常见自然语言处理任务

情感分析（sentiment-analysis）：对给定的文本分析其情感极性

文本生成（text-generation）：根据给定的文本进行生成

命名实体识别（ner）：标记句子中的实体

阅读理解（question-answering）：给定上下文与问题，从上下文中抽取答案

掩码填充（fill-mask）：填充给定文本中的掩码词

文本摘要（summarization）：生成一段长文本的摘要

机器翻译（translation）：将文本翻译成另一种语言

特征提取（feature-extraction）：生成给定文本的张量表示

对话机器人（conversional）：根据用户输入文本，产生回应，与用户对话

自然语言处理的几个阶段

- 第一阶段：统计模型 + 数据（特征工程）
  - 决策树、SVM、HMM、CRF、TF-IDF、BOW
- 第二阶段：神经网络 + 数据
  - Linear、CNN、RNN、GRU、LSTM、Transformer、Word2vec、Glove
- 第三阶段：神经网络 + 预训练模型 + （少量）数据
  - GPT、BERT、RoBERTa、ALBERT、BART、T5
- 第四阶段：神经网络 + 更大的预训练模型 + Prompt
  - ChatGPT、Bloom、LLaMA、Alpaca、Vicuna、MOSS、文心一言、通义千问、星火

Transformers简单介绍

- 官方网址：https://huggingface.co/
- HuggingFace出品，当下最热、最常使用的自然语言处理工具包之一，不夸张的说甚至没有之一
- 实现了大量的基于Transformer架构的主流预训练模型，不局限于自然语言处理模型，还包括图像、音频以及多模态的模型
- 提供了海量的预训练模型与数据集，同时支持用户自行传，社区完善，文档全面，三两行代码便可快速实现模型训练推理，上手简单

一句话总结：学就对了！

Transformers及相关库

- Transformers：核心库，模型加载、模型训练、流水线等
- Tokenizer：分词器，对数据进行预处理，文本到token序列的互相转换
- Datasets：数据集库，提供了数据集的加载、处理等方法
- Evaluate：评估函数，提供各种评价指标的计算函数
- PEFT：高效微调模型的库，提供了几种高效微调的方法，小参数量撬动大模型
- Accelerate：分布式训练，提供了分布式训练解决方案，包括大模型的加载与推理解决方案
- Optimum：优化加速库，支持多种后端，如Onnxruntime、OpenVino等
- Gradio：可视化部署库，几行代码快速实现基于Web交互的算法演示系统

基础组件之Tokenizer

Tokenizer 简介

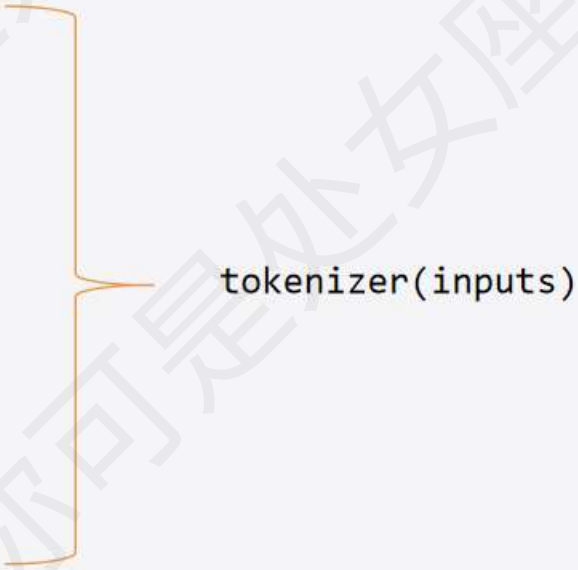
- **数据预处理**
  - **Step1 分词**：使用分词器对文本数据进行分词（字、字词）；
  - **Step2 构建词典**：根据数据集分词的结果，构建词典映射（这一步并不绝对，如果采用预训练词向量，词典映射要根据词向量文件进行处理）；
  - **Step3 数据转换**：根据构建好的词典，将分词处理后的数据做映射，将文本序列转换为数字序列；
  - **Step4 数据填充与截断**：在以batch输入到模型的方式中，需要对过短的数据进行填充，过长的数据进行截断，保证数据长度符合模型能接受的范围，同时batch内的数据维度大小一致。

现在：Tokenizer is all you need!

基础组件之Tokenizer

Tokenizer 基本使用

- 加载保存（from_pretrained / save_pretrained）
- 句子分词（tokenize）
- 查看词典（vocab）
- 索引转换（convert_tokens_to_ids / convert_ids_to_tokens）
- 填充截断（padding / truncation）
- 其他输入（attention_mask / token_type_ids）



基础组件之Tokenizer

Fast / Slow Tokenizer

- **FastTokenizer**
  - 基于Rust实现，速度快
  - offsets_mapping、word_ids
- **SlowTokenizer**
  - 基于Python实现，速度慢

```
%%time
# 单条循环处理
for i in range(10000):
    fast_tokenizer(sen)
```

✓ 0.3s

```
CPU times: total: 78.1 ms
Wall time: 312 ms
```

```
%%time
# 处理batch数据
res = fast_tokenizer([sen] * 10000)
```

✓ 0.1s

```
CPU times: total: 359 ms
Wall time: 87.1 ms
```

```
%%time
# 单条循环处理
for i in range(10000):
    slow_tokenizer(sen)
```

✓ 0.8s

```
CPU times: total: 281 ms
Wall time: 872 ms
```

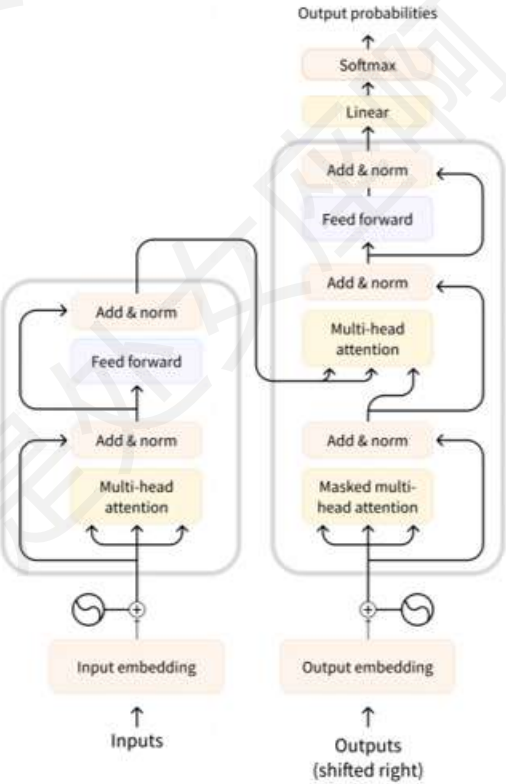
```
%%time
# 处理batch数据
res = slow_tokenizer([sen] * 10000)
```

✓ 0.7s

```
CPU times: total: 188 ms
Wall time: 705 ms
```

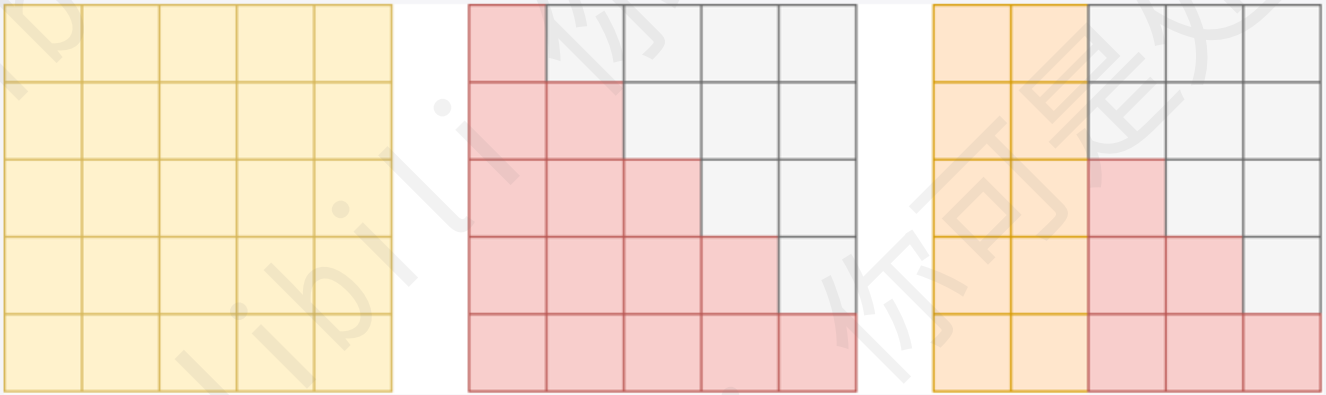
Model简介

- **Transformer**
  - 原始的Transformer为编码器（Encoder）、解码器（Decoder）模型
  - Encoder部分接收输入并构建其完整特征表示，Decoder部分使用Encoder的编码结果以及其他的输入生成目标序列
  - 无论是编码器还是解码器，均由多个TransformerBlock堆叠而成
  - TransformerBlock由注意力机制（Attention）和FFN组成
- **注意力机制**
  - 注意力机制的使用是Transformer的一个核心特性，在计算当前词的特征表示时，可以通过注意力机制有选择性的告诉模型要使用哪些上下文



Model简介

- **模型类型**
  - 编码器模型：自编码模型，使用Encoder，拥有双向的注意力机制，即计算每一个词的特征时都看到完整上下文
  - 解码器模型：自回归模型，使用Decoder，拥有单向的注意力机制，即计算每一个词的特征时都只能看到上文，无法看到下文
  - 编码器解码器模型：序列到序列模型，使用Encoder+Decoder，Encoder部分使用双向的注意力，Decoder部分使用单向注意力



编码器模型　　解码器模型　　编码器解码器模型

Model简介

- **模型类型**
  - 编码器模型：自编码模型，使用Encoder，拥有双向的注意力机制，即计算每一个词的特征时都看到完整上下文
  - 解码器模型：自回归模型，使用Decoder，拥有单向的注意力机制，即计算每一个词的特征时都只能看到上文，无法看到下文
  - 编码器解码器模型：序列到序列模型，使用Encoder+Decoder，Encoder部分使用双向的注意力，Decoder部分使用单向注意力

模型类型	常用预训练模型	适用任务
编码器模型，自编码模型	ALBERT, BERT, DistilBERT, RoBERTa	文本分类、命名实体识别、阅读理解
解码器模型，自回归模型	GPT, GPT-2, Bloom, LLaMA	文本生成
编码器解码器模型，序列到序列模型	BART, T5, Marian, mBART, GLM	文本摘要、机器翻译

显存优化策略，4G显存也能跑BERT-Large

- **显存占用简单分析**
  - 模型权重
    - 4Bytes * 模型参数量
  - 优化器状态
    - 8Bytes * 模型参数量，对于常用的AdamW优化器而言
  - 梯度
    - 4Bytes * 模型参数量
  - 前向激活值
    - 取决于序列长度、隐层维度、Batch大小等多个因素

Transformers显存优化

显存优化策略，4G显存也能跑BERT-Large

- **显存优化策略**
  - hfl/chinese-macbert-large，330M

优化策略	优化对象	显存占用	训练时间
Baseline (BS 32, MaxLength 128)	-	15.2G	64s
+ Gradient Accumulation (BS 1, GA 32)	前向激活值	7.4G	259s
+ Gradient Checkpoints (BS 1, GA 32)	前向激活值	7.2G	422s
+ Adafactor Optiomizer (BS 1, GA 32)	优化器状态	5.0G	406s
+ Freeze Model (BS 1, GA 32)	前向激活值 / 梯度	3.5G	178s
+ Data Length (BS 1, GA 32, MaxLength 32)	前向激活值	3.4G	126s

关于参数高效微调（如Lora）、cpu offload、flash attention等技巧将在LLM章节进行讲解

预训练介绍

预训练简介

- **什么是预训练**
  - 预训练（Pretrain）是指通过自监督学习从大规模数据中获得与具体任务无关的预训练模型的过程，最终产出为预训练模型（Pretrained Model）。

模型类型	常用预训练模型	适用任务
编码器模型，自编码模型	ALBERT, BERT, DistilBERT, RoBERTa	文本分类、命名实体识别、阅读理解
解码器模型，自回归模型	GPT, GPT-2, Bloom, LLaMA	文本生成
编码器解码器模型，序列到序列模型	BART, T5, Marian, mBART	文本摘要、机器翻译

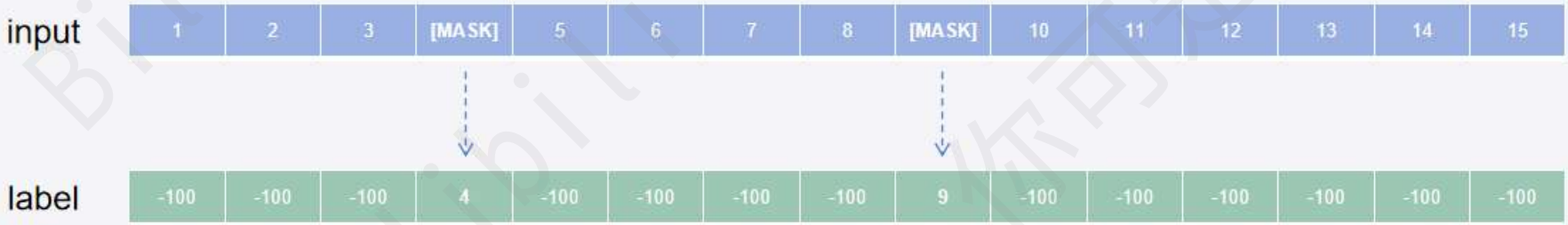
预训练介绍

预训练简介

- **预训练任务**
  - 掩码语言模型，自编码模型
    - 将一些位置的token替换成特殊的[MASK]字符，预测这些被替换的字符
  - 因果语言模型，自回归模型
    - 将完整序列输入，基于上文的token预测当前token
  - 序列到序列模型
    - 任务较为多样化，只是采用编码器解码器的方式，预测部分放在解码器中

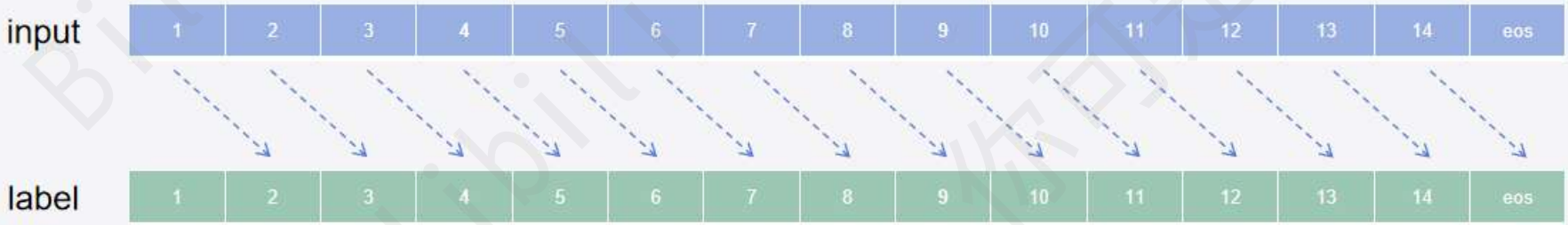
预训练简介

- **预训练任务**
  - 掩码语言模型，自编码模型
    - 将一些位置的token替换成特殊的[MASK]字符，预测这些被替换的字符
    - 只计算掩码部分的loss，其余部分不计算loss



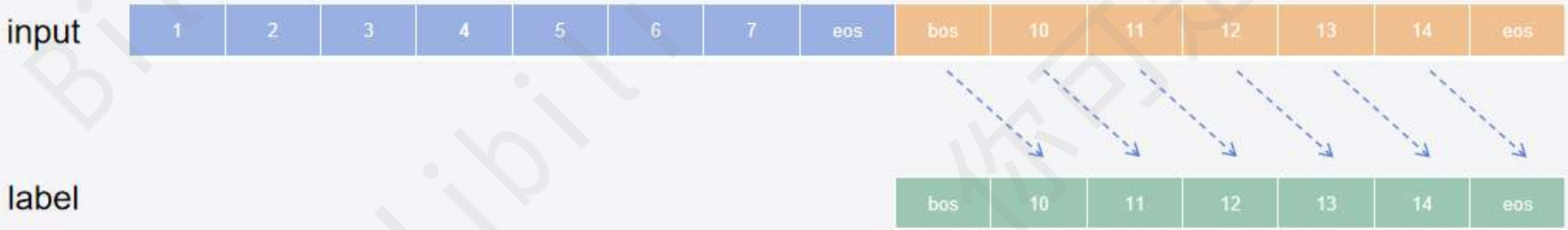
预训练简介

- **预训练任务**
  - 因果语言模型，自回归模型
    - 将完整序列输入，基于上文的token预测当前token
    - 结束位置要有特殊token，eos_token



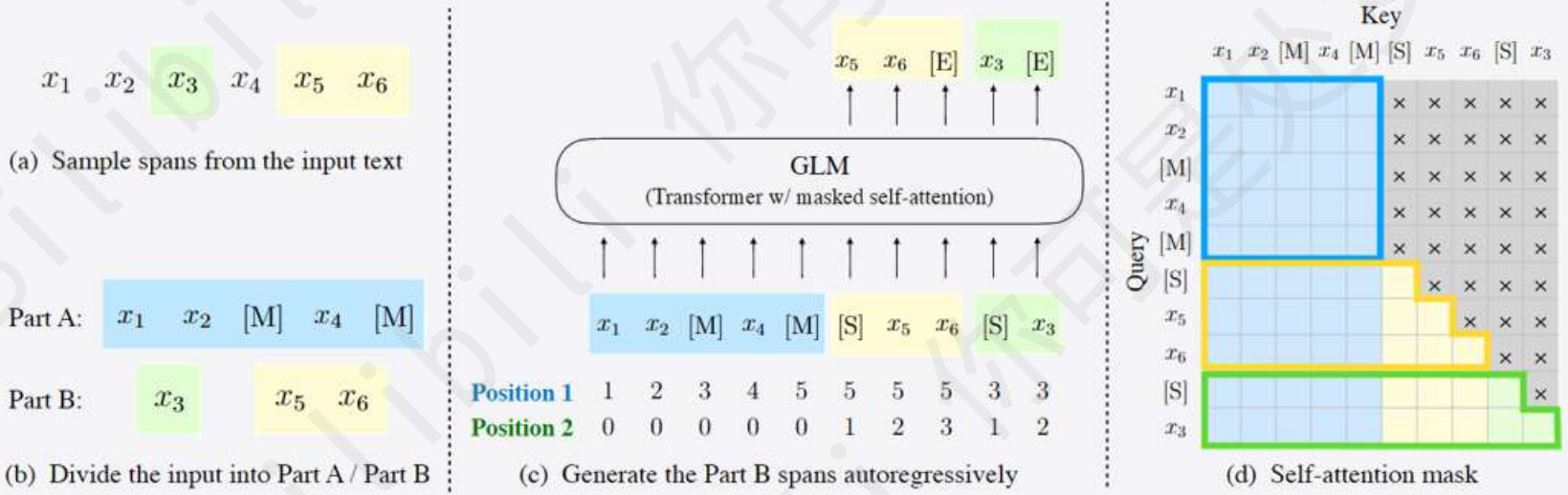
预训练简介

- **预训练任务**
  - 序列到序列模型，前缀语言模型（Prefix Language Model）
    - 任务较为多样化，如掩码生成、片段自回归、乱序修正等
    - 采用编码器解码器的方式进行实现，计算解码器部分的loss



前缀语言模型

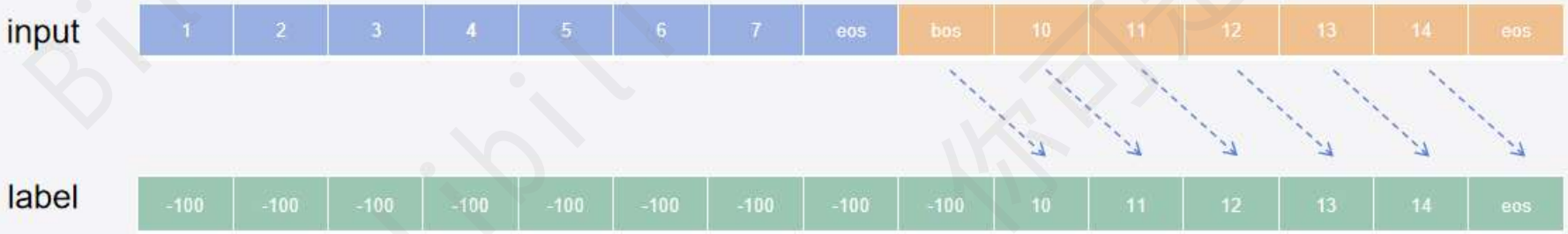
- **数据处理与模型原理**
  - 模型原理
    - 只使用编码器，借助注意力掩码实现编码器解码器的效果，只计算目标部分损失



基于Transformers的解决方案介绍

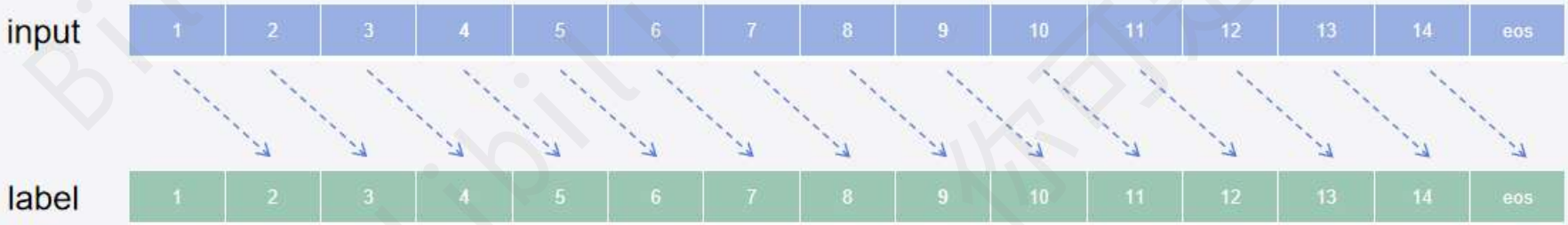
前缀语言模型

- **数据处理与模型原理**
  - 模型原理
    - 只使用解码器，借助注意力掩码实现编码器解码器的效果，只计算目标部分损失
  - 数据处理
    - input和labels合并在一起处理，labels的最后一定是eos_token



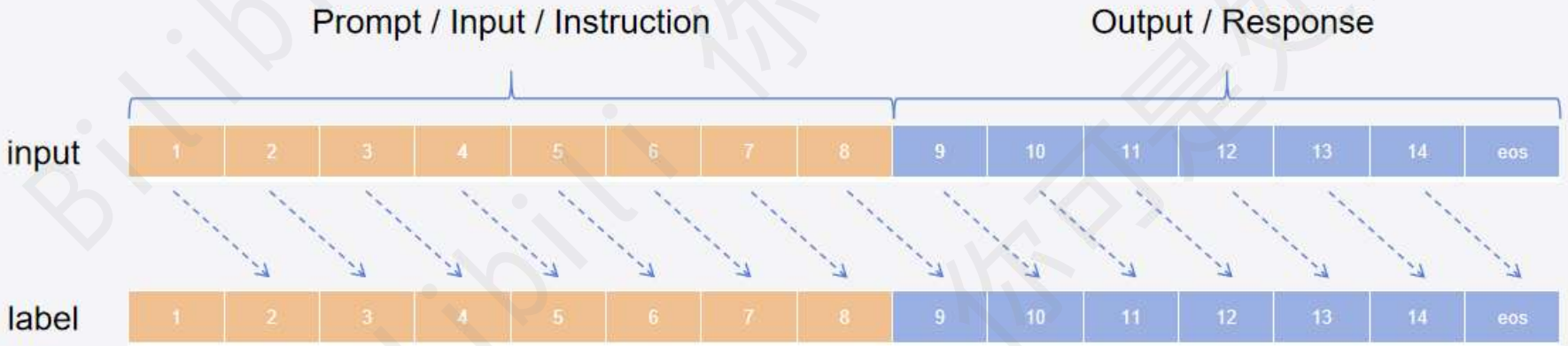
预训练简介

- **预训练任务**
  - 因果语言模型，自回归模型
    - 将完整序列输入，基于上文的token预测当前token
    - 结束位置要有特殊token，eos_token



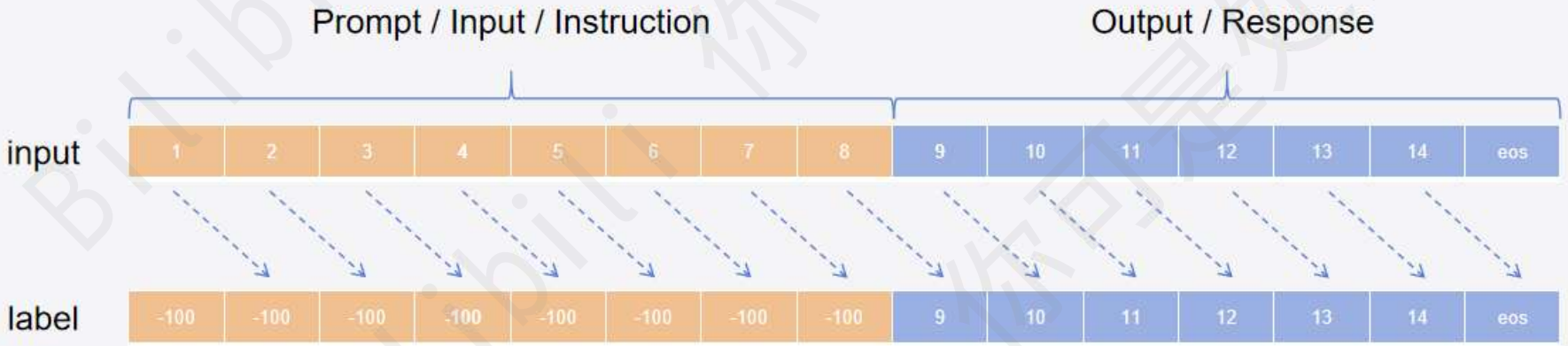
对话机器人解决方案

- **指令微调**
  - 指令微调的方式，赋予回答问题的能力
    - 多类型的任务共同学习，能够解决不同的任务



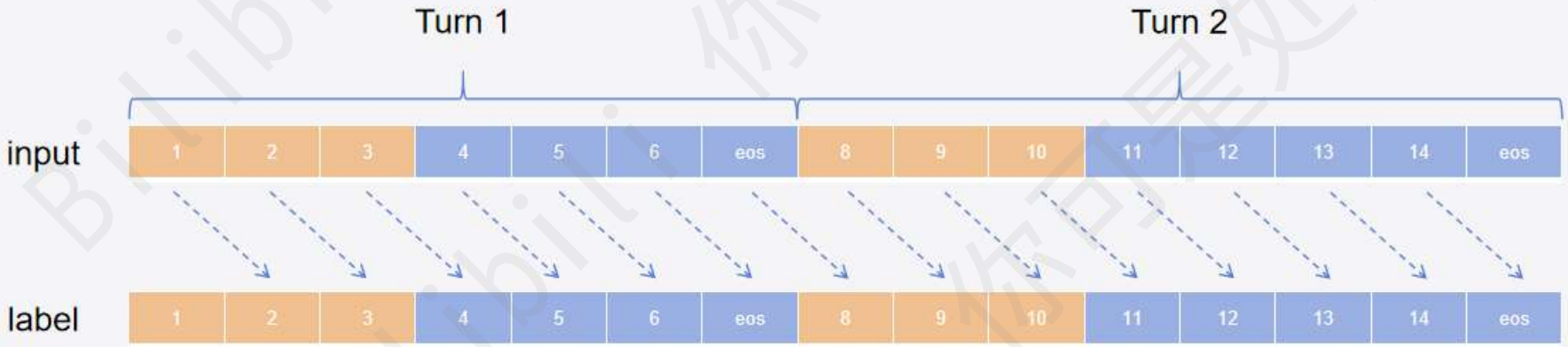
对话机器人解决方案

- **指令微调**
  - 指令微调的方式，赋予回答问题的能力
    - 训练单轮问答模型，计算Loss时只计算Output部分



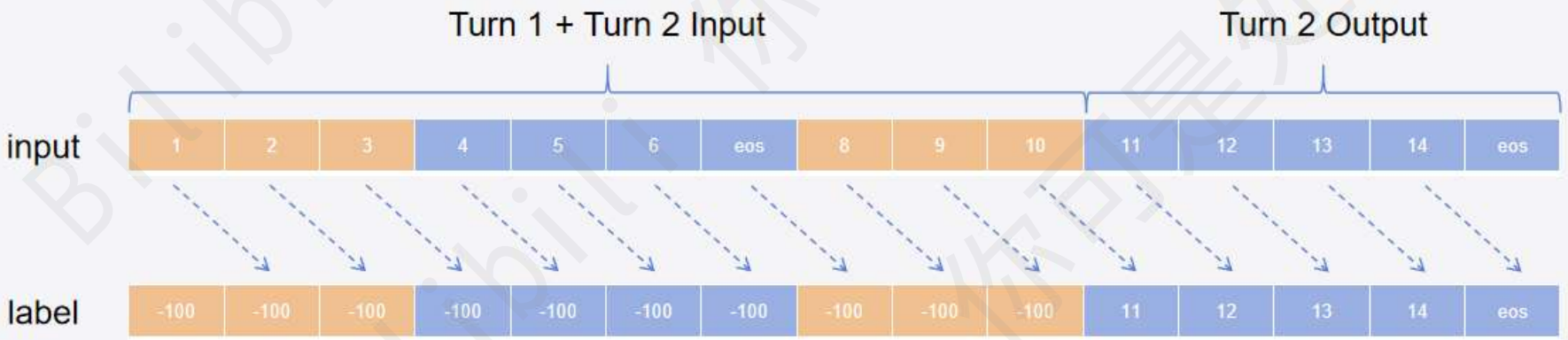
对话机器人解决方案

- **指令微调**
  - 指令微调的方式，赋予回答问题的能力
    - 多轮如何计算？



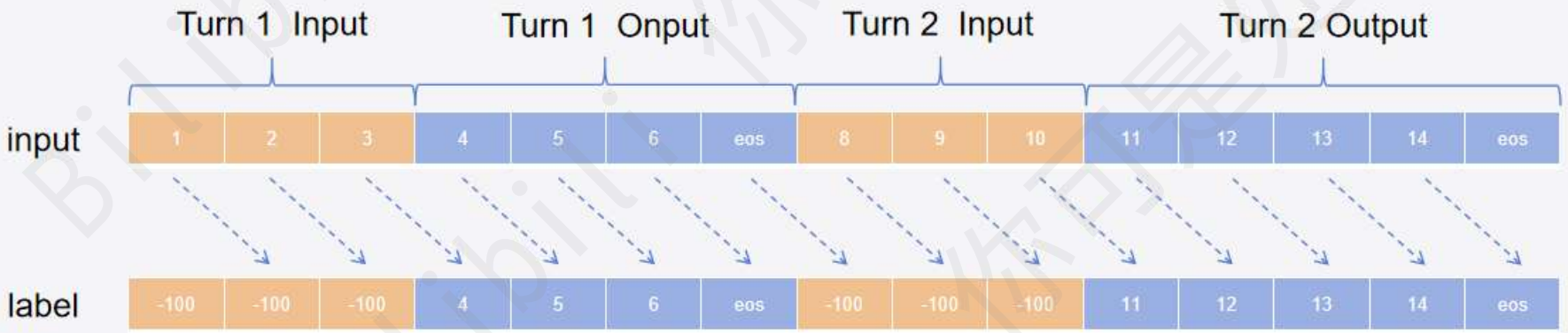
对话机器人解决方案

- **指令微调**
  - 指令微调的方式，赋予回答问题的能力
    - 方式一 计算最后一轮Output的Loss，效率较低



对话机器人解决方案

- **指令微调**
  - 指令微调的方式，赋予回答问题的能力
    - 方式二 计算每一轮Output的Loss，效率更高

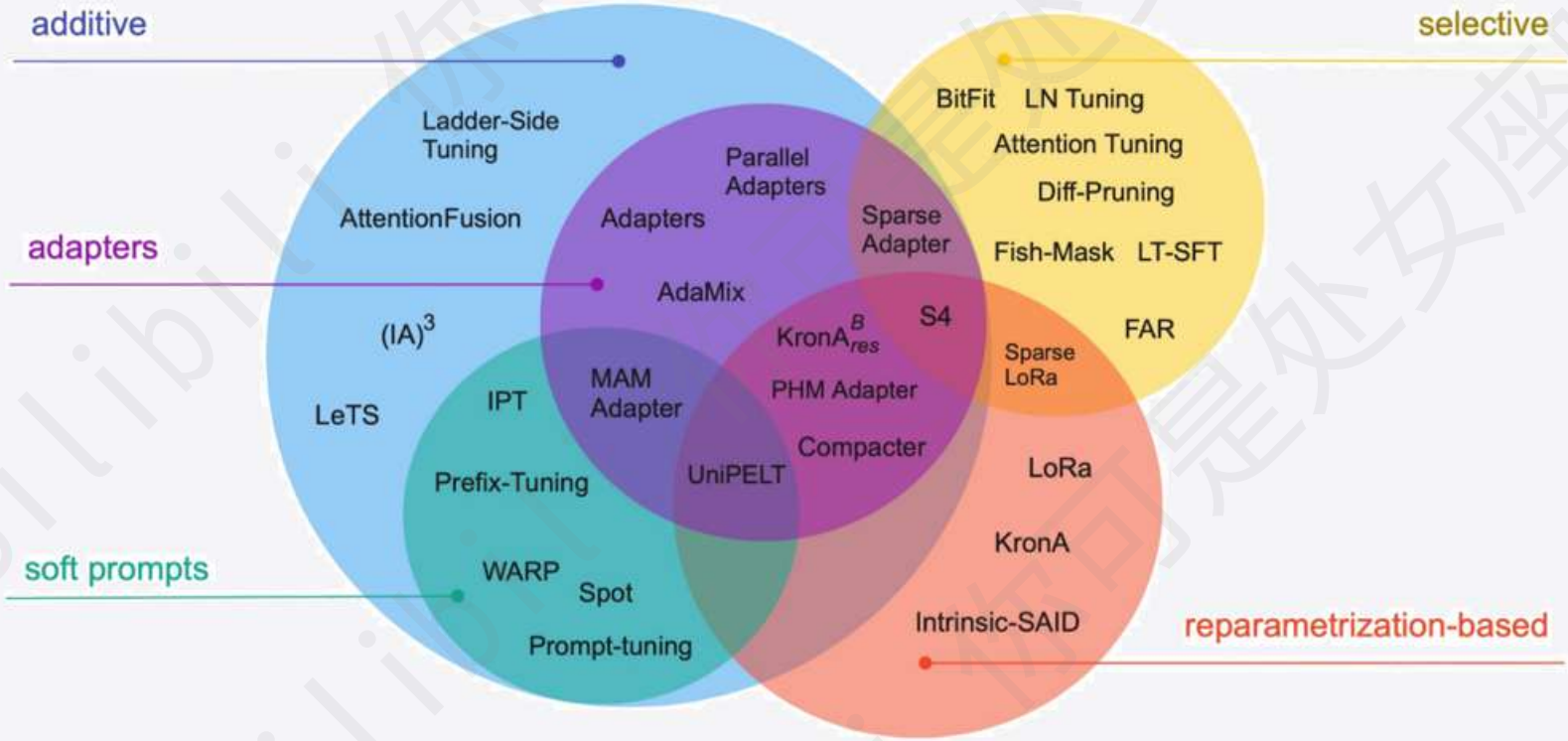


常见解码参数介绍

- **常用推理参数**
  - 长度控制
    - min/max_new_tokens 最小/最大生成的长度
    - min/max_length 序列整体的最小/最大长度
  - 解码策略
    - do_sample 是否启用采样的生成方式
    - num_beams beam_search的大小
  - 采样参数
    - temperature 默认1.0，即原始分布，低于1.0会使得分布更尖锐，高于1.0会使得分布更均匀
    - top_k 将词概率从大到小排列，将采样限制在前K个词
    - top_p 将词概率从大到小排列，将采样限制在前N个词，条件是这N个词的概率超过top_p的值
  - 惩罚项
    - repetition_penalty 重复惩罚项，实现原理是降低已经出现过的token的概率

为什么需要参数高效微调

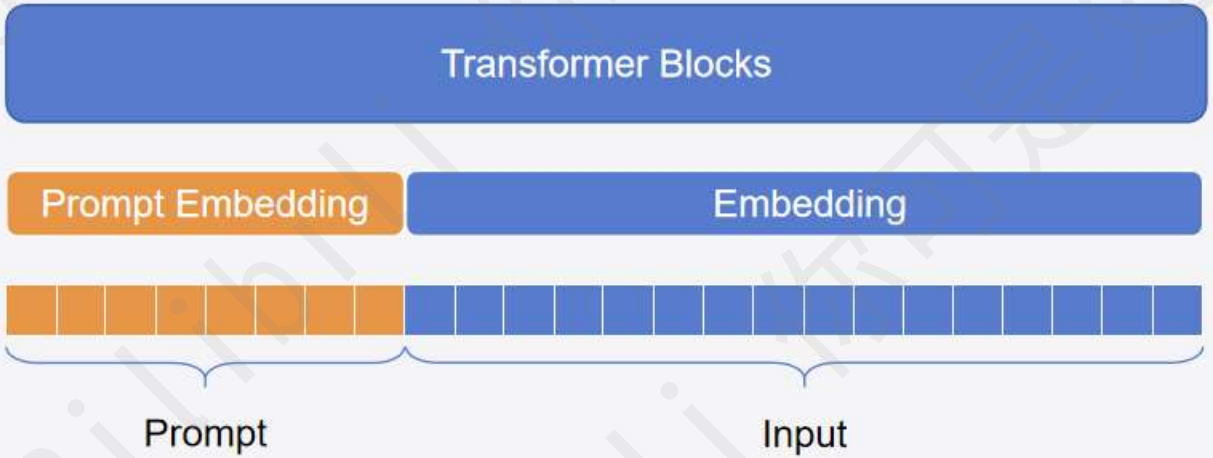
常见的参数高效微调方法



Lialin V, Deshpande V, Rumshisky A. Scaling down to scale up: A guide to parameter-efficient fine-tuning[J]. arXiv preprint arXiv:2303.15647, 2023.

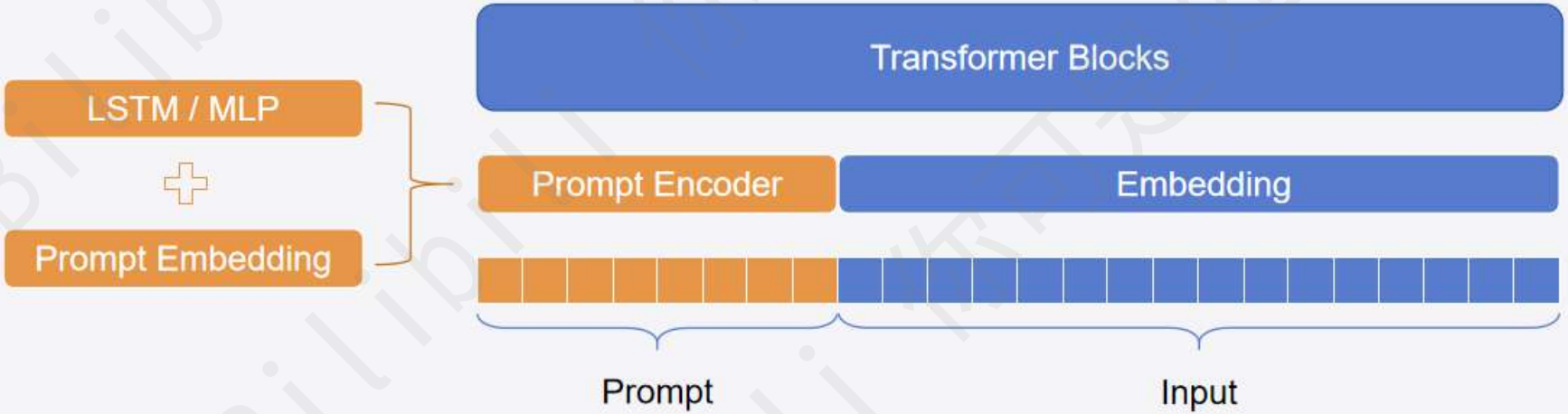
Prompt-Tuning 原理介绍

- **Prompt-Tuning**
  - Prompt-Tuning的思想：冻结主模型全部参数，在训练数据前加入一小段Prompt，只训练Prompt的表示层，即一个Embedding模块。其中，Prompt又存在两种形式，一种是hard prompt，一种是soft prompt。



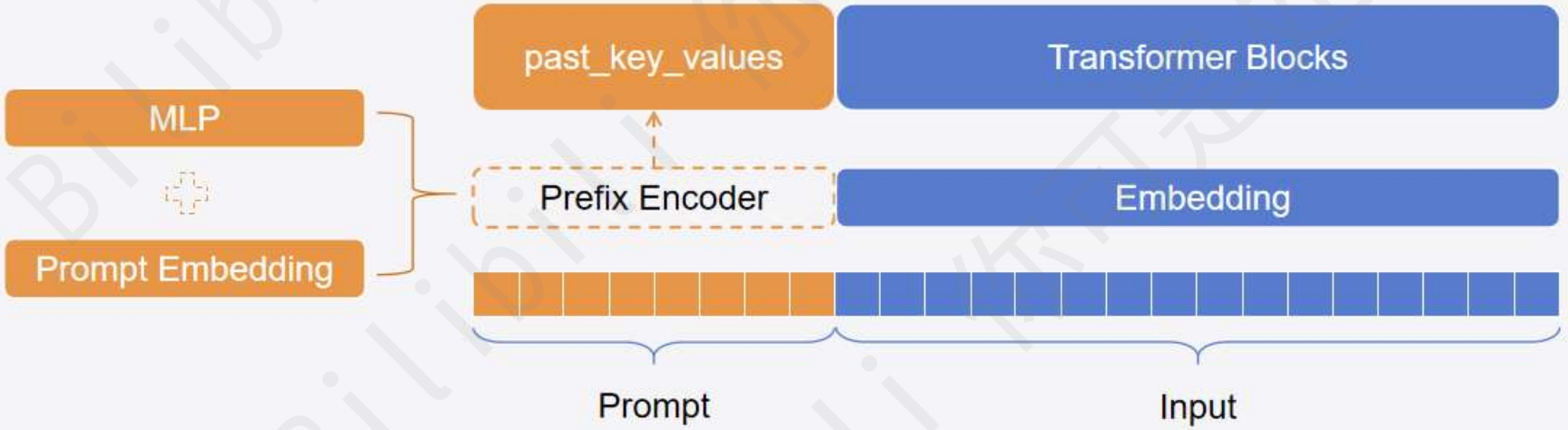
P–Tuning 原理介绍

- **P-Tuning**
  - P-Tuning的思想：在Prompt-Tuning的基础上，对Prompt部分进行进一步的编码计算，加速收敛。具体来说，PEFT中支持两种编码方式，一种是LSTM，一种是MLP。与Prompt-Tuning不同的是，Prompt的形式只有Soft Prompt。



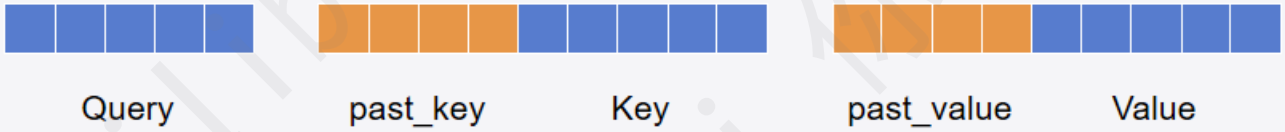
Prefix-Tuning 原理介绍

- **Prefix-Tuning**
  - Prefix-Tuning的思想：相较于Prompt-Tuning和P-tuning，Prefix-Tuning不再将Prompt加在输入的Embedding层，而是将其作为可学习的前缀，放置在Transformer模型中的每一层中，具体表现形式为past_key_values。



Prefix-Tuning 原理介绍

- **如何理解past_key_values**
  - past_key_values：Transformer模型中历史计算过的key和value的结果，最早是用于生成类模型解码加速，解码逻辑是根据历史输入，每次预测一个新的token，然后将新的token加入输入，再预测下一个token。这个过程中，会存在大量的重复计算，因此可以将key和value的计算结果缓存，作为past_key_values输入到下一次的计算中，这一技术又被称之为kv_cache。
  - Prefix-Tuning中，就是通过past_key_values的形式将可学习的部分放到了模型中的每一层，这部分内容又被称之为前缀

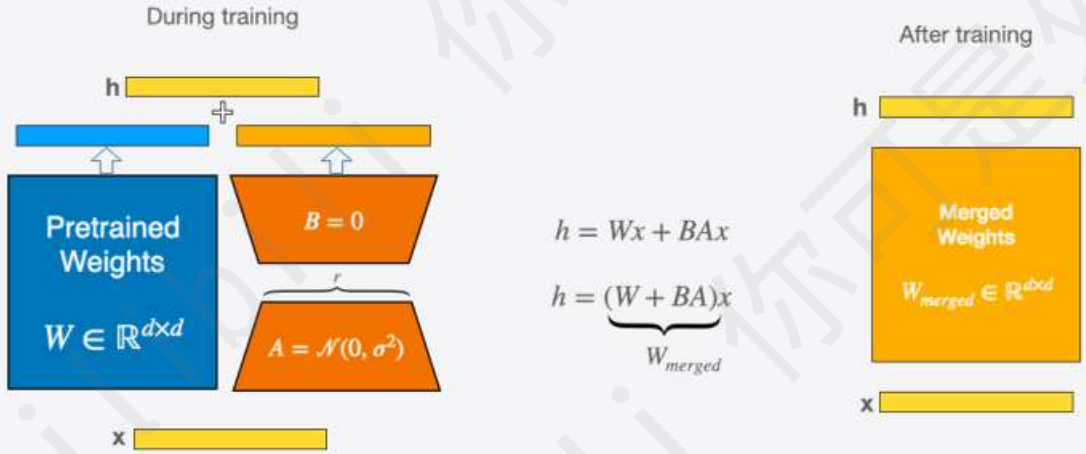


Lora 原理介绍

- **Lora**
  - Lora的思想：
    - 预训练模型中存在一个极小的内在维度，这个内在维度是发挥核心作用的地方。
    - 在继续训练的过程中，权重的更新依然也有如此特点，即也存在一个内在维度（内在秩）。
    - 权重更新：$W = W + \triangle W$
    - 因此，可以通过矩阵分解的方式，将原本要更新的大的矩阵变为两个小的矩阵。
    - 权重更新：$W = W + \triangle W = W + BA$
    - 具体做法，即在矩阵计算中增加一个旁系分支，旁系分支由两个低秩矩阵A和B组成。

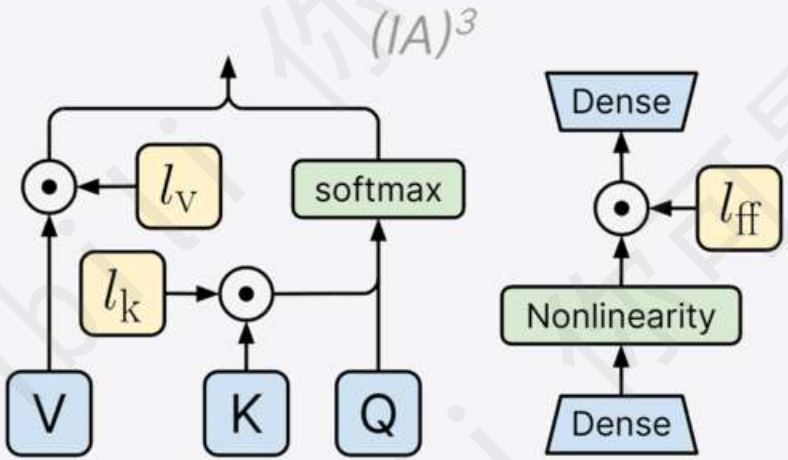
Lora 原理介绍

- **Lora**
  - 训练时，输入分别与原始权重和两个低秩矩阵进行计算，共同得到最终结果，优化则仅优化A和B。
  - 训练完成后，可以将两个低秩矩阵与原始模型中的权重进行合并，合并后的模型与原始模型无异，避免了推理期间Prompt系列方法带来的额外计算量



IA3 原理介绍

- **IA3，Infused Adapter by Inhibiting and Amplifying Inner Activations**
  - IA3的思想：抑制和放大内部激活，通过可学习的向量对激活值进行抑制或放大。具体来说，会对K、V、FFN三部分的值进行调整，训练过程中同样冻结原始模型的权重，只更新可学习的部分向量部分。训练完成后，与Lora类似，也可以将学习部分的参数与原始权重合并，没有额外推理开销。



背景介绍

- **模型训练的显存占用**
  - 模型权重
    - 4Bytes * 模型参数量
  - 优化器状态
    - 8Bytes * 模型参数量，对于常用的AdamW优化器而言
  - 梯度
    - 4Bytes * 模型参数量
  - 前向激活值
    - 取决于序列长度、隐层维度、Batch大小等多个因素

背景介绍

- **如何降低训练时显存占用**
  - 实战演练篇——4G显存，0.3B模型
    - 梯度累计
    - 梯度检查点
    - 优化器配置
    - 输入数据长度
    - 冻结模型参数
  - 参数高效微调篇——8G显存，1.4B模型
    - 参数高效微调（Prompt-Tuning、LoRA等）

背景介绍

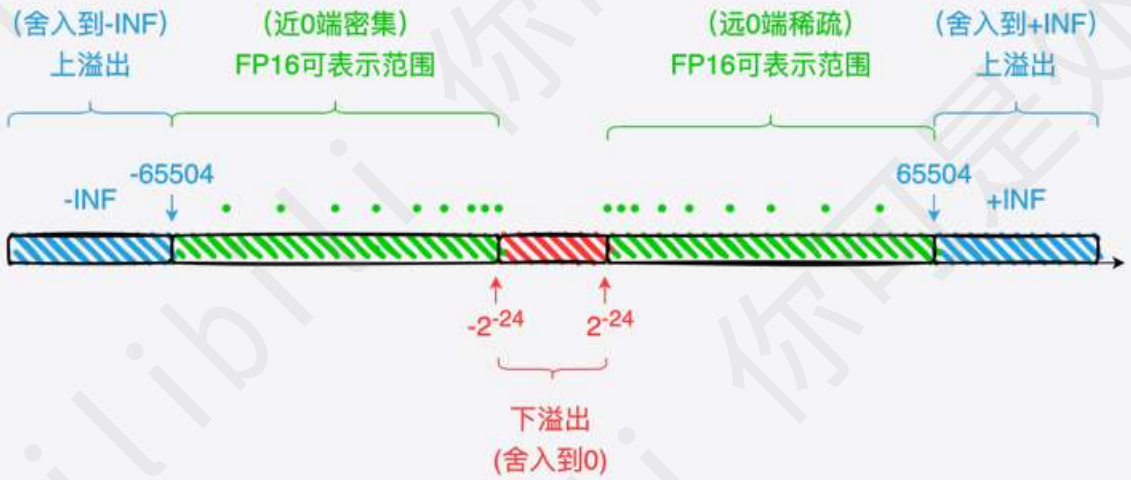
- **模型自身的显存占用**
  - 显存占用
    - 显存占用 ~= 4Bytes * 模型参数量
  - 如何降低显存占用
    - 参数量不变的情况下，降低模型中每个参数占用的字节数即可降低模型的显存占用
  - 如何降低参数占用的字节数
    - 默认的数值精度为单精度fp32，32bits，4Bytes
    - 使用低精度的数据类型表示即可
    - 常见的低精度数据类型：fp16（half、半精度）、bfloat16、int8、fp4、nf4

半精度介绍

- **什么是半精度**
  - 半精度FP16（half precision）是一种浮点数格式，它使用16bit表示一个数字（2个字节）
  - 在训练过程中，启用半精度训练可以有效节约显存，并提升计算速度

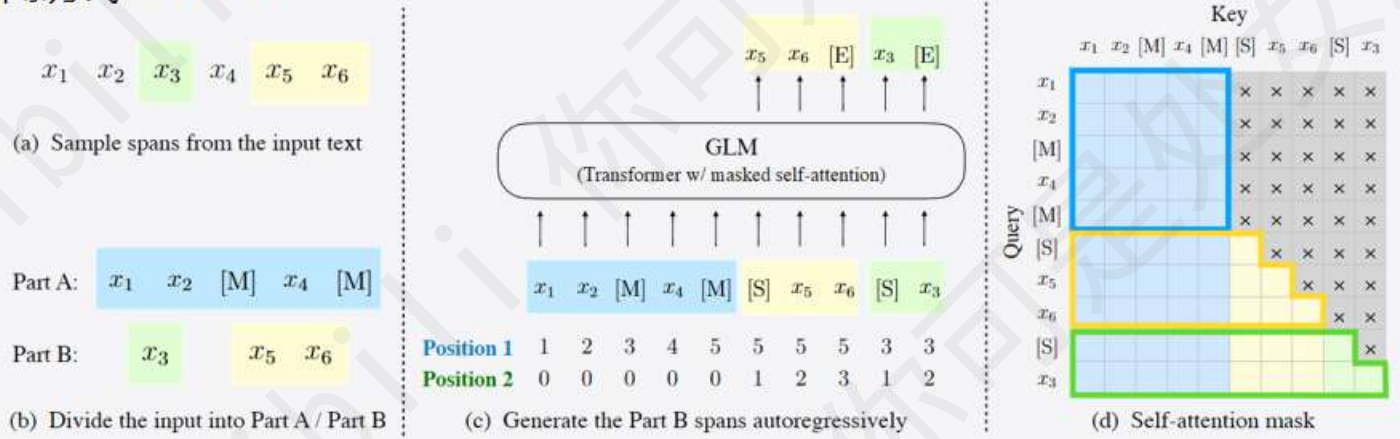
半精度介绍

- **什么是半精度**
  - 半精度FP16（half precision）是一种浮点数格式，它只占用16位（2个字节）
  - 在计算过程中的问题，可能存在溢出问题和舍入问题，可以使用bf16替代



ChatGLM

- **GLM和ChatGLM**
  - GLM模型
    - 训练方式 Prefix LM



    - 数据组织
      - sentenceA with mask（[MASK]、[gMASK]） [sop] sentenceB [eop]

代码实战演练

ChatGLM

- **GLM和ChatGLM**
  - ChatGLM
    - 训练方式
      - v1 Prefix LM
      - v2 Causal LM
    - 数据格式
      - [Round N]\n\n问：Pormpt \n\n答：Response
    - 数据组织
      - v1 Prompt [gMASK] [sop] Response [eop]
      - v2 [gMASK] [sop] Prompt Response [eos]

量化介绍

什么是量化

- 量化是一种模型压缩的方法，使用低精度数据类型对模型权重或者激活值进行表示。简单来说，量化就是将高精度的数字通过某种手段将其转换为低精度的数据
- 量化带来的优势
  - 降低模型加载的内存成本（FP32>FP16>INT8）
  - 减少运行计算成本，加速计算（并非所有情况都可以加速）
- 那么，如何进行量化?

量化介绍

- **INT8量化示例(absmax)**
  - 原始数据：x = [1.52, 2.64, -3.45, 4.32]
  - 量化过程：
    - x_absmax = 4.32
    - scale_factor = 127 / x_absmax ≈ 29.4
    - q_x = round([1.52, 2.64, -3.45, 4.32] * scale_factor) = [45, 78, -101, 127]
  - 反量化过程
    - x’ = q_x / scale_factor = [1.53, 2.61, 3.44, 4.32]

量化介绍

- **量化的问题**
  - 量化过程存在量化误差
    - 原始数据：x = [1.52, 2.64, -3.45, 4.32]
    - 反量化结果：x’ = [1.53, 2.61, 3.44, 4.32]
  - 如何降低量化误差，提高量化精度?
    - 使用更多的量化参数（scale_factor）
    - 矩阵乘法A*B可以看作是A的每一行乘上B的每一列，为A的每一行和B的每一列单独设置scale_factor，这种方式被称之为Vector-wise量化

量化介绍

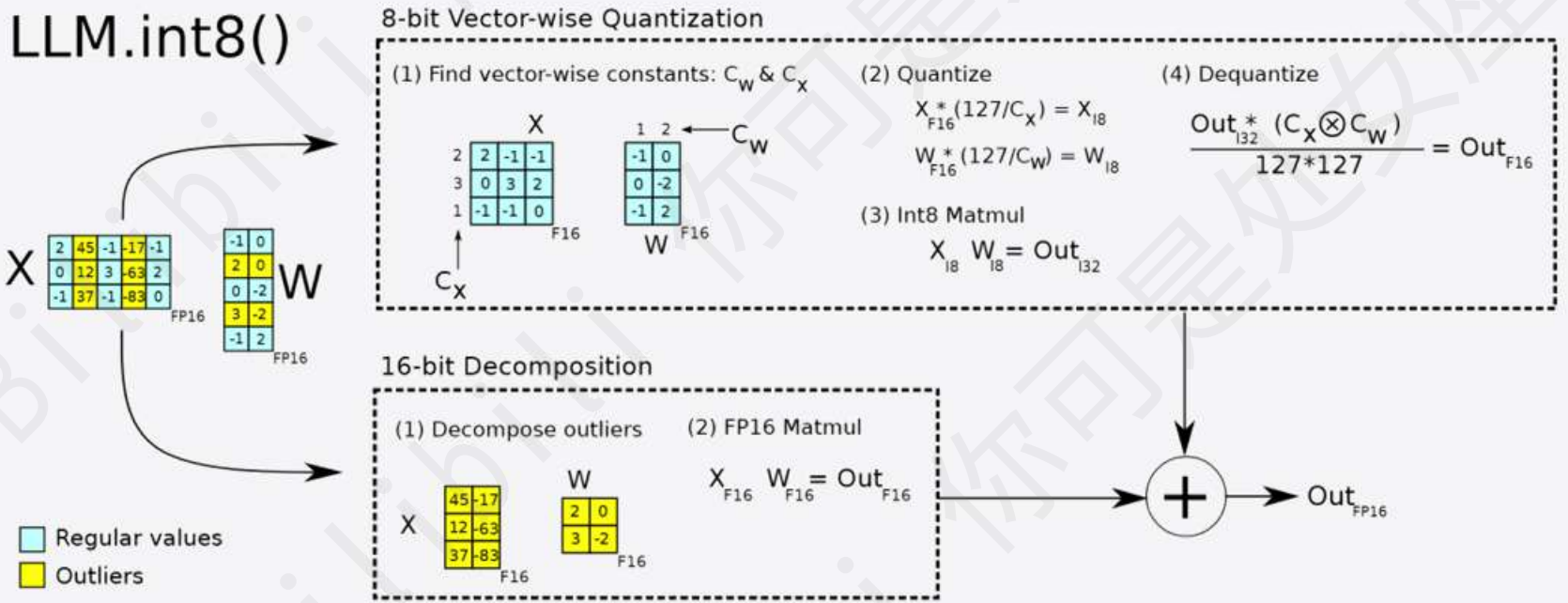
- **INT8量化示例2(absmax)**
  - 原始数据：x = [1.42, 1.51, 1.54, 45.3]
  - 量化过程：
    - x_absmax = 45.3
    - scale_factor = 127 / x_absmax ≈ 2.8
    - q_x = round([1.42, 1.51, 1.54, 4.32] * scale_factor) = [4, 4, 4, 127]
  - 反量化过程
    - x’ = q_x / scale_factor = [1.43, 1.43, 1.43, 45.36]

量化介绍

- **量化的问题**
  - 离群值：超出某个分布范围的值通常称为离群值
    - x = [1.42, 1.51, 1.54, **45.3**]
  - 8 位精度的动态范围极其有限，因此量化具有多个大值的向量会产生严重误差
  - 误差在一点点累积的过程中会导致模型的最终性能大幅度下降
  - 那么，如何解决这一问题？

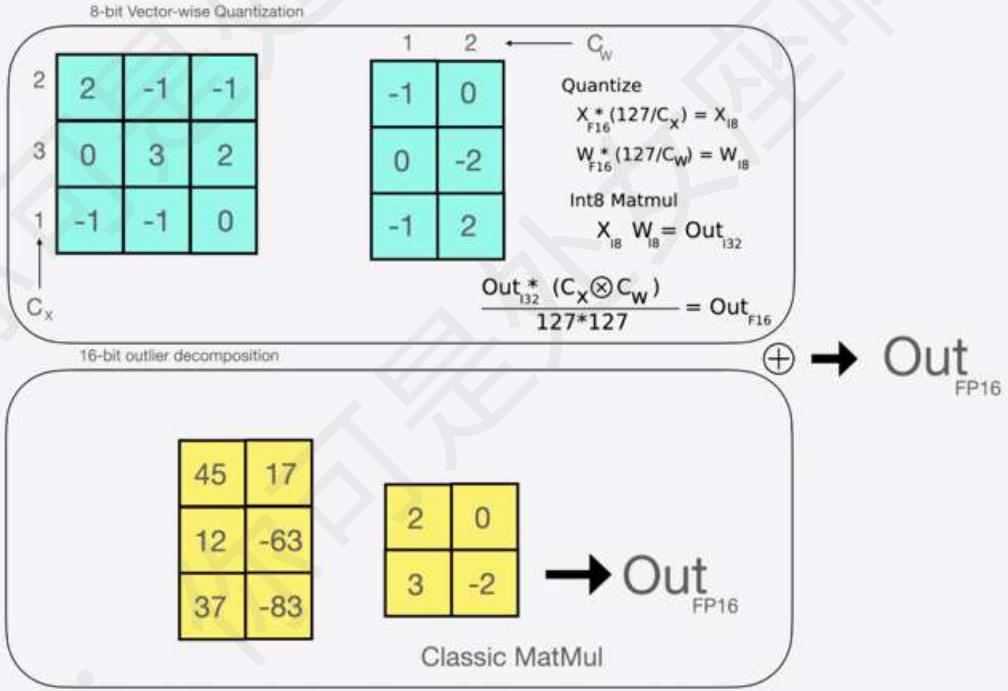
量化介绍

- **混合精度分解量化**



量化介绍

- **混合精度分解量化**
  - 从输入的隐含状态中，按列提取离群值 (即大于某个阈值的值)。
  - 对 FP16 离群值矩阵和 INT8 非离群值矩阵分别作矩阵乘法。
  - 反量化非离群值的矩阵乘结果并其与离群值矩阵乘结果相加，获得最终的 FP16 结果。



QLoRA介绍

- **还是从量化说起**
  - 量化的本质
    - 将原本一个空间的数映射到另外一个空间
  - 回顾8bit量化
    - 将FP32映射到INT8，使用了简单的线性变换
    - (x / scale_factor) × 127 -> [-127，127]

QLoRA介绍

- **8bit 线性量化**
  - 原始数据：x = [1.55, 1.62, 1.83, 4.32]
  - 8bit 量化过程：
    - x_absmax = 4.32
    - scale_factor = 127 / x_absmax ≈ 29.4
    - q_x = round([1.52, 1.62, 1.83, 4.32] * scale_factor) = [46, 48, 54, 127]
  - 反量化：
    - x’ = [46, 48, 54, 127] / scale_factor = [1.56, 1.63, 1.84, 4.32]

QLoRA介绍

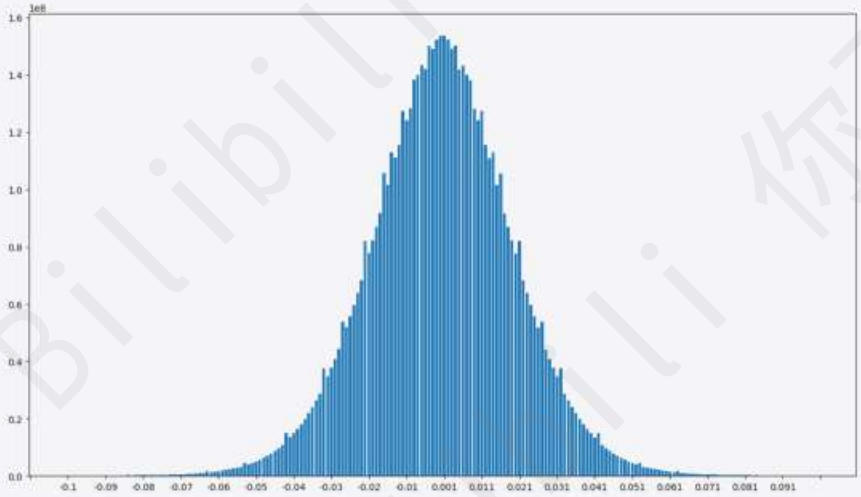
- **4bit 线性量化**
  - 原始数据：x = [1.55, 1.62, 1.83, 4.32]
  - 4bit 量化过程：
    - x_absmax = 4.32
    - scale_factor = 7 / x_absmax ≈ 1.62
    - q_x = round([1.52, 1.62, 1.83, 4.32] * scale_factor) = [3, 3, 3, 7]
  - 反量化：
    - x’ = [3, 3, 3, 7] / scale_factor = [1.85, 1.85, 1.85, 4.32]

QLoRA介绍

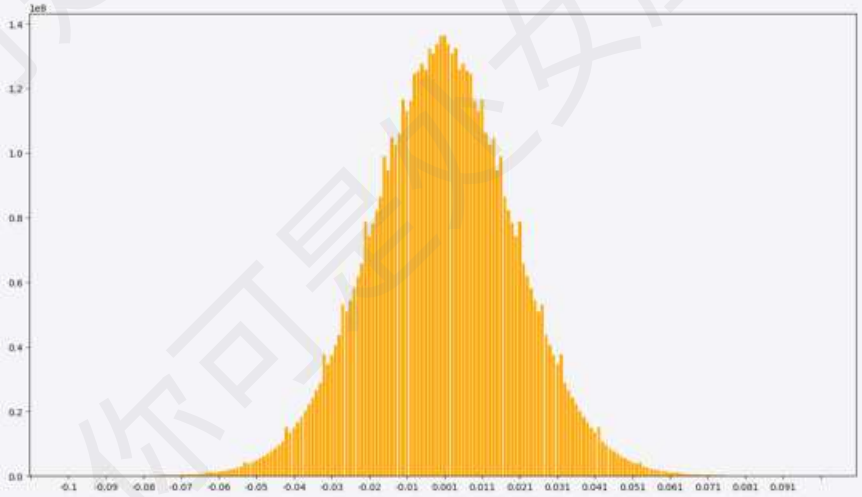
- **线性量化**
  - 简单线性量化的问题
    - 4 bit的表示范围比8bit更小，粒度更粗
    - 不用非常大的离群值，就使得量化后的参数都集中在某数值上
    - 量化误差会非常大
  - 如何理解线性量化
    - 数据归一化到[-1, 1]，把[-1, 1]均匀切分成N区间（N取决于量化bit）
    - 归一化的结果归属于第几个区间，量化结果便为几，数据的分布对量化结果非常重要
    - 如果待量化的数据为均匀分布，那么线性量化的结果即使是4bit也不会太差

QLoRA介绍

- **模型权重的真实分布**
  - 模型权重的真实分布



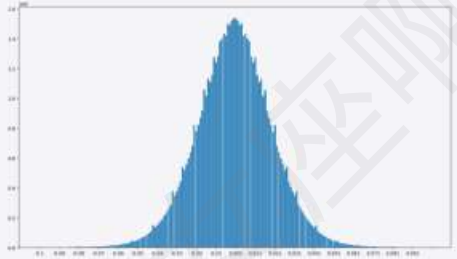
LLaMA2



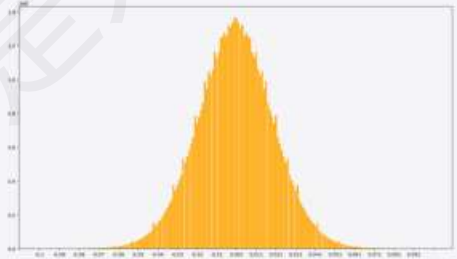
ChatGLM3

QLoRA介绍

- **正态分布**
  - 模型权重的真实分布
    - 权重的分布并非是均匀的
    - 权重的分布是呈现出一种正态分布的趋势的
  - 正态分布有什么特点
    - 中间多，两边少
    - 数值集中分布在均值两侧，离均值越远的部分越少



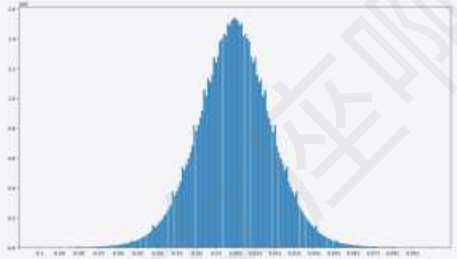
LLaMA2



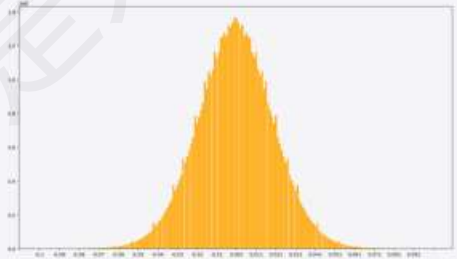
ChatGLM3

QLoRA介绍

- **如何与量化结合**
  - 更好的量化
    - 每部分的权重值均匀的落在N个区间中时量化效果会好
  - 线性量化的问题
    - 在目标域做均匀切分，无法将近似正态分布的值均匀的落在每个分区内
  - 换个思路
    - 目标域无法切分，那可以在源域上进行切分
    - 分位数量化



LLaMA2



ChatGLM3

QLoRA介绍

- **分位数量化**
  - 什么是分位数
    - 把顺序排列的一组数据分割为若干相等部分的分割点的数值即为相应的分位数
    - 中位数是分位数中最简单的一种，它将数据等分成两分。
    - 四分位数则是将数据按照大小顺序排序后，把数据分割成四等分的三个分割点上的数值。
  - 示例
    - {6, 7, 15, 36, 39, 40, 41, 42, 43, 47, 49}
    - 二分位数（中位数）：40
    - 四分位数：15，40，43

QLoRA介绍

- **分位数量化**
  - 分位数量化
    - 以4bit为例，表示范围为16个值，将权重从小到大排序，找到十五个分位数，将其切分为十六块，权重数值落在第几块，量化的表示就是多少，范围[0-15]
    - 此外，由于涉及到反量化，还需要给这16个值一个浮点数的映射，这个映射可以取分位区间两侧分位点的均值，用于反量化，这部分称之为量化值
    - 具体操作时，我们只需要把待量化的数跟量化值进行比较，取最相近的值即可作为该数值的量化值，对应的表示可以通过分位数进行确定，存储时同时存储4bit的表示与对应量化值，计算时使用量化值反量化后进行计算

QLoRA介绍

- **分位数量化改进-NF4**
  - 从正态分布入手
    - 大多数权重整体呈现正态分布，那么可以将其统一缩放至[-1，1]，根据标准正态分布得到16个量化值，并将量化值也缩放至[-1，1]，此时，便可利用前面提到的方法，将权重进行量化
    - 为了减少异常值的问题，采用分块量化，块大小为64，即64个值为一组进行量化
  - NF4
    - [-1.0, -0.6961928009986877, -0.5250730514526367, -0.39491748809814453, -0.28444138169288635, -0.18477343022823334, -0.09105003625154495, 0.0, 0.07958029955625534, 0.16093020141124725, 0.24611230194568634, 0.33791524171829224, 0.44070982933044434, 0.5626170039176941, 0.7229568362236023, 1.0]

QLoRA介绍

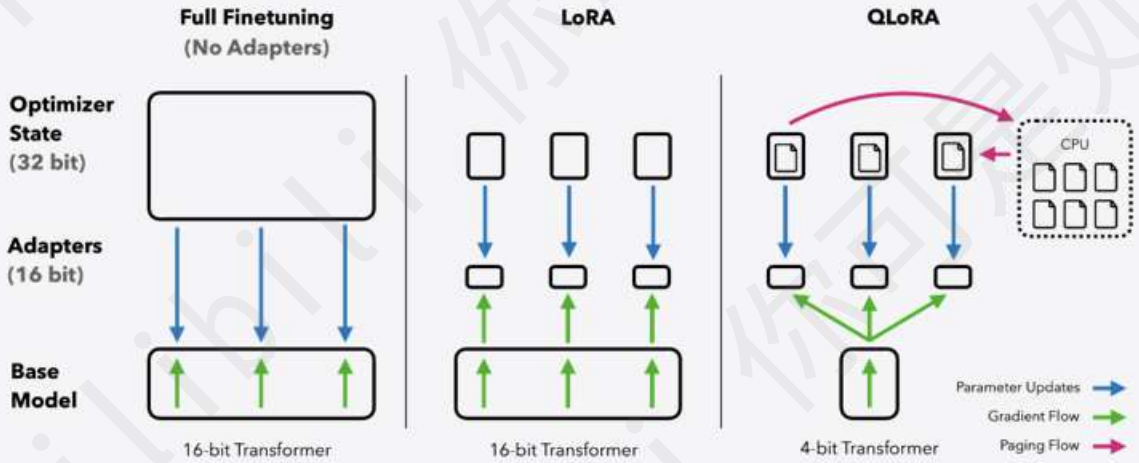
- **NF4 量化完整示例**
  - 原始数据：x = [1.55, 1.62, 1.83, 4.32]
  - 4bit 量化过程：
    - x_absmax = 4.32
    - x_norm = [1.55, 1.62, 1.67, 4.32] / 4.32 = [0.3587, 0.375, 0.4236, 1]
    - NF4 match：q_x = [0.3379, 0.3379, 0.4407, 1.0] = [11, 11, 12, 15]
  - 反量化过程：
    - x’ = [0.3379, 0.3379, 0.4407, 1.0] * x_absmax = [1.46, 1.46, 1.90, 4.32]

QLoRA介绍

- **双重量化**
  - NF4量化存储空间
    - NF4 分位点：16个常量值，可忽略不计
    - 量化常数（absmax）：每个块需要一个FP32的量化常数，32 / 64 = 0.5 bit，即每个参数除了4bit的值外，还会增加0.5bit的额外存储
  - 双重量化
    - 对量化常数进行二次量化，256个一组进行8bit量化，量化为fp8，
    - 8 / 64 + 32/(64 · 256) ≈ 0.127 bit
    - 相较于原始的0.5bit，每个参数额外的资源消耗减少了0.373bit

QLoRA介绍

- **分页优化器**
  - 分页优化器
    - 当显存不足时，将优化器的参数转移到CPU内存上，在需要时再将其取回，防止显存峰值时OOM



知识回顾

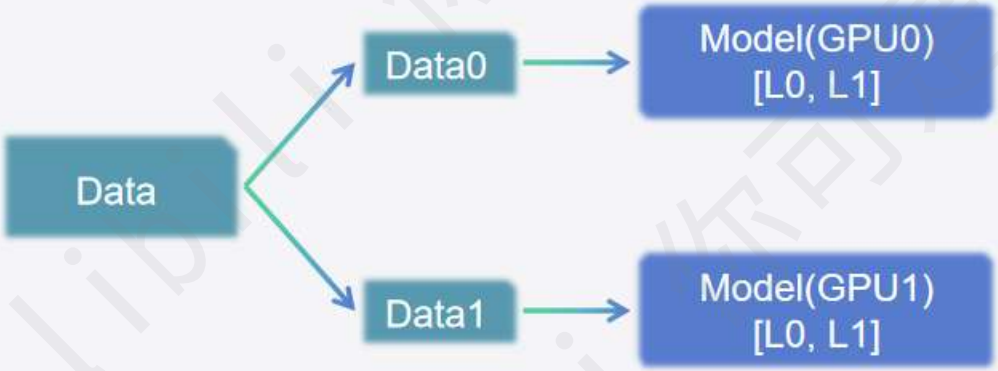
- **单卡场景如何解决显存问题**
  - 可训练参数量
    - 参数高效微调——PEFT
    - Prompt-Tuning、Prefix-Tuning、LoRA等
  - 参数精度
    - 低精度模型训练——Bitsandbytes
    - 半精度、INT8、NF4

分布式训练

- **分布式训练简介**
  - 什么是分布式模型训练
    - 分布式（Distributed）是指系统或计算任务被分布到多个独立的节点或计算资源上进行处理，而不是集中在单个节点或计算机上。
    - 分布式模型训练是一种机器学习和深度学习领域中的训练方法，它通过将计算任务分发到多个计算资源或设备上来加速模型的训练过程。分布式训练通过并行计算的方式，将数据和计算任务分配到多个节点上，从而提高训练速度和处理大规模数据的能力。

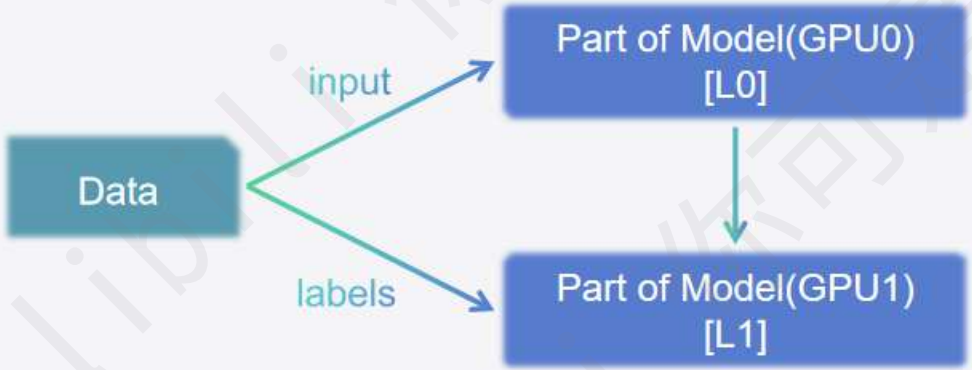
分布式训练

- **如何进行分布式模型训练**
  - 数据并行，Data Parrallel, DP
    - 每个GPU上都复制一份完整模型，但是每个GPU上训练的数据不同
    - 要求每张卡内都可以完整执行训练过程



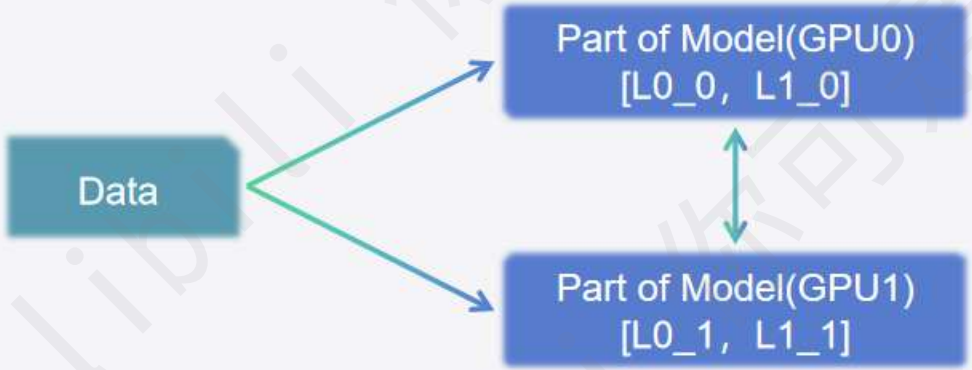
分布式训练

- **如何进行分布式模型训练**
  - 流水并行，Pipeline Parrallel，PP
    - 将模型按层拆开，每个GPU上包含部分的层，保证能够正常训练
    - 不要求每张卡内都可以完整执行训练过程



分布式训练

- **如何进行分布式模型训练**
  - 张量并行，Tensor Parrallel, TP
    - 将模型每层的权重拆开，对于一份权重，每个GPU上各包含一部分，保证能够正常训练
    - 不要求每张卡内都可以完整执行训练过程

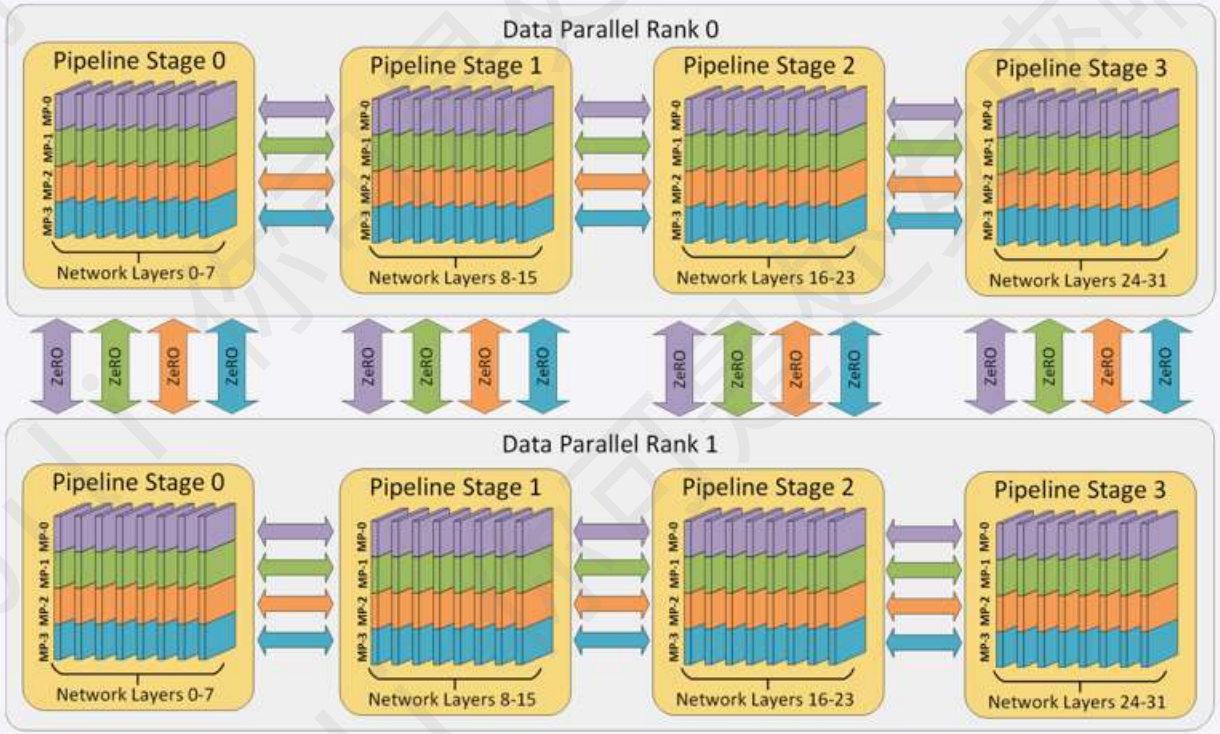


分布式训练

- **如何进行分布式模型训练**
  - 单卡可以完成训练流程的模型
    - 每个GPU上都复制一份完整模型，但是每个GPU上训练的数据不同（数据并行，Data Parrallel, DP）
  - 单卡无法完成训练流程的模型
    - 将模型按层拆开，每个GPU上包含部分的层，保证能够正常训练（流水并行，Pipeline Parrallel, PP）
    - 将模型每层的权重拆开，对于一份权重，每个GPU上各包含一部分，保证能够正常训练（张量并行，Tensor Parrallel, TP）
  - 混合策略
    - 数据并行 + 流水并行 + 张量并行（3D并行）

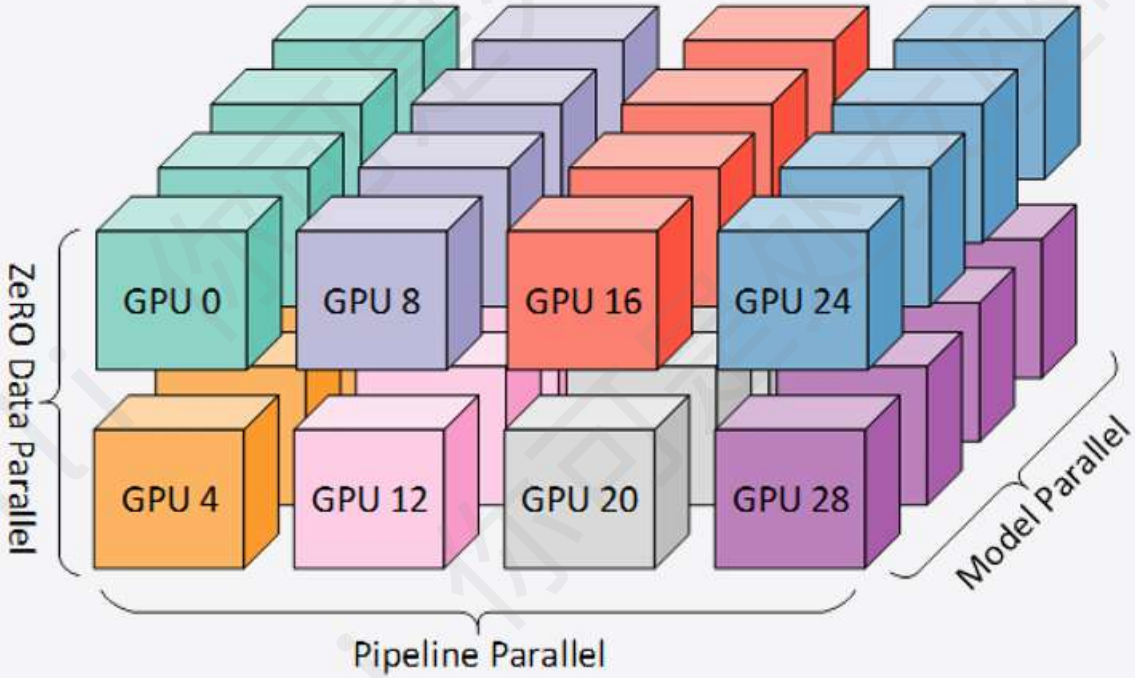
分布式训练

- **如何进行分布式模型训练**
  - 3D并行示例
    - 2路数据并行
    - 4路流水并行
    - 4路张量并行



分布式训练

- **如何进行分布式模型训练**
  - 3D并行示例
    - 2路数据并行
    - 4路流水并行
    - 4路张量并行



Data Parrallel 原理

- **Data Parrallel 原理**
  - 什么是数据并行
    - 每个GPU里都存一份完整的模型，训练时每个GPU里的模型训练不同的数据
    - 适用于单卡能够运行完整训练流程的情况，如果单卡无法运行完整流程，则无法使用
  - Data Parrallel
    - 这里特指Pytorch框架中的nn.DataParallel所实现的数据并行方法

Data Parrallel 原理

- **Data Parrallel 原理**
  - 训练流程
    - Step1 GPU0 加载model和batch数据
    - Step2 将batch数据从GPU0均分至各卡
    - Step3 将model从GPU0复制到各卡
    - Step4 各卡同时进行前向传播
    - Step5 GPU0收集各卡上的输出，并计算Loss
    - Step6 将Loss分发至各卡，进行反向传播，计算梯度
    - Step7 GPU0收集各卡的梯度，进行汇总
    - Step8 GPU0更新模型

Data Parrallel

- **Data Parrallel 训练实战**
  - 实际效果
    - 调用了多GPU进行训练，但是训练速度没有多大，甚至有可能下降（可能大家会观测到不同的现象）
  - Data Parrallel 的问题
    - 单进程，多线程，由于GIL锁的问题，不能充分发挥多卡的优势
    - 由于Data Parrallel的训练策略问题，会存在一个主节点占用比其他节点高很多
    - 效率较低，每次训练开始都要重新同步模型，大模型的同步时间会较难接受
    - 只适用于单机训练，无法支持真正的分布式多节点训练

Data Parrallel

- **Data Parrallel 训练实战**
  - 实际效果
    - 调用了多GPU进行训练，但是训练速度没有多大，甚至有可能下降（可能大家会观测到不同的现象）
  - Data Parrallel 的问题
    - 单进程，多线程，由于GIL锁的问题，不能充分发挥多卡的优势
    - 由于Data Parrallel的训练策略问题，会存在一个主节点占用比其他节点高很多
    - 只适用于单机训练，无法支持真正的分布式多节点训练

nn.DataParrallel是目前不推荐的一种数据并行训练策略

Data Parrallel

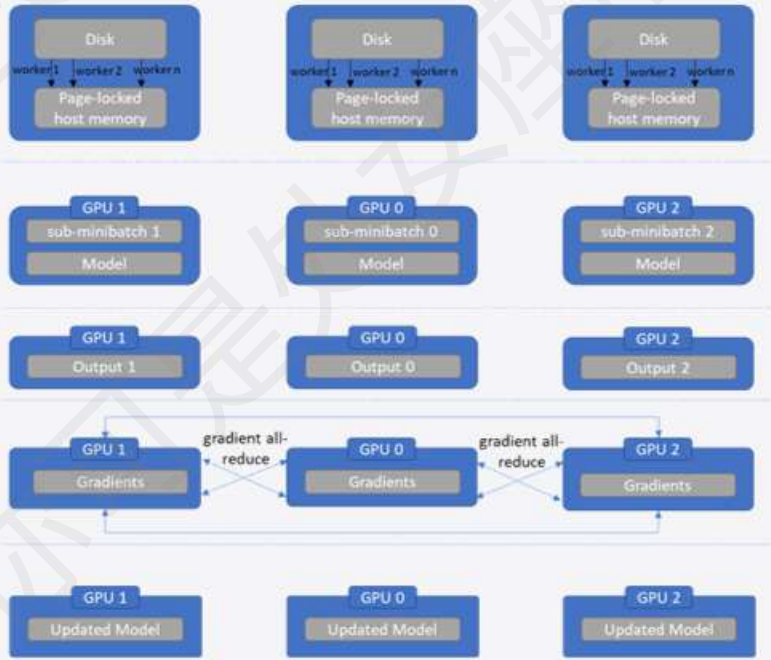
- **Data Parrallel 训练实战**
  - DataParallel真的没有用吗
    - 并非如此
    - 对于并行推理，DataParallel可以派上用场！
  - DataParallel 并行推理验证
    - DataParallel.module.forward()
    - DataParallel.forward()
    - DataParallel.forward()改进版本

Distributed Data Parrallel

- **为什么需要 Distributed Data Parrallel**
  - Data Parrallel 的问题
    - 单进程，多线程，由于GIL锁的问题，不能充分发挥多卡的优势
    - 由于Data Parrallel的训练策略问题，会存在一个主节点占用比其他节点高很多
    - 只适用于单机训练，无法支持真正的分布式多节点训练
    - nn.DataParrallel是目前不推荐的一种数据并行训练策略
  - 真正的分布式数据并行
    - Distributed Data Parrallel

Distributed Data Parallel 原理

- **Distributed Data Parallel 原理**
  - 训练流程
    - Step1 使用多个进程，每个进程都加载数据和模型
    - Step2 各进程同时进行前向传播，得到输出
    - Step3 各进程分别计算Loss，反向传播，计算梯度
    - Step4 各进程间通信，将梯度在各卡同步
    - Step5 各进程分别更新模型



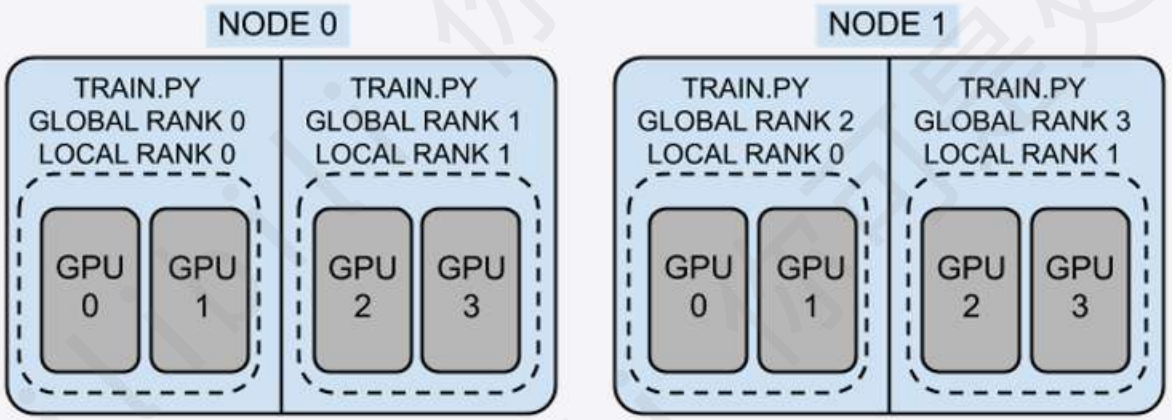
分布式训练中的通信

分布式训练中基本概念

- **分布式训练中的基本概念**
  - **group**：进程组，一个分布式任务对应一个进程组，一般就是所有卡都在一个组里
  - **world_size**：全局的并行数，一般情况下等于总的卡数
  - **node**：节点，可以是一台机器，或者一个容器，节点内包含多个GPU
  - **rank(global_rank)**：整个分布式训练任务内的进程序号
  - **local_rank**：每个node内部的相对进程序号

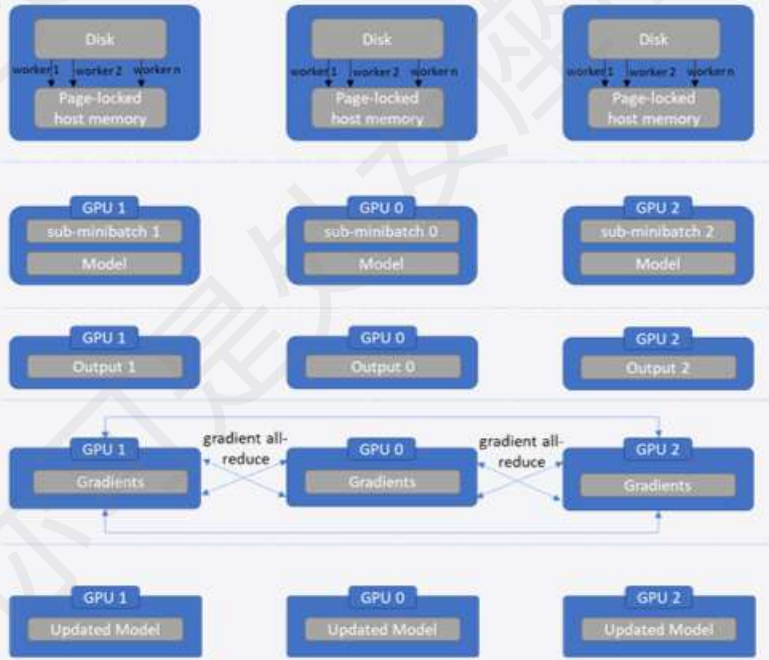
分布式训练中基本概念

- **分布式训练中的基本概念**
  - 2机4卡分布式训练示例
    - node=2, world_size=4,
    - 每个进程占用两个GPU



Distributed Data Parallel 原理

- **Distributed Data Parallel 原理**
  - 训练流程
    - Step1 使用多个进程，每个进程都加载数据和模型
    - Step2 各进程同时进行前向传播，得到输出
    - Step3 各进程分别计算Loss，反向传播，计算梯度
    - **Step4 各进程间通信，将梯度在各卡同步**
    - Step5 各进程分别更新模型

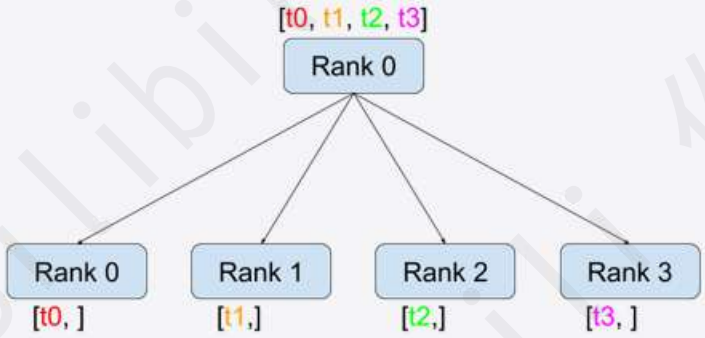


分布式训练中的通信

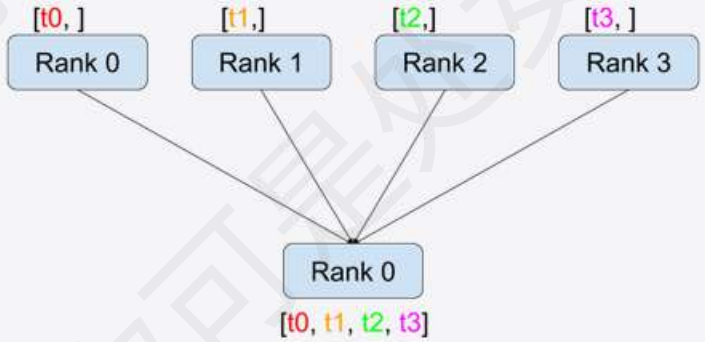
- **分布式训练中的通信**
  - 什么是通信
    - 在分布式模型训练中，通信是不同计算节点之间进行信息交换以协调训练任务的关键组成部分。
  - 通信类型
    - 点对点通信：将数据从一个进程传输到另一个进程称为点对点通信
    - 集合通信：一个分组中所有进程的通信模式称之为集合通信
      - 6种通信类型：Scatter、Gather、Reduce、All Reduce、Broadcast、All Gather

分布式训练中的通信

- **分布式训练中的通信**
  - 集合通信



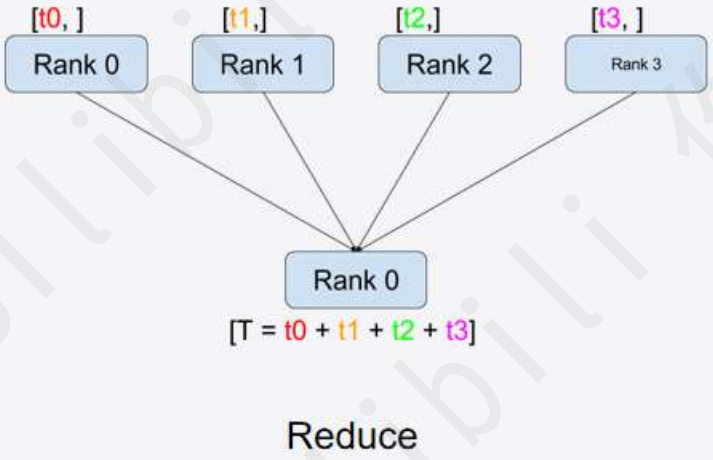
Scatter



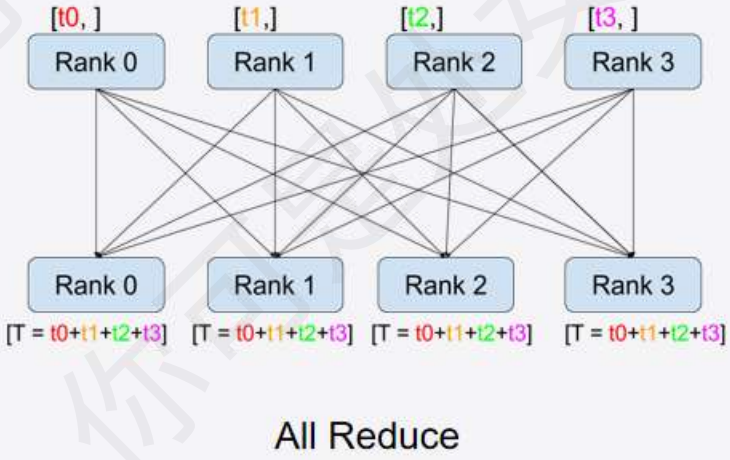
Gather

分布式训练中的通信

- **分布式训练中的通信**
  - 集合通信



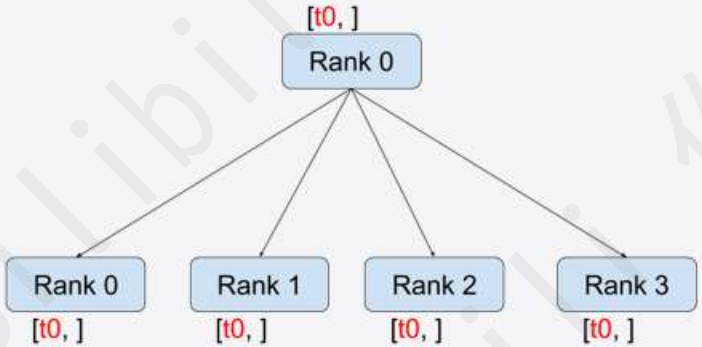
Reduce



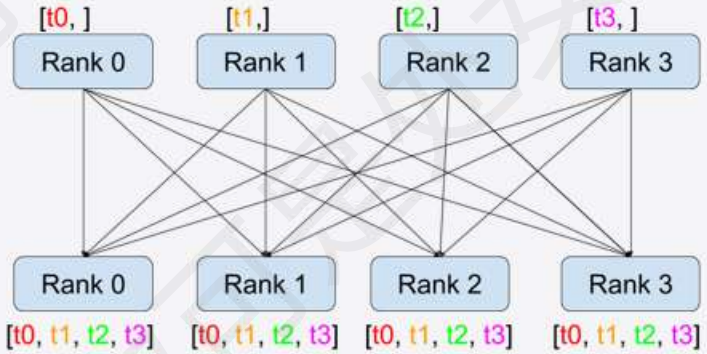
All Reduce

分布式训练中的通信

- **分布式训练中的通信**
  - 集合通信



Broadcast



All Gather

Distributed Data Parallel 训练实战

- **Distributed Data Parallel 实现注意细节**
  - 数据部分
    - 如果是在训练进程内对数据集进行划分，注意保证数据划分的一致性，可以通过随机种子控制
    - 分布式采样器会为了保证每个进程内数据大小一致，做额外的填充，评估指标可能会存在误差
  - 书写逻辑
    - 将分布式的代码看作单进程的代码即可，只是需要分布式的数据采样器以及启动略有不同
    - print打印的都是各自进程内的信息，需要全局的信息则需要自行调用通信计算结果
    - 数据放置到指定设备上时需要注意使用正确的device_id，一般用local_rank

Accelerate 基础入门

- **Accelerate 简介**
  - 什么是Accelerate
    - Accelerate是Huggingface生态中针对分布式训练推理提供的库，目标是简化分布式训练的流程
    - Accelerate库本身不提供分布式训练的内容，但是其内部集成了多种分布式训练框架
      - DDP、FSDP、Deepspeed等
    - Accelerate库提供了统一的接口，一套代码搞定多种分布式训练框架，简单的几行代码（4行），便可让单机训练的程序变为分布式训练程序
    - Transformers库中也是通过Accelerate集成的分布式训练，因此Accelerate库的学习非常有必要！

Accelerate 使用进阶

- **混合精度精训练**
  - 什么是混合精度训练
    - 混合精度训练是一种提高神经网络训练效率的技术，它结合了32位的单精度（FP32）浮点数和16位的半精度（FP16/BF16）浮点数来进行模型的训练。混合精度训的方法可以减少GPU内存的使用，同时加速训练过程。

Accelerate 使用进阶

- **混合精度精训练**
  - 混合精度训练中的显存占用
    - 假设模型参数量为M

	混合精度训练	单精度训练
模型	（4+2）Bytes * M	4 Bytes * M
优化器	8 Bytes * M	8 Bytes * M
梯度	（2 + ） Bytes * M	4 Bytes * M
汇总	(16 +) Bytes * M	16 Bytes * M

混合精度训练可以加速训练，但并不一定会降低显存占用！

Accelerate 使用进阶

- **混合精度精训练**
  - 混合精度训练中的显存占用
    - 假设模型参数量为M

	混合精度训练	单精度训练
模型	（4+2）Bytes * M	4 Bytes * M
优化器	8 Bytes * M	8 Bytes * M
梯度	（2 + ）Bytes * M	4 Bytes * M
激活值	2 Bytes * A	4 Bytes * A
汇总	(16 +) Bytes * M + 2 Bytes * A	16 Bytes * M + 4 Bytes * A

激活值占比较大时，可以看到明显的显存占用降低！

Accelerate 使用进阶

- **梯度积累功能**
  - 什么是梯度累积
    - 梯度累积是一种深度学习训练技术，它允许模型在有限的硬件资源下模拟更大批量大小的训练效果。
    - 梯度累积的具体做法
      - 分割Batch：将大的训练数据Batch分割成多个小的Mini-Batch。
      - 计算梯度：对每个Mini-Batch独立进行前向和反向传播，计算梯度。
      - 累积梯度：不立即更新模型参数，而是将这些小Batch的梯度累积起来。
      - 更新参数：当累积到一定数量的梯度后，再统一使用这些累积的梯度来更新模型参数

DDP 背景回顾

- **DDP 背景回顾**
  - 数据并行
    - 每个GPU里都存一份完整的模型，训练时每个GPU里的模型训练不同的数据
  - DDP 存在的问题
    - 单卡至少需要16 * M * Bytes的资源，M为模型参数量
    - 对于大模型全量参数微调，这基本是一件不可能的事情

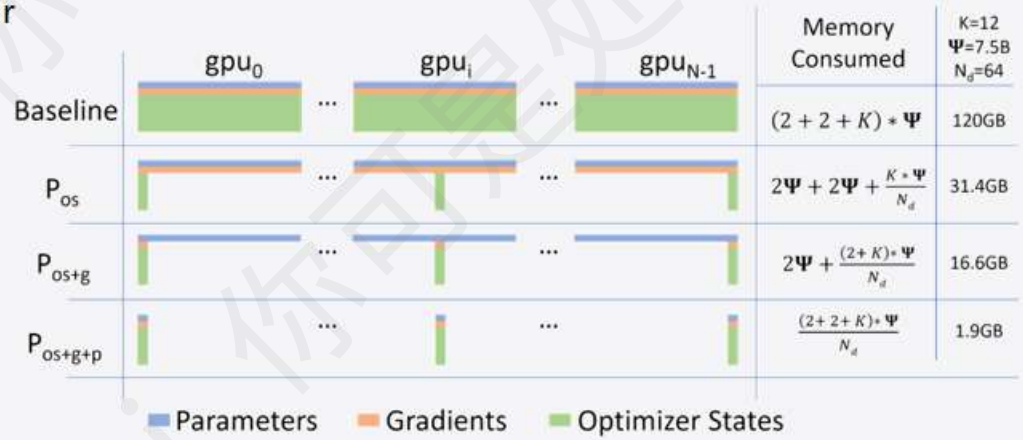
DDP 背景回顾

- **DDP 背景回顾**
  - 数据并行
    - 每个GPU里都存一份完整的模型，训练时每个GPU里的模型训练不同的数据
  - DDP 存在的问题
    - 单卡至少需要16 * M * Bytes的资源，M为模型参数量
    - 对于大模型全量参数微调，这基本是一件不可能的事情

存在冗余，N张卡上进行DDP训练，内存中需要加载N份模型，N份梯度，N份优化器

Deepspeed 介绍

- **Deepspeed 介绍**
  - Deepspeed介绍
    - DeepSpeed 是一个由微软开发的深度学习优化库，目标是使分布式训练变得简单、高效、有效。
    - ZeRO, Zero Redundancy Optimizer
      - ZeRO1, Optimizer States
      - ZeRO2, OS+Gradient
      - ZeRO3, OS+G+Parameter



Deepspeed 介绍

- **Deepspeed 介绍**
  - ZeRO策略通信量分析
    - ZeRO1, Optimizer States
      - 通信与优化器状态无关，因此与DDP相比通信总量不变
    - ZeRO2, Optimizer States+Gradient
      - 每个设备仅需要部分梯度，因此与DDP相比通信总量仍然不变
    - ZeRO3, Optimizer States+Gradient+Parameter
      - 由于需要额外的参数通信，与DDP相比通信总量提升1.5倍

Deepspeed 介绍

Deepspeed 介绍

- **Deepspeed 介绍**
  - Deepspeed介绍
    - ZeRO ++
      - 针对ZeRO3通信进行优化，通过权重量化、权重分层存储、梯度量化降低跨节点通信量
    - ZeRO offload
      - 将优化器、模型参数从显存卸载至内存，或者二者协同搭配（offload ++）
    - DeepSpeed Ulysses
      - 针对长序列训练，将各个样本在序列维度上分割给参与的GPU