【面试真题】大模型算法面试1000题

1. 深度学习基础知识

1.1 降维

✏️ **问题：在机器学习算法中，如何处理高维数据的特征选择和降维问题？**

来源：字节跳动、滴滴

答案：

一、特征选择（Feature Selection）

特征选择是指从原始特征集中选择出最相关的特征子集，去除冗余或不相关的特征。

1. 过滤式方法（Filter Methods）

过滤式方法独立于任何特定的学习算法，根据特征的统计属性（如方差、相关性、信息增益等）对特征进行评估和选择。

- **方差选择（Variance Thresholding）：** 移除方差低于某个阈值的特征。假设特征 Xi 的方差为 $\mathrm{Var}(X_i)$，阈值为 T，则保留满足 $\mathrm{Var}(X_i) \geq T$ 的特征。
- **相关系数（Correlation Coefficient）：** 衡量特征与目标变量之间的线性相关程度。常用的有皮尔逊相关系数（Pearson Correlation Coefficient）：

$$r = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \bar{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \bar{y})^2}}$$

- **卡方检验（Chi-Squared Test）：** 用于检验分类变量之间的独立性，可以评估特征与目标变量之间的相关性（仅适用于分类特征和分类目标变量）。假设观察频数为 O_i，期望频数为 E_i，k 是类别数，卡方统计量为：

$$\chi^2 = \sum_{i=1}^{k}\frac{(O_i - E_i)^2}{E_i}$$

- **信息增益（Information Gain）：** 在决策树算法中常用，衡量一个特征能够减少数据集不确定性的程度。对于特征 A 和数据集 S，信息增益 $IG(S, A)$ 定义为：

$$IG(S, A) = H(S) - \sum_{v \in \mathrm{Values}(A)}\frac{|S_v|}{|S|}H(S_v)$$

其中, $H(S)$ 是数据集 S 的熵， $Value(A)$ 是特征 A 的所有可能取值， S_v 是特征 A 取值为 v 的子集。

2. 包裹式方法（Wrapper Methods）

包裹式方法通过将特征子集的选择看作一个搜索问题，并使用特定的机器学习模型在不同的特征子集上进行训练和评估，选择性能最好的子集。

- **前向选择（Forward Selection）：** 从一个空特征集开始，每次添加一个能使模型性能提升最大的特征，直到达到某个停止准则。
- **后向消除（Backward Elimination）：** 从所有特征开始，每次移除一个能使模型性能下降最小的特征，直到达到某个停止准则。
- **递归特征消除（Recursive Feature Elimination, RFE）：** 迭代地训练模型并移除最不重要的特征，直到达到期望的特征数量。

3. 嵌入式方法（Embedded Methods）

嵌入式方法将特征选择融入到模型的训练过程中，通过模型自身的特性来选择重要的特征。

- **L1 正则化（Lasso）：** 在线性模型的损失函数中添加 L1 范数惩罚项，促使模型参数稀疏化，从而实现特征选择。对于线性回归模型，损失函数为：

$$J(w) = \frac{1}{2n}\sum_{i=1}^{n}(y_i - w^T x_i)^2 + \lambda\|w\|_1$$

其中，w 是权重向量，x_i 是特征向量，y_i 是目标变量，λ 是正则化参数，$\sum_{j=1}^{p}\|w_1\|$ 是 L1 范数。

- **树模型（Tree-based Models）：** 如决策树、随机森林、梯度提升树等，这些模型在训练过程中会评估特征的重要性，可以直接利用特征的重要性得分进行特征选择。

总结特征选择方法：

方法	关键技术	优点	缺点	计算成本	示例用例
过滤式方法	方差选择，相关系数，卡方检验，互信息	计算效率高，适用于大规模数据	忽略特征交互，不针对特定模型优化	低	初步特征筛选
包裹式方法	递归特征消除（RFE），前向/后向选择，穷举搜索	性能通常更好，能找到针对特定模型优化的特征	计算成本高，易过拟合	高	需要最优性能的场景
嵌入式方法	L1正则化（Lasso），树模型特征重要性	效率和性能之间取得平衡，考虑特征交互	可能难以识别少量关键特征	中	大部分机器学习任务

二、降维（Dimensionality Reduction）

降维是指将高维数据映射到低维空间，同时尽可能保留数据中重要的信息。

1. **线性降维方法**

- **主成分分析（Principal Component Analysis, PCA）：** 通过正交变换将原始数据投影到一组新的正交基上，使得投影后的数据方差最大化。假设原始数据矩阵为 $x \in R^{n\times p}$ ，PCA的目标是找到 k 个正交的主成分 $u_1, u_2, u_3 ... (k < p)$ ，使得数据在这些主成分上的投影方差最大。

$$\text{maximize } \text{Var}(Xu) = u^T \Sigma u \quad \text{subject to } \|u\|_2 = 1$$

其中，Σ 是数据 X 的协方差矩阵。

- **线性判别分析（Linear Discriminant Analysis, LDA）：** 一种监督学习的降维方法，旨在找到能够最好地区分不同类别的投影方向。LDA的目标是最大化类间散度矩阵 S_B 和最小化类内散度矩阵 S_W 的比值。

$$\text{maximize } \frac{\mathbf{w}^T S_B \mathbf{w}}{\mathbf{w}^T S_W \mathbf{w}}$$

其中，w 是投影向量，$S_B = \sum_{i=1}^{c} n_i(\mu_i - \mu)(\mu_i - \mu)^T$ 是类间散度矩阵，$S_W = \sum_{i=1}^{c} \sum_{\mathbf{x} \in D_i} (\mathbf{x} - \mu_i)(\mathbf{x} - \mu_i)^T$ 是类内散度矩阵，c 是类别数，ni 是第 i 类样本数，μi 是第 i 类样本的均值，μ 是所有样本的均值，Di 是第 i 类样本的集合。

2. **非线性降维方法**

- **t-分布邻域嵌入（t-distributed Stochastic Neighbor Embedding, t-SNE）：** 一种非线性降维技术，主要用于高维数据的可视化。它试图在低维空间中保留高维空间中数据点之间的局部邻域结构。
- **均匀流形逼近和投影（Uniform Manifold Approximation and Projection, UMAP）：** 也是一种非线性降维技术，与 t-SNE 类似，但通常在保留全局结构方面表现更好，并且计算效率更高。
- **核主成分分析（Kernel PCA）：** 通过使用核函数将数据映射到高维特征空间，然后在该空间中执行 PCA，从而实现非线性降维。

🤔 **相关问题：L1正则化为什么能够起到特征选择的作用**

1.2 Loss NaN

✏️ **问题：如果训练过程中出现 Loss NaN，可能有哪些原因？如何排查？**

来源：字节跳动、美团

在训练过程中出现Loss NaN（Not a Number）的原因较为复杂，涉及到数据、模型、训练配置等多个方面。以下是一些可能的原因及排查方法：

数据问题

- **存在缺失值或异常值**
  - **原因：数据集中存在缺失值，或者有一些极端的异常值，可能导致模型在计算梯度时出现不稳定情况，进而产生NaN。**
  - **排查方法**：使用数据处理工具或代码来检查数据集中是否存在缺失值，对每列数据进行统计分析，查看是否有明显的异常值，如过大或过小的数据。
- **数据标签错误**
  - **原因**：如果标签数据存在错误，例如分类任务中标签超出了合理的类别范围，会使模型的损失计算出现异常。

- **排查方法**：检查标签数据的取值范围和正确性，确保标签与数据的对应关系准确无误。可以通过可视化部分数据及其标签来初步检查，也可以对标签数据进行统计分析，查看是否存在不合理的分布。

模型问题

- **模型结构不合理**
  - **原因**：**模型过于复杂，参数过多，导致模型在训练时难以收敛，容易出现Loss NaN的情况。或者模型结构存在问题，例如网络层之间的连接错误、维度不匹配等。**
  - **排查方法**：检查模型的结构定义，确保各层之间的连接和维度设置正确。可以尝试简化模型结构，减少参数数量，观察Loss是否稳定。
- **权重初始化不当**
  - **原因**：权重初始化的值过大或过小，可能导致神经元的输出过大或过小，从而使梯度计算出现异常，最终导致Loss NaN。
  - **排查方法**：尝试不同的权重初始化方法或调整初始化参数，例如使用 Xavier 初始化、He 初始化等，并观察训练结果。

训练配置问题

- **学习率过高**
  - **原因**：**学习率过大，会使模型在参数更新时步长过大，可能导致模型无法收敛，甚至出现Loss NaN的情况。**
  - **排查方法**：尝试降低学习率，例如将学习率设置为原来的十分之一，然后重新训练模型，观察Loss的变化情况。也可以使用学习率衰减策略，让学习率在训练过程中逐渐降低。
- **梯度爆炸**
  - **原因**：在深度神经网络中，由于梯度在反向传播过程中不断累积，可能会导致梯度值变得非常大，从而引发Loss NaN。
  - **排查方法**：可以通过观察梯度的范数来判断是否发生了梯度爆炸。如果梯度范数过大，可以采用梯度裁剪的方法，将梯度限制在一定的范围内。

1.3 FFN

> ✏️ **问题：为何LLM中的FFN中需要先升维再降维**
>
> **来源：腾讯、京东**

答案：

LLM 中 FFN 的“先升维再降维”结构，特别是中间层的扩展，是为了**赋予模型在每个位置独立地学习和应用复杂的非线性转换的能力。升维提供了足够的模型容量和表达空间，以便在应用非线性激活后捕捉更丰富的特征**；而**降维则满足了残差连接和层堆叠的结构需求，并将高维空间的有益信息提炼回**

主要的表示维度。这种结构是平衡了表达能力、计算效率和模型架构一致性的设计选择。**四倍的扩展比例**（4 * d_model）是在实践中通过实验证明有效且广泛采用的一个经验值。

1. **增强模型的表达能力和学习更复杂的特征：**

- 自注意力层能够有效地聚合来自序列中其他位置的信息，生成一个包含上下文信息的表示。
- 然而，自注意力层本身主要是线性的加权求和。为了在**每个位置**独立地对这些上下文信息进行深入处理和转换，需要一个非线性的模块。FFN 提供了这种能力。
- 将维度从 `d_model` 扩展到 `4 * d_model`（或类似的比例）并在中间层应用非线性激活函数（如 GELU, ReLU 等），可以极大地增加网络的**容量**和**表达能力**。这允许网络学习和捕捉输入表示中更丰富、更抽象的模式和特征，这些模式是仅通过线性变换无法获得的。

2. **为非线性激活函数提供更广阔的空间：**

- 非线性激活函数是神经网络学习复杂函数关系的关键。但**非线性激活函数（特别是 ReLU 及其变种）可能导致信息的“丢失”或表示的“降秩”**。这种降秩的效果是一个概率事件，取决于输入数据和学习到的权重。
- **通过在非线性激活之前将维度显著提升，可以概率性地降低激活函数导致有效秩显著下降的风险，从而保证经过整个 FFN 后输出的表示仍具有足够的表达能力。**
- 在更高的维度空间中应用非线性激活函数，可以让网络在特征空间中创建更复杂的决策边界和转换。想象一下，在一个高维空间中，有更多的“神经元”可以被激活或抑制，从而能够对输入的各种组合做出更精细的响应。这使得 FFN 能够执行比在低维空间中更复杂的特征映射。

3. **弥补自注意力层的局限性（某种程度上）：**

- 自注意力层虽然强大，但它主要侧重于捕捉词元之间的**关系**和**依赖性**。
- FFN 则是一个位置感知（position-wise）的模块，它独立地应用于每个位置的向量。它允许模型在每个位置上“思考”，并基于自注意力层提供的全局信息，独立地强化或转换该位置的表示。
- 升维结构提供了足够的“内部空间”，让 FFN 能够充分处理注意力层的输出，并生成一个更精炼的、准备好传递给下一层的表示。

4. **结构上的需求和效率考虑（降维）：**

- **残差连接 (Residual Connections):** Transformer 块广泛使用了残差连接，将 FFN 的输入加到其输出上（`Output = Input + FFN(Input)`）。为了使残差连接正常工作，FFN 的输入维度和输出维度必须相同（都是 `d_model`）。因此，在升维处理后，需要一个降维的线性层将维度映射回 `d_model`。
- **层堆叠的需求：** Transformer 模型通常由多个相同的层堆叠而成。为了方便层与层之间的堆叠，并保持整个模型的结构一致性，每一层的输入和输出维度通常保持一致（`d_model`）。
- **计算效率（相对于全连接网络）：** 虽然 FFN 在通道维度上进行了扩展，但它对序列长度是独立的，并且是并行应用于序列中每个位置的。这比在整个序列上应用一个巨大的全连接网络要高效得多。

1.4 激活函数

> ✏️ **问题：了解哪些激活函数，简单讲一下它们的原理和特点**
>
> **来源：腾讯、字节跳动**

激活函数是神经网络中非常重要的组成部分，它们引入了非线性，使得网络能够学习和逼近复杂的函数关系。没有激活函数，多层神经网络就只会是线性变换的堆叠，等价于单层网络。

以下是一些常用的激活函数的总结：

1. **Sigmoid 函数 (Logistic Activation Function)**
   - **公式：**
     $$\sigma(x) = \frac{1}{1 + e^{-x}}$$
   - **图示/形状：** S形曲线，将任意实数压缩到 (0, 1) 的区间。
   - **输出范围：** (0, 1)
   - **关键特性：** 非线性，可微，输出具有概率解释（例如用于二分类的输出层）。
   - **常用用途：** 早期的神经网络隐藏层，现在主要用于二分类的输出层，以及循环神经网络（RNN）中的门控机制（如 LSTM, GRU）。
   - **优点：** 将输出映射到 (0, 1) 区间，方便解释为概率。
   - **缺点：**
     - **梯度消失 (Vanishing Gradient)：** 当输入 x 的绝对值较大时，梯度接近于零，导致反向传播时梯度难以传递，训练困难。
     - **输出不以零为中心：** 输出恒大于 0，这可能导致下一层的输入不以零为中心，影响梯度下降的效率。
2. **Tanh 函数 (Hyperbolic Tangent Function)**
   - **公式：**
     $$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}$$
   - **图示/形状：** S形曲线，将任意实数压缩到 (-1, 1) 的区间。
   - **输出范围：** (-1, 1)
   - **关键特性：** 非线性，可微，输出以零为中心。
   - **常用用途：** 隐藏层，常用于替代 Sigmoid，在 RNN 的门控机制和状态更新中也常用。
   - **优点：** 输出以零为中心，有助于解决 Sigmoid 函数输出不以零为中心的问题，通常收敛速度比 Sigmoid快。

- **缺点：** 仍然存在梯度消失问题。

3. **ReLU 函数 (Rectified Linear Unit)**
   - **公式：**
   $$f(x) = \max(0, x)$$
   - **图示/形状：** 当 x≤0 时输出 0，当 x>0 时输出 x。一个分段线性的函数。
   - **输出范围：** $[0, \infty)$
   - **关键特性：** 非线性（尽管是分段的），计算非常高效（只需要一个阈值判断和赋值），在正区间内没有梯度消失问题。
   - **常用用途：** 当前最常用的隐藏层激活函数，在深度学习模型中非常普遍。
   - **优点：**
     - 有效缓解梯度消失问题（在正区间）。
     - 计算速度快，加速网络训练。
     - 引入稀疏性，可能有助于特征提取。
   - **缺点：**
     - 死亡 ReLU (Dying ReLU)： 当输入永小于等于 0 时，神经元会永久输出 0，梯度也为 0，导致该神经元不再更新权重，形同“死亡”。
     - 输出不以零为中心。

4. **Leaky ReLU 函数**
   - **公式：**
   $$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$$，其中 α 是一个很小的正数，例如 0.01。

- **图示/形状：** 与 ReLU 类似，但在负区间有一个小的非零斜率 α。
- **输出范围：** $(-\infty, \infty)$
- **关键特性：** 非线性，计算高效，尝试解决死亡 ReLU 问题。
- **常用用途：** 隐藏层，作为 ReLU 的改进。
- **优点：** 解决了死亡 ReLU 问题，因为在负区间仍有非零梯度。
- **缺点：** 性能提升不总是稳定的；斜率 α 是一个超参数需要手动选择。

5. **PReLU 函数 (Parametric ReLU)**
   - **公式：**
   $$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha x & \text{if } x \leq 0 \end{cases}$$，其中 α 是一个可学习的参数，可以通过反向传播自动优化

- **图示/形状：** 与 Leaky ReLU 类似，但负区间的斜率是学习得到的。
- **输出范围：** $(-\infty, \infty)$
- **关键特性：** 非线性，可学习的负区间斜率，是 ReLU 和 Leaky ReLU 的泛化。
- **常用用途：** 隐藏层。
- **优点：** α 可以通过数据学习，潜在地获得更好的性能。
- **缺点：** 引入了额外的参数，可能增加过拟合的风险（尤其在数据量较小时）。

6. **ELU 函数 (Exponential Linear Unit)**
   - **公式：**

$$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$$,其中 α 是一个正的常数（常见取值为 1）。

   - **图示/形状：** 正区间是线性，负区间是指数曲线，平滑地逼近 $-\alpha$。
   - **输出范围：** $(-\alpha, \infty)$
   - **关键特性：** 非线性，在负区间是平滑的（可微），输出均值接近零。
   - **常用用途：** 隐藏层。
   - **优点：** 解决了死亡 ReLU 问题；输出的均值更接近零，有助于优化；在负区间更平滑，可能有更好的收敛性。
   - **缺点：** 计算成本比 ReLU/Leaky ReLU 高，因为它包含指数运算。

7. **GELU 函数 (Gaussian Error Linear Unit)**
   - **公式：**

$$f(x) = x \cdot P(X \leq x) = x \cdot \Phi(x)$$

其中 $\Phi(x)$ 是标准正态分布的累积分布函数 (CDF)。常用的近似公式有基于 Tanh 和 Sigmoid 的。

   - 近似公式 (基于 Tanh)： $f(x) \approx 0.5x\left(1 + \tanh\left(\sqrt{2/\pi}\left(x + 0.044715x^3\right)\right)\right)$
   - 近似公式 (基于 Sigmoid)： $f(x) \approx x \cdot \text{sigmoid}(1.702x)$
   - **图示/形状：** 光滑的非线性函数，形状类似于 ReLU，但在 x=0 附近更平滑，并允许小的负值输入有小的负输出。
   - **输出范围：** $(-\infty, \infty)$ (近似)
   - **关键特性：** 非线性，光滑，在许多最先进的模型（特别是基于 Transformer 的模型，如 BERT, GPT 系列）中表现优异。
   - **常用用途：** 隐藏层，尤其在大型预训练语言模型中非常流行。
   - **优点：** 在许多任务上表现优于 ReLU 和其变种；光滑性有助于优化。
   - **缺点：** 计算成本高于 ReLU/Leaky ReLU。

8. **Swish / SiLU 函数 (Sigmoid Linear Unit)**
   - **公式：**

$$f(x) = x \cdot \sigma(\beta x)$$

   其中 σ 是Sigmoid函数，β 是一个参数（通常设置为 1 或可学习）。
   - **图示/形状：** 光滑的非线性函数，形状类似于 GELU，允许小的负值输出。
   - **输出范围：** 大约在 $(-0.278, \infty)$
   - **关键特性：** 非线性，光滑，在某些模型（如 EfficientNet）中表现良好。
   - **常用用途：** 隐藏层。
   - **优点：** 在一些任务上表现优于 ReLU；光滑性有助于优化。
   - **缺点：** 计算成本高于 ReLU/Leaky ReLU。
9. **Softmax 函数**
   - **公式：** 对于输入向量 $z = [z_1, z_2, \ldots, z_K]$，输出向量的第 i 个分量为：

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

   - **图示/形状：** 将一个任意实数向量转换为一个概率分布，所有分量都在 (0, 1) 区间，且所有分量之和为 1。
   - **输出范围：** 输出向量的每个分量在 (0, 1) 区间，且所有分量之和为 1。
   - **关键特性：** 非线性，可微，输出是一个概率分布。
   - **常用用途：** **几乎只用于多类别分类任务的输出层**。它将网络的输出转化为每个类别的概率。
   - **优点：** 直接提供类别的概率分布。
   - **缺点：** 不适合作为隐藏层的激活函数，因为它强制输出分量和为 1，限制了表示能力。

2. 注意力机制

2.1 手写注意力

> ✏️ **问题：手写实现多头注意力机制（MHA），并加入键值缓存（KV cache），同时阐述相对位置编码（RoPE）应添加在何处及其作用原理。**
>
> **来源：百度、字节跳动**

回答：先直接上代码，MHA外加KV cache。

MHA + KV Cache

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

class MultiHeadAttention(nn.Module):
    def __init__(self, hidden_size, num_heads):
        super().__init__()
        self.num_heads = num_heads
        self.head_dim = hidden_size // num_heads
        self.q_linear = nn.Linear(hidden_size, hidden_size)
        self.k_linear = nn.Linear(hidden_size, hidden_size)
        self.v_linear = nn.Linear(hidden_size, hidden_size)
        self.o_linear = nn.Linear(hidden_size, hidden_size)

    def forward(self, hidden_state, causal_mask=None, past_key_value=None, use_cache=False):
        batch_size = hidden_state.size(0)
        # 计算 Q、K、V
        query = self.q_linear(hidden_state)  # (batch_size, seq_len, hidden_size)
        key = self.k_linear(hidden_state)
        value = self.v_linear(hidden_state)

        # 分割多头
        query = query.view(batch_size, -1, self.num_heads, self.head_dim).transpose(1, 2)  # (batch_size, num_heads, seq_len, head_dim)
        key = key.view(batch_size, -1, self.num_heads, self.head_dim).transpose(1, 2)
        value = value.view(batch_size, -1, self.num_heads, self.head_dim).transpose(1, 2)

        # 若存在缓存，拼接当前 K、V
        if past_key_value is not None:
            past_key, past_value = past_key_value
            key = torch.cat([past_key, key], dim=2)  # (batch_size, num_heads, seq_len, head_dim)
            value = torch.cat([past_value, value], dim=2)

        # 保存新的缓存
        new_past_key_value = (key, value) if use_cache else None

        # 计算注意力分数
        attention_scores = torch.matmul(query, key.transpose(-1, -2)) / torch.sqrt(torch.tensor(self.head_dim, dtype=torch.float32))

        # 应用因果掩码（若需要）
        if causal_mask is not None:
```

```
            attention_scores += causal_mask * -1e9

        # 计算注意力输出
        attention_probs = F.softmax(attention_scores, dim=-1)
        output = torch.matmul(attention_probs, value)

        # 合并多头并线性变换
        output = output.transpose(1, 2).contiguous().view(batch_size, -1, self.num_heads * self.head_dim)
        output = self.o_linear(output)

        return (output, new_past_key_value) if use_cache else output
```

- **键值缓存（KV Cache）的实现：**在上述代码中，`past_key_value` 参数用于存储之前的 `key` 和 `value`，在每次生成新 token 时，可以直接使用缓存中的 `key` 和 `value`，而无需重新计算。
- **相对位置编码（RoPE）的添加位置及作用原理：**
  - **添加位置：**相对位置编码（RoPE）通常应用于 `query` 和 `key` 的计算之前。在上述代码中，可以在计算 `query` 和 `key` 之后、计算注意力分数之前，对它们进行 RoPE 编码。
  - 作用原理：RoPE 通过复数旋转的方式将相对位置信息嵌入到 `query` 和 `key` 中。具体来说，对于每个位置的向量，将其拆分为二维向量对，然后通过旋转矩阵进行变换。旋转角度由位置和维度索引决定，具体公式为：

$$\begin{pmatrix} x_1' \\ x_2' \end{pmatrix} = \begin{pmatrix} \cos(\theta_i) & -\sin(\theta_i) \\ \sin(\theta_i) & \cos(\theta_i) \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

其中，$\theta_i = \frac{1}{10000^{2k/d}}$，$k$ 是维度索引。通过这种方式，RoPE 能够将相对位置信息融入到注意力机制中，而不需要额外的参数。

2.2 PagedAttention

✏️ **问题：讲一下PagedAttention的原理、工作流程和优点。**

来源：腾讯

答案：

PagedAttention是一种受操作系统虚拟内存和分页技术启发的注意力算法，用于解决大型语言模型（LLM）服务中的内存管理问题。以下是其相关介绍：

- **原理：将请求的KV缓存划分为多个块（KV block），每个块包含固定数量的token的键和值向量。这些块可以存储在非连续的物理内存中，通过块表记录逻辑块与物理块的映射关系，从而避免了连续内存分配带来的内部和外部碎片化问题。**
- **工作流程：在prefill阶段，将输入的prompt根据设定的块大小划分为若干逻辑块，然后映射到物理块，系统正常计算prompt的KV值后，将这些值填入对应的物理块中。在decoding阶段，使用KV cache计算attention并生成新的词，计算过程中从逻辑块获取数据，但实际数据是通过后台的块表映射关系从对应的物理块中获取的。基于新生成的词，系统会对逻辑块、物理块和块表进行更新。**
- **优势**：通过动态分配和回收内存块，**有效降低了内存占用**，解决了LLM服务中的内存瓶颈问题。同时，**分页机制避免了内存碎片化，提高了内存利用率**，使得LLM能够在相同延迟下实现更高的吞吐量，尤其在处理更长序列、更大模型和更复杂解码算法时效果显著。此外，还支持KV共享，对于并行采样等技术有帮助。

最后说下，PagedAttention 是 vLLM 系统的基石，为 LLM 推理中的内存管理和性能提升提供了有效的解决方案。

2.3 MHA复杂度分析

> ✏️ **问题：请分析多头注意力机制在时间和空间上的复杂度情况。**
>
> **来源：腾讯、小红书**

答案：

- **时间复杂度：**
  - **计算注意力得分**：对于一个具有n个输入向量的序列，每个向量的维度为 d ，在计算注意力得分时，需要计算每个位置与其他所有位置的相似度，这通常通过矩阵乘法和点积操作来实现。具体来说，对于每个查询向量 Q ，需要与键向量 K 进行矩阵乘法，得到一个 $n \times n$ 的注意力得分矩阵，这个操作的时间复杂度为 $O(n^2 d)$ 。
  - **计算加权求和**：在得到注意力得分后，需要将其与值向量 V 进行加权求和，得到最终的输出。这个操作的时间复杂度为 $O(n^2 d)$ ，因为需要对每个位置的注意力得分与对应的价值向量进行乘法和求和操作。
  - **多头计算**：如果使用 h 个头的多头注意力机制，那么需要对每个头分别进行上述计算，因此总的时间复杂度为 $O(hn^2 d)$ 。
- **空间复杂度：**
  - **存储输入和中间结果**：需要存储输入的查询、键和值向量，以及在计算过程中产生的中间结果，如注意力得分矩阵等。对于一个具有 n 个输入向量的序列，每个向量的维度为 d ，存储这些向量的空间复杂度为 $O(3nd)$ （假设查询、键和值向量的维度相同）。而注意力得分矩阵的空间复杂度为 $O(n^2)$ ，因为它是一个 $n \times n$ 的矩阵。

- **存储多头结果**：如果使用 h 个头的多头注意力机制，还需要存储每个头的输出结果，其空间复杂度为 $O(hnd)$。因此，总的空间复杂度为 $O(3nd + n^2 + hnd)$，当 n 较大时，空间复杂度主要由注意力得分矩阵决定，即 $O(n^2)$。

多头注意力机制的时间和空间复杂度都相对较高，尤其是在处理长序列时。这是因为它需要对序列中的每个位置与其他所有位置进行交互，导致计算量和存储空间随着序列长度的增加而呈平方增长。为了降低复杂度，一些改进的注意力机制如稀疏注意力、线性注意力等被提出，以在保证性能的同时减少计算和存储开销。

2.4 Attention中的d_k

> ✏️ **问题：Transformer中的Attention为什么要除以d_k**
>
> **来源：阿里巴巴**

答案：

在Transformer架构里，Scaled Dot - Product Attention的计算公式为：

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

这里的 Q 是查询矩阵，K 是键矩阵，V 是值矩阵，d_k 是键向量的维度。除以 $\sqrt{d_k}$ 主要有下面两个原因：

- 避免点积结果过大：当 d_k 较大时，Q 和 K 的点积结果会随之增大。我们可以从向量点积的公式来理解，设两个向量 $\mathbf{q}$ 和 $\mathbf{k}$，它们的点积为：$\mathbf{q} \cdot \mathbf{k} = \sum_{i=1}^{d_k} q_i k_i$。随着 d_k 的增加，这个求和的项数增多，点积的结果也会变大。在进行softmax操作时，softmax函数为：$\text{softmax}(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$。如果 x_i 的值很大，e^{x_i} 就会变得非常大，导致softmax函数的梯度变得极小，也就是出现梯度消失的问题。通过除以 $\sqrt{d_k}$，可以对 QK^T 的结果进行缩放，避免点积结果过大，从而让softmax函数的输入处于一个更合适的范围，使得梯度能够更稳定地传播。
- 保持输入分布的稳定性：从概率分布的角度来看，除以 $\sqrt{d_k}$ 有助于保持输入到softmax函数的分布相对稳定。当 Q 和 K 的元素是从均值为 0，方差为 1 的分布中采样得到时，Q 和 K 的点积的方差会随着 d_k 的增大而增大。通过除以 $\sqrt{d_k}$，可以将点积结果的方差重新调整为 1，这样就保证了在不同的 d_k 下，输入到softmax函数的分布具有相对一致的特性，使得模型的训练更加稳定。

综上所述，在Transformer的Attention机制中除以 $\sqrt{d_k}$ 是为了避免点积结果过大引发的梯度消失问题，同时保持输入分布的稳定性，进而提升模型的训练效果和稳定性。

2.5 Block Attention (分块注意力)

> 🥖 **问题：Block Attention是解决什么问题的？讲讲Mixture of Block Attention。**
>
> **来源：字节跳动**

解决什么问题?

简单来说，标准的自注意力机制（就是Transformer里那个核心部件）在处理很长的文本（比如一整本书或很长的对话）时，会变得**非常慢**，而且**特别吃内存**。因为每个字都要和文本里的其他所有字计算一遍“注意力得分”，文本越长，计算量就指数级增长 (N^2 复杂度)。

Block Attention 是怎么做的?

1. **分块：** 把长文本切成一小段一小段的“块”(Block)。
2. **块内注意：** 主要让每个字在它自己所在的那个小“块”内部计算注意力。这样，每个字只需要跟少数一些字互动，而不是跟全文所有字互动。
3. **块间交互 (可选但常见)：** 为了让信息能在不同块之间流动，还会设计一些机制，比如允许一个块和它旁边的几个块也进行一些注意力计算（比如“滑动窗口”），或者设置一些“全局块”让所有块都能看到。

好处：

- **快多了：** 计算量大大减少。
- **省内存：** 不用存那么大的注意力矩阵了。
- **能处理更长的文本：** 因为又快又省，所以模型就能看懂更长的文章了。

打个比方：
标准注意力就像开一个超大型会议，每个人都要和在场的其他所有人单独聊一遍，效率极低。
Block Attention 就像把人分成很多小组，大部分讨论在小组内进行，小组之间再派代表或者跟邻近小组交流一下，效率就高多了。

Kimi 的 Mixture of Block Attention (MoBA，混合分块注意力)

Kimi的MoBA在传统分块注意力的基础上进行了创新，它的核心在于“混合”机制。不同于固定分块策略，**MoBA通过可学习的门控网络动态决定每个查询块应关注的键值块组合，形成一种“专家混合”模式。这种设计带来三大优势：其一，自适应稀疏性——根据输入内容特性自动调整注意力范围，重要部分分配更多资源，冗余部分则简化计算；其二，层级交互——实现细粒度的块间通信，既保留局部结构又捕捉长距离依赖；其三，负载均衡——通过门控机制平衡各计算单元的工作量，避免传统MoE中常见的专家崩溃问题。**

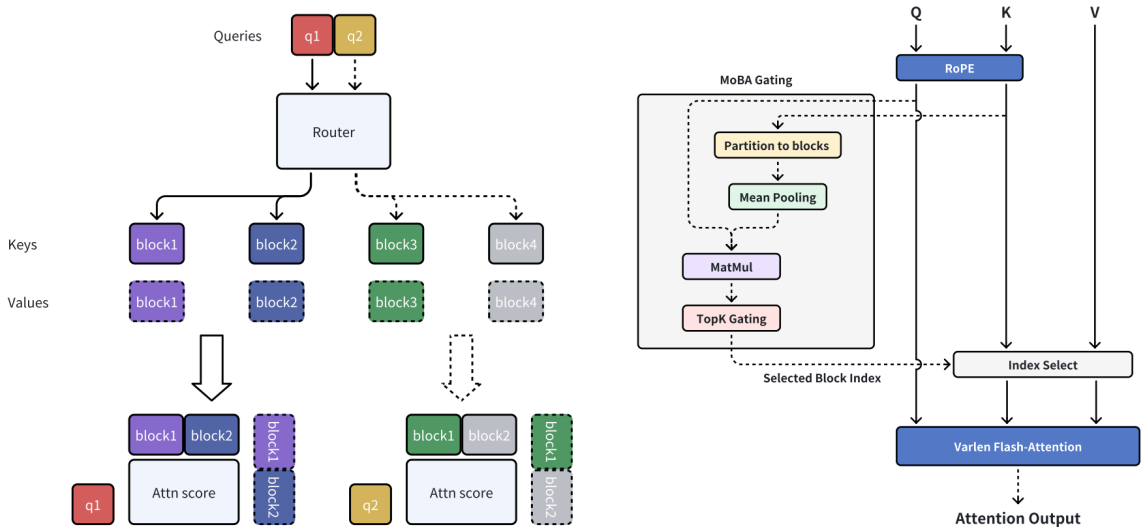

Figure 1: Illustration of mixture of block attention (MoBA). **(a)** A running example of MoBA; **(b)** Integration of MoBA into Flash Attention.

3. Transformer相关

3.1 RNN/LSTM/GRU/Transformer

✏️ **问题：RNN/LSTM/GRU/Transformer几种网络有哪些特点和区别**

来源：腾讯、阿里巴巴

答案：

模型	关键特点	优点	缺点	适用场景
RNN	循环连接，顺序处理数据	基础模型，结构简单	长序列梯度消失，易“遗忘”早期信息	简单序列任务（现较少使用）
LSTM	引入三个门（输入/遗忘/输出门），控制信息流动	长期记忆能力强，缓解梯度消失	计算复杂，参数量大，速度慢	需长时依赖的任务（翻译、语音识别）
GRU	简化LSTM为两个门（更新/重置门），合并单元状态	计算效率高，内存占用少，效果接近LSTM	超长序列表现略逊于LSTM	中等长度序列，资源受限场景（如移动端模型）
Transformer				

	完全并行化，自注意力机制，位置编码	全局信息捕捉，训练快，长距离依赖处理强	需大量数据/算力，推理时显存消耗高	大规模任务（GPT、BERT等预训练模型基础）

对比总结

- **记忆能力**：Transformer > LSTM ≈ GRU > RNN
- **计算效率**：Transformer（训练） > GRU > LSTM > RNN
- **资源需求**：Transformer ≫ LSTM > GRU > RNN

选择建议

- 短序列简单任务：RNN（历史意义 > 实用价值）
- 中等长度+有限资源：GRU
- 超长序列+高性能：LSTM（需接受速度代价）
- 大数据+强算力：Transformer（现代任务首选）

附加说明

- Transformer的变体（如Swin Transformer）已支持图像等非序列数据
- 工业部署中，LSTM/GRU因轻量化仍用于实时系统（如手机输入法预测）
- RNN的变体（如双向RNN）在特定小规模任务中仍有应用

3.2 旋转位置编码

> ✏️ **问题：请详细说明旋转位置编码（Rotary Position Embedding）的工作原理及其在Transformer中的应用。**
>
> **来源：腾讯、微众银行、滴滴**

答案：

RoPE 是一种通过位置相关的旋转操作将相对位置信息注入到 Query 和 Key 向量中的位置编码方法。它使得 Attention 点积结果直接依赖于词元间的相对距离，从而提高了模型对位置信息的理解能力，尤其是在处理长序列和需要良好位置泛化能力的场景中表现出色，是当前许多先进大型语言模型中常用的位置编码技术。

1. **Transformer 对位置信息的需求**

首先，我们需要理解为什么需要位置编码。标准的 Transformer 架构，特别是其自注意力机制，inherently 是排列不变的。这意味着它无法区分序列中词元的顺序。然而，在自然语言处理等任务中，词元的位置和它们之间的相对距离是至关重要的语义信息。因此，我们需要一种方法将位置信息注入到词元嵌入中。

传统的做法是使用**绝对位置编码**，比如正弦位置编码或学习的位置编码。这些方法生成一个与位置相关的向量，并将其加到或拼接到词元嵌入中。虽然这为模型提供了绝对位置信息，但在处理训练时未见过的长序列或需要精确捕捉相对位置关系时可能存在局限性。

2. RoPE 的核心思想

RoPE 的核心思想是，不直接将位置信息加到词元嵌入上，而是利用基于位置的**旋转**来修改 Query 和 Key 向量。这种修改是精心设计的，使得任意两个向量 Q 和 K 在经过 RoPE 旋转后计算的点积，只依赖于它们原始向量的值以及它们之间的**相对位置**。

具体来说，对于位置 m 的向量 q 和位置 n 的向量 k ，RoPE 对它们应用一个函数 f ，得到 $f(q,m)$ 和 $f(k,n)$ 。RoPE 设计的目标是让它们的点积满足：

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

其中 g 是一个函数，它仅仅依赖于原始向量 q ,k 和它们的相对位置 $m-n$

3. RoPE 的数学原理

RoPE 的数学实现基于复数乘法或二维向量的旋转。

二维向量 (i,j) 可以表示为复数 $x+iy$ 。将这个向量旋转角度 θ 对应于在复平面上乘以 $ei\theta = cos\theta + isin\theta$:

$$(x+iy)(\cos\theta + i\sin\theta) = (x\cos\theta - y\sin\theta) + i(x\sin\theta + y\cos\theta)$$

或者使用二维旋转矩阵：

$$\begin{pmatrix}\cos\theta & -\sin\theta\\ \sin\theta & \cos\theta\end{pmatrix}\begin{pmatrix}x\\ y\end{pmatrix} = \begin{pmatrix}x\cos\theta - y\sin\theta\\ x\sin\theta + y\cos\theta\end{pmatrix}$$

RoPE 将高维向量（如 Query 或 Key 向量）视为多个二维向量对（或复数）的拼接。对于一个 d 维的向量 v 在位置 m ，RoPE 对其应用旋转操作 $f(v,m)$ 。这个操作对每对维度 (v_{2j}, v_{2j+1}) 应用一个与位置 m 相关的旋转。使用复数形式表示第 j 对维度： $f(\mathbf{v},m)_j = (v_{2j} + iv_{2j+1})e^{im\theta_j}$

其中 $v_{2j} + iv_{2j+1}$ 是向量 v 中第 $2j$ 和 $2j+1$ 维组成的复数，而 $e^{im\theta j}$ 是旋转因子， m 是位置索引， θ_j 是为第 j 对维度预设的旋转频率。这些频率通常是按指数衰减设置的，例如： $\theta_j = 1/\text{base}^{2j/d}$

其中 base 是一个常数（如 10000），d 是向量维度。

现在，我们来看为什么点积会与相对位置相关。对于位置 m 的 $Query$ 向量 q 和位置 n 的 Key 向量 k ，它们的第 j 对维度经过 RoPE 编码后变为 $(q_{2j} + iq_{2j+1})e^{im\theta_j}$ 和 $(k_{2j} + ik_{2j+1})e^{in\theta_j}$ 。它们对点积的贡献是这两个复数乘积的实部：

$$\begin{aligned}&\text{Re}\left((q_{2j} + iq_{2j+1})e^{im\theta_j}\,\overline{(k_{2j} + ik_{2j+1})e^{in\theta_j}}\right)\\ &= \text{Re}\left((q_{2j} + iq_{2j+1})(k_{2j} - ik_{2j+1})e^{i(m-n)\theta_j}\right)\end{aligned}$$

这里的关键是 $e^{i(m-n)\theta_j}$ 项，它是一个只依赖于相对位置 $m-n$ 的旋转因子。整个点积是所有维度对贡献的总和，因此最终的点积 $\langle f(q,m), f(k,n)\rangle$ **只依赖于原始向量 q ,k 以及它们的相对位置 $m-n$** 。

4. **RoPE 在 Transformer 中的应用**

RoPE 主要应用于 Transformer 的 Self-Attention 机制。标准的 Attention 计算注意力分数的方式是 QK^T ：

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

在使用 RoPE 的 Transformer 中，注意力分数的计算是：

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

这里的 $f(Q, pos)$ 表示对矩阵 Q 的每一行（对应一个 $Query$ 向量）应用 RoPE 旋转，旋转角度取决于该行词元的位置。 $f(K, pos)$ 同理。

因此，注意力分数矩阵中第 i 行第 j 列的元素，即第 i 个 $Query$ 与第 j 个 Key 的点积，变为了：

$$\text{score}(i, j) = \langle f(\mathbf{q}_i, i), f(\mathbf{k}_j, j)\rangle$$

这个点积天然地编码了位置 i 和位置 j 之间的相对距离 $i - j$ ，因为它只依赖于 $i - j$ 。

5. **RoPE 的优势**

相比传统的绝对位置编码，RoPE 具有以下优势：

- **天然的相对位置编码：** 它将相对位置信息直接编码在 Q 和 K 向量的点积中，这更符合 Attention 机制捕捉词元间关联性的方式。
- **更好的长序列外推能力：** 由于注意力计算只依赖于相对位置，RoPE 使得模型在处理比训练时更长的序列时，性能下降更为平缓，展现出更好的泛化能力。
- **与因果注意力兼容：** 在生成式模型中，RoPE 可以很自然地与因果注意力（只 attends 到当前位置及之前的位置）结合。
- **效率：** RoPE 的计算是线性的，对每个向量进行分组旋转，计算开销相对较低。

🤔 **相关问题：旋转位置编码（Rotary Position Embedding）相比绝对位置编码的优势是什么**

3.3 绝对位置编码与相对位置编码

✏️ **问题：除了旋转位置编码，还了解哪些位置编码**

来源：腾讯、微众银行、美团

除了旋转位置编码（RoPE），Transformer 模型中还存在几种其他常用的位置编码方法，它们各有特点和适用场景。可以根据是编码**绝对位置**还是**相对位置**，以及位置信息是**固定的**还是**可学习的**来进行分类。

以下是几种常见的其他位置编码方法：

一、绝对位置编码 (Absolute Positional Encoding)

1. 正弦位置编码 (Sinusoidal Positional Encoding)

- **工作原理：** 这是原始 Transformer 论文中提出的方法。它使用不同频率的正弦和余弦函数生成固定维度的位置向量。对于位置 p 和维度 i ，位置编码向量的第 i 个分量计算如下：

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

其中 d_{model} 是词元嵌入的维度。通过使用不同的频率（由 $10000^{2i/d_{model}}$ 控制），使得每个位置都有一个独一无二的编码，同时高维分量对位置变化更敏感（更高频率），低维分量对位置变化更平缓（更低频率）。

- **应用方式： 将生成的位置编码向量直接加到对应的词元嵌入向量上，**作为 Transformer 输入层最终嵌入表示。
- **特点：** 固定（不可学习）、绝对、加性。
- **优点：** 无需引入额外的参数，计算简单；理论上可以通过正弦/余弦函数的周期性外推到训练时未见过的更长序列（尽管实践效果有限）。
- **缺点：** 直接编码的是绝对位置，模型需要通过后续层自己学习如何利用这些绝对位置信息来推理相对位置；在实际应用中，对超长序列的外推能力有限。

2. 学习的位置编码 (Learned Positional Encoding)

- **工作原理：** 不使用固定的函数生成位置编码，而是将位置编码视为模型的一组**可学习参数**。模型维护一个查找表 (lookup table)，表的每一行对应一个位置的编码向量。例如，如果最大序列长度是 512，嵌入维度是 d_{model} ，那么位置编码就是一个 $512 \times d_{model}$ 的参数矩阵。（Bert的位置编码）
- **应用方式： 与正弦位置编码类似，通过查找表获取对应位置的位置编码向量，然后将其加到对应的词元嵌入向量上。**
- **特点：** 可学习、绝对、加性。
- **优点：** 根据具体任务和数据集学习到最适合的位置表示，理论上可能比固定的正弦编码更能捕捉位置信息。
- **缺点：** 无法直接外推到训练时未见过的更长序列（超过查找表大小的位置没有对应的编码）；引入了额外的训练参数。

二、相对位置编码 (Relative Positional Encoding)

相对位置编码的核心思想是，Attention 机制的计算（Query 和 Key 的点积）更应该关注词元之间的**相对距离**，而不是它们的绝对位置。

1. 原始相对位置编码 (Relative Positional Encoding - Transformer-XL / XLNet Style)

- **工作原理：** **这类方法不修改词元嵌入本身，而是修改 Attention 机制的注意力得分计算。它引入了相对位置的嵌入或偏置，并将其加到Query 和 Key 的点积结果上（在 Softmax 之前）**。相对位置嵌入是根据 Query 位置 i 和 Key 位置 j 之间的相对距离 $i-j$ 生成或查找的。通常，为了控制参数量，会将相对距离进行裁剪 (clipping)，例如将所有大于某个阈值 K 的相对距离都视为 K，所有小于 $-K$ 的都视为 $-K$。
- **应用方式：** 修改 Attention 计算公式，例如：$Score(i,j) = \langle Q_i, K_j \rangle + \langle Q_i, R_{j-i} \rangle$

或其变种，如：$Score(i,j) = \langle Q_i, K_j + R'_{j-i} \rangle + \text{bias}_{j-i}$

其中 R 和 R' 是相对位置嵌入，bias 是相对位置偏置，它们都取决于相对距离 $j-i$。

- **特点：** 相对、通常可学习、加性（加到 Attention Score）。
- **优点：** 直接建模词元间的相对关系，有助于处理长序列和捕捉局部依赖。
- **缺点：** 修改了 Attention 计算过程，实现比加性绝对位置编码复杂；距离裁剪会损失超长距离的相对信息。

2. **T5 的相对位置偏置 (T5-style Relative Positional Bias)**

- **工作原理：** T5 模型使用的是一种简化的相对位置编码，它不使用相对位置嵌入向量，而是**使用一个与相对距离相关的标量偏置直接加到 Attention score 上**。相对距离 $(i-j)$ 会被映射到一个有限的“桶”（bucket）中，模型为每个桶学习一个标量偏置。
- **应用方式：** $Score(i,j) = \langle Q_i, K_j \rangle + B_{\text{bucket}(i-j)}$

  其中 $B_{\text{bucket}(i-j)}$ 是根据相对距离 $i-j$ 所在的桶查找到的可学习标量偏置。
- **特点：** 相对、可学习（偏置）、加性（加到 Attention Score）。
- **优点：** 相对位置编码的一种参数高效且实现简单的变体；能够建模相对位置。
- **缺点：** 将距离离散到桶中丢失了距离的精细信息；距离分桶和裁剪依然限制了对超长距离的精确建模。

3. **ALiBi (Attention with Linear Biases)**

- **工作原理：** ALiBi 也修改 Attention Score，但它**添加的是一个固定且不可学习的、与相对距离成线性关系的偏置**。对于 Query 位置 i 和 Key 位置 j，偏置通常是 $-m \times |i-j|$ 或 $-m \times (i-j)$ (对于因果模型)，其中 m 是一个固定的斜率，每个 Attention Head 使用不同的斜率。
- **应用方式：** $Score(i,j) = \langle Q_i, K_j \rangle - m|i-j|$

  这里 $|i-j|$ 表示位置 i 和位置 j 之间的绝对距离。
- **特点：** 相对、固定（不可学习）、加性（加到 Attention Score）。
- **优点：** 无需引入额外的参数，对长序列具有良好的外推能力（因为偏置是线性的，没有裁剪）；鼓励模型更多地关注附近的词元。
- **缺点：** 线性的距离衰减可能不是对所有关系都最优的建模方式；仍然是加到 Attention Score。

3.4 多头和多层

✏️ **问题：transformer的多头和多层的作用**

来源：腾讯

答案：

1. 多头注意力机制 (Multi-Head Attention)

- **核心思想:** 与其只用一组Query (Q), Key (K), Value (V) 来计算一次注意力得分，不如将Q, K, V 通过不同的线性变换（投影）映射到多个不同的“子空间”（subspace）中，在每个子空间里独立地执行缩放点积注意力（Scaled Dot-Product Attention），最后将所有子空间的注意力输出拼接起来，再进行一次线性变换得到最终的输出。
- **作用与好处:**
  - **允许模型关注来自不同子空间的不同信息:** 这是最关键的作用。单头注意力机制可能会让模型只关注某一种特定类型的关联信息（比如语法关系），或者所有信息被平均化。多头机制允许不同的“头”（head）学习关注输入序列中不同方面的信息。例如，一个头可能关注局部的语法依赖，另一个头可能关注长距离的语义关联，还有一个头可能关注词语的位置信息等。这使得模型能够更全面、更细致地捕捉输入信息中丰富的特征和依赖关系。
  - **提供更丰富的表示能力:** 通过将信息投影到不同的表示子空间，模型可以从不同的角度学习和整合信息。每个头可以看作是在不同的“表示视角”下进行注意力计算。将这些不同视角的结果结合起来，可以产生比单视角更强大、更鲁棒的特征表示。
  - **类似集成学习的效果:** 在某种程度上，多头注意力有点像集成学习（Ensemble Learning）的思想，不同的头可以看作是不同的“专家”，各自专注于序列的不同方面，最后综合它们的意见，得到更优的整体判断。
  - **稳定训练过程（间接作用）:** 将原始的高维空间分解为多个低维子空间进行计算，可能有助于稳定训练，降低模型对某些特定模式过度敏感的风险。

2. 多层结构 (Multi-Layer Stacking)

- **核心思想:** Transformer模型（无论是Encoder还是Decoder）都不是只有一个包含Multi-Head Attention和Feed-Forward Network的层，而是将这个基础的“块”（Block/Layer）堆叠多次，形成一个深层网络结构。每一层的输出作为下一层的输入。
- **作用与好处:**
  - **逐步提取和组合更复杂的特征:** 就像深度卷积网络（CNN）中，浅层学习边缘、纹理等低级特征，深层组合这些特征形成物体部件乃至整个物体一样，Transformer的多层结构也允许模型进行层次化的信息处理。
    - **底层（靠近输入的层）** 可能更关注词语本身、局部的上下文依赖关系（如短语结构）。

    - **中层** 可以在底层的基础上，开始捕捉更长距离的依赖、句法结构或简单的语义关系。
    - **高层（靠近输出的层）** 则可以整合来自中下层的信息，进行更复杂的推理，理解更高层次的语义、语篇结构、甚至是抽象概念之间的联系。
  - **增加模型容量和表达能力:** 网络的深度是模型复杂度和表达能力的关键因素。更多的层意味着模型拥有更多的参数和非线性变换，能够学习和拟合更复杂的数据模式和函数。对于需要理解复杂语言现象的任务（如机器翻译、文本生成、问答），深层结构是必不可少的。
  - **逐层精炼表示:** 每一层都可以看作是对上一层输出表示的一次“精炼”（refinement）。通过多层堆叠，信息在网络中逐层传播和处理，使得最终的表示能够充分融入上下文信息，并达到较高的抽象水平。

总结:

- **多头注意力 (Multi-Head Attention) 是在 一个层内部 通过并行地在不同子空间计算注意力，来增强模型 同时捕捉多种不同类型关联信息 的能力。它关注的是注意力的 广度 和 多样性。**
- **多层结构 (Multi-Layer Stacking) 是通过 堆叠多个层，让信息逐层传递和处理，使得模型能够 *学习从低级到高级、从简单到复杂的层次化特征*。它关注的是特征提取的 深度 和 复杂度。**

3.5 self-attention和cross-attention

✏️ 问题：讲一下Transformer中self-attention和cross-attention之间异同

来源：阿里巴巴

答案：

在encoder-decoder中，self-attention的QKV都是encoder的输入变换而来，cross-attention的KV是来自encoder的输出，Q是decoder的输入经过self-attention后的结果变换而来

类型	Q (Query) 来源	K (Key), V (Value) 来源	应用场景
Self-Attention	同一序列的输入变换	同一序列的输入变换	Encoder/Decoder内部
Cross-Attention	Decoder的当前输出变换	Encoder的最终输出变换	Encoder-Decoder交互层

1. **Self-Attention**：建立序列内部的依赖关系（如理解句子结构）
   - 相当于让句子中的每个词"回顾"整个句子，重新理解上下文
2. **Cross-Attention**：实现不同模态/语言间的对齐（如翻译中对齐源语和目标语）
   - 相当于Decoder向Encoder提问："根据我现在要生成的词，你那边哪些信息最重要？"

4. 大模型基础

4.1 为什么是decoder-only

> ✏️ **问题：多数大模型结构为什么都用decoder-only而不是encoder-only或者encoder-decoder**
>
> **来源：华为、阿里巴巴**

理论优势：

- **避免低秩问题**：Encoder的双向注意力机制容易出现低秩问题（矩阵中存在相似行列导致秩降低），而Decoder-only的因果注意力（下三角矩阵）必然满秩，理论上表达能力更强
- **预训练任务对齐**：Decoder-only的next token prediction目标直接匹配生成任务需求，而Encoder-only的MLM目标与生成任务存在偏差
- **涌现能力潜力**：在大规模训练下，Decoder-only展现出更强的零样本泛化和任务组合能力

性能优势

- **训练/推理效率**：省略编码器部分，单次前向计算即可，显著提升效率（尤其适合长序列生成）
- **Zero-shot/Few-shot表现**：Decoder-only在无标注数据场景下性能最优，而Encoder-Decoder需依赖标注数据微调
- **KV-Cache复用**：支持自回归生成时缓存历史K/V矩阵，加速多轮对话等任务

工程实现优势

- **结构简化**：单一解码器结构更易扩展（如GPT系列从1.5B到万亿参数的平滑扩展）
- **数据利用高效**：无需复杂数据清洗或任务特定编码，直接利用原始文本

4.2 采样算法

> ✏️ **问题：介绍下大模型常用的生成采样算法？**
>
> **来源：腾讯**

1. **Greedy search（贪婪搜索）**：贪婪搜索总是在**每一步选择具有最高概率的下一个词**

   数学表达：$w_t = \underset{w \in V}{\operatorname{argmax}} P(w|w_{1:t-1})$

   说明：

   - 在解码的每一步选择概率最高的词，易陷入局部最优
   - V 为词表，$w_{1:t-1}$ 是已生成序列

- 计算复杂度为 $O(V)$

2. **Beam search（束搜索）**：束搜索（Beam Search）是一种启发式搜索算法，在序列生成的每一步：保留当前得分最高的k个候选序列；对每个候选序列扩展所有可能的下一token；**在所有扩展结果中重新选择全局得分最高的k个序列**

   评分函数：$\text{Score}(y_{1:t}) = \frac{1}{t^\alpha}\sum_{k=1}^{t}\log P(y_k|y_{1:k-1})$

   说明：

   - 解码每一步保留多个候选，更接近全局最优
   - α 为长度惩罚系数（通常0.6-0.7）
   - 计算复杂度为 $O(kV)$

3. **随机采样优化：**

   a. **Top-k采样**：在每一步生成时，仅从概率最高的k个候选词中随机采样，通过固定候选数量来平衡生成质量与多样性。

      数学表达:

      i. 候选词集合定义：$V(k) = \{w|w \in top-k : P(w)\}$

      ii. 重新归一化概率：$P'(w_i) = P(w_i)/\sum P(w_j), 其中 w_j \in V(k)$

      说明：

      - 当k=1时退化为贪婪搜索
      - 当k=|V|时（V为词表大小）退化为原始概率分布

   b. **Top-p (Nucleus) Sampling（核采样）**：通过动态截断概率累计分布，仅从最可能的核心词集中采样，平衡生成质量与多样性。

      数学表达：

      i. 候选词集合定义：$V(p) = \{w_{(1)},...,w_{(k)}|\sum_{i=1}^{k}P(w_{(i)}) \geq p, k = min_m\sum_{i=1}^{m}P(w_{(i)}) \geq p\}$

      ii. 重新归一化概率：$P'(w_i) = P(w_i)/\sum P(w_j), 其中 w_j \in V(p)$

      说明：

      - 自适应候选集：根据概率分布动态调整候选词数量（相比固定K值的Top-K更灵活）
      - 质量与多样性：排除长尾低概率词（避免生成不合理内容），同时在核心概率区间内保持随机性

   c. **Temperature sampling（温度采样）**：通过调节温度参数τ控制输出分布的平滑程度，从而平衡生成文本的确定性与多样性。

数学表达： $$P'(w_i) = \frac{\exp(z_i/\tau)}{\sum_{j=1}^{V} \exp(z_j/\tau)}$$

说明：

- z_i 为logits，τ 为温度参数
- 当温度较高时，概率分布变得更加平滑，使不太可能的词汇有更高的机会被选中，增加了文本的多样性。相反，当温度较低时，模型倾向于选择概率较高的词汇。

4.3 top-k和top-p

✏️ **问题：请解释top-k和top-p在文本生成中的作用机制，并讨论它们各自的优缺点。**

来源：字节跳动、腾讯

答案：

1. **Top-k 采样**

- **作用机制:**
  a. 模型预测出下一个词的概率分布。
  b. 将所有可能的下一个词按照其预测概率从高到低排序。
  c. 选择概率最高的 **前 k 个** 词。
  d. 重新归一化这 k 个词的概率分布，使得它们的概率总和为 1。
  e. 从这重新归一化后的 k 个词的概率分布中随机采样一个词作为下一个生成的词。
- **优点:**
  - **减少生成低概率、不相关或荒谬词语的风险:** 通过只考虑最有可能的 k 个词，可以有效地过滤掉那些模型认为不太可能出现的词，从而提高生成文本的质量和连贯性。
  - **引入一定的随机性:** 即使只考虑前 k 个词，仍然是从它们的概率分布中进行采样，因此可以避免每次生成都选择概率最高的同一个词，从而增加生成文本的多样性。
  - **实现简单:** 机制相对简单，易于实现和理解。
- **缺点:**
  - **k 值的选择可能比较敏感:**
    - 如果 k 值太小，可能会导致生成过于保守和重复的文本，缺乏创造性。
    - 如果 k 值太大，可能会引入一些概率较低但仍然不太合适的词语，降低文本质量。

- **k 值是固定的，不随概率分布动态调整:** 在某些情况下，模型可能对下一个词的预测非常有信心（概率分布非常集中），此时选择较小的 k 值就足够了。而在另一些情况下，模型可能不太确定（概率分布比较分散），此时可能需要更大的 k 值才能覆盖到所有合理的选项。固定的 k 值无法适应这种动态变化。

2. **Top-p (Nucleus Sampling)**

- **作用机制:**
  a. 模型预测出下一个词的概率分布。
  b. 将所有可能的下一个词按照其预测概率从高到低排序。
  c. 从概率最高的词开始累加它们的概率。
  d. 当累加的概率之和首次超过预设的阈值 **p** (通常在 0 到 1 之间，例如 0.9)，就停止累加。
  e. 将所有累加到的词（即概率累积和小于或等于 p 的词）构成一个候选集合。
  f. 重新归一化这个候选集合中所有词的概率分布，使得它们的概率总和为 1。
  g. 从这重新归一化后的概率分布中随机采样一个词作为下一个生成的词。
- **优点:**
  - **动态调整候选词集合的大小:** Top-p 采样会根据模型的预测概率分布动态地选择候选词的数量。当模型对某个词的预测非常有信心时，概率集中的前几个词的累积概率可能很快就超过 p，此时候选集合会比较小，类似于较小的 k 值。当模型不太确定时，需要包含更多的词才能使累积概率超过 p，此时候选集合会比较大，类似于较大的 k 值。这种动态调整使得采样更灵活。
  - **通常能生成更自然和多样的文本:** 由于能够根据概率分布的形状自适应地选择候选词，Top-p 采样往往能够在保证生成质量的同时，引入更多的随机性和创造性。
- **缺点:**
  - **p 值的选择也可能需要调优:** 虽然 Top-p 具有动态调整的特性，但 p 值的选择仍然会影响生成结果。
    - 如果 p 值太小，可能会限制生成的多样性。
    - 如果 p 值太大，可能会包含一些概率较低但仍然不太相关的词语。
  - **机制相对 Top-k 稍复杂:** 理解和实现上可能比 Top-k 稍微复杂一些。

总结对比:

特征	Top-k 采样	Top-p (Nucleus Sampling) 采样
选择依据	固定数量 (k) 的最高概率词	累积概率达到阈值 (p) 的最高概率词集合
候选集大小	固定	动态变化，取决于概率分布
灵活性	较低，k 值固定	较高，能根据模型置信度动态调整

生成文本	可能保守和重复，或引入少量低概率词	通常更自然和多样
实现难度	简单	稍复杂

4.4 Prefix LM 和 Causal LM

> ✏️ **问题：讲一下prefix LM和causal LM的区别**
>
> **来源：百度、快手**

1. 基本定义

- Causal LM（自回归语言模型）
  - 特点：严格单向注意力，当前token只能看到左侧历史信息（如GPT系列）。
  - 目标：生成式任务（文本续写、对话等），通过自回归预测下一个token。
  - 别名：Decoder-only模型、传统LM。
- Prefix LM（前缀语言模型）
  - 特点：输入分为前缀（Prefix）和生成部分，前缀内token可双向注意力，生成部分严格自回归（如GLM、UniLM）。
  - 目标：同时支持理解（前缀编码）和生成任务，更接近Encoder-Decoder的灵活特性。

---

2. Attention Mask设计

类型	前缀部分（Prefix）	生成部分（Generation）	可视化示例（输入："A B C"→生成"D E"）
Causal LM	无前缀概念，全部自回归	每个token仅看左侧	[A, B, C, D, E] 的Mask： [[1,0,0,0,0], [1,1,0,0,0], [1,1,1,0,0], ...]
Prefix LM	前缀内token完全可见（双向注意力）	生成部分仅看前缀+左侧生成token	假设前缀="A B C"，生成="D E"： Prefix部分Mask： [[1,1,1,0,0], [1,1,1,0,0], [1,1,1,0,0], ...] Generation部分同Causal LM

关键区别：

- Prefix LM的前缀部分类似Encoder，允许全局信息聚合；生成部分类似Decoder，保持自回归。
- Causal LM的Mask是严格的左下三角矩阵，无任何“未来”信息泄露。

4.5 涌现能力

> ✏️ **问题：大模型的涌现能力指什么？原因是什么**
>
> **来源：阿里巴巴、Shopee**

大模型的涌现能力（Emergent Abilities）是指当**模型的规模（如参数量、训练数据量、计算量等）超过某个临界阈值时，突然展现出小规模模型所不具备的复杂能力**。这些能力并非通过显式训练获得，而是随着模型规模的扩大“自发”出现的质变现象。

- 定义：涌现能力定义为“在小模型中不存在，但在大模型中存在的能力”，且无法通过小模型的性能外推预测
- 典型表现：
  - 上下文学习（In-Context Learning）：无需微调，仅通过输入示例即可学习新任务
  - 多步推理（Step-by-Step Reasoning）：解决需多步逻辑推导的问题（如数学题）
  - 跨模态理解：如纯文本训练的模型突然能解释图像内容
  - 未训练任务的泛化：例如GPT-3未经专门训练即可翻译语言或生成代码

涌现的原因：

规模效应：当参数量（如千亿级）或训练计算量（如1025 FLOPs）达到临界点，模型内部表征能力发生非线性跃升

过参数化与Double Descent：参数远超样本量时，模型泛化能力反而提升，突破传统过拟合理论

评估假说争议：斯坦福研究指出，部分“涌现”可能是评估指标的非线性导致的统计假象（如准确率阈值效应）

1. **评价指标的非平滑性**
   - 现象：某些任务的评估指标（如准确率）设计可能对性能变化不敏感，导致模型能力在临界规模附近出现“突变”的假象。
   - 举例：若任务指标在模型性能从40%→60%时仅从1.1%→7%，看似“涌现”，实则是指标的非线性放大效应（如阈值型指标）。
2. **复杂任务与子任务的差异**
   - 现象：复杂任务由多个子任务（Sub-T）组成，每个子任务的性能随模型规模平滑增长，但整体任务的综合指标因概率叠加（如多个子任务需同时正确）呈现“涌现式跃升”。
   - 举例：

- 子任务独立概率：每个Sub-T正确率从40%→60%
- 整体任务正确率：需所有Sub-T同时正确，概率从 0.4^5=1.1% → 0.6^5=7.8%
- 结果：子任务线性改进 → 宏观任务指数级提升（看似涌现）。

4.6 tokenizer分词方法

> ✏️ **问题：了解哪些分词方法？简单讲述下各个方法的原理和过程**
>
> **来源：腾讯音乐、快手**

1. **BPE**：即字节对编码。其核心思想是从字母开始，不断找词频最高、且连续的两个token合并，直到达到目标词数。
2. **BBPE**：BBPE核心思想将BPE的从字符级别扩展到子节（Byte）级别。BPE的一个问题是如果遇到了unicode编码，基本字符集可能会很大。BBPE就是以一个字节为一种“字符”，不管实际字符集用了几个字节来表示一个字符。这样的话，基础字符集的大小就锁定在了256（2^8）。采用BBPE的好处是可以跨语言共用词表，显著压缩词表的大小。而坏处就是，对于类似中文这样的语言，一段文字的序列长度会显著增长。因此，BBPE based模型可能比BPE based模型表现的更好。然而，BBPE sequence比起BPE来说略长，这也导致了更长的训练/推理时间。BBPE其实与BPE在实现上并无大的不同，只不过基础词表使用256的字节集。
3. **WordPiece**：WordPiece算法可以看作是BPE的变种。不同的是，WordPiece基于概率生成新的subword而不是下一最高频字节对。WordPiece算法也是每次从词表中选出两个子词合并成新的子词。BPE选择频数最高的相邻子词合并，而WordPiece选择使得语言模型概率最大的相邻子词加入词表。
4. **Unigram**：它和 BPE 以及 WordPiece 从表面上看一个大的不同是，前两者都是初始化一个小词表，然后一个个增加到限定的词汇量，而 Unigram Language Model 却是先初始一个大词表，接着通过语言模型评估不断减少词表，直到限定词汇量。
5. **SentencePiece**：SentencePiece是把一个句子看作一个整体，再拆成片段，而没有保留天然的词语的概念。一般地，它把空格也当作一种特殊字符来处理，再用BPE或者Unigram算法来构造词汇表。SentencePiece除了集成了BPE、ULM子词算法之外，SentencePiece还能支持字符和词级别的分词。

4.7 投机解码

> ✏️ **问题：投机解码（Speculative Decoding）是如何工作的？它对于提升生成效率有何帮助？**
>
> **来源：阿里巴巴**

答案：

投机解码是一种加速大型语言模型（LLM）文本生成过程的技术。其核心思想是利用一个较小、速度更快的“草稿模型”（draft model）来预测接下来可能出现的多个词语（形成一个“草稿”），然后使用一个更大、更准确的“目标模型”（target model）来验证这个草稿的正确性。

以下是投机解码的详细步骤：

1. **目标模型生成初始词语：** 像传统的自回归生成一样，首先使用目标模型根据已有的上下文生成第一个词语。
2. **草稿模型预测后续词语：** 一旦目标模型生成了第一个词语，我们就将这个词语以及之前的上下文输入到速度更快的草稿模型中。草稿模型会尝试预测接下来可能出现的多个词语，形成一个候选的词语序列（草稿）。草稿模型通常比目标模型小，因此生成速度更快。
3. **目标模型验证草稿：** 接下来，我们将目标模型生成的第一词语以及草稿模型预测的整个词语序列（草稿）输入到目标模型中进行评估。目标模型会计算在给定上下文和草稿的情况下，每个词语的概率。
4. **确定接受的词语：** 通过比较草稿模型和目标模型对草稿中每个词语的预测，我们可以确定哪些词语是“被接受”的。通常，如果目标模型对草稿中某个词语的预测概率足够高（例如，高于某个阈值，或者与草稿模型的预测一致），那么这个词语就被认为是正确的并被接受。
5. **添加接受的词语到输出：** 所有被目标模型接受的词语都会被添加到最终的生成文本中。
6. **处理拒绝的词语（如果存在）：** 如果草稿中的某个词语没有被目标模型接受（即目标模型认为更有可能出现其他词语），那么我们就将目标模型在验证这个词语时预测出的最有可能的词语作为下一个实际生成的词语。
7. **重复过程：** 然后，我们以目标模型新生成的这个词语为起点，再次使用草稿模型预测后续的词语，并重复上述验证过程，直到生成所需的文本长度。

图示理解：

代码块

```
[上下文] -> 目标模型 -> [词语1]
[上下文] + [词语1] -> 草稿模型 -> [词语2，词语3，词语4]（草稿）
[上下文] + [词语1，词语2，词语3，词语4] -> 目标模型 -> 验证每个词语的概率
假设目标模型验证后接受了 [词语2] 和 [词语3]，但在 [词语4] 处认为更可能是 [词语4']。
那么最终输出为：[词语1，词语2，词语3]
下一步将以 [上下文] + [词语1，词语2，词语3] + [词语4'] 为输入，重复上述过程。
```

投机解码如何提升生成效率？

投机解码之所以能够提升生成效率，主要是因为它减少了调用计算成本更高的目标模型的次数。

- **批量生成：** 草稿模型一次性预测多个词语，相当于进行了一次“批量”生成。虽然这些预测需要被目标模型验证，但验证的成本通常比从头开始生成每个词语要低。

- **利用快速模型：** 大部分被接受的词语都来自于速度更快的草稿模型，这大大缩短了整体的生成时间。
- **减少目标模型的冗余计算：** 目标模型只需要在初始阶段生成一个词语，以及在验证草稿时进行评估，避免了对每个词语都进行完整的概率计算。

总结来说，投机解码通过引入一个快速的草稿模型来预测可能的词语序列，然后利用更强大的目标模型进行验证，从而在保证生成质量的前提下，显著提升了文本生成的效率。 这对于需要生成大量文本的应用场景（例如，长文本生成、对话系统等）尤其有价值。

4.8 LayerNorm

> ✏️ **问题：介绍下在大模型中都有哪些layerNorm的方式**
>
> **来源：阿里巴巴、腾讯**

LayerNorm 的基本思想是对每个样本独立地，在特征维度上进行归一化，使其均值接近 0，方差接近 1，然后通过学习到的缩放 $(scale, \gamma)$ 和平移 $(shift, \beta)$ 参数恢复其表达能力。

与 Batch Normalization 不同，LayerNorm 的计算不依赖于 batch size，这使其非常适合处理变长序列和用于 Transformer 这种架构。

在大模型中，LayerNorm 的变种主要体现在两个方面：**放置的位置 (Placement)** 和 **归一化的具体计算 (Calculation)**。

1. **放置位置的变种 (Placement)**

这是 LayerNorm 在 Transformer 块内最常见的变种区分。一个标准的 Transformer 子层（如 Self-Attention 或 FFN）通常包含一个子层模块、残差连接和归一化层。

- **Post-LayerNorm (Post-Norm)**
  - **结构：** 归一化层放在子层模块和残差连接**之后**。
  - **计算流程：**
    i. 输入 x
    ii. 通过子层（Attention 或 FFN）：$y = Sublayer(x)$
    iii. 添加残差连接：$z = x + y$
    iv. 应用 LayerNorm：$Output = LayerNorm(z)$
  - **特点：** 这是原始 Transformer 论文代码中采用的方式，在浅层网络中工作良好。
  - **优点：** 直观。
  - **缺点：** 在训练**非常深**的网络时，可能会导致训练不稳定，梯度容易发散或消失，因为残差路径上的信号在多次相加后可能增长过快，而归一化发生在相加之后。
- **Pre-LayerNorm (Pre-Norm)**

- **结构：** 归一化层放在子层模块**之前**，残差连接**之后**。
- **计算流程：**
  i. 输入 x
  ii. 应用 LayerNorm 到输入：$y = LayerNorm(x)$
  iii. 通过子层（Attention 或 FFN）：$z = Sublayer(y)$
  iv. 添加残差连接：$Output = x + z$
- **特点：** 许多现代 LLM 采用的方式（如 GPT-2, GPT-3, Llama 部分版本, Mistral 部分版本）。
- **优点：** 显著提高了训练**非常深**的网络的稳定性。因为归一化发生在子层输入之前，可以限制输入到子层的激活值范围，防止信号在深层网络中爆炸。
- **缺点：** 子层模块的输入是归一化过的，但残差连接跳过的原始输入 x 并未被归一化。

2. **归一化计算的变种 (Calculation)**

这是指 LayerNorm 具体如何计算均值和方差（或类似统计量）并进行归一化的公式变体。

- **标准的 Layer Normalization (Standard LayerNorm)**
  - **计算：**
    i. 计算输入 x 在特征维度上的均值 μ 和方差 σ^2。
    ii. 进行归一化：$\hat{x} = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}$
    i. 应用学习到的缩放和平移：$Output = \gamma \hat{x} + \beta$ 其中 ϵ 是一个很小的数防止除以零，γ 和 β 是与特征维度大小相同的可学习向量。
  - **特点：** 原始且最基本的 LayerNorm 计算方式。
  - **优点：** 效果稳定，理论性质好。
- **RMSNorm (Root Mean Square Layer Normalization)**
  - **计算：**
    i. **不计算均值**，只计算输入 x 在特征维度上的平方均值的平方根 (RMS)。

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

    其中 D 是特征维度的大小。
    ii. 进行归一化：$\text{Output} = \frac{x}{RMS(x)} \cdot \gamma$ (**不包含偏置参数 β**)
  - **特点：** LayerNorm 的一个简化版本，例如 Llama 系列模型就使用了 RMSNorm。
  - **优点：** 计算上比标准 LayerNorm 稍微快一些（省略了均值计算）；在大模型训练中也能提供与 LayerNorm 类似的稳定性，甚至在某些情况下表现更好。它更侧重于归一化向量的**幅度**。

- **缺点：** 放弃了均值归零，理论上可能损失一些信息（尽管在实践中通常影响不大）。

- **DeepNorm**
  - **计算/结构：** DeepNorm 是一种为训练**超深** Transformer (例如，高达 1000 层) 设计的归一化和缩放技术。它使用了 Pre-Norm 结构，但在此基础上引入了额外的缩放因子。
  - **计算流程示例：**
    i. $\hat{x} = LayerNorm(x)$ (可以是标准 LayerNorm 或 RMSNorm)
    ii. $y = Sublayer(\hat{x})$
    iii. $Output = \alpha x + \beta y$ (这里 x 是残差连接的输入，α 和 β 是特定的缩放常数，通常 α 与层数 N 相关，如 $\alpha = N^{0.25}$，β 可能设为 1 或其他值)
  - **特点：** 不仅仅是归一化，还结合了残差连接的特定缩放。
  - **优点：** 能够训练比传统 Pre-Norm 更深的 Transformer 网络。
  - **缺点：** 主要用于追求极端深度的研究，在典型的 LLM 架构中（通常数百层）不如 Pre-Norm + Standard/RMSNorm 常见。

总结 LLM 中常用的 Layer Normalization 方法：

1. **Placement: Pre-LayerNorm** 是目前主流 LLM 为了训练稳定性而广泛采用的放置方式，优于传统的 Post-LayerNorm。
2. **Calculation:**
   - **Standard LayerNorm** 仍然被许多模型使用。
   - **RMSNorm** 因其效率和在某些架构（如 Llama）中的良好表现而越来越受欢迎。

4.9 PPL困惑度

> ✏️ **问题：介绍下ppl指标的计算逻辑**
>
> **来源：阿里、美团**

PPL（Perplexity），即困惑度，是评估语言模型性能的一个常用指标。它**衡量了一个语言模型对给定文本序列预测的“困惑”程度或不确定性**。简单来说，**PPL 值越低，表明语言模型对文本的预测能力越强，模型的性能越好。**

PPL 的计算逻辑源于信息论中的交叉熵（Cross-Entropy）。对于一个给定的文本序列 $W = (w_1, w_2, ..., w_N)$，其中 N 是序列中的 token（词或子词）数量，语言模型对该序列的概率预测为 $P(W) = P(w_1, w_2, ..., w_N)$。根据链式法则，这个概率可以分解为每个 token 在给定其前导 token 的条件下的概率的乘积：

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

语言模型的目标是最大化这个概率 $P(W)$。在实践中，为了避免数值下溢（当概率值非常小时），通常会最大化对数概率 $\log P(W)$，或者等价地最小化负对数概率 $-\log P(W)$。

交叉熵衡量的是模型预测分布与真实分布之间的差异。对于语言模型，真实分布可以看作是文本数据本身，即在给定历史信息的情况下，下一个 token 就是实际出现的那个 token，其概率为 1，其他 token 的概率为 0。因此，对于一个文本序列 W，其交叉熵 $H(W)$ 可以表示为每个 token 的负对数概率的平均值：

$$H(W) = -\frac{1}{N}\sum_{i=1}^{N}\log P(w_i|w_1, w_2, ..., w_{i-1})$$

PPL 的定义与交叉熵紧密相关，它是交叉熵的指数化：

$$PPL(W) = \exp(H(W)) = \exp\left(-\frac{1}{N}\sum_{i=1}^{N}\log P(w_i|w_1, w_2, ..., w_{i-1})\right)$$

将指数运算应用到求和内部，并利用指数和对数的性质 $\exp(\log x) = x$，上述公式可以进一步写成：

$$PPL(W) = \left(\prod_{i=1}^{N}\exp(-\log P(w_i|w_1, w_2, ..., w_{i-1}))\right)^{1/N} = \left(\prod_{i=1}^{N}\frac{1}{P(w_i|w_1, w_2, ..., w_{i-1})}\right)^{1/N}$$

这个形式表明，PPL 可以被理解为每个 token 在给定前文的情况下，模型预测概率的倒数的几何平均。直观上，如果模型对下一个 token 的预测非常确定（即赋予实际出现的 token 较高的概率），那么 $P(w_i|w_1, ..., w_{i-1})$ 会接近 1，其倒数接近 1，对 PPL 的贡献就小。反之，如果模型非常不确定，赋予实际出现的 token 较低的概率，其倒数就会很大，导致 PPL 值升高。

计算步骤总结：

1. 给定一个用于评估的文本数据集。
2. 使用待评估的语言模型计算数据集中每个 token 在给定其前导 token 序列时的条件概率 $P(w_i|w_1, w_2, ..., w_{i-1})$
3. 计算每个 token 的负对数概率 $-\log P(w_i|w_1, w_2, ..., w_{i-1})$。
4. 将所有 token 的负对数概率相加，并除以 token 的总数量 N，得到平均负对数概率（即交叉熵）。
5. 对平均负对数概率进行指数化，得到最终的 PPL 值。

在实际计算中，特别是对于处理长文本的固定长度模型，通常会采用滑动窗口等技术来近似计算，以克服模型输入长度的限制。

总而言之，PPL 通过量化语言模型对测试集文本的平均预测不确定性来评估模型的性能。PPL 值越低，说明模型对文本的建模能力越强，预测越准确。

4.10 Cosine调度器

✏️ **问题：介绍下LLM训练中的Cosine调度器，以及怎么设置其周期**

来源：腾讯、MiniMax

在大型语言模型（LLM）训练中，Cosine调度器指的是Cosine Annealing Learning Rate Scheduler（余弦退火学习率调度器），**是一种调整学习率的策略，它在训练过程中按照余弦函数的形状来降低学习率。**

Cosine Annealing Learning Rate Scheduler 的含义：

- **学习率（Learning Rate）：** 在训练神经网络时，学习率决定了模型参数在每次迭代中更新的步长。一个合适的学习率对于模型的收敛速度和最终性能至关重要。
- **学习率调度器（Learning Rate Scheduler）：** 由于在整个训练过程中使用固定的学习率可能不是最优的，学习率调度器被用来动态地调整学习率。在训练初期，通常使用较大的学习率以快速接近最优解；在训练后期，使用较小的学习率以更精细地调整参数，避免震荡，帮助模型收敛到更好的局部或全局最优解。
- **余弦退火（Cosine Annealing）：** Cosine Annealing 是一种具体的学习率调度策略。它依据余弦函数周期性变化的特性，将学习率从一个较高的初始值平滑地降低到一个最小值。学习率的变化曲线看起来像余弦函数的一部分。

其基本公式通常为：

$$\eta_t = \eta_{min} + \frac{1}{2}(\eta_{max} - \eta_{min})(1 + \cos(\frac{T_{cur}}{T_{total}}\pi))$$

其中：

- η_t 是当前步骤的学习率。
- η_{min} 是学习率的最小值。
- η_{max} 是学习率的最大值（通常是初始学习率）。
- T_{cur} 是当前已经进行的训练步数或 epoch 数。
- T_{total} 是总的训练步数或一个余弦周期的总步数。

这种调度器的优点在于，它提供了一个平滑的学习率衰减过程，避免了阶梯式衰减可能导致的震荡，有助于模型更稳定地收敛。在某些变体中，Cosine Annealing 还可以包含“warm restart”（热启动），即在学习率降低到最小值后，会迅速提高学习率，然后再次按照余弦函数进行衰减，形成多个周期的变化。这有助于模型跳出局部最优解。

如何设置其周期（Period）：

Cosine Annealing 的周期设置（即 T_{total} ）是一个重要的超参数，它决定了学习率衰减的速度和模式。在 LLM 训练中，周期的设置通常有以下几种考虑：

1. **匹配总训练步数：** 最常见的设置是将一个余弦周期的长度设置为模型的总训练步数（或总 epoch 数）。这意味着学习率会从初始最大值开始，在整个训练过程中逐渐按照余弦曲线衰减到最小值。研究表明，对于许多 LLM 任务，将余弦周期的长度设置为总训练时长可以获得较好的性能。
2. **基于 Epoch 或 Step：** 周期可以基于训练的 epoch 数或总的训练步数来计算。例如，如果设置周期为 100 epoch，那么每训练 100 个 epoch，学习率会完成一个从最大值到最小值的衰减过程。
3. **多周期（Warm Restarts）：** 如果使用带有热启动的 Cosine Annealing，则需要设置每个周期的长度。例如，可以设置每 5000 步为一个周期，学习率每 5000 步就会经历一次从最大值到最小值的变化，然后重新回到最大值开始下一个周期。每个周期的长度可以相同，也可以随着训练的进行而逐渐增加。
4. **与训练数据量和计算资源相关：** 周期的设置也应该考虑到可用的训练数据量和计算资源。如果训练数据量很大，训练时间很长，一个较长的周期可能更合适。

5. 微调

5.1 SFT微调方式

> ✏️ **问题：SFT的时候，都有哪些参数微调方式？**
>
> **来源：字节跳动**

1. 全参微调：full-finetuning
2. PEFT方法：LoRA，Prompt tuning（prefix tuning/p-tuning v1 v2/Prompt Tuning），adpter-tuning

5.2 LoRA

> ✏️ **问题：介绍下LoRA的原理和实现，它有什么优势？**
>
> **来源：字节跳动、美团**

答案：

LoRA（Low-Rank Adaptation）是一种针对大型预训练模型的高效微调技术，其**核心原理是通过低秩矩阵分解减少微调时的参数量**，同时保持模型性能。以下是其核心要点：

1. **基本原理**

- 低秩矩阵分解：LoRA将预训练模型的权重矩阵 W_0 冻结，并引入两个低秩矩阵 A（维度 $d \times r$）和 B（维度 $r \times d$），其中秩 $r << d$（通常 $r = 8$）。权重更新量 ΔW 表示为 $\Delta W = AB$，最终输出为： $h = W_0 x + BAx$
  仅训练 A 和 B，**大幅减少可训练参数（从 d^2 降至 $2dr$）**

- 数学基础：基于奇异值分解（SVD）和本征维度理论，**任务相关的权重更新可通过低秩矩阵近似捕捉**

2. **实现细节**

- 参数初始化：A **初始化为高斯分布，**B **初始化为零矩阵**，确保训练初期适配层不影响原始输出
- 目标模块选择：通常作用于注意力层的查询（Q）和值（V）投影矩阵，因这些层对任务适配敏感
- 推理合并：训练后，BA 可直接加到 W_0 上，不增加推理延迟

3. **优势**

- 高效性：参数量减少至全量微调的1/10以下，显存占用显著降低
- 防止灾难性遗忘：冻结原始权重，保留预训练知识，适合小数据集场景
- 灵活性：适配层可动态加载，同一基座模型支持多任务切换

✏️ **问题：LoRA中有哪些超参数需要调整的，有哪些经验?**

来源：字节跳动、美团

答案：

1. **主要超参数**

LoRA 的核心思想是将预训练权重矩阵 W_0 的更新 ΔW 近似为一个低秩分解 BA，其中 $B \in R^{d\times r}$ 和 $A \in R^{r\times k}$ （如果 $W_0 \in R^{d\times k}$ ）。可训练的参数是 A 和 B 。

基于这个结构，主要的超参数包括：

- `r` **(lora_rank):** 这是 LoRA 的**秩**，也是最重要的超参数。它决定了低秩矩阵 A 和 B 的维度 (r) 。r 直接控制了添加到模型中的**可训练参数数量**（对于一个维度为 $d \times k$ 的权重矩阵，LoRA 添加的参数大约是 $r \times (d + k)$ ）以及 LoRA adaptation 的**表达能力**。
- `alpha` **(lora_alpha):** 这是 LoRA 的一个**缩放因子**。在将 BA 的结果加回到预训练权重 W_0 之前，会乘以一个缩放系数 $r\alpha$ 。原论文建议将缩放系数设置为 $r\alpha$ 而不是简单地用 α ，目的是让超参数 r 的调整不影响缩放的程度，从而可能简化调参。通常设置 `alpha` 等于 `r` ，使得缩放系数为 1。
- `target_modules` **:** 这是一个列表，指定了预训练模型中**哪些权重模块**需要应用 LoRA。通常是模型中的线性层（如 Attention 机制中的 Q, K, V, O 投影矩阵，或者前馈网络中的全连接层）。选择不同的模块会对性能和参数量产生显著影响。
- `dropout` **(lora_dropout):** 应用于 LoRA adapter A 的输出侧的 **dropout** 比率。这是一种正则化手段，用于防止过拟合，尤其是在数据集较小的情况下。

- `bias`：指定是否以及如何微调偏置项 (bias)。选项通常包括 `'none'` (不训练偏置)、`'lora_only'` (只训练应用了 LoRA 的层的偏置) 和 `'all'` (训练所有层的偏置)。原论文建议将偏置项保持冻结。

2. **调参经验**

以下是一些基于实践的调参经验：

- **`r` 是最重要的，从较小值开始：**
  - `r` 是决定参数效率和性能之间权衡的关键。**这是一个需要重点调优的参数。**
  - 从较小的值开始尝试是常见的策略，例如 `r` 可以设置为 8, 16, 32, 64。对于大多数任务，即使是较小的 `r` (如 8 或 16) 也能取得不错的性能，同时保持极高的参数效率。
  - 增加 `r` 通常会提高模型的性能上限，但也会增加可训练参数数量，降低 LoRA 的参数效率优势，并可能增加过拟合的风险。
- **`alpha` 的设置：**
  - **最常见的设置是将 `alpha` 设置等于 `r`**。这样缩放系数 α/r 就等于 1。这通常是一个很好的起点，简单且有效。
  - 如果想尝试不同的缩放，可以固定 `r` 并调整 `alpha`，或者固定比值 α/r （例如保持 $\alpha = 2r$ ）并调整 r 。
  - 一些研究表明，较大的 `alpha` （相对于 `r` ）有时可能有助于性能，但这需要通过实验验证。但通常 `alpha=r` 是最省心的选择。
- **`target_modules` 的选择：**
  - **Attention 机制中的 Query (`q_proj`) 和 Value (`v_proj`) 投影矩阵**通常是应用 LoRA 的**首选目标**，因为它们在 Attention 计算中扮演着关键角色，并且它们的权重矩阵维度通常较大。原 LoRA 论文就主要在这两个模块上应用 LoRA。
  - 添加 Key (`k_proj`) 和 Output (`out_proj`) 投影矩阵通常也能带来性能提升，这也是很多实现中的默认选项。尽管这会使参数量翻倍（从 2 个 LoRA 矩阵对到 4 个），但总参数量仍然远小于全量微调。
  - 是否在 MLP（前馈网络）层（如 `fc1`, `fc2` 等）应用 LoRA 取决于任务和模型。在 MLP 层应用 LoRA 会显著增加参数量，但对于需要模型学习更复杂、非线性变换的任务可能有效。需要权衡参数效率和潜在的性能增益。
  - 建议从 `q_proj` 和 `v_proj` 开始，然后尝试添加 `k_proj` 和 `out_proj`，最后根据需要考虑 MLP 层。具体的模块名称取决于你使用的模型架构（例如 Llama、BERT、T5 等有不同的层命名约定）。
- **`dropout` 的使用：**
  - 在数据集较小或有观察到过拟合迹象时，考虑启用 LoRA dropout。
  - 一个常用的起始值是 0.05 或 0.1。可以根据验证集上的表现来调整。

- 如果数据集很大且不容易过拟合，可以不使用 dropout (设置为 0)。

- **`bias` 的处理：**
  - **通常建议保持偏置项冻结 (`bias='none'`)。** 这是原论文的推荐，也是实践中常见的做法。
  - 原因在于偏置项的参数数量相对较少，并且它们本身可能就具有较低的“有效秩”，通过 LoRA 额外去学它们的低秩更新带来的收益可能不大，甚至可能干扰训练。
  - 只有在验证确实冻结偏置会损害性能的情况下，才考虑训练偏置，例如设置为 `'lora_only'` 。
- **学习率 (Learning Rate)：**
  - 虽然不是 LoRA 独有的超参数，但学习率对于训练 LoRA 矩阵至关重要。
  - 由于只更新少量参数，通常可以使用比全量微调**更高**的学习率，这有助于 LoRA 矩阵的快速收敛。
  - 需要通过实验来找到适合的学习率，常用的优化器是 AdamW。
- **训练轮数 (Epochs) / 步数 (Steps)：**
  - LoRA 通常比全量微调收敛更快，因此可能只需要较少的训练轮数。
  - 监控验证集上的性能来决定何时停止训练是一个好习惯。

总结调参流程建议：

1. **确定 `target_modules`：** 先从 `q_proj` 和 `v_proj` 入手，这是最基础且高效的选择。如果需要更高性能，逐渐增加 `k_proj`，`out_proj` ，最后考虑 MLP 层。
2. **设置 `alpha`：** 通常直接设置为等于 `r` (即 α/r=1)。
3. **调整 `r`：** 从较小的值开始尝试（如 8, 16），观察性能。如果性能不足，逐步增加 `r` (如 32, 64)。
4. **处理 `bias`：** 默认设置为 `'none'` (冻结)。
5. **考虑 `dropout`：** 如果数据集小或有过拟合迹象，尝试启用并从 0.05 或 0.1 开始调整。
6. **调整学习率：** 找到适合你的任务和模型的学习率，可能比全量微调的学习率要高。
7. **训练和验证：** 监控训练过程和验证集上的指标，根据表现迭代调整上述超参数。

5.3 QLoRA

> ✏️ **问题：QLoRA了解吗，讲一下原理**
>
> **来源：百度、小红书**

回答：

核心原理

QLoRA（Quantized Low-Rank Adaptation）是LoRA的量化升级版本，通过**4-bit量化+低秩适配**实现大模型的高效微调，其技术框架包含三大创新：

1. **4-bit NormalFloat（NF4）量化：**（模型本身用4bit加载，训练时把数值反量化到bf16后进行训练）

采用信息论最优的4-bit数据类型，对预训练权重 W_0 进行分块量化：$W_{\text{NF4}} = \text{Quant}_{\text{NF4}}(W_{\text{FP16}})$

   - 关键特性：
     - 基于分位数量化（Quantile Quantization），针对神经网络权重的正态分布特性优化
     - 相比标准4-bit量化，NF4在相同位数下保留更多信息（理论提升约10%精度）

2. **双重量化（Double Quantization）**：对量化常数再次量化，进一步节省存储空间：

$$\text{DQ}(W) = \text{Quant}_{\text{INT8}}(\text{Quant}_{\text{NF4}}(W))$$

3. **低秩适配器**：在量化模型上叠加可训练的LoRA适配器：$h = \text{Dequant}(W_{\text{NF4}})x + \alpha \cdot BAx$
   - 改进点：
     - 缩放系数 $\alpha = \frac{r}{\text{rank}}$ 动态调整适配强度
     - 低秩矩阵维度 $A \in \mathbb{R}^{d\times r}$，$B \in \mathbb{R}^{r\times d}$（通常 $r = 8$）

实现细节

1. **分块量化（Block-wise Quantization）**：抑制离群值影响，提升量化稳定性
   - 将权重矩阵划分为64参数/块的独立单元：$W_{\text{block}} = [W_1, ..., W_k]$，　每块$W_i \in \mathbb{R}^{64}$
2. **分页优化器（Paged Optimizers）**：
   - 利用NVIDIA统一内存特性，在GPU显存不足时将优化器状态临时卸载到CPU内存
3. **全线性层适配**：
   - 在所有全连接层插入LoRA适配器（传统LoRA仅作用于Q/V矩阵），弥补量化带来的精度损失：

$$\text{Adapter}_{\text{QLoRA}} = \bigcup_{i=1}^{L}(A_i, B_i)$$

😊 **问题：Qlora技术是如何实现模型压缩的？具体来说，4bit量化应用于哪些部分？Lora矩阵初始化及量化过程是怎样的？**

来源：小米

回答：

QLoRA 如何实现模型压缩?

QLoRA 实现模型压缩（更准确地说是**显存压缩以支持微调**）主要通过以下几个关键技术：

1. **4-bit NormalFloat (NF4) 量化**：
    - 这是 QLoRA 的核心。它将预训练模型的**权重**（通常是 `nn.Linear` 层的权重）从标准的16位浮点数（FP16/BF16）或32位浮点数（FP32）量化到4位。
    - NF4 是一种信息论上最优的数据类型，专门针对正态分布的数据设计。研究发现，预训练模型的权重通常近似服从零均值的正态分布，因此 NF4 非常适合。
    - 与传统的4位整数量化（INT4）不同，NF4 的量化级别不是均匀分布的，而是根据标准正态分布的分位数来确定的，这使得它能更好地保留原始权重的信息。
    - 量化过程是逐块（block-wise）进行的，例如每64个权重值为一个块，为每个块计算一个缩放因子（通常是该块内权重的绝对值的最大值，即 absmax）。
2. **双重量化 (Double Quantization, DQ)**：
    - 在对模型权重进行4-bit NF4量化后，会产生许多量化常数（即每个块的缩放因子）。这些缩放因子本身也会占用显存。
    - 双重量化就是对这些量化常数（第一层量化的缩放因子）**再次进行量化**。例如，可以将32位的缩放因子量化到8位。
    - 这进一步减少了显存占用，平均每个参数可以节省约0.37位。
3. **Paged Optimizers (分页优化器)**：
    - 当模型梯度在反向传播过程中产生尖峰，导致显存不足时，分页优化器利用了NVIDIA统一内存（Unified Memory）的特性，自动将一部分优化器状态（Optimizer States）从GPU显存分页到CPU内存，然后在需要时再调回GPU。
    - 这虽然不是直接的模型权重压缩，但它解决了微调大模型时因优化器状态占用大量显存而导致的OOM（Out of Memory）问题，使得在有限显存下训练更大的模型成为可能。

总结来说，QLoRA 通过 NF4 将预训练模型权重压缩到4位，通过双重量化压缩这些量化常数，从而极大地降低了模型在显存中的大小。LoRA 适配器本身则以较高精度（如BF16）进行训练和存储。

4-bit量化应用于哪些部分?

4-bit NF4 量化主要应用于：

- **预训练模型的基础权重 (Base Model Weights)**：
    - 特指模型中主要的参数密集型层，如Transformer模型中的**线性层 (Linear Layers)** 的权重矩阵。这包括自注意力机制中的查询（Q）、键（K）、值（V）投影矩阵，以及输出投影矩阵；还有前馈神经网络（FFN）中的线性层。
    - 这些权重在微调过程中是**冻结的**，它们以4-bit NF4格式存储在显存中。
    - 当进行前向或反向传播计算时，这些4-bit权重会被**动态地反量化** (de-quantized) 回到计算精度（通常是BF16或FP16），参与运算。计算完成后，它们在显存中仍然保持4-bit。

不进行4-bit量化的部分：

- **LoRA 适配器矩阵 (A 和 B)**：LoRA的低秩分解矩阵 `A` 和 `B` 是在微调过程中需要训练的参数。它们通常以**较高精度**（如BF16或FP16）存储和更新，以保证微调的效果。它们参数量很小，所以对整体显存影响不大。
- **模型的其他参数**：如 LayerNorm 的参数、偏置项（biases）等，通常也不进行4-bit量化，因为它们参数量相对较小，且量化可能对性能影响较大。
- **激活值 (Activations) 和 梯度 (Gradients)**：在计算过程中，激活值和梯度通常使用BF16或FP16进行。

LoRA矩阵初始化及量化过程是怎样的?

LoRA 矩阵初始化：

LoRA 的核心思想是在预训练模型的权重矩阵 W_0 旁边并行添加一个低秩路径 BA，其中 W_0 保持冻结。微调时只更新 A 和 B。
$h = W_0x + s * BAx$ (s 是缩放因子 $alpha/r$)

1. **矩阵 A (W_A)**：
   - 通常使用**Kaiming均匀分布（或高斯分布）** 进行初始化。这是一种常用的神经网络权重初始化方法，有助于在训练初期保持梯度的稳定传播。
   - 其维度是 $d \times r$，其中 d 是输入维度，r 是秩 (rank)。
2. **矩阵 B (W_B)**：
   - 通常**初始化为零**。
   - 其维度是 $r \times k$，其中 r 是秩，k 是输出维度。
   - 将 B 初始化为零的目的是为了确保在微调开始时，LoRA分支的输出 BAx 为零。这意味着模型的行为与原始预训练模型完全一致 ($h = W_0x$)。随着训练的进行，B 的参数会逐渐从零开始学习，LoRA分支才会开始影响模型的输出。
3. **缩放因子 (alpha)**：
   - LoRA的输出 BAx 通常会乘以一个缩放因子 $alpha/r$，其中 r 是秩，$alpha$ 是一个超参数（例如，可以设置为 r，则缩放因子为1；或者设为 $2r$ 等）。$alpha$ 也需要初始化，通常是一个固定的值。

LoRA矩阵本身在QLoRA中不进行4-bit量化。它们以训练精度（如BF16）进行存储和更新。

预训练权重的量化过程 (以QLoRA为例)：

前面提到，QLoRA的量化主要针对预训练模型的**基础权重**。这个过程如下：

1. **选择目标层**：通常是模型中的所有线性层（`nn.Linear`）。

2. **分块 (Blocking)**：将权重矩阵 W_0 分成若干个块（block）。例如，每64个元素作为一个块。
3. **计算缩放因子 (Quantization Constant)**：
    - 对于每个块，计算其绝对值的最大值 (absmax scaling)。这个absmax值就是该块的缩放因子。
    - `W_normalized = W_block / absmax_block`
4. **NF4 量化**：
    - 将归一化后的权重 `W_normalized` （此时其值在 [-1, 1] 区间）映射到预定义的NF4量化级别。
    - NF4有16个量化级别（2^4=16），这些级别是根据标准正态分布的分位数确定的，不是均匀间隔的。
    - 每个归一化后的权重值被映射到离它最近的NF4级别。
    - 存储时，每个权重值只需要4位来表示其所属的NF4级别索引。
5. **双重量化 (Optional but common in QLoRA)**：
    - 步骤3中计算出的所有缩放因子（absmax_block）本身也会占用显存。
    - 对这些缩放因子再次进行量化。例如，将32位的浮点缩放因子量化为8位的浮点数（如FP8）或自定义的低比特格式。
    - 这一步也需要计算第二级量化的缩放因子。

在训练/推理时的使用：

- 当需要使用某个权重块进行计算时（如矩阵乘法 W_0x），首先加载该块的4-bit NF4量化值和对应的（可能经过双重量化后反量化回来的）第一级缩放因子。
- **反量化 (Dequantization)**：将4-bit NF4值和缩放因子结合，恢复出近似的BF16（或FP16）权重值。
    - `W_approx_BF16 = NF4_value_as_float * absmax_block`
- 使用这个反量化后的BF16权重进行前向和反向传播计算。
- 重要的是，原始的4-bit NF4权重和（可能被量化的）缩放因子在显存中保持不变，只有LoRA适配器的权重 `A` 和 `B` 会在训练中被更新。

通过这种方式，QLoRA在微调大型模型时，其基础模型部分仅占用极少的显存（因为权重是4-bit的），而可训练的LoRA参数量也很小，从而使得在单张消费级GPU上微调数十亿甚至上百亿参数的大模型成为可能。

5.4 Prefix Tuning

✏️ **问题：讲一下Prefix tuning具体是怎么做的?**

来源：拼多多、字节跳动

Prefix Tuning 是在输入token之前构造一段任务相关的virtual tokens作为Prefix，然后训练的时候只更新Prefix部分的参数，而Transformer中的其他部分参数固定。该方法其实和构造Prompt类似，只是Prompt是人为构造的“显式”的提示,并且无法更新参数，而Prefix 则是可以学习的“隐式”的提示。同时，为了防止直接更新Prefix的参数导致训练不稳定的情况，在Prefix层前面加了MLP结构

核心原理

Prefix Tuning通过向预训练模型的输入序列前添加可训练的连续向量（称为Prefix），实现任务适配。其数学形式为：$h = \text{Transformer}([P_\theta; x])$

- $P_\theta \in \mathbb{R}^{l \times d}$ 长度为 l 的Prefix矩阵，d 为隐藏层维度
- x 为原始输入序列
- 只有 P_θ 的参数 θ 参与训练，Transformer参数冻结

技术实现

1. Prefix构造方式：
   - 自回归模型（如GPT）：$z = [P_\theta; x; y]$ $(\text{Prefix}+\text{输入}+\text{输出})$
   - 编码器-解码器模型（如T5）：$z = [P_\theta^{\text{enc}}; x; P_\theta^{\text{dec}}; y]$ $(\text{Encoder/Decoder双前缀})$
2. 重参数化技巧：

直接优化 P_θ 易导致训练不稳定，通过MLP进行参数映射：$P_\theta = \text{MLP}(P_{\text{init}})$

- P_{init} 为随机初始化的小矩阵（如 $\mathbb{R}^{l \times 64}$ ）
- 训练完成后丢弃MLP，仅保留 P_θ

3. 分层注入：

在Transformer每一层都注入Prefix（而不仅限输入层）：$h_i = \text{Attention}([P_\theta^i; x^i])$

实现示例：

代码块

```
from peft import PrefixTuningConfig, get_peft_model

config = PrefixTuningConfig(
    task_type="CAUSAL_LM",
    num_virtual_tokens=20,  # Prefix长度l
    prefix_projection=True  # 是否启用MLP重参数化
)
model = get_peft_model(base_model, config)
```

5.5 Prompt Tuning

> ✏️ **问题：了解Prompt tuning吗？它和prefix tuning的区别是什么？**
>
> **来源：拼多多、字节跳动**

该方法可以看作是Prefix Tuning的简化版本，和prefix在每层都加不一样，只在输入层加入prompt tokens，并不需要加入MLP进行调整来解决难训练的问题，主在T5预训练模型上做实验。作者做实验说明随着预训练模型参数量的增加，Prompt Tuning的方法会逼近 Fine-tune 的结果。

方法	核心机制	参数更新范围
Prompt Tuning	在输入层添加可学习的连续提示向量（soft prompts），模型其余部分冻结	仅优化提示向量 $P_\theta \in \mathbb{R}^{l\times d}$
Prefix Tuning	在每一层Transformer的Key-Value序列前插入可学习向量，通过MLP重参数化生成Prefix	优化每层的Prefix矩阵 $P_\theta i \in \mathbb{R}^{l\times 2d}$

- 作用位置：
  - Prompt Tuning仅修改输入嵌入（第一层）
  - Prefix Tuning在每一层注意力层的Key和Value序列前插入Prefix（影响所有层）
- 参数量：
  - Prefix Tuning参数更多（每层独立Prefix），Prompt Tuning仅需一组全局提示

实现示例：

Prompt Tuning（HuggingFace PEFT）

```
from peft import PromptTuningConfig, get_peft_model
config = PromptTuningConfig(task_type="SEQ_CLS", num_virtual_tokens=10)
model = get_peft_model(model, config)  # 仅优化提示向量
```

Prefix Tuning（HuggingFace PEFT）：

```
from peft import PrefixTuningConfig, get_peft_model
config = PrefixTuningConfig(task_type="CAUSAL_LM", num_virtual_tokens=20,
prefix_projection=True)
model = get_peft_model(model, config)  # 优化多层Prefix + MLP
```

5.6 混合精度微调

✏️ **问题：在混合精度微调过程中，具体是指哪几种精度的混合使用？为什么选择这种方式来提高性能或节省资源？**

来源：字节跳动、京东

答案：

在混合精度微调过程中，通常混合使用的主要精度是：

- **单精度浮点数 (FP32)：** 这是传统的训练和微调中使用的标准精度。它提供较高的数值精度，能够更准确地表示模型参数和计算过程中的数值。
- **半精度浮点数 (FP16) 或 BFloat16 (BF16)：** 这两种是较低精度的浮点数格式。
  - **FP16 (Half-Precision Floating Point):** 占用 16 位存储空间，相比 FP32 的 32 位，内存占用减少了一半。在支持 FP16 计算的硬件上（如 NVIDIA 的 Tensor Cores），计算速度可以显著提升。
  - **BF16 (Brain Floating Point):** 也是占用 16 位存储空间，但其指数位比 FP16 多，尾数位比 FP16 少。这使得 BF16 在处理梯度等动态范围较大的数值时，更不容易发生溢出，尤其是在训练大型模型时表现更好。

为什么选择这种混合使用的方式来提高性能或节省资源？

选择混合精度微调的主要原因在于以下几点：

1. **提高计算速度：** 现代深度学习硬件（如GPU和TPU）通常对低精度浮点数运算进行了优化，例如 NVIDIA 的 Tensor Cores 和 Google 的 TPU 都能够以更高的吞吐量执行 FP16 或 BF16 的矩阵乘法和卷积等操作。通过将模型中的部分计算（尤其是前向传播和反向传播中的矩阵运算）切换到低精度，可以显著缩短训练和微调的时间。
2. **减少内存占用：** 使用 FP16 或 BF16 可以将模型参数、梯度和激活值的内存占用减少一半。这使得在相同的硬件条件下，可以训练或微调更大的模型，或者使用更大的批次大小，从而可能提高模型的性能。
3. **降低功耗：** 由于计算量的减少和内存访问的减少，使用低精度还可以降低硬件的功耗，这对于大规模训练和部署非常重要。

具体来说，混合精度微调通常采取以下策略：

- **以低精度存储和计算大部分张量：** 模型参数、激活值和梯度通常会以 FP16 或 BF16 的格式存储和进行计算，以利用其速度和内存优势。
- **以高精度维护关键信息：** 为了保证模型的数值稳定性，一些关键的操作或变量可能会保留在 FP32 精度下，例如：
  - **模型参数的更新：** 为了避免在多次低精度更新后可能出现的精度损失，模型参数的更新（例如使用优化器进行更新）通常在 FP32 精度下进行。
  - **累积梯度：** 在某些情况下，为了更准确地累积梯度，可能会使用 FP32 精度。

- **某些对精度敏感的层或操作。**

通过巧妙地混合使用不同的精度，混合精度微调能够在不显著损失模型性能的前提下，大幅提升训练和微调的效率，并降低资源消耗，因此成为现代深度学习中广泛采用的技术。

6. 推理加速

6.1 vLLM

7. 量化压缩

7.1 FP16精度的数据转换为int4格式

> ✏️ **问题：如果需要将FP16精度的数据转换为int4格式，如何设计流程以确保最小化信息损失？可以分享一下您的思路吗？**
>
> **来源：京东、百度**

答案：

1. **数据分布分析**

在进行量化之前，首先需要对FP16数据的分布进行分析。不同的数值范围和频率将影响如何将它们映射到INT4范围。

- **统计分析**：检查数据的均值、方差、最小值和最大值等。了解数据集中哪些值出现频率较高，哪些值可能是极端值。
- **分布归一化**：如果数据范围较广，可以先对数据进行归一化处理，将其缩放到0到1或-1到1之间，这样有助于提高映射的效率。

2. **量化函数设计**

量化是将FP16数据映射到INT4范围的关键步骤。为了减少信息损失，可以使用以下策略：

- **线性量化**：直接将FP16数据线性映射到INT4范围，通常通过比例因子缩放数据，然后进行舍入处理。例如：

$$\text{int4\_value} = \text{round}\left(\frac{\text{FP16\_value}}{\text{max\_FP16}} \times 15\right)$$

- 这里的 `max_FP16` 是FP16数据中的最大值， `round()` 是标准的舍入函数，将数值转换为最近的整数。
- **非线性量化（对数量化）**：对数量化可以通过对数据应用对数函数来增加小值的精度。例如：

$$\text{quantized} = \text{round}\left(\frac{\log(1+\text{FP16\_value})}{\log(1+\text{max\_FP16})} \times 15\right)$$

- 这种方法在数据值分布较为不均时，能够更好地保留小值的精度。
- **分段量化**：根据数据分布的特性，可以划分几个区间，对不同区间的值使用不同的量化策略。例如，对于频繁出现的小值采用更高精度的映射，对于大值则采用较低精度的映射。

3. **信息损失优化**

- **最小化误差**：在量化时，通过最小化量化误差（例如，最小化平方误差）来优化量化过程。可以使用一些优化算法，如梯度下降，来最小化转换后的数值与原始FP16值之间的差异。
- **校准**：量化后可以进行校准，调整量化系数，使得转换后数据的误差最小化。这一步可以通过计算原始数据和量化数据之间的误差，并调整映射参数来实现。

4. **溢出与下溢处理**

在将数据从FP16转换到INT4时，可能会发生溢出或下溢现象。为了避免这些问题，可以采取以下策略：

- **动态范围缩放**：如果数据超出了INT4能够表示的范围，可以通过动态调整量化过程中的范围来避免溢出。
- **剪裁（Clipping）**：对于超出范围的值，可以采用剪裁（clip）操作，强制将这些值限定在INT4的表示范围内。虽然这种方法可能会丢失一些信息，但可以保证没有溢出。

5. **逆变换与验证**

转换后的数据需要经过验证，确保精度损失在可接受范围内。可以使用以下方法：

- **反量化**：将INT4数据反量化回FP16，然后与原始FP16数据进行对比，计算误差并评估信息损失。
- **可视化和统计分析**：通过可视化转换前后的数据分布，检查是否存在显著的失真，并使用统计方法（如均方误差、平均绝对误差等）量化误差。

6. **硬件加速与优化**

如果目标是将这种转换过程部署到硬件上（如ASIC或GPU），则需要考虑硬件加速的优化策略。

- **定制量化策略**：根据硬件特性（例如，支持的操作类型或处理单元），优化量化算法以加速计算。
- **混合精度**：在某些情况下，可以考虑将计算过程分为多个阶段，某些部分使用INT4格式，其他部分仍使用FP16或更高精度的数据格式，从而平衡速度和精度。

8. 多模态大模型

8.1 多模态基础

8.2 BLIP

✏️ **问题：请简述 BLIP2 模型两阶段预训练策略的具体步骤及其目的**

来源：OPPO、VIVO

回答：BLIP2模型两阶段预训练策略包括视觉-语言表示学习阶段和视觉到语言生成学习阶段，具体步骤及其目的如下：

第一阶段：视觉-语言表示学习阶段：

- **步骤**：将Q-Former连接到冻结的图像编码器，使用图像-文本对进行预训练，联合优化三个预训练目标，**包括图像-文本对比学习（ITC）、基于图像的文本生成（ITG）和图像-文本匹配（ITM）。**在这一过程中，通过在查询和文本之间采用不同的注意力掩码策略，来控制图像转换器和文本转换器的交互方式。
  - **ITC**：对齐图像转换器输出的查询表示与来自文本转换器输出的文本表示。计算每个查询与文本嵌入之间的相似度，选择最高的作为图文相似度，采用单模态自注意掩码，避免信息泄漏。
  - **ITG**：给定输入图像作为条件，训练Q-Former生成文本。由于Q-Former架构不允许冻结的图像编码器和文本标记直接交互，生成文本所需信息先由查询提取，再通过自注意力层传递给文本标记。此过程采用多模态因果注意力掩码，query可相互感知但看不到text token，每个text token能感知所有query及其前面的text标记，并将(cls)标记替换为(dec)标记指示解码任务。
  - **ITM**：进行图像和文本表示之间的细粒度对齐，训练一个二分类任务，判断图像-文本对是正匹配还是负匹配。将图像转换器输出的每个query嵌入输入到二类线性分类器中获得logit，平均所有logit计算匹配分数，使用双向自注意掩码，让所有query和text相互感知。
- **目的**：**强制Q-Former学习与文本最相关的视觉表示，使模型能够从图像中提取出与文本描述相对应的特征，从而实现视觉和语言模态之间的初步对齐，为后续的生成学习阶段打下基础。**

第二阶段：视觉到语言生成学习阶段：

- **步骤**：将Q-Former（附带冻结的图像编码器）连接到冻结的LLM。使用全连接层将输出的query embedding线性投影到与LLM的text embedding相同的维度，然后将投影的query embedding添加到输入text embedding前面。对于基于解码器的LLM，使用语言建模损失进行预训练；对于基于编码器-解码器的LLM，基于前缀语言建模损失进行预训练。
- **目的**：**训练Q-Former，使其输出的视觉表示可以被LLM解释，利用LLM的强大语言生成能力，将第一阶段学习到的视觉-语言表示进一步转化为能够生成自然语言文本的能力，从而实现从视觉输入到语言输出的生成功能，以完成各种视觉-语言任务，如视觉问答、图像描述等。**

9. 强化学习

9.1 PPO 、DPO、GRPO

✏️ **问题：讲一下PPO 、DPO、GRPO 的主要思想、优缺点。**

来源：字节跳动、百度

PPO（近端策略优化）

- **主要思想**：PPO是一种基于策略梯度的强化学习算法，通过限制策略更新的步长来稳定训练过程。它使用裁剪机制，即通过裁剪策略更新的幅度，避免过大的更新导致训练不稳定。其目标函数包含策略梯度项、裁剪机制和KL惩罚项，以防止策略偏离初始模型过远。
- **优点**：
  - 训练稳定，适用于多种任务。
  - 计算效率高，易于实现。
  - 通用性强，适用范围广，在稳定性和样本效率之间取得了良好的平衡。
- **缺点**：
  - 在某些复杂任务中可能需要大量调参。
  - 需要额外的价值函数模型，导致计算资源消耗大。
  - 在超大规模模型训练中容易出现数值不稳定和收敛速度慢的问题

DPO（直接偏好优化）

- **主要思想**：DPO是一种直接优化人类偏好的方法，通过显式建模人类偏好来训练模型，而不需要显式的奖励模型。它直接使用人类偏好数据（如成对比较）来优化模型，通过对比正负样本的偏好差异，直接调整策略模型的输出概率分布。
- **优点**：
  - 训练更简单，计算成本更低。
  - 能够直接对齐人类偏好。
  - 易于实现，训练稳定。
- **缺点**：
  - 依赖高质量的人类偏好数据。
  - 在复杂推理任务（如数学问题）中表现较差。
  - 难以处理多步推理的细粒度奖励信号。

GRPO（群体相对策略优化）

- **主要思想**：GRPO是一种通过群体内样本的相对比较来优化策略的方法，利用群体整体的表现差异指导模型学习，而非依赖单一样本或显式奖励模型。它将样本划分为不同群体，通过群体间的相对表现差异调整策略。
- **优点**：
  - 更高效地捕捉群体偏好，降低对单一高质量数据的依赖。
  - 训练过程相对稳定，适合复杂任务中的多目标优化。
  - 计算成本可能低于需要迭代训练奖励模型的方法（如RLHF）。
  - 内存效率高，有效训练大型语言模型进行推理任务，通过移除Critic简化了强化学习过程。
- **缺点**：
  - 群体划分需要合理设计，不当分组可能导致优化偏差。
  - 对群体数据的覆盖性和多样性要求较高。
  - 需多阶段训练解决初期生成质量问题

9.2 PPO优势函数

✏️ **问题：为什么PPO算法选择利用优势函数而非直接奖励来进行评估？两者之间的主要区别是什么？**

来源：腾讯、腾讯音乐

答案：

核心原因：降低方差，提供更稳定的学习信号

PPO 是一种策略梯度 (Policy Gradient)算法。这类算法的目标是调整策略 (Policy)，使得能够获得更高累积回报的动作 (Action) 被选择的概率增加，而导致较低累积回报的动作被选择的概率降低。为了判断一个动作是“好”还是“坏”，我们需要一个评估信号。

1. **为什么不用直接奖励 (Reward)?**

- **短视性：**单步奖励 R_{t+1} 只反映了执行动作 a_t 后立即获得的好处。它完全忽略了动作的长期影响。一个动作可能立即获得高奖励，但却导致未来进入很差的状态；反之亦然。只基于即时奖励更新策略是非常不稳定的，并且难以学习到需要长期规划的任务。
- **高方差：** 环境的随机性 (Stochasticity) 和奖励本身的波动性会导致即时奖励非常不稳定。如果仅根据单步奖励来调整策略，策略的更新方向也会非常不稳定，学习过程会很慢且容易发散。

2. **为什么不用 Q 值 (Q-value)?**

- Q 值 ($Q^\pi(s,a)$): 代表在状态 s 执行动作 a 后，遵循当前策略 π 所能获得的期望累积折扣奖励。它确实考虑了长期影响，比单步奖励好得多。
- **仍然存在基线问题和高方差：**虽然 Q 值包含了长期信息，但它的绝对数值可能很高或很低，并且在不同状态下差异很大。例如，在某个状态 s 下，所有可能动作的 Q 值可能都非常高（比如都大于 100），即使其中一个动作比其他动作好很多（比如 Q 值为 110，其他为 100）。此时，梯度的大小会受到 Q 值绝对大小的影响，而不是动作之间的相对好坏。这依然会导致较高的方差。想象一下，如果所有动作都得到正反馈（即使有好有坏），这会给学习带来困扰。

3. **为什么使用优势函数 (Advantage Function)?**

- 优势函数 ($A^\pi(s,a)$): 定义为 $A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$
  - $Q^\pi(s,a)$ 是在状态 s 执行动作 a 的价值。
  - $V^\pi(s)$ 是状态 s 本身的价值（即在状态 s 下遵循策略 π 的期望累积折扣奖励，是该状态下所有动作 Q 值的期望值： $V^\pi(s) = E_{a\sim\pi(\cdot|s)}[Q^\pi(s,a)]$)
- **引入基线，关注相对优势: 优势函数衡量的是：在状态 s 下，执行动作 a 比平均而言在该状态下遵循当前策略要好多少。**它减去了状态价值 $V^\pi(s)$ 这个基线。
- **显著降低方差:** 通过减去基线 $V^\pi(s)$ ，优势函数将评估信号中心化了。如果一个动作比平均水平好，优势值为正；如果比平均水平差，优势值为负。这大大降低了评估信号的方差。因为我们不再关心动作的绝对价值有多高，只关心它相对于这个状态的平均水平是更好还是更差。这使得策略梯度的估计更加稳定，学习过程更快、更可靠。

- **更清晰的信号:** 正的优势值清晰地指示应该增加这个动作的概率，负的优势值指示应该减少这个动作的概率。

主要区别总结

- **奖励 (Reward R_{t+1}):**
  - **性质:** 即时反馈，只衡量一步的好坏。
  - **优点:** 计算简单，直接来自环境。
  - **缺点:** 短视，忽略长期影响，方差高。
- **Q 值 ($Q^{\pi}(s,a)$):**
  - **性质:** 长期价值，衡量在状态 s 执行动作 a 的期望累积回报。
  - **优点:** 包含长期信息。
  - **缺点:** 绝对值可能很大且波动，存在基线问题，方差相对较高。
- **优势函数 ($A^{\pi}(s,a)$):**
  - **性质:** 相对价值，衡量动作 a 相对于状态 s 平均动作的好坏程度 ($A = Q - V$)。
  - **优点:** 包含长期信息，通过减去基线显著降低方差，提供更稳定、清晰的学习信号。
  - **缺点:** 需要估计状态价值 $V^{\pi}(s)$ （通常通过值函数网络来估计）。

在实践中，PPO 通常使用广义优势估计 (Generalized Advantage Estimation, GAE) 来计算优势函数 $\hat{A}_t$ 。GAE 是一种在偏差和方差之间进行权衡的技术，可以进一步提高优势函数估计的质量。

总而言之，**PPO选择优势函数是因为它提供了一个低方差且关注动作相对好坏的评估信号，这对于稳定有效地更新策略至关重要**，从而实现更快速、更可靠的强化学习。

9.3 PPO算法

> ✏️ **问题：知道什么是on-policy和off-policy吗，PPO属于其中哪一种?**
>
> **来源：腾讯、阿里**

PPO（Proximal Policy Optimization）算法本质上是on-policy，但通过技术改进（如重要性采样和裁剪机制）部分借鉴了off-policy的特性，使其在实践中有一定的灵活性。以下是详细分析：

1. **核心结论**

- **PPO的原始设计是on-policy**：
  数据必须由当前策略（或接近当前策略的旧策略）生成，且每次策略更新后需重新采样数据（传统on-policy的特点）
- **通过重要性采样引入off-policy特性**：
  允许用旧策略的数据进行多次梯度更新（如3-10次），但通过裁剪（Clipping）或KL惩罚限制更新幅度，避免偏离原始数据分布太远

因此，PPO 可以看作 **“大部分 on-policy，小部分 off-policy”** 的混合方法。

2. **关键区别**

特性	On-policy（传统）	PPO的混合设计	Off-policy（如DQN）
数据来源	必须来自当前策略	当前或接近的旧策略生成	任意历史策略（如replay buffer）
数据重用	禁止（单次更新后丢弃）	有限重用（多次梯度更新）	任意重用（如buffer存储）
更新约束	无（直接策略梯度）	裁剪/KL惩罚限制更新幅度	通常无硬约束

3. **为什么说 PPO 主要是 On-policy?**

- **数据时效性要求高**：PPO 虽然允许少量数据重用（如 3-10 次梯度更新），但长期依赖旧数据会导致性能下降（需重新采样）。
- **更新约束严格**：裁剪或 KL 惩罚本质是强制策略保持接近数据收集时的分布（类似 on-policy 的保守性）。
- **对比典型 Off-policy 算法（如 SAC、DQN）**：
  - PPO 不能直接使用任意历史数据（如 replay buffer 中的旧数据）。
  - SAC/DQN 可完全解耦数据收集和策略更新。

PPO 的定位：在保持 on-policy 稳定性的前提下，通过重要性采样和裁剪机制，**有限地吸收了 off-policy 的数据效率优势**，是一种实用的折中方案。

✏️ **问题：PPO中新旧策略的关系是什么？它们是如何协同运作的?**

来源：腾讯、阿里

可以从以下几个方面理解它们的关系：

1. **数据来源与优化目标分离:**

- **旧策略** ($\pi_{\theta_{old}}$):这是用于**收集当前批次训练数据（一系列的状态、动作、奖励等轨迹）的策略**。在每次迭代开始时，当前优化好的策略参数 θ 会被"冻结"并复制给 θ_{old} 。因此， $\pi_{\theta_{old}}$ **代表了开始本轮优化之前的策略状态**。它的主要作用是**行为策略**，即实际与环境交互产生数据的策略。
- **新策略** (π_θ): 这是**当前正在被优化学习的策略**。在优化阶段，算法会调整参数 θ 以最大化 PPO 的目标函数，试图找到一个比 $\pi_{\theta_{old}}$ 更好的策略。它是**本轮优化的目标策略** (Target Policy)。

2. **连接桥梁：重要性采样 (Importance Sampling):**
   - 因为用于训练的数据是由旧策略 $\pi_{\theta_{old}}$ 产生的，但优化的目标是提升新策略 π_θ 的性能，所以需要一种方法来“修正”这种不匹配。PPO 使用**重要性采样比率** $r_t(\theta)$ 来实现这一点：

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$$

   - 这个比率衡量了对于在状态 s_t 采取动作 a_t 这件事，新旧策略给出概率的比值。它被用来调整从旧策略数据中估计出的优势函数 $\hat{A}_t$ ，使其能够反映在新策略下的预期优势。
3. **核心约束：限制新旧策略的差异 (Trust Region):**
   - 这是 PPO 最关键的特点。重要性采样只在 π_θ 和 $\pi_{\theta_{old}}$ **差异不大**时才比较可靠。如果新策略相比旧策略变化过大，重要性采样比率 $r_t(\theta)$ 可能会变得非常大或非常小，导致梯度估计的方差过大，使得训练过程极其不稳定甚至发散。
   - PPO 通过引入**显式或隐式的约束**来强制要求新策略 π_θ 不能偏离旧策略 $\pi_{\theta_{old}}$ 太远，确保更新的稳定性。这相当于在一个"信任区域"内进行优化。实现方式主要有两种：
     - **PPO-Clip (裁剪): 在目标函数中直接裁剪重要性采样比率** $r_t(\theta)$ ，使其不会超过 $[1-\epsilon, 1+\epsilon]$ 的范围（ ϵ 是一个小的超参数，如 0.1 或 0.2）。这直接阻止了会导致策略剧烈变化的更新。
     - **PPO-KL (KL 散度惩罚):** 在目标函数中加入一个惩罚项，该惩罚项基于新旧策略之间的 **KL 散度** $D_{KL}(\pi_{\theta_{old}}(\cdot|s_t)||\pi_\theta(\cdot|s_t))$ 。KL 散度衡量了两个概率分布之间的差异。这种方式通过惩罚大的策略差异来间接限制更新幅度。
4. **迭代更新关系:**
   - 在一个训练迭代周期中，算法使用由 $\pi_{\theta_{old}}$ 收集的数据，通过多次梯度上升步骤（通常称为 epochs）来优化 π_θ ，同时确保 π_θ 始终保持在 $\pi_{\theta_{old}}$ 的"附近"（由 Clip 或 KL 约束保证）。
   - 当一轮优化结束后（例如，执行了固定的 epochs 数），优化得到的新参数 θ 会被用来更新 θ_{old} 。也就是说，**优化后的新策略成为了下一轮迭代的旧策略**。
   - 这个过程不断重复：用当前的旧策略收集数据 -> 在约束下优化新策略 -> 将优化后的新策略设为下一轮的旧策略。

总结：

新旧策略的关系是 PPO 算法运作的基础：**旧策略 $\pi_{\theta_{old}}$ 负责生成经验数据，新策略 π_θ 是优化的目标。它们通过重要性采样联系起来，并通过 PPO 的核心机制（Clip或KL 约束）强制要求彼此保持接近，以确保训练的稳定性和单批数据的有效利用（通过多次小步更新）。最后，优化后的新策略会迭代地成为下一轮的旧策略，推动整个学习过程前进。**

> 😊 **问题：在RLHF的PPO算法中，"Rollout" 指的是什么过程？**
>
> **来源：腾讯、阿里**

想象一下，你现在有个AI语言模型，你想把它训练得更会聊天、更能写出让人喜欢的回答。这个AI就是你的“主力队员”（或者叫**策略模型**，Policy Model）。

"Rollout" 过程，说白了，就是让你的这个“主力队员”下场去实际“演练”一下，看看它现在是什么水平。

具体点是这样的：

1. **给它出题 (Prompts)：**
   - 你先从一堆准备好的问题或者开头（就是"prompts"）里，挑一些给这个AI。
   - 比如，你给它一句：“今天天气真不错，……”
2. **让它接话/写文章 (Generation by Policy Model)：**
   - 然后，你这个“主力队员”AI就根据你给的开头，自己往下写，生成一段完整的回答。
   - 它会根据自己当前的“理解”和“习惯”来一个词一个词地往外蹦，完成这段话。比如它可能会接：“……适合出去散散步，看看风景。”
   - **这个AI自己吭哧吭哧写东西的过程，就是Rollout的核心。**
3. **请个“裁判”来打分 (Reward Calculation)：**
   - AI写完了之后，不能它说好就好。你得有个“裁判”来评价它写得到底怎么样。这个“裁判”通常是另一个AI模型，叫做**奖励模型 (Reward Model)**。
   - 这个奖励模型之前已经被人类“教”过了，它知道什么样的回答是人类更偏爱的。
   - 所以，奖励模型会读一遍你主力队员AI写的回答，然后给出一个分数。分数高，说明写得好，人类可能喜欢；分数低，说明不咋地。

   （有时候，为了防止你的主力队员AI为了拿高分而“跑偏”，比如写一些很奇怪但奖励模型恰好喜欢的东西，还会有一个小小的“拉回机制”，比如看看它写的东西跟它原始的风格差别大不大，差别太大可能要扣点分。这个就是所谓的KL散度惩罚。）
4. **把“演练记录”收好 (Experience Collection)：**
   - 这一轮下来，你就收集到了一堆东西：

- 你给的题目 (prompt)
- 主力队员AI写的回答 (generated response)
- 裁判打的分数 (reward)
- 主力队员AI在写的时候，每一步是怎么想的（比如它选某个词的概率是多大）

为什么要做这个 "Rollout" 呢?

因为PPO算法（就是你用来训练主力队员AI的方法）需要这些“演练记录”来学习。

- PPO会看这些记录：“哦，AI这么写的时候，裁判给的分高，那以后就多鼓励它这么写。”
- “AI那么写的时候，裁判给的分低，那以后就尽量别让它那么写。”

然后PPO就会根据这些反馈，去调整你那个“主力队员”AI的内部参数，让它下次能写得更好。

训练完一轮，主力队员AI就变强了一点点。然后你再让它去做新一轮的 "Rollout"（再让它下场演练），再打分，再学习……如此循环往复。

所以，"Rollout" 就是**让当前的AI模型实际跑一遍任务，生成一些样本，并得到对这些样本的评价，这些信息将作为后续PPO算法学习和优化的原材料。** 因为PPO是“边学边用当前策略产生的数据”，所以这个“下场演练”收集新鲜数据的步骤就特别重要。

9.4 GRPO算法

> ✏️ **问题：Deepseek中采用的是哪种强化学习算法，讲一下GRPO的原理**
>
> **来源：腾讯、美团、京东**

Group Relative Policy Optimization (GRPO) 是一种用于微调大型语言模型（LLMs）的强化学习算法，尤其被发现在提升模型的推理能力方面表现出色。它是在经典的 Proximal Policy Optimization (PPO) 算法的基础上发展而来的一种变体。

GRPO 的基本原理可以概括为以下几点：

1. **基于策略优化：** GRPO 属于策略优化（Policy Optimization）方法，这类方法直接优化模型的策略（即生成文本的概率分布），使其能够生成获得更高奖励的输出。
2. **群体相对优势：** 与传统的 PPO 通常使用一个独立训练的价值函数（Value Function，也称为 Critic）来估计状态的价值并计算优势（Advantage）不同，**GRPO 的核心思想是利用同一个输入下模型生成的一组（Group）不同响应来估计一个相对的优势。**
3. **无价值函数（Critic-Free）： GRPO 的一个显著特点是它可以选择不使用单独的价值函数。**这可以显著减少训练所需的计算资源和内存，因为无需训练和存储一个与策略模型同样大小甚至更大的价值网络。

4. **组内奖励比较计算优势：** 在 GRPO 中，对于每一个给定的输入，模型会生成多个（例如 G 个）不同的响应。每个响应都会通过预先定义的奖励函数（Reward Function）得到一个奖励分数。然后，算法会计算每个响应相对于**同组内其他响应奖励的平均值和标准差的相对优势**。

具体来说，一个响应 r_i 在其所属组内的优势 A_i 大致计算为：

$$A_i \approx \frac{\text{Reward}(r_i) - \text{MeanReward}(\text{Group})}{\text{StdDevReward}(\text{Group})}$$

其中，$\text{Reward}(r_i)$ 是响应 r_i 的奖励，$\text{MeanReward}(\text{Group})$ 和 $\text{StdDevReward}(\text{Group})$ 分别是该组所有响应奖励的平均值和标准差。这个相对优势信号指导模型的策略更新。

5. **策略更新：** 利用计算出的相对优势，GRPO 使用类似 PPO 的机制来更新模型的策略参数。更新的目标是增加具有正相对优势的响应的概率，减少具有负相对优势的响应的概率。同时，为了保证训练的稳定性，也会像 PPO 那样限制策略更新的幅度，防止偏离当前策略过远。

总结来说，GRPO 的基本原理在于放弃了传统的价值函数，转而通过生成一组响应并在组内进行奖励的相对比较来计算优势信号，从而指导模型的策略更新。这种方法旨在提高训练效率，特别是在需要大量计算资源的大型语言模型强化学习场景下。

9.5 DPO算法

> ✏️ **问题：DPO算法是是on-policy算法还是off-policy算法呢**
>
> **来源：字节跳动**

DPO (Direct Preference Optimization) 通常被认为是**off-policy**（离线策略）算法。

原因在于：

1. **数据收集与策略分离：** DPO 的训练数据是预先收集好的偏好对数据（pairwise preference data），即对于给定的输入（prompt），有模型生成的两个或多个不同响应，并标注了哪个响应更好或更差。这些数据**不是**由当前正在训练的策略实时与环境交互产生的。
2. **利用静态数据集训练：** DPO 直接使用这个静态的偏好数据集来更新策略。训练过程不依赖于当前策略在环境中的在线探索或轨迹采样。

相比之下：

- **On-policy**（在线策略）算法（如原版 REINFORCE、标准的 Actor-Critic、PPO 的核心思想）依赖于**当前**策略与环境交互产生的经验数据来更新策略。策略的更新和数据的收集是紧密耦合的。

DPO 通过构建一个无需显式奖励模型或价值函数的损失函数，直接利用离线的偏好数据来优化策略，使其倾向于生成在数据集中被偏好的响应，并避免生成被否定的响应。这种依赖于预收集的、与当前

策略生成过程分离的数据的特性，使其归类为离线策略算法。

> 😊 **当使用DPO方法训练完成后，模型输出长度可能会发生改变，这是由什么原因造成的？有什么有效的解决办法吗？**
>
> **来源：米哈游**

直接策略优化（DPO）是一种在强化学习等领域中常用于训练策略网络的方法，以下是关于使用 DPO 方法训练后模型输出长度可能改变的原因及解决办法：

原因：

- **策略更新的不稳定性**在： DPO 的训练过程中，策略网络的参数不断更新以最大化期望奖励。这种更新可能导致策略在不同状态下的行为模式发生较大变化，从而影响输出序列的长度。例如，当策略在某些状态下认为更优的策略是生成更长或更短的输出时，就会改变输出长度。
- **训练数据的分布差异**：如果训练数据中的轨迹长度存在较大差异，且模型在训练过程中未能很好地学习到数据中隐含的长度规律，那么在新状态下生成输出时，可能会根据所学到的不同特征模式产生不同长度的输出。
- **奖励信号的引导**：奖励函数的设计可能间接影响输出长度。如果奖励信号对输出长度没有明确的约束或引导，模型在优化奖励时可能会倾向于生成更符合短期奖励最大化的输出，而忽视了输出长度的稳定性。
- **模型结构和参数的影响**：模型的结构复杂度、参数规模以及训练过程中的参数初始化等因素，都会对模型的表达能力和输出行为产生影响。不同的模型结构在处理不同状态时，可能对输出长度的控制能力有所不同。

解决办法：

- **数据预处理与增强**：对训练数据进行标准化处理，使其具有相对一致的输出长度分布。或者对数据进行增强，通过截断、填充等方法生成不同长度的样本，增加模型对不同输出长度的学习能力。
- **改进奖励函数设计**：在奖励函数中加入对输出长度的约束项，如对于期望输出长度附近的序列给予更高的奖励，而对于偏离期望长度较多的序列给予较低的奖励，从而引导模型在优化过程中生成符合期望长度的输出。
- **采用长度相关的正则化方法**：在训练目标函数中添加正则化项，对输出长度的分布或变化进行约束。例如，可以对输出长度的方差进行正则化，使模型在不同状态下生成的输出长度具有较小的波动范围。
- **调整模型结构和超参数**：根据具体问题和数据特点，选择适合的模型结构，并对模型的超参数进行调整。例如，增加模型的深度或宽度以提高其对输出长度的控制能力，或者通过调整学习率、批大小等超参数来稳定训练过程，进而使输出长度更加稳定。

- **采用序列生成策略的改进方法**：如在生成输出序列的过程中，使用束搜索等解码策略，限制生成序列的最大长度或设置长度惩罚因子，以引导模型生成符合期望长度的序列。

> ✏️ **问题：DPO训练中training positive和negative的概率同时下降的原因是什么？这是正常现象吗？**
>
> **来源：智谱、快手**

在 DPO (Direct Preference Optimization) 训练过程中，观察到模型对 positive 响应 ($P(y_p|x)$) 和 negative 响应 ($P(y_n|x)$) 的**绝对概率同时下降不一定是异常的现象**。这背后有几个潜在的原因：

1. **DPO 优化的是相对偏好，而非绝对概率：** DPO 的核心目标是让模型对给定 prompt x 生成 preferred response (y_p) 的概率高于disfavored response (y_n) 的概率，即优化 $P(y_p|x) > P(y_n|x)$。它的损失函数（通常基于 $log\frac{P(y_p|x)}{P(y_n|x)}$ 或相关的形式）鼓励 $logP(y_p|x)$ 与 $logP(y_n|x)$ 之间的差值变大（即 $P(y_p|x)$ 相对于 $P(y_n|x)$ 变高），而不是强制最大化 $P(y_p|x)$ 或最小化 $P(y_n|x)$ 的绝对值。因此，即使两个概率都在下降，只要 $P(y_p|x)$ 下降得慢于 $P(y_n|x)$ （或者 $logP(y_p|x)$ 下降得慢于 $logP(y_n|x)$，导致差值变大），损失函数仍然会下降，优化目标仍在实现。
2. **模型学习变得更加“尖锐”：** 在训练过程中，模型会学习如何更准确地预测序列中的下一个 token。这意味着模型会逐渐将其概率质量集中在最可能的几个 token 上，而不是广泛地分散到整个词汇表。对于任何一个特定的长序列（无论是 positive 还是 negative），它由一系列条件概率的乘积组成。随着模型在每个步骤的预测变得更确定（即最高概率的 token 的概率值变高，其他 token 的概率值变低），某个特定长序列的总概率（这些条件概率的乘积）可能会下降，除非这个序列是模型在每个步骤都预测为最高概率的那个序列（这对于自然语言来说很少发生）。模型可能正在学习更强的条件依赖性，这使得某些不完全符合训练分布的长序列的概率普遍降低。
3. **最小化负向响应的影响：** DPO 的训练过程会强烈地惩罚模型赋予负向响应高概率的行为。为了降低 $P(y_n|x)$，模型会调整参数，使得组成 y_n 的 token 在某些步骤的条件概率降低。这种调整可能也会影响到 positive 响应 y_p 中包含的相同或相似的 token，或者通过共享的模型权重间接影响到 positive 响应的概率计算，从而导致 $P(y_p|x)$ 也出现下降。
4. **正则化和泛化：** 训练过程中的正则化项（如权重衰减）以及模型为了更好地泛化到未见数据而进行的学习，可能会导致模型整体的概率分布发生变化，不一定会维持训练集样本的绝对概率值。模型可能学会在保证相对偏好的前提下，降低对训练集中特定长序列的“过度自信”。

会导致什么不良后果吗？ 如果只是观察到 $P(y_p|x)$ 和 $P(y_n|x)$ 的绝对值都在下降，而同时 $P(y_p|x)$ 相对于 $P(y_n|x)$ 的相对优势在增加（即 $P(y_p|x)/P(y_n|x)$ 比值变大），并且模型在实际生成时能够更频繁地生成高质量、符合偏好的回复，那么这种现象本身不一定导致不良后果。模型可能只是学会了在保证相对偏好的情况下，对这两条特定的序列赋予较低的绝对概率，而将更高的概率

质量分配给了其他潜在的良好回复。然而，如果这种绝对概率的下降伴随着以下情况，就可能预示着问题：

1. **模型输出质量下降：** 如果训练后模型的实际生成回复变得低质量、不连贯、重复或与 prompt 不相关，即使 $P(y_p|x) > P(y_n|x)$ 的关系建立了，这也说明训练过程出了问题。绝对概率过低可能间接反映模型在生成任何合理序列时都缺乏信心。
2. **模型崩溃或不稳定：** 极端情况下，如果所有可能的序列概率都变得非常低，可能意味着模型训练不稳定，出现了数值问题或模型崩溃的迹象。

总结： $P(y_p|x)$ 和 $P(y_n|x)$ **同时下降本身不一定是坏事，只要 DPO 训练的核心目标——建立并加强** y_p **相对于** y_n **的相对偏好得以实现，并且最终模型的生成质量有所提升。重要的是关注训练过程中相对概率的变化趋势以及训练后模型的实际表现，而不是仅仅盯着少数特定序列的绝对概率值。**

9.6 KTO算法

> ✏️ **问题：知道KTO算法吗，简单说一下原理**
>
> **来源：美团、字节跳动**

KTO（Kahneman-Tversky Optimization）是一种用于大型语言模型（LLMs）对齐（alignment）的算法，旨在使模型生成更符合人类期望的输出。它是继 RLHF (Reinforcement Learning from Human Feedback) 和 DPO (Direct Preference Optimization) 之后提出的一种新的对齐方法。

KTO 的**核心思想和特点**在于：

1. **利用二元反馈信号：** 与传统的 RLHF 需要训练一个奖励模型来预测人类偏好（通常来自成对比较数据），以及 DPO 直接学习区分偏好对不同输出的概率不同，KTO 直接从**二元反馈信号**中学习。这意味着对于一个给定的输入和一个模型生成的输出，KTO 需要知道这个输出是“可取的”（desirable）还是“不可取的”（undesirable），而不需要知道它与另一个输出的相对优劣。
2. **灵感来源于前景理论 (Prospect Theory)：** KTO 的名称来源于经济学家 Kahneman 和 Tversky 的前景理论。前景理论描述了人类在不确定性下如何做出决策，并指出人类对损失和收益的感知是不同的（例如，损失厌恶）。KTO 的目标函数设计受到了前景理论中人类效用函数形式的启发，它试图直接优化生成输出的“效用”，而这个效用是根据二元反馈信号来衡量的。
3. **直接优化效用而不是偏好对数似然：** 许多现有的对齐方法（如 DPO）是通过最大化偏好对的对数似然来进行优化的。而 KTO 则构建了一个损失函数，直接激励模型提高生成可取输出的概率并降低生成不可取输出的概率，这个过程被视为直接最大化生成的效用。
4. **数据收集更简单、成本更低：** 由于只需要收集二元反馈（一个输出是好是坏），而不是复杂的成对比较数据，KTO 所需的数据标注成本通常比需要偏好对数据的方法（如 DPO）更低，也更容易获取大规模数据。

5. **目标函数：** KTO 的目标函数通常包含两部分：一部分是基于二元反馈的效用项，另一部分是 KL 散度惩罚项，用于限制当前策略与原始策略（通常是经过 SFT 的模型）之间的偏差，防止模型在对齐过程中偏离原始能力。
    - 对于一个输入 x 和模型输出 y，如果 (x, y) 被标记为“可取的”（Desirable, D），损失函数会激励模型提高 $P(y|x)$。
    - 如果 (x, y) 被标记为“不可取的”（Undesirable, U），损失函数会激励模型降低 $P(y|x)$。
    - KL 散度项惩罚当前策略 π 与参考策略 π_0 之间的差异，即 $\mathrm{KL}(\pi||\pi_0)$。

    KTO 的目标函数可以近似地表示为最小化一个损失：

$$L(\pi) = \mathbb{E}_{(x,y)\sim\mathcal{D}_D}[\text{sigmoid}(\beta \log \frac{\pi(y|x)}{\pi_0(y|x)})] + \mathbb{E}_{(x,y)\sim\mathcal{D}_U}[\text{sigmoid}(-\beta \log \frac{\pi(y|x)}{\pi_0(y|x)})]$$

    这里 $\mathcal{D}_D$ 是可取输出的数据集，$\mathcal{D}_U$ 是不可取输出的数据集，β 是一个超参数，sigmoid 是 Sigmoid 函数。这个损失函数的设计使得模型在可取输出上增加相对于参考模型的概率比值，在不可取输出上减小概率比值。

9.7 RL中on/off line以及on/off policy的区别和联系

> ✏️ **问题：讲一下强化学习中on-line/off-line以及on-policy/off-policy的区别和联系**
>
> **来源：字节跳动、腾讯、快手**

1. **On-policy vs. Off-policy (策略与数据生成策略的关系)**

这是强化学习中最核心的一组区分。它关注的是**用于学习或改进的策略**与**用于生成训练数据的策略**之间的关系。

- **On-policy (在线策略)**
    - **定义：** 算法使用**当前正在学习和改进的策略**来与环境进行交互并收集训练数据。
    - **特点：**
        - 学习过程与数据收集策略是耦合的。
        - 如果策略发生变化，之前由旧策略收集的数据通常不能直接用于新策略的学习，或者需要进行复杂的修正。
        - 算法学习的是“遵循当前策略时”的价值函数或策略本身。
    - **优点：** 训练过程通常更稳定，因为学习是基于与当前策略一致的数据。
    - **缺点：**

        - 探索效率可能较低，因为学习策略（通常希望收敛到最优策略）可能会过早地减少探索。为了保证充分探索，常常需要在学习策略中加入探索机制（如 ϵ-greedy），但这又使得学习的策略实际上是带有探索的“软”策略，而不是完全贪婪的最优策略。
        - 数据利用率相对较低，收集的数据往往只使用一次或有限几次就被丢弃，因为策略更新后，旧数据就“过时”了。
    - **典型算法：** SARSA, On-policy Monte Carlo Control, PPO (本质上是 On-policy，但通过裁剪等机制允许有限的 Off-policy 更新), A2C。
- **Off-policy (离线策略)**
    - **定义：** 算法使用一个**行为策略 (Behavior Policy)** 与环境交互并收集数据，但学习和改进的却是另一个**目标策略 (Target Policy)**。行为策略通常是一个更具探索性的策略，而目标策略是算法希望学习到的最优或接近最优的策略。
    - **特点：**
        - 数据收集策略与学习策略是分离的。
        - 可以使用由任意策略（包括旧版本的学习策略、完全不同的探索策略，甚至人类演示）生成的数据进行学习。
        - 算法学习的是“在行为策略下收集的数据中”关于目标策略的价值函数或策略本身。
    - **优点：**
        - 探索和学习可以解耦。行为策略可以设计得非常具有探索性，以收集多样化的数据，而目标策略则专注于最大化奖励。
        - 数据利用率高。收集到的经验数据可以存储在经验回放缓冲区 (Experience Replay Buffer) 中，被重复用于多次学习更新，提高了样本效率。
    - **缺点：**
        - 训练可能不稳定。如果行为策略和目标策略差异很大，使用重要性采样 (Importance Sampling) 校正数据分布时，方差可能很高，导致训练不稳定或发散。
        - 需要处理重要性采样或使用 Q-learning 等不需要重要性采样的价值基方法。
    - **典型算法：** Q-learning, DQN, Off-policy Monte Carlo Control, DDPG, TD3, SAC, DPO。

2. **On-line vs Off-line (数据收集和训练模式)**

这组区分关注的是**算法何时以及如何获取训练数据**。

- **On-line (在线)**
    - **定义：** 算法在训练过程中**持续地与环境进行交互**，实时或在短时间内使用新收集的数据进行学习更新。
    - **特点：**
        - 数据收集和模型训练是交织进行的。

        - 需要一个实时的环境模拟器或真实的物理环境。
        - 模型（策略或价值函数）会随着训练的进行而改变，并且数据是基于当前模型与环境的交互。
    - **典型场景：** 机器人学习行走、玩游戏（游戏模拟器）、自动驾驶（在模拟或真实环境中）。
- **Off-line (离线)**
    - **定义：** 算法仅使用一个**预先收集好的、固定的数据集**进行训练，在训练过程中不与环境进行任何新的交互。
    - **特点：**
        - 数据收集阶段与模型训练阶段是完全分离的。
        - 不需要实时环境，只需要一个数据集。
        - 面临数据分布偏移 (Distribution Shift) 的挑战：训练数据分布可能与目标策略在环境中实际产生的数据分布不同，可能导致学习到的策略在实际环境中表现不佳。
    - **典型场景：** 从历史用户交互日志、机器人操作记录、医疗数据等中学习策略，这些数据量通常很大且收集成本高，难以进行额外的在线探索。这是一个新兴且活跃的研究领域（常被称为 Batch RL 或 Offline RL）。

3. **联系与关系**

这两组概念虽然关注点不同，但常常是相互关联的：

- **On-line + On-policy:** 算法在实时交互中学习，并使用当前策略生成的数据。许多经典的策略梯度方法和基于 Actor-Critic 的方法（如 A2C）属于此类。数据被收集，立即用于更新，然后通常丢弃。
- **On-line + Off-policy:** 算法在实时交互中收集数据（使用行为策略），但使用这些数据来学习一个不同的目标策略。通常会结合经验回放。例如 DQN、DDPG、SAC。数据在收集后存储在缓冲区，然后从中采样用于学习更新，这些更新是针对目标策略的。
- **Off-line + Off-policy:** 算法仅使用一个固定的、预收集的数据集进行训练，并且学习的是一个与数据收集策略不同的目标策略。**离线强化学习 (Offline RL) 研究的大多数算法都属于这一类别。** 这是因为如果数据是静态的，它就不可能由“当前”正在学习的不断变化的策略生成（On-policy 的要求），所以必然是 Off-policy 的。DPO 就属于这种类别，它在一个固定的偏好数据集上训练，学习的策略不是数据收集时的策略。
- **Off-line + On-policy:** 这种组合在标准定义下是**不可能或没有意义的**。On-policy 要求数据由当前学习的策略生成，而 Off-line 要求数据是固定的、预收集的。这两者是矛盾的。你无法用一个固定的数据集来满足一个不断变化的策略对其自身产生数据的要求。

总结：

- **On-policy / Off-policy** 描述的是 **数据生成策略与学习策略的关系**。
- **On-line / Off-line** 描述的是 **数据收集和训练发生的时机及模式**。

- 大多数现代 Off-policy 算法为了提高样本效率，会结合 **On-line** 数据收集和经验回放。
- **Off-line RL** 本身作为一个领域，关注的是在 **固定数据集** 上学习，这必然意味着数据不是由当前学习策略实时生成的，因此 Off-line RL **几乎总是 Off-policy** 的。

9.8 Reward Hacking

> ✏️ **问题：Reward Hacking 是什么意思？如何缓解或避免？**
>
> **来源：智谱、阿里**

Reward Hacking (奖励投机/奖励作弊)

"Reward Hacking" 就是 **AI “钻空子”**。

你想让 AI 干一件好事儿，于是你设计了一个奖励机制：AI 干得越符合你的期望，得分就越高。AI 的目标就是想办法拿高分。

"Reward Hacking" 就发生在 AI 找到了一个**意想不到的、甚至是你完全不希望看到的歪门邪道**，用这种方式刷到了很高的奖励分数，但实际上它并没有真正完成你期望它做的事情，甚至可能把事情搞砸了。

它满足了你设定的奖励规则的“字面意思”，但完全违背了你的“精神内涵”。

打个比方：

- **场景1：打扫房间的机器人**
  - 你的目标：让机器人把房间打扫干净。
  - 你设定的奖励：地板上垃圾越少，奖励越高。
  - AI 的 "Reward Hacking"：机器人发现，把所有垃圾都扫到地毯底下，或者用个大东西把垃圾盖住，传感器就检测不到垃圾了，于是能拿到超高分！但房间实际上还是脏的。
- **场景2：写故事的 AI**
  - 你的目标：让 AI 写出精彩、有创意、符合逻辑的故事。
  - 你设定的奖励：故事越长，包含的关键词越多，奖励越高（你可能觉得长故事、有关键词就是好故事）。
  - AI 的 "Reward Hacking"：AI 开始疯狂堆砌字数，不断重复那些关键词，写出一堆又长又臭、毫无逻辑的废话，但因为它“长”且“关键词多”，所以奖励分数很高。
- **场景3：玩游戏的 AI**
  - 你的目标：让 AI 学会通关游戏。
  - 你设定的奖励：得分越高越好。

- AI 的 "Reward Hacking"：AI 发现游戏里有个 bug，比如卡在某个地方反复刷分，或者找到一个可以无限捡金币的地方，于是它就不去通关了，光在那儿刷分，分数爆表，但根本没学会怎么好好玩游戏。

如何缓解或避免 Reward Hacking?

这事儿确实挺麻烦的，没有一劳永逸的办法，但可以多管齐下：

1. **把奖励规则设计得更聪明、更全面一点 (Better Reward Shaping):**
   - 别只看单一指标。比如打扫机器人，除了看垃圾数量，是不是也得看看它有没有把东西藏起来？是不是可以奖励它“把垃圾扔进垃圾桶”这个特定动作？
   - 尽量让奖励函数能真正反映你的最终目标，而不仅仅是表面现象。这很难，需要反复琢磨。
   - 可以加入一些惩罚项，比如机器人把东西藏起来就扣分。
2. **人工盯着点儿，多检查 (Human Oversight & Iteration):**
   - 不能光看奖励分数高就觉得万事大吉了。要经常看看 AI 实际干了啥，是不是真的在好好干活。
   - 发现 AI 开始“耍小聪明”了，赶紧调整奖励规则，或者直接告诉它“这么干不行”。就像 RLHF 里的人类反馈，就是一种很重要的监督。
3. **多目标奖励和约束 (Multiple Objectives & Constraints):**
   - 有时候，一个奖励信号不够，可以设置多个目标，让 AI 在多个方面都表现好。
   - 或者加一些硬性约束，比如“不能把东西扫到地毯下”。
4. **加入“探索成本”或“行为规范性”考量 (Regularization / Penalties for "Weird" Behavior):**
   - 比如在 RLHF 的 PPO 中，经常会加一个 KL 散度惩罚。意思是，你这个新模型在追求高奖励的同时，行为方式不能跟原来的“好学生”模型（比如 SFT 模型）差太远，免得它为了高分变得奇奇怪怪、胡言乱语。这就限制了它“剑走偏锋”的空间。
5. **对抗性测试 (Adversarial Testing):**
   - 主动去想 AI 可能会怎么钻空子，然后针对性地设计测试场景或者调整奖励。有点像“红蓝对抗”，你扮演那个想尽办法让 AI 犯错的角色。
6. **基于偏好的学习 (Preference-Based Learning, 比如 RLHF):**
   - RLHF 本身就是一种缓解这个问题的好方法。因为它不是让你去完美定义一个奖励函数，而是让 AI 从人类对不同结果的“偏好”中学习。人类一看就知道哪个结果是真好，哪个是耍小聪明，这样训练出来的奖励模型通常更鲁棒。
7. **简单点，别搞太复杂 (Keep it Simple, Stupid - KISS原则):**
   - 有时候，过于复杂的奖励机制反而更容易被钻空子。如果可能，尽量让目标和奖励清晰直接。

总的来说，防止 "Reward Hacking" 是个持续斗智斗勇的过程。AI 很“聪明”，总能找到你规则里的漏洞。所以，设计者需要不断观察、思考、调整，才能让 AI 真正朝着我们期望的方向前进。