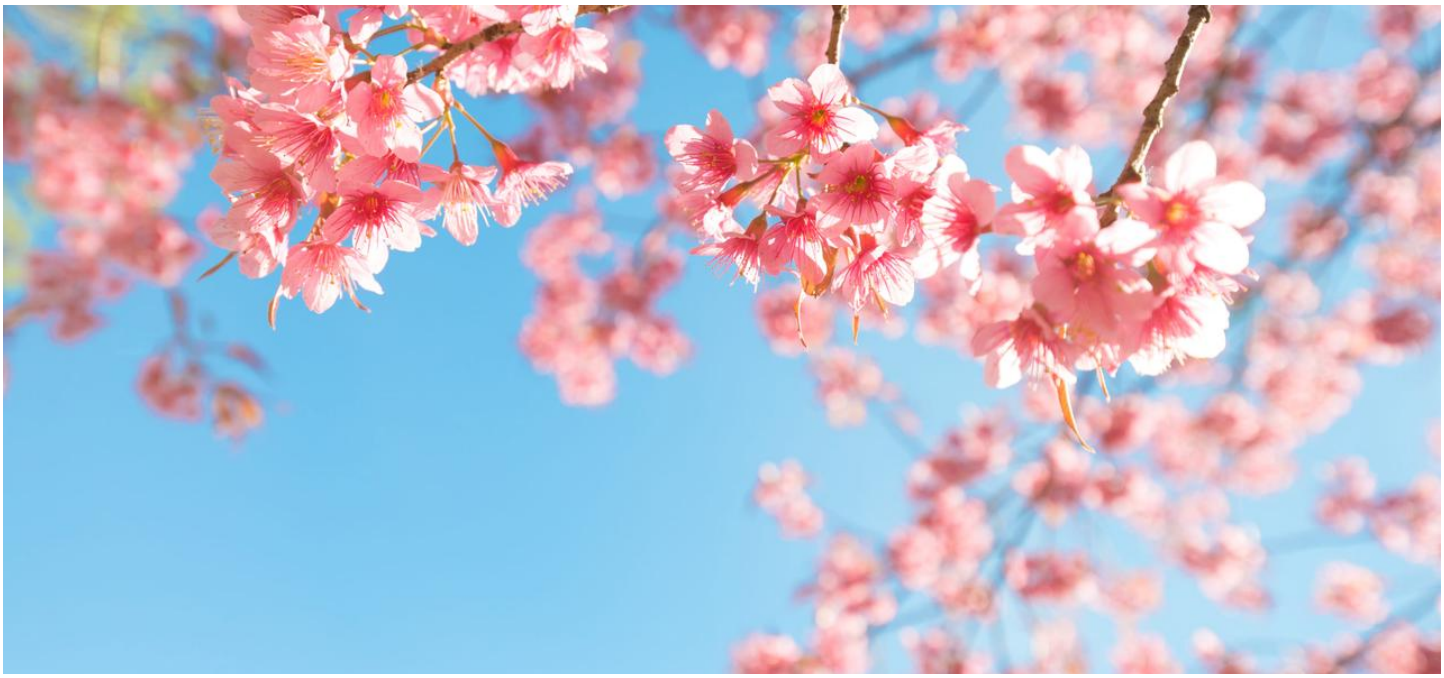


大语言模型基础原理学习笔记 副本(添加新内容) 副本

💡 大模型相关基础原理，总共七章，分别是**大模型整体架构、预训练、后训练、常见模型、大模型应用、训练推理优化和模型压缩**

新增第八章为持续更新内容

1. Architecture 大模型整体架构

1.1 Tokenization 分词

1.1.1 分词介绍

1.1.2 分词算法

1.1.3 常用分词库

1.1.4 分词方法对比

1.2 Embedding 词嵌入

1.2.1 词嵌入介绍

1.2.2 词嵌入方法

1.3 Attention 注意力

1.3.1 Attention介绍

o o1.3.2 常用Attention方法

1.4 FFN & Add & LN 前馈层、残差连接、层归一化

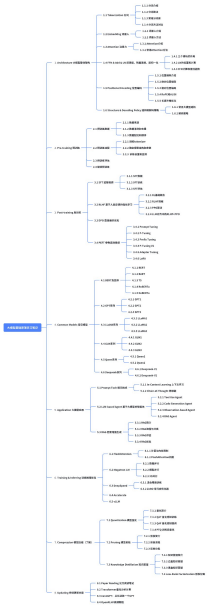
1.4.1 三个模块的作用

1.4.2 LN的位置和计算

1.4.3 FFN计算和激活函数

1.5 Positional Encoding 位置编码

1.5.1 位置编码介绍



1. Architecture 大模型整体架构

经典论文 ***Attention Is All You Need*** 中提出的**Transformer结构，成为了大模型的架构基础**

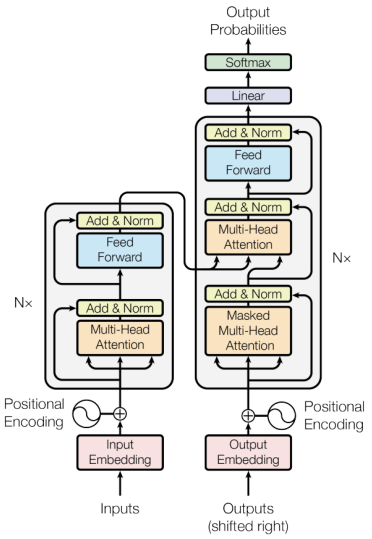


Figure 1: The Transformer - model architecture.

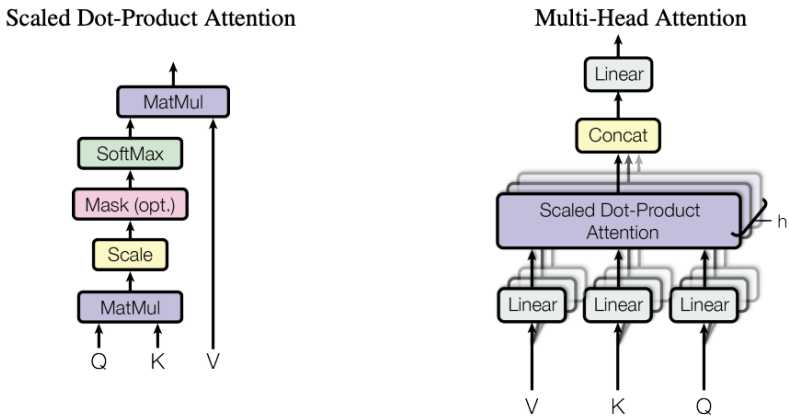


Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

1.1 Tokenization 分词

1.1.1 分词介绍

1. **分词目的：**将输入文本分成一个个**词元**，保证**各个词元拥有相对完整和独立的语义**，以供后续任务（比如学习embedding或者作为高级模型的输入）使用
2. **分词的三种粒度：**
   a. **词粒度 word：**英文天生空格分开词汇，中文可以使用**jieba分词工具**

      优点：词的边界和含义得到保留

      缺点：

      i. **词粒度的词表由于长尾效应可能会非常大，包含很多的稀有词，存储和训练的成本都很高，并且稀有词往往很难学好**

      ii. **OOV(out of vocabulary)问题，对于词表之外的词无能为力**

      iii. **无法处理单词的形态关系和词缀关系，同一个词的不同形态，语义相近，完全当做不同的单词不仅增加了训练成本，而且无法很好的捕捉这些单词之间的关系；同时，也无法学习词缀在不同单词之间的泛化**

   b. **字符粒度 char：**OOV问题迎刃而解

      优点：词表极小，比如26个英文字母可以组合出所有词，5000多个中文常用字基本也能组合出足够的词汇，再加上一些常用字符

      缺点：

      i. **无法承载丰富的语义**

      ii. **序列长度增长，带来计算成本**的增长

c. **子词粒度 subword：**粒度介于char和word之间，基本思想为常用词应该保持原状，**生僻词应该拆分成子词以共享token压缩空间**

   i. **优点：**可以**较好的平衡词表大小与语义表达能力**，比如OOV问题可以通过subword的组合来解决；三种主流的Subword分词算法，分别是**BPE，WordPiece**和**Unigram Language Model**

3. **分词的实例：**一个可以可视化分词结果的网站 **https://tiktokenizer.vercel.app/ ，**下面展示了一些分词可视化的例子，分别用**gpt2**和**gpt4o**的Tokenizer去可视化中文、英文、数字和代码的分词结果，可以看到相比gpt2，**gpt4o的分词器词汇表更大**，相同的内容最后的**token总数更少**，每一个token包含的**连贯的语义更加准确**

gpt2

Token count
502

```
大型语言模型（LLM）是最近几年被开发出来的深度神经网络模型，如 OpenAI 所提供的 ChatGPT 模型。它们开创了自然语言处理（NLP）的新时代。

Large language models (LLMs) are deep neural network models that have been developed in recent years, such as the ChatGPT model provided by OpenAI. They ushered in a new era of natural language processing(NLP).

456 + 1234 = 1690

class SimpleTokenizerV2:
    def __init__(self, vocab):
        self.str_to_int = vocab
        self.int_to_str = { i:s for s,i in vocab.items()}
    def encode(self, text):
        preprocessed = re.split(r'([,.?_!"()\']|--|\s)', text)
        preprocessed = [item.strip() for item in preprocessed if item.strip()]
        preprocessed = [item if item in self.str_to_int else "<|unk|>" for item in preprocessed] #A
        ids = [self.str_to_int[s] for s in preprocessed]
        return ids
    def decode(self, ids):
        text = " ".join([self.int_to_str[i] for i in ids])
        text = re.sub(r'\s+([,.?!"()\'])', r'\1', text) #B
        return text
```

gpt-4o

Token count
286

```
大型语言模型（LLM）是最近几年被开发出来的深度神经网络模型，如 OpenAI 所提供的 ChatGPT 模型。它们开创了自然语言处理（NLP）的新时代。

Large language models (LLMs) are deep neural network models that have been developed in recent years, such as the ChatGPT model provided by OpenAI. They ushered in a new era of natural language processing(NLP).

456 + 1234 = 1690

class SimpleTokenizerV2:
    def __init__(self, vocab):
        self.str_to_int = vocab
        self.int_to_str = { i:s for s,i in vocab.items()}
    def encode(self, text):
        preprocessed = re.split(r'([,.?_!"()\']|--|\s)', text)
        preprocessed = [item.strip() for item in preprocessed if item.strip()]
        preprocessed = [item if item in self.str_to_int else "<|unk|>" for item in preprocessed] #A
        ids = [self.str_to_int[s] for s in preprocessed]
        return ids
    def decode(self, ids):
        text = " ".join([self.int_to_str[i] for i in ids])
        text = re.sub(r'\s+([,.?!"()\'])', r'\1', text) #B
        return text
```

4. **提升分词效果的意义：**Tokenization是很多LLM问题的核心

Tokenization :(

Tokenization is at the heart of much weirdness of LLMs. Do not brush it off.

- Why can't LLM spell words? **Tokenization**.
- Why can't LLM do super simple string processing tasks like reversing a string? **Tokenization**.
- Why is LLM worse at non-English languages (e.g. Japanese)? **Tokenization**.
- Why is LLM bad at simple arithmetic? **Tokenization**.
- Why did GPT-2 have more than necessary trouble coding in Python? **Tokenization**.
- Why did my LLM abruptly halt when it sees the string "<|endoftext|>"? **Tokenization**.
- What is this weird warning I get about a "trailing whitespace"? **Tokenization**.
- Why the LLM break if I ask it about "SolidGoldMagikarp"? **Tokenization**.
- Why should I prefer to use YAML over JSON with LLMs? **Tokenization**.
- Why is LLM not actually end-to-end language modeling? **Tokenization**.
- What is the real root of suffering? **Tokenization**.

1.1.2 分词算法

1. **Byte Pair Encoding(BPE)**
   - **核心思想：从一个基础小词表开始，通过不断合并最高频的连续token对来产生新的token**
   - **具体做法：**输入训练语料和期望词表大小V
     i. **准备基础词表**：比如英文中26个字母加上各种符号，并初始化ID
     ii. 基于基础词表将准备的语料拆分为最小单元
     iii. 在语料上**统计单词内相邻单元对的频率，选择频率最高的单元对进行合并**
     iv. 重复第3步**直到达到预先设定的subword词表大小或下一个最高频率为1**
   - **优点：**可以**有效地平衡词汇表大小和编码步数**(编码句子所需的token数量，与词表大小和粒度有关)
   - **缺点：**基于贪婪和确定的符号替换，**不能提供带概率的多个分词结果**(这是相对于ULM而言的)；**解码的时候面临歧义问题**（比如对于同一个句子 "Hello World"，分词结果可能不同"Hell/o/world"或者"He/llo/world"）
2. **Byte-level BPE(BBPE)**
   - **核心思想：**将BPE从字符级别扩展到**字节级别，**来自**噪声文本或字符丰富的语言（如日语和中文）的稀有字符**可能会不必要地**占用词汇表并限制其紧凑性**
   - **具体做法：**基础词表使用**256的字节集，UTF-8编码**
   - **优点：**
     i. 效果与BPE相当，但**词表大为减小**
     ii. 可以在**多语言之间通过字节级别的子词实现更好的共享**
     iii. 即使字符集不重叠，也可以通过字节层面的共享来实现良好的迁移
   - **缺点：**

i. 编码序列时，长度可能会略长于BPE，**计算成本更高**

ii. 由**byte解码时可能会遇到歧义**，需要通过**上下文信息和动态规划来进行解码，**保证输出有效的句子

3. **WordPiece**
   - **核心思想：**与BPE类似，也是从一个基础小词表出发，通过不断合并来产生最终的词表。主要的差别在于，BPE按频率来选择合并的token对，而**wordpiece按token间的互信息来进行合并**
   - **具体做法：**输入训练语料和词表大小V
     i. **准备基础词表**：比如英文中26个字母加上各种符号

     ii. 基于基础词表将语料拆分为最小单元

     iii. 基于第2步数据**训练语言模型，可以是unigram语言模型，通过极大似然进行估计即可**

     iv. 从所有可能得token对中选择，选择**合并后可以最大程度地增加训练数据概率的token对进行合并**，假设句子由N个子词组成 $S=(t_1,t_2,...,t_N)$，句子 S 的**似然值等于所有子词概率的乘积**：

     $$\log P(s)=\sum_{i=1}^{N}\log P(t_i)$$

     假设各个子词相互独立。合并相邻两个子词 x 和 y 产生新的子词 z，**句子的似然值变化为两个子词之间的互信息**：

     $$\log P(t_z)-(\log P(t_x)+\log P(t_y))=\log\frac{P(t_z)}{P(t_x)+P(t_y)}$$

     其实就是每次选择**互信息最大的两个子词合并**，两个子词在语言模型上有较强的关联性

     v. 重复第4步**直到达到预先设定的subword词表大小或概率增量低于某一阈值**
   - **优点：**可以**较好的平衡词表大小和OOV问题**
   - **缺点：**可能会产生一些**不太合理的子词或者说错误的切分**；对**拼写错误非常敏感**；对**前缀的支持不够好；**一种解决方案是：**将复合词拆开、将前缀也拆开**

4. **Unigram Language Model(ULM)**
   - **核心思想：**初始化一个大词表，然后**通过unigram 语言模型计算删除不同subword造成的损失来代表subword的重要性，保留loss较大或者说重要性较高的subword，**ULM会倾向于保留那些以**较高频率出现在很多句子的分词结果中的子词**，因为这些子词如果被删除，其损失会很大
   - **具体做法：**输入训练语料和词表大小V
     i. **准备基础词表：**初始化一个很大的词表，比如**所有字符+高频ngram，也可以通过BPE算法初始化**

     ii. 针对当前词表，**用语言模型（unigram lm)估计每个子词在语料上的概率**

     iii. **计算删除每个subword后对总loss的影响，作为该subword的loss**

iv. 将**子词按照loss大小进行排序，保留前x%的子词；注意，单字符不能被丢弃，以免OOV**

v. 重复步骤2到4，直到词表大小减少到设定值

- **优点：**
  i. 使用的训练算法可以**利用所有可能的分词结果，这是通过data sampling算法实现的**
  ii. 提出一种基于语言模型的分词算法，这种语言模型可以**给多种分词结果赋予概率，从而可以学到其中的噪声**
  iii. 使用时**也可以给出带概率的多个分词结果**
- **缺点：**1）**效果与初始词表息息相关，初始的大词表要足够好，比如可以通过BPE来初始化**；2）略显复杂

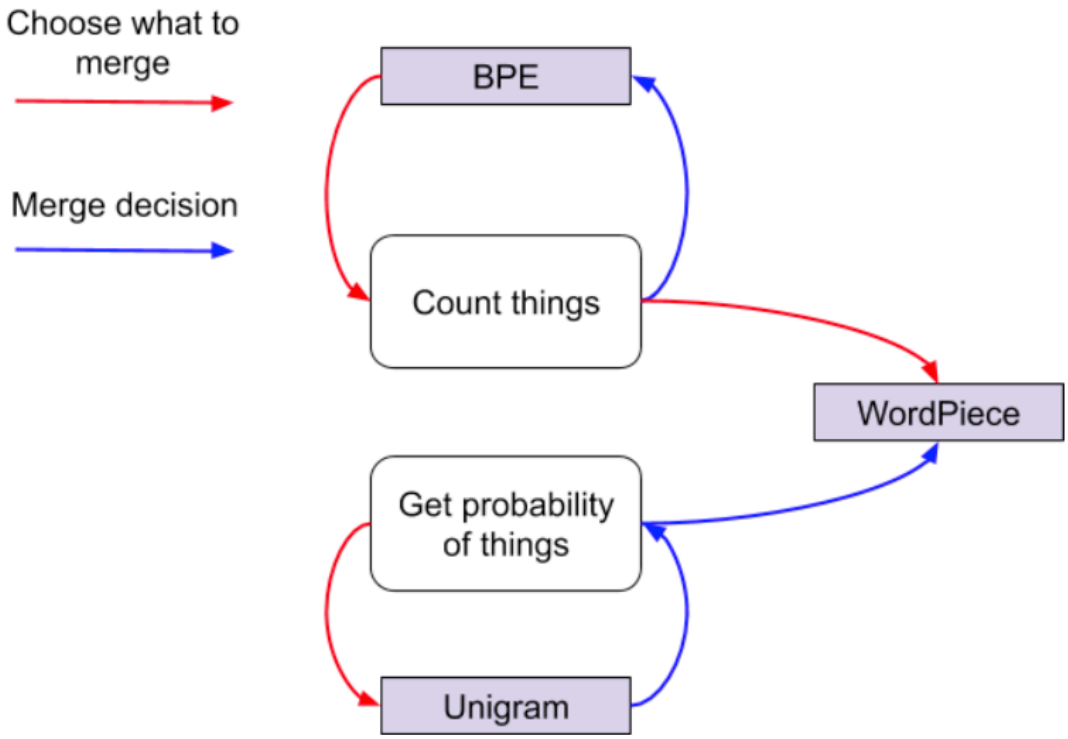
1.1.3 常用分词库

1. **SentencePiece**
   - **多分词粒度**：支持完整和高性能的BPE、ULM子词算法，也支持char, word分词
   - **多语言**：以unicode方式编码字符，将所有的输入（英文、中文等不同语言）都转化为unicode字符，解决了多语言编码方式不同的问题
   - **编解码的可逆性**：之前几种分词算法对空格的处理略显粗暴，有时是无法还原的。Sentencepiece显式地将空白作为基本标记来处理，用一个元符号 “▁”（U+2581）转义空白，这样就可以实现简单且可逆的编解码（**编解码的可逆性：Decode(Encode(Normalized(text)))= Normalized(text)**）
   - **无须Pre-tokenization**：Sentencepiece可以直接从raw text/setences进行训练，无须Pre-tokenization
   - 快速和轻量化
2. **Tokenizers库**

   编码时的pipeline：
   - **normalization**：清理、删除空格、删除变音符号、小写化、Unicode normalization
   - **pre-tokenization**：将文本拆分为更小的对象的行为，这些对象给出了训练结束时token的上限。 考虑这一点的一个好方法是预分词器会将您的文本拆分为“单词”，然后您的最终token将是这些单词的一部分
   - **model**：具体分词算法
   - **post-processing**

1.1.4 分词方法对比

- **wordpiece和BPE的对比：**都是走的**合并的思路**，将语料拆分成最小单元（英文中26个字母加上各种符号，这些作为初始词表）然后进行合并，词表从小到大；核心区别就在于wordpiece是按token间的**互信息**来进行合并而BPE是按照token一同**出现的频率**来合并的
- **wordpiece和ULM的对比：**都使用**语言模型来挑选子词**；区别在于前者**词表由小到大**，而后者**词表由大到小**，先初始化一个大词表，根据评估准则不断丢弃词表，直到满足限定条件。ULM算法考虑了句子的不同分词可能，因而能够输出带概率的多个分词结果



1.2 Embedding 词嵌入

1.2.1 词嵌入介绍

关于词向量与Embedding：Embedding是以每个字在给定分布随机初始化的随机向量而组成的**可学习参数矩阵**，也就是一个全连接Dense层，其**以onehot为输入，稠密向量为输出，即词向量**，因此在实现上，用**lookup查表**来代替矩阵乘积以提高性能。PyTorch中常用的实现为**nn.Embedding(vocab_size, embed_dim)**，embed_dim为词向量的表征纬度大小，vocab_size为词表大小

1.2.2 词嵌入方法

1. **OneHot**

   采用One-Hot编码方式来表示词向量非常简单，但**缺点**也是显而易见的：

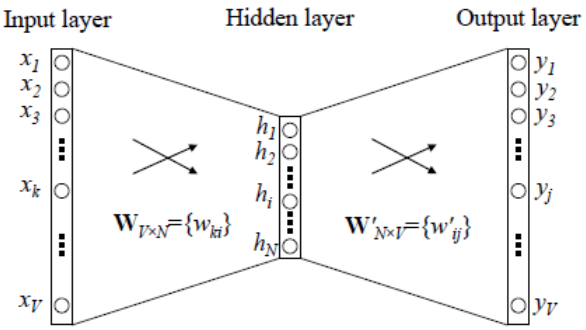
   - 一方面实际使用的**词汇表很大**，经常是百万级以上，这么**高维的数据处理起来会消耗大量的计算资源与时间**

- 另一方面，One-Hot编码中**所有词向量之间彼此正交，没有体现词与词之间的相似关系**

2. **Word2Vec**

Distributed representation可以解决One-Hot编码存在的问题，它的思路是通过训练，将原来One-Hot编码的每个词都映射到一个**较短的词向量**上来，而这个较短的词向量的维度可以由自己在训练时根据任务需要来指定

Word2Vec 的训练模型本质上是**只具有一个隐含层的神经元网络，最后取隐藏层的表示作为词向量。主要分为两种任务类型 CBOW 和 Skip-gram**



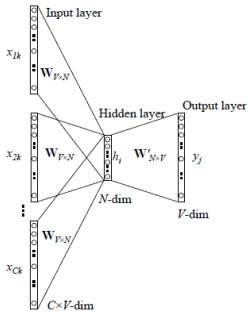


Figure 2: Continuous bag-of-word model

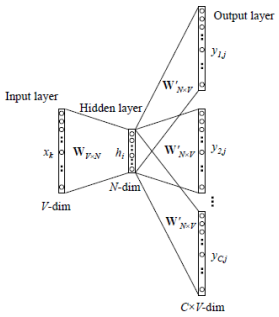


Figure 3: The skip-gram model.

- **CBOW(Continus bag-of-word)：根据语境预测当前词**
- **Skip-gram：根据当前词预测语境**

一般神经网络语言模型在预测的时候，输出的是预测目标词的概率，通过softmax得到，也就是每一次预测都要基于全部的数据集进行计算，这无疑会带来很大的时间开销。Word2Vec提出两种加快训练速度的方式，一种是**Hierarchical softmax (霍夫曼树)**，另一种是**Negative Sampling**

i. **霍夫曼树**

为了避免计算所有词的softmax概率，word2vec采用了霍夫曼树来代替从隐层到softmax层的映射，根据词频来建立哈夫曼树。**将多分类转为二分类问题**。哈夫曼树的所有内部节点就类似之前神经网络隐藏层的神经元，其中，根节点的词向量对应我们的投影后的词向量，而**所有叶子节点就类似于之前神经网络softmax输出层的神经元**，叶子节点的个数就是词汇表的大小。其优点总结如下：

- **计算效率高**：由于是二叉树，使得计算量从 V 变为了 $log_2 V$
- **符合贪心优化思想**：高频词更加接近树根，能更快速得被检索，符合贪心优化的思想

霍夫曼树的缺陷：如果我们的训练样本里的中心词是一个**很生僻的词**，那么就得在霍夫曼树中辛苦的向下走很久了。负采样可以解决该问题。

ii. **负采样**

霍夫曼树使用霍夫曼树来代替传统的神经网络，可以提高模型训练的效率。但是如果我们的训练样本里的中心词是一个很生僻的词，那么就得在霍夫曼树中辛苦的向下走很久了。

负采样：对于给定的词W的上下文Context(w)，词w是一个正样本，其他词是负样本。使用了sigmoid函数：

负样本子集：$NEG(w)$。

给定正样本：$(Context(w), w)$，希望最大化：$g(w) = \prod_{u \in w \cup NEG(w)} p(u|Context(w))$

其中：

$$p(u|x_w, \theta^u) = \begin{cases} \sigma(x_w^T \theta^u), & L^w(u) = 1 \\ 1 - \sigma(x_w^T \theta^u), & L^w(u) = 0 \end{cases}$$

负采样的本质：每次让一个训练样本只更新部分权重，其他权重全部固定；减少计算量；（一定程度上还可以增加随机性）

负采样实际例子：

- 当我们用训练样本（input word:"fox", output word:"quick"）来训练我们的神经网时，“fox”和“quick”都是经过one-hot编码的。如果我们的vocabulary大小为10000时，在输出层，我们希望“quick”单词那个位置输出1，其余都是0。这些其余我们期望输出0的位置所对应的单词我们成为“negative” word。
- **当使用负采样时，我们将随机选择一小部分的negative words（比如选5个negative words）来更新对应的权重。我们也会对我们的positive word进行权重更新**（上面的例子指的是"quick"）。
- 假设隐层-输出层拥有300 x 10000的权重矩阵。如果使用了负采样的方法我们仅仅去更新我们的positive word-“quick”的和我们选择的其他5个negative words的结点对应的权重，共计6个输出神经元，相当于每次只更新300* 6 = 1800个权重。

采样方式：概率采样，可以根据词频进行随机抽样，我们倾向于选择词频比较大的负样本，比如“的”，这种词语其实是对我们的目标单词没有很大贡献的。Word2vec则在词频基础上取了0.75次幂，减小词频之间差异过大所带来的影响，使得词频比较小的负样本也有机会被采到。

3. **FastText**

fastText是一个**快速文本分类算法**，本质和**CBOW**一样

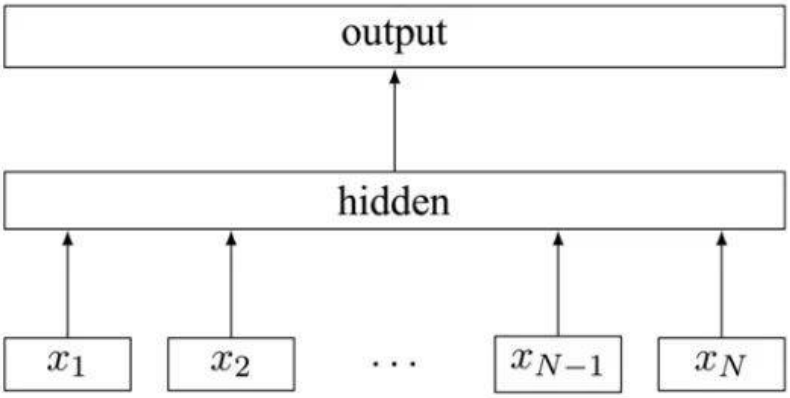


Figure 1: Model architecture of `fastText` for a sentence with N ngram features $x_1, \ldots, x_N$. The features are embedded and averaged to form the hidden variable.

和CBOW一样，fastText模型也只有三层：输入层、隐含层、输出层，输入都是多个经向量表示的单词，输出都是一个特定的target，隐含层都是对多个词向量的叠加平均；**不同的是，**CBOW的输入是目标单词的上下文，fastText的输入是多个单词及其**n-gram特征**，这些特征用来表示单个文档；CBOW的输入单词被onehot编码过，fastText的输入特征是被embedding过；CBOW的输出是目标词汇，fastText的输出是**文档对应的类标**

1.3 Attention 注意力

1.3.1 Attention介绍

- **核心思想：**Attention机制处理时序问题，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**，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。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
- **思想理解：**
  - **Query：寻找的信息**
  - **Key：包含的信息**
  - **Value：需要进行加权的值，**跟key类似

  序列当中**某个位置的Query点积序列中其他所有位置的Keys，产生了相应的权重**，然后**了解有关特定token的更多信息**，而不是序列中任何其他token

  一个很形象的例子，第N个token，再经过vocab embed和position embed之后，我知道我有什么信息，我知道我在哪，基于此生成query即我需要什么样的信息，key的维度里面就是表示自己的信息和位置（对应的地方值会较大）。当在embed空间里面进行点乘的时候，这些查询和键对应的地方就会得到很大的值，（不对应则会产生很小的值）也就是对于我来说，我最终会聚合更多的这个key的信息在我的输出中，然后从这个相关性高的里面学习更多信息

> 我是一个元音，我在第八个位置, 我正在寻找直到当前位置的任何辅音

最终的weights表示，我从各个token中聚合了多少信息

- **计算公式：Scaled Dot-Product**

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

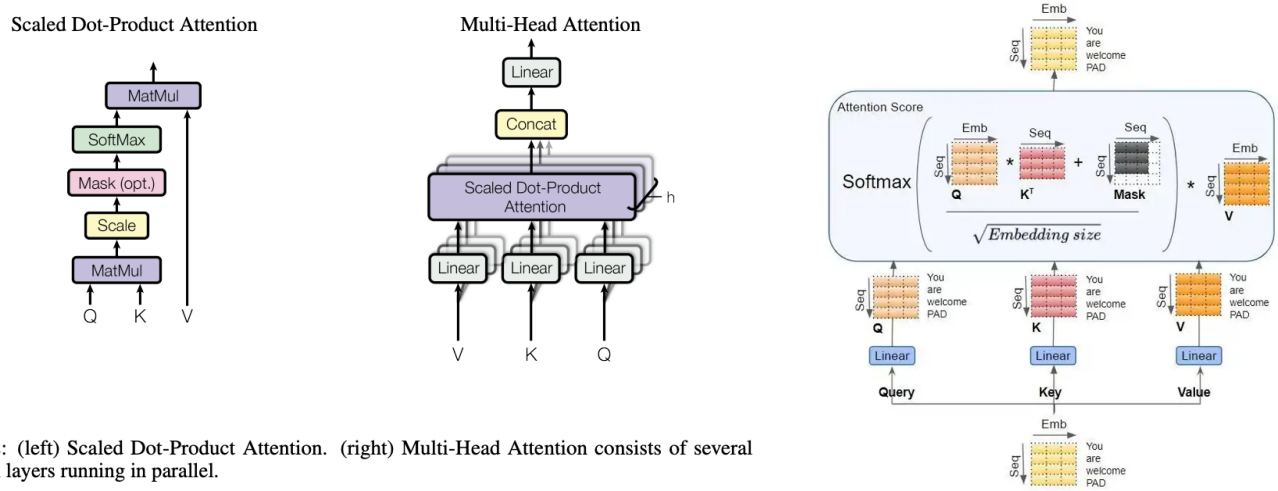


Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

Scaled是指对注意力权重进行缩放，以确保数值的稳定性，通过除以 $sqrt(d_k)$ 实现

当Query和Key向量的维度 d_k 较大时，两个向量的长度比较长时，两个向量的相对差距就会变大。值最大的那个值做出来的softmax会更加靠近1，剩下的会更加靠近0。值的分布会更加向两端靠拢，此时计算梯度的时候，梯度比较小，就会跑不动。联想给数据乘以更大的正数，softmax会更接近onehot，这样初始的时候就去聚集一个不准的节点的信息是不利于学习的。Scale相当于控制初始化的时候的方差

> 刚开始训练的时候，梯度较小不是一个好的情况。因为训练初期，模型还没找到一个合适的参数空间，梯度较小表明模型还没有开始学习，需要更多的训练才能得到好的结果；当模型的初始参数已经非常接近最优解，此时的梯度可能会很小，因为模型已经处于一个相对平坦的区域，此时小梯度可以帮助模型更稳定地收敛到最优解，避免出现震荡和过拟合的情况

Q	A
为什么transformer最后一层不做scale，其他层要做?	最后一层没有两个随机变量相乘的情况，不会出现数值过大导致梯度消失
注意力机制softmax溢出怎么解决?	说明原始概率分布过于陡峭，可以通过提高temperature解决。 temperature原理如下：其中 P(x) 是模型为单词 x 分配的原始概率，P’ (x) 是经过 Temperature 调整后的概率。可以看到，当 Temperature 增大时，原始概率分布会变得更加平坦，各个单词的概率相对更加接近。反之，当 Temperature 减小时，概率分布会变得更加陡峭，使得具有较高概率的单词更有可能被选中。

	$P'(x) = P(x)^{(1/T)}$

o o1.3.2 常用Attention方法

1. **Transformer中的Attention**

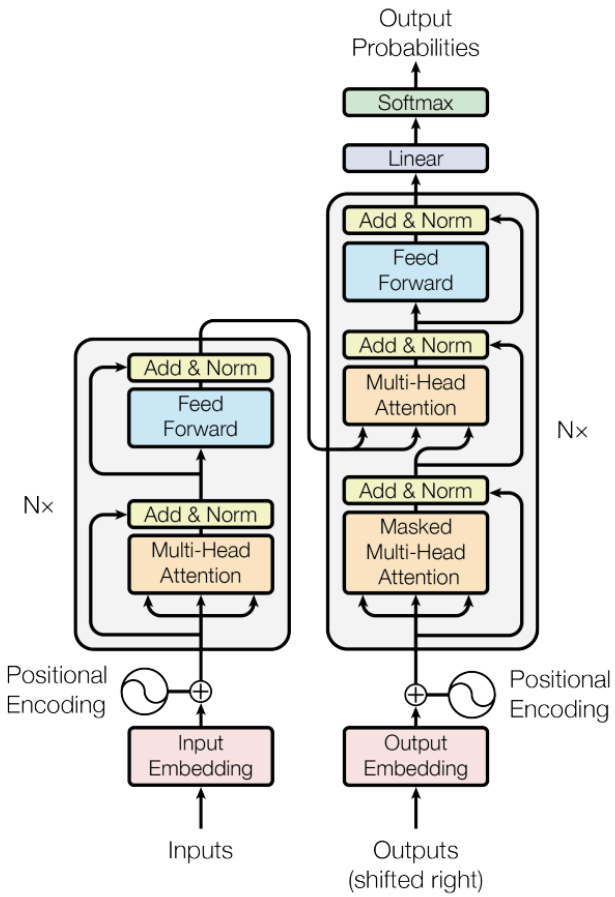
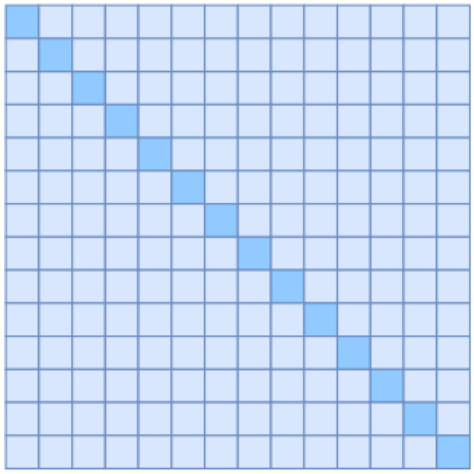


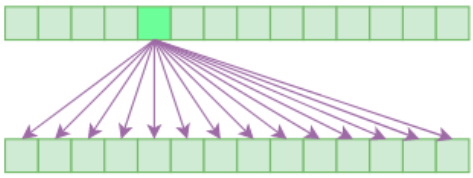
Figure 1: The Transformer - model architecture.

- **Encoder和Decoder中的Self-Attention：** **QKV来自同样的原始序列的**， 只希望关注自己节点的互相交流的信息，查询、键和值都来自同一个输入序列。主要目的是**捕捉输入序列内部的依赖关系**。在Transformer的编码器（Encoder）和解码器（Decoder）的每一层都有自注意力，它允许输入序列的每个部分关注序列中的其他部分，区别在于：**Encoder中的self-attention是当前位置的token与序列全部token计算，Decoder中的self-attention是当前位置的token只与在他之前的token计算（Masked Attention/Casual Attention），为了避免解码过程中的信息泄漏**
- **Decoder中的Cross-Attention：** **QKV来自不同的序列**，有一些其他的节点，我们希望把他们的信息融合到自己的节点上。**查询来自一个输入序列，而键和值来自另一个输入序列**。主要出现在Transformer的解码器。它允许解码器关注编码器的输出。**交叉注意力的思想是使一个序列能够“关注”另一个序列。在许多场景中，这可能很有用，例如在机器翻译中，将输入序列（源语言）的部分与输出序列（目标语言）的部分对齐是有益的**

2. **Attention计算复杂度的优化**

Self Attention是 $O(N^2)$ 的，它要对序列中的任意两个向量都要计算相关度，得到一个 N^2 大小的相关度矩阵





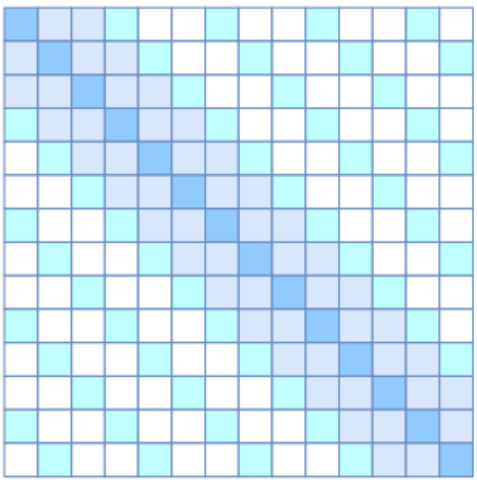
- **Sparse Attention：主要原理是空洞attention和local attention的mixture**

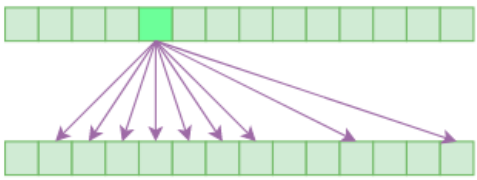
  从注意力矩阵上看**除了相对距离不超过k的、相对距离为k,2k,3k,...的注意力都设为0**，这样一来Attention就具有“**局部紧密相关和远程稀疏相关**”的特性，这对很多任务来说可能是一个不错的先验，因为真正需要密集的长程关联的任务事实上是很少的。

  但是很明显，这种思路有两个**不足之处**:

  i. 如何选择要保留的注意力区域，这是人工主观决定的，带有很大的不智能性

  ii. 它需要从编程上进行特定的设计优化，才能得到一个高效的实现，所以它不容易推广





- **Linear Attention：主要思想就是将softmax拿掉，然后先算K转置V，这样算法复杂度从 $O(N^2)$ 变为 $O(Nd^2)$ 的接近线性**

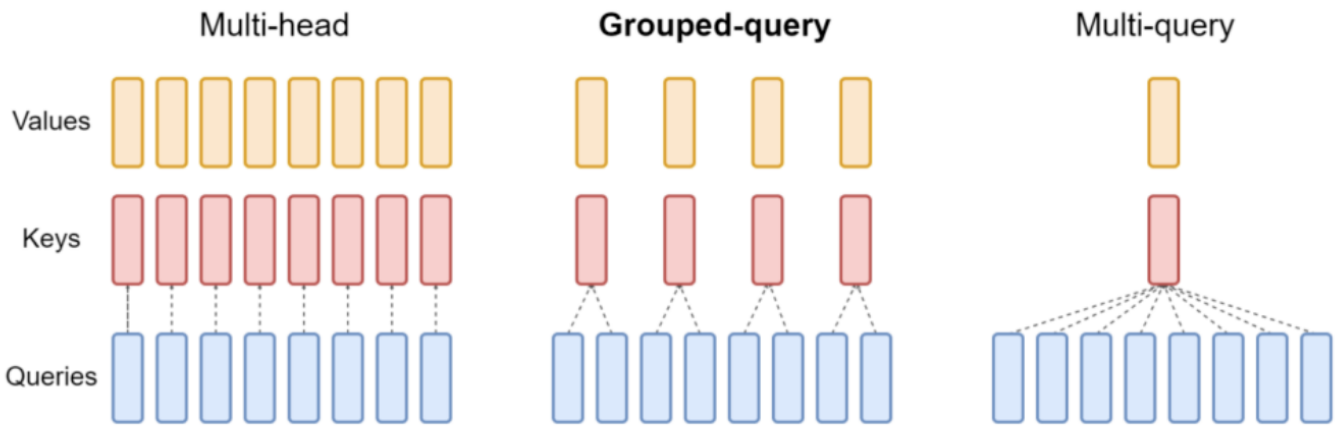
  Softmax是为了从 QK^T 得到非负的权重值，如果有其他方法也可以得到，那么就可以去掉softmax，从而根据矩阵运算规律去先算 K^TV ，注意QKV向量的矩阵都是 $N \times d$ 的

  可以使用**核函数形式**的线性attention，使用两个非负的激活函数处理：

$$Attention(Q,K,V)_i = \frac{\sum_{j=1}^{n} \mathrm{sim}(q_i,k_j)v_j}{\sum_{j=1}^{n} \mathrm{sim}(q_i,k_j)}$$

$$sim(q_i, k_j) = \phi(q_i)^\top \phi(k_j)$$

3. **缓存与效果的取舍：MHA到GQA**



- **Multi-head Attention**：原始transformer中的attention也是MHA，每一个head的head_size将变为原始embed_size的1/num_head，类似group卷积，类似于**建立了很多个交流通道，每个交流通道关注的信息细节不一样，也就是每一个头可以关注到序列中不同子空间的特征**

  手撕MHA：

```python
import torch
from torch import nn
class MutiHeadAttention(torch.nn.Module):
    def __init__(self, hidden_size, num_heads):
        super(MutiHeadAttention, self).__init__()
        self.num_heads = num_heads
        self.head_dim = hidden_size // num_heads

        ## 初始化Q、K、V投影矩阵
        self.q_linear = nn.Linear(hidden_size, hidden_size)
        self.k_linear = nn.Linear(hidden_size, hidden_size)
        self.v_linear = nn.Linear(hidden_size, hidden_size)
        ## 输出线性层
        self.o_linear = nn.Linear(hidden_size, hidden_size)

    def forward(self, hidden_state, attention_mask=None):
        batch_size = hidden_state.size()[0]

        query = self.q_linear(hidden_state)
        key = self.k_linear(hidden_state)
        value = self.v_linear(hidden_state)

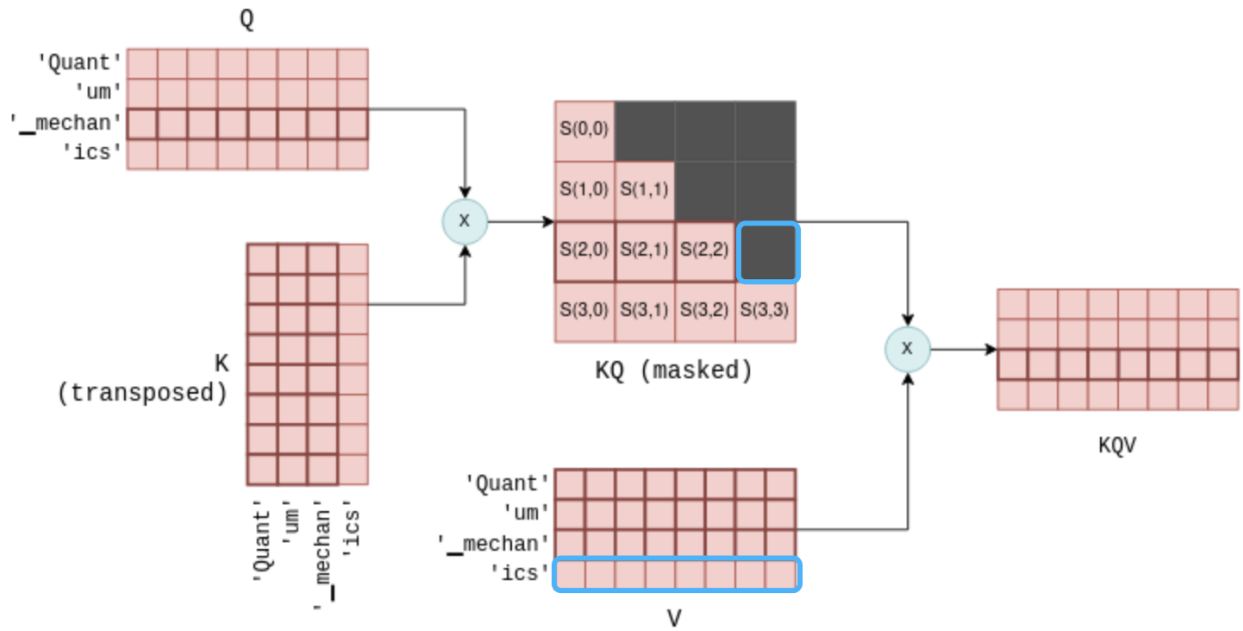
        query = self.split_head(query)
        key = self.split_head(key)
        value = self.split_head(value)
```

```
26
27          ## 计算注意力分数
28          attention_scores = torch.matmul(query, key.transpose(-1, -2)) /
    torch.sqrt(torch.tensor(self.head_dim))
29
30          if attention_mask != None:
31              attention_scores += attention_mask * -1e-9
32          ## 对注意力分数进行归一化
33          attention_probs = torch.softmax(attention_scores, dim=-1)
34
35          output = torch.matmul(attention_probs, value)
36          ## 对注意力输出进行拼接
37          output = output.transpose(-1, -2).contiguous().view(batch_size, -1,
    self.head_dim * self.num_heads)
38
39          output = self.o_linear(output)
40          return output
41
42      def split_head(self, x):
43          batch_size = x.size()[0]
44          return x.view(batch_size, -1, self.num_heads,
    self.head_dim).transpose(1,2)
```

- **Multi-query Attention：所有的query共享KV**

  KVCache：主流的自回归LLM所用的**Causal Attention**，在token by token递归推理生成时，每次计算当前token的attention的时候，**都需要用到序列前面的所有token的K和V**，因此为了避免重新再计算一遍这部分，可以将之前序列token计算过的KV缓存下来用，这就是**KVCache技术**

  下图所示，第三行仅根据第三个Q和前三个KV向量确定，后续token不会影响它



MQA对KV Cache的压缩太严重，以至于会影响模型的学习效率以及最终效果

手撕MQA：

```
 ## 多查询注意力
import torch
from torch import nn
class MutiQueryAttention(torch.nn.Module):
    def __init__(self, hidden_size, num_heads):
        super(MutiQueryAttention, self).__init__()
        self.num_heads = num_heads
        self.head_dim = hidden_size // num_heads

        ## 初始化Q、K、V投影矩阵
        self.q_linear = nn.Linear(hidden_size, hidden_size)
        self.k_linear = nn.Linear(hidden_size, self.head_dim) ###
        self.v_linear = nn.Linear(hidden_size, self.head_dim) ###

        ## 输出线性层
        self.o_linear = nn.Linear(hidden_size, hidden_size)

    def forward(self, hidden_state, attention_mask=None):
        batch_size = hidden_state.size()[0]

        query = self.q_linear(hidden_state)
        key = self.k_linear(hidden_state)
        value = self.v_linear(hidden_state)

        query = self.split_head(query)
        key = self.split_head(key, 1)
        value = self.split_head(value, 1)

        key = k.unsqueeze(1).expand(-1, self.num_heads, -1, -1)
        value = v.unsqueeze(1).expand(-1, self.num_heads, -1, -1)

        ## 计算注意力分数
        attention_scores = torch.matmul(query, key.transpose(-1, -2)) /
torch.sqrt(torch.tensor(self.head_dim))

        if attention_mask != None:
            attention_scores += attention_mask * -1e-9

        ## 对注意力分数进行归一化
        attention_probs = torch.softmax(attention_scores, dim=-1)

        output = torch.matmul(attention_probs, value)

```

```
        output = output.transpose(-1, -2).contiguous().view(batch_size, -1, self.head_dim * self.num_heads)

        output = self.o_linear(output)

        return output

    def split_head(self, x, head_num=None):

        batch_size = x.size()[0]

        if head_num == None:
            return x.view(batch_size, -1, self.num_heads, self.head_dim).transpose(1,2)
        else:
            return x.view(batch_size, -1, head_num, self.head_dim).transpose(1,2)
```

- **Grouped-query Attention：将query分成g组，每组的query共享KV，平衡效果和缓存**

  手撕GQA：

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

class GroupedQueryAttention(nn.Module):
    def __init__(self, d_model, num_heads, num_groups):
        super(GroupedQueryAttention, self).__init__()
        assert d_model % num_heads == 0
        assert num_heads % num_groups == 0

        self.num_heads = num_heads
        self.num_groups = num_groups
        self.d_model = d_model
        self.depth = d_model // num_heads

        # Linear layers for Query, Key, and Value
        self.wq = nn.Linear(d_model, d_model)
        self.wk = nn.Linear(d_model, d_model // num_groups)  # Shared Key per group
        self.wv = nn.Linear(d_model, d_model // num_groups)  # Shared Value per group

        self.fc = nn.Linear(d_model, d_model)
```

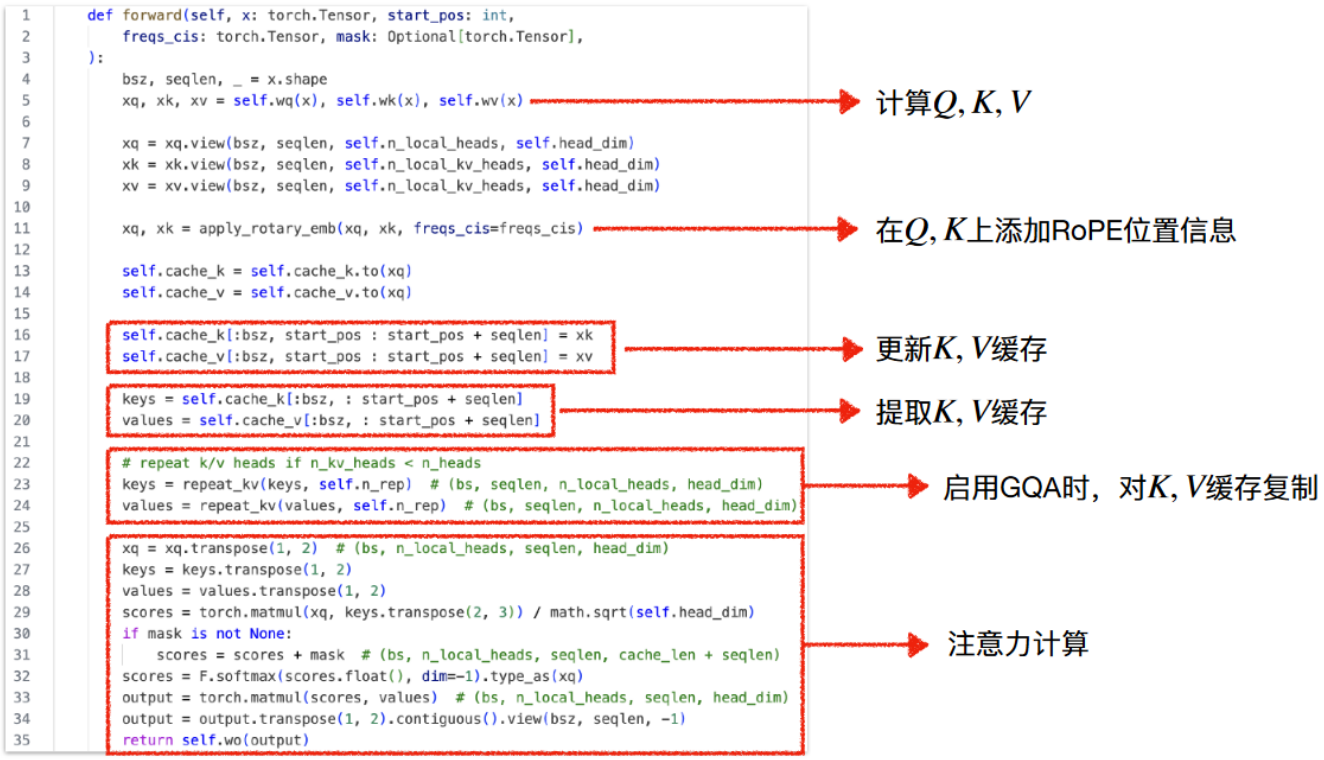
```
23
24      def split_heads(self, x, batch_size):
25          x = x.view(batch_size, -1, self.num_heads, self.depth)
26          return x.permute(0, 2, 1, 3)  # (batch_size, num_heads, seq_len,
    depth)
27
28      def forward(self, x, mask=None):
29          batch_size = x.size(0)
30
31          # Linear projections
32          q = self.wq(x)  # (batch_size, seq_len, d_model)
33          k = self.wk(x)  # (batch_size, seq_len, d_model // num_groups)
34          v = self.wv(x)  # (batch_size, seq_len, d_model // num_groups)
35
36          # Split Query into heads
37          q = self.split_heads(q, batch_size)  # (batch_size, num_heads,
    seq_len, depth)
38
39          # Split Key and Value into groups and expand within the group
40          k = k.view(batch_size, -1, self.num_groups, self.depth *
    (self.num_heads // self.num_groups))
41          v = v.view(batch_size, -1, self.num_groups, self.depth *
    (self.num_heads // self.num_groups))
42
43          k = k.repeat(1, 1, 1, self.num_heads // self.num_groups)
44          v = v.repeat(1, 1, 1, self.num_heads // self.num_groups)
45
46          k = k.view(batch_size, -1, self.num_heads, self.depth)
47          v = v.view(batch_size, -1, self.num_heads, self.depth)
48
49          # Scaled Dot-Product Attention
50          scores = torch.matmul(q, k.transpose(-1, -2)) /
    math.sqrt(self.depth)  # (batch_size, num_heads, seq_len, seq_len)
51          if mask is not None:
52              scores = scores.masked_fill(mask == 0, -1e9)
53
54          attention_weights = F.softmax(scores, dim=-1)  # (batch_size,
    num_heads, seq_len, seq_len)
55          output = torch.matmul(attention_weights, v)  # (batch_size,
    num_heads, seq_len, depth)
56
57          # Concatenate heads
58          output = output.permute(0, 2, 1, 3).contiguous()  # (batch_size,
    seq_len, num_heads, depth)
59          output = output.view(batch_size, -1, self.d_model)  #
    (batch_size, seq_len, d_model)
60
```

```
        # Final linear layer
        output = self.fc(output)  # (batch_size, seq_len, d_model)

        return output, attention_weights

```

带KVCache版本的手撕：



4. **DCA(Dual Chunk Attention)**

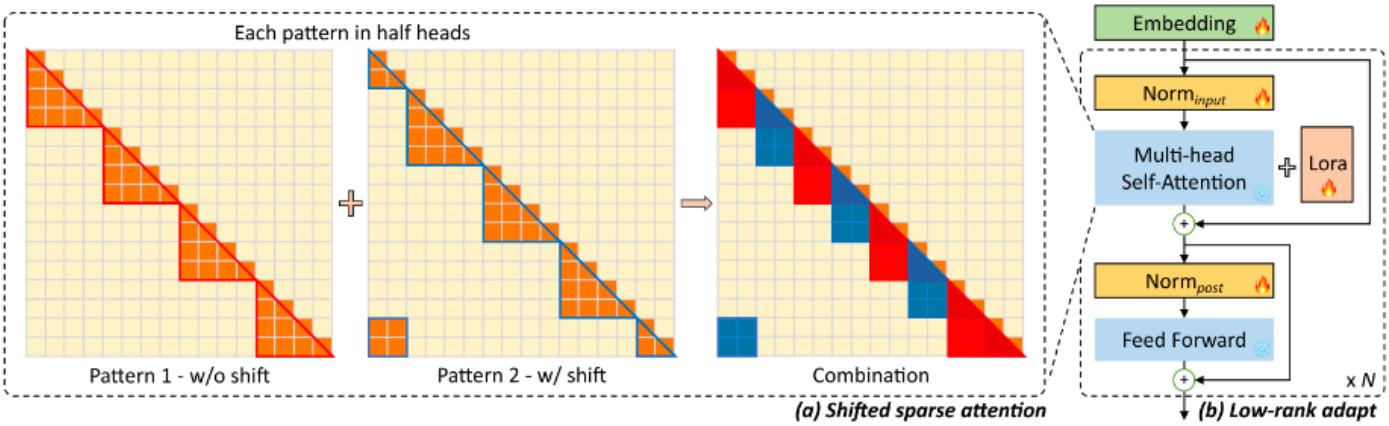
   DCA的核心思想是**将长文本分割成多个较小的“块”(chunks)**，然后分别在这些块内和块之间应用注意力机制。具体步骤如下：

   - **分块：**将长文本分割成若干个小块，每个小块包含一部分文本。例如，一个2000词的文本可以分割成4个每块500词的小块
   - **块内注意力：**对每个小块单独应用注意力机制。这意味着每个块内的单词只与同一个块内的其他单词进行注意力计算，这样可以显著减少计算量
   - **块间注意力：**在计算完块内注意力后，再在这些块之间应用注意力机制。这意味着每个块会与其他块进行全局信息的交互，以捕捉整个文本的上下文关系

   通过这种方式，DCA能够有效地处理长文本，同时保持较高的计算效率和较低的内存占用。

5. **S2-Attention(Shifted Sparse Attention)**

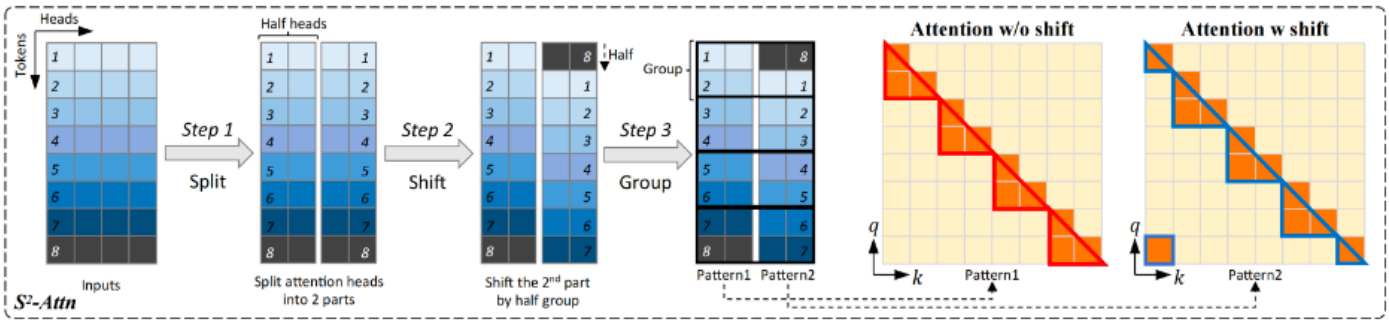
   将上下文长度分成几个组，并在每个组中单独计算注意力。在半注意力头中，将token按半组大小进行移位，这保证了相邻组之间的信息流动。例如，使用组大小为2048的S2-Attn来近似总共8192个上下文长度训练



a. 首先，它将沿头部维度的特征分成两大块 (比如8行4列，8行相当于8个token，4列可以认为是有4个头，然后竖着一切为二)

b. 其次，其中一个块中的标记被移动组大小的一半，**第2个part的第8个token的后一半表示(也即原始inputs第8个token的后两个heads)移动到第2个part的第1行，而第2个part中原来的「第1-7个token的后一半表示」整体往下移动一行**

c. 第三，将token分组并重塑为批量维度，注意力只在每个组内计算，信息通过移位在不同组之间流动。虽然移位可能会引入潜在的信息泄漏，但这可以通过对注意力掩码进行微调来避免



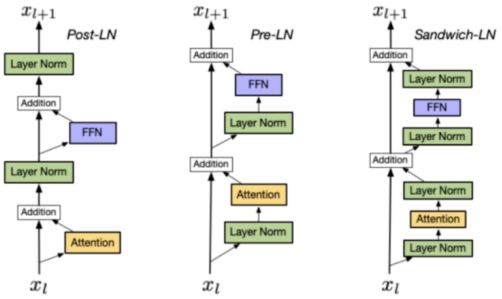
1.4 FFN & Add & LN 前馈层、残差连接、层归一化

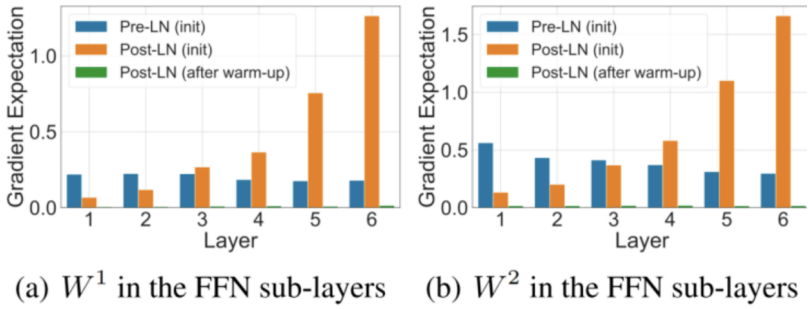
1.4.1 三个模块的作用

- **FFN 前馈层：**过了MHA之后，tokens只是互相查看了一下信息，但是还没思考他们从其他token那里发现了什么。所以**当token通过MHA把信息聚集起来之后，再通过前馈网络独立的去思考学习这些信息，简而言之就是交流加计算**
- **Add 残差连接：**transformer结构一般会堆叠多个block，产生类似神经网络深度增加的优化问题。**残差链接**提供梯度回传的高速公路，一开始残差块的影响比较小，这样梯度也可以很好的回传到输入，即使网络很深，随着训练继续，残差块的梯度逐渐扩大影响。加法会平等的分散梯度。这样的话，至少在初始化的时候，梯度是可以很好的从监督信号回传到输入，从而避免因为很深的网络对优化的难度
- **Layer Norm 层归一化**：加速模型收敛，缓解梯度消失和爆炸问题（**相关：Batch Norm，注意二者的区别和应用场景**）

1.4.2 LN的位置和计算

1. **LN的位置影响**





(a) W^1 in the FFN sub-layers (b) W^2 in the FFN sub-layers

- **Post-norm：深层容易出现训练不稳定的情况，深层的梯度范数逐渐增大**
- **Pre-norm：每层的梯度范数近似相等，训练比较稳定，但是牺牲了深度**
- **Sandwich-norm：平衡，有效控制每一层的激活值，避免它们过大，模型能够更好地学习数据特征，但是训练不稳定可能导致崩溃**

一般认为，**Post-Norm在残差之后做归一化，对参数正则化的效果更强，进而模型的收敛性也会更好；而Pre-Norm有一部分参数直接加在了后面，没有对这部分参数进行正则化，可以在反向时防止梯度爆炸或者梯度消失**，大模型的训练难度大，因而使**用Pre-Norm较多**。目前比较明确的结论是：**同一设置之下，Pre-Norm结构往往更容易训练，但最终效果通常不如Post Norm**。一个□层的Pre Norm模型，其**实际等效层数不如□层的Post Norm模型**，而层数少了导致效果变差了

Pre-Norm结构**无形地增加了模型的宽度而降低了模型的深度**，而深度通常比宽度更重要，所以是无形之中的降低深度导致最终效果变差了。而Post-Norm刚刚相反，它每Norm一次就削弱一次恒等分支的权重，所以Post Norm反而是更突出残差分支的，因此Post-Norm中一旦训练好之后效果更优，但是因为残差支路被削弱了，所以一开始不好训练，需要**warmup**

2. **LN的计算**

- **Layer Norm**：其中 γ 和 β 是**可学习的缩放和偏移参数**

$$\mu = E(X) \leftarrow \frac{1}{H}\sum_{i=1}^{H} x_i$$

$$\sigma \leftarrow \sqrt{\mathrm{Var}(x)} = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(x_i-\mu)^2 + \epsilon}$$

$$y = \frac{x - E(x)}{\sqrt{\mathrm{Var}(X)}} \cdot \gamma + \beta$$

- **RMS Norm**：均方根Norm，**去除掉计算均值进行平移的部分**，计算速度更快效果基本相当

$$RMS(x) = \sqrt{\frac{1}{H}\sum_{i=1}^{H} x_i^2}$$

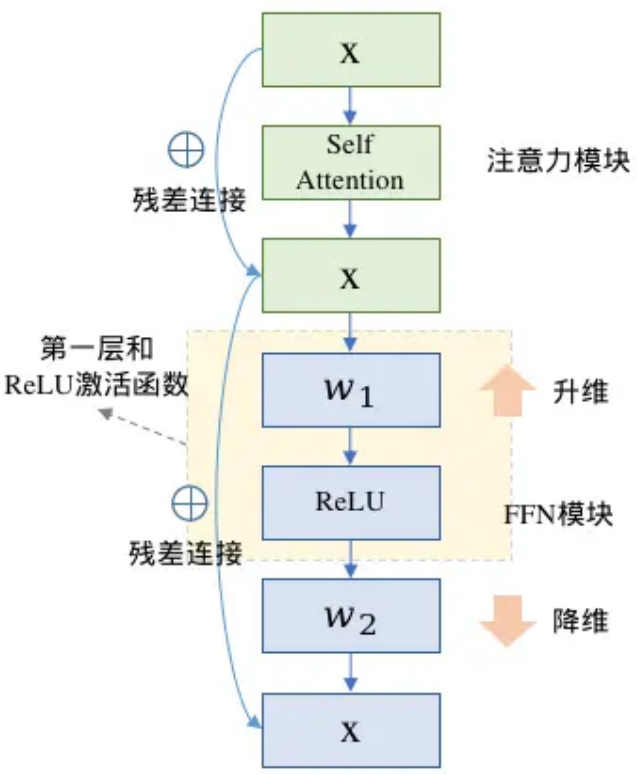
$$x = \frac{x}{RMS(x)} \cdot \gamma$$

- **Deep Norm**：可以缓解爆炸式模型更新的问题，把模型更新限制在常数，使得模型训练过程更稳定。Deep Norm方法在**执行Layer Norm之前，up-scale了残差连接(alpha>1)另外，在初始化阶段down-scale了模型参数(beta<1)**

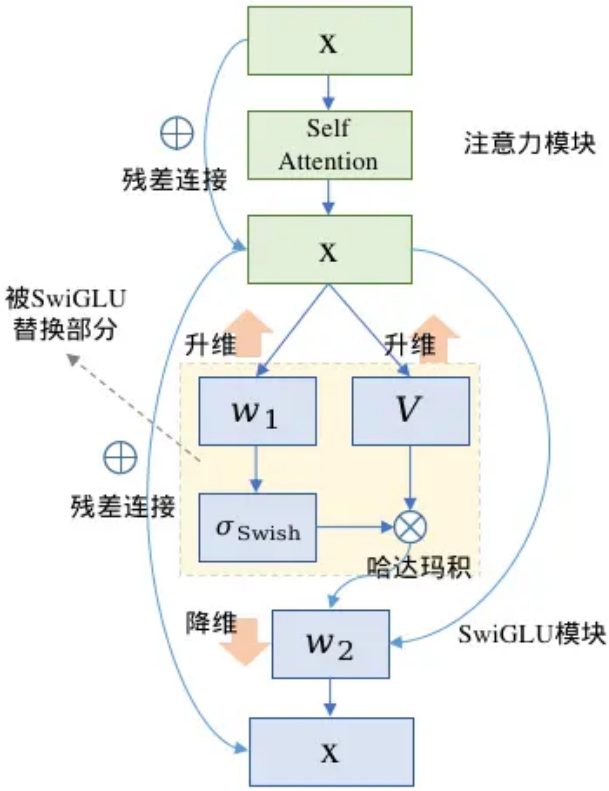
```
def deepnorm(x):
    return LayerNorm(x * α + f(x))
def deepnorm_init(w):
    if w in ['ffn', 'v_proj', 'out_proj']:
        nn.init.xavier_normal_(w, gain=β)
    elif w in ['q_proj', 'k_proj']:
        nn.init.xavier_normal_(w, gain=1)
```

1.4.3 FFN计算和激活函数

- **一般激活函数计的FFN块算公式：** $FFN(x) = ReLU(xW_1 + b1)W_2 + b_2$



一般激活函数



GLU

- **GLU线性门控单元的FFN块计算公式：**

  $GLU(x) = xV \cdot \sigma(xW + b) \quad FFN_{GLU} = (xV \cdot \sigma(xW_1 + b))W_2$

  输出两个线性变换，然后对一个做sigmoid，乘以线性变换的，**相当于做了门控机制**，选择哪些过，哪些不过

  SwiGLU、GeGLU就是用**Swish、GeLU激活函数**分别去**替换GLU中的sigmoid激活函数。一个例子**：在LLaMA2-7B中，FFN的原始输入维度为4096，一般而言**中间层是输入维度的4倍等于**

16384，由于SwiGLU的原因**FFN从2个矩阵变成3个矩阵**，为了使得模型的参数量大体保持不变，**中间层维度做了缩减，缩减为原来的2/3即10922，进一步为了使得中间层是256的整数倍，又做了取模再还原的操作，最终中间层维度为11008**

- **Swish激活函数：** $Swish(x) = x \times sigmoid(\beta * x)$
- **GeLU激活函数：** $GeLU(x) \approx 0.5x\left(1 + tanh\left(\sqrt{\frac{2}{\pi}}\left(x + 0.044715x^3\right)\right)\right)$
- 现在大模型**通常使用SwiGLU**

1.5 Positional Encoding 位置编码

1.5.1 位置编码介绍

RNN的结构包含了序列的时序信息，而Transformer却完全把时序信息给丢掉了，比如**“他欠我100万”，和“我欠他100万”，两者的意思千差万别**，故为了解决时序的问题，Transformer的作者用了一个绝妙的办法：位置编码(Positional Encoding)

Attention计算是无向的，而各种位置信息在tokens间的关系是很重要的。对于人来说，我们很容易知道tokens的位置信息，比如：

- **绝对位置信息，**a1是第一个token，a2是第二个token.....
- **相对位置信息，**a2在a1的后面一位，a4在a2的后面两位.....
- **不同位置间的距离，**a1和a3差两个位置，a1和a4差三个位置....

因此，我们需要这样一种位置表示方式，满足于：

- 它能用来表示一个token在序列中的**绝对位置**
- 在序列长度不同的情况下，**不同序列中token的相对位置/距离也要保持一致**
- 可以用来表示模型在训练过程中从来**没有看到过的句子长度（长度外推问题）**

需要一个**有界又连续**的函数，最简单的，正弦函数sin就可以满足这一点。频率设成非常低，这样几乎不会有不同T的位置编码重合的情况了：

$$PE_t = [sin(w_0 t), sin(w_1 t) \ldots, sin(w_{i-1} t), \ldots, sin(w_{d_{model}-1} t)] \qquad w_i = \frac{1}{10000^{i/(d_{model}-1)}}$$

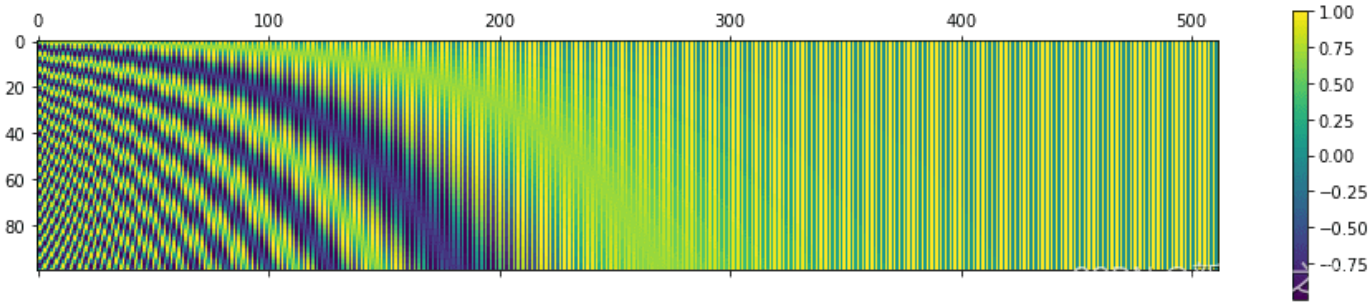
1.5.2 绝对位置编码

1. **Transformer的位置编码：加在embedding上**，但是由于使用的是sin cos 交替，可以通过**线性变换矩阵得到其他位置的表示**，所以可以期望他包含了**相对位置的信息**，而且由于**三角函数有显示的生成规律，所以可以期望有外推性质**
    - **计算公式：**

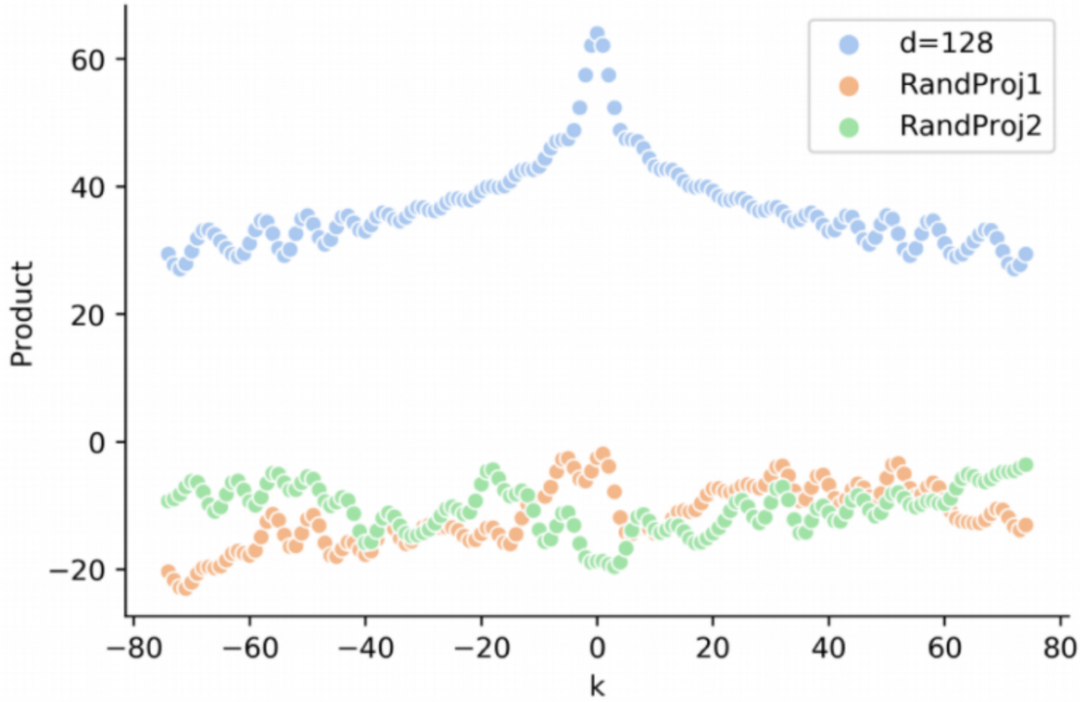
$$PE_t = [sin(w_0 t), cos(w_0 t), sin(w_1 t), cos(w_1 t), \ldots, sin(w_{\frac{d_{model}}{2}-1} t), cos(w_{\frac{d_{model}}{2}-1} t)]$$

$$\begin{pmatrix} \sin(t+\triangle t) \\ \cos((t+\triangle t) \end{pmatrix} = \begin{pmatrix} \cos \triangle t & \sin \triangle t \\ -\sin \triangle t & \cos \triangle t \end{pmatrix} \begin{pmatrix} \sin t \\ \cos t \end{pmatrix}$$

- **可视化：**下图是长度为100，编码维度为512的序列的位置编码可视化，可以发现，由于sin/cos函数的性质，位置向量的每一个值都位于[-1, 1]之间。同时，纵向来看，图的右半边几乎都是黄色的，这是因为越往后的位置，频率越小，波长越长，所以不同的t对最终的结果影响不大。而越往左边走，颜色交替的频率越频繁



- **Transformer位置编码的缺点：**由于**位置编码点积的无向性**，$PE_t^T * PE_{t+\Delta t} = PE_t^T * PE_{t-\Delta t}$ 即两个位置编码的乘积仅取决于 ΔT，距离是成对分布的，不能用来表示位置的方向性。当随着input embedding被喂入attention的时候会出现**距离意识被破坏的现象**，即**正弦位置编码的相对位置表达能力被投影矩阵破坏掉了**，所以在后续BERT的改进中，采用了可学习的位置编码



2. **BERT的可学习位置编码：**直接将**位置编码当作可训练参数**，比如最大长度为512，编码维度为768，那么就初始化一个512×768的矩阵作为位置向量，让它随着训练过程更新。对于这种训练式的绝对位置编码，一般的认为它的**缺点是没有长度外推性**，即如果预训练最大长度为512的话，那么最多就只能处理长度为512的句子，再长就处理不了了。当然，也可以将超过512的位置向量随机初始化，然后继续微调
3. **RNN 位置编码：递归式的绝对位置编码，**input后面接一层RNN，可以用RNN学每个位置的位置编码。**优点是外推、灵活，缺点是丧失transformed的并行处理性质**

1.5.3 相对位置编码

1. **Relative Position Representation：**先对attention公式中的query和key列向量进行拆分，x_i, p_i 分别为输入embedding和位置编码，W_Q, W_K 分别为query和key的投影矩阵，同时展开output部分，W_V 为value的投影矩阵

$$q_i k_j^T = (x_i + p_i) W_Q W_K^T (x_j + p_j)^T = (x_i W_Q + p_i W_Q)(W_K^T x_j^T + W_K^T p_j^T)$$
$$a_{i,j} = \text{softmax}(q_i k_j^T)$$
$$o_i = \sum_j a_{i,j} v_j = \sum_j a_{i,j} (x_j W_V + p_j W_V)$$

为了引入相对位置信息，Google将第一项位置去掉，第二项 $W_K^T p_j^T$ 改为二元位置向量 $R_{i,j}^K$，同时对output部分也改动得到下面的形式

$$a_{i,j} = \text{softmax}\left(x_i W_Q \left(x_j W_K + R_{i,j}^K\right)^\top\right)$$
$$o_i = \sum_j a_{i,j} v_j = \sum_j a_{i,j} (x_j W_V + R_{i,j}^V)$$

二元相对位置向量只依赖于相对位置，而且通常会进行截断处理，这样只需要有限个位置编码（不管是三角函数式还是训练式），都可以表达出任意长度的相对位置

$$R^K_{i,j} = p_K[\text{clip}(i-j, p_{\min}, p_{\max})]$$
$$R^V_{i,j} = p_V[\text{clip}(i-j, p_{\min}, p_{\max})]$$

2. **XLNET式的位置编码：**源自于Transformer-XL论文，继续对attention公式进行完全展开

$$q_i k_j^\top = x_i W_Q W_K^\top x_j^\top + x_i W_Q W_K^\top p_j^\top + p_i W_Q W_K^\top x_j^\top + p_i W_Q W_K^\top p_j^\top$$

Transformer - XL的做法很简单，直接将 p_j 替换为相对位置向量 R_{i-j}，至于两个 p_i，则干脆替换为两个可训练的向量 u, v：

$$x_i W_Q W_K^\top x_j^\top + x_i W_Q W_K^\top R_{i-j}^\top + u W_Q W_K^\top x_j^\top + v W_Q W_K^\top R_{i-j}^\top$$

该编码方式中的相对位置向量没有像经典模型那样进行截断，而是直接用了Sinusoidal式的生成方案，由于 R_{i-j} 的编码空间与 x_j 不一定相同，所以前面的 $W_K^\top$ 换了另一个独立的矩阵 $W_{K,R}^\top$，还有 uW_Q, vW_Q 可以直接合并为单个 u, v，所以最终使用的式子是：

$$x_i W_Q W_K^\top x_j^\top + x_i W_Q W_{K,R}^\top R_{i-j}^\top + u W_K^\top x_j^\top + v W_{K,R}^\top R_{i-j}^\top$$

此外， v_j 上的位置偏置就直接去掉了，即直接令 $o_i = \sum_j a_{i,j} x_j W_V$。似乎从这个工作开始，后面的相对位置编码都只加到Attention矩阵上去，而不加到 v_j 上去了。

3. **T5式的位置编码：**T5模型里边用到了一种更简单的相对位置编码。思路依然源自之前的展开式，如果非要分析每一项的含义，那么可以分别理解为**“输入-输入”、“输入-位置”、“位置-输入”、“位置-位置”四项注意力的组合**。如果我们认为输入信息与位置信息应该是独立（解耦）的，那么它们就不应该有过多的交互，所以**“输入-位置”、“位置-输入”**两项Attention可以删掉，而 $p_i W_Q W_K^\top p_j^\top$ 实际上只是一个只依赖于 (i, j) 的标量，我们可以直接将它作为参数训练出来，即简化为：

$$x_i W_Q W_K^\top x_j^\top + \beta_{i,j}$$

该方法仅仅是**在Attention矩阵的基础上加一个可训练的偏置项**，包含同样的思想的还有微软在ICLR 2021的论文《Rethinking Positional Encoding in Language Pre-training》中提出的**TUPE位置编码，**T5对相对位置进行了一个“分桶”处理，即相对位置是 $i-j$ 的位置实际上对应的是 $f(i-j)$ 位置，映射关系如下：

$i-j$	0	1	2	3	4	5	6	7	8	9	10	11	12
$f(i-j)$	0	1	2	3	4	5	6	7	8	8	9	9	9
$i-j$	16	17	18	19	20	21	22	23	24	25	26	27	28
$f(i-j)$	10	10	10	10	10	10	10	11	11	11	11	11	11

这个设计的思路其实也很直观，就是比较邻近的位置(0~7)，需要比较得精细一些，所以给它们都分配一个独立的位置编码，至于稍远的位置(比如8~11)，我们不用区分得太清楚，所以它们可以共

用一个位置编码，距离越远，共用的范围就可以越大，直到达到指定范围再clip

4. **DeBERTa的位置编码：**同样还是从 $q_{i,k}^j$ 展开式出发，T5干脆去掉了第2、3项，只保留第4项并替换为相对位置编码，而DeBERTa则刚刚相反，**扔掉了第4项，保留第2、3项并替换为相对位置编码：**

$$q_{i,k}^j = x_i W_Q W_K^T x_k^j + x_i W_Q W_K^T R_{i,j}^k + R_{j,i}^k x_i W_Q W_K^T x_k^j$$

DeBERTa提供了使用相对位置和绝对位置编码的一个新视角，它指出NLP的大多数任务可能都只需要相对位置信息，但确实有些场景下绝对位置信息更有帮助，于是它将整个模型分为两部分来理解。以Base版的MLM预训练模型为例，它一共有13层，前11层只是用相对位置编码，这部分称为Encoder，后面2层加入绝对位置信息，这部分它称之为Decoder，还弄了个简称EMD（Enhanced Mask Decoder）；至于下游任务的微调截断，则是使用前11层的Encoder加上1层的Decoder来进行（**需要注意的是他这里命名的encoder和decoder不是传统意义的，不要混淆了**

1.5.4 RoPE和ALiBi

1. **RoPE：旋转式位置编码 Rotary Position Embedding，通过绝对位置编码的方式实现相对位置编码，综合了绝对位置编码和相对位置编码的优点。主要就是对attention中的q, k向量注入了绝对位置信息，然后用更新的q,k向量做attention中的内积就会引入相对位置信息了。RoPE广泛应用在目前的大模型生态中**

二维情况下，对于向量 $\mathbf{q}$ 用复数表示的RoPE如下所示：

$$f(q_m, m) = R_m(f(q_m, m))e^{i\theta(q_m, m)} = \|q_m\| e^{i(\theta(q_m)+m\theta)} = qe^{im\theta}$$

根据复数乘法的几何意义，该变换实际上对应着向量的旋转，所以称之为“旋转式位置编码”，它还可以写成矩阵形式：

$$f(q_m, m) = \begin{pmatrix} \cos m\theta & -\sin m\theta \\ \sin m\theta & \cos m\theta \end{pmatrix} \begin{pmatrix} q_0 \\ q_1 \end{pmatrix}$$

由于内积满足线性叠加性，因此任意偶数维的RoPE，都可以表示为二维情形的拼接，即

$$R_m \begin{pmatrix} \cos m\theta & -\sin m\theta & 0 & 0 & \cdots & 0 & 0 \\ \sin m\theta & \cos m\theta & 0 & 0 & \cdots & 0 & 0 \\ 0 & 0 & \cos m\theta & -\sin m\theta & \cdots & 0 & 0 \\ 0 & 0 & \sin m\theta & \cos m\theta & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & \cos m\theta & -\sin m\theta \\ 0 & 0 & 0 & 0 & \cdots & \sin m\theta & \cos m\theta \end{pmatrix} \begin{pmatrix} q_0 \\ q_1 \\ q_2 \\ q_3 \\ \vdots \\ q_{d-2} \\ q_{d-1} \end{pmatrix}$$

也就是说，给位置为m的向量q乘上矩阵 R_m、位置为n的向量k乘上矩阵 R_n，用变换后的Q、K序列做Attention，那么Attention就自动包含相对位置信息了，因为成立恒等式：

$$(R_m q)^T (R_n k) = q^T R_m^T R_n k = q^T R_{m-n} k$$

值得指出的是，R_m 是一个正交矩阵，它不会改变向量的模长，因此通常来说它不会改变原模型的稳定性。

2. **ALiBi：**ALIBI所做的改动非常简单，只是在Softmax之前，将Attention的计算从 $q_m^T k_n$ 改为

$$q_m^T k_n - \lambda|m-n|$$

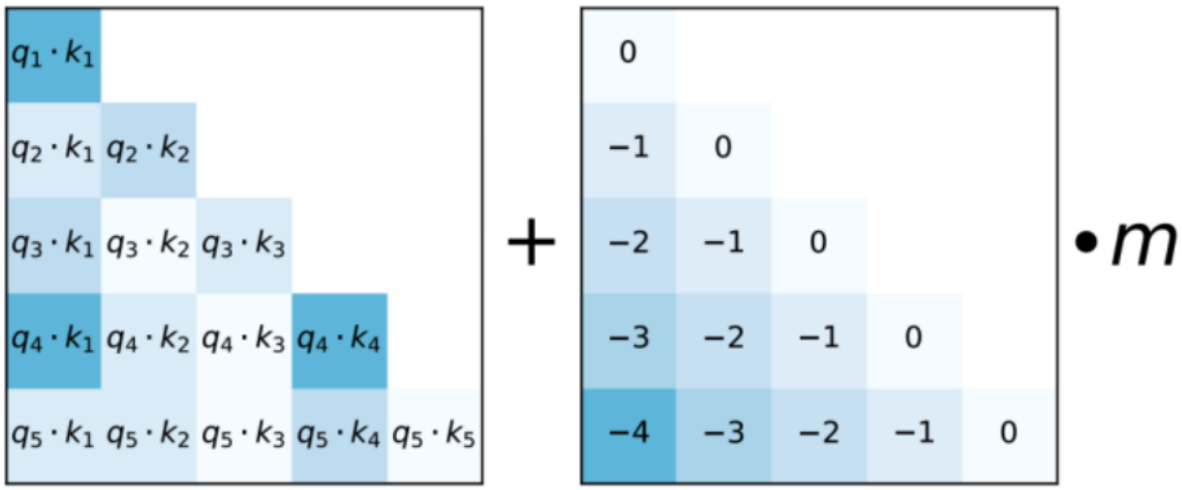
其中 $\lambda > 0$ 是超参数，每个head设置不同的值。从这个定义就可以看出ALIBI跟**局部注意力**的相似之处了，两者都是在Softmax之前减去一个非负矩阵，只不过被减去的非负矩阵有所不同，ALIBI可以看成是**局部注意力**的“平滑版”：

0	0	0	0	∞	∞	∞	∞	∞	∞	∞	∞
0	0	0	0	0	∞	∞	∞	∞	∞	∞	∞
0	0	0	0	0	0	∞	∞	∞	∞	∞	∞
0	0	0	0	0	0	0	∞	∞	∞	∞	∞
∞	0	0	0	0	0	0	0	∞	∞	∞	∞
∞	∞	0	0	0	0	0	0	0	∞	∞	∞
∞	∞	∞	0	0	0	0	0	0	0	∞	∞
∞	∞	∞	∞	0	0	0	0	0	0	0	∞
∞	∞	∞	∞	∞	0	0	0	0	0	0	0
∞	∞	∞	∞	∞	∞	0	0	0	0	0	0
∞	∞	∞	∞	∞	∞	∞	0	0	0	0	0
∞	∞	∞	∞	∞	∞	∞	∞	0	0	0	0

局部注意力等效减去的矩阵

0	1	2	3	4	5	6	7	8	9	10	11
1	0	1	2	3	4	5	6	7	8	9	10
2	1	0	1	2	3	4	5	6	7	8	9
3	2	1	0	1	2	3	4	5	6	7	8
4	3	2	1	0	1	2	3	4	5	6	7
5	4	3	2	1	0	1	2	3	4	5	6
6	5	4	3	2	1	0	1	2	3	4	5
7	6	5	4	3	2	1	0	1	2	3	4
8	7	6	5	4	3	2	1	0	1	2	3
9	8	7	6	5	4	3	2	1	0	1	2
10	9	8	7	6	5	4	3	2	1	0	1
11	10	9	8	7	6	5	4	3	2	1	0

ALiBi减去的矩阵



ALiBi的偏置矩阵**根据q和k的相对距离来惩罚attention score，相对距离越大，惩罚项越大**。相当于两个token的距离越远，相互贡献就越小。

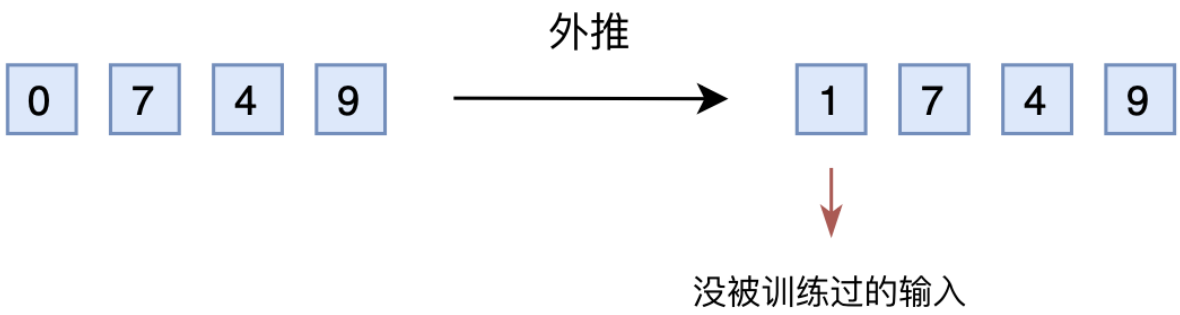
长度外推问题：长度外推性是一个训练和预测的长度不一致的问题

- 预测的时候用到了**没训练过的位置编码**（不管绝对还是相对）
- 预测的时候注意力机制所**处理的token数量远超训练时的数量**

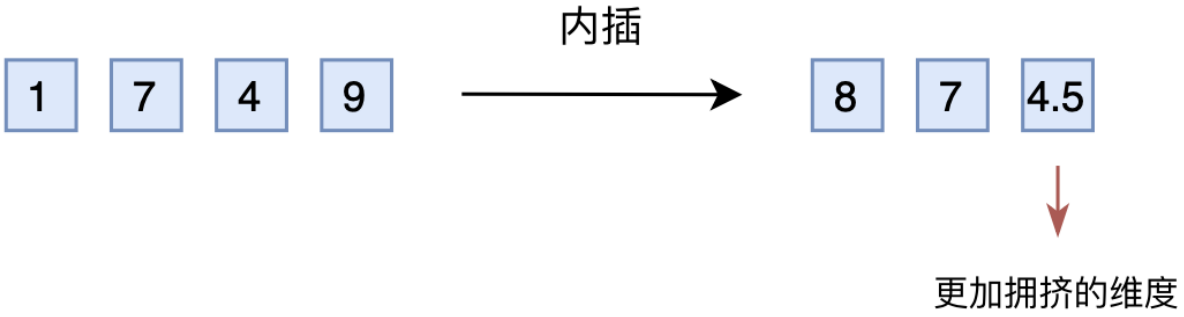
RoPE等周期震荡函数必须进行位置衰减，到远处的**位置信息趋于直线震荡，基本很难有位置信息区分了，所以外推性比训练式的好不了多少，旋转位置编码基于此改进的自然也是如此**

1.5.5 长度外推优化

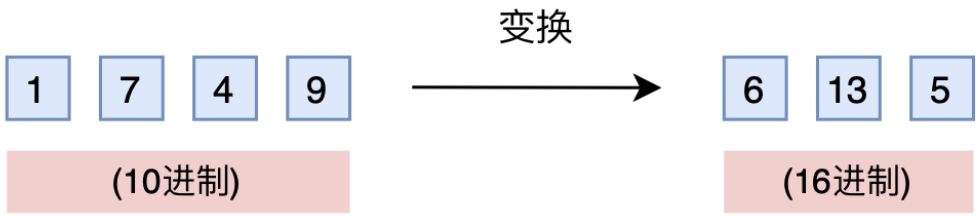
1. **进制表示到直接外推**
   - **进制表示：**假设我们有一个1000以内（不包含1000）的整数N要作为条件输入到模型中，那么要以哪种方式比较好？
     - **一维浮点向量输入**，然而0～999这涉及到近千的跨度，对基于梯度的优化器来说并不容易优化，缩放到0～1之间也不大好，因为此时相邻的差距从1变成了0.001，模型和优化器都不容易分辨相邻的数字
     - **10进制表示直接输入，**将整数N以一个**三维向量[a,b,c]**来输入，a,b,c分别是n的百位、十位、个位。至于如果想要进一步缩小数字的跨度，我们还可以进一步缩小进制的基数，如使用8进制、6进制甚至2进制，代价是进一步增加输入的维度
   - **直接外推：**如果用三维10进制表示训练了模型，又需要将n上限增加到2000以内，那么此时的输入就是一个四维向量了。然而，原本的模型是针对三维向量设计和训练的，所以新增一个维度后，模型就无法处理了。所以解决方案就是**提前预留多几维，训练阶段设为0**，推理阶段改为其他数字，这就是外推Extrapolation。但是推理阶段改为其他数字，因为模型对没被训练过的情况不一定具有适应能力，所以**直接进行外推通常会导致模型的性能严重下降**



2. **线性内插到进制转换**
   - 线性内插：外推改为内插Interpolation，就是将2000以内压缩到1000以内



        - 比如通过除以2，1749就变成了874.5，然后转为三维向量[8,7,4.5]输入到原来的模型中，从绝对数值来看，新的[7,4,9]实际上对应的是1498，是原本对应的2倍，映射方式不一致；从相对数值来看，原本相邻数字的差距为1，现在是0.5，最后一个维度更加拥挤
        - 内插后需要微调训练，以便模型**重新适应拥挤的映射关系**
        - 当处理范围进一步增大时，相邻差异则更小，且集中在个位数，剩下的百位、十位，还是保留了相邻差异为1，内插方法使得不同维度的分布情况不一样，每个维度不对等，模型进一步学习难度也更大
    - **进制转换：**不用新增维度，又能保持相邻差距
        - 三个数字的10进制编码可以表示0~999，**16进制可以表示0~4095**，三维向量就可以覆盖目标范围，代价是**每个维度的数字从0～9变为0～15**



        - 原来训练好的模型已经学会了 875>874，而在16进制下同样有875>874，比较规则是一样的
        - 每个维度超过9之后(10～15)模型，由于一般模型也有一定的泛化能力，所以每个维度稍微往外推一些是没问题，可以正常比较，转换进制可能不微调原来模型也有效

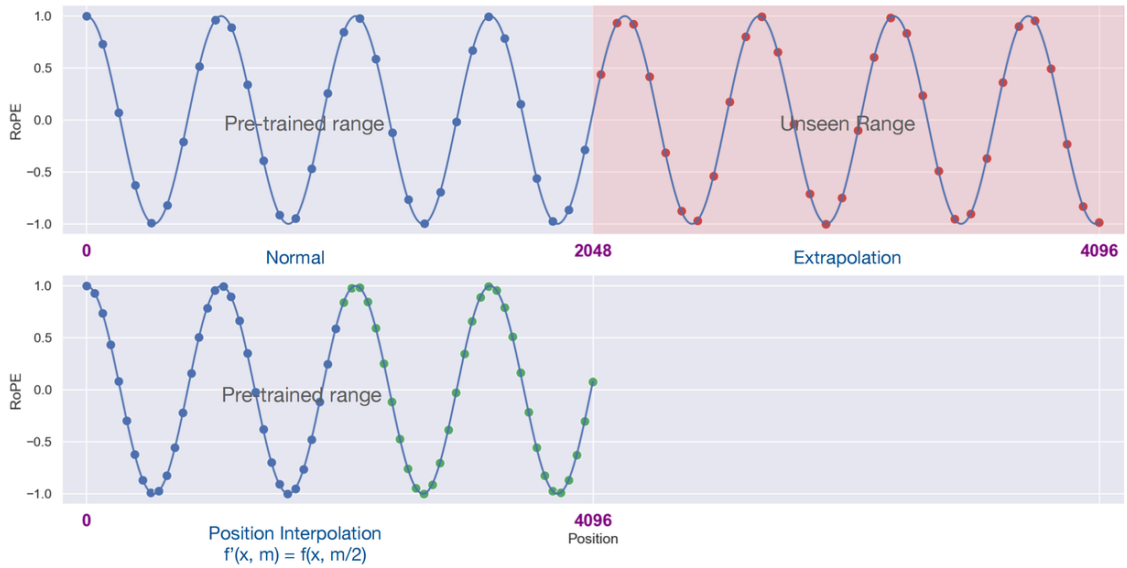
3. **Positional Interpolation 位置内插**

语言模型通常是用固定的上下文长度进行预训练的，如何通过在相对较少的数据量上进行微调来扩展上下文长度，位置插值将上下文长度扩展到预训练极限之外

> 📌 **RoPE的问题：直接外推会出现比较大的Attention Score**
>
> RoPE（相对位置编码）使用正弦和余弦函数将位置信息嵌入到词汇向量的旋转矩阵中。然而，由于以下原因，RoPE直接外推会导致Attention Score显著增加：
>
> - **正弦和余弦函数的周期性：**
>   - **正弦和余弦函数是周期性的**，周期为 2π ，在训练数据中，位置通常在一个相对较小的范围内（例如，0到512或0到2048），这些位置的编码值会保持在周期的某一部
>   - 当位置超出这个范围时（例如，位置变为3000或3500），编码值会进入正弦和余弦函数的另一个周期。由于这些函数的周期性，**这些位置的编码值可能与训练数据中的编码值非常不同**，导致模型在计算注意力分数时出现剧烈变化
> - **高频成分的影响：**

- 在RoPE编码中，较高维度的编码（即频率较高的正弦和余弦成分）会对**较大的位置变化更加敏感，这意味着，随着位置数值的增加，这些高频成分会迅速变化**
- 对于较大的位置值，**正弦和余弦函数的值可能会经历快速变化**，这种快速变化会导致Attention机制中query和key的点积（即Attention Score）出现显著波动、

- **关键思想：**不进行外推，而是直接**将位置索引减小，使得最大位置索引与目标长度大小，即预训练阶段的先前上下文窗口限制相匹配。**可以在相邻的整数位置上插值位置编码，毕竟位置编码可以应用在非整数的位置上(而非在训练位置之外 进行外推)。下图所示，如果直接使用位置(2048,4096]进行推理，那么因为模型没有见过这一部分的位置，效果会出现灾难性的下降，就把[0,4096]这个区间压缩到[0,2048]，原先的1就变成了0.5，4096就变成了2048，这就是**位置内插法**，即把没见过的位置映射到见过的位置



- **内插公式：**对于绝对位置 m，缩放变成 $\frac{mL}{L'}$， L 为原先支持的长度（如2048）， L' 为需要扩展的长度（如4096），计算 query 和 key 的时候，就有 $f'_w(x_m, m, \theta_d) = f_w(x_m, g(m), \theta_d)$，定义缩放比例 $s = \frac{L'}{L}, g(m) = \frac{m}{s}$
- **PI之后是否微调：**效果有所区别
  - PI之后，在没有微调的情况下(在步骤0)，模型可以展示出一定的语言建模能力，如扩展到8192上下文窗口的困惑度<20所示(相比之下，直接外推方法导致困惑度>1000)
  - PI之后，经过微调，**困惑度（perplexity）**迅速改善。 在200步时，模型超过了2048上下文窗口大小的原始模型困惑度，表明模型能够有效地使用比预训练设置更长的序列进行语言建模。 在1000步时，我们可以看到模型稳步改善，并取得了显著更好的困惑度

Model Size	Context Window	Number of fine-tuning steps 0	200	400	600	800	1000
7B	8192	16.10	7.12	7.10	7.02	6.99	6.95
7B	16384	112.13	7.05	6.93	6.88	6.84	6.83

📌 **困惑度**（Perplexity）是自然语言处理中常用的一个评价指标，用于衡量语言模型的好坏。语言模型Model在测试集数据 $T=\{w_1,w_2,\ldots,w_N\}$ 上的困惑度计算，困惑度越低，说明模型对下一个单词的预测越准确，模型性能越好：

$$\text{Perplexity}(Model)=\exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log P\left(w_i\mid w_1,\ldots,w_{i-1}\right)\right)$$

- **PI的问题：**
  - 三角函数 $\sin(\omega x)$，它的周期是 $T=2\pi/\omega$ 对应到RoPE里的每个维度 $(\sin m\theta_j,\cos m\theta_j)$ 其中 $\theta_j=10000^{-2(j-1)/d},j\in[1,2,\ldots,d/2]$（$m$ 为位置，j 为维度）
  - 计算得到周期为：$\frac{2\pi}{m}b^{\frac{2(j-1)}{d}}$，其中，用 b 表示base，即10000。从周期计算的公式我们可以知道，针对不同的维度编码 j，每个维度对应的三角函数周期是越来越大的（即对应到低频、高频）
  - 如果插值是针对绝对位置 m，那么对每个维度 j 都同等地生效；但是周期小（高频）维度，插值之后会变得很密集（本来一个周期包含10个值，内插之后能包含20个值），这样高频的维度就变得很拥挤

4. **NTK-aware 插值到 Dynamic NTK插值**
   - **NTK-aware插值：**核心思想 **高频外推，低频内插，**不像PI针对所有维度平均缩放，而是减少对高频区域的缩放和增加对低频区域的缩放，从而将插值压力分散到多个维度

$$\left[\cos\left(\frac{m}{\beta^0}\right),\sin\left(\frac{m}{\beta^0}\right),\cos\left(\frac{m}{\beta^1}\right),\sin\left(\frac{m}{\beta^1}\right),\ldots,\cos\left(\frac{m}{\beta^{d/2-1}}\right),\sin\left(\frac{m}{\beta^{d/2-1}}\right)\right]$$

   - 上述公式最后面的是最低频 $\frac{m}{\beta^{d/2-1}}$，引入参数 λ 变为 $\frac{m}{\lambda\beta^{d/2-1}}$，让这一项**与内插一致做缩放：**

$$\frac{m}{\lambda\beta^{d/2-1}}=\frac{m/s}{\beta^{d/2-1}}$$
$$\lambda=s^{2/(d-2)}$$

   - 上述公式最前面的是最高频 $\frac{m}{\beta^0}$，引入参数 λ 变为 $\frac{m}{\lambda\beta^0}$，由于 d_{model} 一般比较大，所以 λ 很接近1，即**不做缩放等价于外推，从而NTK-aware插值把外推和内插结合起来**

- **缺点：**一些维度被轻微外推到超出边界的值，因此使用NTK-aware插值进行微调的结果有可能不如PI；此外，由于存在“越界”值，理论尺度因子s并不能准确描述真实的上下文扩展尺度。在实践中，对于给定的上下文长度扩展，尺度值s必须设置得高于预期尺度

- **NTK-by-parts 插值：考虑了波长于上下文的关系**
  - **波长：**维度 j 上嵌入的RoPE执行完整旋转 2π 所需要的token长度 $\lambda_j = \frac{2\pi}{\theta_j} = 2\pi b^{\frac{2(j-1)}{d}}$ 。PI和NTK-aware插值不关心波长的维数
  - **存在问题：**有一些维数的波长长于预训练期间看到的最大上下文长度 $\lambda > L$，这表明一些维数的嵌入可能在旋转域中不均匀分布。**当波长很长时，这些维度上的嵌入几乎不变，可以认为它们保持了绝对位置信息，即每个位置的嵌入不因相对位置变化而变化；当波长较短时，嵌入会在较短的距离内完成多次旋转，这使得这些维度上的嵌入反映的是相对位置信息，即它们可以捕捉到标记之间的相对距离变化。**此外，用比例s去对所有维度进行缩放的时候，所有tokens都变得更彼此接近，同样的位移对应的旋转角度变化减小，向量指向更加相似的方向，**这种缩放严重损害了LLM理解其内部嵌入之间的小型和局部关系的能力，**导致模型在邻近标记的位置顺序上被混淆，从而损害模型的能力
  - **解决方法：**
    - 如果波长 λ 比上下文长度 L 小得多，此时不插值
    - 如果波长 λ 等于或大于上下文长度 L ，此时只做插值，不做任何外推
    - 两者之间的维数可以兼备

    引入比率 $r(j) = \frac{L}{\lambda_j} = \frac{L}{2\pi b^{\frac{2(j-1)}{d}}}$ ，引入边界参数 α, β ，定义斜坡函数

    $$\gamma(r) = \begin{cases} 0, & \text{if } r < \alpha \\ 1, & \text{if } r > \beta \\ \frac{r-\alpha}{\beta-\alpha}, & \text{otherwise} \end{cases}$$ 可以得到NTK-by-parts方法的公式：

    $$g(m) = m$$
    $$h(\theta_j) = (1 - \gamma(r(j)))\frac{\theta_j}{s} + \gamma(r(j))\theta_j$$

    位置索引m不做变化，对不同维度的 θ_j 通过第二个公式进行调整

- **Dynamic NTK 插值：动态调整缩放因子 s**
  - **存在问题：**在很多用例中，以从1到最大上下文大小不等的序列长度进行多次前向传递。一个典型的例子是自回归生成，其中序列长度在每一步之后递增1。之前的方法缩放因子不变，模型在长度小于 L 时可能出现性能折扣，当序列长度大 L' 时可能出现突然退化
  - **动态插值：** $s = max(1, l'/L)$ ， l' 是当前序列的长度

5. **YaRN (Yet another RoPE extensioN method)**

无论数据样本和扩展上下文窗口上的token位置如何，在对logits进行softmax操作之前引入温度t可以统一地影响困惑度，所以可以将注意力权重的计算修改为：

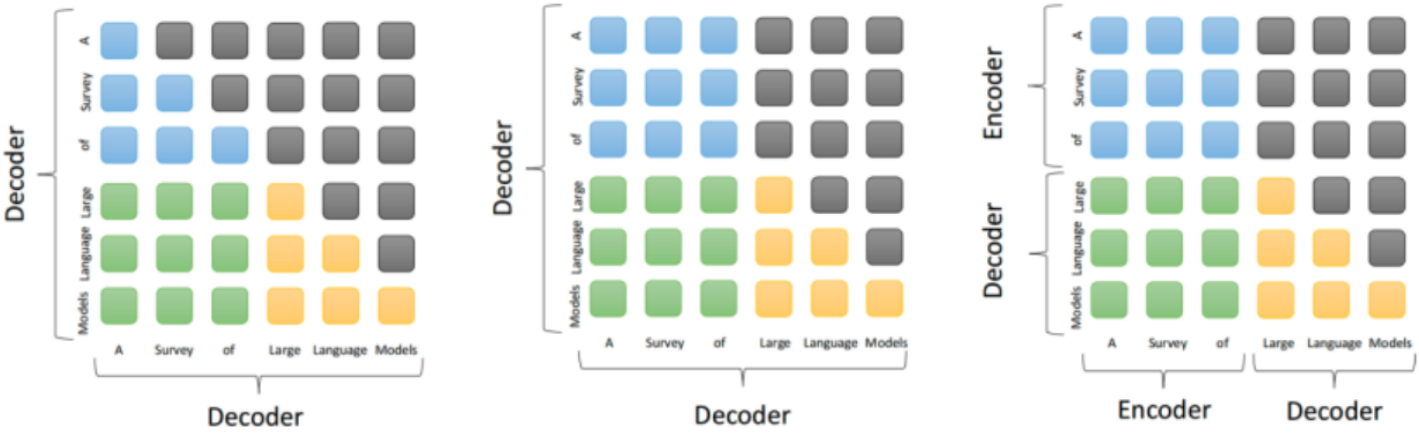
$$\text{softmax}\left(\frac{\mathbf{q_m}^T\mathbf{k_n}}{t\sqrt{d_k}}\right)$$

将RoPE嵌入按相同比例缩放，使得query和key都以 $\sqrt{1/t}$ 进行缩放，然后再结合NTK-by-parts方法得到YaRN，推理和训练阶段没有额外开销，因为RoPE嵌入是提前生成的，而且可以重复使用

对于LLaMA1，2推荐： $\sqrt{1/t} = 0.1\ln(s) + 1$

1.6 Structure & Decoding Policy 结构和解码策略

1.6.1 常见大模型结构



Casual Decoder/Decoder-only　　Prefix LM　　Encoder-Decoder

LLM的结构可以分为四种：**Decoder-only、Encoder-only、Encoder-Decoder、Prefix LM**(可以看成特殊的Encoder-Decoder)

- **Decoder-only**：从左到右的**单向注意力**，**自回归语言模型**，预训练和下游应用完全一致，文本生产效果好。训练效率高，**zero-shot能力强**。**典型模型：GPT、Llama、BLOOM、OPT**
- **Encoder-only**：**典型模型BERT**，目标是生成语言模型，提取表征用的，预训练的语言模型
- **Encoder-Decoder**：输入双向注意力，输出单向注意力。对问题的编码理解更充分，在偏理解的任务上表现较好，但是训练效率低，文本生成任务表现差。**典型模型：T5、Flan-T5、BART**
- **Prefix Decoder**：折中，**典型模型：GLM、U-PaLm**

MoE结构（Mixture of Experts）混合专家

一般的LLM都是Dense模型是相对MoE模型而言的。**每个混合专家（MoE）层的组成形式通常是 ☒ 个「专家网络」{☒_1, … , ☒_☒} 搭配一个「门控网络」G**。这个门控网络的形式通常是一个使用 softmax 激活函数的线性网络，其作用是将输入引导至合适的专家网络。**MoE 层的放置位置是在 Transformer 模块内，作用是选取前向网络（FFN），通常位于自注意力（SA）子层之后**。这种放置方式很关键，因为随着模型增大，FFN 的计算需求也会增加。举个例子，在参数量达到 5400 亿的 PaLM 模型中，90% 的参数都位于其 FFN 层中

门控函数（也被称为路由函数或路由器）是所有 MoE 架构的基础组件，其作用是**协调使用专家计算以及组合各专家的输出**。根据对每个输入的处理方法，该门控可分为三种类型：**稀疏式、密集式和 soft**

式。其中稀疏式门控机制是激活部分专家，而密集式是激活所有专家，soft 式则包括完全可微方法，包括输入 token 融合和专家融合

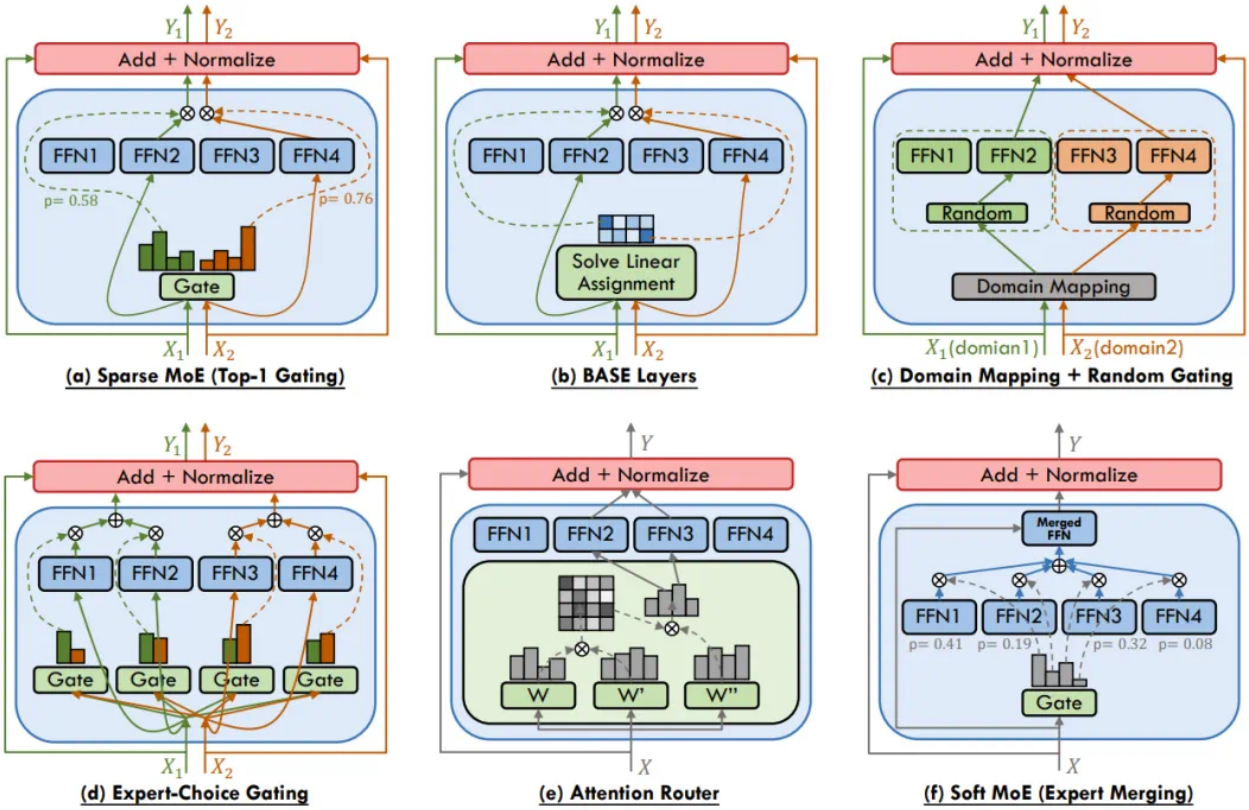


Fig. 4. The illustration of various gating functions employed in MoE models, including (a) sparse MoE with top-1 gating [49], (b) BASE layers [87], (c) the combination of grouped domain mapping and random gating [127], (d) expert-choice gating [193], (e) attention router [166], and (f) soft MoE with expert merging [105].

1.6.2 解码策略

大模型在输出token的时候的几种策略

- **Greedy Search：**贪心的**每次选择概率最大的token（词）**，简单高效，但是可能会导致生成的**文本过于单调和重复**
- **Beam Search：维护一个大小为 k 的候选序列集合**，每一步从每个候选序列的概率分布中选择概率最高的 k 个单词，然后保留总概率最高的 k 个候选序列。这种方法可以平衡生成的质量和多样性，但是可能会**导致生成的文本过于保守和不自然**
- **Top-K Sampling：从排名前 k 的 token 中进行抽样**，允许其他分数或概率较高的token 也有机会被选中。在很多情况下，这种抽样带来的随机性有助于提高生成质量。但是将采样池限制为固定大小 *K* 可能会在**分布比较尖锐的时候产生胡言乱语，**而在**分布比较平坦的时候限制模型的创造力**
- **Top-P Sampling：**在**累积概率超过设定值P的最小单词集中采样**
- **Temperature Sampling：**加在Softmax上的温度系数，用来调节大模型输出token的概率分布的平坦程度，**越大概率分布越平坦，输出越随机，越小概率分布越极端，输出越稳定**
- **KPT：** 一种综合的方法，先后进行 top-k->top-p->Temperature 采样

2. Pre-training 预训练

大模型预训练的主要目的是为了让模型掌握通用能力，产生一个base模型，采用的方法是在大量语料上进行无监督学习，具体训练目标是**Next Token Prediction loss**：

$$L = -\sum_{n=1}^{N} \log p(x_n | x_1, x_2, \ldots, x_{n-1}; \theta)$$

其基本思想是，模型根据上下文预测下一个最可能的单词，Next Token Prediction Loss 通过计算每个预测的对数似然损失，帮助训练模型进行更准确的下一个单词预测

2.1 预训练数据

整个预训练阶段最终的就是数据的获取和处理

2.1.1 数据来源

预训练阶段涉及学习大量数据，以全面了解世界及其各种复杂性。这不仅包括基本的语言能力，还包括诸如math、code和逻辑推理等高级技能，所以需要收集大量的多领域的语料，目前一些常见的数据来源有以下几种：

- **Common Crawl**：这是一个开放的数据共享平台，里面是大量的网页数据
- **GitHub**：从 GitHub 上收集代码相关的数据
- **网页文本**：收集与代码相关和与数学相关的网络文本，如选择代码论坛、库网站、数学网站等
- **电子书、教育材料和论文资料**：这些资源涵盖丰富的知识内容，有助于提升模型在不同领域的理解和生成能力
- **内部数据**：做大模型的公司组织的内部相关数据，可能是业务上的垂域数据

一般用到的总体数据量会达到**10T tokens**的量级，此外对于通用base模型来说，**不同语种的语料**也都要收集，主要是中英文，小语种看具体需求收集

2.1.2 数据清洗和去重

数据清洗是预训练处理数据最重要的环节，得到的数据的质量、多样性决定了模型的通用能力效果。

- 利用**模型对 pretrain 数据的质量进行打分**，已经成了数据清洗工作的标配，llama3、qwen2 的技术报告都有提及。同 size 下，**BERT 结构的模型的表征能力是强于 transformer-decoder 模型的**，因此打分模型可以使用BERT类型的模型进行训练，或者利用**强闭源模型**来对训练数据进行打分，比如GPT4o
- 利用启发式规则进行 **pretrain 数据质量筛选，**比如数据长度是否少于某个值，数据中某个 token 的比例超过某个阈值，数据的中文占比、英文占比、数字占比，数据是否有“http”字段，数据是否包含了“新冠”、“疫情”等低质量关键词，数据是否包含某些反动词汇，数据是否包含某些黄色字眼等。用规则清洗或者过滤数据的时候，一定不要把数据搞成分布有偏的数据

- 利用**minhash等技术进行文本去重**，要先**确定需要多少训练数据，再确定去重的粒度**。需要 10T 训练数据，就卡相似度在 80% 的阈值进行去重

以**Qwen**技术报告为例：

- Qwen 的预训练数据清洗流程较为复杂。包括溯源处理、URL 过滤、英文文本提取、文本校正与规范化、文本去重等步骤。首先，对原始 HTML 网页格式进行处理，去除广告、网页导航等无效内容。然后，利用黑名单过滤掉包含不健康内容的域名，同时使用训练关键词过滤工具对 URL 进行过滤。接着，通过语种鉴别工具从原始数据中提取出英文文本，并修正文本中的拼写错误和 Unicode 乱码。最后，进行文本去重，包括基于 HashLSH 的模糊去重和基于 N-Gram 的精确去重，以降低文本的重复率
- 对于 Qwen-Coder，其数据清洗也非常严格。通过细致的数据清理、可扩展的**合成数据**生成和平衡的数据混合，在保持通用性的同时展示了令人印象深刻的代码生成能力。例如，对从 GitHub 获取的代码数据，采用了 **fastText** 进行多级过滤，每级都会筛掉一批数据，从 582B tokens 筛选四级降到 118B。同时，为了 FIM（fill-in-the-middle，根据上下文补全代码）任务引入了一些特殊 token，如 <|repo_name|> 和 <|file_sep|> 等，用于 repo 级别的代码补全
- 数据清洗的过程不仅提高了数据的质量，还为模型的训练提供了更加纯净和有效的数据，有助于提升模型的性能和泛化能力

2.1.3 数据配比和顺序

- 大部分的开源大模型技术报告里，都提及训练的数据是如何配比的，基本上都是“知识 + 代码 + 逻辑”三个大类目，其中知识数据分文中文知识和英文知识，逻辑数据则可以认为是 math 数据和 cot 数据的混合体。整体上，大部分中文模型的配比都差不多为 **中：英：code = 4:4:2**
- 进一步到领域专项模型等训练的话会加入**领域数据**做进一步配比，上述的数据配比一般是先在小模型上实验，然后利用scaling law去推算应用到大模型上的效果，因此一般会在小模型上做多组数据配比实验，观测loss情况
- 大模型在大量预训练数据上学习的时候一般会采用**课程学习**的方式，就是先学什么，再学什么，考虑到大模型在训练过程中会出现遗忘的情况，数据的训练顺序也很重要，这部分也需要先在小模型上进行实验

2.2 预训练流程

2.2.1 训练tokenizer

根据收集到的数据以及具体需求，训练一个 tokenizer。在收集到的数据集以及更大的通用数据集上，利用 BPE / BBPE 算法进行训练：

1. 数字切分（避免 9.9 > 9.11 的问题回答不正确）
2. 控制压缩率，1 个 token 对应多少个汉字：压缩率太低，那就是字太多、词太少，很影响解码效率；压缩率太大，也就是词太多，又会影响模型的知识能力。通常，压缩率越低的模型，loss 也会低，大部份中文大模型的1 个 token 会映射成 1.5 个汉字左右

3. 手动移除脏的、敏感token
4. 如果提前知道自己的业务场景，那就应该补充业务场景对应的 token，增加业务场景文本的压缩率，比如医疗场景，就提前把阿莫西林、青霉素等作为一个 token
5. 词表的中、英覆盖率要足够大，至于其他小语种是否要加，则看业务需求
6. tokenizer 的 词表大小vocab_size 和模型的 embedding_size 之间，要有一千个左右的余量，后续的对齐环节，需要大量在 pretrain 阶段没见过的全新 token 来做训练

2.2.2 确定模型结构和参数

模型结构一般就使用**Llama架构**，**也就是RoPE + GQA + RMS_Norm + SwiGLU，模型参数则根据拥有的训练资源进行选择**

2.2.3 训练设置和监测

pretrain 训练流程目前基本是固定的，当训练数据和训练代码都准备就绪后，按照以下四个流程来设置学习率和超参数就可以啦：

- 开局 warmup，学习率缓慢上升到最大
- 中间 cos_decay，学习率比较大，是否 decay 看情况决定
- 后期 改变 RoPE的频率，增加文本长度，开始让模型适应长文本
- 收尾 anneal，用高精数据、IFT数据等来给强化模型的考试能力，做完退火就该上考场刷 benchmark 了

总结一下，一般来说预训练也通常会用两阶段或者多阶段的训练流程，先在全部语料上训练，然后在小部分长文本上训练，最后在小部分高质量数据上退火训练

训练 loss 分析：

1. 不同的语料的loss要分开观察，比如中文知识，英文知识，代码这三类数据的 loss 得分开
2. loss毛刺不可忽视，无论是 loss 突然激增还是突然激减，大概率是数据问题（脏数据不仅 loss 高，也会 loss 低，全是乱码的数据 loss 很高，全是换行符的数据 loss 很低）

2.3 预训练评估

pretrain 的评估相对来说是 llm 全链路评估工作中最简单的环节，因为它并不需要考察模型的指令 follow 能力、安全能力、幻觉现象等，只需要看模型整体的知识掌握程度即可

- **PPL 困惑度：**准备一些百科、逻辑、code 等数据，观察模型在这些测试集合上的 loss 表现，偶尔浮动是正常的，但整体趋势肯定是持续下降，最后趋于稳定的。只要训练集中没这些语料，和训练曲线应该是基本一致的。
- **Benchmark：**一些知识性的开源benchmark评测

- **概率探针：**从概率的角度来监控模型的知识能力有没有遗忘或者提升，适用于要观察模型的某一项具体能力。思路也非常简单，就观察某个 token 的概率是否增加，某个句子的概率是否增加，唯一麻烦的地方是探针测试集往往需要训练者一条一条亲自去构造，而不能批量生成，比如
  a. Prob('北京'｜'中国的首都是')，就看一下这个概率值随着 pretrain 推进是否持续在增大
  b. PPL('台湾属于中国')，这个句子的 ppl 是否持续在增大；PPL('台湾不属于中国')，这个句子的 ppl 是否持续在减小
  c. 对比探针，PPL('尊重同性恋') > PPL('反对同性恋') 是否成立
  d. 指令 follow 能力，Prob('{ ' | '以 json输出')

2.4 继续预训练

Continue Pre-train，一般用于基于base模型去训练一个领域专项的模型，进行领域知识注入，例如code模型，法律模型等等。训练上跟pre-train完全一样，不过用到的数据和量不一样，一般来说也要做数据配比，大概有 **通用语料：领域语料：指令数据=7:2:1**，通用数据可以复用base模型的部分，领域数据需要额外进行收集、筛选等，可以复用base模型数据筛选、清洗和去重的pipeline

3. Post-training 后训练

3.1 SFT 监督微调

Supervised Fine-Tune 与 pretrain 的区别，在训练方式上没有任何区别，主要区别在于

数据的组成形式上：

- pretrain 的每条数据都是满长度，即达到模型设置的输入长度上限，sft 的每条数据原本多长就是多长
- sft 会引入 pretrain 阶段未见过的 special_token，来让它们学习全新的语义
- sft 会让模型见到最重要的 eos_token，pretrain 模型因为没见过该 token 而无法停止生成
- 借助 special_token，sft 会把语料切分成不同的角色，标配的有 system、user、assistant，根据业务需求也可以有“背景”、“旁白”、“事件”等等
- sft 的 prompt 部分不做 loss（不绝对，不一定）。主要原因是 prompt 的同质化比较严重，不做 loss_mask 的话，同样的一句话会被翻来覆去的学，但如果你能保证你的每条 prompt 都是独一无二的，就完全可以省去 prompt 的 loss_mask 环节。多轮对话数据一定要想清楚是每一个 answer 都算 loss，还是只对最后一轮的 answer 算 loss

训练目的也不一样：

- pretrain 是在背书，纯粹的学习知识；sft 则是在做题，学习的是**指令遵循能力**。切勿在 sft 阶段强行给模型做知识注入，比如训个 50W 条的 code 数据，所有的**知识注入工作应该采用 continue-pretrain 的思路进行**，否则都会使得模型的通用能力掉点明显（sft 做知识注入基本上是 100% 某个知识，但 continue-pretrain 做知识注入会控制在 10% ～ 20% 左右的比例）

3.1.1 SFT数据

SFT训练数据的核心是数据多样性和数据质量，数据数量并不重要

- **数据多样性：**

  数据多样性主要包含两个维度，**数据用途**和**数据形式**

  数据用途，也就是 **task_type**，可以结合这几个思路进行数据收集：

  - OpenAI 官网列出了 ChatGPT 擅长的所有任务项，诸如翻译、emoji 聊天……之类的
  - LLM 毕竟是个语言模型，传统的每个 NLP 模型它都应该能胜任，那就把 NER、机器阅读理解、意图识别等传统的 NLP 任务也给模型补充一点，如果已有类似任务就不补充了
  - 参考业务需求，下游业务需要某个特殊场景的任务，那就让 sft 阶段提前见一见，这种数据的典型代表就是过年前给模型灌一些对春联、猜灯谜的的数据。只要数据质量没问题，一般都不会破坏模型能力

  在实际工作中，双层 task_type 都很常见，比如“逻辑推理 - 常识推理”，“逻辑推理 - cot 多步骤推理” 这种。至于每种 task_type 的数据量，别搞平均主义：**难 task_type 数据多点，简单 task_type 就数据少点**，也要结合自己的 base 模型能力动态调整

  还需要从**数据形式**的角度来兼顾数据的多样性：

  - **prompt 表达方式多样性**，不要千篇一律的“把中文句子 A 翻译成英文”，也要适当有一些“我在英国旅游，我现在需要向路人问路，我想表达 A 的意思，该怎么说”，“我是一个英文老师，我需要向我的学生讲解句子 A 用英文怎么写，请你用最正宗的表达方式帮我完成。”这么做的目的是防止模型只认识 prompt 中的几个关键 token，进而导致训练过拟合或者泛化性变差
  - **prompt 长度均衡**，既要有短数据，也要有长数据，避免模型的 attention 退化到无法聚焦长 prompt。长数据还不能是字面意思的长，要有那种关键信息藏在 开头 / 中间 / 结尾 的各种数据场景，避免模型在训练时偷懒，只对 prompt 的起始 token 或结束 token 有 attention
  - **answer 长度均衡**，不能让模型没出输几个 token 就停止，适当的有一些语料让它学会输出尽量长的 answer，否则模型会很难 follow “不少于2000字” 这种指令
  - **多轮聊天的切换 topic 能力**，也就是说，有的数据当前 query 是和 session 有关系的，有的数据则是当前 query 和 session 毫无关系，要让模型自己学会判断 query 是否和 session 有关
  - **answer 分布的多样性**，这最重要，千万别总共一万条训练数据，一千条数据的 answer 都说同一句话，answer 可是算 loss 的，太单一的话会严重让模型过拟合

  概括起来，所有的数据形式多样性都可以总结为一句话：**数据形式不能让模型找到规律，关键信息在 prompt 中的位置分布要足够随机**。目的是避免模型在训练时退化，只聚焦于某些或某些位置的 token，而不是聚焦于完整的 prompt

- **数据生产：**

  生产 prompt

- 比较有名的工作**Self-Instruct**，给每个 task_type 准备一些 seed prompt，然后随机采样 seed，在喂给一个能力很强的模型，让它基于这些 seed 问题再续写出一些问题
- 利用各种**启发式规则合成数据造 prompt**，按照数据多样性，去搜集不同 task_type 的数据集集合，然后适当做改写，改写也可以用强大模型来做

生产 answer

- **GPT4 is all you need**，利用一个效果好的模型来生产 answer是不二之选，不在乎成本，就选 GPT4 / Claude 3，在乎成本，就在自己的机器上部署 Qwen_72B / deepseek_MOE
- **小模型 + SFT ≈ GPT4 + zero_shot / few_shot / cot**（复杂指令和逻辑推理可能不行）。对于某种**特殊需求的 task_type**，利用 GPT4 生产一千条 answer，然后去训小模型，再利用小模型去预测出上万条数据

- **专项数据：包括长文本、Agent/function_call、RAG SFT数据**

3.1.2 SFT训练

一般在资源充足的情况下都是做全参微调的SFT。小模型大学习率，大模型小学习率，epoch 基本上就是 1～3个，数据是 10W 条级别左右，稍点多点都行，但少不能少于 1W，多也不能到达 100W；起始训练适当做点 warmup，几种主流的 lr_scheduler 都试一下，gradient_accumulation_steps 是个比较重要的参数，就 16 / 32 / 64 / 128 等数字都尝试下；按需求开 dropout

loss 曲线需要重点留意几个细节：

- **不同 task_type 要有不同的 channel_loss，分别观察**
- special_token 的 loss 一开始会有点高，但是下降也是很快
- 创作类任务的 loss 会比其他任务的 loss 更高一点，这个现象很 make sense，答案越固定，搜索结果越单一的语料，loss 越低，反之亦然
- 只要训练语料是通用数据，且数据进行了采样，那么模型的初始 loss 就不会特别高，7B / 13B 可能在 2 左右，数据难了也有可能到 3，72B 则大多在 1 ～ 2 之间这个水平状态；最终 loss 则大概在 0.5 左右，根据语言模型定义，如果 loss 更低，那基本上模型只会说这一句话了，别的 token 都没概率了
- 如果 loss 持续升高，不要对自己的训练数据产生任何质疑，想着是不是数据太难了不好学之类，这就是训练代码有问题。**next_token_prediction 的训练方式就是在背书，它不存在学不会的情况，只存在学会了但不会泛化的情况**。就算是一堆随机乱码，模型啥也学不到，也应该是 loss 持平，而不是 loss 升高

3.1.3 SFT评估

事先准备好一个高质量的评测集合，这个评测集合要和 sft 训练集合一样有**明确的 task_type**

不同于 pretrain 的评估只需要看知识能力，sft 的评估是需要看经典的 3H 原则的：**Helpfulness、Honesty、Harmlessness**，或者按需求制定指标：指令遵循，内容准确，是否产生幻觉，是否安全等等

评估的时候，每个维度有一个单独的得分，根据相应的加权公式来确定这条回复的可用性，当模型在某个 case 上的得分变低的时候，可以比较直观的看出是哪个维度变差了，结合训练数据做 case 分析

评估方式目前基本是两种：

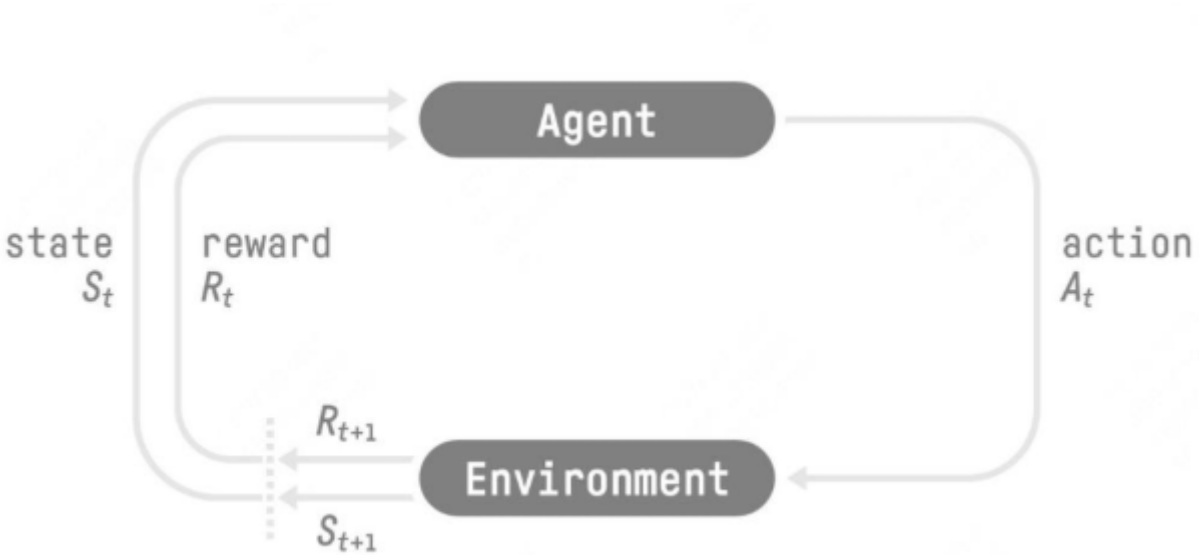
- **机评：在利用 GPT4、Claude进行评估的时候，prompt一定要仔细设计**。大模型的评估是有明显偏好的，对比评估的时候，A 和 B 倾向于选 A，长句子和短句子倾向于选长的；打分评估的时候，一个正确的 answer 让模型打分三次，很可能就分别是 3、4、5分。Alignbench 的机评 prompt可以进行仿照，此外给模型一个参考答案很多时候能让模型的打分更准一些，毕竟模型会考虑“候选 answer”和 “gold answer”的相似度来进行打分
- **人评：字面意思**

3.2 RLHF 基于人类反馈的强化学习

Reinforcement Learning from Human Feedback/Preference-based Reinforcement Learning

3.2.1 RL基础概念

1. **强化学习流程**



强化学习是机器通过与环境交互来实现目标的一种计算方法。机器和环境的一轮交互是指，机器在环境的一个状态下做一个动作决策，把这个动作作用到环境当中，环境发生相应的改变并且将相应的奖励反馈和下一轮状态传回机器。这种交互是迭代进行的，目标是**最大化在多轮交互过程中获得的累积奖励的期望**。强化学习用智能体（agent）这个概念来表示做决策的机器。相比于有监督学习中的“模型”，强化学习中的“智能体”强调机器不但可以感知周围的环境信息，可以通过做决策来直接改变这个环境，而不只是给出一些预测信号。**一般来说在经典的强化学习环境中agent的实现可以用一些简单的MLP、RNN、CNN等神经网络实现，与现在流行的LLM-based agent有区别**

智能体和环境之间具体的交互方式如图所示。在每一轮交互中，智能体感知到环境目前所处的状态，经过自身的计算给出本轮的动作，将其作用到环境中；环境得到智能体的动作后，产生相应的

即时奖励信号并发生相应的状态转移，智能体则在下一轮交互中感知到新的环境状态直到到达终止状态

智能体在这个过程中学习，它的最终目标是：**找到一个策略，这个策略根据当前观测到的环境状态和奖励反馈，来选择最佳的动作**

2. **价值函数和优势函数**

奖励值 R_t 是在状态 S_t 下的即时奖励，想要表示一个状态的好坏可以用**考虑总的期望收益的价值函数**来描述：

$$V(s) = \mathbb{E}_\pi[R_t + \gamma V(S_{t+1})|S_t = s]$$

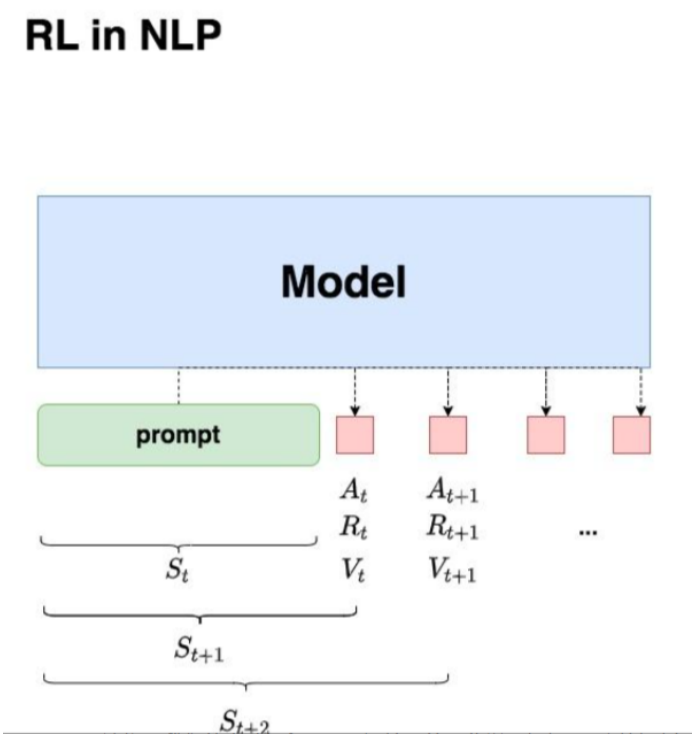
其中 γ 是折扣因子。优势函数 Adv 是用来描述在一个状态下执行了一个动作之后，状态转移到一个确定的状态，相较所有动作的均值 $V(S_t)$ 的优势：

$$Adv_t = R_t + \gamma V(S_{t+1}) - V(S_t)$$

此处不过多展开强化学习相关概念，不然就太长了

3. **RL in NLP**

强化学习要应用到NLP任务中，对应的**agent、环境、状态和动作**都需要明确的定义



给定一个prompt，大模型会**在 t 时刻生成一个token，然后下一个时刻根据prompt+上一时刻的token再去生成下一个token**，进行自回归的生成，所以可以定义强化学习中的各个概念为：

- **动作 A_t：** 生成的token，对应这个动作模型这边会有一个策略policy，也就是生成token的概率分布
- **状态 S_t：** 上文以及 t 时刻前生成出来的所有token加起来
- **奖励 R_t 和价值 V_t：** 这里的定义就是一般强化学习的定义，即时奖励以及价值函数，都是对于NLP中的状态和动作来说的

3.2.2 RLHF流程

深度学习RLHF首次在2017年的论文 ***Deep Reinforcement Learning from Human Preferences*** 中被提出，一开始是用于解决**复杂强化学习任务中环境奖励函数设计问题**的

- **研究背景：**
  - 强化学习在许多任务中面临**目标复杂、难以定义奖励函数**的问题，导致**难以将人类实际目标传达给智能体**
  - 不正确的、有偏的奖励函数会导致**智能体过分利用(exploit)奖励函数**，产生**reward hacking**问题，即**实际学到的行为与人类期望不符合，甚至有害**
  - 奖励函数的设计工程需要**大量的专业人士的精力**
  - 现有方法如**逆强化学习和模仿学习在处理复杂行为时存在局限性，直接使用人类反馈作为奖励函数成本过高**
- **研究目标：**用于**解决没有明确定义奖励函数的强化学习**问题，需要满足以下几点：
  - 能够解决那些人类**只能识别期望行为**，但**不一定能提供demonstration的任务**
  - 允许**非专家用户对智能体进行教导**
  - 能够**扩展到大型问题**
  - 在**用户反馈方面经济高效**
- **研究方法：**算法将**奖励函数与人类偏好进行拟合**，同时**用RL算法训练一个策略来优化当前预测的奖励函**。**给人类提供两个智能体行为轨迹的片段(一般来说是视频、动图)，让人给出自己的偏好标签(就是那个片段更好)，而不是提供绝对数值分数**。在某些领域中，人类更容易进行比较，而且这种比较对于学习人类偏好同样有用。比较智能体轨迹片段与比较单个状态几乎一样快，由此产生的比较结果明显更有帮助

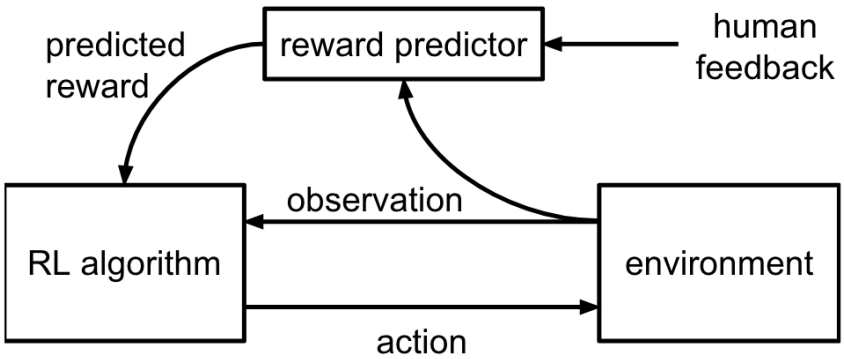


Figure 1: Schematic illustration of our approach: the reward predictor is trained asynchronously from comparisons of trajectory segments, and the agent maximizes predicted reward.

  - **对比标签：**对于智能体轨迹片段 σ^1 和 σ^2 来说，下面的式子表示 σ^1 比 σ^2 更被人偏好，得到的标签 y 也可以表示如下，0.5代表同等偏好程度。 o, a 分别表示智能体的观测和动作

$$\sigma^1 \succ \sigma^2 = \left(\left(o_0^1, a_0^1\right), \ldots, \left(o_{k-1}^1, a_{k-1}^1\right)\right) \succ \left(\left(o_0^2, a_0^2\right), \ldots, \left(o_{k-1}^2, a_{k-1}^2\right)\right)$$
$$y = \{0, 1, 0.5\} \text{ if } \{\sigma^1 \succ \sigma^2, \sigma^2 \succ \sigma^1, \sigma^1 = \sigma^2\}$$

- **偏好建模：**由于RLHF的一个目标是将**奖励函数与人类偏好进行拟合，**就是**利用人类的比较偏好标签来学出一个reward model**，那就涉及到了奖励函数和偏好之间的关联问题，这里给出的方法是，将奖励函数视为解释人类判断的潜在因素，并**假设人类偏好一个片段的概率与潜在奖励在该片段长度上的总和呈指数相关**，基于 **Bradley-Terry 模型**，可以给出**人类偏好片段 σ^1 超过 σ^2 的概率**：

$$\hat{P}[\sigma^1 \succ \sigma^2] = \frac{\exp \sum \hat{r}(o_t^1, a_t^1)}{\exp \sum \hat{r}(o_t^1, a_t^1) + \exp \sum \hat{r}(o_t^2, a_t^2)}$$

- **奖励学习：**得到这个偏好建模以及收集到的人类偏好标签之后，就可以简单的使用**二分类的思路来隐式的学习我们的奖励函数**了，损失函数是分类常用的**交叉熵，**然后利用这个loss训练优化得到最后的符合人类偏好的奖励函数

$$\text{loss}(\hat{r}) = -\sum_{(\sigma^1, \sigma^2, y) \in \mathcal{D}} y(\sigma^1 \succ \sigma^2) \log \hat{P}[\sigma^1 \succ \sigma^2] + y(\sigma^2 \succ \sigma^1) \log \hat{P}[\sigma^2 \succ \sigma^1]$$

- **策略学习：**得到奖励函数之后就可以应用任何一个强化学习算法去最大化奖励用来产出相应的策略了
- **在线学习：**这篇文章提出的方法是在线RLHF(Online RLHF)，就是奖励函数和策略学习是交替同时进行的，伴随着智能体不断的和环境交互产生新的轨迹数据用来给人类打反馈标签

- **总结：**
  - **算法原理：**算法通过将奖励函数与人类偏好进行拟合，**使智能体的行为朝着符合人类期望的方向发展**。在训练过程中，同时优化策略以最大化预测的奖励，从而在没有明确奖励函数的情况下，让智能体学会做出符合人类偏好的决策
  - **反馈方式优势：**
    - **易于提供：**相比提供绝对数值分数，**人类更容易对智能体轨迹片段进行比较**，降低了反馈的难度，使得非专家用户也能更轻松地参与到智能体的训练过程中
    - **信息丰富：**智能体轨迹片段包含了一定的行为序列信息，比单个状态更能反映智能体的行为特点和趋势，因此在学习人类偏好方面更有帮助，能够**为奖励函数的拟合提供更有价值的信息**
    - **在线反馈的好处：**在线收集反馈意味着系统可以**实时获取人类的偏好信息，并根据新的反馈及时调整策略和奖励函数**。这样可以**避免系统过度依赖之前学习到的奖励函数，防止因奖励函数的不准确性或局限性而导致的不良行为**，从而持续提高系统的性能，使其更好地适应复杂多变的任务环境；缺点就是**实时收集人类偏好标签成本很高**，所以之后在LLM中应用的时候，很多工作都在研究**自动化偏好标签，比如RLAIF，利用大模型代替人类给偏好**

3.2.3 PPO算法

PPO是一种**Actor-Critic架构**的强化学习算法，其中Actor就是执行动作的策略，用来跟环境交互产生轨迹数据，Critic则是用来评估Actor访问状态、动作执行的好坏的模型，并给出指导方向

PPO有两个版本，PPO-penalty和PPO-clip，常用的PPO-clip的优化目标如下：

$$\arg\max_{\theta} \mathbb{E}_{s\sim\nu^{\beta},a\sim\pi_{\theta_k}(\cdot|s)}\left[\min\left(\frac{\pi_\theta(a\mid s)}{\pi_{\theta_k}(a\mid s)}Adv^{\pi_{\theta_k}}(s,a),\text{clip}\left(\frac{\pi_\theta(a\mid s)}{\pi_{\theta_k}(a\mid s)},1-\epsilon,1+\epsilon\right)Adv^{\pi_{\theta_k}}(s,a)\right)\right]$$

其中 π_θ 为Actor的策略， Adv 为Critic给出的优势函数估计，逆向看一下为什么PPO最后长这个样子，先看一下最简化的版本：

$$\arg\max_{\theta} \mathbb{E}_{s\sim\nu^{\beta},a\sim\pi_{\theta}(\cdot|s)}\left[\pi_\theta(a\mid s)Adv^{\pi_{\theta_k}}(s,a)\right]$$

Critic给出的优势函数指导Actor的策略优化的方向，如果优势函数值是正的，那说明这个动作的价值高于平均，那么最大化这个式子会增大策略在这个状态下采取对应动作的概率；如果优势函数是负数。则说明这个动作的价值低于平均，那么最大化这个式子会降低策略采取这个动作的概率

PPO在相当于此基础上进一步做了一些优化：

- **重要性采样：**上述简化的式子可以看出来PPO是一个on-policy算法，即采样的策略和更新的策略是同一个策略（动作是从 π_θ 采样出来的，优化的也是 π_θ ），但是这样存在一个问题就是**样本利用效率太低**，所以PPO利用了重要性采样技术，这样就可以利用之前迭代的策略 π_{θ_k} 产生的数据了

  > 📌 **重要性采样**
  >
  > 1. 设我们要估计函数h(x)关于目标分布p(x)的期望 $E_p[h(x)]=\int h(x)p(x)dx$
  > 2. 假设我们有一个提议分布q(x)，且q(x) > 0，只要p(x) > 0，我们可以将期望改写为：
  >
  > $$E_p[h(x)]=\int h(x)\frac{p(x)}{q(x)}q(x)dx$$

- **策略截断：**PPO实际上是基于TRPO算法改进的，TRPO给优化目标引入了当前策略和历史策略的**KL散度约束，规范策略优化不要太过偏离上一个迭代回合的策略**，即策略更新的步长通过KL散度进行调节，但是TRPO用了很复杂的数学方法去估计KL散度的值，PPO这里直接进行固定的策略截断，也就是clip操作，将当前策略的更新的幅度控制在 $[1-\epsilon,1+\epsilon]$ 之间。**PPO-Clip 其实就是通过设置一个“安全区间”，确保策略的更新不会偏离太远，每次策略更新时，PPO 会计算新策略与旧策略的差异，如果这个差异过大，PPO 会将其“裁剪”回一个合理的范围内**

- **GAE估计：**PPO采用了**广义优势估计（Generalized Advantage Estimation），**传统的优势函数估计通常基于**单步的 TD 误差（Temporal Difference Error）或使用蒙特卡洛方法（Monte Carlo Methods）计算的回报**。GAE用于**减少策略梯度方法中的方差，同时保持低偏差**，从而**提高算法的样本效率和收敛速度**

  根据**多步时序差分**的思想，有：

$$
\begin{aligned}
A_t^{(1)} &= \delta_t = -V(s_t) + r_t + \gamma V(s_{t+1}) \\
A_t^{(2)} &= \delta_t + \gamma\delta_{t+1} = -V(s_t) + r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2}) \\
A_t^{(3)} &= \delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2} = -V(s_t) + r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 V(s_{t+3}) \\
&\cdots \\
A_t^{(k)} &= \sum_{i=0}^{k-1}\gamma^i\delta_{t+i} = -V(s_t) + r_t + \gamma r_{t+1} + \cdots + \gamma^{k-1} r_{t+k-1} + \gamma^k V(s_{t+k})
\end{aligned}
$$

然后，GAE将这些**不同步数的优势估计进行指数加权平均**：

$$
\begin{aligned}
A_t^{\mathrm{GAE}} &= (1-\lambda)(A_t^{(1)} + \lambda A_t^{(2)} + \lambda^2 A_t^{(3)} + \cdots) \\
&= (1-\lambda)(\delta_t + \lambda(\delta_t + \gamma\delta_{t+1}) + \lambda^2(\delta_t + \gamma\delta_{t+1} + \gamma^2\delta_{t+2}) + \cdots) \\
&= (1- \\
&\lambda)\left(\delta_t(1 + \lambda + \lambda^2 + \cdots) + \gamma\delta_{t+1}(\lambda + \lambda^2 + \lambda^3 + \cdots) + \gamma^2\delta_{t+2}(\lambda^2 + \lambda^3 + \lambda^4 + \cdots) + \cdots\right. \\
&= (1-\lambda)\left(\frac{\delta_t}{1-\lambda} + \frac{\gamma\delta_{t+1}}{1-\lambda} + \frac{\gamma^2\delta_{t+2}}{1-\lambda} + \cdots\right) \\
&= \sum_{l=0}^{\infty}(\gamma\lambda)^l\delta_{t+l}
\end{aligned}
$$

其中， $\lambda \in [0,1]$ 是在GAE中额外引入的一个超参数。当其为0时，也是仅仅只看一步差分得到的优势：

$$
A_t^{\mathrm{GAE}} = \delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)
$$

当其为1时，则是看每一步差分得到优势的完全平均值:

$$
A_t^{\mathrm{GAE}} = \sum_{l=0}^{\infty}\gamma^l\delta_{t+l} = \sum_{l=0}^{\infty}\gamma^l r_{t+l} - V(s_t)
$$

```python
def compute_advantage(gamma, lmbda, td_delta):
    td_delta = td_delta.detach().numpy()
    advantage_list = []
    advantage = 0.0
    for delta in td_delta[::-1]:
        advantage = gamma * lmbda * advantage + delta
        advantage_list.append(advantage)
    advantage_list.reverse()
    return torch.tensor(advantage_list, dtype=torch.float)
```

ppo优点：**稳定性强，实现容易，适合高维，平衡探索和利用，无约束优化等**

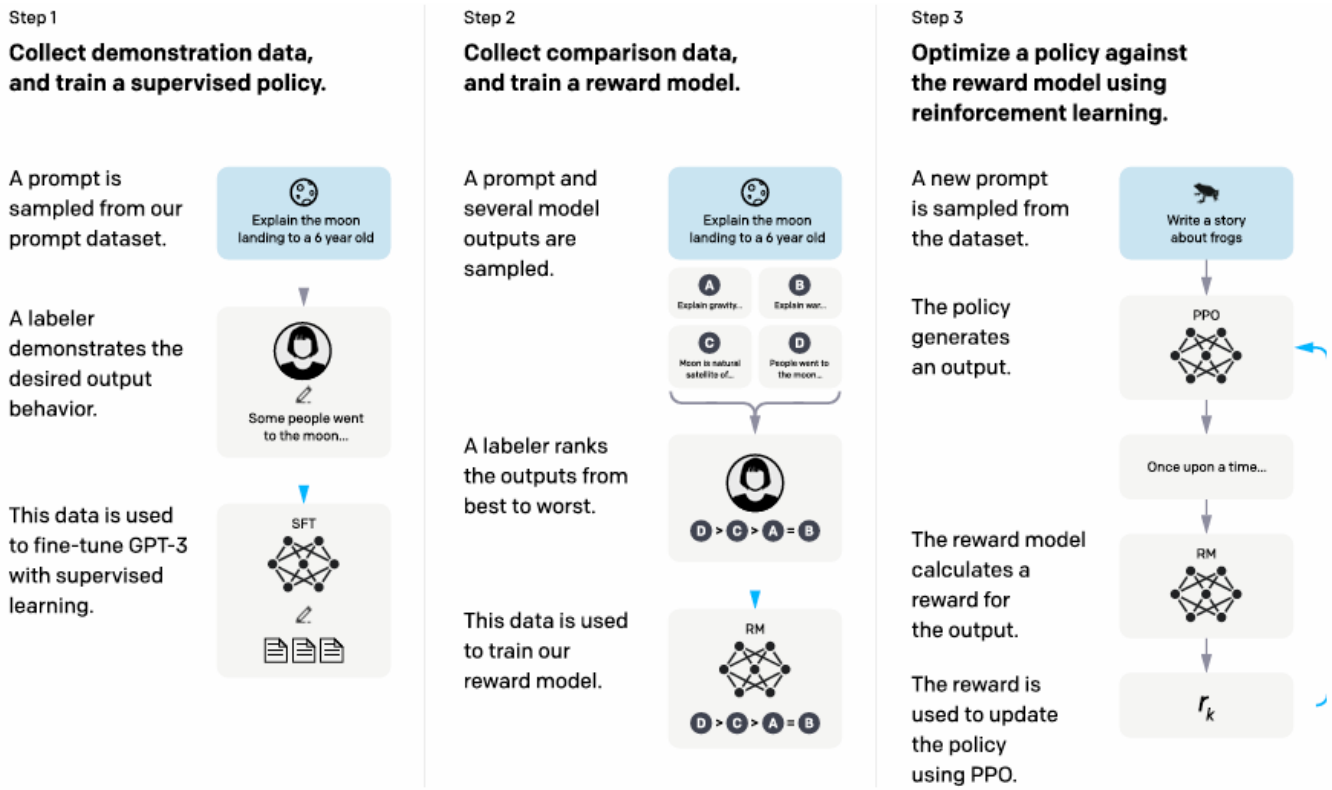
```
td_target = rewards + self.gamma * self.critic(next_states) * (1 - dones)
td_delta = td_target - self.critic(states)
advantage
=rl_utils.compute_advantage(self.gamma,self.lmbda,td_delta.cpu()).to(self.devic
e)
```

```
old_log_probs = torch.log(self.actor(states).gather(1,actions)).detach()
for i in epochs:
    log_probs = torch.log(self.actor(states).gather(1, actions))
    ratio = torch.exp(log_probs - old_log_probs)
    surr1 = ratio * advantage
    surr2 = torch.clamp(ratio, 1 - self.eps, 1 + self.eps) * advantage  # 截断
    actor_loss = torch.mean(-torch.min(surr1, surr2))  # PPO损失函数
    critic_loss = torch.mean(F.mse_loss(self.critic(states),
td_target.detach()))
```

3.2.4 LLM对齐中的RLHF+PPO

1. **InstructGPT**

现在又看回来具体在LLM的对齐中是怎么应用上面那一套RLHF的流程和PPO算法的，OpenAI最早在**InstructGPT**中将RLHF技术应用在LLM post-training中，使模型真正与人类意图对齐



InstructGPT的整个流程是先基于GPT-3进行SFT训练，增强模型的指令遵循能力，然后人力给数据打偏好标签，得到的偏好数据利用前面提到的RLHF的偏好建模训练一个Reward Model，最后再应用PPO算法最大化这个Reward Model提供的奖励值来基于SFT模型训练得到InstructGPT，整个过程有几个值得注意的地方：

- **Reward Model是在移除了最后非嵌入层(unembedding)的 SFT 模型的基础上训练的**，使其接收一个prompt和response，并输出一个标量reward值

- 图中可以看到，**对于同一个prompt，人工可能需要打标签的response不止两个**，也就是需要给多个回答排序，然后InstructGPT的做法是通过排序标签，再将其转换为 C_K^2 个两两对比的偏好标签
- InstructGPT在**实现PPO的时候对强化学习的目标进行了修改**，原本的强化学习的目标是**最大化奖励**：

$$\text{objective}(\phi) = \mathbb{E}_{(x,y)\sim D_{\pi_\phi^{\text{RL}}}}\left[r_\phi(x,y)\right]$$

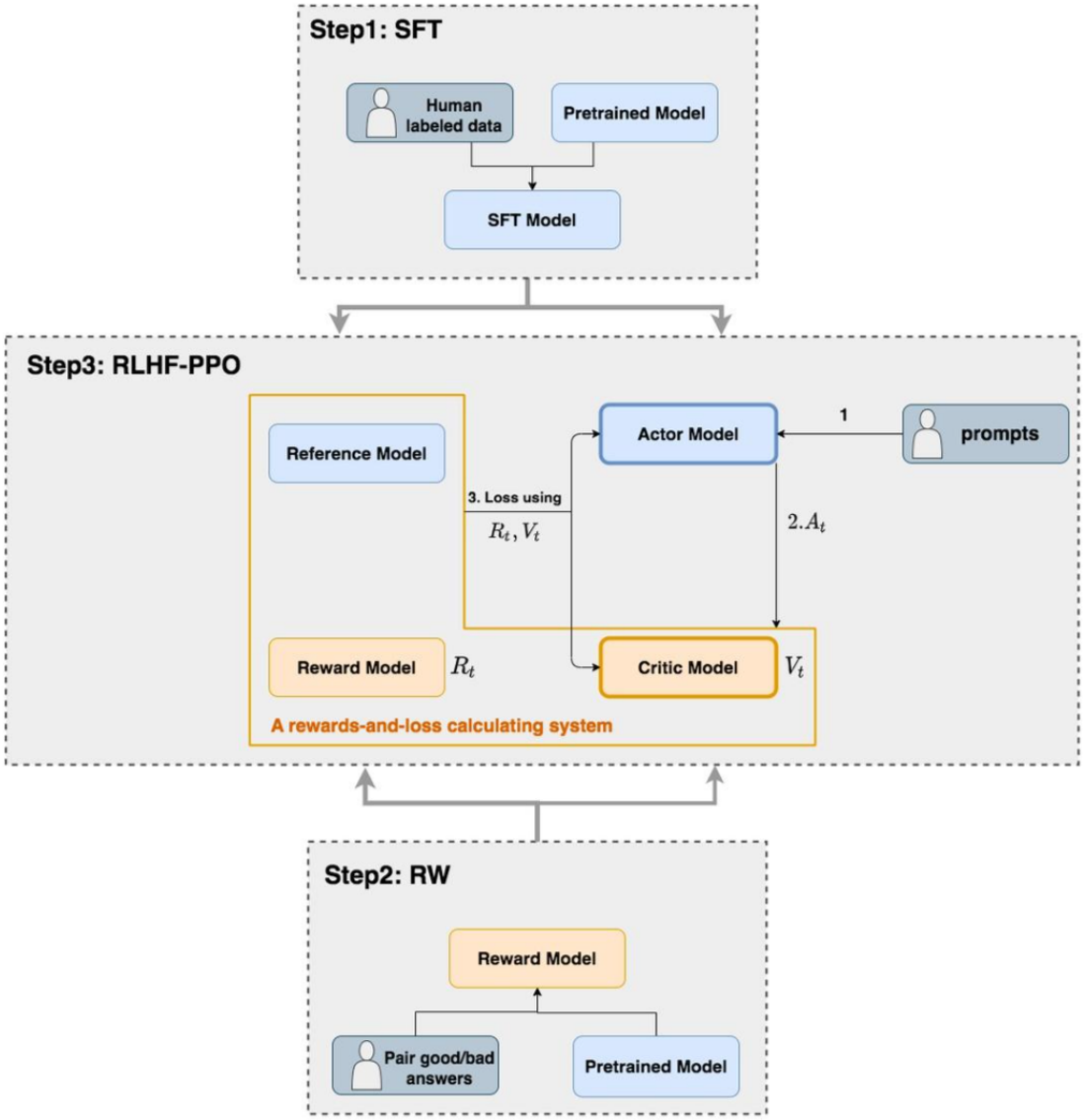
InstructGPT在此基础上，参考之前的一些文献，对奖励函数加上了**RL policy和SFT policy的per-token KL散度来防止RL policy的过优化**，导致偏离SFT模型太远。这样的操作最常见在**Behavior-Regularized Offline RL（行为约束的离线强化学习）**中：

$$\text{objective}(\phi) = \mathbb{E}_{(x,y)\sim D_{\pi_\phi^{\text{RL}}}}\left[r_\phi(x,y) - \beta \log\left(\frac{\pi_\phi^{\text{RL}}(y \mid x)}{\pi^{\text{SFT}}(y \mid x)}\right)\right]$$

形式上等同于重新设计了奖励函数，其中包含了行为约束项，使得RL策略的行为不要太偏离SFT模型。除此之外，InstructGPT还加入了**pretrain数据的梯度**，来修复在公开NLP数据集上的性能回归：

$$\text{objective}(\phi) = \mathbb{E}_{(x,y)\sim D_{\pi_\phi^{\text{RL}}}}\left[r_\phi(x,y) - \beta \log\left(\frac{\pi_\phi^{\text{RL}}(y \mid x)}{\pi^{\text{SFT}}(y \mid x)}\right)\right] + \gamma \mathbb{E}_{x\sim D_{\text{pretrain}}}\left[\log\left(\pi_{\text{RL}}^{\phi}(x)\right)\right]$$

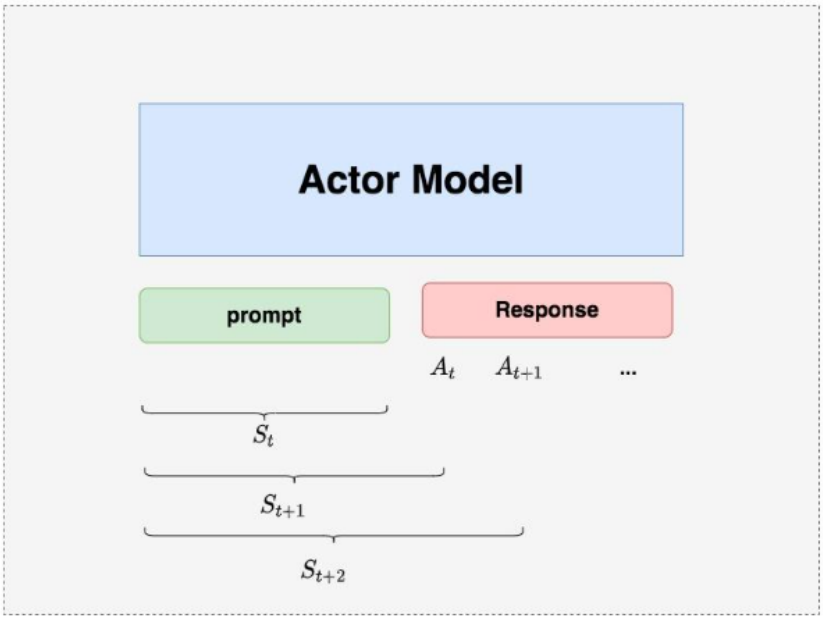
2. **模型和Loss细节**



整体的流程中包含**Actor、Critic、Reward和Reference(SFT)四个模型**

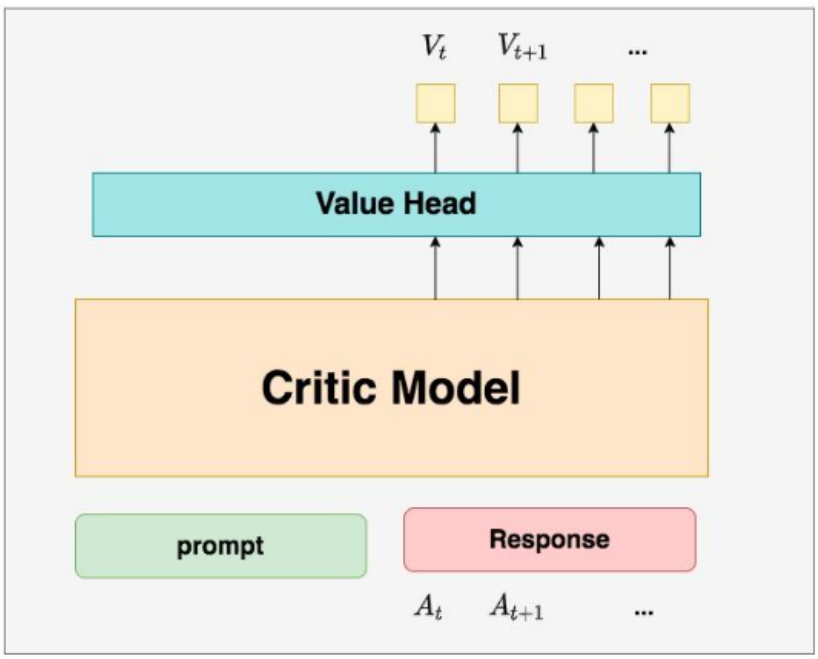
- **Actor Model**：input一条prompt （batch_size = 1），output对应的response，再将prompt + response去计算得到loss，用于更新actor

$$\text{actor_loss} = -\min\left(Adv_t \cdot \frac{P(A_t|S_t)}{P_{\text{old}}(A_t|S_t)}, Adv_t \cdot \text{clip}\left(\frac{P(A_t|S_t)}{P_{\text{old}}(A_t|S_t)}, 0.8, 1.2\right)\right)$$

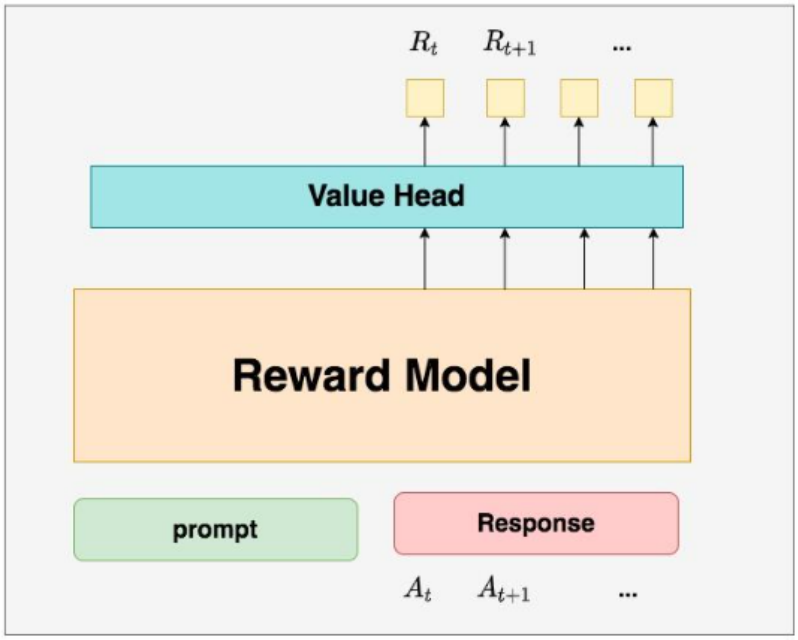


- **Critic Model**：用于预测期望总收益 V_t ，初始化方式有很多种，例如**和Actor共享部分参数**、从**Reward Model初始化**而来。Reward/Critic模型在最后一层增加了一个Value Head层，该层是个简单的线形层，用于将原始输出结果映射成单一的值

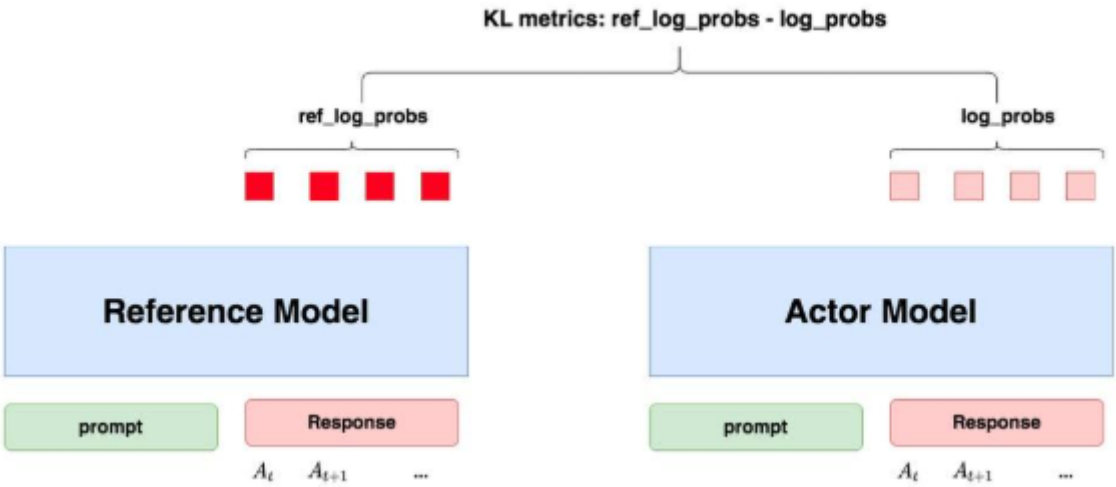
$$\text{Critic_loss} = (R_t + \gamma \cdot V_{t+1} - V_t)^2$$



- **Reward Model**：在PPO训练过程中Reward Model参数冻结，只用来提供奖励值

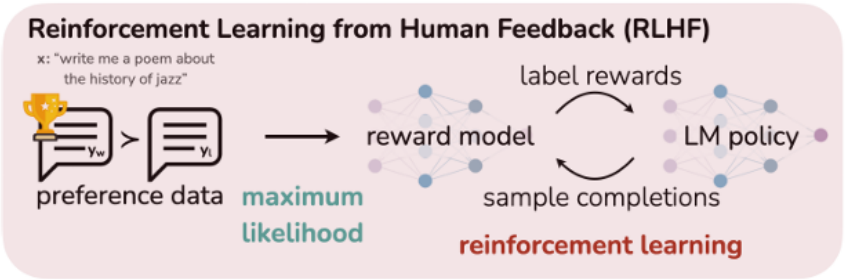


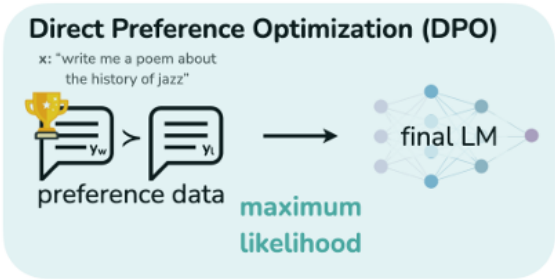
- **Reference Model**：一般也用**SFT阶段得到的SFT模型做初始化**，在PPO训练过程中，它的参数是冻结的



3.3 DPO 直接偏好优化

Direct Preference Optimization





RLHF-PPO存在几个缺点：

- **两阶段训练带来的信息损失：**RLHF的过程是先利用偏好数据训练一个奖励函数模型，然后再用PPO或者其他强化学习算法训练最后的策略。这个过程中如果**奖励函数模型学习存在偏差**，比如奖励实际上并没有和人类偏好对齐的很好，那么后续的强化学习过程也会导致策略陷入次优
- **PPO算法带来的额外训练资源需求：**强化学习的训练会伴随着探索和利用（explore and exploit）的过程，一般会较不稳定，PPO算法在工程实现上利用了很多trick去保障训练的稳定和收敛。但是PPO仍然引入了Actor、Critic、Reward和Reference四个模型，在传统强化学习环境中，Actor和Critic以及Reference都是简单的网络实现，reward是环境自带的人为设计好的，所以并不存在大规模的资源需求，而到了LLM这里，所有的模型都是基于LLM（SFT）模型初始化或者改进的，那么即使在PPO训练过程中只有Actor和Critic需要更新参数，四个模型的推理和训练就需要大量的计算资源，以及四个模型也会带来更多的累积误差

DPO**基于 Bradley-Terry 偏好模型假设，以及强化学习的目标，推导得出 DPO 目标函数**，去掉了Reward和Critic两个模型，优化了对齐的流程，只需要端到端的一步就能从偏好数据到最终策略

💡 **DPO的推导流程：**

让我们从**带行为约束的强化学习(Behavior-Regularized RL)**的目标出发：

$$
\begin{aligned}
&\max_{\pi} \mathbb{E}_{x\sim\mathcal{D},y\sim\pi}[r(x,y)] - \beta D_{\mathrm{KL}}[\pi(y|x)\|\pi_{\mathrm{ref}}(y|x)] \\
&= \max_{\pi} \mathbb{E}_{x\sim\mathcal{D}}\mathbb{E}_{y\sim\pi(y|x)}\left[r(x,y) - \beta\log\frac{\pi(y|x)}{\pi_{\mathrm{ref}}(y|x)}\right] \\
&= \min_{\pi} \mathbb{E}_{x\sim\mathcal{D}}\mathbb{E}_{y\sim\pi(y|x)}\left[\log\frac{\pi(y|x)}{\pi_{\mathrm{ref}}(y|x)} - \frac{1}{\beta}r(x,y)\right] \\
&= \min_{\pi} \mathbb{E}_{x\sim\mathcal{D}}\mathbb{E}_{y\sim\pi(y|x)}\left[\log\frac{\frac{1}{Z(x)}\pi(y|x)\exp\left(\frac{1}{\beta}r(x,y)\right)}{\frac{1}{Z(x)}\pi_{\mathrm{ref}}(y|x)} - \log Z(x)\right]
\end{aligned}
$$

其中 $Z(x) = \sum_y \pi_{\mathrm{ref}}(y|x)\exp\left(\frac{1}{\beta}r(x,y)\right)$ 是配分函数

> 暂时将其理解为是作者看出来有一个这种形式可以提取出来，实际上这个配分函数在**逆强化学习、离线强化学习中大量存在**，涉及到比较麻烦的理论推导，这里就不展开解释了

注意到这个配分函数只是 π_{ref} 和 x 的函数（x, y 是prompt和response）不依赖于 π ，所以可以定义：

$$
\pi^*(y|x) = \frac{1}{Z(x)}\pi_{\mathrm{ref}}(y|x)\exp\left(\frac{1}{\beta}r(x,y)\right)
$$

可以证明**上述 π^* 的满足概率分布大于0且和为1的定义，是一个合法的概率分布**，那么原始的目标可以写成

$$
\begin{aligned}
&\min_{\pi} \mathbb{E}_{x\sim\mathcal{D}}\left[\mathbb{E}_{y\sim\pi(y|x)}\left[\log\frac{\pi(y|x)}{\pi^*(y|x)}\right] - \log Z(x)\right] = \\
&\min_{\pi} \mathbb{E}_{x\sim\mathcal{D}}\left[D_{\mathrm{KL}}(\pi(y|x)\|\pi^*(y|x)) - \log Z(x)\right]
\end{aligned}
$$

那么可以直接从上述目标得到我们的最优策略：

$$\pi(y|x) = \pi^*(y|x) = \frac{1}{Z(x)}\pi_{\text{ref}}(y|x)\exp\left(\frac{1}{\beta}r(x,y)\right)$$

从上式就可以反过来推导出最优策略表示的奖励函数：

$$r^*(x,y) = \beta\log\frac{\pi^*(y|x)}{\pi_{\text{ref}}(y|x)} + \beta\log Z(x)$$

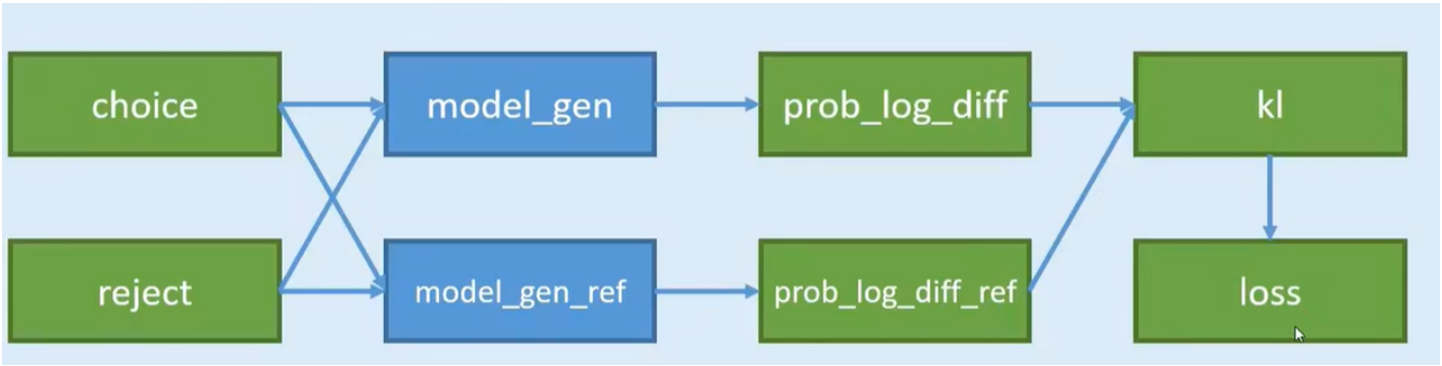
记得我们之前讲过的**Bradley-Terry偏好建模模型**吗，将上述式子带入可以得到：

$$\begin{aligned}
&p^*_\beta(y_1 \succ y_2|x) \\
&= \frac{\exp\left(\beta\log\frac{\pi^*(y_1|x)}{\pi_{\text{ref}}(y_1|x)} + \beta\log Z(x)\right)}{\exp\left(\beta\log\frac{\pi^*(y_1|x)}{\pi_{\text{ref}}(y_1|x)} + \beta\log Z(x)\right) + \exp\left(\beta\log\frac{\pi^*(y_2|x)}{\pi_{\text{ref}}(y_2|x)} + \beta\log Z(x)\right)} \\
&= \frac{1}{1+\exp\left(\beta\log\frac{\pi^*(y_2|x)}{\pi_{\text{ref}}(y_2|x)} - \beta\log\frac{\pi^*(y_1|x)}{\pi_{\text{ref}}(y_1|x)}\right)} \\
&= \sigma\left(\beta\log\frac{\pi^*(y_1|x)}{\pi_{\text{ref}}(y_1|x)} - \beta\log\frac{\pi^*(y_2|x)}{\pi_{\text{ref}}(y_2|x)}\right)
\end{aligned}$$

其中 σ 代表logistic函数。可以惊奇的发现由于指数函数的特性，**配分函数被消掉了，那么我们的偏好模型最后只剩下了参考策略和需要优化的策略，Critic和Reward model都被优化掉了**，然后再利用交叉熵和上面的偏好概率模型计算loss进行训练，就可以完成直接从偏好数据一步到优化策略

完整的DPO loss如下：

$$\mathcal{L}_{\text{DPO}}(\pi_\theta, \pi_{\text{ref}}) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}\left[\log\sigma\left(\beta\log\frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta\log\frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)}\right)\right]$$



如何理解DPO loss：其实从最后的loss公式来看，这是一个类似**对比学习**的loss（个人觉得就是对比学习），通过上述loss **DPO会最大化偏好的回答和不偏好的回答（win/loss，chosen/reject）的概率的差值（除过reference概率之后的）**，那么理想的结果就是，**chosen的回答概率上升，reject的回答概率下降**，从而达到跟人类偏好对齐的目的。当然**实践过程中可能会到chosen和reject的概率都下降的情况**，毕竟DPO只约束差值，DPO之后就有一些工作通过增加chosen的NLL loss或者其他正则化手段等等方法优化DPO

> ❓ **PPO VS DPO**
>
> 理论上PPO拥有更高的效果上限因为强化学习包含探索过程，有可能能探索到没见过的更好的状态，而DPO（offline DPO）则是完全局限于已经标好的数据，但是DPO实现起来更方便，PPO则需要很多工程实现技巧，包括如何训练好一个reward model都是一个技术活，所以很多工作也在研究DPO和PPO中间的状态，比如Online DPO等

3.4 PEFT 参数高效微调

微调大模型时只训练一小部分参数，而不是训练全量的模型参数，对于下游任务只需要部分参数就可以

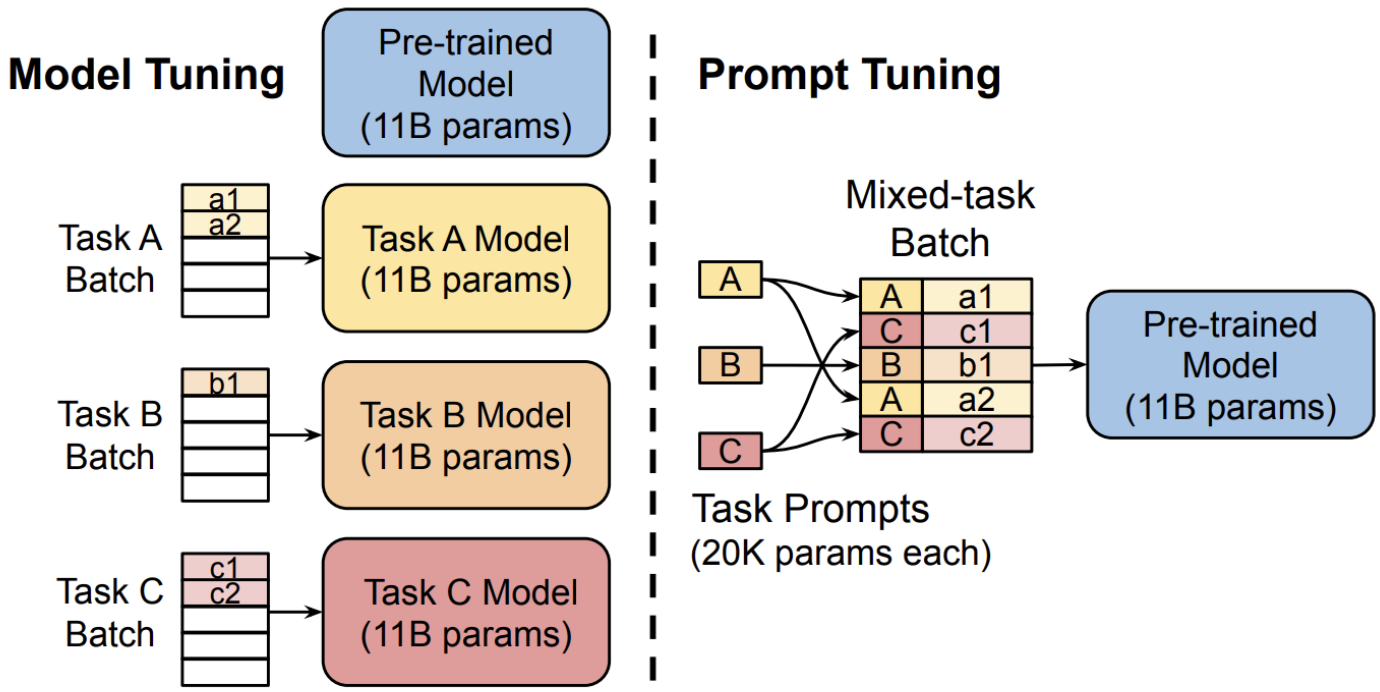
优点：

- 显存占用少，对硬件资源要求低
- 训练速度快，耗时更短
- 更低的存储成本，不同的任务可以共享大部分的权重参数
- 可能会有更好的模型性能，减轻了过拟合问题

但是一般有条件做SFT的时候还都是全量微调的

3.4.1 Prompt Tuning

prompt tuning 其实是一个比较范围比较广的概念，通过调整prompt模板来实现对模型的微调。不同的任务可以定义自己的Prompt，然后拼接到数据上作为输入，但只在输入层加入prompt tokens，并且不需要加入到后续的模型层MLP中进行调整来解决难训练的问题



- **Model Tuning**需要为每个下游任务都搞一个预训练模型，并且推理必须在单独的batch中进行
- **Prompt Tuning**只需要为每个任务存储一个小的特定于任务的prompt，并能够使用原始预训练模型进行混合任务推理

Prompt Tuning主要可以分为Hard Prompt和Soft Prompt：

- **Hard Prompt：**
  - 人工构建（Manual Template）
  - 启发式法（Heuristic-based Template）：通过规则、启发式搜索等方法构建合适的模板
  - 生成（Generation）：根据给定的任务训练数据（通常是小样本场景），生成出合适的模板
- **Soft Prompt：**
  - **词向量微调（Word Embedding）**：显式地定义离散字符的模板，但在训练时这些模板字符的词向量参与梯度下降，初始定义的离散字符用于作为向量的初始化
  - **伪标记（Pseudo Token）：**不显式地定义离散的模板，而是将模板作为**可训练的参数**

Hard Prompt旨在直接与原始文本拼接显式离散的字符，且在训练中始终保持不变 。这里的保持不变是指这些离散字符的词向量（Word Embedding）在训练过程中保持固定 ，不需要引入任何参数。

Soft Prompt旨在让模型在训练过程中根据具体的上下文语义和任务目标对模板参数进行连续可调。这套方案的动机则是认为**离散不变的模板无法参与模型的训练环节，容易陷入局部最优**，而如果**将模板变为可训练的参数**，那么不同的样本都可以在连续的向量空间中寻找合适的伪标记，同时也增加模型的泛化能力，需要引入少量的参数并让模型在训练时进行参数更新

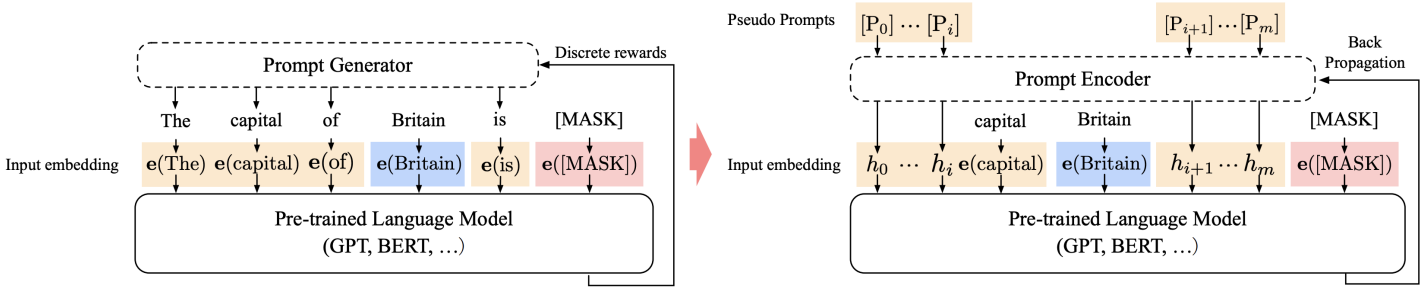


Prefix tuning、P-tuning、P-tuning v2都属于连续的prompt构建，**通过把传统人工设计模版中的真实token替换成可微的virtual token，转换为模型中可以学习的参数进行更新**，只是他们之间存在细微差别

3.4.2 P-Tuning

该方法将 **Prompt 转换为可以学习的 Embedding 层**，并用MLP+LSTM的方式来对Prompt Embedding进行一层处理

需要注意：**P-tuning只限于embedding层**，也就是输入层，没有在每一层都加；另外，virtual token的位置也不一定是前缀，插入的位置是可选的



(a) Discrete Prompt Search (b) P-tuning

主要改进：

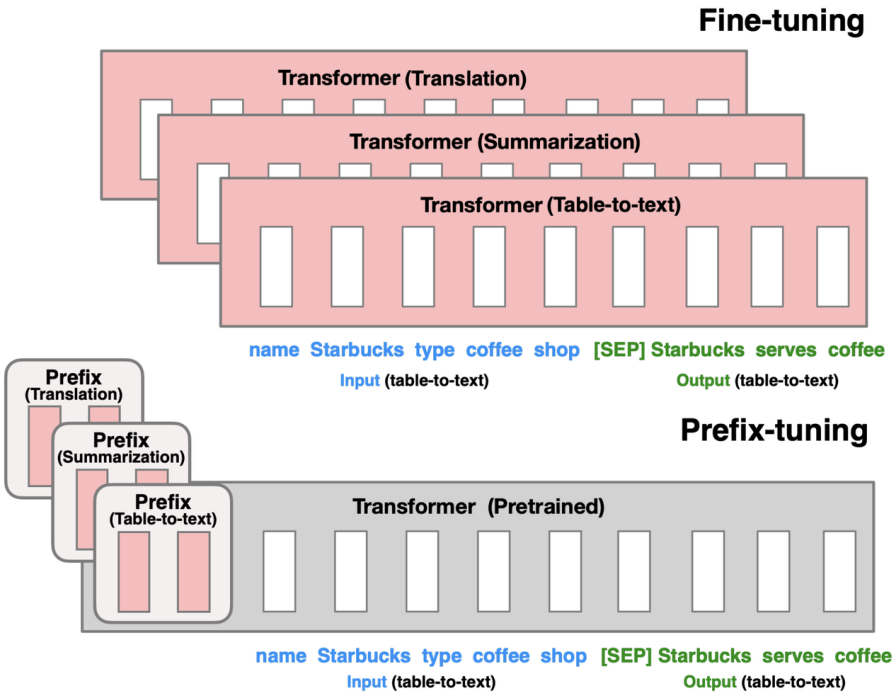
- **考虑到伪标记的相互依赖关系：**认为"capital"与"Britain"是有先后关系的，而prompt部分的双向attention无法显式地刻画这层关系，因此引入**Prompt Encoder，实际过程中的实现是采用一层RNN**
- **指定上下文词：**如果模板全部是伪标记，在训练时无法很好地控制这些模板朝着与对应句子相似的语义上优化，因此**选定部分具有与当前句子语义代表性的一些词作为一些伪标记的初始化**（例如上图中“capital”、“Britain”等）
- **重参数（Reparameterization）：**具体到代码实现上，**P-tuning先通过一个Prompt Encoder表征这些伪标记后，直接将这些新的表征覆盖到对应的embedding table上**，换句话说，Prompt Encoder只在训练时候会使用到，而在推理阶段则不再使用
- 混合提示（Hydride Prompt）：将连续提示与离散token进行混合。
- 需要被MASK的token如何选择：P-tuning中通过MLP或者LSTM选择
- 需要参与微调的参数量级：P-tuning中仅更新embedding层中virtual token的部分参数

为什么P-tuning要先经过MLP或者LSTM选择virtual token：经过预训练的LM的词嵌入已经变得高度离散，如果随机初始化virtual token，容易优化到局部最优值。因此，用一个**Prompt Encoder**（即用一个LSTM+MLP去编码这些virtual token以后，再输入到模型）来编码会收敛更快，效果更好

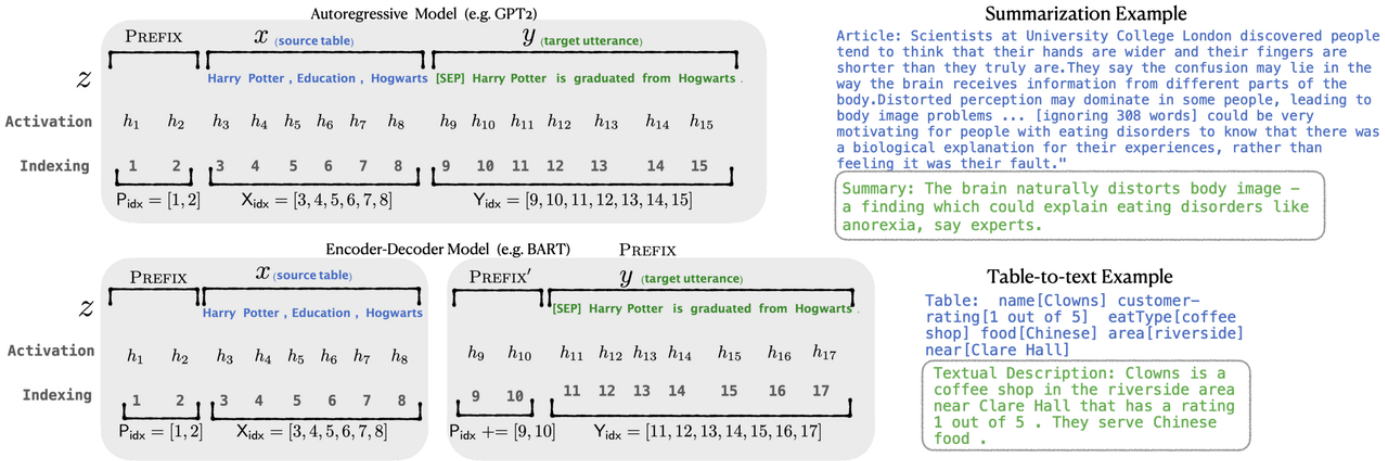
方法	对virtual token的处理	参与微调的参数	适配的下游任务
P-tuning	MLP+LSTM 或 MLP	仅embedding层中virtual token部分	将GPT适配NLU任务

3.4.3 Prefix Tuning

本质也是把传统人工设计模版中的真实token替换成可微的virtual token；该方法在**输入token之前构造一段任务相关的virtual tokens作为Prefix，**然后，在训练的时候**只更新Prefix部分的参数**，而LLM 中的其他部分参数固定

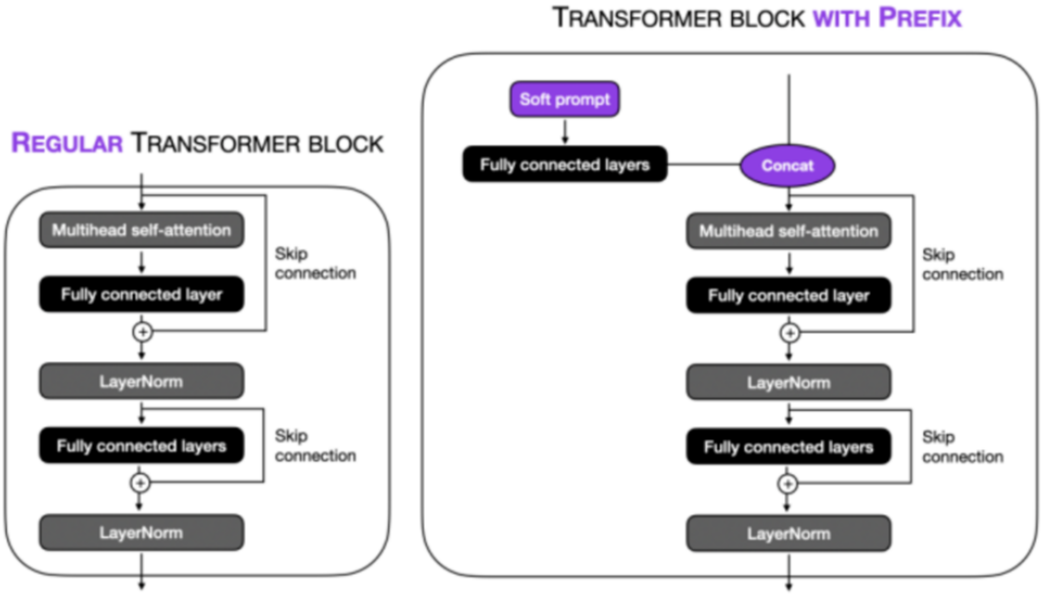


1. 针对不同的模型结构需要构造不同的Prefix
   - 针对**自回归架构模型：**在句子前面添加前缀，得到 `z = [PREFIX; x; y]`，合适的上文能够在固定 LM 的情况下去引导生成下文（比如：GPT3的上下文学习）
   - 针对**编码器-解码器架构模型：**Encoder和Decoder都增加了前缀，得到 `z = [PREFIX; x; PREFIX0; y]`。Encoder端增加前缀是为了引导输入部分的编码，Decoder 端增加前缀是为了引导后续token的生成



2. 对 virtual token的编码方式

   同时，为了防止直接更新 Prefix 的参数导致训练不稳定和性能下降的情况，在 Prefix 层前面加了 MLP 结构，训练完成后，只保留 Prefix 的参数。除此之外，**通过消融实验证实，只调整embedding层的表现力不够，将导致性能显著下降，因此，在每层都加了prompt的参数，改动较大**



对比Prompt Tuning和Prefix Tuning：

- **Prompt Tuning**：将可训练的张量添加到输入embedding
- **Prefix Tuning**：将可训练张量添加到所有Transformer Block

3.4.4 P-Tuning V2

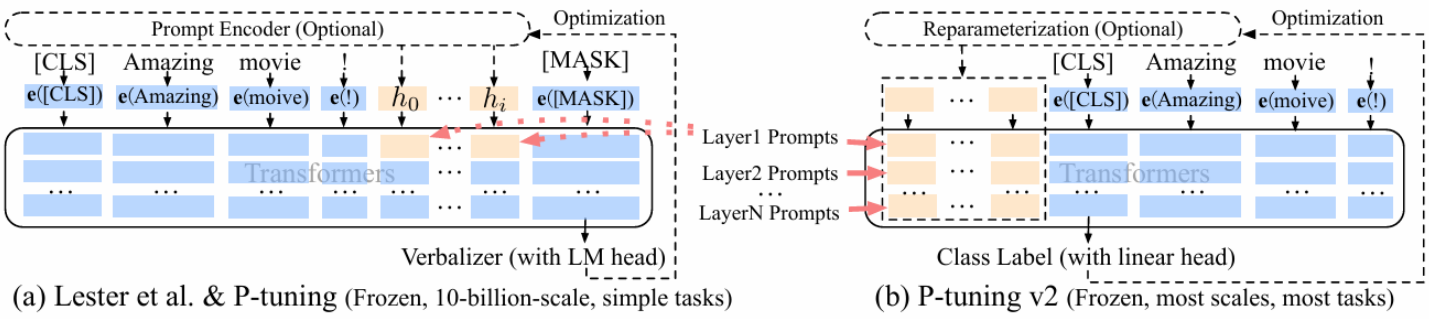
之前的Prompt Tuning和P-Tuning等方法存在两个主要的问题：

1. **缺乏模型参数规模和任务通用性**
    - **缺乏规模通用性**：Prompt Tuning论文中表明当模型规模超过100亿个参数时，提示优化可以与全量微调相媲美。但是对于那些较小的模型（从100M到1B），提示优化和全量微调的表现有很大差异，这大大限制了提示优化的适用性
    - **缺乏任务普遍性**：尽管Prompt Tuning和P-tuning在一些 NLU（自然语言理解） 基准测试中表现出优势，但提示调优对硬序列标记任务（即序列标注）的有效性尚未得到验证
2. **缺少深度提示优化**，在Prompt Tuning和P-tuning中，连续提示只被插入transformer第一层的输入embedding序列中，在接下来的transformer层中，插入连续提示的位置的embedding是由之前的transformer层计算出来的，这可能导致两个可能的优化挑战。
    - **由于序列长度的限制，可调参数的数量是有限的**。
    - **输入embedding对模型预测只有相对间接的影响**。

该方法在每一层都加入了Prompts tokens作为输入，而不是仅仅加在输入层，这带来两个方面的好处：

- **更多可学习的参数**（从P-tuning和Prompt Tuning的0.01%增加到0.1%-3%），同时也足够参数高效
- 加入到更深层结构中的Prompt能给模型预测带来更直接的影响

P-Tuning V2的具体做法：



做法**基本同Prefix Tuning**，可以看作是**文本生成的Prefix Tuning技术适配到NLU任务中**，然后做了一些改进：

- **移除重参数化的编码器：以前的方法利用重参数化功能来提高训练速度和鲁棒性**（如：Prefix Tuning 中的 MLP 、P-Tuning 中的 LSTM），在 P-tuning v2 中，作者发现重参数化的改进很小，尤其是对于较小的模型，同时还会影响模型的表现
- **针对不同任务采用不同的提示长度：**提示长度在提示优化方法的超参数搜索中起着核心作用。不同的理解任务通常用不同的提示长度来实现其最佳性能，这与 Prefix-Tuning 中的发现一致，不同的文本生成任务可能有不同的最佳提示长度
- **引入多任务学习：**先在多任务的Prompt上进行预训练，然后再适配下游任务。一方面，连续提示的随机性给优化带来了困难，这可以通过更多的训练数据或与任务相关的无监督预训练来缓解；另一方面，连续提示是跨任务和数据集的特定任务知识的完美载体。在一些困难的序列任务中，多任务学习可以作为P-tuning v2的补充
- **回归传统的分类标签范式而不是映射器**：标签词映射器（Label Word Verbalizer）一直是提示优化的核心组成部分，它将one-hot类标签变成有意义的词，以利用预训练语言模型头。尽管它在few-shot设置中具有潜在的必要性，但在全数据监督设置中，Verbalizer并不是必须的，它阻碍了提示调优在需要无实际意义的标签和句子嵌入的场景中的应用。因此，P-Tuning v2回归传统的CLS标签分类范式，采用随机初始化的分类头（Classification Head）应用于tokens之上，以增强通用性，可以适配到序列标注任务

实现的时候prefix tuning和P-tuning v2都是通过**past_key_values**进入attention运算内部，在每一层中运算的

Huggingface PEFT实现

```
#设置virtual_token的数量及past_key_values维度
prompt_tokens = prompt_tokens[:, : peft_config.num_virtual_tokens]
past_key_values = past_key_values.view(
        batch_size,peft_config.num_virtual_tokens,
        peft_config.num_layers * 2,
        peft_config.num_attention_heads,
        peft_config.token_dim // peft_config.num_attention_heads,
)
# prefix encoder类的定义
class PrefixEncoder(torch.nn.Module):
```

```
    """
    The torch.nn model to encode the prefix    Input shape: (batch-size, prefix-length)    Output shape: (batch-size, prefix-length, 2*layers*hidden)
    """    def __init__(self, config: ChatGLMConfig):
        super().__init__()
        self.prefix_projection = config.prefix_projection

        # self.prefix_projection=True-->prefix tuning
        # self.prefix_projection=False-->p-tuning v2
        if self.prefix_projection:
            # Use a two-layer MLP to encode the prefix
            kv_size = config.num_layers * config.kv_channels * config.multi_query_group_num * 2
            self.embedding = torch.nn.Embedding(config.pre_seq_len, kv_size)
            self.trans = torch.nn.Sequential(
                torch.nn.Linear(kv_size, config.hidden_size),
                torch.nn.Tanh(),
                torch.nn.Linear(config.hidden_size, kv_size)
            )
        else:
            self.embedding = torch.nn.Embedding(config.pre_seq_len,
                                                config.num_layers * config.kv_channels * config.multi_query_group_num * 2)

    def forward(self, prefix: torch.Tensor):
        if self.prefix_projection:
            prefix_tokens = self.embedding(prefix)
            past_key_values = self.trans(prefix_tokens)
        else:
            past_key_values = self.embedding(prefix)
        return past_key_values
```

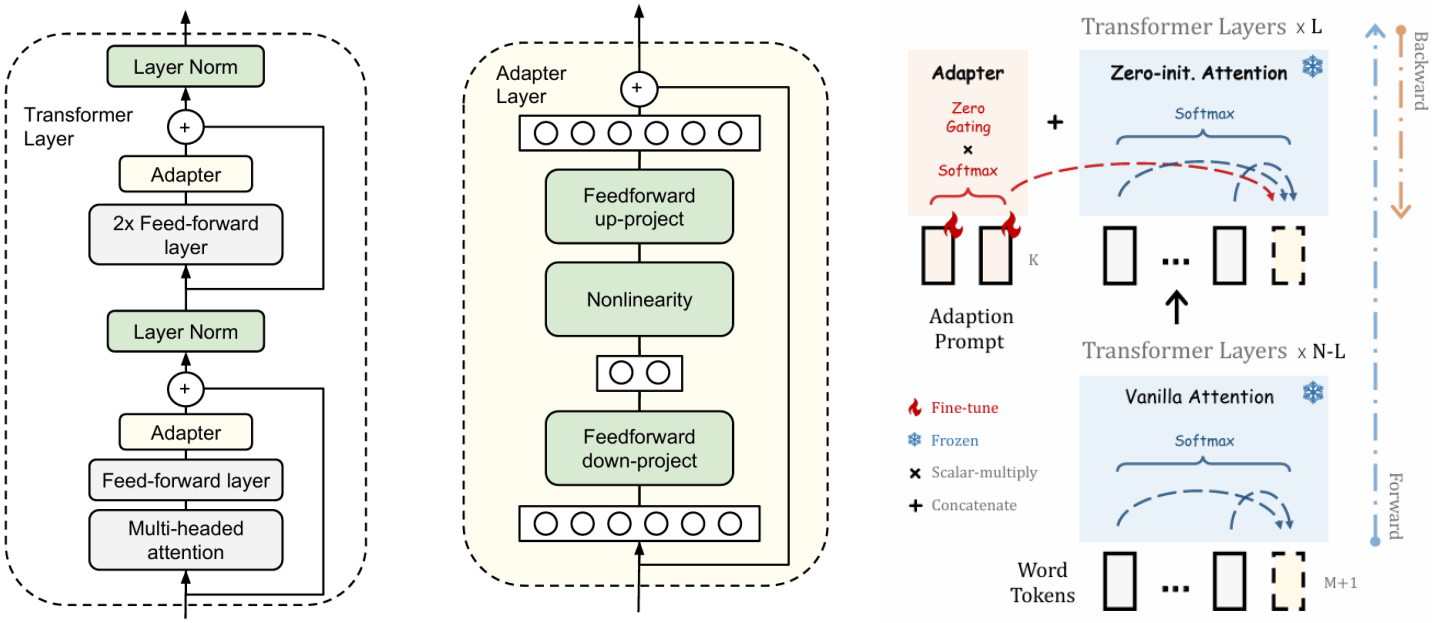
总结一下soft prompt方法：

方法	对virtual token的处理	参与微调的参数	适配的下游任务
P-tuning	MLP + LSTM或MLP	仅embedding层中virtual token部分	使GPT适配NLU任务
Prefix tuning	MLP	Prefix encoder，通过past_key_values代入每一层运算	主要用于NLG任务
P-tuning v2	\		NLG/NLU任务

3.4.5 Adapter Tuning

核心思想：通过引入一些额外的模块来适配下游任务

通过在每一层的transformer layer中加入额外的Adapter的模块来在下游任务微调，特点是**微调训练和推理的时候都额外增加了成本，因为Adapter是线性的加入transformer结构中的**



Adapter　　LLaMA-adapter

LLaMA-adapter：

Adapter和Prefix tuning的问题是结合了随机初始化的张量可能会损害预训练的语义学知识，导致**微调不稳定和性能损失**，而LLaMA - adapter通过下面两个改进实现稳定训练

- 引入**零初始化的注意力机制和门控机制**
- **只修改深层的L个Transformer层**

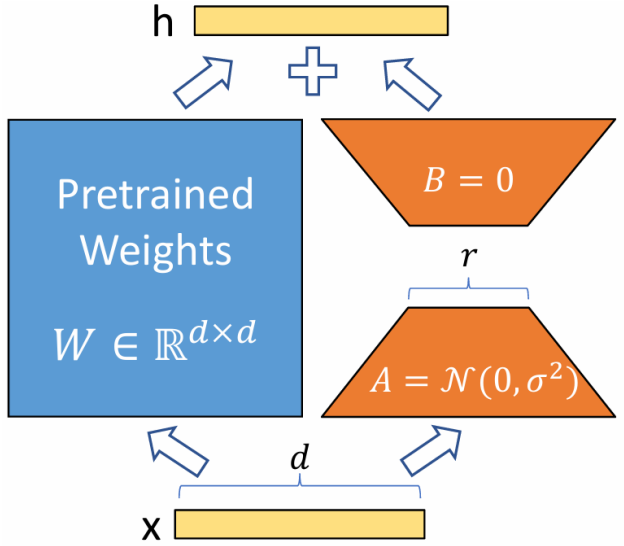
3.4.6 LoRA

Low-rank Adaption 低秩微调

LORA是一种低资源微调大模型方法，使用LORA，训练参数仅为整体参数的万分之一、GPU显存使用量减少2/3且**不会引入额外的推理耗时（相比Adapter的优势）**

- **高效微调的原理：**在微调模型的过程中加载预训练参数 ϕ 进行初始化，并通过FT的优化目标最大化语言模型概率更新得到参数 $\phi + \Delta\phi$。这样的微调方式的主要缺点是得到的增量参数 $\Delta\phi$ 和初始化参数的维度是一致的，称为全量微调，需要大量资源。高效微调则是用更少的参数表示上述需要学到的参数增量，相当于只微调一部分参数
- **LoRA的实现：**LoRA旨在**使用低秩矩阵来编码参数增量 $\Delta\phi$，在训练过程中，低秩的适应矩阵仅仅放大了对下游任务有用的特征，而不是预训练模型中的主要特征**
  - **Instrisic Dimension 内在维度**：LLM表现出很好的few-shot能力，以及利用数千条样本就可以微调一个数十亿参数的LLM，其内在原意是**预训练模型拥有极小的内在维度(instrisic dimension)，即存在一个极低维度的参数，微调它和在全参数空间中微调能起到相同的效果**

- **Low Rank 低秩：**根据内在维度的启发，LoRA认为参数更新过程中也存在一个内在秩，对于预训练权重矩阵可以用一个低秩分解来表示参数更新：



$$W_0 + \Delta W = W_0 + BA$$
$$W_0 \in \mathbb{R}^{d\times k}, B \in \mathbb{R}^{d\times r}, A \in \mathbb{R}^{r\times k}$$
$$\text{and} \quad r \ll \min(d, k)$$

训练过程中冻结参数 W_0 ，仅训练A和B中的参数，前向forward过程也变成了：

$$h = W_0 x + \Delta W x = W_0 x + BAx$$

两边的forward推理是并行的可以同时进行，所以相比Adapter的串行就减少了推理耗时

- **LoRA应该作用在哪些参数矩阵？：**

	# of Trainable Parameters = 18M						
Weight Type	W_q	W_k	W_v	W_o	W_q, W_k	W_q, W_v	W_q, W_k, W_v, W_o
Rank r	8	8	8	8	4	4	2
WikiSQL (±0.5%)	70.4	70.0	73.0	73.2	71.4	**73.7**	**73.7**
MultiNLI (±0.1%)	91.0	90.8	91.0	91.3	91.3	91.3	**91.7**

  - 将所有微调参数都放到attention的某一个参数矩阵的效果并不好，将可微调参数平均分配到□□和□v的效果最好
  - 即使是秩仅取4也能在Δ□中获得足够的信息

- **LoRA的最优秩r是多少？：**

	Weight Type	$r = 1$	$r = 2$	$r = 4$	$r = 8$	$r = 64$
WikiSQL(±0.5%)	W_q	68.8	69.6	70.5	70.4	70.0
	W_q, W_v	73.4	73.3	73.7	73.8	73.5
	W_q, W_k, W_v, W_o	74.1	73.7	74.0	74.0	73.9
MultiNLI (±0.1%)	W_q	90.7	90.9	91.1	90.7	90.7
	W_q, W_v	91.3	91.4	91.3	91.6	91.4
	W_q, W_k, W_v, W_o	91.2	91.7	91.7	91.5	91.4

  - 在秩小到1或者2的时候，LORA的仍有不错的效果，因此更新参数矩阵ΔW可能拥有极小的内在秩

- **LoRA的矩阵初始化：**下采样矩阵A为随机高斯初始化，上采样矩阵B为全0初始化

📌 这么初始化是为了保证训练的开始旁路矩阵依然是0矩阵，对预训练的参数没有影响

- **为什么B可以初始化为全0?**

  如果B初始化为全0，则初始的时候 $\Delta W = A \cdot B = A \cdot 0 = 0$，意味着模型的权重未发生任何变化，仍然是原始权重，随着训练进行，B的梯度会逐渐更新，最终学得有意义的权重调整。B的梯度：$\frac{\partial \mathcal{L}}{\partial B} = A^T \cdot \frac{\partial \mathcal{L}}{\partial \Delta W}$ 因此，B初始化为0不会阻碍训练，因为它仍然可以从梯度中学习到非零的值

- **为什么A不能初始化为全0?**

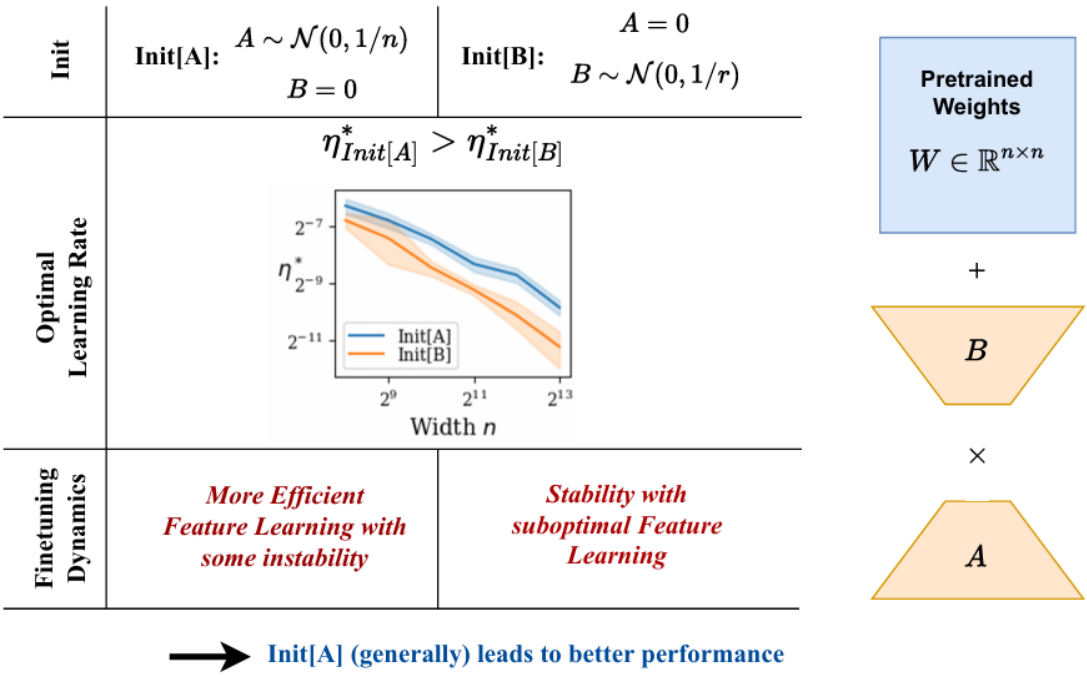
  如果A初始化为全0，$\Delta W = 0$，A的梯度 $\frac{\partial \mathcal{L}}{\partial A} = \frac{\partial \mathcal{L}}{\partial \Delta W} \cdot B^T = 0$ 由于梯度为0，A将无法更新。这会导致模型的训练停滞，权重无法学到有意义的调整

实际上上述的解释对于AB反过来也成立，并不能说明为什么这么设计

这篇文献 ***The Impact of Initialization on LoRA Finetuning Dynamics*** 做了实验给出了AB两种初始化的对比

- Init[A]拥有更好的特征学习效率和一些训练不稳定性
- Init[B]拥有稳定训练和次优的特征学习

Init[A]通常情况下拥有更好的表现，源自于更好的特征学习，一点训练不稳定性可以接受



LoRA变体：

- **QLoRA：**基于LoRA的微调方法，引入**4-bit NormalFloat、双重量化和Paged Optimizers**等方法，降低内存使用同时保持性能。通过在每个网络层添加适配器，QLORA避免以前工作中观察到的几乎所有的准确性折衷。这种方法将65B参数模型的内存需求从>780GB降低到<48GB，使得单GPU上微调最大的公开可用模型成为可能

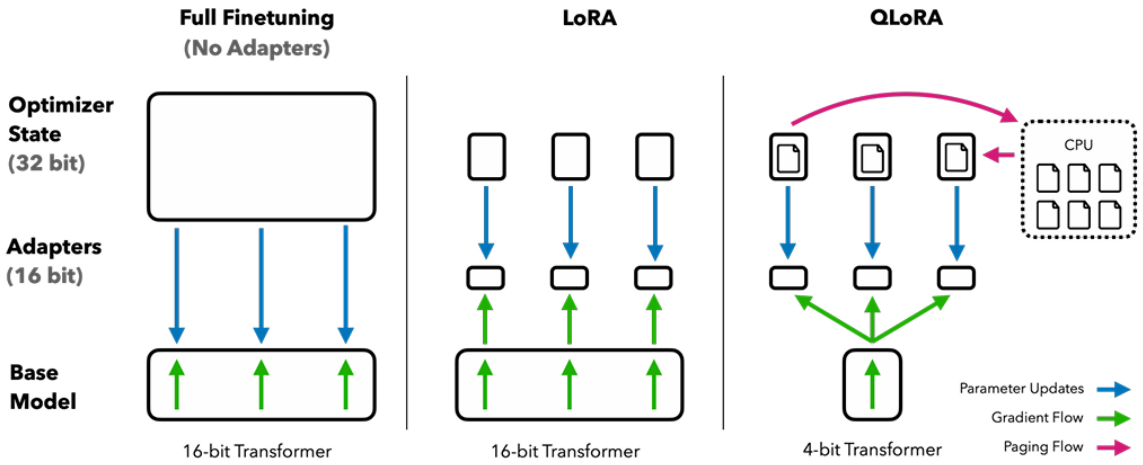
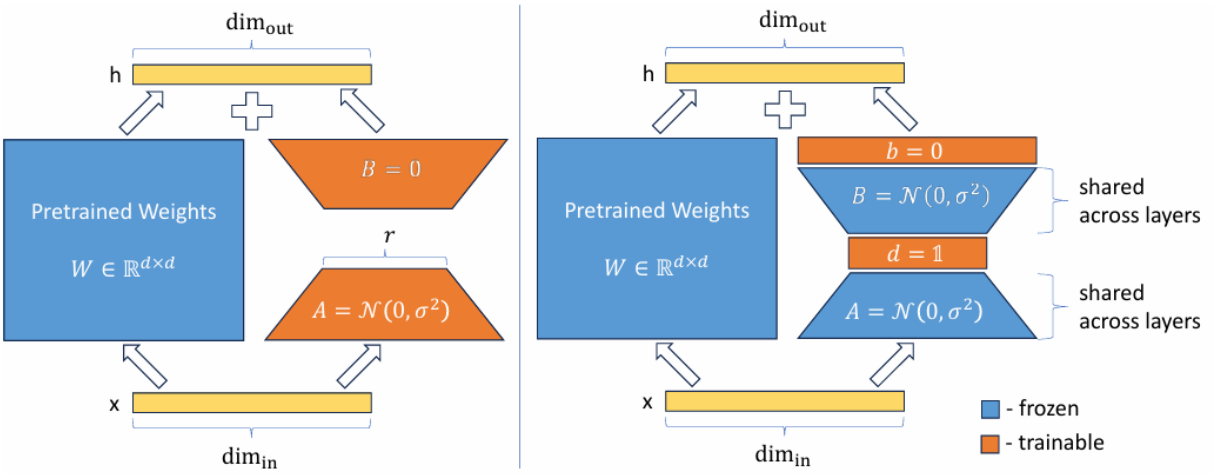


Figure 1: Different finetuning methods and their memory requirements. QLoRA improves over LoRA by quantizing the transformer model to 4-bit precision and using paged optimizers to handle memory spikes.

- **LoRA+：**为矩阵**a和b引入不同的学习率**，引入了一种更有效的训练LoRA适配器的方法

	LoRA	LoRA+
Parameterization	Pretrained Weights $W \in \mathbb{R}^{n \times n}$ $+$ B $\times$ A	
Training	$A \leftarrow A - \eta \times G_A$ $B \leftarrow B - \eta \times G_B$	$A \leftarrow A - \eta \times G_A$ $B \leftarrow B - \lambda\eta \times G_B$ $\lambda \gg 1$

- **VeRA：**Vector-based Random Matrix Adaptation，引入了一种方法来大幅减少LoRA参数大小，**没有训练矩阵A和B而是用共享的随机权值初始化这些矩阵(即所有层中所有矩阵A和B都具有相同的权值)，并添加两个新的向量d和b，微调的时候只训练向量d和b**



- **AdaLoRA：**LoRA每个Adapter的秩都是一样的，**AdaLoRA对于不同层、类型参数根据下游任务动态分配秩**

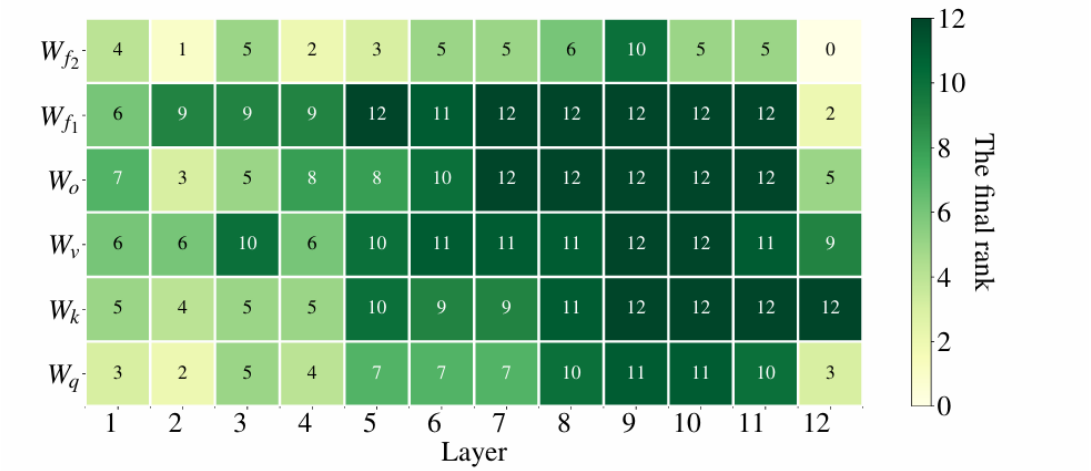


Figure 3: The resulting rank of each incremental matrix when fine-tuning DeBERTaV3-base on MNLI with AdaLoRA. Here the x-axis is the layer index and the y-axis represents different types of adapted weight matrices.

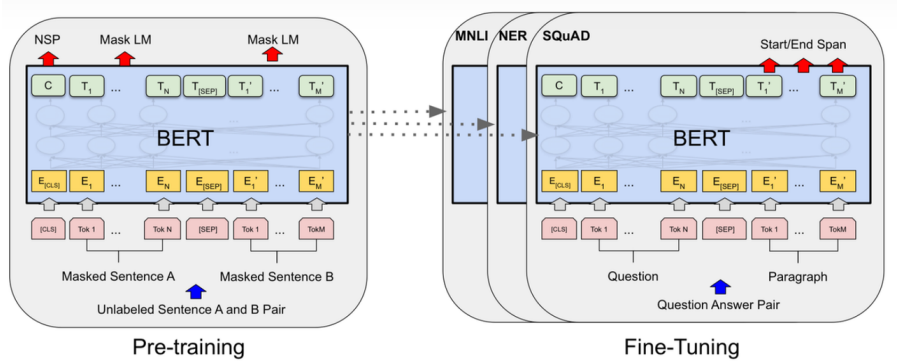
4. Common Models 常见模型

4.1 BERT及变体

4.1.1 BERT

Bidirectional Encoder Representation from Transformers，encoder-only

BERT主要是用来替代Word2Vec的预训练模型，通过**MLM和NSP两大预训练任务学习表征**，对于下游任务加上一个layer微调即可





BERT的输入的编码向量（长度是512）是3个嵌入特征的单位和，这三个词嵌入特征是：

1. **Position Embedding**：位置嵌入是指将单词的位置信息编码成特征向量，位置嵌入是向模型中引入单词位置关系的至关重要的一环；BERT直接去训练了一个position embedding，给每个位置词一个随机初始化的词向量，再训练
2. **Token Embedding**：上图的示例中‘playing’被拆分成了‘play’和‘ing’的token
3. **Segment Embedding**：用于区分两个句子，例如B是否是A的下文（对话场景，问答场景等）。对于句子对，第一个句子的特征值是0，第二个句子的特征值是1

Masked LM (MLM)：在将单词序列输入给 BERT 之前，每个序列中有 15% 的单词被 `[MASK]` token 替换。 然后模型尝试基于序列中其他未被 mask 的单词的上下文来预测被掩盖的原单词。在BERT的实验中，15%的WordPiece Token会被随机Mask掉。在训练模型时，一个句子会被多次喂到模型中用于参数学习，但是Google并没有在每次都mask掉这些单词，而是在确定要Mask掉的单词之后，80%的概率会直接替换为 `[Mask]` ，10%的概率将其替换为其它任意单词，10%的概率会保留原始Token。

1. **80% 的 tokens 会被替换为 [MASK] token**：是 Masked LM 中的主要部分，**可以在不泄露 label 的情况下融合真双向语义信息**；
2. **10% 的 tokens 会称替换为随机的 token**：因为**需要在最后一层随机替换的这个 token 位去预测它真实的词，模型并不知道输入对应位置的词汇是否为正确的词汇（ 10%概率），这就迫使模型更多地依赖于上下文信息去预测词汇，并且赋予了模型一定的纠错能力，因而也能够让 BERT 获得更好的语境相关的词向量（这正是解决一词多义的最重要特性）**；
3. **10% 的 tokens 会保持不变但需要被预测**：这样能够给模型一定的 **bias ，相当于是额外的奖励，将模型对于词的表征能够拉向词的 真实表征**

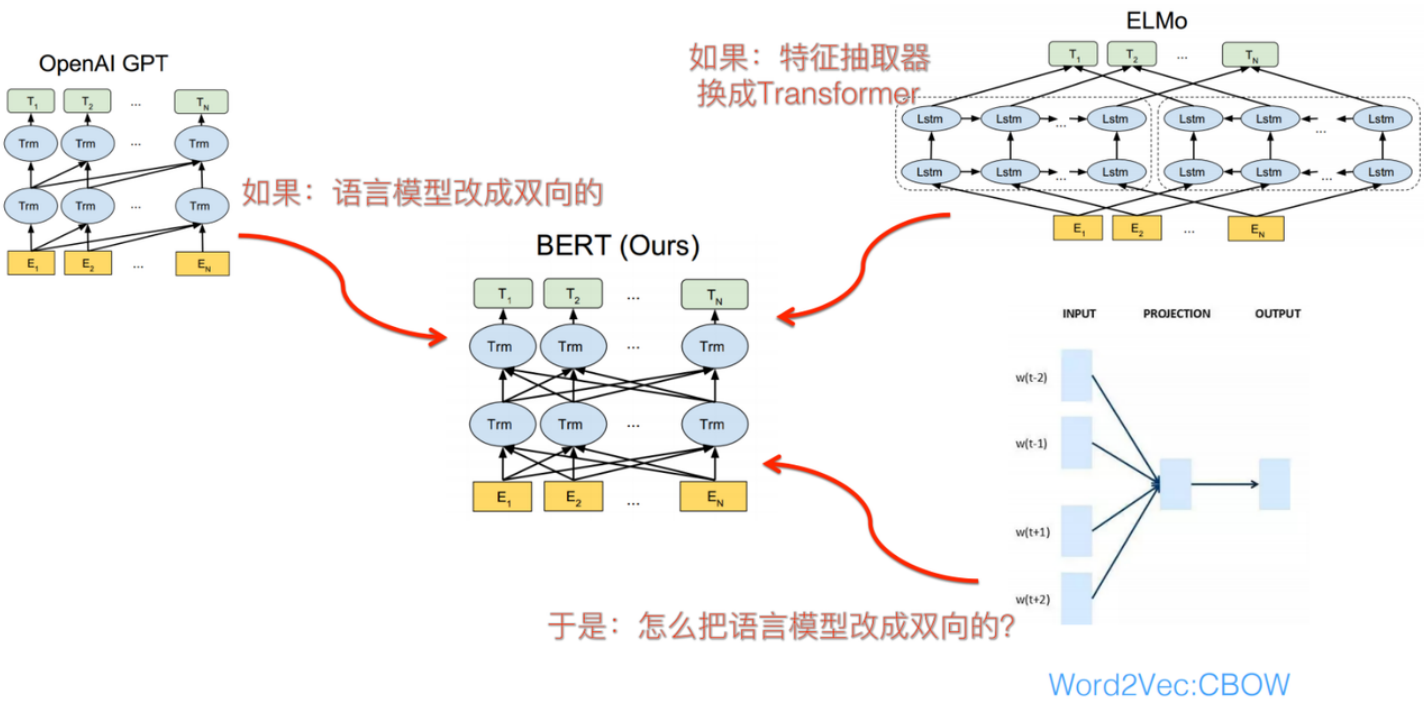
Next Sentence Prediction (NSP)：在 BERT 的训练过程中，模型接收成对的句子作为输入，并且预测其中第二个句子是否在原始文档中也是后续句子

1. 在训练期间，50% 的输入对在原始文档中是前后关系，另外 50% 中是从语料库中随机组成的，并且是与第一句断开的。
2. 在**第一个句子的开头插入 `[CLS]` 标记，表示该特征用于分类模型，对非分类模型，该符号可以省去，在每个句子的末尾插入 `[SEP]` 标记，表示分句符号，用于断开输入语料中的两个句子**。

优点：BERT使用了**双向Transformer提取特征**，使得模型能力大幅提升；添加了两个预训练任务，MLM + NSP的多任务方式进行模型预训练.

缺点：

1. 模型过于庞大, 参数量太多, 需要的数据和算力要求过高, 训练好的模型应用场景要求高；**更适合用于语言嵌入表达，语言理解方面的任务，不适合用于生成式的任务**
2. 因为Bert用于下游任务微调时， `[MASK]` 标记不会出现，它只出现在预训练任务中。这就造成了预训练和微调之间的不匹配，微调不出现 `[MASK]` 这个标记，模型好像就没有了着力点、不知从哪入手。所以只将80%的替换为 `[mask]` ，但这也只是缓解、不能解决。
3. 相较于传统语言模型，Bert的每批次训练数据中只有 15% 的标记被预测，这导致模型需要更多的训练步骤来收敛



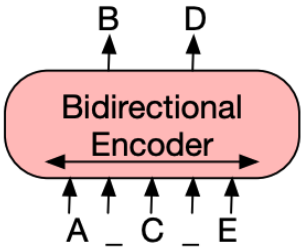
GPT、BERT和ELMo的对比

4.1.2 BART

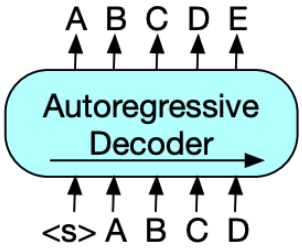
Bidirectional and Auto-Regressive Transformers

BART使用**标准的Transformer模型，encoder- decoder**，不过做了一些改变：

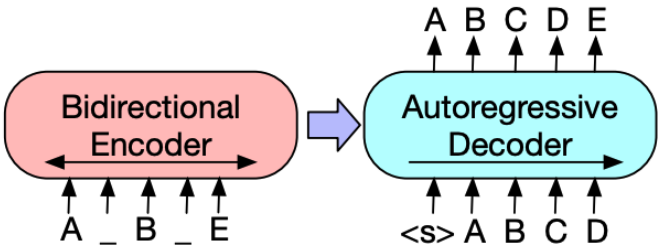
- 同GPT一样，将ReLU激活函数改为GeLU，并且参数初始化服从正态分布N(0,0.02)
- BART base模型的Encoder和Decoder各有6层，large模型增加到了12层
- BART解码器的各层对编码器最终隐藏层额外执行cross-attention
- BERT在词预测之前使用了额外的Feed Forward Layer，而BART没有



(a) BERT: Random tokens are replaced with masks, and the document is encoded bidirectionally. Missing tokens are predicted independently, so BERT cannot easily be used for generation.



(b) GPT: Tokens are predicted auto-regressively, meaning GPT can be used for generation. However words can only condition on leftward context, so it cannot learn bidirectional interactions.



(c) BART: Inputs to the encoder need not be aligned with decoder outputs, allowing arbitary noise transformations. Here, a document has been corrupted by replacing spans of text with mask symbols. The corrupted document (left) is encoded with a bidirectional model, and then the likelihood of the original document (right) is calculated with an autoregressive decoder. For fine-tuning, an uncorrupted document is input to both the encoder and decoder, and we use representations from the final hidden state of the decoder.

BERT, GPT, BART的区别

BART比BERT更适合文本生成的场景；相比GPT，也多了双向上下文语境信息。在生成任务上获得进步的同时，它也可以在一些文本理解类任务上取得SOTA

4.1.3 T5

Text-to-Text Transfor Transformer， encoder-decoder 架构

预训练语言模型领域的通用模型，该模型的**基本思想是将所有自然语言问题都转化成文本到文本（“text-to-text”）的形式**，并用一个统一的模型解决，即**输入是带有任务前缀声明的文本序列，输出的文本序列是相应任务的结果**。具体到这个工作，主要考虑了**machine translation、question answering、abstractive summarization和text classification**四个任务，顺便贡献了个语料库C4（Colossal Clean Crawled Corpus）

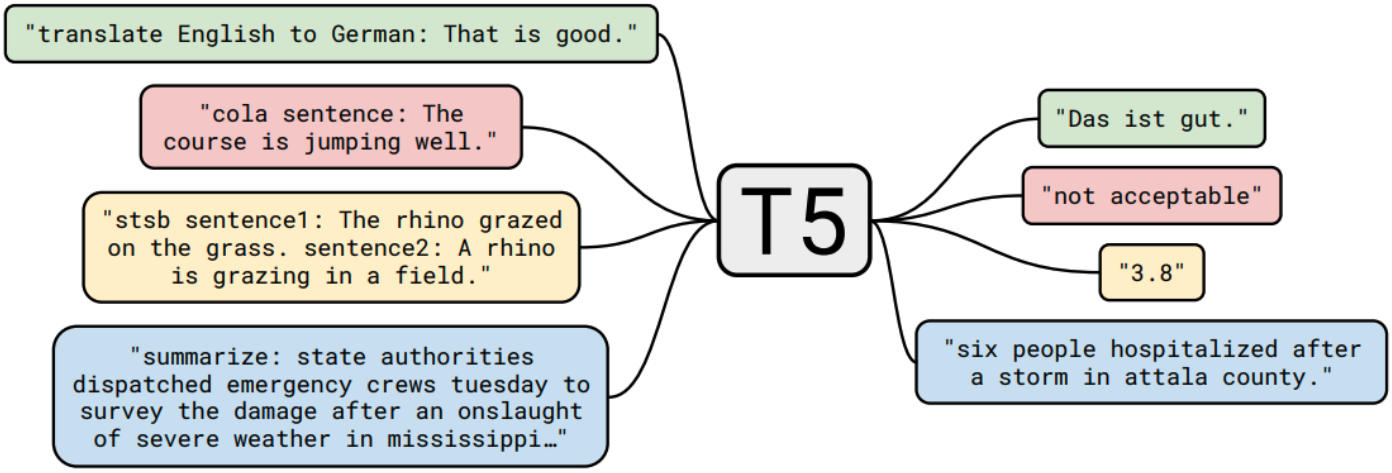


Figure 1: A diagram of our text-to-text framework. Every task we consider—including translation, question answering, and classification—is cast as feeding our model text as input and training it to generate some target text. This allows us to use the same model, loss function, hyperparameters, etc. across our diverse set of tasks. It also provides a standard testbed for the methods included in our empirical survey. "T5" refers to our model, which we dub the "**T**ext-**t**o-**T**ext **T**ransfer **T**ransformer".

对下游任务微调训练更友好，不需要做任何模型侧的改动，只需要对下游任务的训练数据做简单改写，就可以用T5完成相应的任务。训练方式采用和BERT一样的方法，但是**mask连续的三个词，只预测被mask的部分词**，重点还是**大量的数据，和更大的模型11B参数**

4.1.4 RoBERTa

A Robustly Optimized BERT Pretraining Approach

在**模型规模、算力和数据上**，与BERT相比主要有以下几点改进：

- **更大的模型参数量**，模型使用 1024 块 V100 GPU 训练了 1 天的时间
- **更大bacth size**，RoBERTa 在训练过程中使用了更大的bacth size。尝试过从 256 到 8000 不等的 bacth size
- **更多的训练数据，**包括：CC-NEWS 等在内的 160GB 纯文本。而最初的BERT使用16GB BookCorpus数据集和英语维基百科进行训练

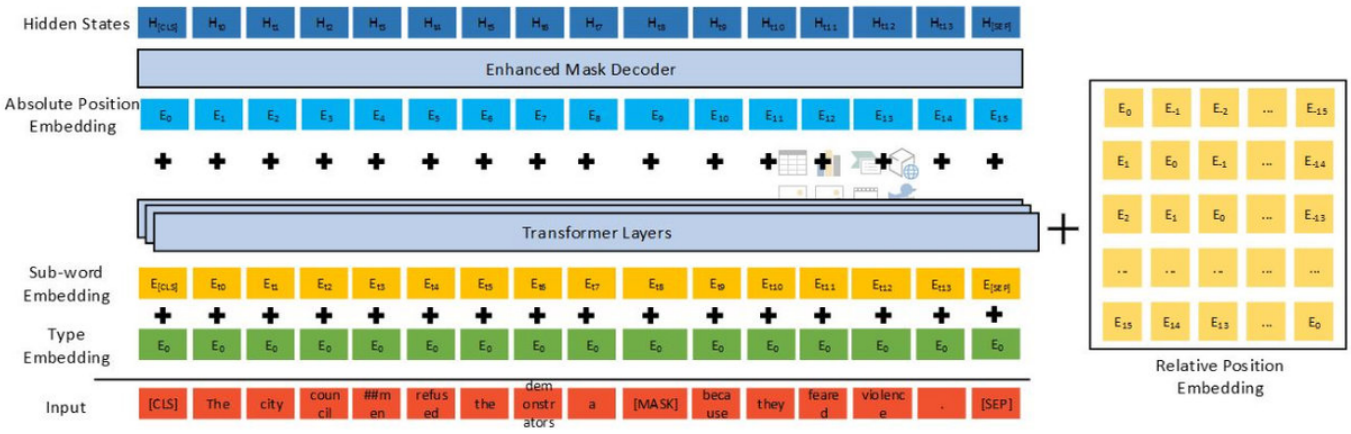
另外，RoBERTa在训练方法上有以下改进：

- **去掉下一句预测(NSP)任务**
- **动态掩码，**BERT 依赖随机掩码和预测 token。原版的 BERT 实现在数据预处理期间执行一次掩码，得到一个静态掩码。 而 RoBERTa 使用了动态掩码：每次向模型输入一个序列时都会生成新的掩码模式。这样，在大量数据不断输入的过程中，模型会逐渐适应不同的掩码策略，学习不同的语言表征。

- **文本编码，**Byte-Pair Encoding（BPE）是字符级和词级别表征的混合，支持处理自然语言语料库中的众多常见词汇。原版的 BERT 实现使用**字符级别的 BPE 词汇**，大小为 30K，是在利用**启发式分词规则**对输入进行预处理之后学得的。而BART考虑用**更大的 byte 级别 BPE 词汇表来训练 BERT**，这一词汇表包含 50K 的子词单元，且没有对输入作任何额外的预处理或分词

4.1.5 DeBERTa

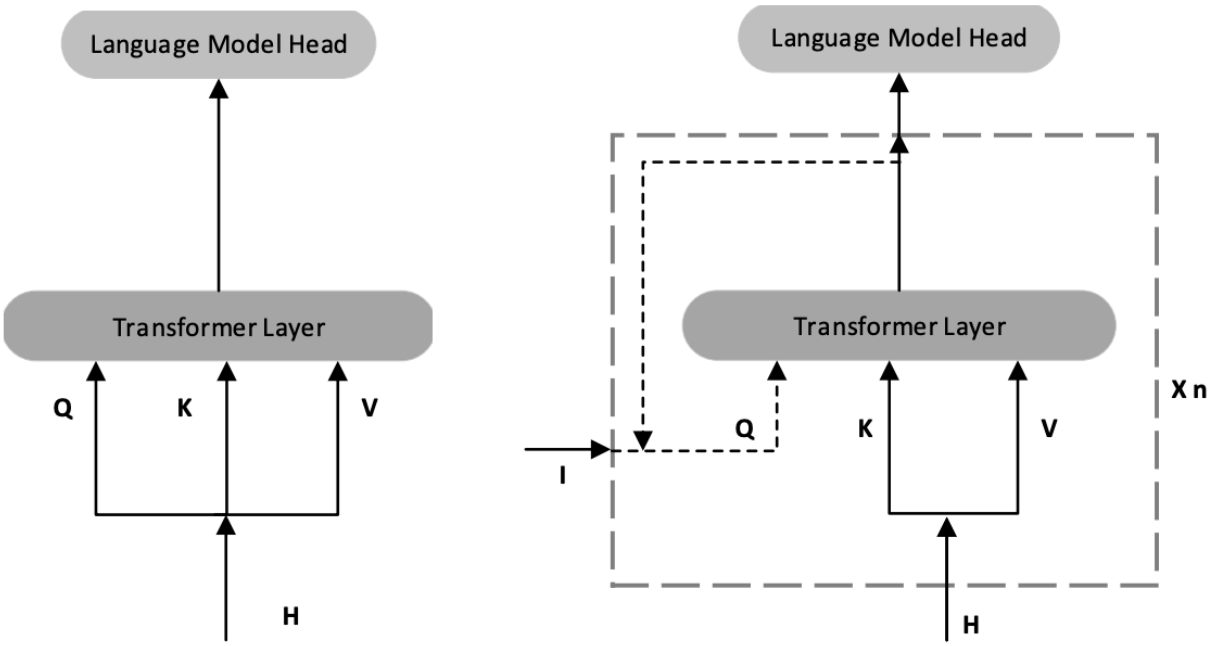
Decoding-enhanced BERT with disentangled attention



DeBERTa 模型使用了**注意力解耦机制**和**增强的mask解码器**改进了 BERT 和 RoBERTa 模型，同时还引入了一种新的微调方法 **虚拟对抗训练** 以提高模型的泛化能力

- **注意力解耦机制：**
  - 在BERT 中，输入层的每个单词都使用一个向量来表示，该向量是其单词（内容）嵌入和位置嵌入的总和
  - 而 DeBERTa 中的每个单词使用两个对其内容和位置分别进行编码的向量来表示，并且注意力单词之间的权重分别使用基于它们的内容和相对位置的解码矩阵来计算
  - 单词对的注意力权重不仅取决于它们的内容，还取决于它们的相对位置。 例如，“deep”和“learning”这两个词相邻出现时，它们之间的依赖性比它们出现在不同句子中时要强得多
- **增强的掩码解码器：**
  - DeBERTa和BERT模型一样，也是使用MLM进行预训练的，在该模型中，模型被训练为使用 mask token 周围的单词来预测mask词应该是什么。 DeBERTa将上下文的内容和位置信息用于 MLM。 解耦注意力机制已经考虑了上下文词的内容和相对位置，但没有考虑这些词的绝对位置，这在很多情况下对于预测至关重要。

- 如：给定一个句子“a new store opened beside the new mall”，并用“store”和“mall”两个词 mask 以进行预测。 仅使用局部上下文（即相对位置和周围的单词）不足以使模型在此句子中区分store和mall，因为两者都以相同的相对位置在new单词之后。 为了解决这个限制，模型需要考虑绝对位置，作为相对位置的补充信息。 例如，句子的主题是“store”而不是“mall”。 这些语法上的细微差别在很大程度上取决于单词在句子中的绝对位置。
- 有两种合并绝对位置的方法。 BERT模型在输入层中合并了绝对位置。 在DeBERTa中，微软在所有Transformer层之后将它们合并，然后在softmax层之前进行 mask token 预测。这样，DeBERTa捕获了所有Transformer层中的相对位置，同时解码被mask的单词时将绝对位置用作补充信息



(a) BERT decoding layer　　(b) Enhanced Mask Decoder

I 是绝对位置embedding或者先前的层输出的信息

- **虚拟对抗训练：**
  - 虚拟对抗训练是一种改进模型泛化的正则化方法。 它通过对抗性样本提高模型的鲁棒性，对抗性样本是通过对输入进行细微扰动而创建的。 对模型进行正则化，以便在给出特定于任务的样本时，该模型产生的输出分布与该样本的对抗性扰动所产生的输出分布相同
  - 对于之前的NLP任务，一般会把扰动应用于单词嵌入，而不是原始单词序列。 但是嵌入向量值的范围在不同的单词和模型之间有所不同。 对于具有数十亿个参数的较大模型，方差会变大，从而导致对抗训练有些不稳定
  - SiFT通过应用扰动的归一化词嵌入来提高训练稳定性，即在实验中将DeBERTa微调到下游NLP任务时，SiFT首先将单词嵌入向量归一化为随机向量，然后将扰动应用于归一化的嵌入向量

4.2 GPT系列

4.2.1 GPT1

Generative Pre-Training

模型结构：Transformer only-decoder 12层，具体细节跟transformer一样，但是位置编码是可训练的。原transformer的decoder包含2个attention：cross-attention(k,v来自encoder，q来自decoder)，mask multi-head attention。gpt只用了mask multi-head attention

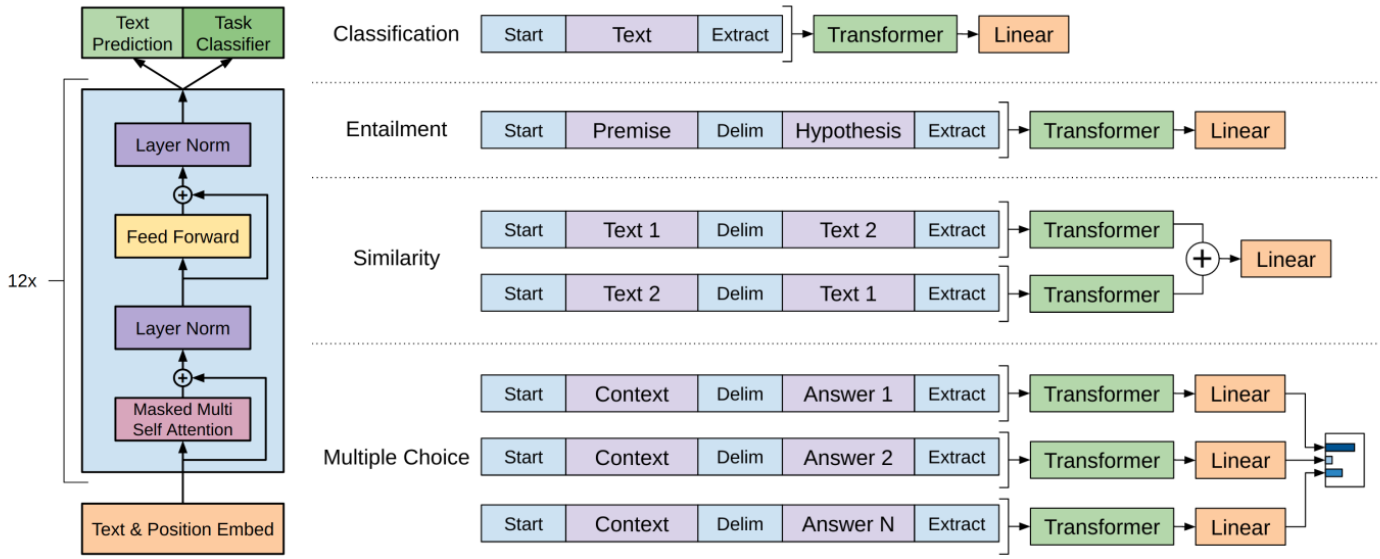


Figure 1: **(left)** Transformer architecture and training objectives used in this work. **(right)** Input transformations for fine-tuning on different tasks. We convert all structured inputs into token sequences to be processed by our pre-trained model, followed by a linear+softmax layer.

训练范式：自监督预训练 + 有监督fine-tune 主要思想是无监督学习

- **预训练的标准语言模型目标函数**，根据前面K个词预测下一个词

$$L_1(\boldsymbol{u}) = \sum_i \log P(u_i|u_{i-k}, \ldots, u_{i-1}; \Theta)$$

- **微调的目标函数：**用的是完整的输入序列加标签，有监督目标函数加无监督的目标函数，y是标签，**加入无监督目标函数的作用：1. 增加SFT模型的泛化性，2. 加速收敛**

$$L_2(C) = \sum_{(x,y)} \log P(y|x^1, \ldots, x^m)$$

$$L_3(C) = L_2(C) + \lambda * L_1(C)$$

- 改变输入形式【通过在序列前后添加 [Start] 和 [Extract] 特殊标识符来表示开始和结束，序列之间添加必要的 [Delim] 标识符来表示分隔】，接上对应下游任务的层，就可实现不同下游任务。**【注：利用最后一层的最后一个token的输出，接下游层即可完成各类任务】**

4.2.2 GPT2

Language Models are Unsupervised Multitask Learners

模型结构：与GPT1基本一致，但是**post-norm改为pre-norm，输入序列512改为1024，48层**

训练范式：预训练 + zero-shot **主要思想是多任务学习**

- 学习目标是**使用无监督的预训练模型做有监督的任务**。基于上面的思想，当一个语言模型的容量足够大时，它就足以覆盖所有的有监督任务，也就是说所有的有监督学习都是无监督语言模型的一个子集。例如当模型训练完“Micheal Jordan is the best basketball player in the history”语料的语言模型之后，便也学会了(question：“who is the best basketball player in the history ?”，answer:“Micheal Jordan”)的Q&A任务
- GPT-2可以在zero-shot设定下实现下游任务，即不需要用有标签的数据再微调训练
- 为实现zero-shot，下游任务的输入就不能像GPT那样在构造输入时加入开始、中间和结束的特殊字符，这些是模型在预训练时没有见过的，而是应该和预训练模型看到的文本一样，更像一个自然语言
- 可以通过做prompt的方式来zero-shot。例如机器翻译和阅读理解，可以把输入构造成，“请将下面的一段英语翻译成法语，英语，法语”
- 为何zero-shot这种方式是有效的呢？从一个尽可能大且多样化的数据集中一定能收集到不同领域不同任务相关的自然语言描述示例，数据集里就存在展示了这些prompt示例，所以训练出来就自然而然有一定zero-shot的能力了

核心思想：GPT-2的核心思想概括为：任何有监督任务都是语言模型的一个子集，当模型的容量非常大且数据量足够丰富时，仅仅靠训练语言模型的学习便可以完成其他有监督学习的任务

实验数据：数据从Reddit中爬取出来的优质文档，共800万个文档，40GB

与GPT-1的区别：

- 模型结构上，layer-norm的位置有所调整；参数初始化的方式有所改变
- gpt2主推zero-shot, 而gpt1主推pre-train+finetune
- 数据量加大，gpt2(40G), gpt1(5G)
- gpt2最大模型为15亿参数，gpt1最大模型为1亿参数

GPT-2的最大贡献是验证了通过海量数据和大量参数训练出来的词向量模型有迁移到其它类别任务中而不需要额外的训练

shot区别	Example
Zero shot	**task description + prompt:** Translate English to French: cheese =>
One shot	**task description + example + prompt:** Translate English to French: **sea otter => loutre de mer** cheese =>
Few shot	**task description + examples** Translate English to French:

	+ prompt:	**sea otter => loutre de mer** **peppermint => menthe poivrée** **plush giraffe => girafe peluche** cheese =>

4.2.3 GPT3

Language Models are Few-Shot Learners

模型结构：与GPT-2一样，但是应用了 **Sparse attention**

- Dense attention：每个 token 之间两两计算 attention，**复杂度 O(n^2)**
- Sparse attention：每个 token 只与其他 token 的一个子集计算 attention，**复杂度 O(n*logn)**

使用 sparse attention 的好处主要有以下两点：

1. **减少注意力层的计算复杂度，节约显存和耗时**，从而能够处理**更长的输入序列**；
2. 具有“局部紧密相关和远程稀疏相关”的特性，对于**距离较近的上下文关注更多，对于距离较远的上下文关注较少**；

训练范式：预训练 + **few-shot / in-context learning**

与GPT-2区别：

- 模型结构上来看，在gpt2的基础上，将attention改为了sparse attention
- 效果上远超gpt2，生成的内容更为真实
- **gpt3主推few-shot，而gpt2主推zero-shot**
- 数据量远大于gpt2：gpt3(45T，清洗后570G)，gpt2(40G)
- gpt3最大模型参数为1750亿，gpt2最大为15亿

GPT-3的 In-context learning 与 元学习的关联：

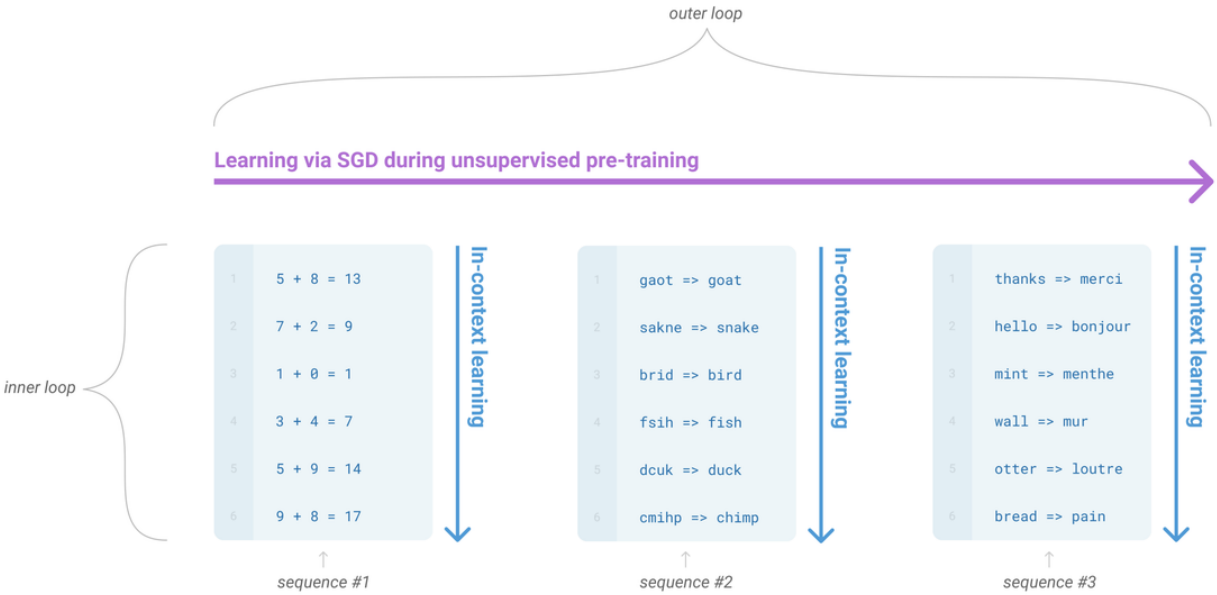


Figure 1.1: Language model meta-learning. During unsupervised pre-training, a language model develops a broad set of skills and pattern recognition abilities. It then uses these abilities at inference time to rapidly adapt to or recognize the desired task. We use the term "in-context learning" to describe the inner loop of this process, which occurs within the forward-pass upon each sequence. The sequences in this diagram are not intended to be representative of the data a model would see during pre-training, but are intended to show that there are sometimes repeated sub-tasks embedded within a single sequence.

4.3 LLaMA系列

Open and Efficient Foundation Language Models，目前主流的开源大模型(Dense)基本上都是LLaMA架构，并在此基础上做各种改动

4.3.1 LLaMA1

模型结构：相比GPT做出来如下改动

- 为了增强训练稳定性，采用**pre-RMSNorm**作为层归一化方法
- 为了提高模型性能，采用**SwiGLU**作为激活函数
- 为了更好地建模长序列数据，采用**RoPE**作为位置编码

为了更好地编码数据，Llama-1使用**BPE**算法进行分词，具体由**sentencepiece进行实现**。值得注意的是，Llama-1将**所有数字分解为单独的数字，并对未知的UTF-8字符回退到字节进行分解，词表大小为32k**

训练数据：海量无标注数据进行自监督学习 1.4T token，这些预训练数据由多个来源混合而成，且

都是公开的数据，各个来源的数据量和采样比例见下表

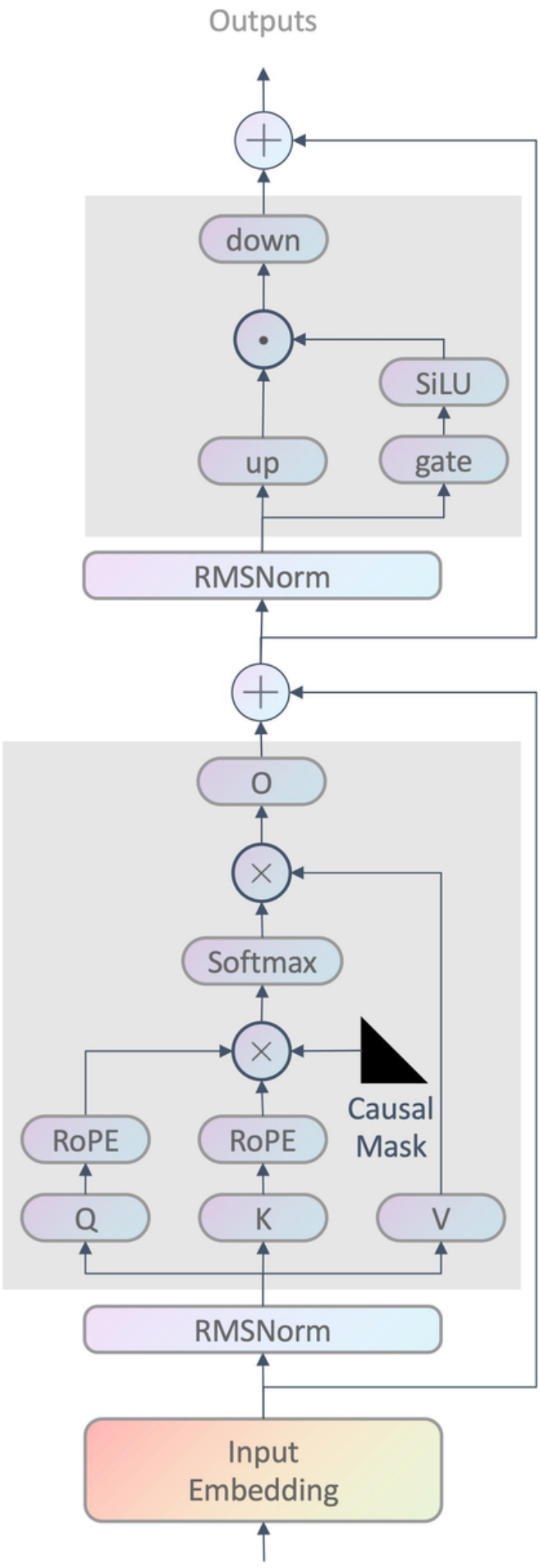
Dataset	Sampling prop.	Epochs	Disk size
CommonCrawl	67.0%	1.10	3.3 TB
C4	15.0%	1.06	783 GB
Github	4.5%	0.64	328 GB
Wikipedia	4.5%	2.45	83 GB
Books	4.5%	2.23	85 GB
ArXiv	2.5%	1.06	92 GB
StackExchange	2.0%	1.03	78 GB

训练方式：基础的**自监督学习模型**，**没有**经过**任何形式的特定任务微调**

Llama-1在公布的技术报告中详细描述了使用**AdamW优化器**的机器学习模型的具体训练配置。AdamW是对Adam优化器的改进，可以更有效地处理权重衰减，从而提高训练的稳定性。β1和β2参数的选择影响训练过程的收敛行为和稳定性

Llama-1描述的**余弦学习率调度**是一种有效的技术，用于在训练期间调整学习率，通过逐渐减少学习率在某些情况下可以导致更好的收敛。实施**0.1的权重衰减和1.0的梯度裁剪是预防过拟合和确保数值稳定性的标准做法**

warmup是一种策略性方法，旨在训练过程初期稳定训练动态。根据模型大小调整学习率和batch是一种优化资源分配和效率的实用方法，有可能提高模型性能

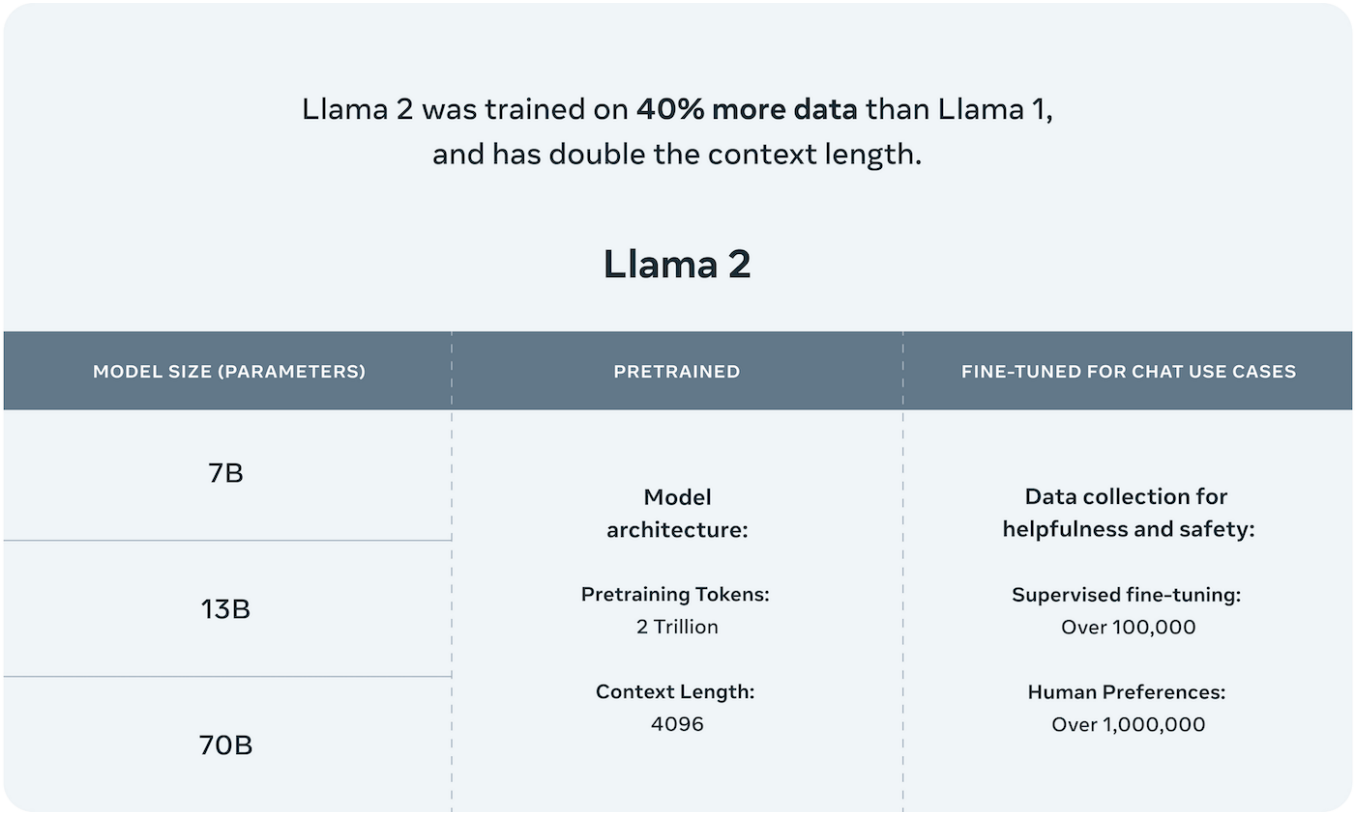


4.3.2 LLaMA2

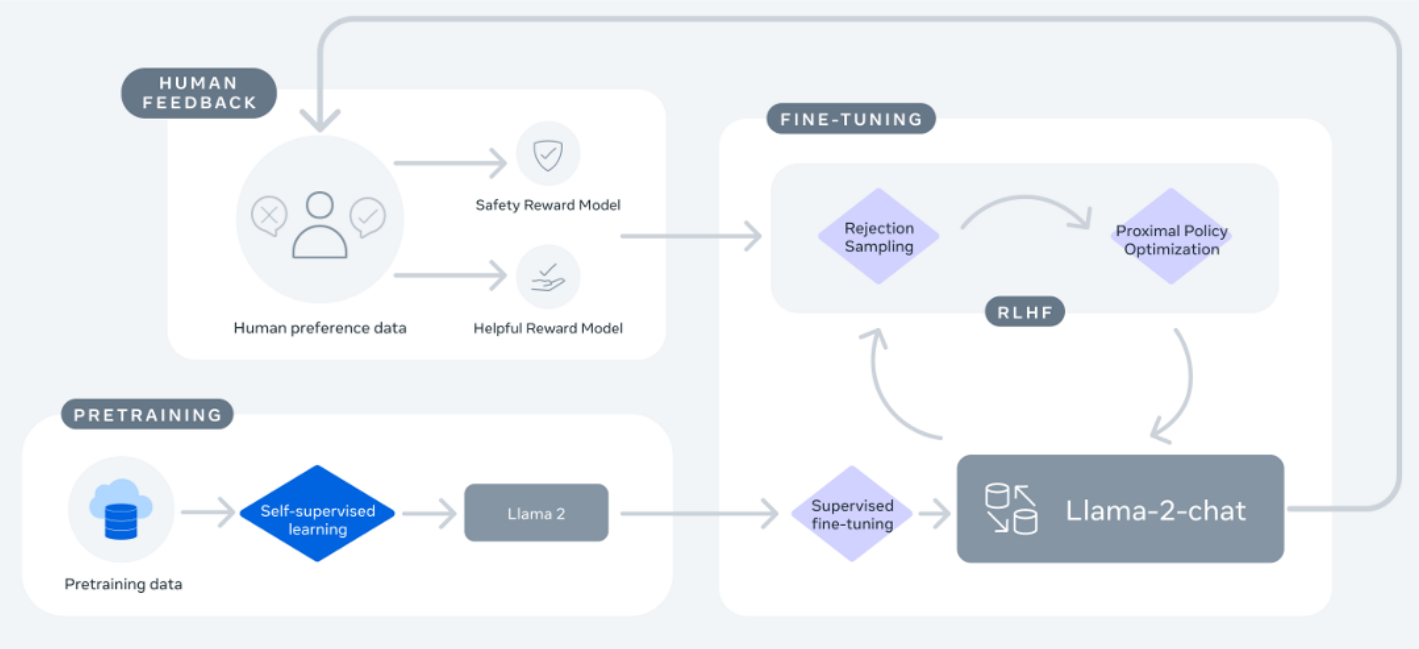
模型结构：

- 跟LLaMA1相比大参数模型**多了GQA**，整体参数量会有减少
- **FFN模块矩阵维度有扩充，**增强泛化能力，整体参数量增加
- 其他区别体现在**更多的训练数据、更长的上下文窗口达到了4K**，LLaMA1是2K

训练数据：Llama-2 预训练使用了来自公开可用源的 2T个数据token（未详细指出具体的开源数据）

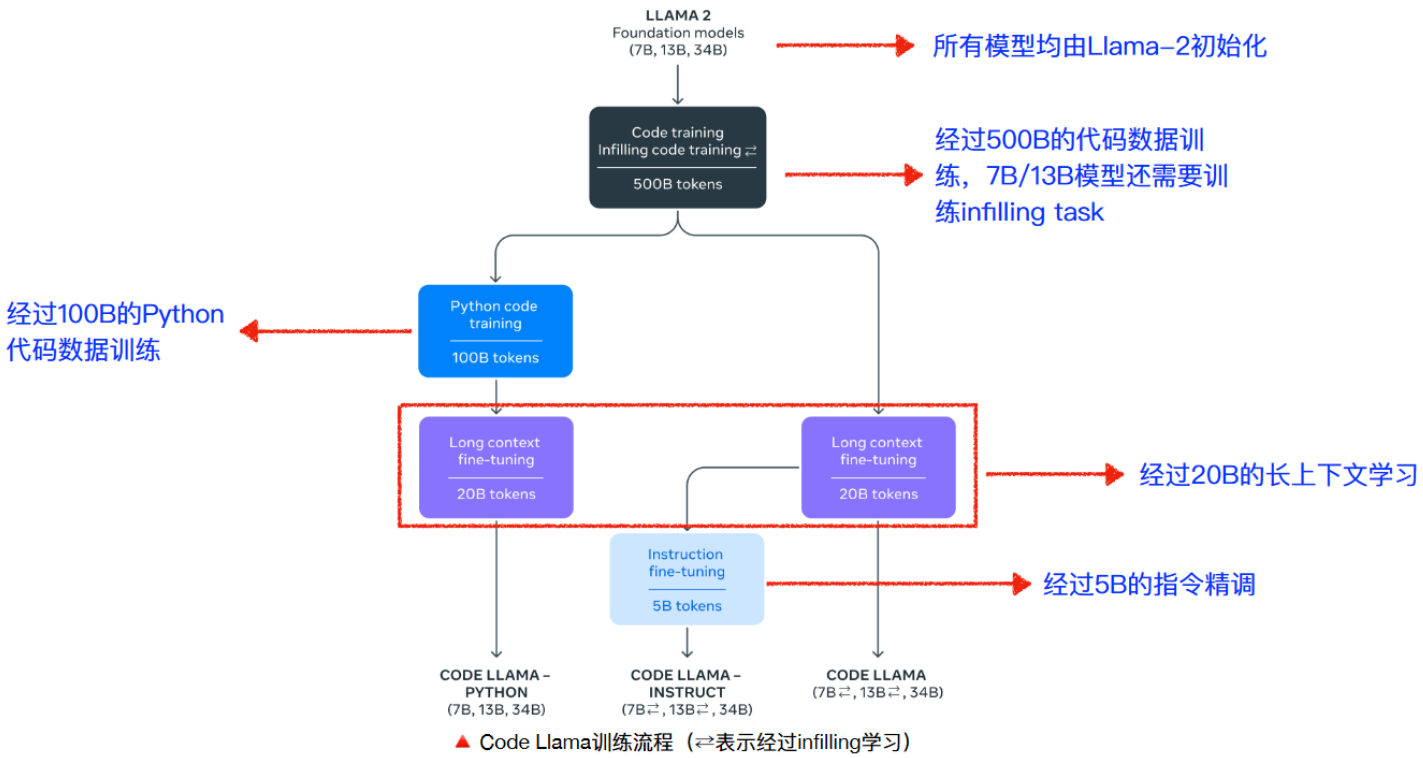


训练过程：通过结合人类反馈和**拒绝采样**，模型能够更有效地利用稀缺的高质量数据，最大化训练的效率和效果



CodeLlama：基于Llama-2训练的领域模型

训练流程：



▲ Code Llama训练流程（⇄表示经过infilling学习）

CodeLLaMA7B，13B模型的infilling task：

任务目标：根据代码的上下文，预测残缺部分的代码

- 从完整的代码中选择一部分进行掩码（mask）并替换为 `<MASK>` 符号，构成上下文
- 利用自回归的方法，根据上下文信息预测解码出被mask的代码部分

Original Document

```
def count_words(filename: str) -> Dict[str, int]:
    """Count the number of occurrences of each word in the file."""
    with open(filename, 'r') as f:
        word_counts = {}
        for line in f:
            for word in line.split():
                if word in word_counts:
                    word_counts[word] += 1
                else:
                    word_counts[word] = 1
    return word_counts
```

Masked Document

```
def count_words(filename: str) -> Dict[str, int]:
    """Count the number of occurrences of each word in the file."""
    with open(filename, 'r') as f:
        <MASK:0> in word_counts:
                    word_counts[word] += 1
                else:
                    word_counts[word] = 1
    return word_counts
<MASK:0> word_counts = {}
        for line in f:
            for word in line.split():
                if word <EOM>
```

4.3.3 LLaMA3

模型结构：

- 与LLaMA2相比，LLaMA3将**tokenizer由sentencepiece换成了tiktoken**，这与GPT4保持一致，**词表大小由32k扩展到了128k**
- LLaMA3 **8B和70B都采用了GQA**
- **同时上下文长度也扩展到了8k（一开始，预训练后期用多阶段长文本训练达到了128K）**

	8B	70B	405B
Layers	32	80	126
Model Dimension	4,096	8192	16,384
FFN Dimension	14,336	28,672	53,248
Attention Heads	32	64	128
Key/Value Heads	8	8	8
Peak Learning Rate	3×10^{-4}	1.5×10^{-4}	8×10^{-5}
Activation Function	SwiGLU		
Vocabulary Size	128,000		
Positional Embeddings	RoPE ($\theta = 500,000$)		

	Training Data	Params	Context length	GOA	Token count
Llama 3	A new mix of publicly available online data	8B	8k	Yes	15T+
		70B	8k	Yes	

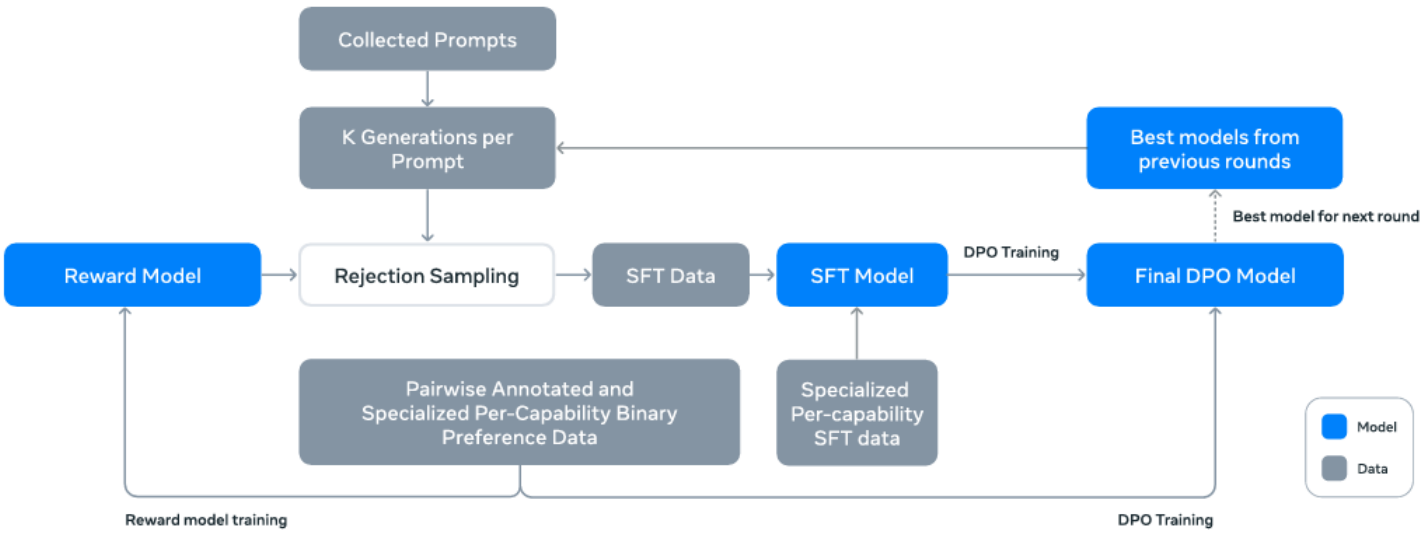
训练数据：精心设计了预训练语料库，数量&质量，扩展到了**15T Tokens**，增长了约8倍。其中，**代码数据扩充了4倍，显著提升了模型在代码能力和逻辑推理能力方面的表现**

值得注意的是，LLaMA-3并没有采用MOE（Mixture of Experts）结构，这种结构主要用于降低训练和推理成本，但在性能上通常无法与同规模的密集型（Dense）模型相比。随着模型规模的扩大，如何降低推理成本将成为一个需要关注的问题。此外，LLaMA-3的训练数据包括了**大量的代码token和超过5%的非英语token，来源于30多种语言**。这不仅使得模型在处理英语内容时更加高效，也显著提升了其多语言处理能力

为确保数据质量，Meta开发了**一系列数据过滤pipeline，包括启发式过滤器、NSFW过滤器、语义重复数据删除技术及用于预测数据质量的文本分类器**。这些工具的有效性得益于先前版本Llama的表现，特别是在识别高质量数据方面

训练流程：Llama-3系列也有两个模型——**预训练模型Llama-3和微调后的模型Llama-3-Instruct**

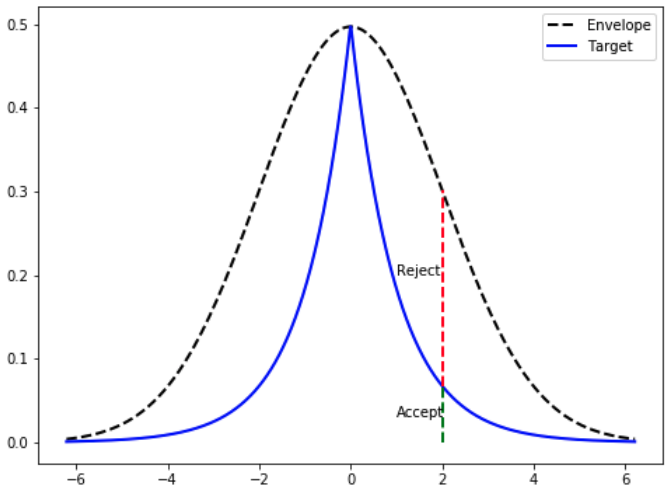
- **预训练阶段：**为了有效地利用预训练数据，Llama-3投入了大量精力来扩大预训练。具体而言，通过为下游基准测试制定一系列**扩展法则（scaling laws）**，使得在训练之前就能预测出模型在关键任务上的性能，进而选择最佳的数据组合
- **后训练阶段：监督式微调（SFT）、拒绝采样、RLHF和直接策略优化（DPO）的组合进行多轮对齐。用于SFT的提示质量和用于PPO和DPO的偏好排名对对齐模型的性能有巨大影响。LLaMA3在模型质量上的一些最大改进来自于仔细筛选这些数据，并对人类标注者提供的多轮质量保证进行多次审查**



整个流程其实就是一个Online DPO/Iterative DPO

Reject Sampling 拒绝采样

- RS是一种从目标概率分布中获取样本的基本蒙特卡洛方法，主要用于在难以直接从目标分布采样时，通过一个更容易采样的提议分布来间接获取目标分布的样本。假设我们想要从一个复杂的目标概率分布中采样，但是直接对采样很困难。我们可以选择一个比较简单的提议分布，这个提议分布通常是比较容易采样的，比如均匀分布、正态分布等
- 在LLM这里通常是**模型对同一个prompt生成K个response，然后利用Reward Model对这些答案进行评分**，挑选出得分最高、表现最优的答案。这一过程不仅提升了模型的生成质量，也为模型的进一步训练提供了优质样本



LLaMA3对DPO的改进

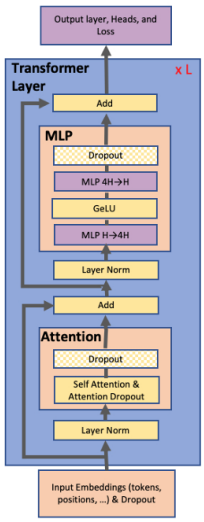
- **在DPO loss中mask格式化的token：**包括标题和终止token，以稳定DPO训练因为他们观察到，这些token对损失的贡献可能导致模型行为不理想，例如尾部重复或突然生成终止token。假设这是由于DPO损失的对比性质——在**chosen和reject response中存在共同token导致学习目标冲突**，因为**模型需要同时增加和减少这些token的概率**
- **使用NLL损失进行正则化：加一个chosen response的负对数似然loss(NLL loss)进行正则化**，loss系数为0.2，有助于通过保持生成的期望格式并防止选择response的对数概率下降，来进一步稳定DPO训练

4.4 GLM系列

General Language Model Pretraining with Autoregressive Blank Infilling

4.4.1 GLM1

模型结构：GLM1是**prefix-decoder结构的Transformer**（实际上就是transformer decoder，只不过通过特殊的mask实现了prefix部分双向attention，后面的部分单向attention），然后做了一些改动：

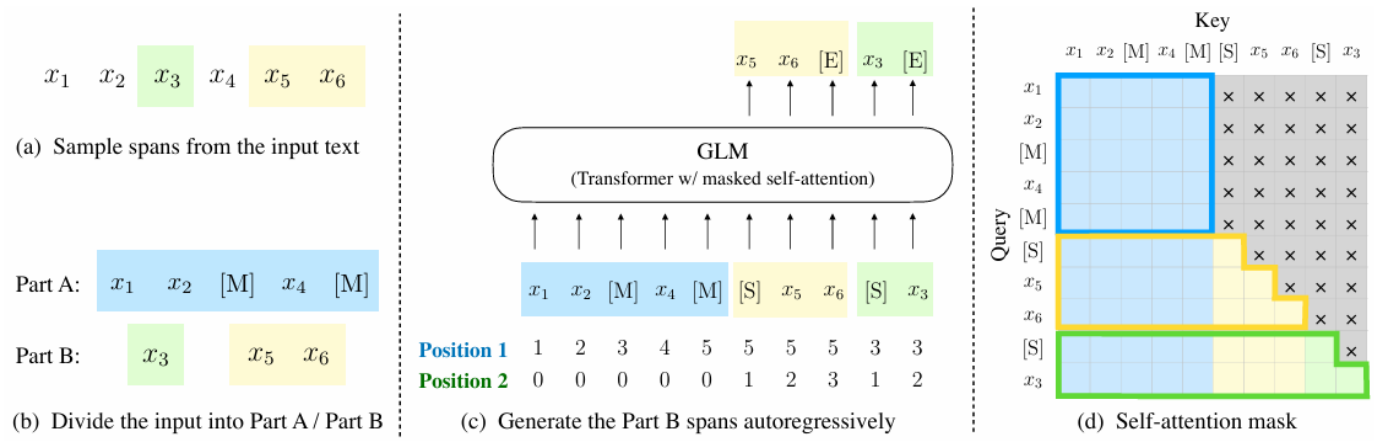


- **Post Deep Norm**
- 用一个单层线性层用来output token预测
- ReLU --> **GeLU 激活函数**

训练目标：

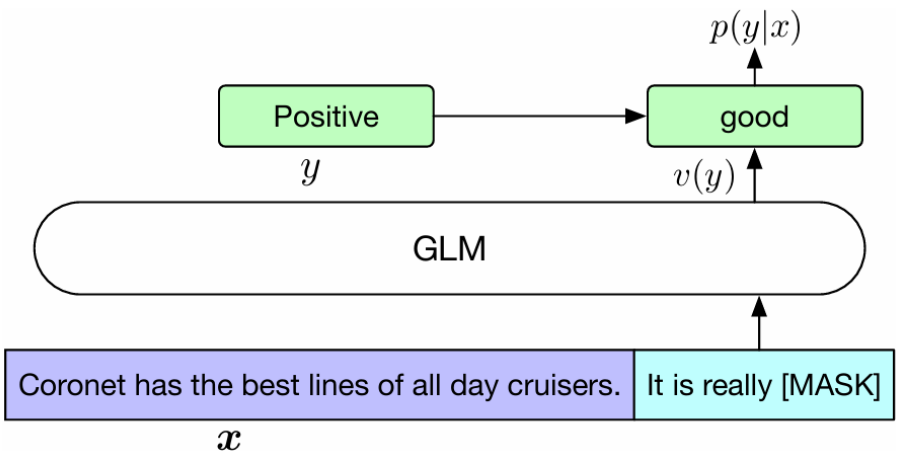
- **自回归填空：**这个任务包含了 **自编码思想：**在输入文本中，随机删除连续的tokens；**自回归思想：**顺序重建连续tokens，在使用自回归方式预测缺失tokens时，模型既可以访问corrupted文本，又可以访问之前已经被预测的spans

  span shuffling + 二维位置编码技术：



- 通过**改变缺失spans的数量和长度**，自回归空格填充目标可以为条件生成和无条件生成任务预训练语言模型
- 输入 x 可以被分成两部分：**Part A是被mask的文本corrupt**，**Part B由masked spans组成**。假设原始输入文本是 $[x_1, x_2, x_3, x_4, x_5, x_6]$，采样的两个文本片段是 $[x_3]$ 以及 $[x_5, x_6]$，那么mask后的文本序列是： $x_1, x_2, [M], x_4, [M]$，即Part A；同时我们需要对Part B的片段进行shuffle，每个片段使用 `[S]` 填充在开头作为输入，使用 `[E]` 填充在末尾作为输出
- **二维位置编码：**在GLM中，使用二维位置编码，第一个位置id用来标记Part A中的位置，第二个位置id用来表示span内部的相对位置，这两个位置id会通过embedding表被投影为两个向量，最终都会被加入到输入token的embedding表达中
- GLM中自定义attention mask的：
  - **Part A中的tokens彼此可见，但是不可见B中的任意tokens(对应prefix部分是双向attention)**
  - **Part B tokens可见Part A，Part B tokens可见B中过去的tokens，不可见B中未来的tokens（对应B中的是masked attention/casual attention）**
- 模型可以**自动学习双向encoder（Part A）以及单向decoder（Part B）**

- **多任务预训练：**生成更长文本与空白填充目标共同优化
  - Doc-level 文档级：从原始长度的 50% - 100% 范围内均匀采样一个span，该目标旨在生成长文本
  - Sentence-level 句子级：限制被mask的span必须是完整的句子，采样多个span以覆盖原始token的 15%（和BERT的15%一致），这个目标适用于 seq2seq（序列到序列）任务，这些任务的预测结果通常是完整的句子或段落
- **GLM做分类任务示例：**



4.4.2 GLM2

模型结构：

- 完全**Decoder-only**，**RoPE**，Gelu-->**SwiGLU**，**Post-RMSnorm**

- **MHA->MQA**，实现更高效的推理
- 更长的上下文（基于**FalshAttention**，2K->32K），并在对话阶段使用8K上下文训练

训练目标：

- ChatGLM2-6B 使用了 GLM 的混合目标函数，经过了 1.4T 中英标识符的预训练与人类偏好对齐训练

📌 **为什么prefix-decoder回归casual-decoder?**

- **多轮对话用prefix-decoder需要构造多个数据来训练，而decoder-only的casual mask可以直接用整个多轮对话数据**，等效且各个对话的权重不一样，这个改动导致了二维编码的去除

  在处理多轮对话的过程中,设有3轮对话,Q1A1，Q2A2，Q3A3，PrefixLM需要构建三条样本:

  - Q1->A1
  - Q1A1Q2->A2
  - Q1A1Q2A2Q3->A3

  Decoder-only利用casual mask的性质在一条样本里面实现多轮对话:

  - 样本构建:Q1 A1 Q2 A2 Q3 A3
  - Loss计算:只需要计算 A1 A2 和 A3 部分
- LLaMA架构展现出decoder-only架构的强大的自回归生成能力

4.4.3 GLM3

模型结构：ChatGLM2与ChatGLM3模型架构是完全一致的，ChatGLM与后继者结构不同

相对于ChatGLM，ChatGLM2、ChatGLM3模型上的变化：

1. 词表的大小从ChatGLM的150528缩小为65024 （是ChatGLM2、3加载比ChatGLM快不少）
2. **位置编码从每个GLMBlock一份提升为全局一份**
3. SelfAttention之后的前馈网络有不同。ChatGLM用GeLU做激活；ChatGLM2,3用SwiGLU做激活

4.5 Qwen系列

4.5.1 Qwen1

模型结构：基于Transformer改进，类似LLaMA结构

- tiktoken BPE，选择词汇表 cl100k base 作为起点扩充词汇表，同时将数字拆分为单个数字，**最终词汇表大小约为 152K**

- **Untied Embedding：**Input embedding和Output embedding不进行权重共享，是单独的两个权重矩阵，但是代价是**增加了内存消耗，不过可以显著提升模型的性能**
- **RoPE**位置编码，**Pre-RMSNorm**，**SwiGLU**激活函数
- 在**大多数层移除偏执bias，但在注意力的 QKV 层中保留**，以增强模型的外推能力。通过添加偏差项，可以更好地捕捉输入数据的特征，从而提升模型在处理未知或未见数据时的表现。这种增强外推能力的做法对于处理复杂任务和应对多样化的数据输入非常重要

模型训练：

- 采用**标准的自回归语言模型训练目标**
- **训练时上下文长度为2048**
- 注意力模块采用**Flash Attention**技术，以提高计算效率并减少内存使用
- 使用**BF16**混合精度加速训练

上下文长度（外推能力）扩展：通过**动态NTK插值、LogN-Scaling和分层窗口Self-Attention等技术**，有效地扩展了模型的上下文长度，同时保持了性能

- **动态NTK插值：**大语言模型基础原理学习笔记
- **LogN - Scaling：**根据**熵不变性**以及一些合理的假设，可以得到一个新的缩放因子，从而得到一种Scaled Dot-Product Attention，LogN-Scaling可以根据上下文长度与训练长度的比值，对Q和V的点积进行重新缩放，确保注意力值的熵随着上下文长度的增长而保持稳定

$$\text{Attention}(Q,K,V)=\text{softmax}\left(\frac{\kappa\log n}{d}QK^{\top}\right)V$$

- **分层窗口Self-Attention：**使用分层窗口Self-Attention，将注意力限制在一个上下文窗口内，防止模型关注到太远的内容；在不同层采用不同的窗口大小，较低的层使用较短的窗口，而较高的层使用较长的窗口

综合上述技术，Qwen预训练长度2028，推理的时候可以处理8192

Reward Model训练：

- **预训练偏好模型（preference model pretraining）：**奖励模型由**同等大小Qwen模型+池化层**得来，用特殊的句子结束标记经过池化层的映射值作为模型奖励值
- **微调奖励模型：**
  - 分类系统：为确保prompt具备一定的多样性和复杂性，创建了一个包含约6600详细标签的分类系统
  - 平衡采样：并采用了一种平衡采样算法，以在选择提示时兼顾多样性和复杂性
  - 多样性采样：为了生成多样的回复，实验过程使**用了不同规模和采样策略的 Qwen 模型**，因为多样化的回复有助于降低标注难度并提高奖励模型的性能

4.5.2 Qwen2

模型结构：

- 与Qwen的区别**GQA、YaRN+双块注意力**（Dual Chunk Attention, DCA）
- Tokenizer BBPE，151643词表

模型训练：这里主要说一下RLHF的训练过程，分为离线+在线两阶段

- 离线训练阶段，使用预先收集的偏好数据集进行DPO训练
- 在线训练阶段，模型利用即时反馈的奖励模型不断改进其性能。从当前策略模型中采样多个response，奖励模型选择最受欢迎和最不受欢迎的响应，形成用于每个情节中DPO的偏好对。采用在线合并优化器（Online Merging Optimizer）来减轻**对齐税，即与人类偏好对齐时模型性能下降的问题**

4.6 Deepseek系列

4.6.1 Deepseek-V1

模型结构：

- 基于 LLaMA，采用 Pre-RMSNorm、SwiGLU和RoPE，67B 模型使用 GQA 优化推理成本，调整了层数
- 使用 BBPE 算法进行分词，训练了约 24GB 的语料库，词汇表大小为 102400

4.6.2 Deepseek-V2

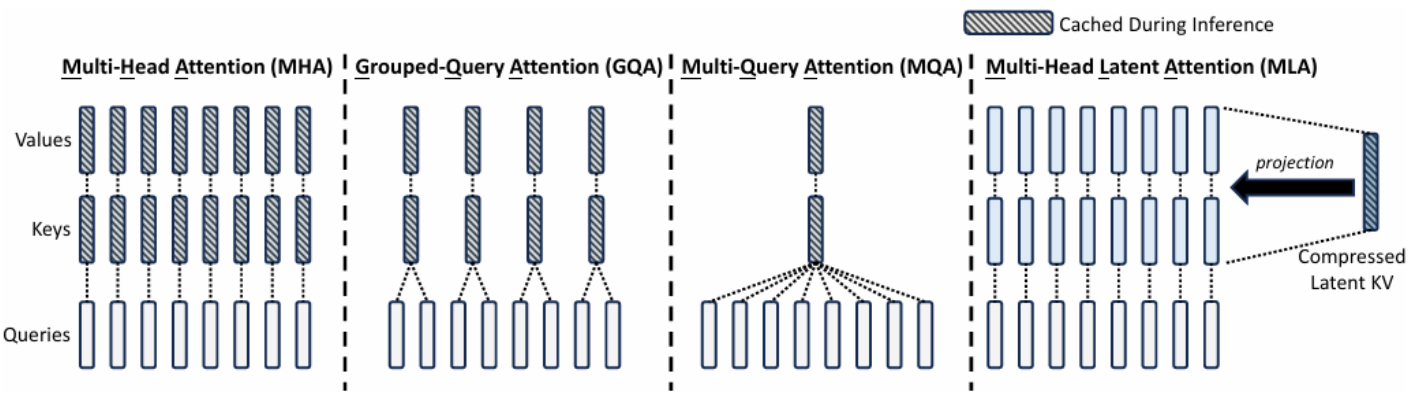
模型结构：

- 用到了**MLA（Multi-Latent Attention）**
- Deepseek-V2是一个**MoE模型**

模型训练：

- 强化学习阶段用到了**GRPO**算法

Multi-latent Attention：KV Cache的低秩投影



一般的MHA会把QKV分成 n_h 个头：

$$
\begin{aligned}
\left[q_{t,1};q_{t,2};\ldots;q_{t,n_h}\right] &= q_t \\
\left[k_{t,1};k_{t,2};\ldots;k_{t,n_h}\right] &= k_t \\
\left[v_{t,1};v_{t,2};\ldots;v_{t,n_h}\right] &= v_t \\
o_{t,i} &= \sum_{j=1}^{t} \mathrm{Softmax}_j\left(\frac{q_{t,i}^T k_{j,i}}{\sqrt{d_h}}\right) v_{j,i} \\
u_t &= W^o\left[o_{t,1};o_{t,2};\ldots;o_{t,n_h}\right]
\end{aligned}
$$

在推理过程中，所有的key和value都需要被cache以加速推理，因此**MHA需要为每个token缓存 $2n_h d_h l$ 个元素**，在模型部署时，这种**大量的KV Cache是一个瓶颈，它限制了最大batch size和序列长度**

- **低秩KV压缩：** $c_t^{KV} \in \mathbb{R}^{d_c}$ 是hidden state压缩了的隐层向量，其中 $d_c << d_h n_h$ 是KV压缩维度，$W^{DKV} \in \mathbb{R}^{d_c \times d}$ 是降采样矩阵， $W^{UK}, W^{UV} \in \mathbb{R}^{d_h n_h \times d_c}$ 是KV的上投影矩阵

$$
\begin{aligned}
c_t^{KV} &= W^{DKV} h_t \\
k_t^C &= W^{UK} c_t^{KV} \\
v_t^C &= W^{UV} c_t^{KV}
\end{aligned}
$$

  在推理的过程中，**MLA只需要缓存 c_t^{KV} ，只有 $d_c l$ 个元素， l 为层数；同时 W^{UK}, W^{UV} 可以被融入 W^Q, W^O （query和output的矩阵），所以并不需要单独再计算KV**

  同时MLA也对query做了低秩压缩，来减少训练中的激活值显存：

$$
\begin{aligned}
c_t^Q &= W^{DQ} h_t \\
q_t^C &= W^{UQ} c_t^Q \\
c_t^Q \in \mathbb{R}^{d_c'}, W^{DQ} &\in \mathbb{R}^{d_c' \times d}, W^{UQ} \in \mathbb{R}^{d_n n_h \times d_c'}
\end{aligned}
$$

- **解耦的RoPE：RoPE与低秩KV Cache并不兼容**，前面的压缩的时候可以将 W^{UK} 融进 W^Q 但是在加上RoPE的情况下不行，回忆一下RoPE： $(R_m q)^T (R_n k) = q^T R_m^T R_n k = q^T R_{m-n} k$ ，query和key中间有一个跟位置相关的矩阵，那么 $W^{UK} R_{m-n} W^Q$ 就不能合并成一个固定的投影矩阵了。

  MLA给出的解决方案是用**额外的多头queries和共享的key去结合RoPE**：

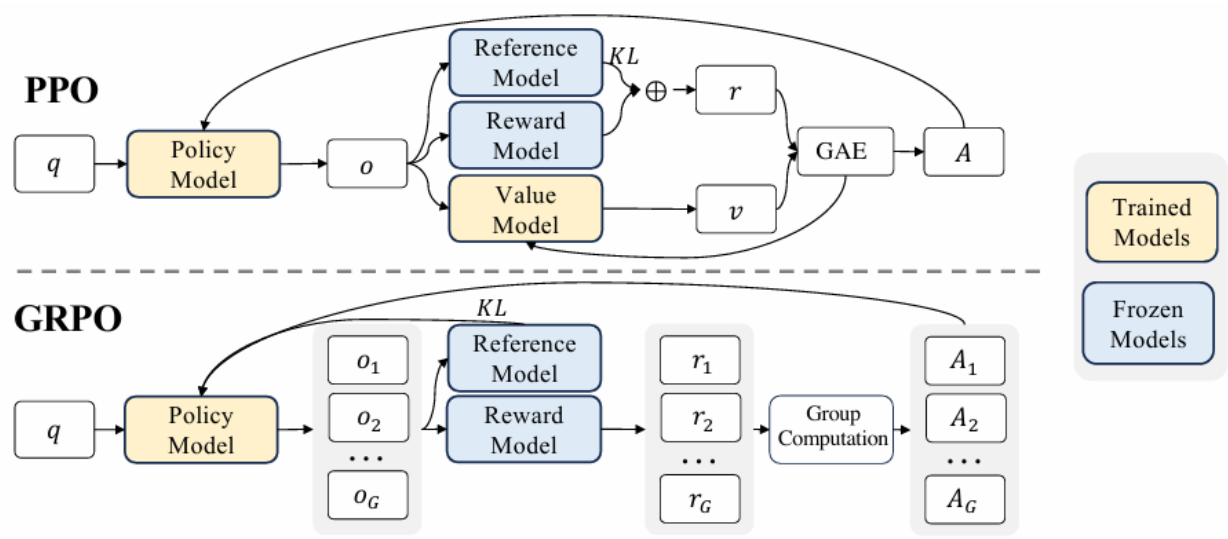
$$
\begin{aligned}
\left[q_{t,1}^R;q_{t,2}^R;\ldots;q_{t,n_h}^R\right] = q_t^R &= \mathrm{RoPE}(W^{QR} c_t^Q) \\
k_t^R &= \mathrm{RoPE}(W^{KR} h_t) \\
q_{t,i} &= [q_{t,i}^C; q_{t,i}^R] \\
k_{t,i} &= [k_{t,i}^C; k_t^R] \\
o_{t,i} &= \sum_{j=1}^{t} \mathrm{Softmax}_j\left(\frac{q_{t,i}^T k_{j,i}}{\sqrt{d_h + d_h^R}}\right) v_{j,i}^C \\
u_t &= W^O\left[o_{t,1};o_{t,2};\ldots;o_{t,n_h}\right]
\end{aligned}
$$

  其中 d_n^R 是单个head的维度， $W^{QR} \in \mathbb{R}^{d_h n_h \times d_c'}, W^{KR} \in \mathbb{R}^{d_h \times d}$ 是用来生成额外的解耦多头queries和共享key的矩阵，推理的时候需要把额外的共享key也缓存了，所以总共的缓存元素为 $(d_c + d_n^R) l$ 个

Attention Mechanism	KV Cache per Token (# Element)	Capability
Multi-Head Attention (MHA)	$2n_h d_h l$	Strong
Grouped-Query Attention (GQA)	$2n_g d_h l$	Moderate
Multi-Query Attention (MQA)	$2d_h l$	Weak
MLA (Ours)	$(d_c + d_h^R)l \approx \frac{9}{2} d_h l$	Stronger

GRPO算法： Group Relative Policy Optimization

回顾一下PPO里面的模型，我们为了计算GAE要用到Reference、Reward和Value三个model，GRPO通过对相同问题产生的多个采样输出的平均奖励作为value的估计值来去掉了Value model，优化强化学习的训练过程。更具体地说，对于每个问题，**GRPO从旧策略中采样一组输出** $\{o_1, o_2, \ldots, o_G\}$，**然后Reward model对这些回答都给出奖励值，最后奖励值的平均值将作为value的估计**



📌 **为什么可以这么做?**

回顾强化学习中价值函数的定义：

$$V(s) = \mathbb{E}_\pi[R_t + \gamma V(S_{t+1}) | S_t = s]$$

本身的定义就是未来的奖励总和的期望，但是由于LLM这里并没有状态转移，就是从图中的q到o，状态是q，动作是o，并没有下一个状态q'（多轮对话可以看成一直在状态转移），所以价值函数可以写为：

$$V(q) = \mathbb{E}_\pi[R_t]$$

那么对问题q进行多次采样回答计算奖励值的平均值作为Value的估计是合理的，前提是采样次数足够

5. Application 大模型应用

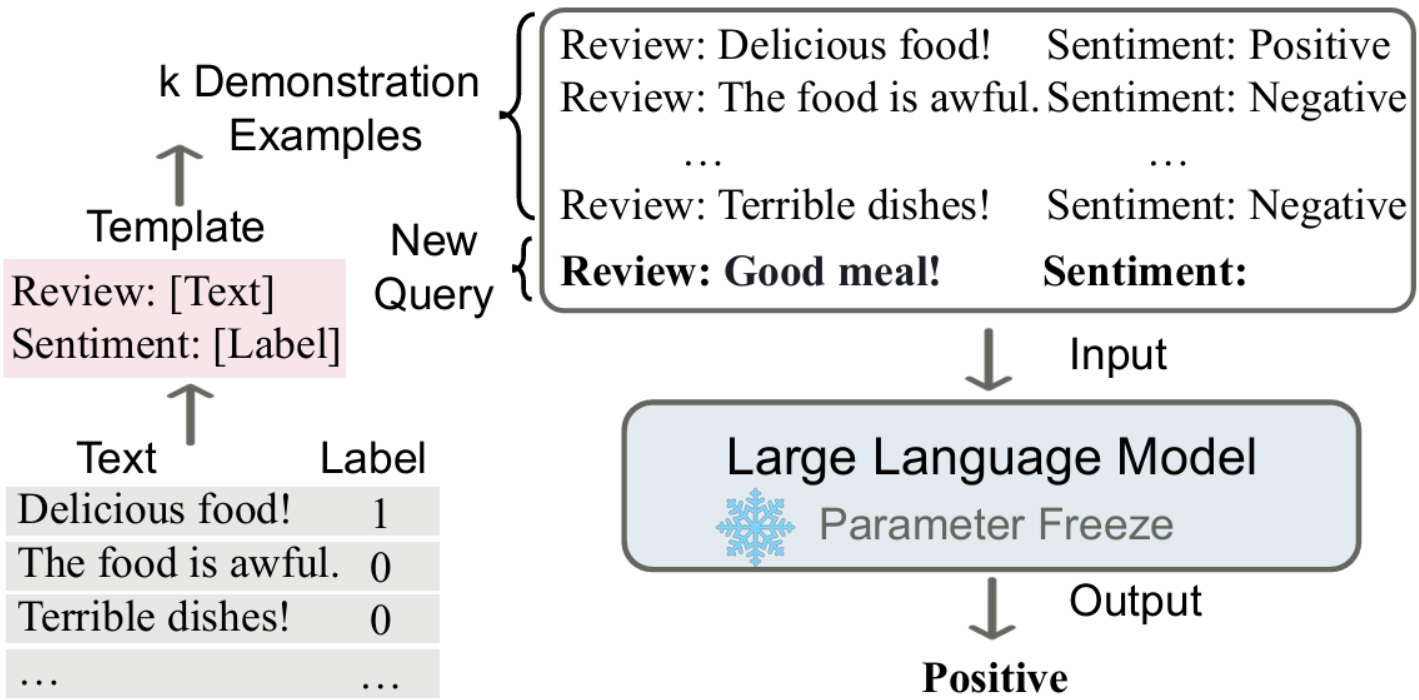
LLM已经得到了广泛的应用，包括各种**提示技术、Agent还有RAG**

5.1 Prompt Tech 提示技术

5.1.1 In-Context Learning 上下文学习

在前面讲GPT系列模型的时候，提到了GPT使用的Few-shot Learning，这就属于ICL的一个部分。

In-context Learning **允许模型在给定的上下文中进行学习和推理，而无需真正更新模型参数**。这种方法充分利用了模型的预训练知识，并通过在推理阶段提供相关的上下文信息来生成或调整模型输出



核心概念：

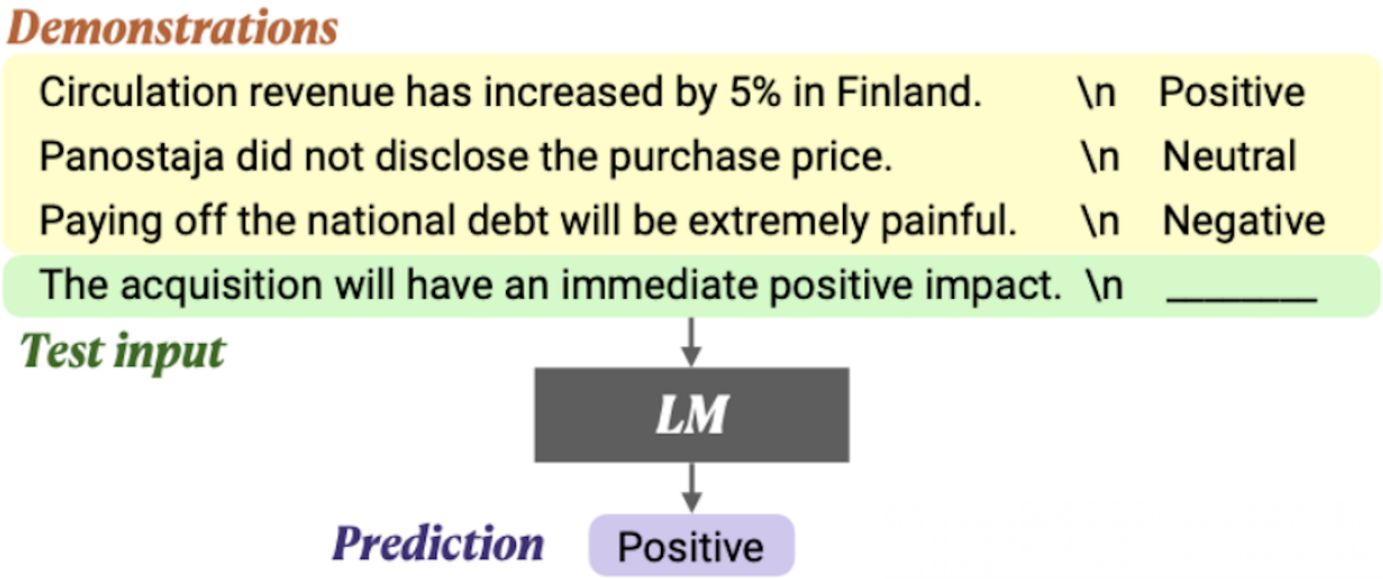
- **上下文依赖：**ICL的核心在于**利用模型的上下文理解能力来完成任务**，模型根据输入的上下文信息(包括示例和任务描述)进行推理，而**不是依赖于显示的训练过程**
- **无参数更新**：ICL**不涉及对模型实际参数的修改**，只是根据提供的上下文信息调整其生成或分类行为
- **动态适应：**模型在推理时会动态地适应给定的上下文，通过分析上下文中的示例或指示来生成合适的输出，这种适应能力来源于模型在预训练阶段学到的通用知识

工作原理：

- **提示词和示例：**ICL常通过提示词来引导模型的生成过程，提示词通常包括任务描述，问题陈述或请求模型执行的操作。在少样本学习(Few-Shot Learning)中，提示词可能包括一些示例输入和输出，帮助模型理解如何处理类似的任务
- **上下文提供**：在ICL中，任务描述用于告诉模型要完成的任务，例如：生成一个关于人工智能的总结。提供几个示例输入和输出对，可以帮助模型理解特定任务的模式或要求，例如：给出一些翻译示例来帮助模型进行语言翻译
- **推理和生成：**模型根据提供的上下文进行推理，生成与上下文相关的响应或输出。在ICL中，生成的文本基于模型对上下文的理解，以及预训练中学到的知识

ICL形式化定义： $\text{LLM}(I, f(x_1, y_1), \ldots, f(x_k, y_k), f(x_{k+1}, _)) \rightarrow \hat{y}_{k+1}$ ，其中 I 是任务定义，$f(x_i, y_i)$ 表示示例， x_{k+1} 是输入，__是要预测的answer

实例理解：



示例设计 Demonstration Design：

1. **示例选择**

   a. **启发式方法：**

   - **基于语义相似度：**使用BERT类模型作作为编码器，选取其CLS embedding的输出向量作为训练样本的文本表征。使用KNN、欧式距离，选取与当前test sentence语义最近的10个训练个样本，作为demonstration

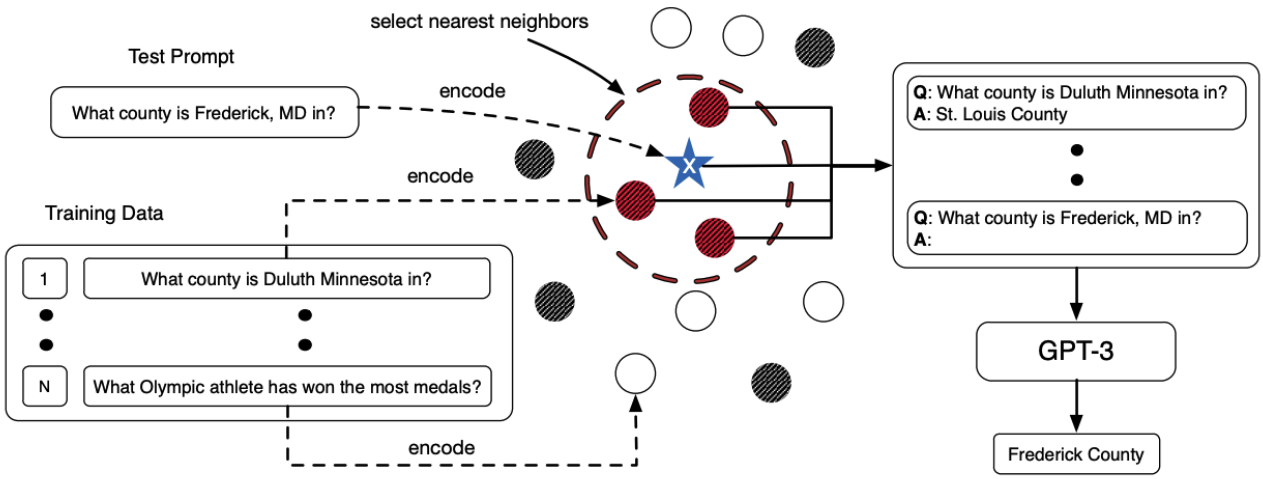


Figure 1: In-context example selection for GPT-3. White dots: unused training samples; grey dots: randomly sampled training samples; red dots: training samples selected by the k-nearest neighbors algorithm in the embedding space of a sentence encoder.

   - **基于多样性：**用来挑选demostration的训练数据和测试数据有时候可能会存在较大的分布差异，这个时候很难通过KNN等基于相似方法来挑选合适的demostration。从多样性的角度去挑选demostration，在原有相似度的基础上，考虑不同demostration之间的差异性。当两个demostration高度语义相似时，剔除其中一个

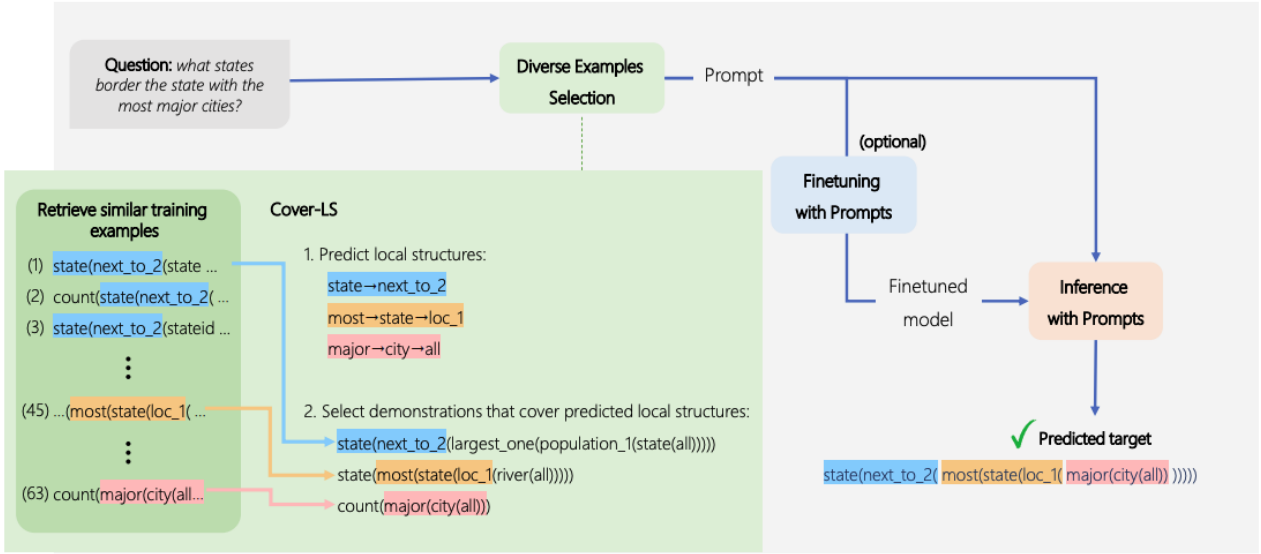


Figure 2: Overview of our framework. Given an utterance, we construct a prompt by selecting a set of diverse demonstrations. Feeding the prompt to the model yields the predicted target. Optionally, models can be fine-tuned (FT setup). In the bottom left corner, we see how *Cover-LS* selects diverse examples: predicting and covering *local structures*, thereby enabling the selection of complementary examples.

b. LLM-based方法：直接用LLM来生成Demonstration

- **基于prompt召回：**将训练集中的每一个样本都当做示例连同input一块输入模型，通过这样得到output的概率大小来评估当前示例的好坏，为了缓解数据集过大导致的计算成本，可以利用BM25、SBERT预先对训练集召回，筛选候选示例集，然后进行评估，选取最高分的k条示例作为正例集，最低分k条作为负例集，使用对比学习的方法，训练retrever。同时生成一个针对输入编码的Utterance encoder和针对示例编码的prompt encoder。基于训练好的retrever，结合Faiss相似度计算框架，找出输入input对应最佳的demostrations

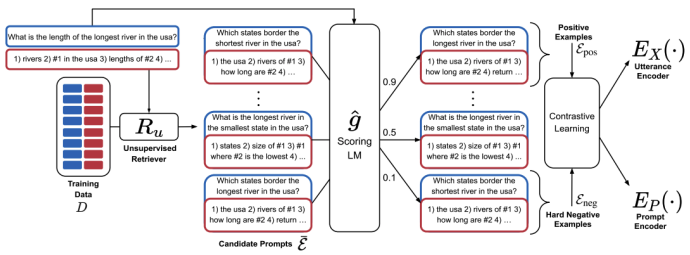
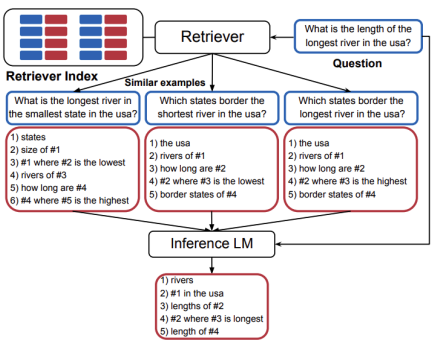


Figure 2: An overview of our approach for training EPR. Given a training example, we use an unsupervised retriever R_u to obtain a set of candidates. We then pass the candidates to a scoring LM and label the top-k and the bottom-k as positive and negative examples, respectively. Last, we use this training data to train a dense retriever.



- **基于主动学习/强化学习：**主动学习的思路是从样本库中选择适合的样本，提供给标注者标注。而示例选择问题与主动学习的思路有点类似，示例选择目标是选择适合的示例，提供给prompt，以使测试样本得到较高的准确率。由于demostrations的可选空间，与样本库（一般是训练集）呈指数关系，要枚举所有的demostrations组合，并不现实。因此可以把示例选择看作是一个序列决策问题，这样就可以基于马尔可夫决策过程（MDP），使用强化学习（RL）的方法去解决。state定义为当前时刻的demostration，即(xi,yi)，action定义为样本库中所有的样本和一个停止信号标识（⊥），reward定义为LLMs利用state与当前action构成的demostrations，在验证集中的准确率。即，prompt是由state+action构造成的demostrations，加上验证集中的样本输入一起组成。通过对比验证集标签与LLMs输出，可计算准确率。然后再选一种RL算法就OK了
- **基于自生成的：**利用LLMs自身的能力，生成合适的demonstration。目的是最小化对外部样本库的依赖。整个过程分为两个阶段：借助预先设计的prompt，让LLMs生成k个合适的示例；在原来输入的基础上，加入第一阶段生成的示例，让LLMs预测最终结果

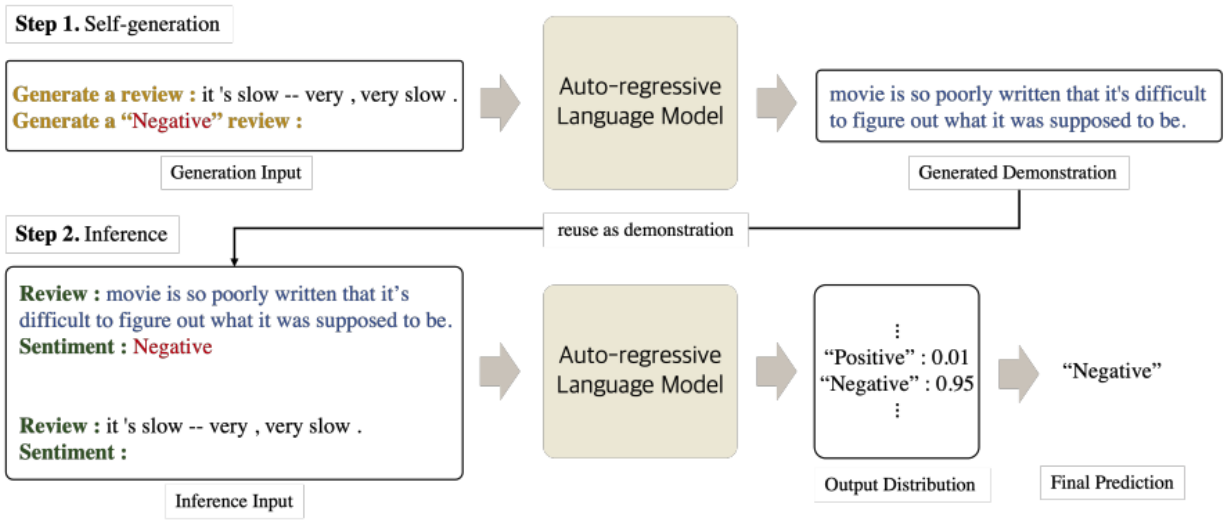


Figure 1: Overall process of SG-ICL. Texts in yellow are manually designed prompts for generation and the texts in red are expected class for generated demonstration. Demonstrations (colored in blue) are generated in the self-generation step and are reused for in-context learning in the inference step. Texts in green are manually designed inference prompts.

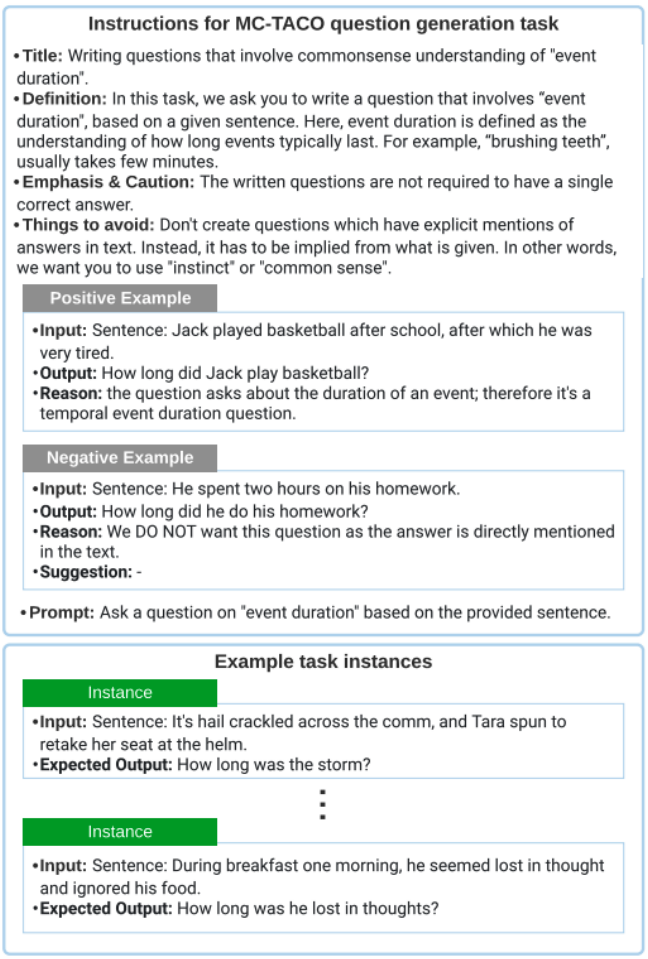
2. **示例格式**

在完成demonstrations的选择后，下一步就是将demonstrations整合成一个自然语言Prompt

《Cross-task generalization via natural language crowdsourcing instructions》

这篇论文提出了一个新的跨任务instruction数据集。他们使用总包的方式，按照规定的instruction格式，对多个开源数据集进行改造。针对每一个promt，其格式包含：

- **title:** 包含一个high-level的任务描述，以及其相关技能，如question generation，answer generation等
- **prompt:** 单独的文本命令，一般出现在输入示例之前
- **definition:** 指令的补充内容，更加详细地描述指令的具体执行细节
- **things to avoid:** 包含模型应该避免的内容，或规则
- **emphasis and caution:** 警告或反对的内容
- **positive examples:** 提供一个类似系统期望的输入、输出例子，使得众包人员更好地理解任务
- **negative examples:**提供一个类似系统期望的输入、输出的负例，让众包人员尽量避免
- **reason:** 解释为什么例子是positive或negative
- **suggestion:** 包含一些建议，主要用于指导如何将负例改成正例。



模型在经过上述instruction数据集微调后，能在unseen样本上，达到较好的生成效果。在新样本推断时，将demonstrations加入到task instances即可

3. **示例顺序**

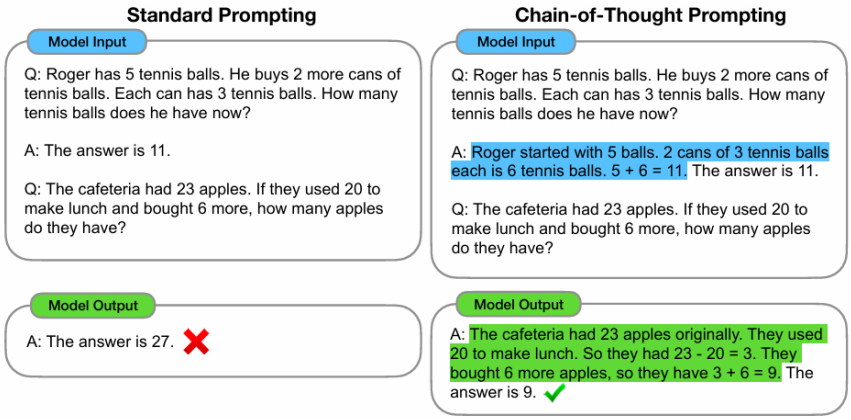
   在整合prompt的时候，各个示例放入的顺序也很关键，排序的方法既有不需要训练的，也有根据示例跟当前输入距离远近进行排序的，也可以根据自定义的熵指标进行重排

ICL的挑战：

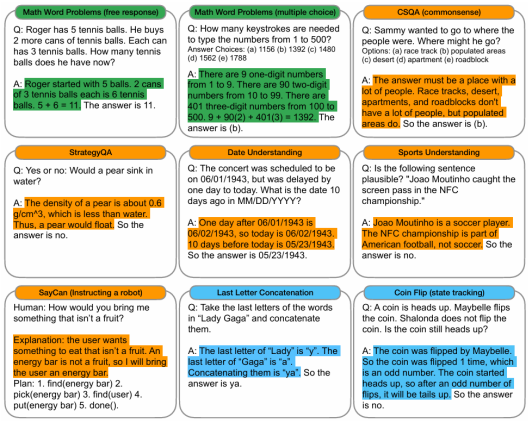
- **长上下文问题：**大规模模型对上下文长度有一定的限制，处理非常长的上下文可能会受到影响
- **选择适当的上下文：**确定哪些信息应该包含在上下文中，以及如何组织这些信息，是ICL的关键挑战之一
- **输出一致性：**在不同上下文中，模型生成的内容可能会有所不同，确保生成结果的一致性和准确性是需要关注的问题

5.1.2 Chain-of-Thought 思维链

Chain-of-Thought(CoT)是一种改进的Prompt技术，目的在于提升大模型LLMs在复杂推理任务上的表现，对于复杂问题尤其是复杂的数学题大模型很难直接给出正确答案。如算术推理（arithmetic reasoning）、常识推理（commonsense reasoning）、符号推理（symbolic reasoning）。COT通过要求模型在输出最终答案之前，显式输出中间逐步的推理步骤这一方法来增强大模型的算数、常识和推理能力，简单但有效。通过让**大模型逐步参与将一个复杂问题分解为一步一步的子问题并依次进行求解的过程可以显著提升大模型的性能**。区别于传统的 Prompt 从输入直接到输出的映射 <input——>output> 的方式，CoT 完成了从输入到思维链再到输出的映射，即 **<input——>reasoning chain——>output>**



CoT和普通提示的区别



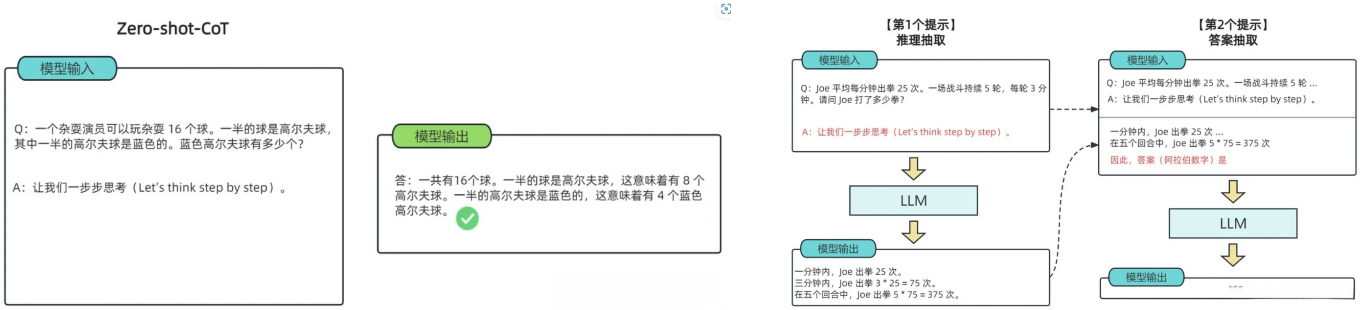
更多的示例

一个完整的包含 CoT 的 Prompt 往往由**指令（Instruction），逻辑依据（Rationale），示例（Exemplars）三部分组成**

- **指令：**用于描述问题并且告知大模型的输出格式
- **逻辑依据：**指 CoT 的中间推理过程，可以包含问题的解决方案、中间推理步骤以及与问题相关的任何外部知识
- **示例：**指以少样本的方式为大模型提供输入输出对的基本格式，每一个示例都包含：问题，推理过程与答案

以**是否包含示例**为区分，可以将 CoT 分为 **Zero-Shot-CoT 与 Few-Shot-CoT**：

- **Zero-Shot-CoT** 不添加示例而仅仅在指令中添加一行经典的“Let’ s think step by step”，就可以“唤醒”大模型的推理能力。其实 Zero-shot-CoT 是一个 pipeline。也就是说“Let's think step by step”这句话，只是通过这个 prompt 让LLM 尽可能生成一些思考过程，然后再将生成的 rationale（理由） 和 question 拼在一起，重新配合一个answer 指向的 prompt 如“The answer is ”来激励模型生成答案。从技术上讲，完整的零样本思维链（Zero-shot-CoT）过程涉及两个单独的提示/补全结果。在下右图中，左侧生成一个思维链，而右侧接收来自第一个提示（包括第一个提示本身）的输出，并从思维链中提取答案。这个第二个提示是一个自我增强的提示



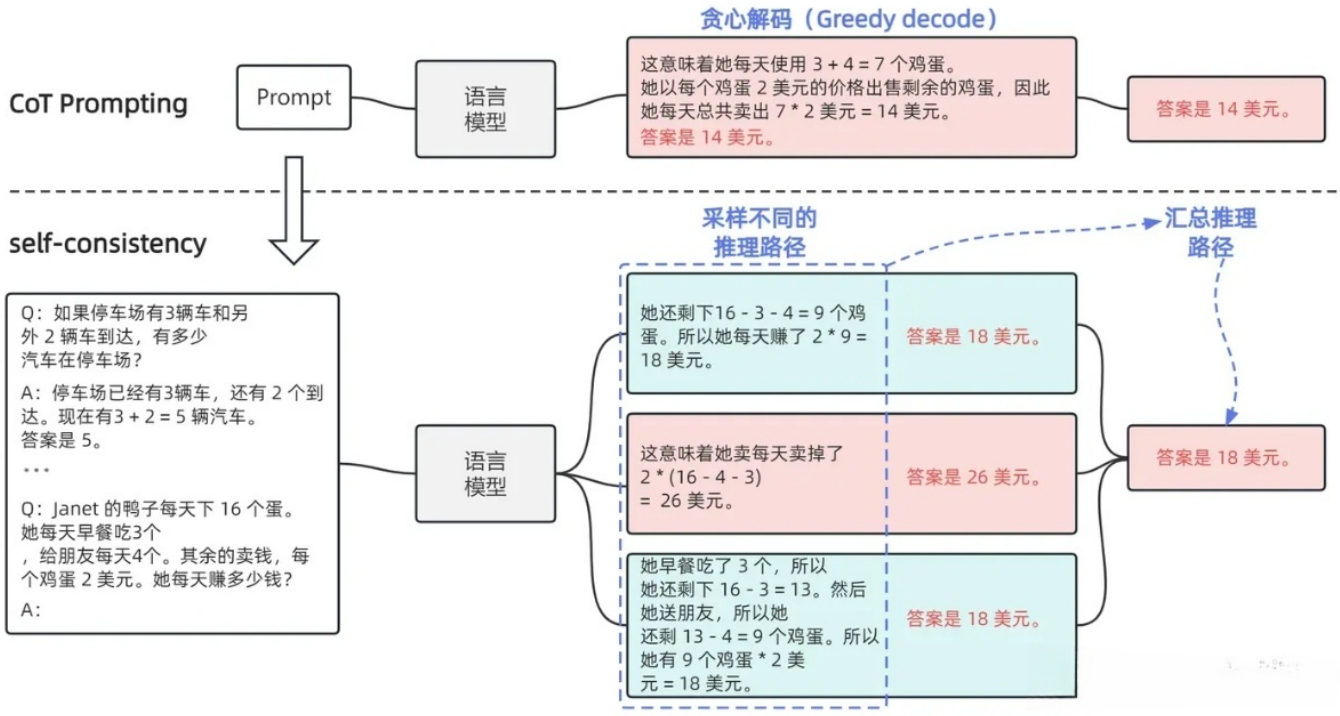
“Let's think step by step”是被验证过的

No.	Category	Template	Accuracy
1	instructive	Let’s think step by step.	**78.7**
2		First, (*1)	77.3
3		Let’s think about this logically.	74.5
4		Let’s solve this problem by splitting it into steps. (*2)	72.2
5		Let’s be realistic and think step by step.	70.8
6		Let’s think like a detective step by step.	70.3
7		Let’s think	57.5
8		Before we dive into the answer,	55.7
9		The answer is after the proof.	45.7
10	misleading	Don’t think. Just feel.	18.8
11		Let’s think step by step but reach an incorrect answer.	18.7
12		Let’s count the number of "a" in the question.	16.7
13		By using the fact that the earth is round,	9.3
14	irrelevant	By the way, I found a good restaurant nearby.	17.5
15		Abrakadabra!	15.5
16		It’s a beautiful day.	13.1

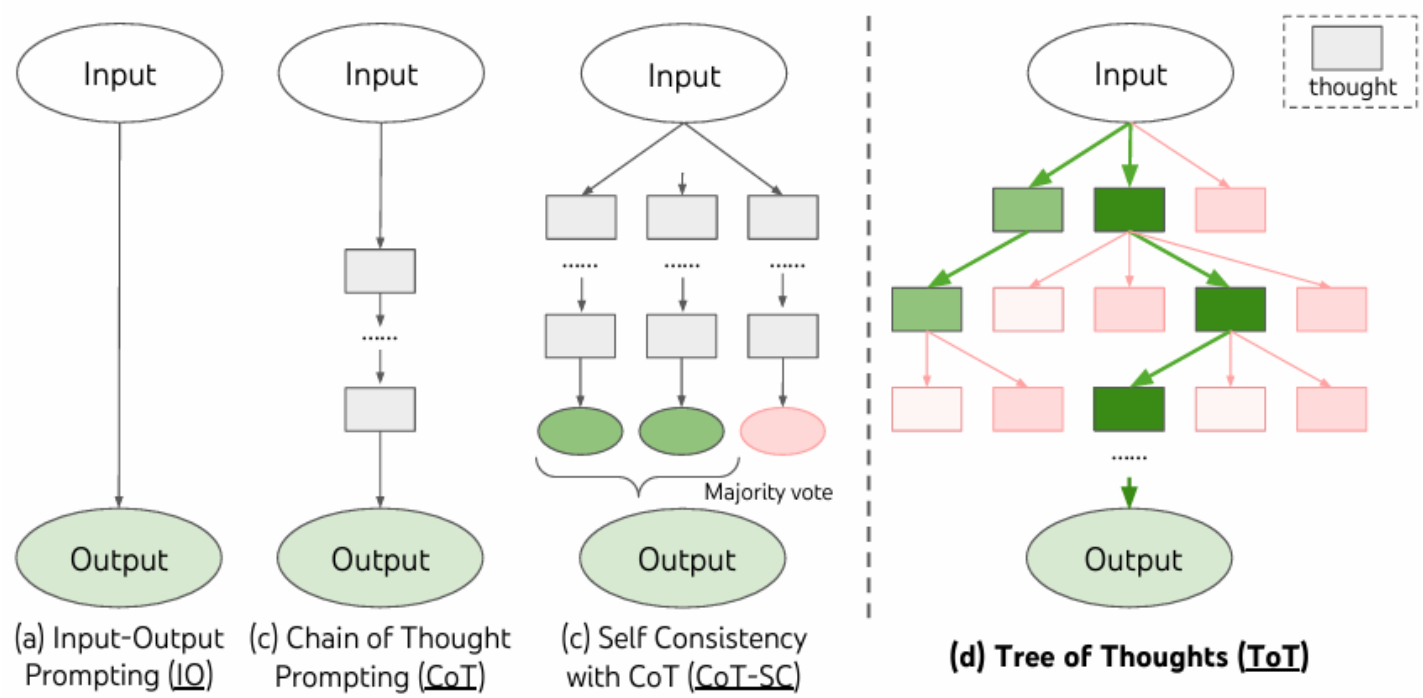
- **Few-Shot-Cot** 则在示例中详细描述了“解题步骤”，让模型照猫画虎得到推理能力

CoT的各种改进方法：

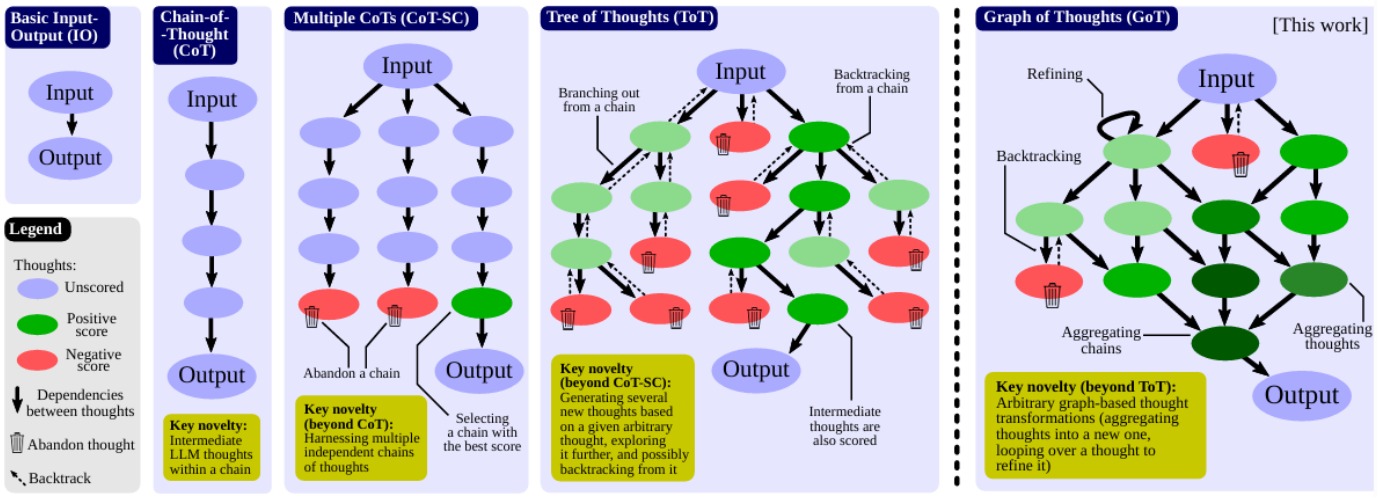
- **CoT-SC：Self Consistency**，主要改进是对答案进行了多数投票（majority vote），是生成多个思路链，然后取多数答案作为最终答案，并且发现其可以显著地提高思维链方法的性能。在下面的图中，左侧的提示是使用少样本思维链范例编写的。使用这个提示，独立生成多个思维链，从每个思维链中提取答案，通过“边缘化推理路径”来计算最终答案。实际上，这意味着取多数答案



- **ToT：**Tree-of-Thoughts，思维树。其做法是通过树（tree）来建模 LLM 推理过程。这能让模型使用不同的思维路径，并能提供全新的功能，比如基于不好的结果反向回溯推理过程。不幸的是，由于 ToT 方法为思维过程强加了严格的树结构，所以会极大限制 prompt 的推理能力。



- **GoT：**Graph-of-Thoughts，思维图。在 GoT 中，一个 LLM 思维会被建模成一个顶点，顶点之间的依赖关系则建模为边。使用 GoT，通过构建有多于一条输入边的顶点，可以将任意思维聚合起来。整体而言，GoT 使用的图抽象方法可无缝地将 CoT 和 ToT 泛化到更复杂的思维模式，而且这个过程无需更新模型。



GoT 由一组交互式模块构成，见下图（蓝色部分）。这些模块是 Prompter（准备用于 LLM 的消息）、Parser（解析器，提取 LLM 答复中的信息）、评分模块（验证 LLM 答复并评分）、Controller（控制器，协调整个推理过程，并决定如何推进推理）。Controller 中包含另外两个重要组件：操作图（GoO）和图推理状态（GRS）。GoO 是一个静态结构，其指定了对给定任务的图分解，即它规定了应用于 LLM 思维的变换及其顺序和依赖关系。GRS 是一个动态结构，其维持着正在进行的 LLM 推理过程的状态（其思维及其状态的历史）

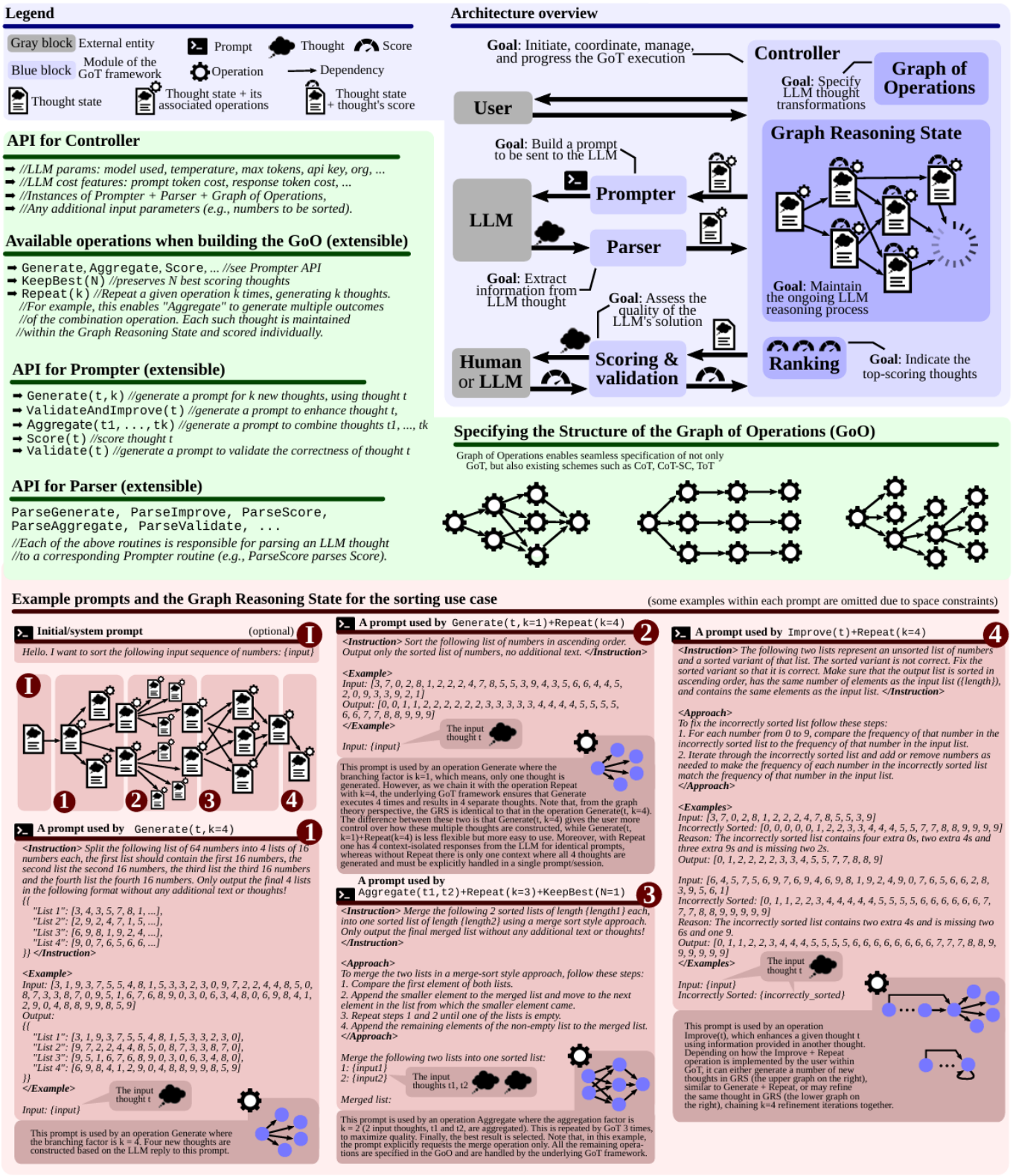
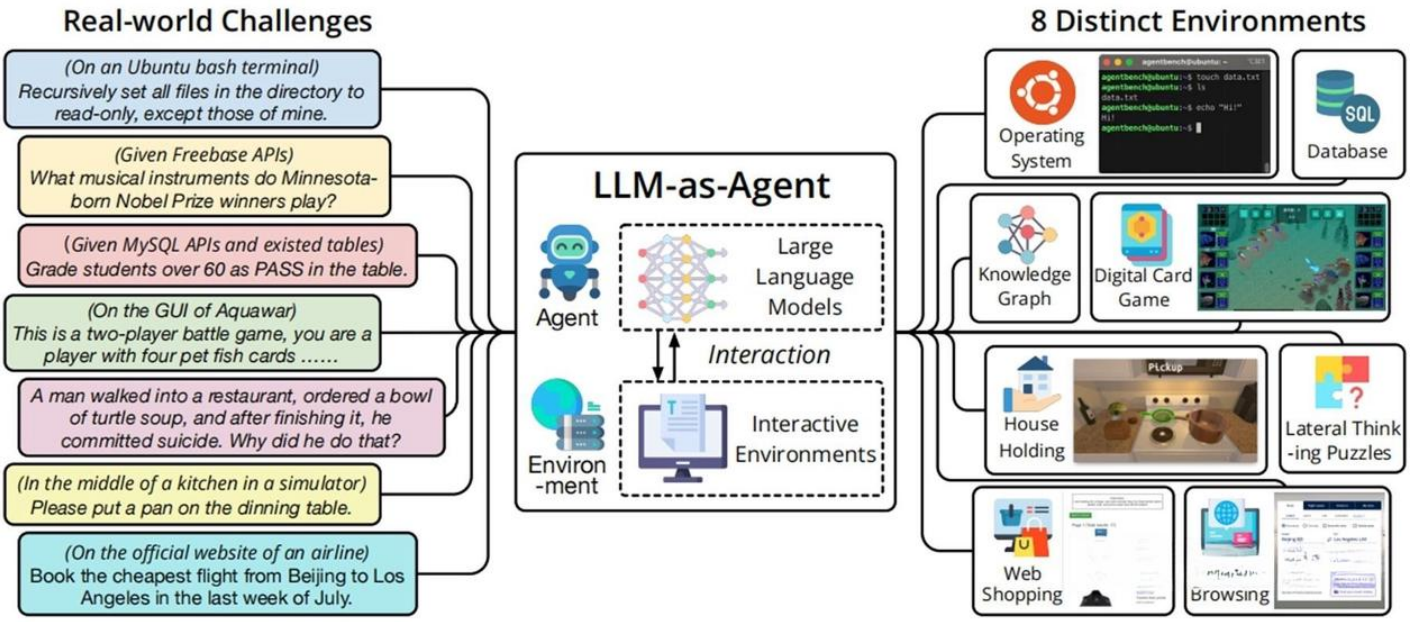


Figure 3: The system architecture of GoT, and the APIs of respective modules. The user can straightforwardly extend the design towards new prompting schemes, experiment with novel thought transformations, and plug in different LLMs. The blue part of the figure contains the architecture overview, the green part lists the API, and the red part contains example prompts together with a GRS and operations involved.

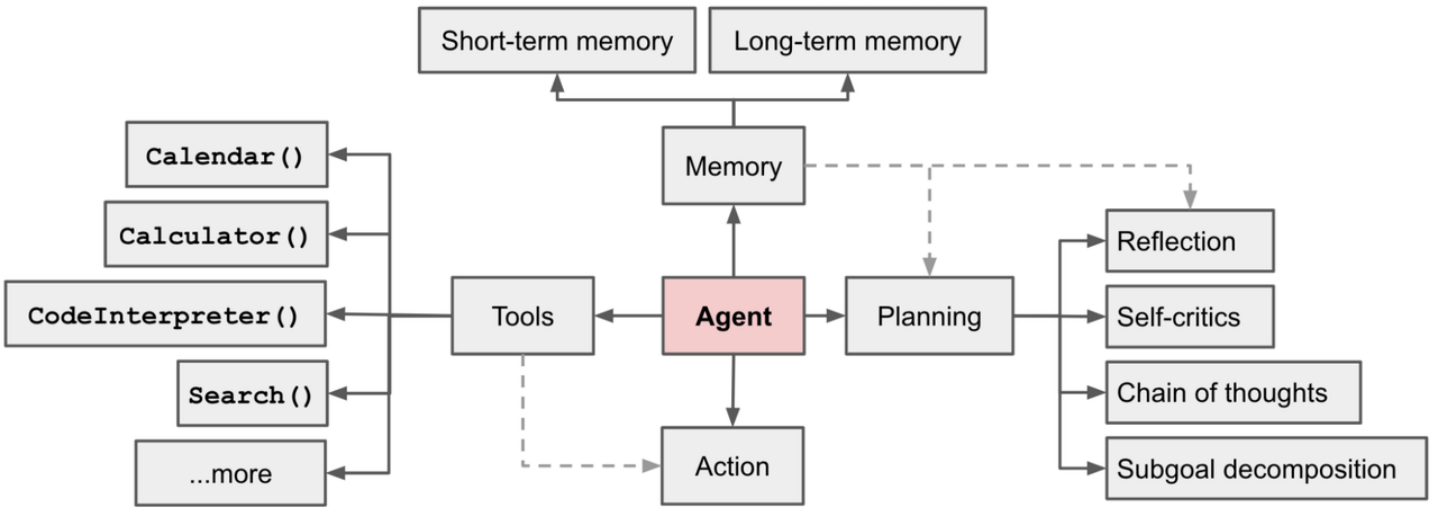
5.2 LLM-based Agent 基于大模型的智能体

LLM Agent是一种超越简单文本生成的人工智能系统。它使用LLM作为其核心计算引擎，使其能够进行对话、执行任务、推理并展现一定程度的自主性。简而言之，代理是一个具有复杂推理能力、记忆和执行任务手段的系统



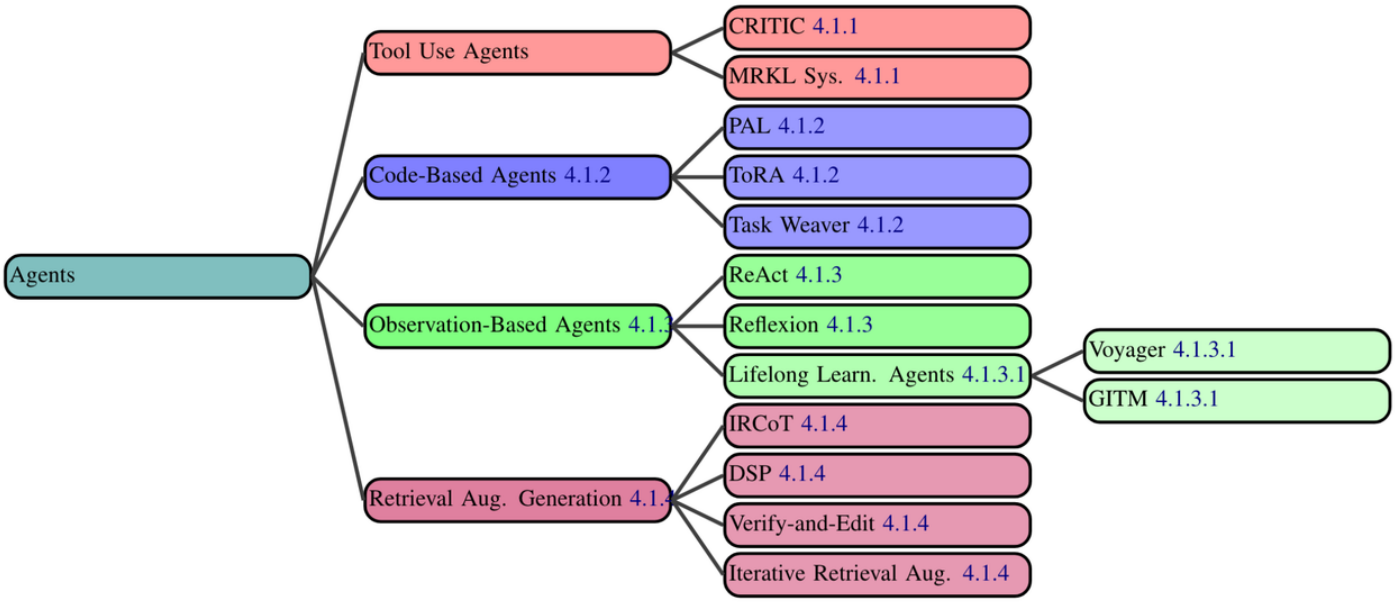
LLM Agent的定义：通过与GenAI本身之外的系统交互的操作来服务于用户的目标的GenAI系统

Agent LLMs 可能涉及单个外部系统（计算器），或者需要解决routing问题决定要去用哪个外部系统（经常包含memory、planning）



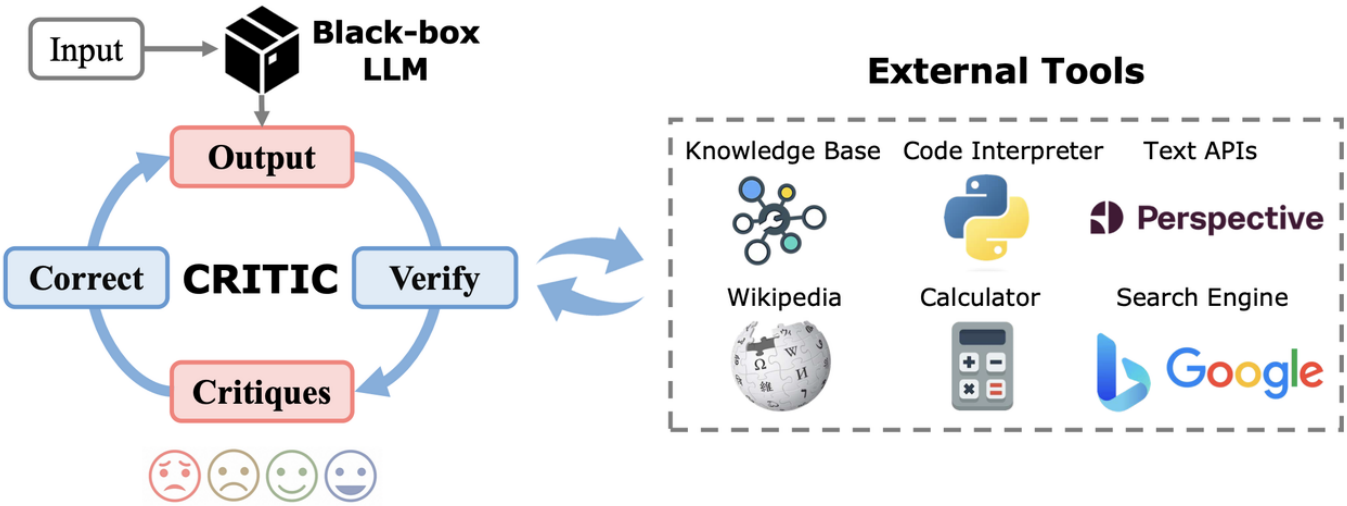
- **规划（planning）**
  - **子目标分解：**agent将大任务拆分为更小的可管理的子目标，使得可以有效处理复杂任务
  - **反思与完善：**agent对历史动作可以自我批评和自我反思，从错误中学习并在后续步骤里完善，从而改善最终结果的质量
- **记忆（Memory）**
  - **短期记忆：上下文学习**即是利用模型的短期记忆学习
  - **长期记忆：**为agent提供保留和召回长期信息的能力，通常利用**外部向量存储和检索实现**
- **工具使用（tool use）**
  - 对模型权重丢失的信息，**agent学习调用外部API获取额外信息，包括当前信息、代码执行能力、专有信息源的访问**等等

Agent分类：



5.2.1 Tool Use Agent

- **MRKL system：Modular Reasoning, Knowledge, and Language** 模块化推理、知识和语言。一个LLM作为router提供对多种工具的访问，然后router可以多次调用工具获取诸如时间、天气等信息，然后结合这些信息给出回复。下面的相关工作都是类似的技术，大部分还包含了微调：**Toolformer、Gorilla、Act-1、Hugging-gpt、ToolkenGPT**，
- **CRITIC：Self-Correcting with Tool-Interactive Critiquing** 利用工具交互评价来自我修正。首先根据prompt生成回答，然后同样的LLM对答案中可能出现的错位进行criticize，最后使用相应的外部工具，如联网搜索、代码解释器对部分回答进行验证或修正。**CRITIC流程图：**



5.2.2 Code Generation Agent

代码编写和执行能力也是一项重要的Agent能力，也可以被划分为 Tool Use Agents（Tool就是代码解释器等）

- **Program-aided LM：**将问题直接转换为code，然后利用Python解释器生成答案

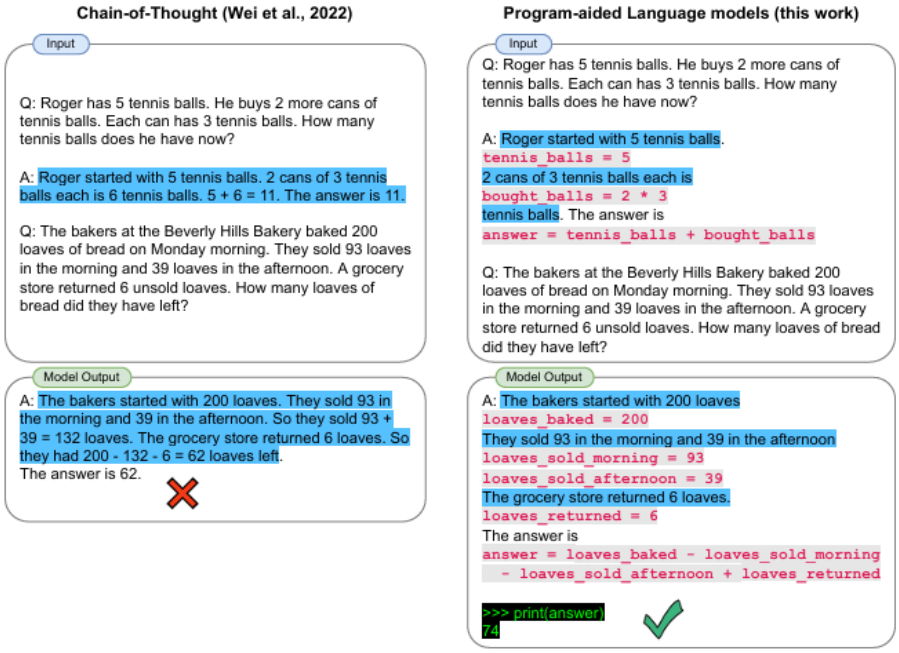
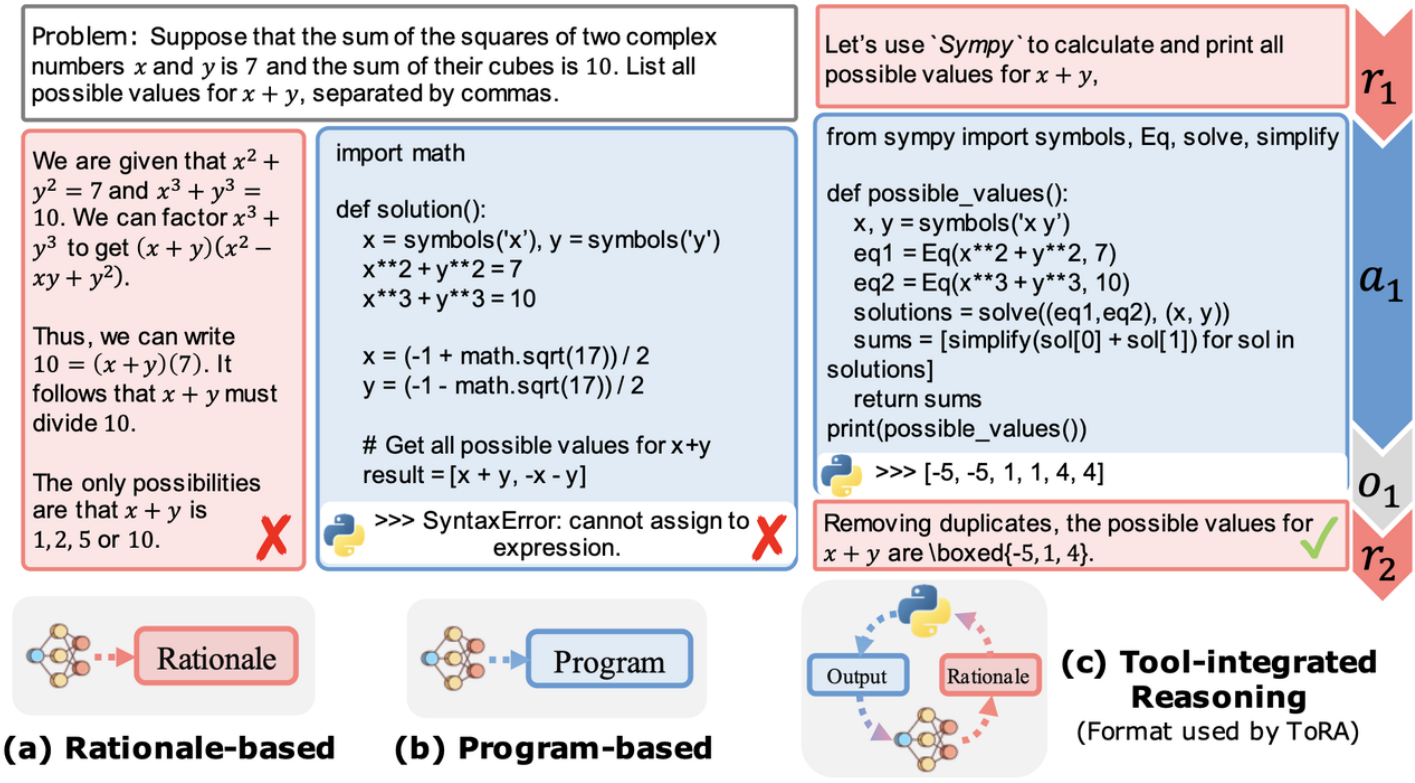
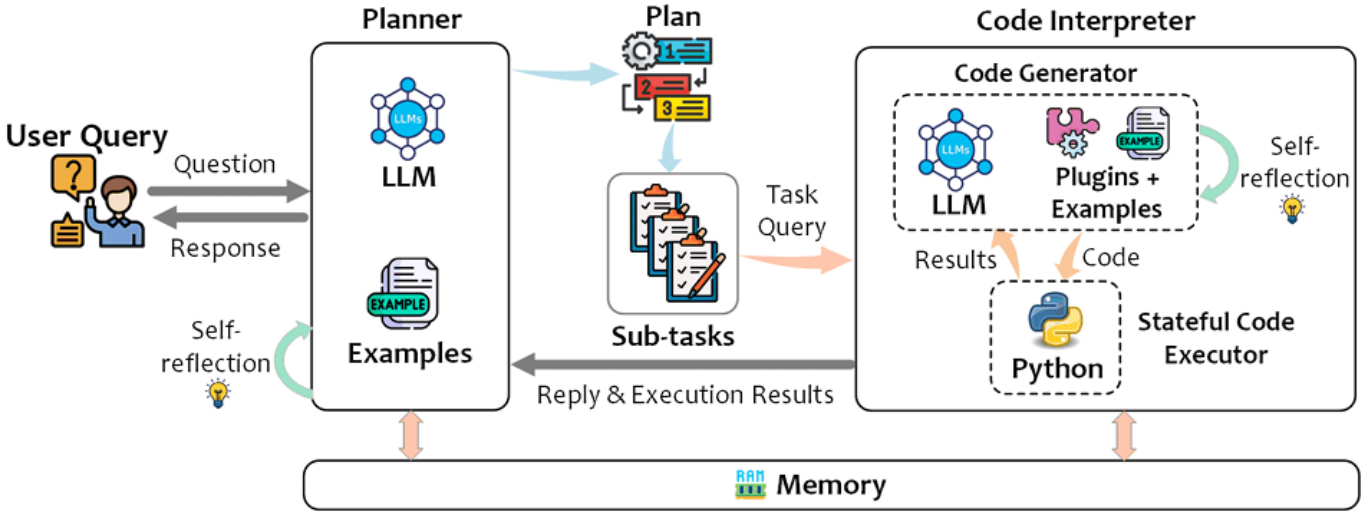


Figure 1: A diagram illustrating PAL: Given a mathematical reasoning question, Chain-of-thought (left) generates intermediate reasoning steps of free-form text. In contrast, Program-aided Language models (PAL, right) generate intermediate steps *and* Python code. This shifts the role of *running* the reasoning steps from the language model to the Python interpreter. The final answer is obtained by running the generated reasoning chain. Chain-of-thought reasoning is highlighted in blue; PAL steps are `highlighted in gray and pink`; the Python interpreter run is highlighted in black and green.

- **Tool-Integrated Reasoning Agent**：类似PAL，但是code和推理多步交叉直到解决问题，PAL是单个code-gen step。下图是CoT、PAL和本文的对比

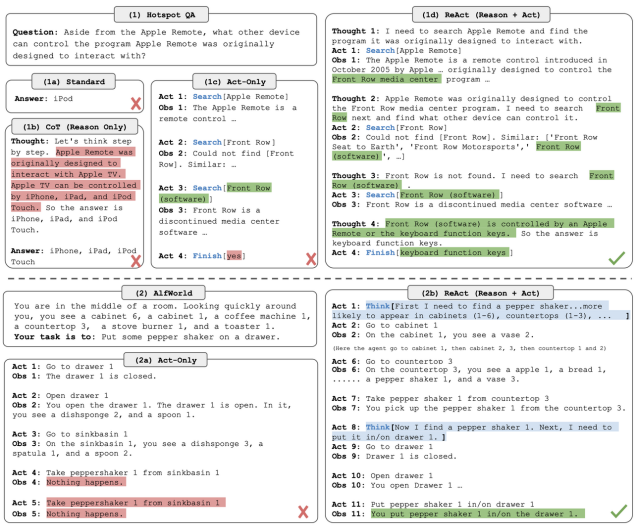


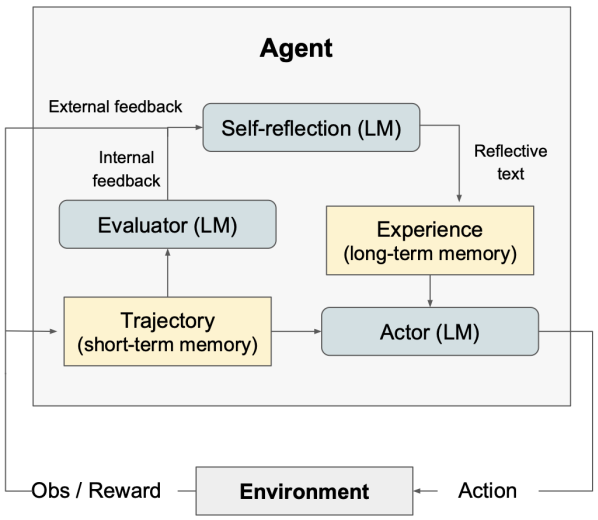
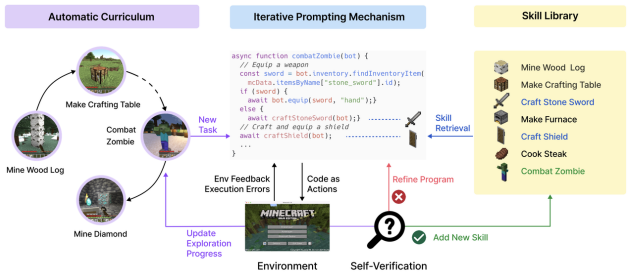
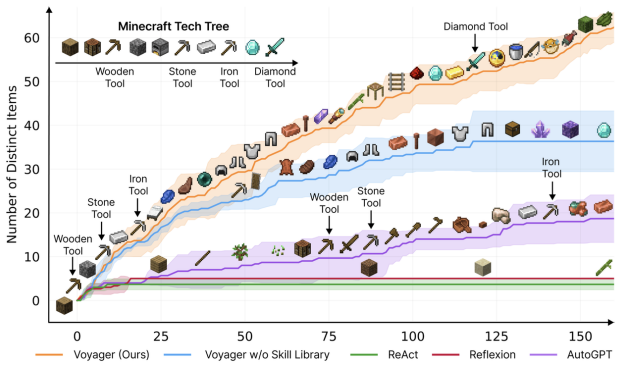
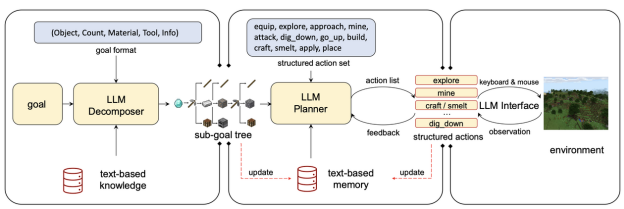
- **TaskWeaver**：也是类似PAL，但是可以利用用户定义的插件

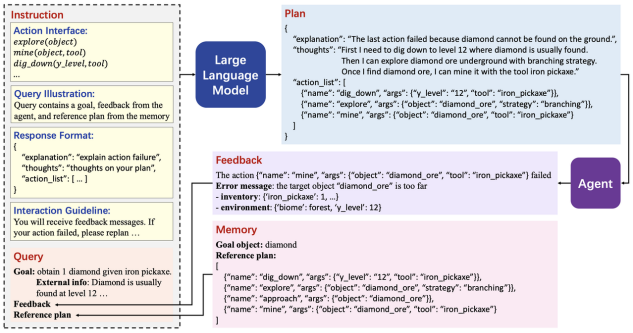


5.2.3 Observation-based Agent

一些Agents被**设计为通过与toy env(gym之类的环境)交互来解决问题**，这些基于Obs的Agent会将接收到的观测插入到Prompt中

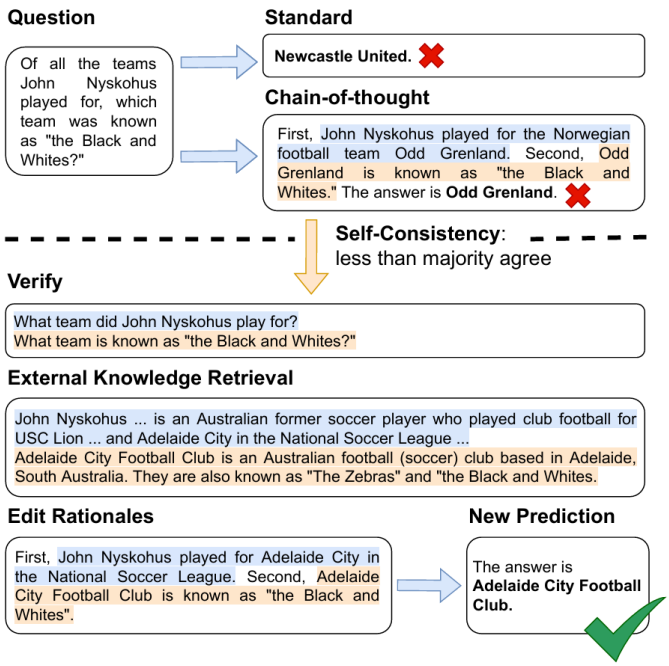
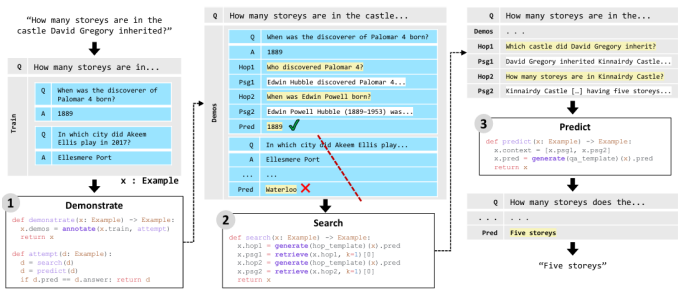
技术	背景	简介	具体方法
Reasoning and Acting	传统CoT中存在的幻觉和错误传播问题; Act-only的方法不能充分利用LLM reasoning的高层目标规划能力	结合LLM推理和生成动作的能力，交替生成推理轨迹和特定任务的动作，重复 **产生一个thought -> 采取一个动作 -> 接收一个观测** 的流程，然后所有的信息加入prompt中，相当于保存了过去的经历的记忆。支持外部工具访问。ReAct仅需学习一到六个context examples即可泛化到新任务实例	**流程及与CoT和Act-only方法的对比** 
Reflexion	LLM快速的从trial-and-error 中学习仍然很有挑战性，因为传统的强化学习方法需要大	由三个不同的模型组成：**Actor、Evaluator和Self-Reflection**。相比ReAct，多了Evaluator和Reflection。Actor来生成文本和动作，并在环境中接收观察结果。Evaluator评估Actor产生的轨迹的质量，并计算一个奖励分数以反映其性能。Self-Reflection模型则对反馈内容	**流程图**

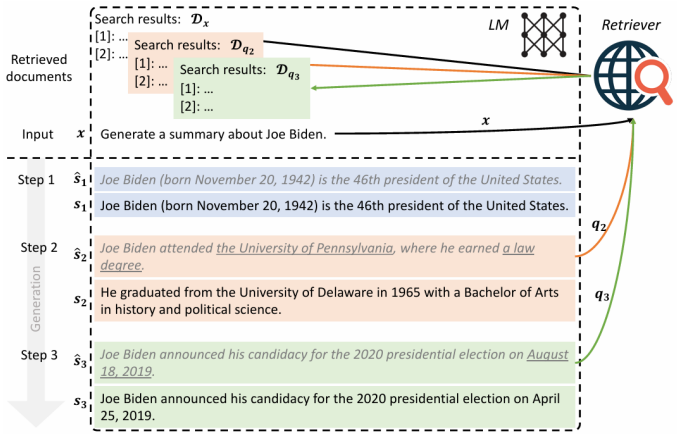
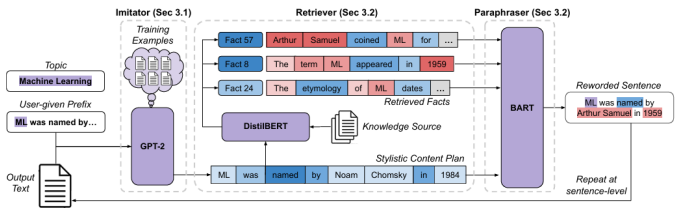
	量的训练样本和昂贵的模型微调	进行反思（做了什么，什么出错了），然后加入prompt作为memory，在任务中不断迭代优化，从而提高决策能力	 内部反馈是Evaluator给出的反馈，外部反馈是环境给出的标量奖励之类的反馈
Lifelong Learning Agents	可以在需要终身学习的真实世界任务中探索的Agents（MC等任务环境）	**Voyager：**主要包含三个部分，自动课程任务生成来探索开放世界，迭代式的生成code与环境交互执行动作，Self-Verification new skill加入skill library用于技能召回。实验中对比了ReAct、Reflexion和AutoGPT，效果碾压	**流程图**  **实验结果** 
		Ghost in the Minecraft (GITM)：利用LLM将一开始的goal分解为subgoals，然后迭代的进行plan和执行structured text（action）。同时GITM还用了外部的知识库辅助目标分解以及存储经验	**流程图**  **LLM Planner 示例**

5.2.4 RAG Agent

从外部来源（本地知识库等）检索信息并插入prompt的范式，可以提高知识密集型任务的性能。**RAG其实也是一种Agent，一般都是调用外部数据库/向量库作为工具**

技术	背景	简介	具体方法
Verify-and-Edit	CoT 通过生成可解释的推理链来提高复杂推理任务的信任和模型性能，但在知识密集型任务中仍然存在事实性问题	生成多条CoT，选择一些进行编辑，编辑是通过对外部信息检索然后允许LLM对这些进行增强	**流程图** 
Demonstrate-Search-Predict	现有工作简单将 RM 检索插入到 LM prompt 中在 multi-hop问题中效果不是很好	利用few-shot将一个问题分解为子问题，然后将对应的子问题的回答结合为最终的answer	**示例流程** 

Iterative Retrieval Augmentation	之前的一些检索增强LM存在两个方面的限制：一次性生成文本在long-form生成任务中容易产生幻觉采用概括性的topic作为query导致查询不到相关细节内容	**FLARE**：迭代的检索生成，先生成一个临时句子，然后检查句子中是否包含低可信度的token，如果有就进行检索然后重新生成句子，重复这个步骤直到完成生成	示例检索query生成：隐式query通过mask掉临时句子中的probs低于阈值的token生成；显式query通过要求LM生成 答案为临时句子中probs低于阈值的span 的问题，这些问题作为query去进行检索，然后将检索文档加入prompt重新生成句子 **流程图** 
		IRP：说明性文本生成任务。包含三个组件，Imitator、Retriever和Paraphraser。Imitator首先生成一个stylistic content plan，概述要包含在下一句中的事实，然后Retriever根据plan在语料库中检索相关facts，最后Paraphraser以plan的风格重新表述这些facts，加到输出后面作为下一个的prefix。sentence-by-sentence生成	**流程**：Imitator是用训练集中说明性文本的expert content plan进行训练的；Retriever部分为了减轻plan中的幻觉事实entity损害模型的检索能力，文章用DistilBERT在训练集上进行了分类任务微调，即预测这个文本中的句子的index（第几个句子），因为来自不同文本的同样index的句子的事实相关entity肯定是不同的，所以这样会使得模型对于事实entity的token的attribution scores更低 训练集GT示例 **流程图** 

5.3 RAG 检索增强生成

Retrieval Augmentation Generation

5.3.1 RAG简介

RAG（Retrieval Augmented Generation）为生成式模型提供了与外部世界互动的一个很有前景的解决方案。主要作用**类似于搜索引擎，找到用户提问最相关的知识或者是相关的对话历史，并结合原始提问（查询），创造信息丰富的prompt，指导模型生成准确输出**。其本质上应用了In-Context Learning的原理

RAG（检索增强生成）= 检索技术 + LLM（大语言模型）提示

LLM本身的局限性：

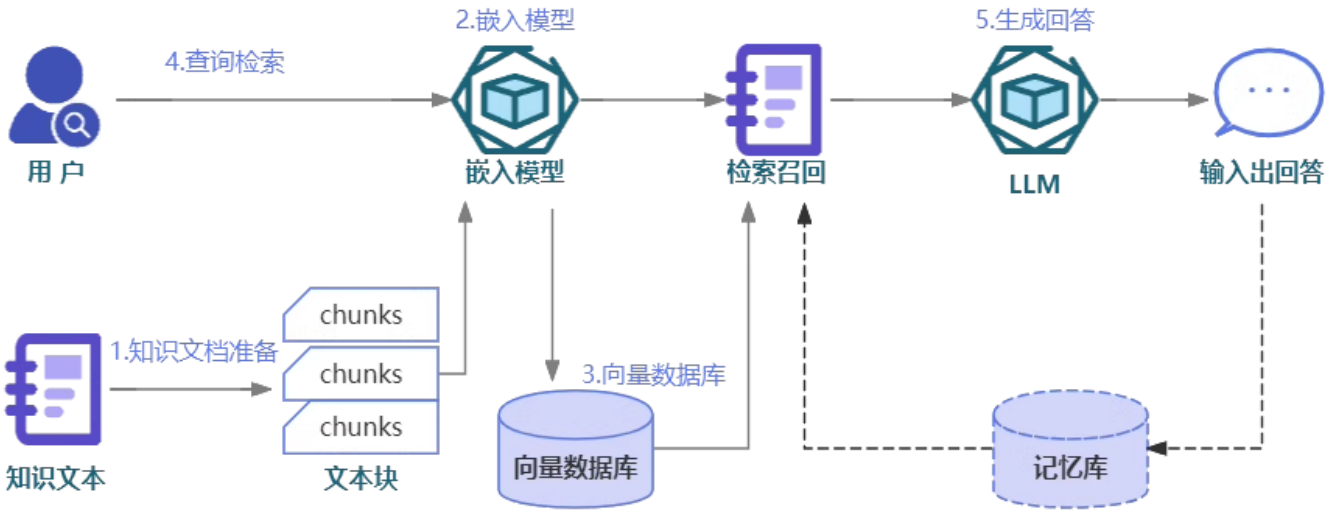
- **知识的局限性：**模型自身的知识完全源于它的训练数据，而现有的主流大模型的训练集基本都是构建于网络公开的数据，对于一些实时性的、非公开的或离线的数据是无法获取到的，这部分知识也就无从具备
- **幻觉问题：**所有的AI模型的底层原理都是基于数学概率，其模型输出实质上是一系列数值运算，大模型也不例外，所以它有时候会一本正经地胡说八道，尤其是在大模型自身不具备某一方面的知识或不擅长的场景。而这种幻觉问题的区分是比较困难的，因为它要求使用者自身具备相应领域的知识
- **数据安全性：**对于企业来说数据安全至关重要，没有企业愿意承担数据泄露的风险，将自身的私域数据上传第三方平台进行训练。这也导致完全依赖通用大模型自身能力的应用方案不得不在数据安全和效果方面进行取舍

RAG的特点：

- **依赖LLM来强化信息检索和输出：**RAG需要结合LLM来进行信息的检索和生成，但如果单独使用RAG，它的能力会受到限制。也就是说，RAG需要依赖强大的语言模型支持，才能更有效地生成和提供信息
- **能与外部数据有效集成：**RAG能够很好地接入和利用外部数据库的数据资源。这一特性弥补了通用大模型在某些垂直或专业领域的知识不足或者数据时效问题，比如行业特定的术语和深度知识，能提供更精确的答案
- **数据隐私和安全保障：**通常RAG所连接的私有数据库不会参与到大模型的数据集中训练。因此，RAG既能提升模型的输出表现，又能有效地保护这些私有数据的隐私性和安全性，不会将敏感信息暴露给大模型的训练过程
- **表现效果因多方面因素而异：**RAG的效果受多个因素影响，比如所使用的语言模型的性能、输入数据的质量、算法以及检索系统的设计等。这意味着不同的RAG系统之间效果差异较大，不能一概而论

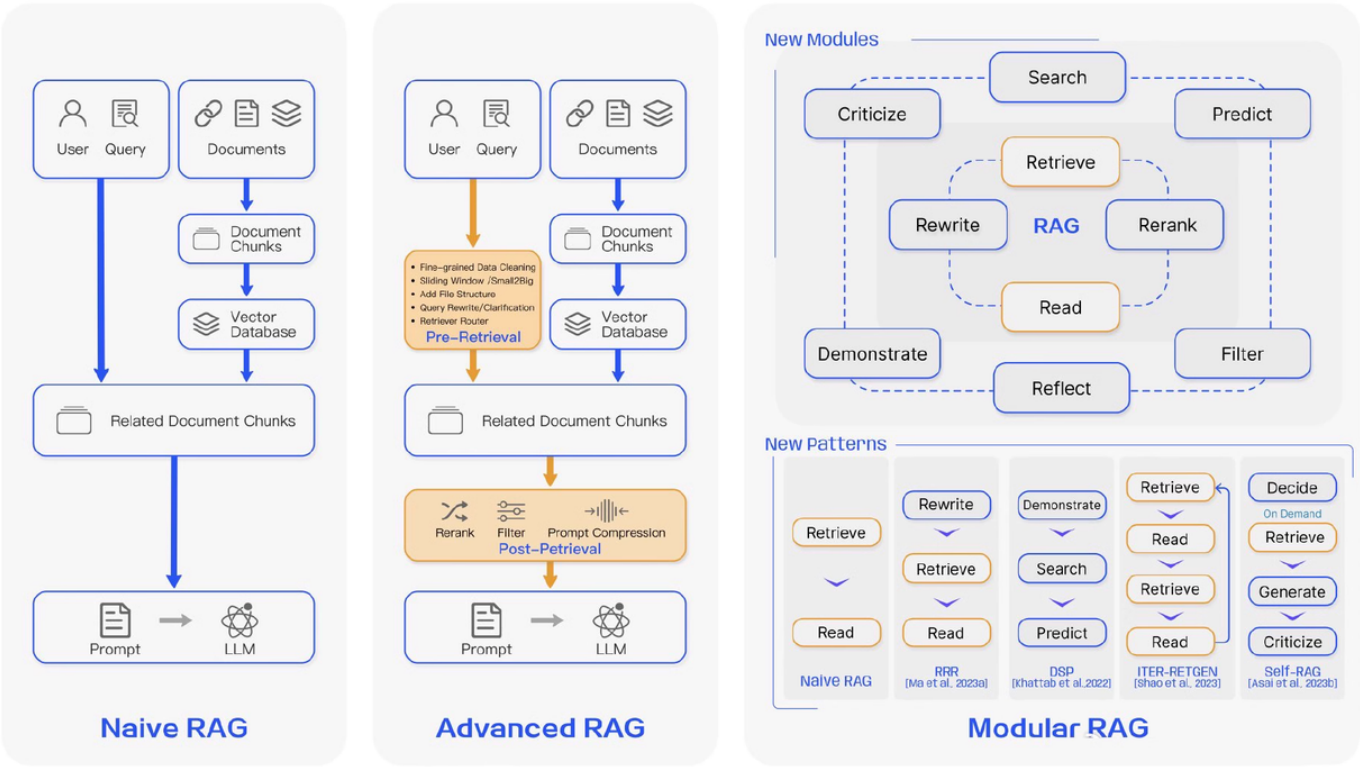
5.3.2 RAG流程与分类

RAG技术的整体思路：可以分为五个基本流程：**知识文档的准备、Embedding模型、向量数据库、查询检索和生成回答**



- **知识文档的准备：**在构建一个高效的RAG系统时，首要步骤是准备知识文档。现实场景中，我们面对的知识源可能包括多种格式，如**Word文档、TXT文件、CSV数据表、Excel表格，甚至是PDF文件、图片和视频**等。因此，第一步需要使用**专门的文档加载器（例如PDF提取器）或多模态模型（如OCR技术）**，将这些丰富的知识源转换为**大语言模型可理解的纯文本数据**。例如，处理PDF文件时，可以利用PDF提取器抽取文本内容；对于图片和视频，OCR技术能够识别并转换其中的文字信息。此外，鉴于文档可能存在过长的问题，我们还需执行一项关键步骤：**文档切片。我们需要将长篇文档分割成多个文本块，以便更高效地处理和检索信息**。这不仅有助于减轻模型的负担，还能提高信息检索的准确性
- **Embedding模型：核心任务是将文本转换为向量形式**，的日常语言中充满歧义和对表达词意无用的助词，而**向量表示则更加密集、精确，能够捕捉到句子的上下文关系和核心含义，**来识别语义上相似的句子。Word2Vec，BERT，GPT系列，BGE系列等等Embedding模型
- **向量数据库：**专门设计用于存储和检索向量数据的数据库系统。在RAG系统中，通过嵌入模型生成的所有向量都会被存储在这样的数据库中。这种数据库优化了处理和存储大规模向量数据的效率，使得在面对海量知识向量时，我们能够迅速检索出与用户查询最相关的信息
- **查询检索：**用户的问题会被输入到嵌入模型中进行向量化处理。然后，系统会在向量数据库中搜索与该问题向量语义上相似的知识文本或历史对话记录并返回
- **生成回答：**将用户提问和上一步中检索到的信息结合，构建出一个提示模版，输入到大语言模型中，静待模型输出答案即可

RAG分类：三个阶段Naive RAG，Advanced RAG、Modular RAG

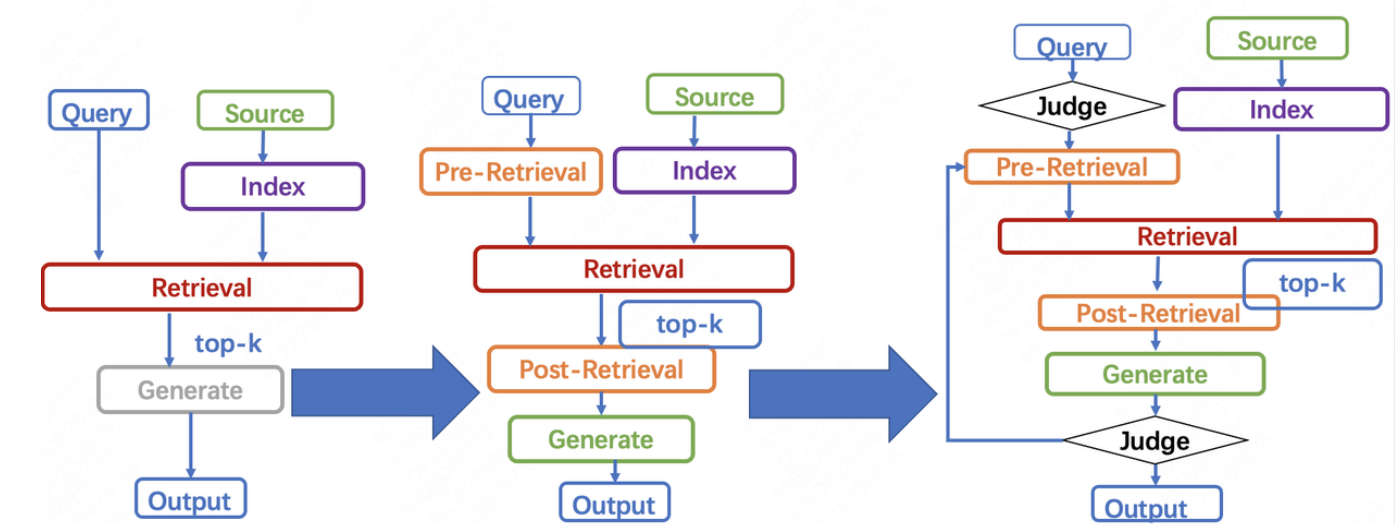


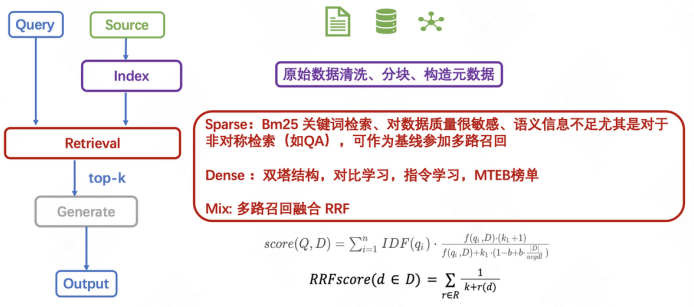
- **Naive RAG**：经典的RAG，主要涉及“检索-阅读”过程。主要包括包括三个基本步骤：
  - **索引**：将文档库分割成较短的 Chunk，并通过编码器构建向量索引
  - **检索**：根据问题和 chunks 的相似度检索相关文档片段
  - **生成**： 以检索到的上下文为条件，生成问题的回答
- **Advanced RAG**：Naive RAG 在检索质量、响应生成质量以及增强过程中存在多个挑战。Advanced RAG在**数据索引、检索前和检索后都进行了额外处理**。通过更精细的数据清洗、设计文档结构和添加元数据等方法提升文本的一致性、准确性和检索效率。在**检索前阶段则可以使用问题的重写、路由和扩充等方式对齐问题和文档块之间的语义差异**。在**检索后阶段则可以通过将检索出来的文档库进行重排序避免 “Lost in the Middle” 现象的发生，或是通过上下文筛选与压缩的方式缩短窗口长度**
- **Modular RAG**：在结构上更加自由的和灵活，引入了更多的具体功能模块，例如**查询搜索引擎、融合多个回答**。技术上将**检索与微调、强化学习等技术融合**。流程上也对 RAG 模块之间进行设计和编排，出现了多种的 RAG 模式

RAG框架种类	优点	缺点
初级RAG (Naive RAG)	1.采用传统的索引、检索和生成流程 2.直接基于用户输入进行查询 3.结合相关文档与问题形成新的提示，供大型语言模 型生成答案	1.检索质量低，可能导致“空中掉落”现象 2.响应生成质量存在幻觉和询矢量转换 不相关性问题 3.增强过程中的集成挑战如冗余和风格一致性
		1.高级优化增加了复杂性

进阶的 RAG（Advanced RAG）	1.优化数据索引，提高索引内容质量 2.实施预检索和后检索方法，提高检索和生成的质量 3.引入混合检索和索引结构优化，如图结构信息	2.对计算资源和处理能力有更高要求 3.需要更多的定制和调优以提高效率和相关性
模块化RAG (Modular RAG)	1.增加功能模块，提供多样性和灵活性 2.适应性强，可针对特定问题上下文替换或重组模块 3.采用串行或端到端训练方法，允许跨多个模块的定制	1.构建和维护模块化系统可能较为复杂 2. 需要仔细管理以确保模块间的协调和一致性

RAG发展路线：





RAG数据相关处理



RAG检索优化

5.3.3 RAG评估

- **针对检索环节的评估：**
  - **MRR平均倒数排名：Mean Reciprocal Rank，**查询或推荐请求的排名倒数

    MRR是一种常用的评估信息检索（Information Retrieval, IR）系统表现的指标，尤其用于衡量搜索引擎、推荐系统等根据查询返回的多个结果中的相关性。结果列表中，第一个结果匹配分

数为1，第二个匹配分数为0.5，第n个匹配分数为1/n，如果没有匹配的句子分数为0。最终的分数为所有得分之和，再平均

MRR的意义：

- MRR值越高，表示系统对用户查询的响应越好，因为第一个相关结果更可能出现在较高的排名位置
- 如果第一个相关结果排名在前几个位置，倒数排名接近1，会提高MRR值
- 如果第一个相关结果排得很靠后，倒数排名就会较小，MRR值较低

假设我们有3个查询：

- 第一个查询的第一个相关结果排在第2位（倒数排名 = 1/2）
- 第二个查询的第一个相关结果排在第5位（倒数排名 = 1/5）
- 第三个查询的第一个相关结果排在第1位（倒数排名 = 1/1）

那么，MRR就会是： $MRR = \frac{1}{3}\left(\frac{1}{2} + \frac{1}{5} + \frac{1}{1}\right) = \frac{1}{3}(0.5 + 0.2 + 1) = \frac{1}{3} \times 1.7 = 0.567$

MRR衡量的是**相关结果首次出现的位置（越靠前越好）**适用于多结果排序任务，如搜索引擎查询、推荐系统

- **Hits Rate 命中率：**前K项中包含正确信息的项的占比
- **NDCG归一化折损累计增益：**Normalized Discounted Cumulative Gain

DCG的思想：list中item的顺序很重要，不同位置的贡献不同，一般来说，排在前面的item影响更大，排在后面的item影响较小。（例如一个返回的网页，肯定是排在前面的item会有更多人点击）**DCG使排在前面的item增加其影响，排在后面的item减弱其影响**

$$DCG_k = \sum_{i=1}^{k} \frac{rel(i)}{\log_2(i+1)}$$

其中rel(i)是item(i)的相关性得分。IDCG是根据rel(i)降序排序后的最好状态的DCG，所以可以得到NDCG：

$$NDCG = \frac{DCG}{IDCG}$$

- **针对生成环节的评估：**
  - **非量化：**完整性、正确性、上下文相关性、答案忠实性和答案相关性
  - **量化：**Rouge、BLEU等文本相似性指标

5.3.4 RAG优化

1. **文档知识准备阶段**

- **数据清洗：**RAG依赖于准确且清洁的原始知识数据。一方面为了**保证数据的准确性，需要优化文档读取器和多模态模型**。特别是处理如CSV表格等文件时，单纯的文本转换可能会丢失表格原有的结构。因此，我们需引入额外的机制以在文本中恢复表格结构，比如使用分号或其他符号来区分数据。另一方面也需要**对知识文档做一些基本数据清洗其中可以包括**：
  - **基本文本清理**：规范文本格式，去除特殊字符和不相关信息。除重复文档或冗余信息
  - **实体解析**：消除实体和术语的歧义以实现一致的引用。例如，将“LLM”、“大语言模型”和“大模型”标准化为通用术语
  - **文档划分**：合理地划分不同主题的文档，不同主题是集中在一处还是分散在多处？如果作为人类都不能轻松地判断出需要查阅哪个文档才能来回答常见的提问，那么检索系统也无法做到
  - **数据增强**：使用同义词、释义甚至其他语言的翻译来增加语料库的多样性
  - **用户反馈循环**：基于现实世界用户的反馈不断更新数据库，标记它们的真实性
  - **时间敏感数据**：对于经常更新的主题，实施一种机制来使过时的文档失效或更新

2. **分块处理**

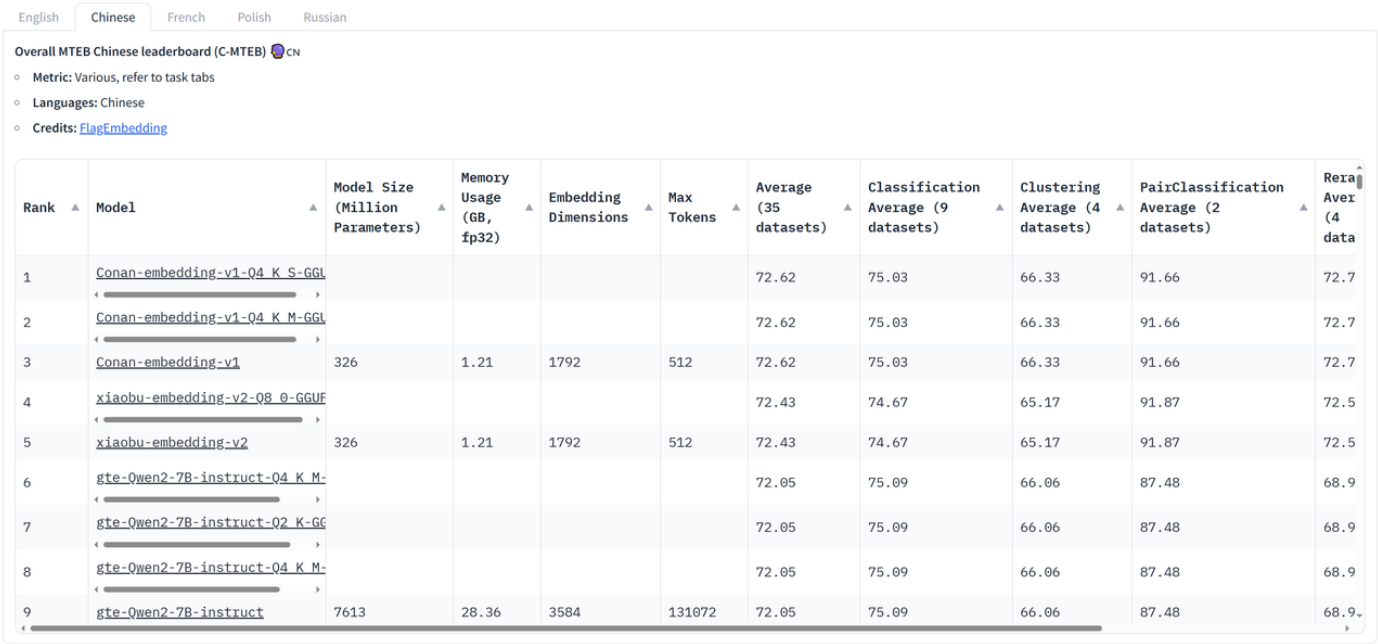
   在RAG系统中，**文档需要被分割成多个文本块后再进行向量嵌入**。不考虑大模型输入长度限制和成本问题的情况下，其目的是在**保持语义上的连贯性的同时，尽可能减少嵌入内容中的噪声，从而更有效地找到与用户查询最相关的文档部分**。如果**分块太大，可能包含太多不相关的信息，从而降低了检索的准确性。相反，分块太小可能会丢失必要的上下文信息，导致生成的回应缺乏连贯性或深度**。在RAG系统中实施合适的文档分块策略，旨在找到这种平衡，确保信息的完整性和相关性。一般来说，理想的文本块应当在没有周围上下文的情况下对人类来说仍然有意义，这样对语言模型来说也是有意义的

   - **固定大小的分块：**最简单和直接的方法，直接设定块中的字数，并选择块之间是否重复内容。通常，我们会保持块之间的一些重叠，以确保语义上下文不会在块之间丢失。与其他形式的分块相比，固定大小分块简单易用且不需要很多计算资源
   - **内容分块：**根据文档的具体内容进行分块，例如根据标点符号（如句号）分割。或者直接使用更高级的NLTK或者spaCy库提供的句子分割功能
   - **递归分块：**在大多数情况下推荐的方法，通过重复地应用分块规则来递归地分解文本。 例如，在langchain中会先通过段落换行符（\n\n）进行分割，然后，检查这些块的大小。如果大小不超过一定阈值，则该块被保留。对于大小超过标准的块，使用单换行符（\n）再次分割。以此类推，不断根据块大小更新更小的分块规则（如空格，句号）。这种方法可以灵活地调整块的大小。例如，对于文本中的密集信息部分，可能需要更细的分割来捕捉细节；而对于信息较少的部分，则可以使用更大的块。它的挑战在于，需要制定精细的规则来决定何时和如何分割文本

- **从小到大分块：**既然小的分块和大的分块各有各的优势，一种更为直接的解决方案是将同一文档从大到小所有尺寸的分块，然后将不同大小的分块全部存进向量数据库，并保存每个分块的上下级关系，进行递归搜索。但可想而知，因为我们需要存储大量重复的内容，这种方案的缺点就是需要更大的存储空间
- **特殊结构分块：**针对特定结构化内容的专门分块器。这些分块器专门设计来处理这些类型的文档，以确保正确地保留和理解其结构。langchain提供的特殊分块器包括：Markdown文件、LaTeX文件，以及各种主流代码语言分块器
- **分块大小的选择：**首先不同的嵌入模型有其最佳输入大小，例如OpenAI的text - embedding - ada - 002模型在256或512大小的块上效果更好；其次，文档的类型和用户查询的长度及复杂性也是决定分块大小的重要因素。处理长篇文章或书籍时，较大的分块有助于保留更多的上下文和主题连贯性；而对于社交媒体帖子，较小的分块可能更适合捕捉每个帖子的精确语义。如果用户的查询通常是简短和具体的，较小的分块可能更为合适；相反，如果查询较为复杂，可能需要更大的分块

3. **Embedding模型阶段**

嵌入模型能帮助我们把文本转换成向量，不同的嵌入模型带来的效果也不尽相同，推荐参考Hugging Face的嵌入模型排行榜MTEB https://huggingface.co/spaces/mteb/leaderboard



English Chinese French Polish Russian

Overall MTEB Chinese leaderboard (C-MTEB) CN

- Metric: Various, refer to task tabs
- Languages: Chinese
- Credits: FlagEmbedding

Rank	Model	Model Size (Million Parameters)	Memory Usage (GB, fp32)	Embedding Dimensions	Max Tokens	Average (35 datasets)	Classification Average (9 datasets)	Clustering Average (4 datasets)	PairClassification Average (2 datasets)	Rera Aver (4 data
1	Conan-embedding-v1-Q4_K_S-GGU					72.62	75.03	66.33	91.66	72.7
2	Conan-embedding-v1-Q4_K_M-GGU					72.62	75.03	66.33	91.66	72.7
3	Conan-embedding-v1	326	1.21	1792	512	72.62	75.03	66.33	91.66	72.7
4	xiaobu-embedding-v2-Q8_0-GGUF					72.43	74.67	65.17	91.87	72.5
5	xiaobu-embedding-v2	326	1.21	1792	512	72.43	74.67	65.17	91.87	72.5
6	gte-Qwen2-7B-instruct-Q4_K_M-					72.05	75.09	66.06	87.48	68.9
7	gte-Qwen2-7B-instruct-Q2_K-GG					72.05	75.09	66.06	87.48	68.9
8	gte-Qwen2-7B-instruct-Q4_K_M-					72.05	75.09	66.06	87.48	68.9
9	gte-Qwen2-7B-instruct	7613	28.36	3584	131072	72.05	75.09	66.06	87.48	68.9

4. **向量数据库阶段**

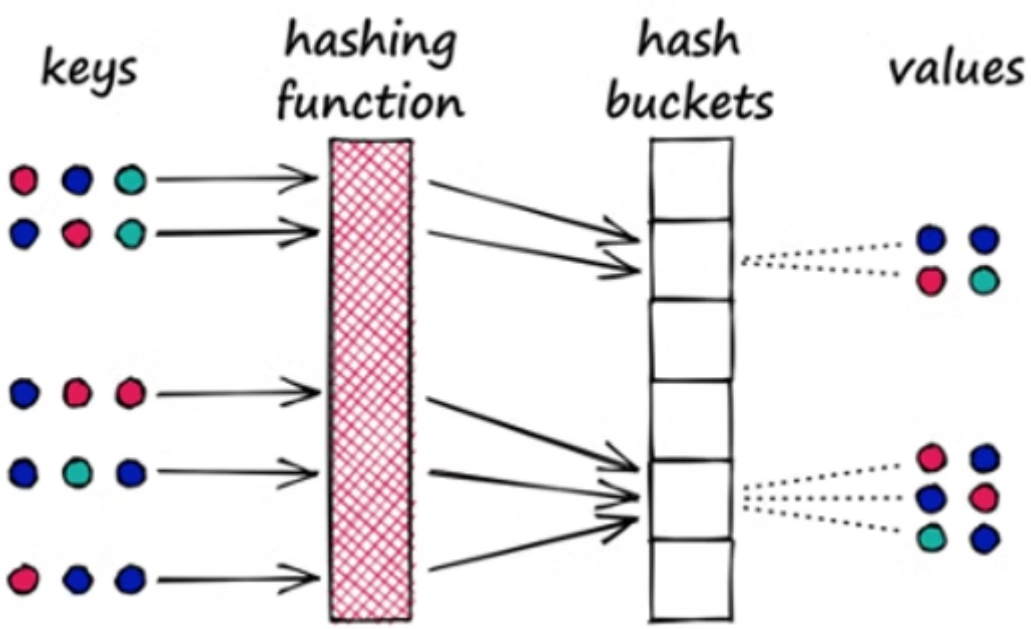
- **元数据：**在向量数据库中存储向量数据时，某些数据库支持将向量与元数据（即非向量化的数据）一同存储。为向量添加元数据标注是一种提高检索效率的有效策略，它在处理搜索结果时发挥着重要作用。

- 例如，日期就是一种常见的元数据标签。它能够帮助我们根据时间顺序进行筛选。设想一下，如果我们正在开发一款允许用户查询他们电子邮件历史记录的应用程序。在这种情况下，日期最近的电子邮件可能与用户的查询更相关。然而，从嵌入的角度来看，我们无法直接判断这些邮件与用户查询的相似度。通过将每封电子邮件的日期作为元数据附加到其嵌入中，我们可以在检索过程中优先考虑最近日期的邮件，从而提高搜索结果的相关性。
- 此外，还可以添加诸如章节或小节的引用，文本的关键信息、小节标题或关键词等作为元数据。这些元数据不仅有助于改进知识检索的准确性，还能为最终用户提供更加丰富和精确的搜索体验

5. **查询索引阶段（检索召回、重排）**
   - **多级索引：元数据无法充分区分不同上下文类型的情况下，可以考虑进一步尝试多重索引技术。**
     - 多重索引技术的核心思想是将庞大的数据和信息需求按类别划分，并在不同层级中组织，以实现更有效的管理和检索。这意味着系统不仅依赖于单一索引，而是建立了多个针对不同数据类型和查询需求的索引
     - 例如，可能有一个索引专门处理摘要类问题，另一个专门应对直接寻求具体答案的问题，还有一个专门针对需要考虑时间因素的问题。这种多重索引策略使RAG系统能够根据查询的性质和上下文，选择最合适的索引进行数据检索，从而提升检索质量和响应速度。但为了引入多重索引技术，还需配套加入多级路由机制
     - 多级路由机制确保每个查询被高效引导至最合适的索引。查询根据其特点（如复杂性、所需信息类型等）被路由至一个或多个特定索引。这不仅提升了处理效率，还优化了资源分配和使用，确保了对各类查询的精确匹配
     - 例如，对于查询“最新上映的科幻电影推荐”，RAG系统可能首先将其路由至专门处理当前热点话题的索引，然后利用专注于娱乐和影视内容的索引来生成相关推荐
   - **索引/查询算法：**
     - 可以利用索引筛选数据，从筛选后的数据中检索出相关的文本向量。由于向量数据量庞大且复杂，寻找绝对的最优解变得计算成本极高，有时甚至是不可行的。加之，大模型本质上并不是完全确定性的系统，这些模型在搜索时追求的是语义上的相似性——一种合理的匹配即可。从应用的角度来看，这种方法是合理的
     - 例如，在推荐系统中，用户不太可能察觉到或关心是否每个推荐的项目都是绝对的最佳匹配；他们更关心的是推荐是否总体上与他们的兴趣相符。因此查找与查询向量完全相同的项通常不是目标，而是要找到“足够接近”或“相似”的项，这便是最近邻搜索（Approximate Nearest Neighbor Search，ANNS）。这样做不仅能满足需求，还为检索优化提供了巨大的优化潜力。下面我们会介绍一些常见的向量搜索算法以便大家在具体使用场景中进行取舍

     i. **聚类：K-means等**

ii. **位置敏感哈希：**沿着缩小搜索范围的思路，位置敏感哈希算法是另外一种实现的策略。在传统的哈希算法中，通常希望每个输入对应一个唯一的输出值，并努力减少输出值的重复。然而，在位置敏感哈希算法中，我们的目标恰恰相反，我们需要增加输出值碰撞的概率。这种碰撞正是分组的关键，哈希值相同的向量将被分配到同一个组中，也就是同一个“桶”里。此外，这种哈希函数还需满足另一个条件：空间上距离较近的向量更有可能被分入同一个桶。这样在进行搜索时，只需获取目标向量的哈希值，找到相应的桶，并在该桶内进行搜索即可



iii. **量化乘积、分层导航小世界**等

- **查询转换：**在RAG系统中，用户的查询问题被转化为向量，然后在向量数据库中进行匹配。不难想象，查询的措辞会直接影响搜索结果。如果搜索结果不理想，可以尝试以下几种方法对问题进行重写，以提升召回效果
  - **结合历史对话的重新表述：**在向量空间中，对人类来说看似相同的两个问题其向量大小并不一定很相似。我们可以直接利用LLM 重新表述问题来进行尝试。此外，在进行多轮对话时，用户的提问中的某个词可能会指代上文中的部分信息，因此可以将历史信息和用户提问一并交给LLM重新表述
  - **假设文档嵌入（HyDE）：**HyDE的核心思想是接收用户提问后，先让LLM在没有外部知识的情况下生成一个假设性的回复。然后，将这个假设性回复和原始查询一起用于向量检索。假设回复可能包含虚假信息，但蕴含着LLM认为相关的信息和文档模式，有助于在知识库中寻找类似的文档
  - **退后提示（Step Back Prompting）：**如果果原始查询太复杂或返回的信息太广泛，我们可以选择生成一个抽象层次更高的“退后”问题，与原始问题一起用于检索，以增加返回结果的数量。例如，原问题是“桌子君在特定时期去了哪所学校”，而退后问题可能是关于他的“教育历史”。这种更高层次的问题可能更容易找到答案
  - **多查询检索/多路召回（Multi Query Retrieval）**使用LLM生成多个搜索查询，特别适用于一个问题可能需要依赖多个子问题的情况

- **检索参数：**具体的检索过程中，我们可以根据向量数据库的特定设置来优化一些检索参数，以下是一些常见的可设定参数
  - **稀疏和稠密搜索权重：**稠密搜索即通过向量进行搜索。然而，在某些场景下可能存在限制，此时可以尝试使用原始字符串进行关键字匹配的稀疏搜索。一种有效的稀疏搜索算法是最佳匹配25（BM25），它基于统计输入短语中的单词频率，频繁出现的单词得分较低，**而稀有的词被视为关键词，得分会较高**。我们可以结合稀疏和稠密搜索得出最终结果。向量数据库通常允许设定两者对最终结果评分的权重比例，如0.6表示40%的得分来自稀疏搜索，60%来自稠密搜索
  - **结果数量（topK）：**检索结果的数量是另一个关键因素。足够的检索结果可以确保系统覆盖到用户查询的各个方面。在回答多方面或复杂问题时，更多的结果提供了丰富的语境，有助于RAG系统更好地理解问题的上下文和隐含细节。但需注意，结果数量过多可能导致信息过载，降低回答准确性并增加系统的时间和资源成
  - **相似度度量方法：**计算两个向量相似度的方法也是一个可选参数。这包括使用欧式距离和Jaccard距离计算两个向量的差异，以及利用余弦相似度衡量夹角的相似性。通常，余弦相似度更受青睐，因为它不受向量长度的影响，只反映方向上的相似度。这使得模型能够忽略文本长度差异，专注于内容的语义相似性。需要注意的是，并非所有嵌入模型都支持所有度量方法，具体可参考所用嵌入模型的说明
- **高级检索策略：**
  - **上下文压缩：**当文档文块过大时，可能包含太多不相关的信息，传递这样的整个文档可能导致更昂贵的LLM调用和更差的响应。上下文压缩的思想就是通过LLM的帮助根据上下文对单个文档内容进行压缩，或者对返回结果进行一定程度的过滤仅返回相关信息
  - **句子窗口搜索：**相反，文档文块太小会导致上下文的缺失。其中一种解决方案就是窗口搜索，该方法的核心思想是当提问匹配好分块后，将该分块周围的块作为上下文一并交给LLM进行输出，来增加LLM对文档上下文的理解
  - **父文档搜索：**父文档搜索也是一种很相似的解决方案，父文档搜索先将文档分为尺寸更大的主文档，再把主文档分割为更短的子文档两个层级，用户问题会与子文档匹配，然后将该子文档所属的主文档和用户提问发送给LLM
  - **自动合并：**自动合并是在父文档搜索上更进一步的复杂解决方案。同样地，我们先对文档进行结构切割，比如将文档按三层树状结构进行切割，顶层节点的块大小为1024，中间层的块大小为512，底层的叶子节点的块大小为128。而在检索时只拿叶子节点和问题进行匹配，当某个父节点下的多数叶子节点都与问题匹配上则将该父节点作为结果返回。
  - **多向量检索：**多向量检索同样会给一个知识文档转化成多个向量存入数据库，不同的是，这些向量不仅包括文档在不同大小下的分块，还可以包括该文档的摘要，用户可能提出的问题等等有助于检索的信息。在使用多向量查询的情况下，每个向量可能代表了文档的不同方面，使得系统能够更全面地考虑文档内容，并在回答复杂或多方面的查询时提供更精确的结果。例如，如果查询与文档的某个具体部分或摘要更相关，那么相应的向量就可以帮助提高这部分内容的检索排名

- **多代理检索：**多代理检索，简而言之就是选取我们提及的12大优化策略中的部分交给一个智能代理合并使用。就比如使用子问题查询，多级索引和多向量查询结合，先让子问题查询代理把用户提问拆解为多个小问题，再让文档代理对每个字问题进行多向量或多索引检索，最后排名代理将所有检索的文档总结再交给LLM。这样做的好处是可以取长补短，比如，子问题查询引擎在探索每个子查询时可能会缺乏深度，尤其是在相互关联或关系数据中。相反，文档代理递归检索在深入研究特定文档和检索详细答案方面表现出色，以此来综合多种方法解决问题。需要注意的是现在网络上存在不同结构的多代理检索，具体在多代理选取哪些优化步骤尚未有确切定论，我们可以结合使用场景进行探索
- **Self-RAG：**自反思搜索增强是一个全新RAG框架，其与传统RAG最大的区别在于通过检索评分和反思评分来提高质量。它主要分为三个步骤：检索、生成和批评。Self-RAG首先用检索评分来评估用户提问是否需要检索，如果需要检索，LLM将调用外部检索模块查找相关文档。接着，LLM分别为每个检索到的知识块生成答案，然后为每个答案生成反思评分来评估检索到的文档是否相关，最后将评分高的文档当作最终结果一并交给LLM

- **重排模型：**Reranking，重排模型通过对初始检索结果进行更深入的相关性评估和排序，确保最终展示给用户的结果更加符合其查询意图。这一过程通常由深度学习模型实现，如Cohere模型。这些模型会考虑更多的特征，如查询意图、词汇的多重语义、用户的历史行为和上下文信息等
  - 举个例子，对于查询“最新上映的科幻电影推荐”，在首次检索阶段，系统可能基于关键词返回包括科幻电影的历史文章、科幻小说介绍、最新电影的新闻等结果。然后，在重排阶段，模型会对这些结果进行深入分析，并将最相关、最符合用户查询意图的结果（如最新上映的科幻电影列表、评论或推荐）排在前面，同时将那些关于科幻电影历史或不太相关的内容排在后面。这样，重排模型就能有效提升检索结果的相关性和准确性，更好地满足用户的需求。在实践中，使用RAG构建系统时都应考虑尝试重排方法，以评估其是否能够提高系统性能

6. **生成回答阶段**
   - **提示词：**大型语言模型的解码器部分通常基于给定输入来预测下一个词。这意味着设计提示词或问题的方式将直接影响模型预测下一个词的概率。这也给了我们一些启示：通过改变提示词的形式，可以有效地影响模型对不同类型问题的接受程度和回答方式，比如修改提示语，让LLM知道它在做什么工作，是十分有帮助的

     为了减少模型产生主观回答和幻觉的概率，一般情况下，**RAG系统中的提示词中应明确指出回答仅基于搜索结果**，不要添加任何其他信息例如，可以设置提示词如：

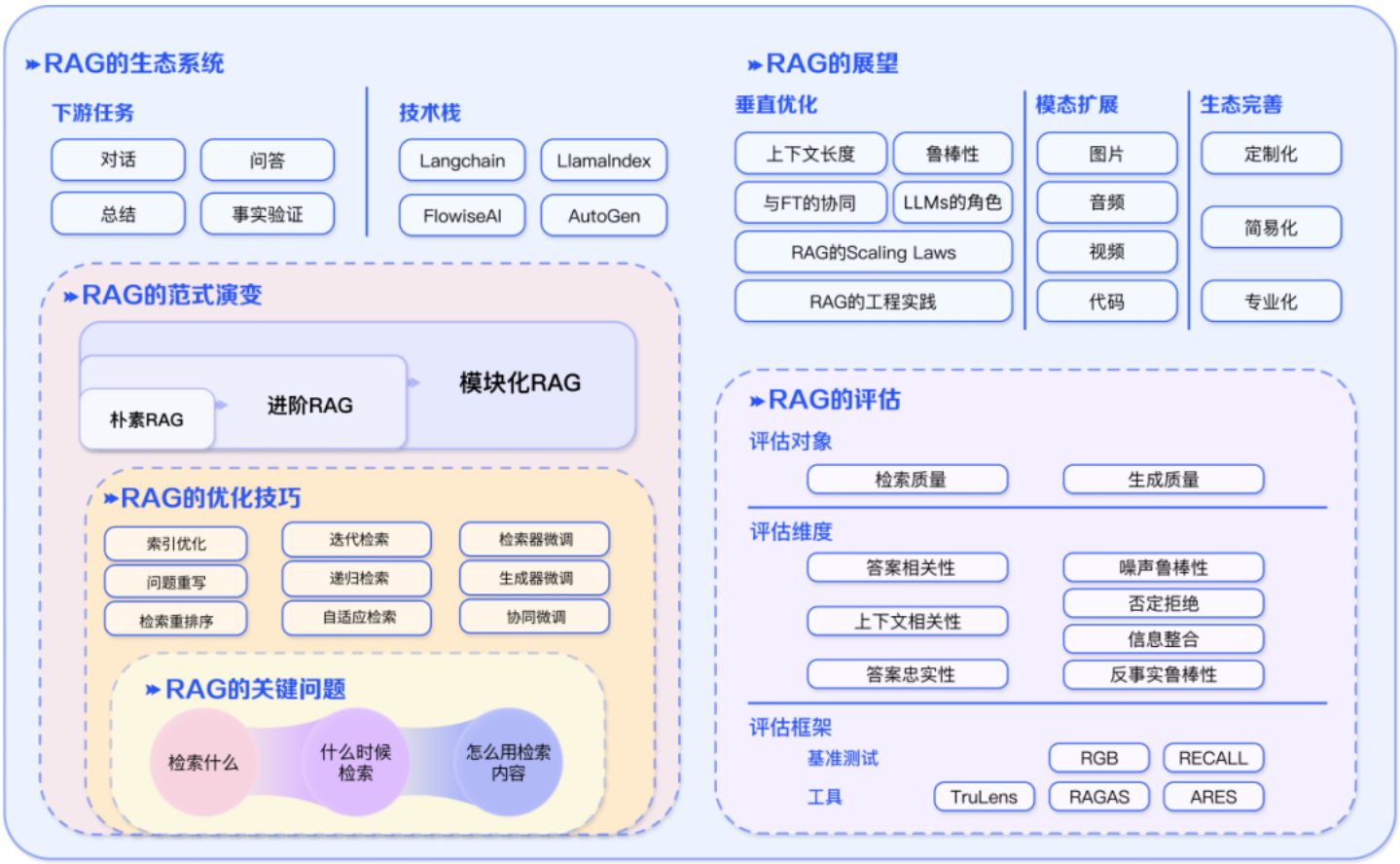
     > “你是一名智能客服。你的目标是提供准确的信息，并尽可能帮助提问者解决问题。你应保持友善，但不要过于啰嗦。请根据提供的上下文信息，在不考虑已有知识的情况下，回答相关查询。”

     当然也可以根据场景需要，也可以适当让模型的回答融入一些主观性或其对知识的理解。此外，使用少量样本（few-shot）的方法，将想要的问答例子加入提示词中，指导LLM如何利用

检索到的知识，也是提升LLM生成内容质量的有效方法。这种方法不仅使模型的回答更加精准，也提高了其在特定情境下的实用性

- **LLM：**可以根据自己的需求选择LLM，例如开放模型与专有模型、推理成本、上下文长度等。此外，可以使用一些LLM开发框架来搭建RAG系统，比如，**LlamaIndex或LangChain**。这两个框架都拥有比较好用的debugging工具，可以让我们定义回调函数，查看使用了哪些上下文，检查检索结果来自哪个文档等等

RAG展望：



6. Training & Inferring 训练推理优化

6.1 FlashAttention

FlashAttention提出了一种加速计算，节省显存，IO感知的精确注意力，可以有效地缓解transformer模型的计算量和储存复杂度随着序列长度 N 呈二次方增长带来的资源和效率问题，而且FlashAttention在训练和推理都可以用

- **加快了计算（Fast）：**Flash Attention并没有减少计算量FLOPs，而是从IO感知出发，**减少了HBM访问次数，从而减少了计算时间**。减少HBM访问次数，是通过**tiling技术分块和算子融合**来实现的
- **节省了显存（Memory-efficient）：**Flash Attention通过引入统计量，改变注意力机制的计算顺序，**避免了实例化注意力矩阵，将显存复杂度从 O(N^2) 降低到了 O(N)**
- **精确注意力（Exact Attention）：**不同于稀疏注意力，Flash Attention只是分块计算，而不是近似计算，**Flash Attention与原生注意力的结果是完全等价的**

6.1.1 计算与内存限制

self-attention块的**计算复杂度和空间复杂度是序列长度 N 的二次方**，有许多近似注意力的方法尝试减少attention的计算和内存要求。例如，稀疏近似和低秩近似的方法，将计算复杂度降低到了序列长度的线性或亚线性，但这些方法并没有得到广泛应用，因为这些方法过于关注FLOPs(浮点数计算次数)的减少，而忽略了IO读写的内存访问开销。在现代GPU中，**计算速度已经远超过了显存访问速度，transformer中的大部分计算操作的瓶颈是显存访问**

> **注意区分FLOPS和FLOPs。**
>
> FLOPS：全大写，floating point operations per second。每秒钟执行的浮点数操作次数，理解为运算速度，是衡量硬件性能的指标
>
> FLOPs：s为小写，floating point operation。表示浮点数运算次数，理解为计算量，衡量模型或算法的复杂度

- **计算带宽（math bandwidth）：**处理器每秒钟可以执行的数学计算次数，单位通常是OPS(即operations/second)。如果用浮点数进行计算，单位是FLOPS
- **内存带宽 （memory bandwidth）：**处理器每秒钟从内存中读取的数据量，单位是bytes/second
- **math-bound：**性能受限于计算带宽，**大矩阵乘法、通道数很大的卷积运算**
- **memory-bound：**性能受限于内存带宽，**逐点运算的操作大多是内存受限的**，比如：**激活函数、dropout、mask，**另外reduction操作也是内存受限的，比如：**softmax，batch normalization和layer normalization**

对于self-attention块，除了大矩阵乘法是计算受限，其他操作**计算softmax，dropout，mask都是内存受限的**

GPU内存分级：GPU内存由多个不同大小和不同读写速度的内存组成。内存越小，读写速度越快。**SRAM内存约20MB ，高带宽内存HBM也就是我们常说的显存，大小为40GB。SRAM的读写速度为19TB/s，HBM的读写速度只有1.5TB/s**，因此减少对HBM的读写次数，有效利用更高速的SRAM来进行计算是非常重要的

GPU运行模式：GPU有大量的线程来执行某个操作，称为kernel。执行操作分为三步：

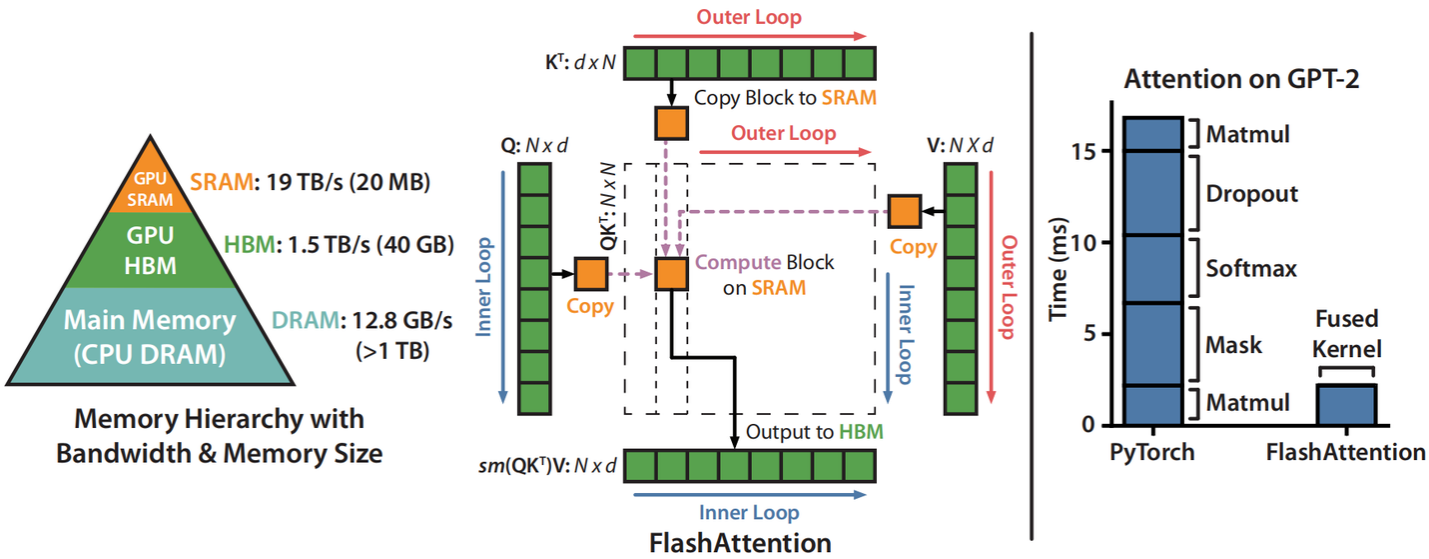
- **每个kernel将输入数据从低速的HBM中加载到高速的SRAM中**
- 在**SRAM中进行计算**
- 将**计算结果从SRAM中写入到HBM中**

对于性能受限于内存带宽的操作，进行加速的常用方式就是**kernel融合，避免反复执行“从HBM中读取输入数据，执行计算，将计算结果写入到HBM中”**，将多个操作融合成一个操作，**减少读写HBM的次数**

Safe Softmax：对于**float32和bfloat16**来说，当 **x≥89 时，exp(x) 就会变成inf，发生数据上溢的问题**。为了避免发生数值溢出的问题，保证数值稳定性，计算时通常会减去最大值，称为**safe softmax**：

$$m = \max_i(x_i), \quad \text{softmax}(x_i) = \frac{e^{x_i - m}}{\sum_{j=1}^{d} e^{x_j - m}}$$

6.1.2 FlashAttention机制



- **Tiling 分块计算：**SRAM的读写速度比HBM高一个数量级，但内存小很多。通过kernel融合的方式，将多个操作融合为一个操作，利用高速的SRAM进行计算，可以减少读写HBM的次数，从而有效减少内存受限操作的运行时间。但SRAM的内存大小有限，不可能一次性计算完整的注意力，因此必须进行分块计算，使得分块计算需要的内存不超过SRAM的大小

  分块计算的难点：注意力计算流程是 **矩阵乘法-->scale-->mask-->softmax-->dropout-->矩阵乘法，矩阵乘法和逐点操作的分块计算是容易实现**的，但是**softmax由于分母需要完整输入数据，所以分块计算很难**

  FlashAttention的做法：引入**额外的统计量** $m(x), l(x)$ 来进行解耦

  - **求Safe Softmax**

    $$m(x) := \max_i x_i, \quad f(x) := [e^{x_1 - m(x)}, \ldots, e^{x_B - m(x)}], \quad l(x) := \sum_i f(x)_i, \quad \text{softmax}(x) := \frac{f(x)}{l(x)}$$

  - **解耦拼接向量的Softmax计算**

    $$m(x) = m([x^{(1)}, x^{(2)}]) = \max(m(x^{(1)}), m(x^{(2)}))$$
    $$f(x) = [e^{m(x^{(1)}) - m(x)} f(x^{(1)}), e^{m(x^{(2)}) - m(x)} f(x^{(2)})]$$
    $$l(x) = l([x^{(1)}, x^{(2)}]) = e^{m(x^{(1)}) - m(x)} l(x^{(1)}) + e^{m(x^{(2)}) - m(x)} l(x^{(2)})$$
    $$\text{softmax}(x) = \frac{f(x)}{l(x)}$$

    通过保持额外的两个统计量可以实现softmax的分块计算，同时多个block的softmax可以并行计算

    > 📌 **一个分块计算Softmax的例子**

对向量[1, 2, 3, 4]计算softmax，分成两块[1, 2]和[3, 4]进行计算

计算block1：

$$
\begin{aligned}
m_1 &= \max(\{1,2\}) = 2 \\
f_1 &= [e^{1-2}, e^{2-2}] = [e^{-1}, e^{0}] \\
l_1 &= \sum f_1 = e^{-1} + e^{0} \\
o_1 &= \frac{f_1}{l_1} = \frac{[e^{-1}, e^{0}]}{e^{-1} + e^{0}}
\end{aligned}
$$

计算block2：

$$
\begin{aligned}
m_2 &= \max(\{3,4\}) = 4 \\
f_2 &= [e^{3-4}, e^{4-4}] = [e^{-1}, e^{0}] \\
l_2 &= \sum f_2 = e^{-1} + e^{0} \\
o_2 &= \frac{f_2}{l_2} = \frac{[e^{-1}, e^{0}]}{e^{-1} + e^{0}}
\end{aligned}
$$

合并得到完整的结果：

$$
\begin{aligned}
m &= \max(m_1, m_2) = 4 \\
f &= [e^{m_1 - m} f_1, e^{m_2 - m} f_2] = [e^{-3}, e^{-2}, e^{-1}, e^{0}] \\
l &= e^{m_1 - m} l_1 + e^{m_2 - m} l_2 = e^{-3} + e^{-2} + e^{-1} + e^{0} \\
o &= \frac{f}{l} = \frac{[e^{-3}, e^{-2}, e^{-1}, e^{0}]}{e^{-3} + e^{-2} + e^{-1} + e^{0}}
\end{aligned}
$$

在忽略mask和dropout的情况下，Flash Attention算法的前向计算在K, V的维度上做外循环，在Q的维度上做内循环

- **kernel融合**

  把mask和dropout加上的forward：

$$
\begin{aligned}
S &= \frac{1}{\sqrt{d_k}} QK^\top \in \mathbb{R}^{N \times N} \\
S_{\text{masked}} &= \text{MASK}(S) \in \mathbb{R}^{N \times N} \\
P &= \text{softmax}(S_{\text{masked}}) \in \mathbb{R}^{N \times N} \\
P_{\text{dropped}} &= \text{dropout}(P, p_{\text{drop}}) \in \mathbb{R}^{N \times N} \\
O &= P_{\text{dropped}} V \in \mathbb{R}^{N \times d}
\end{aligned}
$$

  tiling分块计算使得可以用一个CUDA kernel来执行注意力的所有操作，从HBM中加载输入数据，在SRAM中执行所有的计算操作（矩阵乘法，mask，softmax，dropout，矩阵乘法），再将计算结果写回到HBM中，通过kernel融合将多个操作融合为一个操作，避免了反复地从HBM中读写数据

6.2 Megatron-LM

Megatron-LM 是一个**基于 PyTorch 的分布式训练框架**，用来训练基于Transformer的大型语言模型

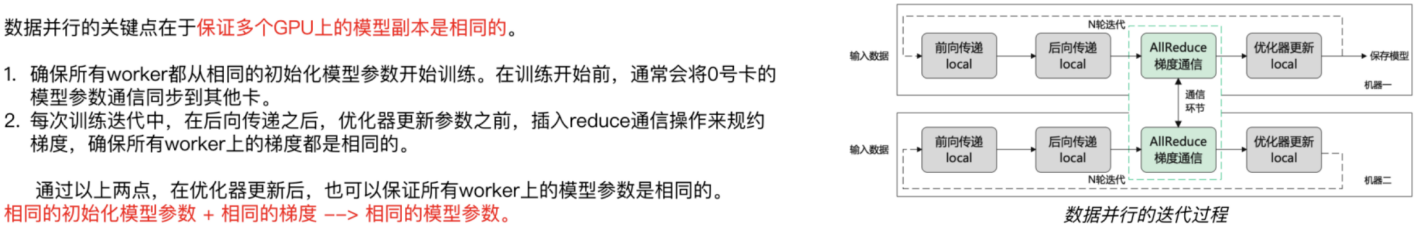
大型模型能够提供更精准和强大的语义理解与推理能力。随着计算资源的普及和数据集的增大，模型参数的数量呈指数级增长。然而，训练这样规模庞大的模型面临着一些挑战：

- **显存限制：** 即便是目前最大的GPU主内存也难以容纳这些模型的参数。举例来说，一个1750亿参数的GPT-3模型需要约700GB的参数空间，对应的梯度约为700GB，而优化器状态还需额外的1400GB，总计需求高达2.8TB
- **计算挑战：** 即使我们设法将模型适应单个GPU（例如通过在主机内存和设备内存之间进行参数交换），模型所需的大量计算操作也会导致训练时间大幅延长。举个例子，使用一块NVIDIA V100 GPU来训练拥有1750亿参数的GPT-3模型，大约需要耗时288年
- **并行策略挑战：** 不同的并行策略对应不同的通信模式和通信量，这也是一个需要考虑的挑战

6.2.1 数据并行

Data Parallelism DP

假设有 N 张卡（worker），**每张卡都保存一个完整模型的副本，每一次迭代（iteration/step）都将输入数据集batch分割成 N 个等大小的micro-batch，每张卡根据拿到的micro-batch数据独立计算梯度，worker定期聚合它们的梯度（调用AllReduce计算梯度均值）**，每张卡再独立进行参数更新。对于无法放进单个Worker的大模型，可以在模型的较小片段上使用数据并行



数据并行的迭代过程

数据并行的扩展通常表现出色，但存在两个限制：

- **降低GPU利用率：** 在一定点之后，每个GPU的批量大小变得太小，这会降低GPU的利用率，并增加通信成本
- **设备数限制：** 可用于训练的GPU设备数量受到批量大小的限制，这限制了可以使用的最大设备数

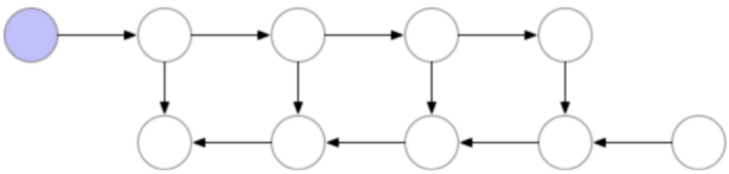
> 📌 `All-Reduce` 是一种在并行计算和分布式计算中常用的通信操作，用于在多个计算节点（例如 GPU 或 CPU）之间汇总数据
>
> **工作原理：**
>
> - **数据分发：**每个节点在完成了自己的计算后会生成一组梯度（或者其他需要汇总的数据），这些数据需要在所有节点之间进行汇总
> - **数据汇总：**All-Reduce 操作将每个节点上的数据在所有节点之间进行相加或其他类型的归约操作（例如，求和、求平均、求最大值等），这意味着最终每个节点都会得到相同的结果，该结果是所有节点数据的汇总
> - **结果广播：**完成汇总操作后，汇总结果会被广播回每个节点，使得每个节点都持有相同的汇总数据。这一步确保了所有节点在下一步的计算中都能使用一致的数据

6.2.2 模型并行

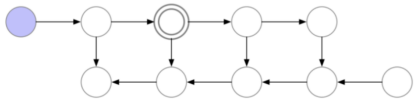
Model Parallelism MP

对于数据并行的限制，可以采用一些内存管理技术，比如激活检查点（**Activation Checkpointing**）

Activation Checkpointing（gradient_checkpointing）：前向传播过程中计算节点的激活值并保存，计算下一个节点完成后丢弃中间节点的激活值，反向传播时如果有保存下来的梯度就直接使用，如果没有就使用保存下来的前一个节点的梯度重新计算当前节点的梯度再使用

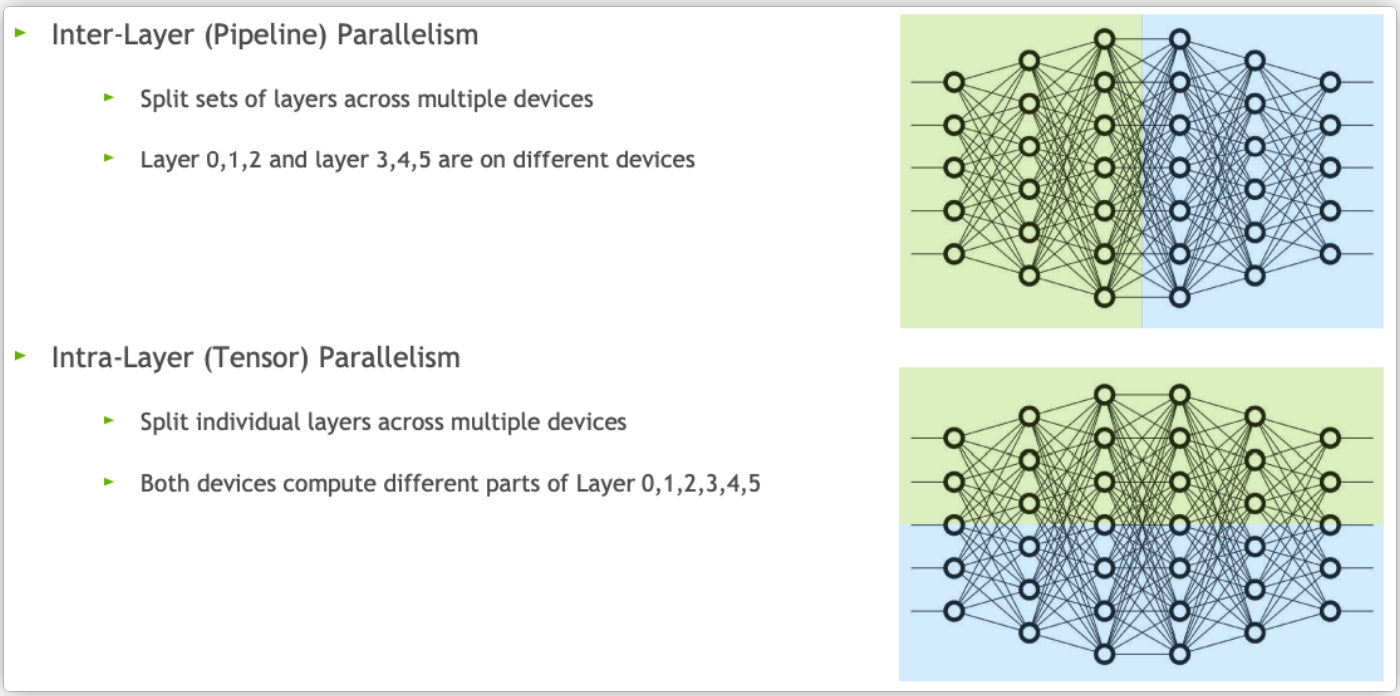


普通反向传播需要把前向传递中所有的激活都保存下来，
这意味着n层的前馈神经网络需要O(n)的内存。
紫色圆圈需要保存在内存中。



在前向传递时，只把检查点节点保存在内存中，检查点以外的中间值全部丢弃。
在反向传播计算梯度时，从最近的检查点开始计算，其余节点最多计算一次。
内存由普通策略的O(n)减少为$O(\sqrt{n})$。检查点策略需要的额外计算基本上相当于一次前向传递。

此外，还可以使用**模型并行**来划分模型的不同阶段，从而解决GPU内存容量和计算限制的问题，使得权重和关联的优化器状态不需要同时存储在一个GPU上。一个模型的内存和计算被分布在多个计算节点上，主要的方式分为**流水线并行和张量并行**



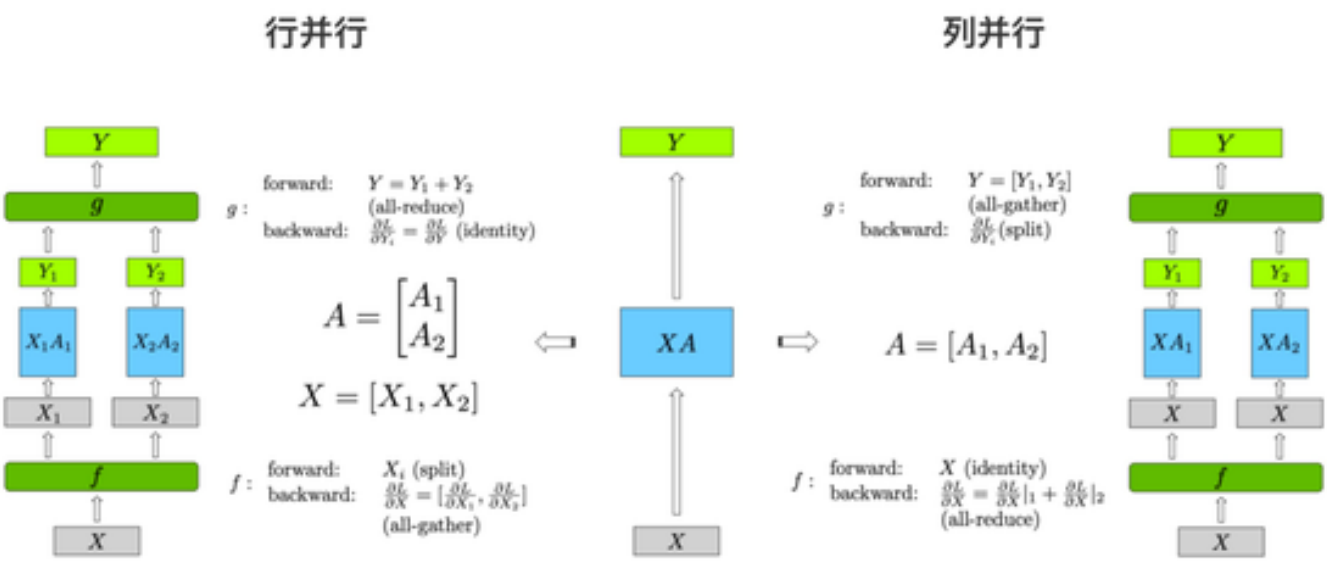
- **张量并行 Intra-Layer Tensor Parallelism：**有的tensor/layer很大，一张卡放不下，将tensor分割成多块，一张卡存一块。这也可以被理解为将大矩阵运算拆分成多个小矩阵运算，然后分布到不同的设备上进行计算

```
1 # 假设模型有三层：L0, L1, L2 每层有两个神经元 两张卡
2 GPU0:
```

```
L0 | L1 | L2
---|----|---
a0 | b0 | c0

GPU1:L0 | L1 | L2
---|----|---
a1 | b1 | c1
```

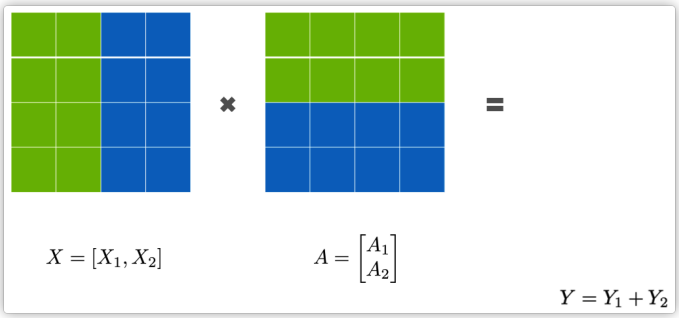
高度的模型并行会产生很多的小矩阵乘法(GEMMS)，对于一个GEMMs $Y = XA$ ，按照对权重矩阵A的分块方式，**张量并行又可以分成行并行和列并行。**神经网络中的线性层（Linear层），可以将其看作是**将输入矩阵分块进行计算，然后将计算结果合并成输出矩阵**。这个过程涉及矩阵乘法和加法操作，其中矩阵乘法涉及到权重矩阵和输入数据之间的乘法，然后再加上偏置向量；对于非线性层（例如激活函数层），通常不需要进行额外的设计。这些层的计算过程是基于输入数据应用某种非线性函数，例如ReLU（修正线性单元）、Sigmoid、Tanh等。这些函数在数学上是已知的，只需要将输入数据传递给这些函数，然后得到输出。整体来看，神经网络的计算可以被抽象为一系列的矩阵和向量操作，其中线性层涉及矩阵乘法和加法，而非线性层涉及特定的函数计算。这些操作在深度学习框架中会被高度优化，以提高计算效率和训练速度

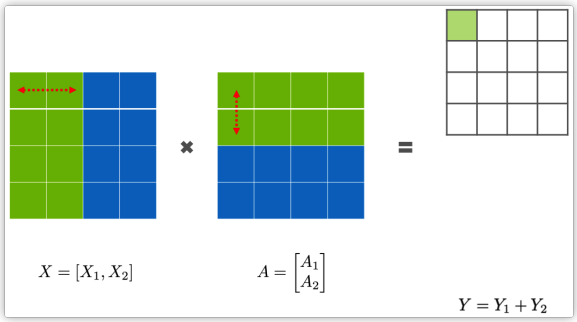


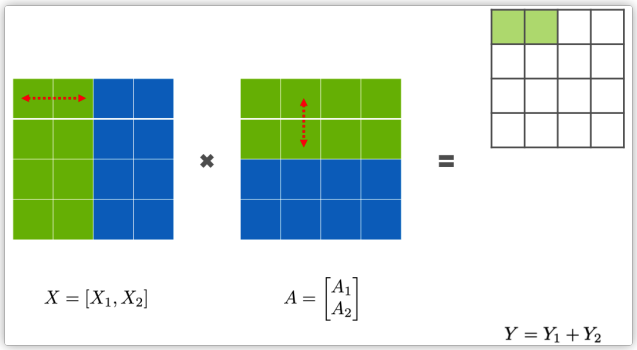
- **GEMMs行并行：**就是把权重矩阵 A 按照行切分成两部分，对应的把 X 按照列切分成两部分

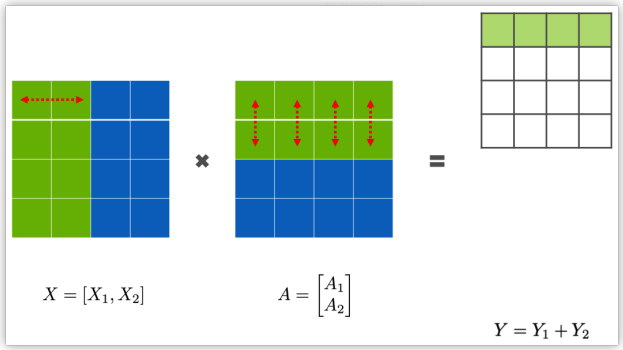
$$XA = [X_1 \quad X_2] \begin{bmatrix} A_1 \\ A_2 \end{bmatrix} = X_1A_1 + X_2A_2 = Y_1 + Y_2 = Y$$

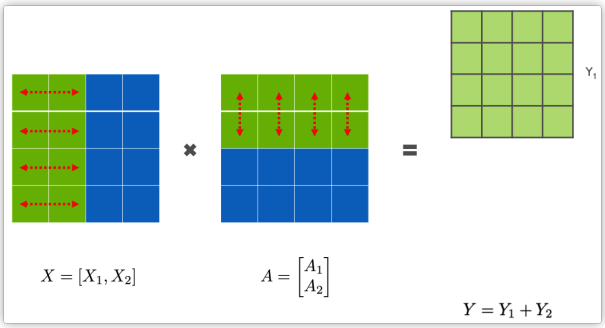
这样 X_1, A_1 可以在第一个GPU上计算， X_2, A_2 可以在第二个GPU上计算，然后进行结果相加即可

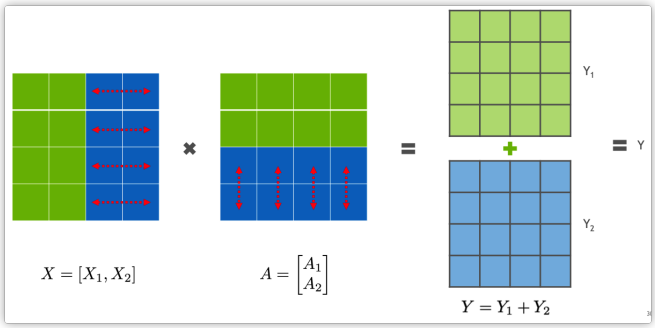




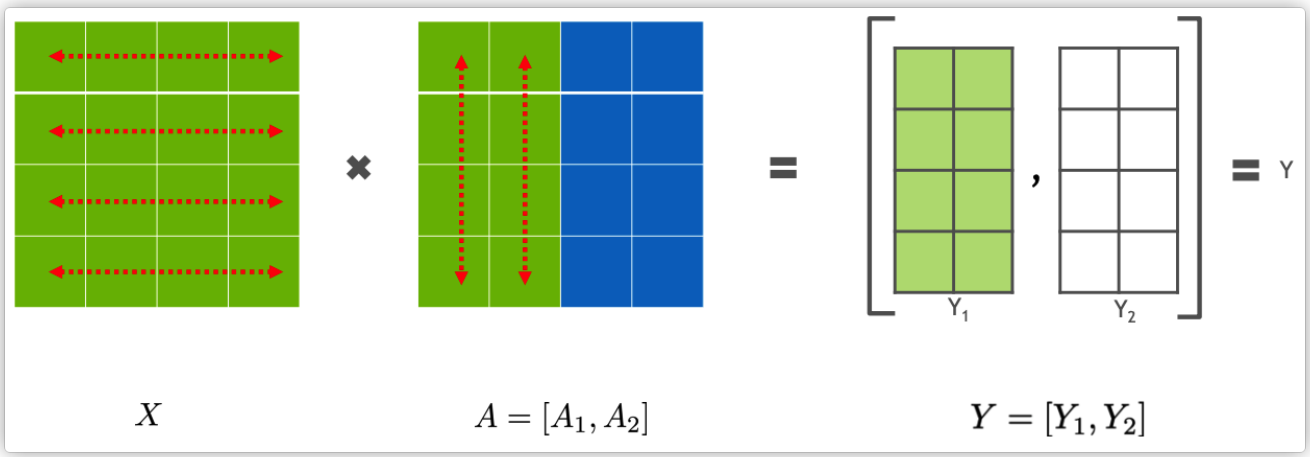


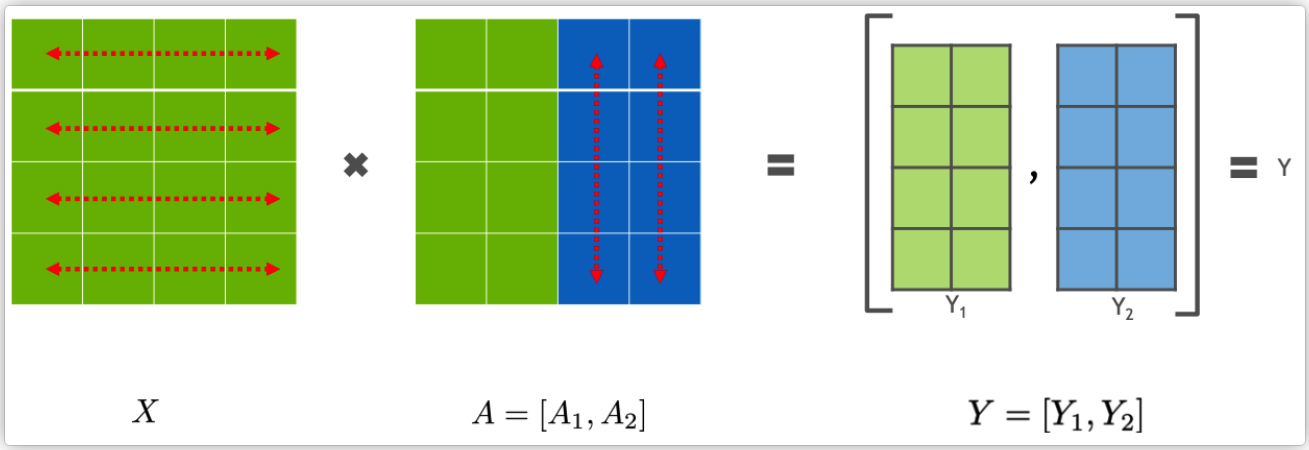






- **GEMMs列并行：**就是把 A 按照列来分割



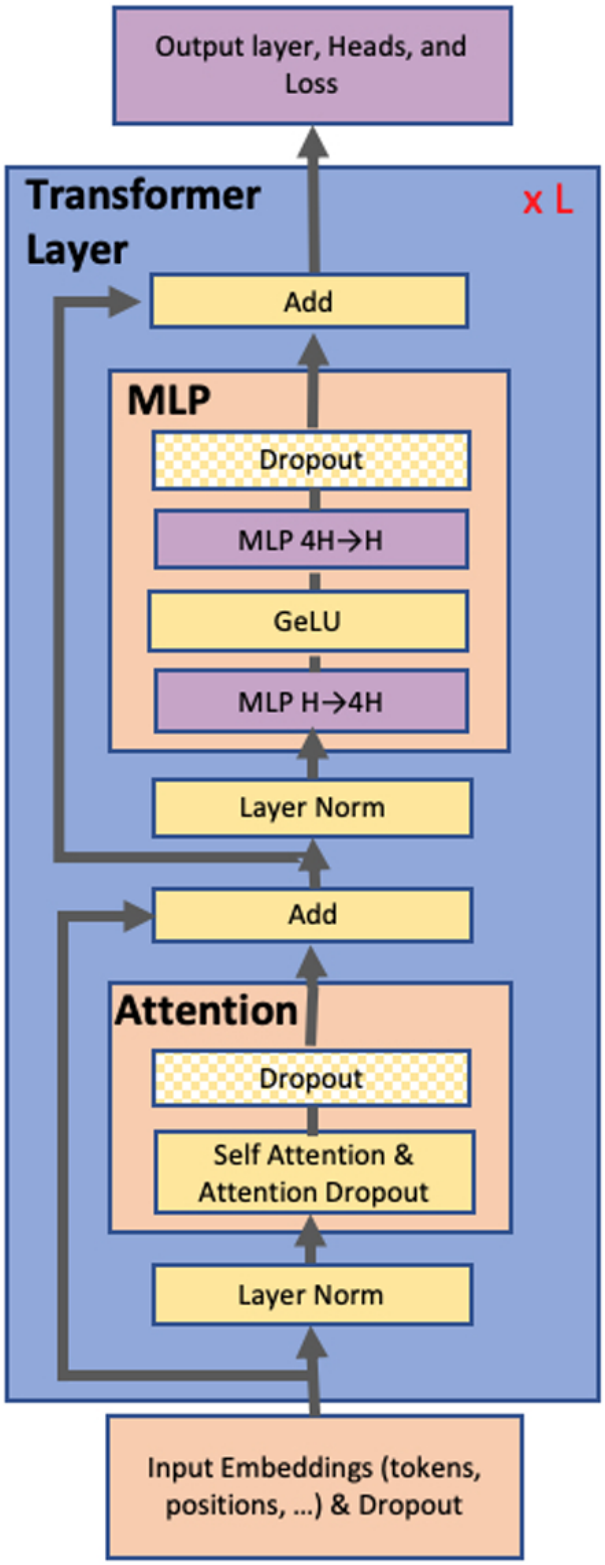


Transformer中的张量并行：

每个Transformer块包含了Self-Attention和Feed Forward Network，这些操作是大量的矩阵计算，适合在GPU上进行并行操作，从而加速模型的训练和推理过程

1. **Masked Multi-Head Self Attention：** 涉及到大量的矩阵乘法操作，这些操作可以被高效地并行执行
2. **Feed Forward Neural Network：** 包含多个全连接层，每个全连接层都涉及矩阵乘法、激活函数（通常是GeLU）和可能的Dropout层。这些操作也是高度并行化的，可以在GPU上迅速执行

Megatron 的 FFN是一个两层MLP，第一层是维度从 H 变成 4H，第二层是维度从 4H 变回到 H，所以Transformer具体架构如下，紫色块对应于全连接层，每个蓝色块表示一个被复制 N 次的transformer层，红色的 x L 代表此蓝色复制 L 次



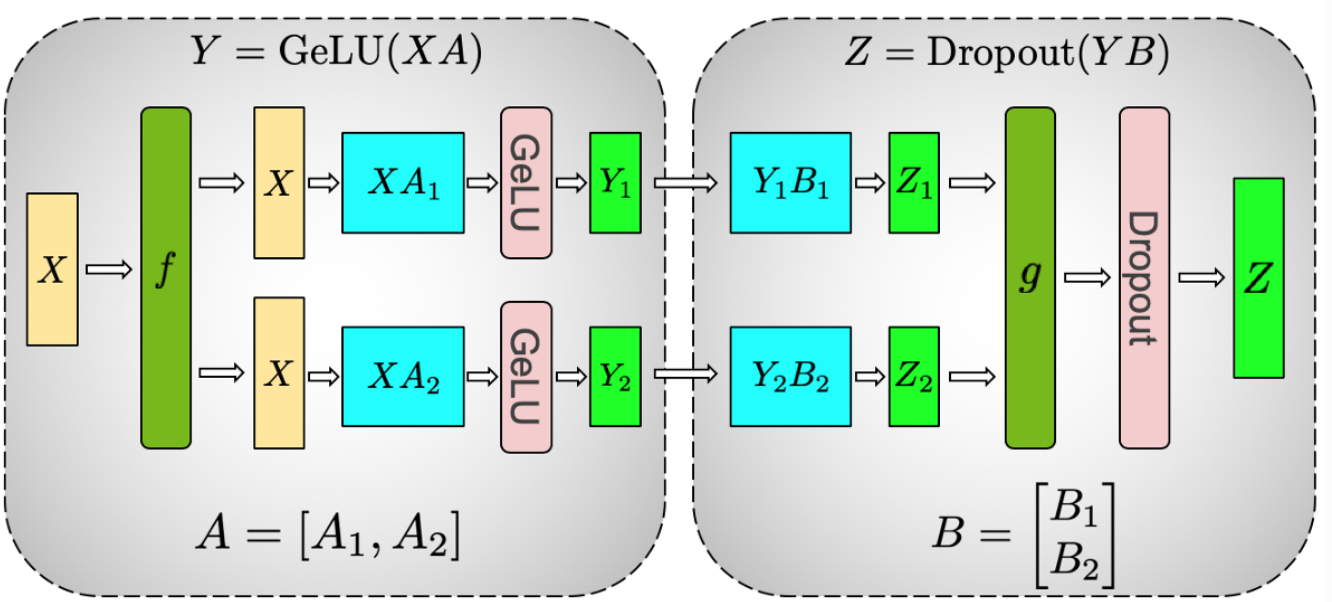
张量并行就是要对Transformer进行切分，Megatron 把 Masked Multi-Head Self Attention 和 Feed Forward 都进行切分以并行化，通过添加一些同步通信操作来创建一个简单的模型并行实现

- **切分MLP：**从MLP开始，MLP的第一部分是GEMM，后面是激活函数GeLU $Y = \mathrm{GeLU}(XA)$

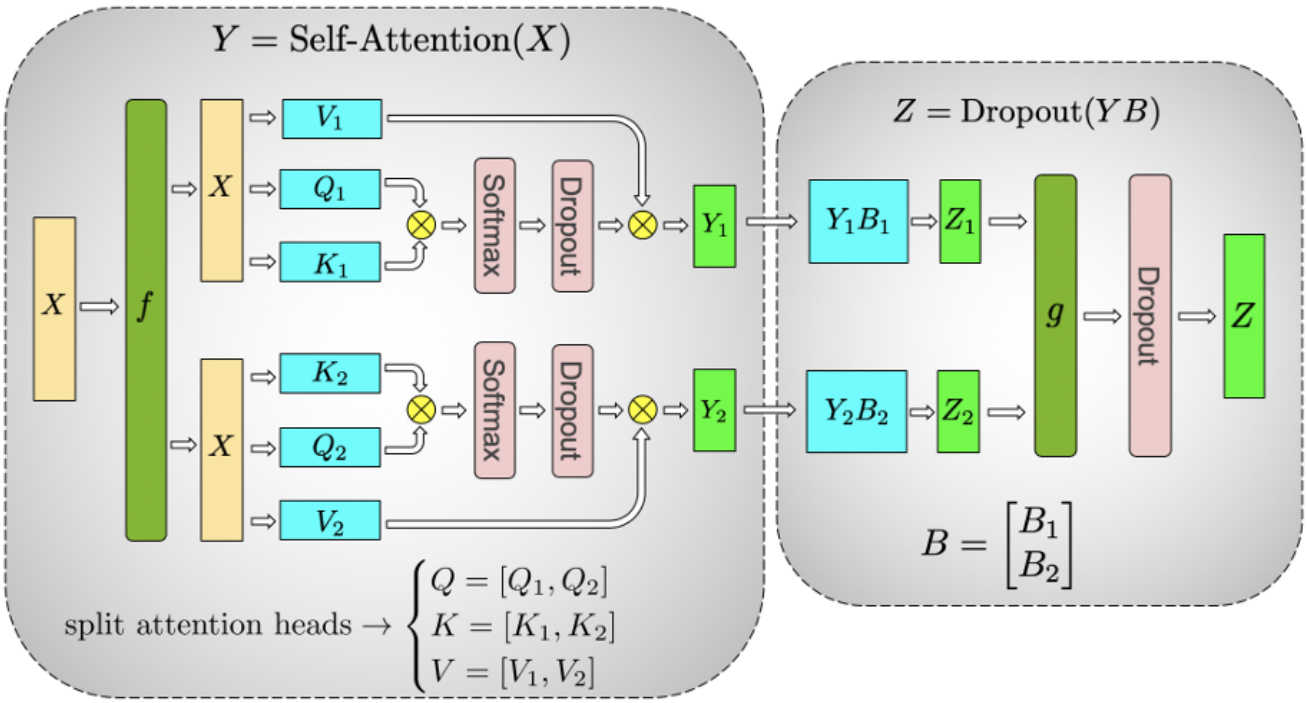
  对比按照行列切分权重的方法：

  - **行并行：** $Y = \mathrm{GeLU}(X_1A_1 + X_2A_2)$，但是**GeLU是非线性计算，也就是** $\mathrm{GeLU}(X_1A_1 + X_2A_2) \neq \mathrm{GeLU}(X_1A_1) + \mathrm{GeLU}(X_2A_2)$**，**所以这种方案需要在GeLU函数之前加上一个同步点，这个同步点让不同GPU之间交换信息

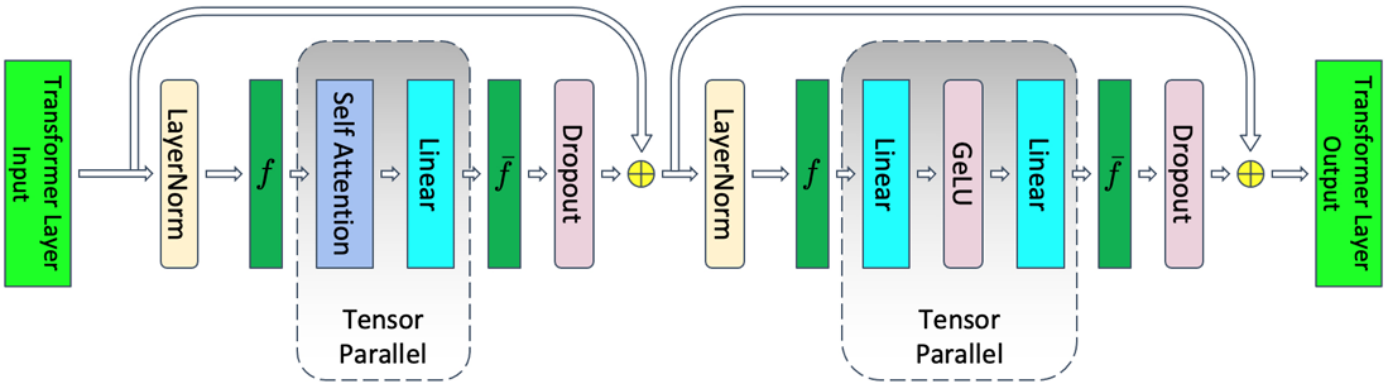
- **列并行：** $[Y_1 \quad Y_2] = [\mathrm{GeLU}(XA_1), \mathrm{GeLU}(XA_2)]$，**不需要同步点，直接把两个GeLU的输出拼接起来就行**，而不需要额外的通信（比如GeLU这里就不用all-reduce了）。下图所示，f, g 分别是在前向和后向传递的时候做一次all-reduce通信操作。对于第二个MLP使用行并行，可以直接与第一个MLP得到的 Y_1, Y_2 对应计算，不需要额外通信操作



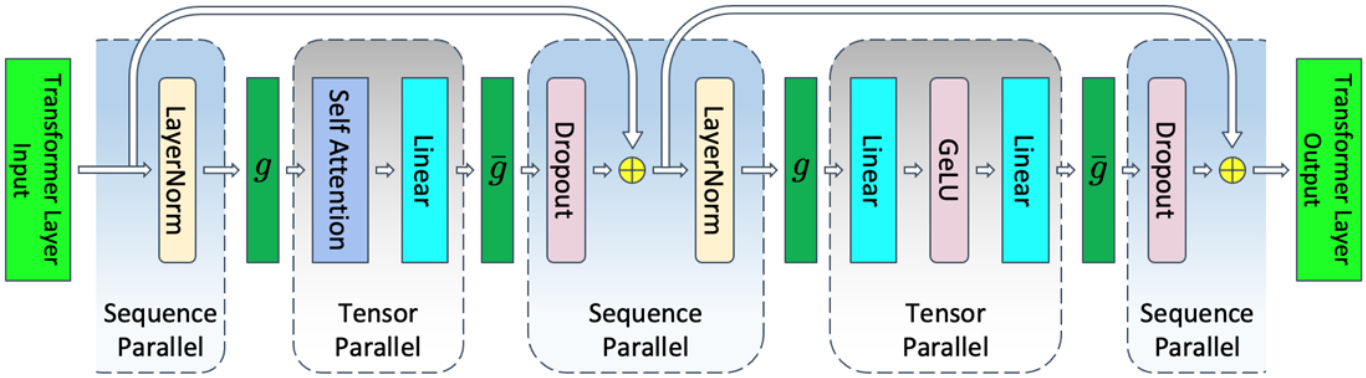
- **切分self-attention：**利用多头注意力固有的并行特性，直接**以列并行的方法切分QKV相关的GEMMs**



一个TransformerBlock需要在前向和后向总共需要4个 all-reduce通信操作



张量并行对AttentionBlock和MLP都进行了并行化，但是**没有对LN和dropout进行并行化**，不过后续在megatron3中用**序列并行**技术对这两部分进行了并行，这部分就不过多展开了

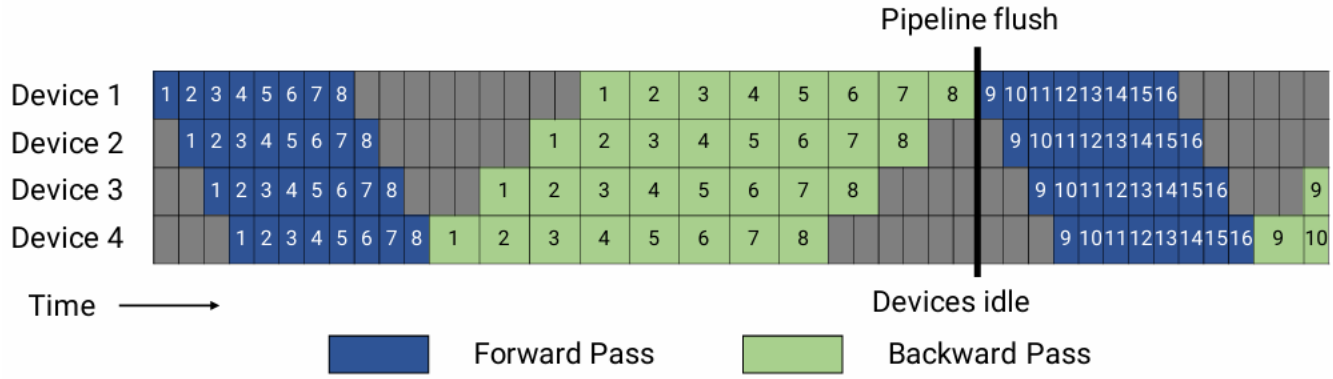


- **流水线并行 Inter-Layer Pipeline Parallelism**：将**网络按层切分，划分成多组，一张卡存一组。**例如，前几层放在一个设备上，中间几层放在另 一个设备上，最后几层放在第三个设备上。这种方式可以在不同设备上并行执行不同的模型阶段，从而提高效率

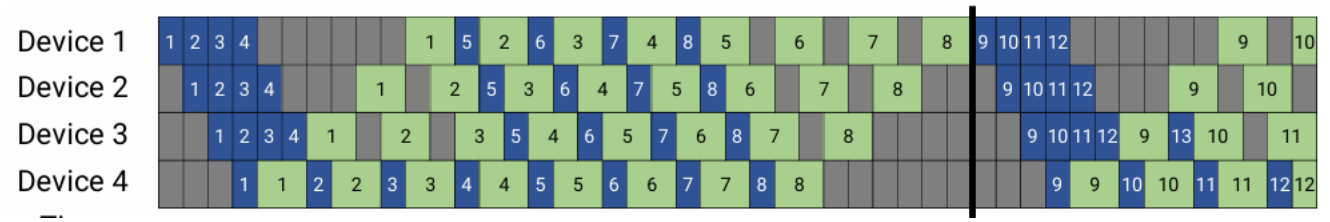
```
# 假设模型有8层 两张卡
======================  ======================
|  L0 | L1 | L2 | L3 |  | L4 | L5 | L6 | L7 |
======================  ======================
          GPU0                    GPU1
# 设想一下，当GPU0在进行（前向/后向）计算时，GPU1在干嘛？闲着
# 当GPU1在进行（前向/后向)计算时，GPU0在干嘛？闲着
# 为了防止”一卡工作，众卡围观“，实践中PP也会把batch数据分割成
# 多个micro-batch，流水线执行
```

目前主流的流水线并行方法包括了两种：**Gpipe和PipeDream**。与这两者相比，Megatron中的流水线并行实现略有不同，采用了Virtual Pipeline的方法。传统的流水线并行通常会在一个设备上放置几个模块，通过在计算和通信之间取得平衡来提高效率。然而，虚拟流水线则采取相反的策略。在设备数量不变的前提下，它将流水线阶段进一步细分，以承载更多的通信量，从而**降低空闲时间的比率，以缩短每个步骤的执行时间**

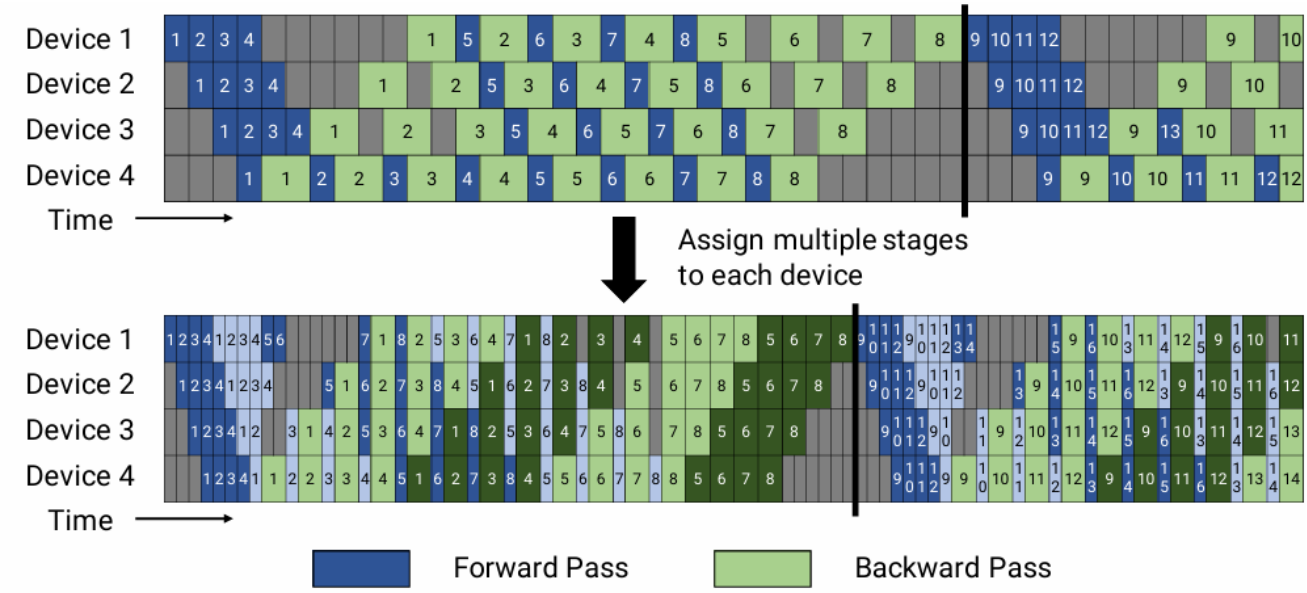
- **Gpipe：**将transformer layer 按层切分放到 Device1-4 上，并将一个 batch 切分成 8 个mini-batch。前向时，每个mini-batch 从 Device 1 流向 Device 4；反向时，在 Device4 上算出梯度并更新 Device4 对应层的参数，将梯度传向 Device3，Device3 做梯度计算和参数更新后传给 Device 2 等。这么做的劣势之一是空泡率(流水线空闲时间)比较高



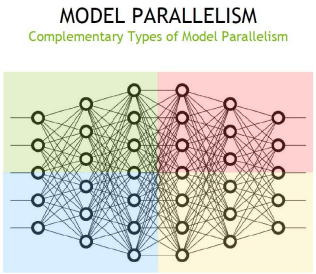
- **PipeDream：**P其相对与 G-pipe 的改进在内存方面， 空泡时间和 G-pipe 一致，但通过合理安排前向和反向过程的顺序，在 step 中间的稳定阶段，形成 1前向1反向 的形式，称为 1F1B 模式，在这个阶段，每个 Device 上最少只需要保存 1 份 micro-batch 的激活值，最多需要保存流水线阶段数份激活值，可以有效降低空泡占比

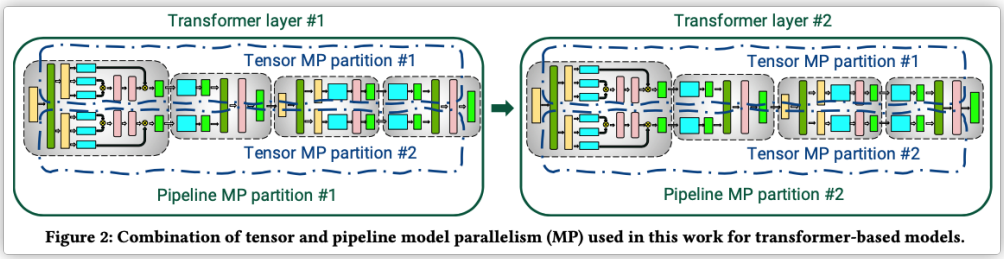


- **Virtual Pipeline：**传统的 pipeline 并行通常会在一个 Device 上放置几个 block，这是为了扩展效率考虑，在计算强度和通信强度中间取一个平衡。但 virtual pipeline 的却反其道而行之，在 device 数量不变的情况下，分出更多的 pipeline stage，以更多的通信量，换取空泡比率降低，减小了 step e2e 用时。virtual 若网络共16层（编号 0-15），4个Device，前述 G-pipe 和 PipeDream 是分成 4 个stage， 按编号 0-3 层放 Device1，4-7层放 Device2 以此类推。virtual pipeline 则是减小切分粒度，以 virtaul_pipeline_stage=2 为例，将 0-1 层放 Device1, 2-3 层放在 Device2，...，6-7 层放到 Device4，8-9 层继续放在 Device1，10-11 层放在 Device2，...，14-15 层放在 Device4，在 steady 的时候也是 1F1B 的形式。按照这种方式，Device之间的点对点通信次数(量)直接翻了 virtual_pipeline_stage 倍，但空泡比率降低了



实际上**流水线并行和张量并行是正交互补的**，而且可以**同时存在**





6.2.3 3D并行

📌 **数据并行：**计算效率高，**实现简单**

- 显存效率：每张卡上都保存了**完整的模型、梯度、优化器状态**，因此显存效率不高
- 计算效率：当增加并行度时，单卡的计算量是保持恒定的，可以实现近乎完美的线性扩展。但规约梯度的通信开销，与模型大小成正相关

张量并行：因模型结构而异，**实现难度大**

- 显存效率：随着并行度增加，**成比例地减少显存占用**。是减少单层神经网络中间激活的唯一方法
- 计算效率：**频繁的通信**，限制了两个通信阶段之间的计算量，影响了计算效率，**计算效率很低**

流水线并行：通信成本最低

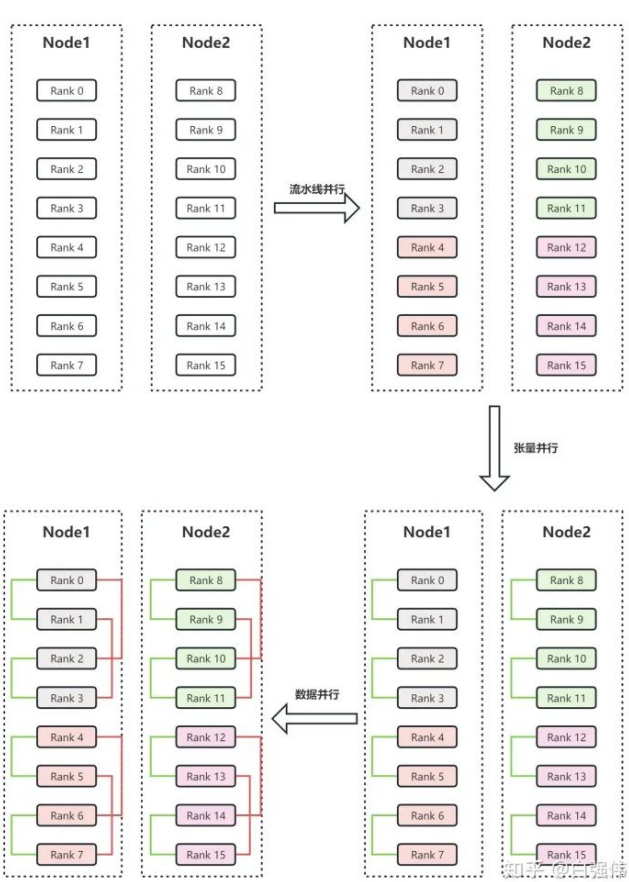
- 显存效率：**减少的显存与流水线并行度成正相关**。但流水线并行不会减少每层中间激活的显存占用
- 计算效率：成本更低的**点对点（P2P）通信**。通信量与流水线各个阶段边界的激活值大小成正相关

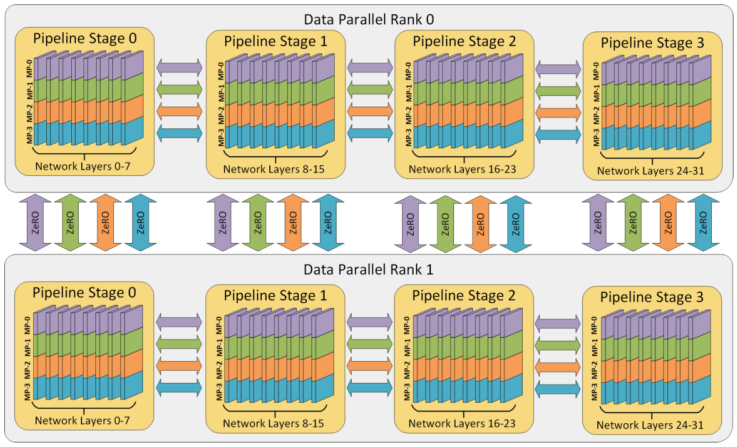
显存效率：模型并行 > 流水线并行 > 数据并行

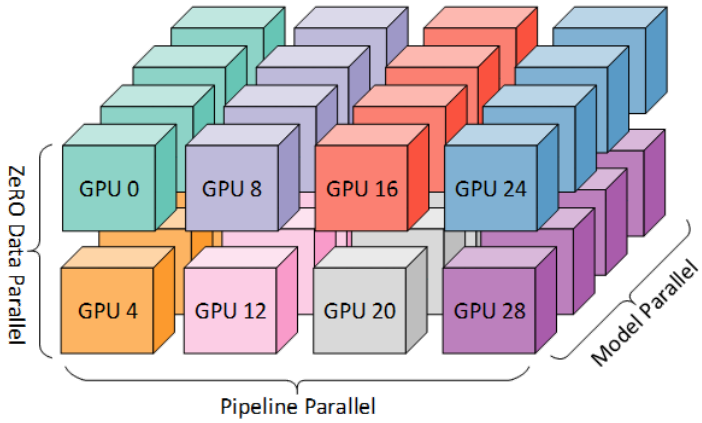
通信效率：流水线并行 > 数据并行 > 模型并行

混合数据并行(DP)、张量并行(TP)和流水线并行(PP)组成

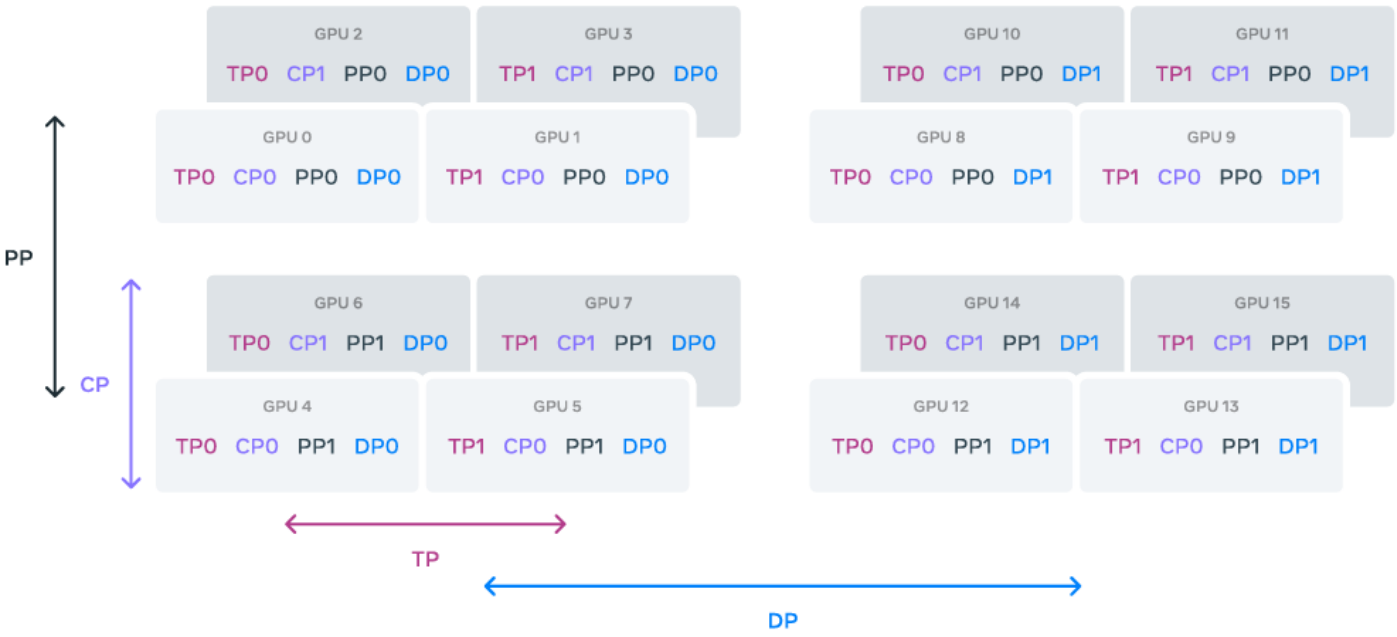
3D并行，四路张量并行，四路流水线并行，两路数据并行，32个worker

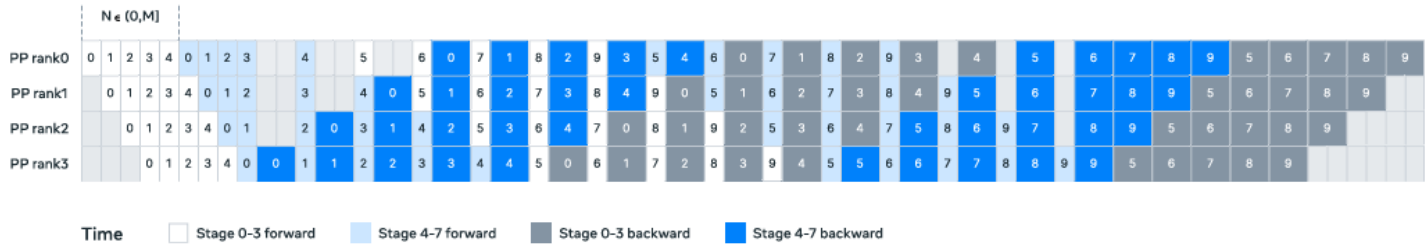






在LLaMA3的技术报告中，他们还使用了**4D并行的技术，除了PP，TP和DP，还加上了CP（context parallel）**





6.3 DeepSpeed

6.3.1 混合精度训练

训练的时候同时使用FP16和FP32两种精度的数据类型，主要在进行forward/backward的时候转FP16，参数更新的时候转FP32更新

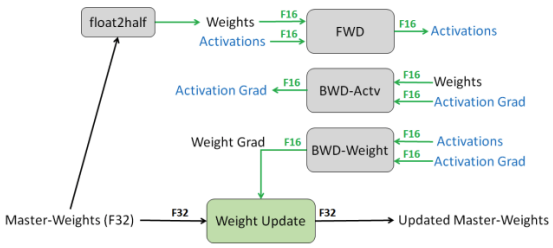
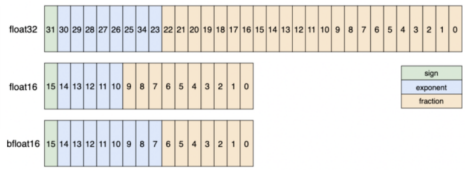


Figure 1: Mixed precision training iteration for a layer.

Format	Bits	Exponent	Fraction
FP32	32	8	23
FP16	16	5	10
BF16	16	8	7



数值动态范围

FP32 $1 \times 2^{-126} \sim (2-\epsilon) \times 2^{127}$

FP16 $1 \times 2^{-14} \sim (2-\epsilon) \times 2^{15}$

BF16 $1 \times 2^{-126} \sim (2-\epsilon) \times 2^{127}$

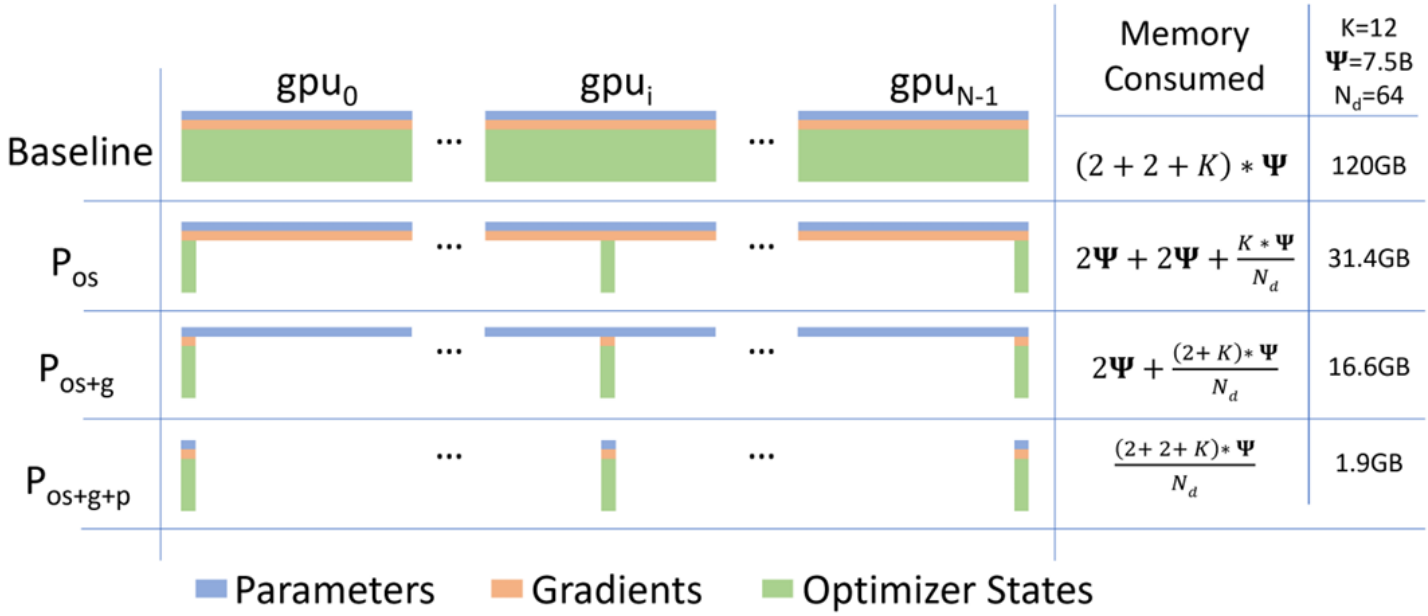
为了保证LLM的性能，bfloat16好于float16。
GPT3、GLM130B使用了float16进行训练。
PaLM、BLOOM使用了bfloat16进行训练。

6.3.2 ZeRO 零冗余优化器

ZERO的显存分类：

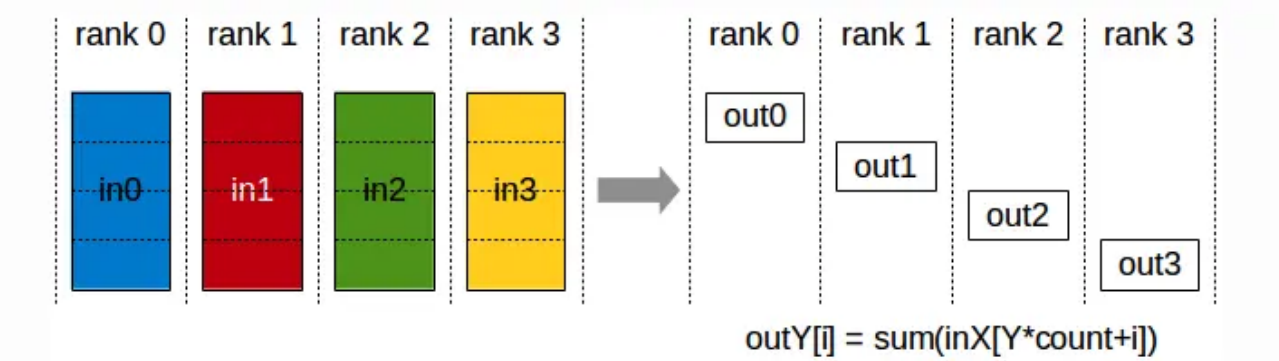
- **模型状态（model states）**：模型参数（fp16）、模型梯度（fp16）和**Adam状态（fp32的模型参数备份，fp32的momentum和fp32的variance）**。假设模型参数量 Φ，则共需要 2Φ+2Φ+（4Φ+4Φ+4Φ）=4Φ+12Φ=16Φ 字节存储，可以看到，Adam优化器状态占比 75%，更新参数的时候还得加上FP32的梯度
- **剩余状态（residual states）**：除了模型状态之外的显存占用，包括激活值（activation）、各种临时缓冲区（buffer）以及无法使用的显存碎片（fragmentation），激活值可以用**activation checkpointing** 来大大减少

ZeRO三个阶段以及显存占用分析：

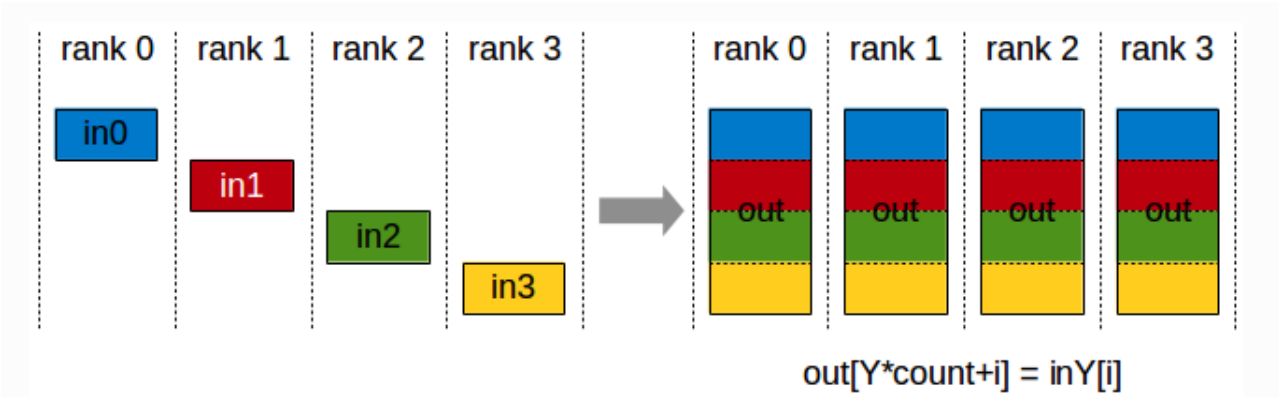


数据通信量分析：DeepSpeed ZeRO主要是数据并行

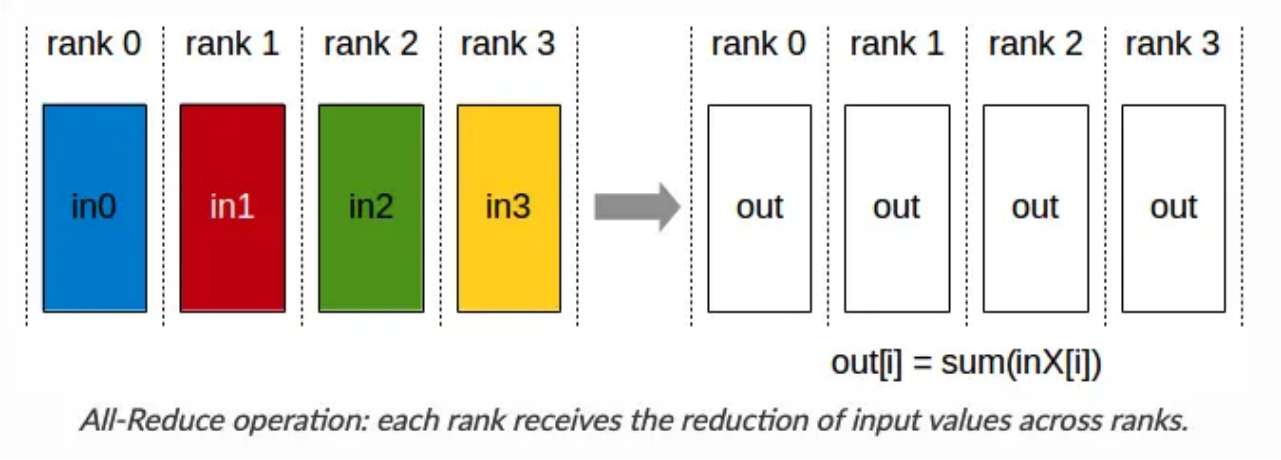
- **传统数据并行：**传统数据数据并行在每一步（step/iteration）计算梯度后，需要进行一次**AllReduce**操作来计算梯度均值，目前常用的是**Ring AllReduce**，分为**ReduceScatter**和**AllGather**两步，**每张卡的通信数据量（发送+接受）近似为 2Φ**



Reduce-Scatter operation: input values are reduced across ranks, with each rank receiving a subpart of the result.



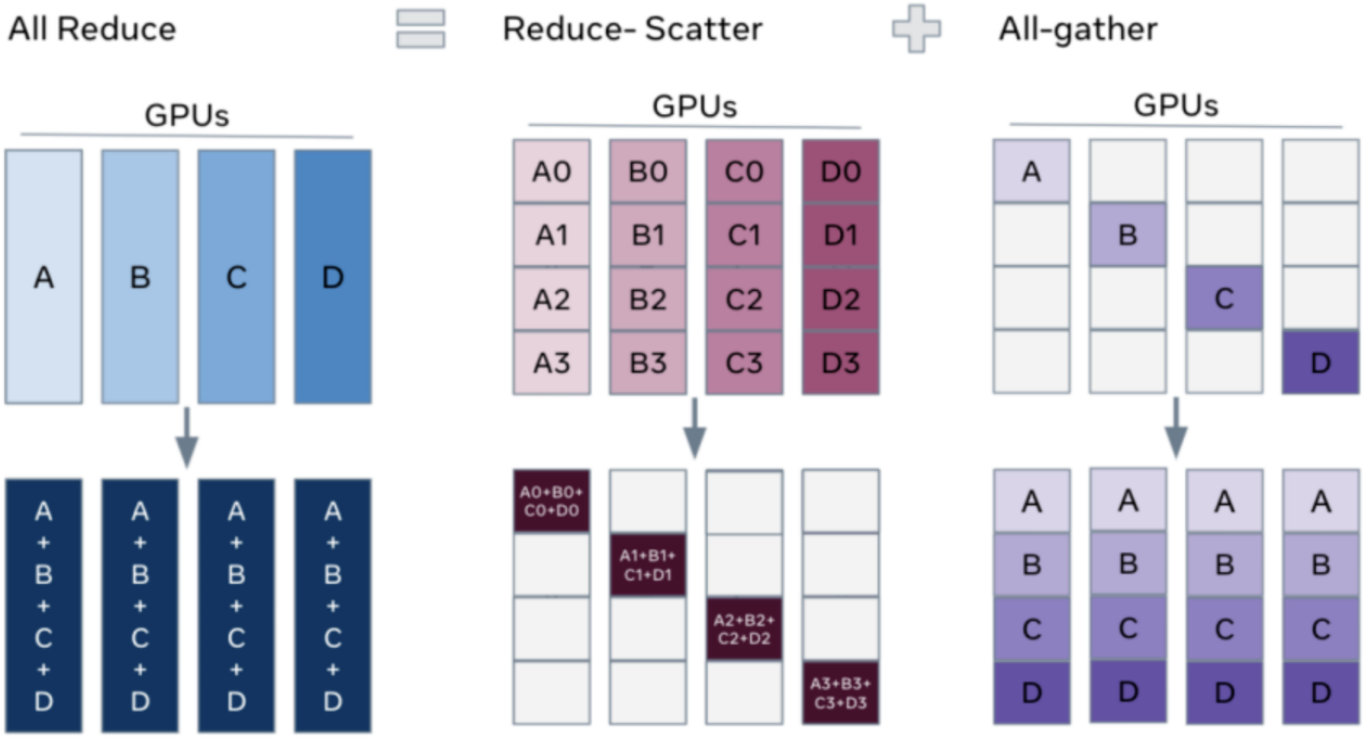
AllGather operation: each rank receives the aggregation of data from all ranks in the order of the ranks.



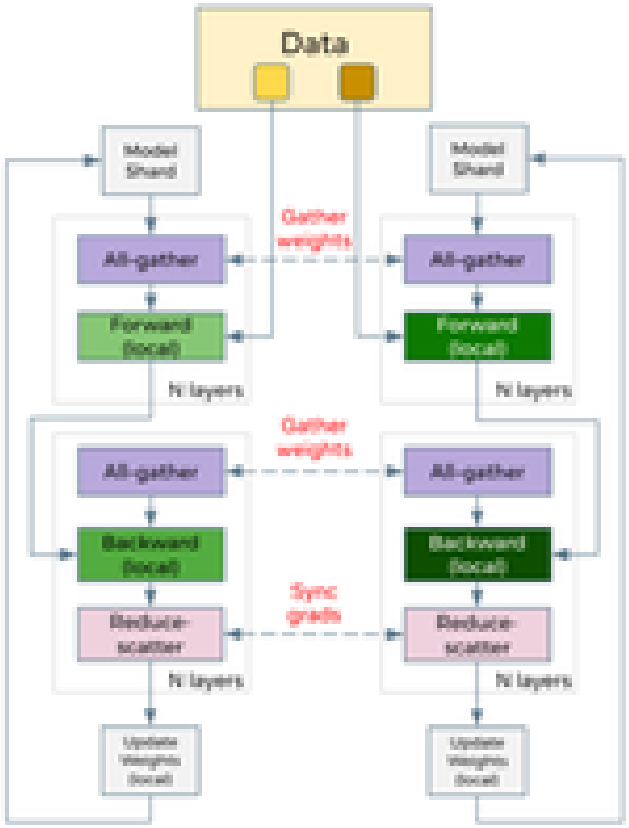
All-Reduce operation: each rank receives the reduction of input values across ranks.

- **ZeRO1、2阶段(优化器状态分区、梯度分区)：**同传统数据并行

  直接分析 P_{os+g}，每张卡只存储 $\frac{1}{N}$ 的优化器状态和梯度，对于gpu_0来说，为了计算它这 $\frac{1}{N}$ 梯度的均值，需要进行一次Reduce操作，通信数据量是 $\frac{1}{N}\Phi \cdot N = \Phi$，然后其余显卡则不需要保存这部分梯度值了。实现中使用了bucket策略，保证 $\frac{1}{N}$ 的梯度每张卡只发送一次.当 gpu_0 计算好梯度均值后，就可以更新局部的优化器状态（包括 $\frac{1}{N}\Phi$ 的参数），当反向传播过程结束，进行一次Gather操作，更新 $(1-\frac{1}{N})\Phi$ 的模型参数，通信数据量是 $\frac{1}{N}\Phi \cdot N = \Phi$。全局来看，相当于用Reduce-Scatter和AllGather两步，和数据并行一致



- **ZERO3：**每张卡只存了 1/N 的参数，不管是在前向计算还是反向传播，都涉及一次Broadcast操作， **All gather + ReduceScatter + AllGather**通信量是**3Φ，是标准数据并行的1.5倍**



- **CPU卸载 ZERO-offload：不仅卸载数据，而且将计算卸载到CPU，显著减少GPU - CPU通信量**

  CPU卸载应满足在以下三个方面最优：

  - 减少CPU的计算量，避免CPU成为性能瓶颈
  - 最小化GPU - CPU之间的通信量，避免通信开销成为性能瓶颈
  - 在最小化通信量时，最大化GPU的显存节省

  数据卸载：

  - FP32的模型参数、优化器状态 --> CPU
  - FP16的模型参数、中间激活 --> GPU

  计算卸载：

  - 优化器更新模型参数 --> CPU
  - 前向传递和后向传递 --> GPU

  通信量：FP16的梯度、模型参数

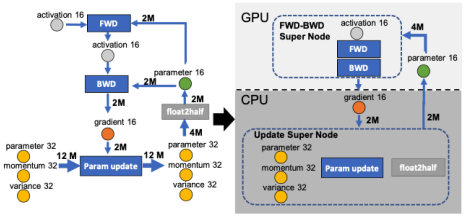


Figure 2: The dataflow of fully connected neural networks with *M* parameters. We use activation checkpoint to reduce activation memory to avoid activation migration between CPU and GPU.

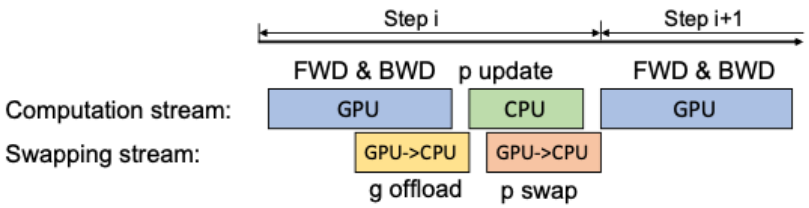


Figure 3: ZeRO-Offload training process on a single GPU.

6.4 Accelerate

Accelerate 是一个为 PyTorch 用户设计的库，旨在帮助简化分布式训练和混合精度训练的过程。它提供了一种简单且灵活的方式来加速和扩展您的 PyTorch 训练脚本，而无需编写冗长的样板代码。使用

户能够更轻松地利用大规模计算资源并加速模型训练过程。在**无需大幅修改代码的情况下完成并行化，支持DeepSpeed的多种ZeRO策略**

- **代码效率高，**支持无论是单机单卡还是多机多卡适配同一套代码
- 允许在**单个实例上训练更大的数据集**：Accelerate 还可以使 `DataLoaders` 更高效。这是通过自定义采样器实现的，它可以在训练期间自动将部分批次发送到不同的设备，从而允许每个设备只需要储存数据的一部分，而不是一次将数据复制四份存入内存
- **支持DeepSpeed：**无需更改代码，只用配置文件即可对DeepSpeed开箱即用、

分布式推理：

当模型参数大到单张卡放不下时候（哪怕batchsize为1也会报显存不足的情况），这里就需要将大模型中的不同layer放到不同的GPU上，而每个GPU只负责其中一部分训练，当然数据在不同卡上流转的时候，都需要自动将数据放到对应的卡上

```
from accelerate import dispatch_model
# device_map设置为自动的最方便了，不用自己设计把模型的layer分配到哪个GPU
model = dispatch_model(model, device_map="auto")
# 打印device_map
print(model.hf_device_map)
print(f'memory_allocated {torch.cuda.memory_allocated()}')
```

分布式训练：

下面是官方的例子，只需要更新几行代码即可开启分布式训练之旅啦！

- `accelerator = Accelerator()` 实例化
- `accelerator.prepare` 把我们的模型、数据、优化器等等都放进accelerate里面，让他帮我们操作
- `accelerator.backward(loss)` 替换掉常用的 `loss.backword()`
- 如果需要梯度裁剪，这里必须使用 `accelerator.clip_grad_norm_()`
- 涉及到模型存储的时候，需要unwrap 下模型,因为在通过 prepare() 方法时，模型可能被 wrap 从而用于分布式训练

```
import torch
import torch.nn.functional as F
from datasets import load_dataset
from accelerate import Accelerator
accelerator = Accelerator()
device = 'cpu'
device = accelerator.device
model = torch.nn.Transformer().to(device)
```

```
optimizer = torch.optim.Adam(model.parameters())
dataset = load_dataset('my_dataset')
data = torch.utils.data.DataLoader(dataset, shuffle=True)
model, optimizer, data = accelerator.prepare(model, optimizer, data)
model.train()
for epoch in range(10):
  for source, targets in data:
      source = source.to(device)
      targets = targets.to(device)
      optimizer.zero_grad()
      output = model(source)
      loss = F.cross_entropy(output, targets)
      loss.backward()
      accelerator.backward(loss)
      optimizer.step()
# 等待所有进程达到一定程度后再执行指令
accelerator.wait_for_everyone()
# 只在主进程中保存模型
if self._accelerator.is_main_process:
     unwrapped_model = accelerator.unwrap_model(model)
     accelerator.save(unwrapped_model,model_path)
     torch.save(unwrapped_model.state_dict, "./model/accelerate.pt")
```

6.5 vLLM

vLLM是一个开源的大模型推理加速框架，通过PagedAttention高效地管理attention中缓存的张量，实现了比HuggingFace Transformers高24倍的吞吐量，其原理是基于PagedAttention的

PagedAttention：

大模型推理的性能瓶颈主要来自于内存。一是**自回归过程中缓存的K和V张量非常大**，在LLaMA-13B中，单个序列输入进来需要占用1.7GB内存。二是**内存占用是动态的，取决于输入序列的长度。由于碎片化和过度预留，现有的系统浪费了60%-80%的内存**。PagedAttention灵感来自于操作系统中虚拟内存和分页的经典思想，它可以允许**在非连续空间里存储连续的KV张量**。具体来说，**PagedAttention把每个序列的KV缓存进行了分块，每个块包含固定长度的token，而在计算attention时可以高效地找到并获取那些块**

https://pic3.zhimg.com/v2-e8a2317d1bc7ba5670ca05f68196453e_b.webp

每个固定长度的块可以看成虚拟内存中的页，token可以看成字节，序列可以看成进程。那么通过一个块表就可以将连续的逻辑块映射到非连续的物理块，而物理块可以根据新生成的token按需分配

https://pic4.zhimg.com/v2-6035b0440dd9f0eb37bc9c221b977799_b.webp

所以序列在分块之后，只有最后一个块可能会浪费内存（实际中浪费的内存低于4%）。高效利用内存的好处很明显：系统可以在一个batch中同时输入更多的序列，提升GPU的利用率，显著地提升吞吐量

PagedAttention的另外一个好处是**高效内存共享**。例如，在并行采样的时候，一个prompt需要生成多个输出序列。这种情况下，对于这个prompt的计算和内存可以在输出序列之间共享

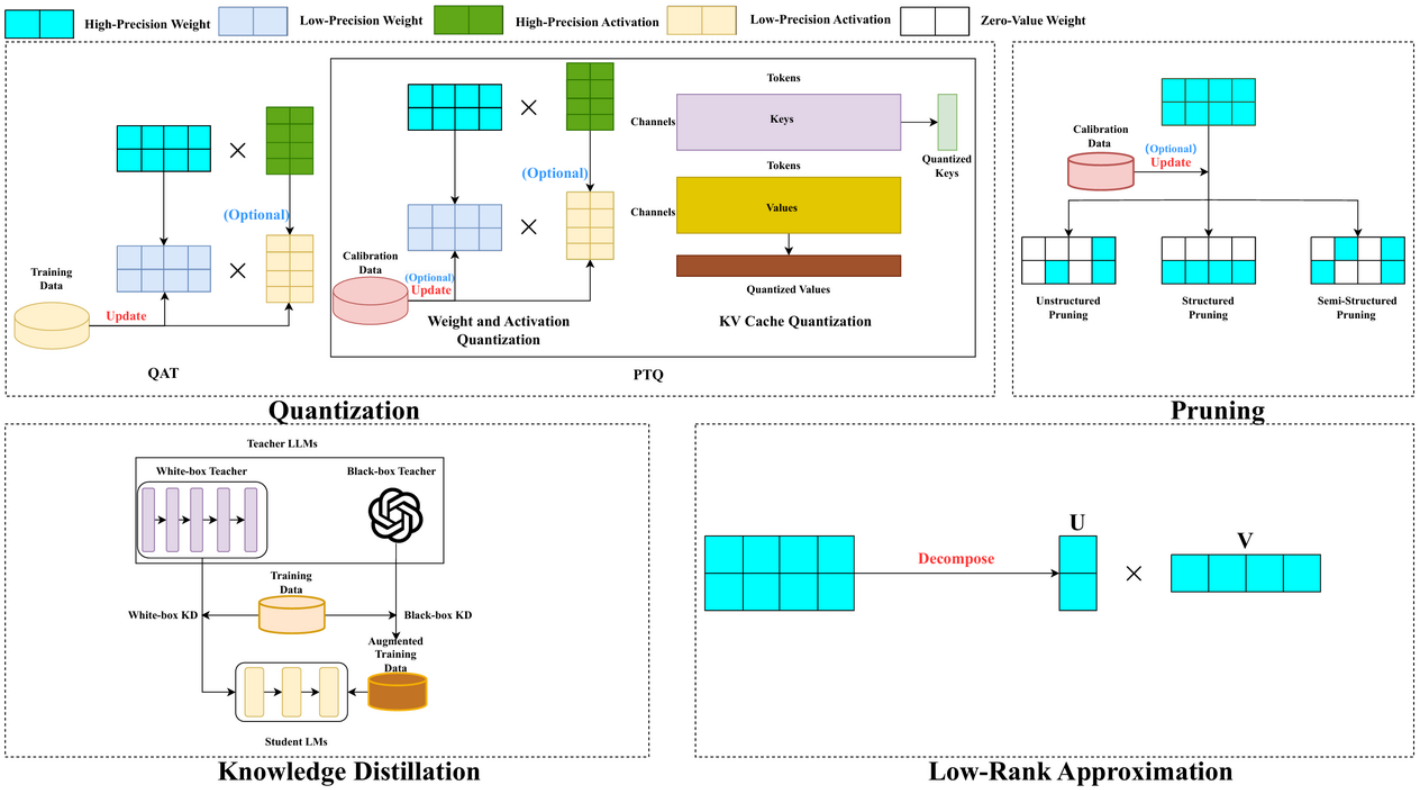
https://pic1.zhimg.com/v2-b22751a85181c355d4acaa222d781afe_b.webp

通过块表可以自然地实现内存共享。类似进程间共享物理页，在PagedAttention中的不同序列通过将逻辑块映射到一样的物理块上可以实现共享块。为了确保安全共享，PagedAttention跟踪物理块的引用计数，并实现了Copy-on-Write机制。 内存共享减少了55%内存使用量，大大降低了采样算法的内存开销，同时提升了高达2.2倍的吞吐量

https://pica.zhimg.com/v2-cab043f5f4d3ed2f4e369a542617fb22_b.webp

7. Compression 模型压缩（了解）

模型变得越来越大，因此需要一些大模型压缩技术来降低模型部署的成本，并提升模型的推理性能



7.1 Quantization 模型量化

7.1.1 量化简介

模型量化是指以较低的推理精度损失将连续取值如**float32，float16的浮点型权重近似为有限多个离散值权重如int8、int4**的过程。通过以更少的位数表示浮点数据，模型量化可以减少模型尺寸，进而减少在推理时的内存消耗，并且在一些低精度运算较快的处理器上可以增加推理速度

📌 **常用的数据类型：**

- **FP32**：32位浮点数，是最常用的高精度表示方式
- **FP16**：16位浮点数，数值范围比FP32小，但占用内存较少
- **BF16**：16位截断的FP32，增加指数位，数值范围更广，常用于深度学习
- **INT8**：8位整数，位数仅为FP32的1/4，适用于模型参数的数据范围映射
- **INT4**：4位整数，进一步减少位数，适用于极端资源受限的场景
- **二值网络（Binary Network）**：1位二值网络，参数只能取0或1，计算效率极高但精度损失较大

工业界目前最常用的量化位数是8比特，低于8比特的量化被称为低比特量化。1比特是模型压缩的极限，可以将模型压缩为1/32，在推理时也可以使用高效的XNOR和BitCount位运算来提升推理速度

量化对象：

- **权重（Weight）**：**权重**的量化是最常规，可达到减少模型大小内存和占用空间
- **激活值（Activation）**：实际中**激活值**往往是占内存使用的大头，量化激活值不仅可以大大减少内存占用，更重要的是，结合权重的量化可以充分利用整数计算获得性能提升

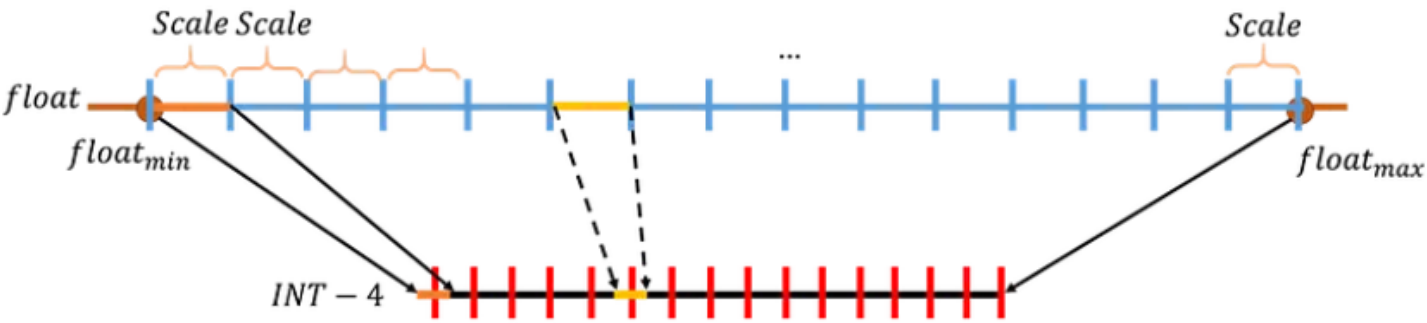
  > 激活值是神经元在接收到输入信号并经过激活函数处理后所产生的输出值

- **KV Cache**：量化 KV 缓存对于提高长序列生成的吞吐量至关重要
- **梯度（Gradients）**：相对上面的量化对象略微小众一些，因为主要用于训练。在训练深度学习模型时，梯度通常是浮点数，它主要作用是在分布式计算中减少通信开销，也可以减少backward时的开销

量化形式：

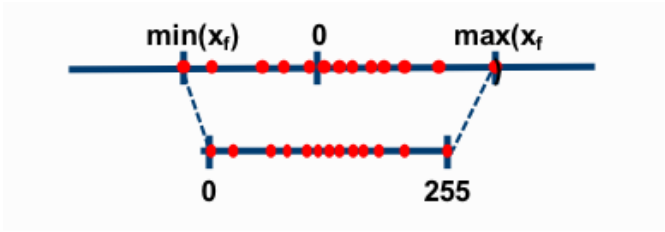
根据量化数据表示的原始数据范围是否均匀，可以将量化方法分为**线性量化和非线性量化**。深度神经网络的权重和激活值通常是不均匀的，因此理论上使用非线性量化导致的精度损失更小，但在实际推理中非线性量化的计算复杂度较高，**通常使用线性量化：**

$$q = clip(round(\frac{r}{s} + z), q_{min}, q_{max})$$

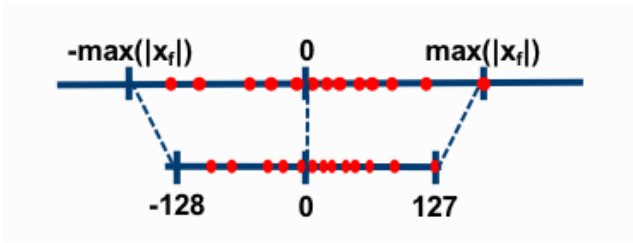


其中 r 为量化前的浮点数，q 为量化后的整数，round表示取整，clip为截断，s 为数据量化的间隔 z 为数据偏移，为0的时候为对称量化，不为0的时候为不对称量化。**对称量化可以避免量化算子在推**

理中计算z相关的部分，降低推理时的计算复杂度；非对称量化可以根据实际数据的分布确定最小值和最小值，可以更加充分的利用量化数据信息，使得量化导致的损失更低



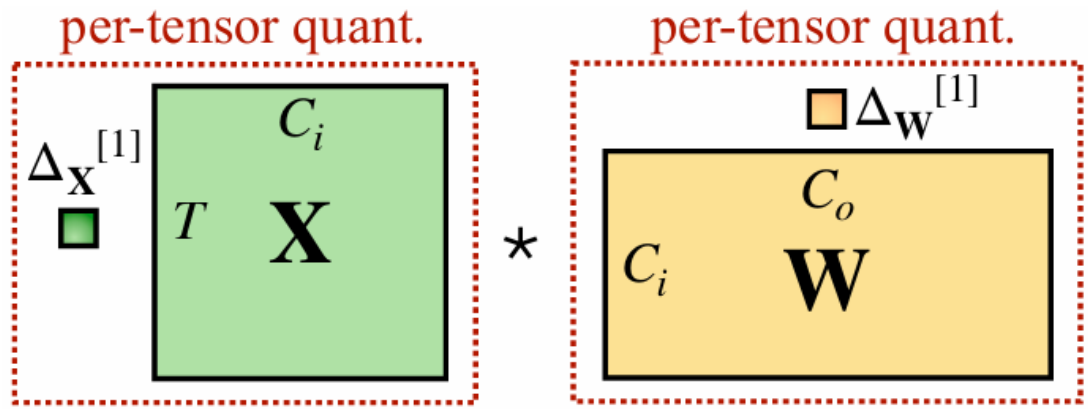
非对称量化



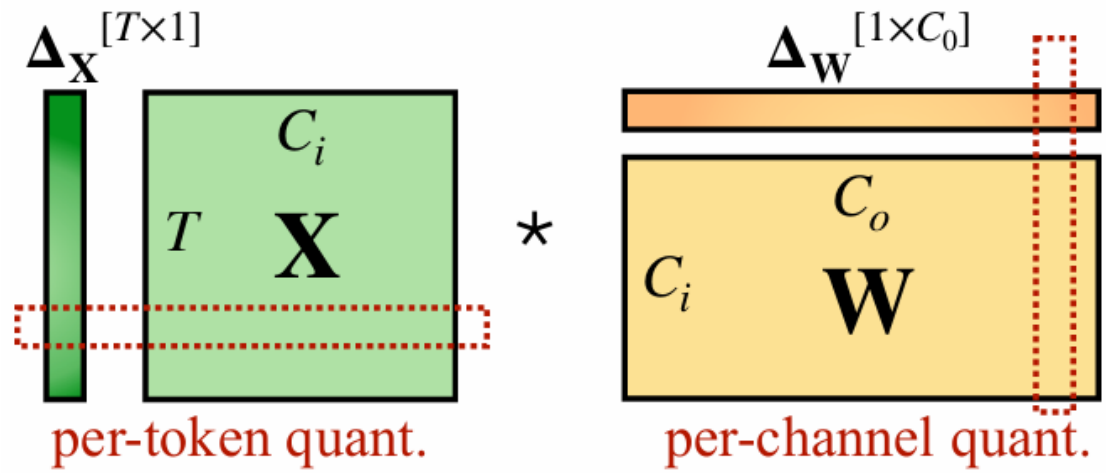
对称量化

根据 s 和 z 的共享范围即量化粒度，量化方法可以进行以下分类：

- **逐层量化 per-tensor：**范围最大，最简单，**以一层网络为单位一组量化参数**



- **逐通道量化 per-token & per-channel：**以一层网络的每个量化通道为单位，每个通道单独用一组量化参数。量化粒度更细，更高的量化精度，计算也更复杂。 其中，per-token 针对激活 x 而言，每行对应一个量化系数；per-channel 针对权重 w 而言，每列对应一个量化系数。结合使用也叫 **vector-wise**



- **逐组量化 per-group：**以组为单位，每个组比如K 行激活值或 K 列权重使用一组s和z；它的粒度处于 per-tensor 和 vector-wise之间。当 groupsize=1 时，逐组量化与逐层量化等价；当 groupsize= 卷积核的数量时，逐组量化与逐通道量化等价

- 此外激活值和权重可以选择不同的粒度进行量化，对于激活值来说还有**动态量化**（推理过程中，实时计算激活的量化系数，对激活进行量化）与**静态量化**（在推理前就计算好激活的量化系数，在推理过程中应用）

量化分类：

根据量化压缩模型的阶段还可以分为：

- **量化感知训练 Quantization Aware Training QAT**
- **量化感知微调 Quantization-Aware Fine-tuning QAF**
- **训练后量化 Post Training Quantization PTQ**

7.1.2 QAT 量化感知训练

首先正常预训练模型，然后在模型中插入“**伪量化节点**”，就是对权重和激活先量化，再反量化。这样**引入了量化误差，让模型在训练过程中“感知”到量化操作，在优化training loss 的同时兼顾quantization error**。适用于对模型精度要求较高的场景，其量化目标无缝地集成到模型的训练过程中，使LLM在训练过程中适应低精度表示，增强其处理由量化引起的精度损失的能力

QAT方法：

- **LLM-QAT：**利用预训练模型生成的结果来实现无数据蒸馏

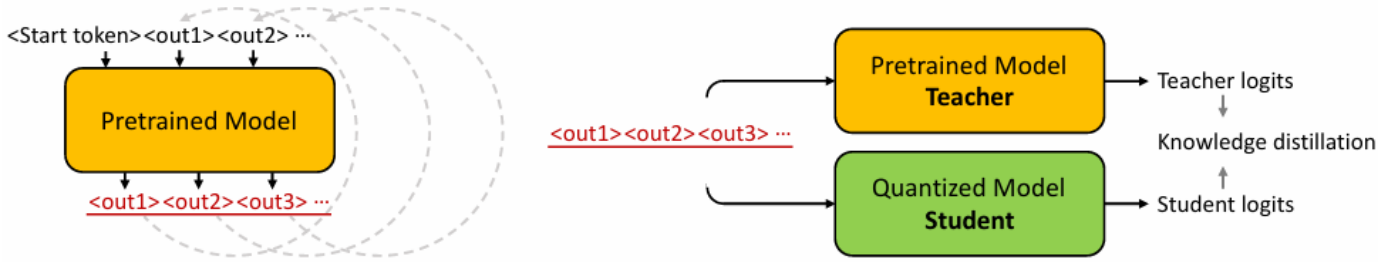


Figure 1: Overview of LLM-QAT. We generate data from the pretrained model with next token generation, which is sampled from top-k candidates. Then we use the generated data as input and the teacher model prediction as label to guide quantized model finetuning.

LLM-QAT不仅**量化了权重和激活**，还**量化了KV缓存**。这个策略旨在增强吞吐量并支持更长的序列。LLM-QAT能够将带有权重和KV缓存量化的LLaMA模型蒸馏为仅有4比特的模型

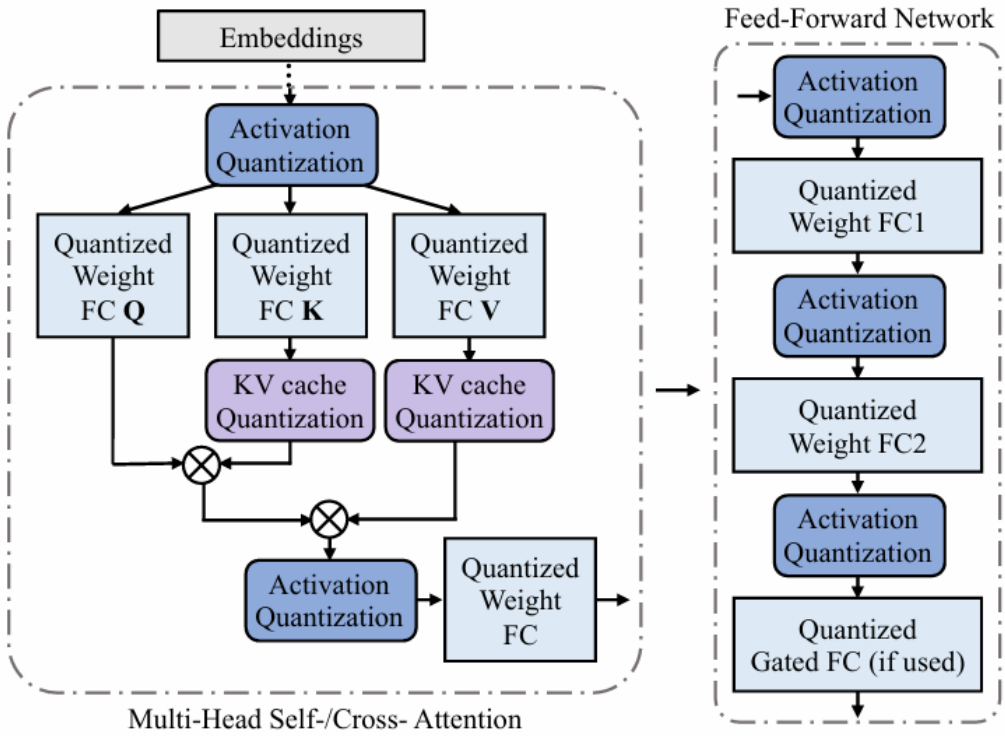


Figure 2: Overview of the quantized transformer in LLM-QAT. We quantize all the weights and input activations in fully-connected linear layers. The KV cache is also quantized if specified.

7.1.3 QAF 量化感知微调

在微调过程中对LLM进行量化，主要目标是确保经过微调的LLM在量化为较低位宽后仍保持性能。通过将量化感知整合到微调中，以在模型压缩和保持性能之间取得平衡

QAF方法：

- **PEQA：** Memory-Efficient Fine-Tuning of Compressed Large Language Models via sub-4-bit Integer Quantization。一种新的量化感知微调技术，可以促进模型压缩并加速推理。采用了双阶段过程运行。在**第一阶段，每个全连接层的参数矩阵被量化为低比特整数矩阵和标量向量**。在**第二阶段，对每个特定下游任务的标量向量进行微调**。这种策略大大压缩了模型的大小，从而降低了部署时的推理延迟并减少了所需的总体内存。 同时使快速的微调和高效的任务切换成为可能

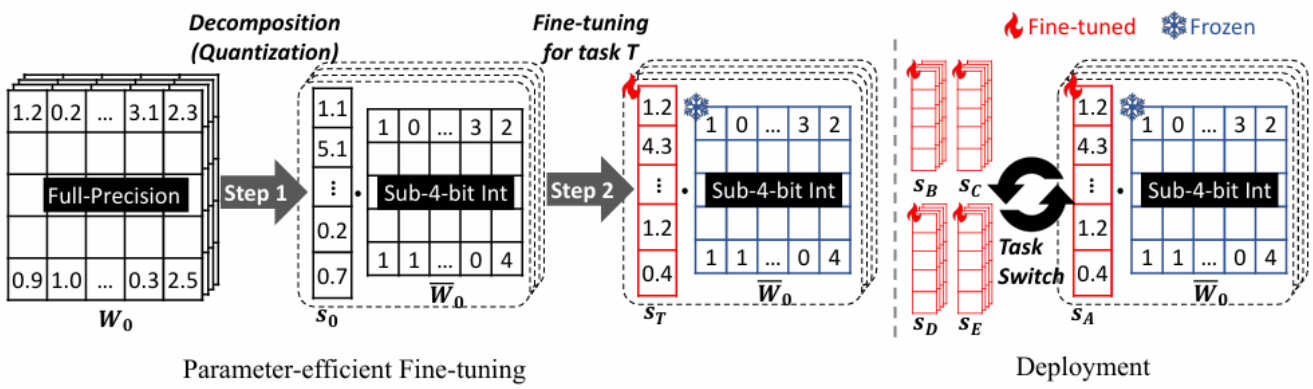


Figure 1: Illustration of our proposed PEQA scheme where $A \cdot B$ indicates the element-wise product of A and B. PEQA is memory-efficient fine-tuning method for quantized large language models that updates only the quantization scale while keeping the integer matrix frozen. Notice a significant reduction in memory footprint when full-precision weights are converted into sub-4-bit integers.

- **QLoRA：**也是一种QAF方法 大语言模型基础原理学习笔记

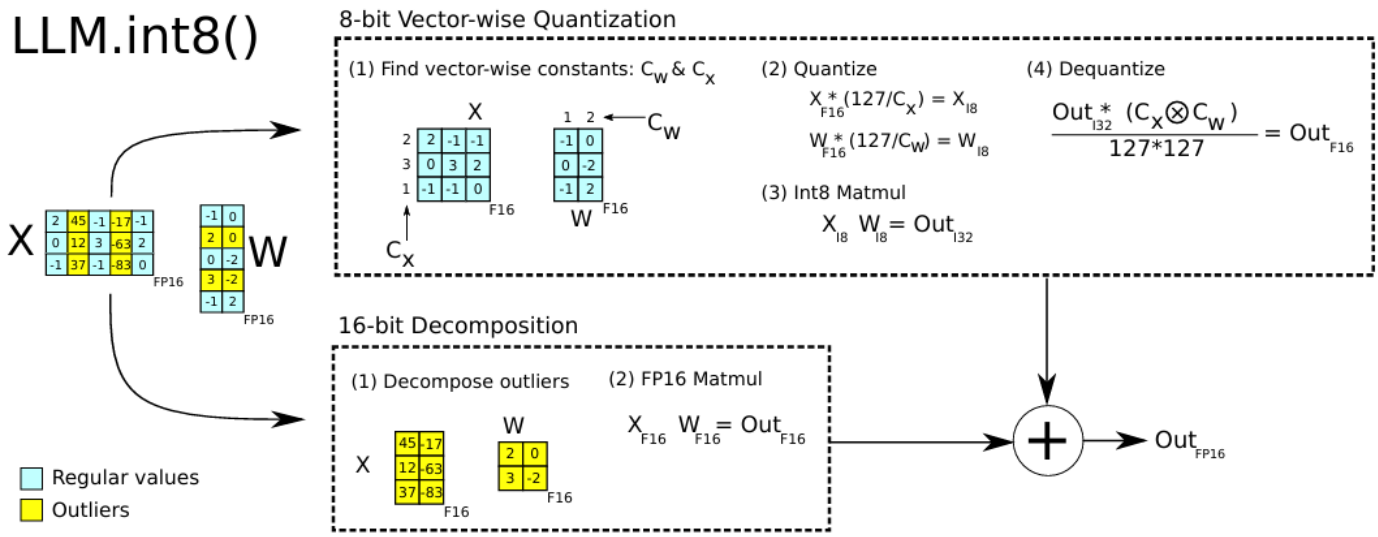
7.1.4 PTQ 训练后量化

QAT 插入“伪量化节点”后微调大大增加了计算成本，尤其是面对超大规模的 LLM。目前针对 LLM 的量化研究都集中在 训练后量化，比如 `LLM.int8()`，`SmoothQuant`，`GPT-Q` 。对于权重而言，可以在推理前事先计算好量化系数，完成量化。但是对于激活（即各层的输入），它们事先是未知的，取决于具体的推理输入

在LLM训练完成后对其参数进行量化，只需要少量校准数据，适用于追求高易用性和缺乏训练资源的场景。主要目标是减少LLM的存储和计算复杂性，而无需对LLM架构进行修改或进行重新训练。**PTQ的主要优势在于其简单性和高效性，但PTQ可能会在量化过程中引入一定程度的精度损失**

PTQ方法：

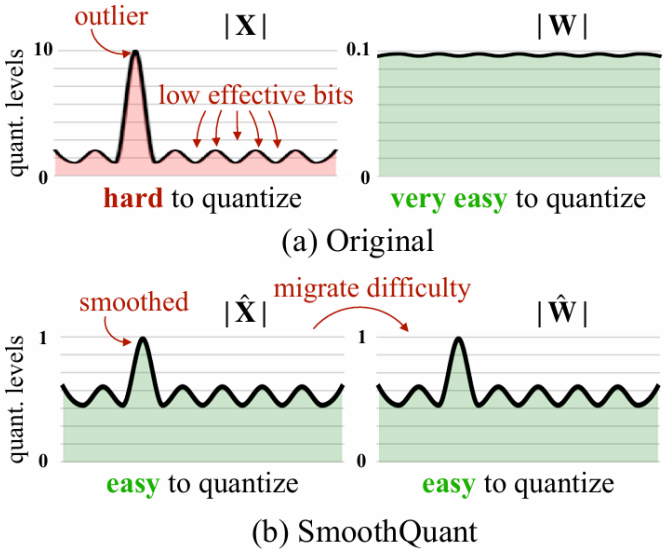
- **LLM.int8()：**激活值X中存在一些离群值，它们的绝对值明显更大；并且这些离群值分布在少量的几个特征 中，称为离群特征。观察下图中黄色的离群值，不论是 per-token 还是 per-channel quantization，都会受到这些离群值的很大影响。LLM.in8() 的思路是，既然只有少量的特征包含离群值，那就把这些特征拿出来单独计算，只对剩余特征做量化，即**采用混合精度分解的量化方法，先做了一个矩阵分解，对绝大部分权重和激活用8比特量化（vector-wise），对离群特征的几个维度保留16bit，对其做高精度的矩阵乘法**



- **SmoothQuant：**针对激活中的离群值，SmoothQuant 给出了与 LLM.int8() 不同的解题思路，激活值的量化比权重的量化难得多，可以通过一个平滑系数，把二者的难度中和一下：

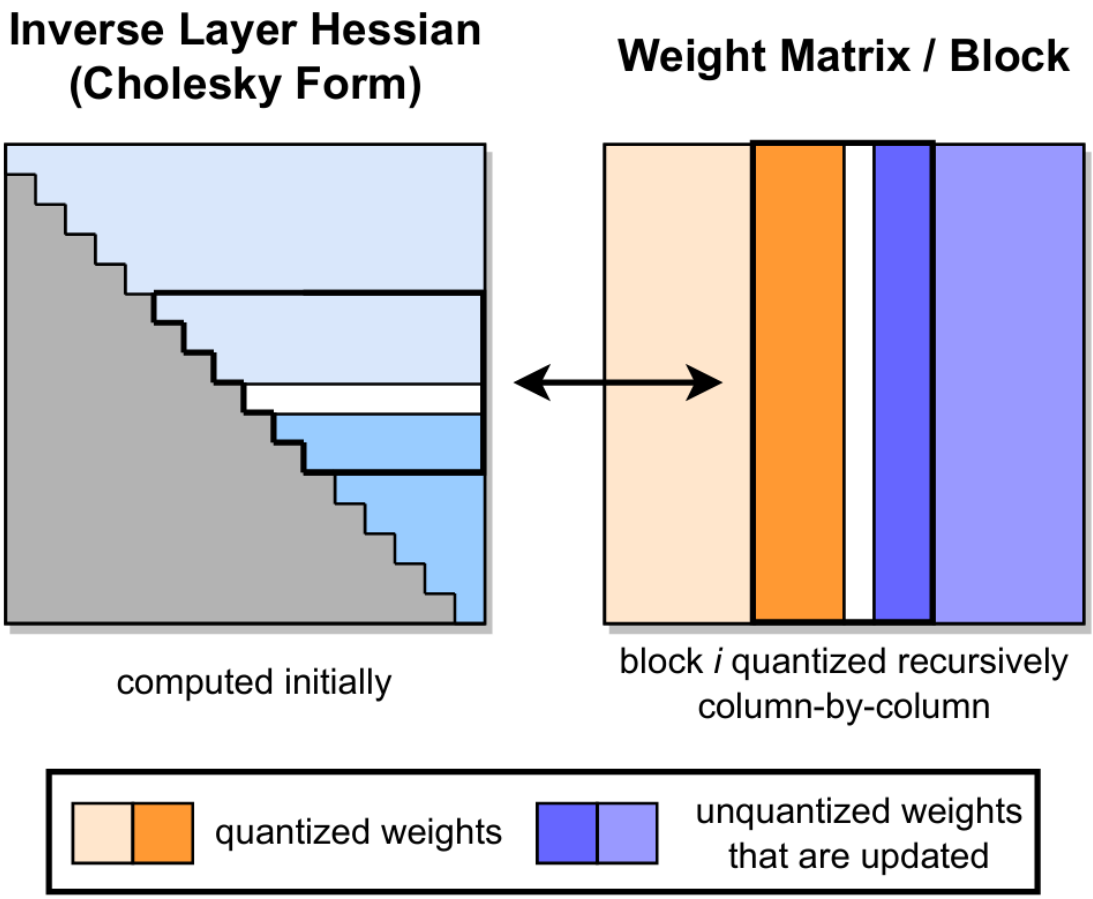
$$Y = (X\mathrm{diag}(s)^{-1}) \cdot (\mathrm{diag}(s)W) = \hat{X}\hat{W}$$
$$s_j = \max(|X_j|)^{\alpha} / \max(|W_j|)^{1-\alpha}$$

对平滑后的激活值和权重进行量化即可，权重采用per-tensor，激活采用不同粒度、不同时机的量化有不同版本：

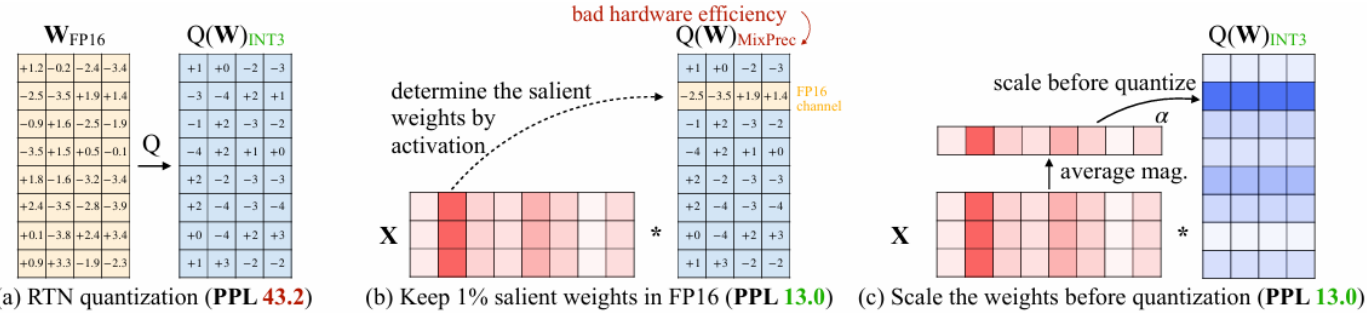


Method	Weight	Activation
W8A8	per-tensor	per-tensor dynamic
ZeroQuant	group-wise	per-token dynamic
`LLM.int8()`	per-channel	per-token dynamic+FP16
Outlier Suppression	per-tensor	per-tensor static
SmoothQuant-O1	per-tensor	per-token dynamic
SmoothQuant-O2	per-tensor	per-tensor dynamic
SmoothQuant-O3	per-tensor	per-tensor static

- **GPT-Q：**LLM.int8() 和 SmoothQuant 都属于 round-to-nearest (RTN) 量化：舍入到最近的定点数，GPT-Q 则是把量化问题视作优化问题，逐层寻找最优的量化权重。**对某个块（block）内的所有参数逐个量化，每个参数量化后，需要适当调整这个块内其他未量化的参数，以弥补量化造成的精度损失。** 此外，GPTQ 量化需要准备校准数据集



- **AWQ：**对于LLM的性能，**权重并不是同等重要的**，通过**保留1%的显著权重可以大大减少量化误差**。在此基础上，**AWQ采用了激活感知方法，考虑与较大激活幅度对应的权重通道的重要性**，这在处理重要特征时起着关键作用。采用**逐通道缩放技术来确定最佳缩放因子**，从而在量化所有权重的同时最小化量化误差



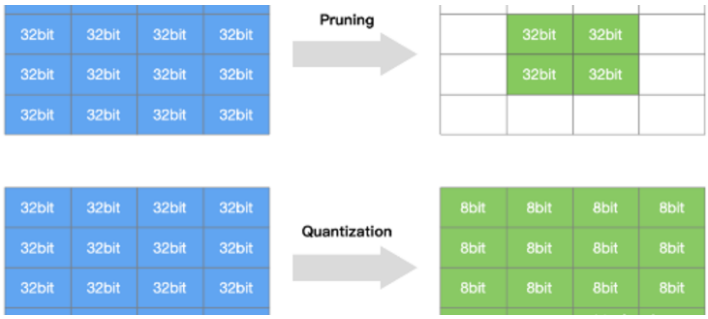
(a) RTN quantization (**PPL 43.2**) (b) Keep 1% salient weights in FP16 (**PPL 13.0**) (c) Scale the weights before quantization (**PPL 13.0**)

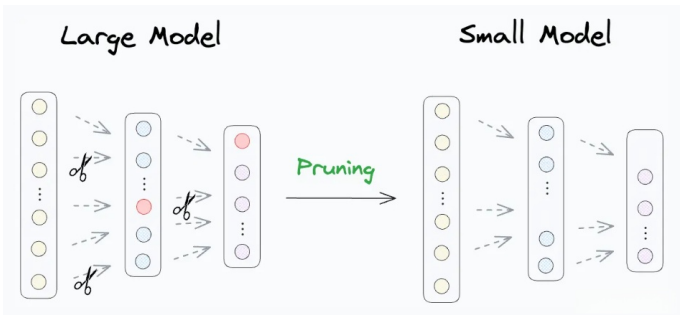
Figure 2. We observe that we can find 1% of the salient weights in LLMs based on the *activation distribution* (middle). Keeping the salient weights in FP16 can significantly improve the quantized performance (PPL from 43.2 (left) to 13.0 (middle)), but the mixed-precision format is not hardware-efficient. We follow the activation-awareness principle and propose AWQ (right). AWQ performs per-channel scaling to protect the salient weights and reduce quantization error. We measure the perplexity of OPT-6.7B under INT3-g128 quantization.

7.2 Pruning 模型剪枝

7.2.1 剪枝简介

- 模型量化是指通过减少权重表示或激活所需的比特数来压缩模型
- **模型剪枝研究模型权重中的冗余，并尝试删除/修剪冗余和非关键的权重，减少模型大小和计算复杂度**

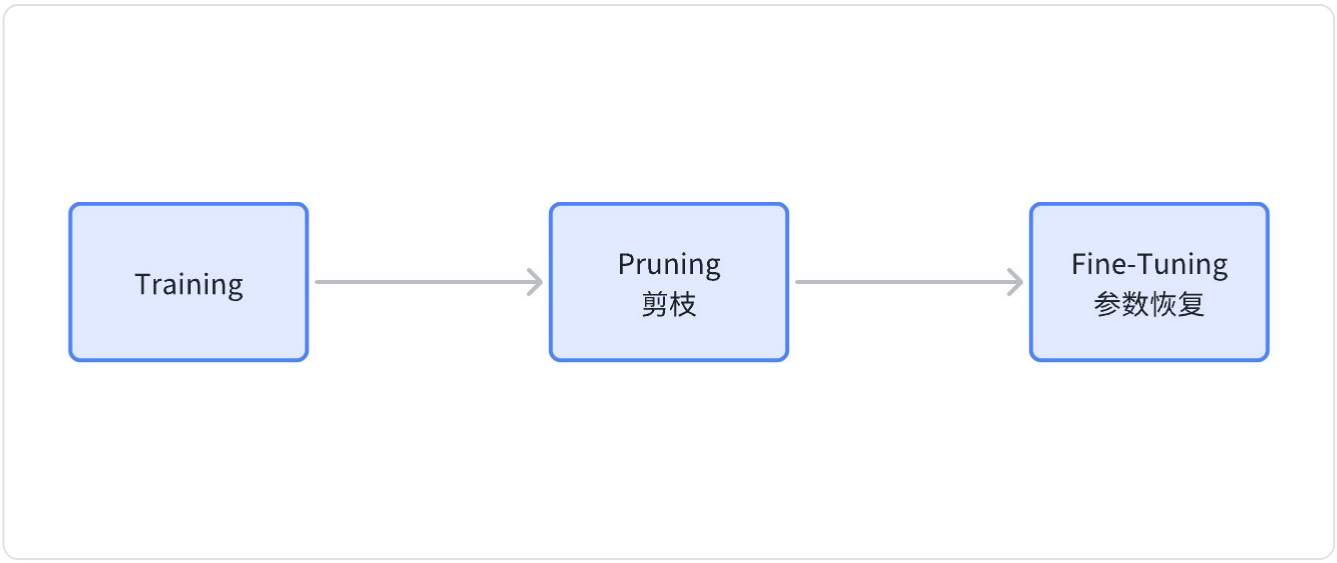




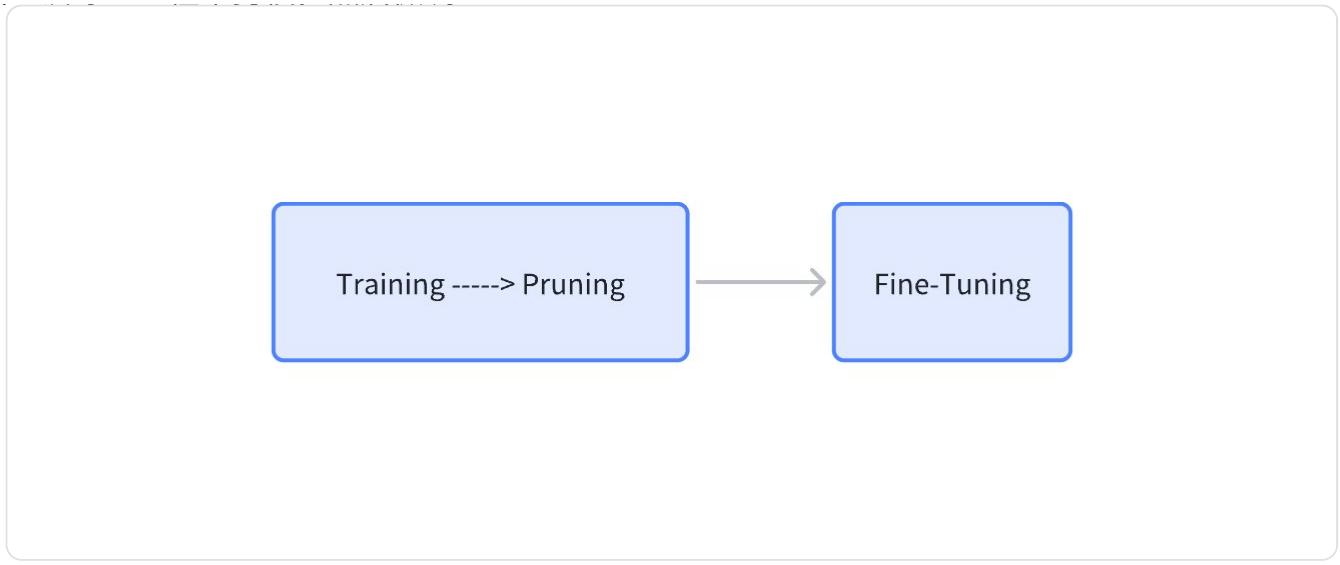
7.2.2 剪枝流程

模型剪枝大的流程大致有三种：

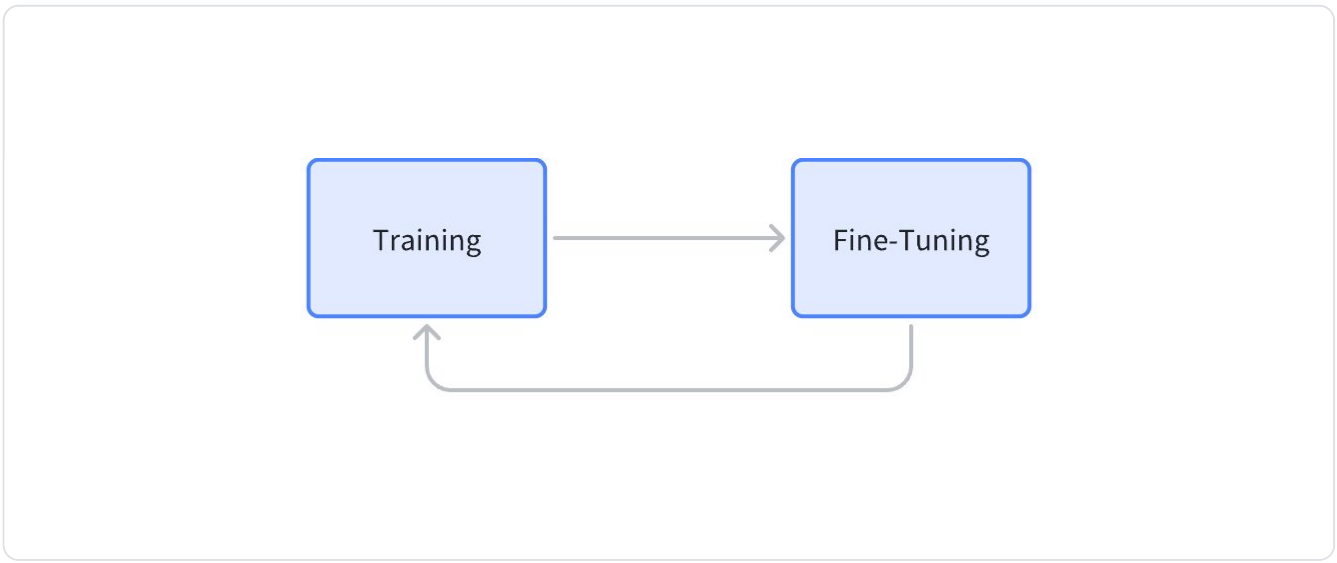
- 先训练模型，然后剪枝，最后微调



- 模型训练的过程中剪枝，然后微调



- 进行剪枝，然后从头训练剪枝的模型

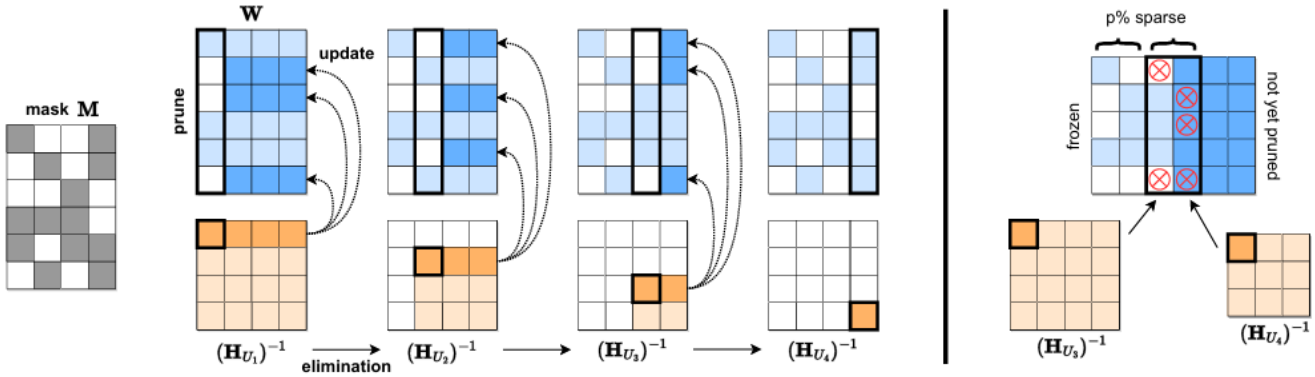


7.2.3 剪枝分类

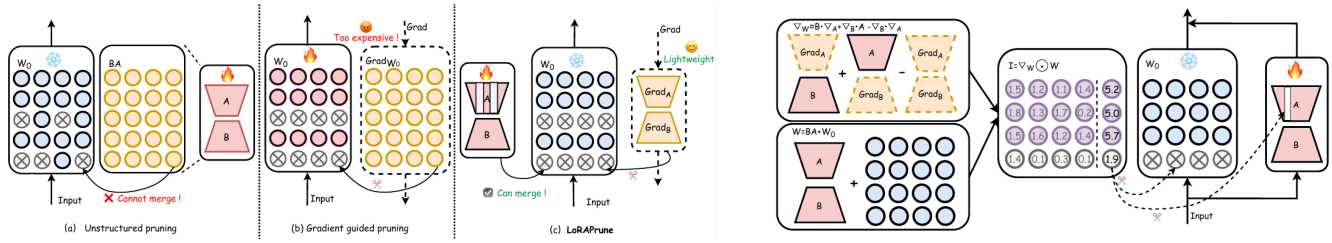
- **非结构化剪枝：随机对独立权重或者神经元链接进行剪枝，指移除个别参数，而不考虑整体网络结构**。这种方法通过将低于阈值的参数置零的方式对个别权重或神经元进行处理。它会导致特定的参数被移除，模型出现不规则的稀疏结构。并且这种不规则性需要专门的压缩技术来存储和计算被剪枝的模型。此外，非结构化剪枝通常需要对LLM进行大量的再训练以恢复准确性，这对于LLM来说尤其昂贵
  - **优点：**剪枝算法简单，模型压缩比高
  - **缺点：**精度不可控，剪枝后权重矩阵稀疏，没有专用硬件难以实现压缩和加速的效果

  非结构化剪枝方法：

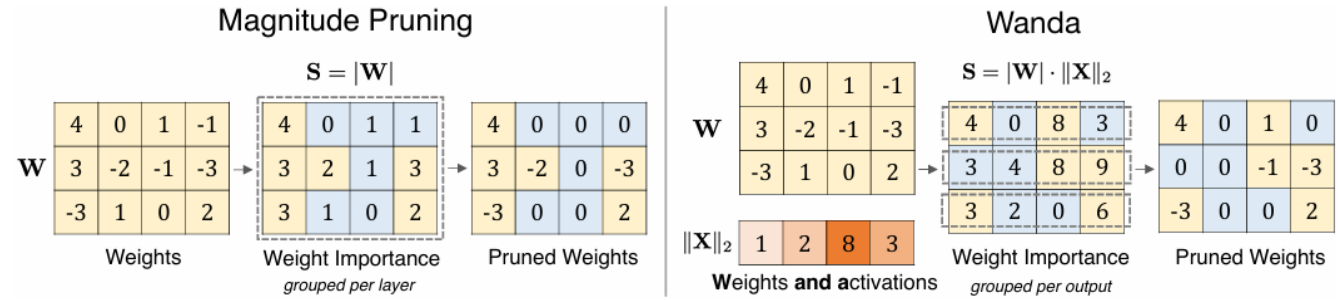
  - **SparseGPT：**引入了**一次性剪枝策略，无需重新训练**。该方法**将剪枝视为一个广泛的稀疏回归问题，并使用近似稀疏回归求解器来解决，实现了显著的非结构化稀疏性**



- **LoRAPrune：**将参数高效微调（PEFT）方法与剪枝相结合，以提高下游任务的性能。引入了一种独特的参数重要性标准，使用了来自LoRA的值和梯度



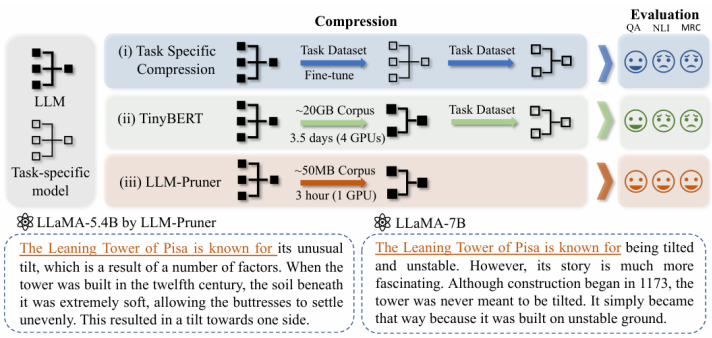
- **Wanda：**提出一种新的剪枝度量，**根据每个权重的大小以及相应输入激活的范数的乘积进行评估，这个乘积是通过使用一个小型校准数据集来近似计算的。这个度量用于线性层输出内的局部比较，使得可以从LLM中移除优先级较低的权重**



- **结构化剪枝：**根据**预定义规则移除连接或分层结构，同时保持整体网络结构**。这种方法一次性地针对整组权重，优势在于降低模型复杂性和内存使用，同时保持整体的LLM结构完整

  结构化剪枝方法：

  - **LLM-Pruner：**采用了一种多功能的方法来压缩LLMs，同时保护它们的多任务解决和语言生成能力。它引入了一个依赖检测算法，以定位模型内部的相互依赖结构。它还实施了一种高效的重要性估计方法，考虑了一阶信息和近似Hessian信息



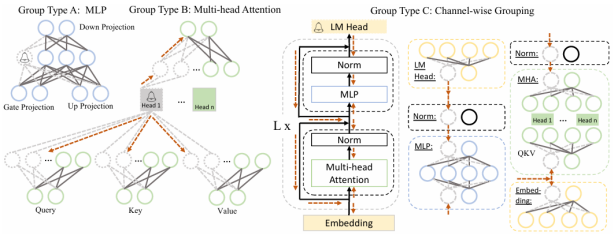


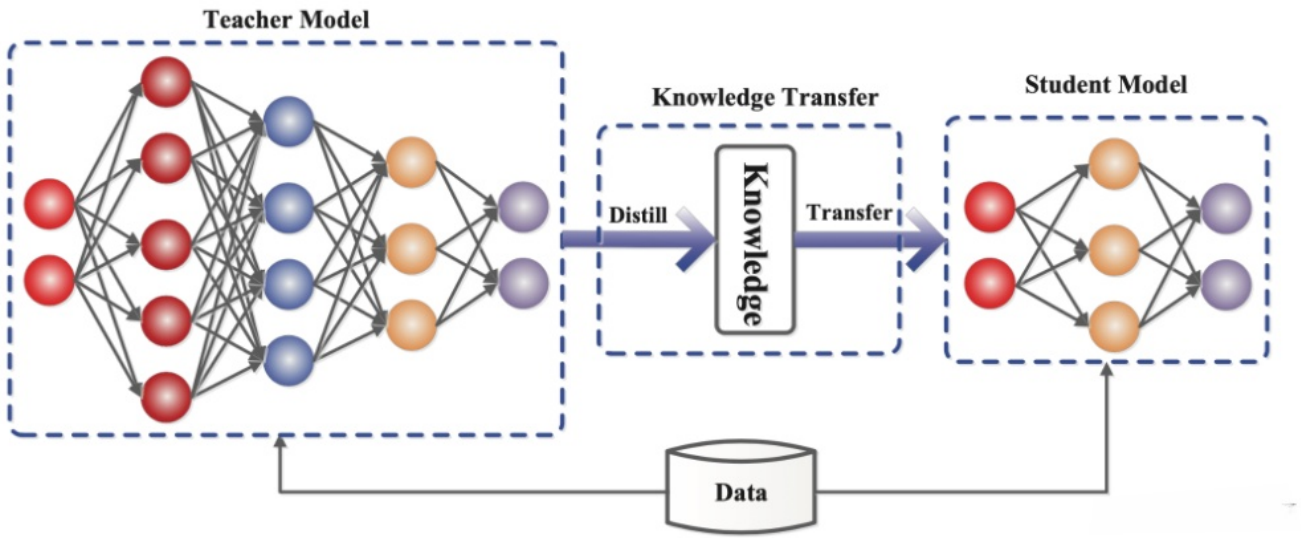
Figure 2: Illustration of the coupled structures in LLaMA. We simplify the neurons in each layer to make the dependent group clear. The trigger neuron, marked as a circle with a bell, cause weights with dependency pruned (dashed lines), which may propagate (red dashed lines) to coupled neurons (dashed circles). A group can be triggered by a variety of trigger neurons. Taking Group Type B as an example, the trigger for this group involves (i) the attention head, (ii) the output neuron in Query, Key or Value, and (iii) the input neuron in the final output projection.

7.3 Knowledge Distillation 知识蒸馏

7.3.1 知识蒸馏简介

知识蒸馏（KD）是一种旨在将知识从大型复杂模型（即教师模型）转移到较小简单模型（即学生模型）的技术。可以分为：**Blackbox知识蒸馏**，通常**只能访问教师模型的输出，这些教师模型一般来自闭源大语言模型**；以及**Whitebox知识蒸馏**，在这种情况下**可以获取教师模型的参数或输出分布，教师模型通常来自开源大语言模型**

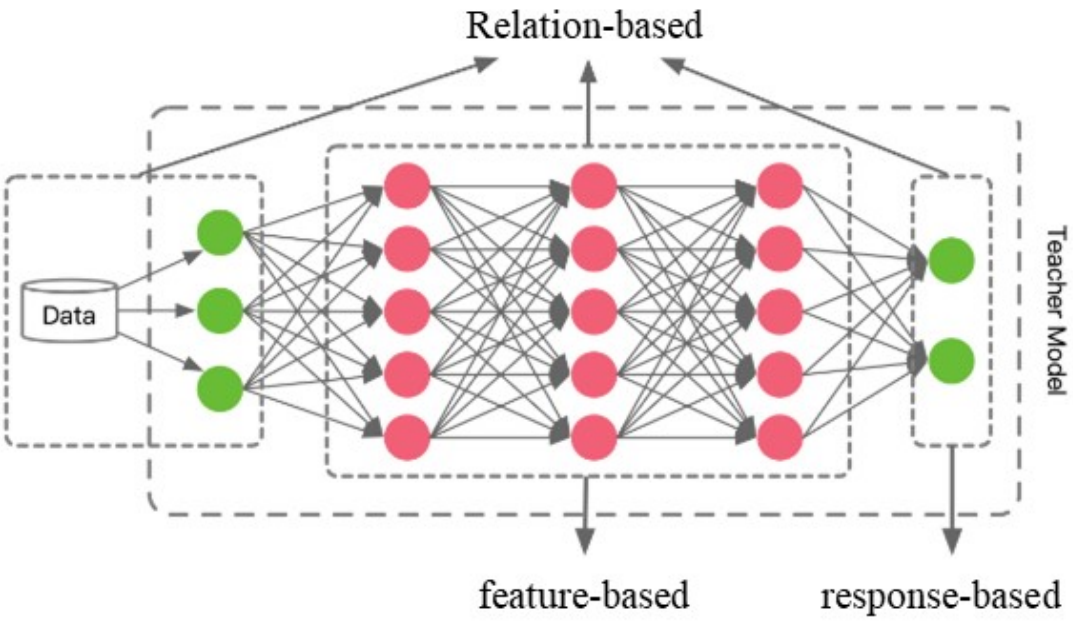
知识蒸馏算法由三部分组成，分别是**知识**（Knowledge）、**蒸馏算法**（Distillation algorithm）、**师生架构**（Teacher-student architecture），一般的师生架构如下图所示：



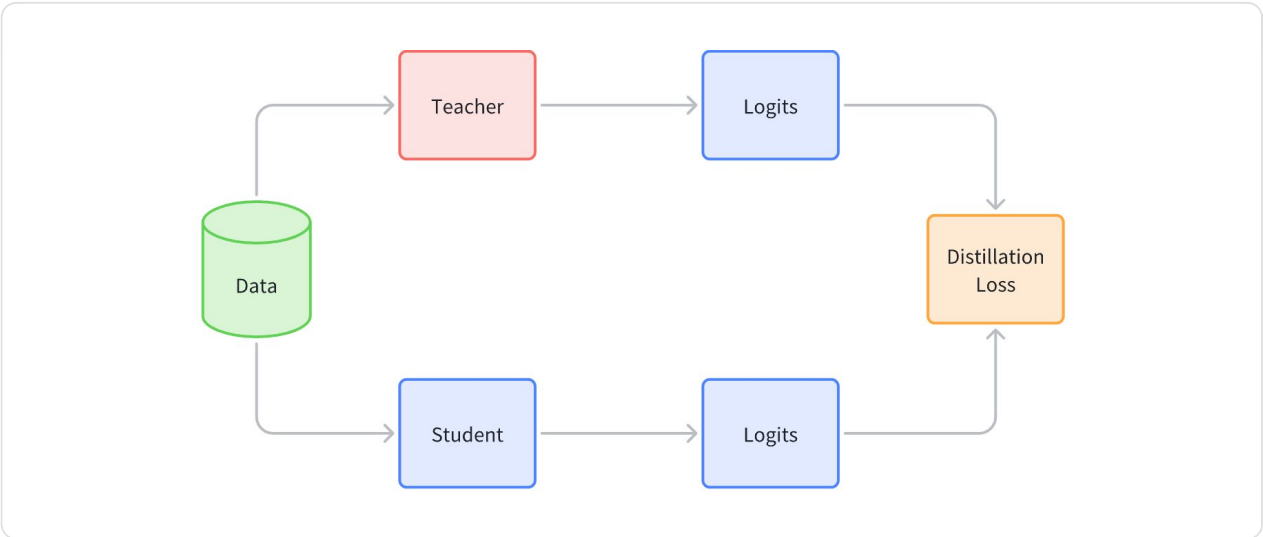
通常，教师网络会比学生网络大，通过知识蒸馏的方法将教师网络的知识转移到学生网络，因此，蒸馏学习可以用于压缩模型，将大模型变成小模型。另外，知识蒸馏的过程需要数据集，这个数据集可以是用于教师模型预训练的数据集，也可以是额外的数据集

7.3.2 白盒知识蒸馏

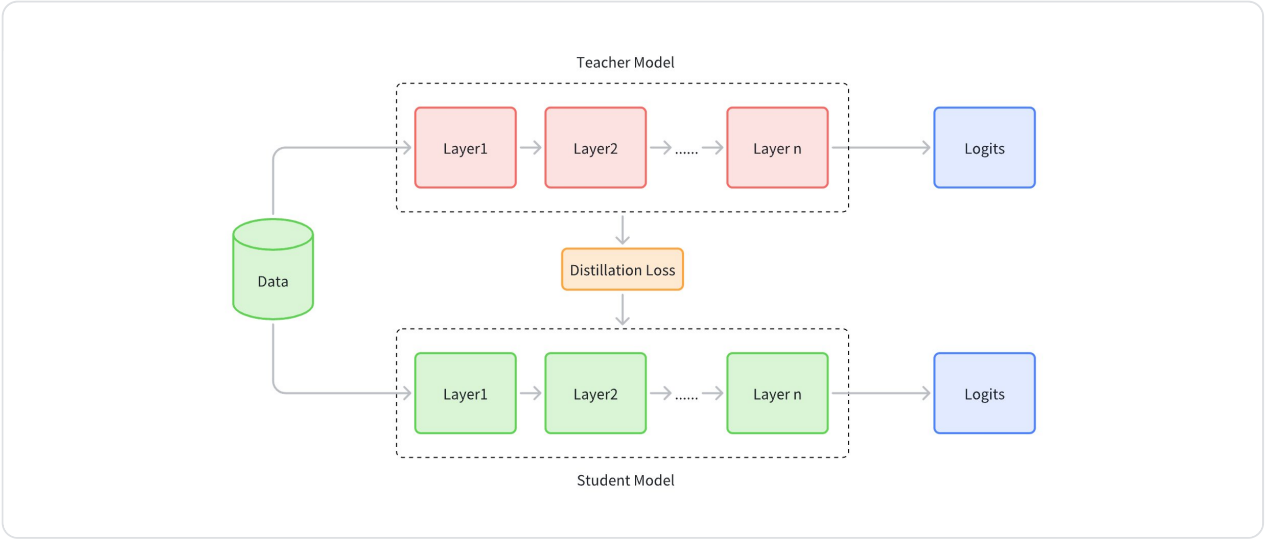
知识的类型：主要有 Response-based、Feature-based、Relation-based 三种



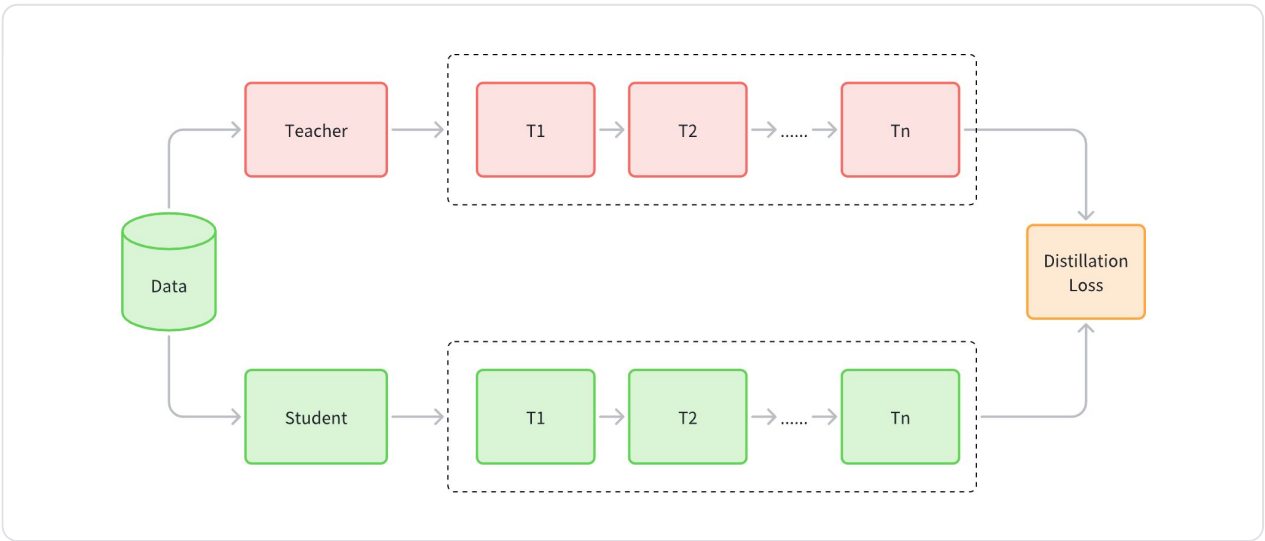
- **Response-based：**当知识蒸馏对这部分知识进行转移时，学生模型直接学习**教师模型输出层的特征**。通俗的说法就是老师充分学习知识后，直接将结论告诉学生



- **Feature-based：**上面一种方法学习目标非常直接，学生模型直接学习教师模型的最后预测结果。考虑到深度神经网络善于学习不同层级的特征，**教师模型的中间层的特征激活也可以作为学生模型的学习目标，对 Response-based knowledge 形成补充**。虽然基于特征的知识转移为学生模型的学习提供了更多信息，但由于学生模型和教师模型的结构不一定相同，如何从教师模型中选择哪一层特征激活（提示层），从学生模型中选择哪一层（引导层）模仿教师模型的特征激活，是一个需要探究的问题。另外，当提示层和引导层大小存在差异时，如何正确匹配教师与学生的特征表示也需要进一步探究，目前还没有成熟的方案



- **Relation-based：**上述两种方法都使用了教师模型中特定网络层中特征的输出，而基于关系的知识进一步探索了各网络层输出之间的关系或样本之间的关系。例如将教师模型中两层 feature maps 之间的 Gram 矩阵（网络层输出之间的关系）作为知识，或者将样本在教师模型上的特征表示的概率分布（样本之间的关系）作为知识



蒸馏的方法：一般分为三种offline distillation，online distillation和self-distillation

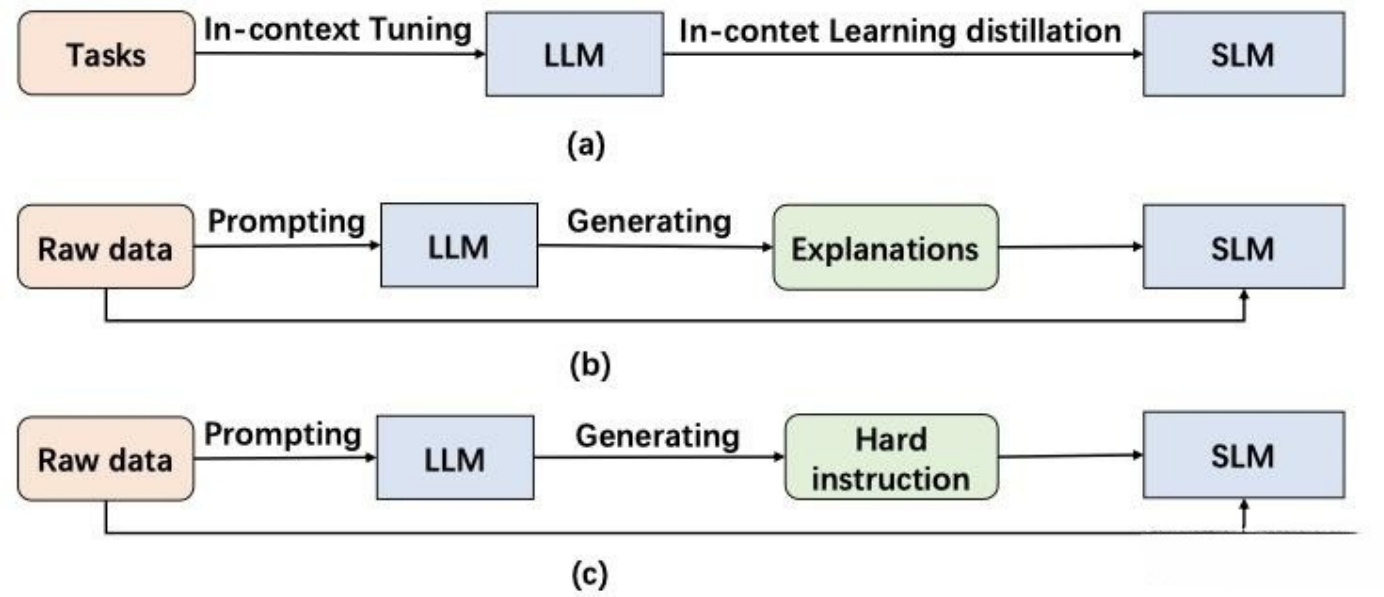
- **Offline distillation：**大部分知识蒸馏算法采用的方法，主要包含两个过程：1）蒸馏前**教师模型预训练** 2）**蒸馏算法迁移知识**。主要侧重于知识迁移部分，教师模型通常参数量大，训练时间比较长，一些大模型会通过这种方式得到小模型，比如 BERT 通过蒸馏学习得到 tinyBERT
- **Online distillation：**要求**教师模型和学生模型同时更新**，主要针对参数量大、精度性能好的教师模型不可获得情况。而现有的方法往往难以获得在线环境下参数量大、精度性能好的教师模型
- **Self distillation：** online distillation 的一种特例，教师模型和学生模型采用相同的网络模型

7.3.3 黑盒知识蒸馏

白盒知识蒸馏也叫标准KD，旨在让学生学习LLM所用的常见知识，如输出分布和特征信息，相比之下，黑盒知识蒸馏通常促使教师大语言模型生成一个蒸馏数据集，用于微调学生语言模型，从而将能力从教师大语言模型转移到学生语言模型。特别的，对于LLM来说，**知识转移还涵盖了蒸馏他们独特**

的涌现能力，所以又叫 Emergent Abilities-based KD，主要是ICL、CoT和Instruction Following三种蒸馏

- **ICL：**采用结构化的自然语言prompt，包含任务描述和一些示例。通过这些任务示例，LLM可以在不需要显式梯度更新的情况下掌握和执行新任务。**ICL蒸馏将LLM的上下文少样本学习和语言建模能力转移到学生模型中，将上下文学习目标与传统的语言建模目标相结合**，并在少样本学习范式下进行ICL蒸馏
- **CoT：**CoT与ICL相比，在**提示中加入了中间推理步骤，这些步骤可以影响最终的输出，而不是使用简单的输入-输出对**。一些基于CoT的蒸馏工作：MT-COT利用LLMs产生的解释来增强较小的推理器的训练，利用多任务学习框架，使较小的模型具备强大的推理能力以及生成解释的能力；Fine-tune-CoT通过随机采样从LLMs生成多个推理解决方案，这种训练数据的增加有助于学生模型的学习过程；SOCRATIC CoT训练了两个蒸馏模型：问题分解器和子问题解决器，分解器将原始问题分解为一系列子问题，而子问题解决器则处理解决这些子问题；DISCO 引入了一种基于LLMs的完全自动反事实知识蒸馏方法，它通过工程化的提示生成短语扰动，然后通过任务特定的教师模型将这些扰动数据过滤，以提取高质量的反事实数据；SCOTT采用对比解码，将每个原理与答案联系起来，鼓励从教师那里提取相关的原理
- **IF：仅依赖于任务描述而不依赖于少量示例**。通过**使用一系列以指令形式表达的任务进行微调，语言模型展现出能够准确执行以前未见过的指令描述的任务的能力**。利用LLMs的适应性来提升学生模型的性能，促使LLM识别和生成“困难”指令，然后利用这些指令来增强学生模型的能力



7.4 Low-Rank Factorization 低秩分解

低秩分解旨在通过将给定的权重矩阵分解成两个或多个较小维度的矩阵，从而对其进行近似。低秩分解背后的核心思想是找到一个大的权重矩阵W的分解，得到两个矩阵U和V，使得W≈U V，其中U是一个m×k矩阵，V是一个k×n矩阵，其中k远小于m和n。U和V的乘积近似于原始的权重矩阵，从而大幅减少了参数数量和计算开销

在LLM研究的模型压缩领域，研究人员通常将多种技术与低秩分解相结合，包括修剪、量化等，例如 **LoRAPrune** 和ZeroQuant-FP，以实现更有效的压缩同时保持性能。随着这一领域的研究继续进行，可能会在将低秩分解应用于压缩LLM方面出现进一步的发展，但仍需要持续的探索和实验来充分发挥其在LLM方面的潜力

8. Updating 持续更新内容

8.1 Paper Reading 论文阅读笔记

子页面列表 [该内容暂不支持预览]

8.2 Transformer量化分析计算

主要分析Decoder-only架构的

模型参数量计算

- **self-attention块**

  QKV 以及输出 O 的四个权重矩阵及bias形状为 (h,h)和$(h,)$，总参数量为 $4h^2+4h$

- **MLP块**

  第一个线性层权重和bias $(h,4h)$和$(4h,)$

  第二个线性层权重和bias $(4h,h)$和$(h,)$

  总参数量 $8h^2+5h$

- **LN部分**

  两个LayerNorm，每个都有两个参数 γ,β 都是 $(h,)$，总参数量 $4h$

- **Embedding层**

  词表大小 V，形状 (V,h)，总参数量 Vh

- **位置编码**

  绝对位置编码 lh,l 为最大序列长度

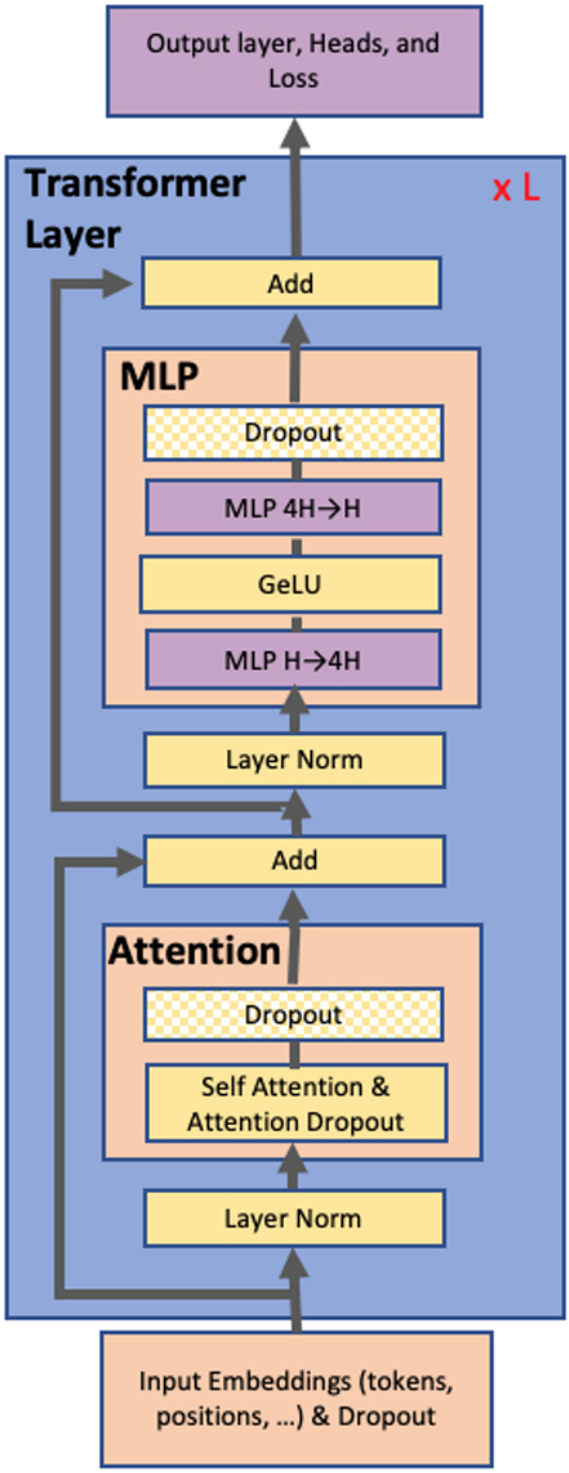
  RoPE、ALiBi等相对位置编码不引入额外可训练参数

- **Output层**

  同Embedding层，(h,V)，参数 Vh

综合计算参数量 L 层transformer参数量：
$L \times \left[(4h^2 + 4h) + (8h^2 + 5h) + 4h\right] + 2Vh$
$= L(12h^2 + 13h) + 2Vh \approx 12Lh^2$

LLaMA-32B	hidden_size	L层数	$12Lh^2$
32.5B	6656	60	31,897,681



前向计算量FLOPs计算

FLOPs表示浮点数运算次数， 对于大小为 $(M, N), (N, L)$ 的矩阵 A, B 来说计算矩阵乘法需要进行 MNL 次乘法和加法运算，所以FLOPs为 $2MNL$ 。输入数据形状 $(batch_size, seq_length) = (b, s)$

- **Embedding层**

  不涉及FLOPs，gather操作，根据ID取token embedding， $(b, s) \to (b, s, h)$

- **self-attention层**
  - QKV 三个映射矩阵计算为 $(b, s, h) \times (h, h)$ ，FLOPs $3 \times 2bsh^2$

- QK^T 计算之前会做split heads操作，所以是 $(b, num_heads, s, h/num_heads) \times (b, num_heads, h/num_heads, s) \rightarrow (b, num_heads, s, s)$ ，总的FLOPs为 $2bs^2h$
- $attention_score \cdot V$ 计算为 $(b, num_heads, s, s) \times (b, num_heads, s, h/num_heads) \rightarrow (b, num_heads, s, h/num_heads)$ ，所以FLOPs为 $2bs^2h$
- Output线性映射之前会对多头进行融合，所以计算是 $(b, s, h) \times (h, h) \rightarrow (b, s, h)$ ，FLOPs为 $2bsh^2$

- **MLP层**
  - MLP $h \rightarrow 4h$ 乘法 $(b, s, h) \times (h, 4h) \rightarrow (b, s, 4h)$ ，FLOPs为 $2 \times 4bsh^2$
  - MLP $4h \rightarrow h$ 乘法 $(b, s, 4h) \times (4h, h) \rightarrow (b, s, h)$ ，FLOPs为 $2 \times 4bsh^2$
- **Output层**

  输出logits从hidden_size转词表大小 $(b, s, h) \times (h, V) \rightarrow (b, s, V)$ ，FLOPS为 $2bshV$

L层的总共FLOPs为 $L * (24bsh^2 + 4bs^2h) + 2bshV$

训练推理显存计算

- **训练阶段：**

  主要先分析模型参数、梯度和优化器状态。LLM训练多使用混合精度训练和AdamW优化器，FP16的模型参数量为 ϕ ，那么FP16的梯度为 ϕ ，AdamW优化器状态中有FP32的一阶动量和FP32的二阶动量一共 2ϕ ，训练的时候优化器更新模型参数的时候还需要FP32的模型参数 ϕ ，FP32的梯度 ϕ ，按照FP16占2字节，FP32占4字节来算，训练阶段的显存占用总共有：$2 \times 2\phi + 4 \times 4\phi = 20\phi$ 字节。这里与第六章的 16ϕ 不太一样，因为额外的FP32的梯度其实不是必须的，不同的实现当中不一样
- **推理阶段：**

  推理阶段没有梯度、优化器状态，不用保存中间激活，占用显存大头为模型参数，还有KVCache也会占用显存

激活值计算

假设混合精度训练，中间激活都是FP16的，每个元素占用2个字节， $bsha$ 为**batch_size, seq_length, hidden_size, num_head**

- **self-attention块的激活值**
  - x, Q, K, V, O 五个都需要保存中间激活，占用显存为 $5 \times 2bsh = 10bsh$
- **MLP块的激活值**
  - 第一个线性层保存输入，占用 $2bsh$
  - 激活函数保存输入，占用 $8bsh$

- $softmax$ 需要保存 QK^T ，占用显存为 $2bs^2a$
- $attention_score \cdot V$ 需要保存分数，$2bs^2a$
- 中间会有两个dropout，dropout操作的mask矩阵每个元素只占一个字节，一个在softmax之后，一个在输出映射之后，总共 $bsh + bs^2a$

总共占用显存 $11bsh + 5bs^2a$

- 第二个线性层保存输入，占用 $8bsh$
- 最后dropout，占用 bsh

总共占用 $19bsh$

- **LN的激活值**

LN需要保存输入，两个LN总共 $4bsh$

对于L层的transformer来说，近似中间激活占用 $L \times (34bsh + 5bs^2a)$

KV Cache计算

推断（reference/generate）阶段用KVcache进行加速，分为两个阶段：

- **预填充 prefilling**：prompt序列，同时生成所有transformer层的KVcache
- **解码 decoding**：利用prefilling的KVcache进行自回归生成一个一个token生成，同时将KV更新到KVcache中

FP16精度存储，显存占用峰值为 $b(s+n)h * L * 2(\text{两个}K,V) * 2(FP16)$ ， s, n 为输入序列长度和输出序列长度，K，V的形状就是 $(b, s+n, h)$

8.3 nanoGPT：从头训练一个GPT

- **准备环境**

新建一个conda环境，并安装依赖的包（如果你不会Ananconda环境管理，建议先学一下这个）

```
conda create -n nano python=3.10
conda activate nano
# 安装pytorch，可以去这个网站https://pytorch.org/get-started/previous-versions/挑选合适的版本
pip install pip install torch==2.4.0 torchvision==0.19.0 torchaudio==2.4.0 --index-url https://download.pytorch.org/whl/cu118
# 安装一些补充包
pip install matplotlib
```

- **准备数据**

下载nanoGPT用到的数据集tinyshakespeare：

wget https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt

如果出现如下报错，可以本地再尝试一下，Windows使用wget可以参考 https://zhuanlan.zhihu.com/p/28826000

```
--2024-12-13 15:53:11--  https://raw.githubusercontent.com/karpathy/char-rnn/master/data/tinyshakespeare/input.txt
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 0.0.0.0, ::
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|0.0.0.0|:443... failed: Connection refused.
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|::|:443... failed: Connection refused.
```

其实这个数据集很小，可以直接打开上面的链接复制到新建txt文件里面😎

- **导入相关包和超参设置**

```python
import torch
import torch.nn as nn
from torch.nn import functional as F
import matplotlib.pyplot as plt

## 超参设置
batch_size = 16 # 批处理大小，并行训练多少个独立序列
block_size = 32 # 推理的最大上下文长度
max_iters = 5000 # 最大训练迭代次数
eval_interval = 100 # 多少次迭代评估一次
learning_rate = 1e-3 # 学习率
device = 'cuda' if torch.cuda.is_available() else 'cpu'# 是否使用GPU
eval_iters = 200 # 评估模型时的迭代次数
n_embd = 64 # embedding后的向量维度
n_head = 4 # 多头注意力的头数
n_layer = 4 # transformer层数
dropout = 0.0 # dropout率
torch.manual_seed(42) # 随机种子
```

- **处理数据**

```python
# 读取文件
with open('input.txt', 'r', encoding='utf-8') as f:
    text = f.read()
print("数据集中的总字符数: ", len(text))
# l看一下前300个字符
print("前300个字符: ", text[:300])
# 文本中出现的所有的字符
chars = sorted(list(set(text)))
vocab_size = len(chars)
print("文本中出现的字符: ", ''.join(chars))
```

```python
print("文本中出现的字符数量: ", vocab_size)
# 简单的tokenizer实现
# 通过列表推导式创建 string to integer(stoi)和 integer to string(itos)的映射字典
# 这是一个简单的tokenizer实现，就是简单的把文本中出现过的所有字符按顺序给予一个整数 0~vocab_size-1
# 这样就把字符映射为网络可以处理的数字
stoi = { ch:i for i,ch in enumerate(chars) }
itos = { i:ch for i,ch in enumerate(chars) }
# 简单的编码和解码规则
# 编码：给定一个字符串输出一个list的整数
encode = lambda s: [stoi[c] for c in s]
# 解码：给定一个list的整数输出一个字符串
decode = lambda l: ''.join([itos[i] for i in l])
print("编码结果: ", encode("Hello World!"))
print("解码结果: ", decode(encode("Hello World!")))
```

```
数据集中的总字符数:  1115393
前300个字符:  First Citizen:
Before we proceed any further, hear me speak.

All:
Speak, speak.

First Citizen:
You are all resolved rather to die than to famish?

All:
Resolved. resolved.

First Citizen:
First, you know Caius Marcius is chief enemy to the people.

All:
We know't, we know't.

First Citizen:
Let us

文本中出现的字符:
 !$&',-.3:;?ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz
文本中出现的字符数量:  65
编码结果:  [20, 43, 50, 50, 53, 1, 35, 53, 56, 50, 42, 2]
解码结果:  Hello World!
```

- **数据集划分及分批**

```
## 数据集划分以及数据分批
# torch.long是长整形
data = torch.tensor(encode(text), dtype=torch.long)
# 90%划分训练集和验证集
n = int(0.9*len(data))
train_data = data[:n]
val_data = data[n:]
def get_batch(split):
    # 生成一个batch的数据，x为输入，y是target
    data = train_data if split == 'train' else val_data
    ix = torch.randint(len(data) - block_size, (batch_size,))
    x = torch.stack([data[i:i+block_size] for i in ix])
    y = torch.stack([data[i+1:i+block_size+1] for i in ix])
    # y相对于x移位一个token，因为训练的目标就是预测下一个token
    x, y = x.to(device), y.to(device)
    return x, y
```

- **模型评估**

```
## 模型评估
# 评估模型在训练集和验证集上的平均损失，评估eval_iters次
@torch.no_grad() # 仅评估
def estimate_loss():
    out = {}
    model.eval() # 模型评估模式
    for split in ['train', 'val']:
        losses = torch.zeros(eval_iters)
        for k in range(eval_iters):
            X, Y = get_batch(split)
            logits, loss = model(X, Y)
            losses[k] = loss.item()
        out[split] = losses.mean()
    model.train() # 模型训练模式
    return out
```

- **模型定义**
  - **多头注意力模型中的单个头**

```
# 多头注意力模型中的单个头
class Head(nn.Module):
```

```
    """ one head of self-attention """
    def __init__(self, head_size):
        super().__init__()
        # QKV映射矩阵
        self.key = nn.Linear(n_embd, head_size, bias=False)
        self.query = nn.Linear(n_embd, head_size, bias=False)
        self.value = nn.Linear(n_embd, head_size, bias=False)
        # 创建一个下三角矩阵，并将其注册为模型的一个缓冲区
        # 用于计算masked self-attention时作为mask
        self.register_buffer('tril', torch.tril(torch.ones(block_size, block_size)))
        self.dropout = nn.Dropout(dropout)
    def forward(self, x):
        # batch_size, block_size, embed_dim
        B,T,C = x.shape
        k = self.key(x)   # (B,T,C)
        q = self.query(x) # (B,T,C)
        v = self.value(x) # (B,T,C)
        # 计算attention分数，1/sqrt(d_k) -> C**-0.5
        wei = q @ k.transpose(-2,-1) * C**-0.5 # (B, T, C) @ (B, C, T) -> (B, T, T)
        # 利用下三角矩阵对attention进行mask填充，保证当前的token只能看到它之前的token，看不到后面的
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf')) # (B, T, T)
        # softmax归一化权重
        wei = F.softmax(wei, dim=-1) # (B, T, T)
        wei = self.dropout(wei)
        # 得到的权重再和value相乘
        out = wei @ v # (B, T, T) @ (B, T, C) -> (B, T, C)
        return out
```

- **多头注意力、FFN和Block**

```
# 多头注意力
# 实现多头自注意力机制，并行计算自注意力，然后将结果拼接起来
class MultiHeadAttention(nn.Module):
    """ multiple heads of self-attention in parallel """
    def __init__(self, num_heads, head_size):
        super().__init__()
        self.heads = nn.ModuleList([Head(head_size) for _ in range(num_heads)])
        # 输出映射线性层
        self.proj = nn.Linear(n_embd, n_embd)
        self.dropout = nn.Dropout(dropout)
```

```
    def forward(self, x):
        # 对每个注意力头h进行计算，然后将结果在最后一个维度上拼接起来，得到out
        out = torch.cat([h(x) for h in self.heads], dim=-1)
        out = self.dropout(self.proj(out))
        return out
# 模型FeedFoward
class FeedFoward(nn.Module):
    """ a simple linear layer followed by a non-linearity """
    def __init__(self, n_embd):
        super().__init__()
        # 两个线性层和一个ReLU激活函数，以及一个dropout层
        # 中间升维 embed_dim->4*embed_dim，然后降维回去
        self.net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd),
            nn.ReLU(),
            nn.Linear(4 * n_embd, n_embd),
            nn.Dropout(dropout),
        )
    def forward(self, x):
        return self.net(x)
# 模型Block定义
# 实现一个Transformer Block
# 包含layerNorm, multiheadattention, layerNorm, feedforward
class Block(nn.Module):
    """ Transformer block: communication followed by computation """
    def __init__(self, n_embd, n_head):
        # n_embd: embedding dimension, n_head: the number of heads we'd like
        super().__init__()
        head_size = n_embd // n_head # 每一个头的思哲
        self.sa = MultiHeadAttention(n_head, head_size)
        self.ffwd = FeedFoward(n_embd)
        self.ln1 = nn.LayerNorm(n_embd)
        self.ln2 = nn.LayerNorm(n_embd)
    # Transformer结构
    # pre-norm->self-attention->残差链接->pre-norm->FFN->残差链接
    def forward(self, x):
        x = x + self.sa(self.ln1(x))
        x = x + self.ffwd(self.ln2(x))
        return x
```

- **二元语言模型**

```
# 定义一个二元语言模型BigramLanguageModel
# 训练一个二元语言模型，并生成新的文本
class BigramLanguageModel(nn.Module):
```

```
    def __init__(self):
        super().__init__()
        # embedding表和位置嵌入表
        self.token_embedding_table = nn.Embedding(vocab_size, n_embd)
        self.position_embedding_table = nn.Embedding(block_size, n_embd)
        # 堆叠多个Transformer Block，总共有n_layer个Transformer Block
        self.blocks = nn.Sequential(*[Block(n_embd, n_head=n_head) for _ in range(n_layer)])
        self.ln_f = nn.LayerNorm(n_embd) # 最后一个layerNorm
        # 输出线性层，将嵌入向量映射到词汇表大小的向量
        self.lm_head = nn.Linear(n_embd, vocab_size)
    def forward(self, idx, targets=None):
        # batch_size, block_size
        B, T = idx.shape
        # 首先，从词嵌入表和位置嵌入表中获取嵌入，将它们相加得到x
        tok_emb = self.token_embedding_table(idx) # (B,T,C)
        pos_emb = self.position_embedding_table(torch.arange(T, device=device)) # (T,C)
        x = tok_emb + pos_emb # (B,T,C)
        # 然后，x输入到Transformer Block中，得到输出x
        x = self.blocks(x) # (B,T,C)
        # 最后，将输出x输入到输出线性层中，得到logits
        x = self.ln_f(x) # (B,T,C)
        logits = self.lm_head(x) # (B,T,vocab_size)
        if targets is None:
            loss = None
        else:
            B, T, C = logits.shape
            logits = logits.view(B*T, C)
            targets = targets.view(B*T)
            # 计算交叉熵损失
            loss = F.cross_entropy(logits, targets)
        return logits, loss
    def generate(self, idx, max_new_tokens):
        # 生成新的文本，从当前的索引idx开始，每次生成一个新的token，直到生成max_new_tokens个token为止
        # idx is (B, T) array of indices in the current context
        for _ in range(max_new_tokens):
            # 将当前的索引裁剪到最后的block_size个token
            idx_cond = idx[:, -block_size:]
            # 获取预测的logits，只关注最后一个时间步
            logits, loss = self(idx_cond)
            logits = logits[:, -1, :] # becomes (B, C)
            # 将logits应用softmax得到概率
            probs = F.softmax(logits, dim=-1) # (B, C)
            # 从分布中采样一个next token
            idx_next = torch.multinomial(probs, num_samples=1) # (B, 1)
```

```
            # 将采样的索引添加到当前的序列中
            idx = torch.cat((idx, idx_next), dim=1) # (B, T+1)
        return idx
```

- **模型训练**

```
## 模型训练
# 实例化
model = BigramLanguageModel()
m = model.to(device) # 移动到GPU上
# 打印模型参数数量
print(sum(p.numel() for p in m.parameters())/1e6, 'M parameters')
# 训练模型
# 创建一个AdamW优化器，学习率为learning_rate，将模型的参数传递给优化器
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)
# 初始化列表来存储损失值
train_losses = []
val_losses = []
for iter in range(max_iters):
    # 每过100次迭代，就评估模型的损失，打印训练损失和验证损失
    if iter % eval_interval == 0 or iter == max_iters - 1:
        losses = estimate_loss()
        print(f"step {iter}: train loss {losses['train']:.4f}, val loss {losses['val']:.4f}")
        # 将损失值添加到列表中
        train_losses.append(losses['train'])
        val_losses.append(losses['val'])
    # 采样一个batch的数据
    xb, yb = get_batch('train')
    # 计算损失
    logits, loss = model(xb, yb)
    # 将梯度清零，PyTorch默认会累积梯度，所以需要手动清零
    optimizer.zero_grad(set_to_none=True)
    # 计算梯度
    loss.backward()
    # 更新优化器的参数
    optimizer.step()
# 绘制损失曲线图
plt.figure(figsize=(10, 5))
plt.plot(train_losses, label='Train Loss')
plt.plot(val_losses, label='Validation Loss')
plt.xlabel('Iteration')
plt.ylabel('Loss')
plt.title('Training and Validation Loss')
plt.legend()
```

```python
# 保存图片
plt.savefig('loss_plot.png')
# 生成新的文本
context = torch.zeros((1, 1), dtype=torch.long, device=device)
print(decode(m.generate(context, max_new_tokens=2000)[0].tolist()))
```

- **实验结果**

首先是模型参数，可以看到上面实现的这个小型GPT-2参数量总共有0.209729M，按照前面介绍过的近似算法 $12Lh^2 = 12 \times 4 \times 64^2 = 196608 = 0.196608M$ ，近似差不多一致

```
0.209729 M parameters
step 0: train loss 4.4396, val loss 4.4404
step 100: train loss 2.6756, val loss 2.6985
step 200: train loss 2.4997, val loss 2.5223
step 300: train loss 2.4155, val loss 2.4224
step 400: train loss 2.3425, val loss 2.3461
step 500: train loss 2.2834, val loss 2.3088
step 600: train loss 2.2488, val loss 2.2666
step 700: train loss 2.2189, val loss 2.2303
step 800: train loss 2.1678, val loss 2.1918
step 900: train loss 2.1298, val loss 2.1589
step 1000: train loss 2.0934, val loss 2.1199
step 1100: train loss 2.0768, val loss 2.1118
step 1200: train loss 2.0413, val loss 2.0829
step 1300: train loss 2.0160, val loss 2.0673
step 1400: train loss 1.9889, val loss 2.0372
step 1500: train loss 1.9678, val loss 2.0205
step 1600: train loss 1.9489, val loss 2.0107
step 1700: train loss 1.9333, val loss 2.0013
step 1800: train loss 1.9130, val loss 1.9823
step 1900: train loss 1.9012, val loss 1.9704
step 2000: train loss 1.8823, val loss 1.9746
step 2100: train loss 1.8730, val loss 1.9648
step 2200: train loss 1.8561, val loss 1.9588
step 2300: train loss 1.8385, val loss 1.9417
step 2400: train loss 1.8260, val loss 1.9368
step 2500: train loss 1.8165, val loss 1.9341
step 2600: train loss 1.8038, val loss 1.9238
step 2700: train loss 1.8053, val loss 1.9110
step 2800: train loss 1.7924, val loss 1.9145
step 2900: train loss 1.7771, val loss 1.9268
step 3000: train loss 1.7647, val loss 1.8941
step 3100: train loss 1.7535, val loss 1.8863
step 3200: train loss 1.7530, val loss 1.8971
```

```
step 3300: train loss 1.7342, val loss 1.8907
step 3400: train loss 1.7429, val loss 1.8870
step 3500: train loss 1.7317, val loss 1.8853
step 3600: train loss 1.7335, val loss 1.8795
step 3700: train loss 1.7152, val loss 1.8584
step 3800: train loss 1.7143, val loss 1.8715
step 3900: train loss 1.7160, val loss 1.8472
step 4000: train loss 1.7080, val loss 1.8487
step 4100: train loss 1.7041, val loss 1.8569
step 4200: train loss 1.6984, val loss 1.8320
step 4300: train loss 1.6860, val loss 1.8352
step 4400: train loss 1.6766, val loss 1.8410
step 4500: train loss 1.6737, val loss 1.8225
step 4600: train loss 1.6727, val loss 1.8243
step 4700: train loss 1.6669, val loss 1.8271
step 4800: train loss 1.6628, val loss 1.8143
step 4900: train loss 1.6598, val loss 1.8113
step 4999: train loss 1.6404, val loss 1.8091
```

生成效果

```
Which Callos thou see thus mody are of your consul your fail An well,
with coundes, thou treather't, of mine devim to formangey in hasterlict.

KING RICHAMPELIT:
No mack not should do win----

Vorticens:
Which my seee
Onvereven to metty's moke mine I have a Boock!-
I know neetly her fair an thoughts thou take.
Upore, bash,--
He'll he hearts of mear now my honour?

ROMERNE:
Frome she reead shound 'win henower
sincens, your have send hone, affected as so taluck! I make as Bose,
The worthat with have senter of Royauble' the lums:
Do acceisy hapt to my, with I say;
As such targeln armity, your such.

COMINIUS:
But have a parking, Richard, to reason
Thinks unruch, and leave head head England;
Truake hand shore hot. I' plead
```

```
That I commy riemen your paster's boanflew-
With I think'd old thee pleemations: thou shreathy, upoer's tive. I did in I convened not sharict heaven theres, he bearty there, foe, who think, my fair
That chare again's thanful's not to my of feear which balt the our have
Mistire mine one your meet, the do kied my spurgh,
Welick, with thou arm holderous from my hearts
Bany fighall Coucius are, too coiste,
Wouse ready which'd not have lick this oody moke books poine hywork trange be a been
diviise and strayh's dright us the keep's!

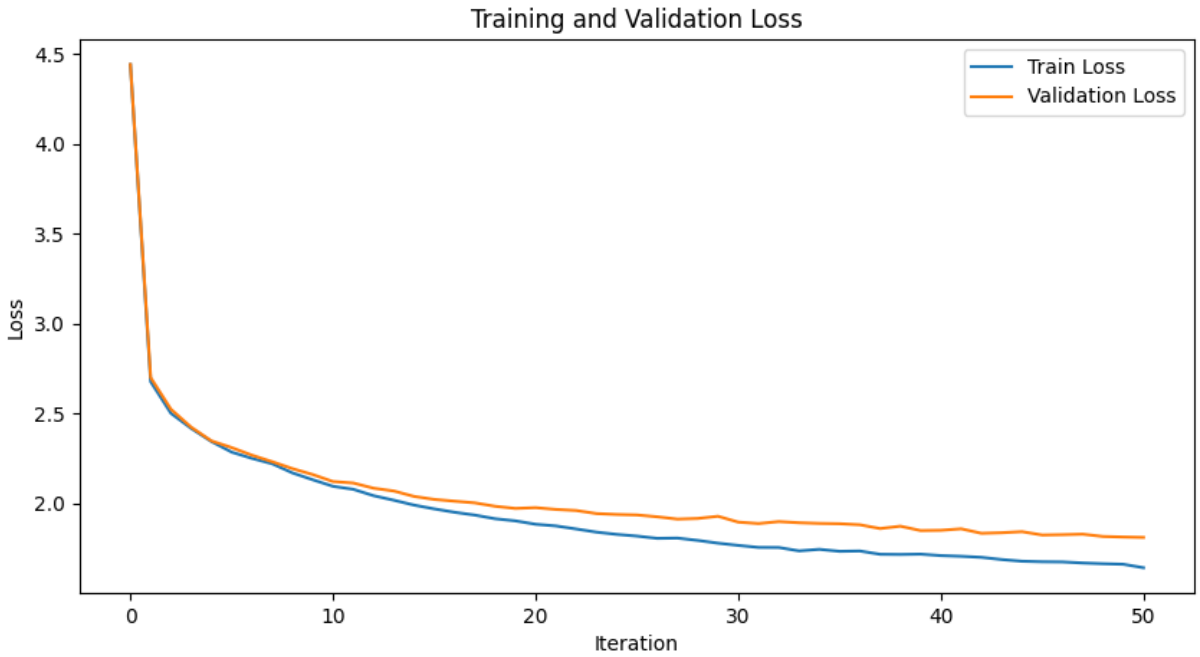
FjUERINCHAM:
Lost, knore: but even, deade was our the both,
How sathe ragge and he be holy and sucke.
Mume 't to shay the hast a baittle
from a trreed his joism sin neever for I am sir to beg.
Lanid you one nor jown, To guiltes to baut,
That you arian's you, to hand and of exiderain's do doubtion Yort how loke than mother's, his matter ever frow's hath in them I pleaceh glack tors;
And wist one low he mother.

KING RICHARD IIII:
That the Lord your prove him, and as honousen,
Te with than Romewing
So off yond our madver our your to child,
'Tis on fould a Strark's hick.

ELIV:
Telmands, Aftienced, to redows with thousandief,
Groon should procien's of Yorse;
I know ailln great all die in weep's hame.

LUCIO:
I will that I shike t
```

loss曲线



训练占用显存

训练时只需要348MB显存，本代码中所用精度全部为FP32，所以每一个元素占用4个字节

```
NVIDIA GeForce RTX 3090 | 63°C, 100 % | 14921 / 24576 MB | [illegible](348M)
```

CUDA上下文初始化的时候占用298MB显存，可以打一个torch.tensor(1).to('cuda')查看

```
NVIDIA GeForce RTX 3090 | 58°C,  54 % |  4969 / 24576 MB | [illegible](298M)
```

剩余为模型参数、梯度、优化器状态、中间激活以及pytorch缓存

```
# pytorch实际使用显存，包含模型参数、梯度、优化器状态、中间激活
torch.cuda.memory_allocated()/1024**2
19.67626953125 MB
# 包含了缓存显存在内
torch.cuda.memory_reserved()/1024**2
34.0 MB
```

这里最好用torch.cuda.memory_allocated()在每一层前后都打印一下，看一下中间激活值到底是多少加深印象

8.4 OpenRLHF微调教程

OpenRLHF 是一个基于 **Ray、DeepSpeed 和 HF Transformers** 构建的高性能 RLHF 框架：

1. **简单易用**: OpenRLHF 是目前可用的最简单的高性能 RLHF 库之一，无缝兼容 Huggingface 模型和数据集

2. **高性能**: RLHF 训练中 80% 的时间用于样本生成阶段。得益于使用 Ray, Packing Samples 以及 vLLM 生成加速的能力，OpenRLHF 的性能是极致优化的 DeepSpeedChat with Hybrid Engine 的 3~4倍以上
3. **分布式 RLHF**: OpenRLHF 使用 Ray 将 Actor、Reward、Reference 和 Critic 模型分布到不同的 GPU 上，同时将 Adam 优化器放在 CPU 上。这使得使用多个 A100 80G GPU 和 vLLM 可以全面微调超过 70B+ 的模型 以及在多个 24GB RTX 4090 GPU 上微调 7B 模型

同时OpenRLHF也支持全量SFT、QLoRA参数高效微调、KTO算法、PRM训练等

- **环境准备**

新建一个conda环境，并按照OpenRLHF的Readme进行操作，自己学习训练的话可以不使用docker，推荐安装 vLLM 0.6.4.post1，然后在安装vllm前，应该先安装pytorch不然会报依赖错误，git clone网络有问题的话可以直接下载zip压缩包。如果你在运行后续训练脚本的时候遇到ninja相关问题，需要apt安装ninja-build

```
conda create -n openrlhf python=3.11
conda activate openrlhf
# 启动 docker container
# docker run --runtime=nvidia -it --rm --shm-size="10g" --cap-add=SYS_ADMIN -v $PWD:/openrlhf nvcr.io/nvidia/pytorch:24.07-py3 bash
# sudo pip uninstall xgboost transformer_engine flash_attn -y

# pip install
pip install openrlhf
pip install pip install torch==2.4.0 torchvision==0.19.0 torchaudio==2.4.0 --index-url https://download.pytorch.org/whl/cu118
# 如果你需要使用 vLLM 加速 (安装 vLLM 0.6.4.post1)
pip install openrlhf[vllm]
# 最新的 vLLM 也是支持的
pip install openrlhf[vllm_latest]
# pip install GitHub 上的最新版
pip install git+https://github.com/OpenRLHF/OpenRLHF.git
# 或者 git clone
git clone https://github.com/OpenRLHF/OpenRLHF.git
cd OpenRLHF
pip install -e .
# 如果报ninja相关错
sudo apt update
sudo apt install ninja-build
```

- **SFT训练脚本**

这里选择了**Qwen2-1.5B作为pretrain model，Alpaca-CoT@firefly作为数据集进行SFT训练**，下面详细解释一下训练脚本中的各个参数，脚本位置在OpenRLHF/examples/scripts/train_sft_llama.sh

```
set -x
read -r -d '' training_commands <<EOF
openrlhf.cli.train_sft \  #SFT训练的py文件
   --max_len 2048 \  #最长序列长度
   --dataset QingyiSi/Alpaca-CoT@firefly \  #数据集name或者path（本地）
   --input_key instruction \  #数据集的输入字段
   --output_key output \  #数据集输出字段
   --train_batch_size 256 \
   --micro_train_batch_size 2 \
   --max_samples 500000 \  #最大训练样本数
   --pretrain Qwen/Qwen2-1.5B \  #pretrain模型name或者path
   --save_path ./checkpoint/qwen2-1.5b-firefly-sft \
   --save_steps -1 \  #多少步保存一个checkpoint，-1表示不保存
   --logging_steps 1 \
   --eval_steps -1 \  #多少步evaluate一下，-1表示最后再evaluate一次
   --zero_stage 2 \  #deepspeed Zero阶段，1，2，3，推荐2，优化器状态和梯度都分到每张卡上
   --max_epochs 1 \  #训练epoch数量
   --bf16 \  #是否用BF16精度训练，默认启用
   --flash_attn \  #是否启用flash attention，默认启用
   --learning_rate 5e-6 \
   --load_checkpoint \  #是否load中间checkpoint
   --gradient_checkpointing \  #是否启用梯度检查点技术
   --use_wandb=[WANDB_TOKENS] \  #wandb的user token
   --wandb_project OpenRLHF \
   --wandb_run_name qwen2-1.5b-firefly-sft
EOF
    # --wandb [WANDB_TOKENS]
    # --packing_samples
# 判断命令第一个是不是slurm，如果不是启用deepspeed命令
if [[ ${1} != "slurm" ]]; then
    deepspeed --module $training_commands
fi
```

此外，还可以在脚本中加入"--num_nodes=1"和"--num_gpus=1"参数来控制训练用到的集群节点数和gpu数量

启动训练脚本之前需要确定你的设备能不能直连Huggingface，如果不能的话请在环境变量中加入镜像站

```
# Linux，最好写到~/.bashrc里面
export HF_ENDPOINT=https://hf-mirror.com
# Windows Power Shell
$env:HF_ENDPOINT = "https://hf-mirror.com"
```

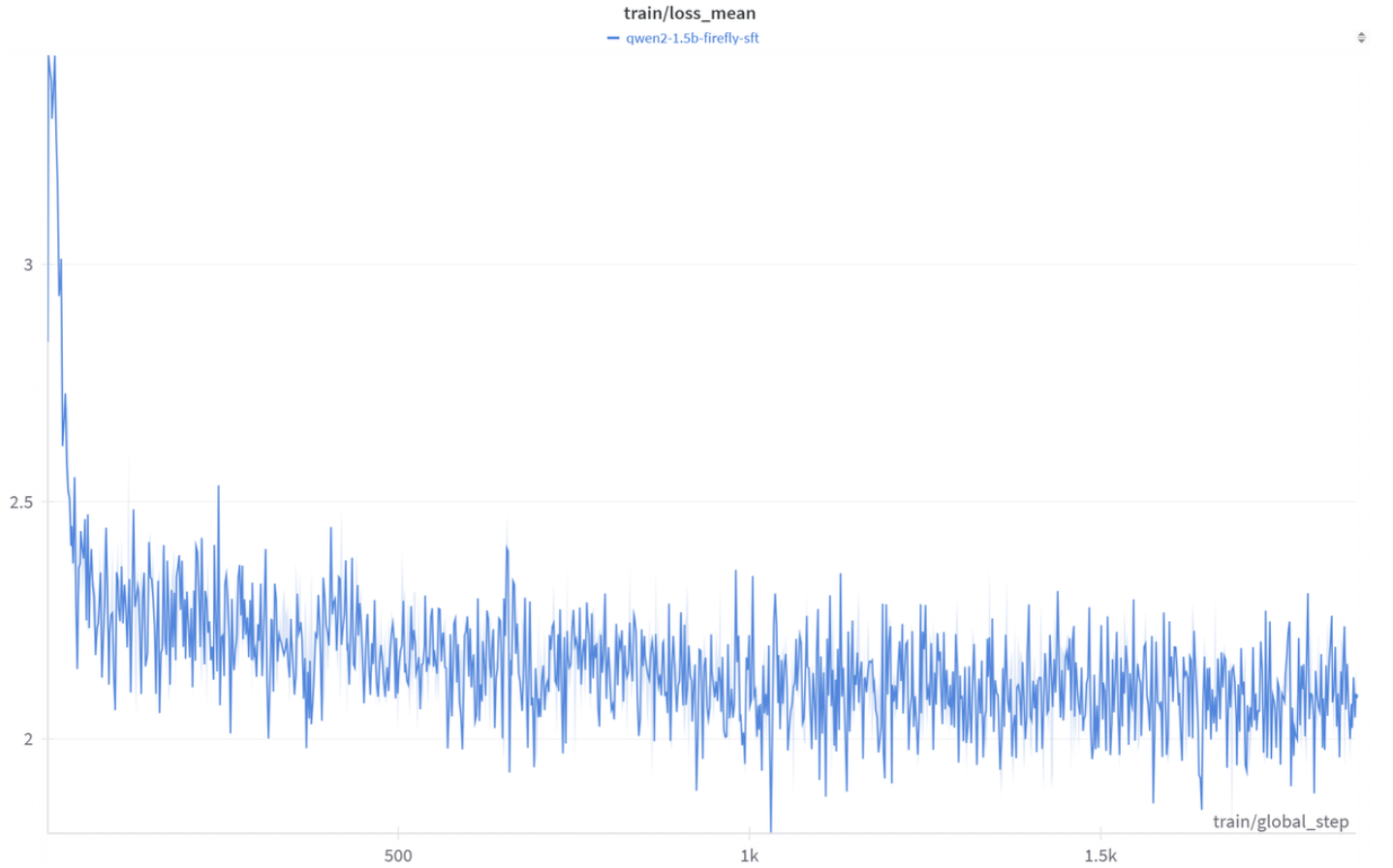
为了节省训练时间，这里建议你提前运行一个加载模型和数据集的脚本，这样模型和数据集会被下载下来并放到你的 ~/.cache/huddingface里面，后面在此运行训练脚本的时候可以直接读取模型和数据集

```
from datasets import load_dataset
# 加载数据集的 firefly 文件夹
dataset = load_dataset("QingyiSi/Alpaca-CoT", data_dir="firefly")
# 打印数据集的一些信息
print(dataset)
print(dataset["train"][0])
from transformers import AutoModelForCausalLM, AutoTokenizer
# 加载模型和tokenizer
model_name = "Qwen/Qwen2-1.5B"
model = AutoModelForCausalLM.from_pretrained(model_name, trust_remote_code=True)
tokenizer = AutoTokenizer.from_pretrained(model_name, trust_remote_code=True)
# 打印模型的一些信息
print(model)
```

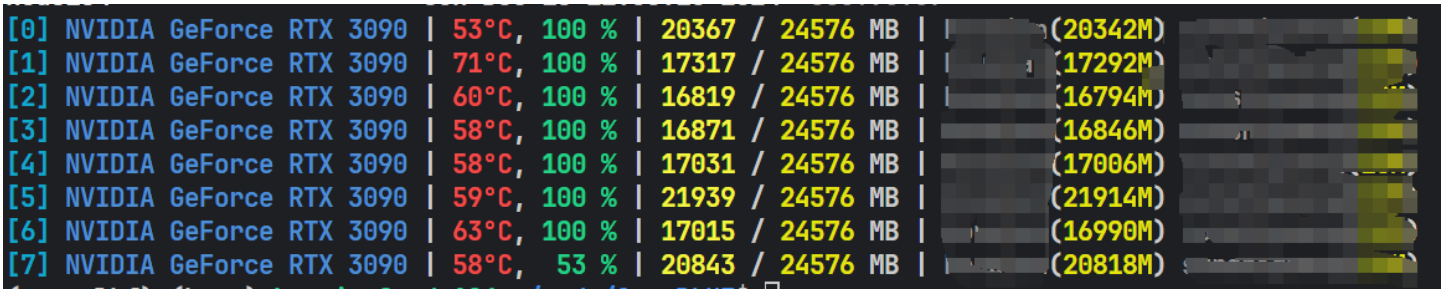
下面就可以 nohup bash examples/scripts/train_sft_qwen1.8b.sh > output.log 2>&1 & 一键开启你的SFT训练了

- **实验结果**

Qwen2-1.5B，八卡3090训练一个epoch，500000samples，256batchsize，24小时，loss_mean曲线，最后收敛在2.1左右，由于我忘了设置save_steps，所以没有保存checkpoint，不能展示训练后的效果



训练期间显存占用



如果资源较少，也可以尝试微调Qwen2-0.5B，一张3090显卡就够用了，下面是根据前面的SFT参数配置只更换模型为Qwen2-0.5B，num_gpus=1的显存占用情况。这是全参数微调的情况，如果开启LoRA显存需求会进一步下降

