

中级AI算法工程师测试题 - 答案与评分标准

本文档仅供内部评分使用

一、算法与数据结构进阶(25分)

1. 算法设计与优化(15分)

1.1 Top-K问题 (8分)

参考答案：

方法一：排序法 (2分)

思路：

- 对整个数组排序
- 返回 `nums[n-k]` (第k大即倒数第k个)

时间复杂度： $O(n \log n)$

空间复杂度： $O(1)$ 或 $O(n)$ (取决于排序算法)

优点：简单直接

缺点：不必要地排序了所有元素

方法二：最小堆 (3分)

思路：

- 维护一个大小为k的最小堆
- 遍历数组，保持堆中是最大的k个元素
- 堆顶就是第k大的元素

伪代码：

```
heap = MinHeap(size=k)
for num in nums:
    if heap.size < k:
        heap.push(num)
    elif num > heap.top():
        heap.pop()
        heap.push(num)
return heap.top()
```

时间复杂度： $O(n \log k)$

空间复杂度： $O(k)$

方法三：快速选择 (QuickSelect) (3分)

思路：

- 类似快速排序的分区思想
- 每次选择pivot，将数组分为大于和小于pivot的两部分
- 根据pivot位置决定继续在哪一部分搜索

平均时间复杂度： $O(n)$

最坏时间复杂度： $O(n^2)$ - 可通过随机选择pivot优化

空间复杂度： $O(1)$

优点：平均情况下最优

频繁查询优化：

- 可以维护一个排序好的数组或平衡二叉搜索树
- 或使用顺序统计树，支持 $O(\log n)$ 查询任意第k大

流式数据处理：

- 使用大小为k的最小堆
- 新数据到来时更新堆
- 近似方法：Count-Min Sketch, Reservoir Sampling

评分标准：

- 给出两种方法各2-3分（共5分）
 - 复杂度分析正确（2分）
 - 频繁查询优化（1分）
 - 流式处理思路（加分项，答出可+1分）
-

1.2 最长递增子序列 (7分)

参考答案：

$O(n^2)$ 动态规划解法 (3分)

状态定义：

$dp[i]$ = 以 $nums[i]$ 结尾的最长递增子序列长度

状态转移：

$dp[i] = \max(dp[j] + 1)$ for all $j < i$ where $nums[j] < nums[i]$

如果没有这样的 j ， $dp[i] = 1$

初始化： $dp[i] = 1$ (每个元素自己构成长度1的序列)

答案： $\max(dp[i])$ for all i

伪代码：

```
n = len(nums)
```

```
dp = [1] * n
```

```
for i from 1 to n-1:
```

```
    for j from 0 to i-1:
```

```
        if nums[j] < nums[i]:
```

```
            dp[i] = max(dp[i], dp[j] + 1)
```

```
return max(dp)
```

时间复杂度： $O(n^2)$

空间复杂度： $O(n)$

$O(n \log n)$ 优化解法 (3分)

核心理想：

- 维护一个数组 tails，tails[i] 表示长度为 i+1 的递增子序列的最小末尾元素
- tails 数组是单调递增的
- 对每个元素，用二分查找找到它在 tails 中的位置

伪代码：

```
tails = []
for num in nums:
    pos = binary_search(tails, num) // 找到第一个 >= num 的位置
    if pos == len(tails):
        tails.append(num)
    else:
        tails[pos] = num
return len(tails)
```

例子：nums = [10,9,2,5,3,7,101,18]

遍历过程：

- 10: tails = [10]
- 9: tails = [9] (9替换10)
- 2: tails = [2] (2替换9)
- 5: tails = [2,5]
- 3: tails = [2,3] (3替换5)
- 7: tails = [2,3,7]
- 101: tails = [2,3,7,101]
- 18: tails = [2,3,7,18] (18替换101)

答案：4

时间复杂度： $O(n \log n)$

空间复杂度： $O(n)$

输出具体子序列 (1分)

需要额外记录每个位置的前驱：

- 在DP过程中，记录 prev[i] = j (表示i的前一个元素是j)
- 从最大的dp[i]反向追溯
- 或在优化解法中，同时维护实际的序列元素索引

评分标准：

- $O(n^2)$ DP状态定义和转移正确 (3分)
- $O(n \log n)$ 优化思路正确 (3分)

- 输出序列的方法（1分）

2. 图算法应用(10分)

2.1 并查集(Union-Find) (10分)

参考答案：

(a) 核心操作实现 (4分)

```
python

class UnionFind:
    def __init__(self, n):
        self.parent = [i for i in range(n)] # 每个节点的父节点
        self.rank = [0] * n                # 树的秩（高度）

    def find(x):
        # 查找x的根节点，同时进行路径压缩
        if parent[x] != x:
            parent[x] = find(parent[x]) // 路径压缩
        return parent[x]

    def union(x, y):
        # 合并x和y所在的集合
        root_x = find(x)
        root_y = find(y)

        if root_x == root_y:
            return # 已经在同一集合

        // 按秩合并：将秩小的树连接到秩大的树
        if rank[root_x] < rank[root_y]:
            parent[root_x] = root_y
        elif rank[root_x] > rank[root_y]:
            parent[root_y] = root_x
        else:
            parent[root_y] = root_x
            rank[root_x] += 1
```

(b) 优化策略 (3分)

路径压缩 (Path Compression):

- 在find操作时，将路径上所有节点直接连到根节点
- 减少树的高度，加速后续查询
- 实现： $\text{parent}[x] = \text{find}(\text{parent}[x])$
- 效果：树变得扁平，查询接近 $O(1)$

按秩合并 (Union by Rank):

- 合并时，将秩小的树连到秩大的树上
- 保持树的平衡，避免退化成链表
- 秩可以是树的高度或大小
- 效果：树的高度控制在 $O(\log n)$

复杂度:

- 单独使用路径压缩：均摊 $O(\log n)$
- 单独使用按秩合并：均摊 $O(\log n)$
- 两者结合：均摊 $O(\alpha(n))$ ， α 是反阿克曼函数，实际上 \approx 常数

(c) 应用场景分析 (3分)

问题:

- n 个节点， m 次操作（连接边或查询连通性）

并查集方法:

- 初始化： $O(n)$
- 每次操作：均摊 $O(\alpha(n)) \approx O(1)$
- 总时间： $O(n + m \cdot \alpha(n)) \approx O(n + m)$

DFS/BFS方法:

- 每次查询需要遍历图： $O(n + \text{edges})$
- m 次查询总时间： $O(m \cdot n)$
- 非常慢，特别是查询次数多时

并查集优势:

- 动态维护连通性非常高效
- 适合"连接-查询"交替的场景
- 实现简单，常数小

实际应用：

- 网络连通性检测
- 图像分割（连通区域）
- Kruskal最小生成树算法
- 社交网络（判断两人是否在同一社交圈）

评分标准：

- find和union实现正确（各2分，共4分）
 - 两种优化解释清楚（各1.5分，共3分）
 - 复杂度对比和优势分析（3分）
-

二、深度学习算法理解(35分)

3. Transformer与Attention机制(15分)

3.1 Attention机制深入理解 (8分)

参考答案：

为什么除以 $\sqrt{d_k}$? (2分)

直觉解释：

- 当 d_k 很大时， QK^T 的点积结果会很大
- 大的点积值进入softmax后，梯度会变得很小（梯度消失）
- 例如：如果 QK^T 中某个值是100，softmax后接近1，其他接近0，梯度几乎为0

数学角度：

- 假设Q和K的元素是均值0、方差1的独立变量
- QK^T 的点积是 d_k 个项相加，方差会累积到 d_k
- 除以 $\sqrt{d_k}$ 后，方差归一化回1
- 保持数值稳定，梯度流动更好

时间和空间复杂度 (2分)

对于序列长度 n ，特征维度 d ：

- QK^T : $(n, d) \times (d, n) = (n, n)$ - 时间 $O(n^2d)$

- **softmax:** 对 (n, n) 矩阵 - 时间 $O(n^2)$
- **乘以V:** $(n, n) \times (n, d) = (n, d)$ - 时间 $O(n^2d)$
- **总时间复杂度:** $O(n^2d)$
- **空间复杂度:** $O(n^2)$ - 存储attention矩阵

瓶颈: 长序列时 n^2 的复杂度, 例如 $n=4096$ 时需要16M的attention矩阵

Multi-Head优势 (2分)

- **多样性:** 不同head学习不同的attention pattern
 - 有的head关注局部信息 (相邻词)
 - 有的head关注长距离依赖
 - 有的head关注句法关系、语义关系等
- **表征子空间:** 类似CNN的多通道, 捕获不同特征
- **集成效果:** 多个head的组合比单个更robust

Encoder-Decoder架构中的Attention (2分)

1. Encoder Self-Attention:

- Q, K, V都来自encoder的前一层输出
- 每个位置关注整个输入序列

2. Decoder Self-Attention (Masked):

- Q, K, V都来自decoder的前一层输出
- 使用mask, 只能关注当前位置之前的token (因果attention)
- 保证生成时的自回归性质

3. Cross-Attention (Encoder-Decoder Attention):

- Q来自decoder当前层
- K, V来自encoder的输出
- Decoder关注Encoder的信息

评分标准:

- $\sqrt{d_k}$ 的解释 (2分)
- 复杂度分析正确 (2分)
- Multi-Head优势 (2分)
- Encoder-Decoder的三种Attention (2分)

3.2 优化长序列Attention (7分)

参考答案（选择任意两种详细解释）：

方法一：Sparse Attention (稀疏注意力)

核心思想：

- 不是每个位置都关注所有其他位置
- 只关注一部分位置，形成稀疏的attention矩阵

常见pattern：

1. **局部窗口**: 每个位置只关注前后 w 个位置
2. **跨步attention**: 每隔 k 个位置关注一次
3. **全局token**: 某些特殊位置（如[CLS]）关注所有位置

复杂度：

- 时间： $O(n \cdot w \cdot d)$ 或 $O(n \cdot \sqrt{n} \cdot d)$ （取决于pattern）
- 空间： $O(n \cdot w)$ 而不是 $O(n^2)$

性能：

- 可能损失一些长距离依赖
- 但在很多任务上效果接近标准attention

例子：Longformer, BigBird

方法二：Sliding Window Attention (滑动窗口)

核心思想：

- 每个token只关注固定窗口大小 w 内的token
- 窗口可以是单向或双向

实现：

- 窗口大小通常是512或1024
- 可以多层堆叠，增加感受野

复杂度：

- 时间: $O(n \cdot w \cdot d)$, w 是窗口大小
- 空间: $O(n \cdot w)$

优缺点:

- 优点: 实现简单, 效果稳定
- 缺点: 难以捕获超出窗口的依赖
- 解决: 通过多层堆叠, 感受野成倍增长

例子: Longformer的局部attention

方法三: Flash Attention (内存优化)

核心思想:

- 不改变计算量, 而是优化GPU内存访问
- 减少HBM (高带宽内存) 和SRAM (片上内存) 之间的数据传输

原理:

- 标准attention需要将整个 $n \times n$ 矩阵存在HBM
- Flash Attention将计算分块, 每次只加载部分到SRAM
- 在SRAM中完成计算, 减少IO

复杂度:

- 时间: 仍然是 $O(n^2d)$, 但wall-clock时间更短
- 空间: $O(n)$ 而不是 $O(n^2)$

效果:

- 训练速度提升2-4倍
- 支持更长的序列
- 数学上完全等价于标准attention

例子: FlashAttention-1, FlashAttention-2

方法四: Linear Attention (线性复杂度)

核心思想:

- 用kernel方法近似softmax attention
- 将attention重新写成线性形式

数学推导：

标准: $\text{Attention} = \text{softmax}(\mathbf{Q}\mathbf{K}^T)\mathbf{V}$

近似: $\text{Attention} \approx \varphi(\mathbf{Q})(\varphi(\mathbf{K})^T \mathbf{V})$

通过改变计算顺序，先算 $(\mathbf{K}^T \mathbf{V})$ ，复杂度降到 $O(nd^2)$

复杂度：

- 时间： $O(nd^2)$ ， d 通常远小于 n
- 空间： $O(nd)$

性能：

- 近似，会有精度损失
- 在某些任务上效果接近
- 特别适合超长序列

例子： Linformer, Performer, Linear Transformer

评分标准（每种方法3.5分，选两种共7分）：

- 核心思想清楚（1.5分）
 - 复杂度分析正确（1分）
 - 说明优缺点或性能（1分）
 - 只答一种方法最多得4分
-

4. 反向传播与优化器(12分)

4.1 反向传播理解 (6分)

参考答案：

Batch Normalization前向计算 (1.5分)

输入: x (batch_size, feature_dim)

参数: γ (可学习的缩放), β (可学习的平移)

步骤:

1. 计算均值: $\mu = \text{mean}(x, \text{axis}=0)$
2. 计算方差: $\sigma^2 = \text{var}(x, \text{axis}=0)$
3. 归一化: $x_{\text{norm}} = (x - \mu) / \sqrt{\sigma^2 + \epsilon}$
4. 缩放平移: $y = \gamma * x_{\text{norm}} + \beta$

其中 ϵ 是小常数($1e-5$)防止除零

反向传播复杂性 (1.5分)

为什么比线性层复杂:

- 均值 μ 和方差 σ^2 都是 x 的函数, 依赖整个batch
- 对 x 的梯度需要考虑:
 1. x 直接影响 x_{norm}
 2. x 通过 μ 影响 x_{norm}
 3. x 通过 σ^2 影响 x_{norm}
- 需要用链式法则处理这三条路径
- 涉及batch维度的求和, 梯度要分配给batch中的所有样本

线性层: $y = Wx + b$, 梯度简单: $\partial L / \partial x = W^T \cdot \partial L / \partial y$

ReLU反向传播 (1分)

前向: $y = \max(0, x)$

反向:

$$\partial L / \partial x = \partial L / \partial y * (x > 0)$$

即:

- 如果 $x > 0$, 梯度正常传递
- 如果 $x \leq 0$, 梯度为0 ("神经元死亡")

伪代码:

$$\text{grad_x} = \text{grad_y} * (x > 0)$$

梯度消失与BN (2分)

梯度消失问题：

- 深层网络中，梯度需要连续相乘传播
- 如果每层梯度 <1 ，多层后梯度 $\rightarrow 0$
- 特别是Sigmoid/Tanh，导数 <1 ，加剧问题
- 导致浅层网络参数几乎不更新

BN如何缓解：

1. **归一化激活值：** 保持激活值在合理范围（均值0方差1）
2. **避免饱和：** 激活值不会太大或太小，激活函数梯度更好
3. **平滑损失曲面：** 使优化更容易，梯度更稳定
4. **允许更大学习率：** 加速训练，更快走出梯度小的区域

评分标准：

- BN前向步骤（1.5分）
 - 反向传播复杂性解释（1.5分）
 - ReLU梯度（1分）
 - 梯度消失和BN作用（2分）
-

4.2 优化器对比 (6分)

参考答案：

SGD vs SGD with Momentum vs Adam (2分)

SGD (随机梯度下降):

$$\theta = \theta - lr * g$$

- 最简单，直接用梯度更新
- 可能在ravines（山谷）中震荡
- 收敛慢

SGD with Momentum:

$$v = \beta * v + g$$

$$\theta = \theta - lr * v$$

- 累积过去梯度的动量
- 减少震荡，加速收敛
- 类似物理中的惯性

Adam (Adaptive Moment Estimation):

$$m = \beta_1 * m + (1 - \beta_1) * g \quad \# \text{ 一阶矩估计}$$

$$v = \beta_2 * v + (1 - \beta_2) * g^2 \quad \# \text{ 二阶矩估计}$$

$$\theta = \theta - lr * m / (\sqrt{v} + \epsilon)$$

- 结合Momentum（一阶矩）和RMSprop（二阶矩）
- 自适应学习率，每个参数不同
- 通常默认选择，效果稳定

Adam为什么更快? (1.5分)

1. 自适应学习率:

- 对频繁更新的参数用小学习率
- 对稀疏更新的参数用大学习率
- 不需要手动调整每个参数的lr

2. 动量加速:

- 一阶矩m平滑梯度方向
- 减少噪声，更稳定的更新方向

3. 适应不同scale:

- 二阶矩v标准化梯度
- 对梯度大小不敏感

Learning Rate Warm-up (1分)

什么是warm-up:

- 训练初期，学习率从很小逐渐增加到目标值
- 例如：从0线性增加到0.001，持续1000步

为什么需要（特别是大模型）:

1. **初始化不稳定**：参数随机初始化，梯度可能很大
2. **Adam的bias**：开始时m和v的估计不准
3. **防止早期震荡**：小学习率让模型先稳定下来
4. **batch size影响**：大batch训练需要warm-up稳定

AdamW改进 (1分)

与Adam的区别：

- **Adam**: L2正则化加在梯度上

$g = g + \lambda * \theta$ (将weight decay加入梯度)
然后用Adam更新

- **AdamW**: Weight decay直接作用于参数

先用Adam更新: $\theta' = \theta - lr * m / \sqrt{v}$
再做weight decay: $\theta = \theta' - lr * \lambda * \theta$

为什么重要：

- 在Adam中，L2正则和weight decay不等价（因为自适应学习率）
- AdamW的weight decay效果更好
- 特别是在视觉任务和大模型上

SGD vs Adam的权衡 (0.5分)

SGD可能更好的情况：

1. **泛化性能**：有研究表明SGD找到的最优点更"平坦"，泛化更好
2. **小数据集**：Adam容易过拟合
3. **长时间训练**：给足够时间，SGD可能收敛到更好的解
4. **计算机视觉**：ResNet等模型传统上用SGD效果好

Adam更好的情况：

- 快速收敛
- NLP任务
- 不想精细调lr

- 大模型训练

评分标准：

- 三种优化器对比（2分）
 - Adam快的原因（1.5分）
 - Warm-up解释（1分）
 - AdamW改进（1分）
 - SGD vs Adam权衡（0.5分）
-

5. 模型架构设计(8分)

5.1 残差连接 (4分)

参考答案：

为什么能训练更深的网络？(2分)

梯度流角度：

假设残差块： $y = F(x) + x$

反向传播时：

$$\begin{aligned}\partial L / \partial x &= \partial L / \partial y * \partial y / \partial x \\ &= \partial L / \partial y * (\partial F / \partial x + 1) \\ &= \partial L / \partial y * \partial F / \partial x + \partial L / \partial y\end{aligned}$$

关键点： "+1"项

- 即使 $\partial F / \partial x$ 很小（甚至为0），梯度仍能通过 "+1" 直接传递
- 创建了一条梯度的 "高速公路"
- 浅层网络可以直接收到梯度信号

对比普通网络：

- 普通： $y = F(x)$ ，梯度 $\partial L / \partial x = \partial L / \partial y * \partial F / \partial x$
- 如果 F 是多层， $\partial F / \partial x$ 是连乘，容易消失
- 残差：至少保证梯度不小于 $\partial L / \partial y$

Transformer 中的位置 (1分)

残差连接在Transformer的两个地方：

1. Multi-Head Attention之后：

$$x' = x + \text{MultiHeadAttention}(x)$$

2. Feed-Forward之后：

$$x'' = x' + \text{FeedForward}(x')$$

每个sub-layer后都有残差连接

Pre-LN vs Post-LN (1分)

Post-LN (原始Transformer):

$$x' = \text{LayerNorm}(x + \text{SubLayer}(x))$$

- 先做残差，再normalize
- 训练可能不稳定（特别是深层网络）

Pre-LN (现代常用):

$$x' = x + \text{SubLayer}(\text{LayerNorm}(x))$$

- 先normalize，再做残差
- 训练更稳定
- 深层网络更容易训练
- GPT-2, GPT-3等都用Pre-LN

哪个更常用： Pre-LN，因为稳定性更好

评分标准：

- 梯度流解释（2分）
 - Transformer中的位置（1分）
 - Pre-LN vs Post-LN（1分）
-

5.2 位置编码 (4分)

参考答案：

正弦位置编码公式 (1分)

$$\text{PE}(\text{pos}, 2i) = \sin(\text{pos} / 10000^{(2i/d)})$$
$$\text{PE}(\text{pos}, 2i+1) = \cos(\text{pos} / 10000^{(2i/d)})$$

其中：

- pos: 位置索引 (0, 1, 2, ...)
- i: 维度索引 (0到d/2)
- d: 模型维度

偶数维用sin，奇数维用cos

为什么用正弦函数? (1分)

1. **周期性：** 不同频率的正弦波可以表示不同的位置关系
2. **相对位置：** 任意位置pos+k可以表示为pos的线性组合（因为三角恒等式）
3. **外推性：** 可以处理比训练时更长的序列
4. **范围固定：** 值在[-1,1]，不会随位置增大

如果用整数索引：

- 位置值没有上界，可能影响训练
- 无法表示相对位置关系
- 难以外推到更长序列

可学习 vs 固定位置编码 (1分)

固定位置编码（如正弦）：

- 优点：不需要学习，参数少；可外推到任意长度
- 缺点：可能不是最优的；无法适应特定任务

可学习位置编码：

- 优点：可以学到任务特定的位置信息；可能效果更好
- 缺点：增加参数；难以外推到更长序列；需要更多数据

实践： 两种效果相近，现代模型（如BERT）常用可学习的

相对位置编码优势 (1分)

绝对位置编码：

- 每个位置有固定的编码
- 问题：位置0和位置100的关系，与位置50和位置150的关系，无法共享

相对位置编码：

- 只关心token之间的相对距离
- 例如：距离为2的两个token，无论在哪里，编码相同
- 优势：
 1. **平移不变性**：序列整体移动，关系不变
 2. **更好的外推**：训练时的相对位置关系可用于更长序列
 3. **性能提升**：在很多任务上效果更好

例子：T5, DeBERTa等使用相对位置编码

评分标准：

- 正弦公式 (1分)
 - 使用正弦的原因 (1分)
 - 可学习vs固定 (1分)
 - 相对位置优势 (1分)
-

三、机器学习理论(25分)

6. 概率与生成模型(10分)

6.1 变分自编码器(VAE) (6分)

参考答案：

编码器和解码器输出 (1.5分)

编码器(Encoder)：

- 输入：数据 x (如图像)
- 输出：后验分布 $q(z|x)$ 的参数
 - μ (均值向量)

- σ^2 (方差向量, 或 $\log(\sigma^2)$)
- 通常假设 $q(z|x)$ 是高斯分布 $N(\mu, \sigma^2)$

解码器(Decoder):

- 输入: 隐变量 z
- 输出: 重构数据的分布 $p(x|z)$ 的参数
 - 对图像: 可能输出像素值 (均值)
 - 对二值图像: 输出伯努利分布参数
- 或直接输出重构的 x'

KL散度项的作用 (1.5分)

$$\text{ELBO} = \underbrace{E[\log p(x|z)]}_{\text{重构损失}} - \underbrace{\text{KL}(q(z|x) \parallel p(z))}_{\text{正则化项}}$$

KL散度项 $\text{KL}(q(z|x) \parallel p(z))$ 的作用:

1. **正则化:** 约束后验 $q(z|x)$ 接近先验 $p(z)$ (通常是 $N(0, I)$)
2. **防止过拟合:** 不让编码器学出任意的分布
3. **结构化隐空间:** 让隐变量 z 有良好的结构, 方便采样生成
4. **平衡:** 如果只有重构损失, 编码器可能把 z 映射到很分散的区域

没有KL项: 编码器可能学到一个完美的编码, 但隐空间混乱, 无法生成

Reparameterization Trick (2分)

问题:

- 需要从 $q(z|x) = N(\mu, \sigma^2)$ 采样 z
- 采样操作不可微, 无法反向传播

不能直接采样的原因:

$z \sim N(\mu, \sigma^2)$ // 这是一个随机操作
 $\text{loss} = f(z)$
 如何计算 $\partial \text{loss} / \partial \mu$ 和 $\partial \text{loss} / \partial \sigma$?

采样操作阻断了梯度流

Reparameterization trick:

不直接采样 $z \sim N(\mu, \sigma^2)$

而是:

1. 采样 $\varepsilon \sim N(0, I)$ (与 μ, σ 无关)
2. 计算 $z = \mu + \sigma \odot \varepsilon$

现在 z 是 μ 和 σ 的确定性函数，可以求梯度:

$$\partial z / \partial \mu = 1$$

$$\partial z / \partial \sigma = \varepsilon$$

效果: 梯度可以通过 z 反向传播到 μ 和 σ

VAE生成模糊的原因 (1分)

1. **重构损失 (MSE):** 鼓励输出是"平均"的图像
 - 如果训练集中有多种可能, VAE输出平均值
 - 导致细节模糊
2. **隐空间正则化:** KL项强制隐空间平滑
 - 相似的 z 生成相似的 x
 - 减少多样性
3. **高斯假设:** 假设 $p(x|z)$ 是高斯, 适合平滑重构

对比GAN:

- GAN用判别器, 直接优化生成质量
- 不需要像素级重构, 可以生成更锐利的图像

评分标准:

- 编码器/解码器输出 (1.5分)
- KL项作用 (1.5分)
- Reparameterization trick (2分)
- 模糊原因 (1分)

6.2 VAE vs GAN (4分)

参考答案:

训练目标的本质区别 (2分)

VAE:

- 目标：最大化ELBO（数据的对数似然下界）

$$\max E[\log p(x|z)] - \text{KL}(q(z|x) \parallel p(z))$$

- 显式建模数据分布 $p(x)$
- 通过最大化似然训练
- 有明确的目标函数

GAN:

- 目标：通过对抗训练让生成分布接近真实分布

Generator: 生成假数据骗过Discriminator



Discriminator: 区分真假数据

- 隐式建模，不直接优化似然
- 通过对抗游戏训练
- 目标是让判别器无法区分



核心差异： VAE是似然方法，GAN是对抗方法

各自优缺点 (1.5分)

VAE:

-  优点：
 - 训练稳定
 - 有理论保证（优化ELBO）
 - 可以直接计算似然
 - 有良好的隐空间结构
-  缺点：
 - 生成质量较差（模糊）
 - 像素级重构损失不理想
 - 假设过强（高斯分布）

GAN:

-  优点:
 - 生成质量高（锐利、逼真）
 - 不需要显式建模
 - 可以生成复杂分布
-  缺点:
 - 训练不稳定（模式崩溃、梯度消失）
 - 难以评估（没有似然）
 - 超参数敏感
 - 没有encoder（不能编码真实数据）

选择VAE的场景 (0.5分)

1. 需要编码能力:
 - 要把真实数据映射到隐空间
 - 如数据压缩、特征提取
2. 训练稳定性重要:
 - 资源有限，不想调参
 - 需要可靠的训练过程
3. 需要似然估计:
 - 异常检测（计算 $p(x)$ ）
 - 需要理论保证
4. 插值和编辑:
 - 隐空间平滑，便于插值
 - 语义编辑

例子:

- 药物分子生成（需要encoder）
- 数据压缩
- 半监督学习（利用隐变量）

评分标准:

- 训练目标差异（2分）

- 优缺点对比 (1.5分)
 - 选择VAE的场景 (0.5分)
-

7. 损失函数与正则化(8分)

7.1 损失函数设计 (4分)

参考答案：

为什么用交叉熵而不是MSE? (1分)

交叉熵 (Cross-Entropy)：

对于分类： $L = -\sum y_{\text{true}} * \log(y_{\text{pred}})$

对于二分类： $L = -[y * \log(p) + (1-y) * \log(1-p)]$

MSE (均方误差)：

$L = (y_{\text{pred}} - y_{\text{true}})^2$

原因：

1. 概率解释：

- 分类输出是概率分布
- 交叉熵衡量两个分布的差异
- MSE更适合连续值预测

2. 梯度特性：

- 交叉熵+softmax：梯度是 $(y_{\text{pred}} - y_{\text{true}})$ ，清晰
- MSE+softmax：梯度包含softmax导数，可能很小

3. 数值稳定性：

- 交叉熵对极端错误惩罚更大
- 加速学习

类别不平衡的调整 (1分)

问题： 类别1有9000个样本，类别2有1000个样本

- 模型可能总预测类别1，准确率90%但无用

解决方法：

1. 类别权重(Class Weights):

```
weight = n_samples / (n_classes * n_samples_per_class)
loss = weight * CE_loss
```

给少数类更大权重

2. Focal Loss:

$$FL = -\alpha * (1-p)^{\gamma} * \log(p)$$

- 降低易分样本的权重
- 关注难分样本
- α 平衡类别, γ 控制关注度

3. 重采样:

- 过采样少数类
- 欠采样多数类

对比学习的InfoNCE (1分)

核心思想:

- 拉近正样本对, 推远负样本对
- 学习语义相似的特征

InfoNCE损失:

对于anchor样本 x 和正样本 x_+ , 负样本 $\{x_-\}$:

$$L = -\log(\exp(\text{sim}(x, x_+)/\tau) / (\exp(\text{sim}(x, x_+)/\tau) + \sum \exp(\text{sim}(x, x_-)/\tau)))$$

其中:

- $\text{sim}(\cdot, \cdot)$ 是相似度 (如余弦相似度)
- τ 是温度参数
- 分子: 正样本对的相似度
- 分母: 正样本+所有负样本

效果： 最大化正样本对的相似度，同时与负样本区分开

Huber Loss使用场景 (1分)

Huber Loss公式：

$$L(y, \hat{y}) = \begin{cases} \frac{1}{2}(y-\hat{y})^2 & \text{if } |y-\hat{y}| \leq \delta \\ \delta|y-\hat{y}| - \frac{1}{2}\delta^2 & \text{otherwise} \end{cases}$$

特点：

- 结合了MSE和MAE
- 小误差时用平方（MSE），大误差时用绝对值（MAE）

对比：

- **MSE：** 对大误差惩罚重，容易受离群点影响
- **MAE：** 对所有误差同等对待，梯度恒定
- **Huber：** 在小误差区域梯度大（快速收敛），大误差区域稳健

使用场景：

1. 数据中有离群点（outliers）
2. 目标检测中的bbox回归
3. 强化学习（Q-learning）
4. 需要对大误差稳健但保持收敛速度

评分标准：

- 交叉熵vs MSE（1分）
- 类别不平衡方法（1分）
- InfoNCE解释（1分）
- Huber Loss（1分）

7.2 正则化技术 (4分)

参考答案：

L1 vs L2正则化 (1分)

L2正则化 (Ridge):

$$\text{Loss} = L_{\text{data}} + \lambda * ||\theta||_2^2 = L_{\text{data}} + \lambda * \sum \theta_i^2$$

$$\text{梯度: } \partial \text{Loss} / \partial \theta = \partial L / \partial \theta + 2\lambda \theta$$

$$\text{更新: } \theta = \theta - \text{lr} * (\text{grad} + 2\lambda \theta) = (1 - 2\lambda \text{lr}) * \theta - \text{lr} * \text{grad}$$

效果: 权重衰减, 参数变小但不为0

L1正则化 (Lasso):

$$\text{Loss} = L_{\text{data}} + \lambda * ||\theta||_1 = L_{\text{data}} + \lambda * \sum |\theta_i|$$

$$\text{梯度: } \partial \text{Loss} / \partial \theta = \partial L / \partial \theta + \lambda * \text{sign}(\theta)$$

效果: 权重可以变为0, 产生稀疏解

为什么L1产生稀疏?

- L1的梯度是常数 $\lambda * \text{sign}(\theta)$, 不管 θ 多小
- 小权重持续被推向0
- L2的梯度是 $2\lambda \theta$, θ 小时梯度小, 不容易到0

几何解释: L1是菱形, 容易在坐标轴交点 (即某维为0)

Dropout训练vs推理 (1分)

训练时:

每个神经元以概率 p 被"丢弃" (输出设为0)

保留的神经元输出不变 (或除以 $(1-p)$ 缩放)

例如: `dropout_rate=0.5`

一半神经元随机置0

推理时:

所有神经元都保留

输出乘以 $(1-p)$

或: 训练时除以 $(1-p)$, 推理时不变 (更常用)

为什么不同?

- **训练：** 随机dropout模拟集成学习，防止过拟合
- **推理：** 要用完整模型，但需要补偿训练时的dropout

效果：

- 类似训练了多个子网络的集成
- 每次dropout是不同的子网络

Label Smoothing原理 (1分)

硬标签 (Hard Label)：

真实类别：[0, 0, 1, 0, 0] (one-hot)

软标签 (Label Smoothing)：

平滑后： $[\epsilon/(K-1), \epsilon/(K-1), 1-\epsilon, \epsilon/(K-1), \epsilon/(K-1)]$

其中 ϵ 是平滑系数 (如0.1)， K 是类别数

例子： $K=5, \epsilon=0.1$

- 原标签：[0, 0, 1, 0, 0]
- 平滑后：[0.025, 0.025, 0.9, 0.025, 0.025]

如何防止过拟合：

1. **减少过自信：** 不鼓励模型输出极端概率 (接近0或1)
2. **改善泛化：** 对其他类别保留一点概率
3. **平滑决策边界：** 类别间的区分不那么绝对
4. **正则化效果：** 相当于最小化KL散度到均匀分布的加权版本

Data Augmentation是正则化吗? (1分)

答案： 是的

理由：

1. **增加数据多样性：** 模拟更多可能的变化
2. **防止记住训练集：** 每个epoch看到的数据都不同
3. **等价于正则化：** 在数据空间添加先验知识
4. **减少过拟合：** 模型必须学习不变性而非记忆

类比：

- L2正则：在参数空间约束
- Dropout：在网络结构上随机化
- Data Augmentation：在数据空间扩充

常见方法：

- 图像：旋转、翻转、裁剪、颜色抖动
- 文本：同义词替换、回译
- 音频：时间拉伸、加噪声

评分标准：

- L1 vs L2（1分）
 - Dropout差异（1分）
 - Label Smoothing（1分）
 - Data Augmentation（1分）
-

8. 评估与调试(7分)

8.1 模型评估指标 (4分)

参考答案：

二分类指标定义 (1.5分)

混淆矩阵：

	预测为正	预测为负
实际为正	TP	FN
实际为负	FP	TN

准确率(Accuracy)：

$$\text{Accuracy} = (TP + TN) / (TP + TN + FP + FN)$$

所有预测正确的比例

精确率(Precision)：

$$\text{Precision} = \text{TP} / (\text{TP} + \text{FP})$$

预测为正的样本中，真正为正的比率

回答："预测的有多准？"

召回率(Recall/TPR):

$$\text{Recall} = \text{TP} / (\text{TP} + \text{FN})$$

实际为正的样本中，被正确预测的比例

回答："找到了多少？"

F1分数:

$$F1 = 2 * (\text{Precision} * \text{Recall}) / (\text{Precision} + \text{Recall})$$

精确率和召回率的调和平均

平衡两者

准确率误导的情况 (1分)

例子:

- 疾病检测：1000个样本，990个健康，10个患病
- 模型总预测"健康"
- 准确率 = $990/1000 = 99\%$
- 但模型完全无用！所有患病都漏检（Recall=0）

为什么误导:

- 类别不平衡时，准确率被多数类主导
- 不能反映少数类（往往是我们关心的）的性能

解决:

- 看Precision, Recall, F1
- 使用混淆矩阵
- 针对每个类别单独评估

ROC vs PR曲线 (1分)

ROC曲线:

- 横轴: $\text{FPR} = \text{FP}/(\text{FP} + \text{TN})$ （假阳性率）

- 纵轴：TPR = TP/(TP+FN) (真阳性率/Recall)
- AUC-ROC：曲线下面积，越大越好
- 适合：类别相对平衡

PR曲线：

- 横轴：Recall
- 纵轴：Precision
- AP (Average Precision)：曲线下面积
- 适合：类别不平衡

何时用PR曲线：

- 正样本很少（如异常检测、稀有疾病）
- 更关注正类的表现
- PR曲线对不平衡更敏感，ROC可能过于乐观

多分类评估 (0.5分)

Macro-averaging：

对每个类别计算指标，然后简单平均

$$\text{Macro-F1} = (\text{F1_class1} + \text{F1_class2} + \dots + \text{F1_classK}) / K$$

- 每个类别权重相同
- 适合关注所有类别表现

Micro-averaging：

全局计算TP, FP, FN，然后算指标

$$\text{Micro-F1} = 2 * \text{TP} / (2 * \text{TP} + \text{FP} + \text{FN})$$

- 每个样本权重相同
- 被大类主导

Weighted-averaging：

按各类别样本数加权平均

$$\text{Weighted-F1} = \sum(n_i * \text{F1}_i) / n_{\text{total}}$$

- 考虑类别大小
- 更全面

评分标准：

- 四个指标定义（1.5分）
 - 准确率误导例子（1分）
 - ROC vs PR（1分）
 - 多分类方法（0.5分）
-

8.2 训练问题诊断 (3分)

参考答案：

Loss变成NaN (1分)

可能原因：

1. 梯度爆炸：

- 梯度太大，参数更新过大
- 导致数值溢出

2. 学习率过大：

- 更新步长太大
- 跳过最优点，发散

3. 数值不稳定：

- $\log(0)$ 或除以0
- softmax输入过大导致exp溢出

4. 数据问题：

- 输入包含NaN或Inf
- 标签错误

解决方法：

- 降低学习率
- 梯度裁剪 (gradient clipping)
- 检查数据预处理
- 使用更稳定的损失函数（如log-sum-exp技巧）

- Batch Normalization

Loss不下降或震荡 (1分)

Loss不下降:

- **学习率过小**: 更新太慢
- **初始化问题**: 权重初始化不当
- **梯度消失**: 深层网络, 梯度传不回来
- **优化器选择**: 可能需要Adam而不是SGD
- **局部最小值/鞍点**: 卡住了 (较少见)

解决:

- 提高学习率或用学习率调度
- 更好的初始化 (Xavier, He)
- 换优化器
- 检查网络架构 (加BN、残差连接)

Loss震荡:

- **学习率过大**: 在最优点附近跳跃
- **Batch size太小**: 梯度噪声大
- **数据问题**: 某些batch特别难

解决:

- 降低学习率
- 增大batch size
- 梯度累积
- 使用学习率warm-up

训练集下降但验证集上升 (1分)

诊断: 过拟合

原因:

- 模型容量太大
- 训练时间太长
- 数据太少

- 没有正则化

解决方法：

1. 模型层面：

- 减小模型（层数、宽度）
- 增加Dropout
- L2正则化
- Early Stopping

2. 数据层面：

- 增加训练数据
- Data Augmentation
- 减少数据泄露

3. 训练层面：

- 降低学习率
- 减少训练epochs
- 监控验证集，及时停止

4. 其他：

- Batch Normalization
- Label Smoothing
- 集成学习

评分标准：

- NaN原因和解决（1分）
- 不下降/震荡诊断（1分）
- 过拟合识别和方法（1分）

四、大模型训练基础(15分)

9. 分布式训练策略(10分)

9.1 并行策略理解 (6分)

参考答案：

训练时存储的内容 (1.5分)

对于一个模型，训练时GPU需要存储：

1. 模型权重 (Parameters):

- FP16: 2 bytes/参数
- $7B \text{ 参数} \times 2 \text{ bytes} = 14GB$

2. 梯度 (Gradients):

- 与权重同样大小
- FP16: 14GB

3. 优化器状态 (Optimizer States):

- Adam需要:
 - 一阶矩 m (FP32): $7B \times 4 = 28GB$
 - 二阶矩 v (FP32): $7B \times 4 = 28GB$
- 共56GB

4. 激活值 (Activations):

- 前向传播的中间结果，用于反向传播
- 大小取决于batch size和序列长度
- 通常是最大的内存消耗

总计: $14 + 14 + 56 = 84GB$ (不含激活值)

- 一张40GB的GPU装不下!

显存估算 (0.5分)

7B参数模型，FP16权重 + FP32优化器：

- 权重: $7B \times 2 \text{ bytes} = 14GB$
- 梯度: $7B \times 2 \text{ bytes} = 14GB$
- 优化器: $7B \times 4 \times 2 = 56GB$ (m 和 v 都是FP32)
- **小计: 84GB**

激活值 (例如batch=8, seq_len=2048, hidden=4096):

- 粗略估计: $\text{batch} \times \text{seq} \times \text{hidden} \times \text{layers} \times \text{bytes}$
- 可能需要几十GB

结论: 单卡40GB远远不够

数据并行 (1.5分)

工作原理：

1. 每张GPU有完整的模型副本
2. 数据切分到不同GPU (mini-batch分片)
3. 各GPU独立前向和反向传播
4. 梯度通过AllReduce同步
5. 所有GPU用相同的梯度更新参数

通信量：

- 每步需要同步梯度： $2 \times \text{model_size}$ (FP16)
- 7B模型：14GB梯度需要通信
- 使用Ring-AllReduce可以优化

瓶颈：

- **通信开销：** GPU间通信带宽有限
- **GPU利用率：** 小模型时，通信时间>计算时间
- **显存限制：** 每张GPU仍需完整模型

适合：模型小，数据大

模型并行 (1.5分)

Tensor Parallelism (张量并行)：

在Transformer中切分：

方式1：按列切分 (Column Parallel)

Attention的Q,K,V投影：

原始：X @ W -> (batch, seq, hidden)

切分：W切成[W1, W2]，分布到2个GPU

GPU1: X @ W1 -> (batch, seq, hidden/2)

GPU2: X @ W2 -> (batch, seq, hidden/2)

结果拼接

方式2：按行切分 (Row Parallel)

Feed-Forward的第二层：

原始：H @ W -> (batch, seq, hidden)

切分：W按行切成[W1; W2]

GPU1: H1 @ W1 -> partial output

GPU2: H2 @ W2 -> partial output

结果求和 (AllReduce)

通信：

- 每层前向和反向都需要通信
- 通信量相对较小

梯度累积 (1分)

什么是梯度累积：

不是每个batch就更新参数，而是：

1. 前向传播 mini-batch 1，计算梯度，累加
2. 前向传播 mini-batch 2，计算梯度，累加
3. ...
4. 累积N个mini-batch后，一次性更新参数
5. 清空梯度，重复

什么时候需要：

1. **显存不足：** 无法用大batch size
 - 想要batch=64，但显存只够batch=16
 - 用4次梯度累积，等效batch=64
2. **模拟大batch：**
 - 大batch通常效果更好（更稳定的梯度）
 - 但受限于硬件

效果：

- 等效于更大的batch size
- 不增加显存（因为逐个mini-batch处理）
- 增加训练时间（更新频率降低）

注意： Batch Normalization统计量仍基于小batch

评分标准：

- 存储内容列举（1.5分）
 - 显存估算（0.5分）
 - 数据并行原理和瓶颈（1.5分）
 - 模型并行切分方法（1.5分）
 - 梯度累积解释（1分）
-

9.2 训练效率优化 (4分)

参考答案：

混合精度训练原理 (1.5分)

原理：

- 大部分计算用FP16（16位浮点）
- 关键部分保留FP32（32位浮点）
- 利用现代GPU的FP16算力（比FP32快2-4倍）

为什么能加速：

1. **计算更快：** Tensor Core加速FP16矩阵乘法
2. **显存更少：** 激活值和中间结果占用减半
3. **带宽更高：** 内存传输减少

具体做法：

1. 权重保留FP32副本（master weights）
2. 前向传播用FP16
3. 损失计算和反向传播用FP16
4. 梯度转回FP32
5. 用FP32梯度更新FP32权重
6. 权重转FP16用于下次前向传播

FP16问题和Loss Scaling (1.5分)

FP16的问题：

1. **数值下溢(Underflow)：**

- FP16最小正数： $\sim 6 \times 10^{-8}$
- 梯度经常小于这个值
- 小梯度被截断为0

2. 数值上溢(Overflow):

- FP16最大值： $\sim 65,000$
- 激活值或损失可能超过
- 变成Inf

Loss Scaling解决下溢:

思路：将loss放大，使梯度变大，避免下溢

1. 前向传播正常 (FP16)
2. Loss乘以scale (如 2^{16}): $\text{loss} = \text{loss} * \text{scale}$
3. 反向传播 (梯度都变大了，不会下溢)
4. 梯度除以scale还原: $\text{grad} = \text{grad} / \text{scale}$
5. 用还原的梯度更新参数

效果：小梯度被放大，不会变成0

为什么动态调整scale:

- scale太小：梯度仍可能下溢
- scale太大：可能导致上溢 (梯度变成Inf)
- 动态调整：
 - 检测到Inf/NaN时，降低scale
 - 连续多步正常时，增大scale
 - 自动找到合适的scale

Gradient Checkpointing (1分)

原理:

- 正常训练：保存所有中间激活值用于反向传播
- Checkpointing：只保存部分激活值
- 需要时重新计算 (用前向传播)

权衡:

- **显存减少：** 只保存checkpoints，其他激活值丢弃
 - 可减少80%激活值显存
- **时间增加：** 反向传播时需要重新计算
 - 约增加30-50%计算时间

使用场景：

- 显存不足，想训练更大模型或更大batch
- 例如：训练BERT-large，用checkpointing能用更大batch

实现： PyTorch的`torch.utils.checkpoint`

监控训练效率指标 (1分)

关键指标：

1. 吞吐量(Throughput)：

- samples/second 或 tokens/second
- 衡量训练速度
- 越高越好

2. GPU利用率：

- GPU使用率应该>90%
- 过低说明有瓶颈（数据加载、通信等）
- 用`nvidia-smi`或`nvitop`监控

3. 显存使用：

- 监控是否接近上限
- 合理利用显存（80-90%）
- 过低说明batch可以更大

4. Loss曲线：

- 训练loss应该平稳下降
- 验证loss与训练loss的gap
- 震荡过大说明学习率可能有问题

5. 梯度统计：

- 梯度范数（grad norm）
- 过大可能爆炸，过小可能消失
- 正常范围：0.1-10

6. 学习率:

- 当前学习率值
- 配合loss看是否需要调整

7. 时间分解:

- 数据加载时间
- 前向传播时间
- 反向传播时间
- 找出瓶颈优化

工具:

- TensorBoard, Weights & Biases
- PyTorch Profiler
- 自定义logging

评分标准:

- 混合精度原理 (1.5分)
 - FP16问题和Loss Scaling (1.5分)
 - Gradient Checkpointing (1分)
 - 监控指标 (至少3个, 每个0.3分, 共1分)
-

10. 代码实现与工程(5分)

10.1 训练代码常见问题 (5分)

参考答案:

代码问题识别 (2分)

原代码:

```
python
```

```
for epoch in range(num_epochs):
    for batch in dataloader:
        outputs = model(batch['input'])
        loss = criterion(outputs, batch['labels'])
        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

问题：

1. zero_grad位置错误 (严重):

- 应该在backward之前清零
- 当前位置：先backward，再step，再zero_grad
- 正确：zero_grad -> forward -> backward -> step

2. 缺少梯度裁剪：

- 大模型训练容易梯度爆炸
- 应该在backward和step之间裁剪

3. 缺少设备管理：

- 没有将数据移到GPU
- 应该：`batch = {k: v.to(device) for k, v in batch.items()}`

4. 缺少.train()模式：

- 训练前应该`model.train()`
- Dropout、BN等需要区分训练/推理

5. 没有梯度累积处理：

- 如果需要大batch，应该支持

6. 没有AMP (混合精度)：

- 现代训练通常用混合精度

添加梯度裁剪 (0.5分)

python

```
# 在backward之后, step之前
loss.backward()

# 方法1: 按范数裁剪 (常用)
torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=1.0)

# 方法2: 按值裁剪
torch.nn.utils.clip_grad_value_(model.parameters(), clip_value=0.5)

optimizer.step()
```

为什么需要:

- 防止梯度爆炸
- 稳定训练
- 特别是RNN、Transformer等

混合精度训练添加 (1分)

```
python
```

```
from torch.cuda.amp import autocast, GradScaler

scaler = GradScaler()

for epoch in range(num_epochs):
    for batch in dataloader:
        optimizer.zero_grad()

        # 前向传播用FP16
        with autocast():
            outputs = model(batch['input'])
            loss = criterion(outputs, batch['labels'])

        # 反向传播 (scaled)
        scaler.scale(loss).backward()

        # 梯度裁剪 (在unscale后)
        scaler.unscale_(optimizer)
        torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)

        # 更新参数
        scaler.step(optimizer)
        scaler.update()
```

保存checkpoint (0.5分)

```
python
```

```

# 保存
checkpoint = {
    'epoch': epoch,
    'model_state_dict': model.state_dict(),
    'optimizer_state_dict': optimizer.state_dict(),
    'loss': loss,
    'scaler_state_dict': scaler.state_dict(), # 如果用AMP
}
torch.save(checkpoint, 'checkpoint.pth')

# 加载
checkpoint = torch.load('checkpoint.pth')
model.load_state_dict(checkpoint['model_state_dict'])
optimizer.load_state_dict(checkpoint['optimizer_state_dict'])
epoch = checkpoint['epoch']

```

应该保存：

- 模型权重
- 优化器状态（Adam的m和v）
- 训练进度（epoch, step）
- 学习率调度器状态
- 随机数种子（可重现）
- 损失/指标（记录）

Loss变NaN的debug (1分)

可能原因：

1. 梯度爆炸：

- 检查：打印梯度范数

```

python

total_norm = 0
for p in model.parameters():
    if p.grad is not None:
        total_norm += p.grad.data.norm(2).item() ** 2
total_norm = total_norm ** 0.5
print(f'Grad norm: {total_norm}')

```

- 解决：梯度裁剪，降低学习率

2. 学习率过大：

- 尝试降低10倍
- 使用warm-up

3. 数值不稳定：

- 检查loss计算：有没有log(0), 除以0
- 检查输入：有没有NaN或Inf

```
python
```

```
assert not torch.isnan(batch['input']).any()
assert not torch.isinf(batch['input']).any()
```

4. Batch Normalization：

- BN的batch size太小
- 方差为0导致除零

5. 混合精度问题：

- FP16溢出
- 调整loss scale

排查步骤：

1. 打印loss每一步的值，找到NaN首次出现
2. 检查那一步的输入、输出、梯度
3. 简化模型，逐步添加组件定位问题
4. 使用(`torch.autograd.set_detect_anomaly(True)`)自动检测

评分标准：

- 识别代码问题（2分）
- 梯度裁剪添加（0.5分）
- 混合精度添加（1分）
- Checkpoint内容（0.5分）
- NaN debug（1分）