小黄搞AI-LLM面试问题

目录

1.Transformer前馈神经网络用的是什么激活函数?

2.解释-下Lora的原理

3.Lora是怎样进行初始化矩阵的

4.解释-下AdaLora和QLora的原理

5.解释-下Adapter

6.介绍几种常见的Adapter

7.解释-下prefix-tuning

8.解释-下P-tuning

9.解释-下Prompt-tuning

10.解释一下混合精度的原理

11.训练的时候用float16、bfloat16还是float32，为什么?

12.怎么解决训练使用float16导致溢出的问题

13.CLM和MLM分别是什么，有什么区别?

14.GLM的自回归空白填充方法与BERT中的使用遮蔽语言模型(MLM)有什么不同，为自然语言理解(NLU)任务带来了哪些优势

15.了解迁移学习吗?大模型中是怎么运用迁移学习的?

16.Encoder-only、Decoder-only和Encoder-Decoder的模型分别有什么区别，怎么运用?

17.为什么现在的大语言模型都用Decoder-only

18.简述-下Transformer的基本结构和原理

19.Transformer为什么使用多头注意力机制?

20.Transformer为何让Q(查询)和K(键)使用独立的权重矩阵进行计算，为什么需要 Q、K、V(查询、键、值)三个矩阵?

21.为什么在attention中要进行scaled(为什么除以\sqrt{d_k})?

22.Transformer的位置编码作用及局限性?

23.为什么Transformer用LayerNorm而不用BatchNorm?

24.解释-下KVcache的原理

小红书账号：小黄搞AI

25.Transformer相比RNN、LSTM的优势何在?

26.Transformer在哪里做了权重共享?

27.介绍一下VLLM的加速方法

28.介绍-下FlashAttention的原理

29.介绍-下Tokenizer怎么训练的

30.WordPiece如何考虑语言模型的概率

31.介绍-下BPE&BBPE这两种Tokenization方法

32.介绍-下RoPE位置编码

33.旋转位置编码为什么比绝对位置编码更好

34.RMSNorm正则化有什么好处和优势

35.介绍一下Bert的结构

36.SwiGLU激活函数的原理

37.为什么大模型用GQA(Group Query Attention)

38.介绍一下Llama的结构

39.Llama模型在训练过程中如何处理梯度消失和梯度爆炸的问题

40.DeepSpeed中ZeRO系列的原理

41.解释下强化学习

42.强化学习中策略函数和值函数是什么

43.DPO的原理

44.PPO的原理

45.分析-下DPO和PPO的区别

46.DPO的正负样本怎么构造

47.DPO 的 loss 是什么?

48.OpenAl对齐为什么要用强化学习，别的方法不行吗?

49.预训练和微调任务有什么区别，两者的目的

50.SFT后会出现哪些问题

51.什么是MOE(混合专家模型)

52.Dense和MOE模型的区别

53.介绍-下Qwen通义千问的结构

54.Qwen2 有哪些提升?

55.介绍-下Llama3.1的创新

小红书账号：小黄搞AI

56.手撕自注意力机制(用Pytorch)(Self Attention)

57.手撕自注意力机制(不用Pytorch)(Self Attention)

58.手撕多头注意力机制(用Pytorch)(Multi Head Attention)

59.手撕多头注意力机制(不用Pytorch)(Multi Head Attention)

60.手撕MQA

61.手撕绝对位置编码

62.手撕RoPE

63.手撕Transformer中FFN代码

64.手撕Layer Norm

65.手撕RMSNorm

66.手撕flash-attention

67.如何缓解大模型inference的时候的重复问题

68.训练时数据长度不一致怎么办，以及如何优化训练速度69.什么是Prefill-Decode分离?为什么分离?

70.什么是PRM 过程监督奖励?OpenAl的秘密武器?

71.从huggingface下载模型时有哪些文件?

72.到底什么是端到端模型

73.什么叫长尾问题?怎么解决长尾分布问题?

74-Transformer模型中，最占用参数的是MLP层吗?

75-Transformer模型中计算量(FLOPs)的分析

76-Transformer模型中计算量(FLOPs)和参数的表格分析

77-什么是交叉熵损失函数？大模型的哪里有交叉熵损失函数?

78-什么是长度外推?

79-ALiBi (Attention with Linear Biases)思路是什么?

80-为什么需要增量预训练?

81-增量预训练的过程当中，loss上升正常吗?

82-在增量预训练过程中如何设置学习率（learning rate, LR）?

83-什么是warmup_ratio？训练过程中怎么设置?

84-Multi-head Latent Attention(多头隐变量注意力)是什么?

85-Linear Attention(线性注意力)是什么?

86-Cross-Attention(交叉注意力)是什么?

小红书账号：小黄搞AI

87-Sparse Attention（稀疏注意力）是什么?

88-不同Attention之间的区别是什么?

89-Group Normalization（GN）和 Instance Normalization（IN)的区别是什么?

90-不同的 Normalization 之间有什么区别?

91-强化学习算法是怎么分类的?

92-DeepSeekV3有啥技术特点?

93-为什么要优化KV cache?

94-MLA（Multi-Linear Attention）是什么?

95-怎么理解DPO的损失函数?

96-DPO的微调流程

97-为什么需要混合精度训练?

98-NTK长度外推方法是什么?

99-为什么推理速度受限于显存带宽?

100-PPL（Perplexity）及其数学公式

101-大海捞针测试和概率探针分别是什么?

102-DeepSeek-R1-Zero里的Zero含义?

103-DeepSeek-R1-Zero中的强化学习（RL）的原理?

104-DeepSeek-R1-Zero有什么弊端

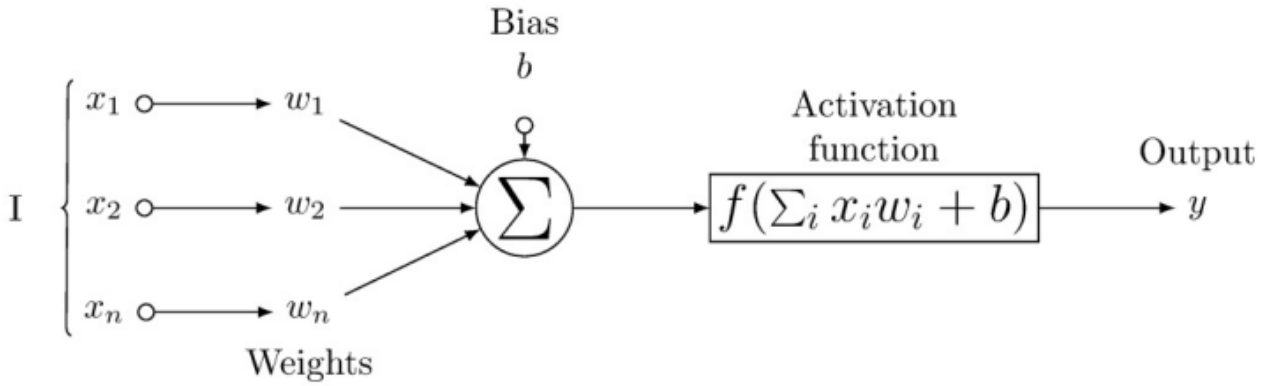
105-什么是冷启动?为什么要冷启动?

106-在DeepSeek-R1上PRM和MCTS是否有用?

107-Deepseek中蒸馏R1是什么?是否比从零RL训练是否更好?

小红书账号：小黄搞AI

1. Transformer前馈神经网络用的是什么激活函数?



1. Transformer前馈神经网络结构

Transformer中的每个编码器和解码器层除了自注意力机制外，还包含一个两层的 **前馈神经网络（Feed-Forward Neural Network, FFN）**，该网络应用在每个位置的向量上，是位置无关的。这个前馈网络通常有以下结构：

- **两层全连接网络**：第一层将输入的向量维度扩展到一个更高的维度，第二层将其缩小回原维度。
- **非线性激活函数**：在两层之间插入一个非线性激活函数来引入非线性能力。

FFN 的通用公式可以表示为：

$$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$$

其中：

- x 是输入向量。
- W_1 和 b_1 是第一层的权重矩阵和偏置向量。
- W_2 和 b_2 是第二层的权重矩阵和偏置向量。
- $\max(0, \cdot)$ 表示激活函数 ReLU（Rectified Linear Unit）

2. 激活函数选择：ReLU

在原始的Transformer模型中，前馈神经网络中使用的**激活函数是ReLU（Rectified Linear Unit）**，即：

$$\text{ReLU}(x) = \max(0, x)$$

- **ReLU的优势：**
  - **非线性能力**：通过ReLU引入非线性，使得模型可以捕捉更复杂的特征。
  - **计算效率高**：ReLU的计算非常简单，只需要取输入和零的最大值，不需要像Sigmoid或Tanh那样计算复杂的指数函数，因此它适合深层神经网络的训练。

小红书账号：小黄搞AI

- **缓解梯度消失问题**：相比Sigmoid或Tanh，ReLU在正区间的梯度为常数，避免了梯度消失的问题。

3. ReLU的使用目的

ReLU在Transformer的前馈网络中被广泛应用，原因在于它能够在两个全连接层之间引入非线性变换，帮助模型学习更复杂的特征表示。它的**计算效率高**且**性能表现良好**，尤其适合大型模型。

4. 其他激活函数的变体

尽管ReLU是原始Transformer模型中的默认激活函数，后续的模型也探索了其他激活函数，如：

> **GELU（Gaussian Error Linear Unit）**：在某些变体中，GELU也被用作替代激活函数，尤其是在更深层的模型中（如BERT）。GELU在零点附近平滑过渡，被认为在某些情况下能够带来更好的性能。

GELU 的公式是基于正态分布的累积分布函数（CDF），定义如下：

$$\text{GELU}(x) = x \cdot \Phi(x)$$

其中 $\Phi(x)$ 是标准正态分布的累积分布函数，x 是输入值。累积分布函数 $\Phi(x)$ 可以通过误差函数 $\Phi(x) = \frac{1}{2}\left[1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right]$ 来计算，其中 erf 是误差函数，定义为：$\text{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2} dt$

在实际应用中，误差函数 $\text{erf}(x)$ 通常通过查表或使用特定的数学库来计算，以提高效率和准确性。

> GELU 函数的直观解释是：它结合了线性和非线性特性，当输入值较小的时候接近于0，当输入值较大的时候接近于1，这使得它在处理不同范围的输入时具有不同的激活特性。这种特性使得 GELU 在某些情况下比传统的ReLU激活函数表现得更好。

5. 总结

- **原始Transformer**前馈神经网络中的**激活函数是ReLU**，因为它能提供简单高效的非线性能力，适合深度学习。
- **ReLU的特点**：计算简单、高效，能够缓解梯度消失问题。
- **其他变体中的激活函数**：有些后续模型（如BERT）使用了**GELU**作为替代激活函数，以实现更好的性能。

2. 解释一下Lora的原理

1. LoRA 的背景和问题

在预训练大规模模型时，微调整个模型的参数可能会带来较大的计算和存储开销。对于像 GPT-3 这样的大型模型，直接微调所有参数不仅非常耗时，还需要大量的存储资源。

2. LoRA 的原理

小红书账号：小黄搞AI

LoRA 通过引入**低秩矩阵分解**，在微调时不再调整原始模型的所有权重，而是通过将权重矩阵分解成两个较小的低秩矩阵来表示权重的变化。尽管预训练模型拥有大量的参数，**但许多参数在特定任务的微调过程中实际上是不活跃的**。LoRA通过低秩分解的方式，只更新那些对特定任务最重要的参数，这样做可以在保持模型性能的同时，显著减少微调所需的计算资源和时间。

假设某个层的权重矩阵为 $W \in \mathbb{R}^{d\times k}$ ，LoRA 不微调 W 本身，而是通过两个低秩矩阵 $A \in \mathbb{R}^{d\times r}$ 和 $B \in \mathbb{R}^{r\times k}$ 来近似微调权重矩阵。LoRA 的权重更新公式为：

$$W_{\text{new}} = W + \frac{\alpha}{r}\Delta W = W + \frac{\alpha}{r}(A \times B)$$

- W_{new} ：微调后的权重矩阵。
- W ：预训练模型的原始权重矩阵，保持不变。
- A 和 B ：低秩矩阵， $A \times B$ 近似于权重变化量 ΔW 。
- α ：一个缩放因子，用来调节低秩矩阵的影响。
- r ：秩的大小，表示低秩矩阵 A 和 B 的维度，通常 $r \ll d, k$ ，从而大大减少需要训练的参数数量。

LoRA 的核心思想是将权重的变化 ΔW 通过两个小型矩阵的乘积进行表示，**利用低秩矩阵分解的方式有效降低参数量**。这样只需要训练低秩矩阵的参数，而不是完整的权重矩阵。

3. LoRA 的优点

🎁 **高效性**：通过低秩矩阵分解，大幅减少了需要微调的参数量。在大规模预训练模型中，LoRA 仅引入一小部分新的参数，节省了存储空间和计算开销。

📍 **冻结主干模型**：LoRA 保持原始模型的主干权重 W 不变，仅调整 A 和 B 矩阵。这使得可以在多个任务之间共享同一个预训练模型，而不同任务只需要微调少量的低秩矩阵。

🌟 **通用性强**：LoRA 可以应用在模型的多种层次结构（如自注意力层、前馈网络层等），广泛适用于各类 Transformer 模型。

3. Lora是怎样进行初始化矩阵的

在 LoRA 的实现中，矩阵 B 初始化为零矩阵，矩阵 A 使用随机高斯分布初始化，这样做的主要目的是在微调开始时确保权重调整 $\Delta W = 0$ 是零矩阵，从而稳定训练过程并保持模型在初始阶段的表现与预训练模型一致。下面详细解释这种初始化方式的原因和优点：

1. 为什么初始化时需要B初始化为零矩阵

小红书账号：小黄搞AI

LoRA 的目标是在不改变预训练模型原始权重 W 的情况下，通过微调来对模型进行轻量级的适配。如果在微调一开始，矩阵 A 和 B 的初始值导致较大的权重变化（即 $\Delta W \neq 0$ ），模型的表现可能会发生突变，偏离原本的预训练模型表现，从而导致不稳定的训练过程。

通过将 B 初始化为零矩阵， $\Delta W = A \times B$ 一开始为零矩阵，这样确保了在微调开始时：

$$Wnew = W + \Delta W = W + 0 = W$$

即，初始时微调模型的权重与预训练模型的权重保持一致。**这确保了模型不会一开始就因为过大的权重变化而导致性能波动。**

2. 随机初始化A的原因

尽管 B 初始化为零，但 A 使用随机高斯分布进行初始化。这是为了确保在训练过程中，当 B 不再为零矩阵时， A **能够提供多样的参数更新方向**。具体原因包括：

📌 **增加表示能力**：随机初始化 A 可以确保它具有丰富的表示能力，使得低秩矩阵 $A \times B$ 能够探索广泛的权重调整空间。随机初始化提供了不同的起始点，从而能够在训练过程中捕捉更多的特征。

避免对称性问题：如果 A 初始为零或相同的值，可能会限制模型在调整过程中学习到多样化的特征。随机初始化打破了这种对称性，使得矩阵 A 能够在训练过程中学到更复杂的特征组合。

提供有效的梯度信息：随机初始化 A 可以防止梯度更新过程中出现梯度消失的问题。初始化时提供的非零梯度信息可以加速收敛，帮助模型更快地找到适应任务的最优参数。

3. 好处与优点

📌 **稳定性**：在训练开始时，保证权重矩阵的变化是零，能够稳定训练过程，使模型在初始阶段的表现与预训练模型一致。这样可以避免模型性能在训练初期因为过大的权重调整而大幅波动。

渐进调整：通过将 B 初始化为零矩阵， ΔW 从零开始逐渐增加，权重调整逐渐介入模型，而不是一次性强烈干扰模型的权重。这样能够让 LoRA 模块在稳定的基础上学习新的特征。

保护预训练模型的效果：因为 LoRA 的初始更新为零矩阵，这样在刚开始的几个训练步骤中，模型仍然依赖于预训练权重，保护了预训练模型在下游任务中的基础表现能力。

减少梯度爆炸或消失的风险：如果 ΔW 初始过大，可能导致梯度计算异常，进而引发梯度爆炸或消失问题。通过保证初始时 ΔW 为零，能够避免这些梯度异常情况的发生。

4. 总结

小红书账号：小黄搞AI

通过将 B 初始化为零矩阵以及将 A 随机初始化，LoRA 保证了模型在微调开始时的稳定性，并能逐渐引入权重更新。**这种初始化方式确保了模型在训练初期不会出现性能波动，同时提供了足够的参数自由度来进行有效的任务适配，从而使 LoRA 在微调过程中能够实现高效且稳定的性能提升。**

4. 解释一下AdaLora和QLora的原理

这两种技术分别解决了不同阶段的优化问题，分别在模型微调和部署上表现出色。下面将详细介绍它们的原理，并通过数学公式和用途的不同来进一步解释。

1. AdaLoRA 原理

原理简介： AdaLoRA（Adaptive Low-Rank Adaptation）通过在微调模型时将权重矩阵分解为低秩矩阵，并且引入了**自适应调整低秩（rank）的机制**。在此过程中，模型在不同层可以根据梯度信息自适应地调整分配给每层的秩值。这个机制通过动态调整秩值，在保持模型性能的同时有效减少了计算和内存需求。

矩阵分解：给定原始模型权重矩阵 W ，通过 LoRA，我们将其分解为两个低秩矩阵：

$$W \approx W_0 + \Delta W = W_0 + AB^T$$

其中， W_0 是预训练模型的固定权重， $A \in \mathbb{R}^{m \times r}$ 和 $B \in \mathbb{R}^{n \times r}$ 分别是低秩矩阵， r 是低秩值。

AdaLoRA 引入的改进是通过自适应调整 r 来使得不同层的权重矩阵分解具有不同的秩值，具体来说，使用 Frobenius 范数等指标来动态调整：

📌

$$\Delta r = \alpha \frac{\|\nabla W\|_F}{\sum_{i=1}^{L} \|\nabla W_i\|_F}$$

- Δr : 表示在某一层上，低秩分解矩阵的秩值 r 需要调整的量。
- α : 是一个调整系数，用来控制变化量的比例
- $\|\nabla W\|_F$: 表示某层权重的梯度的 $Frobenius$ 范数，即对该层的权重矩阵 W 进行梯度更新时的变化量的度量，它衡量了在当前训练步骤中，这一层的权重矩阵被更新的程度
- $\sum_{i=1}^{L} \|\nabla W_i\|_F$: 是所有层的梯度 $Frobenius$ 范数的总和，用来归一化不同层之间的影响力。这个机制确保了复杂度更高的层获得较高的秩值，复杂度较低的层使用较小的秩值。

用途和优势：

- **用途：** AdaLoRA 主要用于**模型微调**。它可以在预训练模型的基础上，通过高效的低秩适应方法，针对特定任务进行细化微调。
- **优势：** 动态调整秩值减少了资源浪费，同时保留了模型的强大性能，适合用于大模型的微调过程。

2. QLoRA 原理

小红书账号：小黄搞AI

原理简介： QLoRA（Quantized Low-Rank Adaptation）结合了量化和 LoRA 的优势。**它通过对模型进行4-bit 量化，将模型权重的精度降低，从而显著减少计算和存储需求**。随后，QLoRA 在量化后的模型上应用 LoRA 微调技术，保持低秩矩阵分解的效率和适应性。

量化公式： 量化的核心思想是将模型的权重 W 从高精度浮点数压缩到低精度表示。4-bit 量化的过程可以表示为：



$$W_q = \text{round}\left(\frac{W - W_{\min}}{W_{\max} - W_{\min}} \cdot (2^4 - 1)\right)$$

- W 表示模型中权重矩阵的值。
- $W_{\min}$ 和 $W_{\max}$ 分别是权重矩阵 W 中的最小值和最大值，它们用于对权重值进行归一化，使得所有值被压缩到一个 0 到 15（即 $2^4 - 1$）的区间内。
- $\frac{W - W_{\min}}{W_{\max} - W_{\min}}$ 将权重值归一化到 0 和 1 之间的浮点数。
- $(2^4 - 1)$ 将归一化后的浮点值映射到 0 到 15 的离散区间。
- `round` 函数则将结果舍入为整数，从而实现离散化。

NormalFloat (NF4)： NormalFloat (NF4) 是一种改进的 4-bit 量化数据类型，**采用非对称的数据表示来优化量化精度**。它将量化区间分为负数部分和正数部分，每部分分别使用 2^{k-1} 个分位数来表示，其中 k 为位数（在此为 4）。具体而言，NF4 将数据区间 $[-1, 0]$ 划分为 8 个部分，将 $[0, 1]$ 划分为 9 个部分，从而生成总共 16 个分位数。

分位数的计算：

- 分位数用于定义每个量化区间的边界。NF4 量化通过分位数函数来计算这些边界，确保量化区间内的数据分布尽可能均匀。
- 对于标准正态分布，分位数函数 $Q_X(\cdot)$ 是标准正态分布 $N(0, 1)$ 的分位数函数。

公式:



$$q_i = \frac{1}{2}\left(Q_X\left(\frac{i}{2^k + 1}\right) + Q_X\left(\frac{i+1}{2^k + 1}\right)\right)$$

- $Q_X(\cdot)$ 是标准正态分布的分位数函数。公式计算每个分位数 q_i ,提供了对数据分布的精确表示。

计算分位数的过程

- 将数据区间 $[-1, 0]$ 和 $[0, 1]$ 划分为多个部分。例如， $[-1, 0]$ 分为 8 份，而 $[0, 1]$ 分为 9 份。
- 计算每个分位数，然后将这两个区间的分位数合并，去掉重复的零，最终得到 16 个分位数。

之后，量化后的模型继续使用 LoRA 技术对权重矩阵进行低秩分解。与全精度模型相比，这种方法显著降低了存储和计算需求，同时在量化模型的基础上，LoRA 的低秩微调机制继续发挥其在适应任务上的

小红书账号：小黄搞AI

作用。

用途和优势

- **用途：** QLoRA 主要用于**模型部署**。特别是在资源有限的硬件环境下，如低显存的 GPU 或 CPU 设备，它能够将大模型在 4-bit 精度下运行，并且维持较高的任务性能。
- **优势：** 通过量化技术大幅降低内存占用，使得大模型可以在低资源环境中运行，同时使用 LoRA 微调来增强模型的适应性。

3. 用途上的区别

- **AdaLoRA 用于模型微调**： 适合用于训练阶段，尤其是在对大语言模型进行**任务特定微调**时，能够高效利用计算资源。**通过动态调整秩值，它能够针对不同任务自适应地分配计算能力**，提升微调效果。
- **QLoRA 用于模型部署**： 主要用于模型的低资源部署场景。通过 4-bit 量化，它显著减少了大模型的计算和存储需求，允许在硬件限制较多的环境中运行大模型。同时，结合 LoRA 微调，QLoRA 保证了模型的任务性能不会因量化而大幅下降。

4. 总结

- **技术核心不同**：
  - AdaLoRA 是通过**动态低秩分解**来提高微调效率，主要应用在训练阶段。
  - QLoRA 则结合了**4-bit 量化和低秩微调**，降低了部署成本，适用于大模型的资源受限环境中的部署。
- **解决问题不同**：
  - AdaLoRA 解决的是如何在模型微调过程中提升效率和性能平衡的问题，适用于复杂任务的定制化训练。
  - QLoRA 解决的是如何在低资源环境中部署大模型的问题，主要是为了减少计算和存储开销。

因此，**AdaLoRA** 和 **QLoRA** 虽然都与 LoRA 相关，但一个偏向训练中的高效性，另一个则偏向部署中的轻量化处理。

5. 解释一下Adapter

✍️ 大模型中的Adapter是一种**轻量级的模型微调技术**，旨在在大规模预训练模型的基础上，**通过插入少量可训练的参数来实现特定任务的微调**，而无需重新训练整个模型。Adapter方法有效地降低了微调大模型的计算和存储成本，同时保留了预训练模型的能力和知识。

1. 背景

小红书账号：小黄搞AI

大模型（如BERT、GPT等）经过大量无监督数据的预训练，拥有广泛的语言理解或生成能力。在下游任务中，通常需要对这些预训练模型进行微调（fine-tuning），以适应特定任务（如情感分类、机器翻译等）。传统的微调方式需要更新模型的全部参数，然而这些模型通常有数十亿甚至数千亿参数，导致微调过程的存储和计算开销巨大。此外，模型参数往往只能在特定任务之间共享，导致每个任务的微调模型都需要单独存储一份。

为了缓解这些问题，Adapter技术应运而生，通过向模型中插入小规模的可训练参数模块，只微调这些模块，而冻结原有的大量预训练参数。

2. Adapter的工作原理

Adapter通常是在Transformer架构中的每个层之间插入的一些可训练层。一个典型的Adapter模块设计如下：

输入降维

通过一个小型的全连接层将输入的特征维度**从原始的高维度降低到一个更小的维度**。

非线性激活

通常**使用ReLU或GELU等激活函数对降维后的特征进行非线性变换**。

输出升维

通过另一个全连接层**将降维后的特征重新映射回原始的高维空间**。

具体数学表示如下：

给定输入特征 $\mathbf{h} \in \mathbb{R}^d$，Adapter模块的操作可以表示为：

$$\mathbf{z} = \mathrm{ReLU}(\mathbf{h}\mathbf{W}_1) + \mathbf{h}$$

$$\mathbf{h}_{\text{out}} = \mathbf{z}\mathbf{W}_2$$

其中，$\mathbf{W}_1 \in \mathbb{R}^{d \times r}$ 和 $\mathbf{W}_2 \in \mathbb{R}^{r \times d}$ 是Adapter中可训练的参数，r 是降维后的中间维度，通常取值远小于 d，因此引入的参数量非常少。

3. Adapter的优势

参数效率高

Adapter只微调少量新引入的参数，避免了对整个大模型的微调。比如，BERT-base有1.1亿参数，而典型的Adapter模块可能只有几百万参数，微调时显著减少了需要调整的参数量。

任务间参数共享

Adapter使得不同任务的参数模块可以独立微调，但基础的预训练模型参数仍然保持共享。这种方法允许多个任务共享相同的大模型，**只需存储各个任务的Adapter参数。**

训练稳定性

由于Adapter只更新小部分参数，**相比全参数微调，训练过程更为稳定，**并且对较小的数据集也更加友好，避免了过拟合

4. Adapter的典型架构与变种

小红书账号：小黄搞AI

1. **Houlsby Adapter**：这是最早的Adapter之一，设计在Transformer的每一层注意力块和前馈网络之间插入Adapter模块。
2. **Parallel Adapter**：不同于将Adapter插入到主网络中，平行结构的Adapter与Transformer层并行工作，类似残差结构，可以在主干网络和Adapter之间进行特征融合。
3. **Compacter**：这是一种在减少参数量的基础上进一步优化的Adapter变体，使用了低秩参数分解技术来减少Adapter中的参数量。
4. **LoRA (Low-Rank Adaptation)**：与Adapter类似的思想，但LoRA通过在注意力权重矩阵上进行低秩近似来减少训练参数。LoRA也是一种插入性微调技术，主要在Transformer的自注意力模块中插入低秩矩阵更新。

5. 应用场景

Adapter在多个领域中都有广泛应用：

- **自然语言处理（NLP）**：任务包括情感分析、命名实体识别、机器翻译等。
- **跨语言模型微调**：Adapter被证明可以适应多语言环境中的任务，在不同语言之间共享大模型，微调不同语言任务的Adapter。
- **多任务学习**：Adapter能够有效实现同一个大模型在多个任务之间的共享，避免了每个任务都需要单独训练一个完整的模型。

6. 总结

> 🌟 **Adapter通过引入少量额外参数以微调预训练大模型，极大降低了微调的计算和存储成本。它为大模型的部署和任务扩展提供了一种高效、灵活的解决方案。**随着模型规模的不断增长，Adapter技术为大规模模型的微调和部署提供了重要支持。

6. 介绍几种常见的Adapter

> 📍 Adapter方法的基本思想是在预训练模型的基础上增加一些可训练的模块（Adapter模块）来实现任务微调，但这些方法并不总是和LoRA一样，直接在权重矩阵上进行融合。**不同的Adapter方法有各自的实现方式和插入位置。**

1. Houlsby Adapter

Houlsby Adapter的实现方式是在Transformer的每一层的自注意力块和前馈网络之间插入一个Adapter模块。这种设计没有直接修改原始权重矩阵，而是**通过增加新的可训练层来实现微调**。

具体结构如下：

- **Adapter模块结构：**通常包括一个降维层、非线性激活函数（如ReLU或GELU）和一个升维层。

小红书账号：小黄搞AI

- **数据流过程：**对于给定的输入 $\mathbf{h}$ ，经过自注意力和前馈网络后，再通过Adapter模块进行额外的特征变换。整个过程可以表示为： $\mathbf{h}_{\text{out}} = \text{Adapter}(\mathbf{h}) + \mathbf{h}$

Adapter模块产生的输出与原始特征进行残差连接，类似于残差网络的设计。

🥖 这种方法的特点是，Adapter模块不与原始权重矩阵直接交互，而是**作为独立的层插入，并在前向过程中通过残差连接与原始网络输出融合。**

2. Parallel Adapter

Parallel Adapter与Houlsby Adapter不同，它与Transformer的层是平行的工作方式。**它的输出与原始Transformer层的输出进行融合，有点类似于残差网络。**具体来说：

- **特征融合**：Parallel Adapter不会直接修改Transformer的层输出，而是将Adapter模块的输出与Transformer层输出相加或进行其他融合操作。
- **结构差异**：不同于直接插入的Houlsby Adapter，Parallel Adapter像是并行执行的子网络，其输出与Transformer的层输出同时存在，然后通过融合策略（如加权平均）得到最终输出。

🏖️ 这种方法依然没有对原始的权重矩阵进行低秩分解或直接修改，而是通过增加并行的结构来辅助特征学习。

3. Compacter

Compacter是Adapter方法的一种优化变体，**它通过低秩分解技术减少了Adapter模块的参数量**。具体而言，它通过将Adapter模块的权重矩阵表示为几个低秩矩阵的乘积来实现参数压缩：

- **低秩分解**：类似于LoRA的思想，Compacter对Adapter模块内部的权重矩阵进行低秩分解，即将一个大的矩阵拆分为两个或多个小的矩阵，这样可以减少训练参数量。
- **参数更新方式**：虽然使用了低秩分解，但它仅限于Adapter模块内部的权重更新，并不会直接去修改Transformer原始的权重矩阵。也就是说，Compacter对原有的大模型参数是保持冻结的，只对插入的低秩结构进行训练。

4. 总结

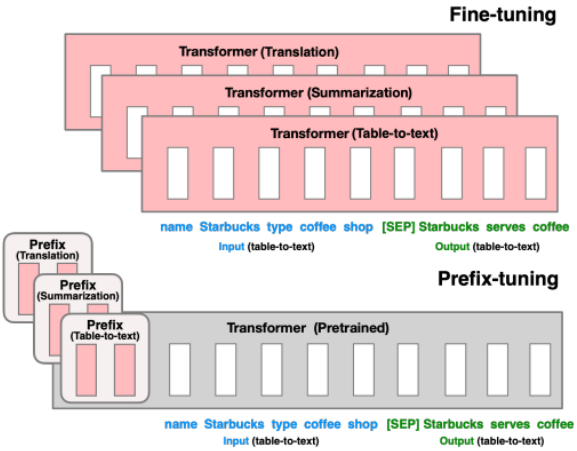
💫 虽然一些Adapter变体（如Compacter）会使用低秩分解技术来减少参数量，但传统的Adapter方法（如Houlsby和Parallel Adapter）主要是通过添加新的层结构来微调模型，而不是直接在原始权重矩阵上进行操作和融合。因此，传统的Adapter方法并不像LoRA那样训练一个低秩矩阵用于与原始的权重矩阵直接融合。

7. 解释一下prefix-tuning

小红书账号：小黄搞AI

💡 **Prefix-tuning**是一种参数高效的微调方法，主要用于自然语言处理任务。它通过在预训练语言模型（如GPT-2、T5等）的**输入序列前面添加一段可训练的前缀（prefix）**，实现对下游任务的适应，而无需更新原有的大量模型参数。

1. 来源和背景



Prefix-tuning的概念出自2021年Li and Liang的论文《Prefix-Tuning: Optimizing Continuous Prompts for Generation Tasks》[1]。

这篇文章提出了在保持预训练语言模型的权重不变的前提下，通过**学习一组连续的前缀向量来微调模型，以适应下游生成任务**。这种方法解决了全参数微调的计算和存储开销问题，同时提供了一种参数高效的模型调整方式。

> [1]: Li, X., & Liang, P. (2021). Prefix-Tuning: Optimizing Continuous Prompts for Generation Tasks. *arXiv preprint arXiv:2101.00190*.

2. 主要观点

💡 Prefix-tuning的主要观点是在输入序列的前面插入一段可训练的前缀，利用这段前缀的特征来引导模型的输出。与全参数微调或其他参数高效的微调方法（如Adapter、LoRA）不同，**prefix-tuning不直接改变模型的内部结构或权重，而是通过对输入进行扩展，以较小的参数开销适应下游任务。**

主要优点包括：

参数效率高：仅微调前缀部分的参数，而冻结预训练模型的所有参数。

训练效率高：由于前缀的参数量远小于整个模型的参数量，训练过程更快。

迁移性强：前缀可以被视为一种可迁移的提示，可应用于多种任务。

3. 技术细节

Prefix-tuning通过在Transformer的输入部分添加一段“可训练的前缀”，这段前缀由若干个连续的向量组成，并且这些向量被看作是额外的“虚拟”输入。具体步骤如下：

1. **前缀构建**
   - 给定一个语言模型的层数为 L，每一层都包含一个自注意力模块和一个前馈网络。

小红书账号：小黄搞AI

- **Prefix-tuning在每一层的自注意力模块的输入前面增加一组可训练的前缀** $\mathbf{P} \in \mathbb{R}^{m \times d}$，其中 m 是前缀长度，d 是模型的隐藏维度。
- 这些前缀向量会在整个训练过程中被优化。

2. **注意力机制中的应用**
   - **这些前缀向量被视为额外的“虚拟token”，与真实的输入token一起通过注意力计算。**
   - 在注意力得分计算中，前缀会影响后续token的注意力权重，从而对生成的输出产生影响。
3. **微调过程**
   - **仅优化这些前缀向量的参数，而冻结原始模型的权重。**这样可以显著减少需要训练的参数量，尤其是在处理大模型时（如GPT-3、T5-11B等），这种方法表现出较高的参数效率。
4. **任务适应**
   - 在任务微调阶段，前缀向量被初始化为随机向量，然后随着训练迭代不断更新，使其能够适应特定任务的需求。**最终，这些向量在模型生成任务中起到类似“指导”的作用。**

4. 数学表述

假设输入序列的表示为 $\mathbf{X} \in \mathbb{R}^{n \times d}$，其中 n 是输入序列长度，d 是隐藏维度。Prefix-tuning的做法是将前缀 $\mathbf{P}$ 与 $\mathbf{X}$ 拼接起来，得到扩展后的输入：

$$\mathbf{X}_{\text{ext}} = [\mathbf{P}; \mathbf{X}] \in \mathbb{R}^{(m+n) \times d}$$

此扩展后的输入 $\mathbf{X}_{\text{ext}}$ 会被传递给Transformer的每一层进行处理。

5. 优势与局限

优势	局限
• 大大减少了微调参数量。例如，在GPT-2模型上，Prefix-tuning只需优化不到0.1%的参数。	• **对于某些任务，前缀长度的选择较为关键，过短可能不足以引导模型完成任务，而过长则会增加训练成本。**
• 不改变原始预训练模型的参数，适用于需要频繁切换任务的多任务学习场景。	• 在一些特定任务上，Prefix-tuning的表现可能不如全参数微调，因为它只能影响输入特征，而无法修改模型的内部参数。

6. 应用场景

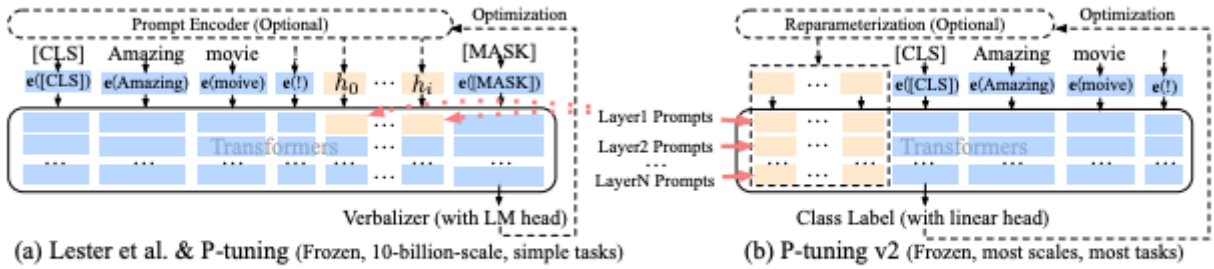
Prefix-tuning主要用于**文本生成任务**（如机器翻译、文本摘要、对话生成等），也可以用于一些分类任务。在多任务和跨领域的应用场景中，prefix-tuning表现出色，因为它能将预训练模型的能力快速转移到新任务上。

7. 结论

小红书账号：小黄搞AI

💡 Prefix-tuning是一种新颖的微调方法，通过在输入序列前添加可训练的前缀，实现参数高效的下游任务适应。其在大规模预训练模型的生成任务中展示了出色的性能，是现代微调技术中的一种重要方法。

8. 解释一下P-tuning



(a) Lester et al. & P-tuning (Frozen, 10-billion-scale, simple tasks) (b) P-tuning v2 (Frozen, most scales, most tasks)

Figure 2: From Lester et al. (2021) & P-tuning to P-tuning v2. Orange blocks (i.e., $h_0, ..., h_i$) refer to trainable prompt embeddings; blue blocks are embeddings stored or computed by frozen pre-trained language models.

💡 **P-tuning**是一种参数高效的微调方法，专注于通过优化连续的提示（prompt）向量来提高预训练语言模型在下游任务中的性能。与传统的离散提示工程（使用自然语言作为提示）不同，P-tuning采用连续可优化的嵌入向量作为提示，以提升模型的适应能力。

> Liu, X., Zheng, Y., Du, Z., Ding, M., Qian, Y., Yang, Z., & Tang, J. (2021). GPT Understands, Too. *arXiv preprint arXiv:2103.10385*.
>
> Liu X, Ji K, Fu Y, et al. P-tuning v2: Prompt tuning can be comparable to fine-tuning universally across scales and tasks[J]. arXiv preprint arXiv:2110.07602, 2021.

1. 来源和背景

📌 P-tuning的概念源自2021年论文《GPT Understands, Too》。该论文提出通过在输入文本的前面插入一段可训练的提示向量，使得大规模预训练语言模型（如GPT）在特定任务（如分类或文本生成）中更好地表现。P-tuning是一种泛化的“提示（prompt）”方法，相较于手工设计的提示词，它可以更灵活地适应任务需求，并且能够显著提升模型的性能。

2. 主要观点

P-tuning的核心思想是在任务输入序列的前面插入一段可训练的连续提示向量。这些提示向量在训练过程中不断优化，以调整语言模型在特定任务中的生成或理解行为。与传统的参数高效微调方法（如Adapter或prefix-tuning）相比，P-tuning不改变模型的结构或权重，而是通过优化前缀（提示向量）的表示来影响模型的行为。

主要观点包括：

小红书账号：小黄搞AI

连续提示优化： P-tuning直接对嵌入层的提示向量进行训练，使提示能够引导模型在下游任务中产生更好的输出。

更高的灵活性： 相比于离散的提示词（使用自然语言作为提示），连续提示向量可以表示更复杂的特征，使其适应性更强。

与下游任务的融合： P-tuning可以被用于多种任务，例如文本分类、序列标注和生成任务，通过引导模型输出与任务目标更匹配的结果。

3. 技术细节

P-tuning的实现过程包括以下几个关键步骤：

1. **连续提示的构建：**
   - 给定一个预训练语言模型（如GPT、BERT等），P-tuning会在输入序列的前面插入一段可训练的提示向量 $\mathbf{P} \in \mathbb{R}^{m \times d}$，其中 m 是提示向量的长度，d 是嵌入层的维度。
   - 这些提示向量被初始化为随机或特定的初值，并在训练过程中不断优化。
2. **输入序列扩展：**
   - 将提示向量 $\mathbf{P}$ 与原始任务输入 $\mathbf{X} \in \mathbb{R}^{n \times d}$ 拼接起来，得到扩展的输入序列 $\mathbf{X}_{\text{ext}} = [\mathbf{P}; \mathbf{X}] \in \mathbb{R}^{(m+n) \times d}$，其中 n 是原始输入序列的长度。
   - 扩展后的输入序列被传递给语言模型的嵌入层，并继续通过模型的所有层进行处理。
3. **参数更新：**
   - 在训练过程中，只有提示向量 $\mathbf{P}$ 的参数会被优化，而预训练模型的参数保持冻结。这种方法显著减少了训练时需要优化的参数量。
4. **多层提示设计（P-tuning v2）：**
   - P-tuning v2进一步扩展了P-tuning的思想，通过在Transformer模型的不同层插入提示向量。具体来说，在多层Transformer中，每一层都可以插入可训练的提示，以更细粒度地影响不同层的特征表示。

4. 数学表述

假设原始输入为 $\mathbf{X} = [x_1, x_2, \ldots, x_n]$，其中 x_i 表示第 i 个输入token的嵌入向量。P-tuning在模型输入前插入提示向量 $\mathbf{P} = [p_1, p_2, \ldots, p_m]$，形成扩展后的输入：

$$\mathbf{X}_{\text{ext}} = [p_1, p_2, \ldots, p_m, x_1, x_2, \ldots, x_n]$$

其中，提示向量 $\mathbf{P}$ 的维度与嵌入层的维度相同。通过训练过程不断优化这些提示向量，使其能够有效地引导模型进行特定任务。

5. 优势与局限

优势	局限

小红书账号：小黄搞AI

- **参数效率高**：只需训练少量提示向量的参数，可以显著减少微调的计算开销。
- **更好的任务适应性**：连续提示向量的表示能力强于手工设计的离散提示词，能够更灵活地适应不同任务。
- **应用广泛**：可用于分类、序列标注、生成等多种自然语言处理任务。
- **提示长度的选择较为关键**：提示向量的长度需要进行调优，不同任务可能需要不同的提示长度。
- **与任务数据的依赖较大**：在一些特定的任务上，P-tuning可能不如全参数微调的效果。

6. 结论

📌 P-tuning是一种灵活的参数高效微调方法，通过对连续提示向量的优化，可以高效地引导预训练语言模型适应不同的下游任务。它为大规模模型的低资源适应和多任务学习提供了一种高效的解决方案，是现代自然语言处理任务中的一种重要技术。

9. 解释一下Prompt-tuning

💫 **Prompt-tuning**是一种专注于优化提示（prompt）来提升预训练语言模型在下游任务中性能的技术。它通过对提示部分进行训练，而不修改模型参数，从而实现对大规模语言模型的高效适应。Prompt-tuning能够以较少的参数调整，获得接近甚至超过传统微调方法的性能，尤其适用于大规模预训练模型（如GPT-3、T5等）。

1. 背景和来源

Prompt-tuning的概念源于利用预训练语言模型的填空能力来解决下游任务。例如，对于文本分类，可以将输入转换为一种填空问题，如“这篇文章的情感是[MASK]”。Prompt-tuning进一步发展了这一思想，通过训练一个可优化的提示，使得提示能够引导模型更好地完成特定任务。

Prompt-tuning的思想最早可以追溯到2021年由Lester等人提出的《The Power of Scale for Parameter-Efficient Prompt Tuning》。这篇文章展示了Prompt-tuning在大规模模型（尤其是100亿参数以上模型）中的有效性，并提出了使用可训练的连续提示向量代替离散文本提示。

> [1]Lester, B., Al-Rfou, R., & Constant, N. (2021). The Power of Scale for Parameter-Efficient Prompt Tuning. *arXiv preprint arXiv:2104.08691*.

2. 主要观点

Prompt-tuning的核心思想是在输入序列前插入一个可训练的提示（prompt），这些提示不再是离散的文本，而是连续的嵌入向量。通过对提示向量进行训练，可以有效引导模型输出更符合任务目标的结果。主要观点包括：

小红书账号：小黄搞AI

1. **参数高效：** 仅优化提示向量，而保持模型的其他参数冻结。相比传统的微调方法，Prompt-tuning在参数更新上具有显著的优势。
2. **大模型适应性强：** Prompt-tuning在大规模预训练模型（如数十亿参数模型）上表现尤为出色，能够充分利用大模型的知识进行任务适应。
3. **任务转换能力强：** 通过修改或优化提示，可以在多个任务之间快速切换，而无需对每个任务都进行全参数训练。

3. 技术细节

Prompt-tuning的实现过程包括以下几个关键步骤：

1. **连续提示的初始化：**
   - 在传统的Prompt工程中，提示通常是手工设计的自然语言文本。Prompt-tuning将这些提示替换为连续的可优化向量，即可训练的嵌入。
   - 这些提示向量的长度m由用户指定，提示的维度d等于模型的嵌入层维度。
   - 提示向量可以被随机初始化或者从现有的嵌入向量中初始化。
2. **输入序列的构造：**
   - 给定一个任务输入序列 $\mathbf{X} = [x_1, x_2, \ldots, x_n]$ ，其中 x_i 表示第 i 个token的嵌入向量，Prompt-tuning在输入序列的前面添加提示向量 $\mathbf{P} = [p_1, p_2, \ldots, p_m]$ 。
   - 扩展后的输入序列表示为 $\mathbf{X}_{\text{ext}} = [p_1, p_2, \ldots, p_m, x_1, x_2, \ldots, x_n]$ 。
3. **冻结模型参数，仅训练提示向量：**
   - 在Prompt-tuning的训练过程中，保持预训练模型的所有参数不变，只对提示向量的参数进行优化。
   - 优化目标通常是任务相关的损失函数，如分类任务的交叉熵损失或生成任务的语言模型损失。
4. **在生成任务中的应用：**
   - Prompt-tuning不仅可以应用于分类任务，还适用于各种生成任务（如机器翻译、文本摘要）。
   - 在这些任务中，提示向量能够引导生成模型朝着特定的方向进行输出，使得生成结果更符合任务需求。

4. 数学表述

假设给定输入序列 $\mathbf{X}$ ，Prompt-tuning的输入扩展为 $\mathbf{X}_{\text{ext}} = [\mathbf{P}; \mathbf{X}]$ ，其中 $\mathbf{P}$ 是长度为 m 的提示向量。Prompt-tuning的目标是优化提示向量 $\mathbf{P}$ 的参数，使得模型的输出能够更好地完成任务。训练过程中优化的目标可以表示为：

$$\min_{\mathbf{P}} \mathcal{L}(f(\mathbf{X}_{\text{ext}}; \theta), y)$$

其中 $\mathcal{L}$ 表示损失函数， f 为语言模型， θ 为冻结的预训练模型参数， y 为任务标签。

小红书账号：小黄搞AI

5. 优势与局限

优势	局限
• **参数效率高**：仅需优化提示向量，显著减少了需要训练的参数量。 • **迁移性强**：可以快速适应不同任务，尤其是在大规模模型中效果更为明显。 • **适用于大规模模型**：Prompt-tuning在大规模模型中可以充分发挥其优势，因为大模型具有更强的泛化能力和知识表示能力。	• **依赖提示长度**：提示向量的长度m需要调优，不同任务可能对提示长度有不同的需求。 • **小模型效果有限**：在小规模预训练模型上，Prompt-tuning的效果可能不如全参数微调。 • **对复杂任务的适应性**：Prompt-tuning对于一些复杂的任务可能需要进一步的改进，如结合其他参数高效的微调方法。

6. 结论

💫 Prompt-tuning是一种灵活的参数高效微调方法，通过优化连续提示向量引导预训练模型在下游任务中实现更优的表现。它为大模型在多任务场景和低资源任务中的高效适应提供了新的途径，是现代NLP任务中的重要技术。

10. 解释一下混合精度的原理

混合精度（Mixed Precision）训练是一种通过结合不同数值精度（如FP32和FP16）进行深度学习模型训练的方法，旨在提高计算效率并减少内存占用，同时在保持模型精度的前提下优化训练性能。其核心思想是，模型中的某些操作对数值精度要求较高，而另一些操作可以使用较低精度完成，从而在性能与精度之间找到平衡。

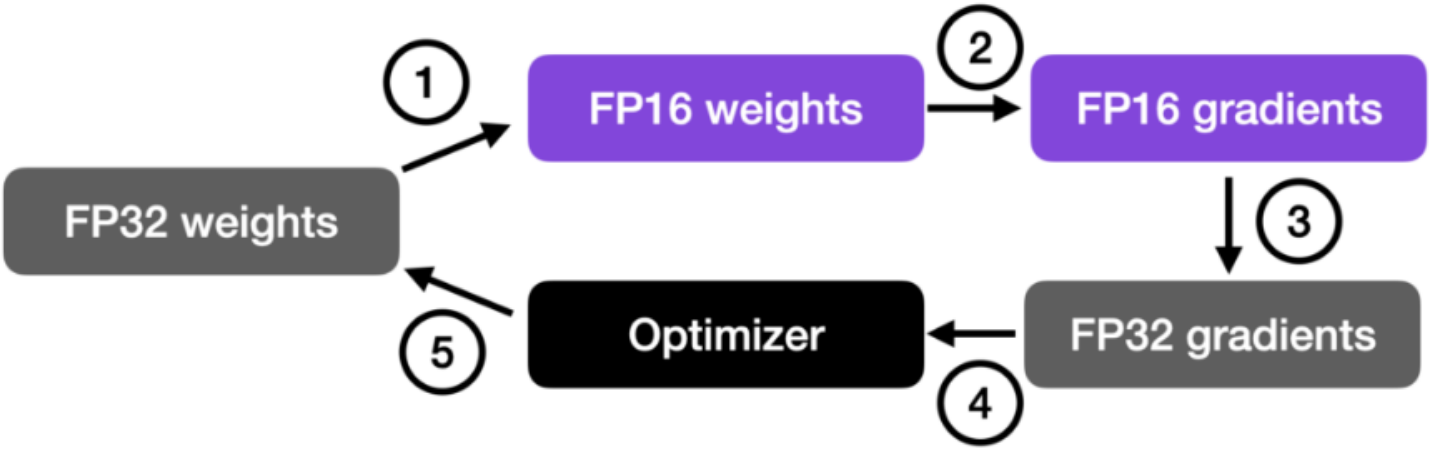
1. 浮点数精度背景

在深度学习中，常见的浮点数精度包括：

- `FP32（32位浮点数）`：通常用于深度学习训练，**提供较高的数值精度，但计算较慢**，内存开销较大。
- `FP16（16位浮点数）`：计算速度更快、**内存占用更少，但表示的数值范围较小**，容易发生数值下溢或上溢问题。

2. 混合精度的原理与使用流程

小红书账号：小黄搞AI



混合精度训练的核心流程是根据不同操作的需求，**在前向传播、反向传播和权重更新中分别使用FP16和FP32来最大化训练效率**，具体步骤如下：

1. **初始化**：模型的神经网络权重参数最初以**FP32**形式存储。
2. **权重转换**：将神经网络的权重参数从**FP32转换为FP16**，用于后续的计算。
3. **前向传播与反向传播**：在前向和反向传播计算中**使用FP16**完成主要的矩阵运算和激活函数计算，以减少内存占用并加速计算。反向传播过程中，梯度也使用FP16进行计算。
4. **梯度转换**：将FP16的梯度转换回FP32，确保在梯度更新过程中使用更高精度的计算，避免精度丢失。
5. **梯度更新**：FP32的梯度与学习率（learning rate）相乘，保证精度较小的数值不会因浮点数的限制而丢失。
6. **权重更新**：使用更新后的FP32梯度更新网络的FP32权重。

3. 损失缩放（Loss Scaling）

在混合精度训练中，由于FP16的数值表示范围较小，特别是在反向传播过程中，较小的梯度值可能会因为精度不足而变为零。为了解决这个问题，引入了**损失缩放**技术。在反向传播前，将损失值按某个系数进行放大，确保反向传播时梯度值足够大，避免FP16下的数值下溢。计算完成后，再将梯度缩小相应倍数，以保证数值正确性。

4. 混合精度的性能提升

混合精度训练通过合理结合FP32和FP16，能够有效提升训练效率，具体优势包括：

- **更快的计算速度**：FP16的计算速度通常比FP32快一倍或更多，尤其在支持混合精度计算的硬件上（如NVIDIA的Tensor Cores），这一加速效应尤为明显。
- **更高的内存利用率**：FP16使用更少的内存，这使得在相同的硬件环境下可以训练更大的模型或使用更大的batch size，从而提高了训练效率。

5. 潜在问题与解决方案

小红书账号：小黄搞AI

- **数值不稳定性**：FP16的精度较低，容易发生数值下溢或上溢问题，尤其是在梯度较小时。引入损失缩放机制能够有效防止这种问题发生。
- **硬件要求**：混合精度训练需要特定硬件（如NVIDIA Tensor Cores）和软件（如CUDA和cuDNN）的支持。幸运的是，主流的深度学习框架如PyTorch和TensorFlow已经原生支持混合精度训练。

6. 实际应用

混合精度训练广泛应用于大规模模型训练任务中，特别是需要大批量数据和复杂网络的场景，如BERT、GPT系列模型等。在计算密集型硬件上，通过混合精度技术能够显著提高模型训练的性能，同时减少内存开销，使得在相同资源下能够训练更大的模型或处理更大的数据集。

7. 总结

混合精度通过结合FP32的精度和FP16的计算效率，既确保了模型的数值稳定性，又在训练速度和内存占用方面带来了显著的优化，是当前深度学习模型训练中的一种关键技术。损失缩放技术的引入进一步确保了FP16计算下的数值精度，从而实现了兼具精度与性能的平衡。

11. 训练的时候用float16、bfloat16还是float32，为什么?

🦄 在训练大型模型时，选择数据类型 (float16、bfloat16 或 float32) 是影响计算精度、内存使用和计算效率的重要因素。以下是对这几种数据类型的详细解释，以及它们在训练中的优势与应用场景。

1. float32

- **概念：** float32（32位浮点数）具有较高的精度，包含**8位指数和23位尾数**。
- **优点**
  - **适用于需要极高精度的计算**，例如模型参数更新等。
  - **能有效避免数值下溢和上溢**，因此稳定性较高。
- **缺点**
  - 内存占用较大，特别是在训练大模型时，**可能导致显存不足**。
  - 计算速度较慢，因为它**需要更多的存储空间和带宽**。
- **应用场景**
  - 通常用于**模型训练的初始阶段或者极度敏感的参数（如某些权重矩阵的更新）**，尤其适合在小模型或高精度模型调优阶段。

2. float16

- **概念：** float16（16位浮点数）仅有**5位指数和10位尾数**，因此它的数值范围和精度都较低。

小红书账号：小黄搞AI

- **优点**
  - 占用的内存和带宽仅为float32的一半，**大幅减少了显存需求**。
  - 计算速度更快，特别是在支持半精度浮点运算的硬件（如NVIDIA的Tensor Cores）上，float16能**显著提高计算效率**。
- **缺点**
  - 数值范围较小，**容易出现下溢和上溢问题**。
  - 由于精度较低，在某些深度模型（如多层网络或需精确计算的任务）中可能不适用，**容易影响训练稳定性**。
- **应用场景**
  - 常用于需要**降低显存使用和提高计算速度的场景**，尤其是大型模型的中间层或加速训练阶段。

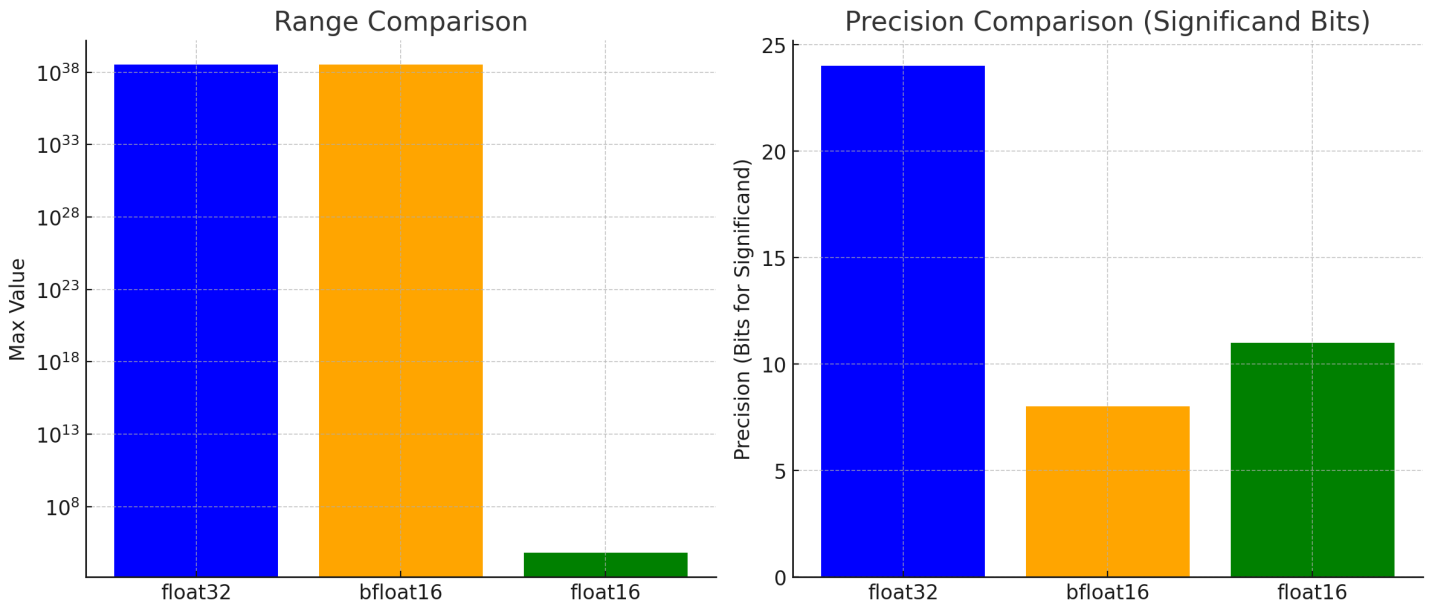
3. bfloat16

- **概念：** bfloat16（16位浮点数）与float16类似，但其**8位指数和7位尾数使得它具有float32的数值范围**，但精度比float32低。
- **优点**
  - 兼具float32的数值范围和float16的存储优势，能够**在低精度存储的情况下提供更好的数值稳定性**。
  - **计算速度和内存占用都比float32小**，在某些硬件（如TPU）中具有良好的支持。
- **缺点**
  - 相较float32精度较低，**可能导致某些极度依赖数值精度的模型出现轻微误差**。
- **应用场景**
  - 适合在深度学习训练中替代float32，尤其是**需要平衡计算效率和数值稳定性的情况下**，如Google TPU环境下的大模型训练。

4. 示意图

float32、bfloat16和float16在数值范围和精度上的差异：

小红书账号：小黄搞AI



范围比较

float32和bfloat16具有相同的数值范围，因为它们都使用8位指数部分，这使得bfloat16能够处理与float32相当的数值范围，降低了下溢和上溢的风险。

而**float16的数值范围较小，容易受到数值溢出影响**。

精度比较

float32的精度最高（23位尾数），适合需要高精度的计算任务。float16的精度次之（10位尾数），在保持范围的同时牺牲了一些精度，但通常仍能满足大多数深度学习任务的需求。bfloat16的精度（7位尾数）最低，但由于其更低的存储需求，适合对精度要求较低且需要计算加速的任务。

🦄 在实际训练中，选择适合的数据类型可以帮助在计算精度与资源使用之间达到良好的平衡。

12. 怎么解决训练使用float16导致溢出的问题

🐠 在深度学习模型的训练过程中，计算资源和训练速度是两个重要的考量因素。**使用较低精度的浮点数（如 float16）可以减少内存占用和提高计算速度。然而，float16 的数值范围和精度较小，可能导致数值溢出（overflow）或下溢（underflow），从而影响模型的训练效果。**

1. float16 溢出问题的原因

浮点数的表示范围

1. **float32（单精度浮点数）**：其范围是约 -3.4×10^{38} 到 3.4×10^{38} 。
2. **float16（半精度浮点数）**：其范围是约 -6.55×10^{4} 到 6.55×10^{4} 。

小红书账号：小黄搞AI

float32（单精度浮点数）

- 指数位宽：8位
- 尾数位宽：23位

float16（半精度浮点数）

- 指数位宽：5位
- 尾数位宽：10位

由于 float16 的指数位和尾数位较少，其能表示的数值范围大大缩小。这在训练深度学习模型时，尤其是在反向传播过程中，可能会导致梯度的溢出或下溢。

2. 解决溢出问题的方法——使用混合精度训练

概念：混合精度训练是在模型中同时**使用多种数值精度（如 float16 和 float32）来兼顾计算效率和数值稳定性。**

实现方法

- **参数和激活值使用 float16**：降低内存占用，加快计算速度。
- **累加和关键计算使用 float32**：提高数值稳定性，防止溢出。

优势

- **提高训练速度**：利用 GPU 对 float16 的优化。
- **减少内存占用**：允许训练更大的模型或使用更大的批量大小。

示意图

```
[输入数据] --(float16)--> [前向传播] --(float16)--> [损失计算]
                                  |
                              (梯度计算)
                                  |
                          [反向传播累加] --(float32)
```

3. 解决溢出问题的方法——损失缩放（Loss Scaling）

原理：通过在计算损失前将其放大一个缩放因子，在反向传播时再将梯度缩小相同的因子，避免梯度下溢。

类型

- **静态损失缩放**：使用固定的缩放因子。
- **动态损失缩放**：根据训练过程动态调整缩放因子。。

实现步骤

1. **损失放大**：将计算的损失乘以缩放因子（如 2^{15} ）。
2. **反向传播**：计算放大后的梯度。
3. **梯度缩小**：在更新参数前，将梯度除以缩放因子。

小红书账号：小黄搞AI

示意图

```
[损失] x [缩放因子] --> [放大后的损失]
                |
[放大后的损失] --(反向传播)--> [梯度]
                |
[梯度] / [缩放因子] --> [缩小后的梯度]
```

4. 解决溢出问题的方法——使用 bfloat16 代替 float16

特点

- **bfloat16（脑浮点16位）**：具有与 float32 相同的指数位数，但尾数位减少。
- **数值范围**：与 float32 相同，但精度较低。

优势

- **更大的数值范围**：减少溢出和下溢的可能性。
- **计算效率**：接近 float16 的计算性能。

5. 解决溢出问题的方法——改进数值稳定性

- **选择数值稳定的激活函数：** 如使用 **Leaky ReLU** 或 **ELU** 代替 ReLU，避免梯度为零。
- **正则化技巧：** 如批量归一化（Batch Normalization），稳定梯度分布。
- **梯度裁剪：** 在反向传播时限制梯度的最大范数，防止梯度爆炸。

6. 解决溢出问题的方法——调整超参数

- **学习率调整**：降低学习率可以减少梯度值的大小，防止溢出。
- **批量大小的选择**：适当减小批量大小，降低每次更新的梯度值。
- **权值初始化**：使用合适的初始化方法（如 He 初始化），防止前期训练中的数值异常。

7. 结论

在使用 float16 进行模型训练时，溢出问题主要源于其有限的数值表示范围。通过混合精度训练、损失缩放、使用 bfloat16、改进数值稳定性和调整超参数等方法，可以有效解决或缓解溢出问题。**在实践中，需要根据具体的模型和数据选择合适的方法，并注意在性能和稳定性之间取得平衡。**

13. CLM和MLM分别是什么，有什么区别?

1. CLM（Causal Language Modeling）：

小红书账号：小黄搞AI

- **定义**：CLM 是**因果语言建模**，训练模型预测一句话中的下一个词。每个词只能利用它前面的上下文信息。
- **应用**：它用于生成任务，例如 GPT 模型，模型按顺序生成句子的每一个单词或字符。
- **训练方式**：在训练过程中，模型只能看到到当前单词之前的所有词，不能看到之后的词。这是一种自回归模型。

2. MLM（Masked Language Modeling）：

- **定义**：MLM 是**掩码语言建模**，训练模型预测在一个句子中被随机掩盖（masked）的词。模型可以使用整个句子的上下文，包括掩盖词的前后内容。
- **应用**：它主要用于表示学习任务，例如 BERT 模型，帮助模型更好地理解整个句子的语义。
- **训练方式**：**在训练过程中，句子中的一些词会被随机掩盖**，模型的任务是根据其余词来预测被掩盖的词。这是一种双向建模。

3. 主要区别：

- **预测目标**：CLM 只预测下一个词，而 MLM 是填补被掩盖的词。
- **上下文使用**：CLM 只能使用之前的词，MLM 则可以使用整个句子的上下文（前后词）。
- **模型类型**：CLM 是自回归的，用于生成任务，MLM 是双向的，用于表示学习任务。
- **来源**：“因果语言建模”就是上一章中说的统计语言模型，只使用前面的词来预测当前词，由 NNLM 首次运用；而“遮盖语言建模”实际上就是 Word2Vec 模型提出的 CBOW。

14. GLM的自回归空白填充方法与BERT中的使用遮蔽语言模型（MLM）有什么不同，为自然语言理解（NLU）任务带来了哪些优势

在自然语言处理领域，预训练语言模型是提升模型性能的关键技术。BERT和GLM是两种常见的预训练模型，它们在预训练任务上采用了不同的方法：BERT使用了遮蔽语言模型（Masked Language Modeling，MLM），而GLM采用了自回归的空白填充方法。理解这两种方法的区别以及GLM在自然语言理解任务中的优势，对于深入掌握预训练模型的原理和应用具有重要意义。

关键概念对照表

概念	解释
预训练语言模型	在大规模文本上训练的语言模型，用于下游任务
自编码模型	通过重建输入来学习表示，如BERT

小红书账号：小黄搞AI

自回归模型	基于先前输出预测下一个元素，如GPT
遮蔽语言模型（MLM）	随机遮蔽输入中的词，模型预测被遮蔽的词
空白填充方法	在输入中引入空白，模型自回归地填充空白

1. 基础概念解释

预训练语言模型

预训练语言模型**通过在大规模无监督文本数据上进行训练，学习语言的语法和语义信息**。然后，可以将预训练得到的模型应用于各种下游任务，如文本分类、问答系统等，以提高模型的性能。

自编码模型与自回归模型

- **自编码模型（Autoencoding Models）**：如BERT，使用双向上下文信息，通过重建被遮蔽的输入来学习语言表示。
- **自回归模型（Autoregressive Models）**：如GPT，使用单向上下文信息，基于前面的词预测下一个词。

2. BERT的遮蔽语言模型（MLM）

工作原理

BERT的MLM方法是在输入序列中随机遮蔽一些词，然后让模型根据上下文预测被遮蔽的词。

示意图

```
输入序列："我爱[MASK]天"
模型预测："我爱北京天"
```

特点

- **双向性**：模型可以同时利用左侧和右侧的上下文信息。
- **信息泄露问题**：在训练过程中，模型可以看到被遮蔽词的未来信息，这在生成任务中不适用。

3. GLM的自回归空白填充方法

工作原理

GLM的自回归空白填充方法是在输入序列中引入空白，让模型以自回归的方式逐步填充这些空白。

示意图

```
输入序列：        "我爱 [ ] 天"
填充过程：
```

小红书账号：小黄搞AI

```
步骤1：预测"北" => "我爱 北 [ ] 天"
步骤2：预测"京" => "我爱 北京 天"
```

特点

- **自回归填充**：模型以自左向右的方式，逐步预测并填充空白。
- **统一的生成和理解**：同时适用于自然语言理解和生成任务。

4. 两种方法的区别

训练目标

BERT的MLM：重建被遮蔽的词，训练目标是最大化被遮蔽词的概率。

GLM的自回归填充：预测空白处的词序列，训练目标是最大化空白填充序列的概率。

上下文利用

BERT：利用双向上下文，但存在信息泄露的问题。

GLM：以自回归方式填充，避免了信息泄露。

应用范围

BERT：主要用于理解任务，如文本分类、问答匹配等。

GLM：兼顾理解和生成任务，如机器翻译、文本摘要等。

5. GLM在自然语言理解任务中的优势

更好的上下文建模

GLM通过自回归的方式，能够更好地捕捉上下文之间的依赖关系，提升对长距离依赖的建模能力。

统一的框架

GLM采用统一的预训练任务，兼容了自编码和自回归模型的优势，使其在多种下游任务中表现出色。

避免信息泄露

自回归的填充方式确保模型在预测时只利用过去的信息，避免了MLM方法中可能存在的信息泄露问题。

提升模型泛化能力

由于GLM在预训练时需要预测连续的空白序列，模型被迫学习更深层次的语言结构和语义信息，从而提升了泛化能力。

🍋 GLM的自回归空白填充方法通过结合自编码和自回归模型的优势，提供了一种统一的预训练框架。**相比于BERT的MLM方法，GLM在避免信息泄露的同时，更好地建模了上下文依赖关系，提升了模型在自然语言理解任务中的性能。**

15. 了解迁移学习吗？大模型中是怎么运用迁移学习的？

1. 迁移学习的概念：

- **定义**：**迁移学习（Transfer Learning）是指将一个领域（源领域）中学到的知识迁移到另一个领域（目标领域）中应用的技术。**通过这种方式，模型在目标任务中可以利用先前训练的模型参数，从而减少对大规模数据的需求和训练时间。

小红书账号：小黄搞AI

- **应用场景**：迁移学习通常用于数据量不足或训练成本较高的任务中，通过使用已经在相关任务上训练过的模型进行微调或调整，使其能够适应新的任务。

2. 迁移学习在大模型中的应用：

预训练和微调：在大模型（例如 GPT、BERT）中，迁移学习的核心思想体现在预训练-微调（Pre-training & Fine-tuning）的范式中。首先在大规模的通用数据（如互联网文本）上进行预训练，获得能够理解广泛语义的通用模型；**接着，在目标任务的小规模数据集上进行微调，使模型能够适应特定任务。**

预训练是一种从头开始训练模型的方式：所有的模型权重都被随机初始化，然后在没有任何先验知识的情况下开始训练，这个过程不仅需要海量的训练数据，而且时间和经济成本都非常高。因此，大部分情况下，我们都不会从头训练模型，而是将别人预训练好的模型权重通过迁移学习应用到自己的模型中，即使用自己的任务语料对模型进行“二次训练”，通过微调参数使模型适用于新任务。

> 🦄 这个微调的过程只需要很少的数据：我们相当于将预训练模型已经获得的知识“迁移”到了新的领域，因此被称为**迁移学习**。

3. 迁移学习的优势：

- **节省计算资源**：避免从零开始训练模型，节省大量计算资源和时间。
- **减少数据需求**：即使目标任务的数据量不足，预训练的知识仍然可以帮助模型在目标任务上取得良好的效果。
- **适应新任务**：迁移学习允许大模型快速适应不同的任务和领域，具有很好的灵活性。

16. Encoder-only、Decoder-only和Encoder-Decoder的模型分别有什么区别，怎么运用？

1. Encoder&Decoder 模型是什么：

标准的 Transformer 模型主要由两个模块构成：

- **Encoder（左边）**：负责理解输入文本，为每个输入构造对应的语义表示（语义特征）；
- **Decoder（右边）**：负责生成输出，使用 Encoder 输出的语义表示结合其他输入来生成目标序列。

2. Encoder-only 模型：

- **架构特点**：Encoder-only 模型只使用编码器部分。编码器的核心是处理输入序列，并生成该序列的上下文向量（即隐藏状态），它能够很好地捕捉输入序列的全局信息。
- **典型模型**：**BERT（Bidirectional Encoder Representations from Transformers）是一个典型的 Encoder-only 模型。**

小红书账号：小黄搞AI

- **应用场景**：这种架构适合于理解类任务，例如：
  - **文本分类**：根据输入的文本进行分类（如情感分析、新闻分类）。
  - **问答任务**（抽取式）：通过输入问题和上下文，模型从上下文中抽取出准确的答案。
  - **命名实体识别**（NER）：标注文本中的特定实体（如人名、地名等）。

3. Decoder-only 模型：

- **架构特点**：Decoder-only 模型只使用解码器部分。解码器主要用于生成序列，它依赖于自回归机制，每次生成一个单词，并根据之前生成的单词继续预测下一个单词。
- **典型模型**：**GPT（Generative Pretrained Transformer）是 Decoder-only 架构的代表。**
- **应用场景**：这种架构适合生成类任务，例如：
  - **文本生成**：根据提示生成一段连贯的文本（如对话生成、自动写作）。
  - **对话系统**：逐步生成自然语言响应，回答用户提问。
  - **机器翻译**（部分场景下）：逐词生成目标语言的翻译结果。

4. Encoder-Decoder 模型：

- **架构特点**：Encoder-Decoder 模型包含编码器和解码器两个部分。编码器处理输入序列，生成上下文向量；解码器则根据编码器的输出生成目标序列。这个结构能够同时处理输入和输出序列的关联。
- **典型模型**：**T5（Text-to-Text Transfer Transformer）和 Transformer（Vaswani et al. 提出的基础模型）是典型的 Encoder-Decoder 模型。**
- **应用场景**：这种架构适合需要输入和输出匹配的任务，例如：
  - **机器翻译**：将一个句子从源语言翻译成目标语言，编码器负责理解源语言句子，解码器负责生成目标语言句子。
  - **文本摘要**：编码器处理长文本，解码器生成简短摘要。
  - **文本生成**（条件生成）：如文本填空或根据输入生成对应的完整文本。

5. 主要区别总结：

- `Encoder-only`：用于理解任务，适合文本分类、问答、信息抽取等场景。
- `Decoder-only`：用于生成任务，擅长文本生成、对话系统、续写等。
- `Encoder-Decoder`：用于处理输入-输出映射的任务，特别适合机器翻译、文本摘要等。

6. 如何运用：

📌 **任务选择：**根据具体任务选择合适的架构：

小红书账号：小黄搞AI

- 文本分类等理解任务优先使用 Encoder-only 模型如 BERT。
- 文本生成等生成任务使用 Decoder-only 模型如 GPT。
- 输入输出序列间有强关联的任务如翻译和摘要，采用 Encoder-Decoder 模型如 T5。

17. 为什么现在的大语言模型都用Decoder-only

🍊 在近年来的自然语言处理（NLP）领域，大规模预训练语言模型取得了革命性的进展。这些模型在各种任务中表现出色，如语言理解、文本生成和对话系统等。值得注意的是，许多最先进的大语言模型（如 GPT 系列、LLaMA、ChatGPT 等）都采用了 **Decoder-only（仅解码器）**架构。那么，为什么现在的大语言模型都倾向于使用 Decoder-only 架构呢？

17.1 Decoder-only 模型的工作流程

```
[开始标记] --> [解码器]
                    |
                [输出词1] --> [解码器]
                                  |
                              [输出词2] --> [解码器]
                                                |
                                            [输出词3] --> ...
```

说明：模型在每一步生成一个词，作为下一步的输入，直到生成结束标记。

17.2 基础概念

Transformer 架构

Transformer 是由 Vaswani 等人在 2017 年提出的一种神经网络架构，彻底改变了 NLP 领域。它主要由两部分组成：

- **编码器（Encoder）**：处理输入序列，捕获其特征表示。
- **解码器（Decoder）**：根据编码器的输出和先前的输出生成目标序列。

示意图

```
[输入序列] --> [编码器] --> [编码表示]
                                |
                            [解码器] --> [输出序列]
```

小红书账号：小黄搞AI

Encoder、Decoder 和 Encoder-Decoder 模型

- **Encoder-only 模型**：仅使用编码器部分，例如 BERT。**主要用于理解任务。**
- **Decoder-only 模型**：仅使用解码器部分，例如 GPT 系列。**主要用于生成任务。**
- **Encoder-Decoder 模型**：同时使用编码器和解码器，例如原始的 Transformer 和 T5。**用于序列到序列任务。**

17.3 不同架构的特点

Encoder-only 模型

- **特点**：**利用双向上下文信息，对输入进行深度理解。**
- **应用**：自然语言理解（NLU）任务，如文本分类、命名实体识别。
- **局限性**：无法直接用于生成任务。

Decoder-only 模型

- **特点**：**以自回归方式，基于先前生成的词预测下一个词。**
- **应用**：自然语言生成（NLG）任务，如文本生成、对话系统。
- **优势**：能够同时处理理解和生成任务。

Encoder-Decoder 模型

- **特点**：**编码器将输入序列编码，解码器生成目标序列。**
- **应用**：序列到序列任务，如机器翻译、摘要生成。

17.4 为什么选择 Decoder-only 架构

1. 自回归语言模型的优势

自回归模型以概率链式法则为基础，模型生成下一个词的概率仅依赖于之前的词。

公式表示

$$P(w_1, w_2, ..., w_n) = \prod_{i=1}^{n} P(w_i | w_1, ..., w_{i-1})$$

优势

- **灵活性高**：适用于任意长度的文本生成。
- **统一框架**：同一模型可以处理多种任务，无需架构修改。
- **上下文建模**：能够捕获长距离依赖，提高生成质量。

自回归生成过程示意图：

```
P(w_1) --> P(w_2 | w_1) --> P(w_3 | w_1, w_2) --> ... --> P(w_n | w_1, ..., w_{n-1})
```

说明：每个词的生成都基于之前生成的词，形成链式依赖。

2. 训练效率和可扩展性

小红书账号：小黄搞AI

- ***并行计算***：自注意力机制允许在训练中高效并行计算。
- ***参数共享***：减少模型参数，提高训练和推理效率。
- ***可扩展性***：容易扩展到超大规模模型（如数十亿参数）。

3.适用于多种任务

- ***生成任务***：如文本续写、对话生成，Decoder-only 模型表现出色。
- ***理解任务***：通过适当的训练，Decoder-only 模型也能处理理解任务。
- ***统一处理***：简化了不同任务之间的模型设计。

4.简化架构，降低复杂度

- ***模型简单***：仅包含解码器，减少了训练和部署的复杂性。
- ***资源效率***：降低了对计算资源的需求，节省成本。

17.5 技术细节和实践考虑

预训练目标的统一性

- **语言模型预训练**：Decoder-only 模型使用语言模型作为预训练任务，目标明确且通用。
- **避免信息泄露**：自回归方式确保模型仅依赖已生成的内容，避免了双向模型中的信息泄露问题。

训练数据的丰富性

- **大规模语料**：互联网文本为语言模型提供了丰富的训练数据。
- **多样性**：模型可以从大量多样化的数据中学习，提升泛化能力。

生成任务的需求增长

- **实际应用**：如聊天机器人、内容创作，对高质量文本生成的需求增加。
- **Decoder-only 优势**：在生成任务中，Decoder-only 模型具有天然的优势。

模型优化和部署

- **优化技术**：针对 Decoder-only 模型的优化方法更为成熟，如混合精度训练、模型压缩。
- **部署便利**：架构简单，易于在实际系统中集成和部署。

🍊 总结

灵活性和统一性：Decoder-only 架构能够统一处理理解和生成任务，满足多样化的需求。

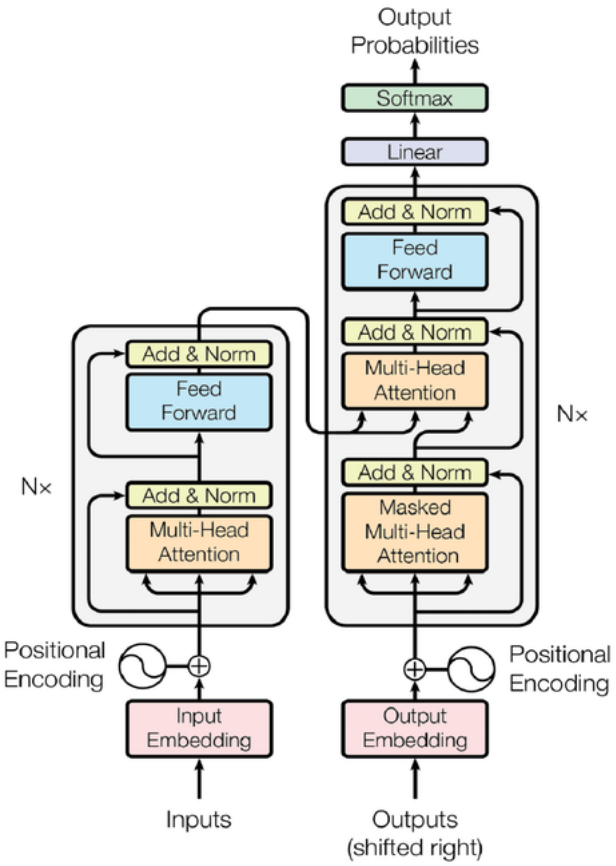
训练和部署效率：架构简单，易于训练和优化，适合大规模模型的开发。

实践验证：大量成功的模型（如 GPT 系列）证明了 Decoder-only 架构的有效性。

18. 简述一下Transformer的基本结构和原理

1. 简要定义与背景

小红书账号：小黄搞AI



Transformer 是一种完全基于注意力机制（Self-Attention）的神经网络模型，由 Vaswani 等人在**2017 年提出，标志着自然语言处理领域的一个重要突破**。Transformer 解决了 RNN（循环神经网络）和 CNN（卷积神经网络）在处理长序列时的效率和性能问题，特别是在**捕捉全局依赖关系、并行计算以及克服梯度消失等方面表现出色**。

与 RNN 不同，Transformer 摒弃了序列顺序的递归处理方式，避免了序列长度增长带来的信息传递瓶颈。在 **RNN 中，信息依赖于时间步长逐步传递，导致序列过长时早期信息容易衰减，且无法并行处理**，训练时间较长。**CNN 虽然可以通过卷积核来捕捉局部特征，但它的感受野有限**，很难处理序列中的远程依赖。

相比之下，**Transformer 通过全局的注意力机制，允许每个位置的词语直接关注序列中的其他所有位置，实现了更高效的长程依赖捕捉**。这样不仅提升了模型处理长序列的能力，还有效减少了序列信息丢失问题。

另外，Transformer 的架构支持大规模并行计算，因为它不需要像 RNN 那样逐步处理序列数据，而是可以同时处理整个输入序列。这大大提高了训练效率，尤其在处理海量数据和复杂任务时表现优异。

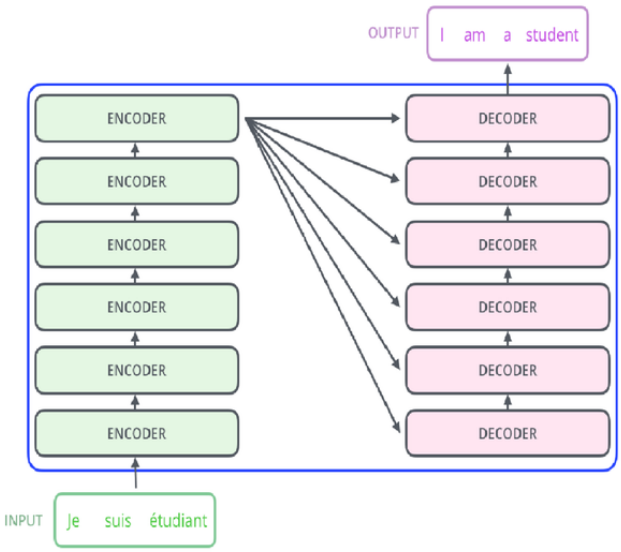
📌 Transformer 模型被广泛应用于各种自然语言处理任务，如机器翻译、文本生成、语义理解、对话系统等领域，**并为之后的大规模预训练模型（如 BERT、GPT 系列）奠定了基础**。

2. 核心架构

Transformer 的架构基于经典的 Encoder-Decoder 结构，其中编码器（Encoder）和解码器（Decoder）各自承担了不同的任务，分工明确。

1. **编码器（Encoder）——理解输入序列：**

小红书账号：小黄搞AI



- **作用**：编码器的任务是对输入序列（例如一段文本或一段音频）进行编码，**提取其中的特征和上下文信息，并将其转化为更高维的隐状态表示（hidden states），即语义向量**。编码器通过自注意力机制（Self-Attention）和前馈神经网络（Feed-Forward Network）捕捉输入序列中各个位置之间的依赖关系，生成能够反映序列全局上下文的表示。

2. **解码器（Decoder）——生成目标序列**：
   - **作用**：解码器的任务是根据编码器生成的隐状态表示，以及已经生成的目标序列部分（在训练时目标序列是已知的，在推理时是逐步生成的），生成最终的目标输出序列（例如翻译后的文本）。**解码器不仅要关注编码器输出的全局语义信息，还需要通过自注意力机制在目标序列中引入依赖**，使其在生成新词时考虑到已经生成的词。

3. 结构详细介绍

📌 编码器负责对输入文本进行编码，生成上下文向量。

解码器根据编码器的输出，生成目标序列中的每个词。

Encoder（编码器）

组成部分：每个编码器层由两个主要子层组成，分别为**多头自注意力机制(Multi-Head Self-Attention Mechanism)和位置前馈神经网络(Position-Wise Feed-Forward Neural Network)**：

a. 多头自注意力机制(Multi-Head Self-Attention)

它通过**并行地计算多个自注意力头**，使模型能够从不同的角度对输入序列进行信息聚合。对于每个输入词的表示，模型通过对比该词与其他所有词的相关性来捕获其上下文关系。这个过程通过查询(Query)、键(Key)和值(Value)的点积来实现。计算过程为:

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

其中，d_k 是查询和键向量的维度，用于归一化点积结果以防止梯度过大或过小。

b. 前馈神经网络(Position-Wise Feed-Forward Network)

这个部分独立地应用于每个位置的词向量表示，包含两个全连接层，中间加入ReLU激活函数:

小红书账号：小黄搞AI

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

通过对每个位置独立应用相同的前馈网络，编码器可以在保留序列结构的同时，**对每个词进行非线性变换**。

c. **残差连接与层归一化(Residual Connections & Layer Normalization)**

每个子层都会加入残差连接，并且在通过子层前进行层归一化，以解决梯度消失和加速模型收敛。归一化公式为:

$$LayerNorm(x + Sublayer(x))$$

残差连接帮助维持信息流动，避免深层网络中信息的丢失。

Decoder(解码器)

组成部分：解码器除了具有与编码器相似的多头自注意力机制和前馈神经网络外，还多了一个编码器-解码器注意力层(Encoder-Decoder Attention Layer)：

a. **多头自注意力机制(Masked Multi-Head Self-Attention)**

- 为了确保模型**在预测下一个词时，只能利用之前生成的词，加入了掩码机制(Masking)**。掩码操作将未来词的注意力权重置为负无穷，防止模型获取不应访问的信息。

b. **多头自注意力机制(Masked Multi-Head Self-Attention)**

- 编码器-解码器注意力层(Encoder-Decoder Attention Layer): 该层作用是通过编码器输出的上下文向量，关注输入序列中的相关部分，来帮助解码器生成与输入相对应的目标序列。

c. **前馈神经网络(Feed-Forward Network)**

- 与编码器中的结构相同，独立地应用于解码器中每个词的表示上。

4. 多头注意力机制的核心原理(Multi-Head Attention Mechanism):

查询(Query)、键(Key)和值(Value)是从输入向量中线性投影出来的不同表示。每个头会通过不同的投影矩阵 W_Q, W_K, W_V 生成多组 Q、K、V，并通过点积注意力来计算它们之间的相关性。通过多头的并行计算，模型能够从多个角度提取序列间的复杂依赖关系，进而提高对不同上下文的捕捉能力。最后，各个头的输出会通过拼接并再次线性变换得到最终的注意力输出:

$$head_i = Attention(QW_{Q_i}, KW_{K_i}, VW_{V_i})$$

$$MultiHead(Q, K, V) = Concat(head_1, \ldots, head_h)W_O$$

5. 位置编码(Positional Encoding)

引入顺序信息

由于Transformer不具有处理序列顺序的内在机制，因此通过为每个位置添加位置编码来为模型提供顺序信息。位置编码采用正弦和余弦函数生成，公式为：

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

小红书账号：小黄搞AI

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

这种函数设计使得模型能够捕捉到不同词之间的相对位置信息，并且可以扩展到任意长度的序列。

6. 模型训练

训练过程中，使用**交叉熵损失函数（Cross-Entropy Loss）**来优化模型权重，并通过并行化计算加速训练。

Transformer 能够非常高效地处理长文本，并避免了 RNN 中存在的梯度消失问题。

7. 优点

并行计算

Transformer 允许对整个输入序列进行并行处理，极大提升了训练速度。

长距离依赖建模

通过自注意力机制，Transformer 能够捕捉远距离词之间的依赖关系。

8. 总结

Transformer 通过引入注意力机制和并行计算，解决了传统序列模型的效率问题，已经成为 NLP 领域的主流架构。

19. Transformer为什么使用多头注意力机制?

📌 Transformer 使用多头注意力机制是为了提升模型捕捉复杂依赖关系的能力，增强表达能力，避免信息瓶颈，并且通过并行计算提高模型的计算效率和注意力分布的多样性。

1. 多头注意力机制的定义：

在 Transformer 中，多头注意力机制（Multi-Head Attention） 的核心思想是**将输入通过多个独立的注意力机制（即“头”）进行并行计算，然后将各个头的输出拼接（concatenate）起来**，最终通过线性变换生成最后的注意力输出。为了理解拼接过程，先简要说明多头注意力机制的流程，再举个例子。

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

通过多个头，模型可以在不同的投影空间中计算注意力，**最终将各个头的输出拼接，并通过线性层进行整合**。具体过程为：

$$MultiHead(Q,K,V) = Concat(head1,\ldots,headh)W_O$$

每个 $head_i$ 代表一个注意力头，它们的并行计算能够为模型提供丰富的上下文信息。

小红书账号：小黄搞AI

2. 多头注意力的目的与优势：

捕捉不同的特征信息（Representation Diversity）：单个注意力头只能聚焦于输入序列中的某些特定关系，而通过多个头，**模型能够并行地从不同的“视角”进行注意力计算**。这些视角可能代表不同的上下文关系、语义层次或距离信息。换句话说，多头机制允许模型在同一层次上学到多种不同的依赖关系，提高了模型对复杂依赖的表达能力。

增强模型的学习能力（Better Feature Learning）：每个注意力头会使用不同的投影矩阵 W_Q、W_K、W_V，这意味着每个头会生成不同的查询、键和值向量。这种多样化的投影可以帮助**模型在更高维度上捕捉更多的信息特征，进而提升模型的表达能力。**

处理不同的关系层次（Hierarchical Relationships）：由于语言中的语义和句法结构具有多层次的特性，**不同的头可以专注于不同的词汇或句法层次，进而使模型能够同时捕捉短程和长程的依赖关系**。例如，一个头可能更关注局部词汇的联系，而另一个头可能会处理句子中远距离的依赖关系。

3. 并行处理提升效率（Parallelization for Efficiency）：

多头注意力机制允许不同的注意力头进行**并行计算**，这相较于传统的序列模型（如 RNNs）显著提高了训练和推理的效率。在多头机制中，模型能够在同一时间捕捉到多个依赖关系，而不是逐步处理，从而大大加速了计算。

4. 避免信息瓶颈（Mitigating the Bottleneck of Single Attention Head）：

使用单个注意力头可能会导致信息过于集中，模型在学习复杂上下文时难以捕捉到多维度的依赖关系。多头注意力通过引入多个注意力头，**降低了单一头在学习复杂关系时可能遇到的信息瓶颈**。

5. 平滑注意力分布（Smoother Attention Distribution）：

在实际应用中，单个注意力头的权重分布可能过于集中，导致模型对某些特定词语的过度依赖。而多**头注意力通过多个头的组合，能够使注意力分布更加均衡和平滑**，从而提高模型对不同上下文的整体理解能力。

6. 总结：

> 📌 Transformer 使用多头注意力机制是为了提升模型捕捉复杂依赖关系的能力，增强表达能力，避免信息瓶颈，并且通过并行计算提高模型的计算效率和注意力分布的多样性。

20. Transformer为何让Q（查询）和K（键）使用独立的权重矩阵进行计算，为什么需要 Q、K、V（查询、键、值）三个矩阵？

1. 独立权重矩阵的设计目的

不同任务、不同功能

小红书账号：小黄搞AI

Q（查询）和 K（键）在注意力机制中的作用不同。Q 代表当前词向其他词发出的“查询”，目的是判断当前词与其他词的相关性；K 则是其他词对这些查询的“响应”，它用来提供与查询匹配的线索。如果 Q 和 K 共用同一矩阵，查询和响应的表达方式会高度相似，无法有效捕捉不同的关系信息，影响注意力机制的灵活性和表达能力。

独立性提供表达多样性

通过不同的线性投影，Q 和 K 能够在不同的向量空间中表征输入的词。这样，Q 可以以一种特定的方式编码查询，而 K 则以另一种方式编码响应，从而允许模型捕捉更丰富和复杂的上下文信息。如果 Q 和 K 共用相同的权重矩阵，模型将难以灵活调整词与词之间的相关性表达。

2. 为什么需要 Q、K、V（查询、键、值）三个矩阵

Q、K、V 的不同角色

Q（Query，查询）：每个词用 Q 来提出问题，去查询其他词和它的相关性，实际上是在当前上下文中，询问“哪些词与我有关系？”。Q 用于构建注意力权重，通过点积计算每个词和其他词的相关性。

K（Key，键）：K 表示的是每个词的“特征描述”，用于和 Q 进行匹配。K 其实是对输入序列中的每个词进行的特征编码，它帮助模型评估每个词对其他词的“响应”能力，类似于如何解释输入词与其他词之间的关系。

V（Value，值）：V 是实际的信息载体，当 Q 和 K 建立了相关性后，最终取值是从 V 中提取的。注意力机制根据 Q 和 K 的相关性对 V 进行加权，提取出有用的信息。V 代表的是输入序列中的“内容”，即真正传递的信息。

Q、K、V 的协同工作

K 提供匹配标准，V 提供信息内容：Q 通过与 K 的点积计算得到各个词之间的相关性，但仅依赖 K 是无法得到最终的上下文信息的。K 是用来建立相关性权重的，而这些权重最终作用于 V，才从 V 中提取出当前词所需的上下文信息。因此，K 负责“衡量相关性”，而 V 则负责提供实际的上下文内容。

V 的必要性：虽然 K 反映了词之间的关联程度，但 V 是用来提供真正的上下文内容。如果只有 Q 和 K，我们只能知道每个词与其他词的相关性，但无法获得与这些相关性对应的具体内容。V 是在相关性确定后，被权重加权的部分，生成的结果便是模型对输入序列的理解。

3. K 为什么不能完全替代 V

信息提取和相关性是两件事

K 提取的是词语之间的依赖关系，而 V 则提供了实际的内容。如果只使用 K，虽然模型可以知道哪些词和当前词关联，但无法得到对应的上下文信息。只有在对 V 进行加权时，模型才能根据与 K 计算出来的相关性提取合适的上下文信息。

区分关联和内容

K 负责提供与 Q 的匹配，而 V 则是对应的值，它们在计算中承担不同的任务。通过计算 Q 和 K 的点积，模型可以知道哪些词与当前词的关联度较高，但这些关联的具体信息，需要通过 V 来提供。

小红书账号：小黄搞AI

4. 总结

Transformer 设计 Q、K 和 V 是为了分离查询、匹配和内容提取三个步骤。Q 是查询向量，用于提问；K 是键向量，用于匹配和判断相关性；V 是值向量，用于传递实际信息。Q 和 K 需要独立的权重矩阵，是为了让模型有能力从不同角度表达输入序列中的依赖关系，而最终的上下文信息是从 V 中提取的，通过 Q 和 K 的相关性来加权得到。

21. 为什么在attention中要进行scaled（为什么除以 $\sqrt{d_k}$ ）？

📌 在 Transformer 的多头注意力机制中，**缩放点积注意力**（Scaled Dot-Product Attention）是核心计算方法之一。计算过程中，查询向量 Q 和键向量 K 进行点积操作，得到注意力分数，然后通过 Softmax 函数将这些分数转换为概率分布。然而，**直接进行点积运算**时，随着向量维度 d_k 的增大，点积的值可能会变得非常大，进而影响 Softmax 的梯度以及模型的训练稳定性。

1. 问题背景

在注意力机制中，点积计算会通过以下公式得出注意力分数：

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

其中 Q 是查询向量，K 是键向量，V 是值向量，d_k 是键向量的维度。

问题1： 点积值过大

查询向量 Q 和键向量 K 是高维向量，点积的期望值会随着维度 d_k 的增加而增大。具体来说，两个随机独立的向量的点积期望值与向量维度成正比，导致点积结果会非常大。进入 Softmax 函数后，这些大值会使得 Softmax 输出变得极端化：**某些位置的输出非常接近 1，而其他位置接近 0**。此时，**注意力机制的注意力分布过于集中**，模型难以有效地从多个位置学习信息。

问题2： 梯度消失

当点积值过大时，Softmax 函数的梯度可能会变得非常小，导致**模型的反向传播过程中梯度消失**，使得模型训练速度变慢或难以收敛。

2. 缩放的解决方案

为了避免这些问题，Transformer 引入了一个缩放因子，将点积的结果除以 d_k 的平方根，即：

$$\frac{QK^T}{\sqrt{d_k}}$$

这样做的原因主要有以下几点：

正则化效果

小红书账号：小黄搞AI

缩放因子 $\frac{1}{\sqrt{d_k}}$ 的引入可以防止随着 d_k 增加，点积值变得过大。点积的期望值大约与向量维度 d_k 成正比，因此通过除以 $\sqrt{d_k}$ ，可以**将点积值缩放到一个合适的范围，确保数值稳定**。理论上，两个独立随机向量的点积的标准差大约与向量维度 d_k 的平方根成比例，因此通过缩放可以将点积值控制在合理范围内。

保持数值稳定性

缩放因子确保了 Softmax 的输入不会过大或过小，保证了注意力机制的数值稳定性。大幅度缩放的点积会使 Softmax 输出接近 0 或 1，这导致注意力集中在特定位置，而通过缩放，**注意力分布更加平滑**，模型能够更稳定地训练。

避免梯度消失

通过引入缩放，点积值进入 Softmax 后，其值不会过于极端，进而避免了梯度消失问题。梯度消失通常出现在激活函数的输出值接近 0 或 1 时，**通过缩放可以确保 Softmax 的输入位于合理范围内，确保反向传播过程中梯度正常传播。**

3. 数值举例

假设 $d_k = 64$ ，那么查询向量 Q 和键向量 K 的点积结果的期望值可能在 64 量级。如果不进行缩放，Softmax 的输入可能非常大，导致其输出值集中在某些位置（接近 1），其他位置则接近 0。此时模型难以有效学习。通过将点积结果除以 $\sqrt{64} = 8$ ，点积结果被缩放至较小范围，使得 Softmax 的输出更加平滑，注意力分布更加合理。

4. 总结

在 Softmax 之前对点积注意力分数进行缩放（除以 $\sqrt{d_k}$ ）是为了应对高维向量下点积值过大的问题。这一操作可以：

1. **防止数值不稳定**，避免注意力权重极端化，确保模型能够有效学习；
2. **避免梯度消失**，通过限制输入值的范围，确保 Softmax 梯度不会过小，保障模型训练效率；
3. **加快模型收敛**，缩放使得多头注意力机制在高维度下保持数值稳定性，从而提高模型的训练速度和表现。

这个缩放机制在实际应用中证明是有效的，能够显著提升模型的稳定性和性能。

22. Transformer的位置编码作用及局限性?

1. Transformer的位置编码介绍

在Transformer模型中，位置编码（Position Encoding）用于引入序列中各个词的位置信息。由于Transformer模型的自注意力机制（Self-Attention）不具备处理序列顺序的能力，因此需要位置编码来补充这一信息，也是苏剑林大神在博客中写的**打破Attention的置换不变性**，位置编码的主要作用是为模型提供词在序列中的相对位置或绝对位置，是加入对Attention的先验认知，或者赋予Attention学习到这些先验认知性质的能力。从而帮助模型理解词语的顺序和语法结构。

小红书账号：小黄搞AI

Transformer中常用的位置编码方法是正弦-余弦位置编码（Sinusoidal Position Encoding）。具体而言，位置编码是一个与输入词嵌入（Word Embedding）相加的向量，其计算方式如下：

$$PE(p, 2i) = \sin\left(\frac{p}{10000^{2i/d}}\right)$$
$$PE(p, 2i+1) = \cos\left(\frac{p}{10000^{2i/d}}\right)$$

2. Transformer的位置编码作用

捕捉序列顺序

位置编码使得Transformer能够区分词语在序列中的位置，从而理解上下文信息。这对于处理自然语言序列尤其重要，因为序列的顺序对语义理解具有关键作用。

保持全局一致性

由于位置编码在不同位置上是具有唯一性的，因此它帮助模型维持序列中的全局一致性，保证对整个序列的理解。

增强模型能力

位置编码使得自注意力机制能够基于位置信息进行加权，有效提升了模型对长距离依赖的处理能力。

3. Transformer的位置编码局限性

固定位置编码的局限性

标准的正弦-余弦位置编码是固定的，不可学习。它无法适应特定任务或数据的变化，因此在某些特定应用中，可能无法提供最优的位置信息。

有限的序列长度

正弦-余弦位置编码对序列长度有一定限制。在处理非常长的序列时，固定位置编码可能无法充分表达序列的全貌。

位置编码的线性组合

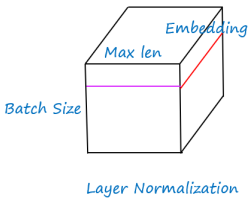
位置编码与词嵌入的线性组合可能导致信息丢失，因为位置编码的信息可能被嵌入中的信息所覆盖，从而影响模型对位置信息的敏感性。

4. 总结

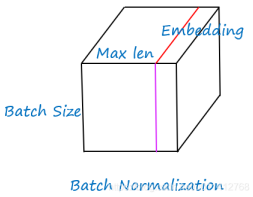
位置编码在Transformer中起到了关键作用，通过引入位置信息，帮助模型理解序列中的词汇顺序。然而，**固定的正弦-余弦位置编码在处理长序列和特定任务时存在一定的局限性**，需要结合具体应用场景来优化或替代。

23. 为什么Transformer用LayerNorm而不用BatchNorm?

小红书账号：小黄搞AI



layer normal



batch normal

首先，我们应该先对我们的数据形状有个直观的认识，当一个batch的数据输入模型的时候。

形状是长方体如图所示，大小为**(batch_size, max_len, embedding)**，其中batch_size为batch的批数，max_len为每一批数据的序列最大长度，embedding则为每一个单词或者字的embedding维度大小。

> Batch Normalization是在batch间选择同一个位置的值做归一化，相当于是对batch里相同位置的字或者单词embedding做归一化。
>
> Layer Normalization是在一个Batch里面的每一行做normalization，相当于是对每句话的embedding做归一化。显然，LN更加符合我们处理文本的直觉。

1. 概念阐述

`BatchNorm（批量归一化）` 主要用于训练深度神经网络时对每一层的激活进行归一化，以减少内部协变量偏移（internal covariate shift）。它通过**对每一批数据进行归一化来稳定训练过程，并通常会加快收敛速度**。

`LayerNorm（层归一化）` 归一化的是每一层的所有特征，不依赖于批量大小，而是对**单个样本的所有特征进行归一化**。这对于处理变长序列数据和动态批量大小尤其重要。

2. Transformer模型的特性

变长序列： Transformer模型处理的是变长的序列数据，**BatchNorm在处理变长序列时可能会遇到问题，因为它需要在每个批次内计算均值和方差**，这对于动态长度的序列不太适用。

自注意力机制： Transformer的自注意力机制允许每个位置对序列中的所有其他位置进行关注。**BatchNorm的统计信息是基于批次的统计，这可能会导致不同位置的信息互相干扰**，从而影响模型的性能。LayerNorm则是对每个样本的所有特征进行归一化，避免了这种干扰。

3. BatchNorm的局限性

批量大小依赖性： BatchNorm的表现依赖于批量的大小。**当批量大小较小或序列长度不一致时，它的统计量可能不稳定**，从而影响模型的训练效果。

推理阶段： 在推理阶段，**BatchNorm需要用到训练阶段的均值和方差进行归一化，这增加了推理的复杂性和开销**。LayerNorm在训练和推理阶段的表现一致，因为它的归一化是基于每个样本的特征，而不是批量统计。

4. LayerNorm的优点

小红书账号：小黄搞AI

独立于批量大小： LayerNorm**不受批量大小**的影响，对于处理变长序列数据非常合适。

更稳定的训练： 由于LayerNorm在每个样本的特征维度上进行归一化，**模型在处理每个样本时的行为是一致的，这使得训练过程更稳定。**

简化推理过程： LayerNorm在训练和推理阶段的行为一致，**不需要额外的训练阶段统计量**，因此推理过程更简单高效。

总结来说，Transformer使用LayerNorm而不是BatchNorm，主要是因为LayerNorm更适合处理变长序列数据，并且在训练和推理阶段的一致性上具有优势。这使得LayerNorm在Transformer架构中能更好地支持自注意力机制的动态特性。

24. 解释一下KVcache的原理

KV Cache（键值对缓存） 是一种优化大规模模型推理的技术，尤其在处理像 Transformer 这样的模型时非常重要。它通过缓存已经计算的键（Key）和值（Value），显著提升推理效率和减少计算复杂度。下面是关于 KV Cache 的详细说明：

1. KV Cache的基本概念

- 键值对缓存: KV Cache 是一种存储结构，用于缓存每个输入 token 的键和值。每个键（Key）映射到一个值（Value），以便快速访问。缓存的目的是减少对原始数据源的重复访问，提高系统性能。

2. Transformer模型中的KV Cache

在 Transformer 模型中，推理过程生成每个 token 时需要计算该 token 的键和值。生成新 token 时，这个过程会重复进行，导致计算复杂度的增加。KV Cache 解决了这个问题：

- 冗余计算: 每次生成新 token 时，未使用 KV Cache 的情况下，模型需要重新计算所有 token 的键和值，这导致计算复杂度呈平方增长。
- 缓存的作用: KV Cache 通过缓存之前计算的键和值，避免了这些重复计算。新的 token 只需与缓存中的数据进行交互，从而将复杂度降低到线性。

3. 推理阶段

KV Cache 技术将推理过程分为两个主要阶段：

- 预填充阶段（Prefill）：
  - 特点: 计算密集型（compute-bound）。
  - 过程: 在生成第一个 token 时，系统计算并存储所有 Transformer 层的键和值。此时的计算开销与未使用 KV Cache 时相当，主要涉及大量的矩阵运算（GEMM），需要强大的计算能力。
- 解码阶段（Decode）：
  - 特点: 内存密集型（memory-bound）。

小红书账号：小黄搞AI

- 过程: 从生成第二个 token 开始到生成最后一个 token，系统利用缓存的键和值，并仅需计算新 token 的键和值。由于 KV Cache 已经存储了之前的结果，每轮推理速度显著提升，但需要更大的内存来存储这些数据。

4. KV Cache的挑战和优化

- 内存占用: 随着序列长度的增加，KV Cache 的内存占用线性增长，可能达到模型大小的数倍。在显存受限的情况下，这种内存需求可能成为限制模型性能的瓶颈。
- 优化策略:
  - 量化技术: 减少模型权重和 KV Cache 的内存占用。
  - 模型并行: 将模型分片到多个 GPU 上。
  - 压缩和内存管理: 采用压缩技术（如 FastGen 方法），通过自适应压缩减少 KV Cache 的内存需求，同时保持模型效率。

5. 总结

KV Cache 是一种重要的推理优化技术，通过缓存计算过的键和值，减少冗余计算，提高了 Transformer 模型的推理效率。尽管它带来了内存占用的问题，但通过各种优化策略，这些问题可以得到有效缓解，从而提升了大规模模型的性能。

25. Transformer相比RNN、LSTM的优势何在?

1. 并行计算能力

- RNN/LSTM：RNN和LSTM是顺序模型，依赖于序列的前后顺序处理数据，当前时刻的输出依赖于前一时刻的输入。这种顺序性限制了模型的并行计算能力，导致训练速度较慢，尤其当序列长度增加时，效率下降明显。
- Transformer：Transformer通过自注意力机制（Self-Attention）消除了序列顺序的依赖，允许模型同时处理输入序列的不同部分。这样大大提升了并行处理能力，训练速度也比RNN/LSTM快得多，尤其是在处理长序列时。

2. 长距离依赖问题

- RNN/LSTM：虽然LSTM通过门控机制（如遗忘门和输入门）在一定程度上解决了长距离依赖问题，但依然可能由于梯度消失或梯度爆炸而导致模型难以捕捉长距离依赖。
- Transformer：Transformer的自注意力机制能够直接建立序列中任何位置的全局依赖，无论距离多远。因此在处理长距离依赖时，Transformer更为有效。

3. 建模灵活性

小红书账号：小黄搞AI

- RNN/LSTM：这些模型由于顺序处理限制，只能按时间步长逐步建模，难以在序列的多个位置之间灵活捕捉关系。
- Transformer：自注意力机制可以对整个输入序列中的每个位置进行建模，捕捉序列中任何两个位置的关系，无论它们之间的距离有多远。这样不仅提高了模型的表达能力，还使得它在处理复杂依赖关系时更加灵活。

4. 应用领域的成功

- Transformer自2017年提出以来，已经在多个领域取得了显著的成功，尤其在自然语言处理（NLP）和计算机视觉领域。模型如BERT、GPT等都是基于Transformer架构，在各类任务中表现远超传统的RNN/LSTM模型。

总结来说，Transformer相较于RNN和LSTM在并行计算能力、长距离依赖处理、建模灵活性等方面都有显著优势，因此在大规模数据和复杂任务中表现得更加优越。

26. Transformer在哪里做了权重共享？

Embedding层可以说是通过onehot去取到对应的embedding向量，FC层可以说是相反的，通过向量（定义为 x）去得到它可能是某个词的softmax概率，取概率最大（贪婪情况下）的作为预测值。

1. 权重共享的概念

权重共享是指在模型的不同层之间复用相同的参数。这可以减少模型的总体参数数量，并使得模型在训练时更容易学习。

2. 在Transformer中的应用

在Transformer中，权重共享主要体现在以下几个方面：

嵌入层和输出层：常见的做法是共享词嵌入层（embedding layer）和输出层（output layer）之间的权重。这意味着在生成词汇的概率分布时，使用的嵌入矩阵和线性层是相同的。这样可以有效地利用已经学习到的词向量，并且减少了参数量。具体来说，如果嵌入矩阵 (W_e) 用于将输入标记转换为嵌入表示，则同样的矩阵也可以用于输出层：

$$\text{Logits} = \text{Output}_d^{(l)} W_e^T$$

位置编码的权重共享：在Transformer中，位置编码通常是通过学习的固定向量来实现的，这些向量与输入的词嵌入相加以表示单词在序列中的位置信息。在编码器和解码器中，可以共享相同的位置编码矩阵，这样不同位置的输入在不同层之间可以有相同的表示。

编码器和解码器的词嵌入权重共享：在机器翻译任务中，源语言和目标语言虽然不同，但它们可以共用一个大型的词表，尤其是考虑到一些词汇如数字、标点符号等在多种语言中是通用的。这使得源语言和目标语言的Embedding层可以共享权重。当使用BPE（Byte Pair Encoding）这样的子词（subword）编码技术时，最小的单元不再是完整的单词，而是更小的子词片段。英语和德语等同属日耳曼语族的语言有许多相同的子词，可以共享相似的语义表示。而对于汉语和英语这种差异较大的

小红书账号：小黄搞AI

语言对，共享的语义可能较少，共享权重的意义不是很大。然而，共享词表会导致词表规模显著增加，这可能增加softmax层的计算负担，影响模型的训练速度和推理效率。因此，在实际应用中，是否进行权重共享需要权衡模型的性能提升与计算资源的消耗。

解码器自注意力中的权重共享：在解码器的自注意力层中，可能会采用权重共享策略，即使用相同的查询、键和值的权重矩阵。这种共享可以提高模型的效率，减少参数数量，并且可以使模型更容易训练。

其他层的部分权重共享： 在一些变体的Transformer模型中，还可能会实现其他层的权重共享，比如部分的前馈神经网络层或者特定的注意力头部。

27. 介绍一下VLLM的加速方法

VLLM (Versatile Large Language Model)是一个专门为大规模语言模型 (LLM) 推理设计的开源加速框架，通过创新的内存管理和并行化技术，显著提高了推理速度和吞吐量。

其中，PagedAttention是 VLLM 的核心技术，专门用于解决 LLM 推理中的内存瓶颈问题，尤其是自回归生成任务中的键值 (KV) 缓存管理。

1. PagedAttention 机制

PagedAttention 的主要目标是高效管理注意力机制中的 **KV Cache**，从而**减少内存碎片和重复存储**带来的显存浪费。在传统的自回归解码过程中，模型需要将所有输入 token 的键和值张量（KV 缓存）存储在 GPU 内存中，以生成下一个 token。这种方法占用了大量显存，尤其当处理长序列时，GPU 内存成为推理过程中的瓶颈。

PagedAttention 通过引入类似于 **虚拟内存管理** 的思想，将 KV 缓存进行**物理分块**管理。即使模型在逻辑上需要连续的输入序列，实际的物理内存块可以是非连续的。这样做的好处是极大减少了 KV 缓存中的内存浪费，并提高了推理过程中的内存利用率。

`PagedAttention 的具体工作原理`

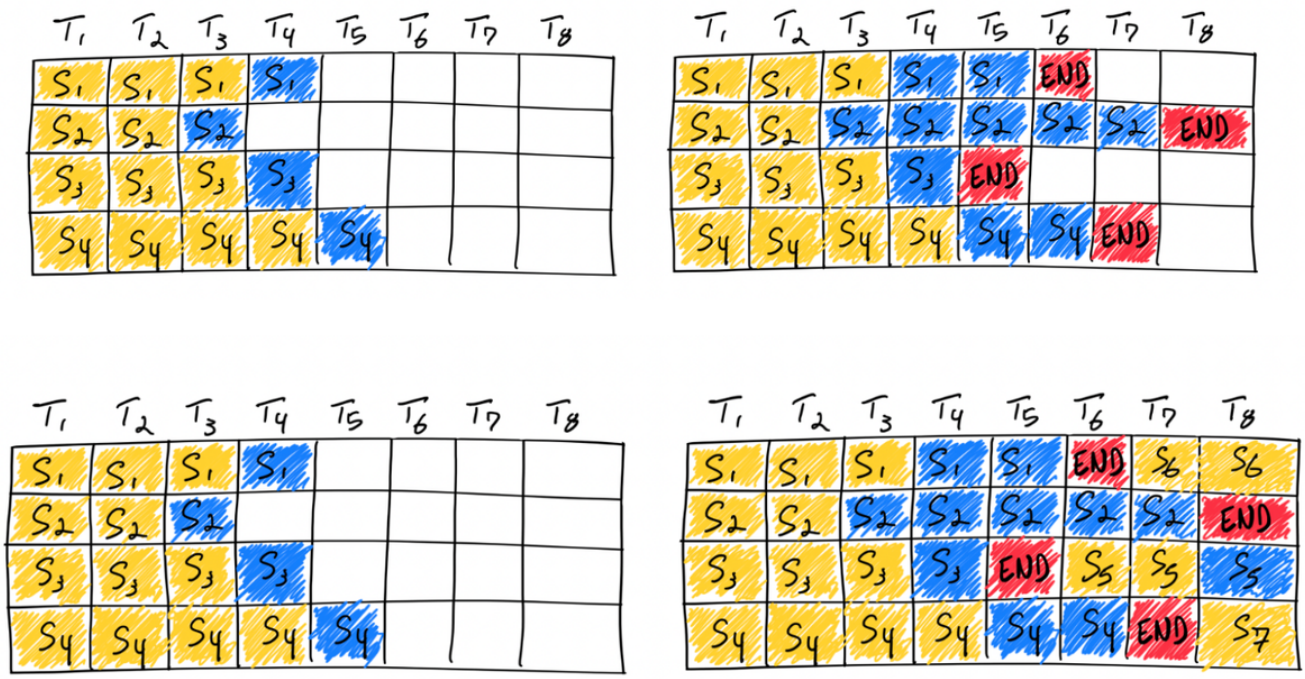
物理分块：PagedAttention 将 KV 缓存进行物理分块，每块显存包含固定长度的 tokens。在进行注意力计算时，VLLM 会根据需要从这些分块中读取键值缓存。**尽管模型在逻辑上处理的是一个连续的序列，但这些序列在内存中的实际位置是分散的。**

减少重复缓存：在处理多个输出时，VLLM 可以将不同的逻辑块映射到同一个物理块，避免了重复存储同样的数据。这种方法显著减少了显存的占用，同时提高了吞吐量。

通过这种分块管理方式，VLLM 可以在保持推理速度的同时，极大降低内存占用，尤其是应对长序列推理时的内存瓶颈。

2. Continuous Batching机制

小红书账号：小黄搞AI



另一个提升 VLLM 推理速度的重要机制是 **Continuous Batching**，它优化了批处理过程。**传统的批处理方法（static batching）要求所有输入序列的长度对齐，这意味着较短的句子需要等待较长句子生成完毕，导致 GPU 计算资源被浪费。**

VLLM 采用 Continuous Batching，即每当某个句子的推理完成时，GPU 会立即填充下一个句子的 token，而**不需要等待整个批次的推理完成**。这种动态的批次管理方式充分利用了 GPU 的计算能力，减少了等待时间，极大提高了吞吐量。

3. PagedAttention 与 Continuous Batching 的协同作用

PagedAttention 和 Continuous Batching 共同作用，优化了 VLLM 的推理性能。PagedAttention 通过精细管理 KV 缓存，减少了推理过程中的内存开销，特别是在处理长序列或多个输出时的内存浪费。而 Continuous Batching 则确保 GPU 计算资源得到最大化利用，使得在推理过程中几乎没有等待时间。

4. VLLM 其他特性

支持张量并行推理

VLLM 支持通过张量并行来进一步加速推理，将模型的计算分布到多个 GPU 上，降低单个 GPU 的内存和计算压力。

VLLM 通过实现高效的张量并行（Tensor Parallelism）来加速推理，将模型分割并分布到多个 GPU 上，使每个 GPU 仅处理部分计算和内存负载。张量并行将大的权重矩阵（例如 Transformer 层中的矩阵）划分为更小的片段，并分布到不同的 GPU 上。

对于矩阵乘法 $W \times X$，其中 $W \in \mathbb{R}^{d_1 \times d_2}$，$X \in \mathbb{R}^{d_2 \times n}$，在使用 p 个 GPU 切分后，每个 GPU 计算矩阵的一部分，具体为 $W_i \times X$，其中 $W_i \in \mathbb{R}^{(d_1/p) \times d_2}$

小红书账号：小黄搞AI

推理的总时间 T_{total} 可以通过将计算分布到多个 GPU 上而减少，理论上的加速比为：

$$T_{total} \approx \frac{T_{single}}{p}$$

其中 T_{single} 为单个 GPU 执行该操作所需的时间。

流式输出：

VLLM 支持流式输出，可以在生成过程中逐步返回生成的 token，减少响应时间，适合需要低延迟的场景。

VLLM 使用**基于流式预取（streaming-based prefetching）**的技术来将通信和计算重叠，尽量减少 GPU 的空闲时间。在模型的一部分正在执行时，下一部分的数据会提前加载并准备好，从而确保 GPU 不会因等待数据而闲置。

与 HuggingFace 模型无缝集成

VLLM 可以与 HuggingFace 的模型兼容使用，方便用户直接在 HuggingFace 提供的模型基础上进行推理加速。

5. 总结（性能提升）

通过这些优化，VLLM 实现了比 HuggingFace Transformers 高 **14-24 倍**的吞吐量，特别在长序列推理和多请求并发的场景中表现尤为突出。

28. 介绍一下FlashAttention的原理

🍊 在深度学习领域，Transformer模型凭借其强大的性能，成为自然语言处理和计算机视觉等领域的主流架构。Transformer 的核心组件是自注意力机制（Self-Attention），然而，传统的自注意力计算在处理长序列时，面临着计算量大和内存占用高的问题。FlashAttention是一种高效的自注意力计算算法，旨在解决这些问题。

28.1 基础概念

Transformer 和自注意力机制

Transformer 模型的核心是自注意力机制，它可以捕获序列中不同位置之间的全局依赖关系。

- **查询（Query）**：表示当前要关注的信息。
- **键（Key）**：表示序列中可被关注的信息。
- **值（Value）**：表示对应键的信息内容。

🦄 **自注意力计算公式**：$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$

其中：

小红书账号：小黄搞AI

- $Q \in \mathbb{R}^{L \times d_k}$：查询矩阵
- $K \in \mathbb{R}^{L \times d_k}$：键矩阵
- $V \in \mathbb{R}^{L \times d_v}$：值矩阵
- L：序列长度
- d_k、d_v：向量维度

计算和内存瓶颈

在传统实现中，自注意力计算涉及到大型矩阵的乘法和存储：

- **计算复杂度**：$O(L^2 d_k)$
- **内存占用**：需要存储 $L \times L$ 的注意力矩阵，内存占用为 $O(L^2)$

对于长序列，内存占用和计算量都会急剧增加，导致效率低下。

28.2 FlashAttention 的原理

问题的本质

- **内存瓶颈**：GPU 的内存带宽限制了自注意力的计算速度。
- **访存开销**：传统算法需要多次读取和写入 GPU 内存，导致大量的访存操作。

FlashAttention 的核心思想

FlashAttention 通过以下方式优化自注意力计算：

- **块处理（Blocking）**：将计算划分为小块，逐块处理，减少一次性需要的内存。
- **IO 感知（IO-Awareness）**：优化内存访问模式，最大化数据在高速缓存中的使用，减少访存开销。
- **数值稳定性**：在块内进行归一化，避免数值溢出。

具体实现步骤

步骤 1：块划分

- 将序列长度 L 划分为多个块，每个块的大小为 B。
- 例如，序列长度为 1024，块大小为 128，则有 8 个块。

步骤 2：逐块计算

- **计算块间的注意力**：对于每个查询块和键块，计算局部的注意力值。
- **内存占用优化**：只在寄存器或高速缓存中存储当前块的数据，避免将整个 $L \times L$ 的注意力矩阵存入内存。

步骤 3：数值稳定的 Softmax 计算

- **局部归一化**：在块内进行 Softmax 计算，使用数值稳定的算法，避免指数计算中的溢出。
- **累积计算**：逐块累积注意力值和归一化因子，确保最终结果的准确性。

算法流程图

小红书账号：小黄搞AI

```
[输入 Q, K, V]
      |
[序列块划分]
      |
[初始化累积结果]
      |
[对每个查询块]
      |
[对每个键块]
      |
[计算局部 QK^T / sqrt(d_k)]
      |
[局部 Softmax 归一化]
      |
[更新累积注意力和值]
      |
[最终输出 Attention(Q, K, V)]
```

传统自注意力的内存占用

```
    [Q]     [K^T]     计算 QK^T       得到 Attention Matrix
 (Lxd_k) x (d_kxL) => (LxL)                  (LxL)

内存占用：需要存储 (LxL) 的矩阵
```

FlashAttention 的块级处理

```
将 Q 和 K^T 分块：

[Q_block_i] x [K_block_j^T] => 局部 Attention Matrix
   (Bxd_k)      (d_kxB)            (BxB)

内存占用：只需存储 (BxB) 的矩阵，多次重复，最终得到完整的结果
```

局部计算与累积

```
对于每个查询块 i：
    初始化局部最大值和累积和
    对于每个键块 j：
        - 计算局部的 QK^T / sqrt(d_k)
```

小红书账号：小黄搞AI

```
        - 更新局部最大值（用于数值稳定）
        - 计算局部 Softmax
        - 更新累积的注意力和值

最终输出 Attention(Q, K, V)
```

28.3 FlashAttention 的优势

减少内存占用

块级处理：避免存储整个注意力矩阵，内存占用从 $O(L^2)$ 降至 $O(LB)$。

适合长序列：能够高效处理长达几千甚至上万长度的序列。

提高计算效率

减少访存操作：优化内存访问模式，提高了数据在高速缓存中的命中率。

GPU 加速：充分利用 GPU 的计算能力和寄存器优势。

数值稳定性

局部归一化：

在计算过程中保持数值稳定。

避免了传统方法中的数值溢出和下溢问题。

数值稳定的 Softmax 计算

在计算 Softmax 时，为了避免指数函数导致的数值溢出，常采用以下技巧：

$$\text{Softmax}(x_i) = \frac{e^{x_i - \max(x)}}{\sum_j e^{x_j - \max(x)}}$$

- **解释**：从每个输入中减去最大值 $\max(x)$，避免了过大的指数。

28.4 与传统自注意力的对比

内存占用对比

传统自注意力：需要存储 $L \times L$ 的矩阵，内存占用大。

FlashAttention：内存占用与块大小 B 成正比，大幅降低了内存需求。

计算速度对比

减少访存操作传统方法：由于大量的访存操作，计算速度受限于内存带宽。

FlashAttention：减少了访存次数，计算速度主要由计算单元决定，更快。

数值精度对比

传统方法：

可能出现数值不稳定，特别是在长序列或高维度下。

FlashAttention：

通过数值稳定的算法，确保了计算精度。

❤️ FlashAttention 提供了一种高效且数值稳定的自注意力计算方法，通过优化内存访问和计算流程，解决了传统自注意力在长序列处理中的内存和计算瓶颈。

总结其优势：

- **内存高效**：大幅降低内存占用，适合处理长序列。
- **计算高效**：减少访存开销，提高计算速度。

小红书账号：小黄搞AI

- **数值稳定**：避免数值溢出和下溢，保证计算精度。

> 推荐阅读
>
> - **Tri Dao**, et al. (2022). FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. *arXiv preprint arXiv:2205.14135*.
> - Vaswani, A., et al. (2017). Attention is All You Need. *Advances in Neural Information Processing Systems*.

29. 介绍一下Tokenizer怎么训练的

在自然语言处理（NLP）领域，Tokenizer（分词器）是将原始文本转换为模型可处理的基本单位（即词元或Token）的工具。Tokenizer的训练对于模型的性能和效率至关重要。

29.1 为什么需要Tokenizer?

1. **计算机对文本的处理需求**：计算机无法直接理解人类语言，需要将文本转换为数值形式才能处理。
2. **模型输入的标准化**：将不规则的自然语言转换为标准化的Token序列，便于模型学习。
3. **降低数据稀疏性**：通过合理的分词，可以减少词汇量，降低模型复杂度，提高训练效率。

29.2 基本概念

1. **Token（词元）**：文本被分割后的最小单位，可能是字符、词或子词。
2. **词汇表（Vocabulary）**：所有Token的集合，模型只能处理词汇表中的Token。
3. **子词（Subword）**：介于字符和词之间的单位，用于解决新词和低频词的问题。

29.3 传统分词方法的局限

基于词的分词	基于字符的分词
将文本按空格或标点分割，**但无法处理新词、拼写错误或形态变化，导致大量的未登录词（OOV）**。	将每个字符作为一个Token，**词汇表小，但序列长度过长，模型难以捕获长距离依赖。**

29.4 引入子词分词

子词分词在字符级和词级之间找到平衡，通过将词拆分为更小的子词单位，解决新词和低频词的问题。

优点：

- **处理未登录词：** 新词可由已知子词组合而成。
- **减少词汇表大小：** 共享子词，降低模型参数量。

小红书账号：小黄搞AI

- **提高模型泛化能力：** 更好地捕获词的形态和词缀信息。

29.5 常见的子词分词算法——Byte Pair Encoding（BPE）

1. 原理

BPE最初用于数据压缩，其在NLP中的应用是：**从字符开始**，迭代地合并频率最高的相邻符号对，构建子词词汇表。

2. 训练步骤

- 1.**初始化**：将语料库中的单词分解为字符序列，末尾加上特殊符号 `</w>` 表示词的结束。

> 示例：`*apple*` 被分解为 `*a p p l e </w>*` 。

- 2.**统计符号对频率**：统计所有相邻符号对的出现频率。

> 示意图：统计如 `*a p*` 、`*p p*` 、`*p l*` 、`*l e*` 、`*e </w>*` *的频率。*

- 3.**合并频率最高的符号对**：将其作为新的符号添加到词汇表中。

> 示例：如果 `*p p*` 频率最高，合并为 `*pp*` ，更新序列为 `*a pp l e </w>*` 。

重复：继续统计新的符号对频率，合并频率最高的，直到达到预设的词汇表大小。

3. 示例

- **初始序列：**

```
1 l o w </w>
2 l o w e r </w>
3 n e w e s t </w>
4 w i d e s t </w>
```

- **第一次合并**（`e s` 频率最高）：

```
1 l o w </w>
2 l o w e r </w>
3 n e w es t </w>
4 w i d es t </w>
```

- **第二次合并**（`es t` 频率最高）：

```
1 l o w </w>
2 l o w e r </w>
3 n e w est </w>
4 w i d est </w>
```

小红书账号：小黄搞AI

- **持续合并**，构建更大的子词。

4. 优缺点

优点	缺点
简单高效，易于实现，适用于多种语言。	合并基于频率，未考虑词的语义信息。

29.6 常见的子词分词算法——WordPiece算法

1. 原理

WordPiece最初用于谷歌的机器翻译系统，与BPE类似，但在合并时考虑了语言模型的概率。

2. 训练步骤

- 1.**初始化**：同样将词分解为字符序列。
- 2.**迭代合并**：
  - ***计算得分***：对每个可能的合并，计算语言模型概率的增益。
  - ***选择合并***：选择使模型概率增益最大的符号对进行合并。
- 3.**更新词汇表**：将新的合并符号加入词汇表。

3. 示例

- **假设语料库**：

```
un affable
un acceptable
un account able
```

- **合并步骤**：
- **计算** `un` 和 `affable` 的合并得分。
- **选择最佳合并**，如 `un` + `affable` 合并为 `unaffable` 。

4. 优缺点

- **优点**：考虑了语言模型的概率，提高了分词的合理性。
- **缺点**：计算复杂度高，训练时间较长。

29.7 Unigram语言模型

1. 原理

基于Unigram（一元）语言模型，将分词视为对词汇表概率分布的估计。

小红书账号：小黄搞AI

2. 训练步骤

- **初始化词汇表**：包含所有可能的子词。
- **估计子词概率**：基于语料库，计算每个子词的概率。
- **词汇表优化**：
  - **计算损失**：基于当前词汇表的子词概率，计算模型的损失。
  - **移除低概率子词**：删除对损失影响最小的子词，缩小词汇表。
- **重复**：迭代优化词汇表，直到达到预设大小。

3. 优缺点

- **优点**：全局优化，考虑了子词的整体概率分布。
- **缺点**：需要大量计算资源，训练复杂度高。

29.8 实践中的考虑

1. **词汇表大小的选择**：需要在模型性能和计算资源之间权衡。
2. **特殊符号处理**：如数字、标点、表情符号，需要特殊处理。
3. **多语言支持**：对于多语言模型，需要确保Tokenizer对不同语言的适用性。

> 💫 **总结**
>
> Tokenizer的训练是NLP模型构建中关键的一步。通过理解不同分词算法的原理和训练过程，我们可以根据具体需求选择合适的Tokenizer，提高模型的性能和效率。无论是BPE的简单高效，WordPiece的概率优化，还是Unigram模型的全局视角，都各有其适用场景。

30. WordPiece如何考虑语言模型的概率

> 🐠 WordPiece最初用于谷歌的机器翻译系统，与BPE类似，但在合并时考虑了语言模型的概率。即WordPiece算法利用语言模型的概率来指导子词（subword）的合并。

30.1 WordPiece算法中的概率考虑

1. 与BPE的区别

- **BPE（字节对编码）**：在每次迭代中，合并频率最高的相邻符号对（字符或子词），不考虑合并对整体语言模型概率的影响。
- **WordPiece**：在每次迭代中，选择能够最大化训练语料库整体概率的符号对进行合并，即考虑了语言模型的概率。

2. 语言模型概率的概念

小红书账号：小黄搞AI

- **语言模型（Language Model，LM）**：给定一系列子词，计算这些子词序列在语料库中出现的概率。
- **在WordPiece中**：使用一元（unigram）语言模型，假设每个子词是独立的，整个语料库的概率是所有子词概率的乘积。

30.2 WordPiece的训练步骤详解

1. **初始化**

- 将语料库中的词分解为字符序列。

2. **统计子词频率**

- 计算所有可能的子词（初始为字符）的出现频率。

3. **估计子词概率**

- 根据频率估计每个子词的概率 $P(s) = \frac{f(s)}{N}$ ，其中 $f(s)$ 是子词 s 的频率， N 是总的子词数。

4. **迭代合并子词**

- **步骤概览**：在每次迭代中，尝试合并所有可能的相邻子词对，**计算合并后对语料库总概率的影响，选择能最大化语料库概率的合并。**

5. **计算每个合并候选的得分**

- **目标**：找到能够最大化语料库对数概率 L 的合并。
- **语料库对数概率**：$L = \sum_{s \in S} f(s) \cdot \log P(s)$， 其中 S 是所有子词的集合。
- **合并后对数概率的增益（得分）**：
  $\Delta L = [f(ab) \cdot \log P(ab)] - [f(a) \cdot \log P(a) + f(b) \cdot \log P(b)]$
  - a 和 b 是待合并的子词。
  - ab 是合并后的新子词。
  - $f(a)$ 、 $f(b)$ 、 $f(ab)$ 是对应子词的频率。

6. **选择最佳合并**

  - 选择使 ΔL 最大的子词对进行合并。

7. **更新子词频率和概率**

- 更新合并后的子词频率 $f(ab)$。
- 从频率表中移除 $f(a)$ 和 $f(b)$。
- 重新计算子词概率。

8. **重复迭代**

- 重复步骤5-7，直到达到预设的词汇表大小或没有合适的合并。

小红书账号：小黄搞AI

30.3 示例解析

假设语料库：

```
unaffable
unacceptable
unaccountable
```

1. 初始状态

- 将词分解为字符序列：

```
u n a f f a b l e
u n a c c e p t a b l e
u n a c c o u n t a b l e
```

- **统计初始子词频率：**

子词	频率
u	3
n	3
a	6
f	2
...	...

- **计算初始子词概率：**

$P(s) = \frac{f(s)}{N}$ ，其中 N 是总子词频率的和。

2. 第一次合并候选

- **可能的合并对：**
- `u n` -> `un`
- `a f` -> `af`
- `f f` -> `ff`
- `a c` -> `ac`

小红书账号：小黄搞AI

- `c c` -> `cc`
- ...
- **计算每个合并的** $DeltaL$
- **示例**：计算合并 `u n` -> `un` 的 $DeltaL$
  - $f(un) = 3$ （`un` 出现了 3 次）
  - $P(un) = \frac{f(un)}{N}$
  - 计算 $\Delta L = [f(un) \cdot \log P(un)] - [f(u) \cdot \log P(u) + f(n) \cdot \log P(n)]$
- **对所有候选合并计算** $DeltaL$

3. 选择最佳合并并更新

- 假设 `u n` -> `un` 的 $DeltaL$ 最大，进行合并。
- **更新语料库**：

```
1 un a f f a b l e
2 un a c c e p t a b l e
3 un a c c o u n t a b l e
```

- **更新子词频率和概率**：
- 移除 `u` 和 `n` 的频率。
- 添加 `un` 的频率为 3。
- 重新计算子词概率。

4. 后续迭代

- 重复上述步骤，合并 `a c` -> `ac`、`c c` -> `cc` 等。
- 在每次迭代中，总是选择能最大化 $DeltaL$ 的合并。

30.4 直观理解

- **最大化语料库概率：** 通过选择合并后能最大提高语料库总概率的子词对，使得Tokenizer更能捕捉到常见的、具有语言学意义的子词。
- **避免低频、不重要的合并：** 由于 $DeltaL$ 考虑了频率和概率的乘积，低频或对概率贡献小的合并会被自然排除。
- **捕获有意义的子词：** 例如，`un`、`able` 等具有实际意义的子词更可能被合并，从而在分词时保留完整。

30.5 与BPE的对比

小红书账号：小黄搞AI

BPE

- **合并策略：**每次合并频率最高的符号对。
- **优点：**简单高效。
- **缺点：**只考虑了符号对的频率，可能合并出一些不具有语言学意义的子词。

WordPiece

- **合并策略：**每次合并能最大化语料库概率的符号对。
- **优点：**考虑了子词对整体语言模型的贡献，生成的子词更有意义。
- **缺点：**计算复杂度较高，需要计算每个合并候选的 $DeltaL$。

🍋 **总结**

- **WordPiece算法通过考虑语言模型的概率，在合并子词时选择能最大化语料库总概率的合并。**
- **这种方法使得生成的子词更符合语言习惯，更有利于下游任务的处理。**
- **与只考虑频率的BPE相比，WordPiece生成的词汇表在实践中往往表现更好。**

31. 介绍一下BPE&BBPE这两种Tokenization方法

BPE（Byte Pair Encoding）和 BBPE（Byte-Level BPE）是自然语言处理中的两种常用的子词分割算法。它们旨在解决词汇表规模和模型泛化能力之间的平衡问题，通过将单词分割为较小的子词单元来降低词汇表的大小，同时保持良好的语言表示能力。

1. BPE（Byte Pair Encoding）

BPE是一种基于频率的贪心算法，最初用于数据压缩。它的核心思想是**通过反复合并出现频率最高的字符对（或子词对），从而生成一组子词单元**。BPE特别适合处理低频词汇的问题，因为它可以有效减少词汇表中的稀有词。

算法步骤

- 初始化时，**将每个单词分解为字符序列**。
- 计算所有字符对的出现频率，找到频率最高的字符对，**并将其合并为一个新子词**。
- 重复这个过程，**直到达到预设的子词个数**。

BPE的伪代码

📌 Input: 单词列表 W 合并次数 N
Output: 子词表
1. 将每个单词表示为字符序列
2. 初始化子词表为单个字符的集合
3. for i in 1 to N:
3.1 计算每个字符对的频率

小红书账号：小黄搞AI

3.2 找到频率最高的字符对 (x, y)

3.3 合并字符对 (x, y) ，更新子词表

4. 返回子词表

数学表示

设字符对为 (x_i, x_{i+1}) ，其频率为 $f(x_i, x_{i+1})$ ，则每次迭代选择频率最高的字符对 $argmax(f(x_i, x_{i+1}))$ ，并将其替换为一个新子词 z ： $x_i, x_{i+1} \rightarrow z$

BPE通过多次这样的操作，最终形成一组有效的子词集合，可以同时处理频繁词汇和低频词汇。

BPE的优点

- 能够有效处理词汇稀疏性，尤其是低频词。
- 动态生成子词单元，使模型的词汇表规模可控。

2. BBPE（Byte-Level BPE）

BBPE是BPE的字节级扩展版本，主要应用于多语言模型（如GPT等）和处理Unicode字符的场景。与标准BPE不同，**BBPE操作在字节层面**，而不是字符层面，这允许它以更加通用的方式处理任意编码的文本（包括非拉丁字母）。

BBPE的主要区别

- BBPE对文本进行字节级别的操作，而不是字符级别。**文本首先被转换为字节序列，再使用BPE进行子词分割。**
- 这种方法特别适合处理复杂的字符集，如汉字、阿拉伯字母等。

BBPE的算法步骤

与BPE类似，不同之处在于：

- 输入的不是字符序列，而是**UTF-8字节序列**。
- 在**字节层面**合并频繁出现的字节对。

3. BPE & BBPE的实际应用

- BPE在各种自然语言处理任务中都有广泛应用，如机器翻译（如Google的神经机器翻译系统）。
- BBPE在多语言模型中使用频繁，特别是在跨语言预训练模型中（如GPT、BERT等）。

4. BPE 和 BBPE 的优缺点对比

小红书账号：小黄搞AI

方法	优点	缺点
BPE	- 子词的生成更符合语言学规则，可以利用词缀和词根的信息。 - 对常见词汇能更好地处理。	- 对于不同语言或复杂字符集（如中文、阿拉伯语）处理不够灵活。 - 无法有效处理表情符号和特殊字符。
BBPE	- 能处理所有字符集（包括Unicode字符），适用于**多语言场景**。 - 对非常规符号和表情支持良好。	- 生成的token化结果在字符层面操作，词汇表可能会增大。 - 子词切分可能不符合语言规律，影响语义一致性。

5. 主流大模型架构采用的方法

目前，主流的大模型（如GPT系列、BERT、T5等）大多数采用的是**BBPE（Byte-Level BPE）**或类似的字节级tokenization方法。这种方法广泛应用的原因包括：

处理多语言文本的能力： 大模型通常需要处理大量**不同语言的文本**，而BBPE在字节级别操作，能够处理所有Unicode字符，特别是对于包含复杂字符集的语言（如中文、韩语、阿拉伯语）表现良好。

统一的tokenization： BBPE方法不依赖于语言特定的字符结构，因此可以在不增加额外复杂性的情况下，统一应用于多语言任务，简化了预训练和下游任务中的tokenization流程。

兼容表情符号和特殊字符： 现代大语言模型需要处理大量互联网数据，这些数据中包含许多表情符号、特殊字符以及非标准符号，而**BBPE可以更好地支持这类符号**。

6. 为什么主流大模型使用BBPE而非BPE?

多语言与统一性： 随着模型规模的扩大，**支持多种语言和字符集成为必然需求**。BBPE通过在字节层面的操作，能够避免各语言tokenization不一致的问题。

处理多种字符： 由于BBPE可以**处理所有Unicode字符**，能够无缝处理非拉丁字符和特殊符号，具备更好的通用性。

7. 总结

BPE 和 BBPE 通过将词分割成子词的方式来平衡词汇表的大小与语言建模的泛化能力。BPE适合处理较为常规的语言任务，而BBPE适合多语言和字符集复杂的场景。

32. 介绍一下RoPE位置编码

在Transformer中，位置编码（Positional Encoding, PE）是为了让模型感知序列中元素的顺序。

传统Transformer使用的是**绝对位置编码**，即对序列中每个位置赋予一个唯一标识符，例如**Sinusoidal位置编码**，通过一组固定的正弦和余弦函数编码每个位置。然而，绝对位置编码存在**局限性：无法直接捕捉元素之间的相对位置信息**，而后者对很多自然语言任务至关重要。相对位置编码能让模型知道两个词之间的距离，这在如句法分析等任务中非常有用。

小红书账号：小黄搞AI

1. RoPE的动机

RoPE的目标是：**通过绝对位置编码的方式实现相对位置编码**，即不改变现有Transformer框架下绝对位置编码的方式，但能让模型自动捕捉元素之间的相对位置信息。

RoPE的理论基础是通过**在查询（query）和键（key）向量上施加一个旋转变换**，将位置信息嵌入这些向量中。RoPE对位置编码的设计不仅适用于传统的Attention机制，还能扩展到**线性Attention**，这一点是其创新所在。

2. RoPE的核心思想

假设序列的查询和键向量分别为 q 和 k，我们需要设计一个函数 $f(q,m)$ 和 $f(k,n)$，将查询和键向量转换为带有位置信息的形式。RoPE通过将位置信息以**旋转变换**的形式嵌入向量，实现相对位置编码。

我们期望Attention机制的核心运算**内积运算（dot product）**带有相对位置信息。因此，需要满足以下恒等式：

$$\langle f(q,m), f(k,n) \rangle = g(q,k,m-n)$$

📌 **m**：表示查询向量 q 在序列中的位置索引。例如，如果你有一个序列 $[w_1, w_2, w_3, \ldots, w_n]$，那么查询 q 对应于某个特定的序列元素，它的索引是 m，即 $q = q_m$，表示第 m 个位置的查询向量。

n：表示键向量 k 在序列中的位置索引。同理，键向量 $k = k_n$ 对应序列中的第 n 个元素。

m - n：表示查询和键之间的相对位置差，这正是RoPE希望通过旋转变换捕捉的位置信息。

即在内积运算中，查询和键向量的相对位置 $m-n$ 影响最终的Attention权重。这要求 $f(q,m)$ 和 $f(k,n)$ 的设计不仅带有绝对位置信息，还能让它们的内积反映出相对位置信息。

3. 推导过程

二维情形下的推导

📌 **通俗解释：**

假设我们有两个向量，一个是查询向量 q，一个是键向量 k，我们希望它们不仅包含内容的信息，还能够通过某种方式带上它们的位置（位置编码）。通过将每个向量旋转一定角度，我们就可以让这些向量携带它们在序列中的位置信息。

旋转角度由它们的序列位置决定（位置越靠后，旋转的角度越大）。当我们计算它们的内积时，复数乘法让旋转后的向量与原始向量结合起来，最终使得Attention计算中能够反映出查询和键的**相对位置**。

总结：

- 用复数是为了方便表达**旋转**，而旋转是用来嵌入位置编码的方式

小红书账号：小黄搞AI

- 复数的旋转特性使得我们可以用一个很简洁的方式在向量中加入位置信息，从而让模型不仅知道查询和键的内容，还知道它们在序列中的相对位置。

为了简化推导，首先考虑查询和键向量在二维空间的形式。可以用**复数**表示它们之间的内积：

$$\langle f(q,m), f(k,n)\rangle = \mathrm{Re}[f(q,m)f^*(k,n)]$$

其中 f^* 是复数的共轭， $\mathrm{Re}[\cdot]$ 代表取实部。我们假设函数 $f(q,m)$ 的形式为一个旋转变换：

$$f(q,m) = qe^{im\theta}$$

即将查询向量 q 乘以一个旋转角度 $m\theta$ 的复数因子。类似地，键向量 k 也会乘以相应的旋转因子。

> ***为什么使用复数?***
>
> 复数在数学中有一个重要的特性——它可以自然地表示**旋转**。复数的形式为 $a+bi$ （其中 i 是虚数单位），而当我们用**极坐标**表示复数时，形式为 $re^{i\theta}$
>
> 其中 r 是复数的模（代表大小），而 θ 是**角度**（代表方向）。
>
> 当我们乘以 $e^{i\theta}$ 时，本质上是在做一个**旋转**操作，将复数绕原点旋转 θ 角度。这是利用复数来表示旋转的关键原因。

矩阵形式的旋转变换： 复数旋转可以推广到高维情形。对于偶数维的查询和键向量，RoPE通过一个旋转矩阵来编码位置。二维情况下的旋转矩阵为：

$$R_m = \begin{pmatrix} \cos(m\theta) & \sin(m\theta) \\ -\sin(m\theta) & \cos(m\theta) \end{pmatrix}$$

这个矩阵用于变换每对相邻的维度，以嵌入位置 m 的信息。对于高维向量，RoPE将多个二维旋转矩阵组合成一个大矩阵，分块处理每两个相邻的维度。

RoPE的高维实现： 假设查询和键向量的维度为 d ，我们可以将它们的每两个相邻维度作为一组，应用二维旋转矩阵。这样，RoPE的编码过程可以描述为对每个位置 m ，将向量 q 乘以如下的旋转矩阵：

$$R_m = \mathrm{diag}\left(\begin{pmatrix} \cos(m\theta_0) & \sin(m\theta_0) \\ -\sin(m\theta_0) & \cos(m\theta_0) \end{pmatrix}, \begin{pmatrix} \cos(m\theta_{d/2-1}) & \sin(m\theta_{d/2-1}) \\ -\sin(m\theta_{d/2-1}) & \cos(m\theta_{d/2-1}) \end{pmatrix}\right)$$

其中，每个 θ_i 是与维度 i 相关的角度偏移量。一般选择 $\theta_i = 10000^{-2i/d}$ ，类似于Sinusoidal位置编码中的设计。

> ### 例子：假设有一个 6 维查询向量
>
> 假设我们有一个查询向量 q ，它的维度是 6，也就是 $q = [q_1, q_2, q_3, q_4, q_5, q_6]$ ，其中每个 q_i 都是一个标量。RoPE 的做法是将这个 6 维向量**分成多个2维的小块**，对每一个小块应用二维旋转。

小红书账号：小黄搞AI

1. 将向量分组

我们把 6 维向量分成 3 组，每组2个维度：

- 第1组：(q_1, q_2)
- 第2组：(q_3, q_4)
- 第3组：(q_5, q_6)

2. 应用二维旋转矩阵

对每一组应用二维旋转矩阵。假设位置 m 是 1（代表第1个位置），旋转角度 θ 是一个固定值，那么对于第1组 (q_1, q_2)，应用旋转矩阵：

$$R_m = \begin{pmatrix} \cos(m\theta) & \sin(m\theta) \\ -\sin(m\theta) & \cos(m\theta) \end{pmatrix} = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ -\sin(\theta) & \cos(\theta) \end{pmatrix}$$

假设 $\theta = 30^\circ$，那么矩阵变为：

$$R_1 = \begin{pmatrix} \cos(30^\circ) & \sin(30^\circ) \\ -\sin(30^\circ) & \cos(30^\circ) \end{pmatrix} = \begin{pmatrix} \frac{\sqrt{3}}{2} & \frac{1}{2} \\ -\frac{1}{2} & \frac{\sqrt{3}}{2} \end{pmatrix}$$

接着用这个旋转矩阵去变换向量 (q_1, q_2)，得到新的值：

$$\begin{pmatrix} q_1' \\ q_2' \end{pmatrix} = R_1 \cdot \begin{pmatrix} q_1 \\ q_2 \end{pmatrix} = \begin{pmatrix} \frac{\sqrt{3}}{2} & \frac{1}{2} \\ -\frac{1}{2} & \frac{\sqrt{3}}{2} \end{pmatrix} \cdot \begin{pmatrix} q_1 \\ q_2 \end{pmatrix}$$

通过矩阵乘法，可以得到新的 q_1' 和 q_2'：

$$q_1' = \frac{\sqrt{3}}{2} q_1 + \frac{1}{2} q_2$$

$$q_2' = -\frac{1}{2} q_1 + \frac{\sqrt{3}}{2}$$

同样的操作对第2组 (q_3, q_4) 和第3组 (q_5, q_6) 分别应用旋转矩阵：

$$\begin{pmatrix} q_3' \\ q_4' \end{pmatrix} = R_1 \cdot \begin{pmatrix} q_3 \\ q_4 \end{pmatrix} \quad \text{以及} \quad \begin{pmatrix} q_5' \\ q_6' \end{pmatrix} = R_1 \cdot \begin{pmatrix} q_5 \\ q_6 \end{pmatrix}$$

这样处理之后，查询向量变成了一个带有位置信息的新向量：

$$q' = [q_1', q_2', q_3', q_4', q_5', q_6']$$

📌 总结：

1. 二维旋转矩阵的性质保证了**对局部维度的操作不会破坏向量的整体特征**。
2. 相对位置信息嵌入是自注意力机制的核心，而**RoPE通过旋转的方式实现**了这一点。
3. **局部两两分组处理可以减少计算复杂度**，同时保持高效的位置信息编码。

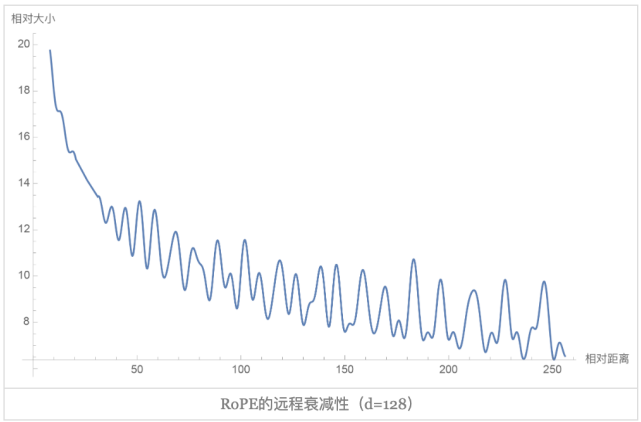
自动实现相对位置编码：通过在查询和键向量上施加旋转变换，RoPE能够保证在计算Attention时，内积运算会自动引入相对位置信息。这是因为：

小红书账号：小黄搞AI

$$\langle R_m q, R_n k \rangle = q^\top R_{m-n} k$$

这里 R_{m-n} 表示一个与位置差值 $m-n$ 相关的旋转矩阵，直接嵌入了相对位置信息。

4. RoPE的远程衰减性



RoPE的远程衰减性（d=128）

RoPE不仅实现了相对位置编码，还具备远程衰减的特性。随着查询和键之间的位置差值 $m-n$ 增大，旋转角度也随之增加，导致内积结果逐渐减小。这种特性对处理长距离依赖非常有帮助，能够让模型在长序列中自然降低对远距离元素的注意力权重。

这个衰减特性可以通过数学证明来展示。将 q 和 k 视为复数，并考虑它们加上RoPE后的内积，最终结果会随着 $m-n$ 的增加而逐渐减小。

5. RoPE在线性Attention中的应用

RoPE的另一个亮点是它可以用于**线性Attention**。大多数相对位置编码方法需要在计算Attention矩阵时进行操作，而线性Attention的设计不直接计算该矩阵，因此其他相对位置编码方案无法应用。

RoPE则不同，它通过对查询和键向量施加旋转变换嵌入位置信息，不依赖于Attention矩阵的具体形式。因此，它可以直接应用到线性Attention中，实现长序列处理时的高效计算。

在RoPE插入线性Attention的公式中，我们只需将查询和键向量用旋转矩阵变换即可：

$$\sum n[Ri\phi(qi)]\top[Rj\phi(kj)]vj$$

其中 R_i 和 R_j 分别是与位置 i 和 j 相关的旋转矩阵， $\phi(\cdot)$ 是非负的激活函数。

6. 总结

RoFormer通过引入RoPE，为Transformer提供了一种**相对位置编码**的方案，保留了绝对位置编码的简单性，同时增强了长序列任务的处理能力。RoPE设计的旋转变换具有理论上的优雅性和实践上的高效性，并且能推广到线性Attention中，是目前唯一适用于该场景的相对位置编码方案。

33. 旋转位置编码为什么比绝对位置编码更好

1. 绝对位置编码（PE）

频率递减机制：

公式：

$$PE(p, 2i) = \sin\left(\frac{p}{10000^{2i/d}}\right)$$

$$PE(p, 2i+1) = \cos\left(\frac{p}{10000^{2i/d}}\right)$$

小红书账号：小黄搞AI

关键点:

- 绝对位置编码**使用正弦和余弦函数来编码每个位置的绝对位置**。
- 频率递减机制是通过指数递增的分母（**10000的幂**）来实现的。对于较高的维度 $2i$，分母的值更大，从而使得频率较低。这使得**高维度的编码可以捕捉长距离的信息，而低维度的编码则捕捉短距离的信息**。

周期性和衰减：

- 每个位置的编码是**周期性**的，因为正弦和余弦函数本身是周期函数。
- 高频成分（低维度）可以捕捉细粒度的位置信息，而低频成分（高维度）捕捉较粗略的位置信息。

2. 旋转位置编码（RoPE）

频率递减机制：

公式:

$$f(q,m) = q \cdot e^{im\theta}$$
$$f(k,n) = k \cdot e^{in\theta}$$

关键点:

- RoPE通过将**查询和键向量与旋转因子相乘来嵌入位置编码**。这个旋转因子是复数形式的旋转矩阵。
- 旋转矩阵用于处理每对相邻的维度，**通过不同的旋转角度（基于位置 m 和 n）来实现频率递减**。
- 旋转矩阵的角度是与位置有关的，因此**相对位置信息能够直接影响注意力计算**。这与绝对位置编码不同，因为RoPE直接嵌入了相对位置信息。

旋转变换：

对于高维向量，RoPE将每两个相邻的维度作为一组，应用二维旋转矩阵。这个旋转矩阵的形式是：

$$R_m = \text{diag}\left(\begin{pmatrix} \cos(m\theta_0) & \sin(m\theta_0) \\ -\sin(m\theta_0) & \cos(m\theta_0) \end{pmatrix}, \ldots, \begin{pmatrix} \cos(m\theta_{d/2-1}) & \sin(m\theta_{d/2-1}) \\ -\sin(m\theta_{d/2-1}) & \cos(m\theta_{d/2-1}) \end{pmatrix}\right)$$

RoPE通过旋转变换将位置信息嵌入到向量中，使得相对位置在编码中得以体现。这种旋转变换基于每个位置的相对差异，而不是绝对位置。

远程衰减：

- 旋转位置编码中的频率递减由旋转角度 $m\theta$ 控制，不同的角度对于不同的维度提供了不同的频率。

小红书账号：小黄搞AI

3. 引入相对位置信息

绝对位置编码：在传统的绝对位置编码方法中，如正弦和余弦函数的编码方式，**每个位置 p 的编码是基于固定的数学公式生成的**。具体来说，位置编码为：$\mathrm{PE}(p, 2i) = \sin\left(\frac{p}{10000^{2i/d}}\right)$ 和 $\mathrm{PE}(p, 2i+1) = \cos\left(\frac{p}{10000^{2i/d}}\right)$，这种方法为每个位置生成唯一的向量表示，能够编码每个位置的绝对信息，但不直接表达位置之间的相对关系。

旋转位置编码（RoPE）：**RoPE通过将位置编码嵌入到旋转变换中**，使得位置之间的相对关系能够直接影响最终的Attention权重。具体而言，RoPE通过旋转矩阵来对每个位置的编码进行变换，使得相对位置 $m - n$ 在计算注意力时自然地被考虑进来。位置编码形式为：$f(q, m) = q \cdot e^{im\theta}$ 和 $f(k, n) = k \cdot e^{in\theta}$。RoPE通过这种设计使得模型不仅保留了绝对位置信息，还能直接学习到位置之间的相对距离和关系。这对于捕捉长程依赖和相对位置信息尤其重要。

4. 增强模型的表示能力

绝对位置编码的局限：绝对位置编码固然有效，但它**不具备自然表达相对距离的能力**。对于长文本或复杂序列，绝对位置编码可能无法有效地捕捉到不同位置之间的细微关系，尤其是在处理长距离依赖时。

RoPE的优势：RoPE通过旋转矩阵对每个位置进行编码，使得向量的**旋转角度反映了位置之间的相对关系**。这样可以使模型在计算注意力时自然地包含相对位置的信息，从而提升模型对序列中元素相对位置关系的敏感度和处理能力。

5. 提升模型的泛化能力

绝对位置编码的限制：绝对位置编码的设计依赖于固定的数学公式（如正弦余弦），这可能在一些情况下限制了模型的泛化能力，因为**它无法适应不同类型的序列或在序列长度变化时的灵活性**。

RoPE的灵活性：RoPE通过对位置编码进行旋转变换，使得位置之间的关系可以被直接学习和调整。**旋转编码的这种设计使得模型能够更好地泛化到不同长度和不同类型的序列**，因为它能够直接捕捉和利用位置之间的相对关系。

6. 理论支持和实践效果

理论支持：在理论上，**RoPE的设计使得位置编码不仅包含绝对信息，还自然地融入了相对信息**。这种设计有助于模型在Attention机制中进行更灵活的计算，使得相对位置的影响在计算注意力时得以体现。

实践效果：在实际应用中，RoPE已被证明在许多任务中表现优于传统的绝对位置编码，**尤其是在需要捕捉长距离依赖关系的任务中**，如机器翻译、文本生成等。这表明RoPE在实际场景中能够更好地处理序列数据，并取得更好的性能。

7. 总结

旋转位置编码（RoPE）通过引入相对位置信息、增强模型的表示能力、提升泛化能力，并且在理论和实践中显示出优越性，使其比传统的绝对位置编码更具优势。这种方法不仅保留了绝对位置信息，还

小红书账号：小黄搞AI

能使模型在处理序列时更有效地捕捉和利用相对位置关系，从而提高模型的整体性能。

34. RMSNorm正则化有什么好处和优势

1. LayerNorm 的定义

LayerNorm 是一种广泛应用于神经网络中的正则化技术，主要用于稳定深层网络的训练，避免梯度消失或爆炸。它通过对输入进行均值和方差归一化来标准化每一层的激活。

给定一个输入向量 $x \in \mathbb{R}^d$，LayerNorm 的标准化过程如下：

> 其中 γ 和 β 是与输入维度 d 大小相同的可学习参数。

1.计算均值

LayerNorm 先计算输入向量的均值 μ：

$$\mu = \frac{1}{d}\sum_{i=1}^{d} x_i$$

2.计算方差

第二步计算输入向量的方差 σ^2：

$$\sigma^2 = \frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2$$

3.归一化

对每个输入进行归一化，使用均值 μ 和方差 σ^2（ϵ 是防止除零的小常数。）

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

4.重缩放与偏移

最终，用可学习的参数 γ 和 β 对归一化的输出进行重缩放和偏移：

$$y = \gamma\hat{x} + \beta$$

> 📌 **数学补给站：方差 (Variance)**
>
> 方差描述的是一组数据的**离散程度**，即每个数据点与**均值**的偏离程度。方差越大，表示数据点的分布越分散；方差越小，表示数据点集中在均值附近。

2. RMSNorm 的定义

RMSNorm 则不涉及均值和方差的计算，而是通过均方根（Root Mean Square, RMS）来进行规范化。其核心思想是基于输入的幅值（magnitude），而不依赖于其均值。

给定相同的输入向量 $x \in \mathbb{R}^d$，RMSNorm 的计算步骤如下：

1.计算均方根

RMSNorm 计算输入向量的均方根 $\mathrm{RMS}(x)$：

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

2.归一化

将输入向量 $x \in \mathbb{R}^d$ 用 RMS 归一化（ϵ 用于防止除零错误）：

$$\hat{x} = \frac{x}{\mathrm{RMS}(x) + \epsilon}$$

3.重缩放与偏移

最后，通过可学习的参数 γ 进行重缩放，并加上偏置 β：

$$y = \gamma\hat{x} + \beta$$

> 📌 **数学补给站：均方根 (RMS)**
>
> 均方根是指一组数据的**平方的平均值的平方根**，它通常用来衡量数据的**幅度大小**，而不关心数据的偏离均值的情况。特别是当数据既有正值又有负值时，均方根通过平方操作消除符号

小红书账号：小黄搞AI

的影响。均方根关注的是每个数据点相对“远离零点”的程度，而不是偏离均值的情况。

3. RMSNorm 与 LayerNorm 的详细对比

特性	LayerNorm	RMSNorm
均值计算	需要计算均值 $\mu = \frac{1}{d}\sum_{i=1}^{d} x_i$	不计算均值
方差计算	需要计算方差 $\sigma^2 = \frac{1}{d}\sum_{i=1}^{d}(x_i - \mu)^2$	不计算方差
归一化方法	基于均值和方差 $x_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$	基于均方根 $\hat{x} = \frac{x}{\mathrm{RMS}(x) + \epsilon}$
正则化对象	对输入进行零均值和单位方差的归一化	仅对输入幅值进行幅度归一化
计算复杂度	需要计算均值、方差，复杂度较高	只计算均方根，计算量更低
对小尺度特征的影响	可能会导致较小的特征值被过度平滑	对特征值影响较小，尤其是小尺度特征
高维数据表现	在高维数据上表现良好	在高维数据上表现更稳定，且具有更高效率
在深层网络中的效果	随着层数增加，过度平滑问题较为显著	深层网络中效果更佳，特征信息保留更好
梯度回传的稳定性	提供一定的梯度稳定性，防止梯度爆炸或消失	更稳定的梯度回传，尤其在复杂模型中
对训练收敛速度的影响	提供更稳定的收敛速度，适用于一般任务	提供较快的收敛速度，适用于需要更快计算的任务

4. 方法有效性分析

幅度归一化 vs. 偏移归一化

LayerNorm 通过均值归零和方差归一化，确保数据的平均值为 0，标准差为 1。这在某些情况下有助于稳定训练，但并不总是必要。对数据的均值偏移并不是所有任务的关键。例如，在一些任务中，输入数据的**幅值大小**（即输入的整体能量）可能比相对位置（即偏移量）更为重要。

RMSNorm 通过均方根来标准化幅度，消除对输入数据的过度缩放，而不会干扰数据的相对位置或特征。这样做的优势是，它保持了输入的**特征分布结构**，但调整了输入的整体尺度。对于一些深度网

小红书账号：小黄搞AI

络，尤其是自注意力机制（如 Transformer 结构）中，这种幅度归一化在不破坏输入特征的情况下，有助于提高模型的训练效果。

特定任务不需要零均值： 并不是所有任务都需要输入数据有零均值。例如，在一些计算过程中（如神经网络的某些层或处理高维数据时），数据的幅度变化对模型更为重要，而数据相对于均值的偏移则无关紧要。因此，RMSNorm 通过幅值归一化来控制数据在一定范围内波动，不必通过均值归零。

计算复杂度

LayerNorm 需要计算均值和方差，每个步骤都涉及对所有输入维度的运算，计算复杂度为 $O(d)$。RMSNorm 只需计算平方和的均方根，省略了均值和方差的步骤，进一步减少了计算量。虽然两者的计算复杂度理论上同为 $O(d)$，但 RMSNorm 的实际计算开销要低。

深层网络的表现

在深层神经网络中，LayerNorm 可能会出现**过度平滑**的问题，特别是当网络深度增加时，**过度依赖均值归一化会导致重要特征被抑制**。RMSNorm 由于只基于输入幅度进行规范化，避免了均值归零的限制，因此在深层网络中能够更好地保留特征信息。

梯度的稳定性

两者都能在一定程度上防止梯度爆炸或消失的问题，但 **RMSNorm 的梯度稳定性在某些复杂的深度模型中表现得更为突出**，因为它通过幅度归一化使得梯度的幅值保持在合理的范围内，不会因为均值和方差的波动而受到影响。

5. 总结

LayerNorm 和 RMSNorm 都是有效的正则化方法，但它们在核心的计算方式和应用场景上有所不同：

- **LayerNorm** 更适合处理**均值与方差**对特征影响较大的任务，特别是小批量数据和 NLP 任务中。
- **RMSNorm** 则适用于**幅度归一化**为主、特征幅度较大的场景，如深层神经网络或高维数据中，同时它的计算效率也更高。

35. 介绍一下Bert的结构

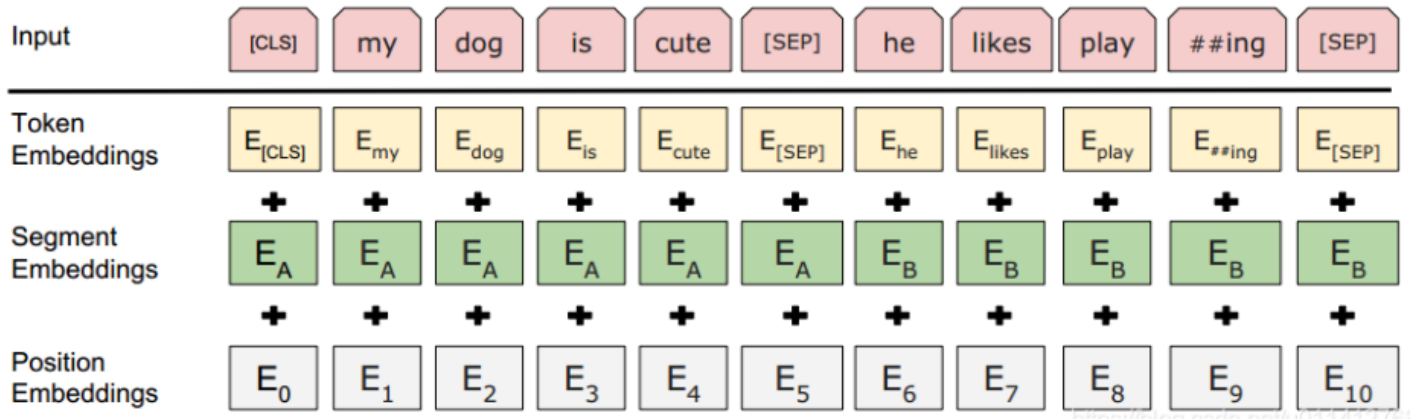
BERT (Bidirectional Encoder Representations from Transformers) 是基于 Transformer 架构的预训练模型。其核心在于利用 Transformer 模型中的 **编码器 (Encoder)** 部分，通过**双向**（从左到右、从右到左）的上下文信息来理解句子中的词语。因此，BERT 能够更好地捕捉语言的语义关系，广泛应用于自然语言处理 (NLP) 任务。

1. BERT 模型的核心结构

输入表示 (Input Representation)

> 《BERT的嵌入层是如何实现的？看完你就明白了》
> https://mp.weixin.qq.com/s/DfIAuo775_sHGYi5z9IZyw

小红书账号：小黄搞AI



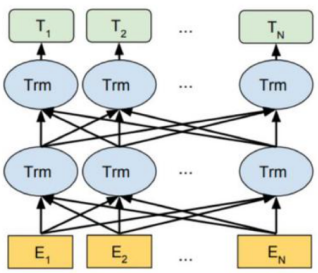
BERT模型的输入

BERT 的输入不仅是简单的单词向量，还包含了丰富的上下文信息，它由以下三种嵌入 (Embedding) 的求和表示：

- `WordPiece Embedding`：**BERT 使用 WordPiece 作为分词方法，将单词划分为子词单元。**这种处理方式既能处理未知词汇，又能提高模型的灵活性和泛化能力。例如，罕见或不规则单词会被分成更常见的子词单位，进而能够在训练中更好地学习到词汇语义。
- `位置嵌入 (Position Embedding)`：由于 BERT 只使用 Transformer 的编码器部分，并不依赖于序列化结构（如 RNN 或 LSTM），因此它无法从输入序列中自然地获取位置信息。为了弥补这一点，**BERT 通过位置嵌入为每个词汇添加了位置特征**，使模型能够感知词汇在序列中的相对位置。BERT 初始化了一个位置嵌入矩阵，并在训练过程中学习这些位置向量。
- `段落嵌入 (Segment Embedding)`：在 BERT 中，**输入通常是两个句子拼接而成，特别是在句子预测任务 (Next Sentence Prediction, NSP)** 中。因此，BERT 为输入中的每个 token 添加一个段落嵌入，用来区分句子 A 和句子 B，帮助模型更好地理解句子之间的关系。

最终，BERT 的输入是这三种嵌入的求和：**wordpiece embedding + position embedding + segment embedding**，这样能够同时捕捉词汇、位置、以及句子间的信息。

`Transformer 编码器层 (Transformer Encoder Layers)`



BERT论文中的网络结构

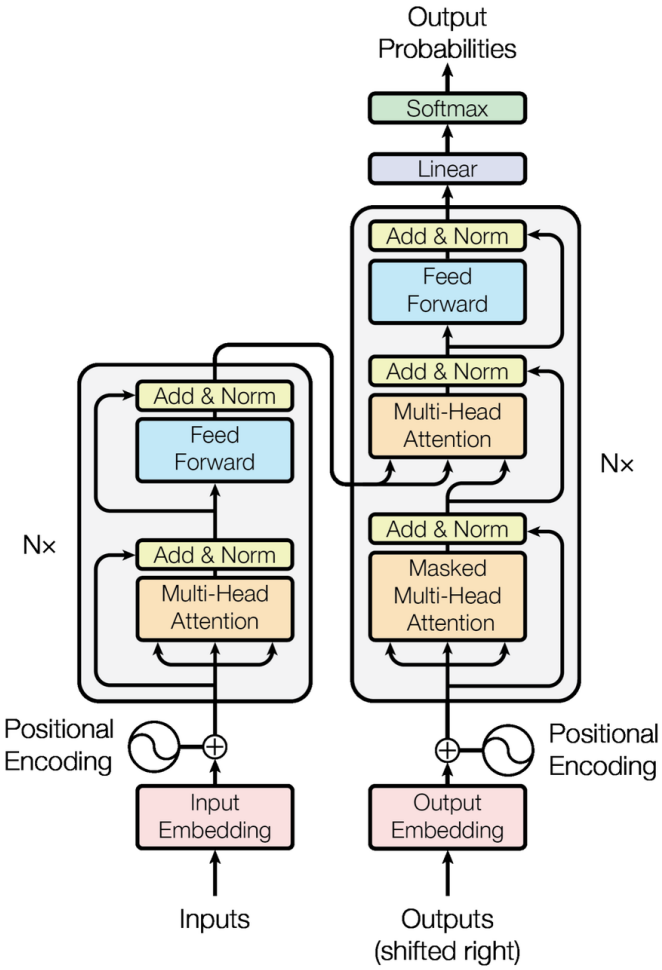
BERT 的主要结构是基于 **Transformer 的编码器部分**，通过堆叠多层编码器来实现深度语义学习。Transformer 编码器包含几个核心部分：

multi-head-Attention + Layer Normalization + feedforword + Layer Normalization 叠加产生，BERT的每一层由一个这样的encoder单元构成。

📌
- **BERT-base** 版本有 12 层编码器，每层有 12 个多头自注意力机制 (Multi-Head Attention)，词向量的维度是 768。

小红书账号：小黄搞AI

- **BERT-large** 版本有 24 层编码器，每层有 16 个多头自注意力机制，词向量的维度是 1024。
- 在所有情况下，将**feed-forward/filter** 的大小设置为 4H（H为词向量的维度），即H = 768时为3072，H = 1024时为4096。



标准的transformer结构（左边是encoder部分）

每一层编码器的结构可以拆解为以下几个组成部分：

- **多头自注意力机制 (Multi-Head Self-Attention)**

  BERT 的核心是自注意力机制，它能在同一句子中计算**每个词与其他词之间的相关性，捕捉句子的全局信息**。自注意力能够解决传统 RNN 模型中长距离依赖问题，适合并行处理大规模数据。

- **前馈神经网络 (Feed-Forward Network, FFN)**

  自注意力层之后，每个词的表示都会经过一个前馈神经网络进行进一步处理，通常是**两层线性变换和激活函数 (如 GELU)**。

- **层归一化 (Layer Normalization)**

  每一层的输出都会经过层归一化，以**确保训练的稳定性和加速收敛**。与传统的批归一化 (Batch Normalization) 不同，层归一化在每个样本的内部进行归一化处理，使其更适合处理序列化数据。

自注意力机制 (Self-Attention Mechanism)

self-attention出现的原因

- 为了解决RNN、LSTM等常用于处理序列化数据的网络结构无法在GPU中**并行加速计算**的问题
- 由于每个目标词是直接与句子中所有词分别计算相关度(attention)的，所以**解决了传统的RNN模型中长距离依赖的问题**。通过attention，可以将两个距离较远的词之间的距离拉近为1直接计算词的相关度，而传统的RNN模型中，随着距离的增加，词之间的相关度会被削弱。

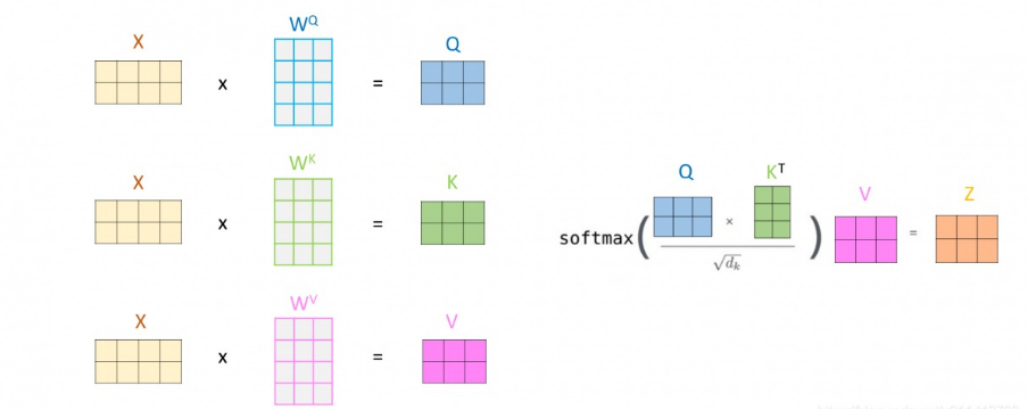
模型的输入

就是前面BERT模型的输入，即 **X=(batch_size, max_len, embedding)**，假设batch_size=1，输入的句子长度为512，每个词的向量表示的长度为768，那么整个模型的输入就是一个512*768的tensor。

小红书账号：小黄搞AI

单个self-attention 的计算过程

每一次的self-attention的计算涉及到三个中间权重矩阵Wq,Wk,Wv，他们分别对输入的X进行线性变换，生成query、key和value这三个新的tensor，整个的计算步骤如下：



单个self-attention的计算过程

step 1

输入 x 分别与 W_q 、W_k 、W_v 矩阵相乘，得到 Q 、K 、V

step 2

Q ，K^T 矩阵相乘，得到 x 中各个词之间的相关度，并scale（为了防止结果过大，除以他们维度的均方根）。

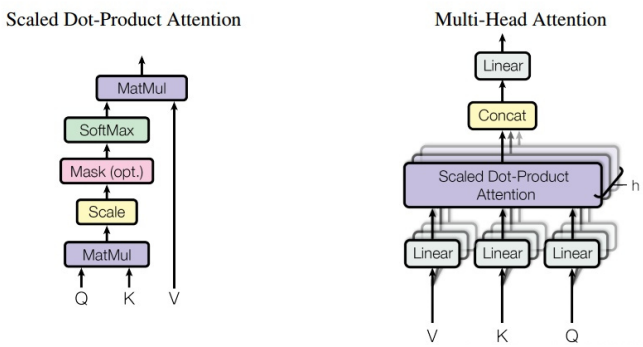
step 3

将第二步的相关度通过**Softmax**函数归一化，得到归一化后各个词与其他词的相关度。

step 4

将第三步的相关度矩阵与 V 相乘，即加权求和，得到每个词新的向量编码。

multi-head attention的计算



self-attention计算的可视化

Multi-Head Self-Attention**将多个不同单头的**Self-Attention输出**Concat**成一条，然后再经过一个全连接层降维输出。例如，一个self-attention计算的输出为output_0 = **(batch_size, max_len, w_length)**，那么n个attention进行concat之后，输出就为output_sum = **(batch_size, max_len,n * w_length)**，这个concat的结果再连一层全连接层即为整个multi-head attention的输出。如下图所示，右边的部分即为一个multi-head attention的计算过程，其中的h指的是attention的个数，即上面例子中的n。

在 **BERT-base** 中，每个自注意力头的输出维度是 64（总维度为 768），而多头注意力机制则是将 12 个自注意力头的输出拼接 (concatenate) 后，再通过一个线性层处理，形成最终的多头注意力输出。

小红书账号：小黄搞AI

预训练任务（Pre-training Tasks）

BERT 在大规模语料上通过自监督学习进行了预训练，主要包括两个任务：

- **Masked Language Model (MLM)**：

  BERT 通过随机掩盖 (mask) 句子中 15% 的 token，并要求模型根据上下文预测这些被掩盖的词。具体操作是：

  - 80%用 [MASK] 替换 token。
  - 10% 用随机 token 替换。
  - 10% 保持原 token 不变。

- 这种策略既能防止模型只依赖 [MASK] 进行猜测，同时保证了模型的泛化能力。

- **Next Sentence Prediction (NSP)**：

BERT 通过给定两个句子，要求模型判断这两个句子是否为连续上下句。这一任务帮助 BERT 学习句子间的关系，提高在问答、推理等任务中的表现。语料中50%的句子，选择其相应的下一句一起形成上下句，作为正样本；其余50%的句子随机选择一句非下一句一起形成上下句，作为负样本。这种设定，有利于sentence-level tasks，例如问答。

> 作者特意说了语料的选取很关键，要选用document-level的而不是sentence-level的，这样可以具备抽象连续长序列特征的能力。

模型的规模与超参数

- BERT 的 **base 版本** 有 110M 参数（12 层、768 隐藏单元、12 个注意力头）。
- **large 版本** 则有 340M 参数（24 层、1024 隐藏单元、16 个注意力头）。

BERT 的预训练采用了 Adam 优化器，初始学习率经过 10,000 步的 warm-up 后逐渐衰减，总共进行了超过 33 亿个训练样本、100 万步的训练过程。

2. BERT模型特点

- 使用transformer作为算法的主要框架，transformer能**更彻底的捕捉语句中的双向关系**；
- 使用了mask language model 和next sentence prediction的多任务训练目标，**是一个自监督的过程，不需要数据的标注**；
- 使用**tpu这种强大的机器训练了大规模的预料**，是NLP的很多任务达到了全新的高度。

BERT本质上是在海量语料的基础上，通过自监督学习的方法为单词学习一个好的特征表示。该模型的优点是可以根据具体的人物进行微调，或者直接使用预训练的模型作为特征提取器。

3. 总结

BERT 的结构基于 Transformer 的编码器部分，通过多层自注意力机制和前馈神经网络的堆叠，捕捉句子的双向上下文关系。在预训练过程中，BERT 通过自监督任务 (MLM 和 NSP) 学习到丰富的语义信

小红书账号：小黄搞AI

息，在预训练完成后，通过微调 (Fine-tuning) 可以应用到各种下游任务，如文本分类、序列标注、机器翻译等。

📌 BERT 之所以强大，主要原因在于：

- 它利用了大规模语料的双向语言模型进行预训练。
- 它不依赖序列化结构 (如 RNN)，能够更高效地并行处理数据。
- 通过微调，BERT 可以适配于不同的具体任务。

这种架构使得 BERT 成为 NLP 领域的一个里程碑式模型。

36. SwiGLU激活函数的原理

SwiGLU（Swish Gated Linear Units，Swish门控线性单元）是GLU（Gated Linear Units，门控线性单元）激活函数的一种变体，它将Swish激活函数和GLU结构结合起来，用于提升深度学习模型的性能，尤其是在像LLaMA这样的语言模型中得到广泛应用。

1. GLU（门控线性单元）概述

> 在 GLU（Gated Linear Unit）中，门控机制是关键组成部分，主要作用是**通过控制信号的流动来实现有选择性的激活**。门控机制类似于神经网络中的“开关”，它决定了某些输入信号是否能通过，进而影响网络的输出。门控机制通常通过 sigmoid 函数来实现，这样可以将输入值压缩到 0 和 1之间，类似于对信息进行筛选。

GLU激活函数公式

GLU是由**两层线性变换**和一个**门控机制**组成的激活函数，其计算公式为：

$$\mathrm{GLU}(x, W, V, b, c) = \sigma(xW + b) \odot (xV + c)$$

其中：

- x 为输入向量；
- W, V 是两个不同的线性变换矩阵；
- b, c 是对应的偏置项；
- $\sigma(x)$ 是Sigmoid激活函数；
- $\odot$ 表示逐元素相乘（即Hadamard积）。

GLU激活函数详细分析

为什么要用门控机制?

选择性信息传递

门控机制通过将**输入与 sigmoid 计算的结果逐元素相乘，可以有选择地抑制或放大某些输入**，这类似于让网络能够“决定”哪些信息是重

为什么要用 Hadamard 积?

逐元素控制：Hadamard 积**允许对每个元素进行独立控制**。在 GLU 中，网络会计算两组输出，一个是线性变换的输出 $XW_1 + b_1$，另一个是通过 sigmoid 激活的门控输出 $\sigma(XW_2 + b_2)$。

小红书账号：小黄搞AI

要的，哪些信息可以被过滤掉。这样做可以使网络在处理输入信息时更有**针对性，增强表达能力**。

避免信息过载

通过门控机制的筛选，可以**防止不必要的信息进入下一层，减轻网络的计算负担**，同时可能有助于减缓梯度消失问题。

提高模型表现

GLU 被认为是一种**比标准的非线性激活函数（如 ReLU）更灵活的机制**，因为它结合了线性变换与门控机制。这可以提高模型的收敛速度，**增强复杂任务中的表现。**

Hadamard 积将这两组结果逐元素相乘，**让门控输出控制每个元素是否被激活**。也就是说，每个输出维度都可以被独立地调节。

简化操作：Hadamard 积的计算量相对较低，**只需进行简单的逐元素相乘**，不像矩阵乘法那样复杂。这使得它成为计算高效且灵活的选择，适合深度学习网络中频繁的大规模计算。

选择性信息流动：通过 Hadamard 积，网络可以**针对不同的输入选择性地激活或抑制信息流动**。这是一种非常精细的控制机制，能让模型只关注重要的信息，同时忽略无关的输入。这种灵活性在优化和模型表现上都有优势。

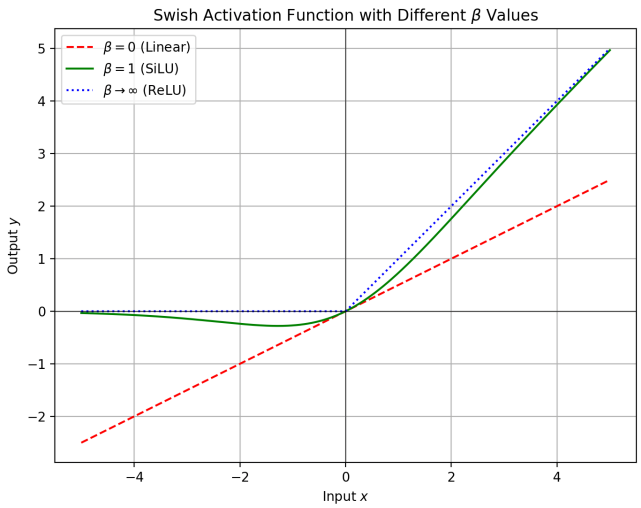
在GLU中，一个线性变换的结果通过Sigmoid函数进行门控，将其输出的值限制在0到1之间，作为“门控因子”去控制另一个线性变换的输出。**这种结构可以动态调整信息流，使得网络对不同输入进行有选择的响应。**

2. Swish激活函数

Swish是一种近年来提出的、性能优异的激活函数，它的表达式为：

$$\text{Swish}_\beta(x) = x \cdot \sigma(\beta x) \tag{2}$$

- $\sigma(x)$ 是Sigmoid函数[$\sigma(x) = \dfrac{1}{1+e^{-x}}$]
- β 是一个可学习的参数，用于控制Swish的形状。



Swish相较于传统的ReLU（Rectified Linear Unit，修正线性单元）激活函数，具有光滑的非单调特性。

具体来说：

当 $\beta = 0$ 时，Swish接近于线性函数；

当 β 无穷大时，它接近ReLU；

当 $\beta = 1$ 时，Swish与SiLU（Sigmoid Linear Unit）等价。Swish的优势在于其对负值的输入有一定的激活，而不像ReLU那样直接将负值置零，这使得它在很多任务中表现更好。

小红书账号：小黄搞AI

3. SwiGLU的计算原理

SwiGLU是通过将GLU中的Sigmoid函数替换为Swish函数得到的，其表达式如下：

$$\text{SwiGLU}(x, W, V) = \text{Swish}_\beta(xW) \odot (xV) \tag{3}$$

在这里，**Swish取代了原有的Sigmoid作为门控激活函数，这样就能在更平滑的激活函数下实现类似的门控机制**。与GLU相比，SwiGLU引入了更复杂的非线性变化，使得网络对不同输入的响应更加灵活。通过这种方式，SwiGLU能够在保持计算效率的同时提升模型的表现。

4. SwiGLU在前馈网络（FFN）中的应用

在如LLaMA等模型的前馈网络（FFN）中，SwiGLU被用作激活函数。FFN层的SwiGLU形式化公式为：

$$\text{FFN}_{\text{SwiGLU}}(x, W, V, W_2) = (\text{Swish}(xW) \odot (xV))W_2 \tag{4}$$

该公式表示：**首先对输入 x 进行两个线性变换，Swish函数作用于其中一个变换的结果，与另一个变换的结果逐元素相乘。最终，结果再通过另一个线性变换 W_2 转换回原始的隐藏层维度**。

此外，Swish激活函数可以用SiLU（Sigmoid-Weighted Linear Unit）代替：

$$\text{SiLU}(x) = x \cdot \sigma(x) \tag{5}$$

从实现的角度看，SiLU和Swish在 $\beta = 1$ 时是等价的，因此在实际应用中，SiLU常被用作SwiGLU的具体激活函数实现。

在PyTorch中，SwiGLU可以通过以下方式实现：

> 这段代码展示了如何使用PyTorch中的 `F.silu` 函数来实现SwiGLU激活函数，其中 `F.silu` 与\(\beta = 1\)时的Swish相同。这样的实现可以用于构建大型语言模型中的前馈网络层。

```python
import torch
from torch import nn
import torch.nn.functional as F

class SwiGLU(nn.Module):
    def __init__(self, w1, w2, w3) -> None:
        super().__init__()
        self.w1 = w1
        self.w2 = w2
        self.w3 = w3

    def forward(self, x):
        x1 = F.linear(x, self.w1.weight)
```

小红书账号：小黄搞AI

```
        x2 = F.linear(x, self.w2.weight)
        hidden = F.silu(x1) * x2
        return F.linear(hidden, self.w3.weight)
```

5. SwiGLU的优势

SwiGLU结合了Swish和GLU的优点，具有以下主要优势：

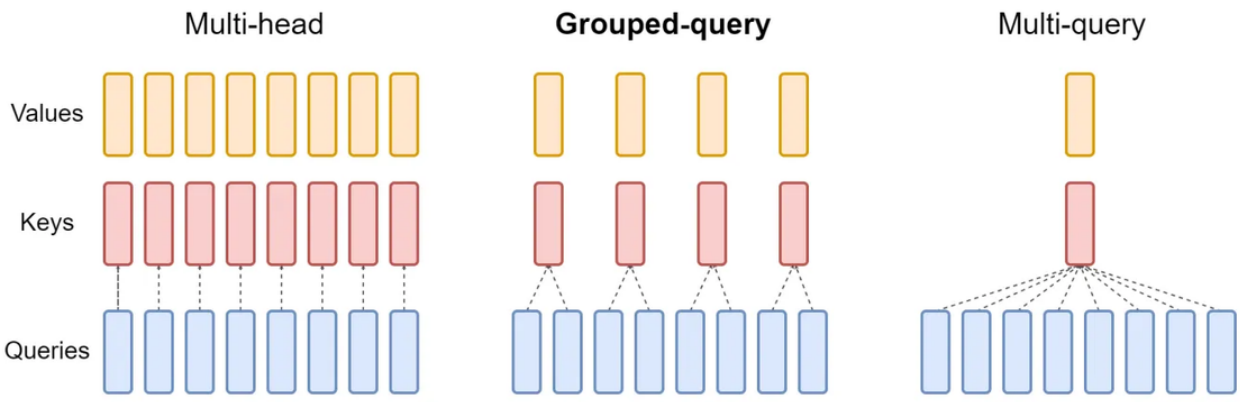
- **平滑激活**：Swish函数提供了平滑的梯度变化，避免了像ReLU那样的“死区”问题。这有助于提升模型的收敛速度和稳定性。
- **门控机制**：通过引入门控机制，SwiGLU可以选择性地控制信息流动，允许模型根据输入调整输出的强度，从而提高表达能力。
- **更好的性能**：与传统的ReLU或GLU相比，SwiGLU在实践中常常带来更好的性能，尤其是在大规模语言模型中。

6. 总结

SwiGLU激活函数作为LLaMA模型的改进之一，通过将GLU中的Sigmoid替换为Swish，使得模型在复杂任务中的表现得到了显著提升。其平滑的激活特性、灵活的非线性响应以及门控机制共同作用，提升了模型的表达能力和计算效率。

37. 为什么大模型用GQA(Group Query Attention)

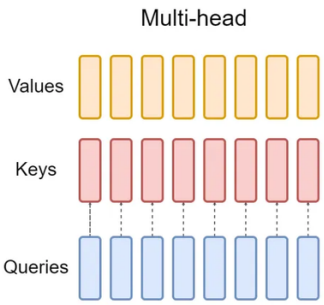
现在大模型广泛使用GQA（Grouped-Query Attention）是为了在**推理速度**和模型性能之间取得更好的平衡，特别是在内存带宽和推理速度成为瓶颈的情况下。要详细理解为什么大模型使用GQA，我们需要先讲解与其相关的两种注意力机制：MHA（Multi-Head Attention）和MQA（Multi-Query Attention）。



1. Multi-Head Attention (MHA)

MHA是在2017年Google团队提出的，是Transformer模型的核心组件。

小红书账号：小黄搞AI



在MHA中，**每个注意力头（head）有自己独立的Query、Key和Value投影矩阵**，允许模型并行地处理输入数据的多个方面。例如，一个注意力头可以专注于捕捉句子中的语法特性，另一个则可以专注于语义特性。

优势

捕捉多样化的特征：不同的注意力头可以**关注输入序列中的不同模式和依赖关系**。

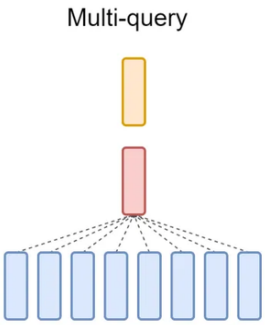
并行化计算：多个头部的自注意力机制是**并行处理的，这在训练期间尤其高效**。

劣势

内存和计算复杂度高

每个注意力头都需要维护自己的Key和Value矩阵，因此**内存需求和计算量随着模型规模增加而急剧上升**。这尤其在大型模型和推理过程中成为性能瓶颈。

2. Multi-Query Attention (MQA)



为了缓解MHA的内存带宽问题，Google在2019年提出了MQA。

在MQA中，虽然每个头依然保留独立的Query矩阵，但**Key和Value矩阵是共享的**。这样做的好处是显著减少了需要加载和存储的Key和Value数据，从而降低了内存带宽的需求，尤其是在自回归推理场景中。**这极大地提升了推理速度，尤其适合解码器场景。**

优势

推理速度大幅提升：共享Key和Value矩阵减少了多头机制的内存带宽需求，**推理速度更快**。

内存占用减少：Key和Value矩阵的数量**从MHA中的多个缩减为一个**，大大减少了内存占用。

劣势

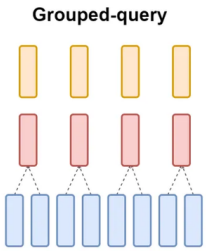
精度下降：由于所有头共享Key和Value矩阵，模型难以捕捉输入数据中复杂和多样化的特征，导致**模型性能相比MHA有所下降**。

3. Grouped-Query Attention (GQA)

📌 **GQA的核心目标是减少推理过程中KV cache的内存占用和带宽需求**，这使得其成为一种专注于**推理加速的技术**，而不是一种全新的训练方法。GQA通过将注意力头分组来共享Key和Value矩阵，从而减少了KV缓存的大小。

GQA是在2023年提出的，是MHA和MQA的折中方案。

小红书账号：小黄搞AI



与MQA相比，GQA并不只使用单个Key和Value矩阵，而是**将注意力头划分为若干组（G组），每个组共享一个Key和Value矩阵**。这种方法在保留了一定的多样性和细粒度特征捕捉能力的同时，大大减少了内存带宽和计算开销。

Pre-Norm

- 在Pre-Norm架构中，规范化操作（RMS Norm或Layer Norm）是在**自注意力（Self-Attention）或前馈神经网络（FFN）计算之前**进行的。也就是说，每一层的输入首先被归一化，然后再传递到注意力或前馈层。
- **Pre-Norm能够确保在深层网络中输入的幅度始终处于稳定范围，这对长链路依赖的模型尤其有益。**通过归一化操作的提前进行，模型能以更加稳定的输入进行学习，从而有助于解决深层模型中梯度消失的问题。

Post-Norm

- 在Post-Norm架构中，规范化操作是在**自注意力或FFN计算之后进行**的。模型首先经过未归一化的操作，最后将结果归一化以确保模型的输出平衡。
- Post-Norm可以在训练初期获得较好的收敛效果，尤其是在浅层模型中表现良好。**然而，在深层网络中，Post-Norm的缺点是可能会导致训练过程中的梯度不稳定**，特别是随着网络深度的增加，梯度可能在传播过程中变得越来越不稳定。

GQA的优势体现在以下几个方面：

平衡速度和精度：相比MQA，GQA通过分组的方式允许部分注意力头共享Key和Value矩阵，而不是所有头都共享，从而**减少了推理速度和内存需求上的负担，同时保留了更多特征捕捉能力**。质量接近MHA，而速度几乎接近MQA。

灵活性高：**GQA可以根据模型的需求灵活调整组数**。例如，GQA-1相当于MQA，GQA-H则相当于MHA。通过控制组数，模型可以在精度和推理速度之间进行权衡，适应不同规模和任务需求。

内存带宽优化：GQA可以**有效减少Key和Value矩阵的大小，减少键值缓存所需的内存带宽**。这在处理大型模型时尤为重要，因为模型的复杂度随着尺寸增加而急剧上升，GQA的优化使得内存带宽消耗与模型规模更好地成比例。

4. 总结

大模型采用GQA的核心原因在于其能够在**推理速度和模型性能之间提供优良的平衡**。相比于MHA，GQA大大减少了内存带宽和计算负担，适用于自回归推理等场景，而相比MQA，GQA通过分组策略保留了更多模型的表现能力。因此，GQA提供了一个在推理效率和模型精度之间折中的解决方案，非常适合当前大模型的实际需求。

38. 介绍一下Llama的结构

LLaMA模型的结构设计包含多个创新点，确保其在高效性和性能方面的提升。以下是LLaMA模型的几个主要特点：

小红书账号：小黄搞AI

1. RMS Pre-Norm（Root Mean Square Pre-Normalization）

LLaMA引入了RMSNorm作为Layer Norm的替代。传统的Layer Norm依赖于对输入进行均值和标准差的标准化，而RMSNorm简化了这一过程，只进行标准化。这意味着它跳过了减去均值的步骤，计算更加高效。这种规范化方式有助于模型的梯度平滑传播，特别是在深度网络中，RMSNorm有助于稳定训练过程，减少梯度消失或爆炸的风险。

RMS Pre-Norm与Layer Norm的详细对比及公式扩展

Layer Normalization (Layer Norm) 公式：

$$\mathrm{LN}(x) = \frac{x - \mu(x)}{\sigma(x)} \cdot \gamma + \beta$$

其中：

- x 表示输入向量；
- $\mu(x)$ 是输入的均值，即 $\mu(x) = \frac{1}{n}\sum_{i=1}^{n} x_i$ ；
- $\sigma(x)$ 是输入的标准差，即 $\sigma(x) = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \mu(x))^2}$ ；
- γ 和 β 是可学习的缩放和平移参数。

RMS Norm (Root Mean Square Norm) 公式：

$$\mathrm{RMS}(x) = \frac{x}{\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}} \cdot \gamma$$

其中：

- x 是输入向量；
- γ 是可学习的缩放参数；
- 与Layer Norm不同，RMS Norm仅基于输入的均方根（Root Mean Square, RMS）值进行标准化，省略了均值调整，计算更简单。

RMS Norm相比Layer Norm的优势分析

计算复杂度的降低

- **Layer Norm** 需要同时计算均值和标准差，**这意味着在每一层的计算过程中增加了两个额外的操作**，尤其是在高维度输入的情况下，这会带来额外的计算负担。
- **RMS Norm** 仅基于均方根（RMS）标准化，**它省略了均值计算步骤，减少了整体计算复杂度**。虽然这个简化看似细微，但在深度神经网络中，特别是对于大型语言模型，它能显著加速计算。

梯度传播的稳定性

- **Layer Norm** 通过标准差规范化确保了输入数据的分布在各层之间保持稳定，尤其是对于深层网络，这可以缓解梯度消失或梯度爆炸问题。然而，**Layer Norm在计算均值时会受到噪声和输入偏移的影响，这可能导致不稳定的梯度传播。**
- **RMS Norm** 通过仅使用RMS来处理标准化，能够提供更加平滑的梯度流动，**尤其是在较深层次网络中，减少了均值对梯度波动的影响**，从而提升了训练的稳定性和速度。

预归一化（Pre-Norm）和后归一化（Post-Norm）

Pre-Norm

Post-Norm

小红书账号：小黄搞AI

- 在Pre-Norm架构中，规范化操作（RMS Norm或Layer Norm）是在**自注意力（Self-Attention）或前馈神经网络（FFN）计算之前**进行的。也就是说，每一层的输入首先被归一化，然后再传递到注意力或前馈层。
- **Pre-Norm能够确保在深层网络中输入的幅度始终处于稳定范围，这对长链路依赖的模型尤其有益。**通过归一化操作的提前进行，模型能以更加稳定的输入进行学习，从而有助于解决深层模型中梯度消失的问题。

- 在Post-Norm架构中，规范化操作是在**自注意力或FFN计算之后进行**的。模型首先经过未归一化的操作，最后将结果归一化以确保模型的输出平衡。
- Post-Norm可以在训练初期获得较好的收敛效果，尤其是在浅层模型中表现良好。**然而，在深层网络中，Post-Norm的缺点是可能会导致训练过程中的梯度不稳定**，特别是随着网络深度的增加，梯度可能在传播过程中变得越来越不稳定。

Pre-Norm 训练过程和性能

- 在深度网络中，Pre-Norm通过提前标准化输入数据，确保**每一层都能接收到具有适当尺度的数据**。这对于减缓深层网络中的梯度消失问题特别重要。

Post-Norm 训练过程和性能

- Post-Norm在较浅的网络中具有较好的性能，因其允许模型在初期较快地收敛。但在**深层网络中，容易出现训练不稳定的问题。**

总结：RMS Pre-Norm的好处

- **计算效率提升**：由于RMS Norm省略了均值计算，它减少了每一层的计算量，这在深层网络中尤为重要。相比传统Layer Norm，RMS Norm能更高效地处理高维输入。
- **梯度稳定性增强**：RMS Pre-Norm能够减少梯度消失的情况，特别是在深度网络中，这种归一化方法通过平滑梯度流动，提升了训练效率。
- **适用于大规模模型**：对于像LLaMA这样的模型，RMS Pre-Norm为保持模型规模相对较小且性能强大提供了支持，使得模型在不增加复杂度的情况下，依然具备很好的泛化能力。

总结来说，RMS Pre-Norm的核心优势在于其计算简单和梯度稳定性，使其成为深度神经网络，尤其是大型语言模型（如LLaMA）中的一种有效规范化选择。

2. SwiGLU激活函数

> SwiGLU 是一种由 **Swish** 函数和线性变换组成的门控激活函数，通常用于提升模型的非线性表达能力。

SwiGLU与ReLU激活函数的详细对比与公式

SwiGLU激活函数公式：

$$\text{SwiGLU}(x, W, V, W_2) = (\text{Swish}(xW) \otimes xV)W_2$$

其中：

Swish 函数的定义为：$\text{Swish}(x) = x \cdot \sigma(x)$

ReLU(Rectified Linear Unit) 激活函数公式：

$$\text{ReLU}(x) = \max(0, x)$$

小红书账号：小黄搞AI

而 $\sigma(x)$ 是标准的 sigmoid 函数，定义为：

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

在 SwiGLU 中，输入 x 首先经过线性变换（与权重矩阵 W 相乘），然后通过 Swish 函数进行非线性激活，之后再与另一个线性变换 xV 进行逐元素乘法，最后再经过权重 W_2 线性变换输出。

其中：

其中，输入 x 大于 0 时输出为 x，否则输出为 0。ReLU的核心特性是简单且计算效率高，能有效缓解梯度消失问题。

ReLU与SwiGLU的详细对比

非线性表达能力

- **ReLU** 是一种简单的**线性分段函数**，虽然它能够引入非线性，但在负数区域它的梯度为 0，**这意味着当输入为负值时，梯度消失，导致神经元在训练过程中可能会被“抛弃”**。
- **SwiGLU** 则通过 **Swish 的平滑特性和门控机制引入更强的非线性**。Swish 函数在负值区域也能保留小幅的梯度，不像 ReLU 那样将负值直接归零，因此它能够保留更多信息，特别是负数区域的信号。这提高了模型对复杂特征的捕捉能力。

信息传递与梯度流动

- **ReLU** 在负数区域完全阻塞了梯度流动，当输入为负时，其导数为 0。这虽然在一定程度上解决了梯度爆炸问题，但也**可能导致部分神经元“死掉”，即永远不再更新**。
- **SwiGLU** 通过 Swish 和线性变换相结合，**不仅在正值区域保持较大的梯度，在负值区域也提供了较小但连续的梯度**，确保信息在网络中传递得更为流畅。这减少了“死神经元”的问题，并且梯度更新更稳定。

计算效率

- **ReLU** 的**计算效率极高，因为它只是一个简单的阈值操作**，对于输入大于 0 的保持原值，小于 0 的直接归零，因此在大规模模型中非常常用。
- **SwiGLU** 的**计算复杂度更高，因为它包含了多重线性变换、非线性 Swish 函数以及逐元素的门控运算**。然而，尽管计算复杂性增加，SwiGLU 提供了显著的性能提升，特别是在处理具有复杂特征的任务时，如自然语言处理任务中的上下文建模。

性能与泛化能力

- **ReLU** 尽管简单高效，但在处理复杂问题时，其线性分段的激活机制可能不足以捕捉到丰富的特征，尤其是**在需要高非线性表达能力的任务中表现出局限性**。
- **SwiGLU** 结合了 Swish 的平滑激活特性与线性变换的门控机制，**增强了模型对复杂特征的学习能力**。这种机制使得 SwiGLU 在处理自然语言任务时表现出更强的泛化能力，能够更好地建模输入之间的复杂关系。

SwiGLU的好处

- **更强的非线性：** SwiGLU 通过 Swish 激活函数引入了更平滑的非线性，不仅保留了正值区域的信息，也允许负值区域的梯度更新，使得模型能够学习到更多的特征。
- **信息的持续传递：** 由于 Swish 函数在负数区域并未完全抑制梯度，SwiGLU 能确保更多的神经元参与到学习过程中，减少了“死神经元”问题，增强了模型的梯度传播能力。

小红书账号：小黄搞AI

- **性能提升**：在自然语言处理等复杂任务中，SwiGLU 提供了显著的性能提升，因为它能够捕捉到更丰富的上下文信息。实验表明，使用 SwiGLU 的模型在诸多任务上的表现优于采用 ReLU 的模型。
- **平滑梯度更新**：与 ReLU 不同，SwiGLU 通过 Swish 的平滑曲线，确保了梯度在正负区域都能稳定传播。这种平滑更新使得模型训练更加稳定，尤其是在大规模深度网络中。

相比于传统的 ReLU 激活函数，SwiGLU 通过结合 Swish 的非线性激活和线性变换的门控机制，增强了模型的表达能力和梯度传播稳定性，尤其在处理复杂特征和需要更高非线性表达能力的任务中表现优越。然而，其计算复杂度也相对更高，需要在应用中进行性能与效率的权衡。

3. RoPE旋转位置编码（Rotary Position Embedding）

RoPE的详细步骤与计算

RoPE的核心思想是将输入向量的每一维都与其位置进行**频率旋转**，这样做有以下几步：

Query和Key的旋转

- 对于给定的位置 m ，RoPE会根据 m 和频率参数 θ 对 Query 向量进行旋转，即乘以 $e^{im\theta}$ ；Key 向量同样进行旋转；
- 这种操作本质上是在每个维度上**引入相位偏移**，赋予了输入向量的每个token特定的位置信息。

相对位置信息的捕捉

- RoPE通过将旋转操作应用于Query和Key，能够有效捕捉它们之间的**相对位置差异**。当计算注意力权重时，RoPE允许模型对相邻token之间的相对距离更加敏感，使得模型更适合长序列的建模。

RoPE旋转位置编码与绝对位置编码的详细对比与公式扩展

RoPE（Rotary Position Embedding）公式：

RoPE是一种通过**旋转操作**编码位置信息的机制，改进了传统的绝对位置编码方式，尤其适用于**捕捉相对位置信息**，RoPE通过将位置信息编码到Q和K向量中实现：

$$\text{RoPE}(q_m, k_n) = (q_m e^{im\theta}, k_n e^{in\theta})$$

其中：

- q_m 和 k_n 分别是位于位置 m 和 n 的 Query 和 Key 向量；
- θ 是频率参数，通常为可学习或固定的常数，决定了位置信息的分布；
- $e^{im\theta}$ 是复数的旋转操作，用来编码位置信息。此处 i 是虚数单位，表示相位旋转的角度。

绝对位置编码(Absolute Position Embedding)公式：

绝对位置编码是一种传统方法，用于直接将位置信息添加到每个token的嵌入向量中。最常见的形式为**正弦-余弦位置编码**:

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{\text{model}}}}}\right)$$

其中：

- pos 是当前token的位置；
- i 是嵌入向量的维度索引；
- d_{model} 是模型的嵌入维度大小。

小红书账号：小黄搞AI

通过这种旋转的方式，RoPE允许模型感知输入序列中各个token之间的**相对位置关系**，而不仅仅是其**绝对位置**。

绝对位置编码的目的是为每个token提供一个独特的绝对位置标识，使得模型可以区分句子中不同位置的token。然而，它只关注token的**绝对位置**，并没有直接捕捉token之间的**相对位置关系**。

RoPE 与绝对位置编码的对比

特性	RoPE旋转位置编码	绝对位置编码
位置信息捕捉	捕捉token之间的**相对位置**关系	仅捕捉每个token的**绝对位置**
序列长度适应性	在处理长序列时表现优越，尤其是长依赖关系	对长序列不够有效，位置信息可能被稀释
数学操作	通过**旋转操作**，引入位置信息的相位偏移	通过直接**嵌入**正弦-余弦函数
注意力机制的效果	能够自然处理注意力机制中的相对位置信息	无法直接捕捉相对位置，适用于较短序列
应用场景	适用于长序列任务，如长文本生成、长依赖建模	更适合短序列或无复杂相对依赖的场景
计算复杂性	更复杂，涉及复数的旋转计算	计算较简单，直接添加到嵌入向量中

RoPE的优势

- **相对位置信息的捕捉**：RoPE的最大优势在于它可以在注意力计算中自然地融入**相对位置信息**，而不像绝对位置编码那样只依赖token的固定位置。因此，它在处理长序列时能够**更好地建模token之间的关系**，尤其是长距离的依赖。
- **处理长序列的能力**：由于RoPE直接编码的是相对位置关系，这使得**它更适合处理长文本、长时间序列等任务**。而传统的绝对位置编码在长序列中，可能由于固定位置的原因逐渐失去位置区分能力。
- **与注意力机制的适配**：RoPE通过旋转操作，**将位置信息直接作用于Query和Key向量，使得它可以更加自然地融入注意力机制的计算过程中**。这种机制保证了模型在长序列中依旧能够捕捉到上下文中有意义的位置信息。

综上，LLaMA模型通过引入RMS Pre-Norm规范化技术、SwiGLU激活函数和RoPE位置编码等创新点，有效地优化了Transformer结构，提升了模型在自然语言处理任务上的性能和效率。这些精心设计的改进使得LLaMA成为一个在多种应用场景下都具有竞争力的语言模型。

小红书账号：小黄搞AI

39. Llama模型在训练过程中如何处理梯度消失和梯度爆炸的问题

🦄 在深度神经网络的训练过程中，梯度消失（vanishing gradients）和梯度爆炸（exploding gradients）是常见的挑战。它们会导致模型训练困难，无法有效学习到数据的特征。下面将从根源上逐个概念进行解释，并探讨Llama模型是如何应对这些问题的。

39.1 梯度消失和梯度爆炸的根源

梯度消失

- **原理：** 在反向传播过程中，梯度需要通过链式法则层层传递。如果激活函数的导数小于1（如Sigmoid或Tanh），多个小于1的数相乘，导致梯度指数级衰减，最终接近于零。
- **影响：** **梯度过小，权重更新缓慢，网络无法有效学习深层次的特征。**

梯度爆炸

- **原理：** 相反，如果激活函数的导数大于1，多个大于1的数相乘，梯度会指数级增长，导致数值不稳定，甚至出现NaN。
- **影响：** **梯度过大，权重更新剧烈，模型容易陷入不稳定状态，无法收敛。**

39.2 Llama模型的架构特点

Llama模型是基于Transformer架构的大型语言模型，具有以下特点：

- **多头自注意力机制（Multi-Head Self-Attention）：** 能够捕捉序列中不同位置的依赖关系。
- **残差连接（Residual Connections）：** 直接将输入添加到输出，缓解梯度消失问题。
- **层归一化（Layer Normalization）：** 在每一层对神经元进行归一化，稳定网络训练。

39.3 Llama模型如何应对梯度消失和梯度爆炸

1. 使用残差连接

- **原理：** 残差连接将每一层的输入直接添加到输出，形成 $\text{输出} = \text{输入} + \text{函数}(\text{输入})$ 的形式。
- **效果：** 提供了梯度传递的“捷径”，避免梯度在深层网络中过度衰减。
- **图示：**

```
输入 -----> (层的运算) -----> 加法 -----> 输出
      \_________________________/

```

2. 层归一化

- **原理：** 对层的输出进行归一化，使其均值为0，方差为1。

小红书账号：小黄搞AI

- **效果：** 稳定了激活值的分布，防止梯度爆炸，提高训练的稳定性和速度。

3. 优化的激活函数

- **使用ReLU或其变体：** ReLU函数的导数为0或1，避免了Sigmoid/Tanh函数导数过小的问题。
- **效果：** 防止梯度消失，加速收敛。

4. 梯度裁剪（Gradient Clipping）

- **原理：** 在反向传播时，对梯度进行裁剪，限制其最大范数。
- **效果：** 防止梯度爆炸，保持数值稳定性。

> **梯度裁剪示意图**
>
> ```
> 梯度计算 ---> [梯度裁剪模块] ---> 梯度更新
> ```

5. 权重初始化

- **原理：** 使用合适的方法（如Xavier初始化）设置初始权重。
- **效果：** 防止初始梯度过大或过小，提供良好的训练开端。

6. 优化器的选择：

- **使用自适应学习率优化器（如Adam）：** 根据梯度的一阶和二阶矩动态调整学习率。
- **效果：** 适应不同参数的更新需求，稳定训练过程。

39.4 总结

Llama模型通过结合Transformer架构的优势，有效地应对了梯度消失和梯度爆炸的问题：

- **残差连接**为梯度提供了直接通路，防止梯度消失。
- **层归一化**和**梯度裁剪**技术稳定了训练过程，防止梯度爆炸。
- **优化的激活函数**和**权重初始化**确保了梯度的合理范围。

这些方法的综合应用，使得Llama模型能够在深层网络中有效学习，捕捉复杂的语言模式和特征。

40. DeepSpeed中ZeRO系列的原理

> 《DeepSpeed之ZeRO系列：将显存优化进行到底》

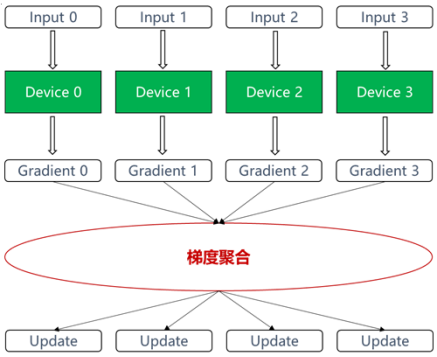
DeepSpeed的ZeRO系列是专为优化大规模深度学习模型训练而设计的技术，主要包括ZeRO-1、ZeRO-2和ZeRO-3，每个版本在内存效率和训练速度上都有显著提升。

1. 常用的并行策略

`数据并行（data parallelism，DP）`

小红书账号：小黄搞AI

参考图



数据并行

假设有 N 张卡，每张卡都保存一个模型，每一次迭代（iteration/step）都将batch数据分割成 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，然后调用AllReduce计算梯度均值**，每张卡再独立进行参数更新。

Step1 输入数据切分

输入数据切分实现上比较简单，一般有两种常用的实现方式：

方式一

在每个训练Epoch开始前，**将整个训练数据集根据并行进程数划分**，每个进程只读取自身切分的数据。

方式二

数据的读取仅由具体某个进程负责(假设为rank0)。**rank0在数据读取后同样根据并行进程数将数据切分成多块**，再将不同数据块发送到对应进程上。

Step2 模型参数同步

数据并行实现的关键问题在于如何保证训练过程中每个进程上模型的参数相同。

因为训练过程的每一个step 都会更新模型参数，每个进程处理不同的数据会得到不同的Loss。由Loss计算反向梯度并更新模型参数后，如何保证进程间模型参数正确同步，是数据并行需要解决的最主要问题。根据下面中的梯度更新公式，只要保证以下两点就能解决这个问题：

数学公式：

$$W_{t+1} = W_t - lr \times \Delta W$$

> 这个公式表示在优化过程中，模型的参数 W 在时间步 $t+1$ 的值是由当前时间步 t 的参数 W_t 减去学习率 lr 乘以梯度 ΔW 来更新的。这是一种常见的梯度下降法的参数更新规则。

1. **每个进程上模型初始化参数 W_0 相同。**
2. **每个进程上每次更新时的梯度 ΔW 相同。**

保证每个进程模型参数初始相同有两种常用的实现方法：

方法一

所有进程在参数初始时使用**相同的随机种子并以相同的顺序初始化所有参数**。

方法二

通过个具体进程初始化全部模型参数，之后**由该进程向其他所有进程广播模型参数**。

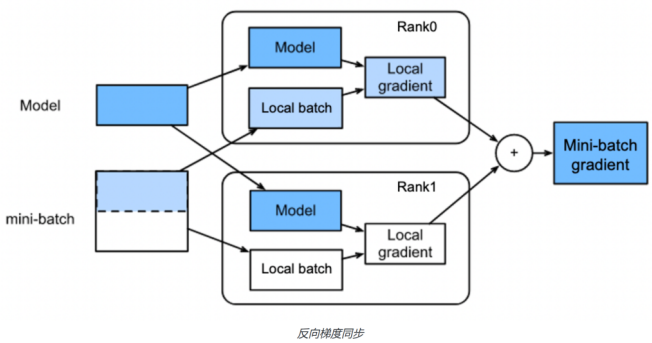
基于上述任意一种方法使每个进程得到一份相同的模型初始化参数后，梯度同步的数据并行训练就可以进一步拆解为如下三个部分：

Step2.1 前向计算

小红书账号：小黄搞AI

每个进程根据自身得到的输入数据独立前向计算，因为输入数据不同每个进程会得到不同的Loss。

Step2.2 反向计算



反向梯度同步

每个进程根据自身的前向计算独立进行反向计算，因为每个进程上的Loss不同，每个进程上在反向中会计算出不同的梯度。这时一个关键的操作是要在**后续的更新步骤之前，对所有进程上的梯度进行同步，保证后续更新步骤中每个进程使用相同的全局梯度更新模型参数。**

这一个梯度同步过程是用一个Allreduce sum同步通信操作实现的，对梯度使用Allreduce sum操作后每个进程上得到的梯度是相同的，这时候的梯度值等于所有进程上梯度对应位置相加的和，然后每个进程用Allreduce后的梯度和除以数据并行中的进程数，这样得到的梯度是同步之前所有进程上梯度的平均值。

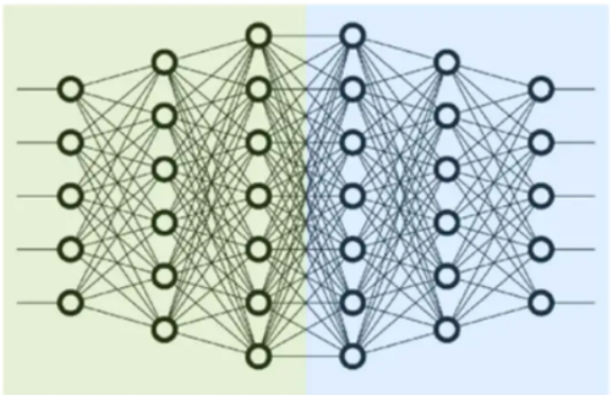
Step2.3 反向计算

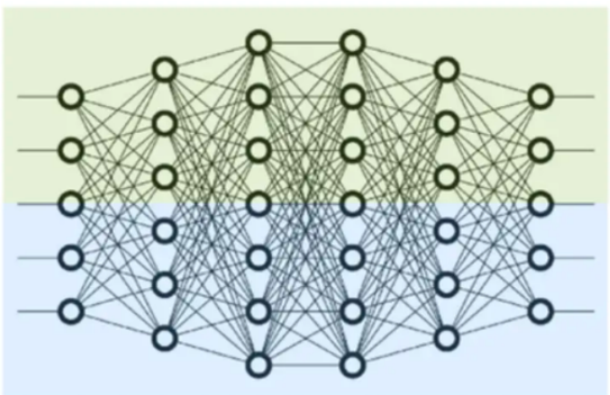
每个进程经过上述步骤后**得到相同全局梯度，然后各自独立地完成参数更新**。因为更新前模型各进程间的参数是相同的，更新中所使用的梯度也是相同的，所以更新后各进程上的参数也是相同的。

上述是主流框架中数据并行的实现过程。**和单卡训练相比，最主要的区别在于反向计算中的梯度需要在所有进程中进行同步，保证每个进程上最终得到的是所有进程上梯度的平均值。**

模型并行（model parallelism/）

流水线并行和张量并行都可以看作是模型并行的一种，只是对模型切分的维度不同，**流水线并行可以看作是层间并行，将模型不同的层放到不同的GPU上，张量并行看作是层内并行，是对层内具体的矩阵运算进行拆分**。


层间并行

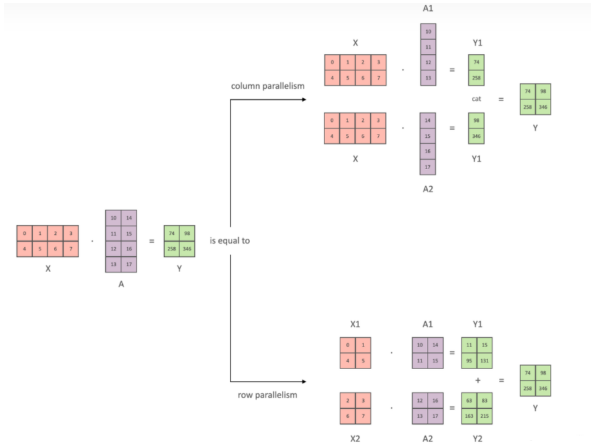

层内并行

A. 张量并行（tensor parallelism, MP/TP）

x为输入，**A**为权重参数，**y**为输出。

按列拆分

小红书账号：小黄搞AI

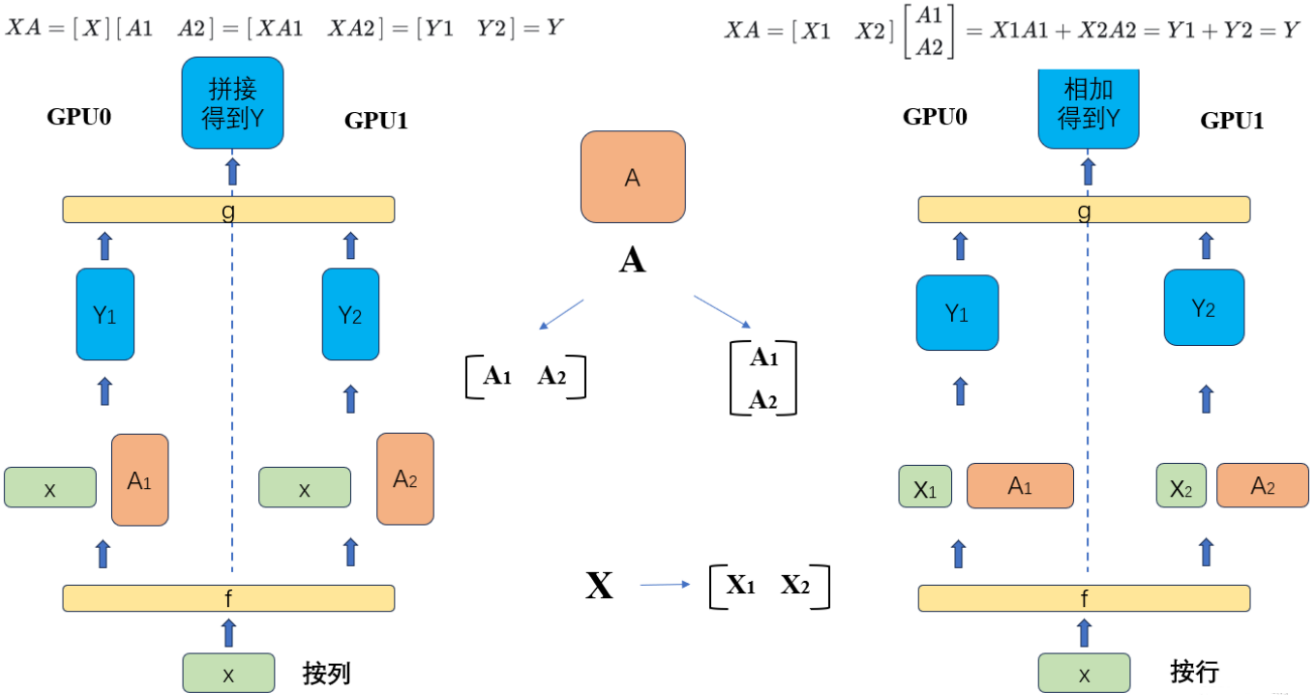


按列拆分是保持x不变，将A按列拆分，分别计算后再**拼接**在一起。

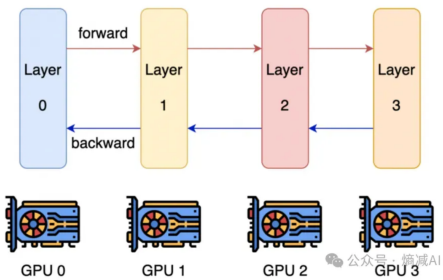
按行拆分

按行拆分是将x按列拆分，A按行拆分，分别计算后，再**相加**得到最终的输出。

> 在两张GPU上的具体的并行操作思想就是将进行运算的矩阵拆分到不同的GPU上，然后将每个GPU上的运算结果进行汇总（GPU之间通信）。



B. 流水并行（pipeline parallelism, PP）



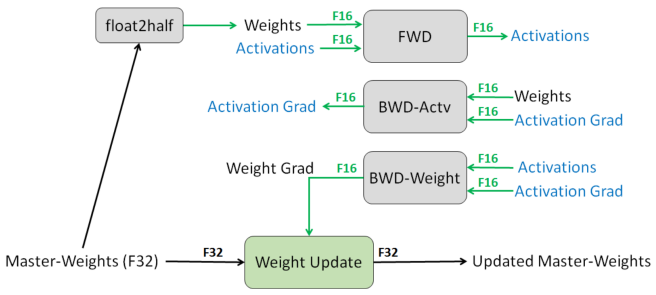
流水线并行PP（Pipeline Parallelism）

把一个参数量较大的模型按照不同的层进行划分，将多个模型的层尽可能均匀的分布在不同的GPU显存上，来起到装载更大模型的能力。

2. 显存消耗

混合精度训练（mixed precision training）和Adam优化器基本上已经是训练语言模型的标配。

小红书账号：小黄搞AI



Mixed precision training iteration for a layer.

Adam在SGD基础上，为每个参数梯度增加了**一阶动量（momentum）和二阶动量（variance）**。

混合精度训练是**同时存在fp16和fp32两种格式**的数值，其中模型参数、模型梯度都是fp16，此外还有fp32的模型参数，如果优化器是Adam，则还有fp32的momentum和variance。

ZeRO将模型训练阶段，每张卡中显存内容分为两类：

1. 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 ：

模型参数（fp16格式）

2Φ

模型参数的总量为 Φ ，每个参数用16位（2字节）来存储。因此模型参数共占用2 Φ 字节。

模型梯度（fp16格式）

2Φ

梯度和模型参数的数量相同，都是 Φ ，每个梯度值也用16位（2字节）存储。因此，梯度共占用2 Φ 字节。

Adam状态

12Φ

Adam优化器是一种常用的自适应梯度优化算法，使用动量（momentum）和方差（variance）来动态调整学习率。在训练中，为了保持数值稳定性，Adam优化器的状态通常以fp32（32位浮点数，4字节）存储。

- **模型参数备份（fp32格式）**
  - Adam优化器会维护模型参数的fp32备份（通常用于更精确的权重更新）。这是 Φ 个fp32参数，每个占4字节。
  - **存储开销：** 4Φ 字节。
- **Momentum（动量，fp32格式）**
  - Adam的动量用于追踪模型参数的变化趋势。Adam为每个参数维护一个动量值，这些动量值以fp32格式存储。
  - **存储开销：** 4Φ 字节。
- **Variance（方差，fp32格式）**
  - Adam的方差用于调整学习率。每个参数有一个对应的方差值，存储为fp32格式。
  - **存储开销：** 4Φ 字节。

小红书账号：小黄搞AI

假设模型参数量 Φ ，则共需要 $2\Phi + 2\Phi + (4\Phi + 4\Phi + 4\Phi) = 4\Phi + 12\Phi = 16\Phi$ 字节存储，可以看到，Adam状态占比 75% 。

2. 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）。

3. DeepSpeed针对模型状态的存储优化（去除冗余）

ZeRO（Zero Redundancy Optimizer）通过 **分区（partition）** 技术优化模型状态的存储，即每张设备（卡）只存储 $\frac{1}{N}$ 的模型状态，从而系统内仅维护一份完整的模型状态。以下描述了分区操作在不同模型状态上的应用：

1. 优化器状态（Optimizer States）的分区

首先进行分区的是模型状态中的 **Adam状态**，下图中的 P_{os} 表示分区的优化器状态，其中 `os` 指的是 **optimizer states**。此时，**模型参数（parameters）** 和 **梯度（gradients）** 仍然在每张设备上保持一份拷贝。

- **每张卡的模型状态所需显存** 为： $4\Phi + \frac{12\Phi}{N}$ 字节
- 当 N 较大时，显存占用趋向于： 4Φ 字节，也就是原来 16Φ 字节 的 $\frac{1}{4}$ 。

2. 梯度的分区

如果继续对 **模型梯度** 进行分区，即图中的 $P_{\text{os+g}}$ （其中 g 代表梯度），**模型参数**仍然在每张卡保持一份拷贝。

- **每张卡的模型状态所需显存** 为： $2\Phi + \frac{2\Phi + 12\Phi}{N}$ 字节
- 当 Φ 较大时，显存占用趋向于： 2Φ 字节，也就是原来 16Φ 字节 的 $\frac{1}{8}$ 。

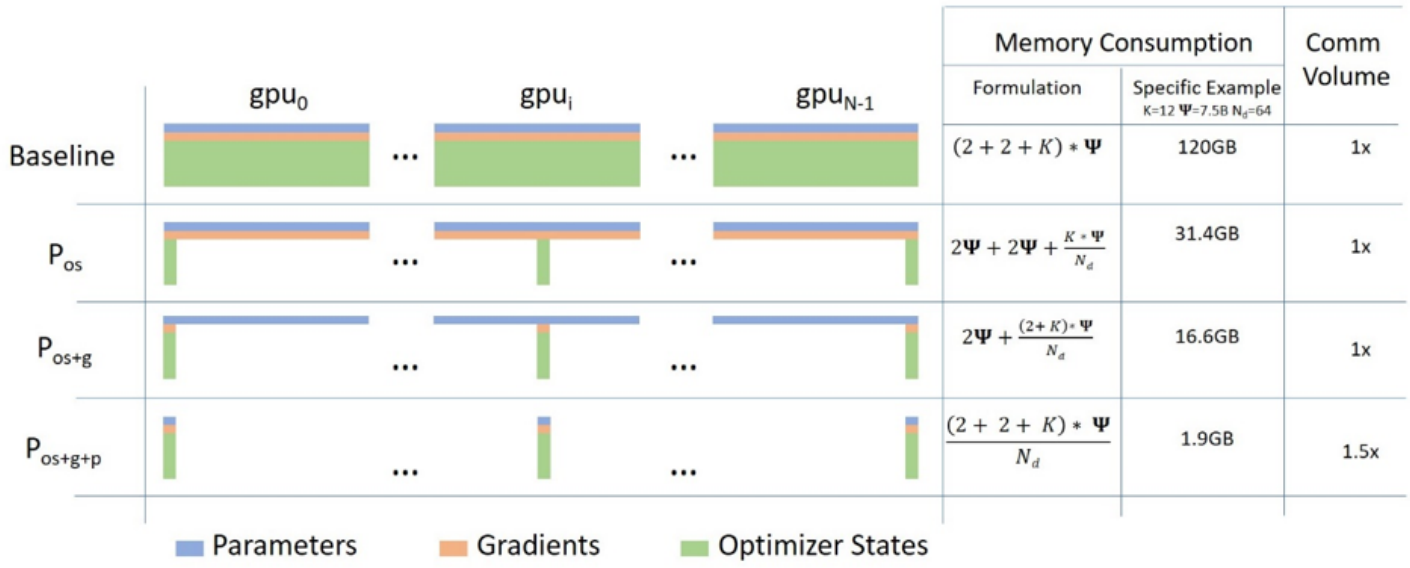
3. 模型参数的分区

如果再进一步对 **模型参数** 进行分区，即图中的 $P_{\text{os+g+p}}$ （其中 P 代表参数），则：

- **每张卡的模型状态所需显存** 为： $\frac{16\Phi}{N}$ 字节，当 N 较大时，显存占用趋向于： 0 字节

通过这种分区方法，ZeRO能够显著减少每张卡上的模型状态存储需求，极大地提升了训练大规模模型的效率和可扩展性。

小红书账号：小黄搞AI



	Memory Consumption		Comm Volume
	Formulation	Specific Example K=12 Ψ=7.5B N_d=64	
Baseline	$(2+2+K)*\Psi$	120GB	1x
P_{os}	$2\Psi+2\Psi+\frac{K*\Psi}{N_d}$	31.4GB	1x
P_{os+g}	$2\Psi+\frac{(2+K)*\Psi}{N_d}$	16.6GB	1x
P_{os+g+p}	$\frac{(2+2+K)*\Psi}{N_d}$	1.9GB	1.5x

模型状态分区

ZeRO-1

它通过将优化器状态分散到多个设备上，减少了单个设备的内存占用。这意味着每个设备只需要存储它所负责的一部分参数和状态，从而支持更大的模型。

ZeRO-2

在ZeRO-1的基础上，进一步分散了梯度。每个设备不仅存储其负责的优化器状态，还可以共享其他设备的梯度。这种方式显著提高了内存利用率，使得训练更大模型成为可能。

ZeRO-3

结合了参数分片、梯度分片和优化器状态分片，几乎实现了“无状态”训练。通过将所有训练所需的数据分散到不同的设备，ZeRO-3使得即使在有限的GPU内存中也能训练数百亿参数的模型。

4. 通信数据量分析

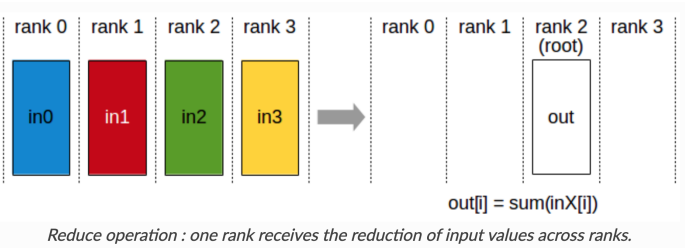
常用的集合通信（collective communication）函数Collective Operations。

Reduce

操作描述：将多个处理器上的数据聚合成一个结果。例如，对每个处理器的数据执行求和操作，并将结果放到某个指定处理器上。

用法：常用于在多个GPU或节点上训练时，汇总各个设备的梯度或参数。

数据流动：**从多个设备到一个设备。**



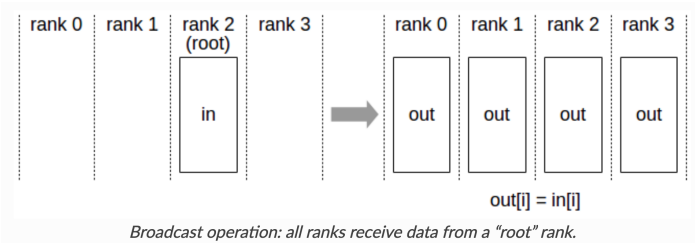
Reduce operation : one rank receives the reduction of input values across ranks.

Broadcast

操作描述：将某个处理器上的数据发送到所有处理器上，确保每个处理器都有相同的数据副本。

用法：通常用于分布式训练的初始化阶段，将模型参数从一个主节点广播到所有参与计算的设备上。

数据流动：**从一个设备到所有设备。**



Broadcast operation: all ranks receive data from a “root” rank.

小红书账号：小黄搞AI

ALLGather

操作描述：每个处理器都参与数据的汇总（如求和或求平均），并将汇总后的结果分发给所有处理器。即所有设备的输入都被聚合为一个结果，并在每个设备上都有一份相同的结果。

用法：这是**数据并行训练**的核心通信操作，通常用于在每一轮迭代之后计算全局梯度的均值，所有设备都获取相同的更新后的梯度。

数据流动：**每个设备互相通信，得到相同的结果。**



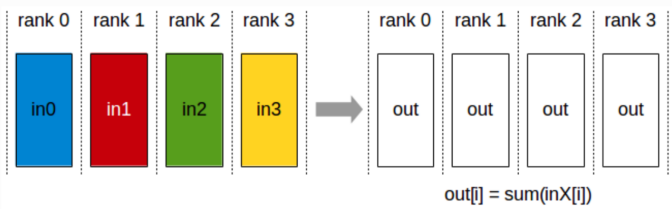
AllGather operation: each rank receives the aggregation of data from all ranks in the order of the ranks.

AllReduce

操作描述：每个处理器都参与数据的汇总（如求和或求平均），并将汇总后的结果分发给所有处理器。即所有设备的输入都被聚合为一个结果，并在每个设备上都有一份相同的结果。

用法：这是**数据并行训练**的核心通信操作，通常用于在每一轮迭代之后计算全局梯度的均值，所有设备都获取相同的更新后的梯度。

数据流动：**每个设备互相通信，得到相同的结果。**



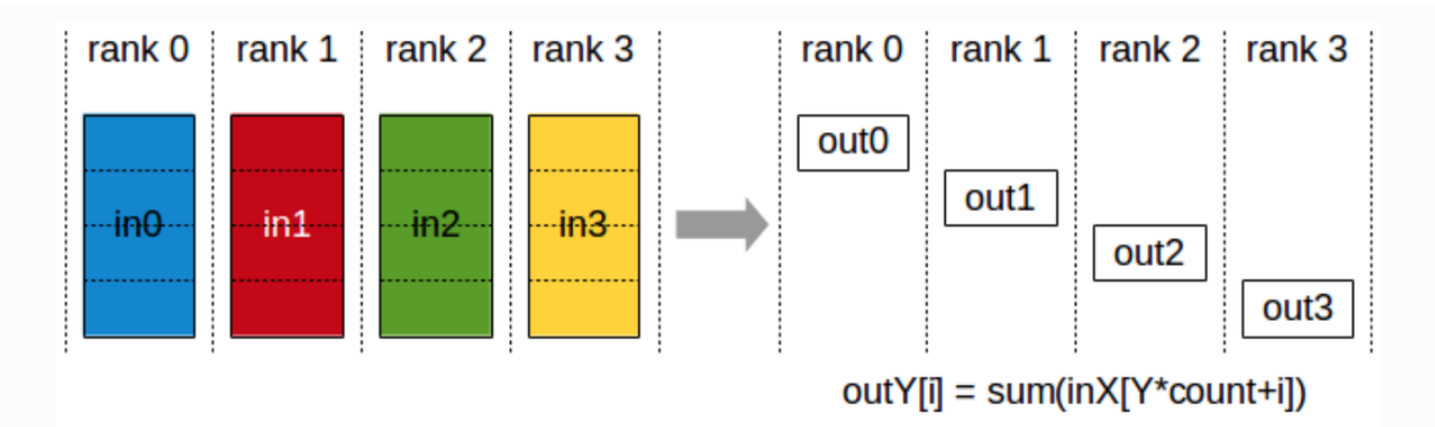
All-Reduce operation: each rank receives the reduction of input values across ranks.

ReduceScatter

操作描述：所有处理器将各自的数据发送给其他所有处理器，并在每个处理器上收集所有其他处理器的数据。

用法：常用于分布式训练的参数收集或数据分片操作。它允许所有设备共享它们独有的数据或参数，以便同步或更新。

数据流动：**每个设备向所有设备发送并接收数据。**



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

📌 **结论**

- P_{os} 和 P_{os+g} 的通信量与传统数据并行相同。

小红书账号：小黄搞AI

- P_{os+g+p} 会增加通信量。

在传统的数据并行中，每一步（iteration）计算梯度后，需要进行一次 **AllReduce** 操作来计算梯度的均值。常用的 **Ring AllReduce** 分为 **ReduceScatter** 和 **AllGather** 两步完成。每张卡的通信数据量（发送+接收）近似为 2Φ 字节。

😈 P_{os+g} 的通信量

在 P_{os+g} 方案中，每张卡只存储 $\frac{1}{N}$ 的优化器状态和梯度。为了计算每张卡上的 $\frac{1}{N}$ 梯度的均值，通信量的具体分析如下：

- 对于 **gpu0** 来说，它存储的是 $\frac{1}{N}$ 的梯度。为了计算该部分的均值，需要进行一次 **Reduce** 操作，通信数据量是 $\frac{1}{N}\Phi \times N = \Phi$ 。
- 其余显卡不需要保存这部分梯度值。为了优化，系统使用了 **bucket策略**，保证 $\frac{1}{N}$ 的梯度只发送一次。

> 📌 **注意：**
>
> 如果模型的最后两层梯度保存在 **gpu0** 上，为了节省显存，其他卡会将这两层的梯度删除。为了计算倒数第三层的梯度，仍然使用 **bucket策略**。这样，其他卡可以在发送梯度的同时计算倒数第三层的梯度。当这两者都完成后，后两层的梯度可以删除。

- 当 **gpu0** 计算完梯度均值后，可以更新它的局部优化器状态。反向传播结束后，再进行一次 **Gather** 操作来更新模型参数，通信数据量为 $\frac{1}{N}\Phi \times N = \Phi$ 。

从全局来看，这个过程相当于通过 **ReduceScatter** 和 **AllGather** 两步完成，通信量和数据并行一致。

😈 P_{os+g+p} 的通信量

在 P_{os+g+p} 方案中，每张卡只存储了 $\frac{1}{N}$ 的模型参数。因此，不论是在前向计算还是反向传播，都涉及到一次 **Broadcast** 操作，这增加了通信量。

参考文章（强烈推荐阅读）



https://www.paddlepaddle.org.cn/support/news?action=detail&id=2762

数据并行:提升训练吞吐的高效方法 |深度学习分布式训练专题

飞桨PaddlePaddle-源于产业实践的开源深度学习平台

小红书账号：小黄搞AI

https://basicv8vc.github.io/posts/zero/

DeepSpeed之ZeRO系列：将显存优化进行到底

前言 目前训练超大规模语言模型主要有两条技术路线：TPU + XLA + TensorFlow/JAX 和 GPU + PyTorch + Megatron-LM + DeepSpeed。前者由Google主导，由于TPU…

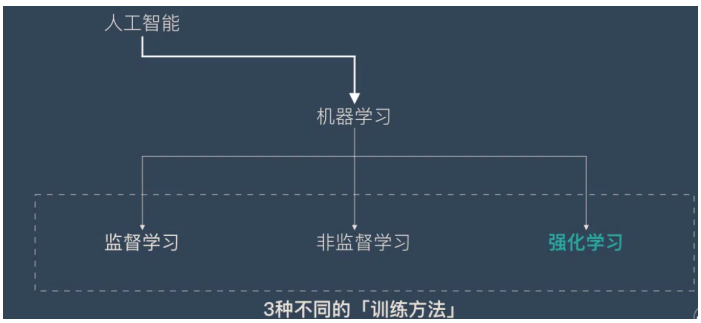
https://medium.com/@zhouboyang1983/llm%E5%88%86%E5%B8%83%E5%BC%8F%E8%AE%AD%E7%BB%83%E7…

medium.com

41. （W）介绍inference的几种方式

42. （W）有了解推理加速吗的原理吗？

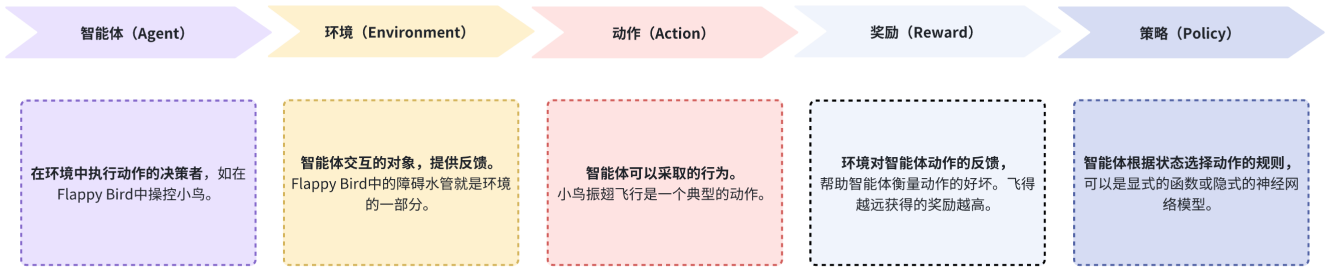
43. 解释下强化学习



强化学习（Reinforcement Learning, RL）

一类机器学习方法，其核心思想是通过与环境的交互、试错探索，逐步优化策略，以最大化长期累计的奖励。不同于监督学习和无监督学习，强化学习不依赖大量的标注数据，而是**通过智能体（agent）与环境的互动来自行学习**。

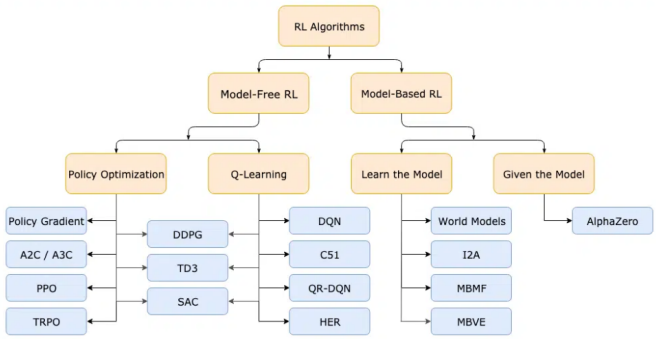
强化学习的主要组成部分包括：



在强化学习中，智能体在每个时间步（timestep）根据当前状态（state）采取一个动作（action），随后环境会反馈一个新的状态和相应的奖励（reward）。**智能体的目标是通过优化策略，使得其在长期内获得的奖励最大化。**

强化学习中的算法大致分为两类：**有模型学习（Model-Based）和免模型学习（Model-Free）**。前者在交互过程中建模环境，便于规划未来的决策，但在真实环境中表现不佳时其效果

小红书账号：小黄搞AI



也会受限。后者不建模环境，而是通过直接优化策略或价值函数来指导决策，如常见的Q-Learning和策略梯度方法（Policy Gradient）。免模型学习实现简单且适用于实际场景，因而得到广泛应用。

为了更专业地解释强化学习中的主流算法，结合具体公式将使理解更加直观。我们会依次深入解释免模型学习（Model-Free Learning）和有模型学习（Model-Based Learning）的代表性算法。

1. 免模型学习（Model-Free Learning）

1.1 策略优化方法（Policy Optimization）

策略优化方法直接优化智能体的策略，使其能够最大化累积回报。其核心思想是直接**通过策略梯度方法更新参数，优化策略函数。**

A3C（Asynchronous Advantage Actor-Critic）

A3C 是一种异步的 Actor-Critic 算法，它将 Actor 和 Critic 结合，并使用多个异步的智能体进行并行训练来提升效率。A3C 的目标是优化**策略函数** $\pi_\theta(a|s)$ 以及**值函数** $V^\pi(s)$。

> **策略函数** $\pi_\theta(a|s)$：定义了在状态 s 下选择动作 a 的概率，决定了智能体的行为。
>
> **值函数** $V^\pi(s)$：定义了在状态 s 下，遵循策略 π 所能获得的期望累积回报，反映了状态的价值。

- **策略梯度**：更新策略的梯度基于优势函数 $A(s,a)$，即
  $\nabla_\theta J(\theta) = \mathbb{E}_{s\sim d^\pi, a\sim\pi_\theta}\left[\nabla_\theta \log \pi_\theta(a|s) A(s,a)\right]$

其中 $A(s,a) = Q(s,a) - V(s)$，用于表示当前动作相比于平均策略的好坏程度。

- **价值函数更新**：值函数使用TD误差（时间差分误差）更新：$\delta_t = R_{t+1} + \gamma V(s_{t+1}) - V(s_t)$

Actor 更新策略参数 θ，Critic 更新值函数参数 w 来估计 $V(s)$。

PPO（Proximal Policy Optimization）

PPO 引入了一个**损失函数的剪辑机制**，限制策略更新的幅度。其关键公式为：

$$L^{\text{CLIP}}(\theta) = \mathbb{E}_t\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right]$$

其中 $r_t(\theta) = \dfrac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$，是策略变化比率，$\hat{A}_t$ 是优势函数，ϵ 是限制变化幅度的超参数。**该损失函数通过对更新幅度进行裁剪，保证了训练过程中的稳定性。**

1.2 Q-Learning 方法（Q-Learning）

Q-Learning 算法通过学习最优的**动作值函数** $Q(s,a)$ 来间接优化策略，它不直接优化策略，而是通过值函数指导动作选择。

小红书账号：小黄搞AI

DQN（Deep Q-Network）

DQN 是 Q-Learning 在深度学习框架下的扩展，通过神经网络估计 $Q(s,a)$ 函数，并引入经验回放和目标网络等技术。Q-Learning 的更新公式为：

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha \left(R_{t+1} + \gamma \max_{a'} Q(s_{t+1},a') - Q(s_t,a_t) \right)$$

其中 α 是学习率，γ 是折扣因子。DQN 用神经网络来近似 Q 值函数 $Q(s,a;\theta)$，并通过损失函数优化网络参数：

$$L(\theta) = \mathbb{E}_{(s,a,r,s')} \left[\left(r + \gamma \max_{a'} Q(s',a';\theta^-) - Q(s,a;\theta) \right)^2 \right]$$

为了提升训练的稳定性，DQN 使用了目标网络 $Q(s,a;\theta^-)$ 来保持 Q 值的平滑更新。

C51 和 Rainbow

C51 将回报的分布作为学习对象，其核心在于学习每个状态-动作对的回报分布，而非单一的期望值：

$$Z(s,a) = \sum_{i=1}^{N} z_i \cdot P(Z = z_i | s,a)$$

其中 $Z(s,a)$ 是分布的离散化估计，C51 通过改进 Q-Learning 来提升性能。

Rainbow 则结合了多个强化学习中的增强技术，包括双 Q-Learning、优先经验回放和分布式 Q-Learning 等，进一步提升了 Q-Learning 的性能和稳定性。

2. 有模型学习（Model-Based Learning）

有模型学习通过学习环境的动态模型来进行规划，这类方法能够模拟环境的未来状态，并提前规划动作。

2.1 模型预测控制（Model Predictive Control, MPC）

MPC 使用环境模型在每一个时间步规划未来一段时间的动作序列。

核心思想是在每个时间步解决如下优化问题：$\max_{a_0,\ldots,a_H} \sum_{t=0}^{H} R(s_t,a_t)$

其中 H 是规划的时域长度，$s_{t+1} = f(s_t,a_t)$ 是由模型 f 预测的状态转移。MPC 执行优化后的第一个动作，然后重新进行规划。

2.2 Expert Iteration（专家迭代）

Expert Iteration 将规划和策略学习结合。通过如蒙特卡洛树搜索（MCTS）等规划方法生成专家动作，并使用这些动作来训练策略函数。其目标是改进当前策略：

$$\pi_{\text{new}}(a|s) = \arg\max_{\pi} \mathbb{E}_{s \sim \pi_{\text{old}}} \left[\sum_{t=0}^{\infty} \gamma^t R(s_t,a_t) \right]$$

AlphaZero 是 Expert Iteration 的一个典型例子，它结合 MCTS 和深度强化学习，在复杂的游戏如围棋中达到了超越人类顶级玩家的水平。

小红书账号：小黄搞AI

3. 总结

主流的强化学习方法可以划分为免模型学习和有模型学习两类。免模型学习通过直接优化策略或值函数来最大化累积奖励，代表方法有 A3C、PPO、DQN 和其改进算法；有模型学习通过构建环境模型来进行长远规划，代表方法有 MPC 和 Expert Iteration。每种方法在不同的应用场景中有其独特的优势和局限性。

44. 强化学习中策略函数和值函数是什么

在强化学习中，**策略函数**和**值函数**是两个核心的概念，它们分别用于描述**智能体如何选择动作以及这些动作的长期回报**。

策略函数（Policy Function）

策略函数 $\pi_\theta(a|s)$ 描述了在给定状态 s 下，智能体选择某个动作 a 的概率或确定性规则。它决定了智能体的行为方式，指导智能体在不同状态下应采取的动作。

- **随机策略：**

  如果智能体根据概率来选择动作，策略函数 $\pi_\theta(a|s)$ 定义了智能体在状态 s 下以概率 $\pi_\theta(a|s)$ 选择动作 a 的分布。

  例如，在随机策略下：$\pi_\theta(a|s) = P(a|s)$，表示给定状态 s ，动作 a 的选择概率。

- **确定性策略：**

  如果智能体在每个状态下总是选择一个固定的动作。

  策略函数就是一个确定性的函数：$a = \pi_\theta(s)$

在这种情况下，策略函数直接输出智能体在状态 s 下所采取的动作 a 。

在策略优化算法（如 A3C、PPO）中，策略函数通常由神经网络近似表示，网络的参数是 θ ，通过优化这些参数使得策略能在不同状态下选择最优的动作。

值函数（Value Function）

值函数 $V^\pi(s)$ 描述了在状态 s 下，按照策略 π 行动时能够获得的**期望累积回报**。它反映了智能体在当前状态下的“好坏”，即从状态 s 开始，智能体期望未来能够获得的奖励总和。

值函数的定义为：

$$V^\pi(s) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \middle| s_0 = s, a_t \sim \pi\right]$$

- **期望累积回报：**这里，γ 是折扣因子，$R(s_t, a_t)$ 是智能体在时间步 t 采取动作 a_t 后获得的即时奖励。值函数是这些奖励的加权和，表示从状态 s 开始，按照策略 π 行动后未来能够获得的总奖励的期望值。
- **状态值函数和动作值函数：**
  - **状态值函数** $V^\pi(s)$ 描述的是在状态 s 下，期望未来的累积回报。
  - **动作值函数** $Q^\pi(s, a)$ 则是考虑在状态 s 下选择某个特定动作 a 时的期望累积回报，即：

    $$Q^\pi(s, a) = \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t) \middle| s_0 = s, a_0 = a, a_t \sim \pi\right]$$

在 Q-Learning 等算法中，动作值函数 $Q(s, a)$ 是核心，用于指导智能体在每个状态下选择最佳动作。

小红书账号：小黄搞AI

总结

- **策略函数** $\pi_\theta(a|s)$：定义了在状态 s 下选择动作 a 的概率，决定了智能体的行为。
- **值函数** $V^\pi(s)$：定义了在状态 s 下，遵循策略 π 所能获得的期望累积回报，反映了状态的价值。

这两者结合起来为强化学习算法提供了强大的工具，策略函数指导行动，值函数评估行动的长期效果。

45. DPO的原理

Direct Preference Optimization (DPO) 是一种新的强化学习算法，特别设计用于解决偏好学习（Preference Learning）问题。**DPO 的核心思想是通过优化用户或专家给定的偏好信号，而非直接优化奖励函数。**该算法的灵感来源于奖励建模和偏好优化相结合的策略。

1. 偏好模型的建立

DPO 通过学习一个偏好模型 $P(a|s)$，该模型用于预测在某一状态 s 下，某个动作 a 是否被用户（或专家）所偏好。这一模型的输入是状态 s 和动作 a，输出是一个偏好概率 $P(a|s)$。偏好模型的作用**类似于传统强化学习中的策略模型，但它基于用户偏好，而不是直接通过奖励信号来学习**。可以表示为：$P(a|s) = \text{softmax}(f_\theta(s,a))$

其中 $f_\theta(s,a)$ 是偏好模型的打分函数，参数为 θ，而 softmax 函数确保输出为有效的概率分布。

2. 偏好比较

为了优化偏好模型，DPO 首先需要定义偏好的比较方式。在给定一对动作 (a_i, a_j) 和状态 s 的情况下，模型需要判断哪个动作更被偏好。这可以通过定义一个偏好比较函数 $P(a_i \succ a_j|s)$，**即在状态 s 下，动作 a_i 相较于 a_j 更被偏好的概率**。

该概率可以通过偏好模型计算得到：

$$P(a_i \succ a_j|s) = \frac{P(a_i|s)}{P(a_i|s) + P(a_j|s)} = \frac{\exp(f_\theta(s,a_i))}{\exp(f_\theta(s,a_i)) + \exp(f_\theta(s,a_j))}$$

这个公式是通过 softmax 函数得出的，表示在两个动作之间，哪个更有可能被偏好。

3. 损失函数的定义

为了训练偏好模型，DPO 使用了基于二元交叉熵的损失函数。这一损失函数旨在最小化模型预测偏好与实际用户（或专家）偏好之间的差异。

对于每一对动作 (a_i, a_j) 和对应的状态 s，我们可以定义以下损失函数：

$$L(\theta) = -\mathbb{E}_{(a_i \succ a_j, s)}[\log P(a_i \succ a_j|s)]$$

通过最小化这个损失函数，偏好模型会不断调整其参数，使得预测出的偏好分布越来越接近真实的用户偏好分布。

4. 策略优化

小红书账号：小黄搞AI

通过学习到的偏好模型 $P(a|s)$，我们可以将其用于生成策略 $\pi_\theta(a|s)$。DPO 的策略优化过程直接依赖于偏好模型，即策略的概率分布可以通过偏好模型的 softmax 输出进行参数化：

$$\pi_\theta(a|s) = P(a|s) = \frac{\exp(f_\theta(s,a))}{\sum_{a'} \exp(f_\theta(s,a'))}$$

这意味着 DPO 的策略优化并不依赖于传统的奖励信号，而是完全基于偏好信号进行优化。通过优化偏好损失函数，我们可以间接优化策略的性能，使其更符合用户的期望。

5. 偏好学习与传统强化学习的区别

传统的强化学习通过最大化累积奖励来训练策略，公式如下：$J(\pi_\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_t \gamma^t r(s_t, a_t)\right]$，其中 $r(s_t, a_t)$ 是在时间步 t 时，状态 s_t 下选择动作 a_t 后得到的奖励。

然而**在 DPO 中，目标是最大化偏好模型的准确性，而不是累积奖励。**因此，DPO 并不直接依赖环境提供的奖励信号，而是通过优化偏好比较损失函数 $L(\theta)$ 来训练策略。这种优化过程更符合人类偏好的决策方式。

6. 扩展与优化

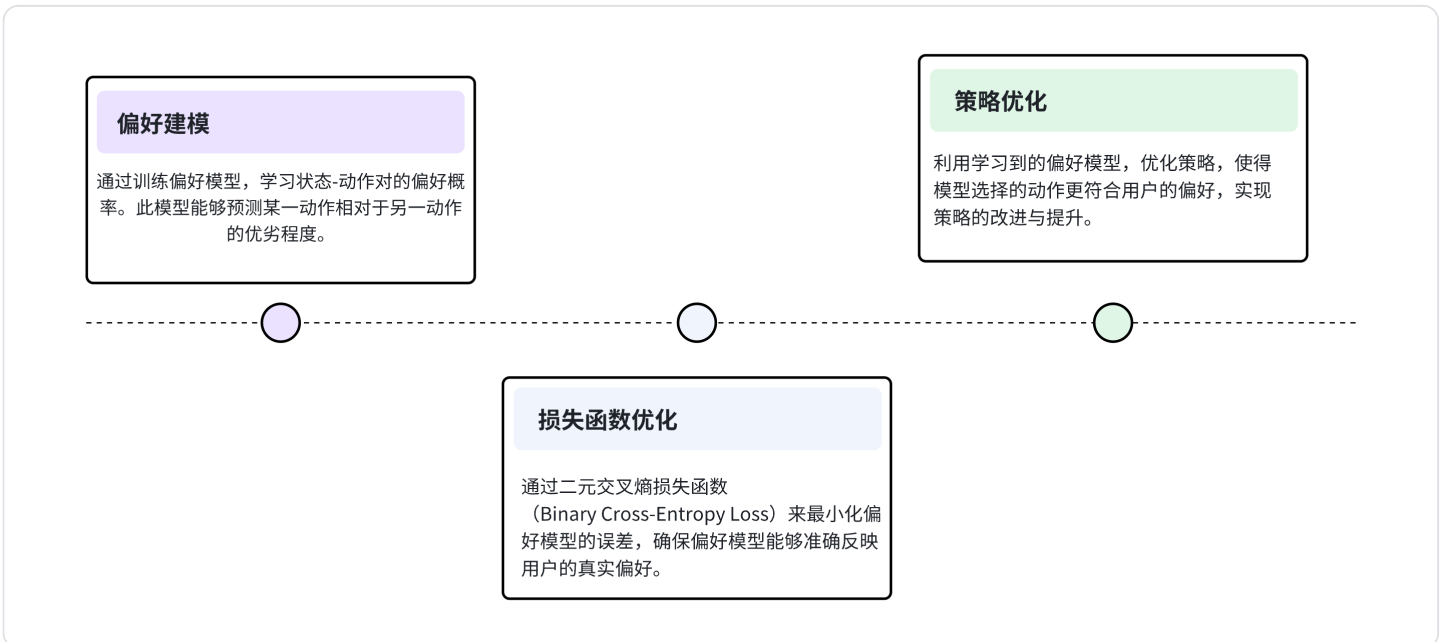
为了提高 DPO 的效率和稳定性，常见的优化技术包括：

偏好采样策略：通过经验回放池采样不同的偏好比较对，确保训练样本的多样性。

策略约束：类似于 PPO（Proximal Policy Optimization）中的剪切策略，通过限制每次策略更新的幅度来避免过度更新，保证训练的稳定性。

7. 总结

DPO 通过优化偏好信号代替直接的奖励信号进行学习，其关键步骤包括偏好建模、损失函数优化、策略优化。



小红书账号：小黄搞AI

DPO 的优势在于它适用于那些无法明确获得奖励信号的场景，尤其是在需要模拟人类偏好的任务中，如推荐系统、对话系统等。

46. PPO的原理

PPO（Proximal Policy Optimization，近端策略优化）是一种常用的强化学习算法，属于**策略优化方法**。其核心思想是**通过限制策略更新的幅度来提升训练的稳定性，避免策略在每次更新中发生较大变化，从而防止训练过程中的不稳定和策略崩溃。**

1. PPO的核心思想

PPO通过一种**剪辑损失函数（Clipped Surrogate Objective）**来实现策略的优化，它的**目标是确保策略更新时不会偏离当前策略太多。它克服了传统策略梯度方法**（如TRPO，Trust Region Policy Optimization）复杂的约束问题，保持了性能的同时降低了实现难度。

传统的策略优化目标是直接优化策略 $\pi_\theta(a|s)$，使得给定状态下的行为能够最大化累计奖励。PPO在此基础上，通过引入一个重要的比例项 **"probability ratio"** $r_t(\theta)$ 来度量新旧策略之间的差异：

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

其中，$\pi_{\theta_{\text{old}}}$ 是更新前的策略，而 π_θ 是更新后的策略。该比率表示在同一状态-行为对 (s_t, a_t) 下，新旧策略选择该行为的概率之比。如果这个比率偏离1过多，说明更新后的策略与原始策略差异较大，可能会导致训练不稳定。

2. PPO的损失函数

PPO的损失函数是基于策略比率 $r_t(\theta)$ 进行定义的，它有两种主要形式：**剪辑形式**和**KL散度形式**。

2.1 剪辑损失函数

PPO的最经典形式是剪辑损失函数，它通过对策略更新的比率进行剪辑来避免策略大幅度更新。

其损失函数可以表示为：$L^{CLIP}(\theta) = \mathbb{E}_t\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right]$

- $r_t(\theta)$：新旧策略的概率比。
- $\hat{A}_t$：优势函数（Advantage Function），表示在时间步 t 的策略相对于基准策略的好坏，通常为优势估计 $A_t = Q(s_t, a_t) - V(s_t)$。
- ϵ：剪辑系数，用于限制策略更新幅度。

这个损失函数的设计核心在于对比两项值：

- 第一项 $r_t(\theta)\hat{A}_t$，即直接优化的目标函数；
- 第二项 $\text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t$ 对比率 $r_t(\theta)$ 进行剪辑，限制其变化范围，防止更新过大。

小红书账号：小黄搞AI

损失函数取这两者中的较小值，意味着如果比率 $r_t(\theta)$ 超过了允许的范围（即 $1-\epsilon$ 到 $1+\epsilon$），则不再继续优化，从而抑制大幅度的策略更新。

2.2 基于KL散度的PPO

除了剪辑形式外，PPO还有一种基于**KL散度约束**的形式。

KL散度用于度量新旧策略分布的差异，通常定义为：

$$D_{KL}[\pi_{\theta_{\text{old}}}||\pi_\theta] = \mathbb{E}_t\left[\log \pi_{\theta_{\text{old}}}(a_t|s_t) - \log \pi_\theta(a_t|s_t)\right]$$



在KL约束下的PPO目标函数为：$L^{KL}(\theta) = \mathbb{E}_t\left[r_t(\theta)\hat{A}_t - \beta D_{KL}[\pi_{\theta_{\text{old}}}||\pi_\theta]\right]$

其中，β 是一个权衡系数，用于控制KL散度的影响。如果新旧策略差异过大，KL项的惩罚会增大，强制策略更新更为谨慎。

3. 优势函数 A_t 的估计

在PPO中，**优势函数** $\hat{A}_t$ 的准确估计对策略的优化效果至关重要。

通常，优势函数被定义为当前策略相对于基准策略的相对表现：$A(s_t, a_t) = Q(s_t, a_t) - V(s_t)$

其中：

- $Q(s_t, a_t)$ 是给定状态和行为下的动作值函数，表示采取动作 a_t 后的预期回报；
- $V(s_t)$ 是状态值函数，表示在状态 s_t 下的预期回报。



PPO算法中，A_t 通常通过**广义优势估计（GAE, Generalized Advantage Estimation）**方法来进行估计，GAE通过加权累积时间差分误差来平衡偏差和方差：

$$\hat{A}_t = \sum_{l=0}^{\infty}(\gamma\lambda)^l \delta_{t+l}$$

其中：

- $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ 是时间差分误差；
- γ 是折扣因子；
- λ 是衰减因子。

4. PPO的优缺点

优点	缺点
更新稳定：通过剪辑比率或KL散度约束，PPO限制了每次更新的幅度，**防止策略发生剧烈变化，从而提高了算法的稳定性**。	**样本效率较低** PPO属于**同策略算法**，每次更新只能使用当前策略采集的数据，导致样本效率较低。

小红书账号：小黄搞AI

易于实现：与TRPO相比，PPO无需复杂的约束优化算法，易于实现和调整超参数。

泛用性强：PPO适用于离散和连续动作空间，广泛应用于各种强化学习任务。

超参数敏感性

PPO对剪辑参数 ϵ 和 GAE 中的 λ 等超参数敏感，需要在实际应用中进行细致的调整。

5. 总结

PPO通过引入剪辑或KL散度约束，在确保策略稳定更新的同时，简化了实现难度。其在连续控制任务、机器人控制、复杂游戏等场景中表现出色，成为强化学习领域应用最为广泛的策略优化算法之一。

47. 分析一下DPO和PPO的区别

DPO（Direct Preference Optimization）和 PPO（Proximal Policy Optimization）是两种用于优化策略的算法，尽管它们的应用场景和优化目标不同，但都属于策略优化方法。以下从**目标函数、算法思路、优化方法、以及公式推导**等方面进行对比，帮助理解它们的核心区别。

1. 目标和应用场景

PPO

PPO是一种用于强化学习（Reinforcement Learning, RL）的**策略优化算法**。其目的是通过与环境交互，**在累计长期奖励的基础上优化策略**。PPO主要用于训练智能体如何通过连续的动作获得最大的累计奖励。

DPO

DPO是一种**偏好优化算法**，主要用于偏好学习任务（如大模型的偏好微调）。**它的目标是通过直接从人类偏好数据中优化策略，使得输出更加符合用户偏好**，而不是通过显式的奖励模型进行优化。

2. 优化目标的对比

PPO的优化目标

PPO通过限制策略更新的幅度，来保证策略的稳定性，避免策略发生过大的变化。其目标函数通过引入一个剪辑函数来实现这一点。PPO的优化目标可以写作：

$$L^{\text{PPO}}(\theta) = \mathbb{E}_t \left[\min \left(r_t(\theta) A_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon) A_t \right) \right]$$

其中：

- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ 是策略更新比率，π_θ 是当前策略，$\pi_{\theta_{\text{old}}}$ 是旧策略。
- A_t 是优势函数（Advantage Function），用于衡量某一动作的优劣。

DPO的优化目标

DPO跳过了显式奖励模型的训练过程，直接从偏好数据中优化策略。它的目标函数是最大化偏好分布下的对数似然：

$$L^{\text{DPO}}(\theta) = \sum_i \log \sigma \left(\frac{1}{\beta} \log \frac{\pi_\theta(y_{i,1}|x_i)}{\pi_\theta(y_{i,2}|x_i)} \right)$$

其中：

- $\sigma(\cdot)$ 是 Sigmoid 函数，用于衡量两个输出 $y_{i,1}$ 和 $y_{i,2}$ 之间的人类偏好差异。

小红书账号：小黄搞AI

- ϵ 是一个超参数，用于限制策略更新的幅度，避免策略发生过大的变化。
- β 是用于控制策略更新幅度的参数。

PPO的核心思路是通过对策略更新的比例进行裁剪，限制策略更新幅度，确保策略不会离旧策略过远，从而保证训练的稳定性。

DPO 的核心思想是直接根据人类偏好数据来调整策略，使策略更符合人类的期望。

3. 策略更新的约束

PPO的策略约束

PPO通过引入剪辑机制控制策略的变化幅度，防止策略在单次更新中偏离太多。通过限制策略更新的比率 $r_t(\theta)$ ，PPO避免了策略不稳定现象。其目标函数中的clip函数对策略更新做了以下限制：

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$$

如果 $r_t(\theta)$ 超出了 $[1-\epsilon, 1+\epsilon]$ 范围，目标函数就会强制剪裁，限制更新幅度，从而实现稳健的更新过程。

DPO的策略约束

DPO 通过**KL散度**进行约束，以确保新策略不会与参考策略偏离过远。DPO 的策略更新形式如下：

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

这里的 $\pi_{\text{ref}}(y|x)$ 是参考策略，它通常是一个预训练的语言模型。通过控制 β ，DPO在参考策略和优化后的策略之间找到平衡，确保策略的变化不会过大。

4. 应用任务的区别

PPO

PPO 适用于典型的强化学习任务，如在游戏环境中训练智能体、机器人控制等。**PPO算法通常在奖励信号稀疏的情况下表现出色，通过不断交互和调整策略，逐步提高累计奖励。**

DPO的策略约束

DPO 更适用于偏好学习任务，如在大型语言模型上进行人类偏好微调。它不需要通过环境交互来优化，而是**通过人类提供的偏好数据直接优化生成模型的输出。相比于PPO的强化学习场景，DPO避免了复杂的奖励模型构建。**

5. 对比总结

特性	PPO	DPO
目标函数	最大化长期奖励，加入KL约束和剪裁机制	最大化人类偏好，直接优化策略不需要显式奖励模型

小红书账号：小黄搞AI

适用场景	强化学习，智能体交互场景	偏好学习，语言模型微调等
策略更新方式	基于剪裁的策略比率更新	基于KL散度和人类偏好直接调整策略
优化方法	通过策略梯度法更新	通过对数似然优化偏好数据下的策略
复杂性	依赖环境交互和奖励模型	不依赖奖励模型，通过偏好数据直接学习
控制策略变化的方式	剪裁策略更新幅度	KL散度约束，控制策略偏离参考策略

6. 总结与结论

🌟 **PPO** 是一种基于强化学习的策略优化方法，通过引入剪裁机制来防止策略更新过大，确保策略的稳定性。PPO主要用于长期累计奖励的优化，适用于典型的RL场景。

DPO 是一种基于人类偏好直接优化策略的方法，通过对人类偏好数据的对数似然优化策略，而不依赖于显式的奖励模型。DPO常用于大型语言模型的微调等任务。

两者在应用场景和优化目标上差异明显，PPO侧重于基于环境奖励的强化学习任务，而DPO则更适合偏好学习的场景。

48. DPO的正负样本怎么构造

DPO正负样本构造过程

在DPO（Direct Preference Optimization）中，正负样本的构造是基于偏好数据的比较结果。具体来说，它通过对不同候选输出的偏好来决定哪一个样本是正样本，哪一个是负样本。这个过程涉及以下步骤：

Step1

收集偏好数据

从人类反馈或其他系统中获取用户偏好的数据集。通常这些数据以成对的形式出现，每对数据表示一个模型输出的选择，用户需要对其中一个输出表示偏好。例如，在对话模型中，给定相同的输入，可以生成不同的回复，而用户会更偏好其中的某一个。

Step2

定义偏好关系

对于每一对候选输出x1和x2。
如果用户更喜欢x1（表现为用户反馈，或通过显式标注的方式），则我们将x1作为正样本，x2作为负样本。反之亦然。

数学上，可以表示为： $x_1 \succ x_2$ ，

其中“$\succ$”表示用户对 x_1 的偏好强于 x_2 。

Step3

构造训练样本

根据偏好关系，我们将偏好高的样本（正样本）与偏好低的样本（负样本）对比，形成训练对。

例如： 正样本：$(x_1, \text{正})$ 负样本：$(x_2, \text{负})$

Step4

采样策略

为了提高训练效率，通常可以采用一定的采样策略，不是对所有可能的样本对进行训练，而是针对那些模型输出差异较大的样本对，或是模型当前预测有不确定性的样本对进行优先训练。

小红书账号：小黄搞AI

在训练过程中，DPO通过最大化偏好模型对这些样本的对数似然来优化目标函数，使得偏好模型能够更好地预测用户的选择。

❤️ **面试建议**：面试时，除了描述理论过程，可以结合具体的应用场景，如对话生成、文本生成等领域，来讲述如何通过用户的反馈数据构造正负样本，使回答更加贴合实际需求。

场景：对话生成模型中的偏好数据收集与正负样本构造

在对话生成任务中，模型需要给定一个输入问题生成适当的回答。为了优化模型的生成效果，我们希望通过人类反馈来调整模型的行为。

1. 数据收集

我们可以通过以下步骤来收集偏好数据：

- **模型生成候选对话回复**：给定一个用户输入（如用户问题或聊天对话），生成模型会给出多种候选回复。假设用户输入是：
  - 用户输入：“天气怎么样？”
  - 模型生成的两条回复分别为：
    i. 回复1：“今天的天气非常好，晴朗无云。”
    ii. 回复2：“不清楚天气，您可以查一下天气预报。”
- **人类反馈**：接下来，系统会展示这两条回复给人类评审员。评审员基于回复的自然性、相关性、信息量等方面，选择其中更符合用户需求的回复。例如，评审员选择了**回复1**，因为它更加具体且回答了用户的问题。

2. 偏好数据生成

根据人类评审员的选择，我们得到如下的偏好数据：

- 回复$1 \succ$ 回复2，即回复1被评为更优。

3. 构造正负样本

根据评审员的反馈，构造正负样本：

- **正样本**：回复1作为正样本，因为它更符合用户的偏好。
- **负样本**：回复2作为负样本，因为它被评为相对较差的回复。

这对正负样本可以用于DPO的训练，其中模型将被优化为更倾向于生成类似回复1的答案，而抑制生成回复2这样的回答。

4. 多轮对话扩展

在多轮对话系统中，用户不仅对单个回复进行评估，还可以对一段完整的对话进行

小红书账号：小黄搞AI



反馈。此时，系统可以根据用户与模型交互过程中整段对话的质量进行偏好标注。

例如，用户对整个对话中的某一轮次的回复给予正向反馈（如点赞或正面评价），那么该轮次中的回复可以作为正样本，其他候选可以作为负样本。

5. 使用反馈优化模型

通过收集大量类似的偏好数据，模型能够不断改进生成的质量。在DPO的框架下，模型通过最大化偏好模型对正负样本的预测准确率，逐步优化输出结果，使其更符合人类偏好。

总结

📍 在这个对话生成的场景中，正负样本的构造基于人类对候选回复的偏好选择，正样本是用户偏好较高的回复，负样本是用户偏好较低的回复。这个流程可以扩展到其他生成任务中，如机器翻译、文本生成等，都可以通过类似的偏好数据收集和样本构造来优化模型。

49. DPO 的 loss 是什么?

🌅 在 Direct Preference Optimization (DPO) 中，损失函数 (loss) 的构造基于偏好数据对的对比学习。**DPO 的核心思想是利用偏好数据，通过优化策略，使得模型在偏好对中选择更优的选项。**

1. 偏好模型的假设

DPO 中假设存在一个偏好模型 f_θ ，其参数为 θ ，**该模型能够对给定的两个候选项进行打分，从而表示出对其中某个选项的偏好**。对于给定的一对数据 (x_+, x_-) ，其中 x_+ 是偏好数据对中更优的选项，x_- 是较差的选项，DPO 希望通过训练使得模型给出的评分满足以下条件： $f_\theta(x_+) > f_\theta(x_-)$ ，即模型倾向于选择被偏好的样本。

2. 偏好对的损失函数定义

为了量化模型评分结果的好坏，DPO 通常使用基于对比学习的损失函数，比如二元交叉熵 (binary cross-entropy) 损失或对数损失。

小红书账号：小黄搞AI

🥛 在 DPO 中，常见的损失函数形式可以表示为：

$\mathcal{L}(\theta) = -\mathbb{E}_{(x_+, x_-)\sim D}\left[\log \sigma(f_\theta(x_+) - f_\theta(x_-))\right]$

其中：

- $\sigma(z) = \frac{1}{1+e^{-z}}$ 是 sigmoid 函数，用于将分数差值映射到 $[0,1]$ 的概率空间。
- $f_\theta(x_+) - f_\theta(x_-)$ 表示偏好对中更优选项和较差选项的评分差。
- D 是偏好数据集的分布。

这个损失函数的目的是使模型对 x_+ 和 x_- 的评分差尽可能大，从而确保模型更倾向于选择 x_+ 作为偏好选项。

3. 损失函数的解释

DPO 的损失函数本质上是一个对比损失，**它通过最大化偏好对中两个选项评分的差异来优化模型**。

Sigmoid 函数 $\sigma(f_\theta(x_+) - f_\theta(x_-))$ 的输出是模型对 x_+ 被偏好的概率。如果模型给出更大的正评分差 $f_\theta(x_+) - f_\theta(x_-)$ ，那么这个概率会更接近于 1，相应的损失会趋向于 0。反之，如果模型未能正确区分 x_+ 和 x_- ，即评分差接近于 0 或为负，则损失值会增大。

通过最小化损失函数，模型会逐渐调整参数 θ ，以使其输出的评分能够更好地反映训练数据中的偏好信息。

4. 总结

🚄 **DPO 的损失函数是通过对比偏好数据对中的评分差来进行优化的，其目标是使模型能够更准确地区分偏好对中的优劣选项**。数学上，损失函数可以用如下公式表示：
$\mathcal{L}(\theta) = -\mathbb{E}_{(x_+, x_-)\sim D}\left[\log \sigma(f_\theta(x_+) - f_\theta(x_-))\right]$ 这个公式通过最大化偏好对之间的评分差，逐步提升模型的性能。

50. OpenAI对齐为什么要用强化学习，别的方法不行吗？

🦄 在人工智能的发展过程中，对齐（Alignment）问题备受关注，**即如何确保AI模型的行为与人类的期望和价值观一致**。OpenAI采用了强化学习（Reinforcement Learning），特别是基于人类反馈的强化学习（Reinforcement Learning from Human Feedback，RLHF），来解决这一问题。下面将从根源上逐个概念进行解释，探讨为什么强化学习是对齐的最佳选择，以及其他方法的局限性。

50.1 对齐问题的背景

小红书账号：小黄搞AI

1. 什么是对齐？

- **定义**：对齐是指使人工智能系统的目标、行为与人类的意图和价值观保持一致。
- **重要性**：未对齐的AI可能会产生意外或不良的行为，导致安全和伦理问题。

2. 对齐的挑战

- **复杂性**：人类价值观复杂多样，难以用简单的规则或目标函数完全描述。
- **不可预测性**：AI模型可能在训练数据之外的情境中表现出未预期的行为。
- **黑箱问题**：深度学习模型的内部决策过程难以解释和控制。

50.2 传统方法的局限性

基于规则的系统

特点：预先定义明确的规则或逻辑，指导AI的行为。

局限性：

- **不灵活**：无法适应复杂、多变的现实环境。
- **难以穷尽**：人类价值观和情境的多样性难以完全编码。

监督学习

特点：使用标注的数据集训练模型，学习输入到输出的映射关系。

局限性：

- **数据依赖**：需要大量高质量的标注数据，获取成本高。
- **泛化能力**：可能在未见过的情境中表现不佳。
- **无法捕捉偏好**：难以反映人类对输出质量的细微偏好和价值判断。

50.3 强化学习在对齐中的优势

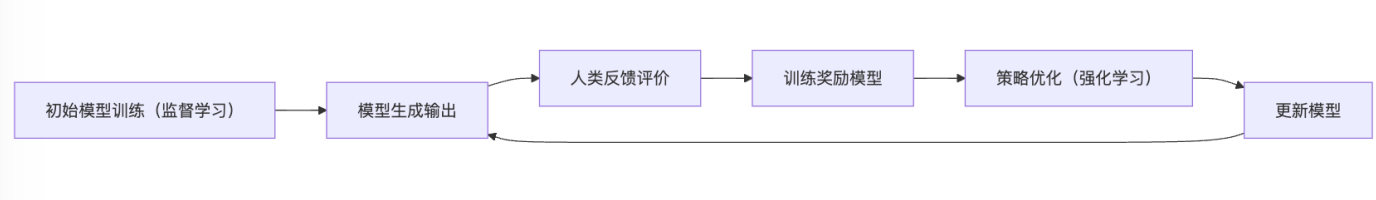
1. 什么是强化学习？

- **定义**：强化学习是一种通过与环境交互，学习如何采取行动以最大化累积奖励的机器学习方法。
- **核心元素**：
  - **智能体（Agent）**：学习和决策的主体。
  - **环境（Environment）**：智能体所处的外部世界。
  - **状态（State）**：环境在某一时刻的信息。
  - **动作（Action）**：智能体可执行的行为。
  - **奖励（Reward）**：环境对智能体动作的反馈，指导学习方向。

2. 基于人类反馈的强化学习（RLHF）

- 流程

小红书账号：小黄搞AI



a. **初始模型训练**：使用监督学习，基于现有数据训练初始语言模型。

b. **收集人类反馈**：模型生成多个响应，由人类对这些响应进行比较和评价。

c. **训练奖励模型**：利用人类反馈的数据，训练一个能够评估模型输出质量的奖励模型。

d. **策略优化**：使用强化学习方法，根据奖励模型的评估结果，优化原始语言模型的策略。

- **优势**
  - **直接融入人类价值观**：通过人类反馈，模型能更好地理解和遵循人类偏好。
  - **高效学习**：强化学习能从较少的反馈中学习到有效的策略。
  - **动态适应**：模型可以持续更新，适应新的反馈和要求。

50.4 为什么其他方法不如强化学习有效

监督学习的不足	强化学习的优势
无法捕捉复杂偏好：监督学习主要关注正确的输入输出映射，难以反映人类对响应质量的细微偏好。	**处理复杂目标**：通过奖励函数，可以表达复杂的、难以明确编码的目标和偏好。
数据限制：获取涵盖所有可能情境的标注数据几乎不可能。	**有效利用反馈**：强化学习能从人类的比较或排名等相对反馈中学习，而不需要绝对的正确答案。
过拟合风险：模型可能过度依赖训练数据，缺乏在新情境下的泛化能力。	**提高模型对齐度**：通过持续优化，模型的行为更符合人类期望。

50.5 总结

OpenAI在对齐过程中选择强化学习，特别是基于人类反馈的强化学习，主要原因在于：

- **直接融入人类偏好**：强化学习能够利用人类的直接反馈，使模型更好地理解和满足人类的需求。
- **处理复杂和抽象的目标**：通过设计合适的奖励函数，强化学习可以优化难以用明确规则描述的目标。
- **提高模型的泛化能力**：模型能更好地适应新情境，减少不期望行为的发生。

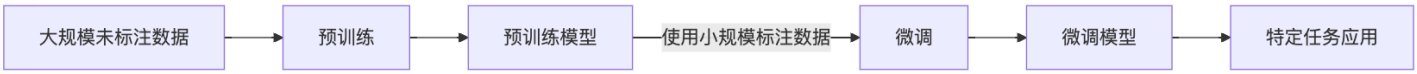
其他方法的局限性使得它们在对齐任务中效果不佳。监督学习无法充分利用人类的偏好信息，基于规则的系统又缺乏灵活性和适应性。因此，强化学习成为对齐AI模型的更有效方法。

> 强化学习，特别是基于人类反馈的强化学习，为AI模型的对齐问题提供了强有力的解决方案。它能够有效地将人类的价值观和偏好融入模型训练，使得AI系统的行为更加符合人类的

小红书账号：小黄搞AI

期望。虽然其他方法在某些方面有其价值，但在对齐任务的核心需求上，强化学习展现出了独特的优势，这也是OpenAI选择其作为主要方法的原因。

51. 预训练和微调任务有什么区别，两者的目的



51.1 概述

在深度学习和自然语言处理（NLP）领域，预训练（Pre-training）和微调（Fine-tuning）是模型训练的两个关键阶段。它们在模型的构建和应用中扮演着不同的角色，具有不同的目的。下面将从根源上逐个概念解释两者的区别和目的。

51.2 预训练

1. 定义

预训练是指在大规模的未标注数据上训练模型，使其学习通用的特征表示。这一阶段不针对特定任务，而是让模型掌握数据的基本结构和模式。

2. 目的

学习通用特征： 通过在大量数据上训练，模型能够捕获数据的通用特征和模式。

提高效率： 预训练好的模型可以作为下游任务的基础，减少训练时间和数据需求。

3. 方法

自监督学习： 利用数据本身的结构信息作为监督信号，

例如：

- **语言模型任务**：预测下一个词（如GPT系列）。
- **遮盖语言模型（Masked Language Model）**：预测被遮盖的词（如BERT）。

4. 特点

- **数据类型：** 通常使用未标注的大规模数据集。
- **训练目标：** 学习数据的内在结构和一般模式。

51.3 微调

1. 定义

小红书账号：小黄搞AI

微调是指在预训练模型的基础上，使用小规模的标注数据针对特定任务进行训练，以适应具体应用需求。

2. 目的

适应特定任务： 使预训练模型在特定任务上达到最佳性能。

利用预训练知识： 通过微调，模型可以在特定任务上充分发挥预训练阶段学到的知识。

3. 方法

监督学习： 使用标注的任务数据，对模型进行训练和优化。

调整模型参数： 在预训练模型的参数基础上，针对新任务进行细微调整。

4. 特点

数据类型： 使用针对特定任务的标注数据集。

训练目标： 优化模型在特定任务上的表现。

51.4 预训练和微调的区别

数据类型	训练目标
预训练：使用未标注的大规模数据集。 **微调**：使用标注的特定任务数据集。	**预训练**：学习通用的特征表示，捕捉数据的基本模式。 **微调**：针对特定任务优化模型性能。
应用范围	**训练方式**
预训练：通用，适用于多种下游任务。 **微调**：专用，针对特定任务进行优化。	**预训练**：自监督学习，无需人工标注数据。 **微调**：监督学习，需要人工标注的数据。

51.5 两者的关系

预训练为微调提供基础： 预训练阶段获得的通用特征表示，为微调阶段提供了良好的起点。

微调提升模型的特定任务性能： 通过在预训练模型上进行微调，可以在特定任务上取得更好的效果，而无需从零开始训练模型。

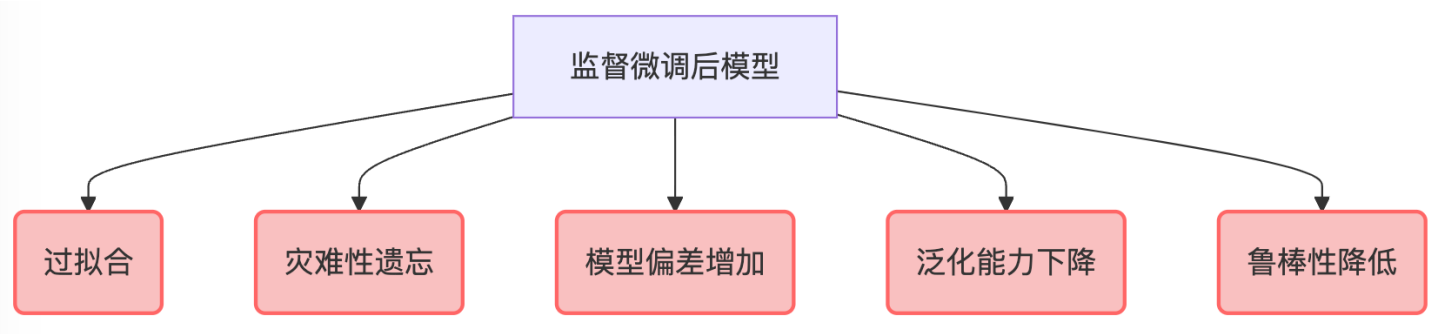
51.6 总结

预训练	微调
目的：学习通用的特征表示，捕获数据的基本模式。 **特点**：在大规模未标注数据上进行自监督学习。	**目的**：使模型适应特定任务，优化其在该任务上的性能。 **特点**：在小规模标注数据上进行监督学习，调整模型参数。

小红书账号：小黄搞AI

通过先预训练再微调的方式，可以充分利用大规模数据的优势，提高模型在特定任务上的性能。这种方法在NLP等领域已成为主流，显著提升了模型的泛化能力和应用效果。

52. SFT后会出现哪些问题



在深度学习和自然语言处理领域，**监督微调（Supervised Fine-Tuning，SFT）**是训练大型语言模型的重要步骤。尽管SFT能够使预训练模型适应特定任务，提高其在该任务上的性能，但在SFT之后可能会出现一些问题。回答从根源上逐个概念进行解释，探讨SFT后可能出现的问题。

在SFT之后，模型可能会面临以下几个主要问题：

1. **过拟合（Overfitting）**
2. **灾难性遗忘（Catastrophic Forgetting）**
3. **模型偏差增加**
4. **泛化能力下降**
5. **鲁棒性降低**

52.1 什么是监督微调（SFT）

1. 定义

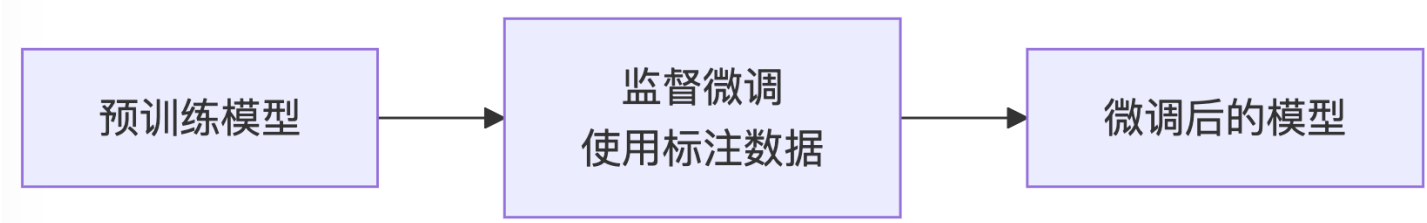
监督微调（SFT）：在预训练模型的基础上，使用标注的任务特定数据集，对模型进行进一步训练，以提高其在该任务上的性能。

2. 目的

适应特定任务：使预训练模型在特定的下游任务（如问答、翻译、分类等）上表现更优异。

3. 流程概览

小红书账号：小黄搞AI



52.2 SFT后可能出现的问题——过拟合（Overfitting）

1. 定义

过拟合是指模型在训练数据上表现良好，但在新数据（测试集或实际应用中）上性能较差。

2. 根源分析

- **数据规模有限**：微调数据集通常比预训练数据集小很多。
- **模型复杂度高**：大型模型具有大量参数，容易记住训练数据的细节。
- **缺乏正则化**：如果在微调过程中没有适当的正则化措施，模型可能过度拟合训练数据。

3. 影响

- **泛化能力差**：模型无法有效处理未见过的数据。
- **实际应用受限**：在真实场景中，模型性能可能无法满足要求。

4. 解决方案

- **数据增广**：增加训练数据的多样性。
- **正则化技术**：如早停（Early Stopping）、权重衰减等。
- **交叉验证**：评估模型的泛化性能。

52.3 SFT后可能出现的问题——灾难性遗忘（Catastrophic Forgetting）

1. 定义

灾难性遗忘是指模型在新任务训练中，遗忘了之前学到的知识。

2. 根源分析

- **参数更新**：微调时，模型参数朝着新任务的最优方向调整，可能偏离了预训练阶段的知识。
- **任务冲突**：新任务的目标可能与预训练任务不一致，导致模型需要“舍弃”部分旧知识。

3. 影响

- **通用能力下降**：模型可能无法再执行预训练阶段擅长的任务。
- **知识遗失**：丢失了在大规模数据上学习到的有用信息。

4. 解决方案

小红书账号：小黄搞AI

- **冻结部分参数**：在微调时，只更新部分参数，保留预训练的知识。
- **使用小学习率**：减缓参数更新的速度，防止过度偏离。
- **渐进式微调**：逐步适应新任务，减少遗忘。

52.4 SFT后可能出现的问题——模型偏差增加

1. 定义

模型偏差是指模型的预测结果偏向某些特定的模式或群体，导致不公平或不准确的结果。

2. 根源分析

- **数据偏差**：微调数据可能存在偏差，如代表性不足或标注不公平。
- **过度适应特定任务**：模型可能强化了特定任务的数据模式，忽略了其他可能性。

3. 影响

- **公平性问题**：模型可能对某些群体不公平。
- **可信度降低**：用户可能对模型的输出产生怀疑。

4. 解决方案

- **数据审查**：确保微调数据的多样性和公平性。
- **偏差检测**：在微调后，对模型进行偏差评估。
- **模型校正**：使用技术手段减轻模型偏差。

52.5 SFT后可能出现的问题——泛化能力下降

1. 定义

泛化能力是指模型在未见过的数据上保持良好性能的能力。

2. 根源分析

- **过度专注于特定任务**：微调可能让模型只擅长于微调任务的数据分布。
- **数据多样性不足**：微调数据集的覆盖范围有限。

3. 影响

- **适用范围缩小**：模型在其他任务或不同数据分布上表现不佳。
- **灵活性降低**：限制了模型的通用性。

4. 解决方案

- **多任务学习**：同时微调模型在多个相关任务上。
- **使用多样化的数据集**：增加微调数据的多样性。

小红书账号：小黄搞AI

52.5 SFT后可能出现的问题——鲁棒性降低

1. 定义

鲁棒性是指模型应对噪声、异常值或对抗样本的能力。

2. 根源分析

- **微调数据质量问题**：噪声数据或异常样本可能影响模型稳定性。
- **过拟合风险**：过度拟合导致模型对数据微小变化敏感。

3. 影响

- **易受攻击**：模型可能容易受到对抗样本的影响。
- **性能不稳定**：在实际应用中，模型可能出现异常行为。

4. 解决方案

- **数据清洗**：确保微调数据的质量。
- **对抗训练**：提高模型对对抗样本的抵抗力。
- **增加噪声训练**：在训练中加入噪声，提高模型的鲁棒性。

52.6 总结

监督微调（SFT）在提升模型特定任务性能的同时，可能引发一系列问题：

- **过拟合**：模型在训练数据上表现良好，但泛化能力不足。
- **灾难性遗忘**：模型遗忘了预训练阶段学到的知识。
- **模型偏差增加**：模型可能强化了数据中的偏差，导致不公平结果。
- **泛化能力下降**：模型在未见过的数据或任务上性能下降。
- **鲁棒性降低**：模型对噪声或对抗样本的抵抗力减弱。

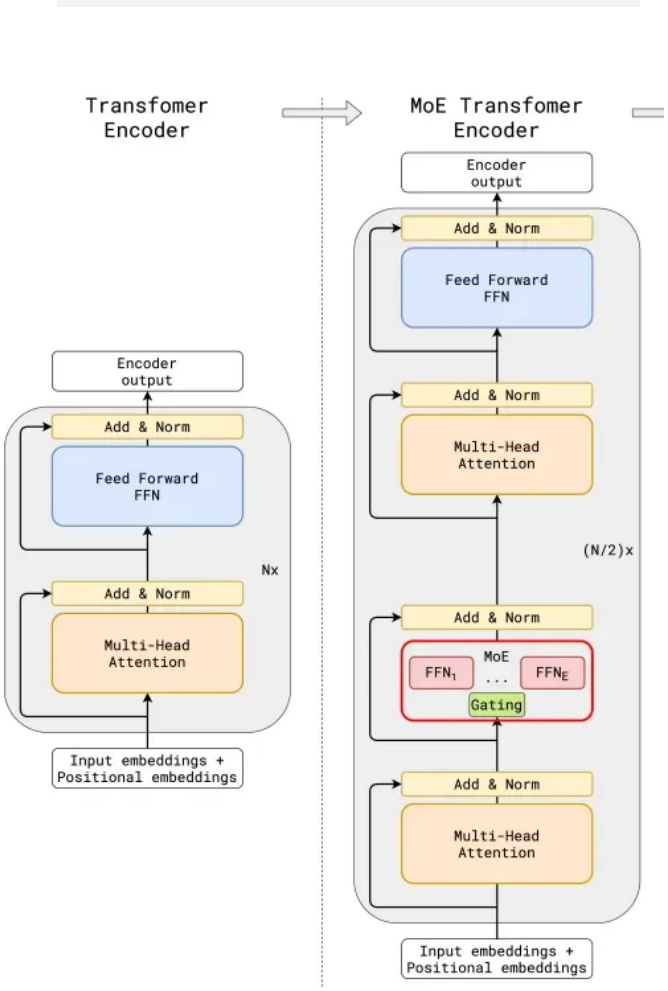
为解决这些问题，需要在微调过程中采取相应的策略，如数据增广、正则化、偏差检测、多任务学习等。

💖 **实践建议**

- **谨慎选择微调数据**：确保数据质量和多样性，减少偏差和过拟合风险。
- **合理设置训练参数**：调整学习率、批次大小等，防止灾难性遗忘和过拟合。
- **定期评估模型**：在微调过程中，持续监控模型在不同数据集上的性能。
- **采用先进技术**：如连续学习、对抗训练等，提高模型的通用性和鲁棒性。

53. 什么是MOE（混合专家模型）

小红书账号：小黄搞AI



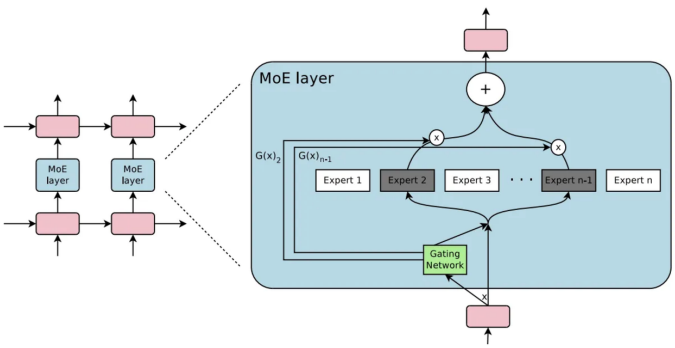
MoE，全称为Mixture of Experts（混合专家模型），是一种大模型架构，其核心设计思想是“术业有专攻”，即将任务分门别类，分配给多个“专家”来解决。与MoE相对应的是稠密（Dense）模型，后者是一个“通才”模型，能够处理多种任务，**而MoE模型则专注于“分而治之”，让每个专家模型解决它最擅长的部分。**

⬅左侧图为传统大模型架构，右图为MoE大模型架构。

两图对比可以看到，与传统大模型架构相比，MoE架构在数据流转过程中集成了一个**专家网络层（红框部分）**。

1. MoE的核心架构

MoE的核心设计在于在数据流转过程中加入了一个**专家网络层**，这一层由**门控网络（Gating Network）**和**一组专家模型（Experts）**构成。



整个工作流程如下：

1. **数据分割**：数据首先会被分割为多个Token。
2. **门控网络**：当每组数据进入专家网络层时，首先会经过门控网络。
3. **专家选择**：门控网络根据输入数据的特征，将数据分配给一个或多个专家模型。每个专家模型专注于处理一部分数据，“让专业的人做专业的事”。
4. **输出融合**：所有专家的输出结果会汇总，系统进行加权融合，得到最终的输出。

这种“术业有专攻”的设计能够在多个领域内提升模型的效率和性能。

2. MoE的详细设计步骤

小红书账号：小黄搞AI

MoE架构的核心思想是将复杂的问题分解成多个更小、更易管理的子问题，并由不同的专家网络分别处理。MoE的设计一般包括以下步骤：



3. MoE与Dense模型的对比

MoE模型与Dense模型在多个方面存在显著差异：

指标	MoE	Dense
模型结构	模型由多个专家组成，每次计算时只有一部分专家被激活，减少了计算量。	所有参数和激活单元都参与每一次前向和后向传播计算。
计算效率	**只激活部分专家，因此计算量和内存需求较少**，处理并发查询时具有更高的吞吐量。	计算量和内存需求随参数规模线性增长。
性能	在保持高效计算的同时，MoE模型的性能接近或优于同等规模的Dense模型。例如，50B的MoE模型可达到34B的Dense模型的性能。	性能稳定，但需要大量计算资源。
时延	只需加载部分激活专家，时延较低，特别是在并发性较低的场景下。	需要加载所有参数，时延较高。
应用场景	适用于需要高效处理并发查询的任务，如大规模在线服务。	适用于需要稳定性能且计算资源充足的任务。

4. MoE的优势

🎁 **更高的吞吐量**：由于激活单元较少，**MoE模型在处理大量并发查询时表现出更高的吞吐量。**

小红书账号：小黄搞AI

较低的时延：只需加载部分激活的专家模型，内存访问时间减少，从而**降低了时延**。

计算更高效：对于单次查询，MoE模型需要从内存中读取的参数更少，因此**在计算效率上优于Dense模型**。

性能优越：MoE模型能够在保持小型模型计算效率的同时，提供与大型Dense模型相近的性能。例如，Mistral MoE模型展示了如何在降低成本的同时提供相似性能。

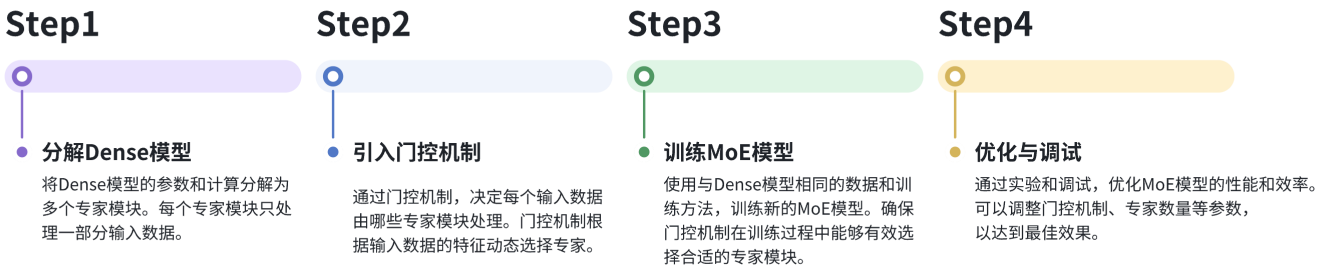
灵活性：可以将训练好的大型Dense模型转换为MoE模型，使其具有小型模型的高效计算和大型Dense模型的强大性能。

5. MoE的挑战

显存需求	训练稳定性与泛化性
由于**MoE需要将所有专家模型加载到内存中，对显存的压力较大**，且涉及复杂的算法和高昂的通信成本，这在资源受限设备上难以实现。	随着模型规模的扩大，**MoE面临着训练不稳定性、过拟合、以及如何确保模型泛化性和鲁棒性等问题**。这些问题需要开发者不断优化和提升。

6. MoE化

将Dense模型转换为MoE模型的过程称为**MoE化（MoEfication）**。其主要目的是保留Dense模型的强大性能，同时通过MoE的高效结构实现更高的计算效率。MoE化的关键步骤如下：



✏️ **MoE模型通过将复杂问题分解为多个子问题，并利用专家网络来分别处理这些子问题**，从而在提高计算效率的同时保持性能优越性。这使得MoE成为了近年来大模型架构中的热门方向，特别是在应对计算资源有限的场景时表现出色。

54. Dense和MOE模型的区别

Dense模型（密集模型）和MOE模型（Mixture of Experts，专家混合模型）在架构和计算效率上有显著的区别。以下是两者的专业比较：

1. 模型架构

Dense模型	MOE模型

小红书账号：小黄搞AI

- 在Dense模型中，所有的参数在每个前向传播步骤中都会被激活和更新。**所有的神经元都对输入数据产生反应，并参与计算。**这种模型通常由多个层次构成，每层包含固定数量的神经元。
- Dense模型在每个训练步骤中都使用所有的参数，导致其计算开销相对较大。

- MOE模型通过引入多个专家（神经网络子模型）来优化计算效率。**每个输入样本只会激活部分专家，通常是通过一个门控机制（Gating Network）来选择最相关的专家进行计算。**
- MOE模型的特点是稀疏性，意味着在每个前向传播中只使用一小部分的参数，从而减少了计算负担。

2. 计算效率

Dense模型

- Dense模型的计算成本随模型规模的增加而增加，特别是在大规模应用中，计算效率可能受到影响。
- 因为**所有参数都被激活，所以在推理和训练时需要处理大量的计算，导致较高的延迟和能耗**。

MOE模型

- MOE模型通过选择少量的专家进行计算，显著提高了计算效率。只激活部分参数意味着减少了所需的计算量，这在处理大规模输入时尤其有效。
- MOE模型**可以在保持或提升模型性能的同时，减少计算资源的消耗**。

3. 应用场景

Dense模型

- 适用于需要全模型输出和全参数激活的任务，例如许多传统的深度学习任务。
- 在模型较小或计算资源丰富的情况下，Dense模型是一个简单有效的选择。

MOE模型

- 适用于需要处理大规模数据集或高维输入的任务。它**特别适合于需要高效并行计算的应用场景**，如自然语言处理和计算机视觉。
- MOE模型能够通过动态选择最相关的专家来提高模型的灵活性和适应性。

4. 训练复杂性

Dense模型

- 训练过程相对简单，**因为所有参数都同时参与更新，优化算法的应用也较为直接**。

MOE模型

- 训练过程较复杂，需要设计有效的门控机制，以确保所选的专家能够为输入样本提供有用的信息。此外，**门控机制的训练可能需要更多的超参数调优**。

5. 总结

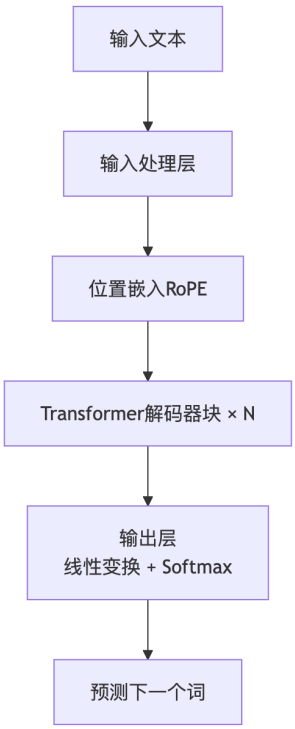
小红书账号：小黄搞AI

📌 Dense模型和MOE模型各有优缺点。Dense模型在实现上更简单，适合小规模应用，而MOE模型在处理大规模数据时效率更高，能够更好地利用计算资源。选择哪种模型取决于具体的应用需求、数据规模和可用的计算资源。

55. 介绍一下Qwen通义千问的结构

🐳 通义千问（Qwen）是阿里巴巴集团研发的一系列大型语言模型，旨在提供强大的自然语言理解和生成能力。该模型系列包括不同参数规模的模型，涵盖基础预训练模型和经过人类对齐技术微调的聊天模型（Qwen-Chat），以及专门用于代码和数学任务的模型（Code-Qwen、Math-Qwen）。

55.1 整体架构概览



Qwen模型的整体结构

通义千问（Qwen）采用了基于Transformer的架构，这是当前大型语言模型的主流选择。其核心组成部分包括：

1. **输入处理层**：对输入文本进行预处理和编码。
2. **嵌入层（Embedding Layer）**：将词或子词转换为连续的向量表示。
3. **多层Transformer解码器块**：由多头自注意力机制和前馈神经网络组成，多个层堆叠以捕捉深层次的语言特征。
4. **输出层**：生成最终的输出，如预测下一个词的概率分布。

55.2 模型结构详解

1. 输入处理层

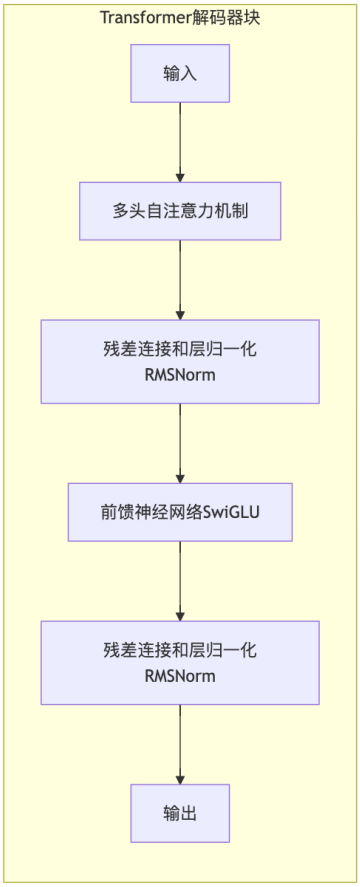
- **分词与编码**：采用字节对编码（Byte Pair Encoding，BPE）方法，将输入文本进行分词和编码。为了增强多语言能力，Qwen的词汇表增加了常用的中文字符和其他语言的词汇，词汇表大小约为152K。
- **压缩效率**：在多种语言中，Qwen的压缩效率高于其他标记器，这意味着服务成本可以显著降低。

小红书账号：小黄搞AI

2. 嵌入层（Embedding Layer）

- **词嵌入（Word Embedding）**：将每个词或子词的索引映射为稠密的向量表示，捕捉词语的语义信息。
- **位置嵌入（Positional Embedding）**：采用**旋转位置嵌入（RoPE）**，将位置信息融入模型中。RoPE通过在输入中加入位置信息，使模型能够识别序列中词语的位置。

3. 多层Transformer解码器块



Transformer解码器块的内部结构

Qwen的核心由多层Transformer解码器块组成，每个块包含以下组件：

a. 多头自注意力机制（Multi-Head Self-Attention）

- **目的**：捕捉序列中不同位置之间的依赖关系。
- **机制**：通过计算查询（Q）、键（K）、值（V）之间的相关性，生成加权的输出表示。

b. 前馈神经网络（Feed-Forward Network, FFN）

- **结构**：由两个线性变换和一个非线性激活函数组成。
- **激活函数**：采用**SwiGLU激活函数**，这是Swish和门控线性单元（GLU）的组合，增强了模型的表达能力。

c. 残差连接和层归一化

- **残差连接（Residual Connection）**：缓解深层网络的梯度消失问题，促进信息的直接传递。
- **层归一化（Layer Normalization）**：采用**RMSNorm**，提升训练的稳定性和效率。

4. 输出层

线性变换和Softmax层：将Transformer块的输出转换为词汇表中每个词的概率分布，用于预测下一个词。

55.3 模型训练方法

1. 预训练

- **数据规模**：使用了多达**3万亿个标记**的数据，包括公共网络文档、百科全书、书籍、代码等，涵盖多种类型和领域。
- **多语言支持**：数据集是多语言的，其中相当一部分是中文和英文。

小红书账号：小黄搞AI

- **数据预处理**：使用了全面的数据预处理程序，包括去重、语言识别、质量过滤等，确保数据的多样性和高质量。
- **目标**：在大规模未标注的文本数据上训练模型，让其学习语言的基本结构和模式。
- **方法**：采用自回归语言建模任务，即给定前面的词，预测下一个词。
- **优化器**：使用AdamW优化器，设置了适当的超参数以保证训练稳定性。

2. 上下文长度扩展

- **挑战**：随着上下文长度的增加，计算和内存成本急剧增加。
- **解决方案**：
  - **NTK-aware插值**：调整RoPE的位置编码，防止在不训练的情况下丢失高频信息。
  - **LogN-Scaling**：调整注意力机制的缩放因子，保持注意力值的稳定性。
  - **窗口注意力**：将注意力限制在有限的上下文窗口内，防止模型注意到过远的标记。
  - **分层窗口分配**：为每一层分配不同的窗口大小，低层使用较短的窗口，高层使用较长的窗口，提升长上下文建模能力。

3. 微调和对齐

- **监督微调（SFT）**：
  - **目标**：使预训练模型适应对话式AI助手的任务。
  - **数据**：使用了精心标注的对话数据，包括任务执行、聊天、工具使用、安全等方面。
  - **方法**：在预训练模型的基础上，使用标注的对话数据，对模型进行微调。
- **人类反馈强化学习（RLHF）**：
  - **目标**：进一步使模型与人类偏好保持一致，生成更符合人类期望的响应。
  - **方法**：
    - **奖励模型训练**：收集人类反馈的数据，训练一个奖励模型来评估模型输出的质量。
    - **策略优化**：使用近端策略优化（PPO），根据奖励模型的评估结果，优化模型的策略。

55.4 Qwen的特殊特性和创新

1. 多语言词汇表

- **优化的Tokenization**：采用了更大的词汇表，包含152K个词元，在中文等多种语言上具有更好的压缩效率，降低了服务成本。

2. 改进的模型架构

- **非绑定的嵌入方法**：输入嵌入和输出投影的权重不共享，提升了模型的性能。
- **偏置调整**：移除了大多数层的偏置，但在注意力的QKV层中添加了偏置，增强了模型的外推能力。

小红书账号：小黄搞AI

3. 激活函数的选择

- **SwiGLU激活函数**：相比于GeLU，SwiGLU在提高模型性能和效率方面表现更好。

4. 长上下文建模

- **上下文长度扩展**：通过NTK-aware插值、LogN-Scaling、窗口注意力等技术，扩展了模型的上下文长度，增强了对长文本的处理能力。

55.5 专用模型

1. Code-Qwen

- **目标**：专门用于编码任务的模型，包括代码生成、调试和解释。
- **训练方法**：
  - **继续预训练**：在基础模型的基础上，继续在大量代码数据上进行预训练。
  - **代码监督微调**：使用与代码生成、调试和解释相关的对话数据，对模型进行微调。
- **性能**：在HumanEval、MBPP等代码基准数据集上表现出色，超过了许多开源模型。

2. Math-Qwen

- **目标**：专门用于解决数学问题的模型，提升模型在算术和数学推理方面的能力。
- **训练方法**：
  - **数学SFT**：在增强的数学指导数据集上进行监督微调，直接得到聊天模型Math-Qwen-Chat。
- **性能**：在GSM8K、MATH等数学基准数据集上表现出色，超过了相同规模的开源模型，接近于GPT-3.5的性能。

55.6 模型评估与结果

1. 基准评估

- **任务**：在MMLU、C-Eval、GSM8K、HumanEval、BBH等多个基准测试中评估模型性能。
- **结果**：
  - **Qwen-14B**：在多个下游任务中表现出色，甚至超过了一些更大参数规模的模型，如LLaMA2-70B。
  - **Qwen-7B**：表现优于同等规模的模型，如LLaMA2-13B。
  - **Code-Qwen**：在代码生成任务上，表现优于同等规模的专有和开源模型。
  - **Math-Qwen**：在数学推理任务上，超过了相同规模的开源模型，接近于GPT-3.5的性能。

2. 上下文长度扩展评估

- **指标**：在arXiv数据集上以困惑度（PPL）为指标进行评估。

小红书账号：小黄搞AI

- **结果**：证明了上下文长度扩展技术的有效性，在超过8192个标记的情况下，模型性能仍然稳定。

55.7 总结

通义千问（Qwen）系列模型通过大规模的预训练和精心设计的架构，展现了强大的自然语言理解和生成能力。其创新之处包括：

- **多语言支持**：优化的词汇表和编码方法，提升了模型在多种语言上的表现。
- **改进的架构设计**：采用了SwiGLU激活函数、RoPE位置嵌入、非绑定的嵌入方法等，提升了模型性能。
- **上下文长度扩展**：通过多种技术手段，扩展了模型的上下文长度，增强了对长文本的处理能力。
- **专用模型**：针对代码和数学任务，开发了专门的模型（Code-Qwen、Math-Qwen），在相应领域取得了优异的性能。

> 🐳 通过结合预训练、监督微调和人类反馈强化学习等方法，Qwen模型在各项基准测试中表现出色，展示了其在自然语言处理领域的强大潜力。

56. Qwen2 有哪些提升？

> 🍊 Qwen2 相较于 Qwen1.5，有了重大升级和改进，下面将从多个方面详细解析这些提升，并逐个解释相关概念。

56.1 模型尺寸的扩展

1. 多种模型尺寸

Qwen2 系列包含了五个不同尺寸的预训练和指令微调模型：

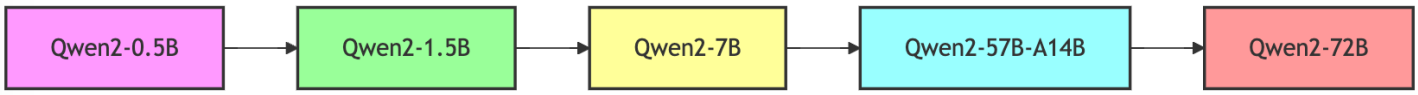
- **Qwen2-0.5B**：约 5 亿参数
- **Qwen2-1.5B**：约 15 亿参数
- **Qwen2-7B**：约 70 亿参数
- **Qwen2-57B-A14B**：约 574 亿参数（激活参数为 140 亿）
- **Qwen2-72B**：约 727 亿参数

为什么模型尺寸多样化很重要？

- **适应不同的应用场景**：小模型适用于资源受限的环境，如移动设备；大模型则在处理复杂任务时表现更佳。
- **灵活性**：开发者可以根据需求选择合适的模型，平衡性能和资源消耗。

小红书账号：小黄搞AI

2. 模型尺寸对比图



58.2 全面采用 GQA 技术

1. 什么是 GQA（Grouped Query Attention）？

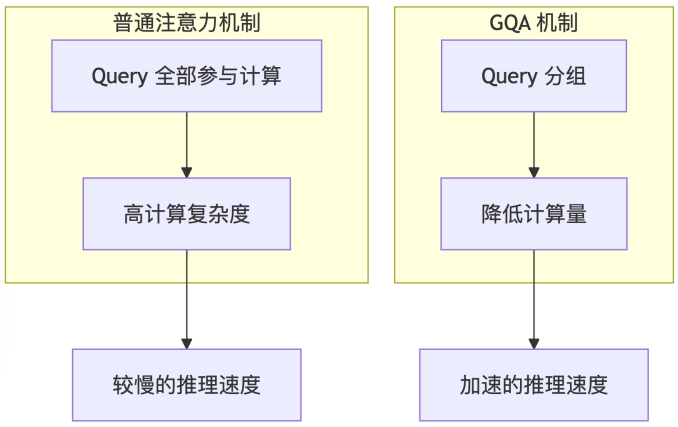
GQA，即**分组查询注意力机制**，是一种改进的注意力机制。它将注意力计算中的查询（Query）部分进行分组，从而减少计算量。

GQA 的优势

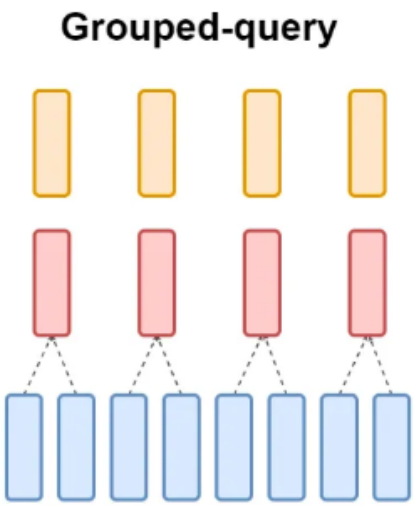
- **推理加速**：通过减少计算复杂度，加快模型的推理速度。
- **显存占用降低**：降低显存需求，使得在同等硬件条件下，可以运行更大的模型或处理更长的序列。

2. Qwen2 全系采用 GQA

在 Qwen1.5 中，只有 32B 和 110B 模型使用了 GQA。而在 Qwen2 中，所有尺寸的模型都采用了 GQA 技术。



GQA 工作原理图



58.3 上下文长度的提升

1. 上下文长度的重要性

上下文长度（Context Length）指模型在一次推理中能够处理的最大文本长度。

- **支持长文本处理**：更长的上下文长度可以处理更长的输入，例如长文档、代码文件等。
- **提高模型的连贯性**：在长对话中，模型可以更好地保持上下文，生成更相关的回复。

2. Qwen2 的上下文长度扩展

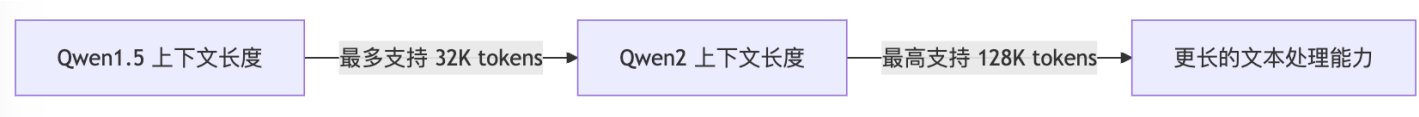
小红书账号：小黄搞AI

- **预训练模型**：在 32K tokens 的数据上训练，并在 128K tokens 时仍有良好表现。
- **指令微调模型**：最高支持到 128K tokens 的上下文长度。

3. 技术实现

- **YARN（Yet Another Rope Extension）**：一种扩展上下文长度的方法，通过调整位置编码实现。
- **Dual Chunk Attention**：将长序列分块处理，降低计算复杂度。

4. 上下文长度对比图



58.4 多语言能力的增强

1. 新增 27 种语言的数据

除了中文和英语，Qwen2 还增加了对 27 种语言的高质量训练数据。

覆盖的语言区域

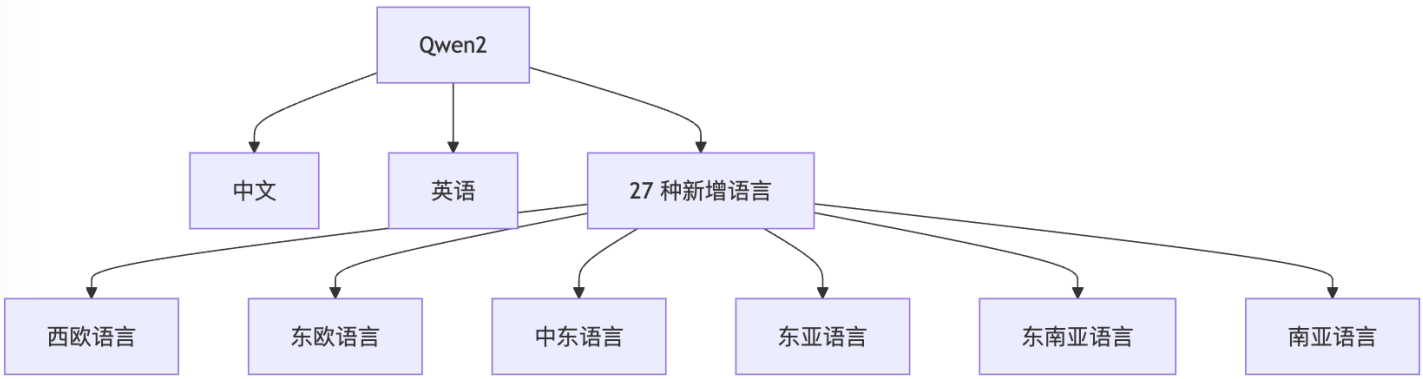
- **西欧**：德语、法语、西班牙语等
- **东欧及中欧**：俄语、捷克语、波兰语
- **中东**：阿拉伯语、希伯来语等
- **东亚**：日语、韩语
- **东南亚**：越南语、泰语等
- **南亚**：印地语、孟加拉语等

2. 解决语言转换（Code Switch）问题

- **问题描述**：在多语言场景中，模型可能会在不恰当的情况下切换语言。
- **改进措施**：通过优化训练数据和模型，使得语言转换的概率大幅降低。

3. 多语言能力示意图

小红书账号：小黄搞AI

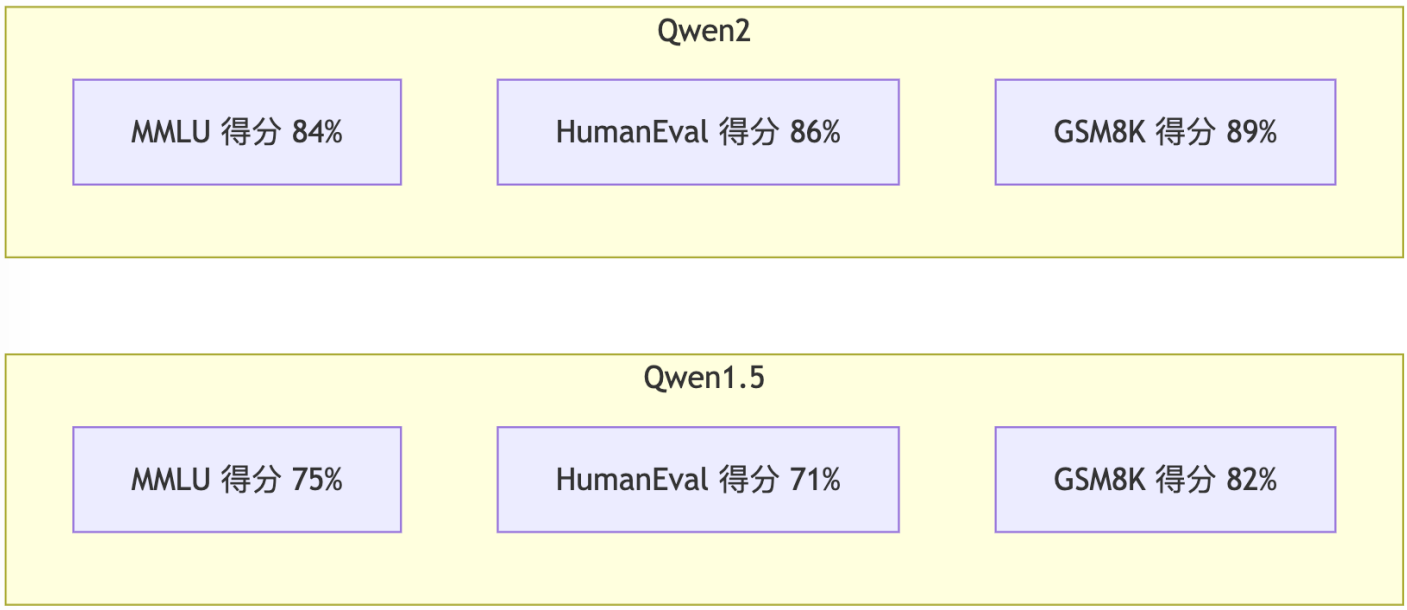


58.5 在多项基准测试中领先

1. 性能提升

- **自然语言理解**：在 MMLU、C-Eval 等数据集上表现优异。
- **代码能力**：在 HumanEval、MBPP 等代码评测中取得高分。
- **数学能力**：在 GSM8K、MATH 等数学评测中有显著提升。
- **多语言理解**：在多语言评测中超越了开源的同类模型。

2. 性能对比图



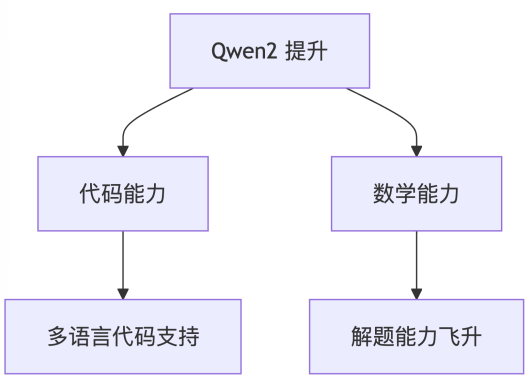
58.6 代码和数学能力的显著提升

1. 代码能力

- **融合 CodeQwen1.5 的成功经验**：将之前版本的代码处理优势融入 Qwen2。
- **多种编程语言支持**：在 Python、Java、C++ 等多种语言上表现出色。

2. 数学能力

小红书账号：小黄搞AI



代码和数学能力提升示意图

- **高质量数据训练**：

  使用大规模高质量的数学数据进行训练。

- **解题能力提升**：

  在数学问题求解上有了飞跃性的进步。

58.7 长文本处理能力

1. 支持长文本的意义

- **处理大规模文档**：如长篇论文、书籍等。
- **上下文关联性**：在长对话或文本生成中保持上下文一致。

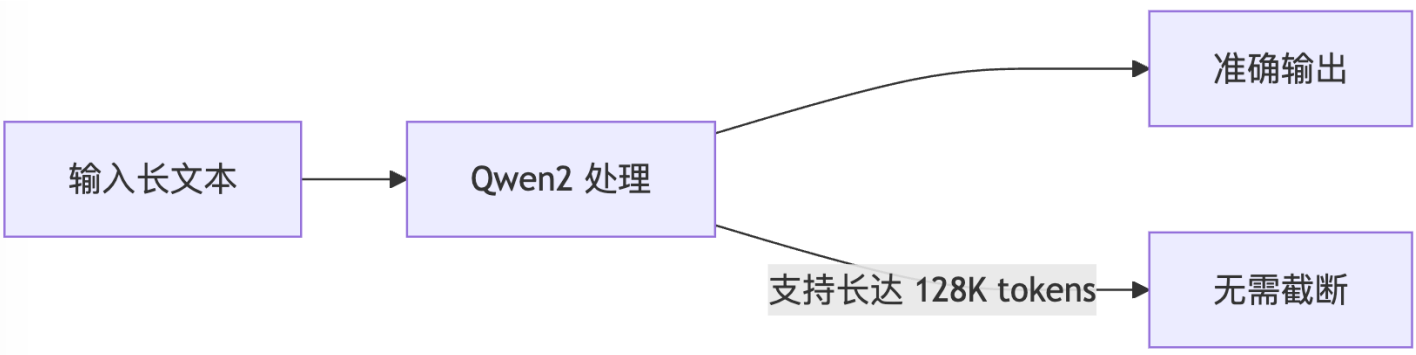
2. 技术实现

- **YARN 或 Dual Chunk Attention**：扩展上下文长度的技术，实现高达 128K tokens 的支持。

3. 实验结果

- **Needle in a Haystack 测试集**：Qwen2-72B-Instruct 能完美处理 128K 上下文长度内的信息抽取任务。

4. 长文本处理示意图



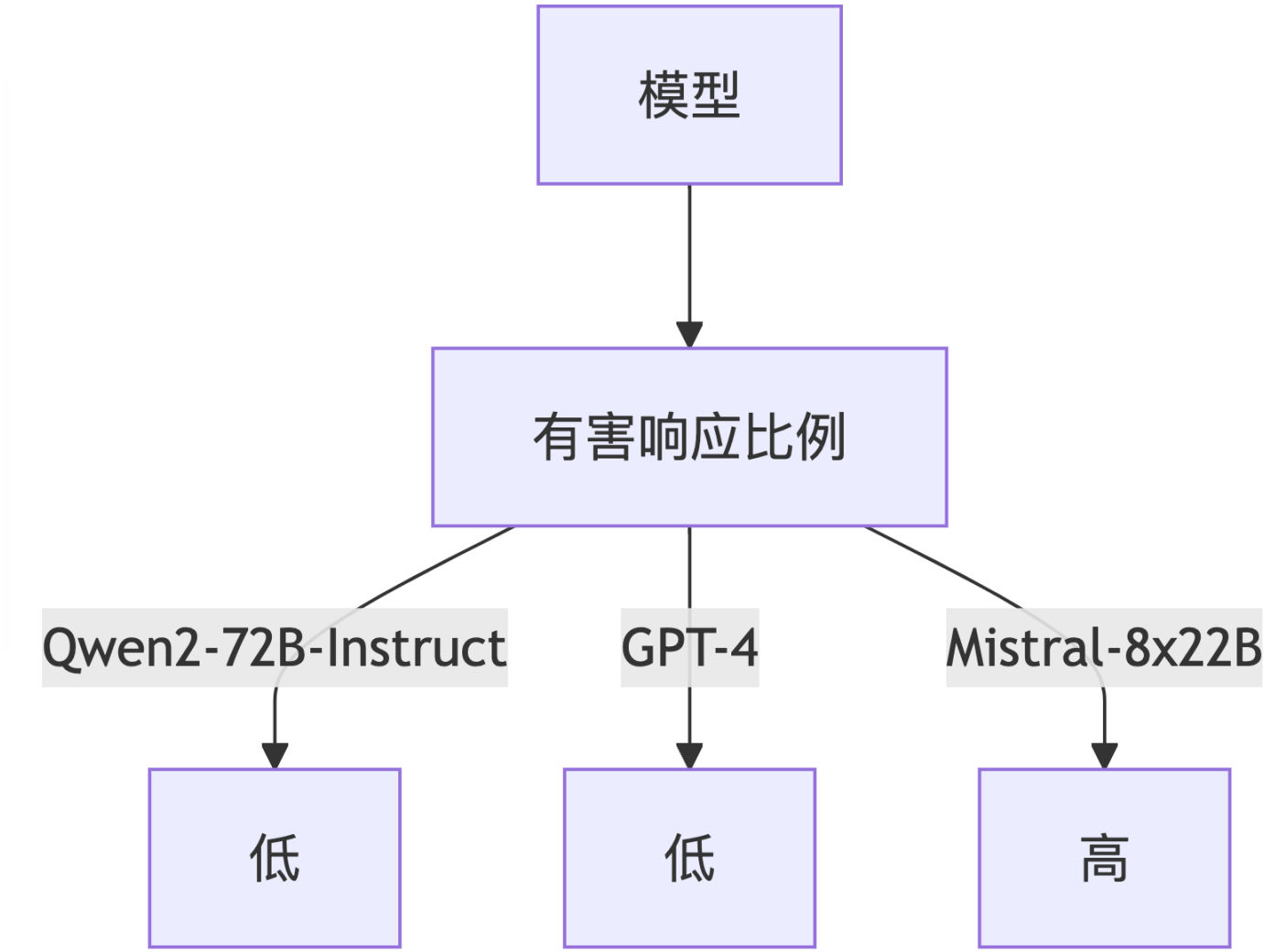
58.8 安全性提升

1. 多语言安全性

- **不安全查询类别**：非法活动、欺诈、色情、隐私暴力。
- **测试结果**：Qwen2-72B-Instruct 在安全性上与 GPT-4 表现相当，显著优于 Mistral-8x22B。

2. 安全性对比图

小红书账号：小黄搞AI



58.9 模型许可的开放

1. 许可变化

- **Qwen2-72B**：继续使用 Qianwen License。
- **其他模型**：Qwen2-0.5B、Qwen2-1.5B、Qwen2-7B、Qwen2-57B-A14B 采用 Apache 2.0 许可。

2. 开放程度提升的意义

- **加速落地和商业应用**：更宽松的许可使得模型可以被更广泛地使用和商业化。
- **促进社区合作**：开放的许可鼓励开发者贡献和改进模型。

58.10 未来计划

1. 更大的模型和多模态扩展

- **训练更大的模型**：继续探索模型和数据的扩展规律。
- **多模态模型**：将视觉和语音理解融入 Qwen2，扩展其能力。

2. 持续开源新模型

- **社区期待**：在不久的将来，继续开源新的模型版本，满足更多需求。

小红书账号：小黄搞AI

🍊 Qwen2 在模型尺寸、多语言支持、上下文长度、代码和数学能力、安全性等多个方面都有了显著的提升。这些改进不仅增强了模型的性能，也扩大了其应用范围。通过采用更开放的许可，Qwen2 将在全球范围内加速落地和商业应用。

57. 介绍一下Llama3.1的创新

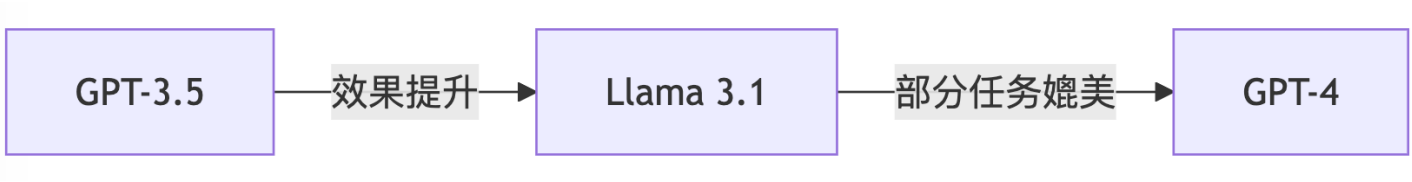
🍊 Llama 3.1 是开源社区在大模型领域的集大成者，代表了当前最成熟、最先进的大模型训练技术。它在多个方面实现了创新和突破，下面将逐个概念进行详细解释。

57.1 性能方面的重大提升

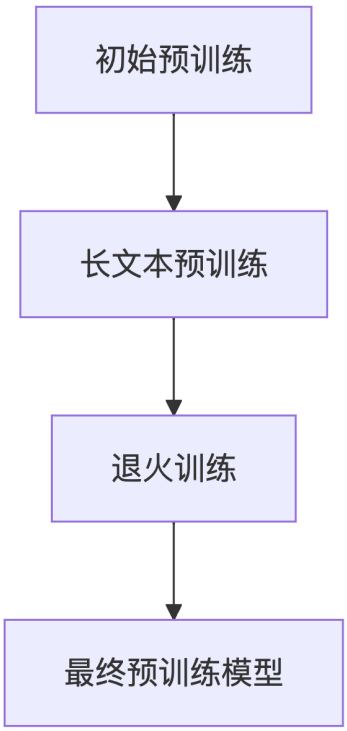
1. 超越同规模模型和 GPT-3.5

- **效果超群**：Llama 3.1 在各个榜单上均超过了 GPT-3.5，在部分任务上甚至可以与 GPT-4 相媲美。
- **使用体验**：虽然在使用体验上略逊于 GPT-4，但稳定性和性能均领先于 GPT-3.5。

2. 性能提升示意图



59.2 预训练方法论的创新：三阶段预训练



三阶段预训练流程图

1. 初始预训练

- **目标**：让模型学习基本的语言知识和结构。
- **方法**：使用短文本进行大规模训练，保证训练效率。

2. 长文本预训练

- **目标**：让模型具备处理长文本的能力。
- **方法**：在初始预训练的基础上，使用长文本继续训练。

3. 退火训练

- **目标**：细化模型能力，提高模型在后续微调阶段的表现。

小红书账号：小黄搞AI

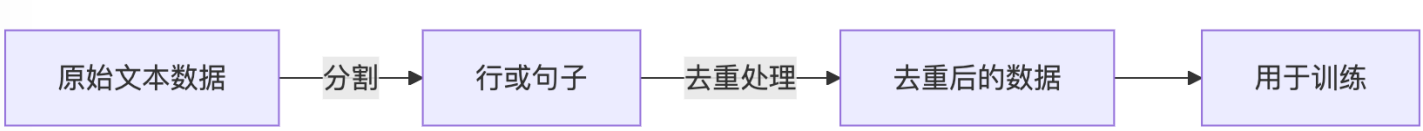
- **方法**：使用高质量的数据，以较小的学习率进行训练。

59.3 数据处理的创新：行级别的去重

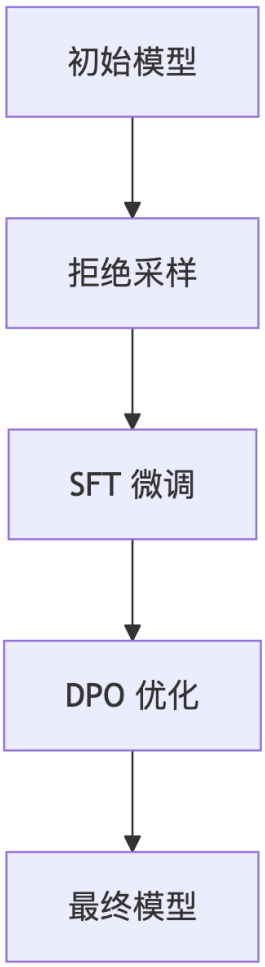
1. 什么是行级别去重？

- **定义**：将文本分割成更小的单元（如行、句子），对这些小单元进行去重处理。
- **优势**：
  - **提升数据质量**：去除重复信息，减少模型过拟合。
  - **提高模型性能**：让模型学习更丰富、多样的内容。

2. 行级别去重的示意图



59.4 后训练方法论的创新：拒绝采样、SFT、DPO 的训练策略



训练流程图

1. 拒绝采样（Rejection Sampling）

- **定义**：从模型生成的多个候选答案中，选择质量较高的样本用于训练。
- **作用**：提升训练数据的质量，使模型更快收敛。

2. SFT（有监督微调）

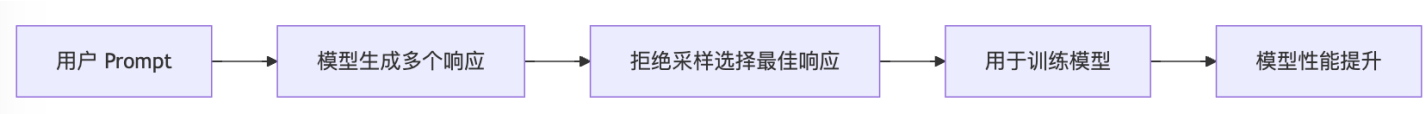
- **定义**：使用高质量的监督数据对模型进行微调。
- **作用**：让模型学会遵循指令，提高回答的准确性和相关性。

3. DPO（直接偏好优化）

- **定义**：利用用户偏好数据，直接优化模型的输出，使其更符合用户期望。
- **作用**：提高模型的用户满意度和实用性。

小红书账号：小黄搞AI

59.5 有效利用用户行为数据



数据利用示意图

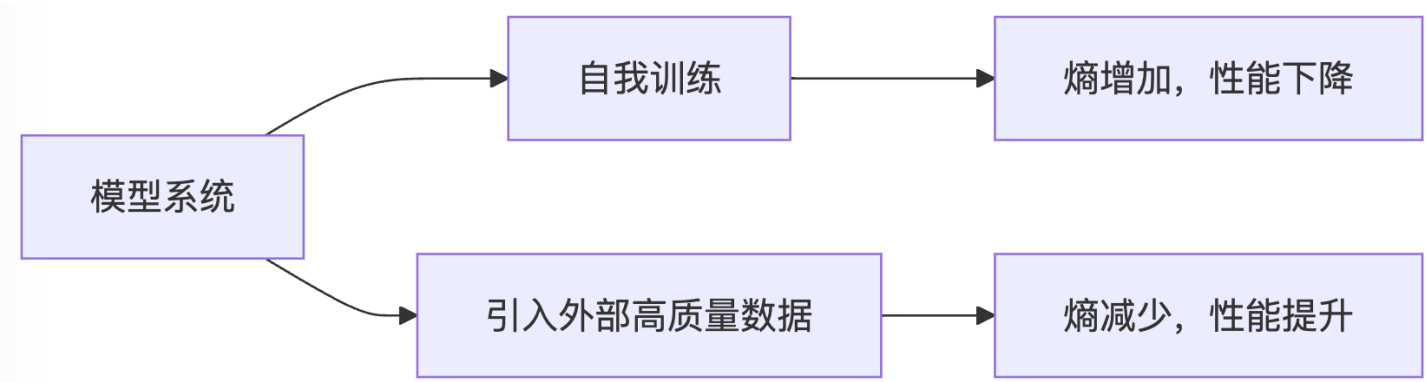
1. 用户行为数据的重要性

- **高质量的 Prompt**：用户在使用模型时，提供了大量高质量的提示词（Prompt）。
- **反馈闭环**：利用用户的反馈，可以持续优化模型性能。

2. 利用方法

- **拒绝采样**：从用户提供的 Prompt 中，选择最优的响应进行训练。
- **DPO**：根据用户的偏好，直接调整模型的输出。

59.6 引入负熵原则



负熵原则示意图

1. 什么是负熵原则?

- **定义**：从外部系统获取有序的信息（负熵），以减少系统的熵，提升系统的有序性。
- **在模型中的体现**：通过引入高质量的数据和反馈，持续优化模型，使其朝着更好的方向发展。

2. 负熵原则的作用

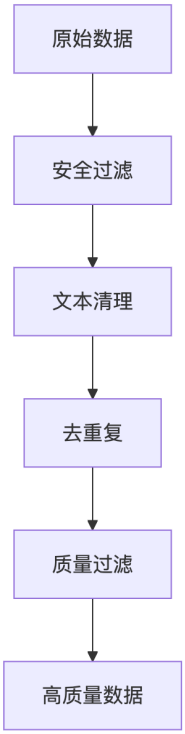
- **防止模型退化**：避免模型在自我生成数据的过程中走向偏差。
- **持续提升性能**：通过外部信息的引入，不断提高模型的能力。

59.7 预训练去重和数据质量提升

1. 数据质量的挑战

- **数据量巨大**：需要处理海量的互联网数据。
- **数据质量参差不齐**：存在重复、低质量、甚至有害的信息。

小红书账号：小黄搞AI



数据处理流程图

2. 数据质量处理方法

- **安全过滤**：移除包含个人信息、不安全内容、成人内容等。
- **文本清理**：提取有用的文本信息，去除杂乱内容。
- **去重复**：在网页级别、文本级别、行级别进行去重。
- **质量过滤**：使用规则和模型，过滤掉低质量的数据。

59.8 使用 Scaling Law 确定训练参数



Scaling Law 示意图

1. 什么是 Scaling Law?

- **定义**：一种基于实验总结的经验规律，指导模型规模、数据量和计算量的合理配置。
- **作用**：在有限的计算资源下，找到最优的模型规模和数据量组合。

2. 应用在 Llama 3.1 中

- **实验方法**：在小规模数据上进行实验，拟合出模型性能与参数的关系曲线。
- **推导结果**：确定在给定计算量下，模型规模和数据量的最佳配置。

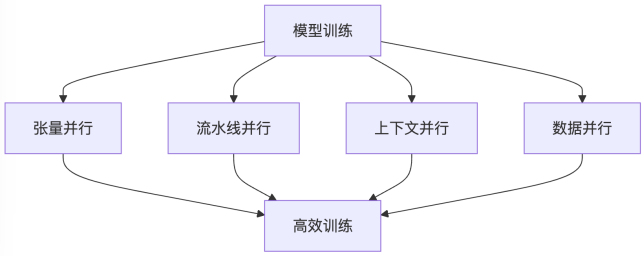
59.9 4D 并行训练模型

1. 训练大型模型的挑战

- **计算资源**：需要大量的 GPU 和存储资源。
- **并行化策略**：如何高效地利用多机多卡进行训练。

2. 4D 并行策略

小红书账号：小黄搞AI



4D 并行示意图

- **张量并行（TP）**：切分模型的权重张量。
- **流水线并行（PP）**：将模型的层级切分，进行流水线式训练。
- **上下文并行（CP）**：在长序列上进行并行。
- **数据并行（DP）**：在多个 GPU 上并行处理不同的数据。

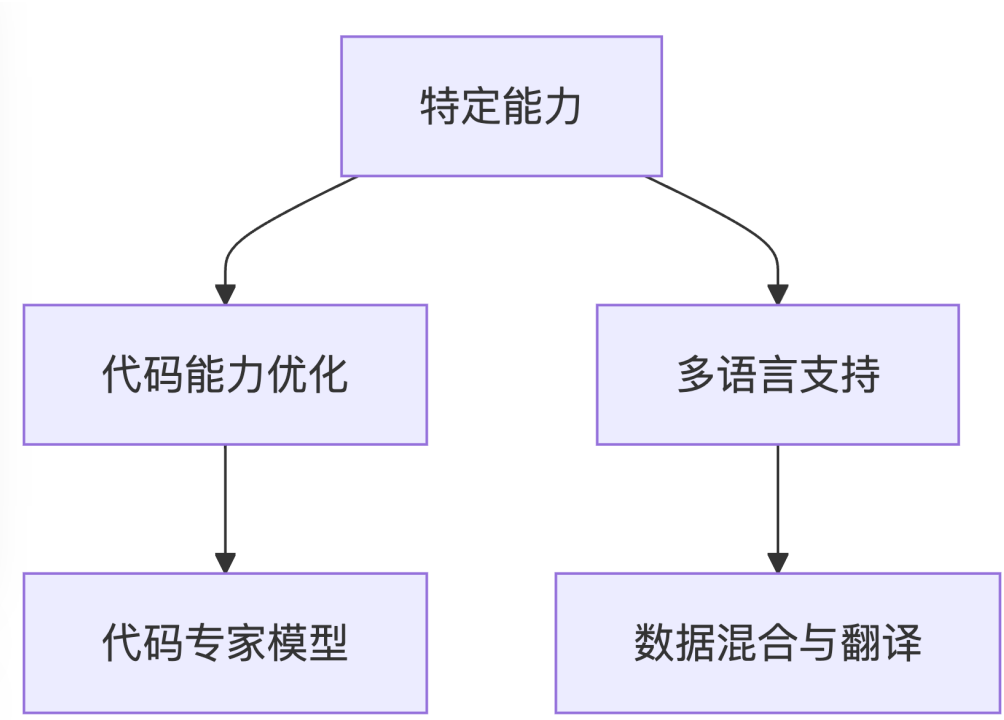
59.10 针对特定能力的优化

1. 代码能力

- **挑战**：代码需要专业背景，人工标注成本高。
- **解决方案**：
  - **代码专家模型**：专门预训练一个代码模型。
  - **合成数据**：利用模型生成代码数据，进行质量过滤。

2. 多语言支持

- **挑战**：不同语言的数据量和质量不一致。
- **解决方案**：
  - **数据混合比例**：合理配置不同语言的数据比例。
  - **翻译和反向翻译**：生成小语种的数据。



特定能力优化示意图

小红书账号：小黄搞AI

📌 Llama 3.1 通过在预训练和后训练阶段的多项创新，实现了性能和能力的全面提升。这些创新包括：

1. **性能大幅提升**：超越同规模模型，部分任务媲美 GPT-4。
2. **三阶段预训练**：初始预训练、长文本预训练、退火训练。
3. **行级别去重**：提高数据质量，提升模型性能。
4. **拒绝采样、SFT、DPO 训练策略**：有效利用合成数据和用户数据，提升模型能力。
5. **负熵原则**：引入外部高质量数据，持续优化模型。
6. **数据质量处理**：多层次的数据过滤和清理。
7. **Scaling Law 应用**：合理配置模型规模和数据量。
8. **4D 并行训练**：高效利用计算资源，训练超大规模模型。
9. **特定能力优化**：针对代码、多语言等领域进行专项优化。

58. 手撕自注意力机制（用Pytorch）（Self Attention）

自注意力（Self-Attention）机制是Transformer的核心机制之一。它允许模型在处理某个序列中的一个元素时，参考序列中其他元素的表示。这种机制通过计算查询（Query）和键（Key）的相似性，来加权求和值（Value）。输出就是这些值的加权和，其中的权重是查询和键的相似性分数。

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

Query（查询）	Key（键）	Value（值）
代表模型要对输入序列中的哪些部分进行注意	每个输入元素都有对应的键，表示它在整个序列中的表示	每个输入元素对应的值，最终会通过注意力权重加权求

```python
from math import sqrt
import torch
import torch.nn as nn

class SelfAttention(nn.Module):
    def __init__(self, dim_q, dim_k, dim_v):
        super(SelfAttention, self).__init__()
        self.dim_q = dim_q
        self.dim_k = dim_k
        self.dim_v = dim_v
        self.linear_q = nn.Linear(dim_q, dim_k, bias=False)
        self.linear_k = nn.Linear(dim_q, dim_k, bias=False)
        self.linear_v = nn.Linear(dim_q, dim_v, bias=False)
        self._norm_fact = 1 / sqrt(dim_k)
```

小红书账号：小黄搞AI

```
15
16      def forward(self, x):
17          batch, n, dim_q = x.shape
18          assert dim_q == self.dim_q
19          q = self.linear_q(x)  # batch, n, dim_k
20          k = self.linear_k(x)  # batch, n, dim_k
21          v = self.linear_v(x)  # batch, n, dim_v
22          dist = torch.bmm(q, k.transpose(1, 2)) * self._norm_fact  # batch, n, n
23          dist = torch.softmax(dist, dim=-1)  # batch, n, n
24          att = torch.bmm(dist, v)
25          return att
```

逐行代码解释

`class SelfAttention(nn.Module)`

```
1 #定义了一个继承自PyTorch的nn.Module类的自注意力机制模块。
2 class SelfAttention(nn.Module):
```

`def __init__(self, dim_q, dim_k, dim_v)`

```
1      def __init__(self, dim_q, dim_k, dim_v):
2          super(SelfAttention, self).__init__()
3          self.dim_q = dim_q
4          self.dim_k = dim_k
5          self.dim_v = dim_v
6          self.linear_q = nn.Linear(dim_q, dim_k, bias=False)
7          self.linear_k = nn.Linear(dim_q, dim_k, bias=False)
8          self.linear_v = nn.Linear(dim_q, dim_v, bias=False)
9          self._norm_fact = 1 / sqrt(dim_k)
```

- 这里 `dim_q`、`dim_k` 和 `dim_v` 分别代表查询、键和值的向量维度。
- `linear_q`、`linear_k` 和 `linear_v` 是线性变换（全连接层），将输入转化为查询、键和值的向量表示。
  - `linear_q`：对输入进行线性变换，得到查询向量。
  - `linear_k`：对输入进行线性变换，得到键向量。
  - `linear_v`：对输入进行线性变换，得到值向量。
- `_norm_fact = 1 / sqrt(dim_k)`：这是对注意力得分进行归一化的因子，用来避免数值不稳定性，防止内积结果过大。

小红书账号：小黄搞AI

`def forward(self, x)`

```
    def forward(self, x):
        batch, n, dim_q = x.shape
        assert dim_q == self.dim_q
```

- `forward` 函数定义了模型的前向传播逻辑。
- 输入 `x` 的形状是 `(batch_size, sequence_length, dim_q)`，即 `x` 包含了一个批次的序列数据。
  - `batch`：表示批次大小。
  - `n`：表示序列长度。
  - `dim_q`：表示输入的特征维度。
- `assert` 语句确保输入的维度与预定义的查询维度一致。

q、k、v计算

```
        q = self.linear_q(x)  # batch, n, dim_k
        k = self.linear_k(x)  # batch, n, dim_k
        v = self.linear_v(x)  # batch, n, dim_v
```

- `q = self.linear_q(x)`：对输入 `x` 进行线性变换，得到查询矩阵 `q`。
- `k = self.linear_k(x)`：对输入进行线性变换，得到键矩阵 `k`。
- `v = self.linear_v(x)`：对输入进行线性变换，得到值矩阵 `v`。
- 变换后，`q` 和 `k` 的形状为 `(batch_size, sequence_length, dim_k)`，而 `v` 的形状为 `(batch_size, sequence_length, dim_v)`。

点积相似度计算

```
        dist = torch.bmm(q, k.transpose(1, 2)) * self._norm_fact  # batch, n, n
```

- `torch.bmm(q, k.transpose(1, 2))`：使用批次矩阵乘法 `bmm` 计算查询 `q` 与键 `k` 的点积相似度。`k.transpose(1, 2)` 将 `k` 的形状从 `(batch, n, dim_k)` 转置为 `(batch, dim_k, n)`，使得矩阵相乘能够计算各个查询与键的相似度。
- 计算结果 `dist` 的形状为 `(batch, n, n)`，即每个查询与每个键之间的相似度矩阵。然后乘以归一化因子 `self._norm_fact`。

小红书账号：小黄搞AI

softmax计算

```
1         dist = torch.softmax(dist, dim=-1)  # batch, n, n
```

- 使用 `softmax` 函数对相似度进行归一化处理，将相似度转换为注意力权重。 `dim=-1` 表示在最后一个维度上（即每个序列位置的所有相似度）进行归一化。

与值矩阵v进行矩阵乘法

```
1         att = torch.bmm(dist, v)
```

- `torch.bmm(dist, v)` ：将归一化后的注意力权重 `dist` 与值矩阵 `v` 进行矩阵乘法。结果 `att` 包含了经过加权求和的上下文信息，即自注意力的输出。

返回自注意力的输出

```
1         return att
```

- 返回自注意力的输出 `att` ，它的形状为 `(batch_size, sequence_length, dim_v)` 。

总结

- 这个实现首先将输入投影为查询、键和值矩阵。
- 通过计算查询与键的内积得到相似度分数，并对其进行归一化处理。
- 最后，使用这些注意力权重对值进行加权求和，输出上下文相关的信息。

这是一个单头自注意力机制的实现，因为这里只计算了一次查询、键和值的组合。多头注意力机制则会并行计算多个这样的组合，每个头会有不同的线性变换矩阵。

59. 手撕自注意力机制（不用Pytorch）（Self Attention）

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

Query（查询）	Key（键）	Value（值）
代表模型要对输入序列中的哪些部分进行注意	每个输入元素都有对应的键，表示它在整个序列中的表示	每个输入元素对应的值，最终会通过注意力权重加权求

小红书账号：小黄搞AI

> 在面试中要求“手撕代码”，通常是指**不依赖高层次的库函数（例如PyTorch中的 `nn.Linear` 等）**，而是从头实现出自注意力机制的功能。

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

class SelfAttentionScratch:
    def __init__(self, dim_q, dim_k, dim_v):
        self.dim_q = dim_q
        self.dim_k = dim_k
        self.dim_v = dim_v
        # 手动初始化查询、键和值的权重矩阵
        self.W_q = torch.randn(dim_q, dim_k) / dim_k**0.5  # 查询的线性变换矩阵
        self.W_k = torch.randn(dim_q, dim_k) / dim_k**0.5  # 键的线性变换矩阵
        self.W_v = torch.randn(dim_q, dim_v) / dim_v**0.5  # 值的线性变换矩阵
        self._norm_fact = 1 / (dim_k ** 0.5)  # 归一化因子

    def forward(self, x):
        batch, n, dim_q = x.shape
        # 计算查询、键和值矩阵
        q = torch.matmul(x, self.W_q)  # (batch, n, dim_k)
        k = torch.matmul(x, self.W_k)  # (batch, n, dim_k)
        v = torch.matmul(x, self.W_v)  # (batch, n, dim_v)
        # 计算相似度（查询与键的点积）
        scores = torch.matmul(q, k.transpose(1, 2)) * self._norm_fact  # (batch, n, n)
        # 对相似度进行softmax归一化，得到注意力权重
        attention_weights = F.softmax(scores, dim=-1)  # (batch, n, n)
        # 根据注意力权重加权求和值
        attention_output = torch.matmul(attention_weights, v)  # (batch, n, dim_v)
        return attention_output

# 测试一下手撕实现
x = torch.randn(2, 5, 10)  # batch_size=2, sequence_length=5, dim_q=10
attention = SelfAttentionScratch(dim_q=10, dim_k=8, dim_v=8)
output = attention.forward(x)
print(output.shape)  # 输出应该是 (2, 5, 8)
```

逐行代码解释

定义 SelfAttentionScratch 类

```python
class SelfAttentionScratch:
```

小红书账号：小黄搞AI

这里定义了一个 `SelfAttentionScratch` 类，用来实现自注意力机制。与基于 `nn.Module` 的高层实现不同，它没有依赖现有的库函数，而是直接定义矩阵计算。

`__init__` 构造函数

```
    def __init__(self, dim_q, dim_k, dim_v):
        self.dim_q = dim_q
        self.dim_k = dim_k
        self.dim_v = dim_v
```

这里的 `__init__` 函数接收三个参数：

- `dim_q`：输入特征（Query）的维度。
- `dim_k`：Key 和 Query 的映射维度。
- `dim_v`：Value 的映射维度。

接下来，类内会基于这些维度参数初始化一些必要的权重矩阵。

初始化查询、键和值的权重矩阵

```
        self.W_q = torch.randn(dim_q, dim_k) / dim_k**0.5  # 查询的线性变换矩阵
        self.W_k = torch.randn(dim_q, dim_k) / dim_k**0.5  # 键的线性变换矩阵
        self.W_v = torch.randn(dim_q, dim_v) / dim_v**0.5  # 值的线性变换矩阵
```

在这里：

- `self.W_q`，`self.W_k`，`self.W_v` 是查询（Q），键（K）和值（V）的权重矩阵，分别用高斯分布随机初始化。
- 初始化时除以 `dim_k**0.5` 和 `dim_v**0.5` 主要是为了防止数值过大引起梯度爆炸。特别是在注意力机制中，常用这种缩放技术来稳定训练。
  - `W_q` 大小为 `(dim_q, dim_k)`：将输入的 Query 映射到 `dim_k` 维度。
  - `W_k` 大小为 `(dim_q, dim_k)`：将输入的 Key 映射到 `dim_k` 维度。
  - `W_v` 大小为 `(dim_q, dim_v)`：将输入的 Value 映射到 `dim_v` 维度。

定义归一化因子

```
        self._norm_fact = 1 / (dim_k ** 0.5)  # 归一化因子
```

小红书账号：小黄搞AI

根据标准的注意力机制公式，`Query` 和 `Key` 的点积在计算之前需要除以 `sqrt(dim_k)`，这就是归一化因子。目的是避免在点积操作中，值的绝对数值过大，从而导致梯度消失或爆炸问题。

forward 函数

```
    def forward(self, x):
        batch, n, dim_q = x.shape
```

- 这里的 `forward` 函数接收输入 `x`，假设 `x` 是一个三维张量，形状为 `(batch_size, sequence_length, dim_q)`，即批次大小、序列长度、和输入特征维度。
- `batch` 是批次大小，`n` 是序列长度，`dim_q` 是输入的特征维度。

计算查询、键和值矩阵

```
        q = torch.matmul(x, self.W_q)  # (batch, n, dim_k)
        k = torch.matmul(x, self.W_k)  # (batch, n, dim_k)
        v = torch.matmul(x, self.W_v)  # (batch, n, dim_v)
```

- **查询矩阵 `q`**：输入张量 `x` 与查询权重矩阵 `self.W_q` 相乘，结果是 `(batch_size, sequence_length, dim_k)`。
- **键矩阵 `k`**：输入张量 `x` 与键权重矩阵 `self.W_k` 相乘，结果是 `(batch_size, sequence_length, dim_k)`。
- **值矩阵 `v`**：输入张量 `x` 与值权重矩阵 `self.W_v` 相乘，结果是 `(batch_size, sequence_length, dim_v)`。

此时，`q` 和 `k` 可以用于相似性度量，而 `v` 会用于最终输出。

计算相似度分数

```
        scores = torch.matmul(q, k.transpose(1, 2)) * self._norm_fact  # (batch, n, n)
```

- 这里我们通过点积操作 `torch.matmul(q, k.transpose(1, 2))` 计算 Query 和 Key 的相似度。
  - `k.transpose(1, 2)` 是对 `k` 进行转置，交换其序列长度维度和最后一个维度，方便与 `q` 做点积操作。最终 `q` 的形状是 `(batch_size, sequence_length, dim_k)`，`k.transpose(1, 2)` 的形状是 `(batch_size, dim_k, sequence_length)`，二者相乘得到 `(batch_size, sequence_length, sequence_length)` 的相似度矩阵。

小红书账号：小黄搞AI

- 然后，结果乘以归一化因子 `self._norm_fact`，用于缩放。

对相似度分数进行 softmax 归一化

```
1         attention_weights = F.softmax(scores, dim=-1)  # (batch, n, n)
```

- 通过 `F.softmax` 函数将相似度矩阵转换为注意力权重。`dim=-1` 表示对最后一个维度（即 `sequence_length` 维度）进行 softmax，确保每个序列位置的注意力权重总和为 1。

根据注意力权重加权求和值

```
1         attention_output = torch.matmul(attention_weights, v)  # (batch, n,
  dim_v)
```

- 使用计算得到的注意力权重 `attention_weights` 去加权 `v`，即 `torch.matmul(attention_weights, v)`。这样得到的 `attention_output` 是注意力机制的输出，形状为 `(batch_size, sequence_length, dim_v)`。

返回结果

```
1         return attention_output
```

最终输出的是 `attention_output`，它表示输入序列经过自注意力机制后的输出。

测试

```
1 x = torch.randn(2, 5, 10)  # batch_size=2, sequence_length=5, dim_q=10
2 attention = SelfAttentionScratch(dim_q=10, dim_k=8, dim_v=8)
3 output = attention.forward(x)
4 print(output.shape)  # 输出应该是 (2, 5, 8)
```

这里对手撕的自注意力机制进行测试：

- 输入 `x` 是一个随机生成的张量，形状为 `(2, 5, 10)`，表示有 2 个样本，每个样本的序列长度是 5，特征维度是 10。
- 创建了一个 `SelfAttentionScratch` 对象，并调用 `forward` 方法。
- 最后输出的形状是 `(2, 5, 8)`，这与 `dim_v=8` 的设定一致。

小红书账号：小黄搞AI

总结

- **手撕自注意力机制：**在面试中，要求手撕代码是指从头实现一个功能，而不依赖于现成的高层次库。在这里，我们手动实现了自注意力机制，涉及到矩阵相乘、转置、点积、softmax 等核心操作。
- **主要步骤：**包括生成查询、键和值矩阵，计算相似度分数，使用 softmax 归一化权重，并根据注意力权重加权 `Value`。

60. 手撕多头注意力机制（用Pytorch）（Multi Head Attention）

多头注意力机制（Multi-Head Attention）是 Transformer 模型中的核心组件，用于捕捉不同位置之间的依赖关系。通过将注意力计算分成多个头部，通过多个头，模型可以在不同的投影空间中计算注意力，**最终将各个头的输出拼接，并通过线性层进行整合**。具体过程为：

$$MultiHead(Q,K,V) = Concat(head1,\ldots,headh)W_O$$

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

class MultiHeadAttention(nn.Module):
    def __init__(self, heads, d_model, dropout=0.1):
        super().__init__()
        # 保存输入的d_model（模型维度）和头的数量
        self.d_model = d_model
        # d_k是每个头的特征维度，通常是d_model / heads
        self.d_k = d_model // heads
        self.h = heads
        # 三个线性变换层，分别用于将输入映射为Q、K、V
        self.q_linear = nn.Linear(d_model, d_model)  # 线性层，映射输入到Q
        self.v_linear = nn.Linear(d_model, d_model)  # 线性层，映射输入到V
        self.k_linear = nn.Linear(d_model, d_model)  # 线性层，映射输入到K
        # Dropout层，用于防止过拟合
        self.dropout = nn.Dropout(dropout)
        # 输出的线性层，最终将多头输出的结果合并回原始d_model维度
        self.out = nn.Linear(d_model, d_model)

    def attention(self, q, k, v, d_k, mask=None, dropout=None):
        # 1. 计算注意力分数（Q*K^T / sqrt(d_k)）
        # 其中q的维度是 (batch_size, heads, seq_len, d_k)
        # k的维度是 (batch_size, heads, seq_len, d_k)
        # 计算的结果是 (batch_size, heads, seq_len_q, seq_len_k)
        scores = torch.matmul(q, k.transpose(-2, -1)) / math.sqrt(d_k)

```

小红书账号：小黄搞AI

```
        # 2. 如果有mask，则将mask的部分填充为一个极小的值，表示这些部分不计算注意力
        if mask is not None:
            mask = mask.unsqueeze(1)  # 扩展mask的维度以适配多头的操作
            scores = scores.masked_fill(mask == 0, -1e9)  # 将mask位置设为极小值

        # 3. 使用softmax计算权重，使得权重总和为1，表示每个位置对其他位置的相关性
        scores = F.softmax(scores, dim=-1)

        # 4. 如果有dropout层，应用于注意力权重上
        if dropout is not None:
            scores = dropout(scores)

        # 5. 使用注意力权重对V矩阵进行加权求和
        # q和k的相似性决定了v的加权方式，输出的维度是 (batch_size, heads, seq_len_q, d_k)
        output = torch.matmul(scores, v)
        return output

    def forward(self, q, k, v, mask=None):
        bs = q.size(0)  # 获取batch_size大小

        # 1. 对Q、K、V分别进行线性变换
        # 输入q, k, v的维度为 (batch_size, seq_len, d_model)
        # 通过线性层后，维度依旧为 (batch_size, seq_len, d_model)
        k = self.k_linear(k)  # 线性变换后的k
        q = self.q_linear(q)  # 线性变换后的q
        v = self.v_linear(v)  # 线性变换后的v

        # 2. 维度变换：将d_model拆分为多个头 (heads)
        # 将Q、K、V重新reshape为 (batch_size, seq_len, heads, d_k)
        # 然后再交换维度，得到 (batch_size, heads, seq_len, d_k)
        k = k.view(bs, -1, self.h, self.d_k).transpose(1, 2)
        q = q.view(bs, -1, self.h, self.d_k).transpose(1, 2)
        v = v.view(bs, -1, self.h, self.d_k).transpose(1, 2)

        # 3. 计算注意力，返回的 scores 维度为 (batch_size, heads, seq_len_q, d_k)
        scores = self.attention(q, k, v, self.d_k, mask, self.dropout)

        # 4. 把多头的输出拼接起来
        # 先将 (batch_size, heads, seq_len, d_k) 变回 (batch_size, seq_len, heads, d_k)
        # 然后将多个头拼接为原始的 (batch_size, seq_len, d_model)
        concat = scores.transpose(1, 2).contiguous().view(bs, -1, self.d_model)

        # 5. 通过输出线性层，再次映射回 (batch_size, seq_len, d_model)
        output = self.out(concat)
        return output
```

小红书账号：小黄搞AI

```
75
```

维度变化示例

假设 `batch_size=2`，`seq_len=10`，`d_model=512`，`heads=8`，那么：

1. 输入的 `q, k, v` 维度是 `(2, 10, 512)`。
2. 经过线性层后，`q, k, v` 的维度仍然是 `(2, 10, 512)`。
3. 使用 `.view(2, 10, 8, 64)` 将其拆分为8个头，每个头大小为64（`d_k=64`），变成 `(2, 8, 10, 64)`。
4. `transpose(1, 2)` 后维度变为 `(2, 8, 10, 64)`，以便每个头独立进行计算。
5. 计算注意力后的结果维度是 `(2, 8, 10, 64)`。
6. 最后通过 `concat` 和 `out` 映射回原始的 `(2, 10, 512)`。

逐行解释代码

导入 PyTorch 库及其模块

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

- 导入 PyTorch 库及其模块，包括 `torch`（张量操作），`torch.nn`（神经网络组件）和 `torch.nn.functional`（各种功能性操作），以及 `math` 模块用于数学计算。

定义 MultiHeadAttention 类

```python
class MultiHeadAttention(nn.Module):
    def __init__(self, heads, d_model, dropout=0.1):
        super().__init__()
        self.d_model = d_model
        self.d_k = d_model // heads
        self.h = heads
```

- 定义 `MultiHeadAttention` 类，继承自 `nn.Module`。
- 在初始化函数中，接受参数 `heads`（头的数量），`d_model`（模型的维度），和 `dropout`（dropout 概率）。
- `self.d_k` 计算每个头的键的维度。

小红书账号：小黄搞AI

- `self.h` 存储头的数量。

创建线性层

```
        self.q_linear = nn.Linear(d_model, d_model)
        self.v_linear = nn.Linear(d_model, d_model)
        self.k_linear = nn.Linear(d_model, d_model)
        self.dropout = nn.Dropout(dropout)
        self.out = nn.Linear(d_model, d_model)
```

- 创建线性层，用于将输入的 `d_model` 维度映射到查询（Q）、键（K）和值（V）。这些层的输出维度仍为 `d_model`。
- `self.dropout` 定义一个 dropout 层。
- `self.out` 定义一个输出层，用于将最终的注意力输出映射回 `d_model` 维度。

定义 attention 函数

```
    def attention(self, q, k, v, d_k, mask=None, dropout=None):
        scores = torch.matmul(q, k.transpose(-2, -1)) / math.sqrt(d_k)
```

- 定义 `attention` 函数，接受查询 `q`、键 `k`、值 `v`、键的维度 `d_k`、可选的 `mask` 和 `dropout`。
- 计算注意力分数：对 `q` 和 `k` 进行矩阵乘法，并除以 $\sqrt{d_k}$ 进行缩放。

使用 masked_fill 掩码

```
        if mask is not None:
            mask = mask.unsqueeze(1)
            scores = scores.masked_fill(mask == 0, -1e9)
```

- 如果存在掩码，使用 `unsqueeze(1)` 扩展掩码的维度，以便与 `scores` 进行广播。
- 使用 `masked_fill` 将掩码为 0 的位置的分数设置为负无穷，以确保在 softmax 中被忽略。

对 scores 进行 softmax 操作

```
        scores = F.softmax(scores, dim=-1)
```

- 对 `scores` 进行 softmax 操作，得到注意力权重。

小红书账号：小黄搞AI

在注意力权重上应用 dropout

```
        if dropout is not None:
            scores = dropout(scores)
```

- 如果传入了 dropout，则在注意力权重上应用 dropout，防止过拟合。

注意力权重对值v进行加权求和

```
        output = torch.matmul(scores, v)
        return output
```

- 使用计算得到的注意力权重对值 `v` 进行加权求和，返回最终的输出。

定义 forward 方法

```
    def forward(self, q, k, v, mask=None):
        bs = q.size(0)
```

- 定义 `forward` 方法，接受查询、键和值，以及可选的掩码。
- `bs` 代表批次大小，通过查询的第一个维度获得。

对输入的 k、q 和 v 应用线性变换

```
        k = self.k_linear(k).view(bs, -1, self.h, self.d_k).transpose(1, 2)
        q = self.q_linear(q).view(bs, -1, self.h, self.d_k).transpose(1, 2)
        v = self.v_linear(v).view(bs, -1, self.h, self.d_k).transpose(1, 2)
```

- 对输入的 `k`、`q` 和 `v` 应用线性变换，并调整形状为 `(bs, -1, h, d_k)`，然后进行转置，变为 `(bs, h, n, d_k)` 形式，以便进行多头计算。

调用 attention 函数

```
        scores = self.attention(q, k, v, self.d_k, mask, self.dropout)
```

- 调用 `attention` 函数，计算注意力输出。

将 scores 转置回原来的维度顺序

小红书账号：小黄搞AI

```
        concat = scores.transpose(1, 2).contiguous().view(bs, -1, self.d_model)
```

- 将 `scores` 转置回原来的维度顺序，并调整形状为 `(bs, -1, d_model)`，以便拼接多头输出。

使用线性层将拼接后的输出映射回 d_model 维度

```
        output = self.out(concat)
        return output
```

- 使用线性层将拼接后的输出映射回 `d_model` 维度，返回最终的输出。

总结

这段代码实现了一个多头注意力机制，主要包括输入的线性变换、注意力计算、以及多头输出的拼接和映射，能有效捕捉输入序列中不同位置之间的依赖关系。

61. 手撕多头注意力机制（不用Pytorch）（Multi Head Attention）

> 在面试中要求“手撕代码”，有时面试官要求**不依赖高层次的库函数（例如PyTorch中的 `nn.Linear` 等）**，而是从头实现出多头注意力机制的功能。

代码实现

```python
import numpy as np

# 定义softmax函数
def softmax(x):
    e_x = np.exp(x - np.max(x, axis=-1, keepdims=True))  # 稳定性处理，避免溢出
    return e_x / np.sum(e_x, axis=-1, keepdims=True)

# 单头注意力机制
def scaled_dot_product_attention(q, k, v, mask=None):
    d_k = q.shape[-1]  # d_k = Q的最后一个维度
    scores = np.dot(q, k.T) / np.sqrt(d_k)  # 计算 Q*K^T / sqrt(d_k)
    if mask is not None:
        scores = np.where(mask == 0, -1e9, scores)  # 使用mask对scores进行填充
    # 计算注意力权重
    attention_weights = softmax(scores)
    # 计算最终输出
    output = np.dot(attention_weights, v)  # 将注意力权重与V相乘
    return output, attention_weights
```

小红书账号：小黄搞AI

```python

# 多头注意力机制
class MultiHeadAttention:
    def __init__(self, heads, d_model):
        self.d_model = d_model  # 输入embedding的维度
        self.h = heads  # 头的数量
        self.d_k = d_model // heads  # 每个头的维度
        # 初始化Q, K, V的权重矩阵（简单起见用随机初始化）
        self.q_linear = np.random.rand(d_model, d_model)
        self.k_linear = np.random.rand(d_model, d_model)
        self.v_linear = np.random.rand(d_model, d_model)
        # 输出的全连接层
        self.out_linear = np.random.rand(d_model, d_model)

    def split_heads(self, x, batch_size):
        """
        将输入拆分为多个头。将每个输入的 d_model 维度拆成 heads 个部分，每个部分的维度是 d_k。
        x: 输入是 (batch_size, seq_len, d_model) 形状的张量。
        """
        x = x.reshape(batch_size, -1, self.h, self.d_k)  # 形状调整为 (batch_size, seq_len, heads, d_k)
        x = x.transpose(0, 2, 1, 3)  # 交换维度 (batch_size, heads, seq_len, d_k)
        return x

    def forward(self, q, k, v, mask=None):
        batch_size = q.shape[0]
        # 1. 线性投影：将输入线性映射到Q, K, V空间
        q = np.dot(q, self.q_linear)  # 形状变为 (batch_size, seq_len, d_model)
        k = np.dot(k, self.k_linear)  # 形状变为 (batch_size, seq_len, d_model)
        v = np.dot(v, self.v_linear)  # 形状变为 (batch_size, seq_len, d_model)
        # 2. 拆分多个头
        q = self.split_heads(q, batch_size)  # (batch_size, heads, seq_len, d_k)
        k = self.split_heads(k, batch_size)  # (batch_size, heads, seq_len, d_k)
        v = self.split_heads(v, batch_size)  # (batch_size, heads, seq_len, d_k)
        # 3. 对每个头计算单头注意力
        outputs = []
        for i in range(self.h):  # 对每个注意力头进行单独计算
            output, _ = scaled_dot_product_attention(q[:, i], k[:, i], v[:, i], mask)  # 每个头独立计算注意力
            outputs.append(output)
        # 4. 拼接多个头的输出
```

小红书账号：小黄搞AI

```
        concatenated = np.concatenate(outputs, axis=-1)  # (batch_size, seq_len, d_model)
        # 5. 通过最后的线性层
        output = np.dot(concatenated, self.out_linear)
        return output

# 测试多头注意力机制
batch_size = 2
seq_len = 3
d_model = 8  # 模型维度
heads = 2  # 多头数量

# 假设输入是 (batch_size, seq_len, d_model) 的张量
q = np.random.rand(batch_size, seq_len, d_model)
k = np.random.rand(batch_size, seq_len, d_model)
v = np.random.rand(batch_size, seq_len, d_model)
mask = None  # 暂时不使用mask

mha = MultiHeadAttention(heads, d_model)
output = mha.forward(q, k, v, mask)

print("输出:", output)
```

逐行解释

`Softmax函数`

- `softmax` 函数计算分数的归一化权重，确保注意力分数按比例作用于值向量。`np.max(x)` 用来避免溢出问题。

`scaled_dot_product_attention 函数`

- 这是单个头的注意力计算过程。
- `q` 和 `k.T` 矩阵相乘，生成注意力分数，并使用 `np.sqrt(d_k)` 进行缩放。
- 如果有 `mask`，它会用极小值填充 `mask` 为 0 的位置。
- 使用 `softmax` 函数计算注意力权重。
- 最终，注意力权重与 `v` 相乘，得到输出。

`MultiHeadAttention 类`

- 初始化时，我们定义了 `q_linear`、`k_linear` 和 `v_linear`，这三个随机矩阵用于将输入线性变换为 Q、K、V。
- `out_linear` 是多头输出的最终线性变换。

`split_heads 函数`

小红书账号：小黄搞AI

- 将 `q`、`k`、`v` 拆分为多个头。首先，通过 `.reshape` 将张量的 `d_model` 维度划分为 `heads` 个头，每个头的维度是 `d_k`。然后，使用 `.transpose` 交换维度，使其适配后续计算。

`forward` 函数

- 线性变换：通过 `q_linear`、`k_linear` 和 `v_linear` 对输入进行线性投影。
- 拆分头：调用 `split_heads`，将 `q`、`k`、`v` 划分为多个头。
- 单头计算：遍历每个头，调用 `scaled_dot_product_attention` 计算单头注意力输出。
- 拼接输出：将所有头的输出拼接在一起，并通过最终线性层。

62. 手撕MQA

63. 手撕绝对位置编码

由于Transformer不具有处理序列顺序的内在机制，因此通过为每个位置添加位置编码来为模型提供顺序信息。位置编码采用正弦和余弦函数生成，公式为：

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

```python
import torch
import torch.nn as nn
import math

class PositionalEncoder(nn.Module):
    def __init__(self, d_model, max_seq_len=80):
        super().__init__()
        self.d_model = d_model
        pe = torch.zeros(max_seq_len, d_model)
        for pos in range(max_seq_len):
            for i in range(0, d_model, 2):
                pe[pos, i] = math.sin(pos / (10000**((2 * i) / d_model)))
                pe[pos, i + 1] = math.cos(pos / (10000**((2 * (i + 1)) / d_model)))
        pe = pe.unsqueeze(0)
        self.register_buffer('pe', pe)

    def forward(self, x):
        x = x * math.sqrt(self.d_model)
        seq_len = x.size(1)
        x = x + self.pe[:, :seq_len]
        return x
```

小红书账号：小黄搞AI

逐行解释

导入库

```
1 import torch
2 import torch.nn as nn
3 import math
```

- **导入库**：导入了 `torch`（用于张量操作）、`torch.nn`（用于构建神经网络相关模块）和 `math`（用于数学函数，比如 `sin` 和 `cos`）。

类定义

```
1 class PositionalEncoder(nn.Module):
```

- **类定义**：定义一个新的类 `PositionalEncoder`，继承自 `nn.Module`，使其能够与 PyTorch 神经网络框架集成，像其他层一样使用。

构造函数

```
1     def __init__(self, d_model, max_seq_len=80):
2         super().__init__()
```

- **构造函数**：`__init__` 是类的构造函数，在创建 `PositionalEncoder` 对象时调用。`d_model` 是输入嵌入的维度，`max_seq_len` 是序列的最大长度（默认为 80）。`super().__init__()` 调用父类 `nn.Module` 的初始化函数。

存储 d_model

```
1         self.d_model = d_model
```

- **存储 d_model**：将模型维度 `d_model` 存储为实例变量，供后续使用。

创建零张量用于位置编码（pe）

```
1         pe = torch.zeros(max_seq_len, d_model)
```

小红书账号：小黄搞AI

- **创建零张量用于位置编码（pe）**：创建一个形状为 `(max_seq_len, d_model)` 的零张量 `pe`，用于存储每个位置和每个维度的编码值。

嵌套循环填充位置编码矩阵

```
        for pos in range(max_seq_len):
            for i in range(0, d_model, 2):
```

- **嵌套循环填充位置编码矩阵**：
  - 外层循环遍历每个位置 `pos`，范围为 0 到 `max_seq_len - 1`。
  - 内层循环遍历每个偶数维度 `i`，范围是 `d_model`，步长为 2（只遍历偶数维度索引）。

为偶数和奇数维度赋值

```
                pe[pos, i] = math.sin(pos / (10000**((2 * i) / d_model)))
                pe[pos, i + 1] = math.cos(pos / (10000**((2 * (i + 1)) / d_model)))
```

- **为偶数和奇数维度赋值**：
  - 对于偶数索引 `i`，使用正弦函数计算位置编码：

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

  - 对于奇数索引 `i + 1`，使用余弦函数计算位置编码：

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{model}}}}\right)$$

  - 这种正弦和余弦交替的模式帮助模型区分序列中的不同位置。

增加批次维度

```
        pe = pe.unsqueeze(0)
```

- **增加批次维度**：将 `pe` 张量增加一个维度，使其形状从 `(max_seq_len, d_model)` 变为 `(1, max_seq_len, d_model)`。这使得后续与输入嵌入相加时，维度匹配更加方便。

注册缓冲区

```
        self.register_buffer('pe', pe)
```

小红书账号：小黄搞AI

- **注册缓冲区**：将 `pe` 张量注册为缓冲区。这意味着它会与模型一起保存，但在反向传播过程中不会更新（它不是可学习参数）。

前向传播函数

```
    def forward(self, x):
        x = x * math.sqrt(self.d_model)
```

- **前向传播函数**：`forward` 定义了如何处理输入 `x`。这里 `x` 是输入嵌入。
  - 输入嵌入 `x` 乘以 `d_model` 的平方根来缩放，这有助于稳定模型的优化过程（这一操作来自 Transformer 论文）。

获取序列长度

```
        seq_len = x.size(1)
```

- **获取序列长度**：从 `x` 的第二维度获取序列长度（即输入的实际序列长度），并存储在 `seq_len` 中。

加上位置编码

```
        x = x + self.pe[:, :seq_len]
```

- **加上位置编码**：将位置编码 `pe` 加到输入 `x` 上。由于 `pe` 的形状为 `(1, max_seq_len, d_model)`，我们根据输入的序列长度 `seq_len` 对 `pe` 进行切片，使其匹配输入的序列长度，然后在批次维度上进行广播。

总结

这个 **PositionalEncoder** 类使用正弦和余弦函数为序列中的每个位置生成固定的位置编码，并将它们添加到输入嵌入中。通过这种方式，Transformer 模型能够感知序列中各个词的顺序信息，帮助模型更好地处理序列数据。

64. 手撕RoPE

Rotary Position Embedding (RoPE) 通过将固定的旋转变换应用于查询和键的向量来编码位置信息。给定一个查询或键向量 q 或 k，RoPE将它们与位置信息结合起来。其核心公式如下

小红书账号：小黄搞AI

对于每个位置 i 和维度 m ，RoPE 在偶数和奇数维度上应用旋转变换：

$$\text{RoPE}(q_i) = \left[q_i^{(2m)} \cos(i \cdot \theta_m) - q_i^{(2m+1)} \sin(i \cdot \theta_m), q_i^{(2m)} \sin(i \cdot \theta_m) + q_i^{(2m+1)} \cos(i \cdot \theta_m) \right]$$

其中，频率 θ_m 是每个维度的变化频率：$\theta_m = \frac{1}{10000^{2m/d}}$

其中，d 是嵌入的维度，m 是维度索引的一半。

```python
import torch
import math

class RoPE:
    def __init__(self, dim):
        """
        RoPE初始化，指定嵌入维度。
        参数:
        - dim: 旋转位置编码的维度，通常等于模型的嵌入维度。
        """
        self.dim = dim
        self.freqs = self._build_freqs()

    def _build_freqs(self):
        """
        构建频率序列，返回形状为 (dim // 2,) 的频率tensor。
        """
        # 生成频率序列，基于公式 theta_m = 10000^(-2m/d)
        return torch.pow(10000, -torch.arange(0, self.dim, 2).float() / self.dim)

    def apply(self, x, positions):
        """
        将RoPE应用到输入向量。
        参数:
        - x: 输入向量，形状为 (batch_size, seq_len, dim)
        - positions: 位置信息，形状为 (seq_len,)
        """
        # 将位置信息与频率相乘，得到旋转角度
        angle = torch.einsum('i,j->ij', positions, self.freqs)  # (seq_len, dim//2)

        # 将偶数维度和奇数维度分开进行旋转
        cos_angle = torch.cos(angle).unsqueeze(-1)  # (seq_len, dim//2, 1)
        sin_angle = torch.sin(angle).unsqueeze(-1)  # (seq_len, dim//2, 1)

        # 重新组织输入x的维度为 (batch_size, seq_len, dim//2, 2)
        x_reshaped = x.view(x.shape[0], x.shape[1], self.dim // 2, 2)

```

小红书账号：小黄搞AI

```
        # 应用旋转变换
        x_rotated = torch.cat([x_reshaped[..., 0] * cos_angle - x_reshaped[..., 1] * sin_angle,
                               x_reshaped[..., 0] * sin_angle + x_reshaped[..., 1] * cos_angle], dim=-1)

        # 恢复原始维度形状 (batch_size, seq_len, dim)
        return x_rotated.view_as(x)

# 示例用法
dim = 16  # 嵌入维度
rope = RoPE(dim)
x = torch.randn(2, 4, dim)  # batch_size=2, seq_len=4, dim=16 的输入向量
positions = torch.arange(4)  # 位置为 [0, 1, 2, 3]

# 应用RoPE
x_encoded = rope.apply(x, positions)
print(x_encoded)
```

逐行解释

1. `import torch` 和 `import math` 导入 PyTorch 和数学库以进行张量操作和数学计算。
2. `class RoPE:` 定义 `RoPE` 类，它负责计算旋转位置编码并将其应用到输入向量中。
3. `__init__(self, dim):` 初始化函数，接受嵌入维度 `dim` 作为参数。
4. `self.freqs = self._build_freqs()` 构建频率序列，通过 `_build_freqs()` 函数生成一个频率向量。
5. `_build_freqs(self):` 这个函数生成每个维度的频率。频率的计算基于公式 $\theta_m = \frac{1}{10000^{2m/d}}$，因此使用 `torch.pow(10000, -torch.arange(0, self.dim, 2).float() / self.dim)` 来构造频率序列。
6. `apply(self, x, positions):` 将RoPE应用到输入向量 `x` 上，`x` 的形状为 `(batch_size, seq_len, dim)`，并使用 `positions` (表示位置序列)。
7. `angle = torch.einsum('i,j->ij', positions, self.freqs)` 计算每个位置与频率相乘得到的旋转角度。使用爱因斯坦求和表示法 `einsum` 来计算位置序列和频率序列的外积。
8. `cos_angle = torch.cos(angle).unsqueeze(-1)` 计算旋转角度的余弦值，并通过 `unsqueeze` 在最后一维添加一个维度以匹配输入张量的形状。
9. `sin_angle = torch.sin(angle).unsqueeze(-1)` 计算旋转角度的正弦值，并同样调整其形状。

小红书账号：小黄搞AI

10. `x_reshaped = x.view(x.shape[0], x.shape[1], self.dim // 2, 2)` 将输入向量 `x` 重塑为形状 `(batch_size, seq_len, dim//2, 2)`，将每两个相邻维度作为一组（偶数维和奇数维）进行旋转。
11. `x_rotated = torch.cat([...], dim=-1)` 应用旋转公式：对偶数维度和奇数维度分别使用余弦和正弦旋转后，使用 `torch.cat` 将结果拼接回一起。
12. `return x_rotated.view_as(x)` 将旋转后的向量恢复为原始的形状 `(batch_size, seq_len, dim)`，返回旋转编码后的向量。

示例说明

在示例中，输入是一个形状为 `(2, 4, 16)` 的随机向量，即 batch size 为 2，序列长度为 4，嵌入维度为 16。`positions` 则是从 0 到 3 的位置索引。

65. 手撕Transformer中FFN代码

Transformer中的前馈神经网络（Feed-Forward Network, FFN）是每个编码器或解码器层的核心组件之一。它通常由两个线性变换和一个非线性激活函数（如ReLU或GELU）组成。

FFN的公式如下：$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$

其中：

- x 是输入向量，
- W_1 和 W_2 是权重矩阵，
- b_1 和 b_2 是偏置项，
- 通过激活函数（如 ReLU）应用非线性变换。

Python代码

```python
import torch
import torch.nn as nn

class FeedForwardNetwork(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        """
        初始化前馈神经网络。
        参数:
        - input_dim: 输入维度
        - hidden_dim: 隐藏层的维度
        - output_dim: 输出维度，通常与输入维度相同
        """
        super(FeedForwardNetwork, self).__init__()
```

小红书账号：小黄搞AI

```
        # 第一个线性层，将输入维度映射到隐藏维度
        self.linear1 = nn.Linear(input_dim, hidden_dim)
        # 激活函数，使用ReLU非线性激活
        self.relu = nn.ReLU()
        # 第二个线性层，将隐藏层维度映射回输出维度
        self.linear2 = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        """
        前向传播函数，定义如何将输入 x 通过前馈神经网络。
        参数:
        - x: 输入tensor，形状为 (batch_size, seq_len, input_dim)
        """
        # 通过第一个线性层
        x = self.linear1(x)
        # 应用激活函数ReLU
        x = self.relu(x)
        # 通过第二个线性层
        x = self.linear2(x)
        return x

# 示例用法
input_dim = 512  # 输入维度
hidden_dim = 2048  # 隐藏层维度
output_dim = 512  # 输出维度，通常等于输入维度

ffn = FeedForwardNetwork(input_dim, hidden_dim, output_dim)
x = torch.randn(2, 10, input_dim)  # batch_size=2, seq_len=10, input_dim=512

# 通过FFN进行前向传播
output = ffn(x)
print(output.shape)  # 输出形状为 (2, 10, 512)
```

逐行解释

1. `import torch` 和 `import torch.nn as nn` 导入PyTorch及其神经网络模块，用于定义模型和进行张量操作。
2. `class FeedForwardNetwork(nn.Module):` 定义一个继承自 `nn.Module` 的 `FeedForwardNetwork` 类，表示Transformer中的前馈神经网络。
3. `__init__(self, input_dim, hidden_dim, output_dim):` 初始化函数，定义输入维度、隐藏层维度和输出维度。一般情况下，输入维度和输出维度相等，隐藏层维度通常较大。
4. `super(FeedForwardNetwork, self).__init__()` 调用父类 `nn.Module` 的初始化方法。

小红书账号：小黄搞AI

5. `self.linear1 = nn.Linear(input_dim, hidden_dim)` 定义第一个线性层，将输入维度 `input_dim` 映射到隐藏层维度 `hidden_dim`。这个线性变换用于增加模型的容量。
6. `self.relu = nn.ReLU()` 定义激活函数，使用ReLU（Rectified Linear Unit），它将所有负值设置为0，从而引入非线性。
7. `self.linear2 = nn.Linear(hidden_dim, output_dim)` 定义第二个线性层，将隐藏层维度 `hidden_dim` 映射回输出维度 `output_dim`，通常等于输入维度。
8. `def forward(self, x):` 定义前向传播函数 `forward()`，描述输入张量如何通过网络层流动。
9. `x = self.linear1(x)` 将输入 `x` 通过第一个线性层 `linear1`，执行线性变换 $xW_1 + b_1$
10. `x = self.relu(x)` 通过ReLU激活函数，将线性变换后的结果非线性化，确保模型具有足够的表达能力。
11. `x = self.linear2(x)` 将经过激活函数的输出通过第二个线性层 `linear2`，得到最终的输出 $\text{ReLU}(xW_1 + b_1)W_2 + b_2$。
12. `return x` 返回前馈神经网络的输出。

66. 手撕Layer Norm

手撕 `LayerNorm` 的 Python 代码，首先需要理解其核心公式和计算步骤。`LayerNorm`（Layer Normalization）用于标准化神经网络层的输出，控制层输出的均值和方差。它的作用在于减小模型训练中的内部协变量偏移，使得训练更加稳定。

1. LayerNorm的数学公式

给定输入向量 x 及其每个维度的值 $x_1, x_2, \ldots, x_n$，`LayerNorm` 按如下公式进行标准化：

均值计算

$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

方差计算

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$$

标准化（Z-score标准化）

$$\hat{x}_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$$

缩放和平移（可选）

$$y_i = \gamma \hat{x}_i + \beta$$

> 其中 ϵ 是一个小值，用于避免分母为零的情况。
>
> 其中 γ 是可学习的缩放参数，β 是可学习的偏移参数。

2. 手撕LayerNorm代码

```python
import torch
import torch.nn as nn

class LayerNorm(nn.Module):
    def __init__(self, normalized_shape, eps=1e-5, elementwise_affine=True):
```

小红书账号：小黄搞AI

```
        super(LayerNorm, self).__init__()
        self.eps = eps
        self.elementwise_affine = elementwise_affine
        # 如果需要可学习参数，则初始化 gamma 和 beta
        if self.elementwise_affine:
            self.gamma = nn.Parameter(torch.ones(normalized_shape))
            self.beta = nn.Parameter(torch.zeros(normalized_shape))
        else:
            self.gamma = None
            self.beta = None

    def forward(self, x):
        # 计算均值和方差
        mean = x.mean(dim=-1, keepdim=True)
        var = x.var(dim=-1, keepdim=True, unbiased=False)
        # 进行标准化
        x_normalized = (x - mean) / torch.sqrt(var + self.eps)
        # 如果有 gamma 和 beta，则进行缩放和平移
        if self.elementwise_affine:
            x_normalized = self.gamma * x_normalized + self.beta
        return x_normalized

# 示例输入
x = torch.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
layer_norm = LayerNorm(x.shape[-1])
output = layer_norm(x)
print(output)
```

3. 逐行解释

1. `__init__` 方法：
   - `normalized_shape`：指的是要标准化的维度大小（通常是最后一维）。
   - `eps`：防止除零错误的小常数。
   - `elementwise_affine`：控制是否进行可学习的缩放和平移（默认为True）。

   初始化时，我们需要根据 `elementwise_affine` 的值来决定是否引入可学习的参数 `gamma` 和 `beta`。`gamma` 用于缩放，`beta` 用于平移，它们的形状都与输入数据的最后一维相同。
2. `forward` 方法：
   - **计算均值**：`x.mean(dim=-1, keepdim=True)` 计算输入 `x` 在最后一维的均值，使用 `keepdim=True` 保持原有的维度以便后续操作。
   - **计算方差**：`x.var(dim=-1, keepdim=True, unbiased=False)` 计算输入 `x` 的方差，其中 `unbiased=False` 确保我们使用有偏方差估计（在深度学习中更常用）。

小红书账号：小黄搞AI

- **标准化**：将 `x` 的每个元素减去均值，然后除以标准差（即方差的平方根），并且加上一个小的 `eps` 来防止除零错误。
- **缩放和平移**：如果 `elementwise_affine` 为 `True`，则对标准化后的结果进行缩放（乘以 `gamma`）和平移（加上 `beta`）。

3. **返回结果**：最后，标准化后的张量被返回，经过可选的缩放和平移。

67. 手撕RMSNorm

📌 RMSNorm（Root Mean Square Normalization）是LayerNorm的一种变体，它的不同点在于不计算均值，而是直接使用输入的范数（即均方根，Root Mean Square）来进行标准化。这种方法减少了计算开销，同时也能提供类似的归一化效果。

1. RMSNorm的数学公式

与 `LayerNorm` 不同，`RMSNorm` 只计算输入向量的均方根，公式如下：

均方根（RMS）计算	标准化	缩放和平移
$\text{RMS}(x) = \sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}$	$\hat{x}_i = \frac{x_i}{\text{RMS}(x) + \epsilon}$	$y_i = \gamma \hat{x}_i$

> 其中 ϵ 是一个小的常数，避免除零错误。
>
> RMSNorm不包含偏移项 β，只引入了可学习的缩放参数 γ

2. 手撕RMSNorm代码

```python
import torch
import torch.nn as nn

class RMSNorm(nn.Module):
    def __init__(self, normalized_shape, eps=1e-5, elementwise_affine=True):
        super(RMSNorm, self).__init__()
        self.eps = eps
        self.elementwise_affine = elementwise_affine
        # 如果需要可学习参数，则初始化 gamma
        if self.elementwise_affine:
            self.gamma = nn.Parameter(torch.ones(normalized_shape))
        else:
            self.gamma = None

    def forward(self, x):
        # 计算均方根（RMS）
```

小红书账号：小黄搞AI

```
        rms = torch.sqrt(torch.mean(x ** 2, dim=-1, keepdim=True) + self.eps)
        # 进行标准化
        x_normalized = x / rms
        # 如果有 gamma，则进行缩放
        if self.elementwise_affine:
            x_normalized = self.gamma * x_normalized
        return x_normalized

# 示例输入
x = torch.tensor([[1.0, 2.0, 3.0], [4.0, 5.0, 6.0]])
rms_norm = RMSNorm(x.shape[-1])
output = rms_norm(x)
print(output)
```

3. 逐行解释

1. `__init__` 方法：
   - `normalized_shape`：要标准化的维度大小。
   - `eps`：用于防止除零错误的小常数。
   - `elementwise_affine`：控制是否引入可学习的缩放参数（默认值为True）。

   如果 `elementwise_affine` 为 `True`，初始化可学习的参数 `gamma`，用于缩放。
2. `forward` 方法：
   - **计算均方根（RMS）**：通过 `torch.sqrt(torch.mean(x ** 2, dim=-1, keepdim=True) + self.eps)` 计算输入的均方根，`x ** 2` 表示对输入张量的每个元素平方，`torch.mean` 则计算均值，`torch.sqrt` 取平方根。我们加上一个小的 `eps` 来防止分母为零。
   - **标准化**：将输入 `x` 的每个元素除以均方根 `rms`，实现标准化。
   - **缩放**：如果 `elementwise_affine` 为 `True`，将标准化后的结果乘以 `gamma` 进行缩放。
3. **返回结果**：经过标准化并且可能缩放后的张量被返回。



RMSNorm与LayerNorm的主要区别，即不再计算均值，只计算均方根，从而降低了计算开销，同时保持标准化效果。

68. 手撕flash-attention



FlashAttention是一种优化的注意力机制实现，旨在提高大模型中的注意力计算效率，特别是降低了内存占用和加速计算。它**通过将常规的注意力计算步骤重组为块状矩阵运算，并通**

小红书账号：小黄搞AI

过减少中间结果的存储来减少内存的占用。FlashAttention使用了部分矩阵乘法和片段（chunked）计算，以优化内存使用，特别适用于长序列的计算。

1. FlashAttention的基本步骤

在常规的注意力计算中，我们有：

Query、Key 和 Value 矩阵的计算

$$Q = XW_q, \quad K = XW_k, \quad V = XW_v$$

注意力分数的计算

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

FlashAttention将这个过程优化为分块操作，通过减少矩阵乘法中间结果的存储需求来减少内存占用。

2. 手撕FlashAttention代码

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

class FlashAttention(torch.nn.Module):
    def __init__(self, embed_dim, num_heads, scale=None):
        super(FlashAttention, self).__init__()
        self.embed_dim = embed_dim
        self.num_heads = num_heads
        self.head_dim = embed_dim // num_heads
        self.scale = scale or self.head_dim ** -0.5

        # 初始化 Query, Key, Value 的线性映射
        self.q_proj = torch.nn.Linear(embed_dim, embed_dim)
        self.k_proj = torch.nn.Linear(embed_dim, embed_dim)
        self.v_proj = torch.nn.Linear(embed_dim, embed_dim)
        self.out_proj = torch.nn.Linear(embed_dim, embed_dim)

    def forward(self, x):
        batch_size, seq_len, embed_dim = x.size()

        # 计算 Q, K, V 矩阵
        Q = self.q_proj(x).view(batch_size, seq_len, self.num_heads, self.head_dim).transpose(1, 2)
        K = self.k_proj(x).view(batch_size, seq_len, self.num_heads, self.head_dim).transpose(1, 2)
        V = self.v_proj(x).view(batch_size, seq_len, self.num_heads, self.head_dim).transpose(1, 2)

        # 初始化输出
```

小红书账号：小黄搞AI

```
        out = torch.zeros_like(Q)
        block_size = 64  # 分块大小，可调整以优化性能

        for i in range(0, seq_len, block_size):
            # 取当前块的 Key 和 Value
            K_block = K[:, :, i:i + block_size]
            V_block = V[:, :, i:i + block_size]

            # 初始化块内的注意力得分和 softmax 缓存
            attention_scores = torch.zeros(batch_size, self.num_heads, seq_len, block_size, device=x.device)
            max_scores = torch.full((batch_size, self.num_heads, seq_len, 1), float('-inf'), device=x.device)
            softmax_sum = torch.zeros(batch_size, self.num_heads, seq_len, 1, device=x.device)

            for j in range(0, seq_len, block_size):
                # 取 Query 当前块
                Q_block = Q[:, :, j:j + block_size]

                # 计算 Q_block 和 K_block 的相似度，并进行缩放
                attention_chunk = torch.matmul(Q_block, K_block.transpose(-1, -2)) * self.scale

                # 使用局部 softmax 减少内存开销
                max_scores = torch.maximum(max_scores, attention_chunk.max(dim=-1, keepdim=True)[0])
                exp_scores = torch.exp(attention_chunk - max_scores)
                softmax_sum += exp_scores.sum(dim=-1, keepdim=True)

                # 计算注意力输出
                attention_scores[:, :, j:j + block_size] = exp_scores / softmax_sum

            # 累积输出
            out[:, :, i:i + block_size] = torch.matmul(attention_scores, V_block)

        # 通过线性层返回最终输出
        out = out.transpose(1, 2).contiguous().view(batch_size, seq_len, embed_dim)
        return self.out_proj(out)

# 示例输入
x = torch.randn(2, 128, 512)  # batch_size = 2, seq_len = 128, embed_dim = 512
flash_attention = FlashAttention(embed_dim=512, num_heads=8)
output = flash_attention(x)
```

小红书账号：小黄搞AI

```
66 print(output)
```

3. 逐行解释

1. `__init__` 方法：
   - `embed_dim`：输入嵌入的维度（例如 512）。
   - `num_heads`：多头注意力机制的头数。
   - `scale`：缩放因子，通常为 `1/√d_k`，即 `head_dim` 的倒数。

   初始化中，我们使用线性层 `q_proj`，`k_proj`，`v_proj` 来生成 Query、Key、Value 矩阵，同时 `out_proj` 负责最后的输出投影。
2. `forward` 方法：
   - 首先，将输入 `x` 通过线性层生成 Query、Key 和 Value 矩阵，并重塑为 `(batch_size, num_heads, seq_len, head_dim)` 的形状。
   - **分块处理**：通过 `for` 循环将 Key 和 Value 矩阵按块分割，每次处理一个大小为 `block_size` 的子块。
     - 对每个 Query 块和 Key 块进行矩阵乘法，计算注意力得分。
     - 对得分进行局部的 `softmax` 操作，使用块内最大值减少浮点数溢出的可能。
   - 使用得到的注意力得分与对应的 Value 块进行矩阵乘法，得到最终的输出。
3. **分块计算**：`FlashAttention` 的关键在于分块处理，使得在处理长序列时，减少了整个注意力矩阵的显式存储需求，每次只计算并存储一个块的注意力得分，从而减少内存消耗。

4. 总结

⚽ 这段代码实现了一个简化版的**FlashAttention**，其中通过分块操作减少了内存占用。在训练长序列数据时，它可以显著提高注意力计算的速度和内存效率，特别适合处理大规模的 transformer 模型。

小红书账号：小黄搞AI

小红书账号：小黄搞AI

69. 如何缓解大模型inference的时候的重复问题

🏝️ 在大型语言模型（如GPT-4）的推理过程中，生成的文本有时会出现重复。这种重复不仅影响文本质量，还会降低用户体验。回答应该从根源上解释导致重复的原因，并提供缓解重复问题的具体策略。

一、重复问题的根源

1.1 语言模型的概率分布倾向

语言模型通过对训练数据中模式的学习，生成每个可能词的概率分布。然而，以下原因可能导致模型倾向于重复：

1. **高频词汇权重过高**：在语言模型中，高频词汇的概率往往被模型学到较高的权重（如“的”、“是”），这使得它们容易在生成过程中反复出现。
2. **局部依赖问题**：模型倾向于基于当前上下文生成词汇，但如果上下文中的重复信息被赋予较高权重，模型可能更倾向重复。
3. **缺乏长程规划**：生成过程中，模型并没有整体规划文本的结构，可能在局部生成时不断循环相似的内容。

1.2 解码策略的影响

模型的解码策略直接影响生成文本的质量。以下是常见解码策略导致重复的原因：

1. **贪心搜索（Greedy Search）**
   - 每一步选择概率最高的词会让模型倾向于选择安全且重复的词汇。
   - 示例：`"我喜欢吃苹果。" → "我喜欢吃苹果苹果苹果苹果苹果苹果苹果苹果苹果苹果苹果..."`
2. **束搜索（Beam Search）**
   - 束搜索会同时跟踪多个候选序列，选择概率最高的路径。然而，如果没有多样性惩罚机制，多条路径可能趋同，导致相似句子反复生成。
   - 示例：在生成一段描述性文本时，可能会重复生成多次类似的句子：“这个房间很大。这个房间很大。这个房间很大。”
3. **缺乏熵正则化**

小红书账号：小黄搞AI

- 如果未引入熵正则化，模型的输出分布可能变得过于尖锐（集中在少数高概率词上），从而导致重复。

1.3 训练数据的问题

1. **重复样本的存在**
   - 如果训练数据中包含大量重复句子或片段，模型可能会认为这些重复是合适的语言结构。
   - 示例：如“你好，世界！你好，世界！你好，世界！”在训练数据中频繁出现，可能导致模型生成类似的重复内容。
2. **语料不平衡**
   - 某些主题或语言结构占比过高，导致模型偏向这些模式。
3. **低质量数据的影响**
   - 含有噪声或人为重复的训练数据会误导模型生成。

1.4 模型的局部最优

在生成过程中，模型可能进入局部最优解状态：

1. **高概率路径陷阱**
   - 模型生成的路径一旦形成某种模式，就可能不断重复该模式，因为它无法识别到更优的全局路径。
2. **缺乏多样性激励**
   - 生成过程中未对多样性进行奖励，导致模型倾向于安全但重复的选择。

二、缓解重复问题的策略

2.1 调整温度参数

温度参数 T 控制模型生成概率分布的平滑程度：

高温度 T：

- 平滑概率分布，增加不常见词的选择机会，从而提升多样性。
- 示例：在温度为1.2时，模型可能生成更多不确定但有趣的内容。
- **注意：温度过高可能导致生成的文本不连贯。**

低温度 $T < 1$：

- 集中概率分布，提升文本确定性，但容易导致重复。
- **建议：低温度适合任务型生成（如问答），而非开放式生成。**

🏆 **优化实践：**

小红书账号：小黄搞AI

- 对于需要严谨性的任务（如翻译），设置 $T = 0.7$。
- 对于创意任务（如故事生成），尝试 $T = 1.2$。

2.2 采用采样策略

Top-k 采样

原理：

- 每次生成时，仅从概率排名前 k 的词中采样，忽略其余低概率词。

实施细节：

- 选择合适的 k 值（如 $k = 40$）。k 值越小，生成内容越集中，重复概率越高；k 值越大，多样性越高，但可能降低连贯性。

核心采样（Top-p采样）

原理：

- 累计概率达到阈值 p 的词集合中随机采样。
- p 越小，生成的多样性越低；p 越大，生成的不确定性增加。

建议：

- 常用 p 值范围：0.8-0.95。
- 在生成较长段落时优先考虑核心采样。

2.3 重复惩罚

在生成过程中，对已经生成的词或短语降低其概率。

实现方法：

- 定义惩罚系数 α，在每次生成时动态调整词的概率：$P_{\text{adjusted}}(w) = \frac{P(w)}{\alpha}$ 其中 $\alpha > 1$ 表示该词被惩罚。

优势：

- 对已生成的高频词（如 n-gram 中的重复短语）进行动态控制，避免文本冗长。

2.4 n-gram 阻止

原理：

- 记录已生成的 n-gram 序列，避免模型再次生成。

实施细节：

1. 设置 n 的值，通常选择 $n = 3$ 或 $n = 4$。
2. 在每次生成新词时，检查其是否会导致生成的 n-gram 与已生成部分重复。

小红书账号：小黄搞AI

3. 如果重复，则跳过该词，选择概率次高的词。

2.5 引入多样性

在束搜索中引入多样性惩罚机制，鼓励生成不同的序列。

具体策略：

- 为每条束路径添加多样性奖励，抑制同一束内路径的相似度。
- 多样性激励公式： $P_{\text{adjusted}}(w) = P(w) - \lambda \cdot \text{similarity}(w)$ 其中 λ 为控制多样性的权重。

2.6 数据清洗

1. **消除重复：**
   - 对训练语料进行去重处理，删除重复率过高的句子。
2. **平衡语料：**
   - 保证不同主题和语言结构的分布均衡。
3. **增强多样性：**
   - 引入多样化的训练数据，如同义句、不同语言风格的句子。

三、实践建议

1. **调整解码参数：**
   - 根据任务需求，选择合适的温度、Top-k、Top-p参数。
   - **示例：**对摘要生成设置 $T = 0.8$ 、 $k = 50$ 、 $p = 0.9$ 。
2. **使用高级解码策略：**
   - 优先使用重复惩罚和 n-gram 阻止等策略。
3. **优化训练数据：**
   - 确保语料多样性，并减少人为重复。
4. **模型微调：**
   - 对模型进行特定任务的微调，增强其生成逻辑和语义流畅性。
5. **引入后处理：**
   - 在生成完成后，应用文本后处理算法检测并移除重复内容。

> 🌰 通过从调整解码策略、优化参数到改善训练数据和模型微调，可以有效减少大型语言模型的重复问题，提升生成文本的质量和用户体验。

70. 训练时数据长度不一致怎么办，以及如何优化训练速度

小红书账号：小黄搞AI

在深度学习领域，处理变长数据（如文本序列、语音信号、时间序列等）是一项核心挑战。数据长度不一致会导致批处理困难、资源浪费和训练效率低下。回答时应该深入探讨应对训练时数据长度不一致的具体方法，并详细阐述优化训练速度的策略。

一、引言

深度学习中的许多任务需要处理变长数据，例如自然语言处理中的句子长度不一、语音识别中的音频样本时长不同等。这些变长数据在处理时面临以下挑战：

- **批量训练依赖矩阵运算，而矩阵需要固定形状**；
- 数据填充或截断可能**影响模型的效率与性能**；
- 变长数据的动态特性可能导致资源分配不均，**增加训练成本**。

针对这些问题，提出合理的解决方法和优化策略，可以显著提升训练效率和模型性能。

二、数据长度不一致带来的问题

2.1 批处理复杂化

固定形状要求：深度学习框架（如TensorFlow和PyTorch）需要固定形状的张量作为输入，变长数据无法直接构成批次。

实现难度增加：变长数据需要额外的预处理（如填充或动态调整），增加实现复杂度和调试难度。

2.2 计算资源浪费

填充代价高：对较短的序列填充到最长序列的长度，导致大量无效计算。

内存占用不均：填充值占用显存和内存，降低硬件资源的利用率。

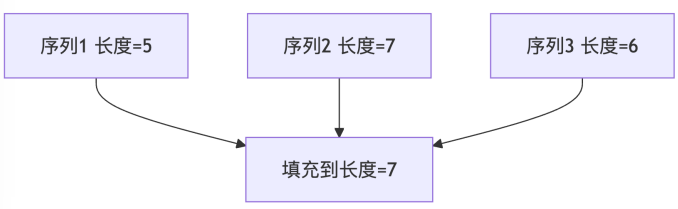
2.3 训练效率低下

计算瓶颈：填充值虽无信息，但仍会参与计算，浪费计算资源。

梯度传播问题：填充部分可能会导致梯度的无意义传播，影响训练效果。

三、处理变长数据的方法

3.1 填充（Padding）



原理：

将所有序列填充到同一长度（通常为批次中最长序列的长度），并使用掩码（mask）忽略填充部分的计算。

实施细节

小红书账号：小黄搞AI

- **如何填充**：在较短序列后添加零或其他占位符。
- **掩码使用**：对于填充部分，设置掩码矩阵，使其在损失计算和梯度更新中不参与运算。

示例：在Transformer中，掩码用于避免填充值影响注意力机制的计算。

优点	缺点
确保所有序列可以批量处理。	对填充量较大的批次，计算资源浪费严重。
易于实现，与主流框架兼容性好。	长短序列之间差距过大会影响模型性能。

3.2 截断（Truncation）

原理：

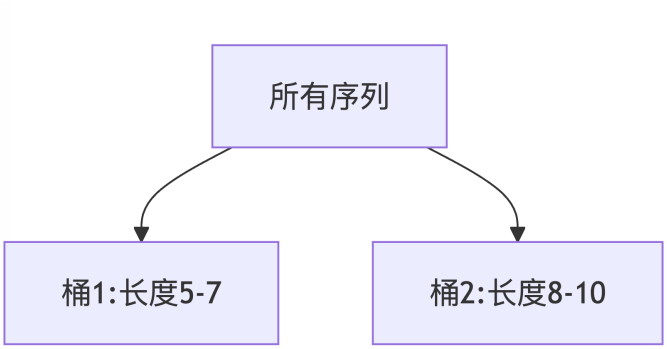
对于超出设定最大长度的序列，截断到指定长度，丢弃其余部分。

实施细节：

- **适用场景**：当长序列的尾部信息对任务无关紧要时（如情感分类）。
- **注意事项**：截断可能丢失关键信息，尤其在长序列依赖性较强的任务（如文档摘要生成）中。

优点	缺点
减少内存和计算开销。	丢失部分信息，可能影响模型性能。
实现简单，适合硬件受限场景。	不适合依赖全局信息的任务。

3.3 分桶（Bucketing）



原理： 将长度相近的序列分入同一批次，减少填充量，提升计算效率。

实施细节：

- **分桶策略**：预定义长度区间（如5-10、11-15），将序列按照长度分组。
- **动态分组**：在每次训练时动态调整桶的范围，以适应数据分布的变化。

优点	缺点
显著减少填充长度，提升硬件资源利用率。	数据加载和分组过程复杂，增加预处理时间。
减少填充值对模型性能的干扰。	随机性下降，可能导致批次间数据分布不均。

3.4 动态批处理

原理：

根据每个批次中最长序列的长度动态调整填充长度，避免全局填充导致的资源浪费。

小红书账号：小黄搞AI

实施细节：

- **实时调整**：在每次生成批次时，计算该批次最长序列的长度，并仅填充到该长度。
- **框架支持**：TensorFlow和PyTorch等框架支持动态计算图，适合这种方法。

优点	缺点
最小化填充，最大限度减少资源浪费。	实现复杂度高。
适应性强，适合序列长度分布变化较大的任务。	可能对硬件性能优化（如GPU并行）不够友好。

3.5 使用可变长度模型

原理：

选择天然支持变长输入的模型结构，如基于RNN或Transformer的模型。

实施细节：

- **注意力机制适配**：在Transformer中，注意力权重的计算可以通过掩码忽略填充值。
- **框架优化**：部分框架支持动态长度的RNN或Transformer（如TensorFlow的dynamic_rnn）。

优点	缺点
避免填充带来的额外计算。	实现难度较高。
保留变长数据的灵活性。	对硬件的依赖性较强。

四、优化训练速度的方法

4.1 数据预处理优化

提前生成掩码

- 在数据加载阶段**计算掩码，避免训练时重复计算。**
- 存储序列和掩码对，提高数据加载效率。

数据加载并行化

- **使用多线程或多进程加载数据，加快数据读取速度。**
- 利用框架内置的 `DataLoader` 提供并行读取功能。

4.2 硬件加速

使用GPU/TPU

- 利用**GPU的并行计算能力**加速矩阵运算。
- TPU特别适合处理大规模批量运算，适配模型需求。

混合精度训练

- 使用FP16（半精度浮点）训练，减少显存占用。

小红书账号：小黄搞AI

- 框架支持：如PyTorch的AMP（Automatic Mixed Precision）。

4.3 批次大小调整

增大批次大小

- 在硬件允许范围内增大批次大小，提高并行度。
- 使用 `gradient accumulation` 技术，模拟大批次的效果。

4.4 模型优化

简化模型结构

- 使用更高效的模型（如Lightweight Transformer）。
- 适当减少层数和隐藏单元，降低复杂度。

剪枝和量化

- 剪枝：移除冗余权重。
- 量化：将权重从浮点数降至整型。

4.5 软件优化

高效框架

- 启用TensorFlow的XLA或PyTorch的TorchScript优化模型图。

CUDA优化

- 使用cuDNN等库，最大化GPU加速效果。

> 🎁 处理训练时数据长度不一致和优化训练速度需要结合任务特点，采用多层次的方法。从填充、截断到硬件加速、软件优化的综合应用，可以显著提升效率，为深度学习模型训练提供更优解。

71. 什么是Prefill-Decode分离？为什么分离？

> 💡 PD分离（Prefill-Decode分离）是大模型推理优化的重要策略之一，涉及将推理过程中的 **Prefill（预填充）** 和 **Decode（解码）** 分开处理，以提升灵活性、效率和资源利用率。

71.1 Prefill 的作用

> **Prefill 是推理的第一阶段**，其作用主要是**处理输入的 Prompt**，即生成语言模型的上下文表示（hidden states）。

> 💎 **Prefill的特点包括：**

小红书账号：小黄搞AI

- **计算密集型**：Prefill 阶段主要依赖于 Transformer 的前向传播计算，**对 GPU 的算力要求较高**。
- **静态特性**：由于用户的输入 Prompt 通常是固定的，**可以以批量方式一次性处理多个样本**。这种批量处理的特点使 Prefill 在相同的计算资源下可以实现更高的吞吐量。

`Prefill 的输出：`

- 计算完成后，Prefill 会生成 Transformer 的激活值，主要包括 **Key-Value Cache（KV Cache）**。
- KV Cache 是一种存储机制，保存了注意力机制中的 Key 和 Value 向量，**供后续解码阶段重复使用，避免重复计算。**

71.2 Decode 的作用

> Decode 是**推理的第二阶段**，主要**负责生成输出文本**。每一步生成的 token 都需要依赖之前生成的 token 以及 Prefill 阶段的 KV Cache。

🌵 `Decode 的特点包括：`

- **访存密集型**：Decode 阶段主要涉及动态的逐步生成，每生成一个 token 都需要**访问存储的 KV Cache**，因此**对内存和带宽的依赖较高**。
- **动态特性**：Decode 是 token-by-token 的过程，每次生成一个新 token 都需要重新计算注意力分数和更新生成序列，**难以像 Prefill 那样进行批量处理**。

71.3 Prefill 和 Decode 的区别与优化方向

小红书账号：小黄搞AI

	Prefill	Decode
任务目标	处理 Prompt，生成 KV Cache	生成 token-by-token 的输出
计算模式	处理 Prompt，生成 KV Cache 高并行性、批量处理	串行处理、逐步生成
瓶颈	计算密集型，需要强大的 GPU 运算能力	访存密集型，需要快速访问 KV Cache
优化方向	提高吞吐量，利用批量输入提升效率	减少网络传输和 KV Cache 访问的开销

71.4 KV Cache 在 PD 中的作用

> KV Cache 是**连接 Prefill 和 Decode 的关键数据结构**。它在注意力机制中保存了每个输入 token 的 Key 和 Value，**用于后续生成的注意力计算**。

KV Cache 的主要作用

1. **减少重复计算：** 在 Decode 阶段，解码每个新 token 都需要与所有历史 token 交互。如果没有 KV Cache，模**型需要重新计算所有历史 token 的 Key 和 Value**，而 KV Cache 可以**直接复用 Prefill 阶段的结果，显著减少计算开销**。
2. **支持高效解码：** Decode 阶段**通过 KV Cache 直接检索历史 token 的信息**，与新生成的 token 交互，完成注意力计算。这种机制**支持逐步生成，并确保解码过程的一致性和高效**性。

KV Cache 的代价

- **存储开销：** KV Cache 的**大小与模型的隐藏层维度和序列长度成正比**。例如，LLaMA-65B 在 2K 输入长度下的 KV Cache 大小约为 5GB。
- **网络带宽需求：** 在分布式场景中，**Decode 阶段可能需要跨设备访问 KV Cache**，这对网络的吞吐量和延迟提出了较高要求。

71.5 PD 分离

小红书账号：小黄搞AI

> PD分离是指在大模型推理中，将**Prefill（预处理）和Decode（解码）**两个任务模块化处理

Prefill（P）

- 负责**解析用户输入（Prompt）**，进行一次性计算。
- 高度依赖矩阵运算和算力，可以通过**批量化处理实现高效利用**。

Decode（D）

- 按序**逐步生成输出（token-by-token）**。
- 强**依赖于访问内存的速度（KV缓存）和低延迟特性**。

🐋 **分离层次**

根据计算和硬件的解耦程度，PD分离可以分为以下层次：

- **Level -1**：**无分离**，P和D任务**完全融合在一个流程中完成**。
- **Level 0**：**逻辑分离**，P和D虽在软件上独立，但仍**使用同一硬件资源**。
- **Level 1**：**硬件内部分离**，例如在**同构GPU集群**中分别为P和D分配计算资源。
- **Level 2**：**分布式同构硬件分离**，**利用高速网络连接不同节点**，分别执行P和D任务。
- **Level 3**：**异构硬件分离**，例如使用CPU或FPGA处理P部分，GPU或专用加速器完成D部

🌵 **PD分离趋势思考结论**

1. 对大型企业而言，PD分离是一个不可逆的技术演进方向。这不仅是架构升级的需求，更是战略布局的必然选择。目前多数企业已经在这个方向上进行了不同程度的尝试，只是在**激进程度上有所区别**。
2. 中小企业则需要在**业务体量与架构复杂度之间寻找平衡点**，采取**务实的渐进式方案**。
3. 产业演进的下一个拐点可能在于**标准化解决方案的出现**。

> 参考：《从MoonCake看大模型推理的PD分离，太快会扯到蛋吗？》
> https://zhuanlan.zhihu.com/p/8056351077

72. 什么是PRM 过程监督奖励？OpenAI的秘密武器？

72.1 什么是 PRM?

过程监督奖励（PRM, Process Supervision Reward） 是强化学习的一种新方法，用于复杂推理任务中对多步骤生成过程进行细粒度的奖励反馈。与传统的结果监督奖励（ORM, Outcome Reward Model）仅在最终答案正确与否进行奖励不同，PRM 聚焦于每一步的中间结果，为每个推理步骤提供奖励估计。这种方法能够更好地优化模型在推理任务中的表现，尤其是在逻辑性要求高的场景，例如数学推理和推导。在 OpenAI 的 Let's verify step by step 论文中，**过程监督奖励（PRM，即为每一步推**

小红书账号：小黄搞AI

理提供反馈的奖励）比结果监督奖励（ORM，仅在最终结果上进行反馈的结果监督奖励）表现明显更好。

🎓 **PRM优势**

- **细粒度监督**：通过**为每一步生成提供反馈**，提升了复杂推理任务的整体质量。
- **过程优化**：对多步决策路径**进行奖励引导**，帮助模型找到更优解答路径。

72.2 如何构建 PRM 数据?

> 复杂推理任务的一个独特挑战是通常需要经过多个步骤 (multi-step) 才能生成最终的答案，这与传统的 RLHF 研究中的单步构造极其不同。**构建 PRM 数据的核心在于对每个中间步骤的奖励值进行标注，然而，由于逐步人工标注耗时高昂。**

🎓 **Deepseek 的 Math-Shepherd 提出一种自动化的过程监督数据生成流程：**

1. **基础路径采样**

   针对每个问题（Prompt），**从训练好的模型中生成一定数量的基础推理路径**（如 Math-Shepherd 中生成 15 条路径）。**这些路径代表模型对问题的初步解答方案**。

2. **中间步骤扩展**

   对于每个路径的中间步骤，**从当前步骤继续采样若干条完整路径（如每步采样 8 条路径），直到生成最终答案。**

3. **奖励估计**

   根据扩展路径中生成正确答案的比例，**为每个步骤生成奖励值**。例如：若从某一步继续扩展生成的路径中，正确率为 50%，则该步骤的奖励值为 0.5。

72.3 PRM 与 Q-Learning 的关联

> PRM 可以看作一种适配于**大模型的特殊 Q-Learning 方法**。其本质是**在推理路径的每个步骤进行奖励传播**，与强化学习中的价值函数更新有异曲同工之妙。

Q-Learning	PRM
通过 Bellman 方程计算长期回报，依赖状态-动作对的价值估计	在推理任务中，**奖励估计直接对应路径中间结果的正确率**，无需显式定义状态转移概率。

这种结合强化学习与生成模型的方式，使得 PRM 能够高效地学习复杂推理中的多步骤最优路径。

72.4 总结与未来趋势

小红书账号：小黄搞AI

🐋 **PRM在复杂推理任务中具有巨大潜力，其优点体现在：**

细粒度反馈：与结果监督相比，过程监督对每一步推理进行评价，**直接鼓励模型生成更连贯的逻辑链条**，而不是仅追求最终结果正确性。

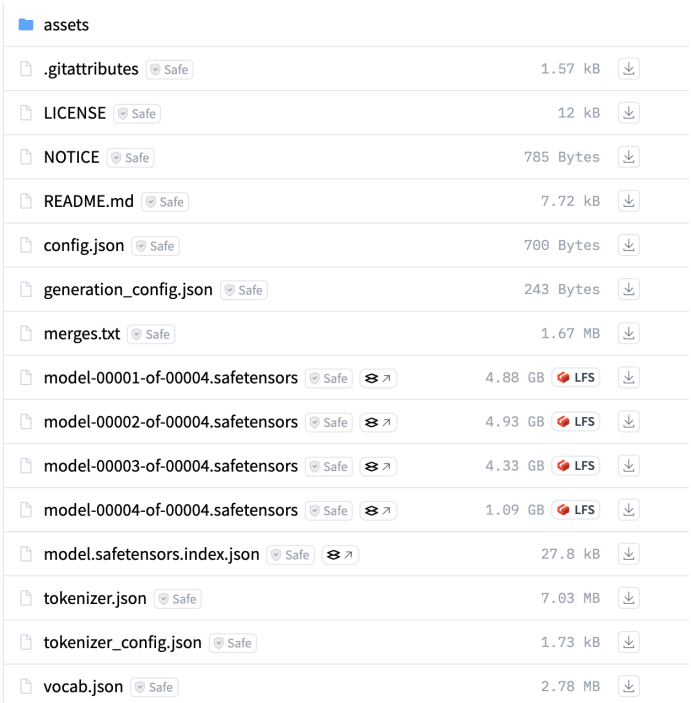
解释性增强：由于每一步推理都受到监督，过程监督能够更容易地生成人类可理解的逻辑链，从而**减少模型产生幻觉的可能性**。

负对齐税（Negative Alignment Tax）：在对数学领域任务的测试中，过程监督不仅提高了性能，**还减少了对齐过程中潜在的性能损失。**

👍 通过结合强化学习与生成模型的优点，PRM 不仅显著提升了模型推理能力，还为未来智能系统中的推理任务开辟了新思路。未来改进方向可能包括：

- **优化路径采样策略**：提高有效路径的生成效率；
- **引入更多多模态数据**：扩展 PRM 的应用场景；
- **结合其他强化学习技术**：如策略优化（PPO）或直接偏好优化（DPO）。

73. 从huggingface下载模型时有哪些文件?



🎓 **从huggingface下载模型时主要有四类文件**

📦**分词器相关文件**（如 `tokenizer.json` 和 `merges.txt`）：**用于预处理输入文本，将其转为 token。**

📦**模型权重文件**（如 `.safetensors`）：**核心文件，包含模型的参数。**

📦**配置文件**（如 `config.json`）：**定义模型结构和生成过程。**

📦**许可证与文档**（如 `LICENSE` 和 `README.md`）：**提供仓库的法律和使用说明。**

73.1 许可证和git文档相关

小红书账号：小黄搞AI

🌵 `.gitattributes`

💚 描述：Git 属性文件，用于**定义 Git LFS（Large File Storage）的设置**和其他版本控制属性。

💙用途：指定哪些文件使用 Git LFS 存储（例如 `.safetensors` 文件）。

`LICENSE`

💚 描述：**模型或代码的许可证文件**。

💙用途：规定此仓库内容的使用和分发规则。

`NOTICE`

💚 描述：法律声明或通知。

💙用途：**提供与模型或库的版权相关的信息**，可能涉及第三方依赖。

`README.md`

💚 描述：**仓库的自述文件**。

💙用途：通常提供关于模型、如何使用它的说明以及相关链接。

73.2 模型配置相关文件

🍁 `config.json`

💛 描述：**模型的配置文件**。

❤️ 用途：包含模型的架构细节，例如**层数、隐藏单元数、注意力头数等**。

`generation_config.json`

💛 描述：**生成配置文件**。

❤️ 用途：定义文本生成时的参数，如**最大生成长度、温度、Top-k采样等**。

73.3 分词器相关文件

🌻 `merges.txt`

💛 描述：BPE（Byte Pair Encoding）**分词器的合并规则文件**。

💚用途：**指定词汇表构建中合并子词单元的规则**。

`tokenizer.json`

💛 描述：**分词器的完整配置和词汇表**。

💚 用途：包含如何**将文本转换为模型可处理的 token**。

小红书账号：小黄搞AI

`tokenizer_config.json`

💛 描述：**分词器的配置文件**。

💚 用途：**定义分词器类型、特殊符号**（如 `[CLS]` 和 `[SEP]`）等。

`vocab.json`

💛 描述：**分词器的词汇表文件**。

💚 用途：**映射 token 与对应的整数 ID**。

73.4 模型权重相关文件

📘 `model-00001-of-00004.safetensors` 至 `model-00004-of-00004.safetensors`

💙 描述：**模型的分块权重文件**，使用 `safetensors` 格式存储。

💚 用途：**包含训练好的模型参数**，分块存储以便处理大文件。

`model.safetensors.index.json`

💙 描述：**权重文件的索引文件**。

💚 用途：**提供各个分块权重文件的位置映射信息**，便于加载。

74. 到底什么是端到端模型

74.1 什么是端到端模型

🌵 **端到端模型指的是一种直接由输入得到输出的过程**。在深度学习框架中，端到端模型不需要显式的中间步骤或手工设计的特征提取过程。简单来说，模型输入原始数据，经过神经网络的处理，最终直接输出结果，而不依赖于复杂的手工特征工程或多个模块的协作。

🍁 端到端模型的广泛应用伴随着深度学习的兴起，尤其是在深度神经网络能够有效地处理大规模数据并通过监督学习自动提取特征时。传统的机器学习方法虽然可以处理大量带标签的数据，但它们往往需要手工设计特征，且在面对复杂任务时效果有限。**相比之下，深度学习通过端到端训练，能够自动学习从输入到输出的映射关系，避免了人工特征设计的繁琐步骤。**

端到端模型的优势体现在它的“自监督学习”能力，尤其是在大模型（如GPT）中，利用自回归（next token prediction）学习突破了传统的监督学习瓶颈。大模型能够通过大量的未标注数据自动进行对齐，进而解决了很多NLP任务的挑战。自监督学习并非完全依赖人工标注数据，而是通过利用大量的原生数据（如自然语言文本）自我学习，使得模型能够进行有效的推理与预测。

小红书账号：小黄搞AI

74.2 端到端模型与传统模型的区别

传统的AI系统往往采用分模块的方式（Pipeline）进行任务处理，比如语音识别中，语音首先需要被转换为文本，然后再传给模型进行处理，最后将模型的输出转换为语音。与之相比，**端到端模型省略了中间步骤**，直接在一个大规模神经网络中进行所有的处理。这使得系统的效率大大提高，尤其是在实时性要求高的任务中（例如自动驾驶）。

然而，端到端模型也存在一些挑战：

系统复杂性：由于**端到端模型的内部结构通常难以解释，它更像是一个黑盒子**。相比之下，传统的Pipeline系统可以清楚地定义各模块之间的接口和处理逻辑。

可解释性差：大模型虽然在处理任务时表现出色，但其**内部决策过程的可解释性较差**。这是当前大模型研究的一个重要难题。

74.3 大模型的优势与应用场景

端到端模型的一个重要应用是跨模态大模型，如GPT模型处理文本、图像、语音等多模态数据。**只要有足够的原生数据，模型就能通过自监督学习实现跨模态任务，例如图生图、文生图等任务。**

实际应用中的优势：

减少复杂性：传统的**机器学习系统往往需要多个处理步骤**，包括特征提取、特征选择、模型训练等，过程繁琐且容易出错。而**端到端方法通过简化流程，不需要手动提取特征，模型可以自动学习最适合的特征和参数**。

提高性能：**端到端系统的训练是联合的，所有参数一起更新**，能够更好地捕捉到输入与输出之间的复杂非线性关系。通过联合优化，通常能够达到比分步式系统更好的性能。

自动化特征学习：在传统系统中，手工设计的特征往往受到先验知识的局限，而端到端方法依赖数据本身，通过训练自动学习特征。**这使得模型能够发现潜在的、有时难以察觉的关系和模式。**

灵活性和通用性：端到端学习不依赖于领域特定的知识，可以灵活地适应不同任务，**减少了针对不同任务的特征工程和模型调整的需求**。

无需人工标注：通过自监督学习，大模型可以**利用大量未标注的原始数据进行训练**，避免了人工标注的成本。

74.4 总结

端到端模型代表了AI任务定义和处理的一个重要发展阶段，尤其在深度学习和大模型的发展中得到了广泛应用。它能够通过自动学习从输入到输出的映射，避免了繁琐的特征工程和模块化处理，极大提

小红书账号：小黄搞AI

> 高了任务处理的效率，尤其在需要高实时性的任务中展现出巨大的优势。**虽然它存在可解释性差、系统复杂等问题，但随着技术的发展，端到端模型在多个领域的应用仍然具有巨大的潜力。**

🎓 **端到端学习带来了极大的便利和灵活性，但也有其局限性和挑战。**首先，虽然端到端方法减少了人工设计的复杂性，但它依然**依赖于大量的数据和计算资源**。尤其是在数据不充足的情况下，端到端学习可能难以表现出与传统方法相媲美的性能。

此外，**端到端学习的“黑盒性”问题也是一个不容忽视的挑战**。传统的机器学习方法，如决策树和线性回归，容易解释和理解模型的内部决策过程，而端到端模型，尤其是深度神经网络，往往难以进行直观的解释。这对于需要高透明度和可信度的领域（如医疗、金融等）是一个限制因素。

为了缓解这些问题，近年来的研究开始关注如何结合端到端学习与可解释性方法，例如可解释的神经网络（XAI）、模型压缩和迁移学习等。通过这些手段，我们可以进一步提升端到端学习在实际应用中的适应性和可操作性。

小红书账号：小黄搞AI

小黄搞AI-LLM面试问题

75. Transformer模型中计算量(FLOPs)的分析

75.1 计算量FLOPs估计方法

FLOPs，floating point operations，表示浮点数运算次数，衡量了计算量的大小。如何计算矩阵乘法的FLOPs呢？对于 $A \in R^{1\times n}, B \in R^{n\times 1}$ ，计算AB需要进行 n 次乘法运算和 n 次加法运算，共计 $2n$ 次浮点运算，需要 $2n$ 的FLOPS,对于 $A \in R^{m\times n}, B \in R^{n\times k}$ ，计算AB需要的浮点运算次数为 $2mnk$ 。

计算AB需要的浮点运算次数推算

对于矩阵相乘： $A \in \mathbb{R}^{m\times n}$， $B \in \mathbb{R}^{n\times k}$ 则 $C = AB \in \mathbb{R}^{m\times k}$

其中 $C_{ij} = \sum_{r=1}^{n} A_{ir} B_{rj}$ 。对于计算一个元素 C_{ij} ：

- 需要进行 n 次乘法运算（将 A_{ir} 与 B_{rj} 相乘）
- 需要进行 $n-1$ 次加法运算（将这 n 个乘积相加）

整个结果矩阵 C 有 $m \times k$ 个元素，每个元素需要 n 次乘法和 $n-1$ 次加法，因此：

- 总的乘法次数为： $m \times k \times n$
- 总的加法次数为： $m \times k \times (n-1)$

在大多数情况下，为了简化近似，人们会认为加法次数与乘法次数相当(当 n 很大时，$n-1 \approx n$)，因此总的浮点运算量(FLOPs)大约为 $2mnk$ （其中包含约 mkn 次乘法和 mkn 次加法）。

75.2 Transformer结构中FLOPs计算

下面将对所提及的计算步骤进行更为详细的解释与展开。这些步骤来源于Transformer中一层（单层）的计算分析，包括Self-Attention部分和MLP（Feed-Forward）部分，以及最终输出logits的计算。假设我们有以下记号：

- b : 批次大小 (batch size)
- s : 序列长度 (sequence length)
- d_{model} : 隐藏层维度
- d_{ff} : 前馈层扩张后的维度，一般为4倍 d_{model}
- h : 注意力头数 (heads)

小红书账号：小黄搞AI

- $d_k = d_{model}/h$: 每个注意力头的维度
- V : 词表大小 (vocab size)

在实际实现中，这些计算往往合并在一起执行，但为方便计算FLOPs和参数量，我们将各个步骤独立拆开说明。

Input输入层分析

💎 在一次训练迭代中，假设输入数据的形状为 $[b, s]$,经embedding层得 $[b, s, h]$ ，矩阵乘法的输入和输出的形状为 $[b, s, V] \times [V, h] \rightarrow [b, s, h]$,计算量为 $2bshV$

假设输入数据的形状为 $[b, s]$ 。其中：

- b 是batch size，即一次输入的样本数。
- s 是序列长度（例如处理一句话中有s个token）。

每个输入样本中的每个位置（token）对应着一个词表索引，这个索引指向词典中大小为 V 的词表，即词表有 V 个可能的词。

Embedding层的本质：
通常，embedding层会将每个词（token）的索引映射到一个长度为 h 的向量（embedding向量）。

- 词嵌入矩阵 $W_{emb} \in \mathbb{R}^{V \times h}$
- V 是词表大小。
- h 是embedding的维度（隐藏维度）。

直观来看，embedding矩阵中的每一行是一个词的向量表示。当给定一个词的索引时，就直接取出该行作为该词的embedding向量。

将Embedding表示为矩阵乘法：
在实际实现中，Embedding查表是通过索引取行完成的，这个操作本身不像传统的矩阵相乘那样显式地执行 V 次乘法和加法。但如果我们将"取embedding"的过程用矩阵乘法的等价形式来表示，就可以得到一个FLOPs的估计方法：

- 将输入的词表索引用**独热(one-hot)编码**表示：
  对于输入的 $[b, s]$ 中的每个元素（一个token ID），用维度为 V 的one-hot向量来表示。
  则输入可以看作是 $[b, s, V]$ 的三维张量，每个位置都是一个one-hot向量（这里原本的 $[b, s]$ 映射为 $[b, s, V]$ 后会很大，但这是一种理论上的表示方法，用于FLOPs计算近似）。
- 在这种表示下：
  - $[b, s, V]$ 表示batch中所有tokens的独热表示
  - $W_{emb} \in [V, h]$ 是embedding矩阵

小红书账号：小黄搞AI

- 将 $[b, s, V]$ 与 $[V, h]$ 相乘，就得到 $[b, s, h]$ 的输出：$[b, s, V] \times [V, h] \rightarrow [b, s, h]$

在这个运算中，每个 $[b, s, h]$ 位置的结果是由长度为 V 的向量（one-hot编码）与 $V \times h$ 矩阵中的对应列进行点积得到的。

计算FLOPs的细节：
对于矩阵乘法 $A \in \mathbb{R}^{b \times s \times V}$ 和 $W_{emb} \in \mathbb{R}^{V \times h}$，计算结果 $C = AW_{emb}$ 的形状是 $[b, s, h]$。

- 对于产生 C_{ij} 的每个元素（这里 i 对应batch内第几个样本与序列位置组合，j 对应embedding维度索引，共有 $b \times s \times h$ 个元素），需要进行：
  - V 次乘法
  - $V - 1$ 次加法（将这 V个乘积相加）

当 V 很大时，$V - 1 \approx V$，为简化计算，往往取总运算约为 $2V$ FLOPs（V 次乘法 + V 次加法 $\approx 2V$）。因此，每计算输出矩阵中的一个元素，需要约 $2V$ 次浮点运算。整个 [b, s, h] 的输出有 $b \times s \times h$ 个元素。对每个元素进行约 $2V$ 次运算：
总FLOPs $\approx b \times s \times h \times 2V = 2bshV$

🍋 Self-Attention部分计算展开

计算 Q，K，V 矩阵

操作：输入形状为 $[b, s, d_{model}]$，通过三个独立的线性层映射得到 Q, K, V 三个张量。

- 每个线性变换将 $[b, s, d_{model}]$ 映射到 $[b, s, d_{model}]$（实际上是 $[b, s, h \times d_k]$，但合并后仍为 $[b, s, d_{model}]$）。
- 对于每个线性层（如计算Q的那一层），矩阵乘法的形状为 $[b \times s, d_{model}] \times [d_{model}, d_{model}] \rightarrow [b \times s, d_{model}]$。转换为FLOPs计算时：
  $FLOPs \approx 2 \times b \times s \times d_{model} \times d_{model}$（2倍是因为矩阵相乘需考虑乘法+加法两类操作）
- 因为Q、K、V各自有一套这样的映射，所以总计约为：
  $3 \times (2 \cdot b \cdot s \cdot d_{model}^2) = 6bsd_{model}^2$

计算注意力权重 $A = QK^T$

操作：

- Q 形状为 $[b, s, h, d_k]$（通常实现中会reshape Q为这种四维形式，但总元素不变），K 同理 $[b, s, h, d_k]$。
- 为计算注意力分数，需要对每个头计算 QK^T，输出形状为 $[b, h, s, s]$。
- 对单个头而言，$Q \in [b, h, s, d_k]$ 与 $K \in [b, h, s, dk]$ 相乘时：$[s, d_k] \times [d_k, s] \rightarrow [s, s]$
- FLOPs约为 $2 \times s \times s \times d_k$（对单头）。有 h 个头并 b 个batch：
  $\text{FLOPs}_{QK^T} \approx 2 \cdot b \cdot h \cdot s \cdot s \cdot d_k$

小红书账号：小黄搞AI

- 由于 $d_k = d_{model}/h$，代入可得到约 $2bs^2d_{model}$ FLOPs。

将注意力分数与V相乘 (即计算 $\text{Attn}(Q,K,V) = \text{softmax}(QK^T)V$)

操作：

- 已获得的注意力矩阵 $[b,h,s,s]$ 与 $V \in [b,h,s,dk]$ 相乘，输出形状为 $[b,h,s,d_k]$。
- 对单个头而言： $[s,s] \times [s,d_k] \to [s,d_k]$ FLOPs约为 $2 \times s \times s \times d_k$
- 考虑 b,h： $\text{FLOPs}_{\text{Attn}*V} \approx 2bhs^2d_k = 2bs^2d_{model}$

注意力输出的线性映射

操作：

- 将拼接后的多头注意力输出 $[b,s,d_{model}]$ 通过一层线性层映射回 $[b,s,d_{model}]$。
- 矩阵乘法 $[bs,d_{model}] \times [d_{model},d_{model}] \to [bs,d_{model}]$， $\text{FLOPs} \approx 2 \cdot b \cdot s \cdot d_{model}^2$

小结Self-Attention计算量

将上述四个部分加在一起，典型总FLOPs约为：

- Q,K,V 投影: $6bsd_{model}^2$
- QK^T : $2bs^2d_{model}$
- $\text{Attn} * V$: $2bs^2d_{model}$
- 输出线性映射: $2bsd_{model}^2$

当序列长度 s 很长时， s^2 项（QK^T 和 $Attn * V$）往往会主导计算复杂度。

MLP (Feed-Forward)部分计算展开

$\text{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2$ ，这里常用设定 $d_{ff} = 4d_{model}$

第一个线性层 ($d_{model} \to d_{ff}$)

操作：

$[b,s,d_{model}] \times [d_{model},d_{ff}] \to [b,s,d_{ff}]$

FLOPs约为： $2 \cdot b \cdot s \cdot d_{model} \cdot d_{ff} = 2bsd_{model}(4d_{model}) = 8bsd_{model}^2$

第二个线性层 ($d_{ff} \to d_{model}$)

操作：

$[b,s,d_{ff}] \times [d_{ff},d_{model}] \to [b,s,d_{model}]$

FLOPs约为： $2 \cdot b \cdot s \cdot d_{ff} \cdot d_{model} = 8bsd_{model}^2$

(两次线性映射加起来约 $16bsd_{model}^2$ FLOPs；实际中非线性激活ReLU等不计入太多额外FLOPs)

小红书账号：小黄搞AI

将Self-Attention和MLP计算量相加

一个典型的Transformer层计算量为：

- Attention部分：约 $\underbrace{6bsd_{model}^2}_{Q,K,V}+\underbrace{2bs^2d_{model}}_{QK^T}+\underbrace{2bs^2d_{model}}_{\text{Attn}*V}+\underbrace{2bsd_{model}^2}_{\text{out linear}}$
- MLP部分：约 $16bsd_{model}^2$

当 s 不大时，d_{model}^2 项占主导；当 s 较大时，s^2 项（注意力的 QK^T 和 $Attn*V$ ）会成为计算瓶颈。

🌽 Output层 (将隐藏向量映射为词表大小)

在训练语言模型或进行推断时，最后往往需要将每个位置的隐状态向量 $[b,s,d_{model}]$ 映射到词表维度 $[b,s,V]$ 来得到每个词的预测分布(logits)：

操作：$[b,s,d_{model}]\times[d_{model},V]\to[b,s,V]$

FLOPs约为：$2\cdot b\cdot s\cdot d_{model}\cdot V$ ，这在词表 V 很大时也可能相当可观。

全部合计（一次训练迭代）

对于一个拥有 LL 层的Transformer，输入数据为 $[b,s]$ 的tokens：

1. 首先经过Embedding $[b,s]\to[b,s,d_{model}]$ 这部分FLOPs已在之前解释过(约 $2bshV$ ，仅用于理论比较)。
2. 每层包括Attention和MLP的计算，将 L 层的计算相加得到总计算量。
3. 最终输出层(logits计算)： $[b,s,d_{model}]\times[d_{model},V]\to[b,s,V]$, $[b,s,V]$ 约 $2bsd_{model}V$ FLOPs。

所以整次训练迭代（不考虑反向传播和优化器更新）中，前向计算FLOPs会是Embedding、LL层Transformer block、以及最终输出层的综合。

上面展示了各个步骤背后的矩阵乘法形状变化和FLOPs数量估计的原因。这种拆解有助于理解大模型（如GPT系列）在训练和推理时的计算开销来源，以及在实践中为什么注意力机制对长序列输入的计算成本如此高昂。

76. Transformer模型中计算量(FLOPs)和参数的表格分析

下面的表格总结了Transformer中各个结构（以decoder-only框架为例）的参数量与计算量(FLOPs)估计。为简化起见，表中给出的参数量与计算量为典型情形下的近似值，实际数值可能会因具体实现与超参数变化而有所调整。

> **符号说明**：
>
> - L : 层数

小红书账号：小黄搞AI

- d_{model}：隐藏层维度
- d_{ff}：FFN的扩张维度（通常为 $4 \times d_{model}$ ）
- V：词表大小
- B：批次大小（batch size）
- n：序列长度（sequence length）
- h：注意力头数（通常满足 $d_{model} = h \cdot d_k$ ）
- 所有计算量(FLOPs)为前向传播的近似值（未包含反向传播和梯度计算），并使用典型的 $2\times$ 乘加运算计数方式（一次点积计算为2次FLOPs：一次乘法和一次加法）。

1. 各组件参数量与计算量对比表

组件	参数量 (约)	前向计算 FLOPs (约)	备注
词嵌入 (Embedding)	$V \times d_{model}$	Embedding查表本身计算量极低；输出logits阶段：$2 \times B \times n \times d_{model} \times V$ ***FLOPs***	输入token索引转换为向量时几乎无矩阵乘法；最后输出层（logits计算）需要大规模矩阵乘法。
Self-Attention	$\approx 4 \times d_{model}^2$	1. Q,K,V投影：$3 \times 2 \times B \times n \times d_{model}^2 \approx 6 \times B \times n \times d_{model}^2$ 2. QK^T 求注意力得分：$\approx 2 \times B \times n^2 \times d_{model}$ 3. 将注意力得分与V相乘：$\approx 2 \times B \times n^2 \times d_{model}$ 4. 输出投影：$\approx 2 \times B \times n \times d_{model}^2$	注意力计算中，QK^T 和 $Attn * V$ 是随序列长度 n 二次增长的主要来源。当序列较长时，注意力部分的计算量(O(n²))可能成为主要瓶颈。
MLP(FFN)	$\approx 8 \times d_{model}^2$ (若 $d_{ff} = 4d_{model}$)	前两层全连接： 1. $d_{model} \to d_{ff}$ ：$2 \times B \times n \times d_{model} \times d_{ff}$ 2. $d_{ff} \to d_{model}$: $2 \times B \times n \times d_{ff} \times d_{model}$ 合计 $\approx 16 \times B \times n \times d_{model}^2$	FFN的计算量与序列长度 n 成线性比例增长 $O(n)$ ，参数量通常比注意力部分更大，但计算复杂度对比注意力的 n^2 关系更为温和。
LayerNorm (每层2个)	$2 \times d_{model}$ 参数量（可忽略）	FLOPs可忽略不计	相比Attention和MLP，LayerNorm的参数和计算成本都很小，一般不计入主要开销。

小红书账号：小黄搞AI

2. 每层总结

- **每层参数量 (不含词嵌入)**：
  Attention参数量约为 $4 \times d_{model}^2$，FFN参数量约 $8 \times d_{model}^2$，两者合计约 $12 \times d_{model}^2$（LayerNorm可忽略不计）。
- **每层计算量 (前向)**：
  当序列长度 n 较大时，Attention的 $O(n^2)$ 部分（QK^T 与 $Attn * V$）往往主导计算复杂度。FFN则是 $O(n)$。因此，对于较大 n，**Attention计算量常常超过FFN，但FFN参数量却更大。**
- **Embedding与Logits阶段**：
  Embedding本身参数量为 $V \times d_{model}$，当词表 V 巨大时，这块参数量可能超过所有层叠加。但从单层内部来看，这一部分不属于每层重复的参数。输出logits计算为 $2 \times B \times n \times d_{model} \times V$ **FLOPs**，随词表大小V呈线性增长。

77. 什么是交叉熵损失函数？大模型的哪里有交叉熵损失函数？

77.1 基础概念

概率分布与随机变量

在统计学与概率论中，我们常常关心某一事件发生的概率分布。例如，对于一个分类问题，如果我们要对一张图片中的物体进行分类，模型会输出这张图片属于各个类别的概率分布。

这些概率可以用向量表示，如：$p = (p_1, p_2, \ldots, p_C)$

其中 p_i 表示模型对类别 i 的预测概率，C 是类别数，并且 $\sum_{i=1}^{C} p_i = 1$。

信息论中的熵概念

在信息论中，“熵”（Entropy）是用来衡量随机变量不确定性的一种度量。如果我们有一个真实的分布 $q = (q_1, q_2, \ldots, q_C)$，它表示真实标签（或真实世界）关于类别的概率分布，那么这个分布的熵可以定义为：$H(q) = -\sum_{i=1}^{C} q_i \log q_i$

这里，log 通常为自然对数，但也可使用2为底或者10为底的对数，不同底数只会影响结果的比例不变性。当所有概率都集中在一个类别上（如真实标签的情况：真实分布通常是 "one-hot" 的，如某个类的概率为1，其余类为0），熵会趋近于0，因为这个分布是完全确定的。

小红书账号：小黄搞AI

🌻 相对熵（KL散度）与交叉熵

当我们有两个不同的概率分布 q 与 p 时，我们希望**衡量它们之间的差异程度。一个常用的度量是KL散度**（Kullback–Leibler divergence）：$D_{KL}(q\|p) = \sum_{i=1}^{C} q_i \log \frac{q_i}{p_i}$

KL散度度量了从分布 p 到分布 q 的额外信息开销。然而，它并不是对称的，而且在深度学习中求最优参数时更常用的是**交叉熵**（Cross-Entropy）来作为优化目标。

交叉熵定义为：$H(q,p) = -\sum_{i=1}^{C} q_i \log p_i$

交叉熵可以理解为：“**用分布 p （模型预测分布）来编码分布 q （真实分布）所需要的期望信息量**”。当 q 是真实标签的one-hot分布时，比如对于真实类别 i 来说 $q_i = 1$ 而对其他类别 $j \neq i$ 来说 $q_j = 0$，则交叉熵简化为：$H(q,p) = -\log p_i$

也就是说，对于one-hot的真实分布，交叉熵损失就是取预测分布中正确类别的概率的负对数。当模型非常自信且正确（即 $p_i \approx 1$ ），那么 $logp_i \approx 0$，表示模型几乎没有信息损耗和不确定性。当模型不确定（p_i 接近较小值）时，$-logp_i$ 会非常大，损失就很高。

77.2 深入理解

> **熵(Entropy)** 是衡量一个分布自身不确定性的量度。
>
> **交叉熵(Cross-Entropy)** 则是用一个分布 p (如模型的预测分布）去描述另一个分布 q（如真实分布）所需要的额外信息量。当真实标签的分布 qq 是“one-hot”时（即某一类为1，其余为0），交叉熵就大大简化了。

💛 例子说明

假设你有一个分类问题，例如“猫”与“狗”的二分类（仅为说明问题简单化），真实情况是图像中确实是一只“猫”，那么真实分布 q 是：$q(\text{猫}) = 1, \quad q(\text{狗}) = 0$

这个分布可以理解为：“真实世界很确定这张图片是猫，没有任何不确定性。”

现在，你的模型给出预测分布 p 可能是：$p(\text{猫}) = p_{\text{猫}}, \quad p(\text{狗}) = 1 - p_{\text{猫}}$

如果你的模型很自信地认为图片是猫，那么 $p_{\text{猫}}$ 会很接近1；如果你的模型很不确定或者错误，那么 $p_{\text{猫}}$ 会接近0。

交叉熵的定义： $H(q,p) = -\sum_{i} q_i \log p_i$

对于二分类（猫、狗）例子，上式中只有两个类别 {猫,狗}，

$H(q,p) = -[q(\text{猫}) \log p(\text{猫}) + q(\text{狗}) \log p(\text{狗})]$

小红书账号：小黄搞AI

但由于真实标签是 one-hot 的，我们知道：$q(\text{猫}) = 1, \quad q(\text{狗}) = 0$

代入后可以得到：$H(q,p) = -[1 \cdot \log p(\text{猫}) + 0 \cdot \log p(\text{狗})] = -\log p(\text{猫})$

为什么这么定义有意义？

从信息量角度理解

信息论中，对于一个事件 x ，如果其发生概率为 $p(x)$，那么该事件带来的“信息量”是 $-\log p(x)$。**事件越罕见（概率越低）**， $-\log p(x)$ 越大，**对你来说信息越“新鲜”，也就是不确定性越高**。当你的模型预测的类别正是正确答案，而它的预测概率接近1，那么 $-\log p(\text{正确类})$ 就接近0，说明模型对这个结果几乎毫无不确定性，也就几乎没有信息损失。

- **举例**：
  假设这个二分类问题中，正确答案是“猫”：
  a. 如果你的模型预测 $p(\text{猫}) = 0.9$，那么损失是 $-\log(0.9) \approx 0.105$。这个值比较小，说明模型预测得较准确。
  b. 如果你的模型预测 $p(\text{猫}) = 0.1$，那么损失是 $-\log(0.1) \approx 2.302$，这是一个很大的值。为什么？因为模型几乎“不相信”正确答案，这种错误代价就会变得很大。

从熵到交叉熵的过渡：

熵 $H(q) = -\sum q_i \log q_i$ 是在真实分布 q 本身下事件的不确定性衡量。如果 q 是one-hot，那么 $H(q) = 0$ （因为真实情况毫无不确定性）。

当你用一个有误差的分布 p 去描述 q 时，就会多出额外的信息消耗，这个额外的信息消耗正是由交叉熵度量：

- $H(q,p) = H(q) + D_{KL}(q\|p)$
- 当 q 是one-hot时， $H(q) = 0$，所以 $H(q,p) = D_{KL}(q\|p)$
- 在这种情况下，最小化交叉熵等价于最小化KL散度，也就是让 p 去逼近 q 。对one-hot标签而言，这就变成单纯地让 p 正确类 $p_{\text{正确类}}$ 越接近1越好，从而使 $-\log p_{\text{正确类}}$ 越小越好。

77.3 为什么用交叉熵作为损失函数？

易于优化

使用交叉熵损失函数会使得在对模型参数进行梯度下降优化时形成良好的梯度信号。因为对数函数的导数特性，它能在预测概率与真实标签差距较大时提供较大的梯度更新信号，从而加快训练收敛速度。

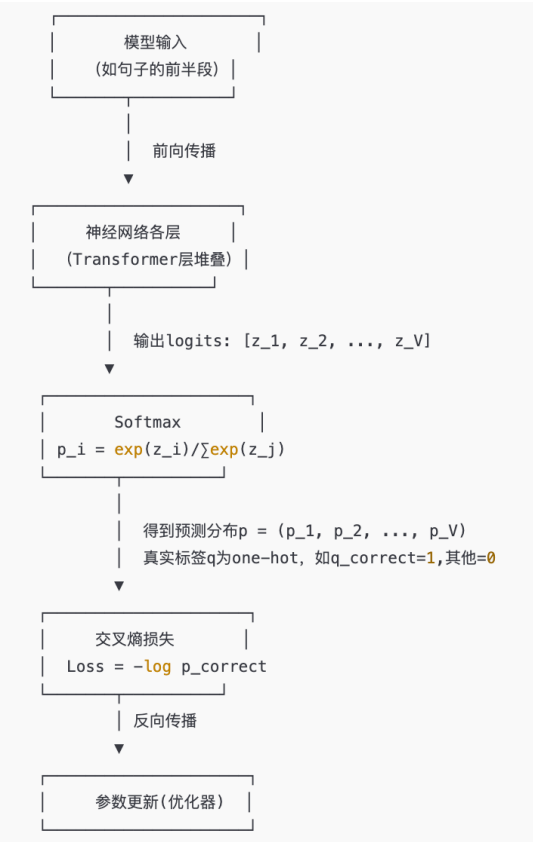
与模型输出的概率分布直接对应

神经网络的最后一层常通过Softmax函数将原始的分值（logits）转化为概率分布，**交叉熵直接将真实分布与预测分布进行对照，是最自然的度量方法。**

小红书账号：小黄搞AI

77.4 大模型中的交叉熵损失函数在哪体现？

在深度学习中，无论是语言模型（如GPT、BERT）还是图像识别模型（如ResNet），在训练过程中都会使用某种形式的损失函数来度量模型预测与真实目标的偏差。对于分类问题，**交叉熵损失**是最常用的损失函数之一。



以大语言模型（Large Language Model, LLM）为例说明：

模型任务性质

大语言模型的核心训练任务通常是“给定上一词（或上一句）的上下文，预测下一个词的概率分布”。在训练数据集中，我们有大量的文本序列。例如一句话分词后为：$w_1, w_2, w_3, ..., w_n$

模型在给定 $(w_1, ..., w_{t-1})$ 的前提下预测 w_t 的概率分布。

实际上，真实的下一个词是已知的（因为在训练集中，这些文本就是已标注的“真实序列”）。那么，我们就可以将真实的下一个词看作有一个"one-hot"分布（即正确词的概率为1，其他词的概率为0），而模型预测出的分布是 p 。

交叉熵损失在大模型训练中的位置

在模型的前向传播阶段，得到输出层的logits（未归一化分值）。通过Softmax函数得到对词典中每个词的预测概率。然后将真实词的one-hot分布与预测概率分布套用交叉熵公式计算损失：

$$Loss = H(q,p) = -\sum_{i=1}^{V} q_i \log p_i$$

对于one-hot标签，仅剩正确词对应的项：

$$Loss = -\log p_{\text{正确词}}$$

对于整篇训练数据，通常会对各个位置的预测损失求平均作为最终的训练目标函数。

通过损失反向传播更新参数

计算出交叉熵损失后，就可以通过反向传播（Backpropagation）计算出对模型参数（如神经网络中的权重和偏置）的梯度。优化器（如Adam或SGD）将利用这些梯度来迭代更新参数，使预测分布 p 趋于接近真实分布 q 。经过多次迭代训练后，模型对下一词的预测将越

小红书账号：小黄搞AI

来越准确。对于整篇训练数据，通常会对各个位置的预测损失求平均作为最终的训练目标函数。

💛 **交叉熵损失函数**是衡量两个概率分布之间差异的重要工具，它将真实分布和模型预测分布进行对比，通过负对数似然度量预测的置信度和偏差程度。

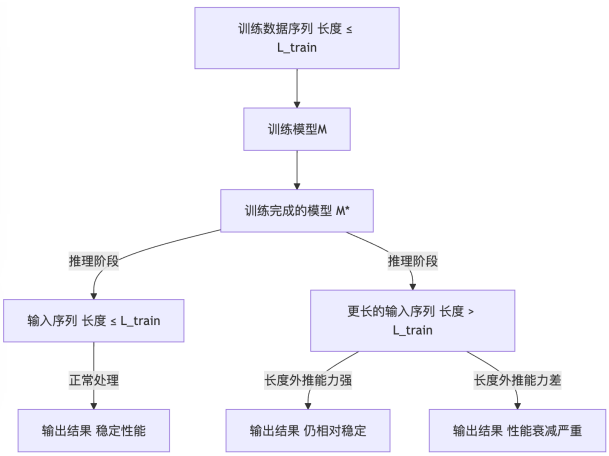
在大模型训练中，交叉熵损失函数典型地用于计算预测下一个词（或分类任务中预测正确类别）的误差。通过最小化交叉熵损失，模型的参数会逐渐被优化，使预测概率分布更加逼近真实分布，从而实现对训练任务的更好拟合和更高预测精度。

78. 什么是长度外推？

🍁 在自然语言处理和序列建模任务中，模型（如Transformer）通常是在有限的最大序列长度（如512或1024个token）上进行训练的。这是因为显存、时间和数据等资源的限制，导致训练和微调过程通常仅在固定长度的序列上进行。然而，**在真实应用场景中，我们经常希望模型能够处理比训练时更长的输入序列，例如更长的文本、长文档摘要、代码文件或长视频字幕的理解。**

78.1 长度外推的概念

“长度外推（Length Extrapolation）”是指模型在推理（inference）阶段处理比其在训练过程中所见过的最大序列长度更长的输入时，仍能保持较好的性能和稳定性。这好比一个模型“学会”在训练时只看了较短片段，却能在推理时对更长的全文进行较为准确和连贯的理解与生成。



训练过程仅在最大长度 `L_train` 下完成；

推理时，如果输入序列长度不超过 `L_train`，模型可稳定工作；当输入序列长度大于训练时长度（在 `L_train` 之外），如果模型具备长度外推能力，则它在 `E->G` 路径中表现出相对稳定的性能。否

🥥 **问题的根源**

固定最大序列长度

绝大多数Transformer模型在训练时**被限制在特定的最大序列长度L以内**。训练时模型学习到的注意力机制和位置编码往往对这个最大长度隐含假设。

位置编码与长度限制

传统位置编码（如正弦位置编码或可训练位置embedding）在序列长度超过训练时所见的最大值后，**会出现未知或无意义的偏置，从而影响模型对更长序列的理解。**

模型不具备天然的线性外推能力

小红书账号：小黄搞AI

则，`E->H` 路径显示了可能出现的性能严重衰减问题。

许多深度模型在面对比训练数据分布更“远”的输入（如超长序列）时性能往往显著下降，因为它们**缺乏在这种条件下合理推断的先验。**

78.2 长度外推的重要性

为什么长度外推很重要?

扩展应用场景：在实际应用中，我们可能需要处理非常长的文档、书籍、长篇维基百科条目或数以千计token的代码，这远超模型的训练长度限制。

提高模型的实用性与灵活性：具备长度外推能力的模型可以无缝适应各种输入长度，不必在每次遇到更长文本时重新对模型进行微调。

减少资源成本：不需要训练更长序列的专门模型，即可在推理阶段灵活应对长序列处理。

长度外推是指模型在推理中面对超出训练长度的输入时依然能保持合理性能的能力。它对于提高模型实用性和适应真实世界中多变的输入长度至关重要。

79. ALiBi (Attention with Linear Biases)思路是什么?



> 线性偏置计算：$b_{i,j} = \alpha * (j - i)$，$\alpha$ 为常数或可设定比例
>
> 无须位置嵌入，仅根据 (i, j) 距离线性添加偏置

在上图中，“Linear Bias”部分说明了如何计算偏置，整个流程中，关键是 $QK^T + LinearBias$ 这一步，它为传统注意力机制提供了一种依赖距离的线性削弱策略。

79.1 位置编码

传统的Transformer模型（如BERT、GPT）在处理序列数据时，需要引入位置信息。常用的方式包括：

绝对位置嵌入（Positional Embedding）

在输入的embedding中加入**固定或可训练的向量**来表示位置。例如Sinusoidal位置编码或可学习的位置embedding。

相对位置编码（Relative Positional Encoding）

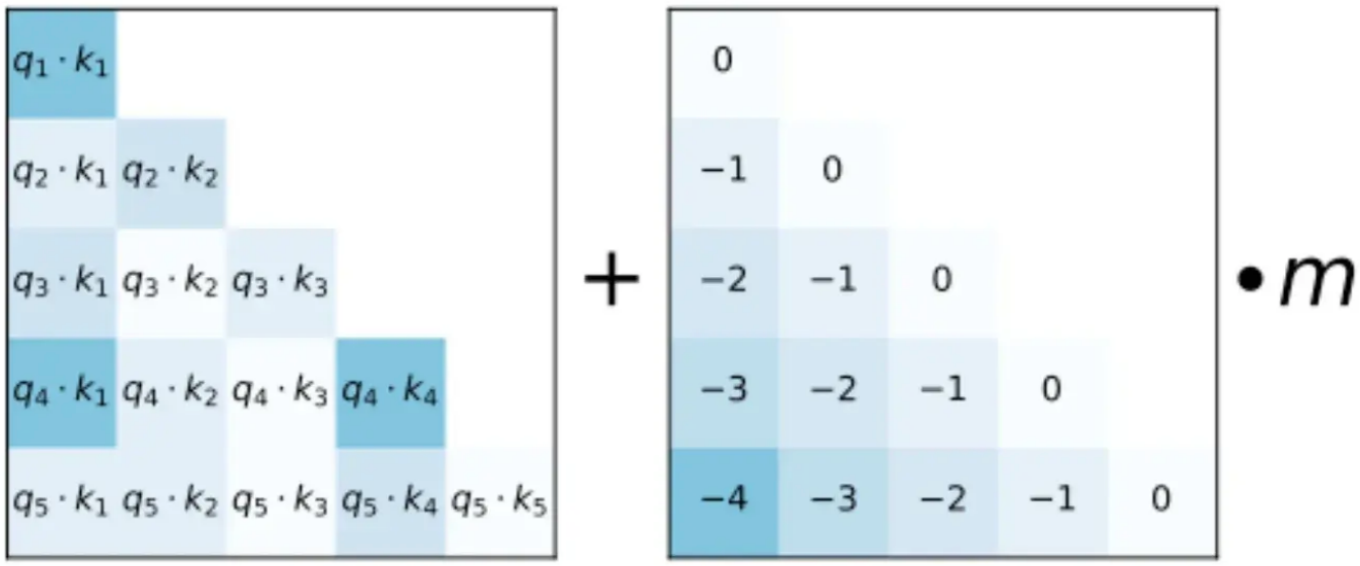
在计算attention时**依据query和key的相对位置差**来添加偏置，使模型能够更好地处理局部依赖和长距离信息。

小红书账号：小黄搞AI

然而，这些方法在**扩展序列长度**（例如，将训练中从未见过的长度应用到推断时）存在性能退化问题。**模型对训练时序列最大长度存在隐含假设，一旦在推理阶段遇到更长的序列，效果会显著下降。**

79.2 ALiBi的核心思想

ALiBi (Attention with Linear Biases) 提出了一种无需显式位置嵌入的方法，它在注意力机制的计算中直接对注意力分数（attention logits）加入一种与位置距离成线性关系的偏置，使得模型对更远位置的token赋予较低的权重。



关键特征

线性偏置（Linear Bias）：对于Transformer中的自注意力层，在计算 $Attention(Q, K, V) = softmax(QK^T / sqrt(d_k)) * V$ 时，ALiBi在 QK^T **的logits上直接添加一个与token距离呈线性衰减的偏置项。**

具体地，若 i 表示query位置、j 表示key位置，则偏置项 $b_{i,j}$ 通常按 $(j - i)$ 的距离给出一个线性下降的分值。这样，当 $j < i$（key在query前面）时，**该key与query的距离越远，logits的得分会越低。**这种线性偏置没有固定长度限制，更长的输入序列只是在此线上继续延展，模型不需要为此重新学习位置编码。

无需训练新的位置参数：不同于可训练的位置嵌入，ALiBi只是在**注意力打分中内置一条线性规则**，不需要在长序列推理时修改参数，也无需重新训练。**这样模型在训练时只需在相对较短序列上完成训练，在推理时就可以较为平滑地拓展至更长序列。**

无缝扩展长度：由于ALiBi的偏置是线性且基于相对距离，不依赖训练中出现过的最大长度，所以在推理阶段可直接将序列长度扩展到训练时未见过的范围，模型依然能保持较好的性能。

优点：

小红书账号：小黄搞AI

- **更好的长度扩展性**：比起传统位置编码方法，ALiBi在面对比训练时更长的序列长度时性能衰减更慢。
- **实现简单**：只需要在注意力计算中添加简单的线性偏置，无需额外的embedding参数。
- **适应性强**：适用于多种变体的Transformer模型。

ALiBi的思路是将位置信息的引入方式从复杂、固定或训练的嵌入向简单的相对距离线性偏置转变。这种方式使模型在面对更长文本时仍能保持合理的注意力分布，从而在测试时对更长序列具有较好的扩展性和稳健性。

80. 为什么需要增量预训练？

> “预训练学习知识，指令微调学习格式，强化学习对齐人类偏好。”

80.1 什么是预训练？

在深度学习（尤其是自然语言处理，NLP）中，**预训练指的是先在海量的无监督或自监督数据上训练一个大规模模型（如BERT、GPT等），得到一个具备通用特征提取能力或通用语言理解能力的模型。**预训练的核心价值在于，它能让模型在大量无标注数据中学习到语言或领域的通用分布、模式和表示方式。

预训练好处

学习通用表示：通过大规模数据的预训练，**模型对词汇、句子结构、上下文关联等形成了统一且强大的表示能力。**

提高数据利用效率：**预训练通常在无标注数据上完成，数据量可大幅增加**；随后针对下游任务（分类、翻译、问答等）仅需少量带标注数据进行微调（Fine-tuning），即可取得优异效果。

80.2 什么是增量预训练？

增量预训练（Incremental Pre-training）是指**在已经完成初始预训练的模型上，通过对新增数据继续训练或微调的方式**，让模型持续学习新知识、适应新领域或更新的环境，同时保留原有模型的知识和能力。它强调**“在已有基础上更新”的思路**，而不是在每次获取新数据后都从头训练一个全新的大模型。

增量预训练主要特点

继承性：沿用原有模型的大部分参数和已学得的知识。

扩展性：接纳新增数据、新事件或新领域信息，使模型的知识库不断扩充。

高效性：无需重复全量训练，因而显著减少计算成本和时间开销。

小红书账号：小黄搞AI

80.3 为什么需要“增量”预训练？

数据与知识不断更新

实时更新的数据

自然语言世界是动态的，新的词汇、术语、事件和知识点在不断出现。如果只使用旧的数据进行预训练，模型的知识就会停留在过去，难以理解和生成当前或未来的新信息。

行业或场景的快速变化

某些领域（如医疗、金融、法律、技术）每天都在产生新的研究、法规、市场变化。为了确保模型掌握最新、最准确的知识，需要持续引入新数据对模型进行更新，这就需要“增量”地预训练。

避免从头训练的高昂代价

全量训练成本高

训练一个大型模型（BERT、GPT、LLM）通常需要数百张GPU、数周乃至数月的训练时间，以及海量的电力和硬件资源。如果每次遇到新数据都从头训练整套模型，资源成本极高。

已经积累的知识和参数不可浪费

大模型在预训练期间已经学习到十分宝贵的通用知识和特征表示。若频繁从头开始训练，既浪费了已经学到的知识，也重复了大量计算。

避免“灾难性遗忘”与参数重置

持续学习场景

如果仅仅对新数据进行训练，而不考虑旧数据的知识，模型有可能逐渐忘却早前学习到的能力，这被称为“灾难性遗忘”（Catastrophic Forgetting）。
增量预训练可以在兼顾先前知识的基础上，又引入新知识，从而平衡两者。

模型稳定性与一致性

增量预训练通常保留了最初的大量参数和知识表示，仅在原有模型基础上进行适度的继续训练或微调，可以较好地保持模型表现的稳定性和一致性，降低训练波动。

80.4 增量预训练带来的价值

保持模型“新鲜度”

通过对新增数据的持续学习，模型随时更新自己的知识库，保持对新词汇、新概念的理解能力。

更加贴合目标领域或场景

当应用场景发生变化或需要深入某个新领域时，只需增量地在该领域的专有数据上进行额外训练，模型就可以快速适应，大大缩短开发周期。

节约资源与成本

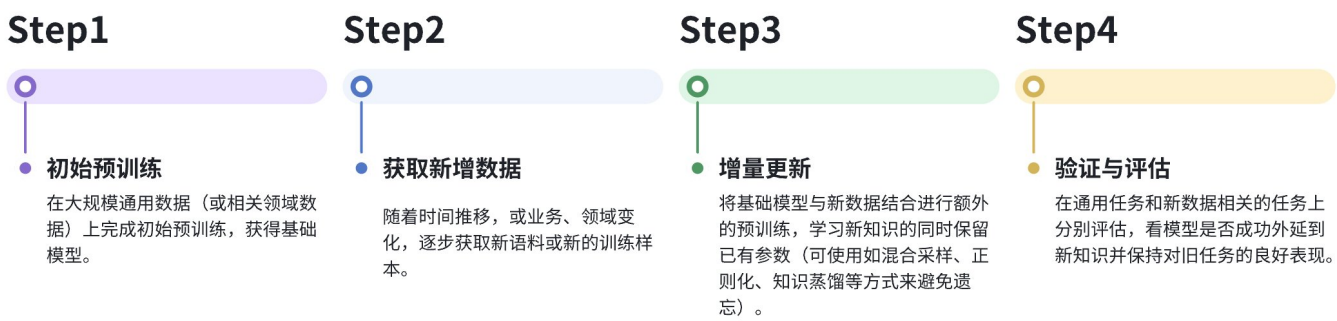
对接持续数据流

小红书账号：小黄搞AI

在已有预训练模型的基础上进行增量更新，比每次训练一个全新的大模型更节约算力、时间与人力。

对于一些在线或实时场景（如对话系统、舆情监控），数据如源源不断的“流”一般进入系统。此时，增量预训练能让模型持续迭代更新，以保持竞争力。

80.5 增量预训练的典型流程



🍆 总结

1. 增量预训练（Incremental Pre-training）是为了**应对不断变化或持续增长的数据与知识需求，而对已经预训练好的模型进行进一步训练与更新的策略**。
2. 它能够**帮助模型保持对新知识的掌握，避免从头训练的资源浪费**，并同时减少对原有知识的遗忘。
3. 在实际应用中，当业务需求、数据分布或目标领域发生变化，**增量预训练能够让模型以更低的成本、更快的速度适应新环境，从而持续保持竞争力。**

当数据的规模与领域不断演化时，**增量预训练可以在保留已有知识的基础上，平滑地融入新知识**，减少训练开销并防止灾难性遗忘。这种对大模型进行“持续学习”的方法对于需要长期维护和更新的应用场景至关重要。

81. 增量预训练的过程当中，loss上升正常吗？

> “通常增量预训练开始的阶段会出现一段时间的loss上升，随后慢慢收敛。”

81.1 什么是增量预训练？

小红书账号：小黄搞AI

什么是增量预训练？

增量预训练（Incremental Pre-training）：在已经预训练好的模型基础上（如一个大型语言模型），**对新收集的数据继续进行训练的过程。**

核心目的：让模型持续**学习新知识、适应新场景或新分布**，同时尽量保留原先学到的有用知识。

在此过程中，模型会接触到不在原训练分布或未充分见过的“新数据”，从而需要**“拓展”或“修正”已有的参数表示**。

81.2 为什么Loss可能会在增量预训练过程中上升？

数据分布改变或新领域的引入

训练分布发生转移（Distribution Shift）

新引入的数据可能与原训练数据差异显著（例如，不同领域、不同时间段、出现新术语或新事实）。

模型需要时间适应新数据

初始阶段，模型尚未对新数据的特征与分布形成有效参数，导致在新数据上表现不佳，从而出现短期的Loss上升。

学习新知识 vs. 保留旧知识

冲突知识（参数冲突）

若新数据与旧数据在语义或任务需求上有部分冲突（或分布重叠区域），模型会在“保留原有表征”和“适配新知识”之间产生张力。

灾难性遗忘（Catastrophic Forgetting）

若增量训练策略未能有效保留原有知识，模型对之前分布的表现会暂时退化（在旧数据上Loss升高），从而也可能导致整体Loss出现波动或上升。

不同训练模式与优化波动

学习率与优化过程

增量训练过程中若使用的学习率过大，或新数据量比例过高，都会造成一段时间内在新数据或旧数据上Loss不稳定甚至上升。

混合训练策略

一些做法会在同一个Batch或同一个Epoch中混合旧数据与新数据，这种混合也会导致短期内Loss出现波动，从而表现为时而上升、时而下降。

小红书账号：小黄搞AI

81.3 Loss 上升是否正常？如何评估？

短期上升是常见现象

在引入和原先分布差异较大的新数据后，短期内Loss上升是正常现象。

因为模型突然面对“陌生”的输入或目标分布；需要修正较多参数来吸收新知识。

这通常表现为**训练过程中Loss先升后降的曲线。**

随着增量训练的继续，模型逐渐学习新数据分布并稳定收敛。

长期趋势才是关键

稳定趋于收敛

若增量预训练策略恰当，模型将逐渐掌握新知识，同时对旧知识的遗忘也在可控范围内，最终整体Loss会再次下降并趋于新的稳定点。

关注验证集或旧数据集表现

仅看训练集Loss有时并不足以判断模型“是否真的学好”。需要同时监控：

a. **新数据的验证集Loss**（看是否学到新知识）

b. **旧数据集或核心评测任务上的Loss**（看是否保留已有能力、避免灾难性遗忘）

🌵 数据分布改变或新领域的引入

如果在增量预训练开始后，Loss出现**大幅度、持续性、不回落**的升高，则应检查：

a. **增量数据是否异常**：标注质量、分布过度偏移等。

b. **超参数设置**：如学习率是否过大，训练策略（混合比例、正则化）是否合理。

c. **模型结构与方法**：是否需要借助“知识蒸馏”“回放训练”等方法来融合新知识、避免遗忘。

81.4 如何平衡对新知识的学习与旧知识的保留？

在增量训练中，出现短期Loss上升固然正常，但我们仍希望能**平衡“学习新知识”与“保留旧知识”**。常见策略包括：

1. **混合采样（Replay/Memory-based）**
   - 保留部分旧数据样本，在训练新数据的同时，**不断让模型回顾旧数据**。
2. **正则化方法**
   - 例如Elastic Weight Consolidation (EWC)等，通过**在参数更新时对重要参数加大惩罚**，避免对旧知识造成“过激”干扰。
3. **知识蒸馏（Knowledge Distillation）**
   - 用增量前的**旧模型（Teacher）指导当前模型（Student）的学习**，减少对旧任务表现的破坏。
4. **分层或模块化训练**
   - 通过在**模型结构上进行模块化区分**，让新知识和旧知识在不同模块中分别优化，减小冲突。

小红书账号：小黄搞AI

结论

- **短期Loss上升在增量预训练中是常见且正常的**，尤其当新数据与旧数据差异较大时。
- 关注训练过程中Loss的动态变化更为重要：**如果Loss先升后降，最终趋于稳定，说明模型在成功吸收新知识、平衡旧知识。**
- 若Loss持续大幅升高且难以收敛，需要从**数据分布、优化策略、增量方法**等多方面进行排查和调优。

在增量预训练过程中，只要最终模型逐渐适应新数据分布、并在验证集上呈现收敛趋势（对新旧任务都保持或提升性能），**短期的Loss波动（包括上升）是可以接受且合乎常理的**。这是模型在接收新知识时必然经历的“磨合过程”。

82. 在增量预训练过程中如何设置学习率（learning rate, LR）？

82.1 学习率是什么？为何如此重要？

> **学习率（LR）**：表示在每次参数更新时，模型沿着梯度方向移动的“步长”大小。

学习率的影响是什么？

学习效率

学习率过低，模型收敛慢或难以学到新知识；学习率过高，参数更新过度，损失函数（Loss）振荡甚至发散。

稳定性

决定了模型在训练过程中的稳定程度，直接影响模型的性能和收敛速度。

82.2 增量预训练与学习率的关系

增量预训练的特殊性

在原有模型基础上继续训练

模型已经在海量数据上完成初始预训练，具备了较丰富的表征能力与知识。增量预训练（Incremental Pre-training）需要在此之上**引入新数据、新领域或新分布**，以更新或扩展模型知识。

风险：旧知识被破坏

学习率过大可能导致对已有权重的“过度”修改，**“旧知识”被大幅覆盖，出**

学习率对增量预训练过程的核心影响

平衡“旧知识保留”与“新知识学习”

- **LR过大**：容易造成模型对新数据的过度拟合、旧知识丢失严重，**Loss震荡大**；
- **LR过小**：**学习新知识缓慢或效果有限**，对新分布适配不佳。

稳定的收敛 vs. 不断探索

- 原模型已经基本“收敛”在较优的参数区域。增量训练时，如果想在此区域“微调”，需要一个相对小一些的步长，避免跳出之前的高质量解。

小红书账号：小黄搞AI

现“灾难性遗忘”；过小又会使得新数据无法有效更新，学不到新知识。

82.3 如何具体设置学习率？(🌟核心考量)

🫐 参考初始预训练阶段的学习率

通常做法

- 若初始预训练（Pre-train）阶段的最大学习率为 α，在增量阶段可使用**约10%**左右的学习率作为尝试。
- 举例：如果 7B 大小的模型在初始预训练时 $LR \approx 3 \times 10^{-4}$，增量预训练时可先尝试 `3e-5`（即 3×10^{-5}。

原因

增量阶段的目标是“在已有能力上增量更新”，而非大规模从零开始学，因此需要**更小**的步长来“细调”原有参数，防止大范围破坏。

🌵 数据集规模与分布差异

新数据集很大（例如与原训练规模同量级）

若增量数据量非常庞大（接近或超过初始训练规模），且与原分布有较大差异，可**适度提升**学习率，但仍需保证初始阶段平稳，可先从小 LR 开始，观察 Loss 曲线再逐渐提高。

新数据集相对较小（数十GB~百GB级）

建议**保守使用更小**的学习率。因为新数据量有限，过大步长容易让模型过度适配新数据而“忘记”老知识。

🍁 Batch Size 与学习率的配合

`Batch size` 缩放原则

- 如果 batch size 相比最初训练时扩大了 k 倍，可以将学习率相应地缩放，但通常不是线性 1:1 的关系，一般经验：
  - Batch size 增加 4 倍，LR 可增加 2 倍左右。

目的

保持“有效更新量”不变，兼顾稳定与效率。若

🍋 动态调参与监控

监控指标

- **训练集与验证集 Loss**：若在增量预训练初期观察到Loss持续上升且无法回落，或旧数据集上的表现陡降，就需要及时降低学习率。
- **学习曲线稳定性**：关注短期震荡（短期Loss急升）与中长期趋势（是否能逐渐收敛）。

学习率调度策略

- 可以在增量训练时继续使用 **Warmup + Cosine decay** 或 **线性衰减** 等方法；
- 也可分阶段设定：前期先用稍小 LR 适应新数据，中后期在确保模型稳定后再逐渐增大或保持衰减。

小红书账号：小黄搞AI

batch size 调整不当，需特别关注震荡情况。

82.4 实践建议

实践建议

初始值设置

参考预训练最大LR的 **1/10** 作为初始增量LR 的起点（例如从 `3e-4` 降到 `3e-5` ）。

按照新数据规模微调

数据规模大、分布差异大时，可略微提高 LR（如从 `1e-5` ~ `3e-5` 区间尝试）；

数据规模小、分布相对相近时，就用相对更低 LR。

同步考虑 batch size

如果 batch size 明显增大，可酌情略增 LR，反之亦然。

分阶段、动态调整

增量预训练开始时，宜谨慎保守；观察损失和评估指标，如能平稳下降并在验证集上提升，可小幅增大；若波动或遗忘严重，及时减小 LR。

保持旧知识的技巧

可以适度混合一定比例的旧数据（或采用知识蒸馏、EWC 等正则化），以减少大幅更改参数导致旧知识流失。

结论

- **学习率在增量预训练中是关键调控参数**：既要确保模型能学到新知识，又不能大幅度毁坏已有能力。
- **通常应比初始预训练时略小**，并根据数据规模、分布差异、batch size 等因素进行相应调整。
- **核心目标**：在合理的步长范围内，**让模型平稳地吸收新知识**，最大限度保留旧知识并避免灾难性遗忘。
- **可操作方案**：以 **原预训练最大LR的 10%** 左右作为起点，观察数据规模和训练曲线，配合合适的衰减策略，动态微调，最终找到适合自身业务场景的最优LR设置。

总结

增量预训练的过程和常规预训练或微调不同之处在于，模型已经包含重要的通用或领域知识，需要在此基础上谨慎地更新参数。学习率过大、过小都有明显弊端，因此在设置学习率时应综合考量 **初始预**

小红书账号：小黄搞AI

训练LR、数据规模、batch size、分布差异 等因素，并配合良好的训练监控与调整策略，方能取得理想效果。

83. 什么是warmup_ratio？训练过程中怎么设置？

83.1 什么是“Warmup”？

在深度学习模型（尤其是大规模语言模型，LLM）的训练中，“**Warmup**”是一种在训练初期对学习率（learning rate, LR）进行**缓慢升高**的技巧。它的核心思想是：

- **先用较小的学习率**开始训练，避免刚开始时模型参数大幅震荡；
- **再逐步提高学习率**到目标值，随后再进行正常的训练或后续的学习率衰减（decay）。

为什么需要Warmup？

稳定性

刚开始训练时，模型参数一般还比较随机、混乱，如果一上来就用较大学习率，容易使Loss剧烈波动甚至导致梯度爆炸。

更好收敛

先用较小LR保证模型在初期能收敛到更稳定的方向，然后再让学习率达到设定的峰值进行主干训练，往往能得到更好的优化效果。

83.2 什么是“warmup_ratio”？

warmup_ratio 是一个**比例（ratio）**，通常指**训练总步数（或总epoch数）中，用于Warmup的比例**。例如：

- 如果 `warmup_ratio = 0.01`，表示在整个训练过程中，前 **1%** 的训练步骤（或epoch）是用来“Warmup”的，学习率从初始较低值逐渐上升到设定的最大学习率。
- 如果 `warmup_ratio = 0.03`，表示前 **3%** 的训练步骤都处于Warmup阶段。

步数（step）与epoch的关系

step：一次前向/反向传播（通常对应一个mini-batch）；

epoch：遍历完一次整个训练集称为一个epoch。

在实践中，warmup_ratio既可基于**总step**（更常见），也可根据**epoch**来估计。比如“预训练只训练1个epoch时，warmup_ratio可以设为1%的epoch长度”，相当于把总训练步数的1%划做warmup阶段。

83.3 训练过程中怎么设置warmup_ratio?

小红书账号：小黄搞AI

🌵 根据训练时长（epoch数）步数来估计

如果只训练1个epoch（比如大型预训练阶段常见）：可以设置 `warmup_ratio ≈ 0.01`（1%），让前1%的训练过程逐渐升高LR。

如果训练3个epoch（如SFT场景）：可以将 `warmup_ratio` 设为 `0.03`，即可让LR平滑上升，防止模型初期“爆炸”。

若数据集非常大（数百GB甚至TB级别），训练epoch较多：

Warmup对最终效果影响往往不会特别大，但依然建议在初期做适当的Warmup，**让模型可以平稳“起步”**。可设置 `warmup_ratio` 在 `0.01 ~ 0.06` 区间内，视学习率和批次大小而定。

🍇 与学习率大小的关系

Warmup_ratio与学习率通常呈正相关

如果你打算使用**较大的学习率**，那么通常会把 **warmup_ratio** 设置得稍大一些（如0.05~0.1），以防止模型在初始阶段“崩溃”；

如果你的学习率本身较小，则warmup_ratio可以相应调小，因为初期参数更新步子本来就不大，风险有限。

warmup防止“烂掉”

在增量预训练场景或SFT场景下，如果模型已经有预训练能力，但你又想用较高LR去快速学习新知识，那么**Warmup阶段能帮你避免“参数过激更新”**，维持模型稳定性。

💡 其他参考因素

batch size

若batch size特别大，在初期可能造成相对更大的梯度累积，建议**适当增加** warmup_ratio，**让LR在前期更缓和地上升。**

观察Loss曲线

训练时应**密切监控Loss曲线**、梯度变化：若在初期依然出现很大震荡或梯度爆炸，可再提高warmup_ratio；若前期学习过于缓慢，可适度下调该比率。

调度策略（scheduler）

常见做法是 **线性warmup**：在Warmup步数内，LR从 `初始值` 线性增长到 `最大LR`。

Warmup结束后，可以采用 **cosine decay、linear decay** 等方式继续衰减学习率，完成后续的训练。

83.4 总结与建议

🍋 **warmup_ratio定义**：在训练的前多少比例的步数（或epoch）内进行“学习率从低到高”的热身过程。

小红书账号：小黄搞AI

典型值：

- 预训练1个epoch，可设 `warmup_ratio=0.01`；
- SFT常见3个epoch，可设 `warmup_ratio=0.03`；
- 大规模、高学习率场景可酌情提高到0.05~0.1；
- 小数据集、低学习率场景可适度降低，甚至0.005亦可尝试。

与学习率、batch size互相配合：当学习率或批次大小变大时，通常需要更长的Warmup来缓冲。

监控训练曲线，灵活调整：根据初期Loss波动或模型稳定性决定是否需要微调warmup_ratio，避免“参数塌陷”或“学不动”的极端情况。

💛 **🌟总结**

warmup_ratio 是训练过程中的关键超参数，决定了**学习率从初值到最大值**所需的“热身”步数比例。

在大模型（LLM）或增量预训练场景中，合理设置warmup_ratio不仅能防止训练初期的梯度爆炸，也能让模型更平滑地过渡到正常学习阶段，取得更稳定、更优的收敛效果。

84. Multi-head Latent Attention（多头隐变量注意力）是什么?

这类方法较为前沿，**“显式” 引入隐变量** 并非 Transformer 的主流做法，但在一些强化学习或复杂多模态系统中会出现，如 Perceiver 系列（DeepMind 提出“Latent Array”）或 Memory-augmented Transformer 等思路。

84.1 概念

🥥 **概念与公式**

概念：在多头注意力的基础上引入额外的隐变量/隐状态（latent states），让模型在计算注意力时可以通过这些“潜在向量”来捕捉更高维或者更全局的交互。

公式(示意)： $\text{head}_i = \text{Attention}\left(QW_i^Q, KW_i^K, VW_i^V, Z\right)$，其中 Z 可能是某种全局隐向量或与外部记忆相连，可以在注意力分配时扮演补充信息的角色。

应用：

- **Perceiver (Jaegle et al. 2021)**：使用一组潜在向量来聚合并处理高维输入（如图像、音频）。
- 需要更高层表征能力的应用：多模态、跨模态、超大规模场景等。

小红书账号：小黄搞AI

84.2 Multi-head Latent Attention（多头隐变量注意力）

为什么引入“隐变量”？

标准多头注意力：只在输入序列 Token 之间（或跨序列）计算 Q/K/V；对每个头来说，Q/K/V 都是显式映射自输入或上一层隐藏态。

隐变量/Latent：希望在注意力过程中，多引入一批“潜在向量”或“外部记忆”。

- 这批向量可以**承载更全局、更浓缩或更抽象的特征**，尤其在**高维数据**（如视频、音频或多模态融合）或**超大规模**上下文中，被认为可引导模型更好地聚合、提炼信息。

核心思路

给定输入序列 $\{x_i\}$，除了在多头注意力中使用 (Q, K, V) 外，还**显式**加入一组“Latent”向量 $\{z_j\}$。这可能有多种操作方式：

1. **Perceiver 系列 (Jaegle et al. 2021)**
   - 定义一组 **Latent Array**（固定长度，如 512 或 1024 个向量），让它与输入序列交互：
     i. 先做**Cross-Attention**: latent 向量 $\rightarrow$ 输入；
     ii. 再做**Cross-Attention**: 输入 $\rightarrow$ latent；
     iii. 或重复多次迭代，让 latent 向量在多轮注意力中**逐步“吸收”输入特征**并输出。
   - 因为 latent 向量的数量一般远小于输入序列长度，可以**降低整体计算量**，同时也能对庞大输入做高度抽象表达。
2. **Memory-augmented Transformer**
   - 在注意力计算时额外加入“记忆槽”（memory slots），作为**Key/Value**或者额外的 Query/Key/Value；模型可以通过注意力**读写**这些外部记忆槽，从而“记住”较长时间跨度的信息。
   - 这在强化学习（RL）或多模态任务中也可见到。
3. **多头隐变量的公式示意**

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

   - 其中 Z 表示**隐变量**（可能是若干向量或某种可学习的全局 token），在注意力分配时**作为额外的信息源或交互对象**。

实际操作流程（以 Perceiver 为例）

定义一组 latent 向量：尺寸 (N_{latent}, d)，比如 256～1024 个。

输入序列（如图像 patch、音频帧等）长度可能很大，记为 (N_{input}, d)。

小红书账号：小黄搞AI

Cross-Attention 步骤：

- **Latent 当作 Q**，输入当作 K/V：
  $Q = ZW^Q, \quad K = XW^K, \quad V = XW^V$, $\text{Attention}(Q, K, V)$ 后得到**更新后的 Latent**。

重复多层：再让 Latent 与自身或与输入继续交互；在每一层中，Latent 会逐渐聚合输入信息，变得更加“浓缩”或“抽象”。

输出：最终在 Latent 上做分类、回归或其他下游任务，这样就无需对庞大的输入序列逐 Token 做大规模计算。

应用与趋势

高维输入场景：图像、视频、音频、3D 点云等，每一帧/每个 patch 都是一个“Token”，可能有数万 Token；用“latent”做中间表示，可显著减少注意力计算规模。

多模态、跨模态：可让一组 latent 向量充当“融合容器”，将不同模态的信息集中。

强化学习、记忆型模型：通过外部隐变量或记忆槽，让模型在长时间跨度或复杂环境中保留更抽象的状态表征。

85. Linear Attention（线性注意力）是什么?

85.1 概念与公式

概念与公式

概念：尝试将注意力的计算复杂度从 $O(n^2)$ 降低到 $O(n)$ 或接近线性的方法，通常通过引入核技巧（kernel trick）或因式分解将 $\text{softmax}(QK^\top)$ 转换为可分离的形式。

公式： $\text{Attention}(Q, K, V) \approx \left(\phi(Q)\left[\phi(K)^\top V\right]\right) / \left(\phi(Q)\left[\phi(K)^\top \mathbf{1}\right]\right)$，其中 $\phi(\cdot)$ 是某种核映射函数，通过这种“先合并 V，再与 Q 相乘”的方式将原本的 $O(n^2)$ 矩阵相乘转化为两个 $O(n)$ 的操作（在理想情况下）。

应用：

- 仍然是解决长序列问题：希望把注意力的时间/空间复杂度降低到可接受范围。
- 文本、语音、基因序列、图像 patch 等长序列场景下都可尝试。

85.2 Linear Attention（线性注意力）

为什么要做“线性化”?

标准注意力计算复杂度：$O(n^2)$，当序列长度 n 很大时（数千、上万 Token），开销（时间、显存）非常惊人。

小红书账号：小黄搞AI

线性注意力目标：将标准注意力 $\mathrm{softmax}(QK^\top)V$ 通过某种**近似或重构**，使其计算复杂度约为 $O(n)$ 或 $O(n\log n)$，从而能在**长序列**场景下更高效地应用。

核心思路：**将 $\mathrm{softmax}(QK^\top)$ 近似为可分解形式**

标准注意力公式：$\mathrm{Attention}(Q,K,V)=\underbrace{\mathrm{softmax}\Big(\frac{QK^\top}{\sqrt{d_k}}\Big)}_{n\times n}V \quad\Longrightarrow\quad O(n^2)$ 计算

线性注意力则尝试把上式中的 $\mathrm{softmax}\big(\frac{QK^\top}{\sqrt{d_k}}\big)$ 用某种 $\phi(\cdot)$ 函数分解为：$\mathrm{softmax}\big(QK^\top\big)\ \approx\ \phi(Q)\,\phi(K)^\top$，这样，就能把原本的 $QK^\top$（$n\times n$级别）分解成若干个**向量乘法**，再配合一些巧妙操作，使得计算不再需要显式构建 $n\times n$ 矩阵。

典型示例

$\phi(\cdot)$ 是一种核映射 / 随机特征近似

Performer (Choromanski et al. 2021) 中：通过 **随机特征映射**（Random Feature Method）来近似 $\exp(x)$，从而把 $\mathrm{softmax}(QK^\top)$ 变为分解形式。

其它方法：也可能使用可学习的核函数 $\phi(\cdot)$，或者使用 ReLU/ELU 的正变换等，核心都在于**将庞大的矩阵相乘改写为可分解的形式**。

公式展开（以 Performer 为例，示意）

假设 $\mathrm{softmax}(QK^\top)\approx\phi(Q)\,\phi(K)^\top$,其中 $\phi(\cdot)$ 是某种 $d\to r$（r 为中间核维度）的映射，那么

$$\mathrm{Attention}(Q,K,V)=\underbrace{\mathrm{softmax}(QK^\top)}_{n\times n}V\ \approx\ \frac{\phi(Q)\,[\phi(K)^\top V]}{\phi(Q)\,[\phi(K)^\top \mathbf{1}]}$$

- $\phi(K)^\top V$ **的计算**：
  - 先把所有 Key 经过 $\phi(K)$ 变成一个 $n\times r$ 矩阵（若有 n 个 Token），再与 V（$n\times d$）相乘，有时可以通过**前向累加**的方式在 $O(n)$ 时间内就得到聚合结果。
- $\phi(Q)\,[\phi(K)^\top V]$ **的计算**：
  - 每个 Query 向量 $\phi(Q_i)$ 只需与 $\phi(K)^\top V$ 的结果做一次矩阵/向量乘，即**不再是与所有 Key 两两计算**。
- 整体上，若设计得当，可以将原本需要显式构造 n×nn\times n 的步骤，替换为**两次 $O(n)$ 的操作**（一个是 $\phi(K)^\top V$ 的预先计算，一个是对每个 Query 做乘法）。

这样就达到了**近似线性化**的目的（虽然要注意，核映射维度 r 可能也会影响实际速度和内存，但大体能显著降低针对 n 的二次方开销）。

实际操作中的流程

1. **输入序列** $x_1,x_2,\ldots,x_n$，投影得到 Q,K,V。

小红书账号：小黄搞AI

2. **对 K, V 做核映射并合并**：
   $\Phi = \phi(K)^\top V, \quad \mathbf{z} = \phi(K)^\top \mathbf{1}$ （$\mathbf{1}$表示全1向量，用于归一化）
3. **对每个 Token i 的 Query** $\phi(Q_i)$，用上面得到的 $\Phi, \mathbf{z}$ 来计算**注意力输出**：
   $$\text{head}_i \approx \frac{\phi(Q_i)\,\Phi}{\phi(Q_i)\,\mathbf{z}}$$
4. **合并结果**：将所有 Token 的注意力输出拼接或在序列维度上堆叠，得到与传统多头注意力类似的结果。

应用与趋势

- **长序列文本**：让 Transformer 可以处理上万 Token，无需 $O(n^2)$ 的大开销。
- **多模态长序列**：图像或视频分块处理、长音频、基因序列等。
- **代表性工作**：
  - **Transformers are RNNs** (ICLR 2020)
  - **Performer** (ICLR 2021)
  - **Linformer、Reformer** 等也在探讨不同线性化/低秩分解思路。

> **Katharopoulos et al. (ICLR 2020)** [*Transformers are RNNs: Fast Autoregressive Transformers with Linear Attention*]，引入线性化注意力思想；
>
> **Choromanski et al. (ICLR 2021)** [*Rethinking Attention with Performers*] 提出 Performer，将 softmax 注意力用随机特征近似进行线性化；
>
> 还有 **Linformer (Wang et al. 2020)**、**Reformer (Kitaev et al. 2020)** 等不同思路。

86. Cross-Attention（交叉注意力）是什么?

86.1 概念与公式

概念与公式

概念： 在计算注意力时，Query（查询）来自一个序列（或特征集），Key / Value（键/值）来自另一个序列（或特征集）。常见情形是 **Decoder** 使用 Cross-Attention 来“关注” Encoder 输出。

公式： 如果我们将 Encoder 的输出记为 H，Decoder 的先前状态记为 Y，那么 Cross-Attention 可能类似： $Q = YW^Q, \quad K = HW^K, \quad V = HW^V$ 其余部分与自注意力相同，仍是 $\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$

86.2 Transformer中的Cross-Attention

小红书账号：小黄搞AI

Transformer中的Cross-Attention

在 **Transformer** 的 **Encoder-Decoder** 结构里，所谓的 **Cross-Attention**（交叉注意力）指的是：

Query（Q） 来自 **Decoder** 端（即目标序列的表示）。

Key（K） 和 **Value（V）** 则来自 **Encoder** 端（即源序列的表示）。

因此，**在做 Cross”的**其实就是：**Decoder 的隐藏状态**（或 Decoder 端的输入）**和 Encoder 输出的隐藏表示**（即源序列经过 Encoder 后得到的一组特征）

Encoder-Decoder 结构概览

在标准的 Transformer 中，分为 **Encoder** 和 **Decoder** 两个主要部分：

Encoder：输入为源序列 X（例如外语句子），经过若干层自注意力与前馈网络后，输出一个隐藏表示矩阵 $\mathbf{H}$。$\mathbf{H} = \text{Encoder}(X)$（大小约为 $n \times d$），其中 n 是源序列的长度，d 是隐藏维度。

Decoder：输入通常包括

- 目标序列（或已生成部分序列）$\mathbf{Y}$ 的嵌入；
- 编码器输出 $\mathbf{H}$

Decoder 也有若干层，每一层通常包含：

a. **Masked Self-Attention**（针对 $\mathbf{Y}$ 本身做自注意力，用 mask 来防止看见后续词）；

b. **Cross-Attention**（也叫 Source-Attention，用 $\mathbf{Y}$ 的表示去“查询” $\mathbf{H}$，从而访问源序列信息）；

c. 前馈网络（Feed-Forward）。

Cross-Attention 的输入与输出

来自 Decoder 端的 Query

- Decoder 在第 ll 层（或任意一层）会先用 Masked Self-Attention 得到一批隐藏向量 $\mathbf{D}^{(l)}$。
- 这批隐藏向量会经过线性变换成为 $Q = \mathbf{D}^{(l)}$，W^Q 这里 $\mathbf{D}^{(l)}$ 对应目标序列（或已生成的目标 Token）在 Decoder 第 ll 层的表示。

来自 Encoder 端的 Key、Value

- Encoder 的输出 $\mathbf{H}$（大小为 $n \times d$）会经过线性变换分别得到 K 和 V，$K = \mathbf{H}W^K, V = \mathbf{H}W^V$
- 这里 $\mathbf{H}$ 包含了源序列（输入句子）在 Encoder 端的上下文信息；经过投影后，就成为用来匹配 Query 的 “Key” 以及最终提供“内容信息”的 “Value”。

Cross-Attention 的计算

小红书账号：小黄搞AI

一旦得到 Q, K, V，就跟普通的 Scaled Dot-Product Attention 一样：

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

- 这里，Q 的行数对应目标序列的长度（Decoder 时间步数），
- K, V 的行数对应源序列的长度（Encoder 输出长度）。

为什么称作“Cross-Attention”?

Self-Attention（自注意力）是 **同一个序列** 内部的各个位置之间相互计算注意力；

Cross-Attention 则是 **目标序列（Decoder 侧）** 和 **源序列（Encoder 侧）** 之间的注意力交互。

应用

- 机器翻译：Decoder 根据源语言（Encoder 输出）生成目标语言。
- 多模态任务（Vision+Language、Speech+Text 等）：图像特征和文本特征通过 Cross-Attention 交互。

> 在机器翻译中，最开始的“attention”就是源序列和目标序列间的注意力，实际上就属于 Cross-Attention 的范畴。
>
> 在 Transformer 中，**Encoder-Decoder 结构**里，Decoder 部分显式地包含一个 Cross-Attention 模块，用来获取源序列信息。

87. Sparse Attention（稀疏注意力）是什么?

87.1 概念与公式

概念与公式

概念：在计算注意力权重时，不再对所有 $(n \times n)$ 的位置两两计算，而是只针对部分位置（token）进行注意力。 例如：局部块（block attention）、滑动窗口（sliding window）、稀疏模式（stride）等。

公式： 虽然核心公式还是 $\text{softmax}(\frac{QK^\top}{\sqrt{d_k}})V$，但 $\frac{QK^\top}{\sqrt{d_k}}$ 会被**稀疏化**为一个稀疏矩阵 S，例如只保留邻近若干位置或固定模式：

$$S_{i,j} = \begin{cases} (QK^\top)_{i,j}/\sqrt{d_k}, & \text{if } (i,j) \in \text{some pattern}, \\ -\infty, & \text{otherwise (masked)}, \end{cases}$$

，然后 softmax 时会将被 Mask 掉的部分归零。

应用：

小红书账号：小黄搞AI

- 需要处理超长文本（成千上万 Token）的 NLP 任务：长文摘要、长上下文 QA、法律/金融文档处理等。
- 序列长度很大的时序预测或推荐系统，也会考虑稀疏注意力。

87.2 稀疏注意力与全连接→Dropout 的思考

背景对比

全连接层到 Dropout：

- 在传统神经网络（尤其是早期全连接层）中，**所有神经元之间都是完全连接**，因此参数量非常大。
- 引入 **Dropout** 的初衷是：对连接或神经元进行随机“断开/关闭”，让网络在训练中保持一定的“稀疏性”，从而**减少过拟合**、提高泛化能力。这种做法在实践中能显著提升模型鲁棒性。

自注意力到稀疏注意力：

- 在 Transformer 等模型中，**标准自注意力是对所有 $(n \times n)$ 的位置两两计算**，当序列长度 n 很大时，计算和显存开销都呈 $O(n^2)$ 级别。
- **稀疏注意力（Sparse Attention）** 则通过“只在部分位置上计算注意力”来降低计算复杂度，例如局部块、滑动窗口等方式，将计算复杂度降到 $O(n)$ 或 $O(n \log n)$。它主要为了解决**长序列的效率问题**。

从这一点看，二者虽然都引入了“稀疏性”，但**动机略有不同**：

- **Dropout**：旨在**防止过拟合**、增强模型的泛化能力；
- **Sparse Attention**：旨在**减少计算量和内存占用**，使长序列处理可行。

“稀疏性”对比

Dropout 的稀疏性

- 训练阶段随机“断开”部分神经元（或隐藏向量维度）连接，**每次迭代随机生成一个“mask”**，被遮蔽的节点不参与当前批次前向传播和反向传播。
- 训练完后，推理阶段往往关闭 Dropout（或按一定缩放比例），因为模型此时不再需要随机失活，需使用所有连接来最大化性能。

Sparse Attention 的稀疏性

- 把原本完整的 $(n \times n)$ 注意力矩阵中“大部分元素”屏蔽为 $-\infty$（或 0），只保留特定位置（邻近块、滑动窗口等）。
- 这不是随机的，而是**固定的或可学习的**稀疏模式：

小红书账号：小黄搞AI

- **固定模式**（如 sliding window、dilated window 等）：在**训练和推理**时都一致地保留同样的稀疏结构。
- **可学习模式**（需要额外参数或方法来学习稀疏性），也会在训练和推理阶段保持一致的结构或经过一定规则来确定关注位置。

- **目标是降低计算量**和显存压力，在很长的序列（如万级 Token）上也能让模型运行起来。

> 可以看到，**稀疏注意力的“稀疏性”更强调对序列的局部/结构化稀疏模式**，并不是像 Dropout 那样“随机丢弃”部分连接，而是在**训练和推理时都大概率保持一个固定或可学习的稀疏结构**。

共同趋势：从“全连接”到“更高效、更具鲁棒性”的模式

虽然 Dropout 和 Sparse Attention 的直接动机不同，但两者确实都体现了“从全连接”到某种形式的“稀疏或部分连接”的思路，核心原因是：

1. **全连接往往带来非常庞大的计算量或过拟合风险**。
2. 通过“稀疏或部分连接”，可以在**效率/泛化能力**上获得显著收益。

在这一层面上，两者确实有**相似之处**：

- 都是在某种程度上减少“连接”或“相互计算”的数量。
- 都是为了在**实践**中得到更好的效果（要么提升训练效率、要么减少过拟合、要么让长序列可行……）。

87.3 稀疏注意力在大模型中的趋势

为何稀疏注意力越来越重要？

大语言模型（LLM）中，序列上下文长度不断增长（从几千 Token 到上万 Token），标准的 $O(n^2)$ 自注意力消耗极大。

稀疏注意力、局部注意力、滑动窗口等技术可以有效地将**时间/空间复杂度**降低，从而让模型在长序列推理时依旧保持较好的性能。

比如：Longformer、BigBird、Sparse Transformer 等已经在大模型的超长文本场景中取得较好效果。

是否会在大模型中成为常见形式？

随着对**长上下文**的需求不断上涨，**稀疏注意力**（以及其它降复杂度注意力，如線性注意力、分块注意力等）**势必会越来越普及**。

不仅仅是文本，在处理多模态数据（视频长帧序列、音频长序列、基因序列）时，也需要类似的稀疏或高效注意力机制来应对超长输入。

未来，**稀疏注意力 + 动态/可学习稀疏模式**，也有可能成为新的研究与应用热点。

小红书账号：小黄搞AI

与 Dropout 的“结合”或“借鉴”？

目前也有一些工作会尝试**随机稀疏注意力**（即训练时随机选择一部分注意力连接），在一定程度上与 Dropout 有“相似”的随机化思路： 既可降低计算量，又能起到正则化作用。

不过，主流的稀疏注意力更多是**固定的局部模式**（如 sliding window）或**可学习模式**，而非像 Dropout 那样在训练时频繁随机变动。

小结与展望

与 Dropout 的相似之处： 都是从“全连接”出发，引入了某种“稀疏/丢弃”手段，以期**提高训练效率和模型性能**。

与 Dropout 的不同点： 动机和实现方式不同：

- Dropout 主要在**随机失活**连接/神经元，防止过拟合；
- 稀疏注意力主要在**固定或可学习的稀疏结构**上进行注意力计算，降低长序列的计算复杂度。

未来在大模型中的趋势： 随着上下文长度需求和多模态数据的增长，稀疏注意力必然会成为**处理长序列的常见工具**，并且会有更多变体（如分块注意力、混合局部+全局 Token、随机稀疏策略等）出现。

因此，也许**稀疏注意力**对于大模型的长序列处理将会像当年 Dropout 之于深度学习的普及一样，**成为一种非常常见且关键的技术手段**。同时，在具体实现层面，它与 Dropout 又有着不小的差异化侧重点——但最终都服务于让模型在资源可控的前提下实现更优的性能与更强的泛化能力。

> **Child et al. (2019)** [*Generating Long Sequences with Sparse Transformers*] 提出多种稀疏模式；
>
> **Beltagy et al. (2020)** [*Longformer*] 采用滑动窗口 + 全局 Token；
>
> **Zaheer et al. (2020)** [*Big Bird*] 综合了随机、滑动窗口、全局 Token 方案。

88. 不同Attention之间的区别是什么？

88.1 各类 Attention 的关系与比较

Self-Attention vs Cross-Attention

- Self-Attention：同一序列内各位置彼此建模
- Cross-Attention：不同序列之间信息交互
- 都是最基础的关注“相关性”的手段

Sparse / Linear Attention vs 普通 Attention

小红书账号：小黄搞AI

- 目的相同：**处理超长序列时减少 O(n2)O(n^2)O(n2) 负担**
- Sparse：通过设计“稀疏模式”跳过大部分计算
- Linear：通过核技巧或矩阵分解把二次复杂度转为线性

Multi-head Attention vs Multi-query / Grouped-query Attention

- Multi-head：每个 head 都有独立 Q/K/V，表达力强，但开销大
- Multi-query：只留一套（或少数）K/V，大幅降低显存占用
- Grouped-query：分组共享 K/V，兼顾一定的表达力与效率

Multi-head Latent Attention

- 在多头的基础上引入“隐变量/全局潜在表示”
- 进一步增强模型的全局表达能力

88.2 各类 Attention表格对比

机制名称	概念/定义	核心相同点	主要区别	典型应用
Self-Attention	**在同一个序列内部**，所有位置之间两两计算注意力，从而获取全局依赖。	与其他注意力一样基于 $\text{Attention}(Q,K,V)=\text{softmax}(\frac{QK^\top}{\sqrt{d_k}})V$ 都是为了捕捉序列/特征间的相关性。	只使用**同一序列**作为 Q/K/V；每个 Token 都与序列中其他所有 Token 交互。	1. Transformer 编码器（如 BERT、GPT 等）2. ViT 等视觉自注意力模型
Cross-Attention	**在不同序列间计算注意力**，Q 来自目标序列（Decoder），K、V 来自源序列（Encoder）。	1. 同样是 Scaled Dot-Product Attention 的变体 2. 仍需计算 QK^T 并进行加权到 V 。	Q、K、V 分属不同序列，使得 Decoder 可“查询” Encoder 的输出，进而**融合外部信息**（典型于 Encoder-Decoder 结构）。	1. 机器翻译：Decoder 从源序列获取信息 2. 多模态任务（图文交互、语音文本交互等）
Sparse Attention	通过**稀疏模式**（如局部块、滑动窗口、stride 等）来减少 $O(n^2)$ 注意力计算量，仅在部分位置计算相关性。	1. 与普通注意力相同的计算公式 2. 都是利用 $\text{dot}(Q,K)$ 来获得注意力权重。	不对所有 Token 两两计算，而是**用稀疏化策略跳过多数不相关位置**，从而降低复杂度到 $O(n)$ 或 $O(n\log n)$ 等。	1. 长序列任务：如长文档摘要、法律/金融文档2. 改进大模型推理效率（Longformer、BigBird 等）
Linear Attention	**通过核函数或矩阵分解**，将注意力计算复杂度从 $O(n^2)$ 近似	1. 仍然遵循注意力核心：计算 Q 与 K 的匹配，得到对 V 的加权	使用**核技巧（kernel trick）或特殊分解**，将	1. 同样解决长序列问题，但思路是“近似/分

小红书账号：小黄搞AI

	降到 $O(n)$，如 Performer、Linformer 等方法。	2. 不改变“注意力”的本质，仅改变计算方式。	$\text{softmax}(QK^\top)$ 近似为可分解形式，从而线性化或近似线性化注意力计算。	解”而非“稀疏化” 2. 典型模型：Performer, Linformer, Reformer 等
Multi-head Attention	**将注意力机制并行拆分为多个独立的注意力头**（head），每个 head 在不同子空间捕捉序列依赖，然后再拼接。	1. 依旧使用 Q,K,VQ, K, V 的注意力公式 2. 都在“谁注意谁”的框架下，只是每个 head 各自独立。	**在同一次注意力计算里使用多个头**；每个头有自己的一套投影参数 W_i^Q, W_i^K, W_i^V，最后再合并，表达力更强。	1. 绝大多数 Transformer 都使用多头 2. NLP、CV、推荐、语音等各种序列建模场景
Multi-query Attention	仅保留一套或少数套 K、V，但保留多个 Query 头，从而**减少存储和计算（尤其是推理阶段）**。	1. 与多头注意力一样，需要多个 Query 头 2. 同样基于标准注意力核心公式。	Key、Value 不再为每个 head 独立，**显著减少显存占用**；只在 Query 端保持多头。	1. 大规模模型推理优化（如多查询减少计算）2. 移动端/边缘设备部署场景
Grouped-query Attention	把多个注意力头分组，每组共用一套 K、V；**兼顾多头灵活性与减少存储开销的需求。**	1. 与多头/多查询同理，也是一种并行注意力 2. 同属“推理加速/显存优化”范畴。	介于多头和多查询之间：只在**“组”级别共用 K、V，而组内的每个 head 依然独立的 Q**；牺牲部分灵活性，减少大量参数。	1. 同样应用在大模型推理优化 2. 工业界大规模模型应用场景
Multi-head Latent Attention	在多头注意力基础上，引入额外的“隐变量/隐状态”，**让模型在注意力计算时也能利用全局潜在向量进行信息交互。**	1. 延续多头注意力的基本机制2. 本质上依然是 $\text{Attention}(Q, K, V)$，但增加一组“latent”向量或“记忆”。	将注意力进一步扩展到隐变量：可看作在**注意力中多加了一个“全局信息”或“外部记忆”容器**，以捕捉更高层次的抽象与依赖。	1. Perceiver 模型（DeepMind）等会用“Latent Array” 2. 一些 memory-augmented Transformer、跨模态/多模态任务

快速记忆

Self-Attention：内部关注，学全局依赖。

Cross-Attention：外部信息交互，学跨序列依赖。

Multi-head：分头并行，丰富表达。

小红书账号：小黄搞AI

Multi-query / Grouped-query：减少 Key/Value 重复，以提高效率。

Sparse / Linear Attention：减少或简化注意力计算，以适应长序列。

Multi-head Latent Attention：在多头的基础上引入隐变量，更丰富的表示。

🌟 用一句通俗语言总结

> **“它们都源自同一个核心：根据‘相关性’分配不同注意力权重，只是根据不同场景需求（是否同一序列、是否跨序列、是否要稀疏或线性化、如何平衡多头与效率、是否加入额外隐变量等）作出不同的改进和变体。”**

从**自注意力**、**交叉注意力**到**多头注意力**是 Transformer 的主干；在此之上，为了**加速长序列处理**，有了**稀疏注意力**与**线性注意力**；为了**优化推理效率**，又发展出**多查询**/ **分组查询**注意力；再进一步，为了**增强表达力**，一些工作研究了**隐变量**结合注意力。所有这些改进都围绕着同一个目标：**在计算、内存、表示能力之间找到最优平衡，并让模型能更好地关注到最重要的信息。**

89. Group Normalization（GN）和 Instance Normalization（IN)的区别是什么?

89.1 Group Normalization（GN）

Batch Normalization（BN）的出现极大地加速了图像分类等任务的训练过程，但它需要在一个较大的 mini-batch 内统计均值和方差，才能得到比较稳定的统计量。

当批量大小很小（小 batch）或在某些特殊的图像/风格任务中，BN 往往**不适用**或效果不佳，这就催生了 **Group Normalization（GN）** 和 **Instance Normalization（IN）** 等替代方案。

🥥 核心思想

Group Normalization（GN）是由 Facebook AI Research 在 2018 年提出的一种归一化方法，它将每个样本的通道（channel）分成多个“组”（Group），再在每个组内部计算均值与方差来执行归一化。

具体流程：

step1: 假设特征图维度为 (N, C, H, W)，其中 N 为 batch size，C 为通道数，H, W 为高宽。

step2: 对同一个样本（即固定 n）下的 C 个通道，划分成若干组（比如 G 组，每组包含 $\frac{C}{G}$ 个通道）。

step3: 对每组中的所有激活值（空间维度 + 分组通道）计算 **均值** 和 **方差**，执行标准化：

$\hat{x} = \frac{x - \mu_g}{\sqrt{\sigma_g^2 + \epsilon}}$ 其中 μ_g 和 σ_g^2 是该组的均值与方差。

小红书账号：小黄搞AI

step4: 最后乘上可学习的缩放系数 γ 并加上偏置 β，与其他 Normalization 的后处理类似。

🌟适用场景与特点🌟

🍇 **小批量训练：** 当显存限制或其他原因导致批量大小无法很大时，BatchNorm 的统计不稳定，此时 GN 更能保证训练的稳定性和加速效果。

🍇 **高分辨率图像或 3D 图像：** 在图像分割、目标检测、医学影像等任务中，图像分辨率很高通常迫使 batch size 不大，GN 是不错的选择。

🍇 **GPU/分布式环境：** 大规模多 GPU 训练时，若每个 GPU 分到的数据很少，BN 统计量偏差大，而 GN 不依赖跨样本计算均值方差，可以在每个 GPU 上独立工作。

特点

1. **对 batch size 不敏感**：与 BN 需要在 mini-batch 上统计不同，GN 的统计仅在“每个样本的分组内”完成。
2. **分组数可调**：通过控制分组数 G 来在“全通道”与“单通道”之间找到平衡。一般小分组更接近 LayerNorm，大分组更接近 InstanceNorm。
3. **计算量与 BN 相当**：与 BN 相比，GN 并不会显著增加额外的计算负担。

89.2 Instance Normalization（IN）

🥥 **核心思想**

Instance Normalization（IN）则是对每个样本、每个通道的激活值**独立**地计算均值和方差。也就是说，在通道维度上进一步细分为单个通道：“**每个样本的每个通道**”都做自己专属的归一化，不和其他通道或其他样本发生数据统计交互。

以 (N, C, H, W) 的特征图为例：

1. 固定某个样本 n、某个通道 c 后，对该通道空间维度 $(H \times W)$ 上的值计算均值 $\mu_{n,c}$ 和方差 $\sigma^2_{n,c}$。
2. 做标准化：$\hat{x}_{n,c,h,w} = \dfrac{x_{n,c,h,w} - \mu_{n,c}}{\sqrt{\sigma^2_{n,c} + \epsilon}}$
3. 同样有可学习的 γ 和 β 用于缩放和平移。

🌟适用场景与特点🌟

🍇 **图像风格迁移（Style Transfer）：** 让每张图的**每个通道都独立归一**化，可以去除不同图像间的全局统计差异，保留风格相关的局部特征。正因如此，IN 在风格迁移中非常流行。

🍇 **GAN 及生成任务：** 有时在生成式网络中，过度依赖批量统计会导致不稳定或多样性不足，IN 帮助消除不同样本间的干扰，让模型更专注于当前样本的特征。

小红书账号：小黄搞AI

特点

1. **最彻底的个体归一化**：每个样本、每个通道都独立计算统计量，忽略了样本间和通道间的分布差异。
2. **对分类等识别任务不一定有利**：因为 IN 可能会过度移除全局特征，导致识别时无法利用跨样本的统计信息；但在需要突出个体差异或风格的生成任务中，反而是优势。
3. **与其他 Normalization 差异大**：BN、LN、GN 通常会对一组通道或一个 Batch 的通道进行统计，而 IN 极端到每个通道自身处理，被称为“**Instance**”归一化。

GN vs. IN 的核心区别

- **Group Normalization（GN）**
  - **归一化维度**：同一个样本的部分通道（按组分割）+ 空间维度
  - **优点**：对小批量更稳定，可控分组数，适合多通道任务或小 batch 训练
  - **使用场景**：图像分割、目标检测、医疗影像等，常伴随高分辨率、小 batch
- **Instance Normalization（IN）**
  - **归一化维度**：单样本单通道 + 空间维度
  - **优点**：彻底消除不同样本、不同通道之间的统计差异，对风格迁移或生成任务有利
  - **使用场景**：图像风格迁移（Style Transfer）、GAN 等注重“个体风格”保留的生成式模型

从根源上理解

- **GN**：在 BN 与 IN 之间找平衡，它不依赖 batch 统计，但仍保留一定的通道组间信息，不会像 IN 那样“极端”。
- **IN**：最小化到“单通道单图”，不顾及跨通道跨样本分布，让每条数据都有独立的统计归一化，突出个体差异、淡化全局分布。

一句话概括

- **Group Normalization（GN）**：小团队体检，适合小批量、多通道图像。
- **Instance Normalization（IN）**：自己给自己量体重（单通道），适合需要保留个体“风格”的生成式任务。

90. 不同的 Normalization 之间有什么区别?

90.1 六种 Normalization 的对比表

下面给出一份涵盖 **Batch Normalization（BN）**、**Layer Normalization（LN）**、**RMSNorm**、**DeepNorm**、**Group Normalization（GN）**、**Instance Normalization（IN）** 的比较汇总。

小红书账号：小黄搞AI

A	B	C	D	
方法	**归一化维度**	**是否依赖 Batch 大小**	**核心特点/适用场景**	**是否需配合**
BN(Batch Normalization)	对 **mini-batch** 内同一通道（或特征维度）计算均值和方差	**依赖**。需要足够大 batch 来得到稳定统计量	- CNN中使用广泛，能显著加速收敛 - 对 batch size 敏感，小批量时统计不稳 - 常用于图像任务(ResNet/各种CV模型)	通常内置在中，测试时或全局统计
LN(Layer Normalization)	对 **单样本在本层的所有神经元**（或特征维度）做归一化	**不依赖** Batch 大小	- 主要应用于 RNN / Transformer / NLP - 对变长序列或小批量场景更友好 - BERT、GPT、Transformer 等模型默认使用	常与残差连Transforme的“Add & LayerNorm
RMSNorm	类似 LN，但只基于 **RMS（均方根）** 进行缩放，不减去均值	**不依赖** Batch 大小	- 计算量更低，不需要减均值 - 在部分大型语言模型（GPT-NeoX、Falcon 等）中流行 - 效果与LN接近，推理效率更高	有时与残差替代常规的
DeepNorm	非单纯归一化方式，而是在 **深层Transformer的残差分支** 中引入特殊缩放/修正	**不依赖** Batch 大小	- 专为 **超深网络**（如100+层Transformer）提高训练稳定性 - 通过在残差连接和 LayerNorm/ RMSNorm搭配时额外加因子来平衡梯度	需要在每层差处做额外LN或RMSN用
GN(Group Normalization)	将通道（channel）分成若干个 **Group**，在每个样本的同一 Group 维度上计算均值和方差	**部分依赖**。对 batch size 要求相对小，但仍对通道分组有依赖	- 在图像任务中 batch size 小、通道数多时，GN可替代BN - 比BN更灵活，对小批量训练更稳定- 常见于一些2D/3D图像、分割、检测等场景	可与残差等适合需要对度控制的场
IN(Instance Normalization)	对 **单张图像/单条数据的每个通道** 独立计算均值和方差（**channel-wise + spatial-wise**），不和其他样本混合	**不依赖** Batch 大小	- 风靡于图像 **风格迁移** (Style Transfer) 等 - 让每个样本、每个通道都独立归一化，消除样本间统计差异 - 适合需要强调“个体风格”的场景	通常与生成GAN或风格起使用

小红书账号：小黄搞AI

90.2 六种 Normalization简要说明

Batch Normalization（BN）

- **归一化维度**：对同一个 mini-batch 内通道维度上计算均值、方差。
- **优势**：在图像任务中有效加速收敛，对大批量训练非常友好。
- **局限**：当批量过小时，统计不稳；对 RNN、变长序列效果不佳；推理时需使用移动平均或全局统计。

Layer Normalization（LN）

- **归一化维度**：对单条样本在本层的所有特征进行均值、方差归一化。
- **优势**：与 batch size 无关，特别适合 NLP、变长序列；在 Transformer 中成为主流做法。
- **局限**：在很大 batch 场景下并没有显著坏处，但相比 BN，有时对CNN而言收益不如后者明显。

RMSNorm

- **归一化方式**：省略了减均值操作，仅使用激活的 RMS（均方根）进行缩放。
- **优势**：计算更简单，性能不逊于 LN；在大型语言模型中也能获得较好收敛稳定性。
- **局限**：主要在 NLP/Transformer 领域使用，尚未像 BN/LN 那样广泛通用。

DeepNorm

- **核心**：并非单纯“归一化公式”更改，而是在 **深层（超深）Transformer** 的残差结构处，添加特定缩放因子。
- **目标**：解决 **网络层数极深时** 的训练不稳定（如梯度爆炸/消失）。
- **使用**：通常与 LN 或 RMSNorm 配合；对 100+ 层的大型预训练模型帮助明显。

Group Normalization（GN）

- **归一化维度**：对每个样本的通道分组，分组内计算均值方差。
- **优势**：对 batch size 小的场景仍然稳定（BN在小batch时就很不稳定）；可在2D/3D图像等高通道任务中细粒度归一化。
- **局限**：需要选择合适的组数（group）才能达到理想效果，存在一定调参难度。

Instance Normalization（IN）

- **归一化维度**：同一个样本、同一通道内（空间维度）计算均值和方差，每个通道彼此独立。
- **优势**：在风格迁移中消除不同样本间的特征差异，让模型更注重“个人风格”。
- **局限**：对于需要全局统计一致性的任务（如分类），IN 可能过于分割信息，反而不利于分类语义保持。

90.3 如何挑选合适的 Normalization？（🌟核心考量）

小红书账号：小黄搞AI

🍋 如何挑选合适的 Normalization

主要是看任务类型

- **图像（CNN）+ 大batch**：BatchNorm 常见且成熟；
- **图像 + 小batch**：可考虑 GroupNorm (GN)；
- **风格迁移**：InstanceNorm (IN)；
- **NLP/Transformer**：LayerNorm (LN)或 RMSNorm；
- **超深Transformer**：DeepNorm（配合LN或RMSNorm）保证稳定性。

看模型规模/深度

- 网络层数极深（几百层）时，需要额外考虑 DeepNorm；
- 一般深度（几十层）用常规 LN/BN/GN 即可。

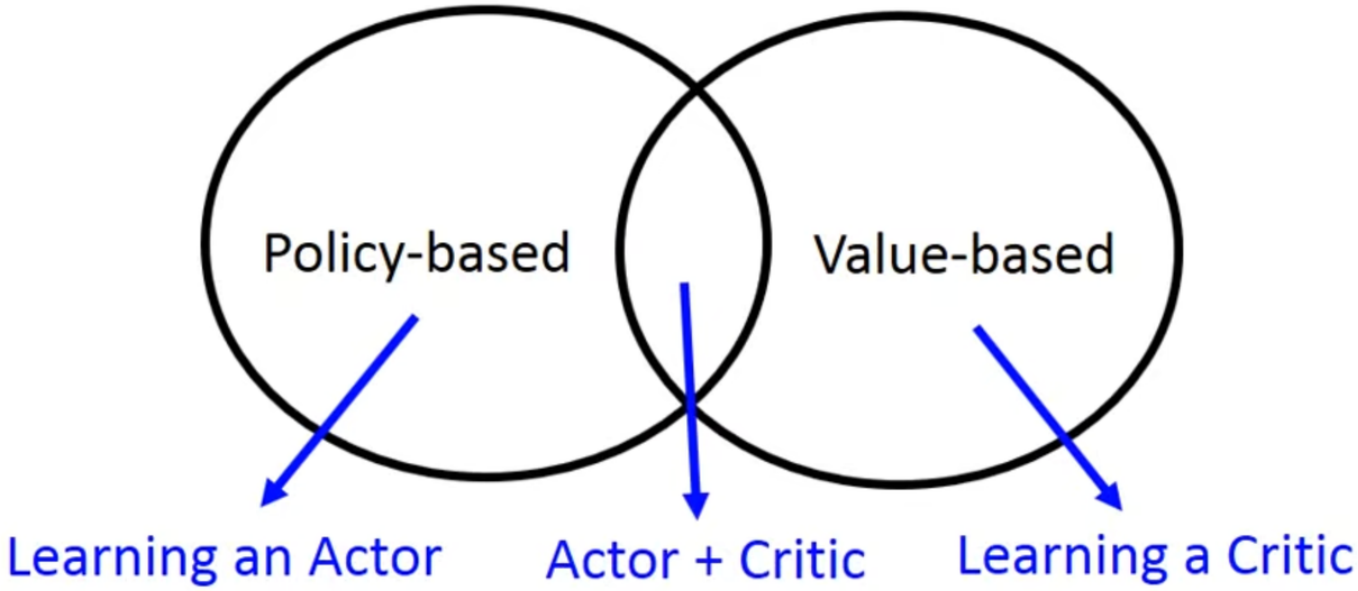
看硬件/批量限制

- 如果显存或数据限制导致 batch size 不大，BN 可能不稳定，应考虑 LN、GN、IN、RMSNorm。

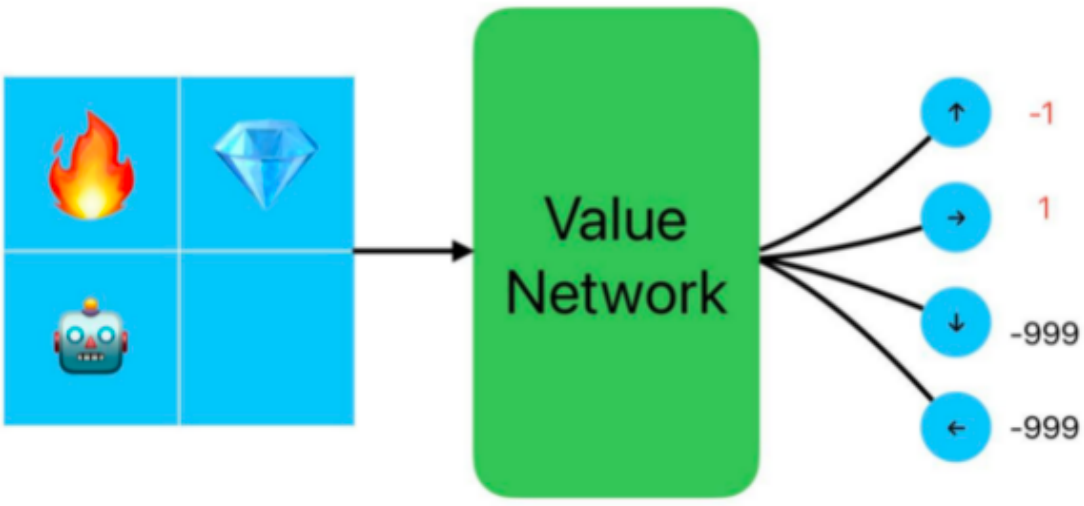
小红书账号：小黄搞AI

91. 强化学习算法是怎么分类的?

强化学习算法多种多样，可以根据不同视角来分类，这里先介绍三种广为人知的分类方式。



91.1 Value-Based 方法



Value-Based 方法的基本思想

- 在每个状态（State）下，**为每个可行的动作（Action）都计算一个“价值”（Value）**。
- 选择 Value 最高的动作来执行。

小红书账号：小黄搞AI

在强化学习中，这个 Value 值通常通过贝尔曼方程（Bellman Equation）进行更新和迭代，既考虑当前能获得的即时奖励，也考虑未来时刻可能获得的长期收益。

示例： 假设某个状态 State A 下可行的三个动作分别是 `Action1, Action2, Action3`，训练过程中会迭代更新其对应的价值 `V(s, a)`（或常见写法 `Q(s, a)`），并最终选取价值最大的动作。

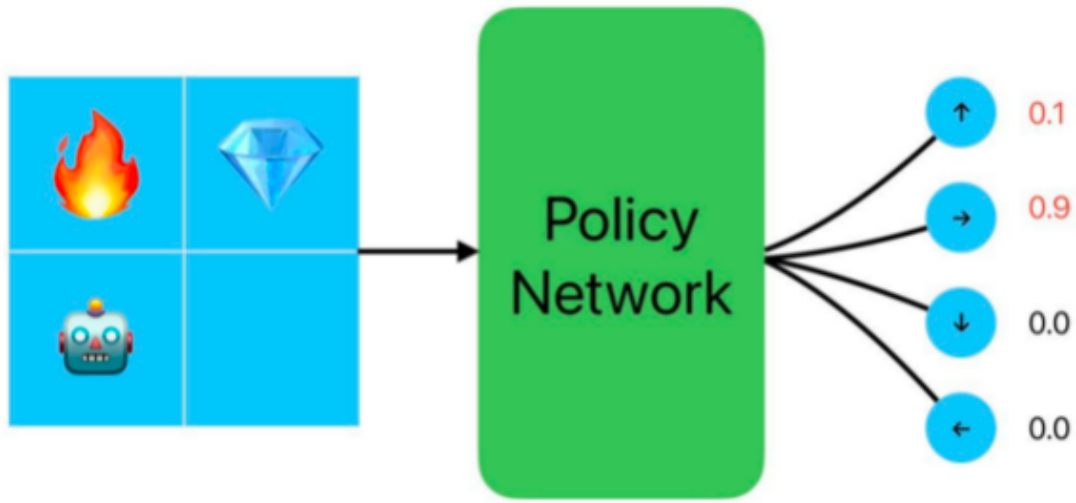
代表性算法

- Q-Learning
- SARSA (State-Action-Reward-State-Action)

适用场景

- 动作空间离散（Discrete）
  - 例如 Pacman 的上下左右
- 或者最终目标策略是**确定性的（Deterministic）**
  - 例如，每次在某个状态就执行同一个动作

91.2 Policy-Based 方法



Policy-Based 方法的基本思想

Policy-Based 方法则是直接对“策略（Policy）”本身进行建模，从而学习在各状态下，动作对应的概率分布，而不是去间接计算 Value。

示例：

在给定状态 State A 时，你可能学得一套策略：

- 以 0.5 的概率选择 `Action1`
- 以 0.3 的概率选择 `Action2`
- 以 0.2 的概率选择 `Action3`

小红书账号：小黄搞AI

通过与环境的交互以及梯度（Gradient）更新，你会得到一个能最大化预期回报的**最优策略**。

代表性算法

Policy Gradient（PG）系列

适用场景

- 动作空间连续
  - 例如机械臂控制中，关节旋转角度是浮点数，而非简单的离散动作
- 在某些状态下需要随机策略
  - 例如剪刀石头布中，最优策略是 **1/3 剪刀 + 1/3 石头 + 1/3 布**

91.3 Actor-Critic 方法

这是 **将 Value-Based 和 Policy-Based 结合在一起** 的思路

- **Actor**部分负责学习出策略（Policy）
- **Critic**部分负责评价策略好坏（类似 Value 计算）
  这类算法兼具两者优点，近年来在深度强化学习场景中非常流行。

91.4 其他分类

除了以上三种主流的算法思路，我们还会以**是否需要环境模型（Model）**进行分类：

- **Model-based**： 先对环境进行建模，学习出环境的动态方程或状态转移规律；此时就能用动态规划（DP）或蒙特卡洛树搜索（MCTS）等方法来选取贪心策略。
- **Model-free**： 不对环境进行明确建模，直接通过 Value 或 Policy 函数的迭代来寻找最优策略；经典的 Q-Learning 即是其中代表。

92. DeepSeekV3有啥技术特点?

92.1 MLA多层注意力架构 (Multi-Head Latent Attention, MLA)

设计理念：MLA通过**低秩压缩技术减少注意力机制中的键值缓存 (Key-Value Cache) 存储需求**。具体而言，它使用降维投影矩阵对键和值进行联合压缩，从而大幅减少推理阶段的KV缓存大小，同时维持性能与标准多头注意力机制 (MHA) 相当。

优势：显著降低了推理过程中显存需求，特别是在处理长上下文时表现优异。

小红书账号：小黄搞AI

92.2 FP8混合精度训练框架

技术细节：

- 通过引入FP8数据格式，将矩阵乘法 (GEMM) 的大部分计算迁移到FP8精度，从而提升训练效率。
- 关键模块（如嵌入层、输出头、归一化操作等）保留较高精度（BF16或FP32），以确保数值稳定性。

创新点：提出了细粒度量化策略 (Fine-Grained Quantization)，通过1×128或128×128分块降低量化误差，同时采用高精度累积方法进一步提高计算精度。

成果：相比BF16基准训练方法，FP8训练框架在性能波动小于0.25%的范围内实现了显著的内存和时间优化。

92.3 DualPipe跨节点通信算法

创新点：**DualPipe通过计算与通信阶段的重叠安排，显著减少了通信开销和流水线气泡（Pipeline Bubbles）。**

技术实现：

- 重新排列正向与反向计算块内的注意力、全连接层等组件，并动态调整通信和计算所占的GPU资源比例。
- 使用高效的跨节点All-to-All通信内核，充分利用InfiniBand和NVLink带宽。

效果：即使在大规模模型训练中，DualPipe算法也实现了近乎零的All-to-All通信开销。

92.4 无辅助损失的负载均衡策略

背景：在专家模型 (MoE) 中，负载不平衡会导致路由崩溃和计算效率低下。传统方法依赖辅助损失进行负载均衡，但可能导致性能退化。

创新方案：DeepSeek采用无辅助损失的负载均衡策略，**通过为每个专家引入动态调整的偏置项来平衡负载**，而不对模型性能造成负面影响。

补充措施：引入序列级的辅助损失以避免单一序列内的极端负载不平衡。

92.5 跨节点全对全通信内核

优化细节：

- 在节点间的通信中，**优先利用NVLink处理节点内通信，InfiniBand处理节点间通信。**

小红书账号：小黄搞AI

- 将令牌路由限制在最多4个节点中，以减少InfiniBand流量。

结果：实现了通信和计算的完全重叠，最大程度利用了GPU的计算能力。

92.6 MTP（多令牌预测）技术

技术实现：引入**多令牌预测目标**，使模型在每个位置预测多个未来令牌，并保持完整的因果链。

优点：

- 增强训练信号密度，提高数据效率。
- 为推理加速提供潜在支持，例如通过投机解码减少延迟。

92.7 数据精炼

技术实现：引入多令牌预测目标，使模型在每个位置预测多个未来令牌，并保持完整的因果链。

优点：

- 增强训练信号密度，提高数据效率。
- 为推理加速提供潜在支持，例如通过投机解码减少延迟。

92.8 DeepSeek-R1蒸馏

创新点：从DeepSeek-R1系列模型中提取长链推理能力，通过**蒸馏方式将验证和反思模式注入到DeepSeek-V3中。**

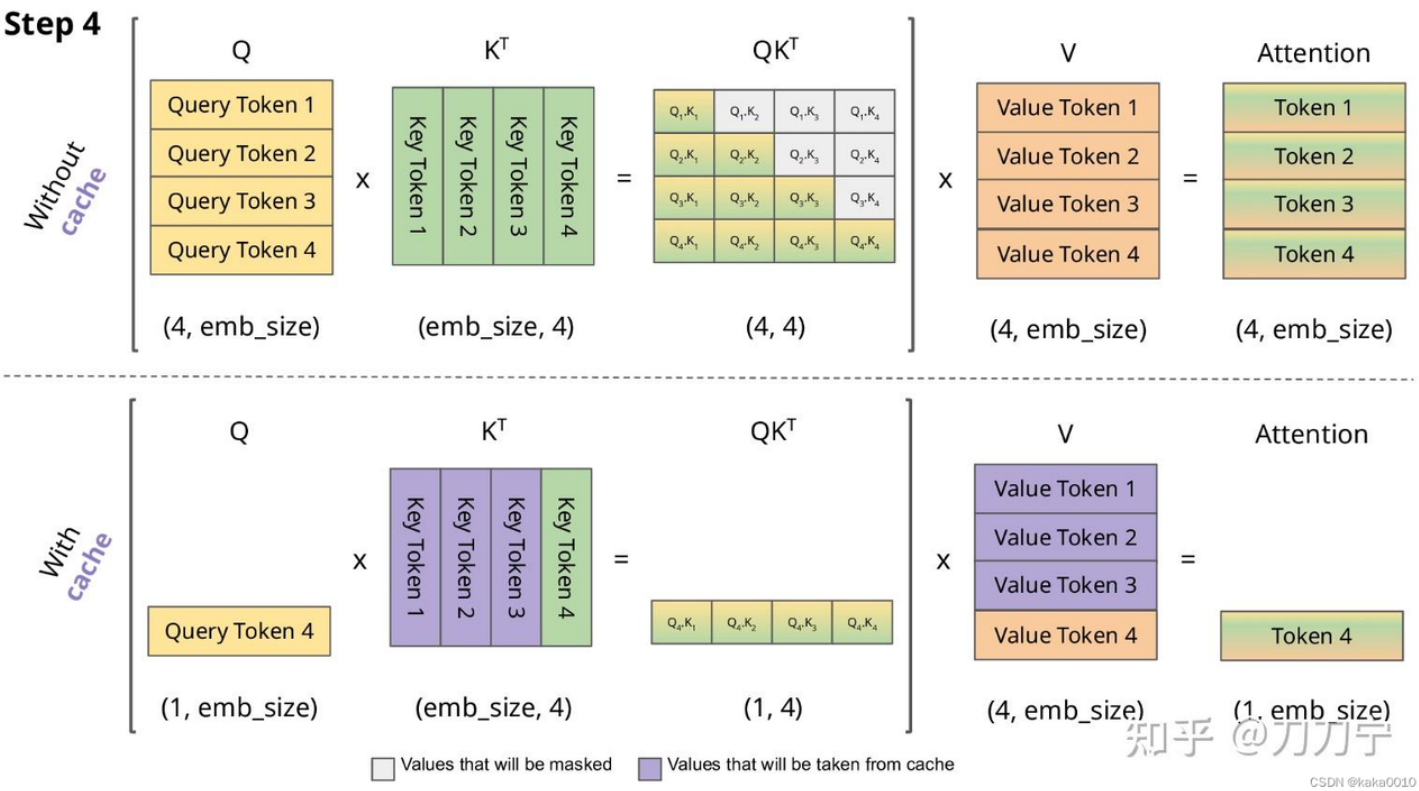
成果：增强了模型的推理能力，同时对输出的风格和长度进行了精细控制。

93. 为什么要优化KV cache?

93.1 KV cache 是什么？为什么它如此重要？

KV cache（Key-Value 缓存）是在 Transformer 模型中保存历史输入的键（Key）和值（Value）的存储机制，目的是在生成序列时**避免重复计算，从而大幅提升效率。**每个生成步骤需要基于之前的输入计算注意力分数，而保存 KV cache 可减少重复操作。

小红书账号：小黄搞AI



KV cache的作用

1. **避免重复计算：**
   - 保存键和值后，后续计算仅需基于缓存数据更新新 token 的注意力分数。
   - 若无 KV cache，每次生成新 token 都需从头计算完整的输入序列，效率大幅下降。
2. **提升推理效率：**
   - KV cache 的核心是通过存储历史计算结果，加速后续序列生成。
3. **减少显存使用的冲突：**
   - 推理中每一层的 Key 和 Value 会持续增长，因此需要优化其占用的显存空间。

93.2 为什么要优化 KV cache?

93.2.1 从显存占用的角度解释

Transformer 模型中，KV cache 的显存需求和模型权重本身相比会迅速膨胀，尤其在长序列生成任务中。以公式计算为例：

$$\text{KVcache 大小} = 2 \times \text{precision} \times n_{layers} \times d_{model} \times seqlen \times batch$$

KV Cache 计算公式解读

KV cache 的总大小与以下几个因素有关：

- batch : Batch size（批处理大小）。每次推理中同时处理的输入序列数量。

小红书账号：小黄搞AI

- $seqlen$：序列长度（$s+n$）（s 是输入序列长度，n 是生成的输出序列长度）。KV cache 中需要保存输入和已生成的所有 token 的信息。
- n_{layers}：模型层数。每一层的注意力机制都会独立保存 Key 和 Value。
- d_{model}：模型的隐藏层维度。通常等于注意力头数量乘以每个注意力头的维度（$\text{num_heads} \times k$）。
- 2：Key 和 Value 两组数据需要缓存。
- $precision$ 数据格式，每个数值占用的字节数（例如，FP16 为 2 字节）。

假设 GPT-3（175B 参数）为例

- Batch size $b = 4$
- 输入序列长度 $s = 4096$
- 输出序列长度 $n = 0$（假设只计算输入序列）
- 模型层数 $\text{nlayers} = 96$
- 每个注意力头维度 $k = 128$
- 注意力头数量 $\text{num_heads} = 96$（假设每层有 96 个注意力头）
- 模型隐藏层维度 $\text{dmodel} = \text{num_heads} \times k = 96 \times 128 = 12,288$

计算 KV cache 的大小：

$$KV\,cache\text{大小} = 2 \times 2 \times 96 \times 12,288 \times 4096 \times 4(bytes) = 4 \times 96 \times 12,288 \times 4096 \times 4(bytes) = 4 \times 96 = 384384 \times 12,288 = 4,718,5924,718,592 \times 4096 = 19,329,373,95219,329,373,952 \times 4 = 77,317,495,808(bytes) \approx 72(GB)$$

最终结果：

在上述假设下，**KV cache 大约需要 72 GB 的显存（若使用 FP16 存储）**。

公式揭示的增长规律

- batch：同时处理的序列数量增加，显存需求线性增加。
- seqlen：序列长度每增加一个 token，显存需求线性增加。
- n_{layers}：模型越深，KV cache 的显存占用越高。
- d_{model}：隐藏层维度和注意力头数量决定了每个 token 的存储大小。
- 2×2：Key 和 Value 分开存储，并使用 FP16 表示（存储位宽会影响大小）。

优化方向

1. **减少 KV cache 的存储位宽**：例如，使用 FP8 代替 FP16，可以将显存需求减半。
2. **分块存储或稀疏存储**：通过分块或稀疏化技术，降低显存峰值需求。
3. **序列压缩**：通过减少存储的冗余信息，优化显存需求。

小红书账号：小黄搞AI

93.2.2 kv cache优化的必要性

> 参考资料：https://blog.csdn.net/javastart/article/details/137948164

batch size	s+n	KV cache(GB)	KV cache/weight
4	4096	75.5	0.22
16	4096	302	0.86
64	4096	1208	3.45

参考上图，随着 batch size 和 长度的增大，KV cache 占用的显存开销快速增大，甚至会超过模型本身。从LLM的趋势上而讲，主要有三个方面来说明kv cache优化的必要性：

1. **总体趋势上LLM 的窗口长度在不断增大**，因此就出现一组主要矛盾，即：对不断增长的 LLM 的窗口长度的需要与有限的 GPU 显存之间的矛盾。因此优化 KV cache 非常必要。OpenAI API场景，API最烧钱的是输入而非输出，输入包括prefill prompt 和conversation，长度动辄数10k token。虽说每输入token比每输出token便宜，但能够降低kv重新计算的开销，无论是硬件资源门槛，还是模型推理降本，都有着极为积极的作用。
2. 对于消费级显卡这种性价比较高的显卡而言，显存容量相对较小，KV cache从一定程度上**降低了模型的batch size**，因而KV cache优化在工程落地中更显重要。

框架	模型	input/output	机器配置	tokens/s	最佳batch_size	gpu/cpu负载
TGI	llama2 70B	128/512	4090*8，32c	291	13	cpu25%，gpu95%
TGI	llama2 70B	128/512	A800，32c	1122	43	cpu25%，gpu95%

从上表能够看出，类似4090等消费级显卡，其主要的瓶颈时batch_size，而影响batch_size的主要因素就是显存容量，而KV cache在推理过程中占用大量的显存。如果能通过KV cache降低显存占用，从一定程度上就能提升消费级显卡的性价比，带来非常高的商业收益。

3. **KV cache 优化在 AIGC 场景中的重要性**

除了语言模型，视频生成和图像生成模型也面临类似问题。以 Stable Diffusion 或视频生成模型（如 Sora）为例：

- 模型特点：虽然模型参数较小（通常 <10B），但生成序列长度非常长（接近 1000k patches）。
- 显存比例：KV cache 对显存的占用可能达到 90%，甚至更多。

实验数据支持

- batch size = 1：KV cache 与模型参数的显存占比达到 **10:1**（例如4090的24G显存，2G分给模型，22G分给kv cache）。
- batch size = 4：这一比例进一步增大到 **40:1**（batch size越大，kv cache的显存越大）。

在此类任务中，优化 KV cache 不仅能够提高显存利用率，还可以直接提升生成速度和性能。

小红书账号：小黄搞AI

93.3 研究方向：如何优化 KV cache?

压缩存储

使用更高效的数值表示（如 FP8）或稀疏表示法，降低单个键值对的存储需求。

动态分配缓存区，仅保存最重要的 KV 数据。

分层缓存

引入多层缓存机制（如显存与主存协作），实现更高效的数据存储与读取。

分块存储

对长序列任务按块拆分 KV 数据，减少显存峰值需求。

推理算法优化

利用注意力机制的低秩特性，减少缓存数据的存储规模。

总结

研究 KV cache 的优化方法，不仅能解决当前大语言模型推理中的显存瓶颈，还能推动多模态生成任务的发展。随着序列长度不断增加，KV cache 的设计将成为 AI 基础设施中的关键一环。

94. MLA（Multi-Head Latent Attention ）是什么?

94.1 多头注意力（MHA）的数学形式

94.1.1 基础定义

对于一个输入 Token 的隐藏向量 $\mathbf{x}_i \in \mathbb{R}^d$ ，在多头注意力中，我们通常会使用三个投影矩阵：

$$\mathbf{W}_Q \in \mathbb{R}^{d\times(n_h\cdot d_h)},\quad \mathbf{W}_K \in \mathbb{R}^{d\times(n_h\cdot d_h)},\quad \mathbf{W}_V \in \mathbb{R}^{d\times(n_h\cdot d_h)}$$

它们分别把 $\mathbf{x}_i$ 投影为：$\mathbf{Q}_i = \mathbf{W}_Q\mathbf{x}_i,\quad \mathbf{K}_i = \mathbf{W}_K\mathbf{x}_i,\quad \mathbf{V}_i = \mathbf{W}_V\mathbf{x}_i$

这里 $\mathbf{Q}_i, \mathbf{K}_i, \mathbf{V}_i$ 各自是 $\mathbb{R}^{n_h\cdot d_h}$ 维，再把它们 reshape（或 split）成 n_h 组，每组维度是 d_h ，即：

$$\mathbf{q}_i^{(h)} \in \mathbb{R}^{d_h},\quad \mathbf{k}_i^{(h)} \in \mathbb{R}^{d_h},\quad \mathbf{v}_i^{(h)} \in \mathbb{R}^{d_h},\quad h = 1, 2, \ldots, n_h.$$

注意力打分时，对于当前查询的第 t 个 Token，我们要跟所有历史 Token（从 1 到 $t-1$）进行计算，以得到加权后的输出（隐状态 $\mathbf{o}_t$）：$\mathbf{o}_t^{(h)} = \sum_{j=1}^{t} \mathrm{Softmax}_j\left(\frac{(\mathbf{q}_t^{(h)})^\top \mathbf{k}_j^{(h)}}{\sqrt{d_h}}\right)\mathbf{v}_j^{(h)}$

最后将所有头的输出拼接后，再乘一个输出矩阵 $\mathbf{W}_O \in \mathbb{R}^{(n_h\cdot d_h)\times d}$ 得到最终结果：

$\mathbf{o}_t = \mathbf{W}_O\left[\mathbf{o}_t^{(1)}; \ldots; \mathbf{o}_t^{(n_h)}\right]$

94.1.2 MHA 的缓存开销

小红书账号：小黄搞AI

在推理时，需要缓存**所有过去 Token** 的 $\mathbf{K}, \mathbf{V}$。因为对每个 Token 都要计算到后续的注意力中，对第 i 个 Token 的头向量是 $\mathbf{k}_i^{(h)}, \mathbf{v}_i^{(h)}$，合计维度是 $2 \times n_h \times d_h$。当有很多 Token 时，缓存量巨大。

94.2 分组注意力（GQA）如何“合并”并减少缓存

94.2.1 GQA 的低秩投影

GQA（Grouped Query/Key/Value Attention）的核心是在**Key、Value 的投影上做“合并”**。与其给每个头都单独一个投影矩阵，不如将所有 Key、Value 的投影矩阵合并为少数几个分组，甚至只有 1 组。举例：设只有 1 组，那么可以定义一个矩阵 $\mathbf{W}_c$：$\mathbf{W}_c \in \mathbb{R}^{d \times d_c}, \quad d_c = g \cdot (d_k + d_v) < d$

对于每个 Token 的隐状态 $\mathbf{x}_i$，得到：$\mathbf{c}_i = \mathbf{W}_c \mathbf{x}_i \in \mathbb{R}^{d_c}$

然后把 $\mathbf{c}_i$ 拆分成两部分（一半用作 K，一半用作 V），再均分成 g 组，而后**简单复制**给 n_h 个头。这样就能用一个低维 $\mathbf{c}_i$ 生成所有头需要的 $\mathbf{k}_i^{(h)}$ 与 $\mathbf{v}_i^{(h)}$。

- 好处：**缓存**只用存 $\mathbf{c}_i$，是 $\mathcal{O}(d_c)$ 维度，显著小于 $\mathcal{O}(n_h \times d_h)$。
- 坏处：因为**所有头都共享**那同一个（或少数几个） Key、Value，只能靠分割/复制来“分头”。这样**各头缺乏灵活性**，表达能力会有所下降。

94.3 MLA：在 GQA 基础上增加“线性上采样矩阵”

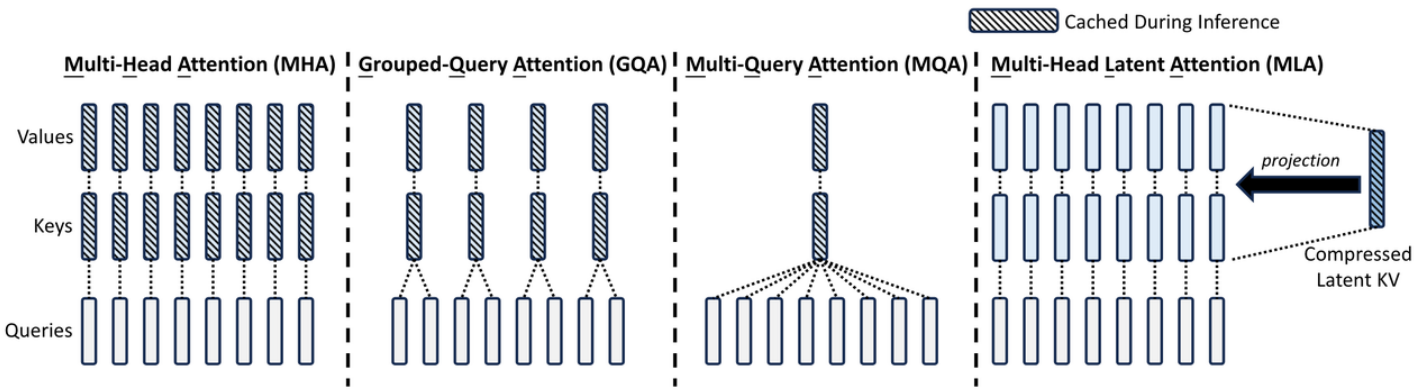


Figure 3 | Simplified illustration of Multi-Head Attention (MHA), Grouped-Query Attention (GQA), Multi-Query Attention (MQA), and Multi-head Latent Attention (MLA). Through jointly compressing the keys and values into a latent vector, MLA significantly reduces the KV cache during inference.

MLA（Multi-Head Latent Attention）保留 GQA 那种**先做低维投影**的主思路，但不满足于后面只做简单的“切分 + 复制”，而是给每个头**一个可学习的线性上采样矩阵**来恢复真正需要的 $\mathbf{k}_i^{(h)}$、$\mathbf{v}_i^{(h)}$。

94.3.1 MLA 的关键公式

先做低秩投影（与 GQA 相同）

$\mathbf{c}_i = \mathbf{W}_c \mathbf{x}_i \in \mathbb{R}^{d_c}$，其中 $\mathbf{W}_c \in \mathbb{R}^{d \times d_c}$

再给每个头都引入“上采样”

对于第 h 个头的 Key、Value，可以定义：

小红书账号：小黄搞AI

$$\mathbf{k}_i^{(h)} = \mathbf{U}_k^{(h)}\,\mathbf{c}_i, \quad \mathbf{v}_i^{(h)} = \mathbf{U}_v^{(h)}\,\mathbf{c}_i$$
$$\mathbf{U}_k^{(h)} \in \mathbb{R}^{d_h \times d_c}, \quad \mathbf{U}_v^{(h)} \in \mathbb{R}^{d_h \times d_c}$$

这样一来，每个头都能得到**不同**的 Key、Value，而不是 GQA 那种“简单复制”。这就增强了表达能力，能逼近甚至超过传统 MHA。

94.3.2 推理阶段如何缓存

如果我们显式缓存 $\mathbf{k}_i^{(h)}, \mathbf{v}_i^{(h)}$，那就变成跟 MHA 一样大了，失去 GQA 的优势。

MLA 的巧思是：只缓存 $\mathbf{c}_i$；在推理计算注意力打分时，再用 $\mathbf{U}_k^{(h)}$ 等恢复 Key、Value。

可进一步将 $\mathbf{U}_k^{(h)}$ “吸收”到 $\mathbf{q}_t^{(h)}$ 的投影中，如此就不必在每次都计算 $\mathbf{k}_j^{(h)}$，从而在实现中大大减少重复和显存带宽消耗。

94.3.3 RoPE 与低秩投影的兼容性：解耦 RoPE

在含有位置编码（RoPE, Rotary Position Embedding）时，通常我们对 Q、K 进行旋转变换：

$$\mathbf{q}_t^{(h)} \leftarrow \mathrm{RoPE}(\mathbf{q}_t^{(h)}), \quad \mathbf{k}_j^{(h)} \leftarrow \mathrm{RoPE}(\mathbf{k}_j^{(h)})$$

但如果我们在低维 $\mathbf{c}_i$ 里再加 RoPE，就可能无法很好地对齐。MLA 的做法是**解耦**：留一套专门的 Key 可以加 RoPE，然后和那条低秩 Key 拼起来：

$$\mathbf{k}_t^{(h)} = \big[\underbrace{\mathbf{k}_t^{(h),C}}_{\text{低秩分支}}; \underbrace{\mathrm{RoPE}(\mathbf{U}_k^{(h),R}\,\mathbf{x}_t)}_{\text{带 RoPE 的分支}}\big]$$

这样既能保持 RoPE 效果，也还能用到低秩压缩带来的缓存优势。

当然，**多出来一条带 RoPE 的分支**也会稍微增加缓存/计算，但依旧比完全 MHA 的缓存要小。

94.4 为什么 MLA 依旧能在推理时带来速度提升？

94.4.1 推理过程分两步：Prefill 与 Generation

Prefill：

一次性处理输入序列，得到所有 Token 的状态并存入 KV Cache。这一步对**带宽和显存**要求最高。

Generation：

逐 Token 生成输出，每产生一个 Token，就访问全部缓存。

小红书账号：小黄搞AI

94.4.2 MLA 在 Prefill & Generation 的表现

Prefill 阶段

MLA 要做**额外的上采样变换**（计算 Key、Value），计算量上比 GQA 多一些；但**KV Cache 小**，带宽压力降低。相对传统 MHA，整体也有可能打个平手或更优。

Generation 阶段

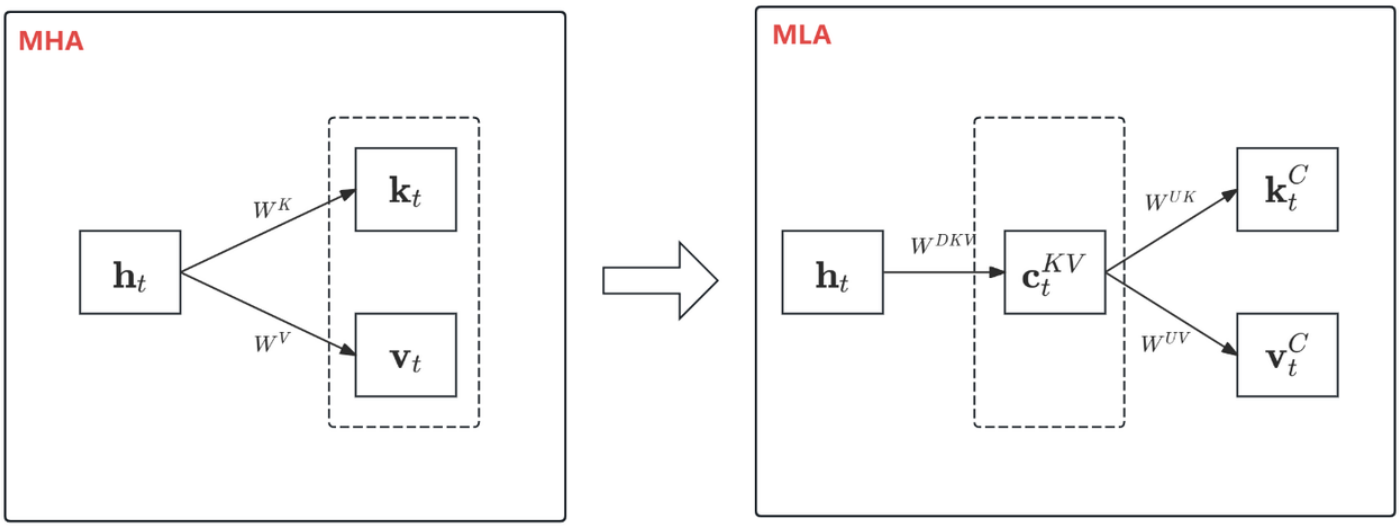
每个新 Token 都要读写过去的 Key、Value；当 Cache 更小，带宽开销也更小；

MLA 通过只缓存 $\mathbf{c}_i$，在需要时再做变换或“吸收到 Q 的投影里”；这样**访问量显著减少**，生成速度就会快不少。

因此，对大语言模型在常见的“长上下文 + 逐 Token 生成”场景下，MLA 往往能带来推理速度和内存占用方面的明显好处。

> **“LLM的推理分两部分：第一个Token的生成（Prefill）和后续每个Token的生成（Generation），Prefill阶段涉及到对输入所有Token的并行计算，然后把对应的KV Cache存下来，这部分对于计算、带宽和显存都是瓶颈，MLA虽然增大了计算量，但KV Cache的减少也降低了显存和带宽的压力，大家半斤八两；但是Generation阶段由于每步只计算一个Token，实际上它更多的是带宽瓶颈和显存瓶颈，因此MLA的引入理论上能明显提高Generation的速度。”——苏神**
>
> 苏剑林. (May. 13, 2024). 《缓存与效果的极限拉扯：从MHA、MQA、GQA到MLA》[Blog post]. Retrieved from https://spaces.ac.cn/archives/10091

94.5 总结



MLA（Multi-Linear Attention） 可以看作是 GQA（分组注意力）的一种增强版：

1. **“低秩投影”**：和 GQA 一样，先把 $\mathbf{x}_i$ 投影到小向量 $\mathbf{c}_i$，只缓存这份小向量。

小红书账号：小黄搞AI

2. **“多头上采样”**：给每个头一个独立可学习的矩阵，把 $\mathbf{c}_i$ 变成**各不相同**的 $\mathbf{k}_i^{(h)}$、$\mathbf{v}_i^{(h)}$，使模型表达能力更接近 MHA。
3. **推理时“吸收”矩阵**：为了不显式缓存所有头的 K、V，MLA 会把上采样矩阵合并到别的部分，只缓存 $\mathbf{c}_i$；这样**保持小缓存**，在 Prefill 和 Generation 阶段都能更好地平衡算力与带宽。
4. **RoPE 解耦**：额外再加一部分 Key 用 RoPE，和低秩 Key 拼接；避免低秩投影与 RoPE 的兼容性冲突。

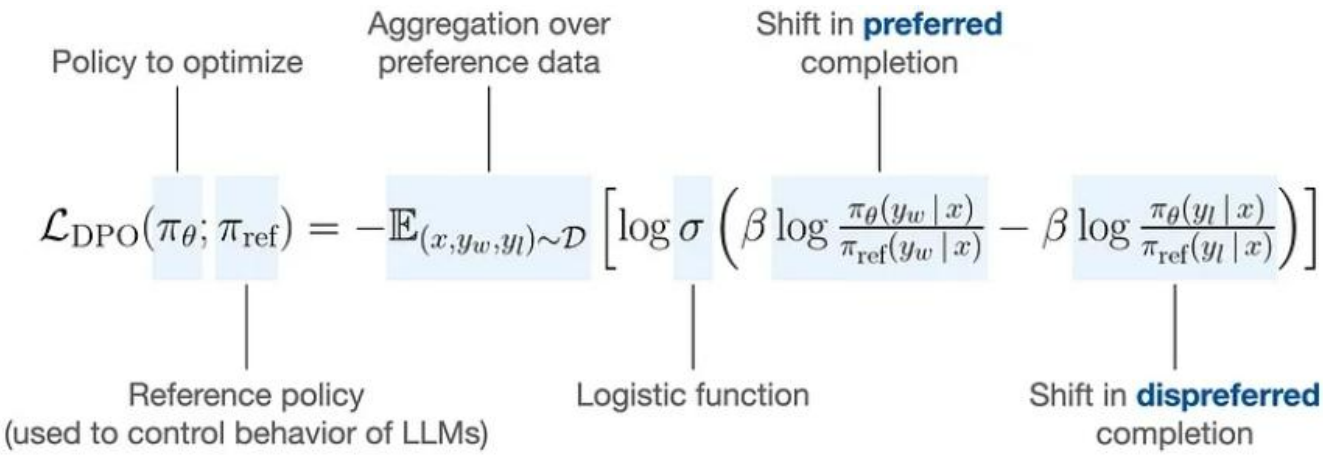
一句话概括：$\mathrm{MLA} = \mathrm{GQA}\ (\text{低缓存}) + \text{线性上采样矩阵}\ (\text{多头的灵活性}) + \text{解耦}\ \mathrm{RoPE}$.
能在尽量少的缓存开销下，提供**媲美或超越 MHA**的注意力表现，特别是对推理速度有很大帮助。

94.6 参考资料

- 官方论文：
  - DeepSeek-V2: Paper Link
- 工程实现：
  - 知乎专栏 GQA/MLA 讲解
- 苏神关于MLA的解释

95. 怎么理解DPO的损失函数？

DPO（Direct Preference Optimization）是一种用于优化语言模型（LLMs）的算法，旨在通过直接优化人类偏好数据来训练模型。DPO 的核心思想是简化传统的 RLHF（Reinforcement Learning from Human Feedback）过程，避免复杂的奖励模型训练，直接通过偏好数据来优化策略。



95.1 DPO 损失函数的基本形式

DPO 的损失函数定义如下：

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

小红书账号：小黄搞AI

其中：

- π_θ 是我们要训练的模型（policy model），目标是让它学会根据人类的偏好生成更好的输出。
- π_{ref} 是参考模型（reference model），通常是一个预训练的语言模型，它的行为是固定的，用来“约束”策略模型，防止策略模型偏离正常行为太远。
- x 是输入文本。
- y_w 是人类偏好的输出（chosen completion）。
- y_l 是人类不喜欢的输出（rejected completion）。
- D 是包含偏好数据的数据集，每个样本由 (x, y_w, y_l) 组成。
- σ 是 logistic 函数（即 sigmoid 函数），用于将差值映射到概率空间。
- β 是一个超参数，用于控制策略模型与参考模型之间的差异。

95.2 损失函数的直观理解

DPO 的目标是最大化策略模型 π_θ 对偏好数据 y_w 的偏好程度，同时最小化对 y_l 的偏好程度。具体来说：

- **分子部分**：$\frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)}$ 表示策略模型对 y_w 的偏好程度相对于参考模型的偏好程度。
- **分母部分**：$\frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)}$ 表示策略模型对 y_l 的偏好程度相对于参考模型的偏好程度。

通过计算这两个比率的对数差值，我们可以衡量策略模型在偏好数据上的表现。如果策略模型更倾向于选择 y_w 而不是 y_l，那么这个差值会变大，损失函数会变小。

策略模型和参考模型的相除是什么意思？

公式中的 $\frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)}$ 和 $\frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)}$ 是关键部分。它们的含义是：

- $\pi_\theta(y_w|x)$：策略模型认为 y_w 是输入 x 的合理输出的概率。
- $\pi_{\text{ref}}(y_w|x)$：参考模型认为 y_w 是输入 x 的合理输出的概率。
- $\frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)}$：策略模型对 y_w 的偏好程度 **相对于** 参考模型对 y_w 的偏好程度。

这个比率的意义是：

- 如果策略模型比参考模型更喜欢 y_w，那么这个比率会大于 1。
- 如果策略模型比参考模型更不喜欢 y_w，那么这个比率会小于 1。

同理，$\frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)}$ 表示策略模型对 y_l 的偏好程度相对于参考模型。

小红书账号：小黄搞AI

为什么要用这个比率？

DPO 的目标是让策略模型更倾向于选择 y_w，而不是 y_l。通过计算这两个比率的差值：

$$\beta \log \frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta \log \frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)}$$

我们可以衡量策略模型对 y_w 和 y_l 的偏好差异。如果策略模型更喜欢 y_w，这个差值会变大；如果策略模型更喜欢 y_l，这个差值会变小。

95.3 损失函数的进一步解释

为了更清楚地理解损失函数，我们可以将其分解为以下几个部分：

1. **Logistic 函数的作用**：
   - σ 函数将差值映射到 $(0, 1)$ 之间，表示策略模型选择 y_w 而不是 y_l 的概率。
   - 我们希望这个概率尽可能接近 1，因此使用 $\log \sigma$ 来最大化这个概率。
2. **超参数 β 的作用**：
   - β 控制策略模型与参考模型之间的差异。较大的 β 会使策略模型更倾向于偏离参考模型，而较小的 β 会使策略模型更接近参考模型。
3. **参考模型的作用**：
   - 参考模型 π_{ref} 用于约束策略模型的行为，防止策略模型过度偏离预训练的语言模型。这样可以避免模型生成不合理或不安全的输出。

95.4 损失函数的简化形式

为了进一步简化损失函数，我们可以对公式进行如下转换：

$$\beta \log \left(\frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} \right) - \beta \log \left(\frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)} \right)$$

利用对数的性质，可以将其改写为：

$$\beta \left(\log \pi_\theta(y_w|x) - \log \pi_{\text{ref}}(y_w|x) \right) - \beta \left(\log \pi_\theta(y_l|x) - \log \pi_{\text{ref}}(y_l|x) \right)$$

进一步简化后，可以得到：

$$\beta \left(\log \pi_\theta(y_w|x) - \log \pi_\theta(y_l|x) \right) - \beta \left(\log \pi_{\text{ref}}(y_w|x) - \log \pi_{\text{ref}}(y_l|x) \right)$$

这个形式与源代码中的计算函数一致。其中：

- 左半部分表示策略模型对 y_w 和 y_l 的偏好差异。
- 右半部分表示参考模型对 y_w 和 y_l 的偏好差异。

损失函数的最终目标

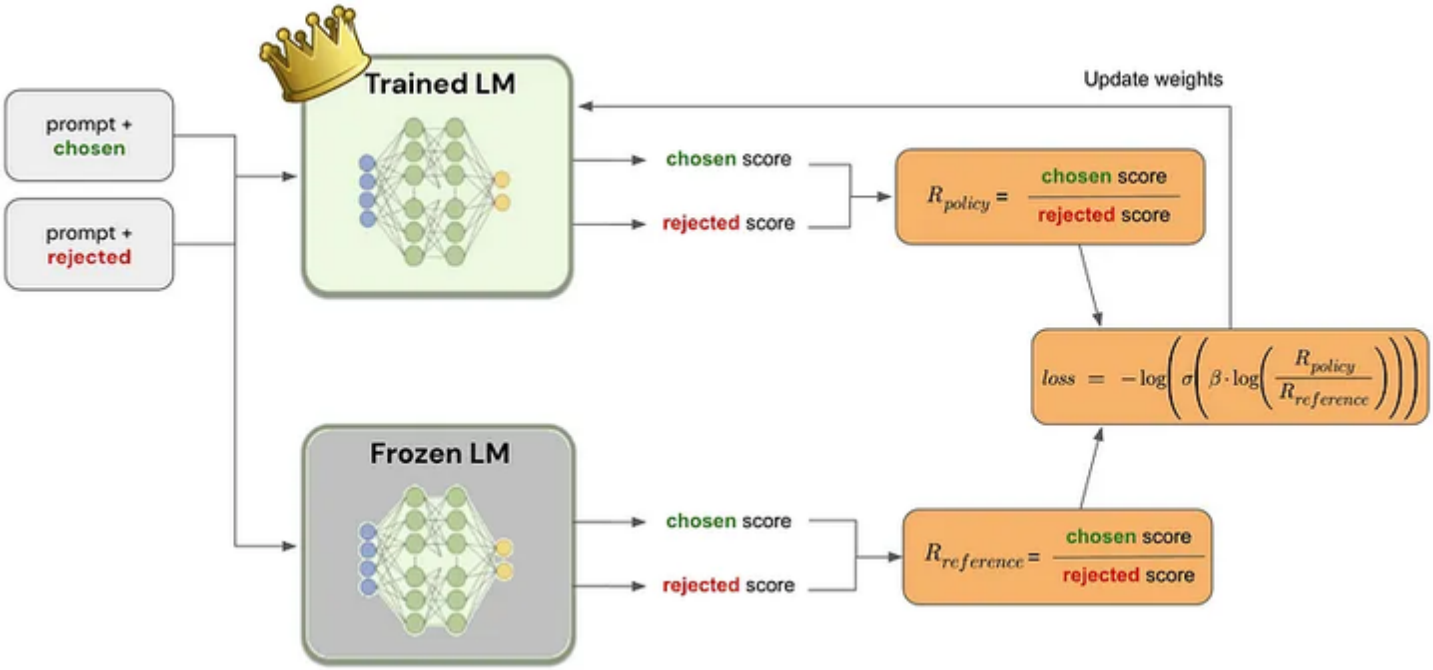
小红书账号：小黄搞AI

DPO 的最终目标是最大化策略模型对偏好数据的偏好程度，同时保持策略模型与参考模型之间的差异在可控范围内。通过这种方式，DPO 能够有效地学习人类偏好，并生成符合人类期望的输出

DPO 的损失函数通过直接优化人类偏好数据，简化了传统的 RLHF 过程。它利用参考模型来约束策略模型的行为，并通过 logistic 函数将偏好差异映射到概率空间。通过这种方式，DPO 能够有效地学习人类偏好，并生成符合人类期望的输出。

96. DPO的微调流程

DPO（Direct Preference Optimization）的微调流程是通过直接优化人类偏好数据来训练模型，使其生成更符合人类期望的输出。



上图展示了DPO微调的大致流程，其中Trained LM即为策略模型，Frozen LM即为参考模型，二者均是先进行SFT微调得到的模型进行初始化，其中Trained LM需要进行训练，Frozen LM不参与训练。两个模型分别针对chosen和rejected进行预测获取对应的得分，再通过DPO的损失函数进行损失计算，进而不断的迭代优化。

96.1 DPO 微调的核心目标

DPO 的目标是让模型学会根据人类的偏好选择更好的输出。具体来说，给定一个输入 x （例如：“写一篇关于环保的文章”），模型会生成两个输出：

- **Chosen (y_w)**：人类喜欢的输出（例如：“环保是保护地球的重要方式，我们应该减少塑料使用。”）

小红书账号：小黄搞AI

- **Rejected (y_l)**：人类不喜欢的输出（例如：“环保不重要，随便扔垃圾也没关系。”）

DPO 的任务是让模型更喜欢 y_w ，而不是 y_l 。

96.2 DPO 微调流程的步骤

步骤 1：初始化模型

- **Trained LM（策略模型）**：这是我们要训练的模型，目标是让它学会根据人类的偏好生成更好的输出。
- **Frozen LM（参考模型）**：这是一个预训练好的模型（比如 GPT-3），它的行为是固定的，用来“约束”策略模型，防止策略模型偏离正常行为太远。

两个模型都是通过 SFT（Supervised Fine-Tuning）微调得到的模型进行初始化的。Trained LM 会参与训练，而 Frozen LM 不参与训练。

步骤 2：输入数据

- 输入数据包括：
  - **Prompt (x)**：输入文本（例如：“写一篇关于环保的文章”）。
  - **Chosen (y_w)**：人类喜欢的输出。
  - **Rejected (y_l)**：人类不喜欢的输出。

步骤 3：计算得分

- **Trained LM** 和 **Frozen LM** 分别对 y_w 和 y_l 进行预测，得到它们的得分：
  - **Chosen Score**：模型认为 y_w 是合理输出的概率。
  - **Rejected Score**：模型认为 y_l 是合理输出的概率。

具体来说：

- **Trained LM** 会计算：
  - $\pi_\theta(y_w|x)$：策略模型对 y_w 的偏好程度。
  - $\pi_\theta(y_l|x)$：策略模型对 y_l 的偏好程度。
- **Frozen LM** 会计算：
  - $\pi_{\text{ref}}(y_w|x)$：参考模型对 y_w 的偏好程度。
  - $\pi_{\text{ref}}(y_l|x)$：参考模型对 y_l 的偏好程度。

步骤 4：计算 DPO 损失

DPO 的损失函数是基于策略模型和参考模型的得分计算的。具体公式为：

小红书账号：小黄搞AI

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

其中：

- σ 是 logistic 函数（即 sigmoid 函数），用于将差值映射到概率空间。
- β 是一个超参数，用于控制策略模型与参考模型之间的差异。

这个损失函数的目标是让策略模型更喜欢 y_w，而不是 y_l。

步骤 5：更新模型权重

- 通过计算 DPO 损失，我们可以得到策略模型的梯度，并使用梯度下降法更新策略模型的权重。
- **Frozen LM** 的权重保持不变，它只用于提供参考得分。

步骤 6：迭代优化

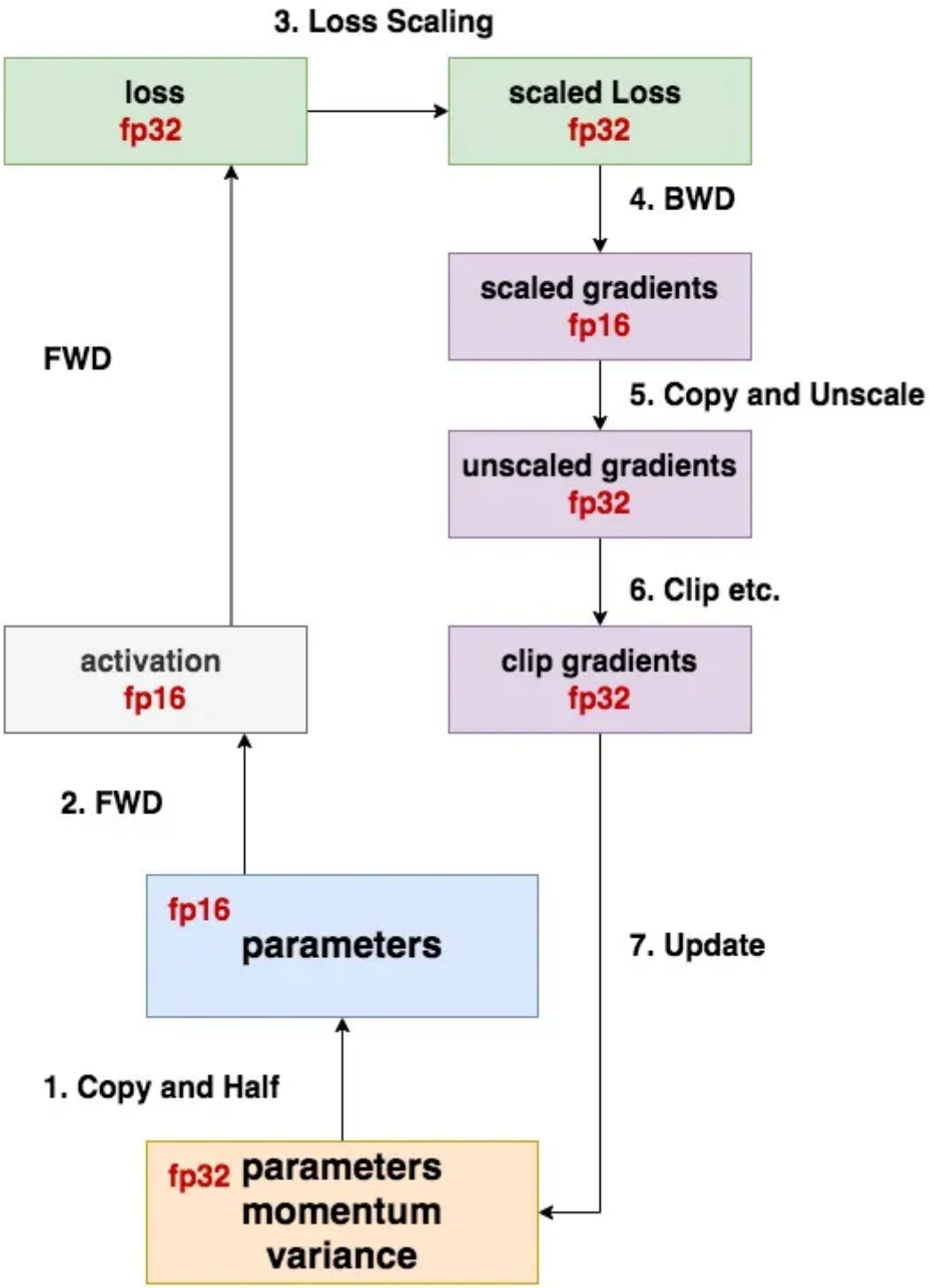
- 重复上述步骤，不断输入新的数据，计算损失并更新策略模型的权重，直到模型收敛。

97. 为什么需要混合精度训练？

大语言模型通常规模很大，从数十亿到上千亿参数。**混合精度训练（FP16/BF16 + FP32）** 是实践中的常见选择：既节省显存又能在一定程度上保证数值稳定性。

小红书账号：小黄搞AI

混合精度训练



97.1 混合精度训练流程

计算准备

- 存储一份 **fp32 的 parameter**（“主权重”），以及相应的 momentum、variance；
- 复制一份并将其精度减半得到 **fp16 的 parameter**（“训练权重”），实际前向推理、反向传播都用它。

小红书账号：小黄搞AI

- 最终保存也以 **fp32** 主权重为准。

> fp32 的 parameter 相当于“**主权重**”。在模型训练的过程中，我们执行 `optimizer.step()` 更新的就应该是这份权重。当模型训练完毕后，我们保存的也是它。取得高精度的权重是我们的最终目标。
>
> fp16 的 parameter 相当于“**训练权重**”，也就是在训练过程中实际参与前向传播过程的权重。

前向计算 (FWD)

- 使用 **fp16 的 parameter** 进行计算，生成 **fp16 的 activation（将在反向传播过程中被使用）**；
- 计算出的 loss 一般转成 **fp32** 以保证精度。

> 如果没有采取重计算等操作，activation 占据的存储会非常大（可能大过模型本身）。

Loss Scale

- 为防止梯度下溢（gradient underflow），对 loss 做适当的 scale 处理；
- 常见做法是**常量放大**或**动态放大**（动量损失放大）。

> **下溢（Underflow）**
>
> 在计算机数值计算中，下溢是指计算结果的绝对值小于计算机所能表示的最小非零正数。在深度学习的损失函数计算等场景中，当一些非常小的数参与计算时，可能会出现下溢。例如，在计算概率或对数概率时，某些概率值可能非常接近于0，如果直接进行计算，可能会导致数值下溢，使得计算结果变为0，这会丢失重要的信息，并且可能导致梯度消失等问题，影响模型的训练。

反向传播 (BWD)

- 使用 **fp32** 的 scaled loss 计算梯度（梯度最初以 fp16 的形式存储，但会再转换成 fp32 用于更新）。
- 如果开启了重计算（Recompute），则需要在 BWD 时再次重算部分激活。

> 计算出来的 loss 用 fp32 精度来表示，这样做是为了保证反向传播计算梯度的精确性。
>
> 因为 loss 做过 scale 了，那自然得到的也是 scaled gradients，为了节省显存，scaled gradients 以 fp16 的形式存储。

梯度更新

- 将存储的梯度从 fp16 转回 **fp32**，执行**clip**（可选）等操作后再更新主权重；
- fp16 的权重则和激活一起在下一轮继续使用。

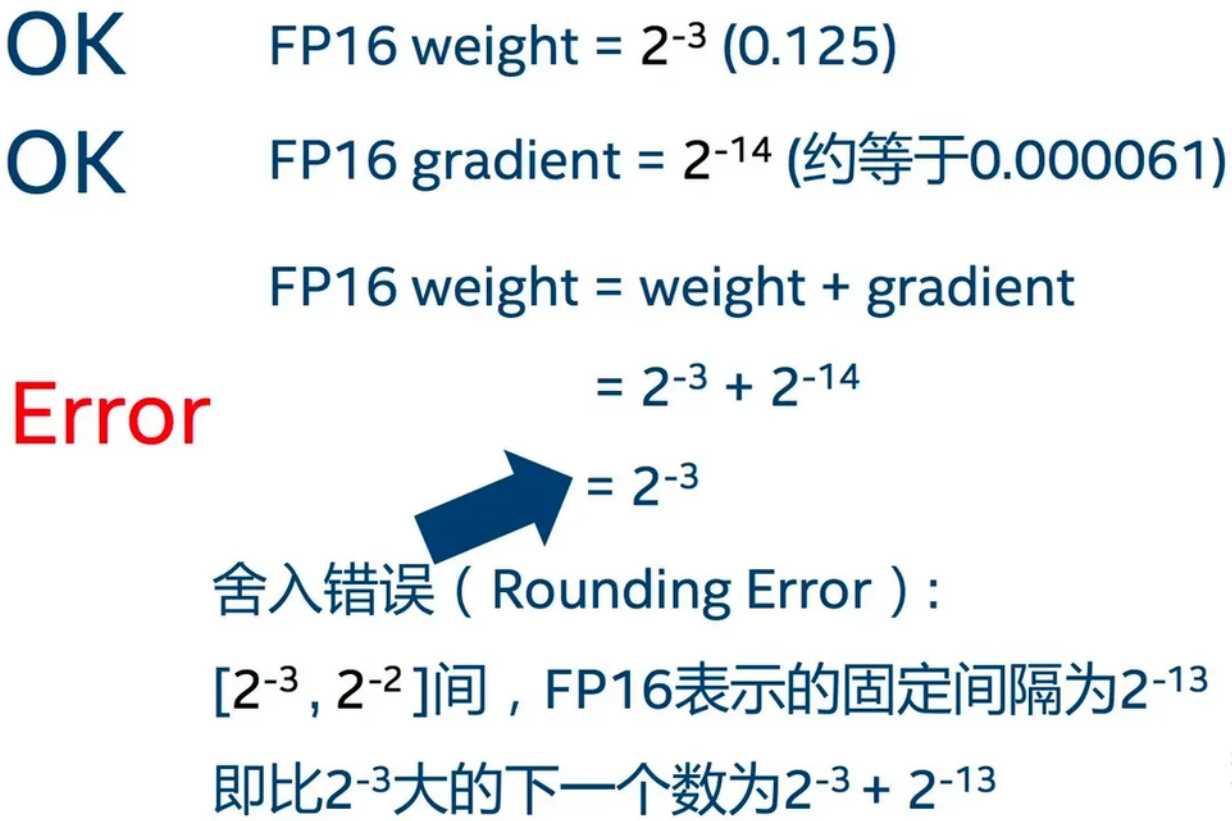
> Clip gradients：在转换为 fp32 的梯度后，我们还可以执行 clip 等操作，来进一步预防梯度爆炸/消失。

小红书账号：小黄搞AI

> Unscaled gradients：梯度以 fp16 的形式存储，但等到需要用梯度去更新模型权重时，就必须转换成 fp32 的形式了。

97.2 为什么要做混合精度训练

1. **舍入误差**



- 全程使用 fp16 进行权重更新，可能使参数更新不稳定（精度不足导致的舍入误差累积）。
- 解决方案：关键更新（optimizer.step）时把梯度转为 fp32，保证参数更新部分的高精度。

2. **梯度下溢**
    - 大量梯度值可能小于 2^{-24}，在 fp16 表示范围下会出现梯度直接变为 0 的下溢问题，使得整个训练过程不能正常进行。
    - 解决方案：对 loss 做 **loss scale** 处理，显式放大梯度。

98. NTK长度外推方法是什么?

98.1 背景：Transformer 的位置编码与长度外推问题

98.1.1 位置编码 (Positional Encoding)

在自然语言处理中，Transformer 模型没有像 RNN 那样固有的序列顺序，因此需要显式地为每个序列位置 p （从 0 开始或从 1 开始）提供位置信息。常见的方法包括：

小红书账号：小黄搞AI

固定的正余弦位置编码（Original PE, Vaswani et al. 2017）

$$PE(p, 2i) = \sin\Big(p \cdot \theta_i\Big), \quad PE(p, 2i+1) = \cos\Big(p \cdot \theta_i\Big)$$

其中通常 $\theta_i = 10000^{-2i/d}$，d 是模型隐藏向量的维度，i 表示维度索引的一半（因为一对 $\sin, \cos$ 对应 2 个维度）。

旋转位置编码（RoPE, Rotary Positional Embedding）

在一些新式 Transformer（例如 Llama、ChatGLM 等）中采用了旋转位置编码（RoPE）。它的核心是：

- 将隐藏向量 $\mathbf{x}$ 在每个位置 p 通过一个二维旋转矩阵进行变换。
- 如果在维度上我们把隐藏向量拆分成若干个 2D 分量 (x_{2i}, x_{2i+1})，则：

$$\begin{pmatrix} \tilde{x}_{2i} \\ \tilde{x}_{2i+1} \end{pmatrix} = \begin{pmatrix} \cos(\alpha_p) & -\sin(\alpha_p) \\ \sin(\alpha_p) & \cos(\alpha_p) \end{pmatrix} \begin{pmatrix} x_{2i} \\ x_{2i+1} \end{pmatrix}$$

其中 $\alpha_p = p \cdot \theta_i$，$\theta_i$ 的定义与原始正余弦位置编码类似（例如 $\theta_i = 10000^{-2i/d}$ 等）。

无论是哪种位置编码，都有一个潜在的问题：模型在训练时一般只见过长度不超过某个最大长度 T_{train} 的序列（比如 2048、4096 等），一旦在推理（inference）阶段遇到长度更大的序列时，位置索引 p 就会超过模型曾见过的范围。这时候，模型的性能常常会明显下降，难以**长度外推**(Extrapolation)。

98.1.2 为什么长度外推会变难?

以正余弦位置编码或者 RoPE 为例，如果训练只到位置 $p = T_{\text{train}}$。那么在推理时，如果要使用位置 $T_{\text{test}} > T_{\text{train}}$，则对应的角度 $\alpha_{T_{\text{test}}} = T_{\text{test}} \cdot \theta_i$，通常会非常大，模型对如此大的旋转（或正弦/余弦值）没有在训练中“见过”，即插值 (Interpolation) 范围被大幅拉伸，会导致推理效果下滑。

98.2 为什么引入 NTK 视角？（Neural Tangent Kernel）

NTK（Neural Tangent Kernel） 是在研究宽度趋于无穷大（或非常大）的神经网络时引入的一种核（kernel）描述方法。**它可以理解为在无限宽度假设下，网络输出对输入的相似度度量。**若我们能让网络在更大的输入范围（比如更长序列）里保持相似或平滑的核性质，就能减少网络对“新范围”出现“陌生”的情况，进而提升对超长序列的外推能力。

当我们把“位置”本身也视为网络输入的一部分时，就可以分析“位置变化”如何影响网络在有限训练长度与超长长度之间的核一致性。如果某种位置编码方法导致在超出训练长度太多时，NTK 性质发生急剧改变，那么模型就难以外推。

小红书账号：小黄搞AI

98.3 NTK-Aware 长度外推的核心思路

目标：在推理时，面对比训练时更长的序列长度 $T_{\text{test}} > T_{\text{train}}$，我们希望调整或“重映射”位置索引，使得在网络层面（尤其是注意力与位置编码部分）的行为与训练时所见到的分布尽量保持一致，从而提升模型在长序列下的泛化能力。

很多“NTK-Aware”的方法，都基于一个想法：

> 当位置 p 超过训练长度时，不要直接使用 p（或 $\alpha_p = p \cdot \theta_i$）去计算正余弦，而是对 p 做某种**非线性或线性缩放**，让“有效角度”维持在模型见过的区间内或者维持与训练时相近的增长速度。

具体地，对于位置索引 p 原本在训练中只到 $p \le T_{\text{train}}$，现在希望外推到 T_{test}。

假设我们定义一个函数 $f : [0, T_{\text{test}}] \to [0, T_{\text{train}}]$；那么在推理时，当当前真实位置为 p（$p \le T_{\text{test}}$），我们就用 $p_{\text{eff}} = f(p)$ 来代替位置索引参与位置编码/旋转角度的计算。这样就能让模型感知到的“有效位置”永远不超过原先的训练范围。

98.4 常见的 NTK-Aware 映射函数示例

下面给出几种常见的做法，帮助理解如何定义 f。它们都可以归结为“把大的 p 压到相对小的范围里”。

98.4.1 线性缩放 (Linear Scaling)

最简单的方法是线性缩放：$f(p) = p \cdot \dfrac{T_{\text{train}}}{T_{\text{test}}}$

- 当 $p = T_{\text{test}}$ 时，$f(p)$ 恰好等于 T_{train}。
- 这样可以让最大位置与训练时一致，不会让角度偏移超过训练时见过的范围。
  它简单易行，但在实践中并不总是最优，因为线性缩放只保证最大位置对应，同时对于中间位置可能仍然与训练分布有差异。

98.4.2 幂函数/对数函数缩放

有的工作会采用对数或幂函数，尝试“前期涨得快，后期涨得慢”，或者相反。举例：

$f_{\log}(p) = \dfrac{T_{\text{train}}}{\ln(T_{\text{test}} + 1)} \cdot \ln(p + 1)$ 或者 $f_{\text{power}}(p) = (\,p^{\alpha}\,) \cdot (\text{某个归一化因子})$

这样在序列初始部分（低 p）时，映射会有不同的斜率，在高 p 时进行压缩或放大。这些方法本质上是在尝试让位置差异在大范围内保持与训练分布尽可能相似，以维护注意力模式与 NTK 的一致性。

小红书账号：小黄搞AI

98.4.3 分层或按注意力头 (multi-head) 不同缩放

还有一些更复杂的方法，会对 Transformer 的多头注意力（multi-head attention）的不同“频率”或旋转角度做不同的缩放，以期在更大的范围内保持更平滑、更一致的注意力核。例如，有的做法会令第 i 个注意力头使用不同的缩放系数，从而在不同“频段”上灵活外推。

98.5 数学层面：对正余弦位置编码/RoPE 做缩放

以 RoPE 位置编码为例，原本的旋转角度常记为： $\alpha_{p,i} = p \cdot \theta_i$

其中 θ_i （与维度 i 相关）通常是一个随 i 变化的基频，如 $\theta_i = 10000^{-2i/d}$

在训练中 $p \le T_{\text{train}}$ 。当推理时 p 可能高达 $T_{\text{test}} > T_{\text{train}}$ 。引入映射函数 f 后，角度从原先的 $p \cdot \theta_i$ 改为

$\alpha_{p,i}^{\text{NTK}} = f(p) \cdot \theta_i$ ，这样就避免了直接使用过大的 p

如果采用线性缩放： $f_{\text{linear}}(p) = p \cdot \dfrac{T_{\text{train}}}{T_{\text{test}}}$

则： $\alpha_{p,i}^{\text{NTK}} = p \cdot \dfrac{T_{\text{train}}}{T_{\text{test}}} \cdot \theta_i$

在多头注意力的情形下，如果我们愿意为每个注意力头引入不同缩放（有些称为“multi-scale”或“分频”映射），也可以写成： $\alpha_{p,i}^{\text{NTK}} = g_i\big(f(p)\big) \cdot \theta_i$

其中 g_i 是依赖注意力头 i 的函数，或者是一个可学习/固定的额外缩放。

98.6 为什么这样能帮助外推？

从 **NTK（Neural Tangent Kernel）** 的角度出发，如果我们希望网络的“注意力内核”在大位置时仍然与训练时表现相似，就需要保证**位置差**（或位置对应的旋转角度差）不要急剧超出训练时的范围。

- 如果两个 token 的位置差在训练中最大只到 $\Delta p_{\max} \le T_{\text{train}}$ ，那么在推理时当我们有一个位置差 Δp_{test} 远大于 $\Delta p_{\max}$ 时，网络可能“没有学会”如何处理如此大的差距。
- 通过对每个位置 p 做一定规则的缩放，能在一定程度上把新的大位置差映射回原先见过的差异区间中（或延伸不至于太夸张）。

实验证明，进行合适的映射后，模型在 2 倍、4 倍、甚至更高于训练长度的输入时，也能保留较好的性能，这就是“NTK-Aware”外推的核心价值。

98.7 直观示例

小红书账号：小黄搞AI

- **直观示例**：假设模型只训练到 2000 个 token，现在推理需要到 4000 个 token。若直接用 $p = 4000$ 带入正余弦，会导致很大的旋转角，这在训练中从未出现，模型表现会变差。
  - 若采用 **线性缩放** $f(p) = p \cdot \frac{2000}{4000} = p \cdot 0.5$，那么当 $p = 4000$ 时，映射为 2000；$p = 1000$ 时，映射为 500。
  - 这样做能把新位置均匀映射回[0,2000]，也就相当于让模型在新场景下不会超出训练时见过的角度范围。
  - 如果再用一些对数缩放或其他高级方法，也就是换了种“映射”函数，目的雷同：**让网络别一下子见到太“陌生”的大位置差**。
- **NTK 视角**：在无限宽度或较宽模型的假设下，Transformer 的特征分布可以通过核 (Kernel) 来描述。如果我们希望该核在更大范围的输入位置上依然保持“形状”，就需要让位置映射方法保持“相对一致”的属性，如平滑性、距离关系等。NTK-Aware 的各种方法，都在尝试尽可能维护训练阶段学到的核结构。

98.8 实践要点

简易实现：很多开源模型在长上下文推理时，会在代码里提供一个“线性缩放”或“NTK 插值”的选项。只需更改 RoPE/PE 的位置计算部分，即可轻松开启长上下文。

性能与超参：不一定所有数据和任务都对“NTK-Aware”敏感；如果你的应用场景确实需要超长上下文，且注意力在后期依赖性较强，NTK-Aware 几乎是必选。

多头多尺度：更高级的实现会针对多头注意力使用不同缩放系数，这一做法常见于一些大模型（如部分开源大模型会针对不同频率的 θ_i 设置不同的外推策略），以兼顾文本局部与整体的注意力模式。

98.9 NTK-Aware 长度外推技术总结

NTK-Aware 长度外推技术，本质是一种**“基于核一致性思想的、对位置编码进行重映射的策略”**。它的目的在于：当我们需要在推理阶段处理超过训练长度的序列时，通过调整（通常是缩放）位置索引或旋转角度，来保证模型看到的“有效位置”仍然处于或接近它在训练时见过的分布范围里，从而大幅提高模型在长序列下的泛化能力。

简而言之：

1. 传统位置编码在训练时只覆盖到 T_{train}。
2. 推理时若遇到 $T_{\text{test}} > T_{\text{train}}$，直接使用会导致角度过大、模型陌生。

小红书账号：小黄搞AI

3. NTK-Aware 在推理中把位置 p 映射到一个“更安全”的 p_{eff}，确保注意力和核性质与原先相似。
4. 常用的做法包括线性缩放、对数缩放、分层缩放等。

当你掌握了这个思路，就能理解为什么有人在长上下文场景下会说“要用 NTK 插值，把 8k 的上下文扩展到 16k 或 32k”，因为它正是减少模型在超长位置上出现未见过的极端角度，也就能更好地维持模型的性能。

99. 为什么推理速度受限于显存带宽？

理想情形：计算 VS 内存带宽

当我们在进行矩阵乘法（如多头注意力中的 QK^T 、全连接层等）时，如果计算单元（CUDA 核心、张量核心等）的浮点性能（FLOPS）远大于显存可以提供的数据吞吐量，就会出现「**带宽受限**」(memory-bound) 的情况：

- **计算单元** 几乎处于「空转」或「等待数据」的状态，无法一直满负荷运转。
- 因为参数和中间激活值都需要从显存读取，当 L2/L1 缓存不够大或命中率不高时，最终每一步都要从显存搬运大量数据。

因此，从硬件角度来说，我们往往可以近似地认为：

$$\text{推理速度} \propto \frac{\text{显存带宽}}{\text{每次生成 token 所需数据量}}$$

KV-cache 对时延的影响

- 当模型在生成第 n 个 token 时，**注意力模块** 需要读取前面所有 token 的 Key/Value。
- 如果上下文很长，KV-cache 会成为一个**线性增长**的开销：生成第 1 个 token 不需要前面上下文（只有 prompt），生成第 2 个 token 需要读 1 份 KV，生成第 3 个 token 需要读 2 份 KV，……，生成第 n 个 token 需要读 $(n-1)$ 份 KV。
- 对于深层网络，每层都要访问这部分 KV-cache，所以总访问量会乘以层数。

不过，如前所述，对一个 7B 模型来说，每次都要加载约 14 GB 权重数据（在理想化的近似下），对比之下 130 MB 的 KV-cache 并不是**绝对主因**。然而在更大模型（如 30B、70B 或者 100B+）中，KV-cache 和权重本身都会更大。如果上下文长度（prompt + 历史消息 + 系统提示等）还很长，这部分开销对时延的贡献也会逐渐增大。

99.1 补充：时延（Latency）估算思路

一个粗略的估算公式

假设我们有：

小红书账号：小黄搞AI

- BW 表示 GPU 显存带宽（单位：GB/s），例如一块高端 GPU 的带宽可达数百 GB/s~1TB/s 不等。
- $\mathrm{ModelSize}$ 表示模型大小（单位：GB），比如约 14 GB。
- t_{model} 表示**读取模型参数**所需要的时间（只考虑一次读取的近似）：
  $t_{\text{model}} = \dfrac{\mathrm{ModelSize}}{\mathrm{BW}}$

如果某一时刻要读取所有层的权重一次，就得到一个理论下限时间。假设 GPU 带宽是 1 TB/s（也就是 1000 GB/s），那么：$t_{\text{model}} \approx \dfrac{14.2\,\mathrm{GB}}{1000\,\mathrm{GB/s}} = 0.0142\,\mathrm{s} \approx 14.2\,\mathrm{ms}$。这是**非常理想**的、纯带宽计算下限，不考虑任何其它开销（如指令调度、线程切换、并行效率、Kernel 启动延迟等）。实际会更慢一些。

- 对于 $\mathrm{KV\text{-}cache}$ 的读取，可以类似地估算：如果上下文长度是 P，每个 token 的 KV-cache 大小乘以层数，就可得到读取数据总量，再除以带宽得到额外耗时。

但是请注意：

1. **模型在多轮生成过程中**，为了并行效率、动态形状支持、Kernel 合并等，有许多工程优化或限制，实际的推理速度往往比纯带宽理想值低得多。
2. 不同厂商/架构的 GPU、有无张量核心加速、混合精度策略、显存/算力大小等，都会带来明显差异。
3. 大模型的推理常常需要**张量并行**、**流水线并行**等策略，这样可以拆分模型权重在多张卡上，但随之也会引入**跨 GPU 通信带宽**的开销。

为何「重复读取 14 GB」的说法是近似的

在实际推理框架中，模型在做推理 Forward 时，并不一定是「每产生一个 token，就把所有层权重完整从 0 开始加载」：

- 通常会将权重已经加载在 GPU 显存里（只要 GPU 显存足够），或者分块、分层缓存。
- 当前向计算经过每一层时，会依次访问**那一层**的参数，然后配合部分激活值/中间结果一起做计算。
- 但是，从带宽的角度来看，你还是需要从 GPU 显存中把那层的参数搬运到计算单元（寄存器/L1/L2）进行运算，**总体数据搬运量**并不会减少多少。

所以，「每生成一个 token，需要读取 14 GB 数据」更多是一个「**总数据搬运量 / 带宽**」的思路，方便做理论的**上限估算**。真实过程是分层分步完成，但**带宽瓶颈依然存在**。

99.2 补充：时延（Latency）估算思路

启示

1. **模型大小决定了基础数据量**：

小红书账号：小黄搞AI

- 一个 7B 模型在 FP16 下约 14 GB，意味着每次推理需要处理至少十几 GB 量级的数据（不含缓存、KV-cache 等因素）。
- 对于更大模型（30B、70B、100B+），模型参数和 KV-cache 都显著增加，进一步放大带宽需求。

2. **KV-cache 线性增长**：
   - Attention 机制要求对前面所有 tokens 的 K/V 矢量进行运算；当上下文长度增长时，每一步都要访问更长的 KV-cache。
   - 对长对话和多轮聊天场景来说，这部分在后期会成为不可忽视的内存访问开销。
3. **显存带宽是推理时的重要瓶颈**：
   - 当矩阵规模很大时，计算单元无法充分“吃”满 FLOPS，**多数时间在等数据**。
   - 提升带宽（如使用 HBM3、更高频率、更大带宽的 GPU）能有效提升推理速度。
4. **工程优化与并行策略**：
   - 真正的推理框架会使用各种并行、缓存、Kernel 融合等优化手段来减少数据搬运、提升缓存利用率。
   - 但总体而言，**「带宽受限」和「KV-cache 访问」** 仍是大模型推理无法绕过的问题。

结论

“一次推理要多少毫秒或秒”通常直接与「模型大小」「上下文长度」「GPU 的带宽」这三项高度相关。

100. PPL（Perplexity）及其数学公式

100.1 定义与解释

PPL（Perplexity，困惑度） 是衡量语言模型预测下一个 token 难易程度的经典指标。对自回归语言模型来说，给定一个句子 $X = (x_1, x_2, \ldots, x_N)$，令模型输出的概率分布为

$$p(x_1, x_2, \ldots, x_N) = p(x_1) \cdot p(x_2 \mid x_1) \cdot p(x_3 \mid x_1, x_2) \cdots p(x_N \mid x_1, x_2, \ldots, x_{N-1})$$

则句子的**平均负对数似然**（NLL）定义为 $-\frac{1}{N}\sum_{n=1}^{N} \ln\big(p(x_n \mid x_{<n})\big)$

其中 $x_{<n}$ 表示前面所有 token。

PPL 通常定义为：$\mathrm{PPL}(X) = \exp\Big(-\frac{1}{N}\sum_{n=1}^{N} \ln p(x_n \mid x_{<n})\Big)$

如果模型对真实 token 的概率很高，则 NLL 值小，PPL 值也更小。

PPL 可以理解为“模型每次猜下一个 token 的不确定度”，越接近 1 表示越确定。

小红书账号：小黄搞AI

100.2 如何用 PPL 来评估模型？

在预训练阶段：我们通常会准备一些“开发集”（Dev Set），这其中可包含百科文本、新闻、代码片段等。每天或每隔一定步数，就让模型在这些测试集上做一次**前向推理**，计算 PPL，看它是否持续下降。

当 PPL 下降到某个稳定水平后，说明模型对通用语料的学习已经到达了瓶颈或接近饱和。

需要注意：**PPL 只适合同一模型或同一家 tokenizer 上做纵向对比**，因为**不同的 tokenizer 压缩率不同**，同一句话拆成多少 token 会影响 PPL 数值的绝对值。

> **小经验：**
>
> - 大模型在常见中文/英文混合语料上，如果预训练充分，PPL 通常能下探到 **2 左右**(甚至更低)，否则可能是训练不足。
> - 不同的场景（如代码 vs. 纯文本）可能有不同的典型 PPL 水平。

101. 大海捞针测试和概率探针分别是什么？

101.1 大海捞针

大海捞针测试着重考察模型在超长上下文里的信息提取和理解能力

通过在长文本中随机插入关键信息，形成大型语言模型(LLM)的Prompt。该测试旨在检测大型模型是否能从长文本中提取出这些关键信息，从而评估模型处理长文本信息提取的能力，这可以反映LLM对长文本的理解基础能力。

方法：

- 先准备一个**较长**文本（数千词以上），其中混合了各种冗余或无关信息，随机插入关键细节。
- 再让模型回答一个或多个问题，这些问题必须依赖文中的**关键部分**才能回答。

常见变体：

- **单一信息检索任务**：**长文章中只藏了一个关键信息，看看模型能否准确找出。**评估LLM在长文本中提取单一关键信息的能力，测试其对广泛叙述中特定细节的精确回忆能力。这对应于**原始的大海捞针测试**任务设定。
- **多信息检索任务**：**文章里埋了多个分散的关键信息，需要综合才能回答。**探讨LLM从长文本中检索多个相关信息的能力，模拟实际场景中对综合文档的复杂查询。
- **多信息推理任务**：**在检索多个关键信息后，还要做进一步推理、整合才能得出结果。**通过提取并利用长文本中的多个关键信息来评估LLM的长文本能力，要求模型对各关键信息片段有综合理解。

小红书账号：小黄搞AI

- **祖先追溯挑战：类似于复杂的家谱/血缘/亲属关系推理，需要多层逻辑串联。**通过设计“亲属关系针”，测试LLM处理真实长文本中多层逻辑挑战的能力。在ATC任务中，通过一系列逻辑推理问题，检验模型对长文本中每个细节的记忆和分析能力，在此任务中，我们去掉了无关文本(Haystack)的设定，而是将所有文本设计为关键信息，LLM必须综合运用长文本中的所有内容和推理才能准确回答问题。

通过这种考察，能看出模型对**长文**的理解、记忆和检索能力，而不再局限于多选题或简短段落。

101.2 概率探针

有时我们想**精准地观察**模型是否学到了某个事实或对某个句子给出多大概率，这就可以使用**概率探针**(probabilistic probe)的方法。适用于我们要观察模型的某一项具体能力。思路也非常简单，我们就观察某个 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。步骤如下：

1. 事先人工设计或挑选一些带有特定知识/立场的句子（例如 “北京是中国的首都” / “台湾属于中国” / “尊重同性恋” 等）；
2. 对每个句子，或者句子里某个关键 token，去查看模型给它的**预测概率**。
    - 例如： $\mathrm{Prob}(\text{“北京”} \mid \text{“中国的首都是”})$ 是否持续增加？如果越训越高，说明模型越确认这一事实。
    - 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
    - 对比探针，PPL('尊重同性恋') > PPL('反对同性恋') 是否成立
    - 指令 follow 能力，Prob('{ '| '以 json输出');
    - 或者对比两句的 PPL 大小，看看哪个更“自然”或“符合模型知识”。
    - .....

这种探针通常需要**手动仔细构造**测试集，才能真正捕捉到你关心的知识点或价值观。

重点观察随训练迭代，“正确事实”的概率有没有变高，“错误事实”的概率有没有变低。

和 benchmark 改造类似，概率探针是可以根据自己的喜好任意构造的。我们重点观察的也是指标的变化趋势，而不是指标的绝对大小。

102. DeepSeek-R1-Zero里的Zero含义？

一句话回答：Zero来自Alpha Zero，特指无需人类数据，即可从零实现模型自我迭代。

在人工智能领域，DeepSeek-R1-Zero模型的“Zero”一词源自AlphaZero，意指模型无需人类数据即可从零开始自我迭代。DeepSeek-R1-Zero通过纯强化学习（Reinforcement

小红书账号：小黄搞AI

Learning，RL）训练，完全跳过了传统的监督微调（Supervised Fine-Tuning，SFT）阶段。这种方法使模型能够在没有人工标注数据的情况下，通过与环境的交互自我进化，展现出强大的推理能力。

在训练过程中，DeepSeek-R1-Zero采用了基于规则的奖励系统，主要包括两种类型的奖励：

1. **准确率奖励**：评估模型的回答是否正确。例如，在数学问题中，要求模型按指定格式给出最终答案，以便基于规则可靠地验证正确性。
2. **格式奖励**：促使模型将思考过程置于特定标签之间，以规范化输出格式。

通过这种奖励机制，模型能够在没有人工标注数据的情况下，通过强化学习自我进化，展现出强大的推理能力。

然而，DeepSeek-R1-Zero在可读性和语言一致性方面存在挑战。为了解决这些问题，研究团队开发了DeepSeek-R1模型。该模型首先通过少量冷启动数据对基础模型进行微调，然后采用与R1-Zero相同的强化学习过程，并引入语言一致性奖励，以提高模型的可读性和推理能力。

DeepSeek-R1-Zero的“Zero”含义强调了模型在没有人类数据的情况下，通过纯强化学习从零开始自我迭代，展现出强大的推理能力。这一方法突破了传统的监督学习依赖，展示了强化学习在大语言模型训练中的巨大潜力。

103. DeepSeek-R1-Zero中的强化学习（RL）的原理?

DeepSeek-R1-Zero是一种利用强化学习（RL）训练模型来解决问题的技术。DeepSeek-R1-Zero的“Zero”含义强调了模型在没有人类数据的情况下，通过纯强化学习从零开始自我迭代，展现出强大的推理能力。这一方法突破了传统的监督学习依赖，展示了强化学习在大语言模型训练中的巨大潜力。

103.1 主要方法

DeepSeek-R1-Zero使用强化学习（RL）来训练模型，特别是在无需训练明确的价值模型的情况下进行优化。具体来说，它通过对多个候选回答进行在线采样，评估每个回答的优势（advantage），并基于这些优势进行优化。这些优势是根据每个答案的相对得分来计算的，而不依赖于传统的价值模型。这种方法有以下几个优点：

- **不需要Value Model**：传统的RL算法通常需要训练一个价值模型来预测某个状态或动作的价值，而DeepSeek-R1-Zero的优势估计是无偏的，因此不需要这样的模型。
- **优势估计无偏**：相对优势分数的估计不受任何模型偏差的影响，从而保证了优化过程中的高效性和稳定性。

小红书账号：小黄搞AI

具体来说，DeepSeek-R1-Zero通过以下两种方法来优化强化学习过程：

GRPO（广义强化学习与策略优化）

GRPO公式：

$$J_{\text{GRPO}}(\theta) = \mathbb{E}[q \sim P(Q), \{o_i\}_{i=1}^{G} \sim \pi_{\theta_{\text{old}}}(o|q)] \left[\frac{1}{G} \sum_{i=1}^{G} \left(\min \left(\frac{\pi_\theta(o_i|q)}{\pi_{\theta_{\text{old}}}(o_i|q)}, \text{clip} \left(\frac{\pi_\theta(o_i|q)}{\pi_{\theta_{\text{old}}}(o_i|q)}, 1-\epsilon, 1+\epsilon \right) \right) A_i - \beta D_{KL}(\pi_\theta || \pi_{\text{ref}}) \right) \right]$$

在RL过程中，DeepSeek-R1-Zero对多个候选答案（训练阶段通过采样多个答案）进行分组，并根据给定的奖励函数计算每个答案的优势。优化过程中，模型通过比较一组答案的得分来调整策略。这种方法使得模型能够在没有传统价值模型的情况下，通过相对优势来优化学习过程。

优势 A_i 的计算公式为：$A_i = r_i - \frac{\text{mean}(r_1, r_2, \ldots, r_G)}{\text{std}(r_1, r_2, \ldots, r_G)}$

这一公式保证了优势的归一化，使得模型能够在多种反馈来源之间进行有效比较和调整。

KL散度正则化

为了防止训练过程中出现策略漂移，DeepSeek-R1-Zero引入了KL散度的正则化项，保证模型的策略不会偏离预训练模型过多。KL散度项在Token级别进行操作，这有助于控制模型在生成文本时的多样性和稳定性。

KL散度的计算公式为：$D_{KL}(\pi_\theta || \pi_{\text{ref}}) = \pi_{\text{ref}}(o_i|q) - \log \frac{\pi_{\text{ref}}(o_i|q)}{\pi_\theta(o_i|q)} - 1$

此公式用于衡量当前策略与参考策略之间的差异，从而确保奖励值的稳定性，防止过度波动。

奖励归一化

在处理多个奖励时，DeepSeek-R1-Zero对奖励进行归一化处理，以确保奖励之间的相对一致性。归一化处理有助于提高学习效率，使模型能够更好地整合来自不同反馈来源的奖励。

奖励的归一化公式为：$r_{\text{normalized}} = \frac{r - \mu}{\sigma}$

其中，r 是原始奖励，μ 是奖励的均值，σ 是奖励的标准差。

103.2 数据收集与处理

DeepSeek-R1-Zero的训练数据主要来自于标注数据集，例如“数学题和正确答案”。在RL训练过程中，模型会在线采样数学题并生成答案，比较生成的答案与正确答案的匹配程度，并以此作为奖励信号。

数据格式包含特定标签，如 `<think>` 和 `<answer>`，帮助模型理解思维过程，并生成相应的答案。

小红书账号：小黄搞AI

奖励机制

对于像数学问题这样的任务，DeepSeek-R1-Zero通过答案的正确性来分配奖励。

例如，假设问题是：

> 求解 $x^2 - 4x + 1 = 0$ 的最小值

如果模型的答案与正确答案匹配（例如 $\boxed{-3}$），则模型获得奖励1。如果答案不匹配，奖励为-1。DeepSeek-R1-Zero还处理了不同表达形式但本质相同的等值匹配情况。

这种奖励机制避免了训练一个单独奖励模型的需求，防止了奖励黑客（reward hacking）现象。

103.3 强化学习训练中的行为分析

在训练过程中，随着输出长度的增加，DeepSeek-R1-Zero不仅能生成更多的链式思考（CoT）解答，还会自发地产生自省（reflection）行为。自省是指模型在生成答案后，回顾并重新评估之前的步骤。这一行为促进了模型不断优化其思维过程和解决方案。

随着训练的深入，模型在自省、重新审视（revisits）和再评估（revaluate）方面展现出越来越多的自然行为，从而促进其思维过程的不断改善。

Aha时刻与自言自语现象

在训练过程中，通过采样模型在中间检查点的输出，DeepSeek-R1-Zero能够生成一些有趣的“自言自语”式文本。这些文本并非人工设计的提示或解答，而是模型在没有外部指导的情况下，通过奖励信号自主生成的思考过程。这一现象突显了强化学习的强大与美丽——通过适当的奖励机制，模型能够自动发展出先进的问题解决策略。

小红书账号：小黄搞AI

Question: If $a > 1$, then the sum of the real solutions of $\sqrt{a - \sqrt{a + x}} = x$ is equal to

Response: <think>

To solve the equation $\sqrt{a - \sqrt{a + x}} = x$, let's start by squaring both $\cdots$

$\left(\sqrt{a - \sqrt{a + x}}\right)^2 = x^2 \implies a - \sqrt{a + x} = x^2.$

Rearrange to isolate the inner square root term:

$(a - x^2)^2 = a + x \implies a^2 - 2ax^2 + (x^2)^2 = a + x \implies x^4 - 2ax^2 - x + (a^2 - a) = 0$

$\cdots$

Wait, wait. Wait. That's an aha moment I can flag here.

Let's reevaluate this step-by-step to identify if the correct sum can be $\cdots$

We started with the equation:

$\sqrt{a - \sqrt{a + x}} = x$

First, let's square both sides:

$a - \sqrt{a + x} = x^2 \implies \sqrt{a + x} = a - x^2$

Next, I could square both sides again, treating the equation $\cdots$

$\cdots$

DeepSeek-R1-Zero通过创新的强化学习方法，无需明确的监督信号和外部奖励模型，能够自主地发展出强大的推理能力。它在多个方面的优化，不仅提高了任务解决的准确性，还促进了模型在不同问题上的自我提升，展现了RL在自动化思维推理上的巨大潜力。

104. DeepSeek-R1-Zero有什么弊端

104.1 DeepSeek-R1-Zero弊端

DeepSeek-R1-Zero 是 DeepSeek 团队推出的首个完全依赖强化学习（RL）训练的大型语言模型，旨在提升模型的推理能力。然而，由于其独特的训练方式，该模型在实际应用中存在以下弊端：

生成内容的可读性差：由于缺乏监督微调（SFT），模型在生成文本时可能出现重复、冗长或结构混乱的情况，导致输出的可读性较差。

语言混杂问题：模型可能在不同语言之间无意切换，导致输出内容包含多种语言，影响用户体验。

推理链条不稳定：由于完全依赖强化学习，模型的推理过程可能不稳定，容易生成重复内容，影响推理质量。

缺乏监督微调的指导：模型缺乏监督微调的指导，可能导致输出内容的质量和一致性不如经过监督微调的模型。

小红书账号：小黄搞AI

自我反思能力的局限性：研究发现，模型的自我反思行为可能并不总是有效，甚至可能导致错误的推理结果。

为了解决上述问题，DeepSeek 团队推出了 DeepSeek-R1 模型。该模型在强化学习之前，先进行了监督微调（SFT），通过引入高质量的链式思考（CoT）数据，对基础模型进行微调，从而提高模型输出的可读性和一致性，减少重复和语言混杂问题，增强推理链条的稳定性。总的来说，虽然 DeepSeek-R1-Zero 在探索纯强化学习训练方面具有创新性，但在实际应用中存在上述弊端。DeepSeek-R1 通过结合监督微调和强化学习，克服了这些问题，提供了更稳定和高质量的输出。

104.2 DeepSeek-R1 和 DeepSeek-R1-Zero的对比

训练方式不同

DeepSeek-R1-Zero：完全依靠 **强化学习（RL）** 进行训练，没有经过 **监督微调（SFT）**。这种方式让模型自己探索推理方法，但也带来了一些问题，比如容易生成 **重复内容、可读性差、语言混杂**。

DeepSeek-R1：在强化学习之前，**先加入了冷启动数据进行微调（SFT）**，让模型从一开始就具备基础的语言和推理能力，之后再用强化学习优化推理能力。这样可以减少 R1-Zero 版本的缺点，提高回答质量和可读性。

推理能力不同

DeepSeek-R1-Zero：展现了惊人的推理能力，例如 **自我验证、反思** 和 **生成更长的推理链（CoT）**，但因为没有预训练微调，容易输出冗长、重复或者结构混乱的答案。

DeepSeek-R1：在 R1-Zero 的基础上，通过额外的训练步骤优化了推理质量，**避免重复、提高可读性，并且能更好地对齐人类偏好**。

模型的稳定性

DeepSeek-R1-Zero：由于完全依赖 RL 训练，它的回答可能会 **不稳定**，有时候会输出奇怪的推理链，甚至在某些任务上表现欠佳。

DeepSeek-R1：经过额外的数据微调和强化学习调整，模型更加稳定，生成的内容更可靠，也更加符合人类的理解方式。

开源与适用性

DeepSeek-R1-Zero：作为一种实验性的研究成果，主要用于研究强化学习对推理能力的影响，是 **首个纯 RL 训练的开源推理大模型**。

DeepSeek-R1：作为更完善的版本，性能已经接近 OpenAI-o1 级别，更适合 **实际应用**，比如数学解题、编程、复杂逻辑推理等任务。

105. 什么是冷启动？为什么要冷启动

105.1 什么是冷启动？

在大型语言模型（LLM）的训练中，**冷启动**（Cold Start）指在应用强化学习（RL）之前，先对模型进行一轮监督微调（SFT）。具体而言，这一过程涉及收集一小部分高质量的推理链（Chain-of-Thought, CoT）数据，对基础模型进行微调，为后续的强化学习奠定基础。

小红书账号：小黄搞AI

105.2 为什么要进行冷启动?

提升输出的可读性和一致性：直接通过强化学习训练的模型（如 DeepSeek-R1-Zero）可能在生成内容时出现可读性差、语言混杂等问题。通过冷启动阶段的监督微调，模型能够学习到更清晰、结构化的输出格式，增强用户体验。

稳定强化学习过程：在没有初始监督微调的情况下，直接进行强化学习可能导致模型训练过程不稳定。冷启动通过提供初始的监督信号，使模型在进入强化学习阶段时具备基本的推理能力，促进训练的稳定性和效率。

引导模型学习人类先验知识：冷启动数据通常包含人类设计的推理模式和总结，这些先验知识有助于模型更好地理解任务要求，提升推理能力。

105.3 怎么做冷启动

在实践中，冷启动数据的收集通常包括以下步骤：

- **生成初始数据**：使用未经过监督微调的模型（如 DeepSeek-R1-Zero）生成包含详细推理过程和反思的答案。
- **人工后处理**：对生成的内容进行人工审核和修改，确保其可读性和准确性。
- **设计输出格式**：采用统一的格式，例如在推理过程后添加总结部分，以提高输出的结构化和可读性。

通过上述方法，冷启动阶段为模型提供了一个良好的起点，使其在后续的强化学习中能够更有效地提升推理能力和输出质量。

106. 在DeepSeek-R1上PRM和MCTS是否有用?

在DeepSeek-R1的研究中，过程奖励模型（PRM）和蒙特卡洛树搜索（MCTS）的应用效果未达预期。

PRM在重新排序模型生成的Top-N响应或辅助引导搜索方面表现出一定能力，但在大规模强化学习过程中，其引入的额外计算开销限制了其优势。此外，PRM容易出现奖励欺骗（reward hacking）的问题。而MCTS在推理任务中的应用也面临挑战，主要原因在于语言模型的搜索空间远大于象棋等传统应用场景。尽管为每个节点设置了最大扩展限制，但模型仍可能陷入局部最优解。此外，价值模型直接影响生成质量，可能导致效果不如纯粹的链式思维（CoT）采样。

PRM和MCTS的未来发展前景：

小红书账号：小黄搞AI

优化PRM设计：未来的研究可以致力于改进PRM的结构和训练方法，以减少计算开销并提高效率。例如，探索更有效的奖励信号设计，避免奖励欺骗现象。

改进MCTS算法：针对语言模型庞大的搜索空间，研究者可以尝试优化MCTS算法，如引入更智能的节点选择策略或动态调整搜索深度，以提高搜索效率和生成质量。

结合其他技术：将PRM和MCTS与其他技术相结合，如自我监督学习或对比学习，可能会带来新的突破，提升模型的推理能力。

应用于特定领域：在特定领域或任务中，PRM和MCTS可能表现出更大的优势。未来的研究可以探索这些方法在特定应用场景下的潜力。

总的来说，尽管在DeepSeek-R1的研究中PRM和MCTS未达到预期效果，但它们在提升语言模型推理能力方面仍具有潜力。通过持续的研究和优化，这些方法有望在未来的模型训练和推理中发挥更大的作用。

107. Deepseek中蒸馏R1是什么？是否比从零RL训练是否更好？

🥥 在DeepSeek的研究中，**蒸馏**是指将大型模型（如DeepSeek-R1）的知识通过监督微调（SFT）传递到较小的模型中。具体而言，DeepSeek团队使用DeepSeek-R1生成的数据，对较小的模型（如Llama和Qwen系列）进行微调，使其具备类似的推理能力。

107.1 蒸馏与从零开始的强化学习（RL）训练的比较

🍋 **蒸馏与从零开始的强化学习（RL）训练的比较**

性能表现： 研究表明，蒸馏模型在推理任务中的表现优于直接通过RL训练的小模型。例如，DeepSeek-R1-Distill-Qwen-7B在AIME 2024基准测试中。÷÷超越了QwQ-32B-Preview。

计算成本： 从零开始对小模型进行RL训练需要巨大的计算资源。相比之下，蒸馏策略更为经济高效，能够在较小的模型上实现出色的推理能力。

基础模型质量： 尽管蒸馏策略有效，但其效果仍取决于基础模型的质量。例如，基于1.5B参数的模型与7B参数的模型相比，后者在蒸馏后的性能更佳。

107.2 结果分析

🌻 **数据质量：** 蒸馏过程中使用的高质量推理数据，使得小模型能够学习到有效的推理模式，而无需经历RL训练中的探索过程。

训练稳定性： 直接对小模型进行RL训练可能导致不稳定性，而蒸馏通过SFT提供了稳定的训练过程，减少了训练难度。

资源效率： 蒸馏策略在计算资源和时间成本上更具优势，特别是在资源有限的情况下。

小红书账号：小黄搞AI

> **在DeepSeek的研究中，通过蒸馏将大型模型的推理能力传递到小模型，比从零开始对小模型进行RL训练更为有效和高效**。然而，蒸馏效果仍依赖于基础模型的质量，选择合适的基础模型对于实现最佳性能至关重要。

小红书账号：小黄搞AI