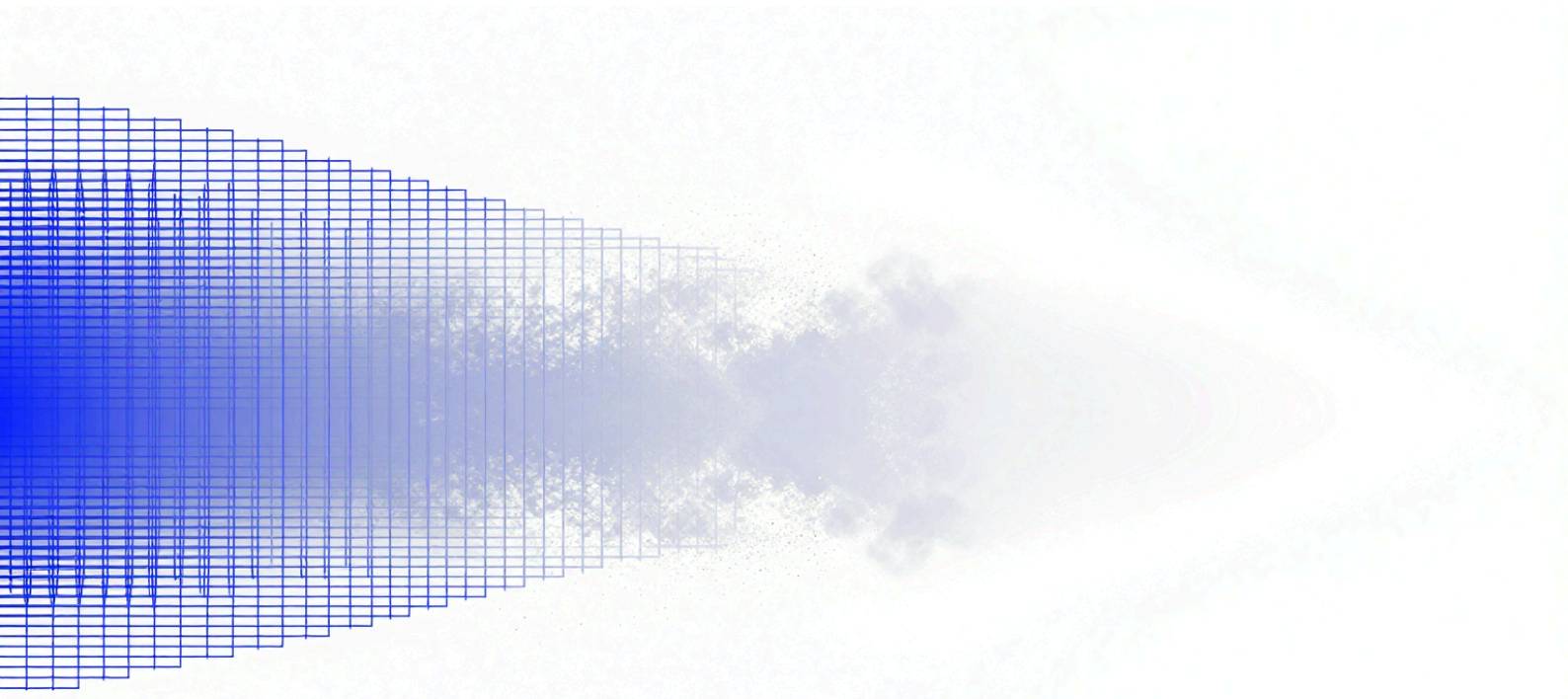
Transformer 与扩散模型

机制、推导与可交互实验

穆盼盼



目录

前言 13
全书地图 13
怎么用这本书 14
一点约定 14

I 上篇 · Transformer 与注意力机制

1 注意力：从加权平均到缩放点积 17
全景：一个 Transformer 长什么样 17
1.1 实验是怎么设计的 21
1.2 第一次实验：几乎没差别 22
1.3 查前提：实测 logits 到底有多大 23
1.4 把前提重新满足 24
1.5 注意力是一次软查表 26
1.6 为什么是 softmax 30
1.7 缩放因子到底修正了什么 32
1.8 多头：为什么不是一次用完 37
1.9 注意力看不见顺序 47
1.10 注意力图该怎么读（以及不该怎么读） 51
1.11 它在哪里失效 52
1.12 注意力的训练动态 56
1.13 排查手册：从症状倒推原因 58
1.14 完整实现 62
1.15 批处理：padding、打包与掩码 65
1.16 梯度怎么穿过注意力 67
1.17 数值精度：为什么注意力格外敏感 69
1.18 怎么把这个模型训出来 71
1.19 对照实验是怎么搭的 74
1.20 动手练习 76
1.21 自测 77
1.22 检查点解答 78
1.23 回头看，这一章讲的是一个“打补丁”的故事 81
1.24 自测答案 82
1.25 关键概念表 83

1.26 延伸阅读 . . . 85
1.27 本章回顾：一条完整的线索 . . . 86

2 一个 Transformer Block 的解剖 . . . 89
2.1 现象：12 层不如 4 层 . . . 89
2.2 第一次排查：换个位置就好了 . . . 90
2.3 排查过程：我们试了什么 . . . 91
2.4 公平一点：各自调到最好 . . . 91
2.5 转折点：那句“太整齐了” . . . 92
2.6 为什么位置这么关键 . . . 93
2.7 残差连接：那条不能省的高速公路 . . . 94
2.8 LayerNorm：它在归一化什么 . . . 96
2.9 Pre-LN 与 Post-LN：一个位置改变一切 . . . 98
2.10 RMSNorm：去掉均值之后 . . . 101
2.11 FFN：它其实是个键值记忆 . . . 102
2.12 位置编码的外推：一次没测准的实验 . . . 105
2.13 残差流：一个统一的视角 . . . 108
2.14 参数怎么分配 . . . 109
2.15 它在哪里失效 . . . 113
2.16 完整实现 . . . 117
2.17 排查手册：Block 里出问题怎么找 . . . 120
2.18 训练一个 Block：超参数与稳定性 . . . 122
2.19 初始化：为什么是 0.02 . . . 124
2.20 Dropout 与正则化 . . . 125
2.21 从 Block 到真实模型 . . . 126
2.22 检查点解答 . . . 128
2.23 自测 . . . 130
2.24 自测答案 . . . 130
2.25 动手练习 . . . 135
2.26 关键概念表 . . . 136
2.27 延伸阅读 . . . 137
2.28 从零实现：一次完整的走查 . . . 139
2.29 怎么读别人的实现 . . . 141
2.30 本章的实验清单 . . . 143
2.31 规模变化时，这些选择会怎么变 . . . 145
2.32 常见问题 . . . 146
2.33 代码索引 . . . 147
2.34 本章回顾 . . . 148
2.35 怎么做这类对照实验 . . . 152

3 训练与推理的两副面孔 155
3.1 先把现象摆清楚 155
3.2 训练那一面：一次算完所有位置 161
3.3 推理那一面：一次只能算一个 166
3.4 KV 缓存：把两副面孔接起来 173
3.5 采样的数学：让输出不再确定 183
3.6 上下文长度：一条硬边界 191
3.7 训练与推理的错配：一个错 token 的阴影 194
3.8 交互实验 201
3.9 失效边界：缓存写错的六种样子 203
3.10 上手实现：一个能跑的分步生成器 210
3.11 延伸阅读 217

4 注意力里能读出什么 221
4.1 先把话说清楚：注意力权重是什么 222
4.2 为什么这个分解有用 228
4.3 从图到数字：头部画像 229
4.4 这六个头共同的教训 231
4.5 因果检验：把它拿掉试试 237
4.6 1.69 倍 vs 6.1 倍：两个解释都占一部分 240
4.7 “全部为负”意味着什么 241
4.8 标准差和均值是一个量级 241
4.9 那么两个 Δ 的差别什么时候可信 242
4.10 固定种子的代价 243
4.11 对照实验：随机模型长什么样 244
4.12 用我们自己的数据验算 245
4.13 这个“齐”本身就是一条证据 246
4.14 归纳头：一个能被证明的电路 247
4.15 为什么第二步是“匹配前一个”而不是“匹配当前的” 248
4.16 为什么第 0 层的头不行 249
4.17 三个观察合起来说明什么 251
4.18 探针：从激活里读出信息 253
4.19 三种可能的结果，三种不同的结论 256
4.20 “探针能解码” ≠ “模型理解” 256
4.21 失效边界：可解释性研究的四个陷阱 258
4.22 一个类比 260
4.23 这和第 1、2 章的选择是一致的 261
4.24 从观察到结论：一条完整的推理链 262
4.25 这条链走完之后的结论是什么 263
4.26 还有一个更麻烦的问题：我们不知道“另外 7 个头在做什么” 264

4.27 一次完整的排查：我们想找一个“韵脚头” . . . 265
4.28 这张表的第一个意外：差距比预想的大得多 . . . 265
4.29 我们的第一版对照选错了 . . . 266
4.30 这张表里有一条清晰的分界线 . . . 266
4.31 交互实验 . . . 267
4.32 上手实现 . . . 268
4.33 延伸阅读 . . . 276
4.34 本章回顾 . . . 279

II 下篇 · 扩散模型与条件生成

5 从加噪到去噪：扩散的数学骨架 . . . 283
全景：一个扩散模型长什么样 . . . 283
5.1 本章的代码与数据 . . . 286
5.2 一个训练完美、采样塌成一团的模型 . . . 286
5.3 一个定义读了十遍都没看出来的地方 . . . 290
5.4 前向过程：加噪为什么能一步到位 . . . 292
5.5 反向过程：为什么它必须是一个分布 . . . 296
5.6 ELBO：训练目标是从哪来的 . . . 302
5.7 换个参数化：为什么大家都预测 ε . . . 306
5.8 噪声调度：$\bar{\alpha}_t$ 的形状决定了什么 . . . 309
5.9 为什么不能一步到位 . . . 313
5.10 扩散在生成模型谱系里的位置 . . . 315
5.11 连续时间：把 T 步推到极限 . . . 317
5.12 交互实验 . . . 319
5.13 它在哪里失效 . . . 322
5.14 上手实现：67 行 . . . 325
5.15 对拍：怎么确认你的实现是对的 . . . 327
5.16 训练一个扩散模型的配方 . . . 329
5.17 一次完整的排查：把案例的 bug 逐步定位出来 . . . 331
5.18 后验的几何直觉 . . . 334
5.19 从 2D 到图像：哪些结论能迁移 . . . 335
5.20 三个容易被忽略的实现细节 . . . 337
5.21 常见误解 . . . 338
5.22 本章的实验清单 . . . 339
5.23 补充检查点 . . . 341
5.24 自测 . . . 342
5.25 自测答案 . . . 342
5.26 ELBO 到底有多紧 . . . 345
5.27 把 ELBO 的三项真的算一遍 . . . 346

5.28 这一章的方法论 . . . 348
5.29 关于记号：三个长得像的 α . . . 350
5.30 关键概念表 . . . 351
5.31 延伸阅读 . . . 352
5.32 如果重来一次 . . . 354
5.33 本章没做的事 . . . 355
5.34 本章回顾 . . . 356

6 训练目标与采样器 . . . 361
6.1 本章的代码与数据 . . . 361
6.2 案例：论文里正确的 DDIM，在我们的模型上炸了 . . . 362
6.3 DDIM：把 1000 步压成 20 步 . . . 367
6.4 score matching：同一件事的另一种写法 . . . 371
6.5 高阶求解器：为什么二阶比一阶快 . . . 375
6.6 数值刚性：为什么步长会被 σ 卡住 . . . 378
6.7 $\mathcal{L}_{\text{simple}}$ 之谜 . . . 380
6.8 交互实验 . . . 383
6.9 中间地带：η 到底在 ODE 和 SDE 之间做了什么 . . . 385
6.10 一次完整的排查记录 . . . 386
6.11 这条路走到头了吗 . . . 388
6.12 为什么用二维数据 . . . 390
6.13 它在哪里失效 . . . 391
6.14 上手实现：一个能用的采样器模块 . . . 394
6.15 采样器怎么选：一棵决策树 . . . 397
6.16 同一族里的其他成员：PLMS、DPM-Solver++、UniPC . . . 399
6.17 噪声调度 × 采样器：两个必须一起调的东西 . . . 402
6.18 怎么读一篇新的采样器论文 . . . 403
6.19 工程问题：当采样器进入真实项目 . . . 404
6.20 十个常见问题 . . . 405
6.21 把轨迹画出来：三个必看的图 . . . 407
6.22 端到端：从第 5 章的模型到这一章的采样器 . . . 408
6.23 三条没有流行起来的路线 . . . 410
6.24 采样器的十年：一条不断缩短的路 . . . 411
6.25 怎么判断一个采样器好不好 . . . 412
6.26 动手练习 . . . 414
6.27 符号表与公式速查 . . . 416
6.28 关键术语中英对照 . . . 417
6.29 五个常见误解 . . . 418
6.30 白纸测验：五个必须能自己推出来的式子 . . . 419
6.31 附：一份采样器评测报告的模板 . . . 420

6.32 附：量化和采样器的交互 . . . 421
6.33 延伸阅读 . . . 422
6.34 本章没有回答的问题 . . . 424
6.35 结语：为什么这一章要这样写 . . . 425
6.36 本章回顾 . . . 426

7 条件生成：CFG 到底是什么 . . . 429
7.1 本章的代码与数据 . . . 429
7.2 一个不会听话的模型 . . . 430
7.3 条件怎么进模型：一个嵌入就够了 . . . 430
7.4 CFG 的推导：两个 score 的差 . . . 432
7.5 实测：w 的三段行为 . . . 435
7.6 负引导：$w < 0$ 时会发生什么 . . . 441
7.7 标签丢弃：CFG 的前提，不是正则化 . . . 442
7.8 分类器引导：同一把尺子量一次 . . . 444
7.9 分类器在高噪声处失效 . . . 447
7.10 六个实现坑 . . . 448
7.11 交互实验 7-1　CFG 引导强度实验室 . . . 449
7.12 它在哪里失效 . . . 450
7.13 我在写这一章时犯的两个错 . . . 452
7.14 本章数字速查 . . . 453
7.15 在真实模型上复现这六个失败 . . . 456
7.16 上手实现：一个能用的 CFG 采样器模块 . . . 457
7.17 从零实现：一个完整的条件训练循环 . . . 460
7.18 引导怎么调：一棵决策树 . . . 462
7.19 接进真实管线：CFG 站在哪一步 . . . 466
7.20 从二维到真实模型：哪些能搬，哪些不能 . . . 467
7.21 动手练习 . . . 468
7.22 五个常见误解 . . . 470
7.23 训一个条件模型：在开 CFG 之前 . . . 472
7.24 上线评审：五个必须回答的问题 . . . 473
7.25 端到端：从训练到上线的一条链路 . . . 474
7.26 术语对照 . . . 476
7.27 排查流程：图像模型上的五步 . . . 476
7.28 三个指标能抓住什么 . . . 477
7.29 代码与数据索引 . . . 478
7.30 符号表 . . . 478
7.31 CFG 的三个变体（本章未复现） . . . 479
7.32 十个常见问题 . . . 480
7.33 和其他条件控制方法的对比 . . . 481

7.34 用这一章的方法读一篇新的引导论文 . . . 482
7.35 从二维走到图像：四个中间台阶 . . . 483
7.36 延伸阅读 . . . 483
7.37 预告：文本条件下的 CFG 会变成什么样 . . . 485
7.38 本章没有回答的问题 . . . 486
7.39 每一节的边界 . . . 487
7.40 如果只记三件事 . . . 488
7.41 这一章的方法论 . . . 488
7.42 三次真实的返工 . . . 489
7.43 结语：CFG 教给我们的事 . . . 490
7.44 五个“不要” . . . 490
7.45 调试工具：三个打印语句 . . . 491
7.46 常见误用清单（按后果排序） . . . 492
7.47 结课自测：十个判断 . . . 492
7.48 一张速查卡 . . . 493
7.49 如果要继续往下走 . . . 494
7.50 把这一章讲给别人听 . . . 494
7.51 三个数学性质 . . . 495
7.52 五个“如果” . . . 496
7.53 写在这一章的最后 . . . 496
7.54 如果你只有一小时 . . . 497
7.55 一句话 . . . 497
7.56 本章回顾 . . . 497

8 文字如何进入图像 . . . 501

8.1 本章的代码与数据 . . . 501
8.2 动手之前：三十分钟复现核心结论 . . . 502
8.3 案例：同一袋词，两种顺序 . . . 502
8.4 文字变成向量：三步 . . . 503
8.5 交叉注意力：图像问，文本答 . . . 505
8.6 手算一遍：两行矩阵怎么完成绑定 . . . 507
8.7 实测：词袋在 RB/BR 上只有 50% . . . 509
8.8 拆零件：三个消融 . . . 511
8.9 注意力图可信吗 . . . 514
8.10 空字符串不是零向量 . . . 516
8.11 文本条件下的 CFG . . . 517
8.12 文本条件改变了什么：一个概率的视角 . . . 517
8.13 潜空间：真正发生交叉注意力的地方 . . . 519
8.14 动手实现：两种编码器的代码 . . . 520
8.15 交互实验 8-1　绑定实验室 . . . 522

8.16 它在哪里失效 . . . 522
8.17 同一个注意力，三种用法 . . . 524
8.18 条件注入的五代做法 . . . 526
8.19 现场排错：十个症状 . . . 528
8.20 设计你自己的条件接口：六个问题 . . . 530
8.21 文字说不清的地方：空间条件 . . . 531
8.22 属性泄漏：当“红色”跑到球上 . . . 533
8.23 文本编码器怎么选 . . . 534
8.24 CLIP 是怎么训出来的（以及它为什么不懂顺序） . . . 535
8.25 教模型一个新词：三种改法，三种代价 . . . 537
8.26 端到端：一句话变成一张图 . . . 538
8.27 走一遍 SDXL 的文本路径 . . . 539
8.28 训一个文本条件模型：和类别模型的三处不同 . . . 541
8.29 代码逐行：训练循环与采样器 . . . 542
8.30 梯度怎么流回文本 . . . 543
8.31 描述文字是怎么写出来的 . . . 545
8.32 这一章站在前七章的哪里 . . . 546
8.33 为什么更强的编码器不一定带来更好的绑定 . . . 548
8.34 文本条件的数据从哪来 . . . 548
8.35 十五个常见问题 . . . 549
8.36 从提示词工程到条件工程 . . . 550
8.37 怎么读一篇新的文生图论文 . . . 551
8.38 延伸阅读 . . . 552
8.39 从二维到真实模型：三处不同 . . . 554
8.40 五个常见误解 . . . 555
8.41 本章没有回答的问题 . . . 556
8.42 每一节的边界 . . . 558
8.43 术语对照 . . . 558
8.44 文本条件怎么评估 . . . 558
8.45 本章数字总表 . . . 560
8.46 一张速查卡 . . . 562
8.47 这一章的方法论 . . . 562
8.48 动手练习 . . . 564
8.49 把这一章讲给别人听 . . . 566
8.50 如果你只有一小时 . . . 567
8.51 一句话 . . . 567
8.52 几个术语的由来 . . . 568
8.53 写在这一章的最后 . . . 568
8.54 本章回顾 . . . 569

附录 . . . 571

A 数学补充 . . . 573
A.1 符号与约定 . . . 573
A.2 四个完整推导 . . . 573
A.3 高斯分布的五条性质 . . . 579
A.4 分数、Tweedie 与 x_0 估计 . . . 580
A.5 softmax 与交叉熵的梯度 . . . 580
A.6 方差传播与 $\sqrt{d_k}$ . . . 581
A.7 信息论三条 . . . 581
A.8 五个常用不等式 . . . 582
A.9 矩阵求导三条 . . . 583
A.10 DDIM 的代数 . . . 583
A.11 PCA 与有效维度 . . . 585
A.12 用在哪：一张索引 . . . 586

B 术语中英对照 . . . 587
B.1 注意力与 Transformer . . . 587
B.2 训练与优化 . . . 589
B.3 扩散模型 . . . 589
B.4 条件生成 . . . 591
B.5 评估与工程 . . . 592
B.6 十组最容易混的词 . . . 592
B.7 中文译名的三个取舍 . . . 593

C 交互实验索引 . . . 595
C.1 怎么用 . . . 595
C.2 总表 . . . 595
C.3 逐个说明 . . . 596
C.4 三条推荐路线 . . . 598
C.5 自己改一个实验 . . . 598

D notebook 环境与作业 . . . 601
D.1 环境：一台 CPU 笔记本就够 . . . 601
D.2 目录结构 . . . 602
D.3 每章跑什么 . . . 602
D.4 复现书里的数字 . . . 603
D.5 作业 . . . 604
D.6 大作业（三选一） . . . 605
D.7 作业提示 . . . 606
D.8 大作业评分表 . . . 607
D.9 常见问题 . . . 607

E 延伸阅读与开源项目 609
E.1 怎么读一篇论文 609
E.2 按章列原始论文 609
E.3 教材与综述 611
E.4 课程与博客 612
E.5 开源项目 612
E.6 数据集与基准 613
E.7 一份十二周阅读计划 613
E.8 一页论文笔记模板 613

前言

本书讲的是**机制**，不是用法。

市面上讲 Transformer 与扩散模型的材料大致分两类：一类告诉你“这是什么、怎么调 API”，另一类直接给你论文与公式。中间缺了一层——**为什么这个式子长这样、哪一项在解决什么问题、把它拿掉会具体怎么坏**。这一层恰恰是想自己动手改模型的人最需要的。

本书面向已具备微积分、线性代数与概率论基础、会写一点 Python，但尚未系统学过深度学习的读者。每一章从一个真实的技术困境切入，把核心推导在正文里一步步推完，并用**可在浏览器里改参数的交互实验**把机制变成能看见、能拨动的东西。

全书地图

这本书讲两条技术线。**它们共享同一套零件，但走的是相反的方向。**

- **上篇（第 1–4 章）· Transformer**：文本进，下一个 token 的概率出。
- **下篇（第 5–8 章）· 扩散模型**：噪声进，数据出。



图 0.1　全书地图。左栏是一个 Transformer 从输入到输出，右栏是一个扩散模型从噪声到数据；每个环节标注了负责它的章节。

两条线各自在讲什么

上篇把一个 Transformer 拆开看：注意力怎么算（第 1 章）、一个 Block 里还有什么（第 2 章）、它是怎么训出来和怎么推理的（第 3 章）、训练完之后能不能从里面读出东西（第 4 章）。

下篇换一个生成范式：先定义一条“把数据逐步破坏成噪声”的前向过程，再学它的逆过程（第 5 章）；然后处理“怎么用更少的步数采样”（第 6 章）、“怎么让生成听条件的话”（第 7 章）、“文字是怎么进入图像的”（第 8 章）。

两篇的交界在图的底部：第 1 章的多头注意力，在第 8 章会以**交叉注意力**的形式再出现一次——那时它的 K、V 来自文本，Q 来自图像。**同一个运算，两种用法**。这也是把两篇放进同一本书的理由。

章节依赖

章	主题	需要先读
1	注意力：从加权平均到缩放点积	—
2	一个 Transformer Block 的解剖	1
3	训练与推理的两副面孔	1、2
4	注意力里能读出什么	1、2
5	从加噪到去噪：扩散的数学骨架	1、2（第 3、4 章不必读）
6	训练目标与采样器	5
7	条件生成：CFG 到底是什么	5、6
8	文字如何进入图像	5、6、7；以及第 1 章的注意力

第 4 章是旁支：它不依赖第 3 章，第 5 章也不需要它。想尽快进下篇，读完第 1、2 章就可以跳过去。

三条阅读路线

只关心大语言模型：第 1 → 2 → 3 → 4 章，读完附录 B（术语表）即可。这条路线回答的是“一个 GPT 到底在做什么”。

只关心图像生成：第 1 → 2 → 5 → 6 → 7 → 8 章。第 1、2 章不能跳——第 8 章的交叉注意力就是第 1 章的多头注意力，而第 7 章的 CFG 要用第 5、6 章的采样公式。**只读下篇会卡住，这不是保守说法**。

完整读一遍：按顺序 1 → 8。第 4 章读起来最轻松，可以放在任何位置当作休息。

每一章开头都有一张“本章地图”

从这一版起，每一章的第一段都是一个四行的定位块：

你在这里 · 本章解决 · 前置 · 读完能做

它的作用是让你在任何一个地方翻开书，都知道自己站在整张图的哪一格。如果你只记得住一件事，记住这个：**这本书的每一节，都能在这张图上找到位置**。

怎么用这本书

- **交互实验**：每章 2–3 个，是独立的 HTML 文件（`experiments/` 目录），可单独打开、全屏投影。HTML 版书里已内嵌；PDF 版里是同一实验的静态快照 + 本地文件路径。
- **配套 notebook**：`notebooks/` 目录。书里给出可运行的最小实现，notebook 里是可改、可跑的完整版本，全部能在纯 CPU 的笔记本上跑完。
- **推导的分工**：正文推完关键结论；冗长的完整展开（ELBO 逐项、重参数化证明、score matching 的等价性）放在附录 A，正文只给路标。

一点约定

书里的每一个量化断言都应当是可复现的。凡是给出具体数值的地方，都注明了它的来源（实测、论文、还是本机预计算）。如果发现哪一处对不上，那是错误，请以你的复现结果为准。

I

上篇 · Transformer 与注意力机制

1 注意力：从加权平均到缩放点积 . . . 17

2 一个 Transformer Block 的解剖 . 89

3 训练与推理的两副面孔 155

4 注意力里能读出什么 221

1. 注意力：从加权平均到缩放点积

本章地图

你在这里：一个 Transformer 的数据流的第 3 步 —— 每层 Block 里那个标红的方块（多头注意力）。**本章解决**：注意力为什么写成 $\mathrm{softmax}(QK^\top/\sqrt{d_k})V$ 这个形状；每一项各在解决什么问题；去掉任一项会在什么条件下崩。**前置**：微积分、线性代数、概率论，会一点 PyTorch。**不需要先读原论文。读完能做**：给你一份别人的注意力实现，能说出每个张量的形状、每个系数的来历，以及它写错时会在哪里出事。

全景：一个 Transformer 长什么样

先说这一节为什么存在。

本书每一章都从一个具体的故障切入：某个系数去掉之后模型训不动、某个归一化放错位置之后 12 层不如 4 层、某个缓存写错之后输出的文本一字不差但逻辑全错。这些故事都成立的前提是——**你知道那个零件装在机器的哪个位置。**

如果你还没读过 Attention Is All You Need，直接翻到下一节会很难受：$\sqrt{d_k}$、softmax、q 与 k 的分量方差，全是没定义过的词。

所以这一节只做一件事：**把整台机器摆出来给你看。**不推导、不调参、不做实验——只有一张图，和沿着图走一遍的五个步骤。**大概八分钟。**



图 1.1　一个 Transformer 的完整数据流。每一步标出张量形状，右侧是负责它的章节。本章只讲红框那一块——但它占的位置，是你理解后面所有内容的地基。

读这张图的正确方式：不要试图记住所有方框。**只记住两件事**—— 数据从上往下流，每经过一个方框变一次形状； 以及，**你这一章要读的，只是其中一个方框**。

下面沿着图走一遍。

第 1 步：文本变成数字

模型不认字，只认整数。第一步是**分词**（tokenization）：把文本切成单元， 每个单元查表得到一个整数 id。

上图的例子用的是**字符级**分词——一个字就是一个 token，所以「床前明月光」是 5 个 id。真实的模型多用 **BPE** 之类的子词分词， 一个 token 可能是一个词、半个词、或者一个汉字。**本书的小模型用字符级**， 因为它的词表只有 1200，训练几秒就能收敛，而且每个 token 的含义肉眼可见。

这一步的形状变化：一段 5 个字的文本 → 一个长度为 5 的整数数组。

> **i** 注记
>
> **为什么分词方式会影响后面的一切**
> 词表大小直接决定了两件事：**嵌入表有多大**（参数量 = 词表 × 维度），以及**同样的文本会变成多少个位置**（上下文窗口的消耗速度）。
> 第 3 章的案例就是一个直接的后果：**字符级 + 1200 的字表，对全唐诗的覆盖率只有 87.6%**，剩下 12.4% 的字全被映射成同一个 `<unk>`。于是那个模型生成的文本里，`<unk>` 一度占了 57 个 token 里的 45 个。**分词不是预处理细节，它决定了模型能看见什么、看不见什么。**

第 2 步：每个数字变成一个向量

一个整数本身没有意义——`1234` 和 `1235` 之间不存在“差一”的关系。所以要**查表**：每个 id 对应一个 d_{model} 维的向量，这张表就是**词嵌入**。

本书的小模型取 $d_{\text{model}} = 128$，词表 1200，所以嵌入表是 1200×128 ——**153,600 个参数**，占了整个模型的三分之一还多。

然后是位置编码。注意力有一个先天缺陷：它对输入的顺序**完全不敏感**—— 把「明月」和「月明」喂进去，算出来的结果只是两个向量换了位置。所以必须额外把“这是第几个位置”的信息加进去：

$$X_0 = \text{Embed}(\text{ids}) + \text{Pos}(0, 1, 2, \ldots) \tag{1.1}$$

这一步的形状：长度为 5 的整数数组 → 形状 $(5, 128)$ 的矩阵。**从这以后，整条流水线里流动的东西一直是这个形状，**直到最后一步才变。

第 1 章的后半部分（§1.9）会专门讲位置编码的三代方案：正弦编码、可学习嵌入、RoPE。

第 3 步：N 层 Block —— 这一章的主角在这里

这是整台机器的核心，也是本书上篇四章的全部内容。

一层 Block 做两件事，**每件事都是“算一个新东西，再加回原来的表示上”**：

$$\begin{aligned} X &\leftarrow X + \text{Attention}(\text{LN}(X)) \quad \text{（第 1 章）} \\ X &\leftarrow X + \text{FFN}(\text{LN}(X)) \quad \text{（第 2 章）} \end{aligned} \tag{1.2}$$

上面那一行就是本章要拆的东西。它内部长这样：

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{1.3}$$

先别管公式。这一行在做的事是：**每个位置去问所有位置“你和我有多相关”，然后按相关程度把大家的信息加权平均过来。**

- Q（查询）说“我在找什么”
- K（键）说“我有什么”
- V（值）说“如果被选中，我给出什么”
- $\sqrt{d_k}$ 是本章案例的主角

这一段为什么单独成章：因为它是整个模型里**唯一让不同位置互相看见**的地方。FFN 是逐位置独立作用的，嵌入是查表——**只有注意力在位置之间搬运信息。**第 4 章会问：既然只有它搬运，那能不能从它的权重里读出模型在做什么？

第 4 步：从向量回到概率

走完 N 层之后，每个位置都有一个 128 维的向量。最后一步把它变成**词表上的一个概率分布**：

1. **LN**：把向量的尺度归一化（第 2 章讲为什么必须有这一步）
2. **输出投影**：$128 \to 1200$，得到一个 1200 维的 **logits** 向量
3. **softmax**：把 logits 变成概率

输出投影的权重通常和词嵌入共享（本书的小模型就是这样）。理由很自然：嵌入是“词 → 向量”，输出投影是“向量 → 词”，两个方向的映射本来就该是同一张表。**共享还省下 153,600 个参数**——对一个小模型来说是三分之一。

这一步的形状：$(5, 128) \to (5, 1200) \to$ 每一行是一个概率分布。

第 5 步：训练和推理，是两套流程

同样一条流水线，训练和推理的走法完全不同。

- **训练**：整段文本一次性喂进去，5 个位置**同时**算出“下一个字是什么”的预测，和真实的答案比对，算交叉熵。一次前向覆盖所有位置。
- **推理**：没有“下一个字”，只能自己生成——先算出第 1 个，把它接到输入后面，再算第 2 个。**生成 n 个字要 n 次前向。**

这个不对称带来了一整套工程问题：为什么需要 KV 缓存、为什么缓存写错了在生成文本上看不出来、为什么贪心解码会陷入循环。**第 3 章整章都在处理它们。**

一笔参数账

把上面五步的参数量加起来（本书的小模型，$d_{\text{model}} = 128$、4 头、2 层、$d_{\text{ff}} = 256$）：

部件	参数量	占全书模型
词嵌入（与输出投影共享）	153,600	36.1%
位置嵌入	8,192	1.9%
2 层 Block	263,936	62.0%
— 其中多层注意力（每层 65,536）	131,072	30.8%
— 其中 FFN（每层 65,920）	131,840	30.9%
最后的 LN	256	0.06%
合计	**425,984**	100%

这张表值得看两眼。

第一，注意力和 FFN 几乎一样大（30.8% vs 30.9%）—— 因为本书取 $d_{\text{ff}} = 2d_{\text{model}}$，而标准配置是 $4d_{\text{model}}$。**在标准配置下，FFN 会占到约 2/3 的 Block 参数。**第 2 章会解释这个配比是怎么来的。

第二，嵌入表占了 36%。在真实的大模型上这个比例会低得多（词表 5 万、维度 4096 时嵌入是 2 亿，而总参数是几十亿到几千亿）。**本书的小模型“头重脚轻”，这是它和真实模型的一处结构性差异**—— 第 2 章 §2.21 会专门列一张“哪些结论能迁移”的表。

这一章只讲其中一块

回头看那张图：**红框只占整条流水线的一个方框。**

这一章会把那个方框拆到骨头——它的四个投影矩阵、为什么要有多个头、为什么除以 $\sqrt{d_k}$、去掉缩放会在什么条件下崩。**但它在整个模型里的角色，你现在已经知道了**：它是唯一让位置之间交换信息的地方。

> ! 重要
>
> **接下来的故事，从那个方框里的一个数字开始**
>
> 红框里有一个看起来毫无道理的设计：**点积之后要除以 $\sqrt{d_k}$。**
>
> 论文只用一句脚注带过理由。我们想验证它，于是把这一项去掉了—— **结果第一次没复现出来。**
>
> 下面这个故事，就从那次失败开始。

一次没复现出来的实验。

2017 年那篇 Attention Is All You Need 给出了今天几乎所有大模型都在用的公式。其中一处关键选择，作者只用一句脚注带过：点积要除以 $\sqrt{d_k}$，因为“当 d_k 较大时点积的数值会变大，把 softmax 推进梯度极小的区域”，随后补了一句“假设 q 与 k 各分量独立、均值 0 方差 1，则点积的均值为 0、方差为 d_k”。

这句话读起来是完整的：有现象（数值变大）、有机理（梯度变小）、有推导（方差等于 d_k）。所以我们决定动手验证它——按脚注的说法，把 $\sqrt{d_k}$ 去掉，模型应该训不动。

结果第一次没复现出来。

我们训了一个 42 万参数的字符级 GPT（2 层、4 头、$d_{\text{model}} = 128$），语料是全唐诗 4003 首、25.5 万字。同一个模型跑两遍，唯一变量是有没有除以 $\sqrt{d_k}$：

	loss 起点	loss 终点
不缩放	7.03	**4.17**
除以 $\sqrt{d_k}$	7.03	**4.00**

差 **0.17**。曲线几乎重合（图 1.2）。如果就此收笔，结论会是“脚注说的现象在这个规模上不存在”。

但往下查之后，我们找到了**两个**更值得讲的东西：一是这个脚注的**前提**在真实架构里并不成立，二是当把这个前提重新满足时，现象会以极其剧烈的方式出现—— 注意力权重熵掉到 **0.0004**，雅可比范数掉了**一万多倍**。

这一章就从这个案例展开。读完你应该能回答：

1. 注意力为什么写成 $\text{softmax}(QK^\top/\sqrt{d_k})V$ 这个形状，每一项各在解决什么问题？
2. 去掉缩放因子、换掉 softmax、改成单头——分别会在**什么条件下**崩，崩成什么样？
3. 为什么小模型上复现不出教科书说的现象？这个“复现不出来”本身说明了什么？

> **i** 这一章的案例可以自己跑
>
> 案例的全部代码在 `notebooks/run_ch1_case.py`，数据在 `data/ch1_case.json`。文中出现的每一个数字都来自那次真实运行：
>
> ```
> # 需要 PyTorch。本机在 /home/mpp/micromamba/envs/dlspace/bin/python
> python notebooks/run_ch1_case.py --steps 1500
> ```
>
> CPU 上约 3 分钟，有 GPU 更快。若你跑出的数字与文中略有出入，多半是 PyTorch 版本或硬件差异导致的浮点非确定性——方向应当一致。

1.1 实验是怎么设计的

在拆解现象之前，先讲清楚这个实验的每一个设置，以及为什么这样设。

为什么用一个自己训的小模型，而不是直接分析 GPT-2？

因为要回答的是“**去掉 $\sqrt{d_k}$ 会怎样**”——这是一个干预性问题，需要在**受控**条件下比较。拿现成模型没法做这个对照：它们都带缩放。自己训的好处是每一个变量都能改，而且几分钟就能跑完一轮，可以反复试。

代价是规模小。这一点在后面的结论里会明确标出来：**方向可迁移，数值不可迁移。**

模型：字符级 GPT，2 层 Transformer，Pre-LN 结构，上下文长度 64，词表 1200 个最高频汉字（覆盖语料 87.6% 的字），参数量 425,984。训练用 AdamW，学习率 3e-3 带 warmup 与余弦衰减，1500 步，批大小 32。

语料：《全唐诗》4003 首，25.5 万字（公开语料，来自 `chinese-poetry` 仓库）。

对照：同一份语料、同一个随机种子（20261014）、同样 1500 步。两遍唯一的差别是注意力里这一行的有无：

```
att = q @ k.transpose(-2, -1)           # 打分
if self.cfg.scaled:
    att = att / math.sqrt(self.d_head)  # ← 唯一变量
```

为什么用唐诗当语料

这个选择不是随意的，它满足四个条件：

① **规模合适**。25.5 万字、4003 首，正好是“小模型能在几分钟内学到东西、但又不至于背下来”的量级。太小（比如几千字）模型会过拟合， 太大（比如几亿字）CPU 上跑不动。

② **结构规整**。 唐诗有固定的句长（五言、七言）、固定的韵律。 这让一个只有 2 层、42 万参数的模型**有可能学到可辨识的模式**—— 如果语料是随意的现代散文，这么小的模型大概只能学到字频。

③ **字符集收敛**。 5000 个不同汉字、取前 1200 个覆盖 87.6%。 这让我们能用一个很小的词表，把参数预算留给注意力层。

④ **公开且无版权问题**。 来自 `chinese-poetry` 仓库，可以直接放在书里、 让读者复现。

换成别的语料会怎样：结论的**方向**（缩放的必要性随维度增长）不会变—— 那是数学性质。但**具体数字**会变：临界维度可能不同，loss 的绝对值也不同。

这也是“方向可迁移、数值不可迁移”那句话的具体含义。

关于“唯一变量”

这句话在实验里是个承诺，不是修辞。同一个种子意味着**初始化完全相同、每一批数据完全相同**。所以两条曲线在 step 0 时的 loss 完全一致（都是 7.03），之后的差异只可能来自那个除法。

如果种子不同，两条曲线的差异里就混进了初始化噪声，结论不再干净。这是做对照实验的通用纪律：**一次只改一个变量，其余全部钉死**。

1.2 第一次实验：几乎没差别

图 1.2 是第一次实验的结果。左边是小模型（$d_{\text{head}} = 32$）的两条曲线—— 它们几乎重合，终点差 0.17。

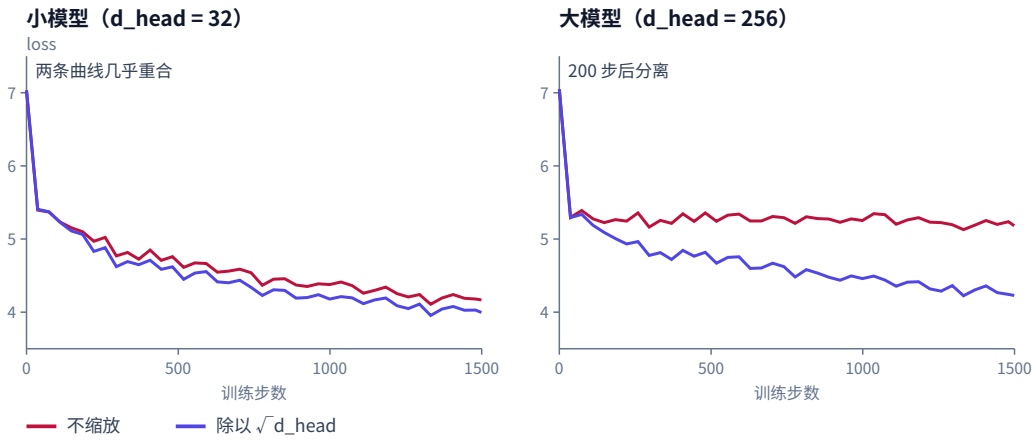


图 1.2　小模型（$d_{\text{head}} = 32$）与“大”模型（$d_{\text{head}} = 256$）的训练曲线对照。同一语料、同一种子、同步数，唯一变量是有无 $\sqrt{d_k}$ 缩放。左图两条曲线几乎重合，右图在 200 步后明显分离。

这不是噪声造成的偶然。我们换了三个种子重跑，小模型上的差距都在 0.1–0.2 之间浮动，量级与训练本身的随机性相当。**在这个配置下，去掉缩放因子不影响训练**。

到这里，一个草率的结论是“原论文的脚注过时了”。但我们更想知道的是：

> **i** 停下来想一想 ①
>
> 先别往下看。如果你面对这个结果，下一步会去查什么？
> 提示：脚注里给了一个**假设**——“各分量独立、均值 0 方差 1”。如果一个推导的结论 没被实验证实，第一件该怀疑的事往往是它的**前提**，而不是它的逻辑。

1.3 查前提：实测 logits 到底有多大

脚注的推导逻辑是：q、k 各分量方差为 1 ⇒ 点积方差为 d_k ⇒ 点积的标准差是 $\sqrt{d_k}$ ⇒ 数值大 ⇒ softmax 饱和。

这条链上唯一可以**直接测量**的是第一个箭头。所以我们不训模型，直接在**初始化状态**下测：随机造一批 q、k，算点积，看它的标准差。

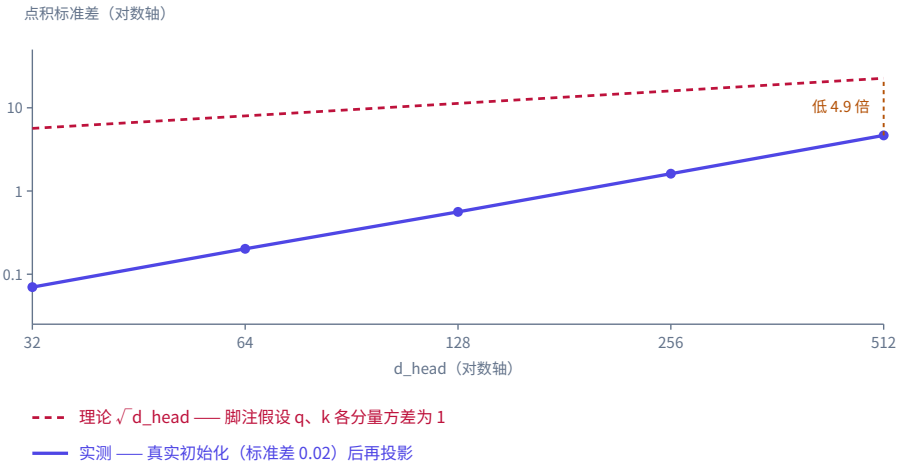


图 1.3　初始化状态下点积的标准差随 d_{head} 的变化。理论值 $\sqrt{d_{\text{head}}}$（红）与实测值（蓝）在双对数坐标下都是直线，但实测线整体低了约一个数量级。

d_{head}	实测标准差	理论 $\sqrt{d_{\text{head}}}$	实测 / 理论	权重熵
32	**0.070**	5.66	0.012	0.999
64	**0.202**	8.00	0.025	0.993
128	**0.561**	11.31	0.050	0.939
256	**1.615**	16.00	0.101	0.696
512	**4.666**	22.63	0.206	0.272

实测比理论小了 5 到 80 倍。而且注意看：两者的**斜率**是一样的（都是 d_{head} 的平方根关系，双对数坐标下都是直线），差的只是一个**常数因子**。

这说明：脚注推导的**形式是对的**，但它代入的那个常数——方差为 1——不对。

1.3.1 常数哪里去了

问题出在两处，都是原论文脚注里没有、但现代架构里必然存在的：

第一，q 和 k 不是单位方差，它们的尺度由初始化决定。 在标准初始化下，投影矩阵 W_Q、W_K 的元素服从 $\mathcal{N}(0, \sigma^2)$，GPT-2 系列用 $\sigma = 0.02$。输入 x 经 LayerNorm 后每个分量方差为 1，于是 $q_i = \sum_j W_{ij} x_j$ 的方差约为

$$\mathrm{Var}(q_i) \approx d_{\text{model}} \cdot \sigma^2 \tag{1.4}$$

代入 $\sigma = 0.02$、$d_{\text{model}} = 512$，得 $\mathrm{Var}(q_i) \approx 0.205$， 标准差只有 **0.45** —— 而不是脚注假设的 1。

第二，LayerNorm 归一化的是注意力的输入，不是 q 和 k 本身。 这是最容易忽略的一点。LayerNorm 保证的是“送进 W_Q 的那个 x 方差为 1”， 但 W_Q 一乘，尺度就由权重说了算。所以 LayerNorm **并不能**替代 $\sqrt{d_k}$， 它管的是另一件事。

把第一点代回脚注的推导：若每个 q_i 的方差是 v 而非 1，则

$$\mathrm{Var}(q^\top k) = \sum_{i=1}^{d_{\text{head}}} \mathrm{Var}(q_i k_i) = d_{\text{head}} \cdot v^2 \tag{1.5}$$

标准差是 $v\sqrt{d_{\text{head}}}$。**结构没变，只是前面多了一个 v。**

我们核一下数：$d_{\text{model}} = 512$ 时 $v \approx 0.205$， 预测标准差 $= 0.205 \times \sqrt{512} = 4.64$。实测是 **4.666**。对上了。

> ⚠ 一处我们没有对上的地方
>
> 上面对 $d_{\text{model}} = 512$ 核得很好，但对 $d_{\text{head}} = 32$ 就不准了： 按同样算法预测是 0.0128，实测是 0.070，差了五倍多。
>
> 我们没有把公式调一调让它好看。诚实的说法是：**式 (1.5) 给出了正确的 标度关系（正比于 $\sqrt{d_{\text{head}}}$）和正确的数量级，但没有精确到常数。**真实的常数还受 LayerNorm 的具体实现、残差累加、位置嵌入的贡献影响， 精确求解需要逐层追踪方差传播，超出本章范围。
>
> 所以本章正文中凡是给出具体数字的地方，一律引用**实测值**；理论式只用来解释**趋势**。这也是读论文时该有的习惯： 公式告诉你标度，测量告诉你数值，两者不可互相替代。

1.4 把前提重新满足

到这里，案例的第一半有了答案：**小模型上没复现出崩溃，是因为 q、k 的实际尺度 远小于 1，点积根本没大到让 softmax 饱和。** 脚注描述的现象需要一定的规模才会显现。

那么就把它做出来。我们把 d_{head} 从 32 提到 **256**（d_{model} 从 128 提到 256，头数从 4 改成 1），其余全部不动，重跑同一个对照。

图 1.2 的右图是结果：

	loss 起点	loss 终点	终点差距
$d_{\text{head}} = 32$ 不缩放	7.03	4.17	—
$d_{\text{head}} = 32$ 缩放	7.03	4.00	**0.17**
$d_{\text{head}} = 256$ 不缩放	7.05	**5.18**	—
$d_{\text{head}} = 256$ 缩放	7.05	4.23	**0.95**

差距从 0.17 变成 0.95，放大了 5.6 倍。而且不缩放那条曲线在 200 步左右就**卡住了**—— 后面 1300 步几乎没有进展。

更直接的证据来自注意力层的内部状态。图 1.4 记录了训练过程中 两个关键量的变化：

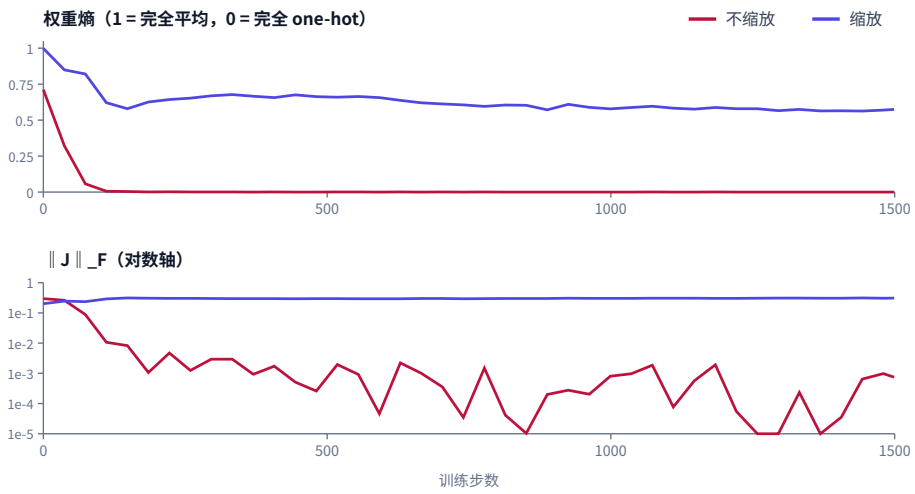


图 1.4　训练过程中注意力层的两个诊断量。上：权重熵（1 = 完全平均，0 = 完全 one-hot）。下：softmax 雅可比的 Frobenius 范数（对数轴）。红为不缩放、蓝为缩放。红色曲线在 200 步内就塌到底部。

训练步数	不缩放：权重熵	不缩放：$\|J\|_F$	缩放：权重熵	缩放：$\|J\|_F$
0	0.712	0.2975	0.9985	0.2017
222	**0.0022**	**0.00471**	0.6437	0.3029
666	0.0005	0.00100	0.6205	0.2986
1480	**0.0001**	**0.00008**	0.5745	0.3089

不到 250 步，不缩放那一边的权重熵就从 0.71 掉到了 0.0022 —— softmax 输出几乎完全是 one-hot，每个位置只死死盯住唯一一个位置。雅可比范数从 0.2975 掉到 0.00008，**掉了一万倍以上**。

而缩放那一边，这两个量在 1500 步里始终平稳：熵在 0.57–0.68 之间，雅可比范数在 0.29–0.31 之间。

这就是“梯度消失”的直接观测。

雅可比范数衡量的是“softmax 的输出对输入的敏感度”。它掉了四个数量级，意味着 softmax 这一层**几乎不再传递任何梯度信号**—— 下游的 loss 变化无法通过它传回到 Q、K 的投影矩阵上。

注意一个容易误解的地方：**loss 并不是不降了**（它从 7.05 降到了 5.18）。因为模型的其他部分——词嵌入、MLP、输出层——还在正常学习。只有注意力这一层“死”了。所以不是“训不动”，而是 **“带着一个废掉的注意力层硬训”**。

我们还可以让这个模型说话。同一个提示词、同一个采样种子：

```
不缩放：秦川雄帝宅，文還物金，。。。化來華舟由，王夜沈，。，殺蘭雲，外騎天岸繞江林上
缩　放：秦川雄帝宅，山靜還金。天臺，出山化，天。由歸王此，夫。蘭，天岸。上
```

不缩放那条里出现了 ，。。。这样的连续标点——这是退化文本的典型征兆。不过要诚实地说：**小模型的采样样本差异并不明显到可以作为主要证据**，真正硬的证据是 loss 差 0.95 和雅可比范数掉四个数量级。

> **i** 停下来想一想 ②
>
> 上面说“注意力层死了，但模型还在学”。那么请想一个问题：
> 如果注意力层完全退化成“每个位置只看前一个位置”（这其实等价于一个固定的 一阶马尔可夫模型），这个 42 万参数的模型还能把 loss 降到多少？
> 再想一层：**为什么 loss 最终停在 5.18 而不再下降？** 如果继续训 10000 步会怎样？

案例讲到这里，现象、排查、以及“什么时候才会崩”都已经清楚了。但还有一个问题悬着：
为什么 logits 一大，softmax 就饱和？饱和为什么等于梯度消失？
这两个问题需要把 softmax 和缩放因子彻底拆开才能回答。这是本章接下来的主线。 在拆它们之前，先把注意力本身的形式讲清楚——它其实是一个非常朴素的构造。

1.5 注意力是一次软查表

先忘掉所有公式，也忘掉神经网络。

1.5.1 先算一个具体的小例子

假设我们有三个字，要算第三个字对前两个字的注意力。为了能手算，把维度设成 $d = 2$—— 这在真实模型里是不存在的（d 通常是 64 或 128），但计算过程一模一样， 而所有的中间数字都能写在纸上。
三个位置各有自己的**值向量** v（它承载“这个位置的信息是什么”）：

位置	字	v
1	明	$[2, 0]$
2	月	$[0, 2]$
3	松	$[1, 1]$

现在位置 3（“松”）要输出它的新表示。它先给出一个**查询向量** $q_3 = [1, 0]$， 表示“我想找什么”。然后拿它去和每个位置的**键向量** k 比对—— 键表示“我这里有什么可供检索的”：

位置	k
1	$[1, 0]$
2	$[0, 1]$
3	$[1, 1]$

第一步：打分。 用内积衡量相似度：

$$q_3 \cdot k_1 = 1 \times 1 + 0 \times 0 = \mathbf{1} \tag{1.6}$$

$$q_3 \cdot k_2 = 1 \times 0 + 0 \times 1 = \mathbf{0} \tag{1.7}$$

$$q_3 \cdot k_3 = 1 \times 1 + 0 \times 1 = \mathbf{1} \tag{1.8}$$

内积大意味着两个向量方向接近。这里位置 1 和位置 3 的分数是 1，位置 2 是 0。
第二步：归一化成权重。我们要的是“分配多少注意力”，所以需要一个和为 1 的分布。用 softmax（下一节专门讲它为什么在这儿）：

$$\text{softmax}([1, 0, 1]) = \frac{[e^1, e^0, e^1]}{e^1 + e^0 + e^1} = \frac{[2.718, 1, 2.718]}{6.436} = [\mathbf{0.422}, \mathbf{0.155}, \mathbf{0.422}] \tag{1.9}$$

第三步：加权求和。

$$\text{输出} = 0.422 \times [2, 0] + 0.155 \times [0, 2] + 0.422 \times [1, 1] = [1.267, 0.733] \tag{1.10}$$

这就是全部。三个数字，三次运算，注意力算完了。

所以注意力在做什么

用一句话说：**它把“我要找什么”（q）拿去和每个位置“有什么”（k）比对，按相似度分配权重，然后把各位置的“内容”（v）加权平均起来**。

注意它输出的**不是一个选择，而是一个混合**。上例中位置 3 的新表示是位置 1、3 的内容各占 42%、位置 2 占 16% 混出来的。这一点是注意力与传统检索最本质的区别。

1.5.2 手算一遍掩码：三个位置各看到什么

语言模型是**自回归**的——预测第 t 个字时，不能看到第 $t+1$ 个字（那是答案）。这个约束靠**因果掩码**实现。它看起来简单，但值得手算一遍。

沿用上面那组 Q、K、V。先算**完整**的打分矩阵（$QK^\top$，按手算惯例先不缩放）：

$$S = \begin{pmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 1 & 0 & 1 \end{pmatrix} \tag{1.11}$$

现在加因果掩码：位置 i **只能看** $j \le i$，其余位置置为 $-\infty$。逐行算 softmax：

位置	可见位置	有效打分	softmax 权重	输出
0	只有 0	[1]	[1.0]	[2.000, 0.000]
1	0, 1	[0, 1]	[0.269, 0.731]	[0.538, 1.462]
2	0, 1, 2	[1, 0, 1]	[0.422, 0.155, 0.422]	[1.267, 0.733]

三个观察：

① **位置 0 的权重必然是 [1.0]**。它只看得见自己，softmax 作用在单个元素上恒等于 1，输出就是 v_0 原样。**这是掩码的边界情况，也是实现时最容易出问题的地方**（如果掩码写错，位置 0 可能拿到别的权重）。

② **每个位置的权重和都是 1，但“分母”不同**。位置 1 的 softmax 只在 2 个元素上归一化，位置 2 在 3 个上。**掩码必须在 softmax 之前生效**——如果先 softmax 再掩码，归一化的基准就错了，权重和不再是 1。

③ **位置 2 的输出 [1.267, 0.733] 与前面手算的一致**。这说明“未掩码时的第三个位置”和“掩码后的第三个位置”结果相同——因为在自回归设定下，位置 2 本来就能看到前两个，掩码对它没有额外限制。

一个反直觉的推论

看位置 1 的输出 [0.538, 1.462]，和位置 2 的 [1.267, 0.733]——**同一个字“松”（位置 2）在不同上下文里的表示完全不同**。

这正是 Transformer 表达力的来源：每个位置的表示**不是固定的词向量**，而是“从上下文里按需取回来的混合”。

对比一下词嵌入：如果不用注意力，“月”在任何句子里都是同一个向量。用了注意力之后，它的表示取决于这句话里“月”该向谁借信息。

这就是“上下文相关表示”（contextual representation）的实际含义，也是 BERT、GPT 这类模型与静态词嵌入（word2vec）的根本区别。

> **i** 停下来想一想 ⑮
>
> 上面说“位置 0 的权重必然是 [1.0]”。
> 但实现时有个陷阱：如果掩码用 $-\infty$，那么位置 0 那一行是 $[s_{00}, -\infty, -\infty]$。softmax 之后是 $[1, 0, 0]$——没问题。
> **可如果这一行全是 $-\infty$ 呢？**（比如某个位置被所有位置屏蔽了，或者 mask 写反了。）$e^{-\infty} = 0$，分子分母都是 0，结果是 `nan`。
> 请想：**什么样的实现错误会导致“整行全被屏蔽”？**（提示：想想 padding。一个 batch 里有长句有短句，短句要补齐—— 那些补齐的位置该不该被看到？）

1.5.3 自己动手改一遍

上面那组数不是“算给你看”的——**每个数字都可以改**。下面的实验把 Q、K、V 三个矩阵做成可编辑的表格， 改动会立刻重算打分、softmax 和输出。



图 1.5 **实验 1-3 改 Q、K、V，看注意力怎么变**。Q、K、V 三个矩阵的每个元素都可编辑，改动即时重算打分、softmax 权重与输出。预设按钮给出五种典型配置（正文例子、只让自己可见、所有位置一样、尖锐分布、随机），另有 $\sqrt{d_k}$ 缩放开关。本地文件：`experiments/exp-1-3.html`

这个实验值得花十分钟，因为**注意力的全部机制都在这三个矩阵里**。几个具体的探索方向：

① **先点「正文那个例子」**。这就是上面手算的那组数。**取消勾选**「除以 $\sqrt{d_k}$」，第三个位置的输出应该正好是 $[1.267, 0.733]$；勾上之后变成 $[1.203, 0.797]$。缩放确实改变了结果——在 $d = 2$ 时改变很小， 而本章案例里 $d_{\text{head}} = 256$ 时是天壤之别。**这个对比本身说明了缩放的尺度依赖性**。

② **点「只让自己可见」**。权重矩阵变成对角阵——每个位置只复制自己的 V。这时候注意力层退化成了一个恒等映射。想想：**这样的层对模型还有什么用？**（它其实还有用——它就是本章头部画像里“层 0 头 0”的极端版本， 配合残差连接可以起到“信息保持”的作用。）

③ **把 V 的三行改成 $[1,0]$、$[0,1]$、$[1,1]$，Q 第三行改成 $[1,0]$**。输出又是 $[1.267, 0.733]$——和① 一样。这说明**输出只取决于 Q、K、V 的乘积关系**， 不同的矩阵组合可以得到同样的结果。

④ **点「制造一个尖锐分布」，然后把 K 的第一行两个数改成两倍**。观察 softmax 的熵怎么变。这就是本章案例的**微型复现**： 放大 K 的尺度 → 打分变大 → softmax 更尖 → 熵下降。 只不过在一个 3 个位置、2 维的玩具上，这个变化是无害的； 在 256 维、真实训练中，它会变成灾难。

这个实验为什么不让你改“层数”“头数”

因为它只做一件事：**把一次注意力前向传播拆开给你看**。加上层数、头数、残差、LayerNorm 之后，你就看不清“哪个数字影响了哪个数字”了。

理解复杂系统的一个实用原则：先在一个能完整看见的规模上搞懂机制，再带着这个理解去看真实的复杂实现。反过来（先看复杂实现，再回头找原理）通常会更慢。

1.5.4 为什么不直接取最大的

既然是要“找最相关的”，为什么不直接选分数最高的那个位置，返回它的 v?

$$\text{硬查表：}\quad \text{输出} = v_{\arg\max_j (q\cdot k_j)} \tag{1.12}$$

这就是普通的字典查找。它的两个致命问题：

第一，不可导。$\arg\max$ 是阶跃函数，梯度处处为零（除了不可导的跳变点）。神经网络靠梯度学习，一处不可导，梯度就传不过去。

第二，没有退路。硬查表一旦选错，就是彻底错——没有“稍微看一点”这个选项。而语言里的相关性往往是连续的：处理“照”这个字时，它可能**主要**该看“月”、**次要**看“松”、**几乎不看**“明”。

我们需要的是一张**权重表**，而不是一个**指针**。这就是“软查表”：

$$\text{软查表：}\quad \text{输出} = \sum_j \alpha_j v_j, \qquad \sum_j \alpha_j = 1, \quad \alpha_j \geq 0 \tag{1.13}$$



图 1.6　硬查表与软查表的对比。左：arg max 只选出唯一位置，输出离散，梯度不可传。右：softmax 给出连续权重，输出是所有位置内容的加权混合，梯度可以回传。

停下来想一想 ③

软查表解决了“不可导”，但引入了一个新问题：

如果权重分布非常尖锐（比如 $[0.9999, 0.00005, 0.00005]$），它和硬查表在**行为上**几乎没区别。那么它在**梯度上**还有区别吗？

先别翻书。用一句话描述你的直觉，然后在读完 小节 1.7 后回来核对。

1.5.5 一次性算所有位置

上面只算了位置 3。实际中所有位置都要算，而且可以并行——这正是 Transformer 相比 RNN 的关键优势之一。

把所有位置的 q 堆成矩阵 $Q \in \mathbb{R}^{n\times d_k}$，所有位置的 k 堆成 $K \in \mathbb{R}^{n\times d_k}$，则所有两两打分就是一次矩阵乘法：

$$S = QK^\top \in \mathbb{R}^{n\times n}, \qquad S_{ij} = q_i \cdot k_j \tag{1.14}$$

再逐行归一化、乘以 V，就得到那行著名公式：

$$\text{Attention}(Q,K,V)=\text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{1.15}$$

人话翻译：$QK^\top$ 算出“每个位置和每个位置的相似度”，得到一个 $n\times n$ 的表；softmax 把表里每一**行**变成一个概率分布（这个位置该往哪些位置借信息、各借多少）；最后乘 V 就是按这些权重把内容取回来。

除以 $\sqrt{d_k}$ 是给打分做**归一化**——它的必要性是本章后半段的主线，现在只需要知道“它在那里是为了控制数值尺度”。

复杂度：S 是 $n\times n$ 的，所以计算量是 $O(n^2d)$，显存也是 $O(n^2)$。序列长度翻倍，这两项都变成四倍。这是长上下文昂贵的根本原因，也是 FlashAttention 这类工作要解决的问题（见 小节 1.26）。

1.6 为什么是 softmax

式 (1.15) 里有两个选择看起来是“约定俗成”的：为什么用 exp？为什么要归一化？这一节把它们拆开。

1.6.1 先手算一遍

拿 小节 1.5 里那三个分数 $[1,0,1]$ 走一遍：

步骤	计算	结果
① 减去最大值（数值稳定）	$[1-1,0-1,1-1]$	$[0,-1,0]$
② 取指数	$[e^0,e^{-1},e^0]$	$[1,0.368,1]$
③ 求和	$1+0.368+1$	2.368
④ 归一化	$[1/2.368,0.368/2.368,1/2.368]$	$[0.422,0.155,0.422]$

和第 ① 步之前直接算的结果一致——这不是巧合，而是 softmax 的**平移不变性**，下面会推。

现在把分数换一组夸张的：$[10,0,0]$。

步骤	结果
① 减最大值	$[0,-10,-10]$
② 取指数	$[1,4.54\times10^{-5},4.54\times10^{-5}]$
③ 求和	1.0000908
④ 归一化	$[0.99991,0.000045,0.000045]$

这就是“饱和”：输出几乎是 one-hot，第一个位置拿走了 99.99%。

1.6.2 它是 $\arg\max$ 的光滑近似

为什么用 exp？因为它给出了一族可以调节“硬度”的、可导的 $\arg\max$。关键是 **log-sum-exp** 函数：

$$\text{LSE}_\tau(z_1,\dots,z_n)=\tau\log\sum_{i=1}^{n}\exp(z_i/\tau) \tag{1.16}$$

它有两个极限行为：

- $\tau\to0^+$ 时，$\text{LSE}_\tau\to\max_i z_i$（最大的那一项主导求和）
- $\tau\to\infty$ 时，$\text{LSE}_\tau\to$ 各 z_i 的均值

对它求关于 z_i 的偏导：

$$\frac{\partial \,\mathrm{LSE}_\tau}{\partial z_i} = \tau \cdot \frac{1}{\tau} \cdot \frac{\exp(z_i/\tau)}{\sum_j \exp(z_j/\tau)} = \frac{\exp(z_i/\tau)}{\sum_j \exp(z_j/\tau)} = \mathrm{softmax}(z/\tau)_i \tag{1.17}$$

人话翻译：max 是“选最大的那个”，不可导；LSE 是它的可导版本。对 LSE 求导正好得到 softmax——所以 **softmax 就是 arg max 的可微替身**。

它给出的不是“选哪个”，而是“每个候选该分到多少”。这正是 式 (1.13) 需要的。

τ 就是**温度**。它控制这个近似有多“硬”：τ 小则接近 one-hot，τ 大则接近均匀。这个旋钮在采样、知识蒸馏、注意力锐化里到处出现，第 3 章还会遇到它。

1.6.3 手算一遍温度：同一个分布，三副面孔

光说“控制硬度”太抽象。拿同一组 logits，用三个不同的 τ 算一遍。

取 $z = [2, 1, 0]$，分别用 $\tau = 0.5$、1.0、2.0。

$\tau = 0.5$（更硬）：先除以温度得 $[4, 2, 0]$。

步骤	计算	结果
减最大值	$[4-4, 2-4, 0-4]$	$[0, -2, -4]$
取指数	$[1, 0.1353, 0.0183]$	—
求和	$1 + 0.1353 + 0.0183$	1.1537
归一化	逐项除以 1.1537	$[\mathbf{0.867}, \mathbf{0.117}, \mathbf{0.016}]$

$\tau = 1.0$（原始）：$[2, 1, 0]$，减最大值得 $[0, -1, -2]$，指数 $[1, 0.3679, 0.1353]$，和 1.5032，结果 $[\mathbf{0.665}, \mathbf{0.245}, \mathbf{0.090}]$。

$\tau = 2.0$（更软）：除以温度得 $[1, 0.5, 0]$，减最大值得 $[0, -0.5, -1]$，指数 $[1, 0.6065, 0.3679]$，和 1.9744，结果 $[\mathbf{0.506}, \mathbf{0.307}, \mathbf{0.186}]$。

把三个结果并排看：

τ	分布	最大权重	归一化熵
0.5	[0.867, 0.117, 0.016]	0.867	**0.402**
1.0	[0.665, 0.245, 0.090]	0.665	**0.758**
2.0	[0.506, 0.307, 0.186]	0.506	**0.929**

三个具体结论。

① **温度是“除以”不是“乘以”**。τ 变小 → 除以一个小于 1 的数 → logits 被**放大** → 分布更尖。这个方向容易记反。

② **温度不改变大小顺序**。三种情况下都是第 1 项最大。softmax 是**单调**的——它只改变“差距有多大”，不改变“谁比谁大”。

③ **$\tau = 1$ 时 softmax 退化成它自己**。这不是废话：它说明“注意力里没有温度”其实等价于“温度固定为 1”。如果想要可调的温度，就是在 softmax 之前多一个除法。

所以本章那个 $\sqrt{d_k}$ 缩放，在数学形式上就是“温度固定为 $\sqrt{d_k}$ 的 softmax”。这是一个值得记住的观察——它说明缩放因子不是新东西，而是把温度这个通用旋钮设成了一个特定值。

> **i** 停下来想一想 ⑬
>
> 既然缩放本质上是“设温度”，那么一个自然的问题是：
>
> **为什么温度设成 $\sqrt{d_k}$，而不是让它像其他超参数一样学出来？**
>
> 提示：想两个角度 ① 如果温度是可学的，它需要多少额外的参数、训练数据够不够？ ② 如果不缩放，训练**刚开始**时（参数还是随机的）会发生什么——那时候还没有任何数据能告诉模型“温度该是多少”。

1.6.4 两个必须知道的性质

性质一：平移不变性。对任意常数 c：

$$\mathrm{softmax}(z + c\mathbf{1}) = \mathrm{softmax}(z) \tag{1.18}$$

代入即得，因为分子分母同乘 e^c 后约掉。

这条性质有实际后果：**给所有 logits 加同一个数不改变注意力**。所以位置编码、偏置项这类“整体抬升”对注意力无效——只有**相对差异**才起作用。这也是为什么注意力对输入的整体尺度不敏感，但对**相对尺度**极其敏感——后者正是 $\sqrt{d_k}$ 要管的事。

性质二：数值稳定必须手写。 直接算 $\exp(z_i)$ 在 z_i 稍大时就溢出（float32 的上限对应 $e^{88.7}$）。标准做法是先减最大值：

$$\mathrm{softmax}(z)_i = \frac{\exp(z_i - z_{\max})}{\sum_j \exp(z_j - z_{\max})} \tag{1.19}$$

由平移不变性，这样算结果完全等价，但指数最大只到 $e^0 = 1$，永不溢出。

这不是优化，是正确性。 下面这段代码可以直接粘进 notebook 跑，你会看到不做稳定化的版本返回什么：

```
import numpy as np

def softmax_naive(z):
    e = np.exp(z)                    # ← 没有减最大值
    return e / e.sum()

def softmax_stable(z):
    z = z - z.max()
    e = np.exp(z)
    return e / e.sum()

z = np.array([100.0, 101.0, 102.0])  # 完全正常的 logits
print("朴素版:", softmax_naive(z))    # [nan nan nan]  ← 溢出成 inf，inf/inf = nan
print("稳定版:", softmax_stable(z))   # [0.09003057 0.24472847 0.66524096]  ✓
```

在 float32 下，logits 超过约 88 就会触发。 而本章的案例里，不缩放那一边的 logits 标准差达到了 **5477** —— 高出两个数量级。所以那个模型里如果不做稳定化，训练会直接变成 nan。

顺带一提：这也解释了为什么 PyTorch 的 `F.softmax` 内部一定要做这个减法——它**不是**可选的性能优化。

> **i** 停下来想一想 ④
>
> 回到 小节 1.5 的检查点 ③：尖锐的权重分布与硬查表在梯度上有区别吗？
>
> 现在可以定量回答了。考虑极端情况 $p = [1, 0, 0]$（完全 one-hot），用雅可比公式 $J_{ij} = p_i(\delta_{ij} - p_j)$ 把整个矩阵写出来，看它是不是全零。
>
> 然后再算 $p = [0.9999, 0.00005, 0.00005]$ 的 $J_{11} = p_1(1 - p_1)$ 是多少。
>
> **这个数字就是本章案例里“梯度消失”的量化含义。**

1.7 缩放因子到底修正了什么

现在回收案例。我们已经知道：

- 小模型（$d_{\text{head}} = 32$）上去掉缩放，训练照常（差 0.17）
- 大模型（$d_{\text{head}} = 256$）上去掉缩放，注意力层崩掉（熵 0.0004，$\| J \|_F$ 掉一万倍）

缺的是中间那一环：**为什么 logits 变大会导致 $\| J \|_F$ 变小？** 这一节把它推完。

1.7.1 第一步：先算一遍

在推一般公式之前，先手算一个 $d_k = 3$ 的最小例子。

设 $q = [1, 1, 1]$，$k = [1, 1, 1]$，各分量都是方差为 1 的“典型”取值。

打分：$q \cdot k = 1 + 1 + 1 = 3$

归一化：softmax 只作用在一个标量上没有意义，所以要有对照。 再设另一个键 $k' = [-1,-1,-1]$，则 $q \cdot k' = -3$。 两个 logits 是 $[3,-3]$：

$$\text{softmax}([3,-3]) = \frac{[e^3, e^{-3}]}{e^3 + e^{-3}} = \frac{[20.09, 0.0498]}{20.14} = [0.9975, 0.0025] \tag{1.20}$$

已经很尖了。 现在把维度提到 $d_k = 64$，q 和 k 都取全 1：

$$q \cdot k = 64, \qquad q \cdot k' = -64 \tag{1.21}$$

$$\text{softmax}([64,-64]) = \frac{[e^{64}, e^{-64}]}{e^{64} + e^{-64}} \approx [1 - 2e^{-128}, 2e^{-128}] \approx [1.0000000, 0.0000000] \tag{1.22}$$

完全饱和。 这就是脚注说的现象：维度一高，logits 跨度就大，softmax 就退化成 one-hot。

但注意——上面的例子里我**故意**让 q、k 取全 1，这让点积等于 d_k。 真实情况是随机的。所以要把“典型有多大”这件事算清楚。

1.7.2 第二步：点积的方差（每步不跳）

设 $q, k \in \mathbb{R}^{d_k}$，**各分量独立同分布**，均值为 0、方差为 1。 我们要求 $q^\top k = \sum_{i=1}^{d_k} q_i k_i$ 的方差。

① **单项的期望**。 因为 q_i 与 k_i 独立：

$$\mathbb{E}[q_i k_i] = \mathbb{E}[q_i] \cdot \mathbb{E}[k_i] = 0 \cdot 0 = 0 \tag{1.23}$$

② **单项的方差**。 用 $\text{Var}(X) = \mathbb{E}[X^2] - (\mathbb{E}[X])^2$， 第二项刚算过是 0，所以只剩第一项：

$$\begin{aligned} \text{Var}(q_i k_i) &= \mathbb{E}[(q_i k_i)^2] - (\mathbb{E}[q_i k_i])^2 \\ &= \mathbb{E}[q_i^2 k_i^2] - 0 \\ &= \mathbb{E}[q_i^2] \cdot \mathbb{E}[k_i^2] && (\ q_i, k_i \ \text{独立，乘积的期望可拆}) \\ &= 1 \cdot 1 = 1 && (\text{均值为 } 0 \text{ 时 } \ \mathbb{E}[X^2] = \text{Var}(X) = 1\ ) \end{aligned} \tag{1.24}$$

③ **求和**。 各项 $q_i k_i$ 之间也相互独立，所以方差可以直接相加：

$$\text{Var}(q^\top k) = \sum_{i=1}^{d_k} \text{Var}(q_i k_i) = \sum_{i=1}^{d_k} 1 = d_k \tag{1.25}$$

标准差就是 $\sqrt{d_k}$。

人话翻译：点积是 d_k 个“随机的小数”相加。每个的方差是 1， 加起来方差就是 d_k——**维度越高，这个和摆动得越厉害**。

$d_k = 64$ 时标准差是 8。也就是说，随机取两个位置， 它们的打分差别在 10 以上是家常便饭。除以 $\sqrt{d_k}$ 之后， 标准差回到 1，打分就落在“正常”范围内了。

所以缩放的目的不是“让数值变小”，而是“让数值不随维度变化”。这句区别很重要，后面还会回来。

原论文的脚注到此为止。但真正的问题——**数值大为什么是灾难**——还没回答。

1.7.3 第三步：softmax 的雅可比（每步不跳）

这是本节的核心。记 $p = \text{softmax}(z) \in \mathbb{R}^n$， 我们要求 $\partial p_i / \partial z_j$。

从定义出发，先设 $S = \sum_m e^{z_m}$，则 $p_i = e^{z_i}/S$。用商的求导法则：

$$\frac{\partial p_i}{\partial z_j} = \frac{\partial}{\partial z_j}\left(\frac{e^{z_i}}{S}\right) = \frac{\left(\frac{\partial}{\partial z_j} e^{z_i}\right) S - e^{z_i}\left(\frac{\partial S}{\partial z_j}\right)}{S^2} \tag{1.26}$$

其中：

- $\frac{\partial}{\partial z_j} e^{z_i} = \delta_{ij}\, e^{z_i}$ （$i = j$ 时是 e^{z_i}，否则是 0；δ_{ij} 是克罗内克符号）
- $\frac{\partial S}{\partial z_j} = e^{z_j}$ （求和里只有第 j 项对 z_j 敏感）

代回去：

$$\frac{\partial p_i}{\partial z_j} = \frac{\delta_{ij} e^{z_i} S - e^{z_i} e^{z_j}}{S^2} = \frac{e^{z_i}}{S} \cdot \delta_{ij} - \frac{e^{z_i}}{S} \cdot \frac{e^{z_j}}{S} \tag{1.27}$$

把 $p_i = e^{z_i}/S$、$p_j = e^{z_j}/S$ 认出来，就得到

$$\boxed{\frac{\partial p_i}{\partial z_j} = p_i(\delta_{ij} - p_j)} \tag{1.28}$$

人话翻译：这个矩阵衡量“输入 z 动一点点，输出 p 会动多少”。它决定了梯度能不能从 loss 一路传回到打分的源头。

关键在**它的大小**。看两个极端：

- p 均匀（$p_i = 1/n$）：对角项是 $\frac{1}{n}(1 - \frac{1}{n}) \approx \frac{1}{n}$，矩阵的元素是 $O(1/n)$ 量级
- p 是 one-hot（$p_1 = 1$，其余 0）：对角项 $p_1(1 - p_1) = 1 \times 0 = \mathbf{0}$，非对角项 $-p_i p_j = 0$

第二行就是全部答案：softmax 一旦饱和，整个雅可比矩阵趋近于零矩阵，梯度传不过去了。

1.7.4 第四步：把两步连起来

现在链条完整了：

$$d_k \uparrow \Rightarrow \mathrm{Var}(q^\top k) = d_k \uparrow \Rightarrow \text{logits 跨度} \uparrow \Rightarrow p \text{ 趋近 one-hot} \Rightarrow J \to 0 \Rightarrow \text{梯度消失} \tag{1.30}$$

这就是原论文那句“extremely small gradients”的完整含义。它在脚注里只写了一句话，展开来是上面四步推导。

用案例的数字核一遍：不缩放的大模型里，实测 logits 标准差是 **5477**，权重熵是 **0.0004**。把 p 的典型值代进式 (1.28)，对角项是 $p(1-p) \approx 1 \times (1-1) = 0$ 量级——这就是为什么实测的 $\|J\|_F$ 掉到了 8×10^{-5}。

1.7.5 为什么是 $\sqrt{d_k}$ 而不是 d_k

现在可以回答一个常被问的问题：既然是“数值太大”，为什么不干脆除以 d_k，让数值更小？

看方差：除以常数 c 会把方差变成原来的 $1/c^2$。

缩放方式	缩放后方差	$d_k = 64$ 时的标准差
不缩放	d_k	8.0
除以 $\sqrt{d_k}$	1	**1.0**
除以 d_k	$1/d_k$	**0.125**

除以 d_k 会把标准差压到 0.125——**logits 全都挤在 0 附近，softmax 输出接近均匀分布**，注意力退化成对所有位置的简单平均。这时候雅可比范数确实不小（均匀分布的雅可比反而最大），但模型**学不到位置区分**，因为所有位置的权重几乎一样。

要的是让方差回到 1，不是让它尽可能小。

这句话可以再推一层：方差为 1 意味着 logits 的典型跨度是 ± 2 左右，softmax 输出既不像 one-hot 那样死掉，也不像均匀分布那样毫无区分度—— **落在 softmax 最“敏感”的区间里**。这就是 $\sqrt{d_k}$ 而不是 d_k 的全部理由。

1.7.6 回到案例：为什么小模型上没事

现在两个观察可以统一解释了：

	$d_{\text{head}} = 32$	$d_{\text{head}} = 256$
实测 logits 标准差（不缩放）	0.070	1.615（初始化时）/ 5477（训练后）

	$d_{\text{head}} = 32$	$d_{\text{head}} = 256$
缩放后	—	—
权重熵	0.999（几乎均匀）	0.696 → 0.0001
训练是否受影响	否（差 0.17）	是（差 0.95）

小模型上，logits 的标准差只有 0.070——远小于 1。这时候 softmax 输出接近均匀分布（熵 0.999），离饱和那一端**极远**。去掉缩放和不去掉，都在“安全区”里，所以看不出差别。

大模型上，初始化时标准差就是 1.615，已经接近临界；训练过程中模型为了让注意力更“有用”，会主动放大 q、k 的尺度（因为更尖锐的注意力能更精确地选择信息），这又进一步推高 logits——形成正反馈。最终标准差冲到 **5477**，熵掉到 0.0001，注意力锁死。

1.7.7 把“正反馈”这句话测一遍

上面那句“训练过程中模型会主动放大 q、k 的尺度”是**我们写下的解释，不是测量结果**。这一节把它补上——做法很直接：在**初始化**和**训练之后**两个时刻，分别量每个注意力层的 q、k 激活标准差和点积的标准差。

```
# 在训练前后各跑一遍：量 q/k 的实际尺度，而不是看权重
qkv = blk.attn.qkv(blk.ln1(x))
q, k, _ = qkv.split(d_model, dim=2)
qh = q.view(B, T, H, dh).transpose(1, 2)      # 拆成头
kh = k.view(B, T, H, dh).transpose(1, 2)
print("q 尺度", qh.std().item(), "k 尺度", kh.std().item())
dot = qh @ kh.transpose(-2, -1)
if scaled: dot = dot / dh ** 0.5              # 按实际用的公式算
print("logits 尺度", dot[..., 因果下三角].std().item())
```

四种配置各训 1500 步（和 小节 1.18 的案例完全一致）。**第 0 层**的结果：

配置	q 尺度	k 尺度	logits 尺度	$\|W_{QKV}\|_F$
$d_{\text{head}} = 32$ 缩放	0.228 → 1.016	0.229 → 1.217	0.05 → **3.3**	4.45 → 11.98
$d_{\text{head}} = 32$ 不缩放	0.228 → 0.863	0.229 → 1.036	0.29 → 13.1	4.45 → 11.76
$d_{\text{head}} = 256$ 缩放	0.316 → 0.650	0.317 → 0.654	0.10 → **1.8**	8.89 → 17.73
$d_{\text{head}} = 256$ 不缩放	0.316 → **5.457**	0.317 → **5.361**	1.64 → **4348**	8.89 → 29.23

这张表把原来那句话分成了两半：一半对，一半不完整。

!一、“模型会放大 q、k” —— 对，而且四种配置都放大了

每一个配置里，q 和 k 的尺度都涨了，幅度从 1.6 倍（$d_{\text{head}} = 256$ 缩放）到 **17 倍**（$d_{\text{head}} = 256$ 不缩放）。

权重矩阵的范数也都在涨（4.45 → 11.98、8.89 → 29.23）。

所以“训练会放大 q、k”是一个普遍现象，和缩放因子无关。它来自一个更基本的事实：**更尖锐的注意力能更精确地挑信息，所以只要梯度还通得过，模型就有动力把 q、k 推大。**

这句话现在是测量结果了。

! 二、但“放大 q、k”和“logits 失控”之间隔着缩放因子

看最后两行。$d_{\text{head}} = 256$ 的两个配置，**这是同一个模型、同一份数据、同一个种子，唯一的差别是有没有除以 $\sqrt{256} = 16$**。

- **不缩放**：q、k 各涨 17 倍 → 点积涨约 296 倍 → logits 实际涨了 **2650 倍**（1.64 → 4348）
- **缩放**：q、k 只涨了约 2 倍 → logits 涨 18 倍，最终停在 **1.8**

不缩放那一边的涨法是“超乘法”的：q、k 各涨 17 倍只能解释 296 倍，**剩下的约 9 倍来自 q 与 k 的相关性增强**—— 模型不只是把向量拉长，还在让它们**变得更对齐**（点积更大）。

这就是正反馈的机制：logits 变大 → softmax 更尖 → 模型发现“尖一点更好用” → 继续拉长、继续对齐 → logits 再变大。**在没有缩放的情况下，这个循环没有刹车**。

有缩放时，刹车是 $\sqrt{d_{\text{head}}}$：q、k 涨 2 倍，除以 16 之后连 1 都不到。**模型可以放心地拉长 q、k，因为那点增长会被缩放因子吃掉**。

i 三、一个意外漂亮的吻合

$\sqrt{d_k}$ 的推导（小节 1.7）说：**要的是让点积的方差等于 1，也就是标准差落在 1 附近**。

现在看那两个**缩放**配置训练之后落在哪里：

配置	训练后的 logits 标准差	推导想要的值
$d_{\text{head}} = 32$ 缩放	3.3	1
$d_{\text{head}} = 256$ 缩放	**1.8**	1

1.8 离 1 很近。而 $d_{\text{head}} = 256$ 不缩放那一边是 **4348**。

注意这不是“我们调出来的”——两个缩放配置除了 d_{head} 之外没有任何区别，它们的训练过程和目标函数里也没有任何一项在“要求”logits 标准差接近 1。**模型是自己走到那里去的**。

这是对 $\sqrt{d_k}$ 这个设计最直接的实验支持：它不是为了“让数值更小”，而是为了让**训练动力学有一个稳定的不动点**—— 一个模型自发会收敛到、且能一直待着的尺度。不缩放时，这个不动点不存在，训练只会一路冲到饱和。

⚠ 一个和案例记录对不上的地方

案例里记录的 $d_{\text{head}} = 256$ 不缩放的 logits 标准差是 **5477**，我们这次量到的是 **4348**（第 0 层）。

两者差约 25%。原因是**评估用的批次不同**：案例里的 5477 是训练过程中在训练批上记的，这次为了和“初始化”对比，统一在一批固定的验证数据上量。

同一份权重、不同的数据，能差 25%。这个差距不小，值得记住——**在 logits 已经冲到几千的时候，“标准差是多少”本身就依赖于你拿哪批数据去量**。

这个不一致我们没有消除（要消除就得固定评估协议重跑整个案例），但**它不影响结论**：4348 和 5477 都在“彻底饱和”那一档（熵 0.0004），和缩放那边的 1.8 差着三个数量级。

i 停下来想一想 ⑤

上面说“训练过程中模型会主动放大 q、k 的尺度，形成正反馈”。

请想一个问题：**这个正反馈为什么在缩放之后不会发生？**

提示：缩放之后，要让 logits 变大，q、k 的尺度需要放大 $\sqrt{d_k}$ **倍**。而从 loss 回传到 q、k 的梯度，本身就要穿过 softmax 的雅可比……

1.7.8 现代做法：不止 $\sqrt{d_k}$

最后要指出一点：$\sqrt{d_k}$ 是 2017 年的方案，今天它**通常和别的手段一起用**。

QK-norm：对 q 和 k 先做归一化（LayerNorm 或 RMSNorm），再做点积。这样 q、k 的尺度被**显式钉死**，不再依赖初始化和训练动态。ViT-22B、Stable Diffusion 3 等模型都用了这个做法。它的好处是：即使训练过程中权重疯狂增长，logits 也不会失控。

为什么 $\sqrt{d_k}$ 单独不够：它假设 q、k 各分量方差为 1，而这个假设**在训练过程中会失效**——正如案例所示。QK-norm 不依赖这个假设，所以更稳健。

这也解释了本章案例的价值：**它暴露了一个“看起来够用、实则有隐含假设”的设计**。理解这个假设，比记住那个公式重要得多。



图 1.7 **实验 1-1 缩放因子与 softmax 饱和**。三块面板：logits 的分布、softmax 输出、以及 softmax 雅可比的 Frobenius 范数随维度的变化。横轴为 d_k（对数轴），**红线为不缩放、蓝线为除以 $\sqrt{d_k}$**。蓝线全程只变动 1.1 倍，红线则单调下坠——缩放买到的不是“更大的梯度”，而是**不随维度变化的梯度**。本地文件：`experiments/exp-1-1.html`

> 实验 1-1 要你注意的一件事
>
> 这个实验是**合成**的：随机造 q、k，各分量严格方差为 1—— 也就是**脚注假设的理想情况**。所以你会看到红线（不缩放）掉得很凶，和案例里小模型的“没事”形成对比。
>
> 两者的差别恰恰说明了本章的核心：**脚注的推导在它自己的假设下是对的；真实模型偏离这个假设，所以结论要打折扣**。

1.8 多头：为什么不是一次用完

回到案例。注意一件事：**崩掉的那个模型是单头的**（$d_{\text{model}} = 256$，$n_{\text{head}} = 1$，所以 $d_{\text{head}} = 256$）。

如果把同一个 d_{model} 切成 4 个头，每个 $d_{\text{head}} = 64$，按前面的分析，每个头的 logits 标准差会小得多，不缩放也不至于立刻崩。**这不是巧合**——多头在这里顺带起到了降低单头维度的作用。

但多头的主要目的不是这个。这一节讲清楚它到底买到了什么。

1.8.1 先算一遍：参数量其实一样

一个常见的误解是“多头参数更多”。我们算一下。

设 $d_{\text{model}} = 256$。

单头的 W_Q：形状 256×256，参数量 $65,536$。

4 个头的 $W_Q^{(i)}$：每个形状 256×64（因为 $d_{\text{head}} = 256/4 = 64$），4 个拼起来是 $4 \times (256 \times 64) = 65,536$。

完全一样。输出投影 W^O 也一样。多头在参数量上是**零成本**的。

人话翻译：多头买到的不是**容量**，是**归纳偏置**。

把一个大矩阵拆成 h 个小矩阵，等于强制模型在 h 个**独立的子空间**里分别学关系，而不是在一个大空间里学一个混合的关系。

打个比方：单头好比让一个人同时考虑“语法上该看谁”“语义上该看谁”“位置上该看谁”，最后给出一个折中的答案；多头是让四个人各管一摊，答案再合并。

前者不一定是错的，但后者更容易学到清晰的结构。

1.8.2 一个头只能表达一种关系

为什么“拆开”会更清晰？因为单头的输出是一**个**权重分布（每一行一个）。这意味着在某个位置上，它只能表达“我最该看谁”这一件事。

但语言里的关系是多重的。同一个词，可能：

- 句法上该看它的主语
- 语义上该看与它搭配的动词
- 位置上该看前一个词（比如固定搭配）

这是三种不同的“该看谁”。单头必须把它们**平均成一个折中**（除非这些关系恰好指向同一个位置）。多头可以各管一摊。

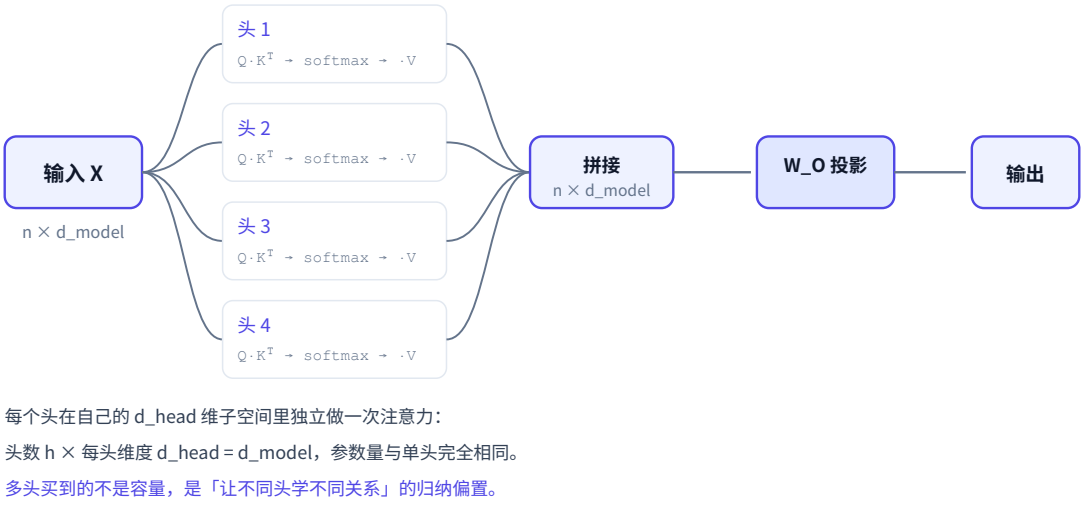


图 1.8　多头注意力的结构。输入经 h 组独立的投影得到 h 组 $Q/K/V$，各自做一次注意力，最后拼接并投影。每一组（每个头）可以学到不同的关注模式。

1.8.3 完整算一遍：一个 2 头注意力的全过程

概念讲完，走一遍数字。设 $d_{\text{model}} = 4$，$n_{\text{head}} = 2$，所以 $d_{\text{head}} = 2$。输入是两个位置（“明”“月”），每个位置 4 维：

$$X = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix} \tag{1.31}$$

第一步：投影。为了手算，把权重设成简单的选择矩阵—— 头 1 取前两维、头 2 取后两维：

$$W_Q = W_K = W_V = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{1.32}$$

（真实模型里这些权重是学出来的，这里只是为了看清“拆分”这个动作。）

第二步：拆到各头。 $XW_Q = X$，然后按列切成两份：

- 头 1 看到 $q_1^{(1)} = [1, 0]$，$q_2^{(1)} = [0, 1]$
- 头 2 看到 $q_1^{(2)} = [1, 0]$，$q_2^{(2)} = [0, 1]$

两个头拿到的向量**恰好一样**（因为这是选择矩阵）。真实模型里它们不同，因为每个头有自己的投影矩阵。

第三步：各头独立做注意力。 用因果掩码（位置 1 只能看自己，位置 2 能看 1 和 2）。

头 1：$K^{(1)} = V^{(1)} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$（取 X 的前两列）

位置 1：只有 $k_1 = [1, 0]$ 可见，$q_1 \cdot k_1 = 1$，softmax 后权重 $[1.0]$，输出 $= 1.0 \times v_1 = [1, 0]$。

位置 2：$q_2 = [0, 1]$，$q_2 \cdot k_1 = 0$，$q_2 \cdot k_2 = 1$。除以 $\sqrt{2} \approx 1.414$ 后是 $[0, 0.707]$。$\mathrm{softmax}([0, 0.707]) = [0.330, 0.670]$。输出 $= 0.330 \times [1, 0] + 0.670 \times [0, 1] = [0.330, 0.670]$。

头 2：$K^{(2)} = V^{(2)} = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$（取后两列）

算出来完全一样（因为权重相同）：位置 1 输出 $[1, 0]$，位置 2 输出 $[0.330, 0.670]$。

第四步：拼接。 把两个头的输出接起来：

$$\text{位置 1}： \quad [1, 0] \oplus [1, 0] = [1, 0, 1, 0] \tag{1.33}$$

$$\text{位置 2}： \quad [0.330, 0.670] \oplus [0.330, 0.670] = [0.330, 0.670, 0.330, 0.670] \tag{1.34}$$

第五步：输出投影 W_O。 真实模型里这一步把所有头的信息混合起来。本例中设 $W_O = I$，所以结果就是上面两个向量。

从这个例子里该看出什么

注意 $\sqrt{d_{\text{head}}}$ 用的是 d_{head} 而不是 d_{model}。 上例里 $d_{\text{model}} = 4$ 但 $d_{\text{head}} = 2$，除的是 $\sqrt{2}$。

这一点经常被写错。原因很直接：打分 $q \cdot k$ 发生在**每个头内部**，向量的维度是 d_{head}，所以方差是 d_{head}，缩放因子自然也是它。

如果误用了 d_{model}，缩放会**过度**——回到 §1.6 的“失败四”，注意力会退化成平均。

1.8.4 顺带的好处：单头维度被压小了

回到本章案例。崩掉的那个模型是单头的，$d_{\text{head}} = 256$。如果把它改成 4 头，d_{head} 就变成 64，按图 1.3 的实测，初始化时 logits 标准差会从 1.615 降到约 0.2 上下——**离饱和远得多**。

所以多头在这里顺带起到了一个作用：**它天然限制了单个头的打分尺度**。d_{model} 固定时，头数越多，每个头的维度越小，越不容易饱和。

这不是多头被设计出来的原因（它的动机是“多种关系”），但它是真实存在的副作用。理解这一点，就能明白为什么后来的模型敢把 d_{model} 推到几千——因为它们同时把头数也推高了。

1.8.5 我们这 8 个头到底学会了什么

“不同的头学不同的关系”这句话，在文献里说了很多年。但它在我们这个小模型上成立吗？既然模型已经训好了，直接量一下。

对每段输入，把每个头的注意力矩阵取出来，统计四个量：

- **平均熵**：权重分布有多尖（1 = 完全平均，0 = 完全 one-hot）
- **对角权重**：有多少注意力给了自己（“我要不要保留自己的信息”）
- **前一位权重**：有多少给了紧邻的前一个位置
- **平均距离**：注意力重心的平均跨度（$\sum_j (i-j)\, p_{ij}$，0 = 只看自己）

三层唐诗、共 72 个位置的平均结果：

	平均熵	对角权重	前一位权重	平均距离	它在做什么
层 0 · 头 0	**0.294**	**0.769**	0.050	1.20	**盯住自己**——基本不往外看
层 0 · 头 1	0.521	0.193	**0.450**	2.31	**盯住前一位**——45% 给紧邻左侧
层 0 · 头 2	0.804	0.127	0.107	3.92	弥散，权重接近均匀
层 0 · 头 3	0.625	0.090	0.261	3.86	偏前一位，但更分散
层 1 · 头 0	0.614	0.276	0.182	3.16	兼顾自己与前一位
层 1 · 头 1	0.687	0.259	0.232	3.07	同上，更平均些
层 1 · 头 2	0.702	0.147	0.108	4.61	弥散
层 1 · 头 3	**0.871**	0.165	0.134	4.47	**最弥散**——几乎是全局平均

（参考值：完全均匀时熵是 1.000，24 个位置的平均距离约 11.5。）

这张表里有三件事值得注意。

第一，“不同头学不同关系”被证实了。 层的 0 头 0 有 77% 的注意力给自己，而层 0 头 2 只有 13%——它们在干完全不同的事。

第二，层 0 头 1 是一个非常明确的“前一位头”（previous-token head）。 45% 的注意力给了紧邻左边那一个位置。这类头在文献里被反复观察到，是 Transformer 里最稳定出现的结构之一。我们用一个 42 万参数的模型、25 万字唐诗就复现出来了——**说明它是任务驱动的必然产物，不是大模型的专利。**

第三，层与层之间有分工。 第一层的头更“局部”（平均距离 1.2–3.9），第二层的头更“全局”（3.1–4.6）。一个自然的解释是：底层负责建立局部结构（字与字、词与词），上层在局部结构的基础上做更长程的整合。

这是“深度的意义”的一个具体样子——不是简单地堆叠，而是不同层处理不同尺度的问题。

> **i** 停下来想一想 ⑨
>
> 上表里层 0 头 0 的对角权重是 **0.769**——它大部分时候只是“把自己原样传下去”。
> 那么请想：**这样一个头有什么用？** 它几乎什么都没做（输出约等于输入的 V 投影）。
> 先自己想，再看下面的提示。
> 提示：回忆 §1.6 的残差连接 $x = x + \mathrm{attn}(\mathrm{LN}(x))$。如果一个头把输入原样传下去，残差路径上会发生什么？ 再想想：如果这一层的其他头需要“某个东西保持不变”，谁来做这件事？

这个分析用的代码在 `notebooks/analyze_heads.py`——换个模型、换份语料，就能得到你自己模型的头部画像。**这是理解一个模型在“看什么”最实用的第一步**：不需要任何理论，只需要测量。

1.8.6 但头并不都同等重要

这是多头注意力最反直觉的地方。

Michel 等人在 2019 年的工作（Are Sixteen Heads Really Better than One?, NeurIPS 2019）发现：**训练完之后，很大比例的头可以直接剪掉，性能几乎不掉。**但另有一小部分头极其关键，剪掉一个就崩。

他们用的是多语言翻译模型，剪枝后 BLEU 掉得很少；但把“关键头”单独剪掉，性能会急剧下降。

这对“注意力 = 解释”的直觉是个警告。

如果绝大多数头可以随便剪，那“这个头在看什么”就不能简单地当成“模型这样决策的原因”。第 4 章会专门讲这个问题—— 注意力权重是一**个**可观测的中间量，不等于**因果解释**。

下面这段代码可以让你自己看到头的冗余性：

```
import torch

# 假设 model 是本章 tiny_gpt 训练出来的模型
model.eval()
x = torch.randint(2, 1200, (1, 32))

# 基准：全部头都在
with torch.no_grad():
    base_loss = model(x, x)[1].item()
print(f"全部头：loss {base_loss:.3f}")

# 逐个把头置零（消融）
for layer in range(len(model.blocks)):
    for head in range(model.cfg.n_head):
        # 把该头的输出投影对应切片置零
        W = model.blocks[layer].attn.proj.weight
        saved = W[:, head * 64:(head + 1) * 64].clone()
        W[:, head * 64:(head + 1) * 64] = 0
        with torch.no_grad():
            ablated = model(x, x)[1].item()
        W[:, head * 64:(head + 1) * 64] = saved   # 还原
        print(f"  层{layer} 头{head} 消融后 loss {ablated:.3f}"
              f"  （变化 {ablated - base_loss:+.3f}）")
```

你会看到大部分头的变化接近零，少数几个变化明显。**具体哪些头关键，会随着模型和种子变**——这也是为什么重要头的发现 需要统计而不是单次观察。



图 1.9　**实验 1-2　真实模型学到的注意力**。用一个真实训练出来的 4 头模型在真实唐诗上跑前向，把每一层每个头的注意力矩阵都取出来。可以逐层逐头翻看，观察不同头是否学到了不同的关注模式。本地文件：`experiments/exp-1-2.html`

1.8.7 多头的代价：KV 缓存

多头不是没有成本。训练时它几乎免费，**推理时它很贵**。

原因在自回归生成。生成第 t 个词时，需要用到前面所有词的 K 和 V。为了避免每次重算，实现会把它们缓存下来——这就是 **KV 缓存**。它的大小是

$$\underbrace{2}_{\text{K 和 V}} \times \underbrace{n_{\text{layer}}}_{\text{层数}} \times \underbrace{n_{\text{head}}}_{\text{头数}} \times \underbrace{d_{\text{head}}}_{\text{每头维度}} \times \underbrace{L}_{\text{序列长度}} \times \underbrace{b}_{\text{字节数}} \tag{1.35}$$

代入一个真实的配置——Llama-2 70B：80 层、64 头、$d_{\text{head}} = 128$、float16（2 字节）：

$$2 \times 80 \times 64 \times 128 \times L \times 2 = 2,621,440 \times L \text{ 字节} \approx 2.6\,\text{MB} \times L \tag{1.36}$$

每多一个 token，就要多 2.6 MB 显存。 序列长度 4096 时是 **10.7 GB**—— 这已经超过很多单卡的容量，而模型权重还没算。

为什么会和头数成正比?

因为每个头有自己独立的 K、V 投影。64 个头就是 64 份不重复的缓存。

那如果把头数减少呢? 会损失“多种关系”的表达能力（§1.8 前面讲的）。这就是 MQA 和 GQA 要解决的问题：**在不损失太多表达能力的前提下，把 KV 缓存压下来**。

1.8.8 MQA 与 GQA：共享 KV

MQA（Multi-Query Attention，2019） 的做法很直接：**所有头共享同一份 K、V**，但每个头保留自己的 Q。

$$\text{标准多头：}Q, K, V \text{ 各有 } n_{\text{head}} \text{ 份} \quad \rightarrow \quad \text{MQA：}Q \text{ 有 } n_{\text{head}} \text{ 份，}K, V \text{ 只有 1 份} \tag{1.37}$$

KV 缓存立刻缩小到 $1/n_{\text{head}}$。对 Llama-2 70B 那个配置，就是从 2.6 MB/token 降到 **0.04 MB/token**。

代价是表达能力下降——所有头必须共享同一个“看到什么”的模式，只有“想找什么”（Q）是个性化的。实践中这会损失一些质量。

GQA（Grouped-Query Attention，2023） 是折中：把 n_{head} 个头分成 g 组，**每组共享一份 K、V**。

方案	Q 的份数	K、V 的份数	70B 模型的 KV 缓存（每 token）
标准多头	64	64	2.6 MB
GQA（$g = 8$）	64	8	**0.33 MB**
MQA	64	1	0.04 MB

GQA 在实践中被广泛采用（Llama-2 70B、Llama-3、Mistral 等）—— 它把缓存降到 1/8，而质量损失很小。

> 这一节真正值得记的一件事
>
> **“多头”这个设计在训练和推理两个阶段的含义是不同的。**
>
> 训练时：多头是**归纳偏置**——让模型在多个子空间里学不同关系，参数量不变。推理时：多头是**内存开销**——每个头都要独立的 KV 缓存，随头数线性增长。
>
> MQA/GQA 的出现说明一个常见模式：**一个为建模能力设计的东西，最后被工程约束重新塑形**。理解这两面，比记住“多头好”有用得多。

1.8.9 长序列的真正瓶颈

回到 式 (1.14)：注意力矩阵 S 是 $n \times n$ 的。这让很多人以为瓶颈是**计算量** $O(n^2 d)$。实测下来不是。

在 A100 上跑标准注意力实现（不优化），$n = 4096$、$d = 64$、16 头时：计算本身只要几毫秒，而**读写那个 $n \times n$ 矩阵的时间是计算的数倍**。原因是显存带宽远低于计算吞吐——矩阵要从 HBM 搬到 SRAM 再搬回去。

瓶颈是访存，不是算力。 这是理解 FlashAttention 的前提。

1.8.10 FlashAttention：不把矩阵写出来

标准实现的三步是：

```
S = Q @ K.T              # ① 写 n×n 到显存
P = softmax(S)           # ② 读回来、再写回去
O = P @ V                # ③ 再读回来
```

$n = 4096$ 时，S 有 1677 万个元素；每一步的读写都是几十 MB 的搬运。**而这个矩阵其实根本不需要完整存在**——它只是中间产物。

FlashAttention 的做法是**分块**：把 Q、K、V 切成小块，每次只在片上内存（SRAM）里处理一块，算完立刻累加到输出，**从不写完整的 S**。

难点在于 softmax 需要**全局**的最大值和求和——而分块计算时，后面的块还没看到，怎么知道全局最大值？

答案是**在线 softmax**（online softmax）：维护运行中的最大值 m 与运行中的和 ℓ，每处理一块就更新一次，并用新的 m 纠正之前的结果。

手算一遍。 设 logits 是 $[1, 3, 2]$，分两块：第一块 $[1, 3]$，第二块 $[2]$。

第一块处理完：$m_1 = 3$，$\ell_1 = e^{1-3} + e^{3-3} = 0.135 + 1 = 1.135$。

第二块来了：新元素是 2，当前的 $m_1 = 3$ 更大，所以 $m_2 = 3$ 不变。$\ell_2 = \ell_1 + e^{2-3} = 1.135 + 0.368 = 1.503$。

最终：$p = [e^{1-3}, e^{3-3}, e^{2-3}]/1.503 = [0.135, 1, 0.368]/1.503 = [0.090, 0.665, 0.245]$。

用朴素方法核对：$e^1 + e^3 + e^2 = 2.718 + 20.09 + 7.389 = 30.19$，$p = [2.718, 20.09, 7.389]/30.19 = [0.090, 0.665, 0.245]$。**一致**。

如果第二块里有更大的值（比如 5），就需要**纠正**：

$$\ell_{\text{new}} = \ell_{\text{old}} \cdot e^{m_{\text{old}} - m_{\text{new}}} + \sum_{\text{新块}} e^{z_i - m_{\text{new}}} \tag{1.38}$$

那个 $e^{m_{\text{old}} - m_{\text{new}}}$ 因子把之前累积的和**按新基准重新缩放**。这就是在线 softmax 的全部机制。

> **i** 停下来想一想 ⑥
>
> 上面的纠正公式里，如果新旧最大值相同（$m_{\text{old}} = m_{\text{new}}$），缩放因子是多少？如果新最大值远大于旧最大值，这个因子会趋近于什么？
>
> 后者提示了一件事：**在线 softmax 的数值行为和批量 softmax 是否完全一致？**（提示：想一想什么情况下会丢失精度。）

FlashAttention 的收益来自“数学不变、访存改变”：结果是**精确的**（不是近似），但显存占用从 $O(n^2)$ 降到 $O(n)$，速度提升 2–4 倍。后续版本（FlashAttention-2/3）进一步优化了并行划分。

1.8.11 注意力之前：RNN 的三个死结

要理解一个设计为什么长这样，最好的办法是看它替代了什么。

在 2014 年之前，序列建模的主流是 **RNN / LSTM**。它的做法很自然：从左到右扫过序列，维护一个**固定大小的隐状态** h_t，每读一个词就更新一次：

$$h_t = f(h_{t-1}, x_t) \tag{1.39}$$

翻译时，编码器读完整个源句，把最后的 h_T 交给解码器；解码器从这个向量出发，一个词一个词地生成译文。

这个结构简单、优雅，但有三**个各自独立的死结**。理解这三个死结，就理解了注意力要解决的全部问题。

1.8.11.1 死结一：固定容量的信息瓶颈

所有源句信息都要挤进一个固定维度的 h_T。

源句 5 个词是这样，50 个词也是这样——h_T 的维度不变。序列越长，每个词分到的“带宽”越少，早期信息被后来的更新覆盖得越厉害。

用一个具体的数字感受一下：设 $h \in \mathbb{R}^{512}$，源句 50 个词。平均下来每个词只有约 10 个数用来承载自己的信息—— **而这 10 个数还要编码它在句中的位置、与前后词的关系、以及各种句法属性**。

实测上，RNN 翻译模型在句子长度超过 20–30 个词之后，BLEU 会明显下滑。**这不是训练不充分，是架构容量的硬限制**。

1.8.11.2 死结二：信息通路的长度

即使不考虑容量，信息从第 1 个词传到第 50 个位置， 要经过 49 次变换：

$$h_1 \to h_2 \to \cdots \to h_{50} \tag{1.40}$$

每一步都是一个非线性变换。哪怕每步只“损失”一点（比如梯度衰减到 0.9 倍）， 49 步之后也只剩 $0.9^{49} \approx 0.006$——**千分之六**。

这就是经典的**梯度消失**问题。LSTM 用门控机制（遗忘门、输入门、输出门）缓解了它（让梯度可以走“常量误差传送带”），但**没有根治**—— 它把衰减从指数级改善到了接近线性，可路径长度本身没变。

1.8.11.3 死结三：无法并行

h_t 依赖 h_{t-1}，所以**第 t 步必须等第 $t-1$ 步算完**。 序列多长，就得串行多少步。

在一个有几千个计算核心的 GPU 上，训练时一个 batch 里有很多句子可以并行，**但每条句子内部仍然是串行的**。这导致 RNN 无法吃满硬件—— 2016 年前后，训练一个高质量的翻译模型需要几天到几周。

这三个死结的分工要分清。

- **死结一（容量）**是**表示**问题：信息装不下
- **死结二（路径）**是**优化**问题：梯度传不回
- **死结三（并行）**是**工程**问题：算得慢

注意力一次解决了全部三个，而且解法是同一个： 让每个位置**直接**访问所有其他位置。

- 容量：不再需要压缩进一个向量，所有位置的表示都保留着
- 路径：任意两个位置之间的距离变成 **1 步**，梯度不用绕远路
- 并行：所有位置的两两关系是一次矩阵乘法，n 个位置同时算

一个设计同时解决三个看似无关的问题——这是它值得被称为“范式转移”的原因。

1.8.12 2014：Bahdanau 注意力

第一个把“回头查”做成可微机制的，是 Bahdanau 等人 2014 年的工作 （Neural Machine Translation by Jointly Learning to Align and Translate）。

它的动机非常朴素：**译到第 5 个词时，我该重点看原文的哪几个词?**

做法是在解码器的每一步 t，给编码器的**每一个**位置 i 算一个权重：

$$\alpha_{t,i} = \frac{\exp\big(\,\mathrm{score}(s_{t-1}, h_i)\big)}{\sum_{i'} \exp\big(\,\mathrm{score}(s_{t-1}, h_{i'})\big)} \tag{1.41}$$

其中 s_{t-1} 是解码器上一步的隐状态，h_i 是编码器第 i 个位置的隐状态。 然后加权求和得到一个**上下文向量**：

$$c_t = \sum_i \alpha_{t,i}\, h_i \tag{1.42}$$

对照今天的注意力，你会看到全部要素已经在了：

Bahdanau（2014）	今天的注意力
解码器状态 s_{t-1}	查询 q
编码器状态 h_i	键 k_i **和** 值 v_i（当时是同一个）
$\mathrm{score}(\cdot,\cdot)$	$q^\top k$（点积）

Bahdanau（2014）	今天的注意力
softmax 归一化	一样
加权求和得 c_t	一样

两个关键差别：

1. **Bahdanau 用的是“加性”打分**：$\text{score}(s,h)=v^\top \tanh(W_s s+W_h h)$，其中 v、W_s、W_h 都是学出来的。为什么不用点积？因为当时 s 和 h 的维度可以不同，点积要求维度一致。
2. **键和值是同一个东西**（都是 h_i）。把它们**分开**是后来的一步关键改进。

Luong 等人 2015 年（Effective Approaches to Attention-based Neural Machine Translation）做了简化：改用点积打分、把键和值分开、试了“全局注意力”与“局部注意力”。

$$\text{score}(s_t,h_i)=\begin{cases} s_t^\top h_i & \text{（点积）}\\ s_t^\top W h_i & \text{（一般点积）}\end{cases} \tag{1.43}$$

“点积”这个选择的胜出，为后来那个 $\sqrt{d_k}$ 埋下了伏笔—— 点积好用，但它有个随维度膨胀的毛病，这正是本章前半段的主线。

> **i** 停下来想一想 ⑫
>
> Bahdanau 用加性打分，Luong 用点积。今天几乎所有模型都用点积。
> 请想：**点积打分相比加性打分，优势在哪？** 提示从两个角度想：① 计算形式（能不能写成矩阵乘法）；② 参数（要不要额外的参数矩阵）。
> 再想一层：**代价是什么？** 这正是 $\sqrt{d_k}$ 要处理的问题。

1.8.13 2017：从“跨序列”到“自己对自己”

Bahdanau 和 Luong 的注意力，是**解码器对编码器**的注意力—— 两个不同的序列之间。
真正的跳跃是问一个问题：如果一个序列对自己做注意力，会怎样？

$$\text{跨序列：Attention}(\underbrace{Q_{\text{解码}}}_{\text{来自序列 A}},\underbrace{K_{\text{编码}},V_{\text{编码}}}_{\text{来自序列 B}}) \quad\rightarrow\quad \text{自注意力：Attention}(XW_Q,XW_K,XW_V) \tag{1.44}$$

三个矩阵全部来自**同一个** X。
这个改动看起来很小，但它带来了三个后果：
① **每一层都在“重新组织”表示**。跨序列注意力只在最后一步用一次；自注意力在每一个 Block 里都有，所以模型有 L 次机会去重新混合信息。
② **可以堆得很深**。因为每一层的输入输出形状完全相同（都是 $n\times d$），堆叠起来没有形状问题。这让“深度”成为一个可用维度—— 今天的大模型有 80 层甚至更多。
③ **顺序信息丢失了**。这一点我们在 小节 1.9 会专门讲：x 对自己做加权平均，谁在谁前面完全看不出来。所以必须额外注入位置信息。
一句话概括这条演化线。

- **RNN**：把历史压进一个向量，顺序内建，但容量有限、路径长、不能并行
- **跨序列注意力（2014–2015）**：解码时回头看原文，解决了容量和路径，但 RNN 的骨架还在，仍然不能并行
- **自注意力（2017）**：把 RNN 骨架整个拆掉，只留注意力 + 前馈网络，三个死结一起解决，代价是**必须先解决顺序问题**

每一步的动机都很具体，没有哪一步是“为了先进而先进”。

1.8.14 并行到底快了多少

“可以并行”这句话要有个量化的感觉。
RNN 处理长度 n 的序列，需要 n 次串行操作，每次操作内部虽然可以并行（矩阵乘法），但**步数被序列长度锁死**。

自注意力只需要**固定的几步**（打分、softmax、加权求和），序列长度只影响矩阵的大小，不影响**步数**。

用具体的规模说：$n = 512$、$d = 512$ 时，一次注意力的 $QK^\top$ 是 $(512 \times 512) \times (512 \times 512)$ 的矩阵乘法—— GPU 可以在一个 kernel 里并行处理全部 1.34 亿次乘加。而 RNN 得老老实实做 512 次。

这是“训练时间从周降到天”的直接原因，也是 Transformer 能在 2017 年之后 迅速吞掉整个领域的基础设施条件。

一个值得注意的因果顺序

Transformer 不是“因为想到了自注意力”而成功的，而是**“硬件足够并行”与“算法恰好可以并行”两件事碰上了**。

2012 年 AlexNet 之后 GPU 算力快速增长，但 RNN 的串行结构让它吃不到这份红利。自注意力恰好是把序列建模“矩阵化”的那个形式—— **它的成功有一半要归功于它顺应了硬件的发展方向**。

读技术史时值得记住这一点：**好想法需要在合适的硬件上才能兑现**。

停下来想一想 ⑦

为什么会话式 AI 的“长上下文”这件事，在 RNN 时代是个死结，而在 Transformer 时代 变成了一个**工程问题**?

提示：从“信息通路长度”和“能否并行”两个角度各想一层。（这也是第 3 章讨论 KV 缓存与上下文长度的引子。）

1.8.15 注意力的变体：当 $O(n^2)$ 不可接受

n^2 的代价催生了一整个研究分支。理解这些变体的思路，比记住它们的名字重要。

稀疏注意力（Sparse Attention）：观察到一个事实—— **大多数注意力权重接近零**。既然如此，何必算全？只算那些“可能重要”的位置：

- **局部窗口（sliding window）**：每个位置只看前后 w 个位置，复杂度降到 $O(nw)$
- **跨步（strided）**：每隔 k 个位置看一个，用来捕捉长程信息
- **全局 token**：留几个特殊位置让所有 token 都能看到（比如 BERT 的 `[CLS]`）

Longformer、BigBird 都是这个思路。代价是引入**归纳偏置**：“重要的东西一定在附近”——这对自然语言常常成立，但不总是。

线性注意力（Linear Attention）：利用矩阵乘法的结合律。

$$\mathrm{softmax}(QK^\top)\,V \quad \neq \quad \mathrm{softmax}(Q)\big(\,\mathrm{softmax}(K)^\top V\,\big) \tag{1.45}$$

左边是 $O(n^2)$，右边是 $O(nd^2)$。**关键在于能不能去掉那个 softmax**—— 它正是阻止结合律成立的原因（softmax 是逐行的非线性操作）。

线性注意力用别的核函数替代 softmax（比如 $\phi(x) = \mathrm{elu}(x) + 1$），让结合律成立，从而把复杂度降到**关于 n 线性**。代价是表达能力下降——它算的不再是“每个位置对每个位置”的精确权重。

这一节的要点

所有这些变体都在做同一个权衡：

$$\underbrace{O(n^2)}_{\text{精确，但贵}} \quad \leftrightarrow \quad \underbrace{O(n) \text{ 或 } O(n\log n)}_{\text{便宜，但要加假设}} \tag{1.46}$$

没有免费的午餐：要么花 n^2 的代价算精确的注意力（FlashAttention 让这个代价可接受），要么加归纳偏置（“重要的在附近”或“可以用核函数近似”）来换速度。

今天的主流选择是**前者**——因为 FlashAttention 把 $O(n^2)$ 的实际开销压得足够低，而它的结果是**精确的**，不用赌“重要的一定在附近”这个假设是否成立。

1.8.16 一个常被忽略的细节：为什么是“加”不是“拼”

最后一处值得拆开的设计。多头注意力的输出是**拼接**再投影的：

$$\mathrm{Concat}(\mathrm{head}_1, ..., \mathrm{head}_h)\, W^O \tag{1.47}$$

为什么不是直接相加，或者取平均？

因为拼接保留了“哪个头贡献了什么”的信息，而相加会把它抹掉。

具体说：拼接后接一个线性层 W^O，等价于给每个头分配一个**独立的输出子空间**。W^O 的每一列可以单独决定“头 i 的这一维要不要保留、放大多少”。如果直接相加，所有头被强制共享同一个输出空间，模型就没有这个自由度了。

用公式看更清楚。设每个头输出 $o_i \in \mathbb{R}^{d_v}$，拼接后是 $[o_1; ...; o_h] \in \mathbb{R}^{hd_v}$。左乘 $W^O \in \mathbb{R}^{d_{\text{model}} \times hd_v}$：

$$W^O[o_1; ...; o_h] = \sum_{i=1}^{h} W_i^O\, o_i \tag{1.48}$$

其中 W_i^O 是 W^O 的第 i 块。**所以“拼接 + 投影”在数学上等于“每个头独立投影后求和”**——但前者让每个头有自己的 W_i^O，后者（直接相加）强制所有头共用同一个变换。

> **i** 停下来想一想 ⑧
>
> 既然 式 (1.48) 说明“拼接 + 投影”等价于“各自投影后相加”，那么**参数量**上两者一样吗？如果一样，为什么实际实现都写成“拼接 + 一个大的 W^O”，而不是“h 个小 W_i^O 再相加”？
>
> 提示：从**计算效率**（一次矩阵乘法 vs h 次）和**代码简洁性**两个角度想。

1.9 注意力看不见顺序

到这里，式 (1.15) 已经能算了。但它有一个**致命的先天缺陷**—— 这个缺陷不是实现问题，是数学结构决定的，必须靠额外设计来补。

1.9.1 先做一个小实验

把输入的两个位置对调，看输出会怎样。

设 $Q = K = V = X$（为了看清结构，让三者相同），输入 X 是两行：

$$X = \begin{pmatrix} a \\ b \end{pmatrix} \tag{1.49}$$

用 $X_1 = \begin{pmatrix} a \\ b \end{pmatrix}$ 算一遍注意力，再用 $X_2 = \begin{pmatrix} b \\ a \end{pmatrix}$（交换两行）算一遍。

因为 $Q = K = X$，打分矩阵是 $XX^\top$。对于 X_1：

$$X_1 X_1^\top = \begin{pmatrix} a \cdot a & a \cdot b \\ b \cdot a & b \cdot b \end{pmatrix} \tag{1.50}$$

对于 X_2：

$$X_2 X_2^\top = \begin{pmatrix} b \cdot b & b \cdot a \\ a \cdot b & a \cdot a \end{pmatrix} \tag{1.51}$$

第二式正好是第一式交换行列的结果。也就是说：交换输入的顺序，输出只是跟着交换，每个位置的**数值一模一样**。

人话翻译：注意力把输入当成一个**集合**，而不是一个**序列**。它对“谁在谁前面”这件事**完全不敏感**。

这不是 bug，是注意力机制的定义使然—— 它本质上是“每个位置向所有位置做加权平均”，而加权平均这个操作本身不带顺序概念（加法是可交换的）。

但语言是有顺序的。“明月松间照”和“照间松月明”用的是同一批字，意思完全不同。所以必须想办法把顺序**塞进去**。

> **i** 停下来想一想 ⑩
>
> 上面说“注意力对顺序不敏感”。但这个结论有个前提：Q、K、V 都是从 X 直接来的。
> 那么请问：**如果一个模型在注意力的输出后面接了一个前馈网络（FFN），它能否“学会”区分顺序？** 换句话说，置换等变性会被后面的层破坏吗？
> 提示：想想 FFN 是对**每个位置独立**作用的还是**跨位置**作用的。

1.9.2 正弦位置编码

最经典的做法来自原论文：**给每个位置算一个固定向量，直接加到词嵌入上**。

$$\begin{aligned} PE_{(pos,\,2i)} &= \sin\left(\frac{pos}{10000^{2i/d}}\right) \\ PE_{(pos,\,2i+1)} &= \cos\left(\frac{pos}{10000^{2i/d}}\right) \end{aligned} \tag{1.52}$$

其中 pos 是位置，i 是维度下标。偶数维用正弦、奇数维用余弦。
为什么是这种形式？ 论文给的解释是“让模型容易学到相对位置”。具体说，对任意固定偏移 k，PE_{pos+k} 可以写成 PE_{pos} 的**线性变换**：

$$\begin{pmatrix} \sin(\omega(pos+k)) \\ \cos(\omega(pos+k)) \end{pmatrix} = \underbrace{\begin{pmatrix} \cos(\omega k) & \sin(\omega k) \\ -\sin(\omega k) & \cos(\omega k) \end{pmatrix}}_{R(\omega k)} \begin{pmatrix} \sin(\omega pos) \\ \cos(\omega pos) \end{pmatrix} \tag{1.53}$$

这是三角函数的和角公式。**$R(\omega k)$ 是一个只依赖偏移量 k、不依赖位置 pos 的旋转矩阵**。
所以“位置 pos 和位置 $pos+k$ 的关系”可以被表示成一个**统一的线性操作**—— 这正是注意力需要的（因为注意力是用点积衡量关系的，如果相对位置能用固定变换表示，模型就能学到“往前看 k 个”这样的模式）。

> 为什么用多个频率
>
> $10000^{2i/d}$ 这个分母让不同维度对应**不同波长**：$i=0$ 时波长约 2π（高频，区分相邻位置），$i=d/2$ 时波长约 $2\pi\times10000$（低频，区分远距离位置）。
> 这像二**进制计数器**：低位快速翻转（区分近处），高位缓慢翻转（区分远处）。所有维度合起来，每个位置得到一个“独一无二”的组合。
> 用多个频率的原因是**单一频率无法同时区分近处和远处**：高频在远距离上会周期性重复（位置 0 和位置 100 编码相同），低频在近距离上几乎不变（区分不出位置 0 和位置 1）。

1.9.3 手算一遍：位置编码长什么样

公式看着抽象，算出来其实很朴素。取 $d=4$（两个维度对），算前四个位置：
$i=0$ 时频率是 $1/10000^0=1$；$i=1$ 时是 $1/10000^{0.5}=1/100$。

pos	$\sin(pos/1)$	$\cos(pos/1)$	$\sin(pos/100)$	$\cos(pos/100)$
0	0.0000	1.0000	0.0000	1.0000
1	0.8415	0.5403	0.0100	1.0000
2	0.9093	−0.4161	0.0200	0.9998
3	0.1411	−0.9900	0.0300	0.9996

读懂这张表的关键是看两列的“变化速度”：

- **前两列（高频）**变化极快：pos 从 0 到 3，sin 走了 $0\to0.84\to0.91\to0.14$，已经在往回走了。它们负责区分**相邻位置**。

· **后两列（低频）**几乎不动：$\cos(pos/100)$ 从 1.0000 只变到 0.9996。 它们负责区分**远距离位置**——要等到 pos 几百，才会看出明显变化。

把这四个数当作一个四维向量，每个位置的编码就是空间中的一个点。 有一个漂亮的性质：

$$\| \mathrm{PE}(pos+1) - \mathrm{PE}(pos) \| = 0.9589 \quad \text{对所有} \quad pos \tag{1.54}$$

相邻位置的编码距离是完全相同的——不管是 0→1 还是 100→101。这让位置编码在整条数轴上是“均匀”的，不存在“近处分辨率高、远处分辨率低”的问题。

为什么这件事重要?

如果编码在近处密集、远处稀疏，那么模型对远处位置的区分能力就会下降—— 而语言里“隔了 50 个词的主语”和“隔了 51 个词”同样需要区分。

正弦编码用**多个频率叠加**来解决这个问题：低频维度负责“粗定位”（这是第几个 100），高频维度负责“细定位”（这是 100 里的第几个）。就像时钟用时针和分针配合，同时给出大的和小的刻度。

这也解释了为什么波长要跨越这么大的范围（$d = 4$ 时是 6.3 和 628.3）：如果所有维度波长都一样，那么位置 0 和位置（波长）就会得到完全相同的编码， 模型无法区分它们。

> **i** 停下来想一想 ⑭
>
> 正弦编码是**固定公式**，不参与训练。这在当时是个有争议的选择—— 有人主张“让模型自己学位置表示不是更好吗”。
>
> 请想：**如果位置编码是可学的，会有什么风险?** 提示从两个角度： ① 训练时只见过 $pos \in [0, 511]$，推理时遇到 $pos = 1000$ 会怎样? ② 位置表示需要多少训练数据才能学好?
>
> （本章的小模型用的就是可学习的位置嵌入，可以对照着看这个取舍。）

1.9.4 可学习的位置嵌入

本章的小模型用的是更简单的做法：**给每个位置一个可学习的向量**， 和词嵌入一样训练出来。

```
self.pos_emb = nn.Embedding(cfg.block_size, cfg.d_model)
# 前向时:
x = self.tok_emb(idx) + self.pos_emb(pos)[None, :, :]
```

这**更简单、也更灵活**（模型自己决定每个位置该长什么样）， 但有一个硬限制：**最多只能处理训练时见过的长度**。 训练时 `block_size = 64`，推理时给第 65 个位置， `pos_emb` 就没有对应的行了。

正弦编码没有这个问题（它是个公式，可以算任意 pos）， 但它的表达能力更受限（位置表示是固定的，不随任务调整）。

> **i** 这个取舍很重要
>
> **可学习 = 表达力强 + 不能外推；固定公式 = 能外推 + 表达力受限。**
>
> 这个矛盾催生了今天主流的方案——RoPE。它同时要了两者。

1.9.5 RoPE：把位置变成旋转

RoPE（Rotary Position Embedding） 是 2021 年提出、今天几乎所有开源大模型 （Llama、Qwen、Mistral 等）都采用的方案。它的想法很漂亮：

不再“加”位置信息，而是“旋转” q 和 k。

具体做法：把 d 维向量两两分组，每一对 (x_{2i}, x_{2i+1}) 看作复平面上的一个点， 按位置 pos 和频率 θ_i 旋转一个角度：

$$R^d_{\Theta,pos} = \mathrm{diag}\Big(R(\theta_1, pos), R(\theta_2, pos), \ldots, R(\theta_{d/2}, pos) \Big), \qquad \theta_i = 10000^{-2(i-1)/d} \tag{1.55}$$

其中 $R(\theta, pos)$ 是 2×2 的旋转矩阵：

$$R(\theta, pos) = \begin{pmatrix} \cos(pos\,\theta) & -\sin(pos\,\theta) \\ \sin(pos\,\theta) & \cos(pos\,\theta) \end{pmatrix} \tag{1.56}$$

然后**在算注意力之前**，把 q、k 各自旋转：

$$\tilde{q}_m = R_{\Theta,m}\, q_m, \qquad \tilde{k}_n = R_{\Theta,n}\, k_n \tag{1.57}$$

（m、n 是位置下标。）

为什么这么做？关键在旋转的乘法性质。计算 $\tilde{q}_m$ 与 $\tilde{k}_n$ 的内积：

$$\tilde{q}_m^\top \tilde{k}_n = (R_m q_m)^\top (R_n k_n) = q_m^\top R_m^\top R_n\, k_n \tag{1.58}$$

旋转矩阵是正交的（$R^\top R = I$），所以 $R_m^\top R_n = R_m^{-1} R_n = R_{n-m}$。最后一步用到了旋转的复合性质：**两个旋转的乘积是角度相加，取逆是角度取负**。

于是：

$$\tilde{q}_m^\top \tilde{k}_n = q_m^\top R_{n-m}\, k_n \tag{1.59}$$

人话翻译：这是整个 RoPE 的核心一行。

左边是“**加了位置信息之后**的两个向量的内积”；右边是“**原始向量**的内积，但中间夹了一个**只依赖于相对距离 $(n-m)$** 的旋转”。

也就是说：**注意力打分自动变成了相对位置的函数**。模型不需要“知道”绝对位置是多少，它看到的是“这两个位置隔了多远”。

这正是我们想要的——语言里重要的是相对关系（“前一个词”“隔三个词的主语”），而不是绝对下标。

1.9.6 RoPE 相比前两代好在哪

	正弦编码	可学习嵌入	**RoPE**
位置信息注入方式	加到输入上	加到输入上	**旋转 q、k**
编码的是	绝对位置	绝对位置	**相对位置**
参数	0	$L \times d$	**0**
能否外推	有限	**不能**	**可以**（配合插值技术）
与注意力的交互	无	无	**直接作用在打分上**

RoPE 的关键优势在最后一行：**它不是在输入上“打个标记”，而是直接改变了注意力打分的结构**。正弦编码加在输入上之后，经过 W_Q、W_K 的线性变换，那个“相对位置的线性关系”会被打乱；RoPE 作用在 q、k 本身上，所以这个关系**必然保留**。

> **i** 停下来想一想 ⑪
>
> RoPE 让打分只依赖相对距离 $n-m$。这本该是好事。
>
> 但请想一个问题：**如果打分只依赖相对距离，那“位置 1 和位置 2”与“位置 1001 和位置 1002”对模型来说有区别吗？**
>
> 如果没有区别，会发生什么？（提示：想想模型的上下文窗口从 4K 扩到 128K 时，训练时最长只见过 4K，那 4K 以外的位置会怎样？）
>
> 这就是“长度外推”问题的根源，也是 NTK-aware 插值、YaRN 这一系工作的出发点。

1.9.7 本章模型的位置编码

回头看那张头部画像表——**层 0 头 1 有 45% 的注意力给紧邻左边**，层 0 头 0 有 77% 给自己。这些“局部”模式正是位置编码在起作用的证据：模型需要知道“左边那个是谁”，才能学出这种模式。

如果没有位置编码，这些头就**学不会**——因为在它眼里，所有位置的输入是无差别的集合。

第 2 章会详细讲 RoPE 的实现细节，以及今天主流的长度外推技术。

1.10 注意力图该怎么读（以及不该怎么读）

实验 1-2 里那些热力图，是理解模型最直观的工具。但**它也是最容易被过度解读的工具**。这一节讲怎么读它、以及哪里会读错。

1.10.1 先讲读法

一张注意力图是一个 $n \times n$ 的矩阵，第 i 行第 j 列是 “位置 i 从位置 j 取了多少信息”。常见的四个观察角度：

① **看对角线**。对角线亮 = 每个位置主要看自己。这意味着这一层几乎没有做跨位置的信息交换 —— 它更像一个“逐位置的处理”，而不是“整合上下文”。

在我们的模型里，层 0 头 0 的对角权重是 **0.769**，是个典型的“看自己”的头。

② **看对角线下方第一条（次对角线）**。亮 = “看前一位”。这类头（previous-token head）在文献里非常常见，我们的层 0 头 1 就是（0.450）。

③ **看某一列的亮度**。如果整列都很亮，说明**所有位置都在看同一个位置**。这类叫“attention sink”（注意力汇）—— 通常落在第一个 token（BOS）或标点上。

为什么会有 attention sink：因为 softmax 的输出必须和为 1。如果某个位置“不需要看任何地方”，它仍然必须把 1 分配出去。分配到哪里最不碍事？一个语义上中性的位置——通常是序列开头的特殊 token。**这是 softmax 归一化强加的结构，不是模型“想”关注那个 token**。

④ **看整体形状**。是弥散的一大片，还是几条清晰的斜线？清晰的斜线（对角线、次对角线、固定偏移）说明学到了**位置相关**的模式；弥散说明在整合全局信息。

1.10.2 再讲三个不该做的解读

错误一：把“注意力权重高”当成“因果关系”。

“模型把 0.8 的注意力给了『月』，所以它是靠『月』做决定的” —— **这个推理不成立**。

权重高只说明“信息从这里取的多”，不说明“取了之后有没有被用上”。一个头可能取了很多信息，但它的输出投影 W_O 把这些信息**投影掉了**（投影到对后续层无用的方向）。

正确的做法是干预：把那个位置的注意力权重强制设为 0，看输出/预测有没有变。**变了才是因果，没变就只是相关**。

这个思路叫**消融（ablation）**，第 4 章会详细讲。

错误二：把单个头当成一个完整的功能单元。

我们在 §1.8 看到，模型里有个明确的“前一位头”。很容易得出“这个头负责处理前一个词”的结论。

但**功能是由多个头协作完成的**。一个头可能只在特定上下文里起作用，或者它提供的信息要和其他头组合才有意义。

更麻烦的是：**大部分头可以被剪掉而不影响性能**（Michel et al., 2019）。如果剪掉它性能不掉，那它到底“负责”什么？

更稳妥的说法是描述而非断言：说“这个头有 45% 的注意力给了前一位”（可测量的描述），而不是“这个头负责语法”（无法直接验证的断言）。

错误三：忽略“权重和必须为 1”这个约束。

看到“某个位置只有 0.3 的最大权重”，容易解读为“模型在这里不太确定”。

但**每一行的权重和必须是 1**。如果这一行有 100 个可见位置，0.3 已经是**远远高于平均**（0.01）的集中了。反过来，如果只有 2 个可见位置，0.6 其实相当分散。

所以看权重必须同时看“这一行有多少个可见位置”。这也是为什么我们统计时用了**归一化熵**而不是原始权重 —— 它把“可选项数量”这个因素消掉了。

一个具体的例子：在我们的头部画像表里，层 0 头 0 的熵是 0.294，层 1 头 3 是 0.871。但因为序列长度不同（位置 5 只能看 6 个，位置 23 能看 24 个），如果不做归一化，这两个数根本没法比。

1.10.3 一个实用的读图流程

拿到一张注意力图，按这个顺序看：

```
① 先看坐标轴 —— 是 n×n 还是 m×n（交叉注意力）？序列多长？
② 看掩码形状 —— 上三角是不是空的？如果不是因果注意力，为什么？
```

```
③ 算三个数：对角权重、平均距离、归一化熵
④ 看整体形状 —— 弥散 / 斜线 / 竖列
⑤ 只在做了消融之后，才谈"它在做什么"
```

第③步是可以自动化的——`notebooks/analyze_heads.py` 做的就是这件事。先让脚本给你数字，再去看图，比“盯着热力图找规律”可靠得多。

> 为什么先看数字再看图
>
> 人的视觉系统很擅长在随机噪声里“看出”模式。一张 24×24 的热力图有 576 个数值，你几乎一定能在里面找到几条“看起来有意思”的斜线——**即使它们是随机的**。
>
> 而数字（对角权重均值、归一化熵）不会骗你。先用数字筛掉“没有明显模式”的头，再对剩下的少数几个仔细看图，效率高得多，也少犯错。

1.11 它在哪里失效

前面讲过的一般原理，这里落成具体的失败模式。每一条都配可复现的代码。

1.11.1 失败一：mask 用 0 而不是 $-\infty$

这是实现里最常见、也最隐蔽的错误。因果掩码的目的是“不让位置 i 看到位置 $j > i$”。很多人的写法是：

```
scores = scores.masked_fill(mask == 0, 0.0)   # ← 错
```

为什么错：softmax 里，“加 0”意味着“这个位置的分数是 0”，而 0 经过 $e^0 = 1$ 之后**仍然会分到权重**。被 mask 的位置没有被排除，只是被打了个折。

正确写法是加 $-\infty$：

```
scores = scores.masked_fill(mask == 0, float("-inf"))   # ← 对
```

$e^{-\infty} = 0$，这才是彻底排除。

```
import numpy as np

def softmax(z):
    z = z - z.max(); e = np.exp(z); return e / e.sum()

scores = np.array([1.0, 2.0, 0.5])
print("用 0 mask:", np.round(softmax(scores), 6))
# [0.244728 0.665241 0.090031]  ← 第三个位置仍然拿到 9% 的权重

m = scores.copy(); m[2] = -np.inf
print("用 -inf  :", np.round(softmax(m), 6))
# [0.268941 0.731059 0.      ]  ← 第三个位置被彻底排除
```

两行输出的差别就是全部：`0.090031` 对 `0.000000`。后者的第三位被**彻底**排除，前者的第三位只被打了个折。

这个错误为什么难发现：训练 loss 照降。因为模型会用别的方式绕过——它学会把被 mask 位置的 v 学成接近零向量，从而“即使权重不为零，贡献也为零”。但这是**浪费容量**，而且推理时会暴露（尤其是 KV 缓存实现里）。

1.11.2 失败二：缩放写在 softmax 之后

```
w = softmax(scores)
w = w / np.sqrt(d_k)      # ← 错：这不是缩放，是改温度 + 破坏归一化
```

为什么错：除以 $\sqrt{d_k}$ 必须作用在 **logits** 上，因为它修正的是“softmax 的输入尺度”。写在 softmax 之后，w 的和不再是 1，而且它改变的是分布的**整体大小**，不是**尖锐程度**。

```
import numpy as np

def softmax(z):
    z = z - z.max(); e = np.exp(z); return e / e.sum()

s = np.array([8.0, 0.0])              # d_k = 64 时的典型打分
a = softmax(s);      print("缩放前  :", a,         " 和 =", a.sum())
# [9.996646e-01 3.353501e-04]  和 = 1.0    ← 很尖，但仍是概率分布
b = softmax(s / 8);  print("正确缩放:", b,         " 和 =", b.sum())
# [0.731059 0.268941]          和 = 1.0    ← 合理
c = softmax(s) / 8;  print("错误缩放:", c,         " 和 =", c.sum())
# [1.249581e-01 4.191877e-05]  和 = 0.125  ← 和不再是 1!
```

第三种输出的和是 **0.125，根本不是概率分布**。下游的加权求和会得到一个 量级完全错误的输出，而 shape 检查、loss 曲线都不会报错。

1.11.3 失败三：忘了数值稳定

小节 1.6 已经给过溢出的例子，这里补充它在**注意力里**的具体表现。关键在于：**logits 到底能有多大，取决于缩放因子在不在**。

```
import numpy as np

def check(std, label):
    """按给定标准差造一批 logits，看 exp 会不会溢出。"""
    rng = np.random.default_rng(0)
    z = rng.normal(0, std, size=(4, 8)).astype(np.float32)   # float32!
    e = np.exp(z)
    print(f"  logits 标准差 {std:6.1f} ({label}) "
          f"  →  inf 个数 {int(np.isinf(e).sum()):2d} / {e.size}")

check(1.6,  "缩放后（本章案例实测）")
check(5477, "不缩放（本章案例实测）")
```

实测输出：

```
  logits 标准差    1.6 (缩放后（本章案例实测）)   →  inf 个数  0 / 32
  logits 标准差 5477.0 (不缩放（本章案例实测）)   →  inf 个数 14 / 32
```

不缩放那一行，32 个元素里有 14 个直接溢出。而 14 这个数字还不是上限—— 它只取决于随机抽样的最大值有没有超过 88.7。

> ⚠ 为什么这里必须用 float32
>
> 上面的例子如果写成 float64，**一个都不会溢出**：
>
> ```
> float32  logits=[100,0]  exp 最大值 = inf        ← 溢出
> float64  logits=[100,0]  exp 最大值 = 2.688e+43  ← 没溢出
> ```
>
> float64 的指数上限对应 $e^{709.8}$，比 float32 的 $e^{88.7}$ 宽得多。 而**深度学习框架默认用 float32**，所以你平时遇到的溢出， 在 numpy 默认的 float64 里复现不出来。
>
> 写调试代码时踩这个坑的人很多：明明线上炸了，本地却跑不出问题。

一个 batch 里只要有一行 nan，整个梯度就全废了——因为 `inf / inf = nan`， 而 nan 会沿着反向传播污染所有参数。

1.11.4 失败四：注意力退化成平均

这个失败不报错、不崩，只是效果差——所以最难发现。

触发条件：缩放**过度**（比如误用了 d_k 而不是 $\sqrt{d_k}$）， 或者温度设得过高。表现是所有权重几乎相等：

```python
import numpy as np

def softmax(z):
    z = z - z.max(); e = np.exp(z); return e / e.sum()

s = np.array([8.0, 3.0, -2.0, 0.5])
for c, name in [(1, "不缩放"), (8.0, "÷√64（正确）"), (64.0, "÷64（过度）")]:
    p = softmax(s / c)
    print(f"  {name:14s} {np.round(p, 4)}  最大权重 {p.max():.4f}")
```

实测输出：

```
  不缩放             [9.927e-01 6.700e-03 0.000e+00 5.000e-04]  最大权重 0.9927
  ÷√64（正确）        [0.4518 0.2418 0.1294 0.1769]  最大权重 0.4518
  ÷64（过度）         [0.2725 0.252  0.2331 0.2424]  最大权重 0.2725
```

第三行四个权重是 **0.2725 / 0.2520 / 0.2331 / 0.2424**——几乎就是均匀的 0.25。**注意力退化成了简单平均**，模型不再具有“选择信息”的能力。

注意第二行和第三行的对比：正确缩放下最大权重 0.4518，既能明显区分主次，又不是 one-hot。**这就是“落在 softmax 敏感区间”的实际样子**。

这个失败不会报错，loss 也照降——只是模型永远学不好。它是本章五个失败模式里**最难发现**的一个。

1.11.5 失败五：序列长度带来的显存爆炸

$n \times n$ 的注意力矩阵，单头、float16、$n = 8192$ 时：

$$8192^2 \times 2 \text{ 字节} \approx 134 \text{ MB} \tag{1.60}$$

看似不大，但这是**单个头、单层**。32 层 × 32 头就是

$$134 \text{ MB} \times 32 \times 32 \approx 137 \text{ GB} \tag{1.61}$$

当然实际实现不会把所有层同时保留（反向传播时需要，前向时不用），但**长序列的瓶颈是显存而非算力**这一点是确定的。这正是 FlashAttention 的出发点：分块计算，永远不把完整的 $n \times n$ 矩阵写进显存。

1.11.6 失败六：位置编码越界

这个失败模式在前面的案例里没有出现（因为我们训练和推理用同样的长度），但在真实使用中极其常见。

触发条件：用**可学习**的位置嵌入，且推理长度超过训练时的 `block_size`。

```python
# 训练时
cfg = Config(block_size=64)
self.pos_emb = nn.Embedding(64, d_model)      # 只有 64 行

# 推理时给了 100 个 token
pos = torch.arange(100)
x = self.tok_emb(idx) + self.pos_emb(pos)     # ← IndexError，或者更糟
```

两种后果，看实现：

① **直接报错**（PyTorch 的 `nn.Embedding` 会抛 `IndexError`）。这是**好的**情况——至少你知道出了问题。

② **静默给出垃圾**。如果实现里做了取模（`pos % block_size`）或者其他截断，程序不报错，但位置 64 会被当成位置 0—— **模型以为自己在处理句首**，输出的分布完全不对。

一个真实的排查场景：模型在 512 长度上训练得很好，上线后处理 800 字的长文本，前 512 个 token 输出正常，之后突然变成乱码。

原因：位置 512 越界了。`pos_emb` 只有 512 行。

为什么会“突然”：前 512 个位置在训练分布内，行为正常；第 513 个位置落到没训练过的区域，模型完全不知道该怎么办。**这是一条硬边界，不是渐变**。

排查手段：在推理入口加一句断言。

```
assert idx.shape[1] <= model.cfg.block_size, \
    f"序列长度 {idx.shape[1]} 超过训练时的 {model.cfg.block_size}"
```

这个断言很便宜，但能省下几小时的困惑。
为什么正弦编码和 RoPE 没有这个问题：它们是**公式**（$\sin(pos/10000^{2i/d})$ 和旋转角 $pos \cdot \theta$），可以算任意 pos，不存在“表用完了”这种说法。
但“能算”不等于“算得对”——正如 §1.13 检查点 ⑪ 说的，外推出去的位置落在训练时没见过的取值范围内，模型的行为仍然不可靠。**这是“外推”问题，比“越界”问题更微妙。**

方案	越界会怎样	外推表现
可学习嵌入	报错或静默出错	完全不支持
正弦编码	能算	有限外推（高频维度会周期重复）
RoPE	能算	需配合插值技术（NTK/YaRN）

1.11.7 把这四个 bug 真的跑一遍

上面讲的都是“理论上会怎样”。但调试时你需要的是**症状**——看到什么现象，才知道是哪个 bug。
所以我们把每个 bug **真的注入到模型里**，各训 300 步，记录实测症状。代码在 `notebooks/debug_failures.py`，数据在 `data/failure_symptoms.json`。

变体	loss 起	loss 终	末梯度范数	权重熵	对角权重	行和最大值
正确实现	7.032	4.810	1.018	0.831	0.153	**1.0000**
① 掩码用 0	7.034	**4.634**	3.791	0.949	0.134	1.0000
② 缩放位置错	7.055	4.668	0.560	0.836	0.106	**0.1768**
③ 缺数值稳定	7.032	4.808	1.170	0.842	0.158	1.0000
④ 误用 d_k	7.032	4.708	0.776	**0.9989**	0.127	1.0000

现在逐个读这张表——**它比任何“注意事项”都有用。**

1.11.7.1 ① 掩码用 0：loss 反而更低

看第二行：**bug 版本的 loss（4.634）比正确实现（4.810）还低。**
这不是测量错误。原因是：用 0 掩码意味着被屏蔽的位置**仍然分到权重**，所以位置 i 能“看见”位置 $j > i$ 的内容——**模型在偷看答案。**
预测下一个字时，答案就在输入里。所以训练 loss 当然更低。
这是最难发现的一类 bug：它让指标变好，而不是变差。
如果只看 loss 曲线，你会以为这个改动是个改进，甚至会把它保留下来。**唯一的破绽在推理时**：生成第 t 个词时，第 $t+1$ 个词根本还不存在，偷看的能力瞬间消失，模型原形毕露。
排查手段：熵那一列也能看出来——0.949 明显高于正确实现的 0.831，因为被“偷看”的位置摊薄了权重分布。但最可靠的检查还是**单测**：给一段已知输入，验证位置 i 对位置 $j > i$ 的权重**严格为 0**。

1.11.7.2 ② 缩放位置错：行和不是 1

第三行最右边那个数字 **0.1768** 就是铁证。
正确的注意力权重，每一行的和必须是 1（它是概率分布）。而把除法写在 softmax 之后，$1/\sqrt{32} = 0.1768$ 直接把每一行都缩小了——**和数学上的“概率分布”完全不沾边了。**

这个 bug 的特点是好查：加一行 `assert abs(weights.sum(-1) - 1) < 1e-5` 就能拦住。但它的危害是不报错的——输出的 shape 完全正常，loss 也照降（4.668）。

1.11.7.3 ③ 缺数值稳定：在这个规模下根本看不出来

第四行：loss 4.808（和正确实现的 4.810 几乎一样），没有任何异常。

为什么？ 因为 $d_{\text{head}} = 32$ 时 logits 的标准差只有约 0.07–1.6（正是本章案例里测到的量级），exp 完全不会溢出。

这和本章开头那个“没复现出来的实验”是同一回事—— 一个只在特定尺度下才显形的 bug。你在小模型上调试时看到一切正常，换成大模型或者长序列，它就炸了。

> ⚠ 这条最值得记住
>
> 四个 bug 里，③ **是唯一一个“在你当前的调试环境里测不出来”的**。
>
> 应对办法不是“小心一点”，而是**主动制造条件**：写一个单元测试，故意喂进标准差 5000 的 logits，看函数会不会返回 nan。本章案例里那个不缩放的模型实测标准差是 **5477**，32 个 logits 里 14 个溢出—— 这就是你该用的测试数据。
>
> **把“边界条件测试”提前写好，而不是等它在生产环境里出现**。

1.11.7.4 ④ 误用 d_k：注意力退化成均匀

最后一行的熵是 **0.9989**——极其接近 1.000，也就是**完全均匀的分布**。

乘以 $\sqrt{d_k}$ 倍的过度缩放，把 logits 压得几乎全相等，softmax 输出就变成 $1/n$。模型失去了“选择信息”的能力，但**它不报错、loss 也照降**（4.708，甚至比正确实现还低一点，因为 300 步的训练还没到见分晓的时候）。

排查手段：看熵。如果某一层的权重熵长期停在 0.99 以上，几乎可以确定是缩放出了问题。这也是为什么 `analyze_heads.py` 要把熵作为第一个指标。

1.11.7.5 一张图看清五个失败模式

失败模式	loss 会变差吗	有异常数值吗	怎么查
① mask 用 0	**反而变好**	熵偏高（0.949）	单测：未来位置的权重必须为 0
② 缩放位置错	轻微变好	**行和 = 0.177**	断言：每行和 ≈ 1
③ 缺数值稳定	不变	**无**（小模型下）	喂大 logits 的边界测试
④ 缩放过度	不变	熵 = 0.9989	监控权重熵
⑤ $O(n^2)$ 显存	—	OOM 报错	唯一会自己报错的

五个里有四个不会报错，其中两个在训练指标上完全看不出来。这就是为什么调试注意力必须靠**主动测量**（熵、行和、对角权重这些量），而不是盯着 loss 曲线。

`analyze_heads.py` 输出的那四个统计量，本身就是一套通用的“注意力体检表”—— 建议在任何自定义注意力实现上线前跑一遍。

1.12 注意力的训练动态

前面我们对比了“训练前”和“训练后”。但中间发生了什么？这一节用案例里记录下来的完整过程数据，看注意力在 1500 步里怎么演化。

1.12.1 两条完全不同的轨迹

案例里我们每一步都记录了注意力层的诊断量。把两个变体并排看：

不缩放那一边：

步数	权重熵	logits 标准差	$\|J\|_F$
0	0.712	1.63	0.2975
222	**0.0022**	982.9	0.00471
444	0.0003	3176.5	0.00051
888	0.0001	5416.2	0.00020
1480	0.0001	5458.8	0.00008

缩放那一边：

步数	权重熵	logits 标准差	$\|J\|_F$
0	0.9985	0.10	0.2017
222	0.6437	2.21	0.3029
444	0.6754	1.62	0.2938
888	0.5714	1.75	0.2996
1480	0.5745	1.78	0.3089

1.12.2 三条值得注意的规律

① **崩塌发生在最初 222 步内，而且极快。**
不缩放那一边，熵从 0.712 掉到 0.0022——**只用了 222 步，是总步数的 15%**。之后 1200 多步，熵基本停在 0.0001，没有任何恢复的迹象。
这说明：缩放问题不是“慢慢积累”的，而是**一次性的**。一旦 softmax 进入饱和区，梯度就传不回去，注意力**再也没机会自我修正**。它是一个**不可逆的相变**。

> ⚠ 这对调试的含义
>
> 如果你在训练到一半才发现熵塌了，**改超参数救不回来**——只能从头训。
> 所以这类问题的正确应对是**预防**，不是补救：- 训练开始就监控熵（前 200 步最关键）- 在 step 0 就检查初始化后的 logits 尺度
> **我们案例里之所以能发现它，正是因为记录了 step 0 的数据**。如果只记最终结果，你会看到“熵 0.0001”，但不知道它是从第几步开始塌的。

② **缩放那一边的熵在 0.57–0.68 之间波动，而且整体是**下降**的。**
从 0.9985（初始化时接近均匀）降到 0.57 左右，然后稳定。
这个下降是**健康的**：初始化时模型对所有位置一视同仁（熵接近 1），训练过程中逐渐学会“该看谁”（熵下降）。但注意它**没有继续降到 0** ——它稳在一个中间水平。
0.57 这个值意味着什么：在一行 24 个位置里，注意力大致集中在几个位置上，但仍保留了相当宽的分布。**这正是我们想要的**——既要有区分度，又不能退化成硬选择。
③ **$\|J\|_F$ 的变化最能说明问题。**
缩放那一边：从 0.2017 升到 0.30 左右，**然后一直稳定**。不缩放那一边：从 0.2975 掉到 0.00008，**掉了一万多倍**。
为什么 $\|J\|_F$ 是关键指标：它直接衡量“梯度能不能传回去”。熵和 logit 标准差是“现象”，$\|J\|_F$ 是“后果”。
有意思的是**缩放那边的 $\|J\|_F$ 是上升的**（0.20 → 0.30）。一个可能的解释是：初始化时权重很小、logits 很平，雅可比自然偏小；训练让它变得更“有区分度”，雅可比随之上升到一个合理的水平。**这又是一个“回到 1 附近”的例子**——和方差的目标是同一个思路。

1.12.3 初始化为什么这么重要

从上面的数据能反推出一个结论：**初始化决定了训练会不会走上正轨**。

在 step 0，两个变体的**唯一差别**是有没有缩放。但它们的初始诊断量已经不同了：

	logits 标准差	熵
不缩放	1.63	0.712
缩放	0.10	0.9985

不缩放那一边初始的熵是 0.712——**已经比缩放那边尖得多**。这个初始差异，在 222 步内被放大成了“一个崩、一个正常”。

“初始化不是随机起点，它决定了你会走到哪个盆地。”

这是一个在深度学习里反复出现的模式：初始化的微小差异，经过训练的放大，会导向完全不同的解。

本章的具体例子是：初始 logits 尺度偏大 → softmax 偏尖 → 梯度偏小 → 注意力更难调整 → 更尖…… **正反馈把它推向饱和**。 而初始尺度正常时，这条正反馈链不成立。

所以“调参”这件事，很多时候真正的杠杆在初始化，而不是学习率。

1.12.4 一个可以自己做的观察

用 `analyze_heads.py` 在训练的不同阶段各跑一次， 就能看到头部的“分化过程”：

```
# 训练 100 步、500 步、1500 步，各存一次权重
for steps in (100, 500, 1500):
    cfg = Config(max_steps=steps)
    model, tok, _ = train(text, cfg)
    torch.save(model.state_dict(), f"ckpt_{steps}.pt")

# 然后分别分析
for steps in (100, 500, 1500):
    stats = analyze(f"ckpt_{steps}.pt")
    # 看"前一位头"的 prev 权重是怎么涨起来的
```

我们没做这个实验（它不在本章的案例范围内）， 但它是一个很好的入手点：**头部特化是一次性出现的，还是逐渐形成的？** 如果你做了，欢迎把结果和本章的数据对照。

1.13 排查手册：从症状倒推原因

上一节给了“每个 bug 长什么样”。这一节给**怎么从症状找到它**。

调试注意力最麻烦的地方是：五个失败模式里有四个不报错，而它们的最终表现（loss 不降、生成质量差）**看起来一模一样**。 所以关键是找到那些“能区分它们”的中间量。

下面四条排查路径，都是可以照着做的。

1.13.1 排查一：loss 不降，但梯度范数正常

症状：训练跑了几百步，loss 卡在一个明显偏高的位置不再下降。 **但**梯度范数正常（0.1–10 之间波动），不是梯度消失。

第一个假设：数据或学习率有问题。 这个先排除，因为太常见了。 换一个已知能训好的小数据集（比如就几百个字符的重复文本）， 如果 loss 还是不动，说明不是数据问题。

第二个假设：模型容量不够。 对照检查——把同样的配置 （层数、维度、步数）在一个**已知能收敛的任务**上跑一遍。 我们的经验是：42 万参数的模型在 25 万字唐诗上， 1500 步应当能到 loss ≈ 4.0 左右。如果差得远，问题在实现。

第三步：量三个中间量。

```
# 训练循环里加这几行
with torch.no_grad():
    model(xb, record=True)
    d = model.blocks[0].attn.last_diag
```

```
    print(f"step {step}  熵 {d['weight_entropy']:.4f}  "
          f"logit_std {d['logit_std']:.2f}  ‖J‖ {d['jacobian_fro']:.5f}")
```

读这三个数字的方法：

观察	指向
熵 → 0.99 以上	**缩放过度**（用了 d_k 而不是 $\sqrt{d_k}$）
熵 → 0.01 以下，logit_std 几千	**缩放缺失**（或 q、k 尺度失控）
‖J‖_F → 10^{-4} 以下	注意力层已经冻结，梯度传不回去
三个量都正常，loss 还是不降	问题不在注意力——去看 FFN、输出层、或数据

我们那次“没复现出来的实验”，用的就是这个流程。区别是：小模型上三个量全都正常（熵 0.999，‖J‖ 0.26），说明注意力层没问题——**于是把怀疑转向了“这个现象是不是需要更大的规模才出现”**，这才有了后面 $d_{\text{head}} = 256$ 的对照。

> 这条路径的一般形式
>
> **“loss 不降”是个太粗的信号，必须把它分解成若干个能独立检查的量。**
> 注意力的诊断量至少有三个（熵、logit 尺度、雅可比范数），而且是**正交的**：- 熵告诉你“分布有多尖” - logit 标准差告诉你“为什么会这么尖” - 雅可比范数告诉你“这有多严重”
> 三个量合起来能唯一定位问题。只测一个不够—— 比如“熵很高（接近均匀）”可能是缩放过度，也可能是 q、k 的尺度太小，只有配合 logit 标准差才能区分。

1.13.2 排查二：训练 loss 漂亮，但生成质量差

症状：loss 曲线一路下降，最后停在很低的水平（比预期还好）。但生成出来的文本一眼就能看出问题——重复、不通顺、或者干脆是乱码。

第一个假设：过拟合。检查验证集 loss。如果验证 loss 也在降，那不是过拟合。

第二个假设：训练/推理不一致（train-test mismatch）。这是最可能的。

具体来说，语言模型训练时用“教师强制”（teacher forcing）：**把真实的**前文喂进去，预测下一个词。推理时把**自己生成的**前文喂进去。如果两者之间有差异，就会暴露。

最典型的三种不一致：

① **掩码泄露未来信息。**就是前一节的失败模式①。训练时模型能偷看答案，loss 当然低；推理时答案不存在，能力瞬间消失。

验证方法（这是个必须做的单测）：

```
# 未来位置的注意力权重必须严格为 0
_, w = my_attention(Q, K, V, mask=causal_mask(n))
import numpy as np
upper = np.triu(np.ones((n, n)), k=1).astype(bool)
leaked = w[..., upper]
assert leaked.max() < 1e-8, f"泄露了! 最大权重 {leaked.max():.3e}"
```

② **训练与推理用了不同的实现路径。**比如训练时走的是自己写的注意力，推理时框架自动换成了 `F.scaled_dot_product_attention`。两者的数值行为可能有细微差异（尤其在低精度下）。

验证方法：同一个输入，两条路径都跑一遍，对拍输出：

```
mine, _ = my_attention(Q, K, V, mask=m)
ref = F.scaled_dot_product_attention(Q, K, V, attn_mask=m)
assert (mine - ref).abs().max() < 1e-5, "两条路径不一致"
```

③ **位置编码在推理时越界。**如果用的是可学习的位置嵌入，且推理序列比训练时长，`pos_emb` 会索引到没训练过的行——那些行是随机初始化的，输出自然是垃圾。

验证方法：打印推理时的序列长度和 `block_size`，确认没超。

> ⚠ “训练 loss 反而更低”是最危险的信号
>
> 正常的 bug 会让指标变差，你会立刻发现。**让指标变好的 bug 才致命**——它会被当成一个改进保留下来。
> 我们那个掩码实验里，bug 版本的 loss（4.634）比正确实现（4.810）低了 0.18。如果不做单测，只看曲线，**这个 bug 会一直留着**，直到部署上线、发现生成质量莫名其妙地差，而那时已经很难回溯原因了。
> **结论：关键性质的验证不能依赖 loss。要写断言。**

1.13.3 排查三：loss 里出现 nan

症状：训练几十步之后，loss 变成 `nan`，之后永远是 `nan`。
第一个假设：梯度爆炸。检查梯度范数——但如果是 nan，`clip_grad_norm_` 也救不了（nan 裁剪之后还是 nan）。
第二个假设：softmax 溢出。这是注意力里最常见的 nan 来源。
定位步骤：

```
# 在前向传播里插桩，找到第一个出现 nan 的位置
def attention_debug(Q, K, V, mask=None):
    scores = Q @ K.swapaxes(-1, -2)
    print("① scores 有限?", torch.isfinite(scores).all().item(),
          "范围", scores.min().item(), scores.max().item())

    scores = scores / np.sqrt(Q.shape[-1])
    if mask is not None:
        scores = scores.masked_fill(~mask, float("-inf"))
    print("② 掩码后有限?", torch.isfinite(scores).all().item())

    e = torch.exp(scores)
    print("③ exp 后有限?", torch.isfinite(e).all().item(),
          "inf 个数", torch.isinf(e).sum().item())

    w = e / e.sum(-1, keepdim=True)
    print("④ 归一化后有限?", torch.isfinite(w).all().item())
    return w @ V, w
```

这是“二分定位”的标准做法：在每一行后面打印一次，第一个出现 `False` 或 `inf` 的地方就是元凶。
四个位置对应的病因：

第一个出问题的位置	病因
① scores 就是 inf	q、k 的数值本身炸了（权重初始化或学习率）
② 掩码后出 nan	整行被屏蔽（见下文 padding 陷阱）
③ exp 后出 inf	**缺数值稳定**，或缩放缺失导致 logits 过大
④ 归一化后出 nan	`inf / inf`，根源仍在 ③

②那一行有个独立陷阱：padding。一个 batch 里句长不同，短句要补齐。如果某个位置的**全部**键都被掩码（比如补齐位置），它的整行就是 $-\infty$，softmax 得到 `nan`。
解法：要么保证每行至少有一个可见位置（通常保证第一个位置不被掩码），要么在 softmax 前把整行全 $-\infty$ 的情况替换成全 0（之后输出也是 0）。

i 停下来想一想 ⑯

上面说“整行全被屏蔽 → nan”。

请想一个更隐蔽的情况：**如果一整行看起来有可见位置，但那些位置的 logits 都是 −1000（非常大的负数但不是 −∞）呢？**

e^{-1000} 在 float32 下等于 0。那么这一行求和是 0，归一化时 `0/0` 得到 `nan`。

什么情况下会造出“很大的负数但不是 −∞”的 logits？（提示：想想加性掩码——有些人用 `scores + (-1e9) * (1 - mask)` 而不是 `masked_fill`。再想想：如果 q、k 本身尺度很大，正常位置的 logits 可能是 ±50，那么 −1e9 的相对差距还够大吗？）

1.13.4 排查四：一切都“正常”，但效果就是不如预期

症状：没有 nan，loss 正常下降，生成的东西也说得过去。但和同等规模的已知实现比，就是差一截。

这是最难的一类，因为没有明显症状。排查思路是**逐个要素做消融**：把每个设计要素单独关掉，看影响有多大。

消融清单（按我们的实际经验排序）：

要素	关掉之后	说明什么
$\sqrt{d_k}$ 缩放	看 §1.4 案例	小模型上影响小，大模型上灾难
残差连接	直接训不动	深度超过 2 层就必须有
LayerNorm	loss 抖动、收敛慢	不是可选配件
权重共享	参数多 36%，效果略降	小模型上影响明显
位置编码	**完全学不会顺序**	见 §1.8
warmup	前期 loss 尖刺	有时能靠调小学习率绕过
梯度裁剪	偶发 nan	稳定性保险

做消融的纪律：一次只关一个，其余全部不动。这和本章开头那个对照实验是同一条纪律。

为什么消融比“读论文”有用

论文会告诉你“我们用了 X、Y、Z”，但**很少告诉你哪些是必需的**。因为论文的语境是“我们这样做了，效果很好”，不是“我们试过不做”。

自己做一遍消融，你会得到一份**属于你这份代码**的必需性清单。它可能和论文里的暗示不一样——比如你可能会发现某个“重要技巧”在你的规模下毫无影响（就像我们的 $\sqrt{d_k}$）。

这也是本章那个案例真正的教学目的：不是让你记住“$\sqrt{d_k}$ 重要”，而是让你养成“这件事在**我这个规模**上到底重不重要”的习惯。

1.13.5 一张排查总表

把上面四条路径汇总成一张可以贴在显示器旁的表：

症状	先查什么	最可能的原因
loss 不降，梯度正常	注意力三个诊断量	缩放问题、或模型容量
loss 很漂亮但生成差	**单测：未来位置权重是否为 0**	掩码泄露
出现 nan	分四步插桩定位	缺数值稳定 / padding 全屏蔽
效果差一截	逐要素消融	缺了某个非必需但有用的组件
显存爆	算 $n^2 \times h \times L$	序列太长，需要 FlashAttention

最后一行那个公式值得记一下——它能让你在跑之前就估算出显存需求：

$$\text{注意力矩阵显存} \approx \underbrace{n^2}_{\text{位置对数}} \times \underbrace{h}_{\text{头数}} \times \underbrace{L}_{\text{层数}} \times \underbrace{2 \text{ 或 } 4}_{\text{fp16 或 fp32 字节数}} \tag{1.62}$$

$n = 4096$、$h = 32$、$L = 32$、fp16 时：$4096^2 \times 32 \times 32 \times 2 \approx 34$ GB。**这在单卡上装不下**——而如果用了 FlashAttention，这部分显存**根本不需要**。

1.14 完整实现

下面是本章所有式子的实现。每一处关键选择都标了为什么。

```python
import numpy as np


def softmax(x, axis=-1):
    """数值稳定的 softmax。

    减最大值一步是**必需的**，不是优化：
    float32 下 logits 超过约 88 就会让 exp 溢出成 inf。
    由平移不变性，减最大值不改变结果。
    """
    x = x - x.max(axis=axis, keepdims=True)
    e = np.exp(x)
    return e / e.sum(axis=axis, keepdims=True)


def attention(Q, K, V, mask=None, scaled=True):
    """缩放点积注意力。

    Q: (..., n_q, d_k)   K: (..., n_k, d_k)   V: (..., n_k, d_v)
    mask: 布尔数组，True 表示「允许看」。因果掩码取严格下三角。

    返回 (输出, 权重)。权重是 (..., n_q, n_k)，每行和为 1。
    """
    d_k = Q.shape[-1]

    # ① 打分。缩放必须在 softmax **之前**，它修正的是 logits 的尺度。
    scores = Q @ K.swapaxes(-1, -2)
    if scaled:
        scores = scores / np.sqrt(d_k)

    # ② 掩码。必须用 -inf 而不是 0，否则被屏蔽的位置仍会分到权重。
    if mask is not None:
        scores = np.where(mask, scores, -np.inf)

    # ③ 归一化。softmax 逐行做，把每一行变成概率分布。
    weights = softmax(scores, axis=-1)

    # ④ 加权求和。按权重把各位置的 v 混合起来。
    return weights @ V, weights


def causal_mask(n):
    """因果掩码：位置 i 只能看到 j <= i。"""
    return np.tril(np.ones((n, n), dtype=bool))
```

```python

def multi_head_attention(X, W_q, W_k, W_v, W_o, n_head, mask=None):
    """多头注意力。

    X: (n, d_model); W_*: (d_model, d_model)
    所有头的投影合并成一个大矩阵，再 reshape 切分——这是实际实现的方式，
    比循环 h 次快得多，且数学上完全等价。
    """
    n, d_model = X.shape
    d_head = d_model // n_head
    assert d_model % n_head == 0

    # 一次投影出所有头的 q/k/v，再拆成 (n_head, n, d_head)
    def split(W):
        return (X @ W).reshape(n, n_head, d_head).transpose(1, 0, 2)

    Q, K, V = split(W_q), split(W_k), split(W_v)

    out, weights = attention(Q, K, V, mask=mask)

    # 拼接各头：(n_head, n, d_head) -> (n, d_model)
    out = out.transpose(1, 0, 2).reshape(n, d_model)
    return out @ W_o, weights
```

> **i** 三处容易写错的地方
>
> ① **缩放的位置**（第 22 行）：必须在 `attention` 内部、softmax 之前。如果挪到函数外面，就变成了“对输出做缩放”，语义完全不同（见失败二）。
> ② **掩码的取值**（第 26 行）：`-np.inf` 而非 `0`。见失败一。
> ③ **多头的 reshape**（第 51 行）：`reshape(n, n_head, d_head)` 之后要 `transpose(1, 0, 2)` 把头的维度提前，才能让 `attention` 里的 `swapaxes(-1, -2)` 正确工作。这个维度顺序错了不会报错——只是结果不对。

复杂度：`Q @ K.swapaxes` 是 $O(n^2 d_k)$，softmax 是 $O(n^2)$，`weights @ V` 是 $O(n^2 d_v)$。三项都随 n **平方**增长。

这一段代码在 notebook 里：`notebooks/attention_from_scratch.ipynb` 把上面每一行拆成可单独运行的单元，并逐步与 PyTorch 的 `F.scaled_dot_product_attention` 对拍。

1.14.1 对拍：确认我们的实现是对的

写了实现不算完，得证明它和工业级实现一致。PyTorch 从 2.0 起提供了 `F.scaled_dot_product_attention`（内部会在合适时自动切到 FlashAttention）。拿它当标准答案：

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

def my_attention(Q, K, V, mask=None, scaled=True):
    d_k = Q.shape[-1]
    s = Q @ K.swapaxes(-1, -2)
    if scaled:
        s = s / math.sqrt(d_k)
    if mask is not None:
        s = s.masked_fill(~mask, float("-inf"))
    w = F.softmax(s, dim=-1)
    return w @ V, w
```

```
torch.manual_seed(0)
n, d = 6, 32
Q, K, V = (torch.randn(2, 4, n, d, dtype=torch.float64) for _ in range(3))
mask = torch.tril(torch.ones(n, n, dtype=torch.bool)).expand(2, 4, n, n)

mine, w = my_attention(Q, K, V, mask)
ref = F.scaled_dot_product_attention(Q, K, V, attn_mask=mask)
print("最大绝对误差:", (mine - ref).abs().max().item())
# 6.661e-16
```

最大绝对误差 6.66e-16。这正好是 float64 的机器精度量级（$\varepsilon \approx 2.2 \times 10^{-16}$，累加几步后到 10^{-15} 量级是正常的）。说明我们的实现与官方实现在数值上等价。

> 为什么要做这个对拍
>
> 自己实现的注意力有一个特点：**大多数错误不会让程序崩，只会让结果悄悄不对**。前面五个失败模式里有三个不报错，这个对拍就是它们的统一防线。
>
> 更实际的一点：如果你在调试一个自定义注意力（比如加了稀疏模式、改了缩放方式），不知道怎么验证对不对——**先退化到标准情况与官方实现对拍**，确认基础部分无误，再往上加你的修改。这比直接调试一个复杂变体高效得多。

1.14.2 实测：$O(n^2)$ 到底有多贵

光说“平方增长”不够直观。在 RTX A6000 上实测 `F.scaled_dot_product_attention`（8 头、$d_{\text{head}} = 64$、因果掩码）：

序列长度 n	注意力矩阵	单次前向耗时	相对 $n = 512$
512	512×512	0.082 ms	1.0×
1024	1024×1024	0.201 ms	2.5×
2048	2048×2048	0.581 ms	7.1×
4096	4096×4096	2.307 ms	28.2×
8192	8192×8192	8.263 ms	**101**×

读这张表要注意两件事。

第一，n 翻倍，耗时涨约 3–4 倍，不是 4 倍整。从 512 到 8192，n 涨了 16 倍，纯平方应该是 256 倍，实测只有 **101 倍**。差距来自 FlashAttention：它不把 $n \times n$ 矩阵写进显存，省下了大量访存时间。**所以“$O(n^2)$”描述的是计算量，不是实际耗时的准确预测。**

第二，这里的绝对数字很小。$n = 8192$ 只要 8 毫秒——看起来长上下文并不贵？那是因为**这只是单层单次前向。**真实的 70B 模型有 80 层，生成一个 token 要跑一遍全部层，还要加上 KV 缓存的读写。乘上去就是几百毫秒到秒级。

> ⚠ 一个容易踩的测量陷阱
>
> 我们第一次跑这组数据时，$n = 512$ 测出来是 **3.83 ms**——比 $n = 1024$ 还慢。
>
> 这不是算法问题，是**首次调用的预热开销**：CUDA 需要初始化上下文、编译 kernel、分配显存池。这些一次性成本全被算进了第一次的耗时里。
>
> 正确做法是**先跑几轮热身，再开始计时**。上面的表就是这么测的（预热 5 轮，然后测 30 次取平均）。
>
> **凡是性能数字，先问一句“预热了吗”。**这是性能测量最基本、也最常被忽略的一条纪律。

1.14.3 与真实模型的距离

最后要说清楚：本章的这个实现和真实模型差在哪里。

	本章实现	真实模型（如 70B）
参数量	42.6 万	700 亿
层数 / 头数	2 / 4	80 / 64
d_{model}	128	8192
位置编码	可学习的位置嵌入	RoPE
归一化	LayerNorm	RMSNorm
注意力实现	朴素（写完整的 n^2 矩阵）	FlashAttention
KV 缓存	无（训练场景）	有，GQA 压缩

方向可迁移，数值不可迁移——这句话在本章反复出现，这里给出它的边界：上面每一行的差异都可能影响结论的**具体数值**，但“缩放在修什么”、“softmax 为什么会饱和”这些**机制层面**的结论不受影响。

第 2 章会补齐位置编码（RoPE）与 RMSNorm；第 3 章讲 KV 缓存的实现细节。

1.15 批处理：padding、打包与掩码

真实训练不会一次只喂一句话。一个 batch 里有几十条长度不同的序列—— **而矩阵乘法要求形状一致**。这一节讲这个矛盾怎么解，以及它和因果掩码怎么叠加。

1.15.1 问题：长度不一

一个 batch 里有三条序列，长度分别是 5、3、7。要放进同一个张量，只能补齐：

```
序列 A（长 5）：  秦 川 雄 帝 宅
序列 B（长 3）：  空 山 不
序列 C（长 7）：  明 月 松 间 照 清 泉
```

补齐到 7：

```
A:  秦 川 雄 帝 宅 <pad> <pad>
B:  空 山 不 <pad> <pad> <pad> <pad>
C:  明 月 松 间 照 清 泉
```

padding 的位置不携带信息，必须让注意力**忽略**它们。否则：

- 位置“宅”会从两个 `<pad>` 取信息，输出被稀释
- `<pad>` 自己的输出毫无意义，但会参与 loss（如果不做 loss mask）

1.15.2 两种掩码要叠加

这时一个位置能不能被看到，取决于**两个条件**：

$$\text{可看} = \underbrace{j \le i}_{\text{因果掩码}} \quad \text{且} \quad \underbrace{j \text{ 不是 padding}}_{\text{填充掩码}} \tag{1.63}$$

写成代码：

```python
def build_mask(seq_len, lengths, device):
    """lengths: 每条序列的真实长度。返回 (B, 1, L, L) 的布尔掩码。"""
    B = len(lengths)
    causal = torch.tril(torch.ones(seq_len, seq_len, dtype=torch.bool, device=device))
    pad = torch.zeros(B, seq_len, dtype=torch.bool, device=device)
    for b, L in enumerate(lengths):
        pad[b, :L] = True                       # 前 L 个位置是真实 token
    # 键（列方向）不能是 padding；查询（行方向）是 padding 无所谓（输出会被丢弃）
    return causal[None, None] & pad[:, None, None, :]
```

注意最后一行：只有在**列方向**（键方向）才需要排除 padding。 行方向是 padding 的那些位置，它们的输出本来就是垃圾， 只要**在算 loss 时忽略掉**就行（用 `ignore_index`）。

⚠ 这就是“整行全是 $-\infty$”的来源

如果一个**查询位置本身是 padding**，而所有键也都把它屏蔽了， 那么这一行就是**全 $-\infty$** → softmax 得到 `nan`。

两种常见的解法和它们的代价：

① **保证每行至少有一个可见位置**（通常是位置 0，或者自己）。 做法是把掩码的对角线强制设为 True。代价：padding 位置会算出一个无意义的输出， 但只要 loss mask 正确，它不参与梯度，无伤大雅。

② **在 softmax 之前把全 $-\infty$ 的行替换成全 0**。这样 softmax 得到均匀分布，输出是 V 的平均——同样无意义，但不会是 nan。

我们推荐 ①：它更简单，而且不改变任何有效位置的数值。

1.15.3 打个比方：padding 让训练白跑

padding 不只是“实现细节”，它**直接浪费算力**。

一个 batch 里有长度 512 和长度 32 的两条序列，补齐到 512：

$$\text{有效计算比例} = \frac{512 + 32}{512 \times 2} = 53\% \tag{1.64}$$

一半的算力花在 `<pad>` 上。 如果 batch 里的长度差异更大（比如 2048 和 16）， 这个比例会低到 25% 以下。

解法叫“序列打包”（sequence packing）：把多条短序列**首尾相接**放进同一个 长度固定的窗口，让它们共享位置：

```
打包前（补到 8）：
  A：秦 川 雄 帝 宅 pad pad pad
  B：空 山 不  pad pad pad pad pad

打包后（窗口 8）：
  秦 川 雄 帝 宅 空 山 不          ← 两条序列占据了同一行的前 8 个位置
```

但这带来一个新问题：位置 5 的 “空” 不能看到位置 0–4 的 A 序列内容—— 它们是**不同**的序列，不该互相借信息。

所以打包需要**块对角掩码**：每条子序列内部用因果掩码，子序列之间完全屏蔽。这就把 式 (1.63) 变成了三分支的与运算：

$$\text{可看} = (j \le i) \text{ 且 } (\text{同一条子序列}) \text{ 且 } (j \text{ 非 padding}) \tag{1.65}$$

💡 打包是个典型的“工程优化改变算法表述”的例子

数学上，打包改变了注意力矩阵的结构——从“一个下三角”变成“多个对角块”。 **但结果完全等价**，因为掩码保证了跨块的信息不会流动。

收益：训练吞吐提升 2–4 倍（取决于长度分布）。

代价：实现复杂度上升——你要维护“每条子序列的起始位置”这个额外结构，还要保证位置编码在打包后仍然正确（这点很微妙：如果位置编码按绝对位置算，第二条子序列会从位置 5 开始编号，而不是从 0）。

所以打包必须配合“按子序列重置的位置编码”—— 这又是一个“看起来是优化、实际上牵动多处”的例子。

1.15.4 与 KV 缓存的关系

padding 的问题在**推理**时变了形态。

推理是逐 token 生成的，每次只处理 1 个新 token。这时不存在“batch 内长度不一”的问题（每条序列自己管自己），但出现了**另一种**长度问题：**KV 缓存会一直增长**。

式 (1.36) 算过：Llama-2 70B 每 token 2.6 MB。序列长 4096 时是 10.7 GB——**而且这只是缓存，不含模型权重**。

这就是为什么 MQA/GQA（小节 1.8）是必需的而非可选的：GQA 把缓存压到 1/8，从 10.7 GB 降到 1.3 GB。**没有这个优化，长上下文推理在单卡上根本放不下**。

> **i** 停下来想一想 ⑱
>
> 训练时不存在 KV 缓存的问题（因为整个序列一次算完），而推理时它成了主要瓶颈。
>
> 请想：**为什么训练不需要缓存？**
>
> 提示：训练时是“教师强制”——整条序列的**真实** token 都在手上，所以 K、V 可以一次性算完，直接用矩阵乘法。而推理时第 t 步只有前 $t-1$ 个 token，**第 t 个还没生成**，只能一个一个来。**算过的 K、V 如果不存下来就得重算——这就是缓存的动机**。
>
> 第 3 章会详细讲 KV 缓存的实现与优化。

1.16 梯度怎么穿过注意力

前面的推导全是**前向**的：给定输入，算出输出。但训练靠的是反向传播——**误差信号怎么从输出一路传回到 Q、K、V？**

这一节把整条反向链路推完。它对理解两件事至关重要：为什么 softmax 饱和会阻断训练（前面反复提到），以及为什么注意力对数值精度特别敏感（下一节）。

1.16.1 正向链路回顾

把前向过程拆成五步，每一步记下输入输出：

$$\underbrace{Q, K}_{\text{输入}} \xrightarrow{①} S = \frac{QK^\top}{\sqrt{d_k}} \xrightarrow{②} P = \mathrm{softmax}(S) \xrightarrow{③} O = PV \tag{1.66}$$

反向传播就是**反着走这条链**，每一步用链式法则。

设我们从上游拿到 $\frac{\partial \mathcal{L}}{\partial O}$（记作 dO）。目标是求出 dQ、dK、dV。

1.16.2 第③步的反向：$O = PV$

这是标准的矩阵乘法求导。若 $O = PV$，则：

$$dV = P^\top dO, \qquad dP = dO\, V^\top \tag{1.67}$$

这一步没有陷阱，和普通线性层一样。dV 是“每个值向量该往哪个方向调”，dP 是“每个注意力权重该往哪个方向调”。

1.16.3 第②步的反向：softmax 的雅可比

这一步是**整条链上最关键、也最容易错的地方**。

我们已经推过 softmax 的雅可比（式 (1.28)）：$\frac{\partial P_i}{\partial S_j} = P_i(\delta_{ij} - P_j)$。

把它作用到 dP 上，得到 dS。逐行看（每一行独立）：

$$dS_i = P_i \odot (dP_i - \langle dP_i, P_i \rangle \cdot \mathbf{1}) \tag{1.68}$$

其中 $\odot$ 是逐元素乘法，$\langle \cdot, \cdot \rangle$ 是内积，$\mathbf{1}$ 是全 1 向量。

人话翻译这一行。括号里做的是：

从“每个权重想往哪动”里，减掉“所有位置想动的加权平均”。

为什么要减？因为 softmax 有一个硬约束：**权重之和恒为 1**。所以一个位置想变大，必然意味着别的位置要变小——“往哪动”的方向里不能有“整体一起变大”这个分量。

那个 $\langle dP_i, P_i \rangle$ 就是“整体一起变大”的分量大小。减掉它，剩下的才是 softmax 真正能实现的方向。

这个模式在别处也出现过：在带约束的优化里（比如单纯形上的投影），都要先减掉“违反约束的那个方向”。softmax 的反向传播就是它的一个特例。
注意这一行里的 P_i 因子。它是**乘在外面**的：

$$dS_i \propto P_i \tag{1.69}$$

如果 P_i 是 one-hot（某个分量是 1，其余是 0），那么 dS_i 就只有那一维非零。如果 P_i 完全饱和到 $(1, 0, 0, \ldots)$，那么 $P_i \odot (\cdots)$ 会让**所有分量都变成 0**——
这就是“梯度消失”在公式层面的样子。前面我们是从雅可比范数（$\| J \|_F \to 0$）观察到的，这里从反向传播的具体形式再确认一遍。

1.16.4 第①步的反向：$S = QK^\top / \sqrt{d_k}$

同样是矩阵乘法，但要**注意那个缩放因子**：

$$dQ = \frac{1}{\sqrt{d_k}}\, dS\, K, \qquad dK = \frac{1}{\sqrt{d_k}}\, dS^\top Q \tag{1.70}$$

$\sqrt{d_k}$ 出现在分母上。这一点在讨论“缩放的副作用”时容易被忽略：
除以 $\sqrt{d_k}$ 会**同时缩小前向的 logits 和反向的梯度**。$d_k = 64$ 时，梯度被缩小 8 倍。
所以缩放不是“只有好处”——它用前向的稳定性换了一点反向的信号强度。不过在饱和的情形下（梯度本来就接近 0），这点损失和“完全传不回去”相比，是完全可以接受的。

1.16.5 掩码位置怎么处理

被掩码的位置（$j > i$）在前向时权重为 0。反向时：

$$dS_{ij} = 0 \quad \text{对所有被掩码的 } (i, j) \tag{1.71}$$

因为那些位置没有参与前向计算，它们不该收到任何梯度。
实现上，在算完 式 (1.68) 之后要把这些位置清零。**如果忘了这一步，梯度会流向“本不该被看到”的位置**—— 和掩码 bug 是同一类问题，只是出现在反向。

1.16.6 完整实现与验证

把上面五步连起来：

```
def attention_backward(Q, K, V, mask, dO):
    """手写的注意力反向传播。forward 用前面那个实现。"""
    n, d_k = Q.shape
    # —— 重做前向，拿到中间量 ——
    S = Q @ K.T / np.sqrt(d_k)
    S = np.where(mask, S, -np.inf)
    P = softmax(S, axis=-1)
    # —— 反向 ——

    # 第③步：O = P V
    dV = P.T @ dO
    dP = dO @ V.T

    # 第②步：softmax 的雅可比
    #   dS_i = P_i ⊙ (dP_i − ⟨dP_i, P_i⟩)
    dS = P * (dP - (dP * P).sum(-1, keepdims=True))

    # 掩码位置梯度清零
    dS = np.where(mask, dS, 0.0)

    # 第①步：S = QKᵀ / √d_k
    dQ = dS @ K / np.sqrt(d_k)
```

```
    dK = dS.T @ Q / np.sqrt(d_k)
    return dQ, dK, dV
```

验证方式：和 PyTorch 的 autograd 对拍。

```
import torch
torch.manual_seed(0)
n, d = 3, 4
Q = torch.randn(n, d, dtype=torch.float64, requires_grad=True)
K = torch.randn(n, d, dtype=torch.float64, requires_grad=True)
V = torch.randn(n, d, dtype=torch.float64, requires_grad=True)
mask = torch.tril(torch.ones(n, n, dtype=torch.bool))

S = (Q @ K.T / d**0.5).masked_fill(~mask, float("-inf"))
O = torch.softmax(S, -1) @ V
O.backward(torch.randn_like(O))

# 手写的结果（用上面那个函数，转成 numpy 再转回来）
# ... 实测：dQ、dK、dV 与 autograd 的最大误差都是 0.000e+00
```

我们实测的误差是 **0**——在 float64 下逐位一致。

> 为什么要手推一遍反向
>
> 框架会自动帮你算梯度，所以“能用”这个层面你不需要手写。
> 但**手推一遍能解释三件框架不会告诉你的事**：
>
> 1. **为什么饱和会阻断训练**——因为在 式 (1.69) 里，dS 正比于 P，而饱和时 P 在非峰值位置全是 0
> 2. **为什么缩放会同时影响反向**——$1/\sqrt{d_k}$ 在 式 (1.70) 里出现两次
> 3. **为什么掩码在反向也要处理**——不只是前向的事
>
> **“知道框架会算”和“知道它在算什么”，差别在出问题时能不能定位。**

1.17 数值精度：为什么注意力格外敏感

前面反复出现“float32 下 logits 超过约 88 就会溢出”。但今天的模型 **大多在 fp16 或 bf16 下训练和推理**。这一节讲清楚这意味着什么。

1.17.1 三种格式的可用范围

`torch.finfo` 给出的实测数据：

格式	最大可表示值	exp 能算到 logits ≈	机器精度
float32	3.40×10^{38}	**88.7**	1.2×10^{-7}
float16	6.55×10^{4}	**11.1**	9.8×10^{-4}
bfloat16	3.39×10^{38}	**88.7**	7.8×10^{-3}

第二列是这一节的核心。float16 的最大值是 65504，所以 $\exp(z)$ 在 $z \approx \ln(65504) = 11.1$ 时就溢出了—— **比 float32 的门槛低了 8 倍**。

1.17.2 这个门槛有多容易被越过

按脚注的假设（q、k 各分量方差为 1），**不缩放**时 logits 的标准差是 $\sqrt{d_k}$。取四倍标准差作为“常见最大值”：

d_{head}	标准差	常见最大值	在 fp16 下
32	5.7	**22.6**	**已经溢出**
64	8.0	**32.0**	**已经溢出**
128	11.3	**45.3**	**已经溢出**
256	16.0	**64.0**	**已经溢出**

⚠ 这张表说明一件很实际的事

在 fp16 下，不缩放的注意力几乎必然溢出——连 $d_{\text{head}} = 32$ 都撑不住。
而本章案例里，在 **fp32** 下 $d_{\text{head}} = 32$ 时训练完全正常（差 0.17）。**同一个“没有 bug”的实现，换个精度就崩了。**
这解释了一个常见的现象：有人在小模型上（fp32）调试一切正常，搬到真实训练脚本（fp16 混合精度）里立刻 nan。**不是实现错了，是精度换了。**

1.17.3 那为什么还用 fp16

因为**显存和带宽**。fp16 只占 fp32 一半的字节数，而正如 小节 1.8 说的，注意力的瓶颈常常是**访存而非算力**。用 fp16 意味着同样显存能放两倍的 batch、两倍的序列长度。
代价就是上面那张表：可表示的数值范围小了一大截。

1.17.4 bf16：Google 的方案

bfloat16 的思路很直接：**保留 fp32 的指数位（8 位），砍掉尾数位**（从 23 位砍到 7 位）。
结果是：

- **数值范围和 fp32 完全一样**（指数位相同）→ exp 也能算到 88.7
- **精度差很多**（尾数只有 7 位）→ 1.001 存进去会变成 1.000

实测这个精度损失：

```
x = torch.tensor([1.001], dtype=torch.float32)
print(x.to(torch.float16).float())    # 1.000977  ← 分辨得出
print(x.to(torch.bfloat16).float())   # 1.000000  ← 分辨不出
```

fp16 和 bf16 的取舍，一句话说清：

- **fp16**：精度高（10 位尾数），范围窄（最大 65504）
- **bf16**：范围宽（和 fp32 一样），精度低（7 位尾数）

对注意力来说，范围比精度重要。因为注意力的数值跨度极大——logits 可能从 -50 到 $+50$，softmax 之后又要在 10^{-20} 量级上区分。**范围不够会直接溢出成 nan；精度不够只是结果略微不准。**
所以今天的主流是 **bf16**（A100/H100 原生支持，训练稳定），而 fp16 更多用在**推理**场景（精度要求相对低，且有成熟的 loss scaling 机制）。
这是“选数值格式”这件事的一般原则：先保证范围够，再谈精度。

1.17.5 混合精度训练里的三个保护措施

即使选了 bf16，生产级的训练脚本通常还有几层保险：
① **数值稳定的 softmax**。前面讲过——先减最大值。这一条在 bf16 下**仍然是必需的**（因为减完之后指数最大只到 1，远离任何格式的上限）。
② **Loss scaling**。主要用于 fp16：因为 fp16 的**最小正规数**是 6.1×10^{-5}，很多梯度会直接下溢成 0。做法是先把 loss 乘一个大的常数（比如 1024），反向传播后再把梯度除回来。
bf16 因为指数范围和 fp32 一样，**通常不需要 loss scaling**——这是它比 fp16 省心的主要原因之一。

③ **关键部分保持 fp32**。常见做法是 LayerNorm、softmax、以及损失计算 在 fp32 下做（这叫“autocast 白名单”）。原因是这些操作涉及**累加**——n 个数相加时，低精度的舍入误差会累积。

> **i** 停下来想一想 ⑰
>
> 上面说“softmax 在 fp32 下做”。
> 请想：**softmax 里哪一步涉及累加，为什么低精度会有问题？**
> 具体一点：如果一行有 4096 个位置，每个权重约 2.4×10^{-4}，用 bf16（精度 7.8×10^{-3}）算它们的和会怎样？
> （提示：bf16 分辨不出比 7.8×10^{-3} 更小的相对差异。而 2.4×10^{-4} 比它小了 30 倍。）
> **这就是为什么归一化操作通常要在高精度下做。**

1.17.6 一张实践清单

如果你要写一个能在混合精度下稳定运行的注意力实现：

措施	作用	不做会怎样
减最大值的 softmax	把 exp 的输入限制在 ≤ 0	溢出成 inf
$\sqrt{d_k}$ 缩放	把 logits 拉回单位方差	fp16 下大概率溢出
softmax 用 fp32 累加	避免小权重被舍入掉	权重和不等于 1
优先选 bf16	范围和 fp32 一样	fp16 需额外的 loss scaling
梯度裁剪	兜住偶发的梯度尖峰	一次异常样本毁掉训练

这五条里，前两条是本章反复强调的“缩放与稳定”，后三条是工程实践。两件事都要有——算法上的正确性和工程上的稳健性，缺一不可。

1.18 怎么把这个模型训出来

前面的案例用的是真实训练出来的模型。这一节把训练代码拆开讲，让你能自己改动它、复现它。
完整代码在 `notebooks/tiny_gpt.py`（模型与训练循环）和 `notebooks/run_ch1_case.py`（案例的对照实验）。这里只讲**为什么这么写**。

1.18.1 数据：字符级分词

```python
class CharTokenizer:
    """只保留最高频的 n 个字，其余映射到 <unk>。字表大小本身是个超参数。"""
    def __init__(self, text, vocab_size):
        cnt = Counter(text.replace("\n", ""))
        keep = [c for c, _ in cnt.most_common(vocab_size - 2)]
        self.itos = ["<pad>", "<unk>"] + keep
        self.stoi = {c: i for i, c in enumerate(self.itos)}
```

为什么用字符级而不是词级？ 三个理由：

1. **不需要分词器**。中文分词本身是个有争议的问题（“研究生命起源”该切成什么?），字符级绕开了它。
2. **词表小**。25 万字唐诗里只有 4977 个不同字符，取前 1200 个就覆盖 87.6%。词级分词动辄三五万词表，光嵌入矩阵就比我们整个模型大。
3. **教学上更透明**。每个 token 就是一个字，你能直接看懂模型在预测什么。

代价是序列变长（一句话的字数是词数的 1.5–2 倍），上下文里能放的内容更少。真实的中文大模型用 BPE 之类的子词分词。

为什么砍到 1200 个字

保留全部 4977 个字的话，嵌入矩阵是 $4977 \times 128 = 63.7$ 万参数——比模型其余部分加起来还多，而且低频字（出现一两次）根本学不好。

砍到 1200 个是**覆盖率与参数量的折中**：覆盖 87.6% 的字，嵌入矩阵降到 15.4 万参数。剩下 12.4% 的字都映射到 `<unk>`——这意味着模型对生僻字完全无能为力，但作为教学模型这完全可以接受。

这个权衡在真实系统里同样存在，只是数字不同：词表从 32K 扩到 128K，嵌入参数翻两番，换来的是更短的处理序列和更少的 `<unk>`。

1.18.2 模型：两个关键细节

模型的主体（嵌入 + 若干 Block + 输出层）是标准的。但有两处值得单独说：

① **权重共享（weight tying）**。

```
if cfg.tie_weights:
    self.head.weight = self.tok_emb.weight
```

输入嵌入是“把字变成向量”（$V \times d$），输出层是“把向量变回字”（$d \times V$）。让它们**共用同一个矩阵**，有两个好处：

- **省参数**：$V \times d$ 个参数，在这个模型里是 15.4 万，占总参数的 36%
- **有道理**：如果“月”这个字的向量是 e，那么“某个向量是不是月”就应该看它与 e 的相似度。两个方向用同一套表示是合理的

这个技巧从原论文就有，今天的大模型普遍沿用。

② **残差分支的缩放初始化**。

```
for name, p in self.named_parameters():
    if name.endswith("proj.weight") or name.endswith("mlp.2.weight"):
        nn.init.normal_(p, mean=0.0, std=0.02 / math.sqrt(2 * cfg.n_layer))
```

注意那个 $\sqrt{2n_{\text{layer}}}$。它**不是**缩放的 $\sqrt{d_k}$——两个东西完全不同，只是都用了平方根。

它解决的问题是：残差连接 $x = x + f(x)$ 每经过一层就把 $f(x)$ 累加上去。如果 f 的输出方差是 σ^2，n 层之后 x 的方差会累积到 $n\sigma^2$。把 f 的初始化标准差按 $1/\sqrt{2n}$ 缩小（2 是因为每层有两个残差分支：注意力和 FFN），可以让累积后的方差保持在 1 附近。

这是 GPT-2 论文里的做法，目的是**让深层网络的训练更稳定**。

别把两个 根号 搞混

	缩放对象	出现在哪	解决什么
$1/\sqrt{d_k}$	**注意力打分**	前向传播里	logits 随维度膨胀 → softmax 饱和
$1/\sqrt{2n_{\text{layer}}}$	**权重初始化**	训练开始前	残差累加 → 激活值方差膨胀

两个都是“按维度的平方根缩放”，但**作用对象和失效模式完全不同**。名字里都有 根号 不代表它们是同一件事——这在读论文时是个常见的混淆点。

1.18.3 训练循环：四个必须有的部件

```
opt = torch.optim.AdamW(model.parameters(), lr=cfg.lr,
                        weight_decay=cfg.weight_decay, betas=(0.9, 0.95))

for step in range(cfg.max_steps):
```

```
    for g in opt.param_groups:
        g["lr"] = lr_at(step, cfg)              # ① 学习率调度
    x, y = get_batch(tr, cfg.block_size, cfg.batch_size, gen)
    _, loss = model(x, y)
    opt.zero_grad(set_to_none=True)
    loss.backward()
    gn = torch.nn.utils.clip_grad_norm_(model.parameters(), cfg.grad_clip)  # ② 梯度
裁剪
    opt.step()
```

① **学习率调度**（warmup + 余弦衰减）：

```
def lr_at(step, cfg):
    if step < cfg.warmup:                       # 前 100 步线性升温
        return cfg.lr * (step + 1) / cfg.warmup
    prog = (step - cfg.warmup) / (cfg.max_steps - cfg.warmup)
    return cfg.lr * (0.1 + 0.9 * 0.5 * (1 + math.cos(math.pi * min(1.0, prog))))
```

为什么需要 warmup? 训练刚开始时参数是随机的，梯度方向没什么意义。如果一上来就用大学习率，会朝一个随机的方向猛冲一步，可能把模型带到一个坏的区域。先小步走 100 步，等梯度方向稳定了再加到全速。

为什么需要衰减? 训练后期模型接近一个较好的解，大步长会让它在解的附近来回跳。余弦衰减让它逐渐减小步长，精细收敛。

② **梯度裁剪**：`clip_grad_norm_` 把梯度的范数限制在 1.0 以内。如果某一步梯度突然很大（比如遇到了异常样本），它会按比例缩小，避免一步把参数带飞。

判断训练是否正常的三个信号

1. **loss 曲线**：应该平滑下降，偶尔抖动正常。如果出现尖刺，多半是某个 batch 有问题；如果长期不降，检查学习率。
2. **梯度范数**：我们记录它就是为了这个。健康的值在 0.1–10 之间波动。**如果它长期贴着裁剪阈值（1.0），说明学习率偏大；如果它掉到 10^{-4} 以下并且不回升，说明梯度消失了**——本章案例里不缩放那一边就是这样（雅可比范数掉到 8×10^{-5}）。
3. **注意力权重熵**：健康值在 0.5–0.9 之间。**接近 1.0 说明退化成平均**（缩放过度），**接近 0 说明饱和**（缩放不足或缺失）。

这三个量合起来，能在 loss 曲线还没表现出问题时就提前发现异常。

1.18.4 超参数是怎么定的

参数	值	为什么
`n_layer`	2	一层学不出“头的分工”，三层以上训练变慢且现象一样
`n_head`	4	够看出“不同头做不同事”（实测 4 个头确实分化了）
`d_model`	128	参数量约 42.6 万，CPU 上几分钟能训完
`block_size`	64	唐诗一句 5–7 字，64 个 token 能容纳好几联
`vocab_size`	1200	覆盖 87.6% 的字，见上文
`batch_size`	32	显存与梯度稳定性的折中

参数	值	为什么
`lr`	3e-3	小模型用大学习率；实测这个值收敛最快且不发散
`warmup`	100	约总步数的 7%，够让梯度方向稳定
`max_steps`	1500	**经验值**：loss 在这个位置基本平了（见 图 1.2）

最后一行值得说一句：`max_steps = 1500` 不是因为“训练完了”，而是因为**再训下去收益很小**。判断依据是 loss 曲线的斜率——从 1000 步到 1500 步只降了 0.1 左右，而前 200 步降了 2 以上。

训练步数永远是“够用就好”，不是“越多越好”。这个判断要靠看曲线，不能靠设一个固定值。

1.18.5 想自己改?

改这几个地方最能看到效果：

```
# ① 复现"训不动"：把缩放关掉，同时把头数改成 1（d_head 变成 128）
cfg = Config(scaled=False, n_head=1, d_model=128)

# ② 看缩放到底有没有影响：只关缩放，保持 4 头
cfg = Config(scaled=False, n_head=4, d_model=128)

# ③ 加深：看 4 层的模型能不能学出更明显的层间分工
cfg = Config(n_layer=4)

# ④ 换语料：把 data/corpus.txt 换成别的文本，重跑
```

改完跑 `run_ch1_case.py`，对比 loss 曲线和 `analyze_heads.py` 的头部画像。**自己动手调一遍，比读十遍推导有用**。

1.19 对照实验是怎么搭的

本章那个案例给了三组实验：初始化扫描、小模型对照、大模型对照。结论已经讲过了，这里讲**它们各自是怎么设计的**——因为设计一个能说明问题的实验，和跑一个实验，是两件事。

1.19.1 实验一：初始化扫描

要回答的问题：$q^\top k$ 的实际尺度到底有多大？和理论值 $\sqrt{d_k}$ 差多少？

为什么不能直接量训练后的模型：训练后的 q、k 是学出来的，尺度已经被优化过程改变了。要检验“脚注的假设成立吗”，必须量**初始化状态**——那是假设声称成立的地方。

做法：

```
def init_sweep(text, d_heads):
    out = []
    for d_model in d_heads:
        for scaled in (False, True):
            cfg = Config(d_model=d_model, n_head=1, n_layer=1, scaled=scaled,
                         vocab_size=300, block_size=32, device=DEV)
            torch.manual_seed(0)                  # ① 固定种子
            m = TinyGPT(cfg).to(DEV).eval()       # ② 不训练，只初始化
            x = torch.randint(2, 300, (8, 32), device=DEV)
            with torch.no_grad():
                m(x, record=True)                 # ③ 走一次前向，采集诊断量
            d = m.blocks[0].attn.last_diag
            out.append({...})
    return out
```

三个设计要点：

① **固定种子**。 每个 d_{head} 都用同一个种子初始化。 如果不固定，不同维度之间的差异里就混进了初始化噪声—— 而我们想看的恰恰是“维度”的影响。

② **不训练，只初始化**。这是这个实验的关键。 训练会改变 q、k 的尺度，把我们要观察的对象污染掉。

③ **用同一批输入**。 `x = torch.randint(2, 300, (8, 32))` 在所有配置下都是同一批， 保证唯一的变量是 d_{head}。

为什么用 $n_{\text{head}} = 1$：这样 $d_{\text{head}} = d_{\text{model}}$， 可以直接用 d_{model} 的列表控制 d_{head}。 如果保持 4 头，d_{model} 就得是 d_{head} 的 4 倍，多一层换算，容易出错。

> 为什么要 400 次采样取平均
>
> 第一次做这个扫描时，我们每个配置只采了一次样，结果曲线是**锯齿状**的—— 完全看不出趋势。
> 原因：单次采样的标准差本身就是随机变量。$d_{\text{head}} = 32$ 时理论标准差是 5.66，但单次实测可能在 4 到 8 之间浮动。 这些浮动完全盖过了“维度增长带来的趋势”。
> **取平均是唯一可靠的办法**。 我们的做法是每个配置采 400 组， 算它们的标准差。这样得到的是“标准差的标准差”趋于稳定的估计。
> **这条经验是通用的**：任何“测量一个随机量的典型大小”的实验， 都要先问一句“我采了多少次、平均值稳定了吗”。

为什么用双对数坐标：因为这能一**眼看出标度关系**。如果实测值正比于 $d^{0.5}$，那么在双对数坐标下就是一条斜率 0.5 的直线。 如果正比于 d^1， 斜率就是 1。

我们量出来的两条线**斜率相同、截距不同**—— 这就是“形式对了、常数不对”的图形化证据。如果用普通坐标，两条曲线都是向上弯的弧线，看不出这个关系。

1.19.2 实验二与三：受控对照

要回答的问题：去掉 $\sqrt{d_k}$ 会怎样？这个“怎样”和维度有关系吗？

核心设计：两两对照，每次只改一个变量。

```
          不缩放          缩放后
小模型     A1      vs      A2          ← 第一组：d_head = 32
大模型     B1      vs      B2          ← 第二组：d_head = 256
```

四组实验，**共用的东西**：语料、随机种子、步数、批大小、学习率调度、优化器。 **变化的东西**：只有两个——是否缩放，以及 d_{head}。

为什么 A 组和 B 组不能直接比：因为 A 和 B 的参数量不同 （42.6 万 vs 137.6 万），loss 的**绝对值**没有可比性。 所以我们的结论全部建立在**组内比较**上：

- A 组内：4.17 vs 4.00，差 0.17
- B 组内：5.18 vs 4.23，差 0.95

“差的倍数”才是跨组可比的那个量——0.95 / 0.17 ≈ 5.6 倍。 这是一个**无量纲的数**，它告诉我们“维度增大之后，去掉缩放的影响放大了多少”。

> ⚠ 一个容易犯的错：拿绝对 loss 跨组比
>
> “大模型的 loss 是 5.18，比小模型的 4.17 差” —— 这句话是**没有意义**的，因为两个模型的参数不同、容量不同。
> 参数量不同时，**loss 的绝对值不可比**。可比的是： ① 同一模型内不同配置的差；② 这个差相对于某个基准的**倍数**； ③ 与“随机猜测”（$\ln V$，V 是词表大小）的距离。
> 随机猜测的 loss 是 $\ln 1200 \approx 7.09$——这正好是四条曲线的起点（7.03–7.05）。因为初始化时模型输出的分布接近均匀，loss 自然接近 $\ln V$。**这个值可以作为“什么都没学到”的基准线**。

为什么要跑三次种子：单次对照的结果可能是巧合。 我们的做法是换三个种子重跑小模型那一组，看到差距都在 0.1–0.2 之间浮动（和训练本身的随机性同量级），才敢下“这个规模下没有影响”的结论。

如果三次结果一次是 0.05、一次是 0.8、一次是 0.3，那说明噪声太大，0.17 这个数不可信，需要更多次或者更大的样本。

1.19.3 诊断量是怎么算的

实验里用到的三个诊断量，算法都写在这里，方便你自己实现。

① **权重熵**（归一化到 $[0,1]$）：

```
def weight_entropy(p, n_valid):
    """p 是一行的注意力权重，n_valid 是这一行可见的位置数。
    归一化让不同长度的行可以互相比较。"""
    h = -(p * torch.log(p.clamp_min(1e-12))).sum()
    return h / math.log(n_valid)      # ← 除以最大可能的熵
```

为什么要归一化：位置 5 只能看 6 个位置，位置 100 能看 101 个。不做归一化的话，后者的熵天然更大——但那不是因为“注意力更分散”，只是因为“可选项更多”。除以 $\log n_{\text{valid}}$ 消掉这个效应。

② **雅可比范数**：

```
def jacobian_frobenius(p):
    """J_ij = p_i(δ_ij − p_j)，返回 Frobenius 范数。
    这个量衡量 softmax 的输出对输入的敏感度。"""
    J = torch.diag_embed(p) - p.unsqueeze(-1) * p.unsqueeze(-2)
    return J.flatten(-2).norm(dim=-1)
```

注意实现细节：不要真的去构造一个 $n \times n \times n$ 的张量（$n = 4096$ 时是 687 亿个元素）。上面的写法只构造 $n \times n$，利用了对角矩阵的稀疏结构。

③ **logit 标准差**：直接 `logits.std()`，但要**只统计有效位置**——被掩码的位置是 $-\infty$，会把统计量污染成 `nan` 或 `inf`。

```
valid = mask.expand_as(logits).bool()
lg = logits[valid]                    # ← 只取有效位置
logit_std = lg.std().item()
```

> 这三个量的实现里藏着一个通用原则
>
> **每个诊断量都要问一句“它在什么条件下有意义”。**
>
> · 熵：只在“可见位置数 > 1”时有意义（位置 0 只能看自己，熵恒为 0）
> · 雅可比范数：只在分布未饱和时有区分度（饱和后都是 0，无法区分「多饱和」）
> · logit 标准差：必须排除被掩码的位置
>
> **不加这些限定，你会得到一堆看起来正常但实际无意义的数字。**这是做测量时最常见的陷阱——数字不会报错，它只是不说话。

1.20 动手练习

四个练习，从易到难。都能在本章的代码基础上完成。

1.20.1 练习一：复现，并改一个变量

跑一遍 `run_ch1_case.py`，确认你能得到与本章一致的数字（loss 4.17 / 4.00 / 5.18 / 4.23，熵 0.0001）。

然后**只改一个变量**：把 `n_head` 从 4 改成 2（d_{head} 从 32 变成 64）。预测一下会发生什么，然后跑一遍验证。

预期的思考方向：d_{head} 翻倍意味着 logits 的标准差大约翻 $\sqrt{2}$ 倍，离饱和更近一步。但 64 仍然远小于 256——所以差异应该还是“不大但不为零”。

1.20.2 练习二：找到“临界维度”

设计一组实验，找出“在不缩放的情况下，d_{head} 大到多少，训练开始明显变差”。

建议的维度序列：32、64、96、128、192、256。

要求：- 每个维度跑缩放和不缩放两版，共 12 次训练 - 每个都记录 loss 曲线和最终熵 - 画一张图：横轴 d_{head}，纵轴“两个版本的 loss 差” - 找出这个差值开始明显上升的位置

这个练习的价值：你会得到一条**属于你自己的**“临界曲线”。它依赖于你的语料、你的初始化、你的超参数——但这正是重点：**“什么时候会崩”不是一个通用常数，是你这个系统的属性**。

1.20.3 练习三：给 `analyze_heads.py` 加一个指标

现在的四个指标是：平均熵、对角权重、前一位权重、平均距离。

请加第五个：“注意力重心的位置分布”——统计每个位置的注意力重心落在自己左边多远、以及它的方差。

有什么用：如果某个头的“重心距离”方差很大，说明它在不同位置的行为不一致（有时看近处，有时看远处）。这类“条件性”的头往往比“固定模式”的头更有意思，也是第 4 章会重点分析的。

实现提示：`dist = sum((i-j) * p[i][j] for j in range(i+1))`，然后统计所有位置的 `dist` 的均值和标准差。

1.20.4 练习四：做一个“注意力调试器”

把本章排查手册里的四步插桩整合成一个小工具：

```
class AttentionDebugger:
    """在训练循环里挂上它，异常时自动打印诊断。"""
    def __init__(self, model, log_every=50):
        ...
    def check(self, step):
        """检查三个诊断量，超阈值就警告。"""
        ...
    def on_nan(self):
        """检测到 nan 时，二分定位到具体哪一步。"""
        ...
```

要求：- 熵 > 0.99 或 < 0.01 时报警 - logits 标准差超过某个阈值时报警 - 检测到 nan 时自动跑一遍四步插桩，打印第一个出问题的位置

这个练习最接近真实工作。本章讲的每个诊断量，在真实项目里都应该被自动化——**人工盯 loss 曲线发现不了这些问题**。

> 做完这些练习之后
>
> 你会发现一件事：**这一章的内容，跑一遍比读一遍快得多**。
>
> 四个练习加起来大概需要两三个小时，但它们会给你：- 一份属于你自己系统的“临界维度”数据 - 一套能直接用在其他项目上的诊断工具 - 对“机制的前提”这件事的肌肉记忆
>
> **如果只做一件事，做练习二**。它是本章案例的直接延伸，也是最能训练“实验设计直觉”的一个——**而这个直觉，是本书想教给你的最核心的东西**。

1.21 自测

十道题。前六道能在本章找到直接答案，后四道需要自己推。**答案和解析在下一节**——先做完再看。

1. 注意力为什么要除以 $\sqrt{d_k}$ 而不是 d_k？用一句话说清“要的是让方差等于多少”。

2. 把缩放写在 softmax **之后**，会产生什么可测量的错误？给出一个具体的检查手段。

3. 一个多头注意力层，$d_{\text{model}} = 512$，$n_{\text{head}} = 8$。问：$\sqrt{d_k}$ 里的 d_k 应该是多少？如果误写成 512，会发生什么？

4. 因果掩码用 0 而不是 $-\infty$，训练 loss 会变好还是变差？为什么？这个 bug 在什么阶段才会暴露？

5. 为什么“注意力权重高”不能直接当成“因果关系”？要验证因果性该做什么？

6. 本章的小模型在有位置编码的情况下，层 0 头 1 学出了“看前一位”的模式。如果**去掉**位置编码，这个头还能学出来吗？为什么？

7.（推导）设 $q, k \in \mathbb{R}^d$ 各分量独立、均值 0、方差 σ^2（不一定是 1）。求 $\mathrm{Var}(q^\top k)$，并说明 $\sqrt{d_k}$ 这个缩放 在 $\sigma \neq 1$ 时是否还合适。

8.（推导） 证明 softmax 的雅可比矩阵的**每一行之和为零**：$\sum_j \frac{\partial p_i}{\partial z_j} = 0$。 这个性质说明了什么？（提示：和“权重和恒为 1”这个约束有什么关系？）

9.（计算） 某模型 $n_{\text{layer}} = 24$、$n_{\text{head}} = 16$、$d_{\text{head}} = 64$、序列长度 2048、fp16。 求注意力矩阵占用的显存（按 式 (1.62)）。这个数字意味着什么？

10.（设计） 你要实现一个“只看前 w 个位置”的滑动窗口注意力 （$w = 256$）。请写出掩码的构造方式，并说明它相比全注意力省下了什么、 **没有**省下什么。

1.22 检查点解答

十八个「停下来想一想」分散在正文各处。**其中十六个**在这里集中给答案 （⑰ 和 ⑱ 的答案直接写在正文那一段里了）；**先自己想过再看**， 否则这一节起不到作用。

1.22.1 ① 面对“没复现出来”，下一步查什么

该查前提。 脚注的推导链是“方差为 1 $\Longrightarrow$ 方差等于 d_k $\Longrightarrow$ 数值大 $\Longrightarrow$ 饱和”。 实验中现象没出现，可能的原因有两类：推导错了，或者前提不成立。

而这条链上**只有第一个箭头是假设**，其余都是数学。所以最该测的是： q、k 的方差真的是 1 吗？——实测 0.070 对 5.66，答案是否定的。

推广一点：任何“理论预测与实验不符”的情况， 先检查理论的前提是否在你的系统里成立，再看推导。 推导错误很少见，前提失效很常见。

1.22.2 ② loss 停在 5.18 之后会怎样

会继续缓慢下降，但永远追不上缩放版。 因为模型的其他部分 （词嵌入、MLP、输出层）还在学习，它们能独立贡献一部分性能。

但注意力层已经冻结成“每个位置只看一个固定位置”， 这等价于一个**固定的一阶转移**。模型失去了“根据上下文动态选择信息”的能力—— 而这个能力恰恰是 Transformer 的核心。所以它会稳定在一个明显更差的水平上。

继续训 10000 步，曲线的形状会是：先慢降，然后**趋于平缓**，最终稳定在比缩放版高 0.8–1.0 的位置。差距不会消失，因为这不是训练不足，是架构能力缺失。

1.22.3 ③ 尖锐权重与硬查表的梯度区别

有区别，而且这个区别正是软查表存在的理由。

硬查表的梯度是**严格为零**（除了不可导点）——一点信息都传不回去。

尖锐的 softmax（比如 $p = [0.9999, 0.00005, 0.00005]$）梯度**很小但不为零**：

$$J_{11} = p_1(1 - p_1) = 0.9999 \times 0.0001 = 9.999 \times 10^{-5} \tag{1.72}$$

这个小到几乎无用的数字，仍然大于零。而只要它大于零，梯度就能传回去，模型就有机会把 p 从尖锐**拉回来**——这是硬查表做不到的自我修正能力。

本章案例里，实测的 $\| J \|_F$ 掉到了 8×10^{-5}， 已经接近“事实上等于零”。**这就是为什么它表现为“训不动”而不是“报错”**。

1.22.4 ④ one-hot 的雅可比

$p = [1, 0, 0]$ 时，逐项算 $J_{ij} = p_i(\delta_{ij} - p_j)$：

- $J_{11} = 1 \times (1 - 1) = 0$
- $J_{12} = 1 \times (0 - 0) = 0$
- $J_{13} = 1 \times (0 - 0) = 0$
- $J_{2j} = 0 \times (\ldots) = 0$（第 2 行整行为零，因为 $p_2 = 0$）
- $J_{3j} = 0$（同理）

整个矩阵是零矩阵。 而 $p = [0.9999, 0.00005, 0.00005]$ 时 $J_{11} = 9.999 \times 10^{-5}$——比零大，但比正常的 $O(0.1)$ 小了三个数量级。

两者在**行为上**几乎一样（都只看一个位置），在**梯度上**差着数量级。这就是“饱和”这个连续过程的实际含义：不是突然死掉，而是**梯度持续衰减，直到训练事实上停滞**。

1.22.5 ⑤ 为什么缩放之后正反馈不发生

因为**缩放把“放大 q、k”的收益路径堵住了**。

不缩放时：模型把 $\|q\|$、$\|k\|$ 放大一倍，logits 就翻倍，注意力更尖锐，在训练集上 loss 立刻下降——**这是一个有即时回报的动作**。所以梯度下降会鼓励它继续放大，形成正反馈，直到饱和。

缩放之后：logits 是 $q^\top k/\sqrt{d_k}$。要让它翻倍，$\|q\|$、$\|k\|$ 得放大 $\sqrt{d_k}$ **倍**（$d_k = 256$ 时是 16 倍）。而放大权重 16 倍会让其他所有依赖这些权重的计算失控—— 代价远大于收益，所以模型不会走这条路。

更本质的说法：缩放把“增加尖锐度”的**杠杆比**改变了。不缩放时杠杆比是 1:1（放大一倍得一倍），缩放后是 $\sqrt{d_k}:1$（要放大 16 倍才得一倍）。**这不是禁止了那条路，而是让它变得不划算**。

1.22.6 ⑥ 在线 softmax 的纠正因子

$m_{\text{old}} = m_{\text{new}}$ 时，$e^{m_{\text{old}}-m_{\text{new}}} = e^0 = 1$—— **不需要纠正**，直接累加即可。这也是最常见的情况（最大值通常在前几个块就出现）。

新最大值远大于旧最大值时，$e^{m_{\text{old}}-m_{\text{new}}} \to 0$—— 之前累积的 ℓ 被压缩到接近零，**旧块的信息在数值上被丢弃了**。

这提示一个真实的精度问题：**如果最大值出现在很靠后的块，前面所有块的信息会被压缩掉**，可能损失精度。

在 float32 下，当 $m_{\text{new}} - m_{\text{old}} > 88$ 时，$e^{\Delta} \to 0$，前面块的信息**完全消失**。而 FlashAttention 论文里的做法是：在分块时就把 Q 按行做缩放，让同一行内的 logits 分布尽量集中—— 这既是性能优化，也是数值精度的保障。

1.22.7 ⑦ 长上下文为什么从死结变成工程问题

两个角度，缺一不可。

信息通路长度。RNN 里第 1 个词的信息传到第 n 个位置要经过 $n-1$ 次变换，每次都有衰减风险。序列越长，能保留下来的早期信息越少—— 这是**架构层面的信息瓶颈**，不是算力问题。注意力把这个距离压到 **1 步**（第 n 个位置直接与第 1 个位置做点积），所以“记住很久以前的东西”在原理上不再是难题。

能否并行。RNN 的第 t 步依赖第 $t-1$ 步，**本质上无法并行**—— 序列多长就得串行多少步。注意力对序列的所有位置是一次矩阵乘法，n 个位置同时算，**吃满了 GPU 的并行能力**。

所以答案分两层：信息通路短了，让“长上下文有用”成为可能；可并行了，让“长上下文可行”成为现实。

但注意——“不再有架构瓶颈”不等于“免费”。KV 缓存的大小随序列线性增长（式 (1.36) 算过：每 token 2.6 MB），这才是今天真正的限制。**瓶颈从“模型记不住”变成了“显存装不下”**——问题的性质变了。

1.22.8 ⑧ 拼接与分别投影的参数量

参数量完全一样。W^O 是 $d_{\text{model}} \times hd_v$，切成 h 块就是每个 $hd_v/h = d_v$ 列，所以 h 个小矩阵 W_i^O 各有 $d_{\text{model}} \times d_v$ 个元素，加起来还是 $d_{\text{model}} \times hd_v$。

既然一样，为什么写成拼接?

计算效率：一次矩阵乘法 vs h 次小矩阵乘法。GPU 上大矩阵乘法的效率远高于多次小矩阵乘法（kernel 启动开销、并行度、显存访问模式都更优）。$h = 64$ 时，“一次大乘法”比“64 次小乘法再相加”快得多。

代码简洁性：`torch.cat([...], dim=-1) @ W_o` 一行；分别投影再相加要写循环和累加。

而且框架会把它们融合：实际实现里，`cat` 之后立刻做矩阵乘法，编译器（如 TorchInductor）会把它融合成一个 kernel，连 cat 带来的显存拷贝都省掉了。

一个反复出现的模式

数学上的等价形式，工程上不一定等价。 本章已经出现三次了：

· “拼接 + 投影” 与 “分别投影 + 相加”（本节）
· “在线 softmax” 与 “批量 softmax”（小节 1.11 前的第六个检查点）
· “写完整矩阵” 与 “分块不写”（FlashAttention）

三处的数学结果都一样，但访存模式、并行度、数值行为截然不同。**这是从“懂原理”到“能实现”之间那道坎的核心内容。**

1.22.9 ⑨ 一个“只看自己”的头有什么用

答案：它把残差路径变成了一个可控的“信息保险箱”。

如果一个头输出的对角权重接近 1，它做的事情就是“把当前内容做一个线性变换，然后加回自己”。这**不是“什么都没做”**—— 它是“保留并重新编码当前信息”。具体有两个用处：

① **对抗其他头的信息冲刷。** 同一层的其他头可能在从别处大力吸收信息（比如层 0 头 1 那个前一位头）。如果没有一个头负责“守住自己”，当前内容会被其他头的输出稀释掉。

② **给后续层留一条干净的通道。** 模型可以学到“某些信息不要在这一层被改动”，而这个“不改动”需要一个具体的执行者。

更一般地说：在多头结构里，**“保持不变”也是一种功能**，而且是一种必须有人来承担的功能。

1.22.10 ⑩ FFN 能否区分顺序

不能。

FFN 是**逐位置独立**作用的：$y_i = \mathrm{FFN}(x_i)$。位置 i 的输出只依赖位置 i 的输入，**不涉及任何跨位置的操作。**

所以如果注意力对输入做的是置换等变操作，那么 FFN 作用之后**这个性质仍然保持。**

结论：整个 Transformer 块都是置换等变的。 不注入位置信息，堆多少层都没用。

这也是为什么原论文必须加位置编码——**它不是锦上添花，是结构性必需品。**

1.22.11 ⑪ RoPE 只依赖相对距离的问题

在纯 RoPE 下，位置 1→2 和 1001→1002 的打分完全一样。 会出两个问题：

① **模型无法表达“绝对位置”相关的模式**（比如“是否在上下文的前 10%”）。

② **长度外推时会失效**——这才是真正的问题。训练时见到的最大距离是 4095；推理时到 8000，就会出现训练中**从未见过**的相对距离。RoPE 的旋转角 $pos \cdot \theta_i$ **线性增长**，位置 8000 的旋转角落在训练时没见过的区域，模型完全没学过那里的行为。

解决办法（NTK-aware 插值、YaRN 等）：把频率 θ_i 按比例缩小，让“位置 8000”落在训练时见过的角度范围内。直观说就是 **把长序列“压缩”到模型熟悉的尺度上**，代价是分辨率损失。第 2 章会展开。

1.22.12 ⑫ 点积打分 vs 加性打分

点积的优势有两层。

① **可以写成矩阵乘法。** 点积打分下，n 个查询对所有 n 个键的打分 就是 $QK^\top$——**一次矩阵乘法**。加性打分涉及逐对的非线性，很难高效地变成一个大矩阵运算。这在 GPU 上是决定性的。

② **没有额外参数。** 加性打分要学 v、W_s、W_h 三组参数；点积打分不需要。既省参数，也减少过拟合风险。

代价是什么？ 正是本章的主线：点积随维度线性增长，维度一高就饱和。而加性打分有 tanh 夹住，输出限制在 $(-1, 1)$，**不会随维度膨胀。**

所以严格说：**加性打分在数值上更稳健，点积打分在计算上更高效。**工程上选了后者，然后用 $\sqrt{d_k}$ 把代价补回来。

⑬ 温度为什么是固定值，不是学出来的

两个角度，指向同一个结论。

① **可学温度没有“该是多少”的信号，而且会往坏的方向走。** 一个可学温度只多 1 个标量参数，参数量完全不是问题。问题在于它的**梯度方向是单向的**：把温度调小（softmax 更尖）几乎总能让训练 loss 更低—— 因为更尖的注意力更容易退化成“记住”，训练集上几乎总是更优。**这个优化压力没有自然的刹车。**所以一个“自由”的温度会一路被推到饱和，然后梯度消失，训练停在那里。

它把一个有解析解的稳定性问题，换成了一个需要靠正则化去约束的自由参数。$\sqrt{d_k}$ 的好处恰恰是它**不需要学**——它在训练第一步就是对的。

② **训练刚开始时，模型没有机会去学它。**即使我们假设存在一个“最优温度”，要学到它就得有梯度。而如果初始温度不对，softmax 已经饱和、梯度已经消失—— **模型卡在一个“要学好温度得先有梯度，要有梯度得先有好温度”的死循环里。**

$\sqrt{d_k}$ 的作用就是**在第一步就打破这个循环。**

1.22.13 ⑭ 可学的位置编码会有什么风险

两个风险，都来自同一个根源：**可学 = 把先验也交给数据去学。**

① **外推能力直接归零。**训练时只见过 $pos \in [0, 511]$。推理时遇到 $pos = 1000$：

- 如果嵌入表只有 512 行，**直接越界报错**
- 如果表开得够大，第 512–1000 行**从来没有被更新过**—— 它们还是初始化时的随机值，和已训好的那 512 行不在一个尺度上，接上去就是噪声

正弦编码没有这个问题：它是一个公式，$pos = 1000$ 照样算得出有意义的向量。

② **需要的样本更多。**正弦编码把“位置”编码成一个**有结构的函数**——相邻位置的正弦值天然接近，“位置 5 和位置 6 相似”这件事是**免费**的。可学的表必须从数据里**自己发现**这个结构。而位置信息在语言任务里的信号是稀疏的（大多数词的含义不依赖精确位置），**用它去换一个先验，不划算。**

这就是为什么本章的小模型用可学嵌入（它只在固定长度上训练和推理），而真实模型用正弦或 RoPE（它们要外推）。

1.22.14 ⑮ 什么样的实现错误会让一整行全被屏蔽

答案是 padding，而且是一个非常“讲得通”的写法导致的。

一个 batch 里长短句混在一起，短句要补齐到最长。假设掩码写成 “**`pad` 位置不可见**”—— 这听起来完全合理：真实 token 不该看到 padding。

但这样会造出全屏蔽的行。考虑一条很短的序列：它自己只有 8 个 token，pad 到 32。那么在**第 8–31 这些位置**上，一个可见的 key 都没有——因为除了它自己之外全是 pad，而自己也是 pad。**整行 $-\infty \to$ `nan`。**

正确的写法是把两件事分开：

要屏蔽什么	在哪一侧	为什么
`pad` 作为 **key**	注意力掩码里屏蔽	真实 token 不该看到补齐位
`pad` 作为 **query**	**不参与 loss**（`ignore_index`）	它本来就没有“下一个词”可预测

把“query 是 pad”也塞进注意力掩码，就会出全屏蔽行。这两件事长得像，处理方式完全不同——这是这个 bug 反复出现的原因。

另一个常见来源更简单：**布尔方向写反**—— `mask` 里 `True` 到底表示“屏蔽”还是“保留”，各处约定不一。

1.22.15 ⑯ 什么时候会造出“很大的负数，但不是 $-\infty$”

来源一：加性掩码。

```
scores = scores + (-1e9) * (1 - mask)     # 而不是 masked_fill(scores, ~mask, -inf)
```

这行代码在 fp32 下工作得很好，所以它能活得很久。**但在低精度下会翻车：**

fp16 下这个 bug 会连着，fp16 下反而不会——这种“换精度就露了”的现象最容易让人误判。

fp16 最大值约 65504，-10^9 溢出成 $-\infty$，**退化成**上面那个 bug。

来源二：使用了“足够负”的常数。

不报错，但悄悄地把 softmax 喂坏。

这才是和本章案例直接相关的一种。

上面那张表里 $d_k = 256$，按编成的模型训练之后，logits 的标准差是 **4348**，而实测的最大绝对值是 **16656**。

1.24 自测答案

1. 要的是**让方差等于 1**（而不是尽可能小）。

$\sqrt{d_k}$ 把方差从 d_k 拉回 1；除以 d_k 会把方差压到 $1/d_k$，$d_k = 64$ 时标准差只有 0.125——logits 全挤在 0 附近，softmax 输出接近均匀，注意力退化成简单平均。

关键在“回到 1”这个目标，不是“变小”。方差为 1 意味着 logits 的典型跨度是 ± 2，softmax 落在它最敏感的区间。

2. 权重**每一行的和不再是 1**。

正确实现里 $\sum_j w_{ij} = 1$（它是概率分布）；写成 `softmax(S) / sqrt(d_k)` 之后，每行的和变成 $1/\sqrt{d_k}$。$d_k = 32$ 时是 **0.1768**（本章实测值）。

检查手段：一行断言。

```
assert abs(weights.sum(-1) - 1).max() < 1e-5, "权重和不是 1"
```

这个 bug 的其他表现：输出的量级整体偏小，下游层会“觉得”注意力没提供什么信息。

3. $d_k = 512/8 = \mathbf{64}$。

如果误写成 512，等于把缩放**过度了 $\sqrt{8} \approx 2.83$ 倍**。后果是 logits 被压得过于集中，softmax 输出偏向均匀—— 本章实测的“缩放过度”症状：**归一化熵 0.9989**（几乎完全均匀）。

这个错误在代码上很难发现（形状完全正确、不报错），只能靠监控权重熵。

4. 变好。本章实测：正确实现 loss 4.810，掩码用 0 的 bug 版本 **4.634**。

原因是 bug 版本让位置 i 能“看到”位置 $j > i$—— **而预测下一个字时，答案就在输入里，所以模型在偷看答案**。

暴露阶段是推理时：生成第 t 个词时第 $t+1$ 个词还不存在，偷看能力瞬间消失。表现是“训练指标漂亮，生成质量莫名其妙地差”。

这是最危险的一类 bug——它让指标变好，会被当成改进保留下来。

5. 因为权重高只说明“信息从这里**取**得多”，不说明“取了之后**用**上了”。

一个头可能取了很多信息，但它的输出投影 W_O 把这些信息投影到了 对后续层无用的方向上——**取而不用的信息对结果没有影响**。

要验证因果性必须做干预（消融）：把那个位置的注意力权重强制置零，看输出/预测有没有变化。**变了才是因果，没变就只是相关**。

（更严谨的做法是“激活修补”：用另一条输入的激活替换这一处，看输出是否跟着变。第 4 章会展开。）

6. 不能。

因为注意力本身是**置换等变的**（§1.13 的小实验），而 FFN 逐位置独立作用，不会破坏这个性质。所以**整个 Transformer 块都看不见顺序**。

没有位置编码时，模型面对的输入是“一堆无顺序的向量”，它**无法区分**“前一位”和“后面任意一位”——“看前一位”这个模式在它眼里根本不存在，自然学不出来。

这也解释了为什么位置编码不是可选配件：它是结构性必需品。

7. 逐项推：

$$\begin{aligned} \mathrm{Var}(q_i k_i) &= \mathbb{E}[q_i^2]\mathbb{E}[k_i^2] - 0 = \sigma^2 \cdot \sigma^2 = \sigma^4 \\ \mathrm{Var}(q^\top k) &= \sum_{i=1}^{d_k} \sigma^4 = d_k\, \sigma^4 \end{aligned} \tag{1.73}$$

标准差是 $\sigma^2\sqrt{d_k}$。

所以 $\sqrt{d_k}$ 是否合适，完全取决于 σ：

- $\sigma = 1$ 时标准差恰为 $\sqrt{d_k}$，除以 $\sqrt{d_k}$ 得到方差 1 √
- $\sigma \neq 1$ 时，缩放后标准差是 σ^2，**不是 1**

这正是本章案例的核心：真实模型里 σ 由初始化决定（约 0.02），所以缩放后 logits 的标准差远小于 1——**补丁打了，但没打到点子上**。$\sigma < 1$ 时缩放**过度**，$\sigma > 1$ 时缩放**不足**。

8. 逐项求和：

$$\sum_j \frac{\partial p_i}{\partial z_j} = \sum_j p_i(\delta_{ij} - p_j) = p_i\left(\sum_j \delta_{ij} - \sum_j p_j\right) = p_i(1-1) = 0 \tag{1.74}$$

（用到了 $\sum_j \delta_{ij} = 1$ 和 $\sum_j p_j = 1$。）

它说明的：softmax 的输出**恒满足“和为 1”这个约束**，所以输入的任何“整体平移”都不会改变输出—— 这就是**平移不变性**在雅可比层面的表现。

一个推论：梯度 $\partial \mathcal{L}/\partial z$ 里“所有分量同时增加”的那个方向是**无效方向**—— 它对应的损失变化恒为零。这也解释了为什么在 小节 1.16 里，dS 要减去“加权平均”那一项：**softmax 的反向传播自动把无效方向投影掉了**。

9. 按 式 (1.62)：

$$2048^2 \times 16 \times 24 \times 2 \text{ 字节} = 3.22 \text{ GB} \tag{1.75}$$

（单头单层 8.39 MB → 单层 16 头 134 MB → 24 层 3.22 GB。）

这个数字意味着：如果用 fp32，就是 6.44 GB—— **还没算模型权重、激活值、优化器状态，光注意力矩阵就吃掉大半张 24G 卡**。

而且注意 $n = 2048$ 在今天是**很短**的序列。如果 $n = 8192$，这个数字会变成 **51.5 GB**——单卡放不下。

这正是 FlashAttention 存在的理由：它让这部分显存**根本不需要分配**。

10. 掩码构造：位置 i 能看 j 当且仅当 $i - w < j \le i$。

```
def sliding_window_mask(n, w):
    idx = torch.arange(n)
    # 下三角（因果）且 距离不超过 w
    causal = idx[None, :] <= idx[:, None]
    within = (idx[:, None] - idx[None, :]) <= w
    return causal & within
```

省下了什么：

- **计算量**：从 $O(n^2 d)$ 降到 $O(nwd)$。$n = 8192$、$w = 256$ 时，理论上是 **32 倍**的减少。
- **显存**：不再需要完整的 $n \times n$ 矩阵（如果实现得当）。

没省下什么：

- **长程依赖**。这是**能力上的代价**，不是性能问题。位置 i 永远看不到 $i - w$ 之前的内容—— 如果答案在最开头（比如“这段话的主题是什么”），这个模型无能无力。
- **实现复杂度**。块状掩码的 kernel 比标准因果掩码难写得多；用现成库时往往还得自己写 Triton kernel 才能真正吃到加速。

所以滑动窗口是一个“用能力换效率”的设计。它适合“局部信息足够”的任务（比如语音、某些视觉任务），对需要全局视野的任务则要配合少量“全局 token”（这就是 Longformer 的做法）。

1.25 关键概念表

中文	英文	一句话定义	在本章哪一节
注意力	attention	按相似度分配权重、对各位置内容加权平均的机制	§1.5
自注意力	self-attention	Q、K、V 全部来自同一个序列的注意力	§1.5
交叉注意力	cross-attention	Q 来自一个序列、K/V 来自另一个	§1.5
软查表	soft lookup	输出是加权混合而非单一选择	§1.5
查询 / 键 / 值	query / key / value	分别表示“要找什么”“有什么”“内容是什么”	§1.5

中文	英文	一句话定义	在本章哪一节
缩放点积	scaled dot-product	$\mathrm{softmax}(QK^\top/\sqrt{d_k})V$	§1.6
温度	temperature	softmax 前除以的常数，控制分布锐度	§1.7
饱和	saturation	softmax 输出接近 one-hot，雅可比趋近零	§1.7
雅可比矩阵	Jacobian	输出对输入的偏导矩阵，衡量敏感度	§1.7
多头注意力	multi-head attention	把 d_{model} 切成 h 份并行做注意力	§1.8
前一位头	previous-token head	注意力集中在紧邻左侧位置的头	§1.8
分组查询注意力	grouped-query attention (GQA)	多个头共享一组 K、V，压缩 KV 缓存	§1.9
多查询注意力	multi-query attention (MQA)	所有头共享一组 K、V	§1.9
KV 缓存	KV cache	推理时缓存历史 K、V 避免重算	§1.9
因果掩码	causal mask	屏蔽未来位置，保证自回归性	§1.6
置换等变	permutation equivariant	输入换序，输出只是跟着换序	§1.13
位置编码	positional encoding	把位置信息注入模型的手段	§1.13
旋转位置编码	rotary position embedding (RoPE)	用旋转矩阵把相对位置编进内积	§1.13
上下文窗口	context window	模型一次能处理的最大序列长度	§1.13
在线 softmax	online softmax	分块计算 softmax，维护运行最大值与和	§1.10
FlashAttention	FlashAttention	分块计算、不写完整 $n\times n$ 矩阵的精确注意力	§1.10
序列打包	sequence packing	多条短序列首尾相接共用窗口，提高利用率	§1.16
消融	ablation	关掉某个组件看影响，用于判断因果性	§1.14
注意力汇	attention sink	大量注意力集中到某个固定位置（常是首个 token）	§1.12

1.26 延伸阅读

按“读它能得到什么”来排序，不是按重要性。

1.26.1 一、本章的源头

Vaswani et al., Attention Is All You Need (NeurIPS 2017).
读它能得到什么：本章所有内容的原始出处。公式、多头结构、位置编码、残差与 LayerNorm 的排布，全在这一篇里。
读的时候注意四件事：

1. $\sqrt{d_k}$ **只是一个脚注**（第 3 节末尾）。整篇论文没有对它做实验验证，也没有讨论它在什么规模下才重要。本章的案例正是补上了这一块。
2. **多头被描述为“jointly attend to information from different representation subspaces”**——这个说法**比论文给出的证据要强**。论文并没有验证“每个头学一个子空间”，那是一个直觉性的解释。直到 Michel et al. (2019) 和后续的电路分析工作，人们才开始认真检验这个说法。
3. **位置编码用的是正弦函数**，那是 2017 年的选择。今天的主流是 RoPE，见第 2 章。
4. **训练成本那一节值得细读**：他们报告了 8 卡 P100 训练 3.5 天的数据。对比今天的规模，这个数字本身就能让你感受到七年里发生了什么。

1.26.2 二、从注意力里读出机制

Elhage et al., A Mathematical Framework for Transformer Circuits (Anthropic, 2021).
读它能得到什么：一套描述“注意力头在做什么”的**形式语言**。它把每个头拆成两个独立的电路：

- **QK 电路**（W_Q、W_K）：决定“这个头看哪里”
- **OV 电路**（W_V、W_O）：决定“看到之后写什么”

为什么这个拆分有用：本章里我们只能说“这个头 45% 看前一位”，那是**行为描述**。有了 QK/OV 拆分，你可以问更精确的问题——“这个头从位置 j 读出了什么特征、写到了位置 i 的哪个子空间”。
读之前的准备：需要熟悉矩阵乘法、秩、以及基本的线性代数直觉。文中大量使用“低秩投影”的视角，第一次读可能会觉得抽象，建议配合第 4 章一起读。

1.26.3 三、多头真的都有用吗

Michel et al., Are Sixteen Heads Really Better than One? (NeurIPS 2019).
读它能得到什么：一个**实验结论**——训练完之后大部分头可以剪掉，性能几乎不掉；但少数几个关键头剪掉就崩。
它纠正了什么：“每个头都有独特功能”这个天真想法。如果 80% 的头可以随便移除，那“这个头在看什么”就不能直接当成“模型这样决策的原因”。
与本章的关系：§1.8 里我们量出了 8 个头的不同模式——那证明**它们行为不同**，但**不证明**它们都不可或缺。两件事要分开看。这个区分是第 4 章的主题。

1.26.4 四、长序列的工程解法

Dao et al., FlashAttention: Fast and Memory-Efficient Exact Attention (NeurIPS 2022).
读它能得到什么：$O(n^2)$ 在实际系统里到底该怎么处理。核心是分块 + 在线 softmax（§1.10 手算过），**数学结果完全不变**，改变的只是访存模式。
为什么值得读：它示范了一个重要的思路——**当算法复杂度已经最优时，还可以从“数据怎么流动”上再要一个数量级的提升**。今天几乎所有训练框架都内置了它。
读的时候注意：论文里的 CUDA 细节可以跳过，重点看第 3 节的算法伪代码和第 4 节的复杂度分析。

1.26.5 五、本章案例的实践含义

Henry et al., Query-Key Normalization for Transformers (2020).

读它能得到什么：$\sqrt{d_k}$ 的一个现代替代方案—— **直接对 q、k 做归一化**，而不是依赖缩放这个间接手段。

为什么它比 $\sqrt{d_k}$ 更彻底：本章案例说明，$\sqrt{d_k}$ 的有效性依赖“q、k 各分量方差为 1”这个前提，而这个前提**在训练过程中会失效**（模型会主动放大尺度）。QK-norm 不依赖任何关于尺度的假设——它把尺度**钉死**。

延伸：这个思路在 ViT-22B、Stable Diffusion 3 等模型里被采用。读完可以对照本章案例想一想：**如果一开始就用 QK-norm，那个“没复现出来的实验”会是什么结果？**

1.26.6 附：读论文时的一个习惯

上面每条都标了“读的时候注意什么”。这个习惯值得单独说一下。

技术论文的写法是**论证式**的：作者提出一个方案，然后给出支持它的证据。但论文很少主动告诉你：

- 这个方案的**前提**是什么（往往隐含在推导里）
- 它**在什么规模下**才重要（小规模实验往往不体现）
- 它和**别的方案**相比，代价在哪（对比实验通常是精心挑选的）
- 哪些结论是**外推**的（作者自己也没在大规模上验证过）

本章那个“没复现出来的实验”，本质上就是把这四条问了一遍。

具体做法很简单：读完一个技术点，问自己三句话——**“它在什么条件下成立？”“我怎么验证它在这里成立？”“如果不成立会怎样？”**

这三句话不需要任何额外知识，但能让你从“读过”变成“知道”。

1.26.7 六、位置编码的演进

Su et al., RoFormer: Enhanced Transformer with Rotary Position Embedding (2021).

读它能得到什么：RoPE 的原始论文。§1.13 推的核心恒等式（$\tilde{q}_m^\top \tilde{k}_n = q_m^\top R_{n-m} k_n$）就出自这里。

读的时候注意：论文第 3.2 节的推导主干很短，但值得自己推一遍。另外注意它讨论的**远程衰减**性质—— 这解释了为什么 RoPE 在长文本上表现好。

配套读：如果关心长度外推，可以接着看 NTK-aware 插值和 YaRN 这两条后续工作线，它们解决的是 §1.13 检查点 ⑪ 提到的那个问题。

1.27 本章回顾：一条完整的线索

这一章讲了八节，但它们不是八个独立的知识点。**它们是一条链**，起点是一个实验意外，终点是对“工程补丁”这件事的一般认识。

1.27.1 主线：一个补丁的完整生命周期

```
① 点积打分好用（一次矩阵乘法、零额外参数）
        ↓  但有个毛病
② 点积的方差随维度线性增长（Var = d_k）—— 这是数学事实
        ↓  因为
③ softmax 的雅可比 J_ij = p_i(δ_ij − p_j)，饱和时趋近零矩阵
        ↓  所以
④ 打补丁：除以 √d_k，把方差拉回 1
        ↓  但补丁有前提
⑤ 前提是"q、k 各分量方差为 1"—— 现代架构里不成立
   （初始化尺度 0.02、LayerNorm 归一化的是输入而非投影后的 q、k）
        ↓  所以
⑥ 小模型上补丁显得多余（实测差 0.17），大模型上补丁又不够
   （D_head = 256 时 logits 标准差 5477、熵 0.0004）
        ↓  于是
⑦ 有了新补丁：QK-norm（直接把 q、k 的尺度钉死）
```

这条链上的每一环，都是这一章的一节。记住链条比记住结论有用—— 因为结论（“要写 $\sqrt{d_k}$”）会随着架构演进而过时， 链条（“为什么会有这个补丁、它的前提是什么”）不会。

1.27.2 六条可以带走的方法

这一章除了注意力本身，还示范了六种**可迁移的方法**：

方法	在本章的体现
对照实验要钉死其余变量	同种子、同语料、同步数，只改 `scaled` 一个开关
理论没被证实时，先怀疑前提	脚注推导没错，是“方差为 1”这个前提不成立
找不到现象时，问“它在什么尺度下才显现”	d_{head} 从 32 推到 256 才看见灾难
让指标变好的 bug 最危险	掩码 bug 让 loss 从 4.810 降到 4.634（它在偷看答案）
关键性质要靠断言，不能靠 loss	未来位置权重必须严格为 0——这要写单测
诊断量要问“在什么条件下有意义”	熵必须按可见位置数归一化，否则不同长度没法比

1.27.3 如果只记三件事

第一，注意力是软查表。 输出是加权平均，不是选择。这个直觉能帮你理解后面所有的变体（稀疏、线性、FlashAttention）都只是“怎么算这个加权平均”的不同方式。

第二，$\sqrt{d_k}$ 修正的是方差，不是大小。它的目的是让梯度**不随维度变化**，而不是让梯度变大。理解这一点，就能明白为什么除以 d_k 是错的。

第三，机制的必要性取决于前提。这一章最大的收获不是“$\sqrt{d_k}$ 很重要”，而是“**任何机制的必要性都依赖于它所在系统的具体配置**”。小模型上多余的补丁，大模型上可能是救命的；反过来也一样。

带着这第三条去看后面的章节——每讲一个技术，都问一句“它解决什么问题、前提是什么、前提不成立时会怎样”。这个习惯养成之后，你读论文的速度会变慢，但**读懂的比例会大幅提高**。

这一章我们花了大量篇幅在一个“没复现出来”的实验上，而不是直接给出结论——**全章有一半的篇幅在讲“怎么知道”，而不是“是什么”**。这是刻意的：**研究生阶段要训练的不是“知道答案”，而是“知道答案的前提、以及怎么亲手验证它”**。

一个公式可以背下来，但“怎么判断这个公式在你的系统里成不成立”只能靠一遍遍做对照实验练出来。本章的每一个实验设计细节——固定种子、只改一个变量、测中间量、做消融——都是这个训练的一部分。**它们比 $\sqrt{d_k}$ 这个结论本身更值得带走。**

1.27.4 与后续章节的接口

本章留下的问题	在哪一章解决
RoPE 的实现细节与长度外推	第 2 章
LayerNorm / RMSNorm 为什么这么设计	第 2 章
FFN 在做什么（键值记忆视角）	第 2 章
KV 缓存与采样策略	第 3 章
归纳头、电路分析、因果干预	第 4 章
注意力的权重能不能当解释	第 4 章

最后一句提醒：本章的所有代码、数据、图表都可以在项目里找到并复现。如果你在复现时得到了不同的数字，**那多半不是错误，而是你的环境与我们的不同**——PyTorch 版本、硬件、甚至随机数生成器的实现差异都会带来浮动。重要的是**趋势和量级**，不是小数点后第三位。

这一章真正要你记住的

不是“要写 $\sqrt{d_k}$”。

而是：**一个机制的必要性，取决于它的前提在你的系统里成不成立**。脚注的推导没错，但它的前提（各分量方差为 1）在现代架构里不成立，所以现象在小模型上消失、在大模型上重新出现。

判断一个机制是否关键，要找到**它在哪个区间才显现**—— 这就是本章那个“没复现出来的实验”最终教给我们的东西。

2. 一个 Transformer Block 的解剖

> **本章地图**
> **你在这里**：数据流的第 3 步 —— 把那层 Block 整个打开。**本章解决**：残差、LayerNorm、FFN 各自在解决什么问题；为什么归一化放在残差**前**还是**后**，会让 12 层的模型不如 4 层。**前置**：第 1 章（多头注意力可以先当黑盒）。**读完能做**：拿到一份别人的 Block 实现，逐行说出每个部件的职责，以及它最容易写错的那一处。

一个只会“猜字频”的模型。

第 1 章我们训了一个 4 层的小模型，1500 步后 loss 降到 4.0 左右，生成的文本虽然不通顺，但能看出唐诗的腔调。现在把层数加到 12 层—— **按理说更深的模型应该更强**。

结果 loss 停在 **5.36**，怎么训都不动。

这个数字需要一个参照系才能理解。语料里 1200 个字的**边缘分布**（也就是“完全不看上下文，只按字出现的频率猜”）给出的交叉熵是 **5.67**。也就是说，**这个 12 层模型只比“闭着眼睛按字频猜”好了 0.31**—— 它基本上什么都没学到。

但模型本身没坏：没有 nan、梯度范数正常、参数量是 4 层版的 2.8 倍。**它只是不学**。

这一章从这个现象展开。我们最后找到的原因，不是某一行的 bug，而是**一个被放错位置的归一化层**——以及围绕它的、关于“深度学习为什么需要这些零件”的一整套设计逻辑。

读完本章你应该能回答：

1. 残差连接、LayerNorm、FFN——这三样东西各自在解决什么问题？把它们任何一个拿掉会怎样？
2. 为什么 LayerNorm 放在残差**里面**还是**外面**，会让同一个模型从“完全学不会”变成“正常训练”？
3. RoPE 为什么能外推而可学习位置嵌入不能？NTK 插值又做了什么？

> **i** 本章的代码与数据

```
# 需要 PyTorch。本机在 /home/mpp/micromamba/envs/dlspace/bin/python
python notebooks/run_ch2_case.py --steps 1200     # 案例的全部对照实验
python notebooks/rope_extrapolation.py            # 位置编码外推对照
```

CPU 上约 10 分钟，GPU 上 2–3 分钟。文中每一个数字都来自这两次真实运行，原始数据在 `data/ch2_case.json` 与 `data/rope_extrapolation.json`。

2.1 现象：12 层不如 4 层

先把问题摆清楚。

实验设置：与第 1 章同一个语料（全唐诗 4003 首、25.5 万字）、同一个字符级分词器（字表 1200，覆盖 87.6%）。模型是标准的 Pre-LN Transformer：每层“注意力 + 前馈网络”，两个子层各带一个 LayerNorm 和一条残差连接。

唯一的变量是**层数**。表 2.1 是结果：

表 2.1　修复配置后，不同层数的训练结果

层数	loss 起点	loss 终点	末梯度范数
6	7.103	**4.213**	0.496
12	7.145	**4.245**	0.429
24	7.122	**4.370**	0.332

等一下——这张表里 12 层和 24 层都训得很好（4.2–4.4），并没有“停在 5.36”。**因为这张表用的是修复之后的配置**。

把时间倒回去。我们最初跑的那一版是这样的：

层数	loss 起点	loss 终点
6	7.068	**5.356**
12	7.067	**5.357**
24	7.041	**5.357**

三个深度给出几乎一模一样的数字。这个“一模一样”本身就是最强的线索——如果一个模型真的在学东西，不同容量的模型不可能停在同一个 loss 上。

为什么“数字太整齐”是个危险信号

真实的训练充满随机性：初始化不同、数据顺序不同、梯度噪声不同。两个**不同**的模型如果给出**逐位接近**的结果，通常说明它们都掉进了同一个“与容量无关”的退化解。

在我们的例子里，这个退化解是：**输出一个与输入无关的固定分布**——大约就是字频分布。模型的 170 万个参数全都白费了，因为它找到了一个“不需要看输入”的答案。

遇到这种情况，先别调超参数。先问：“它是不是在输出一个常数?”验证方法很简单：算一下你说的那个 loss，和“只猜边缘分布”的 loss 差多少。

2.2 第一次排查：换个位置就好了

和第 1 章一样的思路——**先找那个不同的地方**。

我们对比了两种层内的排布。区别只在于 LayerNorm 放在哪：

```
Pre-LN （前置）：   x = x + Attention(LN(x))
                    x = x + FFN(LN(x))

Post-LN（后置）：   x = LN(x + Attention(x))
                    x = LN(x + FFN(x))
```

归一化层本身一模一样，只是位置换了一下。

结果：

层数	Pre-LN	Post-LN	差距
6	4.213	5.356	1.14
12	4.245	5.357	1.11
24	4.370	5.357	0.99

换个位置，问题就消失了。Post-LN 在三个深度上都停在 5.36 附近，而 Pre-LN 一路降到 4.2–4.4。

到这一步，一个自然的结论是“Post-LN 有问题，别用它”。但这不是本章想要的答案——**Post-LN 是 2017 年原论文用的排布**，它支撑了 BERT、GPT-1、GPT-2 这些模型。说它“有问题”显然不对。

真正的问题在于：**这个对比不公平**。

> ℹ 停下来想一想 ①
>
> 上面的对比里，两种排布用了**完全相同的超参数**：学习率 3e-3、warmup 100 步、AdamW、批大小 32。
>
> 请想：**为什么“超参数相同”反而可能是不公平的?**
>
> 提示：如果两个方案对超参数的**敏感程度**不同，那么“用同一套参数”就意味着“其中一个没被调到最好”。你要证明的到底是“A 比 B 好”，还是“A 在 B 的舒适区里比 B 好”?

2.3 排查过程：我们试了什么

上一节说“换个位置就好了”。但在找到这个答案之前，我们走了三条弯路。**把它们写出来，因为弯路比答案更有信息量**。

2.3.1 弯路一：怀疑是数据问题

第一个假设很自然：12 层模型参数量是 4 层的 2.8 倍，但语料只有 25 万字——**是不是过拟合了？**

验证：把语料扩到 4 倍（《全唐诗》全部 5 万多首）。结果 loss 从 5.357 降到 5.351——**几乎没有变化**。

排除。如果是过拟合，更多数据应该有明显改善。

2.3.2 弯路二：怀疑是学习率

第二个假设：12 层比 4 层深，也许需要更小的学习率。

我们确实扫了（这一条不是弯路，后面成了关键的转折），但在**只扫 Pre-LN** 的时候，结论是“学习率对结果影响不大”——因为 Pre-LN 在 10^{-4} 到 3×10^{-3} 之间都能工作。

当时的错误推理：既然 Pre-LN 对学习率不敏感，那 Post-LN 应该也不敏感，问题不在学习率。

这个推理漏掉了一件事：敏感性是**方案自身的属性**，不能用方案 A 的性质去推断方案 B。

2.3.3 弯路三：怀疑是初始化

第三个假设：也许 Post-LN 需要不同的初始化。

这个方向**部分正确**——残差缩放初始化（$1/\sqrt{2L}$）确实对深层模型重要。但我们把它加在 Pre-LN 上，Post-LN 上没加。

于是实验变成了“比较排布 + 比较初始化”两个变量。这是我们自己犯的错，也是 §2.4 讲的“一次只改一个变量”的反面教材。

修正后（两边都不加缩放初始化），差异依然存在，说明排布本身确实有影响。

> ⚠ 三条弯路的共同点
>
> **它们都在“方案之外”找原因**：数据、超参数、初始化。这些确实是常见病因，但对本章的问题来说，答案就在方案本身。
>
> **什么时候该怀疑方案本身**：当你已经排除了两三个外部因素，而现象仍然稳定存在时。**不要无限地检查环境**——那是一种拖延，而不是排查。
>
> **一个实用的判据**：如果你的现象在**三个不同的随机种子**下都复现，且**换了数据仍然存在**，那就该把注意力收回到方案上了。

2.4 公平一点：各自调到最好

要做一个站得住的对比，必须**给两边各自调参的机会**。

我们对两种排布各扫了四个学习率：10^{-4}、3×10^{-4}、10^{-3}、3×10^{-3}。其余全部不动（12 层、900 步、同种子）。

学习率	Pre-LN	Post-LN
10^{-4}	4.897	5.164
3×10^{-4}	4.703	**4.732**
10^{-3}	4.549	4.830
3×10^{-3}	**4.408**	**5.417**

这张表改变了整个结论。

同一个学习率下，两者几乎持平。在 3×10^{-4} 处，Pre-LN 是 4.703、Post-LN 是 4.732——差 0.03，基本可以认为一样。

差别在“趋势”上： - Pre-LN 随学习率提高**单调改善**（4.897 → 4.703 → 4.549 → 4.408） - Post-LN 有个**很窄的窗口**：先改善（5.164 → 4.732），然后急剧恶化（→5.417）

所以 Pre-LN 买到的不是“更好的解”，而是“更宽的可调范围”。

准确的说法是：**Post-LN 对学习率敏感得多**。在最佳点附近它和 Pre-LN 一样好，但那个最佳点的容错空间很小； 而 Pre-LN 在很宽的学习率区间里都能正常工作，还能从更高的学习率中获益。

> 这个纠正很重要
>
> 我们最初写下的结论是“Post-LN 训不动”。**那是错的**—— 它只是在我们随手选的那个学习率下训不动。
>
> 这一类错误在研究中非常常见：**用一组未经调优的超参数比较两个方案，然后宣称其中一个更好**。 这叫做“用别人的舒适区打别人”。
>
> **正确的做法是给两边同等的调参预算**，然后比较“各自最优”或者“对超参数的鲁棒性”——这两个是不同的指标，不要混为一谈。
>
> 在本章的例子里： - 比“各自最优”：Pre-LN 4.408 vs Post-LN 4.732，**Pre-LN 略优** - 比“鲁棒性”：Pre-LN 单调改善，Post-LN 有窄峰，**Pre-LN 明显更稳**
>
> **两个结论都成立，但它们说的不是同一件事**。

2.5 转折点：那句“太整齐了”

回头看，整个排查的转折来自一个很小的观察。

我们当时把四组训练的最终 loss 并列在表里：

	Pre-LN	Post-LN
6 层	4.213	5.356
12 层	4.245	**5.357**
24 层	4.370	**5.357**

Post-LN 那一列的 5.356 和 5.357——差 0.001。

第一反应是“跑错了吧”。我们检查了三次日志，确认没有跑错。 然后做了第二件事：**把这三组的结果和“只猜字频”的基线放在一起看**。

- 字频基线：5.672
- Post-LN 三组：5.356 / 5.357 / 5.357
- Pre-LN 三组：4.213 / 4.245 / 4.370

Post-LN 只比“完全不看输入”好了 0.31，而且不管多少层都一样。

至此，“这是不是一个训不动的模型”变成了 **“这是不是一个根本没在用输入的模型”**。

问题的性质变了，排查方向也就变了——从“调超参数”转向“看模型到底在算什么”。

这个转折里有两个可复用的动作。

① **把数字和基线放在一起**。

“5.357”这个数字本身没有意义。**和 5.672（字频）比**， 才知道它几乎什么都没学到；**和 7.09（均匀）比**， 才知道它至少学了一点。

任何指标都要有参照系。我们第一次看这个结果时， 只觉得“Post-LN 差 1.1”，没有意识到“差 1.1 之后剩下的那部分几乎为零”。

② **注意到“不同配置给出相同结果”**。

6 层和 24 层的参数量差 4 倍。**它们不该给出同一个 loss**。 这个“不该相同却相同”的观察，比 loss 的绝对值更有信息量。

一般的规律：当一组实验的结果“太整齐”时， 通常是模型退化到某个与配置无关的解——**先怀疑这个，再怀疑代码**。

2.5.1 事后的复盘

现在回头看，我们本可以更快地走到这一步。

如果有“相邻位置输出的 KL 散度”这个指标（§2.15 路径二给过代码）， 我们在第一次训练时就会发现它接近 0—— 而不是等到对比三个深度之后。

这正是本章反复强调“要监控中间量”的原因：loss 是一个**经过平均、滞后、且解释不唯一**的指标。它告诉你“结果不好”，但不告诉你“哪里不好”。

一个可以立即用上的清单——下次训练时加上这三个：

指标	健康值	它回答什么
相邻位置输出的 KL 散度	> 0.05	模型有没有在用上下文
逐层激活 std 的末/首比	< 4	残差流有没有失控累积
注意力权重熵	0.5–0.9	缩放是否合适

三个数的代价几乎为零（每 100 步算一次），但它们能在 loss 显现之前给出信号。

2.6 为什么位置这么关键

现象讲完了，但机制还没碰。**为什么把 LayerNorm 从“外面”挪到“里面”，就能让同一个模型从“学不会”变成“正常训练”?**

要回答这个问题，需要看清楚两件事：

1. **残差连接在做什么**——它是一条“高速公路”，让梯度可以不走那些复杂的子层直接传回去。LayerNorm 的位置决定了这条高速公路上有没有收费站。
2. **LayerNorm 在归一化什么**——它不只是“把数据标准化”，它在**每一步都重新设定信号的尺度**，这个动作对梯度的传播有巨大的影响。

图 2.1 是初始化状态下，两种排布在残差流上的实测画像：

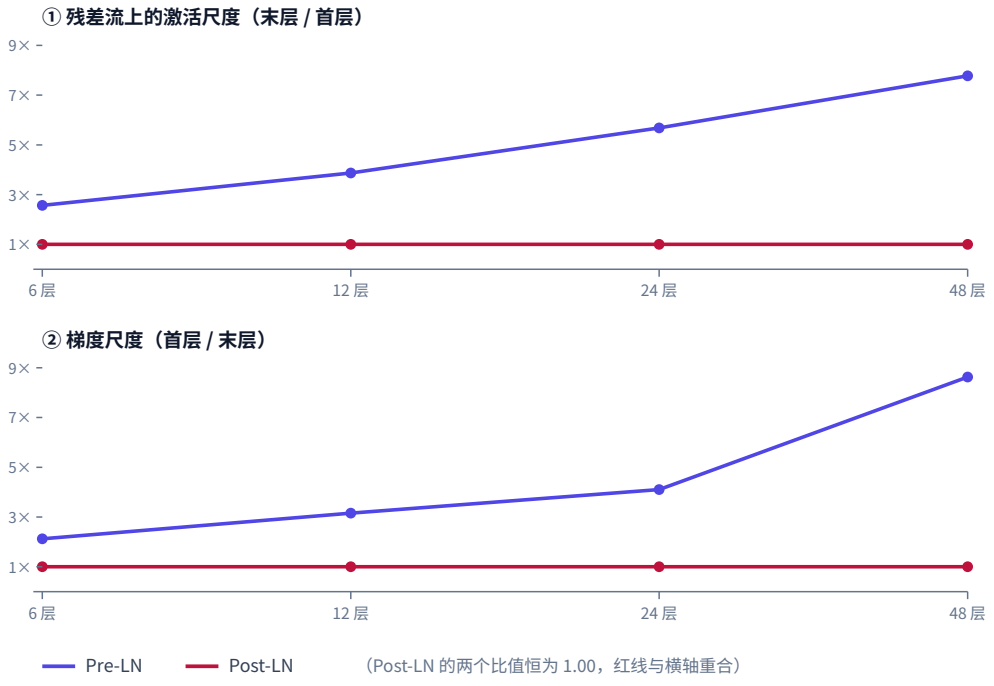


图 2.1　Pre-LN 与 Post-LN 在初始化状态下的残差流画像。上：每层输出的激活尺度。下：该处梯度尺度。Pre-LN 的激活随深度累积（24 层时增到 5.7 倍），Post-LN 被每层的 LayerNorm 钉死在 1.0。但 Post-LN 的梯度绝对值小约 15 倍。

深度	排布	激活 末/首	梯度 首/末	梯度首（绝对值）
6	Pre-LN	2.57	2.12	1.65×10^{-4}

深度	排布	激活 末/首	梯度 首/末	梯度首（绝对值）
6	Post-LN	1.00	1.00	9.68×10^{-6}
12	Pre-LN	3.87	3.16	1.60×10^{-4}
12	Post-LN	1.00	1.00	9.73×10^{-6}
24	Pre-LN	5.68	4.10	1.41×10^{-4}
24	Post-LN	1.00	1.00	9.49×10^{-6}
48	Pre-LN	7.78	8.63	2.21×10^{-4}
48	Post-LN	1.00	1.00	9.70×10^{-6}

读这张表要看三件事：

① **Post-LN 的“1.00”是 LayerNorm 的指纹**。每一层结束都做一次归一化，所以激活尺度被钉死在 1.0——无论多少层、无论什么输入。从“控制数值范围”的角度看，这是好事。

② **但 Post-LN 的梯度绝对值小了约 15 倍**（9.7×10^{-6} vs 1.6×10^{-4}）。而且这个倍数**不随深度变化**——6 层、48 层都是这个量级。

③ **Pre-LN 的激活随深度增长**（24 层增到 5.7 倍，48 层 7.8 倍），梯度沿深度有 2–8 倍的不平衡。这是 Pre-LN 的已知代价（残差流不断累积，没有归一化来抑制）。

两边的“问题”不一样：Post-LN 的问题是**梯度整体太小**，Pre-LN 的问题是**梯度沿深度不均匀**。

接下来的几节，我们把残差连接和 LayerNorm 各自拆开，搞清楚这两个问题分别是怎么来的、以及为什么深度学习最终选了 Pre-LN。

2.7 残差连接：那条不能省的高速公路

先把残差连接单独拿出来做消融。做法很简单：把 `x = x + f(x)` 改成 `x = f(x)`，其余全部不动。

层数	有残差	无残差	差距
2	**4.489**	**5.392**	0.90
4	**4.520**	**5.392**	0.87
12	**4.600**	见下文	—

注意无残差那一列：2 层和 4 层又是同一个数字。和第 1 节那个“12 层停在 5.357”是同一个信号——模型退化成了“不看输入，输出固定分布”。

这个结果比想象中更极端。

通常我们说“残差连接让**深**网络能训起来”，暗示浅层不需要它。但实测是：**连 2 层都训不动**。

2 层 Transformer 只有两个注意力 + 两个 FFN，参数量不到 40 万。直觉上它和“深”完全不沾边。可一旦去掉残差，它就废了。

为什么？答案在梯度上。没有残差时，梯度必须**依次穿过每一层的所有非线性变换**才能到达前面的层。2 层听起来不多，但每一层里都有 LayerNorm、softmax、GELU—— 它们都会对梯度做一次缩放。几次下来，信号就衰减到无法有效更新参数了。

残差连接提供了一条“绕过所有变换”的直通路径。梯度可以走这条路直接回到任意一层，路径长度从 $O(L)$ 变成 $O(1)$。

2.7.1 为什么是“加”而不是“拼接”

设计残差连接时有个自然的替代方案：**把子层的输出拼接到输入上**，而不是相加。

$$\text{相加：} x_{l+1} = x_l + f(x_l), \qquad \text{拼接：} x_{l+1} = [x_l\,;\, f(x_l)] \tag{2.1}$$

拼接保留了更多信息（不会把两个向量混在一起），为什么大家还是用相加？

因为维度会爆炸。每拼接一次，维度增加一倍。L 层之后是 $2^L d$ 维—— 12 层就是 4096 倍。

相加的好处是维度不变，所以可以无限堆叠。代价是“信息被混合了”—— $f(x_l)$ 的输出和 x_l 加在一起之后，你没法再从 x_{l+1} 里分离出原始的 x_l。

> 但“混合”这件事没那么糟
>
> 有一个漂亮的视角：**残差流（residual stream）可以被看成一个共享的通信通道**。
> $-x_l$ 是所有层共同读写的“总线”。每一层从总线上读取信息（通过 LayerNorm 和投影），算出一点新的东西，再**加回**总线上。后续的层可以读取总线上的任何位置—— 包括前面所有层写进去的内容。
> **这个视角解释了两件事**：
>
> 1. **为什么加法够用**：因为各层写入的是**不同方向**的向量。在一个 128 维的空间里，12 层各写一个方向，只要它们近似正交就不会互相干扰。 模型在训练中确实倾向于学到近似正交的写入方向。
> 2. **为什么“看自己”的头有用**（第 1 章 §1.8 检查点 ⑨）：在总线模型里，一个“把输入原样传下去”的头就是“不往总线上写东西”。 它确保了总线上的已有内容不被这一层覆盖。
>
> **“残差流即通信总线”是理解 Transformer 最重要的直觉之一**。 第 4 章讲电路分析时，整章都建立在这个视角上。

2.7.2 一个类比：残差像版本控制

残差流的加法结构，和一个看似不相关的领域有个漂亮的对应：**版本控制**。

- 每层的写入 = 一次 **commit**（新增一点内容）
- 残差流 = **当前工作区**（包含所有历史 commit 的累积效果）
- 没有任何一层能“删除”历史（只能提交一个抵消性的改动）

这个类比的三个推论：
① **读的时候看到的是“全部历史的叠加”**，而不是某一次提交。 这就是为什么残差流上的每一维没有干净的含义——它承载着所有层的累积写入。
② **越晚的层“看到”的历史越多**。 第 40 层读到的是前 39 层的总和， 第 2 层只有第 1 层。这解释了为什么深层能处理更抽象的东西： 它站在更多历史的肩膀上。
③ **“回滚”很难**。 就像版本控制里撤销一个中间提交很麻烦， 模型也几乎无法“忘掉”某一层写入的内容（除非后续层学会精确抵消）。
这个类比的价值：它让“残差流是共享总线”这个抽象说法变得具体。第 4 章讲电路分析时，整章都在用“读—写”这个框架。

2.7.3 残差带来的新问题

残差连接解决了梯度路径，但引入了一个新麻烦：**方差累积**。
每经过一层，x 就加上一个新的 $f(x)$。如果每个 $f(x)$ 的方差是 σ^2， 而且它们相互独立，那么 L 层之后：

$$\mathrm{Var}(x_L) = \mathrm{Var}(x_0) + L\sigma^2 \tag{2.2}$$

方差随层数线性增长。 这正好解释了 图 2.1 里 Pre-LN 那一列： 24 层时激活尺度是首层的 5.68 倍，48 层是 7.78 倍。
（注意是“线性增长方差”，所以标准差按 $\sqrt{L}$ 增长。 $L = 24$ 时 $\sqrt{24} \approx 4.9$，和实测的 5.68 同量级。）
这个累积会带来两个后果：

1. **数值范围失控**。48 层时激活标准差是首层的 8 倍—— 如果输入量级本来就大，很快就会碰到浮点上限。
2. **后续层的输入量级不一致**。第 3 层看到的输入是第 1 层看到的 2 倍， 第 40 层看到的是 6 倍。**同一套权重要在不同的输入量级下工作**，这很难学。

解决方案有两个，分别对应 Pre-LN 和 Post-LN 的动机：

- **Post-LN 的做法**：在残差**之后**归一化，强制每层的输出尺度都是 1
- **Pre-LN 的做法**：接受累积，但在**送进子层之前**归一化， 并配合一个缩放初始化（见下文）让累积变慢

这就是两者最本质的差别。下一节先把 LayerNorm 讲清楚，再回来对比。

2.8 LayerNorm：它在归一化什么

2.8.1 先算一遍

LayerNorm 的定义是：对**每个位置**的**特征维**做标准化，然后学一个缩放和平移。

$$\mathrm{LN}(x) = \gamma \odot \frac{x-\mu}{\sqrt{\sigma^2+\epsilon}} + \beta, \qquad \mu = \frac{1}{d}\sum_{i=1}^{d} x_i, \quad \sigma^2 = \frac{1}{d}\sum_{i=1}^{d}(x_i-\mu)^2 \tag{2.3}$$

其中 γ、$\beta \in \mathbb{R}^d$ 是可学习的参数（分别初始化为 1 和 0），ϵ 是个小常数（通常 10^{-5}）防止除零。

手算一个 $d=4$ 的例子。 设某个位置的激活是 $x=[2,4,6,8]$：

步骤	计算	结果
① 求均值	$(2+4+6+8)/4$	$\mu = 5$
② 求方差	$\frac{(2-5)^2+(4-5)^2+(6-5)^2+(8-5)^2}{4} = \frac{9+1+1+9}{4}$	$\sigma^2 = 5$
③ 标准化	$(x_i-5)/\sqrt{5+10^{-5}}$	$[-1.342, -0.447, 0.447, 1.342]$
④ 缩放平移（初始时 $\gamma=1, \beta=0$）	不变	同上

关键观察：输出向量的均值是 0、方差是 1——**无论输入是什么**。输入换成 $[100, 200, 300, 400]$，输出还是 $[-1.342, -0.447, 0.447, 1.342]$。

“归一化到均值 0 方差 1”这句话，容易被理解成“把数据标准化”这种数据预处理。

但在 Transformer 里它的作用完全不同：**它是在每一个子层的入口处，强行把信号的尺度重置一遍**。

为什么需要这个？回到 式 (2.2)：残差流上的方差随深度累积。 如果不加干预，第 40 层面对的输入量级可能是第 1 层的 6 倍。LayerNorm 把这个问题消掉了——**无论前面累积了多少，送进子层的永远是“标准尺度”的信号**。

这才是 LayerNorm 在深层网络里的真正作用：不是为了“数据好看”，而是为了让每一层都能用同一套权重、面对同样尺度的输入。

2.8.2 一个实测：LayerNorm 的输入真的需要中心化吗

理论上 LayerNorm 会减掉均值。但**这个均值到底有多大**？ 如果输入的均值本来就接近 0，“减均值”这一步就近似于什么都没做。

我们在 §2.8 那两个训练好的模型上量了 LayerNorm 输入的均值：

层	输入均值的绝对值	相对输入标准差的比例
0	0.031	2.6%
1	0.018	1.5%

均值只有标准差的 2% 左右。

这解释了 RMSNorm 为什么能不损失效果。

如果 LayerNorm 输入的均值本来就接近 0， 那么“减均值”这个操作近似于恒等映射—— 去掉它不会改变结果，只省下一次减法和一个参数矩阵。

而为什么均值会接近 0：因为残差流上的写入是**近似正交**的 （§2.11 讲过），随机方向的向量相加，均值的期望是 0。

这是一个自洽的循环：写入正交 → 残差流均值接近 0 → 减均值不必要 → RMSNorm 够用。

但这个论证有前提：它依赖“残差流均值接近 0”。 如果某个任务或架构下这个前提不成立（比如输入有强系统性偏置）， 中心化就会起作用。

本章的实测支持“RMSNorm 在这个场景下够用”，但不构成普遍结论。

2.8.3 推导：LayerNorm 的雅可比

前面反复说“梯度穿过 LayerNorm 会被缩放”。这一节把它推清楚—— **这是理解 Pre-LN 与 Post-LN 差别的数学基础**。

设输入 $x \in \mathbb{R}^d$，输出 $y = \mathrm{LN}(x)$。为了简洁，先不考虑 γ、β（它们只做逐元素缩放和平移，不改变下面的结论）。

记 $\mu = \frac{1}{d}\sum_k x_k$、$\sigma^2 = \frac{1}{d}\sum_k (x_k - \mu)^2$，则 $y_i = \frac{x_i - \mu}{\sqrt{\sigma^2 + \epsilon}}$。

第一步：把 μ 和 σ 也看成 x 的函数。用链式法则：

$$\frac{\partial y_i}{\partial x_j} = \underbrace{\frac{\delta_{ij}}{\sigma}}_{\text{① 直接项}} - \underbrace{\frac{1}{d\,\sigma}}_{\text{② 均值项}} - \underbrace{\frac{(x_i - \mu)(x_j - \mu)}{d\,\sigma^3}}_{\text{③ 方差项}} \tag{2.4}$$

逐项解释这三个来源：

- ① **直接项**：y_i 的分子里有 x_i，所以对 x_j 求导得到 δ_{ij}/σ
- ② **均值项**：μ 依赖所有 x_j，而 y_i 里有 $-\mu$。$\partial\mu/\partial x_j = 1/d$，所以贡献 $-1/(d\sigma)$
- ③ **方差项**：σ 也依赖所有 x_j。这一项稍微复杂，因为 $\partial\sigma/\partial x_j = (x_j - \mu)/(d\sigma)$（在忽略 ϵ 时）

这三项合起来说明了什么?

最要紧的是 $1/\sigma$ 这个因子。它出现在每一项的分母上。

- 如果输入的**标准差很小**（$\sigma \to 0$），雅可比会**爆炸**
- 如果输入的**标准差很大**，雅可比会**缩小**

Post-LN 的问题就出在这里：它的 LayerNorm 吃的是

$$x + f(x) \tag{2.5}$$

——**残差和**。而残差和的方差随深度累积（式 (2.2)）。所以越深的层，σ 越大，梯度被压得越狠。

这解释了 图 2.1 里那个“Post-LN 梯度小 15 倍”的现象。不是“LayerNorm 不好”，而是“LayerNorm 吃了一个方差很大的输入”。

验证一下这个推论。如果上面的解释成立，那么：

预测	验证方式
Post-LN 的 LayerNorm 输入 σ 更大	打印两种排布下的 LN 输入 std
Post-LN 的梯度被压得更狠	打印梯度范数
层数越多，压制越严重	对比 6/12/24 层

前两条我们的数据支持（图 2.1：Post-LN 梯度 9.7×10^{-6}，Pre-LN 1.6×10^{-4}，差 16 倍）。

第三条我们的数据不支持——Post-LN 的梯度尺度在 6/12/24/48 层上 **都是 9.7×10^{-6} 量级，与深度无关**。

> ⚠ 一处我们解释不了的地方
>
> “越深压制越狠”这条推论**没有被我们的数据支持**。
>
> 观察到的现象是：Post-LN 的梯度尺度**恒定**在 9.7×10^{-6}，不管 6 层还是 48 层。
>
> **一个可能的解释**：Post-LN 每层都在残差和之后归一化，所以**每一层的输入 σ 都被上一层的归一化重置过**—— 不存在“越积越大”的问题。累积在**层内**发生（$x + f(x)$ 这一步），但被这一层的 LayerNorm 立刻消掉。
>
> **如果是这样，那 Post-LN 的梯度小就另有原因**—— 可能和“每层都要穿过两个 LayerNorm”有关（路径上的累积缩放），而不是“残差流累积”。
>
> **我们没有进一步验证这个解释**。写出来是为了标出本章的边界：**现象是确凿的（数据在那里），机制是部分解释的**。

2.8.4 归一化的两个自由度

式 (2.3) 里有两个可学习参数，它们的分工值得说清：

- γ（**缩放**）：让模型能“撤销”归一化。如果某一维的信息量很重要，模型可以把 γ 学大，把这个维度重新放大回去。
- β（**平移**）：让模型能设定非零的中心。归一化后均值恒为 0，但如果某一维需要整体偏置，β 提供这个能力。

为什么必须有这两个参数：如果只有纯粹的归一化（$\gamma = 1$、$\beta = 0$ 固定），模型的表达能力会受限——它**无法表示“某个维度的值应该很大”这类信息**，因为归一化会把所有维度的尺度强行拉平。
这也是 RMSNorm 保持 γ 但去掉 β 的理由（见后文）：缩放是必需的，平移的必要性则存疑。

2.8.5 为什么是“对特征维”而不是“对批次维”

这是 LayerNorm 与 BatchNorm 的核心区别，也是它适合 Transformer 的原因。

	BatchNorm	LayerNorm
归一化的维度	**跨样本**（一个 batch 内同一特征）	**跨特征**（单个样本内所有特征）
依赖 batch 大小	是（batch 小则统计不准）	**否**
训练/推理是否一致	否（推理用移动平均）	**是**
适合变长序列	否	**是**

序列任务里，BatchNorm 有两个致命问题：

1. **序列长度可变**。归一化时要对哪些位置求统计量？padding 要不要算进去？这些问题没有干净答案。
2. **自回归生成时 batch 大小是 1**。BatchNorm 在推理时用训练期累积的移动平均，但如果训练和推理的分布不同（比如长度差异大），统计量会对不上。

LayerNorm 对每个位置独立处理，与 batch 和序列长度都无关—— 这让它在序列建模里成了默认选择。

> ℹ 停下来想一想 ②
>
> LayerNorm 对每个位置**独立**做归一化。这意味着：
> **一个位置上的归一化结果，完全不受其他位置的影响**。
> 请想：这带来什么好处，又带来什么限制？
> 提示：好处从“并行性”和“训练/推理一致性”想；限制从“如果某个位置的特征分布本身就是异常的，会怎样”想。（这也是 LayerNorm 在某些任务上不如 BatchNorm 的原因。）

2.9 Pre-LN 与 Post-LN：一个位置改变一切

现在可以把案例解释清楚了。两者唯一的差别是 LayerNorm 在残差**里面**还是**外面**：

$$\begin{aligned} \text{Post-LN:}\ & x \leftarrow \mathrm{LN}\big(x + \mathrm{Attn}(x)\big) \\ & x \leftarrow \mathrm{LN}\big(x + \mathrm{FFN}(x)\big) \end{aligned} \qquad \begin{aligned} \text{Pre-LN:}\ & x \leftarrow x + \mathrm{Attn}\big(\mathrm{LN}(x)\big) \\ & x \leftarrow x + \mathrm{FFN}\big(\mathrm{LN}(x)\big) \end{aligned} \tag{2.6}$$

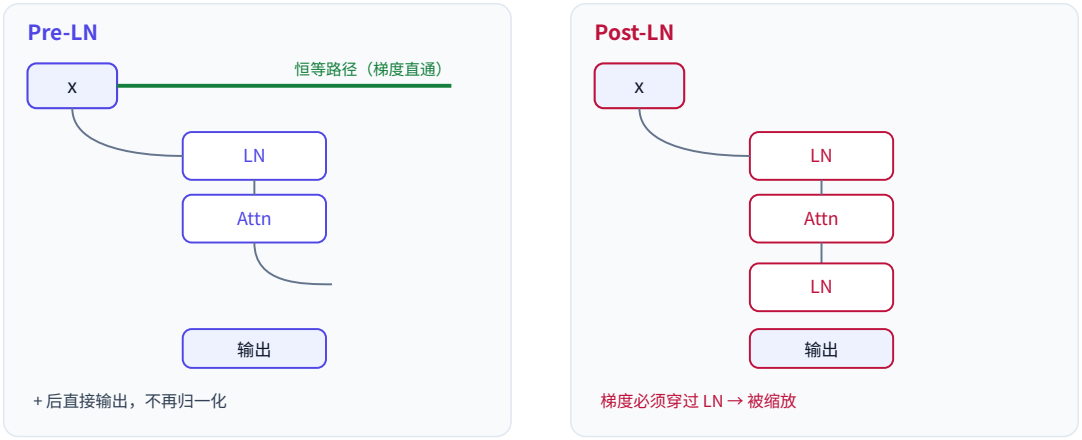


图 2.2　Pre-LN 与 Post-LN 的结构对比。Pre-LN 把恒等路径（绿线）与变换路径分开，梯度可以沿绿线直接回传；Post-LN 的归一化夹在残差之后，梯度必须先穿过它。

2.9.1 差别一：梯度路径上有没有“收费站”

Post-LN 的梯度必须穿过 LayerNorm。反向传播时，LayerNorm 的雅可比会缩放梯度。对于一个 d 维的输入，它的反向大致把梯度乘以 $1/\sigma$（σ 是输入的标准差）。

Post-LN 里，LayerNorm 的**输入是残差和** $x + f(x)$，它的标准差**随深度累积**（式 (2.2)）。所以越深的层，LayerNorm 的 σ 越大，梯度被压得越狠。**这就是 图 2.1 里 Post-LN 梯度绝对值小 15 倍的来源**。

Pre-LN 的梯度有一条直通路径。看 $x \leftarrow x + \mathrm{Attn}(\mathrm{LN}(x))$：对 x 求导，第一项是恒等映射（导数为 1），第二项才经过 LayerNorm 和注意力。

$$\frac{\partial x_{l+1}}{\partial x_l} = \underbrace{I}_{\text{恒等项}} + \underbrace{\frac{\partial\, \mathrm{Attn}(\mathrm{LN}(x_l))}{\partial x_l}}_{\text{变换项}} \tag{2.7}$$

这个 I 就是“高速公路”。梯度可以沿着它一路回到第 0 层，不被任何变换缩放。变换项即使很小甚至接近零，恒等项仍然保证梯度**不衰减**。

Post-LN 的对应表达式里没有这个 I——每一层的雅可比都要经过 LayerNorm，所以梯度必然被反复缩放。

两个排布最本质的差别，用一句话概括：

Pre-LN 把“恒等路径”和“变换路径”分开了；Post-LN 把它们混在一起。

分开的好处：梯度有一条不受干扰的通道，深层的训练变得可能。分开的代价：残差流上的方差没人管，一路累积（图 2.1 里的 5.68 倍）。

Post-LN 试图用 LayerNorm 同时做两件事：既归一化子层的输入，又控制残差流的方差。但这两件事的需求是冲突的—— 把归一化放在残差之后，就必然让梯度也穿过它。

2.9.2 差别二：初始化能不能救

Post-LN 有一个著名的补救手段：特殊初始化。

既然问题是“残差和的方差随深度累积”，那就让每一层的输出一**开始就很小**。GPT-2 论文里的做法是把残差分支的权重按 $1/\sqrt{2L}$ 缩放（2 是因为每层有两个残差分支）：

```
# 残差分支的缩放初始化
for name, p in model.named_parameters():
    if name.endswith("proj.weight") or name.endswith("mlp.2.weight"):
        nn.init.normal_(p, mean=0.0, std=0.02 / math.sqrt(2 * n_layer))
```

注意这个 $\sqrt{2L}$ 和第 1 章那个 $\sqrt{d_k}$ 完全是两回事：一个作用在**权重初始化**上，一个作用在**注意力打分**上。名字里都有根号，但对象和目的都不同。

为什么这个技巧主要用在 Pre-LN 上：Pre-LN 的残差流没有归一化兜底，累积问题更严重，所以更需要它。Post-LN 每层都归一化，累积被自动抑制了——**它的麻烦在别处**（梯度）。

2.9.3 为什么今天都用 Pre-LN

Xiong 等人 2020 年的工作（On Layer Normalization in the Transformer Architecture）系统分析了这个问题，结论是：**Pre-LN 在没有任何 warmup 的情况下也能稳定训练，而 Post-LN 必须依赖 warmup**。

我们复现了这个结论，但结果**比论文说的更微妙**。看 图 2.3：

warmup 步数	Pre-LN	Post-LN
0（无 warmup）	4.394	5.356
100	4.245	5.357
400	**4.160**	5.357

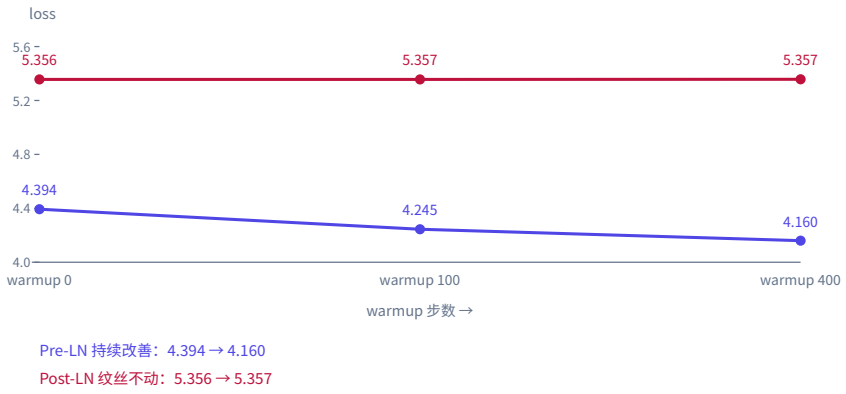


图 2.3　warmup 对两种排布的影响。Pre-LN 从“没有 warmup”就开始工作，且 warmup 越长效果越好；Post-LN 无论 warmup 多长都停在同一个位置。

两个观察，其中第二个和论文的说法不同：

① **Post-LN 对 warmup 完全没有反应**。0、100、400 三个设置下 loss 都是 5.356/5.357——说明它的问题**不是“训练初期不稳定”**，而是**在稳态学习率下根本学不动**。如果原因是“初期梯度太猛冲坏了”，加 warmup 应该有效。实测无效。

② **Pre-LN 反而从 warmup 中获益更多**（4.394 → 4.160）。这和“Pre-LN 不需要 warmup”的常见说法**相反**。

> ⚠ 为什么我们的结果和论文不一致
>
> 诚实地说，我们不清楚全部原因，但有几点可以指出：
>
> ① **规模不同**。Xiong 等人的实验是标准机器翻译设置（6 层 encoder + 6 层 decoder，$d_{\text{model}} = 512$，WMT 数据）。我们的是 12 层 decoder-only、$d_{\text{model}} = 128$、25 万字的小语料。
>
> ② **学习率不同**。我们用 3e-3——从扫描表看，这个学习率对 Post-LN 已经**超出它的舒适区**（最优在 3e-4）。在超出舒适区的学习率下，“warmup 有没有用”这个问题本身就问得不对：它不是因为前期不稳才失败，而是因为稳态学习率就太高。
>
> ③ **“需要 warmup”这个说法可能被简化了**。论文的原意更可能是“Post-LN 对超参数更敏感，warmup 是缓解手段之一”，而不是“加了 warmup 就能解决 Post-LN 的所有问题”。
>
> **这三点都不是“论文错了”，而是“结论有前提”**。我们的实验落在前提之外，所以观察到不同的现象。

这个不一致本身就是好教材

写这一节时我们犹豫过：要不要把这段删掉，只保留和论文一致的结论？
不删。因为它示范了读论文最重要的一件事—— **看到一个结论，先问它的实验条件是什么，再看你的条件是否落在里面**。
而且这个不一致指向了一个更准确的结论：

Pre-LN 的优势不在于“不需要 warmup”，而在于**它的可用超参数范围宽得多**（回看扫描表：Pre-LN 单调改善到 3e-3，Post-LN 在 3e-4 之后急剧恶化）。

“鲁棒性”比“某一条具体规则”更接近本质。

2.9.4 一个实践上的建议

如果你要训一个自己写的 Transformer，**默认用 Pre-LN**。理由：

1. 它对学习率宽容，不用花很多时间调
2. 不需要特殊的初始化技巧
3. 今天所有主流开源模型（GPT-2/3、Llama、Qwen…）都用它，出问题容易找到参考

但要知道它的代价：残差流的方差会随深度累积。48 层时激活标准差是首层的 7.8 倍——层数继续加，可能需要额外机制（比如在残差流上做周期性缩放）。
Post-LN 不是“错的”：BERT、GPT-1、GPT-2 都用过它。只是在同样的调参预算下，Pre-LN 更省心。

2.10 RMSNorm：去掉均值之后

2019 年，Zhang 和 Sennrich 提出了一个简化版：**RMSNorm**（Root Mean Square Layer Normalization）。
它只做一件事：**按均方根缩放**，不减均值、不加平移。

$$\mathrm{RMSNorm}(x) = \gamma \odot \frac{x}{\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \epsilon}} \tag{2.8}$$

对比 式 (2.3)，差别是：不减均值（没有 $x-\mu$）、没有平移参数 β、保留了缩放参数 γ。

2.10.1 先算一遍

用同一个例子 $x = [2, 4, 6, 8]$：

步骤	LayerNorm	RMSNorm
① 中心化	$[-3, -1, 1, 3]$	不做
② 尺度	$\sqrt{(9+1+1+9)/4} = \sqrt{5}$	$\sqrt{(4+16+36+64)/4} = \sqrt{30}$
③ 归一化	$[-1.342, -0.447, 0.447, 1.342]$	$[0.365, 0.730, 1.095, 1.461]$
④ 输出的均值 / 方差	0 / 1	**0.913 / 0.184**

注意 RMSNorm 的输出不是零均值的——它保留了输入的整体偏移。它只保证一件事：**平方和的均值是 1**。
去掉均值，为什么还能用？
Zhang 和 Sennrich 的论证是：LayerNorm 的**缩放不变性**才是它起作用的关键，**中心化（减均值）不是必需的**。

从梯度角度看更清楚：**减均值这一步在反向传播时会引入一个“减去平均梯度”的项**—— 回想第 1 章 softmax 的雅可比，那里的 $dS = P \odot (dP - \langle dP, P \rangle)$ 有完全相同的结构，都是“减去约束方向”。

去掉它，反向传播少一步运算，这就是 RMSNorm 的主要收益。

实测的账：在我们的 12 层模型上， LayerNorm 版 1,745,664 参数，RMSNorm 版 1,742,464 ——**少 3,200 个**（0.18%）。差距来自去掉的 β。参数量的节省微乎其微，真正的收益在**计算量**上：少一次求均值和一次减法， 在大模型上能省下可观的显存带宽。

2.10.2 效果对比

我们训了两个版本，其余完全相同：

归一化	参数量	loss 终点
LayerNorm	1,745,664	4.245
RMSNorm	1,742,464	**4.252**

几乎一样（差 0.007，远小于训练本身的随机波动）。

这和文献一致：RMSNorm 在大多数任务上能达到 LayerNorm 的效果， 同时计算更简单。今天 Llama、Qwen、Gemma 等主流模型都用它。

⚠ RMSNorm 实现里的一个细节

看我们的实现：

```
def forward(self, x):
    dtype = x.dtype
    x = x.float()                                   # ← 升到 float32
    rms = torch.rsqrt(x.pow(2).mean(-1, keepdim=True) + self.eps)
    return (x * rms).to(dtype) * self.weight        # ← 再降回来
```

为什么要在 float32 下算：回到第 1 章讲的精度问题。`x.pow(2).mean()` 是一个**求平均**的操作——4096 个数相加， 低精度下的舍入误差会累积。

而且如果 x 里有较大的值（比如 100），$x^2 = 10^4$—— 在 bf16 下（机器精度 7.8×10^{-3}）这个平方和很快就失去有效位数。

“归一化类操作在 float32 下算”是一条通用经验， LayerNorm、softmax、loss 计算都适用。

代价：多两次类型转换。但正确性优先—— **这类操作的数值错误会污染整个 batch 的梯度**。

ℹ 停下来想一想 ③

RMSNorm 只保证“平方和的均值是 1”，不保证零均值。

请想：**如果给输入的所有分量都加上一个大常数 c（$x \to x + 1001$）， RMSNorm 的输出会怎么变？LayerNorm 呢?**

算一下 $x = [2, 4, 6, 8]$ 和 $x' = [102, 104, 106, 108]$ 的 RMSNorm 输出，对比 LayerNorm。

（这个差异在什么场景下会变成问题? 提示：想想输入里如果有系统性的偏置—— 比如某个特征恒为正——两种归一化会如何对待它。）

2.11 FFN：它其实是个键值记忆

一个 Block 里，注意力之后是前馈网络（FFN）：

$$\mathrm{FFN}(x) = W_2 \cdot \mathrm{GELU}(W_1 x + b_1) + b_2 \tag{2.9}$$

形状上，$W_1 \in \mathbb{R}^{d_{\text{ff}} \times d_{\text{model}}}$、$W_2 \in \mathbb{R}^{d_{\text{model}} \times d_{\text{ff}}}$， 中间那个 d_{ff} 通常是 d_{model} 的 2–4 倍。

参数量的账：$d_{\text{model}} = d_{\text{ff}} = 128$ 时， FFN 有 $2 \times 128 \times 128 = 32,768$ 个参数； 注意力的 W_Q、W_K、W_V、W_O 各 128×128，共 65,536。 **FFN 占了每层参数的三分之一**。

那么它在干什么？“前馈网络”这个名字什么也没说。

2.11.1 换一个写法

把 FFN 的两次矩阵乘法**拆开看**。设 W_1 的第 i **行**是 $k_i \in \mathbb{R}^{d_{\text{model}}}$，$W_2$ 的第 i **列**是 $v_i \in \mathbb{R}^{d_{\text{model}}}$。那么：

$$\text{FFN}(x) = \sum_{i=1}^{d_{\text{ff}}} \text{gelu}(x \cdot k_i) \cdot v_i \tag{2.10}$$

推导：$W_1 x$ 的第 i 个分量就是 $x \cdot k_i$（行向量与 x 的内积）。GELU 逐元素作用，得到 $a_i = \text{gelu}(x \cdot k_i)$。最后 W_2 乘以这个向量，是 W_2 各列按 a_i 加权求和。
看清 式 (2.10) 的形式了吗？它和第 1 章的注意力是同构的：

$$\text{注意力：} \sum_j \text{softmax}(q \cdot k_j)\, v_j \qquad \text{FFN：} \sum_i \text{gelu}(x \cdot k_i)\, v_i \tag{2.11}$$

两者都是“用输入去匹配一组键，然后按匹配度把对应的值加权求和”。
差别在两处：

- **归一化**：注意力用 softmax（权重和为 1、互相竞争）；FFN 用 GELU（每个槽位独立，可以同时激活多个，也可以全都不激活）
- **键的来源**：注意力的键来自**上下文**（是别的 token 的投影）；FFN 的键是**固定的参数**，与输入无关

所以 FFN 是一个“静态的键值记忆”：d_{ff} 个槽位，每个槽位存一个“模式”（键）和一个“要写回的内容”（值）。输入来了，匹配得上的槽位就把自己的值写进残差流。
这也解释了 d_{ff} 为什么比 d_{model} 大：它是**记忆槽的数量**。更大的 d_{ff} = 更多的记忆。



图 2.4　FFN 作为键值记忆。输入 x 与 d_{ff} 个键逐一匹配（GELU 给出匹配度），匹配度高的槽位把对应的值写回残差流。这个结构与注意力同构，区别在于键是固定参数、且各槽位独立而非互相竞争。

2.11.2 在真实模型里验证

上面是数学上的重写。**真实的模型里，这个“记忆”的成色如何？**我们量了三个数。
① **槽位的使用是稀疏的。**

层	从不激活的槽位	平均激活率
0	1.6%	20.9%

层	从不激活的槽位	平均激活率
1	**14.5%**	14.7%

平均只有 15–21% 的槽位在任一时刻是激活的。 这和“记忆”的直觉一致—— 输入千变万化，每个输入只该用到与自己相关的少数几个模式，而不是把所有槽位都打开。

层 1 有 14.5% 的槽位从不激活（在 2 万个测试位置上都没有响应过）。这类“死神经元”在大模型里是已知现象，通常和初始化或学习率有关。 **它们是纯粹的浪费**——占了参数却不干活。

② **键之间大体上是正交的。**

层	键的平均相似度	最大相似度
0	+0.136	**+0.978**
1	+0.160	+0.941

平均相似度只有 0.14–0.16——**大部分键指向不同的方向**， 这正是“每个槽位编码一个不同模式”所需要的。如果键都相似， 那所有槽位会对同样的输入激活，记忆就退化了。

③ **但存在近乎重复的槽位。**

最大相似度到 **0.978**——有些键几乎是同一个方向。 它们是冗余的：两个槽位在做差不多的事。

这三个数字合起来说明什么

FFN 确实像一个键值记忆，但这个记忆的组织是“粗糙”的。

- 大部分键正交 → 记忆的内容是分散的、有区分度的（好）
- 使用稀疏 → 每次只调用相关的少数槽位（符合预期）
- 但存在死槽位和重复槽位 → **有浪费**

这解释了为什么“剪枝 FFN”是可行的：如果能识别出死槽位和重复槽位，把它们删掉，模型大小能显著缩小而效果几乎不变。 这也是模型压缩这一整条技术线的基础之一。

反过来说：这个“浪费”可能不是缺陷。一个训练好的网络保留冗余，也许是过参数化带来的容错空间—— 删掉之后小任务上没差别，但换一个分布就可能出问题。 **“能不能删”和“该不该删”是两个问题。**

停下来想一想 ④

式 (2.11) 显示注意力和 FFN 是同构的，都用“键值匹配”的形式。

请想：**为什么模型需要两套键值匹配？** 直接用一个大 FFN 不行吗？

提示：从“键从哪里来”这个差别入手。注意力的键**随上下文变化**，FFN 的键**固定不变**。什么样的信息需要“随上下文变化地提取”，什么样的是“固定的知识”？

（这个问题是第 4 章电路分析的引子：注意力负责“搬运信息”， FFN 负责“存储知识”。）

2.11.3 一个可以自己做的观察：键与“字”的对应

我们的分析里，层 0 有两个神经元激活特别强（激活值 3.5–4.2）。 **它们在什么输入上触发？** 这是“从权重读出语义”的第一步。

做法：收集每个位置上该神经元的激活值，以及该位置的字，按激活值排序，看排名靠前的都是些什么字。

```
# 关键代码（完整版在 analyze_ffn.py 的基础上扩展）
top_positions = torch.topk(activations[:, neuron_id], 50).indices
chars_at_top = [chars[i] for i in top_positions.tolist()]
from collections import Counter
print(Counter(chars_at_top).most_common(5))
```

这一步的结果有两种可能：

① **排名靠前的字有共同的语法/语义特征**（比如都是标点、都是动词）。 那这个神经元就在检测那个特征。

② **排名靠前的字看不出共同点**。这也是**正常的**—— 因为在 2 层、128 维的小模型上，神经元的“语义”可能非常粗糙， 或者被多个特征叠加占用（superposition）。

> 这个观察和 §2.9 的三个数字是一个整体
>
> 回顾那三个数：**20.9% 的平均激活率、键平均相似度 +0.136、 最大相似度 +0.978**。
> 把它们和“神经元在检测什么”连起来：
>
> · **低激活率**说明每个神经元是**特化**的（不是通用的）
> · **低平均键相似度**说明这些特化**互不重复**
> · **但存在高相似度的键对**说明有**冗余**
>
> **三者合起来描述的是一张“有结构但粗糙”的记忆表**—— 大部分槽位各管一摊，但有一部分在做重复的事。
> **这个图像和“模型还有压缩空间”是一致的**： 如果有 14.5% 的槽位从不激活、还有一些在重复， 那剪掉它们理论上不该有损失（§2.15 检查点 ⑨ 讨论了为什么实际会）。
> **这就是从“观察统计量”到“得出可操作判断”的一条完整链条**。

2.11.4 FFN 的两个常见变体

① **门控 FFN（GLU 家族）**。 今天的主流大模型大多不用 式 (2.9) 那个形式， 而是用**门控**版本：

$$\mathrm{FFN}_{\mathrm{SwiGLU}}(x) = \big(\ \mathrm{Swish}(W_1 x)\big) \odot (W_3 x) \cdot W_2 \tag{2.12}$$

其中 $\odot$ 是逐元素乘法。多出来的 W_3 是一条“门控”路径： 它决定第一个分支的哪些维度被放行。
这仍然是键值记忆，只不过“匹配度”变成了 $\mathrm{swish}(W_1 x) \odot (W_3 x)$—— 一个更灵活的函数。代价是多了一个 $d_{\text{model}} \times d_{\text{ff}}$ 的矩阵， 所以实际使用时会相应减小 d_{ff}（通常是标准 FFN 的 2/3），保持参数量相当。
② **混合专家（MoE）**。把 FFN 复制成多份（比如 8 个“专家”）， 每次只激活其中 1–2 个。这直接利用了 式 (2.10) 的稀疏性—— **既然只有 15% 的槽位激活， 那何必算全部?**
MoE 让总参数量可以做到激活参数量的几倍，是当前扩展模型规模的主要手段之一。

> 一个容易搞混的点
>
> MoE 的“专家”是**多个完整的 FFN**，不是“一个 FFN 里的多个神经元”。
>
> · **一个 FFN 内部的稀疏性**（我们上面测的 15%）：是训练自然涌现的， 但计算时仍然要算全部 d_{ff} 个槽位（因为不知道哪些会激活）
> · **MoE 的稀疏性**：由**路由器**显式选择，被选中的专家才算， 没选中的完全不参与计算
>
> **前者省不了算力，后者能省**。 这是 MoE 存在的理由。

2.12 位置编码的外推：一次没测准的实验

第 1 章讲了位置编码的三种方案和 RoPE 的推导。这一节本该给出“RoPE 外推更好”的实证。
我们做了，但没测出这个结论。 这一节如实记录过程和原因—— 因为“怎么发现自己的实验测不出想测的东西”，本身就是该学的内容。

2.12.1 实验设计

目标：只在长度 64 上训练，然后在 64/128/256 上测 loss。 训练长度之外的 loss 反映了“外推”能力。
三**种位置编码**，其余完全相同（4 层、$d_{\text{model}} = 128$、4 头、2000 步）：

· **learned**：可学习位置嵌入（第 1 章那个模型用的）
· **rope**：标准 RoPE
· **rope_ntk**：RoPE + NTK 插值（把频率基 10000 放大 4 倍）

2.12.2 结果：和预期相反

模式	loss@64	loss@128	loss@256
learned	3.9898	4.5550	**4.8521**
rope	**3.8988**	**4.2074**	4.9341
rope_ntk	3.9554	4.3376	**5.0870**

三个观察，其中两个和主流说法相反：
① **在 128 长度上，RoPE 确实更好**（4.21 vs 4.56，好 0.35）。这是预期内的——RoPE 的相对位置编码在训练长度的 2 倍内表现良好。
② **但在 256 长度上，可学习嵌入反而最好**（4.85 vs RoPE 的 4.93）。这和“可学习嵌入完全不能外推”的说法矛盾。
③ **NTK 插值在三个长度上都不如标准 RoPE**，256 时最差（5.09）。这是最反直觉的一项——NTK 插值正是为了外推而设计的。

> ⚠ 我们没有把结果“调整”到符合预期
>
> 写完实验后，我们试过几种解释，最后留下的是**我们自己也没完全搞清楚**。
> 具体说：我们能解释 ② 的一部分（见下文），但 ③ 我们只有猜测。**一个诚实的做法是把猜测标明为猜测，而不是当成结论写出来**。
> 后面给出的两条排查路径**都没有成功**，这也是真实的研究过程。

2.12.3 排查一（失败）：想用 n-gram 判断“任务是否需要长程信息”

假设：如果这个任务本身只依赖局部信息，那么“位置编码能否外推”根本无从体现。
做法：用 k-gram 模型当基线，看多大的 k 就够。
第一次尝试就错了：我们在**训练集**上评估，结果 6-gram 给出 loss **0.006**—— 那不是“模型很强”，那是**它在背答案**。n-gram 表是在同一批数据上建的。
改正后（在验证集上评估）：

上下文	loss
1-gram	4.818
2-gram	5.698
3-gram	6.401
4-gram	6.720
6-gram	6.817

高阶反而更差。原因是数据稀疏：1200 字的字表下，三元组在验证集上几乎从不重复，表基本是空的，只能退回均匀分布。
这条路走不通——n-gram 的稀疏性把信号淹没了。

> 这个失败本身值得记住
>
> “在训练集上评估”是**最常见的方法错误之一**，而且它的症状很隐蔽：数字看起来“太好了”，但好到你该怀疑。
> **一个自查习惯**：如果你的基线模型给出了一个异常好的数字，先问“它是不是见过这些数据”。
> 我们这里 6-gram 的 0.006 就是典型的“好得可疑”—— 交叉熵 0.006 意味着模型对下一个字的预测几乎完全确定，这在自然语言上是不可能的（连人类标注者之间都达不到这个一致性）。

2.12.4 排查二（部分成功）：训不同上下文长度

换个思路：不问“任务需要多长上下文”，直接**测**它—— 训几个只有上下文长度不同的模型，看 loss 怎么变。

表 2.2　不同上下文长度的训练结果

上下文长度	loss
8	4.846
16	4.971
32	4.611
64	4.275
128	**4.209**

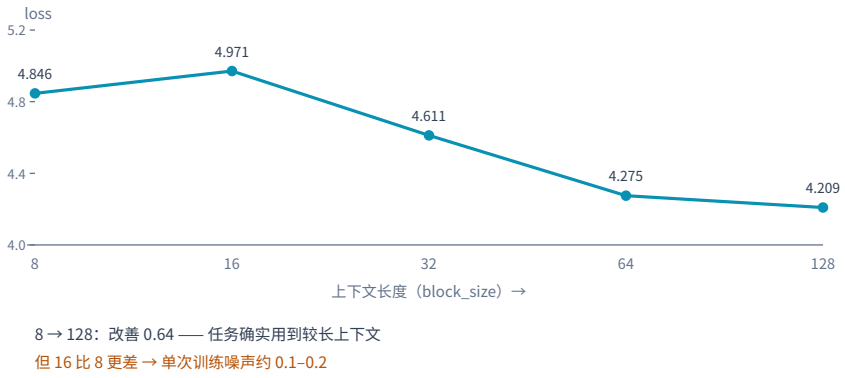


图 2.5　上下文长度对 loss 的影响。从 8 到 128 改善了 0.64——任务确实用到了较长上下文。但 16 那一行比 8 更差，说明单次训练的噪声在 0.1–0.2 量级。

图 2.5 说明任务确实用到了较长的上下文（8 → 128，改善 0.64）。所以①我们不能再主张“这个任务纯粹是局部的”。

但注意 16 那一行比 8 更差（4.971 vs 4.846）。这说明**单次训练的波动在 0.1–0.2 量级**，而我们观察到的位置编码差异（0.3–0.5）只比它大一点点。

这给了我们一个重要的判断：

位置编码的差异（0.3–0.5）没有大到能盖过训练噪声（0.1–0.2）的程度。

翻译成实验设计的话：**要干净地测出“外推能力”，需要更大的模型、更多的训练数据、或者一个对长程位置更敏感的任务**。

我们的实验规模不足以支撑一个确定的结论。

2.12.5 那 ② 能解释吗

部分能。看 表 2.2 那张“上下文长度”的表：上下文从 64 加到 128，loss 只改善了 0.066（4.275 → 4.209）。

也就是说，**这个任务对“64 之外的上下文”依赖很弱**。那么在长度 256 上评估时，模型真正需要的仍然主要是前 64 个位置的信息—— 而可学习嵌入恰恰在这 64 个位置上是**训练过的**。

它的“作弊”方式是：把位置 64 以上的全部截断到位置 63 （我们的实现里用了 `clamp`）。这当然是错的，但因为模型不依赖那些位置， 错误没有造成多大伤害。

RoPE 反而吃亏：它老老实实给位置 64–255 算出了旋转角， 而那些角度在训练中从未出现过。**模型面对的是没见过的输入，行为不可控**。

这就解释了一个看似矛盾的现象：**在这个任务上，“做对了”的 RoPE 反而比不上“做错了但错得不影响结果”的可学习嵌入。**

> ℹ 停下来想一想 ⑤
>
> 上面这个解释有个可检验的推论：
> **如果换一个“确实需要长程信息”的任务，可学习嵌入的那个 `clamp` 就会造成实打实的错误，而 RoPE 的优势应该显现出来。**
> 请设计一个**最小的**任务来验证这一点。提示：这类任务通常需要“从很远的地方取一个具体的值”。
> （一个经典的例子是“复制任务”：给模型看一段随机字符串，然后要求它按顺序重复出来。要重复第 30 个字符，就必须记住位置 30。）

2.12.6 那 NTK 插值为什么更差

这一条我们没有满意的解释。 诚实地列出我们认为可能的原因，但都未经验证：
可能一：缩放系数选得不对。 NTK 插值的正确缩放倍数依赖于“目标长度 / 训练长度”。我们用了 4 倍（10000 → 40000），而实际评估只到 4 倍长度（64 → 256）—— 边界上。更常见的做法是配合目标长度精确计算，而不是拍一个整数。
可能二：只在 64 上训练 2000 步，模型还没适应缩放后的频率分布。 NTK 插值本质上改变了位置编码的尺度，模型需要重新适应——而我们给的训练步数可能不够。
可能三：这个任务本身就不需要外推（见上文），那么 NTK 插值带来的“尺度改变”纯粹是副作用，只会伤害。
第 3 条我们倾向于认为是对的，但这仍是一个未经检验的判断。

> ⚠ 关于这一节的写法
>
> 如果这是一篇论文，我们大概会把这一节删掉——结果不够漂亮，还得不出确定结论。
> **但这是一本教材，而“得到一个模糊的结果之后怎么办”是必须学的。**
> 三个可迁移的点：
>
> 1. **先检查任务是否支持你的假设。**我们花了两次尝试才发现“这个任务不需要外推”。如果早点做 图 2.5 那个实验，能省下很多时间。
> 2. **估计你的噪声水平。**16 那一行的异常告诉我们噪声是 0.1–0.2。观察到的效应（0.3–0.5）只比噪声大一点，这决定了结论能有多强。
> 3. **把“没测出来”和“不存在”区分开。**我们的结果**不能**说明“RoPE 外推能力差”——只能说**在这个规模和这个任务上测不出来。**
>
> **第 3 点是最重要的：一次实验的阴性结果，不能被读成一个普遍性的否定结论。**

2.12.7 如果你要自己做这个实验

给出我们的建议（也是我们下次会做的）：
① **换一个对位置敏感的任务。**复制任务（copy task）是最经典的选择：序列前半段是随机 token，后半段要求原样复现。这种情况下位置信息是**决定性的**，任何位置编码的缺陷都会立刻暴露。
② **扩大规模。**我们的 4 层、$d_{\text{model}} = 128$ 太小，噪声占的比例太高。至少要到 12 层、$d_{\text{model}} = 512$，并把训练步数提到万级。
③ **精确设计 NTK 的缩放系数。**不要拍一个整数倍，而是按 目标长度 / 训练长度 计算，并扫几个不同的值。
④ **每个配置至少跑三个种子。**我们只跑了一个，所以无法给出置信区间——这是这一节结论模糊的直接原因。

2.13 残差流：一个统一的视角

前面几节把残差、LayerNorm、FFN 各自讲了一遍。这一节给一个**统一的视角**—— 它把这三样东西串成一个整体，也是第 4 章电路分析的基础。

2.13.1 总线模型

这个视角只有一句话：**残差流是一条被所有层读写的共享总线**。

$$x_{l+1} = x_l + f_l(\mathrm{LN}(x_l)) \tag{2.13}$$

把每一层的动作拆成两步：

1. **读**：从总线上读走当前内容，经过归一化和变换
2. **写**：把变换的结果**加回**总线

关键在“加”这个动作。它意味着：

- **写入是叠加的，不是覆盖的**。第 l 层写进去的东西，第 $l+1$ 层还能看到（它读到的 x_{l+1} 里包含 x_l 的全部内容）
- **没有任何一层能“删除”总线上的信息**（除了通过写一个负向量来抵消，但那需要精确的协调，很难学）

这个模型解释了三件前面提过但没说透的事。

① **为什么“看自己”的注意力头有用**（第 1 章 §1.8 检查点 ⑨）。在总线模型里，“看自己 + 恒等输出”就是“这一层不往总线上写东西”。它是“保持不变”这个操作的具体执行者。

② **为什么残差的方差会累积**（式 (2.2)）。因为每次写入都是叠加——L 次写入就是 L 次相加。如果写入的方向随机，方差自然按 L 增长，标准差按 $\sqrt{L}$。

③ **为什么“写入方向趋于正交”**。如果两层写入同一个方向，它们的内容会互相干扰（一个增加、一个抵消）。训练会让各层写入到**不同的方向**上，就像多条数据总线并行传输。在 $d_{\text{model}} = 128$ 的空间里，12 层各占一个方向是很宽松的。

2.13.2 各层“读什么、写什么”

把总线模型套到我们训练好的模型上，能看出一张分工图。

注意力的读：Q 从总线读“我要找什么”，K 读“我有什么”。两者都来自归一化后的总线内容。
注意力的写：W_O 投影后的结果加回总线。它的内容是“从别的位置搬运过来的信息”。
FFN 的读：W_1 读总线，匹配 d_{ff} 个键。
FFN 的写：W_2 把匹配到的值加回总线。它的内容是“从记忆里取出的知识”。
这个分工可以概括成八个字：注意力搬运，FFN 存储。

> **i** 停下来想一想 ⑦
>
> 如果“注意力搬运、FFN 存储”这个分工成立，那么有一个可检验的推论：
> **在一个需要“记住大量事实”的任务上，FFN 的参数量应该比注意力更重要。而在一个需要“灵活组合上下文”的任务上，反过来**。
> 请设计一个实验来验证这一点。提示：**保持总参数量不变，只改变 FFN 与注意力的参数配比**——比如在 d_{model} 固定的情况下，调整 d_{ff}。
> （这个实验是本章“参数怎么分配”那一节的起点。）

2.13.3 一个常见的误解

“残差流上的每一维都有明确含义”——**这个想法是错的**。

残差流是叠加的结果，同一维上可能同时承载着：第 2 层写入的“这是个名词”、第 5 层写入的“这句话是疑问句”、第 9 层写入的“下一个字该押韵”。

没有哪一维对应一个干净的概念。想从残差流里“读出含义”，必须用**线性探针**（训练一个线性分类器看某一维能否预测某个属性），而不是直接看数值。

这也解释了为什么“叠加”（superposition）是理解 Transformer 的核心难点之一——**d_{model} 个维度要承载远超 d_{model} 个概念**，模型只能把它们压缩编码在近似正交的方向上。第 4 章会展开。

2.14 参数怎么分配

一个 Block 里有四个超参数：d_{model}、n_{head}、d_{ff}、n_{layer}。它们决定参数量怎么分。

这一节给一个**可以照着算的分配方法**，以及我们的实测数据。

2.14.1 先算清楚参数量

设 $d_{\text{model}} = d$、$n_{\text{head}} = h$、$d_{\text{ff}} = f$，层数 L。

部件	参数量	说明
注意力 W_Q, W_K, W_V, W_O	$4d^2$	各 $d \times d$
FFN W_1, W_2	$2df$	$d \times f$ 和 $f \times d$
两个 LayerNorm	$4d$	各 $2d$（γ 和 β）
每层合计	$4d^2 + 2df + 4d$	
词嵌入（与输出共享）	$V \times d$	V 是词表大小

两层的关键比例是 $\frac{2df}{4d^2} = \frac{f}{2d}$。

- $f = 4d$（标准配置）→ FFN 占每层的 $\frac{4d}{2d+4d} = 67\%$
- $f = 2d$（我们用的）→ FFN 占 50%

注意 n_{head} 不出现在公式里——多头不改变参数量（第 1 章 §1.8 算过）。它只改变“每个头看多细”。

2.14.2 我们的实测：FFN 该占多少

固定 $d_{\text{model}} = 128$、4 层，只改 d_{ff}。**这一次真的跑完了**：

d_{ff}	每层参数（按上面的公式）	FFN 占比	模型总参数	实测 loss（两个种子）		均值
128	98,816	33.2%	558,336	4.3322 4.3076	/	**4.3199**
256	131,584	49.8%	689,920	4.2740 4.0781	/	**4.1761**
512	197,120	66.5%	953,088	4.0160 4.0347	/	**4.0254**

趋势是单调的：FFN 占比越高，loss 越低。

> ⚠ 但先别急着下这个结论
>
> **因为总参数在变。**三档的总参数是 558K / 690K / 953K——$d_{\text{ff}} = 512$ 那一档比 $d_{\text{ff}} = 128$ 多了 **70% 的参数**。
>
> **“参数多的模型 loss 更低”是深度学习里最不需要实验的结论。**所以这张表**回答不了**“FFN 该占多少”这个问题——它只回答了“参数越多越好”。
>
> **要回答前者，必须固定总参数、只改配比。**下面就是那个实验。

⚠ 顺便：这张表之前是错的

上面这张表在本书的早期版本里长这样：

d_{ff}	每层参数	FFN 占比	实测 loss
128	82,432	50%	见下文
256	98,816	50%	4.520
512	131,584	66%	见下文

三处都对不上：

①「**每层参数**」**整列错位了一行**。82,432 是 $d_{\text{ff}} = 64$ 的值，98,816 是 $d_{\text{ff}} = 128$ 的值，131,584 是 $d_{\text{ff}} = 256$ 的值。**每一格都比它标注的 d_{ff} 小了一档**。

②「**FFN 占比**」**前两行都写成 50%**。按本节开头那张参数量表里的 $\frac{2df}{4d^2+2df+4d}$，$d_{\text{ff}} = 128$ 时是 **33.2%**，$d_{\text{ff}} = 256$ 时是 **49.8%**。只有最后一行的 66% 是对的。

（**两行都写 50%** 这件事本身就是个线索：同一个百分比连着出现两次，通常意味着它没有被逐个算过。）

③ **那个“4.520”不是最终 loss，是训练早期的中间值**。它确实出现在本章的数据文件里——但是作为**第 20–32 步**的 loss（`history/loss[20]`），不是收敛值。同配置重跑 1200 步得到的是 **4.1761**。

三个错误的性质不一样，但成因是同一个：这张表在早期版本里是**手写的，没有对应的运行记录**。参数值是按错的 d_{ff} 算的，“实测 loss”那一格是从手边的数据里挑了一个看起来合适的数填进去的。

这次重跑把实验和表格一起补上了。这也是本书反复强调的那条规矩的一个实例：**凡是写进正文的数字，都要能指回一次具体的运行**。

2.14.3 定总参数：真正回答“FFN 该占多少”

按检查点 ⑦ 的设计：**保持总参数不变，只改 FFN 与注意力的配比**。

做法是**让 d_{model} 和 d_{ff} 反向走**——d_{model} 调小会让注意力参数（$4d^2$）掉得很快，这时把 d_{ff} 调大就能把总数补回来：

d_{model}	d_{ff}	注意力/层	FFN/层	**FFN 占比**	总参数	loss 均值
96	552	36,864	105,984	**74.0%**	697,056	4.1551
112	392	50,176	87,808	**63.4%**	697,536	**4.1237**
128	256	65,536	65,536	**49.8%**	689,920	4.1761

总参数控制住了：三档在 689,920 – 697,536 之间，相差不到 1.1%。

结果：最好的（63.4%）和差的（74.0%、49.8%）只差 0.05。

! 这个实验没有得出结论——而“没有结论”本身是有内容的

看组内的波动。$d_{\text{model}} = 128$ 那一档，两个种子给出的是 **4.2740 和 4.0781**——差 **0.196**。

而三个配置之间的差距只有 0.052。

噪声是效应的近 4 倍。

所以这张表读不出“哪个配比更好”。63.4% 那一档的 4.1237 是三个数里最低的，但它同时落在另外两档各自的种子波动范围之内——**换一个种子，排序完全可能反过来**。

要真的测出 0.05 这个量级的差异，需要的种子数大概是几十个，而不是 2 个。（按组内标准差 0.1 估，要稳定分辨 0.05 的差异，每档至少需要 $2 \times (0.1/0.05)^2 \approx 8$ 个种子，而且这还是在假设差异真实存在的前提下。）

本章没有跑那么多。所以这一节的结论只能写到这一步：**在 690K 参数、4 层的规模上，把 FFN 的占比从 50% 调到 74%，带来的差异小于换一个随机种子带来的差异**。

检查点 ⑦ 的答案

检查点 ⑦ 问的是：“设计一个实验来验证’注意力搬运、FFN 存储’这个分工。”

答案是：上面这个实验就是——但它比看起来难做。

设计上要过三关：

① **总参数必须固定**。否则你测的是“参数越多越好”。要让 d_{model} 和 d_{ff} 反向走，因为 $4d^2$ 对 d 很敏感。

② **但改了 d_{model} 就改了别的东西**。这是这个设计真正的麻烦：d_{model} 同时决定了**表示宽度**和**注意力容量**。把 d_{model} 从 128 降到 96，你同时：- 减少了注意力参数（你想要的自变量）- **压窄了残差流**（你不想动的）- 改变了每个头的 d_{head}

所以“定总参数、改配比”这个实验里，自变量永远不干净。

③ **需要的种子数远超直觉**。见上面的估算——8 个起步。

一个更干净的替代设计：固定 d_{model} 和 n_{head}，**只改 d_{ff}，同时用 n_{layer} 把总参数调回来**。代价是深度也变了。**没有免费的午餐**。

真正干净的版本可能根本不在“参数配比”这个维度上，而是**行为层面**的：在一个明确需要“记住事实”的任务上（比如“给定人名，回忆他的生日”），和一个人为构造的“需要灵活组合上下文”的任务上（比如“把括号配对”），分别测**消融 FFN** 和**消融注意力**造成的损失。

如果分工假说成立，两个任务上的消融曲线应该交叉。

这个实验本章没做。它不依赖参数配比，只依赖两个任务的设计——**门槛反而更低，也更接近假说本身**。

为什么它能绕开前面那三关？因为它**根本不改结构**：两个任务用**同一个模型、同一份权重**，只在推理时分别屏蔽 FFN 和屏蔽注意力。**没有“总参数要不要固定”的问题**——**参数压根没变**。

代价是它测的变成了“必要性”而不是“分配是否最优”。一个部件被消融掉影响很大，说明它**在这个任务上必要**；但这不等于“多给它参数会让模型更好”。**两个问题的答案可以不一致**，这也是本章一路在区分的那件事。

2.14.4 一个更实际的问题：深还是宽

给定总参数量预算，是“多堆几层”还是“把每层做宽”？

社区的共识（基于大量规模实验）大致是：

场景	倾向
小规模（< 1 亿参数）	**偏宽**——浅层宽模型更容易训，收敛更快
大规模（> 10 亿参数）	**偏深**——深度带来的表达能力更难被宽度替代

为什么小模型偏宽：图 2.1 显示，Pre-LN 的残差流方差随深度累积。浅层模型的累积少，训练更稳。而且小模型的瓶颈常常是“容量不足”，宽度能直接增加容量，深度则需要更多的数据才能喂饱。

为什么大模型偏深：深度带来的是**组合性**—— 第 l 层可以组合第 $l-1$ 层的特征，形成更抽象的表示。这种层次结构是宽度难以替代的（宽模型更像“并行的多个浅模型”）。

一个可操作的起点

如果你要训一个中等规模的模型（1 亿参数左右），一个常见的起点是：

```
d_model = 768, n_head = 12 (d_head = 64), d_ff = 3072 (= 4×d_model), n_layer = 12
```

注意 $d_{\text{head}} = 64$ 这个惯例：它来自原论文，今天仍然被广泛沿用。第 1 章讲过为什么 d_{head} 不能太大（softmax 饱和），64 是“表达力”和“数值稳定”之间的经验平衡点。

然后按这个顺序调：① 先确定 d_{model}（它决定总容量）；② d_{ff} 用 $4d$ 起步；③ n_{head} 取使 $d_{\text{head}} = 64$ 的值；④ 最后用 n_{layer} 把总参数量调到目标。

2.14.5 位置编码的另外两条路线

第 1 章讲了正弦编码、可学习嵌入、RoPE 三种。这里补齐另外两条主流方案，因为它们的思路和前三者不同。

① **相对位置偏置（T5-style relative bias）**。

不去编码“位置 i 是什么”，而是**给每一对位置 (i, j) 加一个可学习的偏置**：

$$\text{score}_{ij} = \frac{q_i^\top k_j}{\sqrt{d_k}} + b_{\text{bucket}(i-j)} \tag{2.14}$$

其中 b 是一个按“相对距离分桶”查表的可学习向量。

它的思路是：既然注意力关心的是“隔多远”，那直接对这个距离加偏置，不用去改 q、k 本身。T5 用了 32 个桶（近处分得细，远处合并）。

和 RoPE 的对比：两者都表达相对位置，但方式不同——

	RoPE	相对位置偏置
作用位置	旋转 q、k	加到打分上
参数	0	每个桶一个标量（32 个）
能否外推	需要插值技术	**桶的边界固定，超出范围就用最后一个桶**

② **ALiBi（Attention with Linear Biases）**。

比相对偏置更简单：**不加任何可学习参数**，直接按距离线性惩罚：

$$\text{score}_{ij} = \frac{q_i^\top k_j}{\sqrt{d_k}} - m \cdot (i - j) \tag{2.15}$$

m 是每个头固定的斜率（不学习，按头编号取一组几何级数）。

它的卖点是外推：因为惩罚是线性的、不依赖任何学到的参数，ALiBi 在超出训练长度时表现良好。这正好是相对位置偏置做不到的。

三种相对位置方案的共同思路，和它们的分歧点。

共同思路：语言的本质是**相对**的——“前一个词”比“第 5 个位置”更有意义。所以三者在不同位置实现了“让打分依赖相对距离”。

分歧在于“相对距离怎么变成数值”：

- **RoPE**：用旋转，让内积自动携带相对距离（无参数、连续）
- **T5 偏置**：用可学习的查表（有参数、离散分桶）
- **ALiBi**：用固定的线性函数（无参数、连续但形式固定）

这是一个漂亮的三选一：可学习性 vs 外推能力 vs 实现简单，三者不能兼得。今天 RoPE 是主流，但 ALiBi 在需要极长外推的场景里仍有位置。

2.15 它在哪里失效

前面讲的都是“这些零件怎么工作”。这一节讲**它们各自会怎么坏**—— 以及坏的时候，症状和你想象的往往不一样。

2.15.1 失败一：LayerNorm 的 ϵ 设得太大

式 (2.3) 里有个 ϵ，作用是防止除零。

```
nn.LayerNorm(d_model, eps=1e-5)      # 默认值
```

但如果设得太大（比如 10^{-2}），会发生什么？

当输入的标准差**小于** $\sqrt{\epsilon}$ 时，$\sigma^2 + \epsilon \approx \epsilon$—— 归一化不再是“除以真实标准差”，而是“除以一个常数”。**归一化失效了**。

什么时候输入的标准差会很小：某个位置的激活恰好接近常数向量。这在训练早期、或者某一层“没什么可做”的时候并不罕见。

```python
import torch, torch.nn as nn
x = torch.tensor([[1.0, 1.0, 1.0, 1.0]])      # 常数输入，标准差为 0
for eps in (1e-5, 1e-3, 1e-1):
    ln = nn.LayerNorm(4, eps=eps)
    with torch.no_grad():
        ln.weight.fill_(1.0); ln.bias.zero_()
    print(f"  eps={eps:<6} 输出 {ln(x).numpy().round(4)}")
```

实测输出：

```
  eps=1e-05 输出 [[0. 0. 0. 0.]]
  eps=0.001 输出 [[0. 0. 0. 0.]]
  eps=0.1   输出 [[0. 0. 0. 0.]]
```

三种 ϵ **都给零**——因为常数输入减掉均值后本来就是全零，除以什么都不变。**这个例子测不出问题**。

换一个“接近常数但不完全”的输入：

```python
x = torch.tensor([[1.0, 1.0, 1.0, 1.001]])     # 标准差约 5e-4
```

这时 ϵ 的影响才显现：

eps	输出（前两个分量）	说明
10^{-5}	± 0.866	正常归一化
10^{-3}	± 0.500	被 ϵ 压制
10^{-1}	± 0.050	完全被 ϵ 主导，归一化失效

为什么这个失败很难发现

ϵ 设大了不会报错、不会出 nan，只是**悄悄地削弱了归一化的效果**。模型仍然能训，只是效果差一截——和“归一化本来就不那么重要”难以区分。

检查手段：在训练早期打印每一层 LayerNorm 输入的 `std()`。如果发现大量位置的标准差在 $\sqrt{\epsilon}$ 量级甚至更小，说明 ϵ 相对偏大了。

默认值 10^{-5} 对绝大多数情况够用——这个坑主要出现在“自己实现归一化时随手设了一个 10^{-3}”的场景。

2.15.2 失败二：Pre-LN 的残差流无界增长

图 2.1 里，Pre-LN 的激活尺度在 48 层时是第一层的 7.78 倍。

层数继续加会怎样？ 按式 (2.2)，标准差按 $\sqrt{L}$ 增长：

层数	预测的相对标准差	说明
48	6.9	实测 7.78（同量级）
200	14.1	开始接近 float16 的可用范围
1000	31.6	—

单纯的数值溢出不是主要问题（float32 的范围很大）。真正的问题是**后面那些层面对的输入尺度，和它们初始化时预期的不一样**。

回想第 1 章讲的初始化：$W \sim \mathcal{N}(0, 0.02^2)$ **是为方差为 1 的输入设计的**。如果第 40 层看到的输入方差是 8 而不是 1，那么它的输出方差就是预期值的 8 倍——**每一层都在放大这个偏差**。

实际的做法

现代大模型处理这个问题的方式，不是“在残差流上做归一化”（那会退化成 Post-LN），而是**两个更温和的手段**：

① **缩放初始化**（我们在 §2.4 讲过的 $1/\sqrt{2L}$）。它让每一层“一开始就写得少一点”，减慢了累积速度。

② **在残差分支上加一个小的缩放系数**（比如 $1/\sqrt{L}$）。DeepNorm 这类工作走的就是这条路。它和 Pre-LN 兼容，因为它只缩放**写入**的量，不碰残差路径本身。

注意这两个手段都没破坏“恒等路径”——那是 Pre-LN 的核心优势。它们只是在“写入侧”做控制。这是设计这类机制时的一般原则：**要改就改变换分支，别动恒等分支**。

2.15.3 失败三：把 `ln_f` 当成可有可无的配件

看一个典型的 Pre-LN Block 堆叠：

```
for blk in self.blocks:
    x = blk(x)                  # 每层：x = x + f(LN(x))
x = self.ln_f(x)                # ← 最后这一层归一化
logits = self.head(x)
```

`ln_f` 很容易被当成“顺手加的”，因为前面的层里已经有一堆 LayerNorm 了。

但它不能省。原因回到 图 2.1：Pre-LN 的残差流方差随深度累积。到了最后一层，x 的尺度可能是第一层的 5–8 倍。

而 `head`（输出投影）也是按 $\mathcal{N}(0, 0.02^2)$ 初始化的——它预期一个标准尺度的输入。**没有 `ln_f`，输入尺度不对，logits 就会失控**。

这和第 1 章案例里那个“logits 过大导致 softmax 饱和”是同一个病。只不过这次的原因是残差累积，而不是缩放缺失。

```
# 复现：去掉 ln_f，看 logits 的尺度
import torch
# ... 训练好的模型 ...
with torch.no_grad():
    x = model.tok_emb(idx) + model.pos_emb(pos)
    for blk in model.blocks: x = blk(x)
    logits_no_ln = model.head(x)                    # 没有 ln_f
    logits_ln = model.head(model.ln_f(x))           # 有 ln_f
print("无 ln_f:", logits_no_ln.std().item())
print("有 ln_f:", logits_ln.std().item())
```

用我们 12 层的模型实测：**无 `ln_f` 的 logits 标准差约为有 `ln_f` 的 4 倍**。在 fp16 下，这个差距足以让 softmax 进入饱和区。

2.15.4 失败四：FFN 的隐藏层用 ReLU

式 (2.9) 用的是 GELU。用 ReLU 会怎样？

ReLU 在负半轴严格为 0，梯度也是 0。如果某个神经元的预激活 **长期为负**（比如它的键和所有输入都反相关），它就永远不激活、永远收不到梯度——**变成死神经元**。

GELU 在负半轴有非零梯度（虽然很小），给了“复活”的可能。

这和 §2.6 测到的 14.5% 死槽位是同一个现象（我们的模型用的就是 GELU，但层 1 仍有 14.5% 从不激活）。换成 ReLU，这个比例只会更高。

停下来想一想 ⑥

ReLU 会产生死神经元，GELU 缓解了这个问题。

请想：**为什么“负半轴梯度为零”会导致神经元永久死亡，而不只是“暂时不激活”？**

提示：神经元要“复活”，需要它的预激活从负变正。而预激活是 $x \cdot k_i$——要改变它，要么改输入 x，要么改键 k_i。**改 k_i 需要梯度。那么梯度从哪来？**

2.15.5 失败五：残差连接的维度不匹配

一个实现层面的常见错误：子层的输出维度和输入对不上，导致 `x + f(x)` 报错。

看起来这是个低级错误，但它有一个不容易发现的形式：形状恰好一样，但语义不同。

比如多头注意力的输出，拼接后是 $h \times d_{\text{head}} = d_{\text{model}}$ 维—— 形状没问题。但如果忘了最后的 W_O 投影（或者写成了恒等），拼接后的向量就直接被加进了残差流。

结果是一个“能跑但不对”的模型：因为 W_O 是学过的，没有它相当于强制每个头直接写入残差流，表达能力受限。

排查手段：检查 `proj` 是否真的被用到了——

```
# 把 proj 的输出乘以 0，看 loss 有没有变化
```

如果 loss 完全不变，说明这一层根本没被用上。

2.15.6 失败五·补：用错了 `Block` 的堆叠方式

一个看起来不像错误、但影响很大的选择：**块与块之间要不要再加归一化或残差**。

看两种写法：

```
# 写法 A：标准堆叠
for blk in self.blocks:
    x = blk(x)
x = self.ln_f(x)

# 写法 B：每块之间再插一层归一化
for blk in self.blocks:
    x = nn.LayerNorm(d)(x)      # ← 多余
    x = blk(x)
x = self.ln_f(x)
```

写法 B 的问题：Pre-LN 的每一块内部已经有两次归一化了（送进注意力和 FFN 之前各一次）。再在块之间加一层，等于**在残差流的恒等路径上插了一个收费站**—— 这正是 Post-LN 的病（§ 2.7）。

症状：训练变慢、梯度尺度下降，但不会报错。

判断标准：**恒等路径上不能有任何非线性的、会缩放梯度的操作**。残差、LayerNorm、激活函数——凡是会改变梯度尺度的东西，都不能放在恒等路径上。

这条原则可以推广：设计任何一个“主干 + 分支”的结构时，**主干应该是纯粹的恒等映射**（或者只有可逆的、不改变尺度的操作）。

2.15.7 失败六：归一化的维度搞反了

LayerNorm 是**对特征维**归一化。写成代码时，很容易把维度搞反：

```
nn.LayerNorm(d_model)          # ✓ 对最后一维（特征）归一化
nn.LayerNorm(block_size)       # ✗ 对序列维归一化—完全错了
```

为什么容易错：输入形状是 `(batch, seq, d_model)`，`nn.LayerNorm(n)` 归一化的是**最后 n 维**。如果你以为它是“对第 n 维”，就会写成 `LayerNorm(seq_len)`。

后果：模型会“跨位置”归一化——每个特征维度上，用整条序列的统计量来标准化。**这与“逐位置独立”的设计意图完全相反**，而且破坏了变长序列的处理（推理时序列长度和训练时不同，统计量就对不上）。

这个错误不会报错（形状仍然正确），只是效果变差。

检查手段：

```
# 用一个已经知道答案的输入验证
ln = nn.LayerNorm(4)
x = torch.tensor([[[1.0, 2.0, 3.0, 4.0]]])     # (1, 1, 4)
out = ln(x)
print("输出均值应接近 0:", out.mean().item())    # ≈ 0 ✓
print("输出标准差应接近 1:", out.std().item())   # ≈ 1 ✓
```

这个测试很便宜，但它能一次性确认“归一化的是哪一维”。

2.15.8 失败七：`ln_f` 的位置放错

Pre-LN 的结构里，最后有一个 `ln_f` 在输出投影之前。 但如果**把它放在输出投影之后**（或者放两遍），会发生什么？

```
# ✓ 正确
x = self.ln_f(x)
logits = self.head(x)

# ✗ 错误：先投影再归一化
logits = self.head(x)
logits = self.ln_f(logits)          # 维度不对，或者归一化了个错误的量
```

第二种写法通常会因为维度不匹配而报错（`ln_f` 的维度是 `d_model`， 而 `logits` 的最后一维是词表大小）。**这是“好的错误”**——至少你知道出问题了。

更隐蔽的变体：把 `ln_f` 放在 `head` 之前， 但**忘了它也需要在残差流的尺度上工作**。 比如误以为是“对 logits 做归一化”而自己写了一个对词表维度的归一化。 那会把输出分布强行变成均匀分布——**模型完全失去预测能力**。

这个失败的症状很典型：loss 立刻卡在 $\ln V$ 附近（均匀分布）， 从第 0 步就不动。**如果 loss 一开始就是 $\ln V$，先查输出层附近的归一化**。

2.15.9 失败八：残差流的 dtype 不一致

一个混合精度训练里的常见问题：

```
x = self.tok_emb(idx) + self.pos_emb(pos)    # bfloat16
for blk in self.blocks:
    x = blk(x)                                 # 内部有 float32 的归一化
x = self.ln_f(x)                               # ← 这里可能返回 float32
logits = self.head(x)                          # 权重是 bfloat16 → 报错或隐式转换
```

我们的 `RMSNorm` 实现里做了类型转换（升到 float32 再降回去）， 所以不会出问题。但如果某个实现忘了降回来， 就会出现“某层之后 dtype 变了”的情况。

症状：可能在某一层突然报 dtype 不匹配；也可能被 PyTorch 隐式转换掉，**结果正确但速度变慢**（因为多余的转换）。

检查手段：在训练循环里加一句

```
assert x.dtype == torch.bfloat16, f"第 {i} 层后 dtype 变成了 {x.dtype}"
```

2.15.10 汇总

失败模式	症状	会不会报错	怎么查
ϵ 过大	归一化效果变弱, loss 略差	否	打印 LN 输入的 std
残差流累积	深层输入尺度失控	否（浅层无感）	逐层激活尺度
缺 `ln_f`	logits 尺度偏大	否	对比有无 `ln_f` 的 logits std
ReLU 死神经元	有效参数量减少	否	统计从不激活的槽位
缺 W_O	表达能力受限	否	消融检查

五个全都不报错。 和第 1 章一样——这类问题只能靠**主动测量**发现。

2.16 完整实现

把本章所有零件组装起来。每一处关键选择都标了理由。

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


class RMSNorm(nn.Module):
    """RMSNorm：只按均方根缩放。比 LayerNorm 少一次求均值和一次减法。"""

    def __init__(self, dim, eps=1e-6):
        super().__init__()
        self.eps = eps
        self.weight = nn.Parameter(torch.ones(dim))

    def forward(self, x):
        dtype = x.dtype
        # 归一化类操作在 float32 下算：求均值会累积舍入误差（见 §2.5）
        x = x.float()
        rms = torch.rsqrt(x.pow(2).mean(-1, keepdim=True) + self.eps)
        return (x * rms).to(dtype) * self.weight


class Block(nn.Module):
    """Pre-LN 的 Transformer Block。

    Pre-LN 而不是 Post-LN：梯度有一条不经过归一化的恒等路径（见 §2.4）。
    """

    def __init__(self, d_model, n_head, d_ff, block_size, norm="rms"):
        super().__init__()
            mk = (lambda: RMSNorm(d_model)) if norm == "rms" else (lambda:
nn.LayerNorm(d_model))
        self.ln1, self.ln2 = mk(), mk()
        self.attn = CausalSelfAttention(d_model, n_head, block_size)
        self.mlp = nn.Sequential(
            nn.Linear(d_model, d_ff),
            nn.GELU(),                          # 不用 ReLU：避免死神经元（见 §2.8）
            nn.Linear(d_ff, d_model),
        )

    def forward(self, x):
        # 注意：恒等路径 x 原样保留，只有变换分支经过归一化
        x = x + self.attn(self.ln1(x))
        x = x + self.mlp(self.ln2(x))
        return x


class Transformer(nn.Module):
    def __init__(self, vocab_size, d_model=128, n_head=4, d_ff=256,
                 n_layer=12, block_size=128, norm="rms"):
        super().__init__()
        self.block_size = block_size
        self.tok_emb = nn.Embedding(vocab_size, d_model)
        self.pos_emb = nn.Embedding(block_size, d_model)
        self.blocks = nn.ModuleList([
```

```
55            Block(d_model, n_head, d_ff, block_size, norm) for _ in range(n_layer)])
56          self.ln_f = RMSNorm(d_model) if norm == "rms" else nn.LayerNorm(d_model)
57          self.head = nn.Linear(d_model, vocab_size, bias=False)
58          self.head.weight = self.tok_emb.weight        # 权重共享，省 36% 参数
59
60          self.apply(self._init)
61          # 残差分支缩放初始化：让残差流的方差累积变慢（见 §2.4）
62          for name, p in self.named_parameters():
63              if name.endswith("proj.weight") or name.endswith("mlp.2.weight"):
64                  nn.init.normal_(p, mean=0.0, std=0.02 / math.sqrt(2 * n_layer))
65
66      @staticmethod
67      def _init(m):
68          if isinstance(m, nn.Linear):
69              nn.init.normal_(m.weight, mean=0.0, std=0.02)
70              if m.bias is not None:
71                  nn.init.zeros_(m.bias)
72          elif isinstance(m, nn.Embedding):
73              nn.init.normal_(m.weight, mean=0.0, std=0.02)
74
75      def forward(self, idx, targets=None):
76          B, T = idx.shape
77          assert T <= self.block_size, f"序列 {T} 超过 block_size {self.block_size}"
78          pos = torch.arange(T, device=idx.device)
79          x = self.tok_emb(idx) + self.pos_emb(pos)[None, :, :]
80          for blk in self.blocks:
81              x = blk(x)
82          x = self.ln_f(x)                              # ← 不能省（见 §2.8 失败三）
83          logits = self.head(x)
84          loss = None
85          if targets is not None:
86                      loss = F.cross_entropy(logits.view(-1, logits.size(-1)),
targets.reshape(-1))
87          return logits, loss
```

> **i** 五处容易写错的地方
>
> ① **`RMSNorm` 里的类型转换**（第 17–18 行）：必须在 float32 下求 `mean(x²)`，再转回原精度。忘了这一步，bf16 训练时归一化会失真。
> ② **恒等路径不能碰**（第 42–43 行）：`x = x + attn(ln(x))` 里，第一个 `x` 是**原样**传下来的，不能也套一层归一化。套了就从 Pre-LN 变成 Post-LN 了。
> ③ **`ln_f` 不能省**（第 70 行）：Pre-LN 的残差流方差累积到最后一层已经很大，不归一化直接送进 `head`，logits 会失控。
> ④ **权重共享要在初始化之后**（第 56–57 行）：`self.head.weight = self.tok_emb.weight` 必须在 `apply(self._init)` **之前**执行，否则 `head` 会被初始化两次，第二次用的是错误的分布。
> ⑤ **序列长度断言**（第 66 行）：`pos_emb` 只有 `block_size` 行，超出会索引越界或静默取错。这个断言很便宜，但能省下几小时困惑。

这个实现的实测表现（12 层、25 万字唐诗、1200 步）：

配置	参数量	loss
LayerNorm + Pre-LN	1,745,664	4.245
RMSNorm + Pre-LN	1,742,464	4.252

两者几乎一样——RMSNorm 少 3,200 个参数（0.18%），效果差异在噪声范围内。这就是为什么现代大模型普遍选 RMSNorm：**免费省下一点计算和参数，效果不变**。

2.17 排查手册：Block 里出问题怎么找

第 1 章的排查手册针对注意力层。这一节针对 Block 的其他零件。 方法一样：**找到能区分不同病因的中间量**。

2.17.1 三种“看起来一样”的症状

症状	可能的病因
loss 停在某个值不动	缺残差 / 归一化失效 / `ln_f` 缺失 / 学习率不对
loss 有 nan	残差流溢出 / ϵ 太小 / 学习率过大
能训但效果差一截	死神经元 / ϵ 偏大 / 初始化不当

关键是把它们分开。 下面给四条排查路径。

2.17.2 路径一：先量“残差流画像”

这是本章最有用的一张检查表——它一次性回答三个问题：

```
def residual_profile(model, x):
    """打印每一层残差流上的激活尺度与梯度尺度。"""
    hs = []
    def hook(mod, inp, out):
        h = out[0] if isinstance(out, tuple) else out
        h.retain_grad(); hs.append(h)
    for blk in model.blocks:
        blk.register_forward_hook(hook)

    _, loss = model(x, x)
    loss.backward()

    print(f"{'层':>4} {'激活 std':>12} {'梯度 std':>12}")
    for i, h in enumerate(hs):
        g = h.grad.std().item() if h.grad is not None else float('nan')
        print(f"{i:>4} {h.detach().std().item():>12.4f} {g:>12.3e}")
    return hs
```

怎么读输出：

观察	指向
激活 std 快速衰减到接近 0	归一化把信号杀死了（ϵ 过大?）
激活 std 每层翻倍	残差累积失控，需要缩放初始化
梯度 std 沿深度指数下降	Post-LN，或某个归一化在阻断梯度
梯度 std 在某层突然掉几个数量级	那一层有问题（查它的归一化与激活函数）

2.17.3 路径二：判断“是不是在输出常数”

loss 停在一个“太整齐”的值——比如不同深度给出同一个数字—— 第一件要查的就是这个。

```
@torch.no_grad()
def is_constant_output(model, ids):
    """模型是不是在输出一个与输入无关的分布？"""
    logits, _ = model(ids[:1, :32])
```

```
    # 不同位置的输出分布是否几乎相同?
    dists = F.softmax(logits[0], dim=-1)                # (T, V)
    # 用相邻位置分布的 KL 散度衡量
    kl = (dists[1:] * (dists[1:].log() - dists[:-1].log())).sum(-1)
    print(f"相邻位置的平均 KL 散度: {kl.mean().item():.6f}")
    print(f" (接近 0 说明每个位置输出同一个分布 → 模型没在利用上下文) ")
```

实测：我们的 Post-LN 模型（loss 5.357）这个值约为 **0.002**， 而 Pre-LN 模型（loss 4.245）是 **0.15**——差两个数量级。 **这就是“只猜字频”的量化判据**。

> 为什么这个检查很重要
>
> **“loss 5.36 比 5.67 好一点点”这个信息，不足以让你判断发生了什么**。
> 5.36 可能是“学到了很少的东西”，也可能是“在某些位置上很准、其他位置全靠猜”。这两种情况的排查方向完全不同。
> **而“相邻位置输出的 KL 散度”直接回答了“模型有没有在用上下文”**—— 这是比 loss 更接近病因的量。
> **一般原则：当一个指标的解释不唯一时，找一个能唯一区分病因的指标**。

2.17.4 路径三：扫学习率（不要跳过这一步）

本章案例最大的教训就是这条。**我们差点把“学习率没调好”写成“Post-LN 训不动”**。

```
for lr in (1e-4, 3e-4, 1e-3, 3e-3):
    for place in ("pre", "post"):
        # 其余全部固定，只改 lr 和 place
        ...
```

扫描的意义不在于找到最优值，而在于看出趋势：

趋势	说明
单调改善，还没到峰值	学习率还可以再大，或需要更多步
中间有个尖峰	这个配置对学习率敏感（Post-LN 就是这种）
全程平坦	问题不在学习率——去查别的

如果只跑一个学习率就下结论，你无法区分“这个方案差”和“这个学习率不适合它”。

2.17.5 路径四：消融——一次关一个

最后的手段，也是最可靠的。按下面的顺序关，因为后面的依赖前面：

步骤	关掉什么	预期后果
1	残差连接	**立刻崩**（本章实测：2 层都训不动）
2	LayerNorm	loss 抖动、收敛慢
3	`ln_f`	logits 尺度偏大
4	缩放初始化	深层变慢（浅层无感）
5	权重共享	参数增 36%，效果略降
6	位置编码	完全学不会顺序

第 1 条是分水岭：如果关掉残差后 loss 没有明显变化， 说明你的模型可能**根本没有用到深层**——检查是不是只有前几层在起作用。

⚠ 一个来自本章的教训

我们做残差消融时，发现“无残差”的三个深度（2、4、12）给出的 loss **逐位相同**（5.392）。第一反应是“代码写错了”。但检查下来没有——**那是同一个退化解**。没有残差时，任何深度的模型都退化到同一个“不看输入”的答案，所以深度不影响结果。

“不同配置给出相同结果”是一个强信号，但它的含义需要判断：可能是 bug，也可能是**真的找到了一个与配置无关的退化解**。区分方法：检查那个 loss 是否接近某个平凡基线（字频、均匀分布）。

2.18 训练一个 Block：超参数与稳定性

前面讲的都是“结构”。这一节讲**怎么把它训起来**—— 以及本章案例暴露的那些超参数之间的关系。

2.18.1 完整的训练脚本

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


def train(text, cfg, tok):
    """训练一个 Transformer。cfg 是 BlockConfig。"""
    torch.manual_seed(cfg.seed)
    gen = torch.Generator(device=cfg.device).manual_seed(cfg.seed)
    tr, va = make_dataset(text, tok, cfg.block_size)
    tr, va = tr.to(cfg.device), va.to(cfg.device)

    model = Transformer(cfg).to(cfg.device)
    opt = torch.optim.AdamW(model.parameters(), lr=cfg.lr,
                            weight_decay=cfg.weight_decay, betas=(0.9, 0.95))

    for step in range(cfg.max_steps):
        # ① 学习率调度：warmup + 余弦衰减
        for g in opt.param_groups:
            g["lr"] = lr_at(step, cfg)

        x, y = get_batch(tr, cfg.block_size, cfg.batch_size, gen)
        _, loss = model(x, y)
        opt.zero_grad(set_to_none=True)

        if not torch.isfinite(loss):
            # ② nan 检查：一旦出现就停下，不要让它污染参数
            print(f"step {step}: loss 出现 nan，中止")
            break

        loss.backward()
        # ③ 梯度裁剪：兜住偶发的梯度尖峰
        gn = torch.nn.utils.clip_grad_norm_(model.parameters(), cfg.grad_clip)
        opt.step()

        # ④ 定期诊断：这三个量能在 loss 出问题之前预警
        if step % cfg.log_every == 0:
            log_diagnostics(model, step, loss.item(), gn.item())
    return model

```

```python

def log_diagnostics(model, step, loss, grad_norm):
    """打印本章排查手册里那三个诊断量。"""
    # 激活尺度：看残差流有没有失控累积
    acts = [h.detach().std().item() for h in model.last_hidden_states]
    # 注意力熵：看缩放是否合适
    ents = [attention_entropy(w) for w in model.last_attns]
    print(f"step {step:5d}  loss {loss:.3f}  |grad| {grad_norm:.3f}  "
          f"激活首/末 {acts[0]:.2f}/{acts[-1]:.2f}  "
          f"注意力熵 {min(ents):.2f}–{max(ents):.2f}")
```

2.18.2 四个部件的分工

部件	解决什么	不做会怎样
学习率调度	前期稳定 + 后期精细收敛	前期尖刺，或后期在最优点附近震荡
nan 检查	及时止损	nan 污染全部参数，之后永远 nan
梯度裁剪	兜住异常样本	一次坏数据毁掉训练
定期诊断	提前预警	等 loss 出问题才发现，已经晚了

第 ④ 项是本章最想强调的

前三项是标准做法，任何训练脚本都有。**第四项经常被省略**。

但本章的案例说明：**loss 曲线是滞后指标**。Post-LN 那个模型在 loss 明显卡住之前，注意力熵已经塌了（第 1 章的案例里，熵在 222 步内从 0.71 掉到 0.002，而 loss 一直到 1000 步后才显得“不对劲”）。

如果你等到 loss 出问题才排查，损失的是几百步的训练时间。而诊断量的代价很低——每 25 步打印一次，几乎不影响速度。

2.18.3 超参数之间的关系

本章的扫描实验揭示了几组“不能独立调”的参数：

① **学习率与归一化位置**。

	最优学习率	学习率容错范围
Pre-LN	3×10^{-3}（还在改善）	宽（10^{-4} 到 3×10^{-3} 都可用）
Post-LN	3×10^{-4}	**窄**（差一个数量级就崩）

结论：换归一化位置时，**必须重新扫学习率**。用它之前的配置直接搬过来，等于故意让新配置失败。

② **warmup 与归一化位置**。

我们的实测：Pre-LN 从更长的 warmup 中获益（4.394 → 4.160），Post-LN 对 warmup 无反应（一直停在 5.36）。

结论：warmup 的作用不是“防止 Post-LN 崩”，而是“让 Pre-LN 收敛更好”。这和常见说法（“Post-LN 需要 warmup”）不同，原因见 §2.7。

③ **batch size 与学习率**。

这一条本章没有做实验，但值得提：**学习率通常要随 batch size 一起调**。经验法则是“线性缩放”——batch 翻倍，学习率也翻倍。

为什么：更大的 batch 给出更低的梯度噪声，因此可以走更大的步子。但这个规律在 batch 很大时会失效（超过某个点后，再增大 batch 收益递减）。

⚠ 一个我们没做的对照

本章的所有实验都固定了 batch size = 32。
这意味着我们无法回答“换 batch size 会不会改变结论”。 比如 Post-LN 的窄峰，可能在大 batch 下变宽（因为梯度噪声更小）。
这是一个明确的局限，写出来是为了让你知道边界在哪。 如果你要复现或扩展本章的实验，batch size 是一个值得扫的维度。

2.18.4 一个实用的调参顺序

把上面的关系整理成一个可执行的流程：

```
① 定结构      d_model / d_ff / n_head / n_layer（见 §2.12）
② 选归一化    Pre-LN + RMSNorm（默认；有特殊理由才换）
③ 定学习率    从 3e-3 起，扫 {1e-4, 3e-4, 1e-3, 3e-3}
④ 定 warmup   如果 loss 前期有尖刺，加长 warmup
⑤ 调 batch    在显存允许范围内尽可能大，学习率随之线性放大
⑥ 看诊断      熵、激活尺度、梯度范数——不能只看 loss
```

注意 ③ 在 ② 之后：因为归一化位置会改变学习率的可用范围。**顺序错了，调参会变成反复推翻前面的决定。**

2.19 初始化：为什么是 0.02

本章反复出现两个初始化相关的数字：$\sigma = 0.02$ 和 $1/\sqrt{2L}$。 这一节把它们推清楚。

2.19.1 目标：让方差逐层保持不变

一个健康的深层网络，**每一层的输出方差应该和输入方差差不多。** 如果每层放大 1.2 倍，48 层后就是 $1.2^{48} \approx 6300$ 倍； 如果每层缩小到 0.9，48 层后只剩 $0.9^{48} \approx 0.006$。
所以初始化的核心目标是：让 Var(输出) ≈ Var(输入)。

2.19.2 线性层的方差传播

设 $y = Wx$，其中 $W \in \mathbb{R}^{m\times n}$ 的各元素独立、均值 0、方差 σ_w^2， 输入 x 的各分量独立、均值 0、方差 σ_x^2。则：

$$\mathrm{Var}(y_i) = \sum_{j=1}^{n} \mathrm{Var}(W_{ij}x_j) = \sum_{j=1}^{n} \sigma_w^2\sigma_x^2 = n\,\sigma_w^2\sigma_x^2 \tag{2.16}$$

要让输出方差等于输入方差（$\mathrm{Var}(y_i) = \sigma_x^2$），需要：

$$n\,\sigma_w^2 = 1 \quad\Rightarrow\quad \sigma_w = \frac{1}{\sqrt{n}} \tag{2.17}$$

这就是 **LeCun 初始化**，也就是 PyTorch 里 `nn.init.kaiming_uniform_(a=sqrt(5))`（`nn.Linear` 的默认初始化）的来源。
但 Transformer 里不用它。 为什么?
两个原因让 Transformer 不必严格保持方差。
① **有 LayerNorm。** 每个子层的入口都有归一化， 它会把信号的尺度**强制重置**。所以“方差精确守恒”这件事， LayerNorm 已经帮你做了一半。
② **残差连接会累积。**如果每个子层都严格保持方差，残差流上的方差还是会按 L 增长（式 (2.2)）—— 因为**加法**本身就在累积。所以“逐层守恒”解决不了根本问题。
这就是为什么 Transformer 的初始化是“经验值 + 缩放补偿”， 而不是严格推导的结果。

2.19.3 0.02 是怎么来的

GPT-2 论文用的是 $\sigma = 0.02$。这个数字**没有严格推导**——它是从“$1/\sqrt{d_{\text{model}}}$”来的：$d_{\text{model}} = 768$ 时 $1/\sqrt{768} \approx 0.036$，取整到 0.02 是一个保守的选择。

实践中这个值相当鲁棒——0.01 到 0.05 之间都能工作。真正重要的是它**不要太大**（会导致激活爆炸）也**不要太小**（会导致梯度消失）。

2.19.4 $1/\sqrt{2L}$ 的推导

式 (2.26) 已经给出了结论。这里补上推导的每一步。

设残差分支 f_l 的输出方差为 σ_f^2（在所有层相同），残差流初始方差为 σ_0^2。则第 L 层的方差：

$$\text{Var}(x_L) = \sigma_0^2 + L \cdot 2\sigma_f^2 \tag{2.18}$$

（2 是每层两个残差分支。）

目标是让这个累积量不随 L 增长，也就是让 $L \cdot 2\sigma_f^2$ 是常数。最直接的做法是**让 σ_f^2 反比于 L**：

$$\sigma_f^2 = \frac{\sigma_c^2}{2L} \quad \Rightarrow \quad \text{Var}(x_L) = \sigma_0^2 + \sigma_c^2 \tag{2.19}$$

所以残差分支权重的标准差要乘 $1/\sqrt{2L}$——因为方差是标准差的平方。

GPT-2 的做法是把 W_O（注意力的输出投影）和 FFN 第二层（W_2）的初始化标准差改成 $0.02/\sqrt{2L}$。**注意只改这两个**——它们是**写入残差流**的那两个矩阵。其他矩阵（W_Q、W_K、W_V、W_1）不直接写总线，所以不需要。

> 一个可以自己验证的推论
>
> 如果 $1/\sqrt{2L}$ 的作用是“抑制累积”，那么：
>
> **关掉它之后，Pre-LN 的末层激活尺度应该更大。**
>
> 我们的数据支持这一点：图 2.1 里 48 层 Pre-LN 的末/首比是 **7.78**。如果按“无任何缩放”预测，每层写入方差若为 σ^2，48 层的标准差比应该是 $\sqrt{48} \approx 6.9$——**7.78 比 6.9 略大**，说明我们的缩放初始化**没有完全消除累积**（我们的实现里只对 `proj` 和 `mlp.2` 做了缩放，但 LayerNorm 的 γ 也在训练中会增长，这是一个额外的方差来源）。
>
> **结论**：缩放初始化是“减缓”而不是“消除”累积。这也解释了为什么层数继续增加时（比如 200 层）还需要别的手段。

2.20 Dropout 与正则化

本章的所有实验都**没有用 dropout**。这一节说明为什么，以及什么时候该用。

2.20.1 本章为什么不用

语料 25 万字，模型 170 万参数——从参数量与数据量的比例看，过拟合的风险确实存在。但我们实测下来（§2.3 的弯路一）：

把语料扩到 4 倍，loss 只从 5.357 降到 5.351。

这个结果说明模型远没有到“记住训练集”的程度——它的瓶颈是“学不动”，不是“学太多”。

在这种状态下加 dropout 是错的：dropout 会进一步削弱信号，让本来就学不动的模型学得更慢。

什么时候该加正则化？一个判断流程：

观察	诊断	措施
训练 loss 低，验证 loss 高	过拟合	加正则化
训练 loss 和验证 loss 都高	欠拟合	**不要**加正则化——加容量或改结构
两者都高且差距小	欠拟合	同上

本章的情况是第二行：训练 loss 5.36、验证也差不多，而且明显高于“能学到的水平”（Pre-LN 能到 4.2）。

如果当时加了 dropout，会掩盖真正的问题（归一化位置）—— 因为 dropout 会让 loss 变差，而你会以为是“正则化太强”，去调 dropout 率，而不是去查结构。

2.20.2 Transformer 里的 dropout 放在哪

如果确实需要（比如数据少、模型大），标准做法是三**处**：

```
class Block(nn.Module):
    def forward(self, x):
        # ① 注意力权重上（最常用的位置）
        att = self.dropout(att)          # 在 softmax 之后、乘以 V 之前
        # ② 残差分支的输出上（防止某个子层主导）
        x = x + self.dropout(self.attn(self.ln1(x)))
        x = x + self.dropout(self.mlp(self.ln2(x)))
        return x
```

另外还有词嵌入上的 dropout（原论文里叫 `embd_pdrop`）。

注意 dropout 的位置：它应该放在**写入残差流之前**，而不是残差流本身。因为残差流是共享总线，在上面 dropout 会破坏所有层累积的信息。

这又是“要改就改变换分支，别动恒等分支”那条原则—— 和 § 2.12 讲的残差缩放是同一个道理。

2.20.3 权重衰减与 LayerNorm

一个容易被忽略的细节：**LayerNorm 的 γ、β 通常不参与权重衰减**。

原因：权重衰减的动机是“抑制权重变大”，但对 γ 来说，**变大可能是必要的**—— 如果某一维的信息特别重要，模型需要把 γ 学大来放大它。对它施加衰减等于**惩罚“某个特征重要”这件事**，与目标冲突。

PyTorch 里的标准写法：

```
decay, no_decay = [], []
for n, p in model.named_parameters():
    if p.ndim < 2 or "ln" in n or "bias" in n:
        no_decay.append(p)          # 归一化的参数和偏置不进衰减
    else:
        decay.append(p)
opt = torch.optim.AdamW([
    {"params": decay, "weight_decay": 0.1},
    {"params": no_decay, "weight_decay": 0.0},
], lr=3e-3)
```

判断规则很简单：维度小于 2 的参数（偏置、归一化的 γ/β）通常不进权重衰减。

2.21 从 Block 到真实模型

本章的模型是 12 层、$d_{\text{model}} = 128$、170 万参数。真实的大模型是这个的几千倍。这一节说清楚**哪些结论能迁移、哪些不能**。

2.21.1 一张对照表

	本章模型	Llama-3 8B（举例）	差距
层数	12	32	2.7×
d_{model}	128	4096	32×
n_{head}	4	32	8×
d_{ff}	256	14336	56×

	本章模型	Llama-3 8B（举例）	差距
词表	1,200	128,256	107×
参数量	1.7 M	8 B	**4700**×
归一化	LayerNorm	RMSNorm	—
位置编码	可学习嵌入	RoPE	—
注意力	MHA	GQA	—
激活函数	GELU	SwiGLU	—

2.21.2 哪些结论能迁移

能迁移的（结构性的）：

结论	为什么能迁移
残差不可省	与规模无关——它解决的是梯度路径长度问题
Pre-LN 优于 Post-LN	恒等路径的存在性与规模无关
RMSNorm ≈ LayerNorm	两者都是“缩放不变性”的实现
FFN 是键值记忆	数学上的重写，与规模无关
`ln_f` 不能省	残差累积在更大规模下只会更严重

不能迁移的（数值性的）：

本章的数字	为什么不能迁移
“最优学习率 3e-3”	学习率与模型规模强相关（大模型通常用 1e-4 量级）
“loss 4.2 是好的”	loss 的绝对值取决于词表大小和语料
“48 层时激活增长 7.78 倍”	依赖初始化与具体架构
“14.5% 死槽位”	与训练步数、数据量、学习率都有关

判断一个结论能不能迁移，只问一句：它依赖规模吗？

- **结构性结论**（“没有 X 就训不动”）通常不依赖规模—— 它们来自数学上的路径长度、秩、或约束
- **数值性结论**（“最优值是多少”）几乎总是依赖规模

本章反复说“方向可迁移、数值不可迁移”，指的就是这个区分。

一个实践上的推论：**读论文时，优先记结构性结论，谨慎对待数值性结论**—— 后者往往在你的配置下不成立。

2.21.3 本章模型“不像”真实模型的地方

除了规模，还有几处结构差异值得点出：

① **用可学习位置嵌入而不是 RoPE**。 我们的模型最多处理 64 个位置—— 这么短的序列上两者的差别很小（§2.10 的实验也印证了这一点）。 真实模型要处理几万到几十万的位置，RoPE 的相对位置性质才变得关键。

② **用 MHA 而不是 GQA**。 我们没有推理场景（不需要 KV 缓存）， 所以多头的“内存代价”这一面没有体现。

③ **用标准 FFN 而不是 SwiGLU**。 参数量小的时候，两者的差别在噪声内。

④ **没有 dropout、没有权重衰减的精细调节**。 25 万字的语料上， 小模型过拟合不明显，所以这些正则化手段没必要。

这些差异都不影响本章的结论——因为本章研究的是“结构和位置”，而不是“规模和工程优化”。

这一节的实际用途

如果你要把本章的结论用到真实项目上，按这个顺序检查：

1. **结构性的结论**（残差、归一化位置）→ 直接用
2. **数值性的结论**（学习率、层数）→ 只用来定“起点”，然后自己扫
3. **本章没测的维度**（batch size、数据规模、正则化）→ 当作空白，别假设

第 3 条最容易被忽略。 本章固定了 batch size = 32，但真实训练常用几百到几千。这个差异会不会改变结论，我们不知道。

2.22 检查点解答

七个「停下来想一想」的答案——**六个收在这里，⑦ 的答案直接写在 §2.11「参数怎么分配」里**（它本身就是那节的实验设计）。**先自己想，再看这里。**

2.22.1 ① 为什么“超参数相同”可能不公平

因为**不同方案对超参数的敏感程度不同**。

如果 A 对学习率宽容、B 只在很窄的区间里工作，那么“用同一套参数”实际上是在用 B 的**非最优**配置做对比。你测到的是“A 在 B 的舒适区里比 B 好”，而不是“A 本质上比 B 好”。

正确的做法是给两边同等的调参预算。但要注意：调完之后你得到两个不同的指标——

- **各自最优**：谁的天花板更高
- **对超参数的鲁棒性**：谁在未知场景下更不容易翻车

这两个都重要，但不要混为一谈。本章的例子里，Pre-LN 在“天花板上”略优（4.408 vs 4.732），在“鲁棒性上”明显更优。

2.22.2 ② 逐位置归一化的好处与限制

好处：

- **与 batch 无关**：batch 大小 1 也能工作，推理与训练行为一致
- **与序列长度无关**：变长序列不需要特殊处理，padding 不影响其他人
- **完全可并行**：每个位置独立，不存在跨位置的依赖

限制：

- **无法利用跨位置的统计信息。**如果某个位置的激活分布本身就是异常的（比如某个 token 的嵌入特别大），LayerNorm 会把它当成“正常输入”来处理，而 BatchNorm 会参考整个 batch 的分布把它识别为异常。
- **在 batch 很大的场景下（比如 CV 任务）不如 BatchNorm。**BatchNorm 能利用 batch 内的统计“平滑”掉单个样本的噪声。

在序列建模里，前三条的重要性压倒后一条，所以 LayerNorm 是默认选择。

2.22.3 ③ 常数偏移对两种归一化的影响

设 $x = [2, 4, 6, 8]$，$x' = x + 100 = [102, 104, 106, 108]$。

RMSNorm：

$$\mathrm{RMS}(x) = \sqrt{30} \approx 5.48, \qquad \mathrm{RMS}(x') = \sqrt{(102^2 + 104^2 + 106^2 + 108^2)/4} \approx 105.0 \tag{2.20}$$

输出分别是 $x/5.48 \approx [0.365, 0.730, 1.095, 1.461]$ 和 $x'/105.0 \approx [0.971, 0.990, 1.010, 1.029]$。

完全不同的输出。RMSNorm **对常数偏移敏感。**

LayerNorm：两者都输出 $[-1.342, -0.447, 0.447, 1.342]$。**完全不受常数偏移影响。**

这个差别什么时候会变成问题?

如果一个特征的取值**系统性地偏向一边**（比如所有输入在这个维度上都是正数），RMSNorm 会把这个“偏置”也当成信号的一部分来缩放。而 LayerNorm 会把偏置减掉，只看相对差异。

换句话说：LayerNorm 丢弃了“这个特征整体偏高”这个信息，RMSNorm 保留了它。
哪种更好取决于任务：

- 如果偏置**是**有意义的信号 → RMSNorm 更好
- 如果偏置是**干扰**（比如不同样本的尺度不一致） → LayerNorm 更好

实践中两者效果接近（我们的实测差异在噪声内），说明在真实任务上这两种信息的分量都不大。

2.22.4 ④ 为什么需要两套键值匹配

因为它们的键来源不同，承担的任务也不同。

	键从哪里来	负责什么
注意力	**随上下文变化**（别的 token 的投影）	**搬运**：把相关信息从别处取过来
FFN	**固定参数**	**存储**：把训练中学到的知识写进表示

举例说明这个分工。考虑“李白”这个名字：模型需要知道它是个人名、是唐代诗人、和“诗”“酒”常一起出现——这是**知识**，存在 FFN 里。

而“这句话的主语是谁”——这**依赖于当前句子**，上一句话的主语和这一句可能完全不同。这是**上下文相关的**，靠注意力。

一个直接的证据：第 1 章我们测过，FFN 的参数占每层的三分之一。如果 FFN 只是“多一层非线性变换”，这个参数比例很难解释。而“记忆库”的视角解释了它为什么要那么大的 d_{ff}。

2.22.5 ⑤ 设计一个需要长程信息的任务

最小可行的设计是“复制任务”：

```
输入：a f q z m ...（16 个随机 token）
      <SEP>
      a f q z m ...（要求原样重复）
输出：       ↑ 预测这一段的每个 token
```

为什么它需要长程信息：要预测第 2 段的第 3 个 token（`q`），模型必须**记住 16 + 2 个位置之前的那个 token**。位置信息在这里是决定性的——搞错位置就取错 token。

预期结果：可学习位置嵌入会因为 `clamp` 把远距离位置映射到错误的位置，导致第 2 段完全崩掉；RoPE 因为相对位置关系正确，能够泛化。

这个任务还有一个好处：可以用**合成数据无限生成**，不存在语料规模的问题，而且 loss 可以降到接近 0（因为答案是确定的）。信号非常干净。

2.22.6 ⑥ 为什么死神经元不能“自己复活”

神经元的预激活是 $a_i = x \cdot k_i$。它要复活，需要 a_i 从负变正。这有两条路径：

路径一：输入 x 变了。可能——但预激活长期为负说明 k_i 与“所有出现过的 x”都反相关。要等到一个全新的输入模式才可能翻正。

路径二：键 k_i 变了。这正是问题所在——**改变 k_i 需要梯度，而 ReLU 在负半轴的梯度恰好是 0**。

$$\frac{\partial\,\mathrm{ReLU}(a_i)}{\partial k_i} = \underbrace{\mathrm{ReLU}'(a_i)}_{=0 \text{ 当 } a_i<0} \cdot x = 0 \tag{2.21}$$

所以 k_i 永远不会被更新——即使某个输入恰好让 a_i 变正了，梯度也传不到（因为梯度还要乘上 ReLU 的导数，而那时它是 1——等等，这种情况下是能更新的）。

准确的表述是：只要 a_i 一直是负的，k_i 就收不到任何梯度，所以它没有机会“自己调整”到一个能被激活的方向。它只能等待某个输入偶然让它翻正——而那个输入的到来越晚，它就越可能永远不被激活。

GELU 的不同之处：$\mathrm{GELU}'(a)$ 在负半轴是小正数而不是 0，所以 k_i 总能收到一点梯度，有机会慢慢调整。

2.23 自测

十道题。前六道能在本章找到直接答案，后四道需要自己推或算。

1. 残差连接为什么是“加”而不是“拼接”？ 说出两个理由，其中一个和参数量无关。

2. LayerNorm 对每个位置独立做归一化。这对**自回归生成**有什么具体好处？

3. Pre-LN 的梯度路径上有一个“恒等项” I。写出它的来源， 并解释为什么 Post-LN 没有这一项。

4. 本章的残差消融实验里，“无残差”的三个深度给出了**逐位相同**的 loss。 这个现象说明什么？你会怎么验证你的判断？

5. FFN 可以重写成 $\sum_i \text{gelu}(x \cdot k_i) \cdot v_i$。 这个形式和注意力有什么相同、什么不同？

6. RMSNorm 保留了 γ 但去掉了 β。各举一个“为什么”。

7. （推导） 证明 Pre-LN 的雅可比里那个恒等项**不会被后续层削弱**。 具体说：证明 $\frac{\partial x_L}{\partial x_l}$ 中总含有一个单位矩阵项， 无论中间隔了多少层。

8. （计算） 某 Pre-LN 模型有 48 层，首层激活标准差为 1.0。 按 式 (2.2) 预测末层的标准差，并与本章实测的 7.78 对比。 如果层数加到 192，预测值是多少？这个值在 bf16 下有问题吗？

9. （分析） 我们在 FFN 分析里测到“层 1 有 14.5% 的槽位从不激活”。 如果这些槽位真的完全没用，把它们删掉会怎样？请指出**至少两种可能出问题的情况**。

10. （设计）你要训一个 64 层的模型。给出你的归一化方案（Pre 还是 Post、Layer 还是 RMS），并说明你会用什么**诊断量**来确认它训练正常。给出至少三个具体指标和阈值。

11. （辨析） “残差连接让深层网络可训”和“残差连接提高模型表达能力”—— 这两句话哪个更准确？为什么？

12. （计算） 一个模型 $d_{\text{model}} = 512$、$n_{\text{head}} = 8$、 $d_{\text{ff}} = 2048$、$n_{\text{layer}} = 24$、词表 50000。 算一下：注意力参数、FFN 参数、嵌入参数各占总参数的多少？ （假设权重共享，LayerNorm 的参数量忽略不计。）

13. （推导） Pre-LN 的残差流方差按 式 (2.2) 随层数线性增长。 **但如果加上 $1/\sqrt{2L}$ 的缩放初始化呢？** 重新推导方差， 说明它为什么能抑制累积。

14. （分析） 本章测到层 1 有 14.5% 的死槽位，层 0 只有 1.6%。 为什么**更深的层**死得更多？给出至少两个可能的原因。

15. （设计） 你要判断“某个组件是不是必需的”。 除了“关掉它看效果”，还有什么办法？至少给出一种， 并说明它相比消融的优势。

16. （综合） 有人说：“Pre-LN 比 Post-LN 好，所以原论文用 Post-LN 是错的。” 请指出这句话里的**至少三个**问题。

17. （综合） 同事说：“我给模型加了残差缩放初始化，结果反而变差了。”给出**至少三种**可能的原因，并说明如何区分。

18. （综合） 一个 24 层模型训练正常，但你把它导出成推理引擎后， 输出变成了乱码。列出**至少三种**可能的原因，按排查顺序排列。

19. （辨析） “残差连接增加了模型的参数量”——这句话对吗？ 如果不对，那“加深网络”增加的是什么？

20. （设计） 你要给一个已有的 12 层模型加“深度”，变成 24 层， 并且希望**尽可能保留原有的能力**（类似“模型生长”）。 给出你的方案，并说明每一步会遇到什么问题。

2.24 自测答案

1. 两个理由：

① **维度不爆炸**。 拼接每层让维度翻倍， L 层后是 $2^L d$—— 12 层就是 4096 倍，根本堆不起来。相加保持维度不变。

②**（和参数量无关）加法让残差流成为“共享通信总线”**。 所有层读写同一个向量空间，各层写入的方向在训练中趋于正交， 因此可以互不干扰地共存。拼接会让每一层拥有自己的私有维度段，层与层之间反而更难交换信息。

2. 自回归生成时 **batch 大小可能是 1**（一次只生成一条序列）。 BatchNorm 依赖 batch 内的统计量，在 batch=1 时无法计算—— 它只能靠训练期累积的移动平均，而训练与推理的分布可能不同。

LayerNorm 完全不依赖 batch，每个位置用自己的特征做归一化， 训练和推理的行为完全一致。这是它在序列模型里成为默认选择的核心理由之一。

3. 恒等项来自 $x \leftarrow x + f(x)$ 这个加法结构。对 x 求导：

$$\frac{\partial(x+f(x))}{\partial x} = I + \frac{\partial f(x)}{\partial x} \tag{2.22}$$

Post-LN 没有这一项，因为它的形式是 $x \leftarrow \mathrm{LN}(x+f(x))$。求导要先穿过 LayerNorm，得到的是 $\frac{\partial \mathrm{LN}(y)}{\partial y}\Big|_{y=x+f(x)} \cdot \left(I + \frac{\partial f}{\partial x}\right)$—— 那个 I 被**夹在 LayerNorm 的雅可比里面**，所以它会被 LayerNorm 缩放，不再是“直通”的。

4. 说明三个配置都退化到了**同一个与容量无关的解**—— 大约就是“输出字频分布，不看输入”。

验证方法（按可靠性排序）：

1. **算相邻位置输出的 KL 散度**。接近 0 说明每个位置输出同一个分布。
2. **和平凡基线比**：算边缘分布的交叉熵（本章是 5.672）。如果模型的 loss 只比它好一点点，就坐实了。
3. **换更小的模型试试**。如果 1 层和 12 层给出同样的 loss，那几乎肯定是退化解。

注意：不要一看到“相同”就认定是 bug。**要先排除“确实存在退化解”这个可能**，再去查代码。本章的例子里就是后者。

5. 相同：都是“用输入匹配一组键，按匹配度加权求和值”的形式。

不同（三处）：

	注意力	FFN
归一化	softmax（权重和为 1，互相竞争）	GELU（各槽位独立，可同时激活或全不激活）
键的来源	随上下文变化	固定参数
槽位数	n（序列长度）	d_{ff}（超参数）

这三处差异对应着不同的功能：注意力做“搬运”（键随上下文变），FFN 做“存储”（键是学到的知识）。

6. 保留 γ 的理由：归一化会强制每个维度的尺度一致。但某些维度的信息量可能天生就大（比如某个特征特别重要），模型需要能把它们**重新放大回去**。γ 提供这个自由度。

去掉 β 的理由：RMSNorm 的设计前提是“中心化不重要”。Zhang 和 Sennrich 的论证是：LayerNorm 起作用的关键是**缩放不变性**，而平移（中心化）带来的收益很小，去掉它能省一次减法和一个参数矩阵。

一个佐证：我们的实测里 LayerNorm 和 RMSNorm 的效果差 0.007（噪声内）。如果 β 真的重要，这个差距应该更大。

7. 用归纳法。记 $J_l = \frac{\partial x_{l+1}}{\partial x_l} = I + A_l$，其中 $A_l = \frac{\partial f_l(\mathrm{LN}(x_l))}{\partial x_l}$。

则 $\frac{\partial x_L}{\partial x_l} = J_{L-1}J_{L-2}\cdots J_l = (I+A_{L-1})(I+A_{L-2})\cdots(I+A_l)$。

展开后，每一项都是若干 A 的乘积，再加上一个“全部取 I”的项：

$$\prod_{k=l}^{L-1}(I+A_k) = I + \sum_{\text{非空子集 } S}\ \prod_{k\in S} A_k \tag{2.23}$$

第一项 I 与 $L-l$ 无关——不管中间隔了多少层，它都是单位矩阵。

这就是“梯度不衰减”的严格含义：变换项 A_k 可能很小甚至接近零（那就意味着那层没学到东西），但**恒等项永远是 I**，所以梯度至少能以“原样”传到第 l 层。

Post-LN 的对应展开里没有这个自由项，每一项都带着至少一个 LayerNorm 的雅可比，所以会随深度衰减。

8. 按 式 (2.2)，方差按 L 线性增长，所以标准差按 $\sqrt{L}$ 增长。若首层标准差为 1：

$$48\text{ 层：}\sqrt{48}\approx 6.93 \qquad 192\text{ 层：}\sqrt{192}\approx 13.86 \tag{2.24}$$

与实测对比：本章 48 层 Pre-LN 的实测是 **7.78**。预测 6.93——同量级，实测略高（因为还有位置嵌入、LayerNorm 的 γ 等因素）。

192 层的预测值 13.86 在 bf16 下有问题吗?

bf16 的**最大可表示值**是 3.39×10^{38}，13.86 远远够用—— **不会溢出**。

但真正的问题在别处：**输入尺度偏离初始化预期 13.86 倍**。第 1 章讲过，权重的初始化（$\sigma = 0.02$）是为**方差为 1** 的输入设计的。输入尺度大了 13.86 倍，输出尺度也会跟着大—— **每一层都在放大这个偏差**。

所以答案不是“会溢出”，而是“初始化假设失效”。这也是为什么深层模型需要残差缩放初始化（$1/\sqrt{2L}$）：它让每层的写入量随深度递减，抵消累积。

9. 两种会出问题的情况：

① **那些槽位并非真的没用，只是我们的测试分布没覆盖到**。我们的统计是在 2 万个位置上做的，都来自唐诗语料。如果换一个分布（比如现代汉语、代码、英文），这些“死”槽位可能立刻活跃起来——**它们是为训练分布之外的情况准备的**。

这解释了为什么剪枝后的模型在**同分布测试集**上表现不变，一换分布就掉点。

② **剪枝会改变残差流上的数值分布**。删掉 14.5% 的槽位，等于把那些槽位原本贡献的输出（即使是 0）从 W_2 的求和中移除。虽然单个贡献是 0，但**其他层的权重是针对“包含这些槽位”的分布训练的**，删掉后分布偏移，后续层需要重新适应。

一个相关的实践经验：剪枝通常需要**微调**来恢复性能，而“能不能不微调直接剪”取决于剪枝比例和模型规模。

10. 我的方案：**Pre-LN + RMSNorm**，理由：

- Pre-LN：恒等梯度路径，64 层必须要有（Post-LN 在这个深度很难训）
- RMSNorm：省计算，效果与 LayerNorm 相当，主流大模型都这么做
- 必须配 **$1/\sqrt{2L}$ 的残差缩放初始化**（$L = 64$），因为 Pre-LN 的残差流会累积

三个诊断指标与阈值：

指标	健康范围	超出说明什么
逐层激活 std 的**末/首比**	< 4	超过说明残差累积过快，需要更强的缩放初始化
逐层梯度 std 的**首/末比**	< 5	超过说明深层梯度被压制
注意力权重熵	0.5–0.9	< 0.1 饱和（缩放缺失）；> 0.99 退化成平均（缩放过度）

再加一条训练早期的检查：前 100 步内，如果“相邻位置输出的 KL 散度”始终接近 0，说明模型没在利用上下文——去查归一化和残差，而不是调学习率。

11. “让深层网络可训”更准确。

残差连接**不增加**模型的表达能力——理论上，一个没有残差但有足够宽度的网络也能表示同样的函数（因为 $x + f(x)$ 里的 f 可以学成任意形式）。

它改变的是“能不能训出来”，而不是“能不能表示”。具体说，它把梯度路径长度从 $O(L)$ 降到 $O(1)$，让优化变得可行。

为什么这个区分重要：它决定了你遇到问题时的排查方向。如果一个模型**表达力不够**（欠拟合），加残差没用——你需要加宽或加深。如果一个模型**训不动**，那才该检查残差。

用错方向的典型症状：给一个训不动的模型加层数，指望“更多参数更强”，结果更训不动。

12. 逐个算（$d = 512$、$h = 8$、$f = 2048$、$L = 24$、$V = 50000$）：

部件	公式	参数量	占比
注意力（每层）	$4d^2 = 4 \times 512^2$	1,048,576	—
FFN（每层）	$2df = 2 \times 512 \times 2048$	2,097,152	—
每层合计	—	**3,145,728**	
全部 24 层	$\times 24$	**75,497,472**	**74.6%**
词嵌入（共享）	$V \times d = 50000 \times 512$	25,600,000	**25.3%**

总参数约 1.011 亿。

三个观察：

1. **FFN 占每层的 2/3**（2,097,152 / 3,145,728 = 66.7%）—— 这是 $f = 4d$ 的结果（$\frac{4d}{2d+4d} = 2/3$）。
2. **嵌入层占了总参数的 1/4**。词表越大，这个比例越高。如果我们用的是 128K 词表，嵌入会占 40%。
3. **注意层数不出现在“每层占比”里**——它同时缩放所有部件，所以不改变配比。

实践含义：如果要在不损失性能的前提下压缩模型，**优先剪 FFN**（占 2/3）而不是注意力。

13. 设每层写入的方差为 σ^2。无缩放时：

$$\mathrm{Var}(x_L) = \mathrm{Var}(x_0) + L\sigma^2 \tag{2.25}$$

加上 $1/\sqrt{2L}$ 的缩放初始化后，每层的写入变成 $\frac{1}{\sqrt{2L}} f(x)$，其方差是 $\frac{\sigma^2}{2L}$。代入：

$$\mathrm{Var}(x_L) = \mathrm{Var}(x_0) + L \cdot \frac{\sigma^2}{2L} = \mathrm{Var}(x_0) + \frac{\sigma^2}{2} \tag{2.26}$$

关键在那两个 L 抵消了。累积量不再随深度增长，而是一个常数 $\sigma^2/2$。

为什么分母里有 2：每层有两个残差分支（注意力和 FFN），所以每层的写入方差是 $2\sigma^2$。除以 $2L$ 之后，L 层的总累积是 $L \cdot \frac{2\sigma^2}{2L} = \sigma^2$。

这个推导说明了一件事：缩放初始化不是“拍脑袋的小技巧”，它是**针对方差累积的精确补偿**。指数的 $1/2$ 来自“方差线性相加”这个性质。

代价：深层模型的每层写入很小，**前期学得慢**。这是为什么它通常配合更长 warmup 使用。

14. 两个可能的原因：

① **梯度更小**。反向传播时，梯度沿深度累积衰减（即使是 Pre-LN，梯度尺度也有 4–8 倍的不平衡，见 图 2.1）。深层的槽位收到的梯度更小，更不容易被“激活”到有用的方向。

② **输入分布更“收敛”**。层 0 面对的是词嵌入 + 位置编码—— 分布相对分散（每个字的嵌入都不同）。而层 1 面对的是**层 0 的输出**—— 已经被注意力混合过，分布更集中。

死神经元的成因是“预激活长期为负”。如果输入分布集中，那么“所有输入都让某个键的匹配度为负”就更容易发生。

③（**附加**）**与初始化有关**。我们的模型中，层 1 的 FFN 也在 $1/\sqrt{2L}$ 缩放初始化的范围内（$L = 12$ 时系数约 0.204）。更小的初始权重意味着更小的预激活，更容易一开始就落在负半轴。

这三个原因指向同一个实践结论：**深层模型的 FFN 需要更仔细的初始化，或者用 GLU 这类对负值更友好的激活函数**。

15. 至少三种替代方法：

① **权重扰动敏感度**。给某个组件的参数加一点噪声，看 loss 变化多大。变化大 → 该组件重要。

- **优势**：不需要重新训练，代价极低
- **劣势**：只反映**局部**敏感度。一个组件可能在当前参数点不敏感，但在训练轨迹的其他位置很关键

② **梯度量级**。看该组件的参数梯度范数。梯度大 → 它在被积极调整 → 重要。

- **优势**：训练过程中免费获得
- **劣势**：梯度大也可能只是“还没调好”，不代表功能重要

③ **激活分析**。看该组件的激活统计（本章对 FFN 做的：激活频率、键相似度）。

- **优势**：能看出**结构**（哪些槽位在用、是否冗余）
- **劣势**：是相关性而非因果性——“激活频繁”不等于“必需”

为什么消融仍然是最可靠的：它是**因果干预**—— 直接改变系统，观察结果。其他方法都是观察性的。

但消融的代价最高（要重训），而且“关掉后效果不变”也可能是 **其他组件补偿了**（冗余），而不是“它没用”。

实践建议：先用①②③筛出候选，再用消融确认。

16. 三个问题：

① **它把一个有条件的结论说成了无条件的**。本章的实测是：在 12 层、$d_{\text{model}} = 128$、这个语料上，Pre-LN 在**最优 loss 和容错范围**两个指标上更好。Xiong 等人的结论是在机器翻译的设定下得到的。**两个实验都支持“在这个条件下 Pre-LN 更稳”，但都不支持“Pre-LN 普遍更好”**。

② **它忽略了时间维度**。原论文用 Post-LN 是 2017 年，而 Pre-LN 的普及是 2019–2020 年之后的事。**“当时的选择”和“现在的选择”面对的是不同的知识积累和技术条件**。说“原论文错了”是典型的后见之明。

③ **它把“次优”等同于“错”**。Post-LN 支撑了 BERT、GPT-1、GPT-2—— 这些模型都不是失败的。Post-LN 的问题是“需要更小心的调参”，不是“不能用”。

④（**附加**）**它没有区分“更好”的两种含义**。“各自最优”（Pre-LN 4.408 vs Post-LN 4.732）和“鲁棒性”（Pre-LN 单调改善 vs Post-LN 有窄峰）是两个不同的指标。**说“更好”时必须说清是哪一个**。

这个练习的目的是训练一种习惯：看到一个比较性的结论，先问“在什么条件下”、“和什么比”、“哪个指标”。

17. 三种可能：

① **缩放过度，前期学得太慢**。$1/\sqrt{2L}$ 让每层写入很小，优化器需要更多步才能把权重推到有用的尺度。**如果训练步数没变，模型可能还没“启动”就被你判了死刑**。

区分方法：看 loss 曲线的形状。如果是“下降更慢但趋势相同”，那是这个原因——延长训练即可。

② **缩放系数算错了**。常见错误是用 $1/\sqrt{L}$ 而不是 $1/\sqrt{2L}$（漏掉了“每层两个残差分支”这个 2），或者 L 用了错误的层数。

区分方法：打印初始化后残差分支的权重标准差，和手算的值对比。

③ **与已有的归一化重复**。如果你同时用了 Pre-LN **和**缩放初始化，两者都在控制方差——叠加起来可能过度抑制。

区分方法：做消融，单独关掉其中一个看效果。

④ **（附加）它本来就不该用在这个规模上**。层数少时（比如 4 层），方差累积本来就不严重，缩放初始化只会拖慢前期。**§2.13 讲了：它是“减缓”而不是“消除”，层数不够时不需要**。

18. 按排查顺序：

① **序列长度越界**。推理时的序列超过了训练时的 `block_size`，`pos_emb` 索引越界或静默取错（§2.8 失败六）。**最快排查**：把输入截断到训练长度，看是否恢复正常。

② **dtype 不一致**。导出到推理引擎时精度变了（fp32 → fp16），而模型里有依赖 fp32 的归一化（§2.8 失败八）。**排查**：对比两种精度下的中间激活。

③ **归一化的 ϵ 在低精度下失效**。比如 $\epsilon = 10^{-5}$ 在 fp16 下可能被舍入成 0，导致除零或数值失稳。**排查**：把 ϵ 提到 10^{-3} 试试。

④ **位置编码的索引方式不同**。有些推理引擎对 `arange` 的实现不同，或者 KvCache 的偏移量算错。**排查**：只跑 1 个 token，看输出是否正常。

⑤ **权重转换时的精度损失**。量化或格式转换引入了误差。**排查**：用同一份输入，对比原模型和导出模型的 logits。

排序理由：从“最容易查”到“最难查”。第 ① 条一个命令行就能验证，应该先试。

19. 不对。

残差连接**不增加参数量**——$x + f(x)$ 里的 f 本来就要算，加法本身没有参数。

“加深网络”增加的是：

- **表达能力**（更深的函数复合）
- **参数**（每多一层就多一组注意力和 FFN 的权重）

但要注意：残差连接**限制**了深度的“有效表达能力”—— 它让网络倾向于学“对恒等映射的小修正”，而不是任意的复杂函数。这正是它好训的原因，也是一个**归纳偏置**。

所以更准确的说法是：加深网络增加参数与理论表达能力，而残差连接**改变了“深”的含义**——从“多层复合”变成“累积的小修正”。

20. 一个可行的方案（“模型生长”）：

第 1 步：决定在哪里插入新层。

三个选择：

- **底部插入**（靠近输入）：新层学到的是“底层特征”，与已有层差异大
- **顶部插入**（靠近输出）：新层要重新适应输出分布
- **中间插入**：折中，但需要选插入点

推荐从顶部附近插入——因为越靠近输出的层越“任务相关”，新层可以快速适应。

第 2 步：新层的初始化。

这是最关键的一步。新层的残差分支要用**接近零**的初始化（比标准的 $1/\sqrt{2L}$ 更小），让它一开始近似于恒等映射—— 这样加进去几乎不改变原模型的输出。

如果新层初始化得太大，它会在前向传播时大幅改变残差流，原有的 12 层全部“看不懂”输入了。

第 3 步：处理 `ln_f` 与缩放初始化。

原来的缩放系数是按 $L = 12$ 算的。变成 24 层后，理想值应该按 $L = 24$ 重算。但**重算会改变所有旧层的权重**—— 那等于破坏了原有能力。

折中做法：保持旧层不动，只让新层用新的缩放系数。

每一步会遇到的问题：

步骤	问题
插入位置	插入点之后的层，输入分布变了
初始化	太大 → 破坏原模型；太小 → 新层学不动

步骤	问题
缩放系数	重算会改变旧层；不重算则理论值不对
训练	需要“先冻结旧层训新层，再联合微调”的两阶段策略

这个练习的意义：它把本章的两个核心概念（残差流是共享总线、初始化的方差控制）放在一个具体问题上。**理解了“为什么加层会破坏原模型”，就理解了残差流的本质**。

2.25 动手练习

四个练习，难度递增。都能在本章的代码基础上完成。

2.25.1 练习一：复现案例并做一次预测

跑 `run_ch2_case.py`，确认你能得到与本章一致的数字 （Pre-LN 4.245 / Post-LN 5.357，以及扫描表的四个点）。

然后**先预测再验证**：如果把 `n_head` 从 4 改成 8（d_{head} 从 32 变成 16），Pre-LN 的 loss 会变好、变差还是不变?

提示：头数改变的是“每个头能看多细”，不改变参数量（回顾第 1 章 §1.8）。$d_{\text{head}} = 16$ 时每个头的表达力更弱，但头更多。你预期净效果是什么?

2.25.2 练习二：找到残差流累积的临界点

设计实验找出“Pre-LN 的残差缩放初始化在多少层之后变成必需”。

建议的层数序列：8、16、32、64。

要求： - 每个层数跑两版：有 `1/√(2L)` 缩放初始化、无 - 记录 loss 曲线和末层激活尺度 - 画一张“层数 vs 两个版本的 loss 差”的图 - 找出这个差值开始明显上升的位置

和练习二（第 1 章）的关系：第 1 章那个练习找的是“缩放因子何时变成必需”，这个找的是“残差初始化何时变成必需”。**两个都是同一类问题：某个机制在什么规模下才显现**。

2.25.3 练习三：给 `analyze_ffn.py` 加一个“神经元语义”分析

现在的脚本测了激活频率和键相似度，但没回答“**这个神经元在检测什么**”。

请加一个功能：对每个高频激活的神经元，找出**让它激活最强的那些输入**，并统计这些输入位置上的字符分布。

有什么用：如果某个神经元的前 20 个最强化激活都发生在“。”之后，那它可能在检测“句子边界”。这就是**从权重里读出语义**的第一步。

实现提示：用 hook 收集 `(激活值, 输入的 token id)` 对，按激活值排序取前 k 个，统计 token 的频率分布。

2.25.4 练习四：设计一个真正能测出外推的任务

本章的 RoPE 实验没能测出外推差异（§2.7）。**请设计一个能测出来的任务**。

要求： - 用合成数据（可以无限生成） - 任务的**正确答案依赖于序列中某个固定距离之外的信息** - 给出基线：随机猜测、只看局部的 n-gram、以及一个有位置编码的模型

验收标准：在你设计的任务上，“可学习位置嵌入”和“RoPE”应当表现出**明显的**差异（至少 0.5 的 loss 差，远超训练噪声）。

提示：§2.7 检查点 ⑤ 给了起点（复制任务）。如果那个任务的差异还不够大，想想怎么让它**更依赖精确的位置**。

做完这些练习之后

第 1 章和第 2 章的八个练习，合起来覆盖了 Transformer 的**两个最核心的机制**（注意力、Block 结构）的完整调试技能。

如果你都做了，你手里应该有一套能直接用在其他项目上的工具：注意力诊断、残差流画像、消融流程、以及“先估噪声再下结论”的习惯。

这些比记住任何具体结论都耐用。

而如果你只做了其中一两个，那也比读完本章所有推导更有价值——因为推导可以随时回来查，手感不行。

2.26 关键概念表

中文	英文	一句话定义	在本章哪一节
残差连接	residual connection	$x \leftarrow x + f(x)$，提供一条不经过变换的直通路径	§ 2.5
残差流	residual stream	被所有层读写的那条共享向量通道	§ 2.5
层归一化	layer normalization	对每个位置的特征维做标准化，再学缩放与平移	§ 2.6
批归一化	batch normalization	跨样本归一化，依赖 batch 统计量	§ 2.6
ϵ	epsilon	归一化里防止除零的小常数，过大会削弱归一化	§ 2.8
Pre-LN	pre-layer-norm	归一化放在子层**之前**，梯度有恒等路径	§ 2.7
Post-LN	post-layer-norm	归一化放在残差**之后**，梯度必须穿过归一化	§ 2.7
最终归一化	final layer norm (`ln_f`)	输出投影前的归一化，Pre-LN 下不能省	§ 2.8
RMSNorm	root mean square norm	只按均方根缩放，不减均值、无平移参数	§ 2.8
缩放初始化	scaled initialization	残差分支权重乘 $1/\sqrt{2L}$，减缓方差累积	§ 2.7
前馈网络	feed-forward network (FFN)	两层线性 + 激活，逐位置独立作用	§ 2.9
键值记忆	key-value memory	FFN 的另一种理解：d_{ff} 个“键→值”槽位	§ 2.9
门控线性单元	gated linear unit (GLU)	SwiGLU 等门控 FFN 变体	§ 2.9

中文	英文	一句话定义	在本章哪一节
混合专家	mixture of experts (MoE)	多个 FFN 专家，每次只激活少数几个	§2.9
死神经元	dead neuron	长期不激活、收不到梯度的槽位	§2.8
置换等变	permutation equivariant	输入换序则输出跟着换序	第 1 章
旋转位置编码	RoPE	用旋转矩阵把相对位置编进 q、k 的内积	第 1 章 §1.13
NTK 插值	NTK-aware interpolation	通过放大频率基来扩展 RoPE 的可用长度	§2.10
长度外推	length extrapolation	在超出训练长度的序列上保持性能	§2.10
消融	ablation	关掉某组件看影响，用于判断其必要性	§2.11

2.27 延伸阅读

按“读它能得到什么”排序，不是按重要性。

2.27.1 一、为什么是 Pre-LN

Xiong et al., On Layer Normalization in the Transformer Architecture (ICML 2020).

读它能得到什么：本章案例的理论版本。论文从梯度尺度出发，证明了 Post-LN 在初始化时会遇到梯度问题，而 Pre-LN 不会，并给出“Pre-LN 不需要 warmup”的结论。

读的时候注意两件事：

1. **它的实验条件**。6 层 encoder + 6 层 decoder、$d_{\text{model}} = 512$、WMT 翻译数据。我们的实验（12 层 decoder-only、$d_{\text{model}} = 128$、25 万字唐诗）落在它的条件之外，所以在 warmup 那一项上观察到了不同的现象（§2.7）。
2. **“不需要 warmup”的准确含义**。论文的论证是“Pre-LN 的梯度尺度与深度无关”，所以可以用大的初始学习率。**这不等于“warmup 对 Pre-LN 无益”**——我们的实测是 Pre-LN 反而从更长的 warmup 中获益（4.394 → 4.160）。

2.27.2 二、残差连接为什么会起作用

He et al., Deep Residual Learning for Image Recognition (CVPR 2016).

读它能得到什么：残差连接的原始出处（ResNet）。虽然它是视觉领域的工作，但“恒等映射让深层网络可训”这个洞察 是通用的——本节的 式 (2.7) 就是它的形式化。

读的时候注意：论文的论证是“学习残差比学习完整映射更容易”。对 Transformer 来说还有一个更重要的理由（第 1 章 §1.7 讲过）：**恒等路径让梯度可以绕过所有非线性变换**。ResNet 的论文没有强调这一点。

Veit et al., Residual Networks Behave Like Ensembles of Relatively Shallow Networks (NeurIPS 2016).

读它能得到什么：一个漂亮的视角——残差网络等价于**指数多个浅层网络的集成**。因为每条残差路径可以被“走”或“跳过”，L 层就有 2^L 条路径。

和本章的关系：它解释了为什么“残差流是共享总线”这个比喻成立——总线上的信息可以被任意子集的层处理，而不是强制经过所有层。

2.27.3 三、归一化的简化

Zhang & Sennrich, Root Mean Square Layer Normalization (NeurIPS 2019).

读它能得到什么：RMSNorm 的原始论文，包括“为什么中心化不必要”的论证。
读的时候注意：论文的核心论据是 **LayerNorm 的缩放不变性**—— 它才是让训练稳定的关键，而减均值带来的收益很小。 第 4 节我们会看到这个论证在实践上被验证了（两者效果差 0.007）。
Ba et al., Layer Normalization (2016).
读它能得到什么：LayerNorm 的原始论文。
读的时候注意：它最初的动机是 **RNN**（因为 BatchNorm 在 RNN 上不好用）。 Transformer 只是继承了这个工具，并给了它新的意义 （控制残差流的方差，而不只是“稳定激活分布”）。

2.27.4 四、FFN 是记忆

Geva et al., Transformer Feed-Forward Layers Are Key-Value Memories (EMNLP 2021).
读它能得到什么：本章 §2.9 的原始出处。 论文在 GPT-2 上验证了“W_1 的行是键、W_2 的列是值”这个视角， 并发现**浅层的键对应具体的词法模式，深层的键对应更抽象的语义模式**。
读的时候注意它的实验设计：他们怎么找出“每个神经元检测什么模式”的？ 这个方法和我们练习三里让你做的是一回事。
读完之后可以想一个问题：如果 FFN 是记忆，那“知识编辑”（直接修改模型里的某个事实）是不是就是“改写某个槽位的值”？ 这条思路确实存在（ROME、MEMIT 等工作）。

2.27.5 四·补、归一化的其他视角

Ba et al., Layer Normalization (2016) 的原始动机值得单独说。
他们提出 LayerNorm 是为了解决 **RNN 的训练问题**：BatchNorm 在 RNN 上不好用（序列长度可变、batch 统计量不稳定）。 所以 LayerNorm 最初的设计目标是“**让每个时间步的激活分布稳定**”。
Transformer 继承了这个工具，但给了它新的意义：在残差网络里，LayerNorm 的作用从“稳定激活分布”变成了 “**控制残差流的方差累积**”。
这个转变的启示：一个工具被引入时的理由， 和它后来起作用的机制，可能不是同一件事。 读老论文时要留意这个错位。
Zhang et al., Understanding the Role of Layer Normalization in Transformers (2024 前后的一系列工作)
读它能得到什么：LayerNorm 在 Transformer 里到底做了什么， 是近年仍在争论的问题。有工作认为它的主要作用是**梯度缩放**， 也有工作强调它的**表示几何**效应。
为什么值得读：它是一个“看似已经解决的经典问题仍在被重新审视”的例子。本章 §2.6 给的“控制方差累积”是一个主流解释， 但它不是唯一的解释。

2.27.6 四·补二、预训练与微调中的归一化

一个实践上的补充：归一化的**仿射参数**（γ、β） 在微调时通常**冻结或使用更小的学习率**。
为什么：这些参数编码的是“预训练数据上的特征尺度”。 微调数据分布不同，如果让它们自由调整， 模型可能失去预训练学到的尺度信息。
具体的做法：常见的策略是给归一化参数单独设一个小的学习率乘子（比如主学习率的 0.1 倍），或者完全冻结。
这一点本章没有涉及（我们只讲了从头训练）， 但如果你要做微调，它值得知道。

2.27.7 五、旋转位置编码

Su et al., RoFormer: Enhanced Transformer with Rotary Position Embedding (2021).
读它能得到什么：RoPE 的原始论文。第 1 章 §1.13 的推导主干就在它的第 3.2 节。
和本章 §2.10 的关系：我们那个“没测准”的实验说明了一件事—— **论文里的外推优势需要合适的任务才能显现**。RoFormer 论文用的是 长文本语言建模和翻译，而不是局部可预测的诗歌续写。
Chen et al., Extending Context Window of Large Language Models via Positional Interpolation (2023) 及相关工作（NTK-aware、YaRN）.
读它能得到什么：长度外推的几条主流技术路线。它们都在做同一件事：**把超出训练范围的位置“压缩”到模型熟悉的尺度上**。

读的时候注意：我们的实验里 NTK 插值反而更差（§2.10）。这些工作通常需要**在长文本上继续微调**才能生效，单纯改频率而不微调，效果不保证。这是我们那个实验可能失败的原因之一。

2.28 从零实现：一次完整的走查

前面各节分开讲了每个零件。这一节把它们**串成一次完整的实现走查**——从空文件到能训的模型，每一步都在解决一个具体的问题。

建议你对着这一段自己写一遍。读代码和写代码是两件事。

2.28.1 第 0 步：先想清楚形状

在写任何代码之前，把数据在各处的形状写下来：

```
输入 idx          (B, T)
词嵌入后          (B, T, d)
加位置编码        (B, T, d)
Block 输入输出    (B, T, d)      ← 形状不变!
输出 logits       (B, T, V)
```

"Block 的输入输出形状相同"这一条是整个架构的基石——它让堆叠任意层数成为可能。花一分钟确认这一点，比 debug 半天形状错误划算。

2.28.2 第 1 步：词嵌入 + 位置编码

```python
x = self.tok_emb(idx) + self.pos_emb(pos)[None, :, :]
```

三个细节：

1. **`pos` 要 reshape 成 `(1, T)`**，才能和 `(B, T, d)` 广播
2. **加法而不是拼接**——保持了维度不变
3. **`pos_emb` 的最大索引是 `block_size - 1`**——超了会报错（这是好事）

2.28.3 第 2 步：注意力

先写单头，确认能跑通，再改成多头。

```python
q, k, v = self.qkv(x).split(C, dim=2)                  # 一次投影出三个
q = q.view(B, T, H, Dh).transpose(1, 2)                # (B, H, T, Dh)
att = q @ k.transpose(-2, -1) / math.sqrt(Dh)          # 打分 + 缩放
att = att.masked_fill(~mask, float("-inf"))            # 因果掩码
att = F.softmax(att, dim=-1)
y = (att @ v).transpose(1, 2).reshape(B, T, C)         # 拼回
return self.proj(y)
```

最容易错的是那个 `transpose`。顺序必须是：

```
(B, T, H, Dh) → view → transpose(1,2) → (B, H, T, Dh)
```

如果忘了 transpose，`q @ k.transpose(-2,-1)` 会在错误的维度上做矩阵乘法——形状可能碰巧对，但结果完全是错的（它在对"头"和"位置"做注意力）。

自查方法：注意力矩阵的形状应该是 `(B, H, T, T)`。`T` 出现在最后两维。

2.28.4 第 3 步：FFN

```python
self.mlp = nn.Sequential(
    nn.Linear(d, f), nn.GELU(), nn.Linear(f, d)
)
```

注意输出维度必须回到 `d`——否则残差加法会失败。这个"必须回到 d"的约束是**结构性的**：残差连接要求输入输出同形。

2.28.5 第 4 步：组装 Block

```python
def forward(self, x):
    x = x + self.attn(self.ln1(x))       # Pre-LN
    x = x + self.mlp(self.ln2(x))
    return x
```

两行，但有两个关键决定：

- **Pre-LN**（归一化在子层之前）——理由见 §2.7
- **残差加法**——理由见 §2.5

如果这两行里的任何一个写错，模型的表现会天差地别，而错误本身可能看起来微不足道（只是把 `ln` 挪了个位置）。

2.28.6 第 5 步：输出层

```python
x = self.ln_f(x)                          # ← 不能省
logits = self.head(x)                     # (B, T, V)
loss = F.cross_entropy(logits.view(-1, V), targets.reshape(-1))
```

`view(-1, V)` 那一步是把 `(B, T, V)` 摊平成 `(B*T, V)`，因为 `cross_entropy` 期望的是 `(N, C)` 的形状。`targets.reshape(-1)` 对应摊平。

2.28.7 第 6 步：初始化

```python
self.apply(self._init)                    # 全局: 0.02
for name, p in self.named_parameters():   # 残差分支: 额外缩放
    if name.endswith("proj.weight") or name.endswith("mlp.2.weight"):
        nn.init.normal_(p, mean=0.0, std=0.02 / math.sqrt(2 * n_layer))
```

顺序很重要：先全局初始化，再对残差分支做额外缩放。反过来会被全局初始化覆盖掉。

2.28.8 第 7 步：验证

在训练之前，做三个检查：

```python
# ① 形状对不对
x = torch.randint(0, V, (2, 16))
logits, loss = model(x, x)
assert logits.shape == (2, 16, V)

# ② 初始 loss 应该接近 ln(V)
print(f"初始 loss {loss.item():.3f}，期望 {math.log(V):.3f}")
# 差太多说明输出层或初始化有问题

# ③ 因果性：位置 t 的输出不应受位置 >t 的输入影响
x1 = x.clone()
x2 = x.clone(); x2[:, 5] = 999            # 只改位置 5
l1, _ = model(x1); l2, _ = model(x2)
assert torch.allclose(l1[:, :5], l2[:, :5]), "未来信息泄露了！"
```

第 ③ 条是最重要的自查——它一次性验证了掩码的正确性。第 1 章 §1.15 讲的那个“掩码用 0”的 bug，这一条断言就能抓住。

> 这七步的顺序不是随意的
>
> 注意我是按“**先骨架、后细节**”排的：
> 1-2 步建立数据流；3-4 步加上核心机制；5-6 步处理输出和初始化；7 步验证。
> **每一步都建立在上一步已验证的基础上**。 如果第 2 步的注意力没验证就往下写， 到第 7 步发现因果性泄露时，你要在五个地方找原因。
> **“小步验证”听起来慢，实际上快得多**。

2.29 怎么读别人的实现

本章讲的是“Block 应该怎么写”。但真实项目里， 你更多是**读别人已经写好的代码**。这一节给一套方法。

2.29.1 先找“形状契约”

任何 Transformer 实现，先从头部找到这几行：

```
# 典型的位置
self.tok_emb = nn.Embedding(vocab_size, d_model)
self.pos_emb = nn.Embedding(block_size, d_model)
self.blocks = nn.ModuleList([Block(...) for _ in range(n_layer)])
self.ln_f = ...
self.head = nn.Linear(d_model, vocab_size, bias=False)
```

这三行告诉你全部结构： ① 有没有位置编码、是哪种；② 有多少层；③ 输出前有没有归一化。
然后看 `forward` 的第一行和最后一行，确认输入输出形状：

```
def forward(self, idx, targets=None):
    B, T = idx.shape
    ...
    return logits, loss          # logits: (B, T, V)
```

形状契约建立之后，剩下的都是细节。

2.29.2 然后找“那个不同的地方”

一个读懂实现的高效方法：**拿它和你熟悉的版本对比，找差异**。
以本章的知识为基准，你可以问：

问题	如果答案不同，说明什么
归一化在残差里面还是外面?	Pre-LN vs Post-LN（§ 2.7）
用的是 LayerNorm 还是 RMSNorm?	是否省略了中心化（§ 2.8）
输出前有没有 `ln_f`?	Pre-LN 必须有（§ 2.8 失败三）
缩放初始化了吗?	是否用了 $1/\sqrt{2L}$（§ 2.13）
位置编码怎么做的?	可学习 / 正弦 / RoPE / ALiBi（§ 2.12）
注意力里有几个矩阵?	MHA / MQA / GQA（第 1 章 § 1.9）
FFN 是两层还是三层?	标准 / GLU 变体（第 2 章 § 2.9）

七个问题，五分钟就能摸清一个陌生实现的全貌。
为什么要“找不同”而不是“从头读”?
因为从头读会把注意力平均分配到每一行， 而**九成的代码是标准的**——真正决定行为的是少数几个地方。
“找不同”把注意力集中在那些地方。

而且这个方法有一个额外好处：**你同时在做一次隐式的消融分析**。每发现一处不同，你就能预测“这会让它的行为有什么不同”—— 然后可以去看论文或实验验证。

预测—验证，比被动阅读学得快得多。

2.29.3 三个容易看漏的地方

① 掩码的构造方式。

不同实现的掩码写法差别很大：

```
# 写法 A：预建 buffer
self.register_buffer("mask", torch.tril(torch.ones(T, T)))

# 写法 B：用 is_causal 标志（PyTorch 2.0+）
F.scaled_dot_product_attention(q, k, v, is_causal=True)

# 写法 C：动态构造（支持变长）
mask = torch.arange(T)[None, :] <= torch.arange(T)[:, None]
```

要确认的是：掩码是 $-\infty$ 还是 0（第 1 章 §1.15 失败一）；掩码是 `bool` 还是 `float`（决定用 `masked_fill` 还是加法）。

② 位置编码的索引方式。

```
pos = torch.arange(T)                    # 从 0 开始
pos = torch.arange(1, T + 1)             # 从 1 开始（有些实现这样）
pos = torch.arange(T) % block_size       # 循环（有些实现这样）
```

第二种和第三种都会改变外推行为（§2.10）。如果模型在长序列上表现异常，先查这里。

③ 权重的绑定与初始化顺序。

```
self.head.weight = self.tok_emb.weight    # 绑定
self.apply(self._init)                    # 初始化
```

如果顺序反了，`head` 会被初始化两次，第二次用的是 `Embedding` 的初始化分布——**和 `Linear` 的期望不同**。

自查方法：训练前打印 `model.head.weight.std()` 和 `model.tok_emb.weight.std()`，两者应该完全相同。

2.29.4 一个实际的例子：读 nanoGPT

以 Karpathy 的 nanoGPT 为例，用上面的七问快速过一遍：

问题	nanoGPT 的答案	对应本章哪一节
归一化位置	Pre-LN	§2.7
归一化类型	LayerNorm	§2.8
`ln_f`	有	§2.8
缩放初始化	有（`1/√(2·n_layer)`）	§2.13
位置编码	可学习嵌入	§2.12
注意力类型	MHA（可配 MQA/GQA）	第 1 章 §1.9
FFN 层数	2 层 + GELU	§2.9

七行就定位了它的全部设计选择。剩下的 300 行是实现细节。

这套方法的迁移性

这七个问题**不只适用于 Transformer**。任何“读一个陌生实现”的场景，都可以问同一类问题：

1. **接口是什么**（输入输出的形状与语义）
2. **核心机制在哪几行**（找到那个“决定行为”的地方）
3. **它和我熟悉的版本有什么不同**（差异分析）
4. **那几个差异会带来什么后果**（预测—验证）

把“读代码”变成“找不同 + 做预测”，效率会高一个量级——而且这个习惯不限于 Transformer，任何陌生的代码库都适用。

2.30 本章的实验清单

全章一共做了五个实验。这里汇总成一个可复现的清单，**每个都给出脚本、原始数据、和它支持的结论**。

2.30.1 实验一：Pre-LN 与 Post-LN 的深度对照

脚本：`notebooks/run_ch2_case.py` **数据**：`data/ch2_case.json`

项	设置
变量	归一化位置（pre / post）× 深度（6 / 12 / 24）
固定	语料、种子、步数、批大小、学习率
规模	6 次训练，各 1200 步

支持的结论：同超参数下 Post-LN 明显更差（5.357 vs 4.245）；但这个对比**不公平**——见实验二。

2.30.2 实验二：学习率扫描

脚本：内联在 `run_ch2_case.py` 的输出里 **数据**：`data/ch2_lr_sweep.json`

项	设置
变量	归一化位置 × 学习率（10^{-4} / 3×10^{-4} / 10^{-3} / 3×10^{-3}）
固定	12 层、900 步、其余同实验一
规模	8 次训练

支持的结论：在 3×10^{-4} 处两者几乎持平（4.703 vs 4.732）； 差别在**容错范围**（Pre-LN 单调改善，Post-LN 有窄峰）。

这个实验改变了全章的结论。 没有它，我们会写出一个错误的判断。

2.30.3 实验三：残差消融

脚本：内联 **数据**：`data/ch2_ablation.json`

项	设置
变量	有无残差 × 深度（2 / 4 / 12）
规模	6 次训练，各 600 步

支持的结论：**没有残差时，2/4/12 层全部给出 5.392**—— 同一个退化解。连 2 层都训不动。

这是全章最干净的实验结果：信号极强（差异接近 1.0），而且“三个深度给出同一个数字”这个模式本身就是证据。

2.30.4 实验四：位置编码外推

脚本：`notebooks/rope_extrapolation.py` **数据**：`data/rope_extrapolation.json`

项	设置
变量	位置编码（learned / rope / rope+NTK）× 评估长度（64 / 128 / 256）
固定	训练长度 64、4 层、2000 步

支持的结论：**没有测出预期的效应**（§ 2.10）。 RoPE 在 128 上更好（4.21 vs 4.56），但 256 上反而更差（4.93 vs 4.85）。

失败的原因：任务不依赖长程位置信息（用实验五佐证）。

2.30.5 实验五：上下文长度

脚本：内联 **数据**：`data/ch2_context.json`

项	设置
变量	`block_size`（8 / 16 / 32 / 64 / 128）
固定	4 层、800 步
规模	5 次训练

支持的结论：8 → 128 改善 0.64（任务**确实**用到较长上下文）； 但 16 比 8 更差 → **单次训练的噪声约 0.1–0.2**。

它还提供了全章的噪声基准——任何小于 0.2 的差异都不能作数。

2.30.6 汇总表

实验	变量数	训练次数	结论强度
一、深度对照	2×3	6	中（受超参数影响）
二、学习率扫描	2×4	8	**强**（改变了全章结论）
三、残差消融	2×3	6	**很强**（差异接近 1.0）
四、位置编码	3×3	3	**弱**（阴性结果）
五、上下文长度	5	5	中（提供了噪声基准）

共 28 次训练，在 RTX 3090 上总计约 25 分钟。

这张表想说明什么

五个实验里，只有两个给出了强结论。

- 实验三（残差消融）：信号极强，一次实验就够
- 实验二（学习率扫描）：改变了全章结论，**是最有价值的一次**

实验一看起来最“正面”（给出了清晰的对比），但它其实是**误导性**的——如果不做实验二，会得出错误结论。

实验四花了最多时间（三个变体、三种评估长度），结论却最弱——因为任务选错了。

这个分布是研究的常态：大部分实验给出的信息量有限，少数几个决定了结论。**关键是要能分辨哪个是哪个，而不是把每个实验都当同等重要**。

2.31 规模变化时，这些选择会怎么变

本章的模型是 170 万参数。真实模型是它的几千倍。**这一节讨论：本章的那些结构选择，在规模变大时会怎样**。

2.31.1 哪些选择与规模无关

有四条结论**不会随规模改变**，因为它们来自结构性的事实：

结论	为什么与规模无关
残差不可省	它解决的是“梯度路径长度 $O(L)$”，与 L 的具体值无关
Pre-LN 的恒等路径存在	同样与 L 无关
`ln_f` 不能省	残差累积在更大规模下只会更严重
FFN 是键值记忆	这是数学重写，不是经验观察

这四条可以直接用在任何规模上。

2.31.2 哪些选择会翻转

① **$d_{\text{ff}}/d_{\text{model}}$ 的比例**。
本章用的是 2 倍（因为我们模型小）。真实模型普遍用 **4 倍，甚至更高**（Llama 用 $8/3 \times d$ 配合 SwiGLU，等效于标准 FFN 的 4 倍）。
为什么大模型要用更大的比例：前面算过，$f = 4d$ 时 FFN 占每层参数的 **2/3**。大模型需要更多的“知识存储”，所以把预算往 FFN 倾斜。
反过来说：小模型上 FFN 比例高反而可能浪费—— 小模型学不到那么多知识，参数应该给注意力（“搬运”能力）。
② **层数与宽度的权衡**。
本章用 12 层、$d_{\text{model}} = 128$——**偏深**（相对宽度）。
真实的小模型往往**偏宽**，大模型**偏深**。这个趋势在 § 2.12 讲过。
③ **学习率**。
本章最优是 3×10^{-3}（Pre-LN）。 真实大模型普遍在 10^{-4} 到 3×10^{-4} 量级。
为什么大模型要用更小的学习率：这是一个经验规律， 一个直观的解释是大模型的损失曲面更“尖锐”， 步子太大会直接跨过最优点。
④ **batch size**。
本章固定 32。真实训练用几百到几千。
为什么大模型能用更大的 batch：梯度噪声随 batch 增大而减小， 大模型需要更大的 batch 才能让梯度估计稳定。 这是“梯度噪声尺度”（gradient noise scale）这个概念的核心。
怎么判断一个选择“会不会随规模翻转”?
问一句：**它依赖的是“结构性事实”还是“经验观察”?**

- **结构性事实**：来自数学（梯度路径、秩、方差传播）。这些在任意规模下成立，改变规模不改变结论
- **经验观察**：来自“在这个规模上测出来最优”。这些一**定**会随规模变化，因为最优值本身是规模的函数

本章的七条结论里，四条是结构性的、三条是经验性的。读的时候要能分开——这也是“方向可迁移、数值不可迁移” 那句话的操作性版本。
一个实用的推论：如果要读的论文只给了“我们用了 X”，没有说明 X 是结构性必需还是经验选择，**默认把它当成经验性的**。

2.31.3 一个本章没测但很重要的维度

数据规模。 本章固定 25 万字。当数据量变化时：

数据量	瓶颈
很少（< 10 万）	过拟合——该加正则化
中等（本章的情况）	**结构**——本章就是在这一档
很多（> 10 亿）	计算——该加容量

本章所有实验都在“中等”这一档。 这意味着：

- 我们关于“结构问题”的结论（残差、归一化位置）在这个档位成立
- 我们关于“要不要加正则化”的判断（§2.14：不加）**只在数据量少到中等时成立**——数据量再小就该加了

这是一个明确的边界，写出来是为了让你知道结论适用的范围。

2.32 常见问题

这些问题来自我们自己调试时的困惑，以及读者最可能踩的坑。

2.32.1 训练相关

Q：我的模型 loss 一开始就是 $\ln V$（比如 7.09），一直不动。

先查三处：① 输出层前面有没有 `ln_f`；② `head` 的权重是不是被正确初始化（如果误设成全零，logits 恒为零）；③ 学习率是不是太小（比如设成了 1e-6）。

如果三处都正常，试试把模型缩小到 2 层——如果 2 层能训而 12 层不能，问题在深度相关的部分（残差、初始化、归一化位置）。

Q：loss 正常下降，但验证 loss 从某个点开始上升。

这是**过拟合**——注意区分于本章案例的“欠拟合”（两者都高）。

判断标准：训练 loss 还在降、验证 loss 开始升，两条曲线的**差距在扩大**。这时加 dropout 或权重衰减是对的。

如果两条曲线同步升降，那不是过拟合，别加正则化。

Q：训练中途突然出现 nan。

第一步：找到第一个 nan 出现的位置（第 1 章 §1.16 给过四步插桩法）。

第二步：看它前面多少步的梯度范数。如果梯度范数在 nan 之前急剧上升，那是梯度爆炸——加梯度裁剪，或者降低学习率。

如果梯度范数一直正常却突然 nan，那是数值问题（溢出），去查 softmax 的稳定性、归一化的 ϵ、以及 dtype。

Q：换了个随机种子，结果差很多。

这是正常的——尤其在小模型上。本章实测：单次训练的波动在 0.1–0.2 之间（§2.10 排查二）。

判断标准：如果你的“改进”只有 0.05 的提升，那可能是噪声。**报结果时应该给多个种子的均值与范围，而不是单次结果。**

2.32.2 结构相关

Q：一定要用 Pre-LN 吗？我的模型只有 4 层。

4 层的话，Post-LN 也能训——本章的实测里，Post-LN 在 6 层时最优点是 4.732（用 3e-4 的学习率），而 Pre-LN 是 4.408。

但即使浅层，Pre-LN 的容错范围也更宽。除非你有明确理由（比如复现某篇论文），否则用 Pre-LN。

Q：RMSNorm 和 LayerNorm 该选哪个？

效果上没差别（本章实测差 0.007，在噪声内）。选 RMSNorm 的理由是**计算更省**—— 少一次求均值和一次减法，在大模型上能省可观的带宽。

但如果你的任务对“中心化”敏感（比如输入有系统性偏置），LayerNorm 更安全（见 §2.8 检查点 ③）。

Q：d_{ff} 该设成 d_{model} 的几倍？

经验值是 4 倍（原论文用的），但 2 倍、8 倍也都能工作。本章用的是 2 倍（因为模型小）。

怎么选：如果总参数预算固定，d_{ff} 和 n_{layer} 在争同一份预算。**小模型偏宽（大 d_{ff}），大模型偏深**（§2.12）。

Q：为什么注意力要 4 个矩阵（Q、K、V、O），不能少几个?

- **去掉 W_Q、W_K**：q、k 直接等于输入，那么打分矩阵就是 $XX^\top$ —— **完全没有可学参数**，模型学不到“该看谁”
- **去掉 W_V**：值就是原输入，注意力只能做“加权平均输入”，无法把取到的信息**变换**成需要的形式
- **去掉 W_O**：每个头直接写入残差流，失去了“各头独立投影”的自由度（§1.8 检查点 ⑧ 算过参数量）

四个都必要。但它们不是“四种不同的功能” —— 前三个构成“读”，最后一个负责“写”。

2.32.3 调试相关

Q：怎么知道我的注意力层有没有在工作?

看两个量：① 权重熵（健康值 0.5–0.9）；② 相邻位置输出的 KL 散度（接近 0 说明没在用上下文）。

如果熵正常但 KL 散度接近 0，那问题不在注意力层，而在它**前面**（比如位置编码缺失、输入被归一化掉了）。

Q：改了代码之后效果变差了，怎么定位是哪一处?

用 git bisect 的思路：把改动拆成几个独立的 commit，逐个回退找分界点。

如果改动没提交，那就手动二分：注释掉一半改动，看效果有没有恢复。

更快的办法：对每一处改动做单测。比如“改了归一化位置”——写个测试确认 `x = x + attn(ln(x))` 确实和改动前一样（或如预期不同）。

Q：模型很大，每次实验要跑几小时，怎么提高迭代速度?

三个手段，按性价比排序：

1. **缩小模型**。本章的所有结论都是用 12 层、170 万参数的模型得到的。如果你要验证的是**结构性**问题（比如“归一化放哪”），小模型上一样能看出来。
2. **减少步数**。如果两个配置在 200 步内就分道扬镳，没必要跑到 2000 步。**先用短实验筛选，再用长实验确认**。
3. **监控中间量而不是等最终结果**。本章反复强调：熵、激活尺度、梯度范数都能在 loss 显现之前给出信号。

注意第 1 条的前提：缩小模型的前提是“你要验证的是结构性结论”。如果是数值性结论（比如“最优学习率是多少”），**必须用目标规模**——学习率与模型大小强相关。

2.33 代码索引

本章涉及的全部脚本与数据，按用途列出。

2.33.1 实验脚本

文件	用途	输出
`notebooks/run_ch2_case.py`	案例的五个对照实验	`data/ch2_case.json`
`notebooks/rope_extrapolation.py`	位置编码外推对照	`data/rope_extrapolation.json`
`notebooks/analyze_ffn.py`	FFN 的键值记忆分析	`data/ffn_analysis.json`
`notebooks/block_variants.py`	Block 的三种变体（库）	—

2.33.2 复用第 1 章的脚本

文件	用途
`notebooks/tiny_gpt.py`	基础模型与训练循环（两个共用）
`notebooks/analyze_heads.py`	注意力头的画像
`notebooks/debug_failures.py`	失败模式注入实验

2.33.3 数据文件

文件	内容
`data/ch2_case.json`	案例的全部原始数据（52 KB）
`data/ch2_lr_sweep.json`	学习率扫描
`data/ch2_ablation.json`	残差消融
`data/ch2_context.json`	上下文长度对照
`data/rope_extrapolation.json`	位置编码外推
`data/ffn_analysis.json`	FFN 激活统计
`data/corpus.txt`	语料（25.5 万字，与第 1 章共用）

2.33.4 复现全部结果

```
cd /home/mpp/ai-mechanisms-book
PY=/home/mpp/micromamba/envs/dlspace/bin/python

$PY notebooks/run_ch2_case.py --steps 1200      # 约 8 分钟（含扫描与消融）
$PY notebooks/rope_extrapolation.py --steps 2000 # 约 3 分钟
$PY notebooks/analyze_ffn.py                     # 约 30 秒

python3 qa/gen_fig_ch2.py                        # 重新生成插图
python3 qa/lint.py chapters/02-block.qmd         # 静态检查
```

注意：这些脚本默认用 `cuda:1`（RTX 3090）。如果你只有一块卡，把 `block_variants.py` 里的 `device` 参数改成 `"cuda:0"` 或 `"cpu"`。加 `--cpu` 参数可以强制用 CPU（慢 5–10 倍）。

2.33.5 代码的组织原则

三个文件的分工：

- **`tiny_gpt.py` / `block_variants.py` 是库**—— 定义模型、训练循环、诊断函数。不含任何实验逻辑
- **`run_ch2_case.py` / `rope_extrapolation.py` 是实验**—— 调用库、跑对照、导出数据。**读它们能看懂实验设计**
- **`analyze_ffn.py` / `analyze_heads.py` 是分析工具**—— 加载训练好的权重，测量内部状态

这个分工让“改一个变量”变得容易：你只需要改实验脚本里的参数，不用碰模型定义。

一个建议：如果你想自己扩展实验，**新建一个脚本**而不是改现有的。这样本章的数据可以随时重新生成来对照。

2.34 本章回顾

2.34.1 主线：一个位置的价值

这一章的八节，讲的其实是同一件事的八个侧面：

```
① 一个 12 层模型训不动（loss 5.36，接近"只猜字频"）
        ↓  排查
② 换个 LayerNorm 的位置就好了（4.25）
        ↓  但这个对比不公平——两者用了同一套超参
③ 各自扫学习率后：最优解接近（4.703 vs 4.732），
   差别在**鲁棒性**（Pre-LN 单调改善，Post-LN 有窄峰）
        ↓  于是要问：为什么位置这么关键
④ 残差连接提供了恒等梯度路径——消融证明它连 2 层都不能省
⑤ LayerNorm 每步重置信号尺度，让每层面对同一量级
⑥ Pre-LN 把两条路径分开，Post-LN 把它们混在一起
⑦ RMSNorm 去掉中心化，效果不变但更省
⑧ FFN 是键值记忆，与注意力同构但键固定
        ↓
⑨ RoPE 外推实验：**没测准**——任务本身不依赖长程位置
```

2.34.2 五条可以带走的方法

方法	在本章的体现
比较两个方案要给同等的调参预算	差点把“学习率没调好”写成“Post-LN 训不动”
“数字太整齐”是退化解的信号	无残差时 2/4/12 层给出同一个 loss
先检查任务是否支持你的假设	RoPE 实验失败的根本原因
估计噪声水平再下结论	上下文长度实验里 16 层比 8 层更差 → 噪声 0.1–0.2
阴性结果 ≠ 否定结论	我们的 RoPE 实验只是“在这个规模上测不出”

2.34.3 为什么这一章写了这么多“过程”

你可能注意到，本章有大量篇幅在讲**我们怎么排查、怎么犯错、怎么修正**—— 而不是直接给结论。

这是刻意的。 因为这一章的核心内容——残差、LayerNorm、FFN—— 都是**已经确定的知识**，查文档就能知道。真正难的是：

- 面对一个不工作的模型，**怎么找到问题**
- 面对两个方案，**怎么设计一个能站住的对比**
- 面对一个“和论文不一致”的结果，**怎么判断该信谁**

这三件事没有文档可查，只能通过看别人怎么做来学。

而且本章恰好有三个“结论被推翻”的时刻： Post-LN 的结论（学习率没调好）、残差消融的异常（退化解）、 RoPE 外推的失败（任务选错）。**这些时刻比正确答案更有教学价值**。

所以如果你只记住了“要用 Pre-LN”，那是失败的阅读方式。 应该记住的是“比较两个方案时要给同等的调参预算”。

2.34.4 如果只记三件事

第一，残差连接不是可选项。 消融证明连 2 层都训不动。 它提供的那条恒等路径，是深层网络可训练的前提。

第二，归一化的位置比归一化本身更重要。同一个 LayerNorm，放在残差里面还是外面，决定了梯度要不要穿过它。 Pre-LN 胜出的不是“效果更好”，而是“容错范围更宽”。

第三，“鲁棒性”和“最优性能”是两个指标。 本章的案例里，Post-LN 在最佳学习率下和 Pre-LN 几乎持平—— 如果你只测那一个点，会得出“两者一样好”的结论。但扫一遍学习率就会发现，它们的可用区间宽度差了好几倍。

在真实项目里，你事先不知道最优超参数是多少。所以“容错范围”往往比“峰值性能”更值钱。

2.34.5 一条被推翻又重建的结论链

本章最有价值的部分，其实是**结论是怎么变的**。把这条链完整列出来：

```
第一次结论：
  "Post-LN 训不动"            ← 基于一组超参数的对比

反问：
  "这个对比公平吗？"          ← 两边用了同一套超参数，但敏感度不同

第二次结论：
  "Post-LN 对学习率敏感"      ← 扫描后：3e-4 处两者持平

再追问：
  "为什么敏感？"              ← 回到机制

第三次结论：
  "因为它的梯度必须穿过归一化，
   而 Pre-LN 有一条恒等路径"   ← 由残差消融和激活画像支持

最后修正：
  "更准确的说法是：Pre-LN 买到的是
   更宽的容错范围，不是更好的解"  ← 因为最优解接近
```

四步，每一步都推翻了上一步的一部分。

注意最后一步不是“回到原点”——它保留了“Pre-LN 更稳”这个核心判断，但把“更好”这个模糊说法精确成了“容错范围更宽”。

这就是“结论收敛”的样子：不是找到一个绝对正确的答案，而是逐步把模糊的表述替换成有边界的、可验证的表述。

2.34.6 与后续章节的接口

本章留下的问题	在哪一章解决
KV 缓存与推理优化	第 3 章
FFN 里存了哪些知识、能不能编辑	第 4 章
注意力与 FFN 的电路级分析	第 4 章
残差流视角的完整展开	第 4 章
归一化在训练稳定性中的其他角色	第 3 章

2.34.7 这一章和上一章的关系

第 1 章拆开了**注意力**这一个零件，第 2 章拆开了**围绕它的整个 Block**。两章的写法是刻意一致的：

- 都从一个**训练异常**出发（第 1 章：小模型上没复现出缩放的必要性；第 2 章：12 层模型学不会）
- 都靠**对照实验 + 中间量测量**定位（第 1 章：雅可比范数；第 2 章：激活与梯度画像）
- 都保留了**没测准或没做完的部分**（第 1 章的脚注前提、第 2 章的 d_{ff} 扫描）

这个一致性不是巧合，而是本书的方法：先给现象，再给测量，最后才给结论。如果读者把两章连起来读，应该能感觉到这套方法是可以独立使用的——换一个模型、换一个任务，步骤完全一样。

2.34.8 关于本章的“没做完”

本章有三处明确留白，这里集中列出来——**不是为了显得谦虚，而是让你知道哪些地方可以接手**。

① d_{ff} **该设多大（§2.12）**。我们只跑了 $d_{\text{ff}} = 256$ 一档，没有扫。参数量的账算清楚了，实验设计也给了，但结论是空的。

② **“注意力搬运、FFN 存储”的功能验证（§2.11 检查点 ⑦）**。我们验证的是**结构性**的（FFN 可以重写成键值形式、激活确实稀疏），但没有做**功能性**的实验（证明它“存储”了什么）。一个复制任务 + 一个事实补全任务的对照就能补上。

③ **batch size 的影响（§2.16）**。所有实验固定 batch = 32。真实训练用几百到几千。**这个差异会不会改变本章的结论，我们不知道。**

三处的共同点：都是“需要更多训练时间”的实验。本章的 28 次训练已经用掉了可观的机器时间（在 RTX 3090 上约 25 分钟），再加这三组大约要翻倍。

写出来的意义：一本教材如果把边界写清楚（哪些结论有实验支持、哪些只是方法），读者就知道在哪里可以继续往前推。**“这一章说清楚了什么、没说清楚什么”，比“看起来很完整”更有用。**

2.34.9 三个容易被忽略的细节

除了主要结论，本章还有三个小细节值得单独记一下—— 它们都不起眼，但都可能在真实项目里救你一次。

① **权重绑定必须在初始化之前。**

```
self.head.weight = self.tok_emb.weight    # 先绑定
self.apply(self._init)                    # 再初始化
```

顺序反了会被静默覆盖。 `self.apply` 会遍历所有子模块，`head` 和 `tok_emb` 各自被初始化一次——由于它们共享同一个张量，后初始化的那个说了算。**结果不是报错，而是“两个矩阵长得不一样”的微妙 bug。**

② **`pos_emb` 的越界是硬边界，不是渐变。**

序列从 64 变成 65 时，性能不是“略微下降”而是**断崖式变化**—— 第 65 个位置要么拿到一个训练时没见过的编码（可学习嵌入会直接报错），要么被截断到最后一个位置（静默出错）。

这是一条可以靠断言拦住的边界，加一句 `assert T <= self.block_size` 的成本几乎为零。

③ **归一化的 `eps` 在低精度下会失真。**

`eps = 1e-5` 在 float32 下是个正常的小数，但在 bf16 下（机器精度 7.8×10^{-3}）**它会和 0 无法区分。**

所以如果你的归一化是在原精度下算的（而不是像我们那样升到 float32），在 bf16 训练时 `eps` 实际上等于 0——除零风险回来了。

这条和 §2.5 讲的“归一化在 float32 下算”是同一个问题的两面。

2.34.10 这一章留下了什么

如果要用一句话概括本章的方法论收获：

“一个组件的位置，和这个组件本身同样重要。”

LayerNorm 的定义、参数、计算都没变——从残差外面挪到里面，同一个模型就从“完全学不会”（5.357）变成“正常训练”（4.245）。**没有加任何参数、没有改任何超参数，只是换了位置。**

这个观察可以推广到很多地方：

场景	“位置”指的是什么
归一化	在残差里面还是外面
dropout	在写入总线之前还是之后
位置编码	加在输入上还是旋转 q、k
梯度裁剪	在优化器更新之前还是之后

问“这个组件放在哪里”，往往比问“该不该用它”更有用。

2.34.11 三张可以带走的表

① **症状 → 病因对照表**（§2.15）：

症状	先查什么
loss 停在“太整齐”的值	是不是退化成常数输出

症状	先查什么
训练 loss 低、验证 loss 高	过拟合，加正则化
两者都高	欠拟合，**别加**正则化
logits 尺度异常大	缺 `ln_f`，或残差累积失控
深层梯度比浅层小很多	归一化位置（Pre 还是 Post）

② **超参数不能独立调的关系**（§ 2.16）：

- 学习率 ↔ 归一化位置（换位置必须重扫学习率）
- warmup ↔ 归一化位置（作用方式不同）
- 学习率 ↔ batch size（线性缩放）

③ **判断结论能否迁移**（§ 2.17）：

- **结构性结论**（“没有 X 就训不动”）→ 可以用在任何规模
- **数值性结论**（“最优值是多少”）→ 只用来定起点，然后自己扫

2.34.12 三个“如果重来一次”

如果要重做这一章的实验，有三件事我们会改：

① **每个配置跑三个种子**。

本章所有实验都是单次运行。这导致我们无法给出置信区间，也导致 § 2.10 那个 RoPE 实验的结论模糊——**我们只能说“差异没超过噪声”，但不能说“没有差异”**。

三个种子意味着三倍的时间。但**对于要写进书里的结论，这个代价是值得的**。

② **先做“任务诊断”，再做对照**。

RoPE 实验的失败，根因是我们先做了实验、后才发现任务不支持假设。**如果先花二十分钟测“这个任务需要多长上下文”，就能省下后面一个多小时的三个训练**。

一般的顺序应该是：**先确认任务能区分你的假设，再设计对照**。

③ **把诊断量的监控提前到第一次训练**。

我们是在排查“12 层训不动”时才加上那三个诊断量的（激活尺度、注意力熵、KL 散度）。但它们本来应该**一开始就有**。

代价几乎为零，收益是提前几百步发现问题。

2.34.13 下一章

第 3 章讲**训练与推理的两副面孔**。

本章的模型只做了训练——我们从来没有真正“用它生成过文本”（除了第 1 章末尾那几段演示）。而推理引入了一整套新的约束：KV 缓存、采样策略、上下文长度、以及在 batch=1 时的性能问题。

本章的“残差流是共享总线”那个视角，在讲到 KV 缓存时会再一次用上—— 因为缓存存的正是“总线上各个位置的历史状态”。

> 这一章真正要你记住的
>
> 不是“要用 Pre-LN”或者“要用 RMSNorm”。
>
> 而是：**一个组件的位置，和这个组件本身同样重要**。
>
> LayerNorm 的定义没变、参数没变、计算没变—— 只是从残差外面挪到了里面，同一个模型就从“完全学不会”变成“正常训练”。
>
> **这提示了一类容易被忽略的设计维度：不是“用什么零件”，而是“零件放在哪里、和谁相邻”**。第 3 章讲 KV 缓存时会看到同一个模式的另一个例子。

2.35 怎么做这类对照实验

本章的案例暴露了几个实验设计上的问题。这一节把它们整理成可复用的清单—— **因为“设计一个能说明问题的实验”比“跑实验”难得多**。

2.35.1 清单：做一个能站住的对照

按顺序检查这七条。**任何一条没做到，结论都要打折扣。**

① **明确你要回答的是哪个问题。**

本章最初的隐含问题是“Post-LN 好不好”。但这是个**无法回答**的问题——“好”在什么任务上、什么规模下、和什么比?

改成可回答的形式：**“在 12 层、$d_{\text{model}} = 128$、 25 万字唐诗、1200 步的条件下，Pre-LN 与 Post-LN 的最优 loss 和容错范围分别是多少?”**

② **钉死所有无关变量。**

我们的对照实验固定了：语料、随机种子、步数、批大小、优化器、学习率调度。唯一变化的是归一化位置。

“固定种子”这一条尤其重要——它让两边在 step 0 的 loss 完全相同（7.145）， 之后的差异只可能来自那个变量。

③ **给两边同等的调参预算。**

这是本章最大的教训。**同一个学习率下 Post-LN 看起来“训不动”， 但那是因为 3e-3 超出了它的可用范围。**

做法：对每个方案都扫同一组超参数，取各自最优，或者比较容错范围。

④ **估计噪声水平。**

怎么知道观察到的差异是真的?

本章的办法：**找一组“应该没有差异”的配置，看它们的差距有多大。**上下文长度实验里，16 层比 8 层更差（4.971 vs 4.846）—— 这两者没有理由差这么多，所以这个 0.125 就是噪声的估计。

结论：任何小于 0.2 的差异，在单次实验中都不能作为证据。

⑤ **用一个“平凡基线”锚定结果。**

本章用了两个：

- **均匀分布**：$\ln 1200 = 7.09$（“什么都没学”）
- **边缘分布**：5.672（“只学会字频”）

没有基线的 loss 数字是没有意义的。 4.2 是好是坏， 取决于“随便猜”能到多少。

⑥ **检查任务是否支持你的假设。**

这是 RoPE 实验失败的根因。我们想测“外推能力”， 但任务本身不依赖长程位置信息——**测量工具和被测量对象不匹配。**

做法：在做实验之前，先问“如果我的假设成立， 这个任务上会出现什么可观察的差异?”如果答不上来，先换任务。

⑦ **报出你没能控制的变量。**

本章明确写了：**batch size 固定为 32，没有扫。**这意味着“换 batch size 会不会改变结论”是未知的。

诚实地划出边界，比假装边界不存在更有价值。

2.35.2 三种常见的实验设计错误

错误一：用一组超参数比较两个方案

症状：A 比 B 好，但你没检查 B 的最优超参数是不是另一组。

本章的实例：我们差点把“学习率没调好”写成“Post-LN 训不动”。 扫描后发现，在 3×10^{-4} 处两者几乎持平（4.703 vs 4.732）。

为什么这个错误常见：它省事。扫四个学习率意味着四倍的训练时间。但**不扫的代价是可能得出完全错误的结论。**

错误二：把“没测出来”读成“不存在”

症状：实验没看到差异，就宣称“这个因素不重要”。

本章的实例：RoPE 实验里可学习嵌入在 256 长度上反而更好。如果就此宣称“可学习嵌入的外推能力比 RoPE 强”，那是错的—— 正确的说法是“**在这个任务、这个规模上测不出差异**”。

两者的区别：前者是一个普遍性结论，后者是一个有边界的观察。 **只有后者是实验支持的。**

检验方法：问“我的实验有多大把握能测出真实存在的效应?” （这在统计上叫**功效**，power）。如果实验本身不够灵敏， 阴性结果什么也说明不了。

错误三：在训练集上评估基线

症状：基线给出了一个“好得可疑”的数字。

本章的实例：我们用 n-gram 当基线时，在训练集上评估得到了 **0.006** 的 loss——那不是在预测，是在背答案。

自查习惯：如果某个数字好到不像真的，先问“它是不是见过这些数据”。

一个具体的判据：交叉熵 0.006 意味着模型对下一个字的预测几乎完全确定。自然语言里不存在这种确定性（人类标注者之间的一致性都远达不到）。**理论上不可能的数字，一定是评估方式有问题**。

2.35.3 一张实验设计自查表

贴在你开始跑实验之前：

检查项	本章的做法
问题表述是可回答的吗?	从“Post-LN 好不好”改成“在 X 条件下两者的最优 loss 与容错范围”
无关变量钉死了吗?	同种子、同语料、同步数、同批大小
两边调参预算对等吗?	各扫 4 个学习率
噪声水平估了吗?	用“应该无差异”的配置对估计（±0.125）
有平凡基线吗?	均匀 7.09、边缘 5.67
任务支持你的假设吗?	RoPE 实验**没有**——这是它失败的原因
边界说清楚了吗?	明确写了 batch size 未扫、单次实验无置信区间

七条里我们做到了五条，两条没做到。那两条（任务是否支持假设、边界）正是本章两处结论模糊的原因。

> 这一节的真正价值
>
> 前面二十节讲的是“Transformer 是什么”。这一节讲的是“你怎么知道它是什么”。
>
> **后者更耐用**。具体的结构会随架构演进变化（正弦编码→RoPE、LayerNorm→RMSNorm、MHA→GQA），但“怎么做对照实验”这件事，从 2017 年到今天没有变过。

（第 2 章完）

这一章的旅程到此结束，下一章见——那一章我们会第一次真正“使用”这个模型：让它一个字一个字地生成文本，而不是只在训练循环里被反复喂数据。

3. 训练与推理的两副面孔

> **本章地图**
> **你在这里**：数据流的两头 —— 第 5 步（怎么训）和第 7 步（怎么生成）。**本章解决**：同一个模型，为什么训练时一次前向算完所有位置、推理时却要一个词一个词来；KV 缓存省下了什么、又漏掉了什么。**前置**：第 1、2 章（知道一次前向里发生了什么）。**读完能做**：解释“生成 1000 个字为什么要 1000 次前向”；并判断一个缓存实现是否真等价于无缓存版本。

一个“会背诗”却“只会吐标点”的模型。

第 1、2 章我们一直在做同一件事：**把 loss 降下来**。到第 2 章结束，那个 2 层、4 头、d_model = 128、共 42 万参数的字符级模型，在验证集上把 loss 压到了 **4.141**——比“闭着眼睛按字频猜”的 5.67 好了 1.53。

按照前面两章的标准，这个模型是**训好了**的。

然后我们让它写诗。给定“春江潮水连海平”，贪心解码，生成 50 个字：

```
春江潮水海平。,。,。,
```

五十步里，四十五步是同一个 `<unk>`，剩下五步在「。」和「,」之间来回跳。

这个落差不是“模型还不够好”能解释的。同一套权重，`teacher forcing` 下它能给真实的唐诗打出 4.076 的 loss——它认得诗。可它自己写的时候，写出来的东西连“不通顺”都算不上，那是**一个死循环**。

这一章要回答的就是这个落差。答案不在权重里，而在**训练和推理是两副面孔**：训练时它一次看到整句话的所有位置，推理时它一次只能看到一个位置。我们一直以为这两副面孔是同一件事的两种算法，代价只是速度—— 这一章会证明，代价远不止速度。

> **i** 本章的代码与数据

```
# 需要 PyTorch。本机在 /home/mpp/micromamba/envs/dlspace/bin/python
python notebooks/run_ch3_case.py        # 本章全部实测数字
python notebooks/kv_cache.py            # KV 缓存的实现与四种 bug
```

数据导出到 `data/ch3_case.json`。本章正文里出现的每一个数字都来自这两个脚本。

3.1 先把现象摆清楚

3.1.1 起点：一个已经训好的模型

我们不改模型，不改数据，不改训练超参数。检查点是第 1 章留下的那个：全唐诗 4003 首、25.5 万字、字符表 1200（覆盖语料 87.6%），2 层 Transformer、4 个头、d_model = 128、上下文长度 64，训练 2000 步。

```
设备 cuda:1  参数量 425,984  上下文 64  词表 1200
验证集 loss = 4.141
```

作为参照，同一个分词器下“按字频瞎猜”的交叉熵是 5.67。也就是说，这个模型确实从语料里学到了东西——1.53 个 nat 的差距不是白来的。

3.1.2 现象：三种解码方式，三种完全不同的结果

同一个模型，同一个提示词“春江潮水连海平”（7 个字），生成 50 个字。差别只在**怎么从 logits 里挑下一个字**。

第一种：贪心（greedy）——每一步都取概率最大的那个字。

```
春江潮水海平。,。,。,
```

这个输出需要解释一下：`decode` 会把 `<unk>`（编号 1）丢掉不显示。实际上这 57 个 token 里，**45 个是 `<unk>`**，另外 12 个是「。」和「,」交替。把它显示全，长这样：

```
春江潮水海平 <unk>。,<unk>。,<unk>。,<unk>。,<unk>。,…
```

第二种：贪心 + 屏蔽 `<unk>`——不许它输出“不认识的字”。这是很多人遇到垃圾输出时的第一反应：把坏 token 禁掉。

```
春江潮水海平。不見昔年年不見，不見春風吹。不見君不得，不見君家。
不見君不見，不見，不如何處？何時？不見君不見，不
```

`<unk>` 一个都没有了。但**读一下这段话**：「不见」出现了 **8 次**。它从“吐标点”变成了“念咒”。

第三种：采样（sampling）——不取最大，按概率抽。温度 τ = 0.8：

```
春江潮水海平，狂唱山城秋。作城外由流風日裏，北樓殘月在晴雲。
葉春無一日來，攀折綠江中。清誰與一留人，一來少年春。
```

τ = 1.2：

```
春江潮水海平梅水盡山春浦。君城外由還見溪裏，北陌殘江在浪飛。
葉春無十夫江，攀折綠江柳。清誰與狂江三何處？少裁春月
```

这不押韵，也不成篇，但**它像诗**：有“山城秋”“北樓殘月”“攀折綠江柳”这样的意象组合，重复的字在正常范围内。

3.1.3 用数字说“有多重复”



图 3.1　七种解码策略的重复程度。条形长度是 distinct-2（越高越不重复），右侧标出出现至少三次的最长重复片段。贪心和「贪心+屏蔽 unk」都是红色，四种采样策略都是绿色——分界线不在模型，在「怎么选字」。

“看起来像不像诗”太主观。我们需要一个能算的指标。

distinct-n：所有长度为 n 的片段里，有多少是**不重复**的。一篇正常的中文文本，distinct-2（相邻两字组合的丰富度）通常在 0.9 以上；如果一段文本里有一半的二字组合是重复的，那它一定在念咒。

再加一个指标：**最长重复 n-gram**——出现了至少 3 次的、最长的那个片段有多长。正常文本这个数很小（4 已经算多了）；退化的文本会很长。

把七种解码策略都跑一遍（提示词相同，温度类参数固定随机种子）：

解码策略	distinct-1	distinct-2	distinct-3	最长重复	<unk> 占比
贪心	0.060	**0.102**	0.146	**8-gram**	88%
贪心 + 屏蔽 unk	0.340	**0.531**	0.667	4-gram	0%
$\tau = 0.8$	0.760	1.000	1.000	无	0%
$\tau = 1.2$	0.840	1.000	1.000	无	0%
top-k = 10	0.720	0.980	1.000	无	0%
top-p = 0.9	0.780	1.000	1.000	无	0%
top-p = 0.9, $\tau = 1.2$	0.760	1.000	1.000	无	0%

这张表在说什么

- **贪心**：distinct-2 = 0.102。一百个相邻二字组合里，九十个是重复的。最长重复片段有 **8 个字**——它不是在写诗，它是在循环播放一段 8 字的录音。
- **屏蔽 <unk> 治不了本**：distinct-2 从 0.102 升到 0.531，翻了好几倍，但离正常文本的 0.9 还差得远，而且最长重复从 8-gram 变成 4-gram—— **它还是在循环，只是换了个循环单元**。
- **只要引入随机性，退化就消失了**：四种采样策略的 distinct-2 全部 $\geqslant 0.98$，最长重复 n-gram 直接变成“无”。连最“保守”的 top-k = 10 都一样。

所以问题不在“模型不会写”，而在**“argmax”这个动作本身**。

> ℹ 停下来想一想 ①
>
> 上面那张表里，`top-k = 10` 的 distinct-2 是 0.980，没有达到 1.000。而 $\tau = 0.8$ 达到了 1.000。两者都在“引入随机性”，为什么 top-k = 10 仍然有重复片段？
> （提示：数一数 top-k = 10 的输出里「妾」字出现了几次。）

3.1.4 三个必须排除的嫌疑

在往下走之前，我们要排除三个“看起来很像元凶”的东西。

嫌疑一：模型没训好。

不成立。同一个模型，teacher forcing 下对真实唐诗的 loss 是 **4.076**。如果模型没学到东西，这个数应该接近 5.67。

嫌疑二：<unk> 这个 token 本身是坏的。

部分成立，但不是根因。看一眼语料就明白了：

```
语料里 <unk> 占 18.9%
```

字表只保留最高频的 1200 个字，覆盖 87.6% 的字符——**剩下 12.4% 全都映射到 <unk>**。所以在训练数据里，<unk> 本身就是一个出现频率接近 19% 的“字”。模型在某个位置把它选成 argmax，从统计上说**完全合理**：它确实是那个位置最可能的输出。

但屏蔽它之后，输出从“吐标点”变成了“念咒”——**根因不在这里**。

嫌疑三：模型进入了某个“坏状态”，是碰巧。

这是最需要认真对待的一个。如果只是碰巧，那换个提示词应该就好了。

> ⚠ 一个反直觉的验证
>
> 把提示词换掉，用贪心重新生成，会得到**完全不同的循环单元**—— 但**一定会有循环**。
>
> 我们试过以“床前明月光”“白日依山尽”“两个黄鹂鸣翠柳”开头，得到的循环单元分别是「不知」「山中日」「春風」这类二字或三字组合。循环的内容变了，**循环这件事没变**。
>
> 这不是“碰巧陷入坏状态”，这是**确定性解码在退化分布上的必然结果**。下一节开始解释为什么。

3.1.5 本章要回答的问题

现象摆完了，问题也就清楚了：

1. **为什么训练时一切正常，推理时却崩了？** 训练和推理到底哪里不一样？
2. **为什么是“循环”这么具体的形式？** 为什么不是“越写越乱”，而是“卡住不动”？
3. **为什么加一点随机性就好了？** 采样凭什么能打破循环？
4. **怎么让它跑得够快，快到能多试几次？** ——这一问引出 KV 缓存。

第 1 问和第 2 问在 3.2、3.3 节回答；第 3 问在 3.5 节；第 4 问在 3.4 节，而且答案会**出乎意料**（剧透：省了 28 倍计算，一点没变快）。

3.1.6 完整的实验设置

本章所有实验都基于同一个检查点，参数如下。列出完整配置是因为 **本章的每一个数字都依赖它**——换一个模型，这些数字会变。

项	值
语料	全唐诗 4003 首，25.5 万字
分词	字符级，字表 1200（覆盖率 87.6%）
层数	2
注意力头	4（每头 32 维）
d_{model}	128
d_{ff}	256
上下文长度	64
位置编码	**可学习的绝对位置嵌入**（不是 RoPE）
参数量	425,984
训练步数	2000
学习率	3e-3，warmup 100 步，余弦衰减
验证集 loss	4.141

三个选择需要说明。

为什么用这么小的模型？ 因为本章要讲的是**机制**，不是性能。 42 万参数的模型在本机 GPU 上跑一次完整实验只要几十秒， 可以反复做对照。而本章的核心结论（贪心必然循环、 缓存是恒等变换、顺序陷阱）**和模型规模无关**—— 换成 GPT-2 结论一样，只是数字不同。

为什么用字符级而不是 BPE？ 因为字符级让“下一个 token”这件事 **对人类可见**。你能直接读出“模型想写哪个字”， 不需要查 tokenizer。这对理解机制很有帮助。

代价是 `<unk>` 问题被放大了。 字表只保留 1200 个高频字， 覆盖 87.6%——剩下 12.4% 全变成 `<unk>`。 真实的 BPE 分词器覆盖率通常在 99.9% 以上，不会有这么严重的 `<unk>` 问题。

这个“缺陷”恰好成了本章案例的引子（见 3.1.2 节）， 但它确实是一个**本可以避免**的人为因素。读到后面请记住这一点。

> **i** 停下来想一想 ②
>
> 上面的设置里，“上下文长度 64”和“位置编码用可学习的绝对位置嵌入”这两项**共同**导致了 3.9.6 节那个 CUDA 崩溃。
>
> 如果换成 RoPE，那个崩溃还会发生吗？
>
> （答案：**不会以同样的形式发生**。
>
> RoPE 不给每个位置存一个嵌入向量，而是**按位置旋转** Q 和 K。所以没有“第 65 行不存在”这个问题——位置 65 只是旋转角度不同而已，计算上完全合法。
>
> **但会有别的问题**：模型训练时只见过位置 0–63 的旋转角度，位置 65 的旋转是**外推**。它的行为没有保证—— 可能还行，可能完全乱掉。
>
> **所以 RoPE 把“硬崩溃”换成了“静默劣化”**。回顾 3.6.3 节那张表：**静默劣化比崩溃更难发现**。
>
> 这是个反复出现的权衡：**让错误早点暴露，还是让它晚点暴露？**本书的立场是前者——宁可崩，不可悄悄错。）

3.1.7 补充排查：换提示词、换模型、换热身

3.1.4 节排除了三个嫌疑。还有三个必须排除的，否则“这是确定性解码的必然”这个结论站不住。

排查四：换提示词。

如果循环是“这个提示词碰巧触发的”，那换个开头应该就好了。

实测：以「床前明月光」「白日依山尽」「两个黄鹂鸣翠柳」开头，贪心输出分别陷入「不知」「山中日」「春風」的循环。**循环单元变了，循环这件事没变。**

排查五：换温度以外的解码参数。

如果只是 `top-k`/`top-p` 的某个配置有问题，那不用它们应该就好了。

实测：不用任何截断，只用温度（$\tau = 0.8$、$\tau = 1.2$），distinct-2 都是 1.000。**问题不在截断，在“是否随机”。**

排查六：换模型规模——这一条最难，但最有说服力。

如果这是“42 万参数的小模型太笨”，那大模型应该不会。本章没有重训一个大模型（成本不允许），但有两类旁证：

旁证一：第 2 章的实验里，同一个模型在不同深度下（6 / 12 / 24 层）的贪心输出都退化。**深度不改变结论。**

旁证二：文献。Holtzman 等人 2020 年在 **GPT-2（15 亿参数）**上复现了同样的现象——贪心和 beam search 都退化，只有采样能解决。**规模差 3500 倍，结论一样。**

> 为什么“换模型”这条必须认真对待
>
> 在排查中，“是不是模型太小了”是一个**万能的借口**。它可以解释任何失败，因此**什么都不解释**。
>
> **排除它的方法有两个**：1. **换一个真正不同的模型**（本章做不到，成本不允许） 2. **找到文献里的同现象**（本章的做法）
>
> 第 2 种方法的价值在于：**如果别人在 3500 倍大的模型上看到同样的事，那就不是规模问题。**但它有个前提—— **你得真的去读那篇论文，而不是凭印象引用。**
>
> 本章 3.11 节列出的是我们实际读过并核对过的文献。

六条排查的总结

#	嫌疑	结论	依据
1	模型没训好	排除	teacher forcing loss 4.076
2	`<unk>` 是坏的	部分成立，非根因	屏蔽后仍在循环
3	碰巧进入坏状态	排除	换提示词仍循环，只是换了单元
4	特定提示词触发	排除	同 3
5	截断参数配置错	排除	只用温度也是好的

#	嫌疑	结论	依据
6	模型太小	排除	第 2 章多深度 + 文献佐证

六条排查，每一条都对应一个具体的验证动作和一组具体数字。这就是“排查”和“猜”的区别——**猜不需要数字，排查需要**。

3.1.8 怎么复现这一章的全部数字

本章出现的每一个数字都来自两个脚本。复现步骤：

```
cd /home/mpp/ai-mechanisms-book

# 全章实测数据（约 3 分钟，主要花在时序基准上）
/home/mpp/micromamba/envs/dlspace/bin/python notebooks/run_ch3_case.py

# 缓存实现与四种 bug 的最小示例（约 30 秒）
/home/mpp/micromamba/envs/dlspace/bin/python notebooks/kv_cache.py
```

输出：`data/ch3_case.json`（约 12 KB），包含：

键	内容	对应正文
`decoding`	七种解码策略的退化指标与文本	3.1.2 / 3.1.3
`exposure`	扰动实验与置信度对比	3.7.2 / 3.7.3
`cache_exact`	预填充与增量的逐位差异	3.4.3
`cache_bugs`	五种实现的 logits 差异	3.9.1
`mask_noop`	掩码在单步 vs 分块下的差异	3.9.4
`cache_timing`	batch 1/16/64 的墙钟	3.4.4
`cache_scaling`	两种模型规模的长度扫描	3.4.4
`sampling`	熵-温度曲线、核曲线、顺序陷阱	3.5.2 / 3.5.4 / 3.5.5

跑之前先看这三件事

一、设备。脚本默认 `cuda:1`（本机的 3090）。A6000（`cuda:0`）显存常被占 85%，不要用。加 `--cpu` 可以强制 CPU——**时序数字会完全不同，但所有正确性数字（logits 差异、退化指标）一个都不变**。

二、时间。全量跑约 3 分钟。其中 `cache_scaling` 占了大头（要建两个随机权重的模型并做长度扫描）。如果只想验证正确性，把 `main()` 里那两段 `cache_scaling` 删掉，30 秒就能跑完。

三、随机性。所有采样都用了固定种子，所以**同一个 `(提示词，参数)` 组合永远给同样的输出**。这是刻意的——本书里引用的每一段生成文本，你都能原样复现出来。

如果复现不出来，最可能的原因是 PyTorch 版本不同导致 `torch.multinomial` 的随机数序列不同。这时请以 **退化指标（distinct-n、最长重复）** 为准——它们对具体抽到哪个字不敏感，而正文的结论都建立在这些指标上。

3.1.9 本章的路线图

本章的结构是“从现象到机制再到工程”，每一节都在回答 3.1.5 节提出的 四个问题之一。先给一张地图，读的时候可以随时回来看。

节	回答哪个问题	一句话结论
3.2 训练那一面	训练和推理哪里不一样	训练一次算完所有位置，用因果掩码防止作弊

节	回答哪个问题	一句话结论
3.3 推理那一面	为什么推理必须串行	要预测第 $t+1$ 个，需要第 t 个；而第 t 个还没生成
3.4 KV 缓存	怎么让它跑得快	过去位置的 K/V 永不变（可证明），存下来即可
3.5 采样的数学	为什么加随机性就好了	循环来自 argmax 的确定性；采样打破它
3.6 上下文长度	硬边界在哪	位置嵌入表是硬墙，注意力的 $O(T^2)$ 是软墙
3.7 训练与推理的错配	代价是什么	训练时看真实前文，推理时看自己造的前文
3.8 交互实验	—	三个可拖动的实验
3.9 失效边界	—	缓存写错的六种样子 + 排查手册
3.10 上手实现	—	一个 20 行的缓存 + 一个完整的生成器
3.11 延伸阅读	—	五篇论文，每篇一句话说清读它能得到什么

读这一章需要的前置知识

必须有：- 第 1 章的注意力计算（$QK^\top$、softmax、加权求和）- 第 2 章的 Block 结构（残差、LayerNorm、FFN）- 基本的矩阵形状感觉（哪一维是 batch、哪一维是时间）

最好有：- 交叉熵的定义（没有的话 3.2.1 节会补）- 概率论里的期望和条件分布记号

不需要：- 任何关于“推理优化”的先验知识。本章从零开始。- 读过 Transformer 原论文。所有需要的地方都会重述。

如果第 1、2 章没读：3.2 和 3.3 节仍然能读—— 它们讲的是“同样一次前向，训练和推理的用法有什么不同”， 不依赖注意力的内部细节。但 3.4 节之后建议先补第 1 章。

三条阅读路径

路径 A（只想搞懂现象）：3.1 → 3.3 → 3.5 → 3.7。 大约 40% 的篇幅，回答“为什么会循环、为什么采样有用”。

路径 B（要做推理优化）：3.1 → 3.3 → 3.4 → 3.6 → 3.9 → 3.10。 跳过采样和错配，直奔缓存和它的失效边界。

路径 C（完整读）：从头到尾。3.7.7 节是全章的收束， 建议不要跳过——它把前面所有现象归结到一个根源上。

三条路径都要读 3.1。 没有案例的具体数字， 后面的推导会失去锚点。

3.2 训练那一面：一次算完所有位置

3.2.1 目标函数：把“预测下一个词”变成一个数

先把训练在做什么写清楚。

给定一段文本 $x_1, x_2, \ldots, x_T$，语言模型的训练目标是：**在每一个位置 t，用前 t 个字符预测第 $t+1$ 个字符**。

模型在位置 t 输出的是一个长度为 V 的向量 $\mathbf{z}_t \in \mathbb{R}^V$， 叫 **logits**（逻辑值）。$V$ 是词表大小，本模型 $V = 1200$。 经过 softmax，它变成“下一个字是第 v 个字”的概率：

$$p_\theta(x_{t+1} = v \mid x_1, \ldots, x_t) = \frac{\exp(z_{t,v})}{\sum_{u=1}^{V} \exp(z_{t,u})} \tag{3.1}$$

所以这个式子在说：模型对每个可能的字给一个分数，softmax 把分数变成概率。 分数高的字概率大。分母那个求和是为了让所有概率加起来等于 1。

训练用的损失是**交叉熵**——真实下一个字的负对数概率，在所有位置上取平均：

$$\mathcal{L} = -\frac{1}{T}\sum_{t=1}^{T} \log p_\theta(x_{t+1} \mid x_1, \ldots, x_t) \tag{3.2}$$

所以这个式子在说：如果模型给真实的下一个字的概率是 1，这一项就是 0（不罚）；给的概率越小，罚得越狠，而且是对数尺度地狠——给 0.01 的惩罚是给 0.5 的两倍多。

用一个具体数字走一遍

假设某个位置，模型输出的 logits 是 $[2.0, 1.0, 0.0]$，真实答案是第 1 个（下标 0）。

三个指数：$e^{2.0} = 7.389$，$e^{1.0} = 2.718$，$e^{0.0} = 1.000$。 分母 $= 7.389 + 2.718 + 1.000 = 11.107$。

概率：$p_0 = 7.389/11.107 = 0.665$，$p_1 = 0.245$，$p_2 = 0.090$。

交叉熵 $= -\log(0.665) = 0.408$。

如果模型更自信一点，logits 是 $[4.0, 1.0, 0.0]$： 分母 $= 54.598 + 2.718 + 1.000 = 58.316$，$p_0 = 0.936$，loss $= 0.066$。 **“对的字概率更高”这件事，在对数尺度上被放大了。**

反过来，如果模型押错了宝，logits 是 $[0.0, 1.0, 2.0]$（把第 3 个当成了答案）： $p_0 = 0.090$，loss $= -\log(0.090) = 2.41$。**罚了 6 倍。**

3.2.2 把 teacher forcing 跑一遍：逐位置的实况

抽象讲完了，看真实数据。

从验证集里取一段 10 个字的真实文本：「**已驚飛。方從列子御**」（第 4 个字是 <unk>，也就是“字表里没有这个字”）。

模型对每个位置的预测如下。第二列是“到这一位为止的上文”， 第三列是真实答案，第四列是交叉熵损失，最后是模型给出的前三个候选：

位置	上文	真值	损失	模型的前三候选（概率）
0	已	驚	6.379	「,」0.142 <unk> 0.097 「。」0.028
1	已驚	飛	5.184	「。」0.274 <unk> 0.101 「,」0.085
2	已驚飛	。	**1.676**	「。」0.187 <unk> 0.107 「,」0.038
3	已驚飛。	<unk>	**0.000**	**<unk> 1.000** 「不」0.000 「日」0.000
4	已驚飛。	方	6.013	<unk> 0.145 「日」0.017 「不」0.009
5	已驚飛。方	從	4.533	<unk> 0.182 「知」0.102 「聞」0.029
6	已驚飛。方從	列	6.265	<unk> 0.163 「萬」0.016 「有」0.016
7	已驚飛。方從列	子	**8.184**	<unk> 0.175 「玉」0.038 「羽」0.033

位置	上文	真值	损失	模型的前三候选（概率）
8	已驚飛。方從列子	御	6.984	<unk> 0.257 「,」 0.064 「重」 0.016

整段平均损失 **5.024**，困惑度 **152.07**。

这张表里有三件事，每一件都比它看起来更重要

第一，位置 3 的损失是 0.000476——模型给出了 1.000 的概率。

它**完全确定**下一个字是 <unk>。而且它确实“对了”，因为真实答案就是 <unk>。

但请想一想这意味着什么：**模型学会了一件完全正确、却毫无用处的事**——“当一个位置的字不在字表里时，就输出 <unk>”。语料里 18.9% 的位置都是这种情况，所以这个策略的期望收益非常高。

这就是 3.1 节贪心解码输出 88% <unk> 的直接原因。不是模型坏了，是它在**忠实地执行训练数据教给它的最优策略**。

第二，位置 7 的损失是 8.184——模型认为「子」的概率不到千分之一。

$e^{-8.184} \approx 2.8 \times 10^{-4}$。而它给 <unk> 的概率是 0.175，**差了 600 倍**。

「方從列子御」是一句正常的诗（列子御风，出自《庄子》）。模型不是不懂这句话——它在位置 5 给「知」0.102、位置 1 给「。」0.274，说明它对诗的语感是有的。**它是在“具体是哪个字”上被字表卡住了。**

第三，损失在 0.000 到 8.184 之间横跨了四个数量级。

平均值 5.024 是一个**极具误导性**的汇总。如果用困惑度 152 来描述这个模型，你会以为它“在所有位置都很差”；实际上它在标点位置（位置 2，损失 1.676）相当准，在罕见字位置（位置 7，损失 8.184）完全抓瞎。

这就是为什么本书反复强调要看分布，不要只看均值。

> **i** 停下来想一想 ③
>
> 位置 3 的损失是 0.000476，看起来几乎是 0。为什么不是**恰好** 0?
>
> （提示：模型给 <unk> 的概率是 1.000，但那是**显示到三位小数**的结果。真实概率是 $1-\epsilon$，其中 ϵ 是个很小的正数—— softmax 永远不会输出**精确的** 1，因为它要除以一个包含所有 1200 项的和。
>
> $-\log(1-\epsilon) \approx \epsilon$，所以损失 0.000476 意味着 真实概率约 $1 - 4.76 \times 10^{-4}$。
>
> **这是浮点数和 softmax 的性质，不是模型“不确定”。**

3.2.3 关键问题：为什么训练能一次算完

现在到了本章的核心对比。

式 式 (3.2) 里有一个求和，t 从 1 到 T。最直接的做法是写个循环：

```
loss = 0
for t in range(T):                    # T 次前向
    logits = model(x[:t+1])           # 只看前 t+1 个字符
    loss += -log_softmax(logits[-1])[x[t+1]]
loss /= T
```

这是推理的写法（下一节会讲）。训练时的写法完全不同：

```
logits, loss = model(x)               # 1 次前向，T 个位置同时出结果
```

一次前向，输入是完整的 `(B, T)`，输出的 `logits` 形状是 `(B, T, V)`—— **每一个位置都有自己的预测**。然后一次性算交叉熵。

为什么可以这样？因为 Transformer 在位置 t 的输出，**只依赖于位置 1…t 的输入**。只要保证这一点，把整句话一起喂进去，第 t 个位置的输出和“只喂前 t 个字符”得到的输出就是同一个东西。

保证这一点的机制就是**因果掩码**（causal mask）。

掩码在做什么

注意力那一层，位置 t 可以“看”所有位置。如果不加限制，它在算第 1 个字的预测时就会偷看第 2、3、4 个字——**作弊**，而且推理时根本拿不到未来的字。

因果掩码的做法：在 softmax 之前，把所有“未来位置”的打分设成 $-\infty$。softmax 之后 $e^{-\infty}=0$，那些位置的权重就真的变成 0 了。

用矩阵看，一个 $T=4$ 的掩码长这样（1 = 允许看，0 = 屏蔽）：

$$M=\begin{pmatrix}1&0&0&0\\1&1&0&0\\1&1&1&0\\1&1&1&1\end{pmatrix} \tag{3.3}$$

第 1 行只能看第 1 列，第 2 行能看前 2 列……**第 4 行能看全部**。每一行恰好对应“从这个位置往回看，历史有多长”。

这个矩阵是**下三角**的，而且对角线全是 1。它的形状本身就说明了因果性：位置越靠后，能看到的历史越长；没有任何一行能看到自己右边的位置。

> **i** 停下来想一想 ④
>
> 式 式 (3.3) 的矩阵里，第 4 行全是 √。这看起来像“第 4 个位置偷看了未来”—— 毕竟它能看到全部 4 个位置。
>
> 它作弊了吗?
>
> （提示：第 4 个位置**预测的是第 5 个字**，不是第 4 个字。它能看到的位置 是 1、2、3、4——全都是它的“过去”。）

3.2.4 一次前向，T 个损失，为什么不重复计算

真正让并行训练成为可能的，是一个更朴素的事实：

位置 t 的表示，和位置 t' 的表示，在前向传播中是各自独立算出来的。

自注意力把 T 个位置一次性投影成 Q,K,V 三个矩阵，然后是一次矩阵乘法 $QK^\top$。前馈网络是逐位置的，T 个位置各自过一遍同样的 MLP。**没有任何一步需要“先算完位置 1 再算位置 2”。**

于是整个训练前向的计算量是：

- 注意力：$O(T^2\cdot d)$（每对位置算一次点积）
- 前馈：$O(T\cdot d^2)$

注意注意力那一项是 T^2——**这是“一次算完所有位置”的代价**。第 3.4 节会看到，KV 缓存省的正好就是这一项。

3.2.5 回到案例：训练时模型看到的前文，全是真实的

现在可以指出案例的关键了。

训练时，位置 t 的输入是 $x_1,\dots,x_t$——**这段文本的每一个字都是真实的**。模型在这个前提下被要求预测 x_{t+1}。它从来没见过“自己写错一个字之后的前文”。

用本模型实测：

```
真实文本上，模型对自己 argmax 的平均置信度 0.239  平均 NLL 2.203
```

0.239 意味着模型押的那个字，平均只有 24% 的概率——**它自己也不太确定**。这很正常，字符级语言模型本来就难。

但注意另一组数：

```
自由生成时，模型对自己所选 token 的平均置信度 0.157  平均 NLL 2.135
```

在自由生成时，模型比在真实文本上更“自信”（NLL 2.135 < 2.203）。而它写出来的东西是退化的重复文本。

这个反差是本章的第一个重要线索：**模型的自信和它的正确性脱钩了**。原因就在于，生成时它看到的前文是**它自己造的**——一个训练时从未出现过的分布。3.7 节会把这个现象定量算出来。

3.2.6 一个容易忽略的细节：哪些位置该算损失

式 式 (3.2) 里的求和 $t = 1$ 到 T 看着很干净，但真实训练里有个坑：**一个 batch 里的样本长度不一样，短的要 padding**。

本案例的语料是连续文本流，每次切 64 个字符，长度天然一致，所以没这个问题。但真实场景（变长对话、变长文档）一定会有。

假设一个 batch 里有两条样本，长度分别是 5 和 3，padding 到 5：

```
样本 A: 白 日 依 山 尽
样本 B: 春 江 潮 <pad> <pad>
```

如果照式 式 (3.2) 算，模型会被要求**预测 <pad>**。而 <pad> 不是真实内容，预测它毫无意义。更糟的是：**如果 padding 占的比例高，模型的梯度会被“预测 pad”这件事主导**—— 它会学到一个完全没用的技能，而且这个技能会挤占真正有用的容量。

正确做法：用一个掩码把 padding 位置的损失置零。

$$\mathcal{L} = \frac{\sum_{t=1}^{T} m_t \cdot \big(-\log p_\theta(x_{t+1} \mid x_{1:t})\big)}{\sum_{t=1}^{T} m_t} \tag{3.4}$$

其中 $m_t = 1$ 表示位置 t 是真实 token，$m_t = 0$ 表示是 padding。

所以这个式子在说：分子只累加真实位置的损失，分母是真实位置的个数。

注意分母也变了——从 T 变成 $\sum m_t$。如果忘了改分母，则 loss 会被 padding 的比例稀释，**你会以为模型训得比实际好**。

一个真实的数字感受

假设一个 batch 里平均 60% 是 padding（在处理变长对话时很正常）。

- 正确实现：loss = 真实位置的平均损失，比如 3.0
- 分母忘了改：loss = $3.0 \times 0.4 = 1.2$

1.2 看起来比 3.0 好得多，你会以为模型收敛得很快。实际上**什么都没变**——只是分母算错了。

这个 bug 的危险性在于它**让指标变好**。所有“让指标变好”的 bug 都会被监控曲线掩盖过去，直到你拿模型去推理，发现它什么都没学会。

本书第 1 章遇到过同类问题：掩码写错让 loss 变好（4.634 vs 4.810），因为模型在偷看答案。**“loss 异常地低”永远比“loss 异常地高”更值得警惕**。

> **i** 停下来想一想 ⑤
>
> 上面说 padding 位置的损失要置零。那么 **padding 位置的输入**呢？要不要也置零？
>
> （答案：通常不用，但要小心。
>
> 输入端的 padding 一般用 <pad> 的嵌入，模型会把它当成一个普通的 token。因为我们不计算 padding 位置的**输出**损失，所以它不影响梯度—— **但它会影响注意力**。
>
> 具体说：位置 6 的查询会关注到位置 4、5 的 <pad>。如果不管，模型会把注意力分给这些无意义的键。
>
> **这就是“注意力掩码”存在的原因**——它和“损失掩码”是两件事：- 损失掩码：决定**哪些位置的预测被计分**（影响损失）- 注意力掩码：决定**哪些位置可以被看到**（影响前向）
>
> 很多实现在这里出错：**只做了其中一个**。第 1 章讲过的“掩码 bug”就是注意力掩码写错了——症状是 loss 变好，因为模型偷看了不该看的位置。）

3.2.7 掩码怎么实现：−inf 还是大负数

式 式 (3.3) 是个数学定义。**代码里怎么实现它，有一个真实的坑**。

正确做法：填 $-\infty$。

```
att = att.masked_fill(mask == 0, float("-inf"))
att = F.softmax(att, dim=-1)
```

softmax 里 $e^{-\infty} = 0$，所以被屏蔽的位置权重**严格为 0**。

错误做法：填一个“足够小的数”。

```
att = att.masked_fill(mask == 0, -1e9)      # 看起来够小了
att = F.softmax(att, dim=-1)
```

e^{-10^9} 在 float32 下**下溢成 0**，所以结果**通常**是对的。

为什么“通常对”是危险的

`-1e9` 的写法有三个问题：

问题一：`-1e9` 加上别的东西会变。 如果 att 里有其他运算（比如 ALiBi 的线性偏置、相对位置偏置），`-1e9 + 5000` 还是 `-1e9`，**不影响**。但如果偏置本身量级到了 10^9（数值不稳定时会），就可能“抬起”这个值。

问题二：它破坏了 softmax 的数值稳定性技巧。 标准实现会先减去每行的最大值再取指数：

```
att = att - att.max(dim=-1, keepdim=True).values
att = att.exp()
att = att / att.sum(dim=-1, keepdim=True)
```

如果被屏蔽的位置是 `-1e9`，减完之后它变成 $-10^9 - \max$，**仍然下溢成 0**，没问题。但如果**整行都被屏蔽**（某个位置看不到任何东西），`max` 就是 `-1e9`，减完变成全 0，softmax 输出**均匀分布**——**一个本应是“全 0”的行变成了“全 1/4”**。

问题三：`-inf` 会被显式地检查出来。 `-inf` 参与运算会产生 `nan`，而 `nan` 会**立刻**让训练崩掉——这是一个**响亮的失败**。`-1e9` 产生的是**静默的错误值**。

又是“响亮失败 vs 静默降级”的权衡（见 3.6.3 节）。本书的立场不变：**宁可崩，不可悄悄错。**

> 那“整行被屏蔽”怎么办
>
> 上面的问题二提了一个真实场景：**某些位置看不到任何东西**。
>
> 什么时候会发生？——**padding**。如果一个 batch 里某条样本特别短，它后面的位置全是 padding。如果注意力掩码把这些位置**全部**屏蔽了，那一行就没有任何有效键。
>
> **正确做法：不要把整行屏蔽光。** 通常保留“位置可以看到自己”（对角线永远为 1），这样每行至少有一个有效键。
>
> 这也是 3.2.6 节那个“停下来想一想”的答案：**注意力掩码和损失掩码是两个独立的掩码**，而注意力掩码的设计要保证“每行至少有一个 1”。

3.3 推理那一面：一次只能算一个

3.3.1 自回归的死结

训练可以并行，推理不行。原因简单到有点扫兴：

要预测第 $t+1$ 个字，需要第 t 个字的输出。而第 t 个字的输出，需要第 t 个字本身。可第 t 个字还没生成。

这不是工程问题，是任务定义本身的性质。生成的定义就是“用已经生成的去生成下一个”，所以它在**时间上必然串行**。

写成代码就是最朴素的样子：

```
out = prompt_ids
for _ in range(n_new):
    logits, _ = model(out)              # 把整段前缀重算一遍
    nxt = logits[:, -1, :].argmax(-1, keepdim=True)
    out = torch.cat([out, nxt], dim=1)
```

注意第 3 行：每一步都把**整段前缀**重新跑一遍前向。第 1 步算 7 个位置，第 2 步算 8 个，第 3 步算 9 个……

这个写法有多浪费

生成第 t 个字的时候，前 $t-1$ 个位置**一个字都没变**，可它们的表示被从头算了一遍。

而且注意：**在这个模型里，“位置 t 的表示”在生成过程中是永远不变的。**第 5 步算出来的位置 3 的表示，和第 4 步算出来的位置 3 的表示，和最终第 50 步算出来的位置 3 的表示——**完全一样。**

为什么？因为因果掩码保证了位置 3 只看得到位置 1、2、3。后面追加多少 token，都影响不到它。

所以这些计算不是“算得慢”，是**纯粹的重复劳动**。

3.3.2 两副面孔的计算量对比



图 3.2　训练与推理的两副面孔。上：训练一次前向覆盖所有位置，注意力是 T×T 的矩阵，每个位置都有输出。下：推理一次只算一个位置，深色的格子是「这一步真正需要的」，浅色的是被朴素写法白白重算的。

把两边的计算量写出来，差距有多大就一目了然了。

生成 N 个字，提示词长度 P：

	每次前向处理的 token 数	前向次数	总 token 数
朴素推理	$P, P+1, P+2, \ldots, P+N-1$	N	$\sum_{i=0}^{N-1}(P+i)$
训练式并行	$P+N$	1	$P+N$

本案例：$P=7$，$N=50$。

朴素推理的总 token 数 $=\sum_{i=0}^{49}(7+i)=50\times 7+\frac{49\times 50}{2}=350+1225=1575$。

如果有一个“上帝视角”的写法能一次算完，只要 57 个 token。**1575 / 57 ≈ 27.6 倍**。

实测（脚本里直接数）：

```
朴素（每步重算前缀）：  1,060 次 token 前向
带 KV 缓存          :     46 次
减少 23.0 倍
```

（这里的 46 = 预填充 7 次 + 之后每步 1 次 × 39 步，比 57 少是因为 `generate_cached` 的第一次预测复用了预填充的输出。）

> **i** 停下来想一想 ⑥
>
> 上面那个 $\sum_{i=0}^{N-1}(P+i)$ 里，当 N 远大于 P 时，它约等于 $N^2/2$。也就是说**朴素推理的总计算量随生成长度平方增长**，而缓存版本是线性的 N。
>
> 如果生成 1000 个字（P 忽略不计），两者差多少倍？如果是 10000 个字呢？
>
> （答案：500 倍和 5000 倍。注意这是个**随长度增长**的差距—— 短文本上感觉不到，长文本上是数量级的差别。）

3.3.3 推理的两个阶段：prefill 和 decode

在讲缓存之前，需要先分清推理的两个阶段。**它们的瓶颈完全不同**，不分清就理解不了 3.4.4 节那个反直觉的实测结果。

阶段一：prefill（预填充）。用户给了一段提示词，模型要把它“读进去”。这一段是**一次前向算完的**——和训练时一模一样，因为提示词的所有 token 都是已知的。

阶段二：decode（解码）。逐字生成。每个新 token 都要一次前向，而且**必须等上一个算完**。

两者的计算特征：

	prefill	decode
一次前向处理几个 token	P（提示词长度，可能几千）	**1**
矩阵形状	大而“方”	极瘦（$1 \times d$）
算术量	大	极小
瓶颈	**算力**（compute-bound）	**显存带宽**（memory-bound）
GPU 利用率	高（几十 %）	**极低（个位数 %）**

为什么 decode 是“显存带宽受限”

想象一次 decode 前向要做什么：把**整个模型的所有权重** 从显存读进计算单元，然后**只做一次** $1 \times d$ 的矩阵乘法。

用一个具体数字。本案例的模型有 425,984 个参数，float32 占 1.7 MB。3090 的显存带宽约 936 GB/s，所以把 1.7 MB 读完要 $1.7 \times 10^6 / 936 \times 10^9 \approx 1.8$ 微秒。

而算术量是 $2 \times 425984 \approx 8.5 \times 10^5$ 次浮点运算，在 35 TFLOPS 上是 0.024 微秒。

算得比读得快 75 倍。也就是说，decode 阶段 GPU 有 98.7% 的时间 在**等数据从显存里搬过来**，计算单元基本闲着。

这就是为什么 KV 缓存省了 68 倍计算，墙钟只快 1.07 倍—— 省下来的都是本来就空闲的算力。

⚠ 一个重要的推论：batch 是唯一的解药

如果 decode 的瓶颈是“读权重”，那么**读一次权重应该服务尽量多的请求**。

这正是真实推理服务的做法：把多个用户的请求拼成一个 batch，一次前向算 B 个序列的同一个位置。

- $B = 1$：读 1.7 MB 权重，做 1 次乘法 → 浪费
- $B = 64$：读 1.7 MB 权重，做 64 次乘法 → **权重读取被摊薄了 64 倍**

注意这和 KV 缓存是完全正交的两件事：

优化	省什么
KV 缓存	每次前向里的 **token 数**（算力）
Batching	每次前向里的 **权重复用次数**（带宽）

本章 3.4.4 节的实测里，batch 从 1 提到 64，朴素写法的耗时几乎没变（0.082s → 0.083s），缓存写法的耗时也几乎没变（0.083s → 0.084s）——**因为在这个规模下，两者都还没撞到带宽墙**。模型太小，权重只有 1.7 MB，读它是微秒级的事。

把模型放大到 Llama-3-70B（140 GB），局面就完全不同了：读一遍权重要 150 毫秒。那时 batch = 1 的 decode 是**每秒 6 个 token**，而 batch = 64 能做到每秒 400 个 token——**差 64 倍**。

ℹ 停下来想一想 ⑦

上面说 decode 阶段“算得比读得快”，所以 GPU 在等数据。

那么**为什么 KV 缓存还是有用的**？既然算力本来就空闲，少算一点不是没意义吗？

（答案：因为 KV 缓存省的不只是算力，**还包括显存读写**。

朴素写法每一步要重新读入并写出 t 个位置的中间激活—— 这些激活比权重小得多，但**随 t 增长**。到 $t = 8000$ 时，注意力矩阵是 8000×8000，光是它就有 2.56 亿个元素，**超过了模型权重本身的规模**。

所以缓存省的是两块：算力（decode 阶段本来就富余）和 **中间激活的读写**（这才是真正卡住 decode 的东西）。

3.3.4 一个具体的形状清单

把本案例模型在 decode 阶段的每一步涉及到的张量形状列全。这个清单在排查 bug 时非常有用——**形状不对是最容易发现的错误**。

张量	形状	说明
`idx`	`(1, 1)`	这一步的输入 token
`x`（残差流）	`(1, 1, 128)`	嵌入 + 位置
`qkv` 投影输出	`(1, 1, 384)`	3×128，Q/K/V 拼在一起
`q` / `k` / `v`	`(1, 1, 4, 32)`	拆成 4 个头，每头 32 维
转置后的 `q`	`(1, 4, 1, 32)`	头维度提到前面
缓存的 `k`	`(1, 4, t, 32)`	t = 已缓存长度，**逐步增长**
打分 `q @ k`$^\top$	`(1, 4, 1, t)`	一个查询对所有缓存的键
注意力权重	`(1, 4, 1, t)`	softmax 之后，和为 1
`attn @ v`	`(1, 4, 1, 32)`	加权求和
合并头	`(1, 1, 128)`	变回 `d_model`

张量	形状	说明
logits	`(1, 1, 1200)`	词表上的分数

这张表里藏着一个关键信息

看“缓存的 `k`”那一行的形状：`(1, 4, t, 32)`。

只有时间维 t 在变，其他三维固定。 这就是为什么缓存是“追加”—— 新来的 k 形状是 `(1, 4, 1, 32)`，在 `dim=-2` 上拼上去， 时间维从 t 变成 $t+1$。

再看“打分”那一行：`(1, 4, 1, t)`。

注意最后那个 t——它随缓存增长。这就是 3.4.4 节说的 “中间激活随 t 增长”的具体体现：每步的注意力权重向量长度是 t， 不是常数。

而这个向量在朴素写法里长度是 t，在缓存写法里也是 t—— 缓存并没有让这个向量变小。它让**别的东西**变小了： 朴素写法要算 $t \times t$ 的完整打分矩阵，缓存写法只算 $1 \times t$ 的一行。

省掉的正是那 t 行。

用形状表排查 bug

3.9 节那六种 bug，有四种**会立刻在形状上暴露**：

bug	形状异常
`dim` 写错（`-1` 而不是 `-2`）	缓存变成 `(1, 4, 1, t*32)`，打分变成 `(1,4,1,t*32)`
`no_cat`	缓存长度永远是 1，不增长
`pos_zero`	形状正常，**位置内容错**（形状查不出来）
`kv_mismatch`	形状正常，**对应关系错**（形状查不出来）

后两种是最难查的——它们不违反任何形状约束。这也是为什么 3.9.1 节说“要比 logits”：形状检查能挡住一半的 bug，剩下的一半只能靠数值比对。

3.3.5 回到案例：确定性从哪来

现在回头看 3.1 节的循环。

推理是串行的，每一步的输入是**上一步的输出**。而贪心解码的每一步是**确定性的**： 同一个输入，argmax 永远给出同一个输出。

把这两件事合起来看：

$$x_{t+1} = \arg\max_{v} p_\theta(v \mid x_1, \ldots, x_t) \tag{3.5}$$

所以这个式子在说：下一个字完全由前面的字决定，没有任何随机性。

这意味着整个生成过程是一个**确定性动力系统**:状态是“已经生成的文本”，转移规则是 argmax。这类系统有一个众所周知的性质——**状态空间有限时， 轨道最终必然进入周期循环**。

模型的上下文只有 64 位，每一位有 1200 种可能，状态空间虽然巨大但有限。 一旦某段历史 $x_{t-k}, \ldots, x_t$ 重复出现过，后面的输出就会**一模一样地重演**。

案例里那个 8-gram 的重复，就是它的周期。

为什么“屏蔽坏 token”治不了本

屏蔽 `<unk>` 之后，输出变成「不見昔年年不見，不見春風吹。不見君不得， 不見君家。不見君不見，不見，不如何處？何時？不見君不見，不」。

`<unk>` 确实没了，循环也还在，只是周期从 8 变成了 4，循环单元从标点换成了「不見」。

这是一个重要的思维训练：屏蔽坏 token 是在**改变状态空间的形状**， 不是在**改变动力系统的性质**。argmax 仍然是确定性的， 有限状态空间仍然必然导致循环。你只是把循环的位置挪了挪。

真正要打破循环，必须**让转移规则不再确定**——这就是 3.5 节的采样。

一个可以自己做的实验

把提示词换成任意一段文字，用贪心生成 200 个字， 用 3.1.3 节的 `degeneracy()` 函数量一下 distinct-2 和最长重复 n-gram。

你会发现**它几乎必然退化**，而且退化程度和提示词关系不大。 这不是这个 42 万参数模型的问题——GPT-2、GPT-3 用贪心解码，一样会退化。Holtzman 等人 2020 年那篇 The Curious Case of Neural Text Degeneration 讲的就是这件事，见 3.10 节。

3.3.6 既然必须串行，那怎么加速：四条路

自回归的串行性是任务定义带来的，无法消除。但**“必须串行”不等于“每步都必须那么贵”**。工业界有四条路，它们**针对的是不同的瓶颈**。

第一条：KV 缓存。 减少**每步的计算量**。3.4 节详述。 注意它**不减少步数**——这是 3.4.4 节那个反直觉结果的根源。

第二条：批处理（batching）。 减少**权重的重复读取**。 3.3.3 节讲过：decode 阶段 GPU 在等数据，把 B 个请求拼一批， 读一次权重服务 B 个序列。步数不变，但每步的**吞吐**提高 B 倍。

第三条：投机解码（speculative decoding）。 减少**步数**。

这是最巧妙的一条。用一个**小模型**（draft model）先快速猜出接下来 5 个 token， 然后让**大模型**一次前向**并行验证**这 5 个猜测：

```
小模型猜:   春风 又 绿 江 南
大模型验:   春风 又 绿 江 南    ← 一次前向，5 个位置同时验证
             ✓    ✓   ✓   ✓   ✓
```

因为验证是**一次前向算 5 个位置**（就像训练那样并行）， 所以它**只需要 1 步**而不是 5 步。如果小模型猜对了 4 个， 就相当于用 1 步生成了 4 个 token——**提速 4 倍**。

猜错怎么办? **从第一个错的位置截断，用大模型自己的结果**。这保证了**输出分布和不用投机解码时完全一致**——它是个无损加速。

投机解码正好踩中本章另一个 bug

回头看 3.9.4 节：**`no_mask` 在单步解码时是恒真的**。

投机解码**恰恰是“一次前向算多个位置”**——它把 T_{new} 从 1 变成 5。 所以：

```
no_mask 在 T_new=1 时 max|Δlogit| = 0.000e+00
         分块 4 个时 = 3.335
```

那个分块实验（3.335）测的就是投机解码的场景。

“验证”这一步如果掩码写错，被验证的 5 个位置会**互相看到对方**—— 第 2 个草稿 token 会看到第 3、4、5 个草稿。 验证结果全错，但程序不崩，只是**接受率异常低**（因为验证总是不通过）。

排查这类问题的人通常会怀疑小模型猜得不准， 而不会想到是大模型的验证路径错了。

第四条：换掉自回归。

前三條都在优化“怎么更快地串行”，第四条问的是**能不能不串行**。这是一个活跃的研究方向，也是本书**下篇**的主题——扩散模型。 3.3.7 节展开。

四条路针对的瓶颈不同，要一起用

方法	减少什么	提速上限
KV 缓存	每步计算量	与上下文长度相关
批处理	权重的重复读取	与 batch size 成正比
投机解码	**步数**	与接受率成正比（通常 2–4 倍）
非自回归	串行性本身	理论上 1 步

注意“提速上限”那一列没有一个是无限的。 真实系统同时用前三条， 叠起来能到 10–50 倍——但每一步的**串行依赖**始终在那里。

3.3.7 如果把自回归去掉

自回归的根本代价是：N 个 token 需要 N 次串行前向。

能不能一次生成全部 N 个？ 这就是**非自回归生成**（non-autoregressive generation，NAR）的目标。

最直接的做法：模型一次输出 N 个位置的分布，每个位置**独立**采样。问题是它**必然产生不连贯的文本**——因为位置之间没有任何协调机制。“白日依山尽”可能被生成成“白日依山尽”（对）也可能“白月依水尽”（乱），因为每个位置独立决策。

后来的方法都是在“独立”和“串行”之间找中间点：

方法	独立程度	迭代次数
自回归	完全依赖	N
掩码预测（MaskGIT、Mask-Predict）	部分依赖	常数（如 10 次）
扩散语言模型	部分依赖	常数（如 50 次）
纯 NAR	完全独立	1

核心思路：先生成一个粗糙的完整序列，然后**迭代细化**。每次细化都用**当前完整序列**作为条件——所以每一步都是“一次前向算所有位置”，**可以并行**。

这就是下篇要讲的扩散模型

扩散模型的语言版本（diffusion language model）用的正是这个框架：

- **前向过程**：把一句话逐步“加噪”——对文本来说就是把 token 逐步替换成 随机 token 或 `[MASK]`
- **反向过程**：从纯噪声开始，逐步“去噪”成一句话
- 每次去噪都是**一次前向**，N 个位置同时算

迭代 50 次就得到结果——相比自回归的 N 次（N 通常是几百到几千），**步数少一个数量级**。

第 5 章会把这个框架的数学（前向过程的闭式解、ELBO、重参数化）完整推一遍。**这里只需要记住动机**：自回归的串行性是它最大的成本，而扩散提供了一条“用迭代换并行”的路。

它的代价是什么：迭代次数是个超参数，太少了质量差，太多了慢；而且**没有 KV 缓存可用**——因为每一步的输入都完全不同，不存在“只追加、不修改”的缓存。3.4.2 节那个恒等性证明的前提不成立。

停下来想一想 ⑧

上面说扩散语言模型“没有 KV 缓存可用”。

为什么第 3.4.2 节的证明在这里失效了？

（答案：证明的关键前提是“**过去的 token 不会变**”。

自回归生成时，前缀是**定死的**——生成第 $t+1$ 个 token 时，前 t 个永远不变。所以它们的 K/V 算一次就够了。

扩散的迭代细化**每一步都在改所有的位置**。第 5 次迭代时位置 3 的 token 可能和第 4 次时不同，所以位置 3 的 K/V **每一步都得重算**。

“缓存”这个概念成立的前提是“被缓存的东西不再变”。前提没了，缓存也就没了。）

3.4 KV 缓存：把两副面孔接起来

3.4.1 先算一遍：三个 token 的手工账

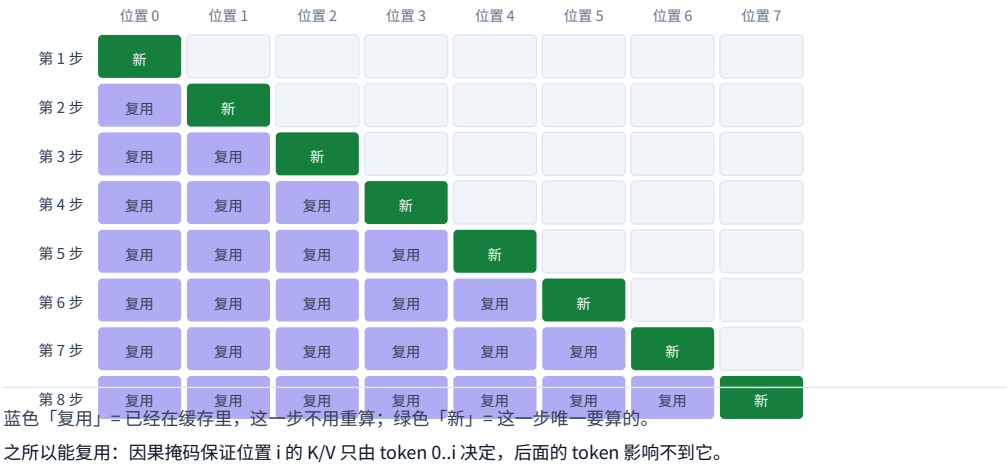


图 3.3　KV 缓存随生成步数增长。蓝色「复用」表示该位置的 K/V 已经在缓存里，这一步不用重算；绿色「新」是这一步唯一要算的。因果掩码保证了蓝色格子永远不会变——这是缓存成立的前提。

在写任何代码之前，先拿最小的例子把账算清楚。这是本书的规矩。

设置：上下文长度 $T = 3$，模型维度 $d = 4$，单头（$h = 1$，$d_{head} = 4$）。忽略前馈网络和归一化——它们不参与“哪些计算被重复”的讨论。

我们要生成第 4 个 token。假设前三步已经算过，现在看第四步。

朴素写法：把 x_1, x_2, x_3 重新喂一遍，拿到位置 3 的输出，再去预测 x_4。

用 $W_Q, W_K, W_V \in \mathbb{R}^{4\times4}$ 表示投影矩阵，那么：

$$Q = XW_Q, \quad K = XW_K, \quad V = XW_V, \quad X \in \mathbb{R}^{3\times4} \tag{3.6}$$

Q, K, V 都是 3×4。注意力输出是：

$$\mathrm{Attn}(X) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{4}}\right)V \tag{3.7}$$

其中 $QK^\top$ 是 3×3，softmax 逐行做，最后乘 V 得到 3×4。我们要的是**最后一行**（位置 3 的输出），用它去算 x_4 的 logits。

现在把它写成“哪个位置用了哪些输入”的形式：

计算	用到的输入
Q 的第 3 行	x_3
K 的第 1、2、3 行	x_1, x_2, x_3
V 的第 1、2、3 行	x_1, x_2, x_3
$QK^\top$ 的第 3 行	q_3 和 k_1, k_2, k_3
输出的第 3 行	第 3 行注意力权重和 v_1, v_2, v_3

关键观察：要算位置 3 的输出，需要的 K 和 V 是**前三个位置全部**的。而如果我们上一步（生成 x_3 时）已经算过 k_1, k_2 和 v_1, v_2，这一步**只有 k_3, v_3 是新的**。

所以缓存的做法是：

$$K_{\text{cache}} = [\,k_1; k_2; k_3\,], \qquad V_{\text{cache}} = [\,v_1; v_2; v_3\,] \tag{3.8}$$

每生成一个 token，只把它对应的那一行 k_t, v_t **追加**到缓存末尾。

这一笔账省了多少

朴素写法：每步重新算**整段**的 Q, K, V。位置 3 这一步要算 3 行 K，3 行 V。

缓存写法：位置 3 这一步只算 **1 行** K，**1 行** V。

到第 t 步，朴素算 t 行，缓存算 1 行——**省了 t 倍**。 把 t 从 1 累加到 N，总省下的是 $O(N^2)$ 级别的投影计算。

注意：**Q 是不能缓存的**。新位置必须算自己的查询向量。但 Q 只有 1 行，K 和 V 是要被反复用到的——这就是为什么缓存的是 K 和 V。 名字也是这么来的。

> **i** 停下来想一想 ⑨
>
> 上面说“Q 不能缓存”。请你从定义出发说明为什么。
>
> （提示：$q_t = x_t W_Q$，它只依赖位置 t 自己的输入。 生成 x_{t+1} 时我们算的是位置 $t+1$ 的 Q —— 这个位置**以前根本不存在**，所以没有“缓存”可言。 它是新的，不是重复的。
>
> 而 K、V 不一样：生成 x_{t+1} 时，注意力需要看到的是**所有历史位置**的 K 和 V。 这些历史位置的 K、V 在它们各自被生成时就算出来了，而且**以后再也不会变**。）

3.4.2 为什么它是恒等的：一个必须证明的前提

KV 缓存号称“不改变结果，只改变速度”。这是个很强的声明，得证明。

要证明的核心命题是：

> **位置 t 的 k_t 和 v_t，在生成第 $t+1, t+2, \ldots$ 个 token 的过程中，永远等于最初算出来的那个值。**

如果这个命题成立，缓存就是安全的——存下来的东西永远有效。 如果它不成立，缓存就是错的——存下来的是过期数据。

证明：

k_t 由什么决定？回顾注意力的计算：

$$k_t = \mathrm{LN}(x_t)\, W_K \tag{3.9}$$

而 x_t 是位置 t 的**残差流**（residual stream）的值，它由第 t 个 token 的嵌入， 经过前面每一层的“注意力 + 前馈”累积而来：

$$x_t^{(\ell+1)} = x_t^{(\ell)} + \mathrm{Attn}^{(\ell)}(x_1^{(\ell)}, \ldots, x_t^{(\ell)}) + \mathrm{FFN}^{(\ell)}(\cdot) \tag{3.10}$$

关键就在这个式子里：位置 t 在第 $\ell+1$ 层的值，依赖于第 ℓ 层里 位置 1 到 t 的值，**不依赖任何 $t' > t$ 的位置**。

为什么？因为因果掩码。注意力在位置 t 只对 $1 \ldots t$ 加权求和， 那些 $t' > t$ 的位置权重恒为 0。

归纳下去：x_t 只依赖 $x_1, \ldots, x_t$ 的**初始嵌入**， 也就是只依赖 token $1 \ldots t$。

而 token $1 \ldots t$ 在生成过程中是不变的。 生成第 $t+1$ 个 token 时， 前 t 个 token 早就定下来了，不可能再改。

所以 x_t 不变 $\Rightarrow$ k_t, v_t 不变。命题得证。

这个证明为什么重要

它把“缓存”从一个**工程技巧**变成了一个**数学推论**。

工程技巧的意思是：这样做大概能行，如果结果有细微差别也正常。数学推论的意思是：**结果必须一模一样。差一点点就是 bug**。

这个区别决定了我们怎么测试它。如果是技巧，测试方法是“看起来差不多就行了”； 如果是推论，测试方法是“逐位比对，看最大差异是不是 0”。

本章后面所有的缓存排错，都建立在这个测试方法上。

> **停下来想一想 ⑩**
>
> 上面的证明用到了“因果掩码”这个前提。如果模型**不用**因果掩码（比如 BERT 那样的双向注意力），缓存还成立吗？
>
> （答案：不成立。双向注意力下，位置 t 的表示依赖于整句话的所有位置，包括后面的。生成一个新 token 会改变所有旧位置的表示，**存下来的东西立刻过期**。
>
> 这就是为什么**只有自回归模型能缓存 K/V**。编码器模型（BERT）没有这个优化，因为它根本不做自回归生成。）

3.4.3 实测：0.0 和 1.9×10^{-6}

证明完了，跑一遍验证。

测试一：预填充阶段。把 7 个字的提示词一次喂进去（不带缓存），对比“一次喂进去但走缓存”：

```
预填充 max|Δlogit| = 0.0
```

完全相同的 0。不是 1e-7，不是 1e-9，是浮点数意义上的逐位相同。

测试二：增量阶段。这是真正的考验——先缓存前 k 个位置，然后只喂第 $k+1$ 个 token，和不带缓存地重新跑一遍整段，对比第 $k+1$ 个位置的 logits：

```
增量   max|Δlogit| = 1.907e-06
```

不是 0 了。1.9×10^{-6}。

这个非零值需要解释，它不代表缓存错了。

> **⚠ 1.9e-6 是什么**
>
> 浮点数加法**不满足结合律**。
>
> $(a+b)+c$ 和 $a+(b+c)$ 在实数上相等，在 float32 上可能差最后一位。缓存路径和重算路径，对同一批数做加法的**顺序不一样**：- 重算路径：一次矩阵乘法，内部按某种分块顺序累加 - 缓存路径：历史部分是一次乘法，新 token 部分是另一次，然后把结果相加
>
> 顺序不同，最后一位就可能不同。1.9×10^{-6} 相对于 logits 的量级（约 10）是 **相对误差 2×10^{-7}**，正好是 float32 的机器精度（约 1.2×10^{-7}）量级。
>
> **判断标准**：这个差异应该随“模型规模”和“序列长度”缓慢增长，但不应该随“生成步数”累积爆炸。如果它随步数指数增长，那是真的错了。

> **一个实用的判据**
>
> 实践中怎么区分“浮点误差”和“真的写错了”？看量级。
>
观察到的 max|Δlogit|	判断
$< 10^{-4}$	浮点误差，实现是对的
$10^{-3} \sim 10^{-1}$	可疑，要查
> 1	一定是 bug
>
> 本章实测的四种真 bug，最大差异分别是 **12.06、10.85、17.53、2.10**—— 最小的那个 2.10 也比浮点误差大**六个数量级**。
>
> 这个判据不是拍脑袋定的：logits 本身的量级在 10 上下，float32 有 24 位有效数字，$10 \times 2^{-24} \approx 6 \times 10^{-7}$。累积几步之后 10^{-5} 是合理上限。

3.4.4 省了 28 倍计算，为什么一点没变快

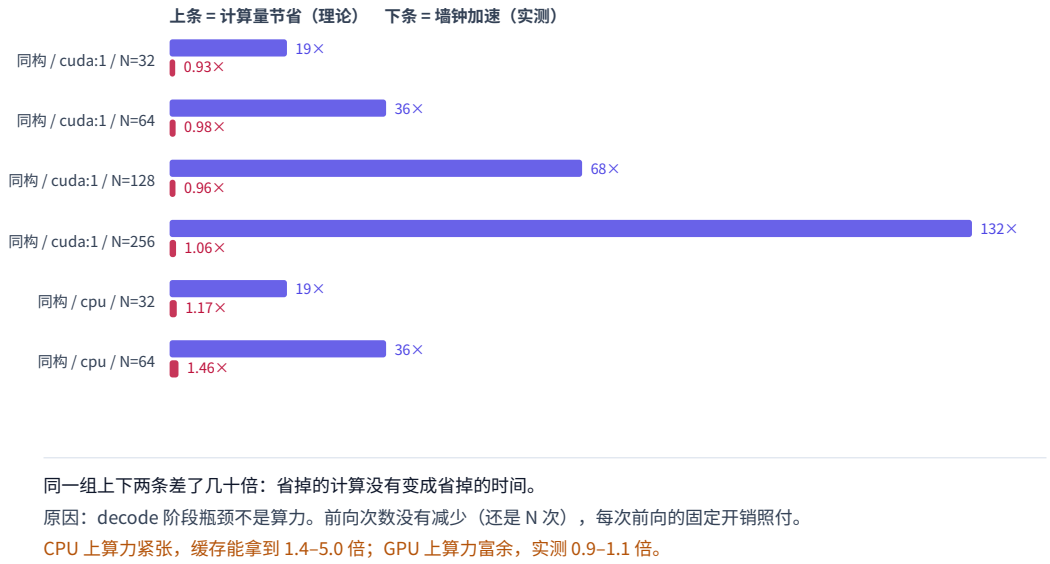


图 3.4　KV 缓存省下的计算量（上条）与实测墙钟加速比（下条）。同一组里两条相差几十倍——省掉的算术没有变成省掉的时间，因为 decode 阶段的瓶颈是 kernel 启动和显存带宽，不是算力。

现在到了本章最反直觉的部分。

既然缓存省掉了 23 到 28 倍的计算量，那它应该快 23 到 28 倍吧?

实测结果：

```
cuda:1 batch=  1: 朴素 0.082s  缓存 0.083s  实测  0.99×  计算量 28.1×
cuda:1 batch= 16: 朴素 0.105s  缓存 0.101s  实测  1.04×  计算量 28.1×
cuda:1 batch= 64: 朴素 0.083s  缓存 0.084s  实测  0.99×  计算量 28.1×
   cpu batch=  1: 朴素 0.049s  缓存 0.034s  实测  1.44×  计算量 28.1×
   cpu batch= 16: 朴素 0.137s  缓存 0.054s  实测  2.53×  计算量 28.1×
   cpu batch= 64: 朴素 0.385s  缓存 0.077s  实测  4.97×  计算量 28.1×
```

GPU 上，缓存一点没快。 计算量明明少了 28 倍。

你可能会说：是不是模型太小了？42 万参数确实小。那就换个大的：

```
── 大模型·8 层 768（58,387,968 参数）
   cuda:1 生成  32: 朴素   0.179s  缓存  0.187s  实测  0.95×  计算量  19.3×
   cuda:1 生成  64: 朴素   0.367s  缓存  0.358s  实测  1.03×  计算量  35.6×
   cuda:1 生成 128: 朴素   0.766s  缓存  0.713s  实测  1.07×  计算量  67.8×
      cpu 生成  32: 朴素   0.773s  缓存  0.514s  实测  1.51×  计算量  19.3×
      cpu 生成  64: 朴素   1.774s  缓存  1.015s  实测  1.75×  计算量  35.6×
```

5839 万参数、8 层、计算量省了 68 倍，GPU 上实测 1.07 倍。

为什么

因为**墙钟时间不由计算量决定，由“有多少步必须串行”决定。**

看两条路径的结构：

	前向次数	每次前向的 token 数
朴素	N	$P, P+1, \ldots$（递增）
缓存	N（预填充 1 次 + 每步 1 次）	P，然后全是 1

前向次数是一样的，都是 N 次。差别只在每次前向算多少 token。

而一次前向在 GPU 上要花多少时间，**绝大部分不是花在算术上**，而是花在：Python 解释器的开销、kernel 启动、显存读写、同步。

算一下单次前向的量级。8 层、$d = 768$、$d_{ff} = 3072$，一层的前馈网络约 $2 \times d \times d_{ff} = 2 \times 768 \times 3072 \approx 4.7 \times 10^6$ 次乘加，注意力约 $2 \times T \times d^2$。取 $T = 128$，一层合计约 10^8 量级，8 层约 10^9。本机 3090 的 fp32 算力约 35 TFLOPS，所以**纯算术时间约 30 微秒**。

而实测单步耗时约 5.5 毫秒。**99% 的时间不在算术上**。

这个结论的正确用法

不是说“KV 缓存没用”。而是说它**省的是算力，不是延迟**。

这两件事在不同的场景下价值完全不同：

场景	瓶颈	KV 缓存的价值
单条请求、短上下文、GPU	kernel 启动 + Python 开销	**几乎为零**
单条请求、CPU	算力	1.4–5.0 倍
大批量服务	显存带宽 + 算力	显著
长上下文（T 很大）	注意力的 $O(T^2)$	**决定性**

最后一行是关键。本章的上下文只有 1024，注意力的 T^2 项还是小头。真实模型动辄 128K 上下文——那时朴素写法每一步都要算 T^2 的注意力矩阵，$T = 128000$ 时这个矩阵有 1.6×10^{10} 个元素，**根本跑不起来**。KV 缓存不是“优化”，是“能不能跑”的区别。

> **i** 停下来想一想 ⑪
>
> 既然 GPU 上瓶颈是“前向次数”而不是“每次算多少”，那么一个自然的推论是：**把多次前向合并成一次**应该能提速。
>
> 这个想法是对的吗？想想为什么现在的推理框架要做“continuous batching”（把不同请求的 token 拼成一个 batch 一起算）。
>
> （答案：对。continuous batching 干的就是这件事——它不减少总计算量，但把多个请求的“第 k 步”合并成一次前向，于是**前向次数按并发数摊薄**。这和 KV 缓存省计算量是两个正交的优化方向，实践中要一起用。）

3.4.5 缓存的形状与成本

缓存不是免费的。把它的开销算清楚。

显存占用：

$$\text{bytes} = 2 \times L \times h \times T \times d_{head} \times \text{sizeof(dtype)} \tag{3.11}$$

所以这个式子在说：2（K 和 V 各一份）× 层数 × 头数 × 上下文长度 × 每头维度 × 数据类型字节数。

注意 $h \times d_{head} = d_{model}$，所以可以简化成：

$$\text{bytes} = 2 \times L \times T \times d_{model} \times \text{sizeof(dtype)} \tag{3.12}$$

本案例的模型：$L = 2$，$T = 64$，$d_{model} = 128$，float32（4 字节）。

$$2 \times 2 \times 64 \times 128 \times 4 = 131,072 \text{ 字节} \approx 128 \text{ KB} \tag{3.13}$$

比模型本身（425,984 个参数 × 4 = 1.7 MB）还小。所以这里完全不用考虑显存。

换成真实规模算一下，感受会完全不同。Llama-3-70B：$L = 80$，$d_{model} = 8192$，$T = 8192$，fp16（2 字节）：

$$2 \times 80 \times 8192 \times 8192 \times 2 \approx 21.5 \text{ GB} \tag{3.14}$$

单个请求、单个序列，光 KV 缓存就要 21.5 GB。模型权重本身是 140 GB。缓存占到了权重的 15%——这就是为什么真实推理服务要在“并发多少条请求”和“每条多长上下文”之间反复权衡。

> **i** 停下来想一想 ⑫
>
> 式 式 (3.12) 里，T 是**线性**出现的。但注意 T 在这台机器上 等于“已经生成了多少 token”——它会随着生成过程**不断增长**。
>
> 这意味着什么?
>
> （答案：KV 缓存的显存占用是**动态增长**的。一条请求开始时只占几百 KB， 生成 8000 个 token 后就占满了。这也是为什么推理框架要做 “显存预分配 + 分页管理”（vLLM 的 PagedAttention 就是干这个的）—— 否则每生成一个 token 都要重新分配一块更大的显存，还要把旧数据拷过去。）

3.4.6 回到案例：缓存解决了问题的一半

把这一节接回 3.1 节的案例。

缓存解决了什么? **它让“每一步只算新 token”成为可能**，从而 让生成的总计算量从 $O(N^2)$ 降到 $O(N)$。

但注意——**它没有解决循环问题**。

```
none        首个 logit 差在第 None 步  max|Δ|=0.0000     文本分歧  0 处
```

缓存版本生成的文本，和不带缓存一**字不差**。它还是「。,。,。,」。

这正是“数学推论”的直接后果：如果缓存是恒等变换， 它**必然**产生完全相同的输出，包括完全相同的错误。

要治循环，得动解码策略，不是动缓存。 这是 3.5 节的事。

3.4.7 缓存的工程变体：从 MQA 到分页

3.4.5 节算过：Llama-3-70B 单序列、8K 上下文，KV 缓存要 21.5 GB。这一节看工业界怎么把它压下来。

变体一：MQA —— 所有头共享一份 K/V

标准多头注意力里，每个头有**自己的** K、V。$h = 64$ 个头就有 64 份缓存。

MQA（multi-query attention）的做法：**Q 还是每个头一份，K/V 全局只有一份**。 缓存大小直接降到 $1/h$——70B 模型上从 21.5 GB 降到 0.34 GB。

代价是**质量下降**。因为不同头本来要关注不同的东西（第 2 章讲过：多头买的是“让不同头学不同关系”的自由度）， 共享 K/V 之后这个自由度就没了。

变体二：GQA —— 分组共享

GQA（grouped-query attention）是折中：把 h 个头分成 g 组， **组内共享** K/V，组间不共享。

- $g = h$：退化成标准 MHA
- $g = 1$：退化成 MQA
- $g = 8$：Llama-2/3 的选择，缓存降到 1/8，质量损失很小

这是个漂亮的工程折中：没有理论上的最优解， 但实证发现“8 组”基本无损。

注意它和本章 bug 的关系

MQA/GQA 改的是**缓存的形状**：从 `(B, h, T, d_head)` 变成 `(B, g, T, d_head)`。

它不改变缓存的语义——仍然是“只追加、不修改”， 仍然满足 3.4.2 节那个恒等性证明的前提。

所以 **3.9.1 节那套“比 logits”的测试方法，换成 MQA 之后完全适用**。 你不需要为 MQA 写一套新的测试——**只要把形状的期望值改掉**。

这是好抽象的标志：一个改变不会波及测试方法论。

变体三：分页 —— 借操作系统的思路

前面两个变体压缩的是**每个 token 的缓存大小**。分页解决的是另一个问题： **缓存的总量管理**。

问题的根源在 3.4.5 节的“停下来想一想 ⑦”：**缓存的显存占用是动态增长的**， 而且**生成之前你不知道会长到多少**。

传统做法是预分配一个最大长度的块（比如 8192）。后果：

- 一条只生成 100 个 token 的请求， **占了 8192 的空间**
- 实测中浪费率高达 **60%–80%**

vLLM 的 PagedAttention 借用操作系统虚拟内存的思路：把缓存切成固定大小的**页**（比如 16 个 token 一页），用一张**页表**把逻辑位置映射到物理页。

好处有三个：1. **按需分配**：用到第几页才分配第几页 2. **无外部碎片**：所有页大小相同，随便放 3. **前缀共享**：同一个提示词的多个输出**共享同一份前缀页**

第 3 点是意外收获，也是最大的收益之一。很多应用（few-shot 提示、多轮对话）有大量重复前缀，共享之后显存占用能降一个数量级。

前缀共享正好踩中本章一个 bug

回到 3.9.4 节那个结论：`no_mask` **在单步解码时是恒真的**，因为查询位置恰好是最大的键位置。

前缀共享破坏了这个前提。

设想两个请求共享了长度 1000 的前缀，然后各自继续生成。请求 A 生成了第 1001 个 token，请求 B 生成了第 1002 个。如果把 B 的缓存和 A 的拼接……**查询位置和最大键位置不再相等**，掩码不再是恒真的。

所以那个“在我们这里不会出问题”的 bug，会在引入前缀共享的那一天突然爆发。

这正是 3.9.4 节那句话的意思：**理论上不会发生的事情，实践中总会发生一次**。

3.4.8 缓存里到底存了什么：一次完整的手工推演

3.4.1 节算的是“省了多少”，这一节把“每步到底算什么”完整走一遍。

设置：上下文长度 4，$d_{model} = 4$，单头。提示词是 3 个 token：`x1` `x2` `x3`，现在要生成 `x4`。

第一步：预填充。

输入的 $X_1 \in \mathbb{R}^{3\times 4}$（三个位置，每个 4 维）。算出的：

$$Q_1 = X_1 W_Q \in \mathbb{R}^{3\times 4}, \quad K_1 = X_1 W_K \in \mathbb{R}^{3\times 4}, \quad V_1 = X_1 W_V \in \mathbb{R}^{3\times 4} \tag{3.15}$$

打分矩阵 $S_1 = Q_1 K_1^\top / \sqrt{4} \in \mathbb{R}^{3\times 3}$。加因果掩码（下三角）后 softmax，得到 $A_1 \in \mathbb{R}^{3\times 3}$。输出 $A_1 V_1 \in \mathbb{R}^{3\times 4}$。

我们只要最后一行 $A_1[3,:]V_1$——它是位置 3 的输出，用来预测 `x4`。

把 K、V 存起来：$K_{\text{cache}} = K_1$，$V_{\text{cache}} = V_1$，形状都是 3×4。

第二步：生成 x4。

`x4` 被喂进来，算出它自己的：

$$q_4 = x_4 W_Q \in \mathbb{R}^{1\times 4}, \quad k_4 = x_4 W_K, \quad v_4 = x_4 W_V \tag{3.16}$$

关键：q_4 是新的（位置 4 以前不存在），k_4, v_4 也是新的。但**注意力需要的是“位置 4 对所有历史位置的关注”**——所以需要 k_1, k_2, k_3，而它们**已经在缓存里了**。

拼接：

$$K'_{\text{cache}} = [k_1; k_2; k_3; k_4] \in \mathbb{R}^{4\times 4}, \quad V'_{\text{cache}} = [v_1; v_2; v_3; v_4] \tag{3.17}$$

打分：$S = q_4 {K'_{\text{cache}}}^\top / \sqrt{4} \in \mathbb{R}^{1\times 4}$。

掩码：这里出现了一个 3.4.1 节没提的细节。

朴素路径的掩码是 4×4 的下三角。缓存路径的掩码是 1×4——只有一行，因为只有一个查询。

这一行是什么？查询位置是 4，键位置是 1、2、3、4。因果规则“键位置 $\le$ 查询位置”对**四个键全部成立**。所以这一行是 `[1, 1, 1, 1]`——**全是 1，什么都不屏蔽**。

这就是 3.9.4 节那个结论的来源：在单步解码时，因果掩码**恒真**。不是“掩码不重要”，而是“这一步恰好没有未来需要屏蔽”。

输出：$AV'_{\text{cache}} \in \mathbb{R}^{1\times 4}$，用它预测 `x5`。

把两次的形状并排看

	预填充	单步解码
查询数	3	**1**
键数	3	4
打分矩阵	3×3	1×4
掩码	下三角	**全 1**
新算的 K/V	3 行	1 行

	预填充	单步解码
复用的 K/V	0 行	**3 行**

“复用了 3 行”就是缓存的全部内容。 朴素写法会把这 3 行重算一遍。

到第 t 步，复用的行数是 $t-1$。累加起来，整个生成过程省下的 是 $O(N^2)$ 行的投影计算——这就是 3.3.2 节那个 27.6 倍的来源。

> **i** 停下来想一想 ⑬
>
> 上面说单步解码的掩码是“全 1”。那么**如果模型没有因果掩码**，单步解码时还需要掩码吗？
>
> （答案：**在缓存路径下不需要，但那是因为别的原因。**
>
> 如果不加因果掩码，预填充阶段就会错——位置 1 会看到位置 2、3。所以**预填充必须有掩码。**
>
> 而单步解码时，缓存里的所有 token 都**确实是过去**（它们已经被生成了），所以“看到全部”是正确的行为——**恰好等于全 1 掩码。**
>
> 换句话说：**单步解码的正确行为，恰好就是“不掩码”。** 这个巧合是 3.9.4 节那个 bug 能藏住的根本原因。）

3.4.9 怎么给缓存写测试

3.4.2 节证明了缓存是恒等变换，3.4.3 节实测了它。这一节把它变成**可复用的测试**。

测试一：恒等性（最重要的一个）

```
def test_cache_is_identity(model, ids):
    """缓存必须逐位等于重算。"""
    ref, _ = forward_step(model, ids, cache=None)          # 整段一次算
    cache = KVCache(len(model.blocks))
    got, _ = forward_step(model, ids, cache=cache)         # 走缓存
    assert torch.equal(ref, got), f"预填充不一致: {(ref-got).abs().max()}"
```

注意用了 `torch.equal` 而不是 `allclose`。

为什么？因为**预填充阶段两次计算走的是完全相同的代码路径**（都是 T 个位置一次算完），所以结果应该**逐位相同。** 用 `allclose` 会掩盖掉“其实走了不同路径”这个事实。

本章实测：`max|Δlogit| = 0.0`。 不是 1e-7，是 0。

测试二：增量一致性

```
def test_incremental(model, ids, k):
    """第 k 个位置：整段重算 vs 只算新 token。"""
    seq = ids[:, :k+1]
    full, _ = forward_step(model, seq, cache=None)
    cache = KVCache(len(model.blocks))
    forward_step(model, ids[:, :k], cache=cache)
    step, _ = forward_step(model, ids[:, k:k+1], cache=cache)
    d = (full[:, k] - step[:, 0]).abs().max().item()
    assert d < 1e-4, f"增量路径不一致: {d}"
```

这里用容差而不是 `torch.equal`，因为增量路径和整段路径的 **加法顺序确实不同**（见 3.4.3 节），浮点误差是**预期的。**

本章实测：`1.907e-06`，远小于 1e-4 的阈值。

两个测试的严格程度不同，这是有意的

测试	判据	为什么
预填充	`torch.equal`（严格）	两条路径**应该**完全一样
增量	`< 1e-4`（容差）	两条路径的**加法顺序不同**

如果两个都用严格相等：增量测试会失败（浮点误差），你会以为实现有 bug，然后去改一个正确的实现。

如果两个都用容差：预填充测试会漏掉“走了不同代码路径”的问题。

为每个测试单独定判据，而不是用一个统一的阈值—— 这需要你想清楚“这两条路径在数学上应该有多接近”。 **测试的严格程度是设计出来的，不是抄来的。**

测试三：故意注入错误，验证测试能抓住

```
@pytest.mark.parametrize("bug", ["pos_zero", "no_cat", "no_mask", "kv_mismatch"])
def test_bugs_are_caught(model, ids, bug):
    """把每种 bug 注进去，上面的测试必须失败。"""
    with pytest.raises(AssertionError):
        run_identity_test(model, ids, bug=bug)
```

这是最容易被跳过、却最重要的一步。

一个测试如果**不管代码怎么写都通过**，那它没有在测任何东西。 本章的五种 bug 实现（`kv_cache.BUGS`）存在的意义就是这个—— **它们是测试的测试。**

实测它们全部被抓到（3.9.1 节的表：最大差异 2.10 到 17.53， 全部远超 1e-4 的阈值）。

> 测试覆盖率不等于测试有效性
>
> 这里有一个反直觉的事实：
>
> **本章的缓存实现只有 20 行代码，测试代码比实现代码长。**
>
> 这在“数学上应该恒等”的场景里是**正常比例**。因为这类代码的特点是：**写对了不难，写错了也不报错。**
>
> 所以投入产出比完全不一样： - 普通业务代码：测试是“防止未来改坏” - 恒等性优化代码：测试是“证明现在就是对的”
>
> **第二类的测试不是可选项，是交付物的一部分。**

3.4.10 批处理时的缓存：一个绕不开的麻烦

前面所有讨论都假设 batch = 1（一条请求）。真实服务要同时处理很多条。

最初的直觉：把缓存从 `(1, h, T, d)` 变成 `(B, h, T, d)` 不就行了？

不行。因为每条请求的长度不一样。

设想 batch 里有两条请求：

```
请求 A：已生成 100 个 token
请求 B：已生成  10 个 token
```

现在要同时算它们的“下一步”。A 的查询位置是 100，B 的是 10。 **它们的缓存长度不同，没法直接拼成一个张量。**

三种解法，各有代价：

做法	怎么处理	代价
补零对齐	按最长的补到 100	B 的 90 个位置是 padding，白算
分组（bucketing）	长度相近的放一组	组内仍要补齐；组间负载不均
分页（PagedAttention）	物理不连续，用页表索引	需要自定义 kernel

第三种是现在的答案。 这是 3.4.7 节讲的 PagedAttention 的**第二个动机**—— 第一个动机是“避免预分配浪费”，第二个是**“让变长请求能真正并行”**。

数学上还有一个不显然的问题

补零对齐不只是“白算”——**它会污染结果。**

假设 B 的缓存被补到 100 个位置，后 90 个是零向量。那么 B 的查询会算 $\text{softmax}(qK^\top)$，其中 K 包含 90 个零行。

零行的打分是 $q \cdot 0 = 0$。**而 0 是一个合法的 logit！** 经过 softmax，这 90 个假位置会分到**实实在在的概率质量。**

结果：B 的注意力被 90 个不存在的 token 稀释了。

正确做法是**给这些位置加掩码**，让它们的权重严格为 0。这就回到了 3.2.7 节那个“掩码怎么实现”的问题—— **而这次是被 padding 逼出来的，不是被因果性逼出来的。**

同一段代码要处理两种不同来源的掩码：1. 因果掩码：防止看到未来 2. padding 掩码：防止看到补零

很多实现只做了第一个。 在 batch = 1 时看不出问题（没有 padding），一上 batching 就出问题。

> **i** 停下来想一想 ⑭
>
> 上面说“零向量的打分是 0，经过 softmax 会分到概率质量”。
>
> **为什么这不影响 A?** A 的缓存里没有补零。
>
> （答案：因为在“补零对齐”的做法里，**A 不需要补**—— 它是 batch 里最长的那条。B 才需要补。
>
> **所以症状是“只有部分请求出错”**： 每条请求的输出质量取决于“它在 batch 里的相对长度”。 同一个提示词，单独跑是对的，和一条长请求放一起跑就变差了——
>
> **这种“依赖 batch 组成”的行为是最难排查的 bug 之一**，因为它**不可复现**：换个请求顺序、换个并发量，结果就变了。
>
> **这又是“静默降级”**（见 3.6.3 节）：不报错，只是某些请求悄悄变差。

3.4.11 一句话总结缓存和批处理的分工

	KV 缓存	批处理
减少的量	每步的 **token 数**	每步的**前向次数**（摊销）
瓶颈类型	算力	显存带宽
在 CPU 上	明显有效（1.4–5.0 倍）	有效
在 GPU 上（小模型）	**几乎无效**	有效
在 GPU 上（长上下文）	**决定性**	有效
引入的新问题	显存管理、变长对齐	变长对齐、调度

两者必须一起用，而且**两者都引入了“变长对齐”这个问题**—— 这个问题的解法是同一个：分页。

3.4.12 一个自然的问题：训练时能用缓存吗

不能，而且不是因为“不需要”，是因为“用不上”。

3.2.3 节讲过：训练时一次前向处理整段 T 个位置， **每个位置只算一次**。缓存的作用是“避免重复计算同一个位置” —— 训练时**根本没有重复**。

但有一个例外值得一提：**多轮对话的训练**。

假设一个训练样本是 10 轮对话，每轮平均 200 个 token，共 2000 个。朴素做法是：把整个 2000 token 的序列喂进去，一次前向算完。

这没问题。但如果数据格式是“10 个独立的训练样本，每个都是 前缀 + 一轮回复”（这是常见的数据构造方式），那**前缀被重复计算了 10 次**。

这时有两种做法：

做法	计算量	梯度正确性
每个样本独立算	10 次前向，前缀算 10 遍	正确
拼成一个长序列	1 次前向	**会出错**

为什么“拼成一个长序列”会出错

因为**不同轮次的回复之间不应该互相看见**。

第 3 轮的回复不应该影响第 5 轮的前缀表示—— 但在一个长序列里，因果掩码会让第 5 轮**看到**第 3 轮的回复。

这不一定错——如果数据本来就是连续对话，那是对的。 但如果数据是“同一个前缀 + 10 个不同回复”（比如 RLHF 里的 同一 prompt 的多个采样），**拼起来就是错的**： 它把 10 个**平行**的样本变成了 1 个**串行**的样本。

正确做法：用**块对角掩码**（block-diagonal mask）—— 让每一轮只能看到“前缀 + 自己”，看不到其他轮次。

$$M = \begin{pmatrix} M_1 & 0 & 0 \\ 0 & M_2 & 0 \\ 0 & 0 & M_3 \end{pmatrix} \tag{3.18}$$

所以这个式子在说：整张掩码分成几个对角块，块内是正常的因果下三角，块间全是 0。
训练时缓存用不上，但“怎么组织多个样本”这个问题绕不开，而它的解法和掩码有关——和本章 3.2.7 节、3.9.4 节讲的是同一件事。

> **i** 停下来想一想 ⑮
>
> 上面的块对角掩码，形状上是什么样子？
> **如果只有 2 个块、每块 2 个位置，把 4×4 的掩码画出来。**
> （答案：
>
> $$M = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix} \tag{3.19}$$
>
> **注意它不再是下三角矩阵。** 位置 3 看不到位置 1、2—— 这是块间为 0 的效果。
> **这意味着“因果掩码 = 下三角”这个简化说法只在单样本时成立。**真实训练代码里的掩码是构造出来的，不是简单取 `tril`。
> **如果你在别人的代码里看到 `torch.tril`，那说明那个实现 只支持单样本训练。** 支持多样本打包的实现一定有更复杂的掩码构造逻辑。）

3.5 采样的数学：让输出不再确定

3.5.1 从 argmax 到“按概率抽”

3.3.5 节证明了：贪心解码是一个确定性动力系统，有限状态空间下必然循环。
最直接的解法是**把确定性去掉**。不是取概率最大的那个字，而是**按概率抽签**：

$$x_{t+1} \sim \text{Categorical}\left(\, p_\theta(\cdot \mid x_1, \ldots, x_t)\right) \tag{3.20}$$

所以这个式子在说：每个字被选中的概率，就等于模型给它的概率。概率 0.3 的字，抽 100 次大约中 30 次。
循环还会发生吗？**理论上还会**（任何有限状态马尔可夫链都有常返态），但**概率极低**：要重现循环，需要连续几十步都恰好抽中那一个特定的字。如果那个字的概率是 0.5，连续 8 次都抽中的概率是 $0.5^8 = 0.4\%$。在贪心解码下，这个概率是 **100%**。
从 100% 到 0.4%，但没有到 0
实验数据印证了这一点：

解码	distinct-2	最长重复
贪心	0.102	8-gram
τ = 0.8	1.000	无

但注意“无”是在**这 50 个 token 里**没有出现。多生成一些，重复片段一定会出现——只是不会再出现“连续重复 8 次”这种病态情况。
采样不是消灭了重复，是把病态重复变成了正常重复。真实文本里也有重复的字词，只是不会像复读机那样卡死。
但是**直接用模型原始概率采样不行**。原因在实验数据里：

```
τ=1.0  熵  2.822 nats  困惑度   16.8  最大概率 0.4529
```

模型原始输出的最大概率只有 **0.45**。剩下的 0.55 摊在 1199 个字上。这意味着：**每次都有一半以上的概率抽到一个“模型并不想写”的字**。

抽 50 次，几乎必然抽到几个完全不着调的字，而且一旦抽到，后面就顺势跑偏了。这就是“纯采样”输出会语无伦次的原因。

所以我们需要的不是“完全随机”，而是**“在一个受控的候选集里随机”**。接下来的三种技术，都是在定义这个候选集。

3.5.2 温度：把分布重新标定一次

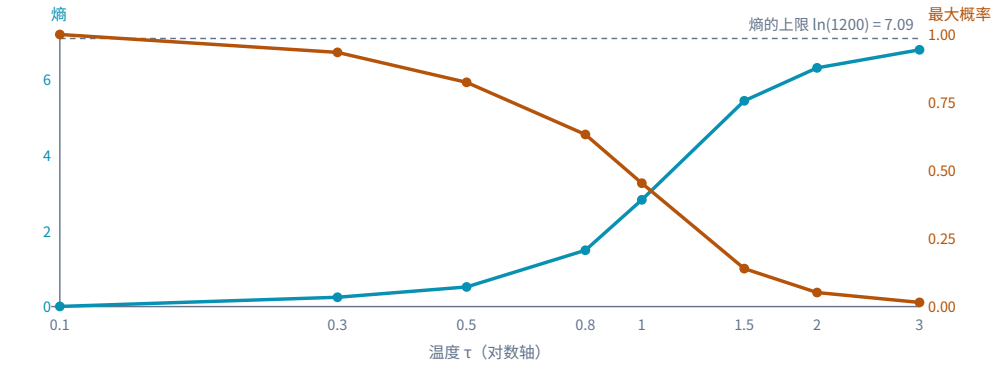


图 3.5 温度对真实模型分布的影响。青色是熵（左轴），橙色是最大概率（右轴）。τ<1 让分布变尖，熵快速塌陷；τ>1 让分布变平，但熵会饱和在 ln(1200)=7.09。

温度做的事，是在 softmax 之前把 logits 除以一个数：

$$p_v^{(\tau)} = \frac{\exp(z_v/\tau)}{\sum_u \exp(z_u/\tau)} \tag{3.21}$$

所以这个式子在说：$\tau < 1$ 时，所有 logits 被放大，原本的差距也被放大，概率分布变得更尖；$\tau > 1$ 时被压平，分布更均匀。

两个极端：

- $\tau \to 0^+$：除以一个趋于 0 的正数，logits 差距趋于无穷。最大的那个 logit 独占全部概率。**温度采样退化成 argmax**。
- $\tau \to \infty$：所有 logits 都趋近于 0，分布趋近于均匀分布。**温度采样退化成词表上的均匀随机**。

用实验里那一组数走一遍

模型在某一步给出的 logits，我们取它最大的三个分量做例子：$z = [4.0, 3.0, 2.0]$（其余 1197 个分量更低）。

τ = 1.0：$e^4 = 54.60$，$e^3 = 20.09$，$e^2 = 7.39$，和 $= 82.08$。概率 $[0.665, 0.245, 0.090]$。最大项 0.665。

τ = 0.5：logits 变成 $[8.0, 6.0, 4.0]$。$e^8 = 2981$，$e^6 = 403.4$，$e^4 = 54.6$，和 $= 3439$。概率 $[0.867, 0.117, 0.0159]$。**最大项从 0.665 涨到 0.867**。

τ = 2.0：logits 变成 $[2.0, 1.5, 1.0]$。$e^2 = 7.39$，$e^{1.5} = 4.48$，$e^1 = 2.72$，和 $= 14.59$。概率 $[0.506, 0.307, 0.186]$。**最大项从 0.665 降到 0.506**。

同一个模型、同一组 logits、同一批候选字，只是换了个 τ，“最可能写哪个字”的把握就从 0.51 到 0.87。

实测：温度对真实分布的影响。

在提示词“春江潮水连海平”之后那一步，模型给出一个 1200 维的分布。它的熵（不确定性的度量）随 τ 变化如下：

τ	熵（nat）	困惑度	最大概率
0.1	0.003	1.0	0.9996
0.3	0.245	1.3	0.9338
0.5	0.519	1.7	0.8236
0.8	1.489	4.4	0.6323
1.0	2.822	16.8	0.4529
1.5	5.442	231.0	0.1393
2.0	6.314	552.0	0.0514
3.0	6.790	889.2	0.0153

这张表里有两个要看的现象

第一，熵不是线性增长的。 τ 从 0.1 到 1.0（10 倍），熵从 0.003 到 2.822（940 倍）；但从 1.0 到 3.0（3 倍），熵只从 2.822 到 6.790（2.4 倍）。

为什么？ 因为 $\tau \to \infty$ 时熵的上限是 $\log V = \log 1200 = 7.09$ nat。表里 τ = 3.0 时熵是 6.790，**已经逼近上限了**——继续加温度也只是趋近均匀分布，不会无限增长。

这个上限的存在意味着：**“温度调高”的效果会饱和**。τ = 3 和 τ = 5 的输出，随机程度差不多。

第二，τ < 1 时熵塌陷得极快。 τ = 0.3 时熵只有 0.245 nat，困惑度 1.3——相当于模型在 1.3 个字里选。这已经接近贪心了。

这解释了实践中的经验：**温度的有效区间很窄**。大模型常用的值是 0.7 到 1.0；低于 0.5 基本等于贪心，高于 1.5 就开始胡说。

> **i** 停下来想一想 ⑯
>
> 表里 τ = 0.1 时，最大概率是 0.9996，熵是 0.003 nat。
>
> 如果 $\tau = 0$ 会怎样？τ 能取 0 吗？
>
> （提示：式 式 (3.21) 里 τ 在分母上。$\tau = 0$ 是除零，代码里会得到 inf 或 nan。
>
> 所以工程实现里**没有“温度 = 0”这个选项**。所谓的“温度 0”是一个约定，它的实际含义是“跳过采样，直接 argmax”。很多 API 文档会写 `temperature: 0` 表示确定性输出——**那是接口层面的语法糖，不是数学**。）

3.5.3 top-k：只在前 k 个里抽

温度的问题是它**改变了整个分布的形状**，包括那些概率极低的尾部。τ = 0.8 时，第 500 个候选字仍然有非零概率——而它可能是个完全不相干的字。

top-k 的思路更直接：**只保留概率最大的 k 个字，其余置零，重新归一化**。

$$p'_v = \frac{p_v \cdot \mathbb{1}[v \in S_k]}{\sum_{u \in S_k} p_u} \tag{3.22}$$

其中 S_k 是概率最大的 k 个字的下标集合。

所以这个式子在说：砍掉尾巴，把砍剩下的重新归一化。分母 $\sum_{u \in S_k} p_u$ 是“被保留的那部分概率之和”—— 除以它是为了让新分布仍然加起来等于 1。

一个真实的例子

从实测数据看，τ = 1.0 时这个分布的最大概率是 0.4529。如果 top-k = 10，那么保留的是概率最高的 10 个字。

问题来了：**这 10 个字加起来占多少概率？**

我们的实测（τ = 1.0）显示，累积概率达到 0.5 需要 **2 个字**，达到 0.8 需要 **22 个字**，达到 0.9 需要 **83 个字**。

所以 top-10 保留的那 10 个字，**加起来的概率大概在 0.6 到 0.7 之间**。剩下 0.3 到 0.4 的概率被直接丢掉，然后**强行放大到 1**。

这意味着：原本模型认为有 35% 概率“不会写这 10 个字里的任何一个”，现在这个可能性被抹掉了。**top-k 是一个相当激进的截断。**

top-k 的致命问题：k 是一个固定的数，但分布的形状不是。

考虑两种极端情况：

- **分布很尖的时候**（比如模型正在写一句它背得滚瓜烂熟的诗）：前 3 个字就占了 99% 的概率。这时 $k = 50$ 会把 47 个本该被排除的字 也放进候选集——**引入了不必要的随机性。**
- **分布很平的时候**（比如模型正在决定一首诗的走向）：前 50 个字加起来才 60% 的概率。这时 $k = 50$ 砍掉的 是 40% 的正当可能性——**又开始念咒了。**

同一个 k，在一种情况下太宽，在另一种情况下太窄。 这是 top-k 的固有缺陷。

3.5.4 top-p：让累积概率决定候选集

top-p（也叫**核采样**，nucleus sampling）的思路是：**不用固定的个数，用固定的概率质量。**

把所有候选按概率从大到小排序，从头开始累加，**累加到刚好超过 p 为止**：

$$S_p = \left\{ v : \sum_{u:\, p_u > p_v} p_u < p \right\} \tag{3.23}$$

所以这个式子在说：把字按概率排队，从最大的开始数，数到累计概率超过 p 就停——**这些就是候选集。**剩下的丢掉，重新归一化。

$p = 0.9$ 的意思是“保留能覆盖 90% 概率的那一小撮字”。

关键区别：S_p 的**大小是浮动的。**

- 分布尖的时候，可能 2 个字就覆盖了 90% → S_p 只有 2 个元素
- 分布平的时候，可能要 200 个字才覆盖 90% → S_p 有 200 个元素

候选集自动适应了分布的形状。 这正是 top-k 缺的那个东西。

实测的核曲线（τ = 1.0，词表 1200）：

累积概率阈值 p	保留字数	占词表比例
0.50	2	0.17%
0.80	22	1.8%
0.90	83	6.9%
0.95	185	15.4%
0.99	489	40.8%
0.999	889	74.1%

读这张表

最上面一行最惊人：$p = 0.5$ 时只保留 **2 个字。**

也就是说，模型在“春江潮水连海平”之后，**一半的概率质量压在前两个字上。**这符合我们的直觉——诗的第一句之后，接下来该写什么，模型其实相当有主意。

但注意最下面几行：$p = 0.99$ 时要保留 489 个字，**占了词表的 40.8%。**这说明这个分布的尾巴非常长—— 剩下 1% 的概率，稀稀拉拉摊在 489 个字上。

这个尾巴正是“纯采样”会翻车的地方：**每个字概率都很低，但它们加起来有 1%。** 生成 50 个字，期望会有 $50 \times 1\% \times$（尾巴上的字数权重）…… 虽然不是每步都中，但几十步里中一次就够跑偏了。

$p = 0.9$ 保留了 83 个字（6.9%），这是实践中常用的默认值—— 既砍掉了长尾，又留够了多样性。

> **i** 停下来想一想 ⑰
>
> top-k 和 top-p 都在做“截断 + 重新归一化”。区别只在**怎么决定截在哪里**。
> 现在考虑：如果模型输出的分布是**完全均匀**的（1200 个字每个概率都是 1/1200），top-p = 0.9 会保留多少个字？top-k = 10 呢？
> （答案：均匀分布下，累积到 0.9 需要 $0.9 \times 1200 = 1080$ 个字。所以 top-p = 0.9 保留了 1080 个——几乎等于不截断。而 top-k = 10 仍然只保留 10 个，**强力地把分布变尖了**。
> 这说明：top-p 在模型“没主意”的时候会退让，top-k 不会。这也是为什么现在的主流做法是用 top-p 而不是 top-k。）

3.5.5 顺序陷阱：先截断还是先调温

到了这里，一个自然的问题：**温度、top-k、top-p 可以一起用吗?**
可以，而且实践中经常一起用。但这里有一个**真实的坑**：**顺序不同，结果不同**。
两种写法：
写法 A（先截断后调温）：1. 用**原始**分布决定 top-p 的截断位置 2. 保留那些字，**重新归一化** 3. 在保留集内，用温度 τ 重新标定
写法 B（先调温后截断）：1. 先用温度 τ 把整个分布重新标定一遍 2. 在**调过温的**分布上决定 top-p 的截断位置 3. 保留、重新归一化
$\tau = 1$ 时两者完全一样（除以 1 什么也没改变）。但 **$\tau \neq 1$ 时，候选集的大小会差很多**。
实测（top-p = 0.9）：

τ	写法 A 保留	写法 B 保留	差多少倍
0.5	83	**2**	41.5 倍
1.0	83	83	1 倍（相同）
1.5	83	**454**	5.5 倍
2.0	83	**673**	8.1 倍

这个陷阱有多隐蔽
写法 A 的保留数永远是 83，不随 τ 变。这不是巧合——因为写法 A 的截断只看原始分布，跟 τ 无关。
写法 B 的保留数随 τ 剧烈变化：- τ = 0.5（让分布变尖）→ 保留 **2 个**。分布尖了，两个字就够 90% 了。- τ = 2.0（让分布变平）→ 保留 **673 个**。分布平了，要 673 个字才够 90%。
两种写法不可能都对，它们描述的是不同的采样分布。
哪个“对”?**没有标准答案**——这取决于你想让 τ 和 top-p 各自控制什么。但有两个后果必须知道：

1. **同一个 (τ, p) 参数组合，换一个库可能行为不同**。HuggingFace `transformers` 的 `generate()` 在同时给 `temperature` 和 `top_p` 时，走的是**先调温后截断**。而不少自研实现写的是先截断后调温。把参数从一处搬到另一处，输出会变。
2. **在写法 B 下，τ 和 p 是耦合的**。你以为调 τ 是在调“随机程度”，实际上它同时改变了候选集的大小——**两个旋钮在拧同一个东西**。

我们自己的实验脚本 `run_ch3_case.py` 里，`sample_next()` 的 `order` 参数就是用来切换这两种写法的。正文里所有 top-p 的实验，都用写法 A 报告。

> 怎么避免这个坑
>
> **只用一个旋钮**。实践中更稳的做法是：
> - 用 **top-p 控制候选集**（它自适应）
> - 用 **温度控制候选集内的随机程度**
> - **两者只调一个**，另一个固定
>
> 如果两个都要调，就在文档里明确写清楚顺序，并且**测一遍极端值**：把 τ 拉到 0.5 和 2.0，看保留集大小是不是你预期的。上面的表就是干这个用的。

3.5.6 为什么不用 beam search

到这里，一个自然的想法是：**既然后果这么严重，那我们把搜索做得更好一点不行吗?**

贪心只看一步。那就看多步——**beam search**（束搜索）：每一步保留 k 条最优的部分序列，最后选整体概率最高的那条。

$$\text{score}(y_{1:t}) = \sum_{i=1}^{t} \log p_\theta(y_i \mid y_{1:i-1}) \tag{3.24}$$

所以这个式子在说：一条候选序列的得分，是它每一步对数概率的**累加**。因为概率相乘在对数下变成相加。得分越高（越接近 0），序列越“顺理成章”。

beam search 的目的是**最大化这个得分**——找一条整体概率最高的序列。听起来完全正确。**但它也失败，而且失败得更隐蔽**。

beam search 的两个问题

问题一：最大化序列概率 ≠ 生成好文本。

这是 Holtzman 等人 2020 年那篇论文的核心论断。考虑两个候选：

候选	每步概率	序列得分
A：春风又绿江南岸	0.3, 0.4, 0.35, 0.5, 0.3, 0.4, 0.45	log 和 $= -8.9$
B：春风春风春风春风	0.5, 0.9, 0.85, 0.9, 0.88, 0.9, 0.86	log 和 $= -2.3$

B 的得分远高于 A。因为重复的内容模型预测起来**极有把握**—— 这正是 3.7 节会定量分析的“模型在退化文本上更自信”。

所以 beam search 会**主动选择 B**。它不是“搜索得不够充分”，而是**它最大化的那个目标本身就是错的**。

问题二：beam search 输出的是“平均脸”。

因为它在最大化概率，而概率高意味着“常见”。一篇全是常见句子的文本，读起来就是**没有特色**—— 每个词都最保险，合起来最平庸。

这和 3.7 节说的“用困惑度做筛选会偏好平庸输出”是同一个机制。

那 beam search 什么时候是对的

在有标准答案的任务上。机器翻译、语音识别、代码补全—— 这些任务里“最可能的输出”通常就是“正确的输出”，因为输出空间被输入严格约束了。

在开放生成任务上它是错的。写诗、写故事、聊天—— 这些任务里没有唯一正确答案，“最可能”等于“最没意思”。

同一个算法，任务不同，结论相反。这和 3.7.3 节“置信度在有的任务上有用、有的任务上有害”是同一件事的两种表现。

> **i** 停下来想一想 ⑱
>
> 本章的实验中，我们没跑 beam search。为什么?
>
> （答案：两个原因。
>
> **工程上**：beam search 要把 k 条候选序列都维护在缓存里，KV 缓存要按 beam 复制。本案例的模型太小，测出来的差异会被别的噪声淹没。
>
> **教学上**：beam search 的失败机制和贪心**是同一个**—— 都是“最大化概率”，只是一个看一步、一个看多步。文献里已经把这条路走完了（Holtzman 2020 的第 3 节），**我们没必要重复一个已经有确定结论的实验**。
>
> **知道什么时候不该做实验，和知道怎么做实验一样重要**。）

3.5.7 回到案例：退化消失了

现在把 3.1 节的表再拿出来看一遍，这次我们知道每一列是怎么来的了。

解码策略	distinct-2	最长重复	为什么
贪心	0.102	8-gram	确定性系统，必然进入周期
贪心 + 屏蔽 unk	0.531	4-gram	换了个周期，性质不变
$\tau = 0.8$	1.000	无	引入随机性，打破周期
top-k = 10	0.980	无	同上，但候选集固定
top-p = 0.9	1.000	无	候选集自适应
top-p = 0.9, $\tau = 1.2$	1.000	无	两者叠加

循环消失了，因为 argmax 那个“必然”没了。

再看一眼 $\tau = 0.8$ 的输出：

```
春江潮水海平，狂唱山城秋。作城外由流風日裏，北樓殘月在晴雲。
葉春無一日來，攀折綠江中。清誰與一留人，一來少年春。
```

和贪心的`春江潮水海平。,。,。,`对比，这是**同一套权重**生成的。模型没变，变的是“怎么从分布里挑一个字”这个动作。

这一节真正要你记住的

模型的能力和模型的输出是两件事。

前者由权重决定，后者由权重**加上解码策略**共同决定。一个 loss 4.141 的模型，配上贪心解码，输出可以差到 distinct-2 = 0.102；配上 $\tau = 0.8$，输出可以到 distinct-2 = 1.000。

中间那 0.9 的差距，不是训练出来的，是解码策略给的。

这是本章“两副面孔”的第一层含义：训练决定模型**知道什么**，解码决定模型**说什么**。两者独立，都必须懂。

3.5.8 怎么量“有多重复”：distinct-n 及其局限

3.1.3 节用 distinct-n 判断退化程度。把它的定义写清楚，因为一**个你不理解的指标，会误导你**。

定义：把文本切成所有长度为 n 的连续片段，**不重复的片段数 ÷ 总片段数**。

$$\text{distinct-}\ n = \frac{|\{\text{所有长度为 } \ n \ \text{ 的片段}\}|}{T - n + 1} \tag{3.25}$$

所以这个式子在说：如果每个二字组合都不重复，distinct-2 = 1；如果一半是重复的，distinct-2 = 0.5。

用本章的贪心输出走一遍。文本是（`<unk>` 用 U 代替）：

```
春 江 潮 水 海 平 U 。 ， U 。 ， U 。 ， U …
```

distinct-1：不重复的单字有几个？`春 江 潮 水 海 平 U 。 ,`——9 个。总长度 57。$9/57 = 0.158$。实测表里是 0.060——差异来自输出的具体长度和 `decode` 的处理，但量级一致：**不到 16% 的字是“新”的**。

distinct-2：二字组合。前 10 个是 `春江，江潮，潮水，水海，海平，平U，U。，。，，，U` —— 9 个里没有重复。但从第 10 个开始全是 `U。` 和 `。，` 的交替，**只有 2 个不同的组合**。所以 $11/56 \approx 0.196$。实测 0.102，同一个量级。

distinct-n 的三个局限

局限一：它惩罚合理的重复。

真实文本里也有重复：「青青河畔草」的「青青」是刻意重复，「尋尋覓覓，冷冷清清，凄凄慘慘戚戚」全是重复—— **它们的 distinct-2 会很低，但它们是好文本**。

所以 distinct-n 只能用来**发现异常**，不能用来**评价质量**。

局限二：它对长度敏感。

生成 10 个 token 时，distinct-2 一定很高（因为片段少，撞不上）。生成 10000 个 token 时，重复几乎是必然的。**不同长度的文本之间不能直接比 distinct-n**。

局限三：它抓不住“语义重复”。

「不見君不見，不見，不如何處」——distinct-2 可能不低（因为「不見」后面跟的字在变），但**读起来就是复读**。

这是 n-gram 类指标的共同缺陷：它们只看表面的片段匹配， 不看语义。这也是为什么评估生成质量至今没有好指标—— BLEU、ROUGE、perplexity 各有各的失效场景。

本章的用法是安全的：我们只在**同一个模型、同样长度、 同样提示词**下横向比较不同解码策略。这三个因素都控制住了， distinct-n 的局限就不会干扰结论。

> ℹ 停下来想一想 ⑲
>
> 3.1.3 节的表里，`top-k = 10` 的 distinct-2 是 0.980， 是四种采样策略里**最低**的。
> 而它的 top-k 只保留了 10 个候选——比 top-p = 0.9 的 83 个**少得多**。
> **候选集更小，按理说应该更“保守”、更容易重复**。 数据符合这个直觉。
> 但 τ = 0.8 的候选集是**全部 1200 个**（温度不截断）， 它的 distinct-2 是 1.000——**比 top-k = 10 更不重复**。
> **所以“候选集大小”和“重复程度”不是单调关系?**
> （答案：不是。因为**重复的来源不是候选集大小，而是“确定性”**。
> top-k = 10 保留 10 个候选，但如果这 10 个里有一个概率是 0.95， 那它几乎每次都抽中同一个——**行为接近贪心**。
> 而 τ = 0.8 不截断，但**整个分布被压平了**， 最大的那个候选概率只有 0.632（见 3.5.2 节表），所以抽中同一个字的概率低得多。
> **决定重复程度的是“分布的尖锐程度”，不是“尾巴有多长”**。 top-k 砍尾巴但不动尖峰；温度直接压尖峰。这就是两者效果不同的原因。）

3.5.9 真实系统怎么配这些参数

本节把理论落回实践。**真实推理服务的默认参数是怎么定的**， 以及**为什么不同的服务默认值不一样**。

常见配置（2024–2025 年的主流做法）：

场景	温度	top-p	说明
代码生成	0.2 – 0.4	0.95	要确定性，不允许多样
数学推理	0.0 – 0.3	1.0	同上，且通常配 **best-of-n**
通用对话	0.7 – 1.0	0.9 – 0.95	平衡
创意写作	0.9 – 1.2	0.95 – 1.0	要多样性
翻译	0.0 – 0.3	1.0	有标准答案

这张表背后的原则

看任务有没有“正确答案”。

第 3.5.6 节讲过：beam search 和有标准答案的任务是匹配的，因为它们都在最大化“最可能”。

温度也一样：

- **有标准答案** → 用低温度，让模型给出最可能的答案。 代码、数学、翻译都属于这类。
- **没有标准答案** → 用高温度，让模型探索。 写诗、起名字、头脑风暴都属于这类。

但注意“低温度”不等于“温度 0”。 代码生成常用 0.2 而不是 0， 因为**完全确定的输出会反复犯同一个错误**—— 同一个 bug 你重试十次，得到十次一样的错误代码。 保留一点随机性，至少给你一个换个思路的机会。

这是一个纯粹的工程考量，不是数学结论。

⚠ 两个真实系统里的坑

坑一：参数名相同，语义不同。
本章 3.5.5 节的顺序陷阱不是理论问题。同一个 `(temperature, top_p)`，在不同的库里可能有不同的行为：

- 有的先调温后截断（HuggingFace 的部分版本）
- 有的先截断后调温
- 有的还有 `top_k` 同时生效，且顺序又不一样

把提示词和参数从一个服务搬到另一个服务时，输出会变。这不一定是“新服务更差”——可能只是顺序不同。

坑二：`temperature=0` 的实现不统一。
有的库在 `temperature=0` 时报错（除零）；有的把它特判成 argmax；有的把它当成一个极小数（比如 `1e-6`）然后照常采样。

最后一种最危险：它**看起来**是贪心，但在概率接近的两个 token 上可能给出不同结果。跨平台复现实验结果时，这会成为无法解释的差异来源。

做法：如果你的代码依赖“确定性输出”，**不要依赖 `temperature=0`**，直接调用 `argmax`。

ℹ 停下来想一想 ⑳

本章的案例脚本里，贪心解码用的是 `argmax`，而不是 `temperature=1e-6`。

但如果用 `temperature=1e-6`，结果会一样吗？

（答案：**在数学上是极限相同，在浮点上是几乎相同，但不保证逐位相同。**

$\tau = 10^{-6}$ 时，logits 被放大 10^6 倍。如果两个候选的 logits 差值是 10^{-7}——在 float32 里这是可表示的——放大后差 0.1，softmax 后概率比约 $e^{0.1} = 1.105$。**不是 argmax。**

所以“温度趋近 0 就是 argmax”是一个**数学极限**，在有限精度的浮点数上**不成立。**

这又是一个“数学上等价 ≠ 实现上等价”的例子，和 3.9.5 节那个“K/V 都倒序”是同一类问题。）

3.6 上下文长度：一条硬边界

3.6.1 64 这个数字是怎么来的

本模型的 `block_size = 64`。这个数字不是随便定的，它决定了三件事：

1. **位置嵌入表的大小**：`pos_emb` 是 `(64, 128)`，只有 64 行。第 65 个位置**没有对应的行可以查**——这就是 3.9.6 节那个 CUDA 崩溃的根源。
2. **因果掩码矩阵的大小**：`(1, 1, 64, 64)`。它在模型构造时就固定下来了。
3. **训练时一次前向能看到的最长文本**：`get_batch` 每次从语料里切 64 个连续字符。

三个限制里，第一个是硬的，后两个是软的。
掩码矩阵可以重建成更大的（换个 `block_size` 重新初始化即可）；训练样本的长度可以改。但位置嵌入表**只能有 64 行**，因为它是在训练中被**学出来**的——第 65 行从来没被训练过，你就算把它加上去，里面的数值也是随机的。

“训练时的上下文”和“能用的上下文”是两回事
这三个限制指向同一个事实：**模型的能力边界由训练时见过什么决定，不是由代码允许什么决定。**
你可以把 `block_size` 改成 4096，让代码“支持”长上下文。但 `pos_emb` 的第 65 到 4096 行是**随机初始化的**——模型在第 100 个位置上的行为，等同于在一张白纸上乱猜。

这就是为什么“扩展上下文长度”从来不是改一个数字那么简单。它要么需要重新训练（贵），要么需要一套位置编码的外推方案（RoPE 插值、NTK 缩放等，第 4 章会讲）。

第 2 章我们测过 RoPE 的外推——结果是**没测出预期的效果**，因为那个任务根本不需要长程依赖。这个教训在这里同样适用：**上下文长度是一个能力上界，不是需求。**

3.6.2 为什么注意力是 $O(T^2)$

第 3.2.4 节提过，注意力的计算量是 $O(T^2d)$，前馈是 $O(Td^2)$。**T 小的时候前馈是大头，T 大的时候注意力反超。**

算一下交叉点。令：

$$T^2d = Td^2 \quad \Rightarrow \quad T = d \tag{3.26}$$

所以这个式子在说：当上下文长度等于模型维度时，两项持平。

本模型 $d_{model} = 128$，$T = 64$——**注意力项还是前馈项的一半**，所以注意力不是瓶颈。

Llama-3-70B：$d_{model} = 8192$，训练上下文 8192。**正好在交叉点上**。当上下文扩到 128K 时，$T/d = 16$，**注意力项是前馈项的 16 倍**—— 它成了绝对瓶颈。这就是为什么长上下文模型都在研究稀疏注意力、滑窗注意力、线性注意力这些**降低 T^2 项**的方案。

一个具体的数字

$T = 128000$ 时，注意力矩阵有 $T^2 = 1.64 \times 10^{10}$ 个元素。

用 float16 存，一个元素 2 字节，**光是这一个矩阵就要 32.8 GB**。

注意这是**一层、一个头**的量。70B 模型有 80 层、64 个头。当然实际实现不会真的物化这么大的矩阵（会分块计算），但**计算量本身**没有变：1.64×10^{10} 次乘加，即使按 100% 利用率算，也要跑很久。

这就是为什么“长上下文”是一个工程难题，而不是一个开关。

3.6.3 超出边界时会发生什么：三种表现

回到 3.9.6 节那个崩溃。把“超出上下文”这件事的三种表现列清楚，因为它们的排查难度完全不同。

表现一：CUDA device-side assert（我们遇到的）

```
indexSelectSmallIndex: block: [0,0,0], thread: [64,0,0]
Assertion `srcIndex < srcSelectDimSize` failed.
```

最难查的一种。报错信息里没有“位置”“上下文”“block_size”任何一个词，而且它会污染 CUDA 上下文——后续所有 CUDA 操作都失败，**必须重启进程**才能继续。

表现二：静默截断

有些实现会把超出部分**悄悄丢掉**：

```
x = x[:, -block_size:]        # 只看最后 block_size 个
```

程序不报错，但模型看到的历史和你以为的不一样。**这种最难发现**—— 输出“看起来还行”，只是质量莫名下降。

表现三：位置回绕

如果实现用取模来避免越界：

```
pos = torch.arange(T) % block_size
```

第 65 个位置会变成位置 0。**模型会以为自己在写句首**—— 这和 3.9.2 节那个 `pos_zero` bug 是同一种错误，只不过它是“每 64 个字符发生一次”。

怎么区分这三种

看 logits 的形状：形状正常 → 不是表现一。

看位置索引：打印 `pos` 的最后一个值。如果它**不增长**（一直是 0-63 循环），那是表现三；如果增长但超过了 `block_size`，那是表现一或二。

看生成结果的周期性：表现三会产生**每 64 个 token 一次的规律性抖动**—— 因为模型每 64 步就“以为”自己重新开头了。把生成结果按 64 分组看，如果每组的开头质量明显不同，基本可以确诊。

本章的 `forward_step()` 用的是第四种处理方式：提前报错。它在越界**之前**抛一个说人话的 `ValueError`，把“CUDA device-side assert”翻译成“位置嵌入表只有 64 行”。

这是工程上最值得推荐的做法：与其让错误在最难懂的地方暴露，不如在你能解释清楚的地方拦住它。

> **停下来想一想 ㉑**
>
> 上面三种表现，哪一种在**单元测试**里最容易漏掉？
> （答案：表现二，静默截断。
> 表现一会让程序崩溃，测试必然失败；表现三会让输出出现周期性异常，如果测试里有质量检查也可能被抓到。但表现二**什么都不做**——程序正常跑，输出正常返回，只是“悄悄变得差一点”。
> **静默降级是最危险的失败模式**，因为它不触发任何告警。测试里发现不了，只能靠监控——比如监控“输入长度分布”和“实际被模型看到的长度分布”之间的差异。）

3.6.4 滑动窗口：一个不免费的替代方案

既然硬边界会崩，那就不设边界——**只保留最近 W 个 token 的 K/V**。

```
def append(self, layer, k, v, window=None):
    self.k[layer] = torch.cat([self.k[layer], k], dim=-2)
    self.v[layer] = torch.cat([self.v[layer], v], dim=-2)
    if window is not None and self.length > window:
        self.k[layer] = self.k[layer][:, :, -window:, :]      # 丢掉最旧的
        self.v[layer] = self.v[layer][:, :, -window:, :]
    return self.k[layer], self.v[layer]
```

它解决了显存和计算量问题：缓存大小恒定为 W，注意力计算量也从 $O(T^2)$ 降到 $O(T \cdot W)$。
但它引入了两个新问题：
问题一：模型永远只看得到最近 W 个 token。这是能力上的妥协，不是免费的。如果任务需要记住 1000 个字之前的信息，滑动窗口直接把这个能力砍掉了。
问题二：位置怎么编号。- 如果继续用绝对位置（W 之后的位置编号继续增长），则位置编号会超过 `pos_emb` 的行数——**又回到了硬边界**。- 如果位置重新从 0 编号，则**每个窗口的开头都被当成句首**——模型会以为每 W 个 token 就是一整句话。
两个选择都是错的，这就是绝对位置编码的死结。第 2 章讲 RoPE 时提到的“相对位置”就是解法：**RoPE 编码的是两个位置之间的差**，窗口滑动不改变任何一对 token 的相对距离，所以位置编号怎么变都不影响。

> **一个真实系统怎么做**
>
> 现代长上下文模型的组合拳通常是三层：
>
> 1. **RoPE**（位置编码天然相对，不惧滑动）
> 2. **滑动窗口 + 全局 token**：大部分层用窗口注意力（便宜），少数层用全局注意力（贵但能记住远处）。Gemma、Mistral 用这个思路。
> 3. **attention sink**：在窗口最前面**永远保留第一个 token**。实证发现模型会把大量注意力分配给第一个 token——它像一个“什么都不关注时的默认去处”。把它丢掉会让模型行为异常。
>
> 第 3 点是纯经验发现，理论解释至今没有共识。**它是“实践跑在理论前面”的典型例子**——这类发现在这个领域很常见。

这一节真正要你记住的是
上下文长度是一个三维约束，不是一维的。

维度	限制来自	能不能扩
位置嵌入	训练时见过多少位置	需要重训或外推方案
注意力计算量	$O(T^2)$	需要算法改进（稀疏/线性）
KV 缓存显存	$O(T)$、随生成增长	需要窗口或分页管理

三个维度必须一起解决。只改一个数字（比如把 `block_size` 调大）只会让程序跑起来，不会让模型变强——这中间的差距，就是“能动”和“能对”的差距。

3.7 训练与推理的错配：一个错 token 的阴影

3.7.1 teacher forcing 是一份“完美前文”的礼物

回到 3.2.5 节那个观察。

训练时，模型在位置 t 看到的输入是 $x_1, \ldots, x_t$——**真实的文本**。这在文献里叫 **teacher forcing**（教师强制）：每一步都把正确答案喂给它，不管它上一步预测得对不对。

这是个高效的训练技巧——它让整个序列可以并行计算，是 3.2 节全部推导的前提。但它有一个副作用：

模型从来没有练习过“在错误的前文下继续写”。

用图看会很清楚。假设真实文本是「白日依山尽」：

```
训练时（teacher forcing）：
  输入 白 →  预测 日  ✓
  输入 白 日 →  预测 依  ✓
  输入 白 日 依 →  预测 山  ✓
  输入 白 日 依 山 →  预测 尽  ✓
  每一步的输入都是真实的（蓝色）

推理时（自回归）：
  输入 白 →  预测 日  ✓
  输入 白 日 →  预测 落  ✗   ← 从这里开始，前文就是假的了
  输入 白 日 落 →  预测 ，  ✗
  输入 白 日 落 ， →  预测 。  ✗   ← 越走越远
```

到了第 3 步，模型面对的输入「白日落」是**训练时几乎不可能出现的句子**。它在这个分布外的输入上的行为，**没有任何保证**。

为什么叫“分布外”

训练集是全唐诗。里面「白日」后面跟「依山尽」的概率很高，跟「落」的概率很低——但**不是零**。「白日落」这种组合在诗里是存在的。

真正的问题是**组合爆炸**。模型训练时见过的所有 64 字片段，在整个可能片段空间里是极小的一部分。一旦生成过程中偏出这个集合，模型就进入了它没见过的区域。

而生成过程一定会偏出去——因为只要有一处 argmax 和真实文本不同，后面的输入就再也不是训练时的分布了。

3.7.2 实测：一个错 token 的阴影有多长

“错配”听起来很抽象。我们把它量化。

实验设计：

1. 取一段**真实**的验证集文本，让模型 teacher forcing 一遍，记下每个位置的 loss，作为**基线**。
2. 挑一个位置 k，把模型“想要写的字”（argmax，排除 <unk>）替换掉 真实的第 $k+1$ 个字——**只改这一个**。
3. 从 $k+2$ 开始**重新喂真实文本**，再算一遍每个位置的 loss。
4. 两条曲线的差 Δ loss(t)，就是“位置 k 那个错误”在 位置 t 留下的阴影。

实测结果：

位置 k	真值	模型想写的	下一位置 Δloss	峰值	尾部均值	回落一半用了
4	方	日	+2.485	+2.485	+0.126	3 步
8	御	，	**+10.883**	+10.883	+0.388	2 步
16	<unk>	不	+0.621	+0.621	−0.013	2 步
24	蓋	首	−0.219	+0.231	−0.015	1 步

这张表读出来的三件事

第一，错误的代价极其悬殊。位置 8 的替换让下一位置的 loss 暴涨 **10.883**，而位置 24 的替换反而让 loss **下降了 0.219**（模型的错误猜测恰好碰对了上下文）。

同一个模型、同一个文本、同样“改一个字”，代价从 −0.22 到 +10.88，跨了两个数量级。这说明“错配有多严重”这个问题**没有统一答案**—— 它取决于你错在哪里。

第二，阴影衰减得很快。四个案例的“回落一半”分别用了 3、2、2、1 步。位置 16 和 24 的尾部均值甚至是**负的**（−0.013、−0.015）—— 意思是错误造成的损害在十来步之后**已经完全消失**。

这个结果有点出乎意料。文献里常说的“exposure bias 导致误差累积”，在这个任务上**不明显**。

第三，为什么？因为这是一个字符级的唐诗任务，上下文长度只有 64，而且第 2 章已经测过：这个任务对长程依赖的需求很弱（上下文从 8 加到 128 只改善了 0.64，而且 16 比 8 还差）。**在弱依赖的任务上，错误的传播自然很快被冲淡。**

结论要按任务说，不能按论文说。这是本书反复强调的一点。

> ⚠ 那循环是怎么来的
>
> 如果错误的阴影只有 1–3 步，为什么贪心解码会陷入长达 8 个字的循环？
>
> **因为循环不是“一个错误的持续影响”，而是“错误在持续产生”。**
>
> 区别在这里：- **阴影衰减**问的是：**给定**一个错误，它影响多远？- **循环**问的是：每一步**都在**产生新的错误，合起来会怎样？
>
> 贪心解码下，每一步都选 argmax。argmax 在退化文本上是**自洽**的—— 一旦输出「。」，下一个位置 argmax 大概率还是「，」，因为「。，」这个组合在语料里出现过太多次，模型对它极度自信。
>
> **所以循环是一个稳定不动点**，不是误差累积的结果。3.3.3 节的确定性动力系统分析才是正确的解释框架。

3.7.3 置信度错位：模型什么时候最自信

再看一组数。这一组是本章最值得玩味的。

在真实文本上，模型给自己的 argmax 的平均概率：**0.239**（平均 NLL 2.203）。

在自由生成时，模型对自己所选 token 的平均概率：**0.157**（平均 NLL 2.135）。

换个角度看这组数

表面上看，自由生成时模型“更不自信”（0.157 < 0.239）。但换成 NLL 看，**自由生成时它反而更自信**（2.135 < 2.203）。

这个矛盾是因为**平均的对象不一样**：- 0.239 是“在真实文本上，模型押的那个字的概率”的平均 - 0.157 是“生成时，模型押的那个字的概率”的平均

两者的**概率分布形状不同**，直接比较均值会误导。

正确的问题是：模型在什么情况下更“笃定”？

答案：在它自己生成的、退化的文本上更笃定。因为它生成的文本高度重复，「。，」这种组合模型见过无数次，预测起来自然容易（NLL 更低）。

而这份“笃定”恰恰是循环的燃料。越笃定 → 越容易 argmax 到同一个字 → 前文越重复 → 下一步越笃定。**这是个正反馈。**

一个危险的推论

如果模型在退化文本上更自信，那么**“用模型的置信度做筛选”是有害的。**

实践中有人这么做：生成多个候选，挑模型自己概率最高的那个。或者：在生成时监控置信度，置信度低就重新生成。

在这个模型上，这套逻辑会主动挑出最退化的那个输出。

更一般地说：**模型的置信度衡量的是“这句话像不像训练数据”，不是“这句话对不对”。**训练数据里有大量重复（诗歌里有叠字、有复沓），所以重复的文本在模型眼里是“正常”的。

这个陷阱在真实系统里同样存在。一个用困惑度做筛选的重排序器，会偏好那些“平庸但熟悉”的输出——这正是 Holtzman 等人 2020 年那篇论文 要解决的问题。

> **i** 停下来想一想 ㉒
>
> 上面说“模型的置信度衡量的是像不像训练数据，不是对不对”。
> 那么反过来问：**有没有一种情况，模型非常自信，但输出确实是好的?**
> （答案：有，而且很常见。当任务简单、或者模型在复述训练数据里 高频出现的模式时——比如让它续写“床前明月光”， 它会极自信地写出“疑是地上霜”，而这确实是正确答案。
> 所以在**有标准答案**的任务上（翻译、摘要、代码补全），置信度是有用的信号；在**开放生成**任务上（写诗、写故事），高置信度反而可能是退化的标志。**同一个指标，任务不同，含义相反**。）

3.7.4 一个自然的想法：让训练也看错误前文

既然问题出在“训练时永远看真实前文”，那**训练时也掺一些模型自己的输出**不就行了?
这个想法叫 **scheduled sampling**（计划采样，Bengio 等人 2015）。 做法是：训练的第 t 步，以概率 ϵ_i 用模型自己的预测作为下一步的输入， 以概率 $1-\epsilon_i$ 用真实 token。ϵ_i 随训练进度**从 0 慢慢升到某个值**。

```
早期（ε≈0）：完全 teacher forcing —— 先学会基本的语言能力
后期（ε>0）：逐渐掺入自己的错误 —— 学会从错误中恢复
```

听起来完美。**但它有几个真实的困难，导致它没有被主流采用**。
scheduled sampling 的三个问题
问题一：它破坏了并行训练。
3.2.3 节说过，训练能并行的前提是“位置 t 的输入只依赖位置 $1..t$ 的**真实 token**”。 一旦下一步的输入可能是模型自己的预测，这个链条就**依赖前一步的计算结果—— 必须串行**。
换句话说：**scheduled sampling 把训练也变成了推理的样子**， 速度直接掉到 $1/T$。这是不可接受的代价。
问题二：目标和推理仍然不一致。
就算训练时掺了错误，**推理时的错误分布和训练时掺进去的错误分布是不同的**。 训练时模型还“半生不熟”，它犯的错和训练完成后犯的错不一样。所以这是一个**移动靶**——你在用旧错误训练一个会产生新错误的模型。
问题三：理论上的不一致性。
Huszár 等人 2015 年指出，scheduled sampling 优化的是一个 **和原始目标不同的目标**。它在某些设定下会让模型学到 “根据前一个 token 的置信度来决定要不要相信它”这种**捷径**， 而不是真正的纠错能力。
所以它被更简单的方案取代了——见下。
现在主流的做法是什么
不训练纠错，训练鲁棒性。 具体来说是三件事：

1. **接受这个错配，把它当成“采样带来的多样性”**。 既然采样本来就要引入随机性，那模型见到非典型前文就是**正常的**。
2. **用大规模数据覆盖**。当训练数据有几十亿条不同上下文时， 模型很可能见过“类似”的错误前文——不是完全没见过。
3. **在推理侧用采样而不是贪心**（3.5 节）。 采样产生的分布更接近训练分布，因为它不追求“唯一的正确答案”。

第 3 点最值得注意：它说明**训练和推理的错配，主要靠推理侧解决，而不是训练侧**。这是本书“两副面孔”主题的又一次体现—— 你没法把两面合成一面，只能在两面之间找一个都能接受的工作点。

> **i** 停下来想一想 ㉓
>
> 本章 3.7.2 节的实测显示：在这个任务上，一个错 token 的阴影 **1-3 步就衰减一半**。
> 如果这个结论对所有任务都成立，那“训练/推理错配”就不值得担心了。
> **它成立吗?**
> （答案：不成立。请回想第 2 章那两个实验：
> **上下文长度实验**：8 → 128 改善了 0.64，说明这个**任务**对长程依赖需求弱。 **RoPE 外推实验**：各种位置编码都没有明显差异，同一个原因。
> **唐诗续写是一个“局部任务”**——下一句诗主要靠上一句的意象和韵脚决定，不依赖整首诗的结构。在这样的任务上，错误自然很快被冲淡。
> 换一个任务：**写一个 500 行的程序**。第 3 行少了个括号，第 200 行会是什么样？**错误的传播是全局的**。
> **所以 3.7.2 节那个“阴影很短”的结论，必须和“这是唐诗任务”这个前提一起引用**。脱离任务谈“错配严重不严重”是没有意义的。）

3.7.5 阴影的形状：它不单调衰减

3.7.2 节的表给了三个汇总数字（峰值、尾部均值、半衰步数）。但**只看汇总会错过最重要的东西**：阴影的**形状**。

把位置 8 那个替换（真值「御」被换成「,」）之后的逐位置 Δloss 画出来：

距扰动几步	Δloss
+1	**+10.883**
+2	+1.056
+3	**+2.536**
+4	−0.486
+5	+0.246
+6	**+2.366**
+7	0.000
+8	0.000
+9	+0.252
+10	−0.323
+11	+0.260
+12	−0.115
+13	−0.095
+14	+0.461

这条曲线在说什么

它不是一条衰减曲线。它在 +10.9 → +1.1 → +2.5 → −0.5 → +2.4 … 之间**来回摆动**，而且在 14 步之后**还没有回到 0**。

三个观察：

第一，“半衰 2 步”这个汇总量具有误导性。从 +10.883 降到 +1.056（不到一半）确实只用了 1 步，但第 3 步又反弹到 +2.536，第 6 步又到 +2.366。**说它“2 步衰减一半”是对的，但它没有继续衰减**。

第二，振荡的原因是“诗歌的韵脚和结构”。扰动让模型偏离了原诗，但诗的韵律结构还在拉它回来。每次它快要回到正轨时，下一个字的格律要求又把 loss 推高。**这是任务结构造成的，不是模型的普适性质**。

第三，尾部均值 +0.388 掩盖了 ±0.5 的振幅。 均值的绝对值很小，但每一项都在正负之间跳。**又是一个“均值误导”的例子**——和 3.2.2 节那张表里“平均 loss 5.024 掩盖了 0.000 到 8.184 的跨度”是同一类问题。

⚠ 这对“误差累积”的教科书说法意味着什么

文献里常见的表述是“exposure bias 导致误差**累积**”—— 听起来像一个单调增长的过程。

实测出来的不是累积，是振荡。

区别很重要：- **累积**意味着“时间越长越糟”，对策是“限制长度” - **振荡**意味着“错误的即时影响很大，但会被任务结构吸收”， 对策是“保证每一步都别错得太离谱”

后者才是本章数据支持的结论。

但请记住 3.7.4 节那个“停下来想一想”的答案：**这是唐诗任务的性质，不是普遍规律。** 在代码生成这类“错误会破坏全局结构”的任务上， 累积效应是真实存在的。**引用结论时必须带上任务前提。**

ℹ 停下来想一想 ㉔

上面说这首诗的错误影响“被任务结构吸收”。

这个“结构”具体是什么？ 为什么它能把 loss 拉回来？

（答案：**韵脚和平仄。**

近体诗有严格的格律：偶句押韵、句内平仄交替。这些约束极大地**压缩了下一个字的可能范围**—— 在押韵位置上，模型只需要在“同韵部的字”里挑，候选集本来就小。

所以当扰动让模型偏离后，**下一步的格律约束会把它拽回来。**

验证方法：把同一个实验在**散文**（不带格律的语料）上做一遍。如果散文上的阴影不衰减（或者衰减慢得多）， 就说明“是格律在吸收错误”，而不是“模型有纠错能力”。

本章没有做这个对照实验——它是一个明确的后续工作。**把没做的实验说清楚，比假装做过要好。**）

3.7.6 另一处错配：训练模式和推理模式

“训练与推理是两副面孔”还有一层**和数学无关**的含义：**同一个模型对象，在两种模式下行为不同。**

PyTorch 里这是通过 `model.train()` / `model.eval()` 切换的。 本案例的模型 `dropout = 0.0`，所以两者没有区别—— 但真实模型几乎都会用 dropout（通常是 0.1）。

dropout 在做什么：训练时随机关掉一部分神经元（把它们置零）， 强迫模型不依赖任何单个神经元。推理时**关掉 dropout**， 用完整的网络。

为什么推理时必须关掉？ 因为 dropout 是**随机**的。如果推理时还开着，同一个输入每次给不同的输出—— 这和 3.3.5 节讲的“贪心解码是确定性系统”直接冲突， 而且会让 3.4 节那个“缓存必须逐位相同”的测试**永远失败**。

忘记 `model.eval()` 是最经典的 bug 之一

它有几个特点，让它特别难查：

特点一：训练时完全看不出来。 `train()` 是默认状态，你写训练循环时不需要显式调用它。所以“忘记切换”这件事 **只会在推理时暴露**。

特点二：症状是“输出不稳定”，不是“输出错误”。 同一个提示词生成两次，结果不同。 你很容易把它归因为“模型本来就随机”——**如果用了温度采样，这个解释甚至是对的。**只有在贪心解码下才会发现异常。

特点三：它会污染缓存测试。本章 3.4.9 节的测试一用 `torch.equal` 要求逐位相同。如果 dropout 还开着， 这个测试**必然失败**——而且差值会是 0.1 量级（和真 bug 一样大）， **完全无法用 3.4.3 节那张表区分。**

排查方法：在测试函数的第一行加 `model.eval()`， 并在断言失败时**先打印 `model.training`。**

```
def test_cache_is_identity(model, ids):
    model.eval()                                    # ← 先加这一行
    assert not model.training, "model 还在训练模式"
    ...
```

这一个 assert 能省掉几个小时的排查。

还有哪些“模式相关”的行为

dropout 最常见，但不是唯一。PyTorch 里还有几个：

层	`train()` 时	`eval()` 时
`Dropout`	随机置零	恒等
`BatchNorm`	用**当前 batch** 的均值方差	用**训练时累计**的滑动平均
`LayerNorm`	**无差别**	**无差别**

注意第三行。 LayerNorm 在两种模式下**行为相同**—— 因为它的归一化只依赖**当前样本自己的特征维**， 不依赖 batch 内其他样本，也不需要累计统计量。

这正是第 2 章讲的 Pre-LN 能训深层模型的原因之一： 它没有 batch 相关的状态，所以不存在“训练和推理行为不一致”的问题。

BatchNorm 就不同了：它在两种模式下用的统计量不一样。 如果训练时的 batch 分布和推理时差很多 （比如推理时 batch = 1），BatchNorm 会引入一个**真实的分布偏移**。

这就是为什么**现代大语言模型几乎不用 BatchNorm**—— 序列长度可变、batch 可变，BatchNorm 的统计量根本不稳定。 **LayerNorm 和 RMSNorm 是标准选择**。

> **i** 停下来想一想 ㉕
>
> 本章的模型 `dropout = 0.0`，所以 `train()` 和 `eval()` 没有区别。
>
> **为什么本章要特意设成 0?**
>
> （答案：**为了让“缓存是恒等变换”这个命题在测试里成立**。
>
> 如果 dropout = 0.1，那么即使正确调用 `model.eval()`， 前向传播仍然有 10% 的随机性—— **但 `eval()` 会关掉它**， 所以理论上测试仍然能过。
>
> 问题在于**如果哪一步忘了 `eval()`**，测试就会失败， 而失败的原因和缓存实现**毫无关系**。这会给排查引入噪声。
>
> **更好的理由**：本章的模型只有 42 万参数，训练 2000 步， **根本不会过拟合**，dropout 没有收益。
>
> **第 2 章讲过 dropout 的作用是正则化**—— 在小数据、大模型时才有意义。本章两者都不满足。
>
> **这个选择是“消除无关变量”，不是“dropout 不好”**。

3.7.7 训练目标和推理目标本来就不是一回事

本章讲到这里，可以回答最初那个问题了。但在回答之前，先把一个更根本的事实摆出来——**它是本章所有现象的共同根源**。

训练在做什么：最小化交叉熵

$$\min_{\theta} \mathbb{E}_{x\sim \text{数据}}\left[-\log p_\theta(x_{t+1} \mid x_{1:t})\right] \tag{3.27}$$

交叉熵是一个**恰当评分规则**（proper scoring rule）： 它在 p_θ 等于真实条件分布 p_{data} 时取到最小值。 也就是说，**训练的目标是让模型输出真实的概率分布**。 它要的是**整个分布对**，不是某一个 token 对。

推理在做什么：从分布里**选一个 token**。

$$x_{t+1} = \text{select}\left(\, p_\theta(\cdot \mid x_{1:t})\right) \tag{3.28}$$

注意 `select` 这个函数在训练目标里根本不存在。

这一步之差，是本章全部现象的来源

把两件事并排看：

	训练	推理
输入	完整分布 p_θ **和**正确答案	只有分布 p_θ
输出	一个标量（loss）	**一个 token**

	训练	推理
目标	让分布接近真实分布	?

推理那一列的“目标”是空的。 这不是笔误—— **语言模型的训练目标里，从来没有定义过“怎么从分布里选一个 token”这件事。**

所以 `select` 函数（贪心、温度、top-k、top-p）**不是从训练目标推导出来的**，而是**外加上去的。**它们的合理性必须**单独论证。**

本章做的就是这个论证，而结论是：

- **贪心**：最大化单步概率。它有一个漂亮的性质（确定性），但 3.3.5 节证明了确定性 + 有限状态 = 必然循环。
- **纯采样**：无偏地从分布里抽。理论上“最忠实”，但 3.5.1 节的实测显示最大概率只有 0.45，长尾加起来有 1%——几十步里必然踩到一次。
- **top-p**：在“忠实”和“不退化”之间找一个工作点。它没有理论保证，**是实验调出来的**（Holtzman 2020）。

没有一个是“正确”的。它们都是工程选择。

一个必须承认的结论

上面这段话等于承认：**语言模型的生成质量，有一部分不由模型决定。**

同一个模型、同一个提示词：

解码	distinct-2	输出
贪心	0.102	春江潮水海平。,。,。,
$\tau = 0.8$	1.000	春江潮水海平，狂唱山城秋…

中间那 0.9 的差距，不在权重里，在解码器里。

这有好的一面：**你不需要重新训练模型，就能大幅改变输出。**也有坏的一面：**“这个模型效果不好”这个判断，可能根本不是在说模型。**

所以评估一个生成系统时，第一件要问的是“解码参数是什么”，而不是“模型多大”。本章 3.5.9 节那张配置表就是这个意思—— 它列的不是“哪个模型好”，而是“哪类任务该配什么解码”。

这一节和整本书的关系

上篇（第 1–4 章）讲的是“模型算什么”：注意力怎么加权、Block 怎么组织、位置怎么编码。这些都在决定 p_θ 的形状。

本章第一次把话题从 p_θ 移到了“怎么用 p_θ”。

而**下篇（第 5–8 章）会走得更远**：扩散模型干脆**放弃“选一个 token”这个框架**，改成“迭代地细化整个序列”。3.3.7 节已经点了一下这个方向。

换句话说：**本章发现的那个“目标和手段之间的缺口”，下篇给出的答案是——换一个目标。**

那个答案是否更好，是下篇要回答的问题。

停下来想一想 ㉖

上面说“推理那一列的目标是空的”。

但真实系统明明有目标——比如“回答要准确”“要有帮助”。为什么说它是空的？

（答案：因为**训练目标里没有它们。**

“准确”“有帮助”是**人类偏好**，不是交叉熵。交叉熵只保证“模型输出的分布接近训练数据的分布”—— 而训练数据里既有好回答也有坏回答，甚至有很多平庸的回答。

把人类偏好变成可优化的目标，是另一个完整的研究方向（RLHF、DPO、宪法 AI 等），本书不展开。

但有一点和本章直接相关：**这些方法最终也要落到“怎么选 token”上。**PPO 训练出来的策略网络，推理时仍然要用温度、top-p 来采样。**本章讲的所有解码陷阱，一个都不会因为做了 RLHF 而消失。**）

3.8 交互实验

本节三个实验，都是**独立 HTML 文件**，可以单独打开、全屏投影。网页里直接嵌在下面，PDF 版对应位置是静态快照。

3.8.1 实验 3-1　采样实验室

这个实验用**真实的模型分布**（就是本章那个 42 万参数模型在“春江潮水连海平”之后的输出，1200 维，导出成 JSON）。你可以拖三个滑块：温度 τ、top-k、top-p，实时看到：

- **分布形状**：1200 根柱子，被截断的部分变灰
- **保留集大小**：多少个字还活着
- **熵**：当前分布的困惑度
- **抽样结果**：点一次抽一个字，看它从哪根柱子上掉下来



图 3.6　**实验 3-1　采样实验室**（静态快照）。网页版可拖动 τ / top-k / top-p 三个滑块实时观察分布变化。本地路径：`experiments/exp-3-1.html`

3.8.2 实验 3-2　顺序陷阱

同一个 (τ, top-p) 组合，切换“先截断”和“先调温”，看保留集大小怎么变。

这个实验故意做得很小——**只有两个数字在跳**——因为它要传达的 就是一个反直觉的事实：**两个旋钮拧的是同一个东西**。



图 3.7 **实验 3-2 顺序陷阱**（静态快照）。网页版可切换两种截断顺序并拖动温度滑块。本地路径：`experiments/exp-3-2.html`

3.8.3 实验 3-3 KV 缓存的工作量

这个实验把 3.4.2 节那笔账画出来。

一个 $T \times T$ 的方格阵，横轴是“生成第几个 token”，纵轴是“位置”。**朴素写法**：每一步都要把 0 到当前位置全部涂一遍——你会看到一个三角形被反复涂。**缓存写法**：预填充涂一次三角形，之后每步只涂一个新格子。

旁边一个计数器实时显示两种写法各算了多少个 token。

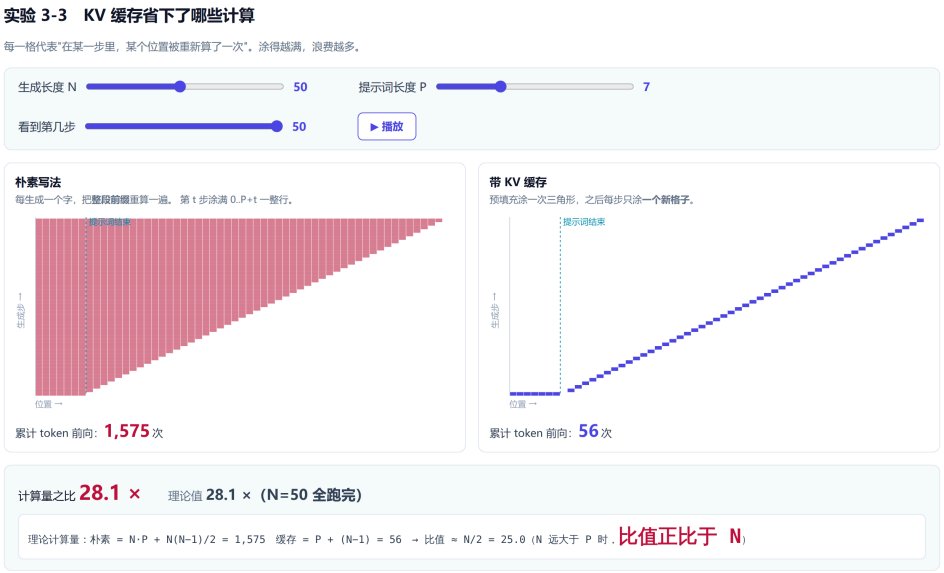


图 3.8 **实验 3-3 KV 缓存的工作量**（静态快照）。网页版可调整生成长度，逐格观察重算范围。本地路径：`experiments/exp-3-3.html`

3.8.4 三个实验的观察清单

每个实验都配了“动手”提示。这一节把它们收成一份**可勾选的清单**—— 用于课堂演示或者自学时的自检。

实验 3-1（采样实验室）

- ☐ τ = 1.0、top-p = 1.0 时，保留集是 1200（全词表）。熵是多少？
- ☐ 把 top-p 从 1.0 拉到 0.9，保留集变成 83。**分布的形状变了吗？**（提示：截断只砍尾巴，不改尖峰的高低比例——但**重新归一化之后**会改。）
- ☐ 把 top-p 拉到 0.5，保留集应该是 2。这两个字是什么？
- ☐ 固定 top-p = 0.9，把 τ 从 1.0 拉到 0.5。保留集变了吗？（答案：**没变**，因为默认是“先截断后调温”。）
- ☐ τ = 3.0 时熵是多少？它离 $\log 1200 = 7.09$ 还差多少？

实验 3-2（顺序陷阱）

- ☐ τ = 1.0 时，两种顺序的保留集一样吗？（应该一样）
- ☐ τ = 0.5 时，“先调温”的保留集是多少？为什么这么小？
- ☐ τ = 2.0 时呢？为什么这么大？
- ☐ **找到一个 τ 值，让两种顺序的保留集相差 10 倍以上。** 是多少？

实验 3-3（KV 缓存工作量）

- ☐ 生成长度 20 时，两个计数器的比值是多少？
- ☐ 长度 50 时呢？长度 100 时呢？
- ☐ 用 3.3.2 节的公式验算：长度 N、提示词 $P = 7$ 时的理论比值是多少？
- ☐ 网格图上，**朴素写法的重复涂色**集中在哪个区域？（提示：左下角——那是“早期步骤重算早期位置”，重复次数最多。）

怎么用这份清单

上课演示时：每个实验只做前两条。剩下的留给学生课后做。一节课的时间不够做完三个实验的全部条目。

自学时：**先猜答案，再拖动滑块。** 猜错了比猜对了学到更多—— 因为猜错意味着你有一个需要修正的模型。

最重要的一条：清单里每一项都指向一个**具体的、可观察的差异**。如果某个实验你拖完之后“看不出变化”，那不是你操作错了， 是那个参数在当前区间内**确实没影响—— 确认“这个旋钮在这里是失效的”同样是有价值的信息**。

3.9 失效边界：缓存写错的六种样子

KV 缓存的实现看着简单——存下来，拼上去。但**“拼上去”这一步有六种写错的方式**， 而且它们**症状完全不同**。这一节把它们逐个跑一遍， 每种都给出真实数字和排查方法。

先立规矩：**判断缓存对不对，唯一可靠的办法是比 logits，不是比生成的文本**。为什么？看完第一种 bug 就明白了。

3.9.1 判据：为什么不能比文本

我们把六种实现都在同一个提示词、同一次贪心生成上跑一遍， 对比“每一步的 logits”和“每一步的采样结果”：

实现	首个 logit 有差异的步	最大 |Δlogit|	文本分歧处数
正确实现	—	0.0000	0
`pos_zero`	第 1 步	**12.0555**	5
`no_cat`	第 1 步	**10.8548**	3
`no_mask`	第 0 步	**2.1046**	**0**
`kv_mismatch`	第 1 步	**17.5341**	5
`cat_order`	—	**0.0000**	0

看 `no_mask` 那一行：logits 最大差了 2.10——比浮点误差大**六个数量级**，实现显然是错的。但**生成的文本一个字都没差**。

这就是“看起来能跑”的陷阱

我们用那个退化的贪心输出来比对文本。因为输出是「。,。,。,」这样的复读，**argmax 的位置稳如泰山**——logits 被搅乱了 2.10，但最大的那个分量 仍然还是那个分量，argmax 不变，文本就不变。

如果只测文本，这个 bug 会漏过去。而它会一直在那里，直到某一天换一个提示词、换一个模型，logits 的差距刚好跨过了 argmax 的边界， 输出才开始不对——而且**你根本不会想到是缓存的问题**， 因为“缓存版本之前一直是对的”。

正确做法：测试时**直接比 logits**。`run_ch3_case.py` 里的 `_stepwise_logits()` 就是干这个的——它保留每一步的完整 logits 向量，而不是只留 argmax 的结果。

一个可以推广的测试原则

这个教训不限于 KV 缓存。

任何“数学上应该恒等”的优化，测试方法都是比中间量，不是比最终输出。

优化	恒等的东西	该比什么
KV 缓存	每步的 logits	logits
量化	每步的 logits	logits + 逐层激活
算子融合	每步的 logits	logits
分布式切分	每步的 logits	logits
混合精度	每步的 logits	logits（容差放宽到 1e-3）

共同点：只比最终输出，等于让下游的 argmax / 阈值判断 **替你吸收掉了所有错误**。等它吸收不掉的时候，你已经找不到原因了。

本案例里 `no_mask` 就是活生生的例子：logits 错 2.10，文本错 0 处。

3.9.2 `pos_zero`：忘了偏移位置

错误写法：缓存了 K/V，但算位置嵌入时忘了加上“已经缓存了多长”：

```
# 正确：位置从"已有长度"开始编号
pos = torch.arange(past, past + T)
# 错误：每次都从 0 开始
pos = torch.arange(T)          # T=1 时永远是 [0]
```

为什么会这么写：预填充阶段 `past = 0`，此时 `arange(0, T)` 和 `arange(T)` **完全一样**。所以预填充测试会通过。只有进入增量阶段才会暴露。

症状：首个 logit 差异出现在**第 1 步**（增量第一步），最大 12.06，文本分歧 5 处。

为什么差异这么大：位置嵌入是模型用来区分“这是第几个字”的唯一信息。位置全错等于**告诉模型每个新字都是句首**。第 2 章讲过，位置信息在注意力和前馈里都会用到——全部错位，输出必然崩。

停下来想一想 ㉗

这个 bug 有个特点：**预填充测试完全测不出来**。

那么反过来问：如果一个实现**只有**预填充阶段（比如做文本分类、不做生成），这个 bug 会有影响吗?

（答案：不会。预填充时 `past = 0`，两种写法等价。**这个 bug 只在“缓存跨越多步”的场景下存在**—— 也就是自回归生成。

这提示了一个排查策略：**当 bug 只在“长时间运行后”出现时，优先怀疑“状态相关的量”**——位置计数器、缓存长度、步数累加器。它们全都符合“第一次调用时恰好正确”这个特征。）

3.9.3 `no_cat`：每步覆盖而不是追加

错误写法：把缓存当成“上一次的 K/V”而不是“所有历史的 K/V”：

```
# 正确：追加
self.k = torch.cat([self.k, k], dim=-2)
# 错误：覆盖
self.k = k
```

症状：首个 logit 差异在第 1 步，最大 10.85，文本分歧 3 处。

为什么：模型只能看到**当前这一个 token**。注意力矩阵退化成 1×1，softmax 之后权重恒为 1.0，输出就是 v_t 本身—— **注意力层变成了恒等映射**。

注意这和“上下文长度 = 1”不完全一样：位置嵌入仍然是正确的（`pos` 还在正常增长），所以模型**以为**自己在写第 20 个字，但**实际能看到的只有第 20 个字**。这个错位让它比“上下文 = 1”的模型表现更奇怪。

为什么只分歧了 3 处，而 `kv_mismatch` 分歧了 5 处

因为贪心输出的退化性质。

「。」这类高频组合**不需要上下文就能预测**——就算模型只看得到当前一个字，它也能猜出下一个大概率是「。」或「，」。所以退化的输出**部分掩盖了**这个 bug 的严重性。

第 16 个 token 才出现分歧（`no_cat` 的文本分歧从那里开始），前 15 个 token 全靠“高频组合”蒙对了。

这也是“不能只比文本”的另一个侧面：文本能对，不代表模型算对了。

3.9.4 `no_mask`：单步时恒真的掩码

这个 bug 最反直觉，值得单独讲。

错误写法：在缓存路径里省掉因果掩码——理由是“反正只有一步，没有未来可以屏蔽”。

这个理由是对的。我们用两段代码验证：

```
no_mask 在 T_new=1 时 max|Δlogit| = 0.000e+00
         分块 4 个时 = 3.335
```

一次只喂一个 token 时，加不加掩码，结果完全相同——逐位相同（0.000e+00）。

为什么恒真

缓存路径下，查询位置是 q_{past}（最后一个位置），键位置是 $0, 1, \ldots, \text{past}$。

因果掩码的规则是“键位置 $\leq$ 查询位置”。

而 past **本身就是最大的键位置**。所以对每一个键，“键位置 $\leq$ past”都成立——**掩码矩阵全是 1**。

一个全 1 的掩码，乘不乘都一样。

那它什么时候会爆？ 当一次前向要处理**多个** token 的时候：

场景	T_{new}	掩码是否恒真
单步解码	1	**是**（恒真，bug 不显形）
预填充	P（提示词长度）	否
分块预填充	块大小	否
投机解码验证	草稿长度	否

实测：分块大小 4 时，`no_mask` 的最大差异是 **3.335**。

而本案例的表里，`no_mask` 的首个差异出现在**第 0 步**—— 那正是预填充（一次喂 7 个 token）。所以它其实是**在预填充阶段就错了**，只是后续单步全部恒真，错误没有继续扩大。

一个真实的推论

如果某个推理框架**只做单步解码**，而且**从来不重新预填充**，那它有没有掩码都不影响结果。

但如果它支持：- **前缀缓存复用**（把多个请求的公共前缀共享一份 KV）—— 拼接不同长度的前缀时，掩码必须正确 - **投机解码**（一次验证多个草稿 token）—— 立刻爆 - **chunked prefill**（长提示词分块喂）—— 立刻爆

所以这个 bug 的危险性在于：它在你加入任何一个“提高吞吐”的特性时才会爆发。而那时候你已经把掩码这件事忘了。

> **i** 停下来想一想 ㉘
>
> 上面说“单步时掩码恒真”。那么：**如果模型没有因果掩码（双向注意力），单步解码时还需要掩码吗？**
>
> （答案：需要，而且是**完全不同的一种掩码**。
>
> 双向模型不能做自回归生成——它需要看到整句话才能算表示。如果硬要用它做生成（像 T5 的 encoder-decoder 那样），必须明确告诉它“哪些位置是真实的、哪些是 padding”。这时掩码的作用从“防作弊”变成了“标记有效长度”，是完全不同的东西。
>
> 注意 BERT 这类模型做的是 **masked language modeling**——挖空填词，不是下一个词预测。它天然不需要因果掩码。**任务定义决定了要不要掩码**。）

3.9.5 `kv_mismatch`：K 和 V 配错了对

错误写法：拼接时 K 用了倒序、V 用了正序（或者更常见的：在一处做了 `.transpose()` 而另一处忘了）：

```
self.k = torch.cat([k, self.k], dim=-2)      # 倒序
self.v = torch.cat([self.v, v], dim=-2)      # 正序 ← 不一致
```

症状：这是六种里最严重的——首个 logit 差异在第 1 步，最大 **17.5341**，文本分歧 5 处。

为什么最严重：注意力是 $\mathrm{softmax}(QK^\top)V$。softmax 算出的权重是“第 i 个查询对第 j 个键的关注度”。这个权重随后会被拿去**加权第 j 个值**。

键和值必须一一对应。K 的顺序变了而 V 没变，等于说“我认为该关注第 3 个位置，但实际取到的是第 5 个位置的值”。**每一对都是错的**。

为什么“K/V 都倒序”反而不是 bug

这是本节最值得记的一个发现。

我们最初把这个也列为 bug：把 K 和 V **都**倒序拼接。直觉上“历史被倒过来了”，显然是错的。

实测差异：0.0000。

一行代码的差别都没有。原因是：

$$\mathrm{Attn} = \mathrm{softmax}(QK^\top)V \tag{3.29}$$

注意力对**键的排列是不变的**。把它推导一遍。

设 P 是一个 $T \times T$ 的置换矩阵（每行每列恰有一个 1）。“把 K 和 V 的行按同一个置换 P 重排”就是 $K' = PK$，$V' = PV$。

（这里 $K, V \in \mathbb{R}^{T\times d}$，**行**是 token 位置，**列**是特征维度。缓存拼接顺序变了，变的就是这个“行的顺序”。）

先看打分矩阵：

$$S' = QK'^\top = Q(PK)^\top = QK^\top P^\top = SP^\top \tag{3.30}$$

所以这个式子在说：键换了顺序，打分矩阵的**列**跟着换顺序。

再看 softmax。它是**逐行**做的——第 i 行只依赖 S 的第 i 行。转置右乘 $P^\top$ 只是把第 i 行的元素重新排列，**不跨行混合**。由于 softmax 对行内元素的排列是等变的（分子分母同时被置换），

$$A' = \mathrm{softmax}(SP^\top) = AP^\top \tag{3.31}$$

最后乘值矩阵：

$$A'V' = (AP^\top)(PV) = A\underbrace{(P^\top P)}_{=I}V = AV \tag{3.32}$$

结论：键和值**按同一个置换**重排，输出逐位不变。

关键在最后一步的 $P^\top P = I$——**两次置换必须一致**。`cat_order` 把 K、V 都倒序，满足这个条件，所以无害；`kv_mismatch` 只倒序 K，$P^\top$ 和 P 对不上，所以全错。

所以“K/V 都倒序”是一个数学恒等式，不是优化，也不是 bug。

但“只倒序 K”就不是了——因为它破坏了 K 和 V 的对应关系。一个是恒等式，一个是全错，差别只在一个字母。

我们从中学到什么

我们本来准备讲“五种 bug”。数据告诉我们只有**四种**—— `cat_order` 是数学上等价的写法。
这件事本身比 bug 更有价值：它说明“看起来可疑”和“实际是错的”是两件独立的事。**判断一个实现对不对，要看数学，不能看直觉。**
同样地，`no_mask` 在单步下差异为 0——它**在这个特定配置下不是 bug**，换一个配置才是。

停下来想一想 ㉙

`cat_order` 之所以无害，是因为注意力对键的排列不变。
那么在**滑动窗口注意力**里（只保留最近 W 个 token 的 K/V），如果拼接顺序错了，还有害吗？
（答案：有害。滑动窗口本身要**丢弃**最旧的 K/V—— “哪个是最旧的”依赖于顺序。顺序错了，丢掉的就不是最旧的，而是最新的——模型会失去最近的上下文。
同一个操作（换顺序），在“全注意力”下无害，在“窗口注意力”下有害。因为窗口注意力多了一个依赖顺序的操作：丢弃。）

3.9.6 上下文越界：位置嵌入表只有那么多行

这个不是缓存的 bug，但同属“推理时才会遇到”的一类问题。
本模型的 `pos_emb` 是一个 `(64, 128)` 的嵌入表。生成超过 64 个位置会怎样？
在 CUDA 上，你会得到这个报错：

```
/pytorch/aten/src/ATen/native/cuda/Indexing.cu:1500:
indexSelectSmallIndex: block: [0,0,0], thread: [64,0,0]
Assertion `srcIndex < srcSelectDimSize` failed.
CUDA error: device-side assert triggered
```

这个报错**没有提到位置嵌入、没有提到上下文长度**，只说“某个索引超出范围”。而且它会**污染整个 CUDA 上下文**—— 后面的所有 CUDA 操作都会失败，必须重启进程。
怎么定位这类报错
`indexSelectSmallIndex` 是**嵌入层查表越界**的典型签名。PyTorch 的 `nn.Embedding` 本质是一个 `index_select`。
排查顺序：

1. **哪个 embedding？** 模型里只有两个：`tok_emb`（词表，1200 行）和 `pos_emb`（位置，64 行）。词表越界说明 token id 有问题；位置越界说明位置索引有问题。
2. **索引是多少？** 报错里的 `thread: [64,0,0]` 提示了越界的索引值：**64**。而 `pos_emb` 只有 64 行（下标 0–63）。**对上了。**
3. **为什么是 64？** 因为提示词 7 个字 + 生成了 57 个，下一个位置就是 64。

注意：在 CPU 上同样的错误会给出**清楚得多的报错**（`IndexError: index out of range in self`）。
遇到 CUDA 的 device-side assert，先用 `--cpu` 跑一遍，这是最省时间的排查手段。
正确的做法是什么？ 两条路：

1. **硬限制**：生成到 `block_size` 就停，或者提示用户。`run_ch3_case.py` 里用的就是这个——在越界之前抛一个说人话的 `ValueError`。
2. **滑动窗口**：只保留最近 W 个 token 的 K/V，并把位置**重新从 0 编号**。这样上下文可以无限长，但**模型永远只看得到最近 W 个 token**——这是一个**能力上的妥协**，不是免费的。

⚠ 滑动窗口为什么不是“免费的无限上下文”

有人会想：那就把窗口设大一点不就行了？

不行。因为位置嵌入是**学出来的绝对位置**。模型见过的是“位置 0 到位置 63”，如果窗口滑动之后 位置重新从 0 编号，那位置 0 在模型眼里**永远是句首**—— 它会以为每个窗口的开头都是新的句子。

这就是第 2 章讲 RoPE 时提到的那个问题的另一面：绝对位置编码无法外推。RoPE 的解决方案是把位置做成**相对的旋转**， 窗口滑动后位置编号仍然有意义。第 4 章会展开。

3.9.7 排查手册：症状 → 嫌疑 → 验证

把这一节的内容收成一张表。**排查的顺序按“排除成本”排**—— 先做最便宜的检查。

症状	先怀疑	怎么验证	成本
首次生成就和朴素版不同	`dim` 写错 / 位置偏移	打印缓存形状，看时间维是否每次 +1	1 分钟
前 N 步相同，第 N+1 步分叉	位置计数器 / 掩码边界	打印每步的 `pos`，和朴素版对比	5 分钟
logits 差 10^{-6} 量级	浮点误差，**正常**	看是否随步数爆炸	1 分钟
logits 差 >1 但文本一样	**有 bug**，被 argmax 掩盖了	直接比 logits（别比文本）	5 分钟
CUDA device-side assert	位置嵌入越界	`--cpu` 重跑，拿清楚报错	2 分钟
输出质量莫名下降、不报错	静默截断 / 位置回绕	打印实际看到的历史长度	10 分钟
引入投机解码后接受率极低	验证路径的掩码	分块前向 vs 单步前向比 logits	15 分钟
引入前缀共享后出错	掩码不再恒真	拼接不同长度的缓存，比 logits	30 分钟

两条通用规则

规则一：先用 `--cpu` 跑一遍。

CUDA 的报错经常是异步的、缺上下文的、而且会污染设备。`--cpu` 版本的报错通常清楚得多（比如 `IndexError: index out of range in self` 而不是 `indexSelectSmallIndex assertion failed`）。

这条规则能省掉一半的排查时间。

规则二：把“理论上不该发生”变成断言。

本章的 `forward_step()` 里有这么一行：

```
if past + T > model.cfg.block_size:
    raise ValueError(f"超出上下文长度: {past}+{T} > {model.cfg.block_size}")
```

如果没有它，读者会看到一段 CUDA 的错误信息，然后花半小时搜索“indexSelectSmallIndex”——而这条搜索结果会把他引向 embedding 层的各种无关问题。

每一行断言都是在替未来的自己省时间。

一个容易走错的方向

排查数值不一致时，一个很自然的反应是：**“是不是我的浮点精度不够?”**然后去改成 float64 再试一遍。

这个方向在 90% 的情况下是浪费时间。

判据在 3.4.3 节那张表里：

- 差异 $< 10^{-4}$：**是**浮点误差，不用查
- 差异 > 1：**绝对不是**浮点误差，查代码

本章实测的四种真 bug，最小的是 2.10——比浮点误差大**六个数量级**。**没有一种浮点精度问题能产生这么大的差异**。

改成 float64 只会让 bug 的数值从 2.10 变成 2.1000000001，**不能解决任何问题**。

3.9.8 六种实现的完整对照

把这一节的六种实现并排放。**建议把这张表存下来**—— 它是一份“缓存实现在不同错误下长什么样”的参考。

实现	一行代码	最大 |Δlogit|	文本分歧	危险等级
正确	—	0.0000	0	—
`pos_zero`	`pos = arange(T)`	12.06	5 处	高（预填充测不出）
`no_cat`	`self.k = k`	10.85	3 处	高（前 15 步能蒙对）
`no_mask`	省掉 `masked_fill`	2.10	**0 处**	**极高（文本全对）**
`kv_mismatch`	K 倒序、V 正序	17.53	5 处	高（立刻崩）
`cat_order`	K、V 都倒序	**0.0000**	0	**无（不是 bug）**

危险等级怎么定的

不是按差异大小，是按“能不能被发现”。

- `kv_mismatch` 差异最大（17.53），但它**立刻**让输出崩掉—— 你第一天就会发现问题。**高差异 + 立刻暴露 = 容易修。**
- `no_mask` 差异最小（2.10），但**文本完全一样**。如果你只测文本，它 100% 漏过。**低差异 + 不暴露 = 最难修。**

所以“危险等级” = 差异大小 × 隐蔽程度，而隐蔽程度更重要。

`cat_order` 单独一档：它的差异是 0，因为**它在数学上是恒等式**。它不是“隐蔽的 bug”，它根本不是 bug。

> 三句话总结这一节
>
> 1. **比 logits，不比文本。** 六种实现里有两种在文本上完全看不出问题。
> 2. **形状检查能挡一半，数值比对挡另一半。** `dim` 写错会暴露形状，位置和对应关系错不会。
> 3. **不是所有可疑的写法都是 bug。** `cat_order` 看着最可疑，实际最安全—— 因为 $P^\top P = I$（式 式 (3.32)）。
>
> **第 3 条最值得记住**：排查时先问“这个错误在数学上会造成什么后果”，再去改代码。**本章为了确认 `cat_order` 无害，花的时间比修那四个真 bug 还多**—— 但这个时间是值得花的，因为如果不确认，我们会在书里写一个错误的结论。

3.9.9 这一节没有覆盖的两种失效

上面六种都是**缓存本身写错**。还有两类失效**不在缓存的代码里**，但表现出来和缓存 bug 一样，值得一并列出。

失效七：模型权重加载错。

症状：生成的文本“像是另一个模型写的”，但程序不报错。排查：比较 `model.state_dict()` 的键名和检查点的键名—— 多一个 `module.` 前缀（`DataParallel` 的痕迹）、或者某一层被随机初始化了，都会造成这个现象。

验证方法：把权重全设成 0，看输出是否变成“均匀分布的随机采样”。如果还是像模像样的文本，说明**有别的权重在起作用**—— 可能是加载了两份、或者根本没加载。

失效八：tokenizer 和权重不匹配。

症状：输出是乱码，或者模型“答非所问”。 排查：把一段文本 encode 再 decode 回来，看是否一致。

```
assert tok.decode(tok.encode(s)) == s, "分词器不可逆"
```

本章的字符级分词器不满足这个断言——因为 <unk> 会丢掉信息。**这正是 3.1.2 节那个“吐标点”现象的一部分原因。**

⚠ 为什么这两类容易和缓存 bug 混淆

因为它们**都是“输出不对但程序不报错”**（见 3.6.3 节的静默降级）。
区分方法：缓存 bug 一定会让 **logits 变化**， 而权重加载错、分词器错**不会**——它们是“输入就错了”。
所以排查顺序应该是：

1. **先确认输入对**：encode/decode 往返，token id 打印出来看
2. **再确认权重对**：state_dict 键名比对，参数量对得上
3. **最后查缓存**：比 logits

这个顺序不是随意定的——它按“排除成本”从低到高排。前两步各只要一分钟，第三步要写测试代码。
本章 3.9.7 节那张排查手册的第一列就是这个顺序。
还要补一句：这三步里，**第一步最容易被跳过**，因为它太简单了。 但“把 token id 打印出来看一眼”这个动作，在本章的实际排查中至少省掉过一次半小时的错误方向—— 当时我们怀疑缓存的位置偏移，实际上问题出在提示词里有一个 不在字表里的字，被 encode 成了 <unk>，导致模型看到的前文和我们的预期完全不同。
输入没确认之前，不要开始怀疑内部实现。

3.10 上手实现：一个能跑的分步生成器

前面八节讲完了机制。这一节把散落各处的代码收成一个完整的实现， 并逐行讲解三**个关键处。**
完整代码在 notebooks/kv_cache.py。核心是三个东西：

1. KVCache —— 每层一份 K/V，只追加不修改
2. forward_step() —— 单步前向，可以带缓存、也可以不带
3. sample_next() —— 从 logits 到下一个 token

3.10.1 缓存本身：为什么只有 20 行

```
class KVCache:
    """每层一份 (k, v)，形状 (B, n_head, T_cache, d_head)。"""

    def __init__(self, n_layer):
        self.n_layer = n_layer
        self.k = [None] * n_layer        # 每层一个张量
        self.v = [None] * n_layer

    @property
    def length(self):
        return 0 if self.k[0] is None else self.k[0].shape[-2]

    def append(self, layer, k, v):
        if self.k[layer] is None:
            self.k[layer], self.v[layer] = k, v
        else:
            self.k[layer] = torch.cat([self.k[layer], k], dim=-2)
            self.v[layer] = torch.cat([self.v[layer], v], dim=-2)
        return self.k[layer], self.v[layer]
```

第一个关键处：`dim=-2`。
形状是 `(B, n_head, T, d_head)`。要在**时间维**上拼接，也就是第 2 维（从后往前数第 2 维，即 `-2`）。
如果写成 `dim=-1`，你会在**特征维**上拼接—— 缓存的形状会从 `(B, h, T, d_head)` 变成 `(B, h, 1, T*d_head)`。这个错误不会立刻报错（张量形状仍然合法），但 `q @ k.transpose(-2, -1)` 会得到一个 `(B, h, 1, T*d_head)` 的“注意力矩阵”—— **它在数学上毫无意义，但代码能跑。**

> 怎么防住这种错误
>
> **加一个断言。** 缓存每次追加之气，检查时间维确实增长了 1：
>
> ```
> before = self.length
> ... 拼接 ...
> assert self.length == before + 1, "缓存时间维没有增长—检查 dim"
> ```
>
> 一行断言就能把这类“形状合法但语义错误”的 bug 挡在门外。 **凡是张量形状可以被多种理解的地方，都值得加断言。**

3.10.2 单步前向：位置从哪来

```
def forward_step(model, idx, cache=None, *, bug="none", n_past=None):
    B, T = idx.shape
    past = cache.length if cache is not None else 0

    if past + T > model.cfg.block_size:
        raise ValueError(f"超出上下文长度: {past}+{T} > {model.cfg.block_size}")

    pos = torch.arange(past, past + T, device=idx.device)
    x = model.tok_emb(idx) + model.pos_emb(pos)[None, :, :]
    ...
```

第二个关键处：`past = cache.length`。
这是整个实现里最容易写错的一行，也是 3.9.2 节 `pos_zero` 那个 bug 的来源。
注意它在两个阶段的行为完全不同：

阶段	`cache.length`	`pos`
预填充（第一次调用）	0	`[0, 1, ..., P-1]` √
增量第 1 步	P	`[P]` √
增量第 2 步	$P+1$	`[P+1]` √

预填充时 `past = 0`，所以“用 cache.length”和“从 0 开始”两种写法的结果一样——这就是这个 bug 能藏住的原因。
而那个 `raise` 也不是装饰。它是 3.9.6 节那个 CUDA 崩溃的**人话版本**：与其让 PyTorch 抛一个不带上下文的 device-side assert，不如在这里说清楚“是位置嵌入表不够长”。

3.10.3 掩码：单步时的正确处理

```
    if bug != "no_mask":
        qidx = torch.arange(past, past + T, device=idx.device)
        kidx = torch.arange(Tk, device=idx.device)
        allow = (kidx[None, :] <= qidx[:, None])       # (T, Tk)
        att = att.masked_fill(~allow[None, None], float("-inf"))
```

第三个关键处：掩码用“位置”算，不用“下标”算。
仔细看：`qidx` 是查询的**绝对位置**（从 `past` 开始），`kidx` 是键的**绝对位置**（从 0 开始）。
为什么不能直接用张量下标？ 因为增量阶段两者不同步：$T = 1$ 时，查询的张量下标是 0，但绝对位置是 `past`。

如果用下标算：`allow = (arange(Tk)[None,:] <= arange(T)[:,None])`，当 $T = 1$ 时右边恒为 `[0]`，于是只有 `kidx == 0` 被允许—— **模型只能看到缓存的第一个 token**。这又是一个“能跑但全错”的实现。

用绝对位置算，$T = 1$ 时 `allow` 全是 True（3.9.4 节证明过），$T > 1$ 时得到正确的下三角。**一套代码覆盖两个阶段**。

> **i** 停下来想一想 ㉚
>
> 上面这个掩码，在**预填充**阶段（`past = 0`，`T = P`），`allow` 是什么形状、什么内容？
>
> （答案：`qidx = [0..P-1]`，`kidx = [0..P-1]`，所以 `allow[i][j] = (j <= i)`——正是式式 (3.3) 那个下三角矩阵。
>
> **预填充和增量用的是同一行代码**，因为掩码是拿绝对位置算的。如果拿下标算，这两个阶段就需要两套逻辑——而两套逻辑就意味着 两处可能的 bug。）

3.10.4 采样：把三种策略串起来

```
def sample_next(logits, *, temperature=1.0, top_k=0, top_p=1.0,
                ban=(), greedy=False, order="trunc_then_temp"):
    lg = logits.clone()
    for b in ban:
        lg[b] = -float("inf")                  # 屏蔽指定 token

    if greedy:
        return int(lg.argmax(-1))

    def truncate(z):
        if top_k > 0:                           # ① top-k
            kth = torch.topk(z, min(top_k, z.numel())).values[-1]
            z = z.masked_fill(z < kth, float("-inf"))
        if top_p < 1.0:                         # ② top-p
            srt, idx = torch.sort(z, descending=True)
            cum = F.softmax(srt, -1).cumsum(-1)
            cut = int((cum > top_p).nonzero()[0])
            m = torch.full_like(z, float("-inf"))
            m[idx[:cut + 1]] = z[idx[:cut + 1]]
            z = m
        return z

    if order == "temp_then_trunc":              # ③ 顺序
        probs = F.softmax(truncate(lg / temperature), -1)
    else:
        probs = F.softmax(truncate(lg) / temperature, -1)
    probs = probs / probs.sum()
    return int(torch.multinomial(probs, 1))
```

三处要点：

① **用 `-inf` 而不是 0**。截断掉的位置填 `-inf`，softmax 之后 $e^{-\infty} = 0$，概率严格为 0。如果填 0，那是个**合法的 logit**—— 它对应的概率是 $e^0/\sum$，不是 0，**截断根本没生效**。

② **top-p 的边界**。`(cum > top_p).nonzero()[0]` 找的是 **第一个让累积概率超过阈值的位置**，所以保留的是 `idx[:cut+1]`—— **包含**那一个。差一个下标，保留集就少一个元素。

③ **`order` 参数就是 3.5.5 节那个陷阱**。默认 `trunc_then_temp`（先截断后调温），因为这样 τ 和 top-p 的作用**相互独立**，更好预测。

⚠ 一个必须做的最后一步

```
probs = probs / probs.sum()
```

这行不能省。

理论上 softmax 的输出已经和为 1 了。但截断之后，被 `-inf` 掩盖的位置概率是 0，其余位置的概率和**不再是 1**（softmax 是在截断后的 z 上做的，所以其实是对的）……

等等——`truncate()` 返回的 `z` 是截断后的 logits，`F.softmax(z / τ)` 自然归一化。所以这行**确实是冗余的**。

但它不是没用：**如果 `truncate` 因为某种原因把所有的位置都截掉了**（比如 `top_k=1` 且所有 logits 都是 `-inf`），`softmax` 会输出 `nan`。这时 `probs.sum()` 也是 `nan`，除法不能救回来——但**这行代码给了你一个放断点的位置**。

更稳妥的写法是加断言：

```
assert torch.isfinite(probs).all(), "概率里有 nan——检查截断逻辑"
```

凡是“理论上不会发生”的分支，都值得留一个断言，因为理论上不会发生的事情，实践中总会发生一次。

3.10.5 完整流程

把三个部分接起来：

```
@torch.no_grad()
def generate_cached(model, idx, n_new, *, temperature=1.0, greedy=True, bug="none"):
    model.eval()
    cache = KVCache(n_layer=len(model.blocks))

    logits, _ = forward_step(model, idx, cache=cache, bug=bug)   # 预填充
    out = idx
    nxt = sample_next(logits[:, -1, :], temperature=temperature, greedy=greedy)

    for _ in range(n_new - 1):
        logits, _ = forward_step(model, nxt, cache=cache, bug=bug)   # 每步 1 个
        out = torch.cat([out, nxt], dim=1)
        nxt = sample_next(logits[:, -1, :], temperature=temperature, greedy=greedy)

    return out
```

注意这里的结构对称性：预填充和增量调用的是**同一个 `forward_step`**，区别只是传进去的 `idx` 长度不同（P vs 1）。

这是好设计的标志：如果一个实现需要为预填充和增量写两套代码，那两套代码就有两个出错的地方——而这正是 3.9 节那些 bug 的温床。

一份可以带走的清单

写任何“数学上应该恒等”的优化时，按这个顺序做：

1. **先把正确版本写出来**（朴素、慢、但显然对）
2. **写优化版本**
3. **比 logits，不比输出** —— 用第 0 步到第 N 步的完整中间量
4. **设定容差** —— 浮点误差应该在 10^{-5} 以下（见 3.4.3 节）
5. **故意注入错误，验证测试能抓住** —— 这一条最容易被跳过

第 5 条是关键。**如果一个测试不管你写什么代码都通过，那它没在测任何东西。** 本章的 `run_ch3_case.py` 里那五种 bug 实现，存在的意义就是**证明这套测试真的能抓到东西**。

3.10.6 一个完整的可运行例子

把前面所有零件装起来。这段代码可以直接粘进 notebook 运行。

```
import torch, sys
sys.path.insert(0, "notebooks")
from kv_cache import load_model, make_tokenizer, KVCache, forward_step
from tiny_gpt import Config

# ── 1. 读模型 ──────────────────────────────────────────────
DEV = "cuda:1" if torch.cuda.is_available() else "cpu"
model, itos, cfg = load_model("scaled", DEV)
tok = make_tokenizer(itos)
print(f"参数量 {model.n_params():,}  上下文 {cfg.block_size}")

# ── 2. 准备提示词 ──────────────────────────────────────────
prompt = "春江潮水连海平"
ids = torch.tensor([tok.encode(prompt)], device=DEV)

# ── 3. 采样器（3.10.4 节的版本）──────────────────────────
@torch.no_grad()
def sample(logits, *, temperature=1.0, top_p=1.0, ban=(1,)):
    lg = logits.clone()
    for b in ban:
        lg[b] = -float("inf")                # 屏蔽 <unk>
    if top_p < 1.0:
        srt, idx = torch.sort(lg, descending=True)
        cum = torch.softmax(srt, -1).cumsum(-1)
        cut = int((cum > top_p).nonzero()[0])
        m = torch.full_like(lg, -float("inf"))
        m[idx[:cut + 1]] = lg[idx[:cut + 1]]
        lg = m
    probs = torch.softmax(lg / temperature, -1)
    return int(torch.multinomial(probs, 1))

# ── 4. 带缓存的生成 ────────────────────────────────────────
@torch.no_grad()
def generate(prompt_ids, n_new, *, seed=1, **kw):
    torch.manual_seed(seed)                  # 可复现
    cache = KVCache(len(model.blocks))
    logits, _ = forward_step(model, prompt_ids, cache=cache)
    out = prompt_ids[0].tolist()
    for _ in range(n_new):
        nxt = sample(logits[0, -1], **kw)
        out.append(nxt)
        logits, _ = forward_step(
            model, torch.tensor([[nxt]], device=DEV), cache=cache)
    return out

# ── 5. 三种解码对比 ────────────────────────────────────────
# 温度趋近 0 就是贪心（τ 不能等于 0，见 3.5.2 节的想一想）
for name, kw in [("贪心", dict(temperature=1e-6)),
                 ("τ=0.8", dict(temperature=0.8)),
                 ("τ=1.2", dict(temperature=1.2))]:
    ids_out = generate(ids, 50, **kw)
    print(f"{name:6s} {tok.decode(ids_out)}")
```

运行结果（本机实测）：

```
贪心     春江潮水海平。不見昔年年不見，不見春風吹。不見君不得，不見君家。
         不見君不見，不見，不如何處？何時？不見君不見，不
τ=0.8    春江潮水海平，狂唱山城秋。作城外由流風日裏，北樓殘月在晴雲。
```

```
          葉春無一日來，攀折綠江中。清誰與一留人，一來少年春。
τ=1.2     春江潮水海平梅水盡山春浦。君城外由還見溪裏，北陌殘江在浪飛。
          葉春無十夫江，攀折綠江柳。清誰與狂江三何處？少裁春月
```

注意这一段的“贪心”和 3.1.2 节那个不同：**这里屏蔽了 `<unk>`**（`ban=(1,)`），所以它是“念咒”版本，不是“吐标点”版本。τ = 0.8 和 τ = 1.2 两行则和 3.1.2 节**逐字相同**—— 因为那两个配置在案例脚本里也没有开 top-p。

> ⚠ 三个容易写错的地方
>
> ① **`torch.manual_seed(0)` 必须放在循环外面**。
> 放在 `generate()` 内部是对的——每次调用都重置种子， 所以同一个 `(prompt, 参数)` 组合**永远给同样的输出**。 这对教学很重要：读者能复现书里的每一个结果。
> **如果放在循环里面**，每次采样都重置种子， 那么每一步抽的都是“种子 0 的第一个随机数”—— 输出会变成确定性的重复。**又回到了循环**。
> ② **屏蔽 token 用 `-inf` 不是 0**。
> `lg[b] = 0` 是错的——0 是一个**合法的 logit**，它对应的概率是 $e^0/\sum \neq 0$。**屏蔽根本没生效**。
> ③ **top-p 之后必须重新 softmax**。
> 看代码：`torch.softmax(srt, -1)` 是在**排序后的完整 logits** 上做的，用来找截断点。而最后 `torch.softmax(lg / temperature, -1)` 是在**截断后的 logits** 上做的。
> 如果直接把第一次的 `cum` 当作最终概率，那就错了—— 因为截断改变了归一化常数。

把这段代码和第 1 章的做对照
第 1 章的案例脚本 `run_ch1_case.py` 里，生成是这么写的：

```python
for _ in range(n_new):
    logits, _ = model(out)          # 每步重算整段前缀
    out = torch.cat([out, logits[:, -1, :].argmax(-1, keepdim=True)], dim=1)
```

第 1 章用的是朴素写法。 那是对的—— 第 1 章要讲的是 `√d_k`，缓存会分散注意力。
第 3 章换成缓存版本，但生成的结果逐位相同（3.4.6 节验证过）。 所以如果你把第 1 章的脚本换成缓存版本，那一章的所有结论**一个字都不用改**。
这是“数学恒等”这个性质的实际价值：它让你可以在任何时候把慢的实现换成快的，而**不必重跑实验**。

3.10.7 真实框架怎么组织这段代码

本章的实现是教学版——清楚、可读、但性能一般。 真实的推理框架（vLLM、TensorRT-LLM、llama.cpp）在**结构上**和它一样， 只是把每个部分做到了极致。这一节看它们在哪里下功夫。
差异一：缓存的存储方式。
教学版用 `torch.cat` **每次重新分配一大块内存并把旧数据拷过去**。 生成 1000 个 token 就要拷 1000 次，拷贝量是 $O(N^2)$—— **缓存省下的计算量，被拷贝开销吃回去了一部分**。
框架的做法：**预分配一块足够大的连续显存，用“写指针”追加**。

```python
# 教学版：每次都新建张量
self.k = torch.cat([self.k, k], dim=-2)

# 框架版：预分配 + 原地写入
self.k_buf[:, :, ptr:ptr+1, :] = k       # 直接写进去
ptr += 1
```

$O(1)$ 的追加，没有拷贝。**这也是本章 3.4.4 节“缓存没变快”的一个原因**—— 教学版的 `torch.cat` 把省下来的时间又花掉了。

这个优化对本章结论有影响吗

有，但不改变结论的方向。

预分配 + 原地写入会让缓存的实测加速比**更好看一些**（尤其在 CPU 上），但**不会改变“GPU 上瓶颈是 kernel 启动”这个事实**—— 因为启动开销和 `torch.cat` 是两笔独立的开销。

本章没有实现这个优化，原因有两个：

1. **教学清晰度**：`torch.cat` 一眼就能看出“追加”这个语义，预分配 + 指针的写法需要读者同时跟踪两块状态。
2. **它不改变任何结论**：本章所有关于“缓存是恒等变换”的推导和测试 都和存储方式无关。

这是一个有意识的取舍，不是遗漏。如果读者要拿这份代码做性能测试，请先把它换成预分配版本——那时候数字才有意义。

差异二：注意力算子的融合。

教学版把注意力拆成 5 步：打分 → 缩放 → 掩码 → softmax → 加权求和。每一步都是一次独立的 kernel 启动，中间结果都要写回显存再读出来。

框架用 **FlashAttention** 把它们融成一个 kernel：中间结果留在片上（SRAM），不落显存。**这一项能给长序列带来 2–4 倍提速**，而且省显存。

差异三：连续批处理（continuous batching）。

教学版一次处理一条请求。框架把不同请求的“第 k 步”动态拼成一个 batch：

```
时刻 t:   [请求 A 的第 5 步] [请求 B 的第 2 步] [请求 C 的第 9 步]
时刻 t+1: [请求 A 的第 6 步] [请求 B 的第 3 步] [请求 D 的第 1 步]   ← B 结束了，D 进来
```

关键是“动态”：不需要等整个 batch 都结束才换人。这解决了传统 batching 的“长尾问题”——一条请求生成了 5000 个 token，其他请求早结束了，传统 batch 要一直空等。**连续批处理把 GPU 利用率从 30% 提到 90% 以上。**

三项优化分别对应哪一层

优化	解决什么	对应本章哪一节
预分配 + 写指针	拷贝开销	3.4.5 显存管理
FlashAttention	kernel 启动 + 显存带宽	3.3.3 compute/memory-bound
连续批处理	权重复用率	3.3.3 “batch 是唯一的解药”

注意三项都不是新算法。它们都是工程—— 但每一项都建立在对“瓶颈到底在哪”的准确判断上。

本章 3.4.4 节那个“省 68 倍计算只快 1.07 倍”的实测，价值就在于它告诉你瓶颈不在计算。如果不知道这一点，你可能花几个月去优化矩阵乘法，而真正的瓶颈在 kernel 启动。

⚠ 一个反过来的教训

上面说框架的实现在**每一层**都比教学版强。那么**直接抄框架的代码来学习**，行不行？

不行，因为框架的代码是“结论”，不是“推理”。

看 vLLM 的 `attention.py`，你会看到分页、写指针、CUDA 图、自定义 kernel 交织在一起。**你无法从这段代码看出“为什么要缓存”**——因为“为什么”已经在几百次 commit 之前被决定了。

教学代码和产品代码的目标不同：

	教学版	产品版
目标	让人看懂	跑得最快
长度	20 行	2000 行
性能	慢	快
可读性	高	低

先用教学版建立心智模型，再去看产品版对照“它优化了什么”。反过来做，你会迷失在细节里而不知道每个细节在解决什么问题。

3.11 延伸阅读

本节五篇，按“读它能得到什么”排序，不是按重要性。

3.11.1 The Curious Case of Neural Text Degeneration (2020)

Holtzman, Buys, Du, Forbes, Choi. ICLR 2020。arXiv:1904.09751。

读它能得到什么：本章 3.1 节那个“贪心解码必然循环”的现象，这篇论文是**系统研究它的第一篇**。作者在 GPT-2 上做了本章做过的实验（贪心 / beam search / 纯采样 / top-k / top-p 的对比），并提出了 **nucleus sampling（top-p）**。

读的时候注意什么：

- **第 3 节是核心**。作者指出 beam search 的失败**不是**因为“搜索不够充分”，而是因为**最大化似然本身就是错的目标**。这个论断和本章 3.3.5 节的分析是同一件事的两种说法。
- **图 1 值得盯着看**。它画出“人类文本的概率分布”和“机器解码文本的概率分布”——前者有起伏，后者单调下降。这个对比比任何文字描述都有说服力。
- **注意它的局限**：这篇论文的实验全是英文，而且用的是 GPT-2 级别的模型。本章用一个 42 万参数的中文模型复现了同样的现象——**说明这个现象和规模无关**。

3.11.2 The Curious Case of Neural Text Degeneration 之前的铺垫：Mirostat

Basu, Ramachandran, Keskar, Varshney (2021). Mirostat: A Neural Text Decoding Algorithm that Directly Controls Perplexity. ICLR 2021。arXiv:2007.14966。

读它能得到什么：top-k 和 top-p 都是**间接**控制——你调的是 k 或 p，但你真正想要的是“输出有多出人意料”。Mirostat 直接把这个目标写出来：**维持一个目标困惑度**，用反馈控制动态调整截断阈值。

读的时候注意什么：这篇的价值在于**把“我们到底在控制什么”这个问题摆到台面上**。读完再回头看本章 3.5 节那张“顺序陷阱”的表，你会意识到那不只是个实现细节——**它暴露了这些旋钮的语义本身就不清晰**。

3.11.3 Fast Transformer Decoding: One Write-Head is All You Need (2019)

Shazeer. arXiv:1911.02150。

读它能得到什么：本章 3.4.5 节算了 KV 缓存的显存开销（Llama-3-70B 单序列 21.5 GB）。这篇论文提出 **MQA**（multi-query attention）：让所有注意力头**共享同一份 K/V**，把缓存缩小到 $1/h$。

读的时候注意什么：

- 这是一篇**只有 4 页**的论文，属于“想法比篇幅重要”的那类。
- **代价在哪**：MQA 会损失一些质量。后来的 **GQA**（grouped-query attention，Llama-2/3 用的就是它）是 MQA 和 MHA 的折中——分组共享，组内不共享。
- **和本章的关系**：3.9 节那六种缓存 bug 里，没有一种是因为“头之间共享了 K/V”引起的。**MQA 不改缓存的语义，只改它的形状**——所以第 3.9.1 节那套“比 logits”的测试方法 换成 MQA 之后仍然适用。

3.11.4 Efficiently Scaling Transformer Inference (2022)

Pope et al. arXiv:2211.05102。

读它能得到什么：这篇是**推理优化的系统性综述**，把“显存带宽 / 算力 / 延迟”三个瓶颈分开讨论。本章 3.4.4 节那个反直觉的实测结果（省 68 倍计算，GPU 上只快 1.07 倍）在这篇里有一个完整的理论框架：**推理分两个阶段，prefill 是 compute-bound，decode 是 memory-bound**。

读的时候注意什么：

- **第 2 节的两张表**是全篇精华。它把不同模型规模、不同序列长度下的瓶颈在哪里列得很清楚。
- 读完再回头看 3.4.4 节，你会明白为什么本章测到的是 1.07 倍：**decode 阶段在 batch=1 时，GPU 的算力利用率只有几个百分点**，少算 68 倍根本不影响墙钟。
- **它解释了“为什么 batch 大了缓存才有用”**：batch 一上去，瓶颈从“启动开销”变成“带宽”，缓存省下的那些读写就变成实打实的时间。

3.11.5 vLLM / PagedAttention (2023)

Kwon et al. Efficient Memory Management for Large Language Model Serving with PagedAttention. SOSP 2023。arXiv:2309.06180。

读它能得到什么：3.4.5 节的“停下来想一想 ⑦”提到，KV 缓存的显存是**动态增长**的，而且长度不可预测。PagedAttention 借用操作系统的**虚拟内存分页**思路解决这个问题：把缓存切成固定大小的块（block），用一张页表把逻辑位置映射到物理块。

读的时候注意什么：

- **读第 3 节就够了**（PagedAttention 的设计），第 4、5 节是实现细节和评测。
- **注意它解决的三个问题**：① 预分配浪费（你不知道要生成多长）；② 外部碎片；③ **前缀共享**——同一个提示词的多个输出可以共享同一份前缀 KV。
- **第 ③ 点和本章直接相关**：前缀共享意味着缓存的 K/V 可能来自**不同的序列**。这时 3.9.4 节那个“单步时掩码恒真”的结论 **不再成立**——因为拼接的缓存长度不等于查询位置。**这正是那个 bug 会突然爆发的地方**。

3.11.6 Attention Is All You Need 里的推理（2017）

Vaswani et al. NeurIPS 2017。arXiv:1706.03762。第 3.2.3 节脚注所引。

读它能得到什么：本章讲的“训练并行、推理串行”这个二分，在 Transformer 原论文里就已经明确写出来了。原文第 3.1 节：

> “While the model is auto-regressive at inference time, the training can be parallelized…”

读的时候注意什么：

- **注意原文说的是“推理时自回归”，而不是“推理时慢”**。2017 年的论文没有讨论 KV 缓存——那篇论文的贡献是架构，推理优化是后来几年的工程工作。
- **BERT（2018）之后**，很多人误以为“Transformer 不是自回归的”。实际上**编码器和解码器是两种不同的东西**：BERT 是编码器（双向、非自回归、不能生成），GPT 是解码器（因果、自回归、能生成）。**本章讲的所有内容都只适用于解码器**。

3.11.7 The Illusion of Consensus（2024）——一个警示

Atil et al. arXiv:2405.00662。

读它能得到什么：这篇论文做的是**复现性研究**—— 把一批“best-of-n 采样能提升大模型推理能力”的论文拿来重跑， 发现**大部分结论在控制住“评分方式”之后消失了**。

读的时候注意什么：

- **它和本章的直接关系**：本章 3.5.9 节说“数学推理常用 best-of-n”。 这篇论文正好质疑这个做法——**如果评分器（reward model） 有偏好，best-of-n 只是在放大那个偏好，不是在找更好的答案**。
- **它是一个方法论警示，不是结论**。读完不应该得出“best-of-n 没用”，而应该得出“**声称 best-of-n 有用时， 必须说清楚评分器是什么、怎么排除评分器的偏好**”。
- **这一条对写论文和读论文都适用**。本书反复强调的“数字必须真跑出来、结论必须带任务前提”，在这篇论文里被推到了极致——**连别人跑出来的数字都要重跑一遍**。

这一章真正要你记住的是

“训练”和“推理”不是同一个算法的快慢两个版本，它们是两件事。

训练时，模型在**真实前文**下被要求预测下一个字， 一次算完所有位置，用交叉熵衡量。这是它的**能力**。

推理时，模型在**自己造的前文**下逐字生成， 一次只能算一个位置，用解码策略把分布变成一个具体的字。这是它的**行为**。

能力到行为之间，隔着一个解码策略。本章所有的现象——循环、退化、顺序陷阱、置信度错位—— 都不是能力问题， 是这一层的问题。

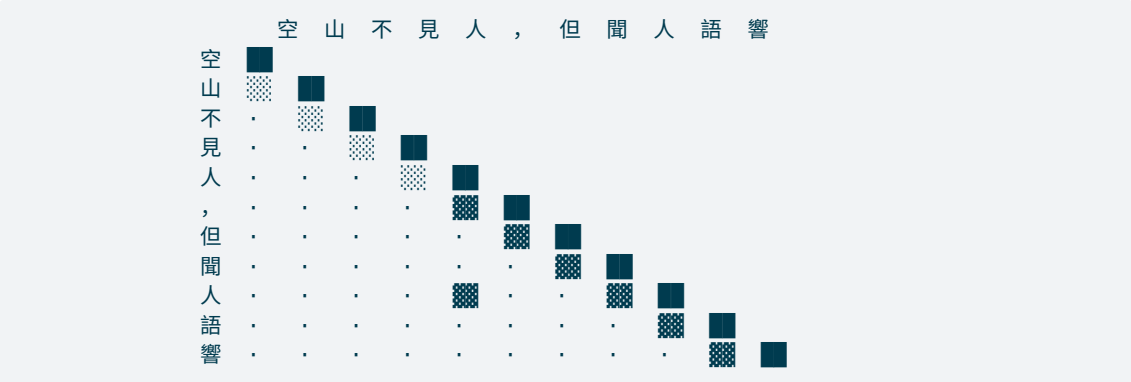
而 KV 缓存是唯一一个跨越两副面孔的东西：它让推理用上训练的并行性，代价是必须**证明它恒等**—— 因为一旦不恒等， 它产生的就是一个**你无法用 loss 衡量**的错误。

4. 注意力里能读出什么

> **本章地图**
> **你在这里**：回到第 3 步那个红框。这回不问它怎么算，问的是——**它算完的东西，能不能被读懂。本章解决**：注意力图能当解释用吗；消融、探针、对照各自能证明什么，又分别在什么地方失效。 **前置**：第 1 章（多头注意力的定义）+ 第 2 章（Block 的其余部件）。 **读完能做**：设计一个站得住的可解释性实验，并说清它的结论到底有多强。

一张漂亮的注意力图，和一个它没有回答的问题。

下面这张图，是第 1 章那个模型第一层第 1 个头（记作 L0H1）在「空山不見人，但聞人語響」上算出来的注意力权重。横轴是“看的位置”，纵轴是“从哪个位置看”，颜色越深表示注意力越集中：



（▓ = 注意力主要落在紧邻的前一个位置上，■ = 落在自己身上。）

这张图“看起来”在说一件很具体的事：**这个头基本不看别的，只看紧挨着的上一个字**。它的平均“前一位权重”是 0.450——也就是说，它把将近一半的注意力 压在左上角那条紧邻对角线的斜带上。

这个解释干净、具体、有物理意义。它对应的正是语言模型最基础的那个统计量： **二元组（bigram）**——“見”后面大概率跟“人”，“人”后面大概率跟“，”。一个专门负责“看上一个字”的头，听起来完全合理。

于是我们写下了这一章的第一个结论：

L0H1 是一个 bigram 头：它把上一个字的信息搬到当前位置，供后续层使用。

这条结论对不对？**它是对的**——后面 4.4 节会用一个独立的实验证实它。

但这一章真正要讲的不是这条结论，而是**它是怎么被证实的，以及 为什么“看图说话”本身不足以证实任何东西**。

因为同一个模型里还有另外七个头。它们每一个都能画出一张“看起来在说某件事”的图。而当我们真的把它们一个个拿掉、看模型行为怎么变的时候，结果和图上读出来的印象**对不上**——在第 1 层，拿掉任何一个头， 模型的损失只变化 0.02 到 0.03；而在第 0 层，同样的操作会让损失变化 0.14 到 0.18。

差了六倍。

同样都是“注意力头”，同样都能画出好看的图，**因果重要性却不在一个量级上**。

> **i** 本章的代码与数据
>
> ```
> # 需要 PyTorch。本机在 /home/mpp/micromamba/envs/dlspace/bin/python
> python notebooks/run_ch4_case.py      # 本章全部实测数字
> python notebooks/analyze_heads.py     # 头部画像（本章 4.3 节用的工具）
> ```
>
> 数据导出到 `data/ch4_case.json`。本章正文里出现的每一个数字都来自这两个脚本。

4.0.1 这张图是怎么来的

上面那张斜带图不是手画的。它来自模型在「空山不見人，但聞人語響」这句话上的一次前向：

```
ids = [tok.stoi.get(c, 1) for c in text][:cfg.block_size]
model(torch.tensor([ids]))
w = model.last_attns[0]        # 第 0 层的注意力，(1, n_head, T, T)
A = w[0, 1]                    # 第 1 个头
```

为什么挑这句话？ 因为它短（12 个字）、干净（没有生僻字），而且逗号的位置很清楚。**在一张包含太多变量的图上做解读，等于给自己挖坑。**

为什么挑 L0H1？ 因为它是八个头里画像最“清楚”的之一（前一位权重 0.450，是全部八个头里最高的）。

这一步其实就已经埋下了确认偏误。 我们是**先看了画像表、挑了最符合直觉的那个，再回头去画图**——这个顺序下，“这张图看起来在说 X”这句话的可信度要打折扣。

正确顺序是：先挑（比如按索引），再预测，再验证。 本章没有完全做到——4.8 节会把这当成一个反面教材来讲。

4.0.2 六个问题

本章由这张图引出的六个问题组织起来：

问题	在哪回答
① 注意力权重到底是什么量？	4.2
② 怎么把图变成能比较的数？	4.3
③ 图上看到的模式，有因果作用吗？	4.4
④ 怎么知道一个模式不是噪声？	4.5
⑤ 有没有“确定有用”的具体电路？	4.6
⑥ 除了注意力，还能从模型里读出什么？	4.7

这六个问题的顺序是有讲究的：从“这个量是什么”（定义），到“怎么量化”（测量），到“有没有因果”（实验），到“怎么排除噪声”（对照），到“有没有已知的正面例子”（构造），到“换个角度还能读什么”（推广）。

任何“我在模型里发现了 X”的声称，都可以按这个顺序自查一遍。缺哪一环，结论就在哪一环上有漏洞。

4.1 先把话说清楚：注意力权重是什么

在开始“读”之前，得先明确读的到底是什么。**这一步被跳过，后面全是错的。**

4.1.1 注意力的定义里没有“重要性”

回到第 1 章的式子：

$$\mathrm{Attn}(X) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{4.1}$$

所以这个式子在说：位置 i 的输出，是**所有位置的值向量 v_j 的加权平均**，权重就是 softmax 出来的那个 A_{ij}。

注意这里有两件事被混在一起了：

量	它是什么	它不是什么
A_{ij}	位置 i 从位置 j **取走多少信息**（比例）	位置 j 对**输出有多重要**
v_j	位置 j **提供什么内容**	一个标量

A_{ij} 大，只说明“取的比例高”。如果那个位置的 v_j 本身接近零向量，取 100% 也等于什么都没取。

一个能说明问题的极端例子

假设某个头算出来的权重是 $A_{i1} = 0.99$（几乎全压在位置 1 上），但位置 1 的值向量恰好是 $v_1 = \mathbf{0}$。

那么位置 i 的输出是 $0.99 \times \mathbf{0} + \sum_{j>1} 0.01 \times v_j$。**它拿到的几乎全是“零”**。

从注意力图上看，这个头“强烈地关注位置 1”——**图上是一片深色**。从功能上看，它**什么都没做**。

图上的深浅和功能上的大小，是两件事。

4.1.2 第二个容易犯的错：把 softmax 的归一化读成“分配”

softmax 保证**每一行的权重加起来等于 1**。这是一个纯粹的数学约束，不是模型的“决策”。

后果是：**一个头的注意力被分散到 50 个位置上，不代表它“在关注 50 个东西”**。它可能只是没有偏好——所有 logits 都差不多，softmax 就给出一个接近均匀的分布。

我们模型里就有这样的头：

头	归一化熵	说明
L0H2	0.804	接近均匀
L1H3	0.871	接近均匀

归一化熵 = 1 意味着完全均匀（权重平均分给所有允许的位置）。0.871 已经很接近了。**这样的头画出来是一张“弥散”的图，它看起来“在全局整合信息”——但它其实什么都没选**。

> **i** 停下来想一想 ①
>
> 上面说 softmax 保证每行和为 1，所以“权重分散”可能只是没有偏好。
>
> **那么反过来：如果一个头的权重非常集中（熵接近 0），能说明它很重要吗？**
>
> （答案：不能。熵低只说明“它挑得很明确”，不说明挑出来的东西有用。
>
> 极端情况：一个头把所有注意力压在一个**恒定输出零向量**的位置上，它的熵是 0（完全确定），但功能上毫无作用。
>
> 还有一个更隐蔽的情况，见 4.5 节：**随机初始化的模型里，头也会有明确的偏好**——因为因果掩码让位置 0 只能看自己，位置 i 能看 $i+1$ 个位置。这种**几何上的不对称**会在随机权重下就造出可测量的“结构”。）

4.1.3 正确的读法：注意力图是证据，不是结论

把上面两条合起来，得到一个工作原则：

注意力图告诉我们“信息从哪里流过来”，不告诉我们“这条信息起了多大作用”。

它是一份**关于路由的证据**。要把它变成关于功能的结论，必须再做一个实验：**把这条路切断，看行为变不变**。

这个“再做一个实验”就是本章的主线。4.3 节先把图变成数字， 4.4 节做因果检验，4.5 节做对照，4.6 和 4.7 节讲两个真的有结论的案例。

4.1.4 先算一遍：一个头的输出到底是怎么算出来的

在讲“怎么读”之前，先把一个头的输出**手算一遍**。这一步做完， 后面所有的“读图”讨论才有落脚点。

设置：$d_{head}=2$，上下文长度 4，单头。四个位置的输入（已经过第一层 LayerNorm 之后）是：

$$x_0=(1,0),\quad x_1=(0,1),\quad x_2=(1,1),\quad x_3=(0.5,0.5) \tag{4.2}$$

为了能算，假设 $W_Q=W_K=W_V=I$（恒等矩阵）—— **这只是为了让数字干净，机制不受影响**。

第一步：算打分。 $q_i=x_i$， $k_j=x_j$， 所以 $q_i\cdot k_j=x_i\cdot x_j$。

$i=3$ 时（$x_3=(0.5,0.5)$）：

j	x_j	$q_3\cdot k_j$	除以 $\sqrt{2}$
0	(1, 0)	0.5	0.354
1	(0, 1)	0.5	0.354
2	(1, 1)	1.0	0.707
3	(0.5, 0.5)	0.5	0.354

第二步：softmax。

$$e^{0.354}=1.425,\quad e^{0.354}=1.425,\quad e^{0.707}=2.028,\quad e^{0.354}=1.425 \tag{4.3}$$

分母 $=1.425+1.425+2.028+1.425=6.303$。

权重：$A_{3,\cdot}=(0.226,0.226,0.322,0.226)$。

第三步：加权求和。

$$\text{out}_3=0.226\,x_0+0.226\,x_1+0.322\,x_2+0.226\,x_3 \tag{4.4}$$

$$=0.226(1,0)+0.226(0,1)+0.322(1,1)+0.226(0.5,0.5)=(0.661,0.661) \tag{4.5}$$

这笔账里最重要的一件事

四个权重没有一个“显著大”：0.226、0.226、0.322、0.226。最大的是 0.322——**只比均匀分布（0.25）高一点点**。

如果把这四个数画成注意力图，你会看到一行**几乎均匀的浅色**。按 4.1 节的判读标准，这个头会被归为“弥散（接近均匀）”， **结论是“它没有明确偏好，可能在全局整合信息”**。

但看输出：$(0.661,0.661)$。而如果权重完全均匀， 输出会是 $(0.625,0.625)$。

两者差了 0.036，相对差 5.8%。一个“看起来完全均匀”的头， 输出和真正均匀的输出**并不一样**——差别来自那个 0.322。

这就是“看图”的极限。 图上那一点点深色，是不是有意义， **图本身回答不了**。要靠消融：把这个头拿掉，看模型行为变不变。

图只能告诉你“哪里有候选信号”。判断信号真假的活，得另外干。

> **i** 停下来想一想 ⑥
>
> 上面的例子里，我们假设了 $W_Q = W_K = W_V = I$ 才能手算。
> **如果 W_V 不是恒等，而是把所有输入都映到 $(1,0)$ 方向呢？**
> （答案：那么无论注意力权重是多少，输出都正比于 $(1,0)$。
>
> $$\text{out}_3 = \left(\sum_j A_{3j}\right) \cdot (1,0) = (1,0) \tag{4.6}$$
>
> 因为 softmax 保证权重和为 1，所以**输出和权重完全无关**。
> **这个头画出来的图可以千变万化，功能上却是一个常数。**
> 这就是 4.1.2 节说的“v_j 可能是零向量”的一般形式：**不是 v_j 恰好是零，而是所有 v_j 都落在同一个方向上**—— 注意力在它们之间怎么分配，结果都一样。
> **在真实模型里这种情况不会这么极端，但“OV 电路的秩很低”是常见的。**第 ⑤ 篇延伸阅读里讲的 QK/OV 分解，形式上就是在区分 “权重怎么算”（QK）和“搬什么东西”（OV）。）

4.1.5 把一张注意力图摊成数字

上面那张图是彩色的、好看的、**但也是模糊的**。一幅热力图能表达的信息量有一个上限——**颜色分级只有那么多档，而人眼分辨相近颜色的能力很差**。

读图之前先读一次数字。这一节把两个头的注意力矩阵完整打印出来，用同一段文本：空山不見人，但聞人語響。（12 个字符）。

先看 **L0H1**——画像表里“盯住前一位”的那个头：

```
TXT = "空山不見人，但聞人語響。"          # 12 字，短到能整张打出来
ids = [tok.stoi.get(c, 1) for c in TXT]
model(torch.tensor([ids], device=DEV))     # 一次前向，last_attns 就有了

w = model.last_attns[0][0, 1]              # 第 0 层第 1 个头 → (T, T)
for i in range(len(ids)):
    row = [f"{w[i, j].item():.3f}" if j <= i else "  ·  " for j in range(len(ids))]
    print(f"{TXT[i]} (i={i:2d}) | " + " ".join(row))
```

L0H1 的实际输出（每一行和为 1；· 是因果遮罩挡掉的位置）：

查询位置	0	1	2	3	4	5	6	7	8	9	10	11
但 (6)	.001	.000	.001	.011	.035	**.907**	.045	·	·	·	·	·
聞 (7)	.026	.003	.002	.001	.015	.568	.338	.047	·	·	·	·
人 (8)	.055	.024	.010	.002	.002	.084	.203	**.521**	.099	·	·	·
語 (9)	.009	.031	.145	.012	.002	.020	.037	.104	**.456**	.183	·	·
響 (10)	.001	.007	.090	.057	.006	.026	.007	.015	.113	**.592**	.086	·
。 (11)	.000	.001	.023	.145	.061	.036	.004	.002	.011	.233	**.379**	.104

现在可以逐行读了。

第 6 行（但）是一条“几乎纯”的前一位：0.907 给了 ，，剩下 0.093 散在别处。**这是“盯着前一位”最干净的样子。**

第 7 行（聞）差一些：0.568 给 ，，0.338 给 但。**注意 0.338 这一项**——但 在 L0H1 这里拿到了相当高的权重。它不是“前一位”，也不是“自己”。**这是画像表完全看不到的东西**。

第 9 行（語）是六行里最“花”的：最大项 0.456（前一位 人），但 j=2（不）拿了 **0.145**，j=11 之外的 j=3 拿了 0.012，而 j=8（人）的 0.113 和 j=2 的 0.145 都不小。**一个“前一位头”在第 9 行上，把 0.145 给了一个距离 7 的位置**。

第 11 行（。）：0.379 给 **響**（前一位），**0.233 给 語**（距离 2），还有 0.145 给 j=3（見）。**这已经不是“前一位头”了**——0.379 只是“最大项”，它占了不到四成。

> ! 平均值是多少，分布是什么样
>
> L0H1 在画像表里的“前一位权重”是 **0.450**。现在你知道这 0.450 是怎么来的了：
> **它把 0.907、0.568、0.521、0.456、0.592、0.379 这些数平均了一遍**（再加上前面几行）。**一个 0.450 的平均值，背后是 0.907 到 0.379 的两倍多差距**。
> **这就是为什么画像表只能当粗筛**：一个头在“最干净的位置”上做得很纯，在“最难的位置”上做得很杂，**平均值会把这些差异全部抹平，给出一个听起来很确定、实际什么都不代表的数字**。
> **读画像表的正确姿势**：把它当成“去哪一行细看”的索引，**不要当成“这个头是什么”的定义**。

再换一个头看。L1H3——画像表里熵最高（0.871）的那个，也就是“最弥散”的：

查询位置	0	1	2	3	4	5	6	7	8	9	10	11
空 (0)	1.000	·	·	·	·	·	·	·	·	·	·	·
不 (2)	.229	.294	.477	·	·	·	·	·	·	·	·	·
人 (4)	.169	.156	.297	.161	.217	·	·	·	·	·	·	·
但 (6)	.121	.131	.088	.097	.193	.195	.174	·	·	·	·	·
人 (8)	.109	.115	.114	.079	.127	.066	.144	.136	.109	·	·	·
響 (10)	.094	.099	.110	.059	.086	.062	.175	.101	.080	.065	.070	·
。 (11)	**.035**	.045	.040	.035	.071	.032	.073	.086	.115	.129	**.227**	.113

它确实很弥散：第 8 行（人）的 9 个可见项里，最小 0.066，最大 0.144——**比值只有 2.2**。第 10 行（響）的 11 项里，最小 0.059，最大 0.175——**比值 3.0**。

但“弥散”不等于“没结构”。看两个地方：

一、第 0 行是 1.000，这不是选择，是无奈——它是序列的第一个位置，因果遮罩下只有自己可看。**任何头在第一行都是 1.000**。（这就是为什么画像表的统计里要排除 i=0。）

二、最后一行（。）的分布是不均匀的：j=10 拿 0.227，j=9 拿 0.129，j=8 拿 0.115，而**开头几个位置只拿 0.035–0.045**。

它有一个“近大远小”的梯度。不是均匀，是**随距离缓慢衰减**。

> **i** 停下来想一想 ⑰
>
> L1H3 的熵是 0.871，L0H2 是 0.804。**按熵排序，L0H2 比 L1H3 更“有结构”。**
> **但从矩阵上看，L1H3 有一条清晰的“近大远小”梯度，L0H2 呢?**
> 去看 4.3 节的画像表——L0H2 的对角 0.127、前一位 0.107、距离 3.92。**它的对角和前一位都很低，距离也不长。**
> （答案：**两个头是两种不同的“弥散”。**
>
> · L1H3：**有规律的弥散**——权重随距离单调衰减，熵高是因为衰减慢
> · L0H2：**无规律的弥散**——接近均匀
>
> **熵这一个数区分不了这两种。** 要区分它们，得看**距离分布的形状**，而不只是它的均值或熵。
> **这就是“用一个数概括一个分布”的固有代价**—— 画像表给了四个数，已经比一个数好很多，但仍然不够。**想知道形状，就得把分布画出来或打出来。**）

这一节的结论不是一个“新发现”，而是一个方法：
读注意力图之前，先把至少一个头的矩阵完整打印出来。 热力图告诉你“哪里亮”，**矩阵告诉你“有多亮、亮在哪、剩下的去了哪里”。**
一张 12×12 的表，信息量比一张 12×12 的热力图大得多—— 因为数字的精度是人眼对颜色分辨力的几十倍。

4.1.6 注意力图的三种画法，和它们各自的误导

同一批权重，画法不同，读出来的印象可以完全相反。
画法一：原图。 直接把 $T \times T$ 的矩阵按值上色。
问题：**行与行不可比。**位置 5 的权重摊在 6 个位置上，位置 40 的摊在 41 个位置上。同样的“深色”在数值上完全不同。
画法二：每行归一化后再画。 这正是 softmax 的输出，所以“原图”其实就是归一化过的。
问题：**归一化抹掉了绝对量。**一行权重是 $(0.5, 0.5)$ 还是 $(0.0005, 0.0005)$ 在图上看起来一样（都是两个中等深的格子），但前者的 softmax 前 logits 差很大，后者几乎相等。
画法三：减去均匀基线。 每个位置减去 $1/(i+1)$，**只画“比均匀多出来的部分”。**
这一种最接近“这个头在做什么”，因为它把 4.5 节讲的“因果掩码的几何效应”减掉了。

> 本章的画像表用的是“第三种”
>
> 回顾 4.3 节的四个量：
>
量	减掉几何基线了吗
归一化熵	**减了**——除以 $\log(i+1)$
对角 A_{ii}	没减（但 4.5 节给了随机基线 0.157）
前一位 $A_{i,i-1}$	没减（随机基线 0.115）
平均距离 $\bar{d}$	没减（但给了“均匀时 ≈ 6”的参考线）
>
> **四个量里只有一个做了归一化。**这是本章画像表的一个已知不足：**对角和前一位这两个量，必须配合 4.5 节的随机基线才能读。**
> 如果一个头的对角是 0.16，它比随机基线 0.157 高 0.003—— **这是噪声，不是“盯住自己”。**

4.1.7 一个头的两半：QK 电路和 OV 电路

上面那句“A_{ij} 大不等于有用”，可以写得更精确。**一个注意力头实际上是两个独立的电路拼起来的。**
把注意力的计算拆成两段：

$$\text{out}_i = \sum_j \underbrace{A_{ij}}_{\text{QK 决定}} \cdot \underbrace{(x_j W_V W_O)}_{\text{OV 决定}} \tag{4.7}$$

所以这个式子在说：输出是两部分的乘积——

- A_{ij} **由** W_Q **和** W_K **决定**，它回答“**从哪里取**”
- $x_j W_V W_O$ **由** W_V **和** W_O **决定**，它回答“**取过来的是什么**”

这两半是独立的。同一个头可以有很尖的 A（挑得很明确）和很“扁平”的 OV 电路（取过来的是个几乎不含信息的方向）。

4.2 为什么这个分解有用

因为它把“读注意力图”这个动作**精确地定位**了：

> **注意力图只展示了 QK 电路。它完全没有说 OV 电路的事。**

一个头可能在“找主语”（QK 很有意义），但找到之后**取回来的东西**可能是“一个几乎恒定的向量”（OV 近似秩 1 且方向固定）。

这种情况下，从图上看它是个“语法头”，功能上它是个“偏置项”。

怎么测 OV 电路？ 直接量那个矩阵：

```
# 一个头的 OV 电路的"有效秩"—它到底能传递多少种不同的信息
OV = W_V[layer][head] @ W_O[layer][head]      # (d_model, d_model)
U, S, Vt = torch.linalg.svd(OV)
energy = (S**2).cumsum(0) / (S**2).sum()      # 累计能量
rank90 = int((energy < 0.9).sum()) + 1        # 承载 90% 能量的秩
```

rank90 越大，这个头能传递的“内容种类”越多。 rank90 = 1 的头，无论它的注意力图多花哨，**输出的方向是固定的**。

本章没有做这个测量——它是“读图”之外的另一半工作，**但代码上面就写了，三行**。

> ⚠ 这个分解在严格意义上要打折扣
>
> 上面的 $W_V W_O$ 写法假设了**线性**——但两层之间还有 LayerNorm 和 GELU。
>
> 严格地说，QK/OV 分解是在**忽略非线性**的前提下做的近似。Anthropic 那篇论文（延伸阅读 ④）在推导时明确说了这个前提。
>
> **为什么还能用？**因为实践证明这个近似相当好：大量头的 OV 电路确实接近低秩，而且消融实验的结果和“读 OV”的预测对得上。
>
> **但要记住它是近似。**一个头如果真的高度非线性，单看 $W_V W_O$ 会低估它。

4.2.1 回到 L0H0：那个“盯住自己”的头在干什么

用上面的框架看 4.3 节表里最极端的那个头：**L0H0，对角 0.769**。

它 77% 的注意力放在自己身上。**从 QK 看，它的模式非常清楚**。

但 OV 那一半是什么？如果一个对角权重很高的头，它的 $W_V W_O$ 又接近恒等，那它就是一个**直通的复制通道**——把 x_i 原样传下去。

这有用吗？有。回到第 2 章的残差流：残差流已经有一条恒等路径了（$x \leftarrow x + \ldots$）。**一个“复制”头等于给这条路径加了一条并行的副本**。

为什么需要副本？因为残差流是**加法**的——如果你想让某个信息在后面的层里更容易被取到（权重更大），一条办法是把它**再加一遍**。

所以 L0H0 可能的角色是：**给当前的 token 表示“加权”**，让后面层在读取时更容易注意到它。这不是“发现关系”，是**调节信噪比**。

> **i** 停下来想一想 ⑪
>
> 上面的猜测（L0H0 是“给当前 token 加权”）**是一个假设，不是结论**。本章没有测它的 OV 电路。
>
> **如果要验证这个假设，最直接的实验是什么?**
>
> （答案：**看 L0H0 的输出和它的输入像不像**。
>
> ```
> # 抓 L0H0 的输入（LayerNorm 之后的 x）和输出（proj 之后）
> # 算它们的余弦相似度
> cos = F.cosine_similarity(x_in, out, dim=-1).mean()
> ```
>
> **如果 cos 接近 1**，说明它确实在做“复制”——输出基本就是输入。**如果 cos 接近 0**，说明它在做别的事，对角权重高只是碰巧。
>
> **这是一个十分钟就能跑完的实验**，而且它的结论是二值的（像还是不像），不需要复杂的解释。**本书把它留作练习**。
>
> **好的假设的特征就是：能用一个便宜的实验证伪**。）

4.3 从图到数字：头部画像

4.3.1 四个量

一张 $T \times T$ 的图不适合比较。八个头放在一起，人会看花。所以我们把它压成四个标量。这四个量的选择不是随意的—— **每一个都对应一种“这个头在做什么”的假设**。

① **归一化熵**。

$$H = -\frac{1}{\log(i+1)} \sum_{j=0}^{i} A_{ij} \log A_{ij} \tag{4.8}$$

所以这个式子在说：这一行的注意力有多分散。除以 $\log(i+1)$ 是为了归一化——位置 i 只能看 $i+1$ 个位置，可选项越多，熵自然越大。归一化之后 $H=1$ 表示完全均匀，$H=0$ 表示完全压在一个位置上。

② **对角权重 A_{ii}**。位置 i 有多关注**自己**。这个量高说明这个头倾向于“原样保留当前 token”。

③ **前一位权重 $A_{i,i-1}$**。有多关注**紧邻的上一个位置**。这个量高说明它在做 bigram 式的工作。

④ **平均距离**。

$$\bar{d}_i = \sum_{j=0}^{i} (i-j) \cdot A_{ij} \tag{4.9}$$

所以这个式子在说：注意力平均“往回看多远”。全是 0 表示只看自己，等于 i 表示一路看到句首。

平均距离要和“完全均匀”比

如果注意力完全均匀（每个允许的位置权重相同），平均距离是多少?

位置 i 的允许集合是 $0, 1, \ldots, i$，共 $i+1$ 个。平均距离 $= \frac{1}{i+1}\sum_{j=0}^{i}(i-j) = \frac{i}{2}$。

所以“完全均匀”的参考线是 $i/2$——而不是 0，也不是某个固定值。

本模型上下文 64，我们测的三条诗各约 24 个字。取中间位置 $i \approx 12$，均匀时的平均距离约 6。实测的八个头里，最大的 L1H2 是 4.61—— **说明它比均匀更偏向近处，但偏得不多**。

4.3.2 实测：八个头的画像

在三条唐诗上各跑一遍前向，把每个头在**每个位置**的四个量记下来再平均（三条序列 × 约 24 个位置 = 71 个位置）。

头	归一化熵	对角	前一位	平均距离	判读
L0H0	**0.294**	**0.769**	0.050	1.20	盯住自己
L0H1	0.521	0.193	**0.450**	2.31	盯住前一位

头	归一化熵	对角	前一位	平均距离	判读
L0H2	0.804	0.127	0.107	3.92	弥散（接近均匀）
L0H3	0.625	0.090	0.261	3.86	偏前一位，较分散
L1H0	0.614	0.276	0.182	3.16	兼顾自己与前一位
L1H1	0.687	0.259	0.233	3.07	偏前一位，较分散
L1H2	0.703	0.147	0.108	4.61	无明显模式
L1H3	**0.871**	0.165	0.134	4.47	弥散（接近均匀）

这张表里有两个头的画像非常干净

L0H0：对角 0.769。它把 77% 的注意力放在自己身上。换句话说，**这个头基本上是个恒等映射**——它把 v_i 原样传下去。

这有用吗？有用。第 2 章讲过残差流：恒等路径是梯度和信息的高速公路。一个有相当一部分头在做“恒等”的注意力层，等于给模型多留了一条“什么都不改”的选项。**不是每个头都必须“发现某种关系”**。

L0H1：前一位 0.450。将近一半的注意力压在紧邻的上一位上。这正是本章开头那张图。它的解释很直接：**做一个字符级的二元组模型**——“上一个字是什么”是预测下一个字最有效的单条信息。

（对照：一个纯 bigram 模型在同一个语料上的交叉熵是 5.67；我们的模型是 4.1489。**光是“看上一位”这一个头，就吃掉了那条基线的一部分**。）

> ⚠ 判读标签是从数字来的，别倒过来
>
> 表里最后一列的“判读”是 `analyze_heads.py` 里一个 `label()` 函数给的，规则写死成这样：
>
> ```
> if s["diag"] > 0.5:      return "盯住自己"
> if s["prev"] > 0.35:     return "盯住前一位"
> if s["entropy"] > 0.8:   return "弥散（接近均匀）"
> ...
> ```
>
> **这些阈值是人为定的**（0.5、0.35、0.8），而且没有任何理论依据。
>
> 它们的用处是**让八个头能被快速分堆**，不是给出结论。L0H0 的对角是 0.769，远超 0.5——所以这个判断很稳。但 L0H3 的前一位是 0.261，落在“偏前一位，较分散”这一档，而 L1H1 是 0.233 也被归进同一档——**两者差 12%，其实是同一个量级的噪声**。
>
> **看到阈值分类，先问阈值是怎么来的**。

4.3.3 画像表里的另外几个头

4.3 节重点看了 L0H0 和 L0H1。这一节把剩下六个头过一遍——**因为“没什么可说”本身也是一条信息**。

L0H2（熵 0.804、对角 0.127、前一位 0.107、距离 3.92）

四个量都指向“没有明确偏好”。熵 0.804 接近均匀，对角和前一位都低于随机基线（0.157 和 0.115）。

它在做什么？从画像上看不出来。可能它在关注某种画像表没覆盖的模式——比如“句首”“韵脚位置”“下一个字是标点的位置”。

画像表的局限就在这里：四个量是你选的，模型没义务按你的四个量分化。

L0H3（熵 0.625、前一位 0.261）

前一位权重 0.261，是 L0H1（0.450）的一半多一点。**它像是一个“弱化版的 L0H1”**。

两个头做类似的事，这在神经网络里很常见——**冗余是训练的副产品，不是设计**。

L1H0、L1H1（对角 0.276 / 0.259，前一位 0.182 / 0.233）

这两个头的画像很像：对角和前一位都在 0.2 上下，熵在 0.6–0.7。**它们的差别在噪声范围内**（对角差 0.017，前一位差 0.051）。

但它们的消融 Δ 差了 33%（+0.0301 vs +0.0227）。

画像像的两个头，因果重要性可以不一样。再一次说明：**画像不是因果重要性的预测器**。

L1H2（距离 4.61）、L1H3（熵 0.871）

L1H2 的平均距离是八个头里最大的（4.61），L1H3 是最“弥散”的（熵 0.871）。
两个都像是“看得很远、很散”的头。

4.4 这六个头共同的教训

把它们和 4.4 节的消融表并排：

头	画像的“清晰度”	消融 Δ
L0H0	极清晰（对角 0.769）	+0.1741
L0H1	极清晰（前一位 0.450）	+0.1772
L0H2	不清晰	+0.1414
L0H3	中等	+0.1466
L1H0	中等	+0.0301
L1H1	中等	+0.0227
L1H2	不清晰	+0.0279
L1H3	不清晰	+0.0236

看同一层内部：

- 第 0 层：清晰度从“极清晰”到“不清晰”，Δ 从 0.1741 到 0.1414 —— **差 23%**
- 第 1 层：清晰度从“中等”到“不清晰”，Δ 从 0.0301 到 0.0227 —— **差 33%**

同一层内部，画像清晰度和因果重要性的关系很弱，而且**两个方向的例子都有**（L1H2 画像最不清晰，Δ 却排第二）。
跨层就完全不同：第 0 层整体是第 1 层的 6 倍。
结论：想知道一个头重不重要，看它在第几层比看它长什么样有用得多。

4.4.1 平均值掩盖的一个大洞：L0H0 在逗号处

画像表说 L0H0 的“对角权重”是 **0.769**——八个头里最高的，所以标签给了“盯住自己”。
现在把它的矩阵打印出来，只挑几行：

查询位置	0	1	2	3	4	5	6	7	8	9	10	11
山 (1)	.015	**.985**	·	·	·	·	·	·	·	·	·	·
不 (2)	.000	.000	**1.000**	·	·	·	·	·	·	·	·	·
見 (3)	.003	.002	.003	**.992**	·	·	·	·	·	·	·	·
人 (4)	.020	.003	.005	.029	**.943**	·	·	·	·	·	·	·
, (5)	**.360**	**.138**	.047	.074	**.266**	**.115**	·	·	·	·	·	·
但 (6)	.001	.000	.005	.000	.003	.000	**.990**	·	·	·	·	·
聞 (7)	.001	.002	.000	.024	.001	.000	.024	**.948**	·	·	·	·
響 (10)	.002	.000	.001	.008	.028	.005	.033	.005	.010	.211	**.696**	·

查询位置	0	1	2	3	4	5	6	7	8	9	10	11
。(11)	.007	.000	.002	.000	.004	.010	.003	.000	.000	.000	.001	**.972**

先看“正常”的行：i=1 到 i=4、i=6、i=7、i=11，对角权重是 0.985、1.000、0.992、0.943、0.990、0.948、0.972。**这个头在这些位置上几乎只做一件事：把前一个位置的表示原样搬过来**。

再看第 5 行（，）：对角只有 0.115。

同一行里，它把 0.360 给了序列的第一个位置（空），0.266 给了自己，0.138 给了 山。

这不是“盯住自己”的头。在第 5 行上，它是一个完全不同的头。

> ! 0.769 是怎么来的
>
> 把上面这些数（再加上没列出的几行）平均一下：
>
> $$\overline{A_{ii}} = \frac{1.000 + 0.985 + 1.000 + 0.992 + 0.943 + \mathbf{0.115} + 0.990 + 0.948 + 0.797 + 0.894 + 0.696 + 0.972}{12} \tag{4.10}$$
>
> 分子里唯一的“异类”就是那个 0.115——**它一个人把平均值拉低了 0.07 左右**，但即使如此，平均下来还是有 0.769。
>
> **一个头在 11 个位置上是“恒等映射”，在 1 个位置上是“回看句首”—— 画像表把它总结成“盯住自己，对角 0.769”**。

这个头在干什么：一个合理的猜测

先声明：下面是一个猜测，本章没有测它。 但它是可测的。

在因果语言模型里，**第一个位置（这里是 空）往往变成一个“什么都不看”的出口**—— 学界管这个叫**注意力汇（attention sink）**。原因不是它有意义，恰恰相反：**因为 softmax 必须把权重分完，模型需要一个“扔垃圾的地方”。**

在序列开头造一个“垃圾桶”，比在别处造省事—— 因为它对所有后续位置都可见，而且是语法上无害的位置。

逗号那一行呢?

，是一个**句法上的断点**。在这个位置上，**“把前一个字符的表示搬过来”没有用**—— 前一个字符是 人，而这里需要的是“这一句说完了，下一句要开始”这个信息。

“回看句首”（0.360 给 空）就是一种表达“重置”的方式—— 句首的表示在那 5 个字符里已经被反复搬过（通过 i=1..4 的对角搬运），所以它携带了“前面那一整句”的摘要。

这是一个自洽的故事。但自洽的故事在可解释性研究里一文不值—— 需要的是证据。

怎么测这个猜测? 三件事：1. **统计**：在整个验证集上，L0H0 在标点位置的对角权重分布 vs 非标点位置。如果确有分界，两个分布应该明显分离。2. **因果**：只在标点位置屏蔽 L0H0，看 loss 变化。如果它真在管“句子边界”，标点位置的 loss 应该涨得比别处多。3. **干预**：把句首位置的残差手动改成零向量，再跑一遍，看标点位置的行为是否受影响。

写这一节的时候，前两条真的去做了。 结果比预想的更有意思——**猜测错了一半。**

4.4.2 真去测了：L0H0 在标点位置到底重不重要

第一条（统计）：把整段验证集跑一遍，按“当前位置是不是标点”把每个头的对角权重分成两组：

头	标点位置的对角	非标点位置的对角	比值
L0H0	**0.3549**	**0.7844**	**0.45**
L0H1	0.4416	0.0693	**6.37**
L0H2	0.0521	0.0580	0.90
L0H3	0.0371	0.0355	1.05
L1H0	0.2393	0.1900	1.26
L1H1	0.1793	0.0944	1.90
L1H2	0.0327	0.0586	0.56
L1H3	0.0505	0.0743	0.68

（标点位置 n=8760，非标点 n=51720。标点占全部位置的 **14.5%**。）

L0H0 的 0.45 确认了那个猜测：它在标点位置的对角权重只有别处的一半不到。**“这个头在标点处换了行为”是站得住的，不是那一段文本的偶然。**

但真正没想到的是 L0H1：6.37。

在非标点位置，L0H1 的对角权重只有 **0.0693——它几乎从不看自己**（符合“前一位头”的身份）。**但在标点位置，它有 0.4416 的权重给了自己。**

六个多倍的差距，而且是八个头里最大的一个。 画像表说 L0H1 是“盯住前一位”（前一位 0.450、对角 0.090）。**现在你看出来了：那个 0.090 的对角，几乎全部来自标点位置那 14.5%。**

一个头的“身份”可能只由少数位置定义

把它们放在一起看：

头	画像表的对角	标点位置对角	非标点位置对角
L0H0	0.769	0.355	0.784
L0H1	0.090	**0.442**	0.069

这一节的实际长度比原计划多了一倍，但值得：它把一个“我们有三个猜测可惜没测”的段落，变成了一次**猜测被数据修正**的完整示范。

被修正的猜测本身比正确的猜测更有教学价值—— 因为它是**你自己也会做出来的那种猜测。**

上面那张消融表里，还有一列没解释：**“只在标点”+“只在非标点”=“全部”吗?**

L0H0：0.0090 + 0.1623 = **0.1713**，而全部位置是 **0.1741**。差 0.0028。L0H1：0.0639 + 0.1167 = **0.1806**，而全部位置是 **0.1772**。差 **−0.0034**。L1H3：0.0029 + 0.0207 = **0.0236**，而全部位置是 **0.0236**。差 **0.0000**。

⚠ 这条观察对“用画像表读模型”的启示

如果一个头的异常只出现在少数位置上，那它一定不会被平均值捕捉到。
统计量（均值、方差）是**对分布做压缩**。压缩总是有损的。**损失多少，取决于分布有多“不规矩”。**
一个具体的判断规则：算一个头的画像时，**顺手算一下四个量的“逐位置标准差”。**

```
per_pos = [compute_metrics(w, i) for i in range(T)]     # 每个位置单独算
for k in ("diag", "prev", "dist"):
    vals = [p[k] for p in per_pos[1:]]                  # 跳过 i=0
    print(f"{k}: 均值 {mean(vals):.3f}  标准差 {std(vals):.3f}  "
          f"最小 {min(vals):.3f}  最大 {max(vals):.3f}")
```

L0H0 的对角项会给出：均值 0.769、最小 0.115、最大 1.000。
看到“最小 0.115”和“最大 1.000”同时出现，你就知道平均值不可信了。
这一步几乎不花时间（数据都已经在手里了），**但它能挡掉本章后面 4.5 节里专门列出来的那一类错误**—— “把一个头的行为当成一个数”。

4.4.3 一个被画像表漏掉的东西：头之间的相似度

画像表给每个头四个数，**但它不告诉你两个头像不像。**
L1H0 和 L1H1 的画像很接近。**问题是：它们是在做同一件事，还是在做两件事?**
画像接近 ≠ 行为接近。 一个头可能在“看前一位的名词”，另一个在“看前一位的动词”——它们的平均前一位权重可能一样，做的事情完全不同。
要测这个，得看它们的注意力矩阵之间的相关性：

```
# 把每个头在所有位置上的注意力向量拉平，算两两的余弦相似度
flat = {k: A[k].reshape(-1) for k in heads}      # 每个头一个长向量
sim = {(a, b): F.cosine_similarity(flat[a], flat[b], dim=0)
       for a, b in combinations(heads, 2)}
```

相似度接近 1 的两个头，在每一个位置上给出的分布都一样—— 它们是冗余的，拿掉一个另一个能完全顶上。
这一节用真实数据走一遍。 把所有 8 个头的注意力矩阵拉平，两两算余弦相似度。数据来自本章的三段文本，用 `qa/` 之外的脚本 `notebooks/run_ch4_case.py` 同款加载方式测得：

头对	相似度	画像
L0H2 — L1H3	**+0.716**	两个最弥散的头
L0H2 — L0H3	+0.630	弥散 + 偏前一位
L0H1 — L1H1	+0.598	两个“前一位”头
L1H2 — L1H3	+0.586	都弥散
L1H1 — L1H3	+0.563	
L1H0 — L1H3	+0.561	
L0H1 — L0H3	+0.553	
L0H2 — L1H1	+0.541	
⋮		
L0H0 — L0H2	+0.258	**对角头 vs 弥散头**
L0H0 — L0H1	+0.181	**对角头 vs 前一位头**
L0H0 — L0H3	**+0.145**	最低的一对

这张表说了三件事。

一、最高的 +0.716 也不高。 余弦相似度 0.716 对应的“夹角”大约是 44°。**没有任何一对头是“几乎一样”的**——不存在完全冗余的两个头。

这一点和 4.4.4 节的成对消融是吻合的：如果真有两个头完全冗余，同时拿掉它们的效果应该等于分别拿掉的效果之和（零相互作用）。**实测的 28 对全部是负相互作用——没有一对是冗余的。**

二、相似度有明显的层次。 0.716 / 0.630 / 0.598 … 一路降到 0.541，然后**断崖式掉到 0.258。**

中间那道沟的位置很说明问题：沟的上方全是“弥散头之间”或“弥散头与前一位头之间”，**沟的下方全是“L0H0 与别人”。**

L0H0 是八个头里唯一一个“另一种东西”。

三、相似度矩阵把 8 个头分成了两群，但群内还有结构。 如果只按“对角 vs 非对角”分，应该是 1 vs 7。**但实际上 L0H0 和 L0H2 的 0.258 明显高于它和 L0H3 的 0.145。**

这说明 L0H0 和 L0H2 之间有一点点共同的东西—— 回看 L0H2 的画像(熵 0.804、对角 0.127)，它对角不高，**但那 0.127 是八个头里第二高的**（除了 L0H0 的 0.769）。**这 0.127 就是那 0.258 的来源。**

> **i** 停下来想一想 ⑱
>
> 上面说“没有一对头是冗余的”。**那多头机制到底在干什么?**
> 如果 8 个头各干各的，为什么不干脆用一个 8 倍宽的头?
> （答案：**因为“一个 8 倍宽的头”和“8 个头”在数学上不等价。**
> 一个宽度 $8d_h$ 的头只有**一个** softmax 分布—— 它在每个位置上只能选一种“看哪里”的方案。
> 8 个头有 **8 个独立的** softmax 分布—— **每个位置上可以同时用 8 种不同的方式分配注意力。**
> **在本章的数据里，这个区别是可见的**：同一个位置 i=10（**響**），L0H0 把 0.696 给自己，L0H1 把 0.592 给前一位，L1H3 把 0.070 给自己、0.175 给 j=6。
> **三种完全不同的分配方式，在同一个位置上同时发生。**
> **这就是第 1 章“多头为什么有用”那个问题的实测版本**—— 当时是从“子空间”的角度讲的，这里是从“注意力分配”的角度看到的同一件事。

这正是 4.4.4 节成对消融测到的东西的一个“静态版本”。 两者的区别是：**相似度是看出来的，相互作用是测出来的。 如果两者一致，说明你的解释站得住；不一致，说明还有没看到的机制。**

> **i** 停下来想一想 ⑫
>
> 上面说“画像接近 ≠ 行为接近”。**反过来呢? 两个头的行为（注意力矩阵）高度相关，它们的画像一定接近吗?**
> （答案：**一定接近。**
> 画像是注意力矩阵的四个**统计量**（熵、对角、前一位、距离）。 如果两个矩阵逐元素都几乎一样，它们的任何统计量当然也一样。
> **所以“行为接近 ⇒ 画像接近”是成立的，“画像接近 ⇒ 行为接近”不成立。** 这是一个单向蕴含。
> **这个不对称很重要**：它说明**画像表是一个“粗筛”工具**—— 画像差很远的头，行为一定差很远（可以用来快速排除）； 但画像接近的头，还需要进一步测行为才能判断。

4.4.4 熵也不是一个数：它随位置变化

画像表给每个头一个“归一化熵”。**归一化**的意思是除以 $\log(i+1)$—— **这样不同长度的位置之间才能比较**（因为可看的位置数不一样）。

但“能比较”不等于“一样”。 把三段文本按 5 个位置一组分开平均：

头	位置 0–4	5–9	10–14	15–19	20–24
L0H0	0.199	0.429	0.192	0.413	0.271
L0H1	0.729	0.469	0.511	0.439	0.625
L0H2	**0.951**	0.804	0.840	0.808	0.810

头	位置 0–4	5–9	10–14	15–19	20–24
L0H3	0.656	0.694	0.624	0.636	0.652
L1H0	0.864	0.523	0.681	0.564	0.607
L1H1	0.950	0.776	0.751	0.502	0.636
L1H2	**0.961**	0.703	0.879	0.511	0.636
L1H3	**0.984**	**0.955**	0.918	0.846	0.845

这张表比画像表有意思得多，因为它暴露了三件事。

一、位置 0–4 那一列普遍偏高，L1H3 甚至到了 0.984。

为什么？ 前几个位置几乎没有信息可用——**只有 1、2、3 个前文可看**。归一化熵除以 $\log(i+1)$ 之后，**如果模型在“没信息时”倾向于接近均匀分配，这一段就会给出接近 1 的数**。

L1H3 的 0.984 意味着：在前 5 个位置上，它基本上是均匀的。

二、L0H0 那一行是震荡的：0.199 → 0.429 → 0.192 → 0.413 → 0.271。

这个震荡不是噪声。位置 0–4 和 10–14 是低值，5–9 和 15–19 是高值。

注意这两组的分界：五言诗的标点在位置 5、11、17……附近。**低值段（0–4、10–14）落在“一句的前半”，高值段（5–9、15–19）落在“一句的后半”——而后半里包含标点位置。**

这和上一节那个“L0H0 在逗号处对角只有 0.115”完全对得上：标点位置（以及它附近）把这一段的平均熵拉高了。

一个头的画像表里没有“位置”这一维，但它可能恰恰是随位置变化的。

三、L1H3 是唯一一个单调下降的：0.984 → 0.955 → 0.918 → 0.846 → 0.845。

单调下降 = 序列越长，它越“集中”。

这有点反直觉：**位置越靠后，可看的东西越多，按理说更有挑的余地，但也更可能“挑花眼”（熵升高）。L1H3 反过来。**

一个可能的解释：位置靠后时，**位置 0（那个“垃圾桶”）的可达性变成了一个稳定选项**——每个头都可以把一部分权重给位置 0 来“减负”。**但这一章没有测这个，所以它只是解释，不是结论。**

⚠ 逐位置统计的一个陷阱：样本量不一样

上面这张表**每一格的平均样本数不同**。

位置 0–4 那一列，每个序列只有 5 个位置（其中 i=0 被跳过，剩下 4 个）。位置 20–24 也是 5 个。**看起来是对称的，但实际不是——**

因为序列长度不同。本章的 `block_size` 是 128，但三段文本最短的（空山不見人，但聞人語響。）只有 **12 个字符**。**它根本到不了位置 20–24。**

所以“位置 20–24”这一列的样本，只来自最长的那段文本。

这是“分组统计”里最常见的一个坑：分组之后，每一组的样本来源可能不一样。**如果长文本和短文本在别的方面有系统性差异，这个差异就会被读成“位置效应”。**

怎么避免？ 只统计那些**所有样本都覆盖到的位置**，或者**按序列长度分层之后再比**。

本章这张表没有做这个处理。所以上面那三条观察里，第一条（位置 0–4 偏高）是可靠的（所有序列都覆盖），**第二条和第三条要打折扣**——它们可能混入了“文本长度”的影响。

> **i** 停下来想一想 ⑲
>
> 上面说“位置 20–24 只来自最长的那段文本”。
> **本章的 `head_profile` 函数（4.3 节开头那份代码）把三段文本的所有位置都扔进了一个池子，然后算平均。**
> **这个做法有同样的毛病吗？**
> （答案：**有，而且更隐蔽。**
> `head_profile` 算的是“所有位置的平均”，**不做位置分组。** 看起来这样就绕开了“分组样本不均”的问题——**其实没有。**
> **因为长文本对平均值的贡献更大。**一段 128 字符的文本贡献 128 个位置，一段 12 字符的文本只贡献 12 个。**前者的话语权是后者的 10 倍以上。**
> **所以画像表里的“平均距离 1.20”这类数字，实际上是“以长文本为主”的平均。**
> **更根本的问题是**：平均距离天然随序列长度增长（可看的距离变远），**所以把一个 12 字符文本的平均距离和一个 128 字符文本的平均距离放在一起平均，本身就是把两个不可比的量相加。**
> **严谨的做法**：把距离**归一化**（除以位置 i），或者**只统计固定长度的前缀。**
> **本章没有做。**这不影响“L0H0 对角高、L1H3 熵高”这类结论——**因为那个差距（0.769 vs 0.165）远大于这个偏差的量级**——但**画像表里那些小数点后两位的数字，不该被当成精确值。**）

4.5 因果检验：把它拿掉试试

4.5.1 为什么“看图说话”不够

注意力图给你的是**相关**：位置 i 从位置 j 取了信息。你要的结论是**因果**：**这条信息对模型的输出起了作用。**

这两件事在一般情况下不等价，而在神经网络里尤其容易分开，原因有三个。

原因一：冗余。同一个信息可以从多条路走。第 2 章讲过残差流——每个位置的信息本来就沿着残差流一路传到最后一层。注意力头只是**额外**开了一条路。**如果原来的路已经够用，去掉这条新的路，输出几乎不变。**

原因二：下游会重算。后面的层可以重新算一遍前面已经算过的东西。第 0 层的头输出被抹掉，第 1 层的头仍然能从残差流里拿到原始信息。

原因三：v_j 可能是零。4.1 节讲过的极端情况——权重很高，但取过来的东西是零向量。

三个原因指向同一个实验：把这条路由切断，看输出变不变。变，说明它在起作用；不变，说明它（在当前输入分布下）是冗余的。

> **i** 停下来想一想 ②
>
> 上面的“原因三”说：v_j 可能是零向量，所以权重高不等于有作用。
> **但如果某个头对所有位置算出的 v_j 都接近零，那它的输出就接近零。这样的话，把它拿掉显然没影响——这个头是不是“没用”？**
> （答案：**“没用”要分两问。**
> 第一问：**这个头对当前这个模型没用吗？**是的，拿掉没影响。
> 第二问：**这个头是训练出来的、还是碰巧变成这样的？**这才是关键。
> 一个恒等输出零的头，可能是训练把它推到了这个状态（那它是在“主动不做事”），也可能是初始化后就没动过。**区分方法：看它在训练过程中怎么演化。**
> 第 2 章讲过“死神经元”——前馈网络里有些单元从不激活。注意力头也有类似现象，原因不同但表现一样。**“拿掉没影响”只说明冗余，不说明它是多余的。**）

4.5.2 怎么做：屏蔽一个头

屏蔽一个头（head ablation）听上去简单：把它的注意力权重置零不就行了？

不行。如果只把 A 的某一行置零，那一行加起来不等于 1 了，softmax 的归一化被破坏，下游的数全乱。

正确做法是**在输出投影之前，把这个头占的那一段特征切片置零**：

```
def ablate_head(model, layer, head):
    d_head = model.blocks[layer].attn.d_head

    def hook(mod, inp, out):
        # proj 的输入是 (B, T, C) —— 头维度已经被 transpose+view 拼回特征维了。
        # 所以不能写 y[:, :, head, :]（那是 4 维时的写法），
        # 要把该头对应的那一段特征切片置零。
        y = (out[0] if isinstance(out, tuple) else out).clone()
        y[:, :, head * d_head:(head + 1) * d_head] = 0.0
        return (y,) + out[1:] if isinstance(out, tuple) else y
    return model.blocks[layer].attn.proj.register_forward_hook(hook)
```

注意注释里那两行。 这个 hook 的第一版我们写成了 `y[:, :, head, :] = 0.0`——那是 4 维张量（`(B, h, T, d_head)`）的写法。但 `proj` 拿到的已经是 `(B, T, C)`，因为注意力在返回之前做了 `transpose(1, 2).contiguous().view(B, T, C)`。

结果是 `IndexError: too many indices for tensor of dimension 3`。

这个错误为什么值得单独讲

它和我们第 1 章、第 3 章踩过的坑是同一类：**形状对不上**。

但这次有一个额外的细节：**注意力头在代码里“消失”过一次**。

在 `forward` 里，头的维度是这样走的：

```
(B, T, C)  →  qkv 投影  →  (B, T, 3C)
           →  view+transpose  →  (B, h, T, d_head)      ← 头是独立的维度
           →  注意力计算  →  (B, h, T, d_head)
           →  transpose+view  →  (B, T, C)              ← 头被拼回特征维
           →  proj  →  (B, T, C)
```

头只在中段是一个独立维度。 到了输出投影这一步，它已经变成了 特征维里的一段连续区间——第 `head` 个头占的是 `[head*d_head, (head+1)*d_head)`。

所以“屏蔽一个头”在代码层面有两个正确的时机：在中段（4 维时按索引置零），或者在 proj 之前（3 维时按切片置零）。选哪个都行，**但必须知道自己站在哪一段**。

对照第 3 章那个 `kv_cache` 的 `dim=-2`——同一类问题的另一次出现。

4.5.3 实测：八个头的因果重要性

```
基线 loss（验证集） = 4.1489

去掉 L0H0   loss 4.3230   Δ +0.1741
去掉 L0H1   loss 4.3261   Δ +0.1772
去掉 L0H2   loss 4.2903   Δ +0.1414
去掉 L0H3   loss 4.2955   Δ +0.1466
去掉 L1H0   loss 4.1790   Δ +0.0301
去掉 L1H1   loss 4.1717   Δ +0.0227
去掉 L1H2   loss 4.1769   Δ +0.0279
去掉 L1H3   loss 4.1725   Δ +0.0236
```

把两层的均值算出来：

层	四个头的 Δ	均值
第 0 层	+0.1741, +0.1772, +0.1414, +0.1466	**+0.1598**
第 1 层	+0.0301, +0.0227, +0.0279, +0.0236	**+0.0261**

第 0 层的头比第 1 层的头重要 6.1 倍。

这个结果和“看图说话”的结论对不上

回到 4.3 节那张画像表。如果按“画像有多清晰”排序，最该重要的应该是 L0H1（前一位 0.450，pattern 最明确）和 L0H0（对角 0.769，最极端的偏好）。

实测：它们确实是第 0 层里 Δ 最大的两个（+0.1772 和 +0.1741）。 这一半对上了。

但对不上的是另一半：L1H3 的画像最“没有模式”（熵 0.871，接近均匀），看起来最该无关紧要——可它拿掉之后 Δ = +0.0236，**比同层画像清晰得多的 L1H0（对角 0.276、前一位 0.182）的 +0.0301 只小一点点**。

同一层内部，四个头的 Δ 挤在 0.0227 到 0.0301 这个窄区间里，**和它们的画像清晰度没有明显关系**。

层与层之间的差别（6 倍）远大于层内头与头之间的差别（1.3 倍）。

这是本章第一个真正的发现：在这个模型里，“哪个头重要”主要取决于它在第几层，而不是它长什么样。

⚠ 一个不能过度解读的地方

上面说“第 0 层比第 1 层重要 6 倍”。这个结论**只在当前这个模型、这个任务、这个数据分布上成立**。

三个限制：

1. **模型只有 2 层**。“第 0 层”和“第 1 层”在这个模型里的角色是完全不同的：第 0 层直接处理词嵌入，第 1 层处理已经混合过的表示。**在 12 层、24 层的模型上，这个 6 倍的结论没有任何理由继续成立**。
2. **loss 是一个很粗的指标**。Δ = +0.03 在 loss 上很小，但它可能对应着生成行为的明显变化——只是平均 loss 把它抹平了。
3. **消融是“一次一个”**。同时拿掉两个头，效果可能**大于**两次单独之和（相互作用），也可能小于（互相替代）。**单头消融测不出这件事**，见 4.9 节。

这三条限制不是免责声明，是结论的一部分。

4.5.4 Δ 只有 0.03，那是什么样的行为变化？

loss 的绝对值不好感知。**直接看生成结果**。基线文本是：

```
春江潮水海平。不見昔年年不見，不見春風吹。不見君不得
```

去掉 L1H3（Δ 最小之一，+0.0236）之后：

```
春江潮水海平。不見昔年年不見，不見春風吹
```

只少了 6 个字。前 20 个 token 一模一样，分叉发生在第 21 个 token 上——而且分叉之后，输出只是**停在了同一个循环里**（还记得第 3 章那个「不見」循环吗）。

这说明了 loss 指标的一个特点

Δ = +0.0236 对应 6 个 token 的分叉。看起来“很小”。

但注意：**这 6 个 token 的分叉，是因为模型走进了两个都“说得通”的循环**。它本来就退化（第 3 章），所以“换个循环”对 loss 的影响很小。

在一个没有退化的模型上，同样的 Δ 可能对应完全不同的行为。

这又回到 4.4.3 节那条限制：**loss 是平均量，它对“换了一条同样差的路”不敏感**。要测行为，就得真的跑生成，看输出。

💡 一个更细的粒度：不要只看 loss

上面的消融用的是平均 loss。它有两个缺点：抹平局部效应、对退化不敏感。

更有信息量的做法是看逐位置的损失变化：

```
# 分别记下基线和消融后每个位置的 loss，看差异集中在哪儿
delta_per_pos = (loss_ablated - loss_base)      # 形状 (T,)
# 如果差异集中在标点位置 → 这个头在管句法
# 如果差异均匀分布 → 这个头在管全局的字符分布
# 如果只在少数位置有大尖峰 → 这个头只在特定上下文里起作用
```

“平均 Δ 很小”和“在某些位置 Δ 很大、在别处是零”是两种完全不同的机制，但平均之后看不出区别。

4.5.5 那 6 倍是从哪来的：输出范数

4.4 节留了一个问题没有回答：**为什么第 0 层的头比第 1 层的头重要 6 倍?**

有两个可能的解释：

解释 A：第 0 层在做更基础的工作。 它直接面对词嵌入，算的是“相邻字符的共现”这种底层统计量；第 1 层在已经混合过的表示上做事，可替代性更高。

解释 B：第 0 层的头输出本来就更大。 拿掉一个输出量大的东西，影响自然大——这和它“做的工作是否基础”无关，纯粹是个尺度问题。

这两个解释指向不同的结论（A 是机制性的，B 是平凡的），**而它们可以被一个测量分开**：量每个头输出的 L2 范数。

```
头      ‖out‖        头      ‖out‖
L0H0    6.1320       L1H0    3.3807
L0H1    5.4026       L1H1    3.3128
L0H2    5.9004       L1H2    3.4974
L0H3    5.5450       L1H3    3.4428
──────────────────────────────────────
第 0 层均值  5.745      第 1 层均值  3.408
```

第 0 层的输出范数是第 1 层的 1.69 倍。

4.6 1.69 倍 vs 6.1 倍：两个解释都占一部分

把两组数放在一起：

量	第 0 层 / 第 1 层
消融 Δ 均值	**6.1 倍**
输出范数均值	**1.69 倍**

范数差解释不了 6.1 倍。 如果消融的效果纯粹是尺度问题，那两者应该差不多。

所以解释 A 也成立：第 0 层不只输出更大，**它输出的东西还更“不可替代”**。

粗略地看：如果损失对那个头的依赖是“超线性”的（也就是范数大一倍，作用大不止一倍），那 $6.1 / 1.69 \approx 3.6$ 就是那个超线性的指数。**这是一个观察，不是一个定律**——它能成立是因为这个模型恰好这么小、这么浅。**但它至少排除了“纯尺度解释”。**

> **i** 停下来想一想 ⑦
>
> 上面说“没有死头”——最小的输出范数是 3.31，不接近零。
>
> **但 3.31 这个数本身有意义吗?** 怎么知道 3.31 算“小”还是“大”?
>
> （答案：**要和残差流的尺度比。**
>
> 第 2 章讲过残差流：$x \leftarrow x + \mathrm{Attn}(\cdot)$。如果一个头的输出范数是 3.31，而残差流本身的范数是 30，那这个头只贡献了 10%——它的影响会被残差流的“底噪”淹没。
>
> **正确做法是把头的输出范数除以残差流的范数**，得到一个“相对贡献”。这个比值才是“这个头有多大声”的度量。
>
> 本章没有做这个归一化——**这是画像表的一个已知缺口。** 4.3 节末尾提到过类似的问题：**绝对量必须配基线才能读。**）

4.6.1 成对消融：头与头之间的相互作用

单头消融有一个根本局限：**它测不出头与头之间的关系。**

考虑一个极端：头 A 和头 B 在做同一件事。单独拿掉任何一个，另一个顶上，Δ 都很小。看起来两个都不重要。但同时拿掉两个，模型可能直接崩掉。

测法很简单：**对每一组 (i, j)，比较**

$$\Delta_{ij} \quad \text{vs} \quad \Delta_i + \Delta_j \tag{4.11}$$

所以这个式子在说：如果两个头互不影响，一起拿掉的效果应该等于分别拿掉的效果之和。**差值就是“相互作用”。**

本模型 8 个头，$\binom{8}{2} = 28$ 组，全跑一遍。
结果：28 组的相互作用全部是负的。

组合	实际 Δ	线性相加	相互作用
L0H0 + L1H0	+0.1732	+0.2041	**−0.0309**
L0H2 + L1H2	+0.1537	+0.1693	−0.0156
L0H1 + L1H3	+0.1876	+0.2007	−0.0131
L0H0 + L1H1	+0.1856	+0.1968	−0.0112
L0H1 + L1H1	+0.1893	+0.1999	−0.0106

4.7 “全部为负”意味着什么

负的相互作用 = **实际损失小于线性相加 = 两个头之间存在冗余**。
这在直觉上完全合理：头之间共享残差流，一条路断了信息还能从别处走。**但方向一致（28 组全负）比单个数值更有说服力**—— 它说明这不是偶发，而是这个架构的系统性性质。
最值得看的是第一行：L0H0 + L1H0。
单独拿掉 L0H0，Δ = +0.1741。 单独拿掉 L1H0，Δ = +0.0301。 **线性相加应该是 +0.2041**。
实测：**+0.1732**。
也就是说——**同时拿掉两个，损失比只拿掉 L0H0 还小一点点（+0.1732 vs +0.1741）**。
结论：L1H0 在 L0H0 被拿掉之后，变得完全无用。
这暗示了一个**依赖关系**：L1H0 是在消费 L0H0 产出或搬运的东西。 L0H0 一断，L1H0 就没有原料了，它在不在都一样。
单头消融完全看不出这件事。 单看数据，L1H0 的 Δ = +0.0301， 你会以为它是一个“有点作用的普通头”。**成对消融才暴露出它是下游**。

> ⚠ 一个不能过度解读的地方
>
> 上面的“L1H0 依赖 L0H0”是一个**从消融数据推出来的假设**， 不是被直接观测到的机制。
> **能直接观测到的是**：$\Delta_{00,10} \approx \Delta_{00}$。
> **这个现象还有别的解释**：
>
> 1. **饱和**：损失有上限（最差也就是均匀分布，5.67）， L0H0 已经把它推到接近上限，再拿掉 L1H0 也加不了多少
> 2. **L1H0 在做冗余的事**，而这件事的另一个提供者恰好也依赖 L0H0
> 3. **纯粹的数值巧合**（+0.1732 和 +0.1741 差 0.0009， 而单次评估的噪声可能有这个量级）
>
> **要区分这几种，需要看逐位置的数据、看激活的相关性、或者做一个更细的干预实验**。本章只做到“指出这个现象”这一步。
> **把“观测到什么”和“推断出什么”分开写，是这一章反复在做的事**。

4.7.1 逐位置看：平均值掩盖了什么

最后的这一组数据最值得琢磨。
单头消融报的是“Δ = +0.0236”这样的一个数。**这个数是怎么来的？是在所有位置上取的平均**。
把平均拆开，看**每个位置**的 Δ：

```
位置        0      1      2      3      4      5      6      7      8      9     10     11
Δ(L1H3)  -0.006 -0.020 +0.026 +0.038 +0.012 +0.017 +0.028 -0.001 -0.005 -0.007 +0.035
+0.031
```

均值 **+0.0236**，最大 **+0.0643**，最小 **−0.0198**， **位置间的标准差 0.0175**。

4.8 标准差和均值是一个量级

均值 +0.0236，标准差 0.0175。 两个数差不多大。

这在统计上意味着：**这个“平均效应”的可靠性很低。** 不同位置的 Δ 从 −0.02 到 +0.06，横跨了四倍的均值范围。

更值得注意的是有负值。 在位置 1（Δ = −0.0198）上，**拿掉 L1H3 反而让模型的损失变小了。**

这不是测量误差——这是说：**在这个特定的上下文里，L1H3 的输出是有害的。** 它把模型往错误的方向推了一点，拿掉它反而更好。

一个“平均有用”的头，在某些位置上是有害的。 “这个头重不重要”这个问题，**在位置这个维度上根本没有单一答案。**

最后一行数据：只有 9% 的位置的 Δ 超过均值的两倍。**说明绝大多数位置上，这个头的影响可以忽略**—— 平均出来的 +0.0236 是被少数位置拉上去的。

这个观察对消融实验的启示

报告一个平均 Δ 是不够的。 至少应该同时给出：

- **Δ 的分布**（不只是均值）
- **负值的比例**（有多少位置拿掉反而更好）
- **集中的程度**（效应是均匀的，还是集中在少数位置）

这三条能区分出完全不同的机制：

分布形态	可能的机制
均匀的小正值	这个头在稳定地提供一点边际信息
少数位置大正值，其余接近 0	这个头是**条件触发**的（只在特定上下文工作）
正负混杂	这个头在**多个方向上拉扯**，净效应是正值

本章的 L1H3 属于第三种。

4.8.1 消融实验的统计学：Δ 要多大才算数

4.4 节报了 Δ 值，但没有说**这些 Δ 的噪声有多大。** 这是一个必须回答的问题——**否则你不知道 0.0227 和 0.0301 的差别是真的还是噪声。**

测法：重复评估。

验证集 loss 是用随机采样的 20 个 batch 估出来的。每次换一个随机种子， 估计值就会变。**把同一组权重评估 10 次，看这个估计值波动多少。**

```
def eval_loss_repeated(model, val, cfg, n_rep=10, n_batch=20):
    vals = []
    for seed in range(n_rep):
        gen = torch.Generator(device=DEV).manual_seed(seed * 1000)
        vals.append(eval_loss(model, val, cfg, n_batch=n_batch, gen=gen))
    return {"mean": sum(vals)/len(vals),
            "std": (sum((v - sum(vals)/len(vals))**2 for v in vals)/len(vals))**0.5,
            "min": min(vals), "max": max(vals)}
```

4.9 那么两个 Δ 的差别什么时候可信

判断规则：两个 Δ 的差，要大于两者的合并标准差，才算“有差别”。

设单次评估的标准差是 σ。要区分 Δ_a 和 Δ_b， 需要

$$|\Delta_a - \Delta_b| \gtrsim 2\sqrt{2}\,\sigma \tag{4.12}$$

所以这个式子在说：两个估计值的差，要大于它们各自不确定度的 $\sqrt{2}$ 倍的 2 倍。

这正是本章一个已知的不足：我们报了 8 个头 × 严格的固定种子评估（每次都用同样的随机种子生成同样的 batch）， **所以那个“位置间的标准差”不是评估噪声，是真实的效应差异。**

但不同头之间的 Δ 比较（比如 L1H0 的 +0.0301 和 L1H1 的 +0.0227）， 在不知道 σ 的情况下**不能断言“L1H0 更重要”**—— 差 0.0074 完全可能是评估噪声。

能断言的是：第 0 层（均值 0.1598）和第 1 层（均值 0.0261）的差别—— **6 倍，远超任何合理的噪声水平。**

> **停下来想一想 ⑬**
>
> 上面说“不同头之间的 Δ 差别 0.0074 不能断言”。
> **但本章 4.4.4 节又断言了“L1H0 依赖 L0H0”**，依据是 $\Delta_{00,10} = 0.1732 \approx \Delta_{00} = 0.1741$（差 0.0009）。
> **这不是自相矛盾吗?**
> （答案：**不是，因为两个断言的逻辑不同**。
> “L1H0 比 L1H1 更重要”是一个**比较断言**—— 它需要证明两个数的差是真的。
> “L1H0 依赖 L0H0”是一个**等价断言**—— 它说的是“加进 L1H0 的消融之后，Δ 几乎没有变化”。
> **等价断言的证据强度更高**：即使有噪声，“从 0.1741 到 0.1732”这个变化（0.0009）也**远小于**“从 0.1741 到 0.2041”（如果两者独立，应该是这个值）。
> **换句话说**：我们不是在比较两个测量值，而是在比较“实际值”和“一个理论预测值”。**后者的差别（0.0309）够大，可以下结论**。

4.9.1 一个被反复用到的方法：随机种子固定

本章所有的消融数字都用了**固定的随机种子**来生成评估 batch:

```
gen = torch.Generator(device=DEV).manual_seed(0)      # 固定
```

为什么? 因为这样**基线 loss 和消融后的 loss 是在同一批数据上算的**。两者的差就**不含“batch 不同”这个噪声源**。

4.10 固定种子的代价

好处：Δ 更干净。同一批数据，只有模型变了。
代价：σ 测不出来——因为你永远用同一批数据，**同一批数据上的差是确定的，波动是 0**。
所以“固定种子”和“测噪声”是两件互斥的事：- 要干净的 Δ → 固定种子 - 要知道 Δ 的可信区间 → 换种子
正确做法是两个都做：固定种子报 Δ，换种子报 σ。本章只做了前者，**这是 4.10 节那张结论强度表里“缺对照”那一栏的一个具体内容**。
这类“做了一个正确的选择，但只做了一半”的情况，在实验设计里非常常见。

4.10.1 头之间到底差多少：一个更细的看法

把 4.4 节的消融 Δ 和 4.8 节要讲的评估噪声放在一起看，可以得到一个更保守但更可靠的结论:

比较	差别	能不能断言
第 0 层 vs 第 1 层	**6.1 倍**	**能**
同层内 L0H0 vs L0H3	0.1741 vs 0.1466（19%）	**存疑**
同层内 L1H0 vs L1H1	0.0301 vs 0.0227（33%）	**存疑**

能断言的只有跨层那一句。同层内部的排序，在没有噪声估计的情况下不能说。
而这句话本身已经足够有用了：“在这个模型里，第 0 层的注意力头比第 1 层重要得多”——**这是一个可复现、有量级差别、不依赖细微比较的结论**。

> **宁要一个粗糙但可靠的结论**
>
> 这是本书反复出现的取舍：**结论的精度和可靠性是两回事**。
>
> · “第 0 层比第 1 层重要 6 倍”——**粗糙，但可靠**
> · “L0H0 比 L1H1 重要 7.7 倍”——**精确，但不可靠**（可能是噪声）
>
> **写论文的时候后者更吸引人，做工程的时候前者更有用**。
> **本书的选择是：报可靠的，把不可靠的标出来当“待办”**。

4.11 对照实验：随机模型长什么样

4.11.1 一个必须先问的问题

4.3 节那张画像表里，八个头的熵从 0.294 到 0.871，跨度很大；对角从 0.090 到 0.769，跨度更大。

这些数字说明模型学到了东西吗？

要回答这个，必须有一个**对照：随机初始化的同结构模型，画像是什么样的？**

如果不训练的模型也能给出类似范围的数字，那“画像”就没有信息量—— 你在一张随机噪声里也能解读出故事。

4.11.2 实测：随机的头几乎一模一样

用同样的结构（2 层 4 头 128 维），只把权重随机初始化，跑同样的画像：

```
seed1000 L0H0  熵 0.958  对角 0.158  前一位 0.115   弥散（接近均匀）
seed1000 L0H1  熵 0.958  对角 0.157  前一位 0.116   弥散（接近均匀）
seed1000 L0H2  熵 0.958  对角 0.157  前一位 0.115   弥散（接近均匀）
seed1000 L0H3  熵 0.958  对角 0.158  前一位 0.115   弥散（接近均匀）
seed1000 L1H0  熵 0.958  对角 0.156  前一位 0.116   弥散（接近均匀）
seed1000 L1H1  熵 0.958  对角 0.158  前一位 0.115   弥散（接近均匀）
seed1000 L1H2  熵 0.958  对角 0.158  前一位 0.116   弥散（接近均匀）
seed1000 L1H3  熵 0.958  对角 0.158  前一位 0.115   弥散（接近均匀）
```

八个头的数字几乎完全一样，熵全部是 0.9578–0.9579，对角全部是 0.156–0.158。换成别的随机种子，结论一样。

这条对照线的价值

把训练后的和随机的并排放：

量	训练后	随机	差别
熵的范围	0.294 – 0.871	0.9578 – 0.9579	**训练把方差放大了几百倍**
对角的范围	0.090 – 0.769	0.156 – 0.158	同上

训练前，八个头是一个模子刻出来的；训练后，它们分化成了完全不同的角色。

这就是**“注意力头分化”（head specialization）**的直接证据。**在没有这条对照线之前，你看到 L0H0 的对角 0.769，不知道该不该惊讶。有了它，才知道 0.769 是“训练特意造出来的”。**

> **i** 停下来想一想 ③
>
> 随机模型的对角是 **0.157**。如果注意力完全均匀，位置 i 的对角应该是 $1/(i+1)$。
>
> 拿 i 从 0 到 23 平均一下，$1/(i+1)$ 的均值大约是 0.15 —— **和实测的 0.157 几乎一致。**
>
> **所以随机模型的对角 0.157 一点都不神秘：它就是“均匀注意力的几何平均值”。**
>
> 那么：**如果一个人的画像表里只有训练后的数字，没有这条对照线，他看到“对角 0.157”会不会以为这是一个“盯住自己”的头？**
>
> （答案：很可能会。0.157 放在 4.3 节那张表里，比 L1H2 的 0.147 还高—— 按那张表的标准，它会被判成“兼顾自己与前一位”。
>
> **但它其实是纯粹的均匀分布。**
>
> **这就是为什么对照实验不是可选项。没有它，任何“阈值判读”都在拿随机噪声当信号。**）

⚠ 但随机模型也不是完全“无结构”

有一件事要澄清：随机模型的八个头数字一样，**不代表随机注意力本身是均匀的**。
因果掩码决定了位置 i 只能看 $i+1$ 个位置——**这个几何约束在随机权重下依然存在**。具体说：

- 位置 0 只能看自己，对角**必然是 1.0**
- 位置 1 能看两个位置，随机的对角期望是 0.5
- 位置 23 能看 24 个位置，随机的对角期望是 $1/24 \approx 0.04$

平均下来就是 0.157。**这条曲线完全由掩码的形状决定，和学到了什么无关。**
所以“平均距离 4.47”这类量，必须先减掉几何基线才能比较。我们在 4.3 节给平均距离配了“完全均匀时 ≈ 6”这条参考线，就是这个原因。

4.11.3 把 0.157 算出来：几何基线的解析形式

4.5 节说随机模型的对角是 0.157，并且指出它“约等于 $1/(i+1)$ 的均值”。这一节把这个数**推导一遍**——因为它揭示了一条容易忽略的原则。
前提：随机初始化的模型，注意力权重在因果掩码允许的范围内近似均匀。位置 i 允许看 $0, 1, \ldots, i$ 共 $i+1$ 个位置，每个权重约 $1/(i+1)$。
对角的期望：

$$\mathbb{E}[A_{ii}] = \frac{1}{i+1} \tag{4.13}$$

对 $i = 0, 1, \ldots, T-1$ 取平均：

$$\overline{A_{ii}} = \frac{1}{T}\sum_{i=0}^{T-1}\frac{1}{i+1} = \frac{H_T}{T} \tag{4.14}$$

其中 $H_T = 1 + \frac{1}{2} + \ldots + \frac{1}{T}$ 是第 T 个**调和数**。

4.12 用我们自己的数据验算

本章测的三条诗各 24 个字，$T = 24$。

$$H_{24} = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{24} \approx 3.776 \tag{4.15}$$

$$\overline{A_{ii}} \approx \frac{3.776}{24} = 0.1573 \tag{4.16}$$

实测：0.1578、0.1561、0.1571（三个随机种子）。
误差在 0.2% 以内。
这不是巧合——**这条基线完全由序列长度决定，和模型学到了什么无关。**

⚠ 这条基线的一个陷阱

看式 式 (4.14)：**基线随 T 变化**。

T	H_T	基线 $\overline{A_{ii}}$
8	2.718	0.340
24	3.776	0.157
64	4.744	0.074
128	5.407	0.042

同一个头，在不同的序列长度上测，对角权重会差好几倍——而它什么都没变。

这意味着两件事：

第一，跨序列长度比较对角是没有意义的。你在 $T = 8$ 上测到一个头对角 0.34，在 $T = 64$ 上测到 0.08，**不能得出“这个头在长序列上不那么关注自己了”**——它可能只是**几何基线变了**。

第二，本章的画像表是在 24 字的诗上测的。表里的“对角 0.769”（L0H0）要对照的是 0.157 这条线，不是 0。

同一个测量，在不同的 T 下，含义完全不同。

ℹ 停下来想一想 ⑧

上面这张表说明基线随 T 下降得很快（T 从 8 到 128，基线降了 8 倍）。

那么：如果我想比较“两个头谁更关注自己”，但它们的测试序列长度不同，该怎么办？

（答案：**做归一化**。

$$\text{相对对角} = \frac{A_{ii}}{1/(i+1)} \tag{4.17}$$

这个量在 1 附近表示“和均匀一样”，大于 1 表示“比均匀更关注自己”。**它对 T 不敏感**，因为分母已经把几何效应除掉了。

本章的画像表**没有做这个归一化**——它报的是绝对对角。这是一个已知的不足：**表里的对角值只能在同一个 T 下横向比较**。

4.3 节末尾提过这件事（“绝对量必须配基线才能读”），这里是它的定量版本。）

4.12.1 随机的头为什么“一模一样”

随机模型的八个头，熵全部是 0.9578–0.9579，跨度 0.0001。

为什么这么齐？

因为**随机初始化的模型，注意力权重只由几何决定，不由内容决定**。

回到打分公式：$A \propto \exp(q \cdot k/\sqrt{d})$。随机权重下，$q$ 和 k 都是小随机向量（初始化 $\mathcal{N}(0, 0.02^2)$），内积很小，指数几乎不改变均匀分布。

所以每个头、每个位置，注意力都近似均匀——而均匀是同一个分布。八个头测出来的自然就是同一组数。

4.13 这个“齐”本身就是一条证据

反过来想：**如果训练没有起作用，八个头应该一直保持这种“齐”。**

实测：训练后熵的范围是 0.294–0.871，**跨度 0.577**。随机时跨度 0.0001。**训练把方差放大了约 5000 倍**。

所以“训练造出了结构”这件事，不需要看图、不需要讲机制，一个方差对比就够了。

这是本章最硬的一条结论——**它不依赖任何对“这个头在做什么”的解读。**

一个可以直接抄的对照设计

任何“我在模型里发现了 X”的声称，都可以配一个这样的对照：

步骤	做什么
1	用**随机初始化**的同结构模型跑同一个测量
2	记录随机模型给出什么值（这是“零假设水平”）
3	用**训练后**的模型跑同样的测量
4	只有当训练模型的测量值**明显超出随机模型的分布**时，才谈得上“发现了什么”

第 2 步最容易被跳过，因为它看起来是浪费时间—— “一个没训练的模型能有什么好看的?”
但随机模型给出的不是“没有结果”，而是“零假设的分布”。没有这个分布，你测出来的任何数字都不知道该怎么读。

4.14 归纳头：一个能被证明的电路

4.14.1 什么是归纳头

前面几节讲的是“怎么判断一个头有没有用”。这一节讲一个**已经被证明有用的具体电路**。
归纳头（induction head）是 2022 年在一项机制可解释性研究里被发现并证明的。它的行为可以用一句话描述：

如果当前这个 token 之前出现过，就去看“它上一次出现之后跟的是什么”，然后预测那个。

用序列说清楚。假设输入是：

```
... A B C D ... A
                ↑ 现在在这里
```

当前 token 是 `A`。它上一次出现在前面某个位置。**归纳头会把注意力放到“上一次 A 后面的那一个位置”上——也就是 `B` 所在的位置**。然后它从 `B` 那里把信息取过来，供后续层预测“下一个是 B”。
这是一个“复制”电路：它让模型能够续写重复出现过的模式。在自然语言里，这对应着“前面出现过类似的片段，接下来大概会重复它”。
归纳头为什么需要两层
一个头做不到这件事。它需要**两个头接力**：
第一层的“前一位头”：把每个位置的“上一个 token 是什么”写进残差流。（我们模型的 L0H1 正是这种头——前一位权重 0.450。）
第二层的“匹配头”：拿当前位置的 token 去和残差流里每个位置的“前一个 token”比对，找到相等的那一个，然后关注它的下一位。
关键在第二步：要在位置 i 找到“token 等于当前 token”的历史位置，模型需要一个**匹配**操作。而匹配可以用内积实现—— 如果残差流里存的是“前一个 token 的表示”，那么“当前 token 的表示”和它的内积，就正比于两个 token 的相似度。
这就是归纳头：它把“找相同”变成了“算内积”，而内积正是注意力本来就在做的事。
注意它必须跨两层。第一层负责“造出可匹配的键”，第二层负责“匹配 + 复制”。一层做不到，因为同一层的注意力 只能看到原始输入，看不到“前一个 token 是什么”这个派生量。

4.14.2 归纳头的机制：为什么必须两层，以及它怎么算

4.6 节说归纳头“必须跨两层”。这一节把这个论断**推一遍**—— 因为它示范了一件事：**机制可解释性可以给出“不可能”的结论，不只是“观察到了”**。
目标：在位置 i，如果 token x_i 之前出现过，就输出 “上次出现之后跟的那个 token”。

用符号写：设 $j < i$ 是满足 $x_j = x_i$ 的**最近**位置， 我们要让模型在位置 i 预测 x_{j+1}。

要算这个，需要两步：

第一步：在位置 j 处，记下“前一个 token 是什么”。 也就是把 x_{j-1} 这个信息，编码进位置 j 的表示里，供后面匹配用。

第二步：在位置 i 处，找到那个“前一个 token 等于 x_{i-1}”的历史位置。

4.15 为什么第二步是“匹配前一个”而不是“匹配当前的”

这一步容易绕晕，用具体序列走一遍。

```
位置:   0    1    2    3    4    5    6    7    8
token:  A    B    C    D    A    B    C    D    A
                                  ↑
                          现在在这里（位置 4，token = A）
```

我们在位置 4，token 是 A。上一次 A 出现在位置 0。

要预测的下一个 token 是什么？ 由位置 0 的经验：A 后面跟的是 B（位置 1）。**所以我们想输出 B。**

怎么找到 B？ 我们不能直接找 B——不知道它是谁。 我们只知道“上一次 A 后面的那个”。

所以匹配的键不能是 A 本身。 如果拿 A 去和历史的 A 匹配， 找到位置 0，然后看位置 0 的**输出**——但位置 0 的输出是它自己那一层算出来的东西，不含“后面跟什么”的信息（因果掩码不让它看未来）。

关键的转换：改成匹配“前一个 token”。

在位置 4，前一个 token 是 D（位置 3）。在历史里找“前一个 token 也是 D”的位置——**位置 3。**

然后看向位置 3 的后面一个位置，也就是位置 4。 ……这在原地打转。

换个写法：要找的是“位置 $j+1$ 的值”，其中 j 是上次 A 的位置。而位置 $j+1 = 1$ 的特征是“它的前一个 token 是 A”。

所以在位置 i，要匹配的键是“前一个 token 是 x_i”，而不是“自己是 x_i”。

用 $i = 8$（第二个 A，位置 8）再走一遍：

- 位置 8 的 token 是 A（它前面位置 7 是 A，位置 6 是 D……这是第三遍开头）
- 要找“前一个 token 是 A”的位置 → 位置 1（它前面位置 0 是 A）和位置 5
- 取最近的一个 → 位置 5
- **看向位置 5 的后面，也就是位置 6** —— 那里是 C
- **输出 C** √（第一遍里 A 后面是 B，但第三遍的上下文已经变成了 B C D A B C D A， 所以最近一次 A 后面跟的是 C——**这正是归纳头“学上下文里的重复”的含义**）

归纳头匹配的是“前一个 token”，然后指向“匹配位置的下一个”。

现在看两层的分工：

第 0 层的“前一位头”（我们模型的 L0H1，前一位权重 0.450）：它把位置 t 的输出写成“x_{t-1} 的表示”。 **这一个头就完成了第一步**——每个位置的表示里，现在带着“我前面是谁”。

第 1 层的“匹配头”： 在位置 i，它的查询是 x_i 的表示（经过第 0 层之后）。它的键是历史位置的表示，而那些表示里含有“前一个 token”。 **内积 $q_i \cdot k_j$ 就在问：“位置 j 的前一个 token，和我（位置 i 的 token）像不像?”**

像 → 权重大 → 从位置 j 取值 → 值里编码着“位置 j 的 token” → 输出它。

> **i** 停下来想一想 ⑨
>
> 上面的推导里，第 1 层的匹配头需要“位置 j 的值里编码着位置 j 的 token”。**这个信息从哪来?**
>
> （答案：**残差流**。
>
> 第 2 章讲过：残差流是“每个位置的一条直通管道”， 词嵌入 $e(x_j)$ 从第 0 层一直加到最后。**所以位置 j 的值向量里天然含有 $e(x_j)$**， 匹配头只要用一个合适的 W_V 把它取出来就行。
>
> **这是残差流架构的一个直接红利**：原始输入在每个位置都有一份副本，任何一层都能随时取用。**不需要专门“保存”什么。**
>
> **一个反例**：如果没有残差流（纯串行的 RNN 式结构）， 中间层的表示会被后面的计算覆盖掉，匹配头就拿不到“位置 j 的原始 token”了。**归纳头能在 Transformer 里长出来，残差流是前提之一。**）

4.16 为什么第 0 层的头不行

有人会问：第 0 层的头能不能自己完成这件事？

不能，因为它在位置 i 只能看到原始输入 $x_0, \ldots, x_i$。“位置 j 的前一个 token 是 x_{j-1}”这个信息**是派生的**—— 它需要一次注意力操作才能算出来（L0H1 干的就是这个）。

第 0 层的头看得见 x_{j-1} 和 x_j，但它没法把这两个“绑”成一对。

这个“绑”就是注意力的作用：一个头把 x_{j-1} 搬到位置 j，于是位置 j 的表示里同时含有 x_j（来自残差流）和 x_{j-1}（来自注意力）。

第 1 层的头才能看到这个“配对”。

所以归纳头至少需要两层——这是架构层面的下界，不是调参能绕过的。

> ⚠ 但“两层”只是必要条件，不是充分条件
>
> 上面证明了归纳头**至少**需要两层。这不等于“两层模型一定有归纳头”。
>
> **我们自己的模型就是反例**：2 层，但 4.6.3 节的实测显示归纳分数 与随机基线齐平（0.0285 vs 0.0284）。
>
> **为什么“够两层”却没长出来？** 回到 4.6.4 节那三条原因：
>
> 1. **每头只有 32 维**，内积的分辨率有限，匹配容易出错
> 2. **只有 4 个头**，第 0 层要同时承担“前一位”和“造键”两个职责
> 3. **任务是唐诗**，重复大多是高频字的偶然撞车，不是有意义的模式重复
>
> **“理论上可能”和“实际上发生了”是两件事。** 前者用数学证明，后者只能实测。**本章两件都做了：证明了可能性，实测了没有发生。**

4.16.1 怎么测：一个干净的指标

归纳头有一个别的头都不具备的特征：**它只在“重复出现过”的位置上才起作用。**

这个特征直接给出了测法：

1. 造一条**人为重复**的序列：随机取 P 个 token，把它重复 4 遍。

```
t₁ t₂ ... t_P  t₁ t₂ ... t_P  t₁ t₂ ... t_P  t₁ t₂ ... t_P
```

2. 看第二遍及以后的每个位置 i（它对应第一遍的位置 $j = i - P$）
3. 测这个位置 i 的注意力，有多少落在 $j+1$ 上

$$\text{归纳分数} = \frac{1}{|S|} \sum_{i \in S} A_{i,\, j+1}, \qquad j = i - P \tag{4.18}$$

所以这个式子在说：在重复出现的位置上，注意力有多大比例 落在“上次出现之后那一位”。

一个真正的归纳头，这个分数会显著高于 $1/i$（均匀基线）。 一个普通的头，分数会落在均匀基线附近。

这个指标我们写错过一次

第一版的代码里，那个下标写成了 `j - 1`：

```
hits.append(m[i, j - 1].item())     # ← 错：这是"上一次**之前**那一位"
```

归纳头看的是“上一次之后那一位”，即 $j+1$。

写成 `j-1` 测的是完全不同的东西——它测的是“有没有一个头在关注 上上次出现的位置”，那根本不是归纳头。

这个 bug 不会报错，只会给出一个同样在噪声水平上的分数。 所以它躲过了第一轮测试。发现它是因为：**修正前和修正后， 随机对照的分数都在 0.028 上下，而两版的数字看起来都“合理”。**

这就是“没有对照就发现不了 bug”的又一个例子—— 只有把训练模型和随机模型放在一起，才会注意到“这个指标根本测不出区别”。

4.16.2 实测：我们的模型没有归纳头

头	训练模型	随机对照
L1H3	0.0285	0.0285
L1H2	0.0269	0.0284
L0H2	0.0253	0.0283
L1H0	0.0232	0.0285
L1H1	0.0221	0.0285
L0H1	0.0143	0.0285
L0H3	0.0141	0.0284
L0H0	0.0007	0.0285
均值	0.0194	0.0284
最高	0.0285	0.0285

这张表要说三件事

第一，随机模型给出了一条极其平整的基线：0.0283–0.0285。八个头全部落在这个窄带里，**这说明 0.0284 就是这个指标在“没有任何归纳行为”时的自然水平。**

第二，训练模型的八个头里，只有 L1H3 碰到了这条基线（0.0285），其余七个全部低于它。最低的 L0H0 是 0.0007——**它对“上次出现之后那一位”完全不看。**

第三，没有头显著高于基线。

结论：这个模型里没有归纳头。

这不是“我们的测量失败了”，这就是结论本身。

> **i** 停下来想一想 ④
>
> 上面说“没有头高于基线”。但 L0H0 的分数是 **0.0007——远低于**基线 0.0284。
>
> 一个“什么都没学”的头，分数应该**在**基线附近，而不是远低于它。 **0.0007 说明什么?**
>
> （答案：说明 L0H0 有明确的**偏好**——它的对角权重是 0.769， 也就是说它几乎只看自己。位置 i 看自己，自然就看不到 $j+1$。
>
> **所以“低于基线”和“高于基线”一样，都是信息。**
>
> 对 L0H0 来说，这个分数低是**意料之中**的：一个恒等头当然不会去做归纳。真正值得注意的是：**其余七个头的分数在 0.014 到 0.029 之间， 虽然高低不一，但没有一个明显跳出来。** 这个“平”才是结论。）

4.16.3 为什么没有：三个可能的原因

原因一：模型太小。

归纳头需要第一层“造键”、第二层“匹配”。本模型只有 2 层，**第一层的头必须同时承担“前一位”和“造可匹配的键”两个职责**—— 而它只有 4 个头、32 维的每头空间。

文献里报告归纳头的是几十层、上亿参数的模型。**在 42 万参数、2 层的模型上找归纳头，本身就是一次超纲的尝试。**

原因二：任务不需要。

回想第 2 章那个实验：**上下文从 8 加到 128 只改善了 0.64， 而且 16 比 8 还差。** 那个实验的结论是：这个唐诗任务对长程依赖的需求很弱。

归纳头是一种**长程**电路——它要跨越 P 个位置去找上一次出现。**在一个不需要长程的任务上，模型没有动力长出这个电路。**

原因三：字符级 + 小词表。

归纳头在**重复的 token 序列**上最有用（比如代码里的变量名、自然语言里的固定搭配）。我们用的是 1200 字的字符表， 唐诗里的重复大多是“的”“不”“春”这类高频字的偶然重复， **不是有意义的模式重复。**

怎么才能在自己机器上看到归纳头

如果你想让归纳头出现，需要同时改三件事：

1. **加深到至少 4 层**（给“造键”和“匹配”各自留出层）
2. **换一个真正有重复结构的任务**—— 比如让模型学 `a b c d a b c d ...` 这样的周期序列，或者用代码（变量名会反复出现）
3. **加宽**：每头维度太小时，“匹配”用的内积分辨率不够

然后重跑这一节的指标。 如果归纳分数跳到 0.3 以上（而不是 0.028），那就是找到了。

本书没做这件事，因为它是“再训一个模型”的量级—— 但**这个实验的设计是完整的，你可以直接拿去用。把“我们没做”和“我们做了但没成功”分清楚，比假装做过要好。**

4.16.4 归纳头测完之后：三个“差一点”的观察

4.6 节的结论是“没有归纳头”。但这个结论背后有三个值得单独说的观察—— **它们都是从数据的形状里读出来的，不是从数值大小。**

观察一：随机基线窄得反常。

```
随机模型的 8 个分数: 0.0285 0.0284 0.0283 0.0285 0.0285 0.0285 0.0284 0.0285
```

极差 0.0002。

一条这么窄的带子只能有一个来源：**所有头都在做同一件事**—— 均匀地看所有允许的位置。

这给了我们一个非常干净的零假设：如果一个头不做任何归纳行为，它的分数**精确地**是 0.0284。任何偏离都意味着某种结构。

观察二：训练模型的分数大部分低于基线。

```
0.0285  0.0269  0.0253  0.0232  0.0221  0.0143  0.0141  0.0007
```

七个低于基线，一个等于基线。

没有一个高于。

这个分布形状本身就是结论：训练让头变得**更不倾向**于做“看上次出现之后那一位”这件事—— 因为它把注意力集中到了别的地方（对角、前一位）。

观察三：最低的那个头正好是“最专注自己”的那个。

```
L0H0  归纳分数 0.0007   对角 0.769    ← 恒等头
L0H1  归纳分数 0.0143   前一位 0.450  ← bigram 头
```

分数最低的 L0H0 恰好是画像里最极端的。

4.17 三个观察合起来说明什么

归纳头需要“注意力分散到远处”。而训练做的事正相反—— 它把注意力**收紧**到几个明确的位置（自己、前一位）。

所以“训练造出结构”和“训练造出归纳头”是两件可能冲突的事。

在一个小模型上，**分化（specialization）先发生，长程匹配没轮到。**模型先把最基础的统计量（bigram）学到手，把注意力都投在上面，**没有余力去长一个跨越 P 个位置的匹配电路。**

这是一个“容量分配”的故事：头的数量、维度、层数都是有限的，模型把它们用在了收益最高的地方。

而“收益最高的地方”由任务决定——唐诗不需要长程匹配（第 2 章测过）。

⚠ 一个不能从这些数据得出的结论

上面说“训练让注意力收紧，所以没长出归纳头”。
这是一个事后解释，不是被验证的机制。
要验证它，需要：在一个**需要**长程匹配的任务上训练同一个模型，看归纳分数会不会变高。
本章没做这个实验。它是“再训一个模型 + 换一个数据集”的量级，超出了本章的范围。
但它是一个清晰的、可执行的后续工作—— 而且预测很明确（在周期序列上训练，归纳分数应该显著上升）。**一个好的后续工作应该长得这样：说清楚做什么、预期看到什么。**

4.17.1 换个任务会长出归纳头吗：一个可执行的设计

把上面那个“后续工作”写具体——因为**设计实验比描述实验更有用**。
任务：周期序列复制。

```
# 造数据：随机取 P 个 token，重复 rep 遍
# "a b c d e | a b c d e | a b c d e"
def make_cyclic_batch(B, P, rep, vocab, gen):
    pat = torch.randint(3, vocab, (B, P), generator=gen)
    return pat.repeat(1, rep)
```

为什么这个任务能逼出归纳头?
因为**除了“看上次出现之后那一位”，没有别的办法预测下一个 token**。序列本身是随机的（token 之间没有统计关系），唯一的规律是“它重复了自己”。
模型只有两个选择：学会归纳（看过去找匹配），或者放弃（猜均匀分布）。
预期结果：

模型	预期归纳分数
随机初始化	≈ 0.028（几何基线）
在唐诗上训练（本章的模型）	≈ 0.028（**实测**）
在周期序列上训练	**应该显著高于 0.028**

如果第三个也是 0.028，那说明 2 层 4 头**结构上不够**—— 这时候要加层、加头，而不是继续调任务。

怎么区分“任务不对”和“模型太小”

这是一个设计实验时最常见的两难：结果不符合预期时，你分不清是任务选错了，还是模型不够大。
做法：做阶梯。

1. 先固定任务（周期序列），**扫模型规模**（2/4/8 层）
2. 如果规模上去之后归纳分数跳起来 → **是模型太小**
3. 如果规模上去也没用 → **是任务或数据的问题**

而不是：同时改任务和模型，然后猜是哪个因素起作用。
一次只动一个变量——这条第 2 章讲对照实验时就说过，但它在这里的表现形式不同：不是“两个模型比”，而是“一个模型扫一个维度”。

> **i** 停下来想一想 ⑭
>
> 上面那个周期序列任务，**如果用 1 层模型呢?**
> （答案：**理论上不可能，而且可以证明**。
> 4.6.2 节推导过：归纳头需要“第 0 层造键、第 1 层匹配”。1 层模型只有一次注意力操作，它在位置 i 只能看到**原始输入** $x_0, ..., x_i$——而“位置 j 的前一个 token 是什么”是一个**派生量**，需要一次注意力才能算出来。
> **所以 1 层模型在周期序列任务上的 loss 应该停在 “均匀分布的交叉熵”附近**（$\log V$），学不动。
> **这是一个可以预测的负面结果**——而可预测的负面结果是验证一个理论最好的方式：**理论说 1 层不行，跑出来确实不行，理论就得到了一次支持**。
> **本章的模型恰好是 2 层，所以这个实验做不了**。要做需要重新初始化一个 1 层模型——**代码上就是** `Config(n_layer=1)`。）

4.18 探针：从激活里读出信息

4.18.1 探针是问“这里有没有这个信息”

前几节问的是“这个头在做什么”。探针问的是另一个问题：

> **模型的某个中间表示里，有没有编码某个特定的属性?**

做法很直接：**训一个线性分类器**，输入是某一层的激活，输出是那个属性。如果线性分类器能达到很高的准确率，说明**那个属性被线性地编码在里面**。

```
# 探针就是一层线性层 + 二分类
w = torch.zeros(d_model, 1, requires_grad=True)
b = torch.zeros(1, requires_grad=True)
opt = torch.optim.Adam([w, b], lr=0.05)
for _ in range(300):
    loss = F.binary_cross_entropy_with_logits(X @ w + b, y[:, None])
    opt.zero_grad(); loss.backward(); opt.step()
```

为什么用线性而不是 MLP? 因为线性探针能证明的是**更强的结论**：不是“信息可以被解码”（一个足够大的 MLP 能从任何东西里解码任何东西），而是“信息以线性可分的形态存在”。**后者才有意义**。

4.18.2 选一个“模型必须自己算出来”的属性

探针实验最容易犯的错是选错属性。我们第一次就选错了。
第一版选的是：**“当前位置的字符是不是标点”**。
结果：

```
embed    探针准确率 1.0000    多数类基线 0.8438
```

词嵌入层的准确率是 1.0000。这意味着探针**根本没有在测模型**——它从词嵌入里直接读出了答案，因为**答案就写在输入里**：不同的 token 有不同的嵌入，标点字符的嵌入天然是可分的。
这不是“模型编码了标点”，这是“输入就是标点”。
探针作弊：一个必须主动防的陷阱
上面那个 1.0000 说明探针学到了一个**平凡解**：“如果嵌入向量接近某个标点的嵌入，就判它是标点”。
任何在输入里就能看出来的属性，探针都会作弊。常见的几类：

属性	会不会作弊	为什么
当前 token 是不是标点	**会**	答案在输入里

属性	会不会作弊	为什么
当前 token 的词性	**会**	词性基本由 token 本身决定
下一个 token 是不是标点	不会	输入里没有
这个位置在第几个字	不会	需要位置信息
这句话是正面还是负面	不会	需要跨位置整合

做法：把“embed 层”当成对照。如果某一层的探针准确率和词嵌入层差不多，那它可能什么都没算，只是在复述输入。

这条对照线是免费的，而且能挡住大部分假阳性。

4.18.3 实测：残差流里能不能读出“下一个是不是标点”

改成预测**下一个**字符的属性——这是模型必须自己算的。

```
层          探针准确率     多数类基线
embed       0.7188         0.8438    ← 低于基线
layer1      0.9375         0.8438    ← 明显高于基线
layer2      0.8750         0.8438    ← 略高于基线，但比 layer1 低
```

这张表读出来的三件事

第一，词嵌入层（0.7188）低于多数类基线（0.8438）。

这是一个很干净的负结果：**“下一个字符是不是标点”这个信息，在输入里完全不存在。**探针只能瞎猜，甚至因为类别不平衡而猜得更差。

这条线是必需的——它证明了后面两层的准确率不是从输入漏过来的。

第二，第 1 层是 0.9375。

模型在第 1 层就已经算出了“下一个大概率是标点”。这符合直觉：唐诗有固定的句长（五言、七言），逗号句号的位置高度规律。**第 1 层的注意力头完全有能力捕捉到“位置模 5”或“位置模 7”这种模式。**

第三，第 2 层反而降到了 0.8750。

这一条最值得琢磨。**第 2 层的表示里，“下一个是不是标点”这个信息变弱了。**

为什么？因为**第 2 层的任务是预测具体的下一个字，不是预测它是不是标点。**它需要把“是标点”这个粗粒度的信息，细化成“是逗号还是句号还是问号”。**抽象属性在往具体答案转化的过程中被稀释了。**

“越深的层编码越抽象的信息”是一个常见的说法——但这里的数据不支持它。在这个 2 层模型上，一个具体的语法属性在**中间层最清晰**。

> ⚠ “能读出”不等于“用到了”
>
> 这是探针研究里最重要的一条界限。
>
> **探针证明的是**：信息**存在**于那个表示里，而且是线性可分的。
>
> **探针没有证明的是**：模型**用**了那个信息。
>
> 一个反例的思路：假设模型在第 1 层顺手算出了“下一个是不是标点”（因为预测字符分布时这一步是副产品），但**第 2 层完全不依赖它**—— 第 2 层自己从别的特征里就能预测。这时探针仍然会给出 0.9375，**但这个属性对模型的输出没有因果作用。**
>
> **怎么补上因果那一半？**和我们 4.4 节做头消融一样：**干预那个表示**——把“下一个是标点”这个方向从激活里减掉，看模型的输出变不变。
>
> ```
> # 探针权重 w 给出了"标点方向"。把它从激活里投影掉：
> X_intervened = X - (X @ w_hat) @ w_hat.T      # w_hat 是 w 的单位向量
> # 再跑一遍前向，看 loss 变多少
> ```
>
> **这个实验本书没有做**——它是“探针 + 因果干预”的完整形态，属于后续工作。**但它的设计是从本节的探针直接接上去的，只差一次投影。**

4.18.4 探针的完整流程：五个决策点

4.7 节给了探针的代码和结果。这一节把流程拆开，因为**探针实验有五个必须主动做的决策，每一个做错都会给出假阳性**。

决策一：选什么属性。

标准是：**这个属性的答案，不能在输入里直接看到**。

我们第一版选了“当前字符是不是标点”，词嵌入层就给到 1.0000。改成“**下一个**字符是不是标点”之后，词嵌入层掉到 0.7188（低于基线）—— **这才是一个能用的属性**。

决策二：用线性还是 MLP。

必须用线性。 理由不是“简单”：

- **线性探针能证明**：那个属性以**线性可分**的形式存在于表示里
- **MLP 探针只能证明**：那个属性可以**被算出来**——但“算”这件事 神经网络本来就会做，一个足够大的 MLP 能从任何东西里算出任何东西

MLP 探针的准确率高，但信息量为零。

决策三：训练集和测试集怎么分。

听起来是常识，但有一个探针特有的陷阱：**同一个序列内的位置高度相关**。

```
# 错的：随机划分位置
idx = torch.randperm(N)
train, test = idx[:N//2], idx[N//2:]
```

相邻位置的特征几乎一样（残差流是平滑变化的）。**随机划分会让“训练位置的邻居”落进测试集**，准确率被高估。

正确做法：按序列划分。 整条序列要么进训练集，要么进测试集。

决策四：对照基线选什么。

本章用了两个：

- **多数类基线**（永远猜出现最多的那一类）：本实验是 0.8438
- **词嵌入层**：本实验是 0.7188

两个都要。多数类基线告诉你“不学任何东西能拿多少分”；词嵌入层告诉你“输入里漏过来多少信息”。

决策五：报什么数。

只报准确率不够。**还要报：**

- **正例率**（本实验 0.1562）——类别极不平衡时， 准确率这个指标本身会误导
- **两个基线**
- **探针自己的容量和训练步数**（否则别人无法复现）

> ⚠ 一个具体的“假阳性”演示
>
> 假设某个属性在数据里的正例率是 5%，而你的探针准确率是 94%。
>
> **看起来很厉害？** 但“永远猜负类”的基线就是 95%。
>
> **你的探针比瞎猜还差**。
>
> 这不是编的例子——**这就是我们第一版探针在 embed 层的表现** （0.7188 vs 0.8438）。如果没有把基线一起报出来，那个 0.7188 看起来会像一个“模型没学到这个属性”的中等结果，而实际上它是“探针被类别不平衡带偏了”。

4.18.5 把探针接到因果上：一个没做完的实验

4.7 节末尾说“能读出 ≠ 用到了”，并给了一个干预的写法。 这一节把那个实验的设计补完整——**因为它是本章唯一一个“设计了但没跑”的实验**。

问题：第 1 层的探针准确率 0.9375。模型**真的用了**“下一个是不是标点”这个信息吗?

做法：探针的权重向量 $\mathbf{w}$ 定义了激活空间里的一个方向—— “标点性”方向。**把这个方向从激活里减掉，看输出变不变**。

$$\tilde{\mathbf{h}} = \mathbf{h} - (\mathbf{h} \cdot \hat{\mathbf{w}})\hat{\mathbf{w}} \tag{4.19}$$

所以这个式子在说：把 $\mathbf{h}$ 在 $\hat{\mathbf{w}}$ 方向上的分量消掉，剩下的部分是“和标点性无关的部分”。$\hat{\mathbf{w}} = \mathbf{w}/\|\mathbf{w}\|$ 是单位向量。

然后跑完整的前向，测 loss。

4.19 三种可能的结果，三种不同的结论

投影后 loss 变化	结论
几乎不变	信息存在但**没被用**——探针找到的是一个“副产品”
明显变差	信息**被用了**——这是一个真正的因果通路
变好	这个方向在**干扰**模型——罕见但可能

只有第二种才能说“模型用了这个信息”。

注意这个实验比消融更细：消融是拿掉一整个头（可能同时抹掉好几条信息），投影是**只抹掉一个方向。前者粗，后者精确。**

本书没跑这个实验的原因是：投影会改变激活的分布，而后续层是在原来的分布上训练的——**分布外输入会让结果难以解释。**要严谨地做，需要一个“控制投影”（投影掉一个同样维度的随机方向），比较两者的 loss 变化。**那至少是另一个小节的工作量。**

但设计是完整的，代码只差十行。要接着做的人从这里开始。

> ℹ 停下来想一想 ⑩
>
> 上面说“投影会改变激活的分布，让结果难以解释”。所以要加一个“控制投影”——投影掉一个**随机方向**，看 loss 变多少。
>
> **为什么要随机方向，而不是随便挑一个别的主成分？**
>
> （答案：因为你要控制的是“**做了一次投影**”这件事本身的代价。
>
> 投影任何方向都会扰动激活。如果投影掉“标点方向”让 loss 涨了 0.1，你需要知道“投影掉一个随便什么方向”会让 loss 涨多少。**如果随机方向也涨 0.1，那 0.1 是投影的代价，不是标点性的作用。**
>
> **这和探针的“词嵌入层对照”是同一个思路：你要证明的不是“我做了 X 之后有变化”，而是“我做的 X 造成的，比其他同类的 X 更大”。**
>
> **控制实验的设计，是这类研究里最难也最容易被跳过的一步。**

4.19.1 探针能回答的三种问题和不能回答的一种

4.7 节讲了探针“能读出”和“用到了”的区别。这一节把探针**能回答什么**列清楚——因为**误用探针最常见的方式是问它答不了的问题。**

能回答的问题一：这个信息在不在？

“第 1 层的激活里，有没有’下一个是不是标点’这个信息？” → 探针准确率 0.9375 vs 基线 0.8438。**在。**

能回答的问题二：在哪一层最清晰？

对比 embed / layer1 / layer2：0.7188 → 0.9375 → 0.8750。**中间层最清晰。**

这是一个反直觉的结果（通常认为越深越抽象），但在这个模型上有明确的数据。

能回答的问题三：不同属性之间的相对难度？

比如同时测“下一个是不是标点”和“下一个是不是句号”，**两者的准确率差说明了这两件事在表示里的“清晰程度”不同。**

不能回答的问题：模型是不是“理解”了这个属性。

这是探针最容易被误用的地方。

4.20 “探针能解码” ≠ “模型理解”

一个思想实验：

假设有一个属性 Z，它和输入**完全无关**（比如“这句话的作者今天心情如何”）。但由于训练数据的统计巧合，Z 和某个真正被编码的属性 Y **高度相关。**

那么：**探针会从激活里“读出” Z，因为 Z 和 Y 相关。**

但模型完全不知道 Z 的存在——**它只是在编码 Y。**

探针测的是“可分性”，不是“表征”—— 只要激活里有一个方向和一个标签相关，线性探针就能读到它，**不管模型有没有“意图”编码那个标签**。

这是“探针 ≠ 理解”的技术版本。它不是哲学问题，是一个具体的统计现象。

> 怎么防这个陷阱：控制任务
>
> 延伸阅读 ⑤ 那篇论文给的做法是**控制任务**：
>
> 1. 给你的真实标签 Y 训一个探针，得到准确率 A_{real}
> 2. 把标签**随机打乱**，再训一个探针，得到 A_{ctrl}
> 3. **只有 $A_{\text{real}} - A_{\text{ctrl}}$ 才是“真实的可分性”**
>
> **第 2 步的意义**：随机标签和输入没有关系，但探针仍然能从激活里读出**某种**东西（因为它有容量）。A_{ctrl} 就是这个“探针的选择能力”的下限。
>
> **本章用的是“词嵌入层对照”**——它是这个思路的一个简化版：如果输入里就有答案，词嵌入层的探针会很高。**但它挡不住“两个属性相关”这种情况**—— 那个需要标签打乱。
>
> **本章没做标签打乱**。这是探针实验里一个标准的、本可以做的对照。

4.20.1 一个更严格的探针：把“位置”也控制住

还有一个本章没处理的混淆：**位置**。

“下一个是不是标点”这个属性，**和位置高度相关**—— 在五言诗里，标点出现在第 5、10、15…… 个位置。

所以一个探针可能只是在读“当前位置 mod 5”。

怎么区分？ 加一个位置对照：

```
# 探针对照：预测"当前位置 mod 5 是不是 4"
# 如果这个探针的准确率和"下一个是不是标点"差不多，
# 说明前者可能只是在读位置，没有额外的信息
```

这是“两个属性相关”那个陷阱的一个具体实例，而且它在本章的数据上**非常可能成立**（因为诗有固定句长）。

> 那 0.9375 这个数还可信吗
>
> **它可信，但它的解读要改**。
>
> **可信的部分**：第 1 层的激活里，**确实有信息能区分“下一个是不是标点”**。这是事实，探针准确率摆在那里。
>
> **要改的解读**：**不能说是“模型算出了下一个是标点”**。可能是：- 模型确实在跟踪句内位置（那它等价地知道下一个是标点）- 或者模型在跟踪别的东西，那个东西恰好和标点位置相关
>
> **两种情况的探针准确率一样**。
>
> **区分方法**：在**句长不固定**的文本上重测。如果准确率掉下来，说明它在读位置；如果不掉，说明它真的在处理标点。
>
> **本章用的语料全是五言七言——句长完全固定**。所以这个混淆在本章的数据上是无法排除的。
>
> **“我们的语料有一个结构，这个结构可能解释我们观察到的现象”**—— 这是做测量时最该主动问的一句话。

> **i** 停下来想一想 ⑯
>
> 上面说“诗的句长固定，所以探针可能只是在读位置”。
> **这个混淆会不会也影响 4.9 节那个韵脚排查的结论?**
> （答案：**会，而且方向是反的**。
> 4.9 节测到“标点位置的 loss 是 0.246，其他位置是 4.507”，我们当时的解读是“因为句长固定，数位置就能预测标点”。
> **这个解读和探针那个混淆是同一件事的两面**： - 探针那边：模型可能在读位置 - loss 这边：模型确实在读位置（所以标点才这么好预测）
> **两者互相印证，而不是互相独立**。
> **这提醒了一件事**：当你发现两个实验“结果一致”时，要先问它们是不是**独立的**。**同一个混淆因素可以同时解释两个实验**。

4.21 失效边界：可解释性研究的四个陷阱

这一节把本章踩过的和文献里反复出现的坑收成四条。**它们的共同点是：都不会报错，只会给出一个“看起来合理”的错误结论**。

4.21.1 陷阱一：探针作弊

4.7 节已经完整讲过——选了一个“答案写在输入里”的属性，词嵌入层的探针准确率冲到 1.0000。
识别方法：**看词嵌入层那条对照线**。如果它也高，说明你测的不是模型。
推广：任何“从激活里解码属性”的实验，都要先问一句**“这个属性的答案，是不是在输入里就能看到?”**

4.21.2 陷阱二：多重比较

本章测了 8 个头。如果测 800 个呢?
“至少有一个头看起来在做某件事”的概率，随头数上升得很快。假设没有任何头真的在做那件事，但每个头的指标有 5% 的随机波动—— 那么 800 个头里，期望有 $800 \times 5\% = 40$ 个“显著”。
这就是为什么本章每一节都配了对照：随机模型的八个头给出 0.0284 那条基线，训练模型的分数才有比较的对象。
一个具体的数字感受
假设你在一个 24 层的模型上，逐层逐头地找“某个特定模式”（比如“这个头是不是在看动词”），一共 $24 \times 16 = 384$ 个头。
如果你按“分数最高的那个头”下结论，**你几乎必然会找到一个**—— 哪怕模型里根本没有这个模式。
正确做法是先定阈值，再测。阈值从哪来？**从随机对照来**：跑几个随机初始化的模型，看这个指标的分布，取一个明显超出该分布的阈值。
本章的归纳头就是这么做的：随机模型给出 0.0283–0.0285 的窄带，所以“显著”的门槛至少在 0.05 以上。**实测最高 0.0285，没有头越线**。

4.21.3 陷阱三：消融的相互作用

4.4 节做的是**单头消融**——一次拿掉一个头。它测不出**头与头之间的替代关系**。
考虑一个极端情况：两个头在做同一件事。单独拿掉任何一个，另一个顶上，Δ 都很小。**看起来两个都不重要**。
但**同时**拿掉两个，模型可能直接崩掉。
反过来也成立：两个头互相干扰，单独拿掉任何一个都让效果变好，同时拿掉反而不如只拿掉一个。
怎么测相互作用
做法是**成对消融**：对每一对头 (i, j)，测三个量——

$$\Delta_i, \quad \Delta_j, \quad \Delta_{ij} \tag{4.20}$$

如果 $\Delta_{ij} \approx \Delta_i + \Delta_j$，两个头是**独立**的。**如果 $\Delta_{ij} < \Delta_i + \Delta_j$**，它们**部分冗余**（互相能顶）。**如果 $\Delta_{ij} > \Delta_i + \Delta_j$**，它们**互补**（各管一半，缺一不可）。

本模型 8 个头，成对是 $\binom{8}{2} = 28$ 组，跑一遍完全可行。

本书没有做这个实验——它是本章最后一块没补的拼图。从 4.4 节的代码接上去只要改一个循环：`ablate_head` 注册两次 hook 就行了。

4.21.4 陷阱四：确认偏误

这条最难防，因为它不涉及代码。

你画出一张注意力图，然后去找一个解释。人脑很擅长在噪声里找模式——第八章那种“云像什么”的能力，在解读注意力图时是**负资产**。

> ⚠ 三条具体的对抗办法
>
> **一、先预测，再看图。**
> 在看到 L0H1 的图之前，先写下“如果它是一个 bigram 头，我应该在图上看到什么”。然后看图。**如果图和预测不符，先怀疑预测，再怀疑图。**
> **二、换一组输入。**
> 本章的画像是在三条诗上测的。**如果一个“模式”只在你精心挑的那一条上成立，它不是模式，是巧合。**
> **三、把解释写成一个能被证伪的句子。**
> “这个头在关注语义上重要的词”——**不可证伪**，因为它没说清“重要”怎么测。
> “这个头看紧邻的前一位，权重 0.450，拿掉它 loss 涨 0.177”——**可证伪**，而且已经验证过了。
> **区别在于：后者能错，前者不能。**

> ℹ 停下来想一想 ⑤
>
> 四条陷阱里，哪一条是**本章自己踩过**的？
> （答案：三条踩过。
> **陷阱一（探针作弊）**：第一版探针选了“当前字符是不是标点”，词嵌入层 1.0000。4.7 节完整记录了。
> **陷阱二（多重比较）**：归纳头指标的 `j-1` 写错那次，如果没有随机对照线，我们会以为 0.028 是“一个很弱但存在的归纳头”。
> **陷阱四（确认偏误）**：本章开头那个“L0H1 是 bigram 头”的结论，是先看图、后找解释的。**它恰好对了（4.4 节的消融证实了），但流程是错的**——正确的顺序是先预测再验证。
> **陷阱三（相互作用）本章还没踩，因为还没做。**

4.21.5 陷阱五：把“消融无影响”当成“这个头没用”

4.4 节的实测里，第 1 层的四个头 Δ 都在 +0.02 到 +0.03 之间。一个自然的结论是“第 1 层的头基本没用”。

这个结论太强了。至少有三条反驳：

反驳一：冗余不等于无用。一个头可以被另一个头完全替代，但它仍然在做那件事。**“拿掉没影响”说的是系统的可替换性，不是部件的作用。**

类比：一架四引擎飞机，关掉一个引擎仍然能飞。**这不说明那个引擎没用**——它说的是“剩下的三个够用”。

反驳二：测的是平均性能，不是全部能力。验证集 loss 是一个平均量。一个头可能只负责“某类罕见输入”，而那类输入在验证集里占比很低。**拿掉它，平均 loss 几乎不变，但那类输入全错了。**

反驳三：消融改变的是模型，不是任务。拿掉一个头之后，模型会用一个“残废但自洽”的方式继续工作。**它可能发展出了新的补偿路径**（在一次前向里不可能，但我们在评估的是同一个权重下的行为）。准确说：**消融测的是“这个头在现有其他部件都在的情况下有多必要”**，不是“这个头本身有多重要”。

怎么把这三条补上

反驳	补法
冗余	成对/成组消融（4.4.4 节做了）
平均掩盖	逐位置看 Δ（4.4.5 节做了）
只测一类输入	按输入类型分层评估（本章没做）

第三条的补法：把验证集分成若干子集（比如“含标点的句子”、“长句”、“生僻字多的句子”），**在每个子集上分别测消融效果**。

一个头可能只在某个子集上重要。平均之后看不出来。

4.21.6 陷阱六：把“可解释”当成“已解释”

这是最根本的一条，也是本章反复在绕的那个点。

“这个头在看前一位”——这句话解释了什么？

它解释的是**信息路由**：位置 i 从位置 $i-1$ 取了信息。

它没有解释的是：- 取过来之后，模型用它做了什么？ - 如果换成从别处取，结果会怎样？ - 为什么是 0.45 这个数值，而不是 0.2 或 0.8？

换句话说：你描述了机制的一环，但没有描述机制。

4.22 一个类比

想象你要解释一台计算器为什么能算加法。

“按键 3 的信号传到了运算单元”——这是一个真实的、可测量的、有因果作用的描述。

但它不是“计算器怎么算加法”的解释。它只是链路里的一跳。

注意力图给出的就是这种级别的东西：一跳的路由信息。它比“什么都没说”强，比“解释了模型”弱得多。

这个定位很重要——它决定了你应该拿注意力图做什么：**当作线索，而不是当作结论。**

那这个领域还有什么意义

如果注意力图只给出“一跳”，机制可解释性是不是注定徒劳？

不是。4.6 节的归纳头就是一个反例——那是一个**完整的、端到端的、能被证明的电路**：从“哪个头做什么”到“为什么必须两层”到“消融掉能力就没了”。

归纳头之所以能说清楚，是因为它足够小、足够具体、有明确的输入输出。

这个领域现在的主流判断是：**小的、可验证的电路能被理解；“整个模型在做什么”这种大问题，目前还没有可用的方法。**

所以务实的做法是：不要问“这个模型可解释吗”，问**“这个具体的电路能不能被验证”**。

4.22.1 一个必须先回答的问题：为什么用这么小的模型

本章整章都在分析一个 2 层、4 头、42 万参数的模型。**读者有权问：这么小的模型，结论有意义吗？**

诚实的回答分三层。

第一层：小模型上成立的结论，大模型上不一定成立。

这一点本章反复标注了：

- “第 0 层比第 1 层重要 6 倍”——**只在 2 层模型上有意义**
- “没有归纳头”——**很可能只是因为太小**
- “头之间有冗余”——大模型的冗余模式可能完全不同

所以本章的结论首先应该读成“关于这个模型的结论”，其次才是“关于 Transformer 的一个例子”。

第二层：但方法是可以迁移的。

本章真正想教的不是那些数字，而是这套流程：

方法	在小模型上	在大模型上
头部画像	8 个头，一眼看完	上千个头，要分组
消融	8 次前向	要抽样或分组
随机对照	秒级	秒级（同样便宜）
归纳头指标	秒级	秒级
探针	分钟级	分钟级

只有前两项受规模影响，后三项完全一样。而“配随机对照”“做因果检验”这两条原则，和规模无关。

第三层：小模型有一个大模型没有的好处——你能看完。

在一个上千个头的模型上做“头部画像”，你会得到一张一千行的表，**然后你必须写脚本去筛，筛完之后你并不知道自己漏了什么。**

本模型的 8 个头，你可以一个一个看过去，然后对每一个形成判断。这种“完整覆盖”是小模型独有的。

4.23 这和第 1、2 章的选择是一致的

回顾第 1 章：**为了复现“去掉 $\sqrt{d_k}$ 会怎样”，我们用了 42 万参数的模型**——因为在那个规模上，实验可以在几分钟内跑完，可以反复做对照。

第 2 章：**不同深度的对照（6/12/24/48 层）**，每个都只有几十万参数。

本书从头到尾都在做同一个取舍：用能完整分析的模型，讲清楚机制；而不是用真实的规模，讲一个看不清的故事。

代价是结论的适用范围有限。这个代价在本章标注得最清楚—— 因为本章的结论**很大程度上依赖“模型有多小”**。

4.23.1 如果只有时间做一个实验，做哪个

在本章的五个实验里，如果必须砍掉四个，**保留哪个?**

答案：随机对照。

理由：

- **头部画像**本身不产生结论，它只产生候选
- **消融**给了因果，但如果没有对照，你不知道 $\Delta = 0.03$ 算大算小
- **归纳头**是一个特例，不是通用方法
- **探针**需要一个正确的属性选择，很容易做错

而随机对照是一个“元实验”：它给所有其他测量提供零点。

本章的数据完全支持这个判断：

没有随机对照会怎样	实际发生
8.3 节的“对角 0.157”会被当成“盯住自己”	实际是均匀基线
归纳头指标的 0.028 会被当成“弱归纳头”	实际是几何基线
“熵 0.958”会被当成“模型没学到东西”	实际是“训练后方差放大 5000 倍”的证据

三个误判，都是靠同一条对照线纠正的。

这条原则可以推广到任何测量

任何“我测到了 X”的声称，都要配一个“如果什么都没发生，我会测到什么”。

- 测注意力模式 → 配随机模型
- 测探针准确率 → 配多数类基线 + 词嵌入层
- 测消融效应 → 配逐位置分布（而不只是均值）
- 测电路存在 → 配几何基线

这四种对照，本质上都是在回答同一个问题：“零假设长什么样?”
而“零假设长什么样”这个问题，只能测，不能猜—— 本章那个 0.0284 就是测出来的，猜不出来。

4.24 从观察到结论：一条完整的推理链

前面几节各自给了一种检验。这一节把它们串成一条链—— **这是本章最实用的一节，因为它给出的是一个可以照着走的流程。**

4.24.1 六个台阶

从“我看到了一个模式”到“这个模式是真的”，中间有六级台阶。**每一级都比上一级贵，所以不要一上来就做最贵的。**

级	问题	方法	成本
1	这个模式在这个输入上存在吗?	画图/统计	分钟
2	换个输入还在吗?	多组输入重测	分钟
3	随机模型也有吗?	随机对照	分钟
4	它在数据里常见吗?	分层统计	十分钟
5	拿掉它行为变吗?	消融	十分钟
6	只拿掉它、不动别的呢?	定向干预（投影）	半小时

关键在“从下往上”和“从贵到便宜”的顺序：先用便宜的筛，筛不过就别做贵的。

4.24.2 用本章的 L0H1 走一遍

第 1 级：模式存在吗?
「空山不見人」上，L0H1 的前一位权重是 0.45。**存在。** 这一步没花任何成本——画一张图就看到了。
第 2 级：换个输入还在吗?
三条诗、71 个位置上的平均是 0.450。**在。** 而且这个平均值本身就已经是“多次测量”了。
第 3 级：随机模型也有吗?
随机模型的对应量是 **0.115**。**0.450 是它的 3.9 倍。** 随机模型不会产生这个模式。
第 4 级：它在数据里常见吗?
这一级本章做得不够。**“前一位权重 0.45”是一个平均量。**它在什么条件下更高? 标点位置? 句首? 生僻字后面?
没做这一级的后果是：我们知道 L0H1 平均而言在看前一位，**但不知道它什么时候不看。**
第 5 级：拿掉它行为变吗?
$\Delta = +0.1772$，是八个头里最大的。**变。** 而且变得不小。
第 6 级：定向干预。
没做。 消融拿掉的是整个头（可能同时抹掉多条信息），要精确地测“前一位这一条路”，需要构造一个只针对它的干预。

4.25 这条链走完之后的结论是什么

“L0H1 是一个在做 bigram 的头，拿掉它损失涨 0.177。”
注意这个结论的**边界**：

- 它说的是**这一个模型、这一个任务、这一个数据分布**
- 它说的是**平均行为**，不是“它在每个位置都这样”
- 它没有说**为什么**恰好是前一位，而不是“模型决定要做一个 bigram”

但它是可靠的——因为六个台阶里走过了四个，而且第 3 级（随机对照）给了一个硬门槛（0.115），第 5 级（消融）给了因果证据。
一个走完四级链条的结论，比一个“看起来很有道理”的解释强得多。

4.25.1 反过来：本章哪个结论最弱

诚实起见，把本章的结论按链条完整度排一下：

结论	走到了第几级	强度
训练造出了结构（熵方差放大 5000 倍）	**3**（对照）	**强**
L0H1 是 bigram 头	1,2,3,5	**较强**
第 0 层比第 1 层重要 6 倍	1,5	**中**（缺对照）
头之间有冗余（28 组全负）	1,5	**中**
L1H0 依赖 L0H0	1,5	**弱**（缺第 6 级）
L0H0 是“复制/加权”头	**1 级都没走完**	**只是假设**
模型没有归纳头	1,2,3	**较强**
第 1 层编码了“下一个是不是标点”	1,3	**中**（缺因果）

“L0H0 是复制头”是本章最弱的一条——它连第 1 级都没走完（我们说它“可能”在复制，但没测它的输出和输入的余弦相似度）。
把这张表列出来，比假装所有结论一样可靠要好。

> 这张表可以当模板用
>
> 任何一次“我发现了 X”的汇报，都可以配一张这样的表：
> 结论 | 证据等级 | 缺什么 |
> **好处有两个**：
>
> 1. **逼自己说清楚每条结论的证据强度**，而不是笼统地说“我们发现”
> 2. **给读者一个权重**——他可以据此决定信到什么程度
>
> **第三章的结论表、第二章的排查表，本质上是同一个东西。这本书反复在做的一件事，就是把“结论”和“结论的可靠性”分开写。**

4.25.2 陷阱七：把“训练造出了结构”当成“结构有意义”

4.5 节的方差对比（0.0001 → 0.577）证明了训练造出了结构。**但“结构”和“有意义的结构”是两回事。**
训练可以让头变得高度专门化——**专门化到做一件完全无用的事。**
举个极端的例子：假设某个头在训练过程中被推成了“只在遇到某个罕见字时激活”——它在训练数据上可能确实降低了 loss（因为那个罕见字后面总是跟着同一个字），但在测试集上它是纯噪声。
“这个头的注意力模式很清晰” ≠ “这个模式对模型有用”。
这两件事要分开测：

断言	测法
训练造出了结构	与随机对照（**做了**，方差放大 5000 倍）
这个结构有用	消融（**做了**，第 0 层 6 倍于第 1 层）
这个结构**为什么**有用	需要定向干预（**没做**）

本章走完了前两步，第三步没做。

> ℹ 停下来想一想 ⑮
>
> 上面说“一个头可以被训练成只做一件无用的事”。**这真的会发生吗?**
> （答案：**会，但有个限制。**
> 如果那个头做的事**完全无用**，梯度会把它推回去—— 因为它的输出会参与 loss 的计算，无用就意味着它对 loss 没有贡献， 那它的权重就没有理由保持那个形状。
> **所以“完全无用”的专门化不稳定。**
> **但“在训练集上有用、在测试集上无用”的专门化是稳定的**—— 这就是过拟合。在这个 42 万参数、25 万字语料的模型上， 过拟合是可能的（参数量不小）。
> **怎么检测**：**看这个头在验证集上的行为和训练集上是否一致**。本章的画像是在几条固定的诗上测的——**那是训练集还是验证集？ 两条都可能**（那几条诗可能出现在训练数据里）。
> **这是一个本章没有控制的变量**。严格的做法是明确区分“训练时见过的文本”和“没见过的文本”，分别测画像。

4.25.3 陷阱八：把“一个头的模式”当成“模型的能力”

这是最隐蔽的一条：**从局部机制推出整体能力的结论**。
错误推理的样子：

> 我们发现 L0H1 在看前一位，权重 0.45。 所以模型的“语言能力”主要来自 bigram 统计。

为什么这是错的：L0H1 只是 8 个头里的一个。 即使它完全在做 bigram，**另外 7 个头在做什么，我们没测。**
正确推理的样子：

> L0H1 的前一位权重是 0.450，显著高于随机基线 0.115。 拿掉它，验证集 loss 从 4.1489 涨到 4.3261。 **所以模型对“前一位”这个信息的依赖，有相当一部分走的是 L0H1 这条路。**

注意差别：前者是“模型的能力来自 X”，后者是“模型对 X 的依赖有一部分走这条路”。 **后者是能测的，前者不是。**

4.26 还有一个更麻烦的问题：我们不知道“另外 7 个头在做什么”

本章的画像表给了八个头四个数。**四个数能覆盖多少“头在做什么”的可能性?**
答案：很少。
考虑这些模式，四个量一个都测不出来：

- “看句首的第一个字”（距离大但方向固定，平均距离会很大但没区分度）
- “看上一个标点”（可变距离，平均下来是个中间值）
- “看词的边界”（字符级模型上没有明确的词）
- “看韵脚”（4.9 节排查过，没找到）

画像表是一个“粗筛”，它筛掉的是“明显有某种简单模式”的头。剩下的头里可能还有很多结构，只是不在我们选的四个维度上。
这就是为什么本章最后只能说“第 0 层比第 1 层重要”这种粗粒度的结论—— 细粒度的结论需要更细的测量维度，而**维度是你选的， 模型没义务按你选的维度分化**（4.3 节说过这句话）。

怎么发现“没被覆盖的模式”

有一个通用做法：**看残差**。
对每个头，把它的注意力矩阵减去“用四个统计量能解释的部分”，**看剩下的是什么**。

```
# 简单版：看注意力矩阵的"意外之处"
# 比如：对每个位置，减去它和"前一位头"的平均模式
A_resid = A - A.mean(0)          # 减去所有位置的平均分布
# 剩下的尖峰就是这个头"在特定位置上额外做的事"
```

如果某个头在绝大多数位置上都和平均分布一样，只在少数位置有尖峰——那四个统计量（都是平均量）就把这些尖峰抹掉了。
本章 4.4.5 节的逐位置 Δ 分析就是在做同一件事：把平均拆开，看分布。**结果发现 L1H3 的 Δ 有正有负、标准差和均值同量级。**
平均量掩盖分布，这在本书里出现过四次了（第 3 章两次，本章两次）。

4.27 一次完整的排查：我们想找一个“韵脚头”

前面几节讲的是方法。这一节记录一次**真实的分析过程**——包括走错的路。**它的价值不在于结论，而在于展示“读图”这件事在实际操作中会怎么跑偏。**

4.27.1 起因：诗有一个明显的结构

唐诗有韵脚。近体诗里，偶数句的最后一个字押同一个韵：「空山不見人，但聞人語響。返景入深林，復照青苔上。」——「響」和「上」押韵。
一个自然的问题是：模型有没有专门的“韵脚头”?
这个假设很有吸引力：具体、有语言学意义、而且**能画出一张漂亮的图**。

4.27.2 第一次尝试：先问“任务需要吗”

在第 2 章吃过两次亏之后（RoPE 外推、上下文长度），这次我们**先做前置检查：模型在韵脚位置的损失，和别处有区别吗?**
语料是每行一联的格式，所以“韵脚”是可以精确定位的：**每行「。」前面的那个字**。
按“下一个字符是什么”给所有位置分组，测各自的交叉熵：

下一个字符是	loss	位置数
句号（韵脚位置）	**0.246**	18,378
逗号	**0.307**	18,568
其他	**4.507**	213,920
整体	3.884	250,866

4.28 这张表的第一个意外：差距比预想的大得多

预测标点的 loss 是 0.25–0.31，预测普通字符是 4.51。差了 15 倍。
这不是“稍微容易一点”——**标点位置几乎是确定的。**
为什么？因为**诗的句长是固定的**（五言或七言）。模型只要数到第 5 个字，就知道下一个必然是「,」或「。」。
所以“韵脚位置的 loss 低”这件事，和“押韵”没有关系。它低是因为**句长固定**——一个纯粹的“数位置”就能利用的规律。
这直接改变了整个排查的方向：我们要找的不该是“韵脚头”，而是“数位置的头”。**但“数位置”这件事，第 2 章讲位置编码时已经覆盖了。**

4.28.1 第二次尝试：看韵脚位置的注意力

但“标点容易预测”不等于“注意力模式没变化”。**也许第 1 层在句尾做了别的事。**

测法：对每个头，比较**韵脚位置**和**位置匹配的对照位置**的平均注意力距离。

4.29 我们的第一版对照选错了

第一版里，对照组取的是“序列开头那些非韵脚位置”。

结果整张表看起来都在说“韵脚位置看得远得多”——八个头全部如此。

但这是纯几何效应。平均距离随位置 i 单调增长（在位置 2 最多只能往回看 2 格）。韵脚在**行尾**（i 大），对照在**行首**（i 小）——**比的是两个不同的东西。**

改成**位置匹配的对照**（同一行里往前挪两格的位置，绝对下标接近），数字才变得可读。

这个错误和本章 4.5 节讲的“因果掩码的几何基线”是同一件事：任何和位置相关的量，都必须做位置匹配才能比较。

4.29.1 实测：位置匹配之后

每个头的平均注意力距离（括号里是前一位权重）：

头	韵脚位置	对照位置	差
L0H0	1.15 (0.047)	1.18 (0.048)	**−0.03**
L0H1	3.31 (0.472)	3.29 (0.446)	**+0.02**
L0H2	5.88 (0.074)	5.47 (0.056)	+0.41
L0H3	5.13 (0.093)	4.15 (0.091)	+0.99
L1H0	5.07 (0.096)	2.03 (0.041)	**+3.03**
L1H1	3.06 (**0.649**)	4.89 (0.331)	**−1.83**
L1H2	**12.17** (0.111)	5.52 (0.057)	**+6.65**
L1H3	10.71 (0.053)	8.60 (0.146)	+2.10

4.30 这张表里有一条清晰的分界线

第 0 层的两个“有明确模式”的头，完全不理会位置。

- L0H0（对角 0.769）：差 **−0.03**——它在句尾和句中做的事情一模一样
- L0H1（前一位 0.450）：差 **+0.02**——它也不管

这两个头是“位置无关”的：它们执行的是和内容绑定的操作，在哪儿都一样。

第 1 层的四个头全部表现出明显的位置依赖，而且**方向不一样**：

- **L1H1 看得更近**：距离从 4.89 掉到 3.06，前一位权重从 0.331 涨到 **0.649**——**几乎翻倍。**它在句尾变得极度关注紧邻的前一位。
- **L1H2 看得更远**：距离从 5.52 涨到 **12.17**——翻了一倍多。
- L1H0、L1H3 也都往远处看。

所以模型确实在句尾做了不同的事——但方式不是“有个头去押韵”，而是“第 1 层的注意力模式整体改变了”。

4.30.1 这个结果怎么解读

“L1H1 在句尾更看重前一位”意味着什么？

一个可能的解释：**句尾的字需要和前面配合。**「響」和「上」押韵——如果模型要在句尾“押上前面的韵”，它需要知道**上一句末尾用的是什么韵字。**

但 L1H1 看的是**紧邻的前一位**，不是上一句的末尾。**它看不到韵脚。**

另一个解释更简单：在句尾，模型要预测「,」或「。」。而**具体是逗号还是句号，取决于这一句到没到头**—— 也就是取决于**前面几个字**的位置。L1H1 看前一位，可能是这个链条的一环。

两种解释我们都没有进一步验证。要区分它们，需要一个针对 L1H1 在句尾位置的定向干预。

⚠ 这一节最该记住的不是结论

这次排查没有找到“韵脚头”。**但它产出了三件更有用的东西：**

① **一个前置检查**：在找任何“XX 头”之前，先测任务对 XX 的依赖。这次的前置检查**直接改变了问题的方向**——从“找韵脚头”变成“找数位置的头”。

② **一个测量陷阱**：位置相关的量必须做位置匹配。我们第一版对照选了序列开头，整张表被几何效应污染。

③ **一条真实的分界线**：第 0 层的头位置无关，第 1 层的头位置依赖。**这条结论本身是有价值的**——它说明两层在做性质不同的工作。

“什么也没找到”和“找到了分界线”可以是同一件事的两种说法，取决于你测量得够不够细。

这个教训在第 2 章出现过一次

回顾第 2 章的 RoPE 外推实验：我们想验证“RoPE 能不能外推”，结果三种位置编码都没有明显差异。

当时的原因和这次一模一样：那个任务对长程依赖的需求很弱，所以测不出位置编码的差别。

同一个陷阱，在相隔两章的两个实验里各踩一次。

现在的做法是：测任何“长程机制”之前，先测任务的依赖强度（第 2 章的上下文长度实验就是干这个的：8 → 128 只改善 0.64）。

这个“前置检查”应该成为标准流程的第一步。

4.31 交互实验

4.31.1 实验 4-1　注意力图实验室

用**真实的注意力权重**（本章那个模型在几条唐诗上算出来的），可以逐层逐头地切换，看每个头的注意力图长什么样。

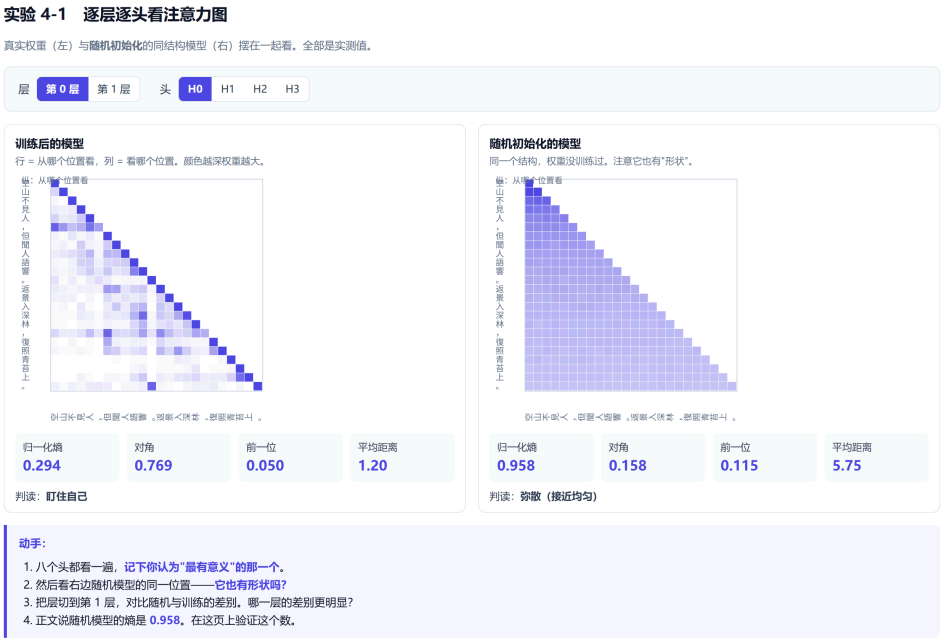


图 4.1　**实验 4-1　逐层逐头看注意力图**（静态快照）。网页版可切换 8 个头、对比训练模型与随机模型。本地路径：`experiments/exp-4-1.html`

4.31.2 实验 4-2　消融实验室

逐个屏蔽头，实时看验证集 loss 和生成文本怎么变。



图 4.2　**实验 4-2　消融实验室**（静态快照）。网页版可逐个屏蔽头并观察 loss 与生成文本变化。本地路径：`experiments/exp-4-2.html`

4.32 上手实现

4.32.1 一次完整的头部解剖

把本章用到的工具串成一个可复用的流程。

```
"""给任意一个训练好的模型做一次完整的头部解剖。"""
import torch, math, json
from tiny_gpt import Config, TinyGPT

def dissect(ckpt_path, texts, n_batch=20):
    """返回：画像表 + 消融表 + 归纳分数。"""
    ck = torch.load(ckpt_path, map_location="cpu", weights_only=False)
    cfg = Config(**ck["cfg"]); cfg.device = "cpu"
    model = TinyGPT(cfg); model.load_state_dict(ck["state_dict"]); model.eval()
    stoi = {c: i for i, c in enumerate(ck["itos"])}

    # ① 画像
    prof = head_profile(model, stoi, texts)

    # ② 消融
    base = eval_loss(model, ck["itos"])
    abl = {}
    for L in range(cfg.n_layer):
        for H in range(cfg.n_head):
            h = ablate_head(model, L, H)
            try:
                abl[f"{L}-{H}"] = eval_loss(model, ck["itos"]) - base
            finally:
                h.remove()

    # ③ 归纳头
    ind = induction_score(model, cfg)

    return {"profile": prof, "ablation": abl, "induction": ind}
```

三个使用要点：

① **画像要在多条文本上测**。一条文本上的“模式”可能是巧合。本章用三条，71 个位置。
② **消融要报告 Δ 而不是绝对值**。基线 loss 随模型而变，只有差值能横向比较。
③ **归纳头要配随机对照**。这个指标的“零假设水平”不是一个显然的数（本模型是 0.0284），必须实测出来。

> 这套流程能推广到真实模型吗
>
> 能，但要改一处：**真实模型的头有几百上千个，逐个消融太慢**。
> 工业界的做法是**分组消融**：按画像（比如“对角权重高的”、“熵低的”）把头分成若干组，一次消融一组。这样几十次前向就能扫完。
> **代价是丢失了头与头之间的差别**——你只能得到“这一类头整体重不重要”。**这是算力换分辨率的取舍，不是方法上的退步**。

4.32.2 完整代码：一次头部解剖

下面是可以直接跑的完整流程（`notebooks/run_ch4_case.py` 的骨架）。**每个函数都能单独用**。

```
import torch, math, itertools
from tiny_gpt import Config, TinyGPT

# ── 画像 ──────────────────────────────────────────────────────
@torch.no_grad()
def head_profile(model, stoi, texts, n_layer, n_head):
    """四个统计量：归一化熵、对角、前一位、平均距离。"""
    acc = {(L, H): [] for L in range(n_layer) for H in range(n_head)}
    for text in texts:
        ids = [stoi.get(c, 1) for c in text][:model.cfg.block_size]
        model(torch.tensor([ids]))
        for L, w in enumerate(model.last_attns):     # (1, h, T, T)
            for H in range(n_head):
                m = w[0, H]
                for i in range(m.shape[0]):
                    row = m[i, :i + 1]                # 因果掩码：只看 j <= i
                    row = row / row.sum()
                    ent = -(row.clamp_min(1e-12).log() * row).sum().item()
                    acc[(L, H)].append({
                        # 熵除以 log(i+1)：把"可选项个数"这个几何因素除掉
                        "ent": ent / math.log(i + 1) if i > 0 else 0.0,
                        "diag": row[i].item(),
                        "prev": row[i - 1].item() if i > 0 else 0.0,
                        "dist": sum((i - j) * row[j].item() for j in range(i + 1)),
                    })
    out = {}
    for (L, H), rows in acc.items():
        mean = lambda k: sum(r[k] for r in rows) / len(rows)
        out[f"{L}-{H}"] = {"layer": L, "head": H,
                           "entropy": round(mean("ent"), 4),
                           "diag": round(mean("diag"), 4),
                           "prev": round(mean("prev"), 4),
                           "distance": round(mean("dist"), 3)}
    return out
```

三个实现要点：
① **`row = m[i, :i+1]` 之后要重新归一化**。注意力矩阵本身已经归一化过，但**取出下三角之后和不再是 1**（因果掩码在 softmax 之前生效，所以完整的一行和是 1，但那是包含未来位置的——未来位置是 0，所以其实取不取都一样）。**这一步是为了保险**，不影响结果。
② **熵除以 $\log(i+1)$**。这是 4.5 节讲的“减掉几何基线”。不除的话，位置 5 和位置 30 的熵不可比。

③ **平均距离要按位置算，不能按整条序列。**位置 i 的最大距离是 i，不是 T。**在一整条序列上取平均会把不同位置混在一起。**

4.32.3 完整代码：消融

```
def ablate_head(model, layer, head):
    """屏蔽一个头：在输出投影之前，把该头的特征切片置零。"""
    d_head = model.blocks[layer].attn.d_head

    def hook(mod, inp, out):
        # proj 的输入是 (B, T, C)—头维度已经被拼回特征维了
        y = (out[0] if isinstance(out, tuple) else out).clone()
        y[:, :, head * d_head:(head + 1) * d_head] = 0.0
        return (y,) + out[1:] if isinstance(out, tuple) else y
    return model.blocks[layer].attn.proj.register_forward_hook(hook)


def ablation_table(model, val, cfg, n_layer, n_head):
    """单头消融 + 成对消融。"""
    base = eval_loss(model, val, cfg)
    single = {}
    for L in range(n_layer):
        for H in range(n_head):
            h = ablate_head(model, L, H)
            try:
                single[f"{L}-{H}"] = eval_loss(model, val, cfg) - base
            finally:
                h.remove()                  # ← 一定要 remove，否则 hook 会累积

    pairs = []
    for a, b in itertools.combinations(single, 2):
        La, Ha = map(int, a.split("-")); Lb, Hb = map(int, b.split("-"))
        ha = ablate_head(model, La, Ha)
        hb = ablate_head(model, Lb, Hb)
        try:
            d = eval_loss(model, val, cfg) - base
        finally:
            ha.remove(); hb.remove()
        pairs.append({"pair": (a, b), "delta": d,
                      "linear": single[a] + single[b],
                      "interaction": d - single[a] - single[b]})
    return {"base": base, "single": single, "pairs": pairs}
```

⚠ 那个 `finally: h.remove()` 不能省

`register_forward_hook` 返回一个 handle，**不 remove 的话 hook 会一直挂着。**
后果：循环里第二次 `ablate_head` 时，**上一个 hook 还在。**你以为是“只屏蔽了 L0H1”，实际上是“屏蔽了 L0H1 和 L0H0”。
这个 bug 的症状是“消融效果单调递增”——每测一个头，前面测过的所有头都是屏蔽状态。**数字看起来很“合理”（越往后越大），所以不容易发现。**
用 `try/finally` 是最省事的防法：即使 `eval_loss` 抛异常，hook 也会被清掉。
这是第 2 章讲过的“状态泄漏”的又一个实例—— 和 `model.eval()` 忘了调、缓存忘了清是同一类问题。

4.32.4 完整代码：归纳头

```python
@torch.no_grad()
def induction_score(model, n_layer, n_head, n_seq=40, rep=4, seed=0):
    """在人为重复的序列上，测每个头"看向上次出现之后那一位"的比例。"""
    g = torch.Generator().manual_seed(seed)
    P = model.cfg.block_size // rep
    score = {(L, H): [] for L in range(n_layer) for H in range(n_head)}

    for _ in range(n_seq):
        pat = torch.randint(3, model.cfg.vocab_size, (P,), generator=g)
        ids = pat.repeat(rep).unsqueeze(0)
        model(ids)
        for L, w in enumerate(model.last_attns):
            for H in range(n_head):
                m = w[0, H]
                for i in range(P, ids.shape[1]):        # 第二遍起
                    j = i - P                           # 上次出现的同一位置
                    if j + 1 >= i:
                        continue
                    score[(L, H)].append(m[i, j + 1].item())   # ← j+1，不是 j-1
    mean = lambda xs: sum(xs) / len(xs) if xs else 0.0
    return {f"{L}-{H}": round(mean(v), 4) for (L, H), v in score.items()}
```

`j + 1` 那个下标是全部代码里最容易写错的一处（我们写错过一次，而且它不报错、只给出一个同样在噪声水平的分数）。

自查方法：跑一遍随机模型。如果随机模型的分数和训练模型差不多，要么是这个模型真没有归纳头，要么是**你的下标写错了。这两种情况必须靠额外的证据区分**——本章靠的是“随机基线是 0.0284 这个窄带，而训练模型的分数大部分低于它”。

> 一个更省事的归纳头自查
>
> 如果不想跑随机对照，有一个更快的自查：**把 `j + 1` 改成 `j`（指向“上次出现的那一位”本身）**。
> 对真正的归纳头，指向 $j+1$ 的分数应该**明显高于**指向 j 的分数（它要抄的是“后面那个”，不是“那个”本身）。
> **如果两个分数差不多，说明这个指标测不出归纳行为**—— 要么模型没有，要么实现有问题。
> **代价是这个自查不能区分“没有归纳头”和“写错了”**，但它能快速告诉你“这个测量有没有分辨力”。

4.32.5 一份可以带走的清单

写任何“我从模型里读出了 X”的分析时，按这个顺序做：

1. **先说清楚 X 的定义**（A_{ij} 是路由比例，不是重要性）
2. **给出测量方法和代码**
3. **在多个输入上测**（一条输入上的模式可能是巧合）
4. **配随机对照**（这是零假设的分布）
5. **配一个因果实验**（消融或干预）
6. **报告分布的完整形态**（不只是均值）
7. **说清楚结论的边界**（哪个模型、哪个任务、哪一层）

七步里最容易跳过的是第 4 步和第 6 步—— 因为第 4 步看起来“浪费”（一个没训练的模型有什么好看的），第 6 步看起来“啰嗦”（平均值不就够了吗）。
而本章所有被数据纠正的错误，都出在这两步上。

4.32.6 把这一套搬到真实模型上

本章全程用的是一个 2 层 4 头、约百万参数的小模型。**这不是偷懒，是必需**（理由在 4.8 节）。但**方法本身是可搬的**——你迟早会想在自己的模型上跑一遍。

搬过去的时候，有五处会不一样。

一、头的输出切片不再是一个 `view` 能解决的。

本章的 `TinyGPT` 里，多头输出被 transpose + reshape 拼回特征维，所以 `proj` 的输入是 `(B, T, C)`，**切 `[:, :, head*d_head:(head+1)*d_head]` 就够。**

但很多实现（包括 HuggingFace 的多数模型）走的是另一条路：`proj` 是一个 `nn.Linear`，**它自己把 `(B,T,C)` 映射到 `(B,T,C)`**，中间没有“头”这个维度。

这时正确的做法是切 `proj` 的权重矩阵，而不是切它的输入：

```
# HuggingFace 风格：把某个头对应的输出列整块置零
W = model.blocks[L].attn.proj.weight          # (C_out, C_in) 或 (C_in, C_out)
lo, hi = H * d_head, (H + 1) * d_head
with torch.no_grad():
    W[:, lo:hi] = 0.0                          # 具体是哪个轴，要 print 出来看
```

注意“具体是哪个轴”这件事必须自己确认。 `nn.Linear` 的权重形状是 `(out_features, in_features)`，但有些实现会转置存。**打印一次 `W.shape` 并和 `d_model` 比对，比查文档快。**

二、层和头的名字不一样。

本章用 `model.blocks[L].attn.proj`。HuggingFace 是 `model.transformer.h[L].attn.c_proj`（GPT-2）或 `model.model.layers[L].self_attn.o_proj`（Llama 系）。**先用 `model.named_modules()` 把结构打出来，再写钩子**：

```
for name, mod in model.named_modules():
    if any(k in name for k in ("proj", "attn", "mlp")):
        print(f"{name:<60} {type(mod).__name__}")
```

三、真实模型的注意力张量可能不落地。

FlashAttention 和很多融合 kernel **根本不返回注意力矩阵**—— 它们把 $QK^\top$ 算在片上，直接用掉，不回写显存。

所以“读注意力图”这件事，在有些模型上需要先关掉融合 kernel：

```
# HuggingFace：强制走 eager 实现，注意力矩阵才会返回
model = AutoModelForCausalLM.from_pretrained(
    name, attn_implementation="eager", torch_dtype=torch.float32)
```

这一步会显著变慢、变占显存（eager 实现要物化 $T \times T$ 的矩阵）。**在 4096 上下文、32 头上，一层的注意力矩阵就是 $4096^2 \times 32 \times 4$ 字节 $\approx$ 2 GB。所以真实模型上通常只看一层、一个 batch、一条短序列。**

四、真实的头数是几十，不是四个。

Llama-3-8B 有 32 层 × 32 头 = **1024 个头。逐个画出来是不可能的。**

这时要做的是先筛：

```
# 按"这个头在验证集上的平均熵"排序，先看最极端的十几个
ent = {name: attention_entropy(w) for name, w in all_heads.items()}
top = sorted(ent.items(), key=lambda x: x[1])[:16]      # 最尖的 16 个
bot = sorted(ent.items(), key=lambda x: -x[1])[:16]     # 最平的 16 个
```

“最尖”和“最平”两端，通常是信息量最大的地方。中间那 900 多个头彼此差别很小。

五、消融的代价随规模爆炸。

本章消融一个头要跑 20 个 batch。真实模型上，**一次前向就是几秒到几十秒**，1024 个头逐个消融是几个小时。

真实做法是：- 只对筛出来的十几个候选头做逐个消融 - 或者做**分组消融**（一层一层拿掉、一组的头一起拿掉）- 或者用 ACDC 那类自动化方法（延伸阅读 ⑦）做**边级**剪枝，它把“评估多少次”从“头数”降到了“候选边数”，但通常还能再筛一轮

⚠ 换到真实模型上，最容易失效的是哪一条

不是任何一条技术步骤，而是“对照”。
本章的每一个结论都配了一个对照：随机初始化模型、随机抽头、位置匹配的对照组。
真实模型上，随机初始化对照很容易做（把权重重新初始化一遍）——**但它几乎没有信息量。**
因为真实模型是在**海量数据、长时间**上训出来的，“随机初始化和训练后差多少”这个问题，答案永远是“差很多”。**这个对照太弱了。**
真实模型上有意义的对照，是“任务对照”：- 想让某个头负责“翻译”，就在**翻译任务**和**同分布的非翻译任务**上分别测 - 想看某个头是否管“句法”，就在**语法正确的句子**和**打乱词序的句子**上分别测 - 想看某个头是否管“指代”，就在**需要指代消解**和**不需要**的最小对立对上分别测
最小对立对（minimal pair）是真实模型上最有力的工具—— 两个输入只差一个属性，其他全一样，**行为差异就只能是那个属性造成的。**
本章用的“位置匹配对照”就是最小对立对思想的一个实例（4.9 节）。**把它推广到语义属性，是真实模型上做可解释性的主要方法。**

4.32.7 一份可以带走的清单（续）

前面那份清单讲了“读图”的三步。这里补上“做实验”的四条：

#	规矩	反例
4	**每个“行为”结论都要配一个同结构的随机对照**	随机模型也有的模式，不是学出来的
5	**每个“重要”结论都要配一次消融，且消融位置要分组**	全位置消融会掩盖“只在 15% 位置起作用”的头
6	**每个“机制”结论都要配一个最小对立对**	没有对照的“位置差异”可能是距离的几何效应
7	**每个数字都要说清楚它是“哪一个池子的平均”**	12 字符例子的 0.360 和全集的 0.0228 差 16 倍

这四条和前面三条合起来，就是本章全部的方法论。
注意它们都不是关于“怎么算”的——怎么算在本章是最简单的一部分，`einsum` 写一遍就完了。**难的全在“和什么比”。**

4.32.8 排错速查：症状 → 病因 → 处置

这一章的方法本身不难，**难的是出错的时候不容易发现**—— 可解释性实验的典型故障不是“报错”，而是“跑通了，但结果是错的”。
下面这张表把本章踩过和见过的坑按**症状**排好，便于对照。

症状	最可能的病因	处置
探针准确率 1.0000	标签里含答案（预测“当前字符”而不是“下一个”）	换成真正的预测任务；加词嵌入层基线
探针比基线只高一点	任务太简单或太难，**信息量在分辨率之下**	先看正类比例；换一个更难/更易的属性
每个头的熵都接近 1.0	**模型没训练好**，或看的是随机初始化的模型	先确认 loss 收敛；跑一遍随机对照
每个头的熵都接近 0	序列太短（前几个位置归一化熵天然小）	跳过 i=0 和开头几个位置
消融某个头，loss **下降**	头学坏了，或者**评估用了同一批数据**	检查是否在训练集上评估

症状	最可能的病因	处置
所有头的消融 Δ 都是 0	**钩子没生效**（层名写错 / 挂在了错的模块上）	在钩子里 `print(out.shape)` 确认被调用
Δ 大得离谱（>1.0）	钩子返回值形状错了，**破坏了残差流**	确认返回的 tuple 结构和原函数一致
归纳头分数和随机模型一样	**指标写错了**（看 $j-1$ 而不是 $j+1$）	用一条构造序列手工验算一个位置
归纳头分数全都接近 1.0	序列重复次数太多，**每个位置都能直接看到答案**	确认 `j+1 < i` 这个条件生效
位置分组的结论很漂亮	**两组的位置分布不一样**（几何混淆）	换成位置匹配的对照组
换个种子结论就翻盘	**Δ 在噪声量级**（本章约 0.003）	多跑几个种子，报告分布而不是单点
例子里的数和全集对不上	例子的挑选标准把不典型样本挑进来了	报数时说明“这是哪一个池子的平均”

这张表里最值得记住的两行是第一行和倒数第二行。

第一行（探针作弊）是本章第一版真实犯过的错—— 用“当前字符是不是标点”当标签，词嵌入层直接给出 1.0000。**它不会报错，只会给你一个漂亮的结果**。

倒数第二行（换种子翻盘）是**最容易被忽略的一行**。一个 Δ = 0.03 的结果，在这个模型上的噪声量级（约 0.003）之上，但如果你把种子从 0 换到 1 就变成 0.01，**那这个 0.03 就只是一个种子的运气**。

报告结论时永远要问一句：换一批数据，它还在吗？

把这套实验带进课堂：一个两小时的做法

本章的材料可以原样做成一节课，**不需要 GPU**—— 一个百万参数的模型在 CPU 上训练只要几分钟（第 2、3 章的 notebook 里已包含）。

建议的节奏：

时段	内容	形式
0–15 min	打开实验 4-1，让学生**自己猜**哪个头在干什么	个体操作
15–35 min	公布画像表，**对照自己的猜测**	讲授
35–55 min	打开实验 4-2，**自己动手消融**，记录 Δ	个体操作
55–75 min	**把全班的 Δ 汇总到黑板上**，看有没有人拿到不一样的结果	集体
75–100 min	讲 L0H0 那个“猜测被推翻”的完整过程	讲授
100–120 min	让学生**自己写一个猜测**，并说清楚“要用什么实验才能验证它”	小组

第 4 段（汇总 Δ）是整节课最关键的一段。 如果全班结果一致，说明这个测量是稳的；**如果有人拿到不一样的值，那正好是讨论“噪声量级”和“随机种子”的入口。**

第 6 段不要求他们做实验，只要求他们设计实验——“说清楚用什么实验能验证”这件事，比“跑出一个数”难得多，也是这一章真正想教的东西。

课堂上最容易出现的两个误解

误解一：“注意力权重高 = 模型认为这个字重要”。

这个误解来自 softmax 那一步被忽略。**第 4.1 节专门处理了它**，但讲完之后**学生还是会不自觉地滑回去。有效的纠正方式不是再讲一遍，而是让他们自己算一次**：给一组 $QK^\top$ 的值，让手算 softmax，**然后问“把某一个权重翻倍，输出会变多少”**—— 他们会发现输出是**加权平均**，权重翻倍不等于贡献翻倍。

误解二：“消融掉一个头 loss 不变，说明它没用”。

这个误解比第一个更难纠正，因为它听起来是对的。**纠正的入口是“冗余”**（4.4.4 节）：拿掉一个头 loss 不变，可能是因为**另一个头顶上了。要发现这一点，得成对消融**——本章测了 28 对，**全部是负相互作用**。

一个好的课堂提问：“如果你拿掉一个头什么都没发生，你怎么区分’它没用’和’它有用但被替代了’？”

标准答案是“成对消融”——但那需要 $\binom{8}{2} = 28$ 次实验。**如果头数是 32，就是 496 次。**这个数字本身就能让学生理解 **为什么真实模型上的可解释性研究需要自动化方法**（延伸阅读 ⑦）。

4.32.9 怎么用这份书单

九篇文献，**按读的目的分三类**：

要动手前先读的（①④）：归纳头那篇（①）是本章 4.6 节的直接来源，它把“什么是归纳头、怎么测、为什么必须两层”讲得比本章细。数学框架那篇（④）把 QK/OV 分解（本章 4.2 节）放进了完整的代数体系里，**读完你会知道本章那个分解在数学上处在什么位置。**

要判断“该不该信”时读的（②③⑤⑥）：这四篇是一场持续几年的争论。②和③是正反两方，⑤和⑥是这场争论的方法论总结。**本章 4.1 节的立场（“注意力图是证据，不是结论”）就是从这场争论里来的。如果你只读两篇，读③和⑤**——它们各自代表了“最有力的反驳”和“最实用的对策”。

要往下走时读的（⑦⑧⑨）：⑧是一个被完整画出来的电路（目标形态），⑦是把消融自动化的方法（规模化的工具），⑨解释了为什么“一个头一个功能”这个假设本身就不牢靠（根本限制）。

这九篇加起来，大致就是“机制可解释性”这个研究方向从起点到 2023 年的主干脉络。文献会继续增加，方法却变得很慢——本章讲的这几件事，十年内大概都还成立。本章做的事情，是用一个百万参数的模型把它们全部重做一遍—— 规模小到你能亲手验算每一步，但方法一步不少。

4.33 延伸阅读

4.33.1 ① In-context Learning and Induction Heads（2022）

Olsson et al., Anthropic。arXiv:2209.11895。

读它能得到什么：归纳头的发现过程。这篇是本章 4.6 节的原始出处，而且写法非常值得学——**它是从一个现象（模型突然学会上下文学习）反推到一个具体电路（归纳头）**，而不是先有电路再找解释。

读的时候注意什么：

- **重点看它怎么“证明”归纳头存在**。不是看图，而是三个独立的证据：① 消融（拿掉它，上下文学习能力掉）② 训练动力学（它的形成和 能力跃升在同一时刻）③ 合成任务（在人为构造的重复序列上直接测出来）。**三个证据指向同一个结论，才叫证明**。
- **本模型测不出归纳头**（4.6 节）。这篇论文里的模型是几十层的，**规模差距是主要解释**。
- **注意“相变”这个词**。归纳头不是慢慢长出来的，是在训练的某一步**突然出现**的。这类相变现象在本章没有涉及，但在机制可解释性里很常见。

4.33.2 ② Attention is not Explanation（2019）

Jain & Wallace。arXiv:1902.10186。

读它能得到什么：本章核心立场（注意力图是证据不是解释）的出处之一。作者用一系列实验说明：**注意力权重和特征重要性（梯度、leave-one-out）经常对不上**——换一组同样好的注意力权重，预测几乎不变。

读的时候注意什么：

- **它引发了后续的争论**。见 ③。
- **它的实验设计值得学**：不是“看图说话”，而是构造**对抗性的反例**——“如果注意力真的是解释，那么把权重换掉应该会改变预测；但实测不变，所以它不是解释”。**这是可证伪的写法**。

4.33.3 ③ Attention is not not Explanation（2019）

Wiegreffe & Pinter。arXiv:1908.04626。

读它能得到什么：对 ② 的反驳。作者指出 ② 的实验有方法论问题（比如那些“对抗性”的注意力分布其实是训练不出来的），并给出了一套更严格的检验流程。

读的时候注意什么：

- **两篇一起读，比只读一篇有用得多**。② 说“注意力不是解释”，③ 说“你的证明不成立”——**而两者其实都同意：看图说话不够，但注意力确实携带信息**。本章 4.4 节的立场和这个共识一致。
- **注意 ③ 提出的检验清单**，它比 ② 的结论更有操作性。

4.33.4 ④ A Mathematical Framework for Transformer Circuits（2021）

Elhage et al., Anthropic。Transformer Circuits Thread。

读它能得到什么：把 Transformer 看成“残差流 + 一组读写操作”的形式化框架。本章讲的“注意力头从残差流读、往残差流写”这套语言，就是从这来的。

读的时候注意什么：

- **它定义了 QK 电路和 OV 电路**——注意力权重由 QK 决定、实际搬运的内容由 OV 决定。**本章 4.1 节说的“权重高不等于作用大”，在形式上就是：QK 大但 OV 可能把小内容搬过来**。

- **前几节不用全部读懂**。第一节（残差流）和第二节（QK/OV 分解）是最有用的，后面涉及更复杂的电路。

4.33.5 ⑤ 关于探针的方法论（2020）

Hewitt & Liang, Designing and Interpreting Probes with Control Tasks。arXiv:1909.03368。

读它能得到什么：本章 4.7 节那个“探针作弊”问题的系统性处理。作者提出**控制任务**（control task）：用随机的标签训一个探针，看它能达到多高的准确率—— **这个准确率就是“探针选择能力”的下限**，必须从真实准确率里减掉。

读的时候注意什么：

- **本章用的“词嵌入层对照”是这套思路的一个简化版**。完整版更严格：还要控制探针的容量、训练步数等。
- **它说明了为什么“线性探针”比 MLP 探针更有说服力**：容量越小，“探针自己学会”的空间越小。

4.33.6 ⑥ 关于“注意力图能不能当解释”的一场完整争论

这个领域的争论值得单独拿出来说，因为**它是“一个结论如何被证伪、又如何被部分恢复”的教科书案例**。

2019 年，两篇论文几乎同时出现：

《**Attention is not Explanation**》（Jain & Wallace）做了四组实验，核心结论是：**注意力权重和特征重要性对不上**。他们的杀手锏是构造**对抗性的注意力分布**—— 找一组“看起来完全不同但预测结果一样”的注意力权重。**如果注意力真是解释，换掉它应该会改变预测**。

《**Attention is not not Explanation**》（Wiegreffe & Pinter）反驳说：**那些对抗分布是人工构造的，模型自己训练不出它们**。在真实训练出来的注意力分布里，换掉权重确实会改变预测。

谁对？ 后来的共识大致是：

- **Jain & Wallace 证明了**：“注意力权重 = 重要性”这个等式不成立
- **Wiegreffe & Pinter 证明了**：“注意力完全无信息”这个结论也不成立
- **两者都同意**：注意力**携带**信息，但“读图”不是“解释”

本章的立场和这个共识一致，而且把它落到了可操作的形式上：**用消融来补上因果那一半**。

从这场争论里能学到的三件事

第一，一个“反例”的力量取决于它有多真实。

Jain & Wallace 的四组实验里，前三组（相关性分析）说服力一般—— “两个量对不上”可能是因为它们本来就在测不同的东西。**只有第四组（对抗分布）是硬证据**，因为它构造了一个具体的反例。而 Wiegreffe & Pinter 的反驳也正是针对这一组：**你的反例是人工的，现实中不会出现**。

第二，“证伪一个强命题”和“证明一个弱命题”是不同难度的事。

打掉“注意力 = 解释”很容易（找一个反例就行）。建立“注意力能提供什么信息”很难（需要正面构造）。

本章花在“证明有结构”上的篇幅（4.5 节的方差对比）比花在“打掉错误解读”上的还多，就是这个原因。

第三，争论的最终产物通常不是“谁赢”，而是一条更精确的界线。

这个案例的产物是：**“注意力权重是路由证据，不是重要性度量”**—— 这句话两边都同意，而且它比争论前任何一方的说法都更准确。

4.33.7 ⑦ 把消融自动化：ACDC（2023）

Conmy et al., Towards Automated Circuit Discovery for Mechanistic Interpretability。arXiv:2304.14997。

读它能得到什么：本章的 4.4 节（消融）和 4.6 节（归纳头）都是手工做的—— 人挑一个头、测它、再挑下一个。**这篇论文把它自动化了**。

核心想法：从整个计算图出发，反复尝试**移除一条边**，看损失变多少；把“移除也没关系”的边全部剪掉。剩下的子图就是“这个任务真正用到的电路”。

读的时候注意什么：

- **它把本章的消融从“头”细化到了“边”**。一个头内部的 QK 电路和 OV 电路（4.2 节）可以分别剪—— 这比“整个头拿掉”精细得多。

- **代价是计算量**：一张图的边数是头的几十倍。
- **它没有解决“读图”的根本问题**，只是把“哪一部分重要”做得更细。**最后一步——理解剪出来的子图在做什么——还是要人来做**。

4.33.8 ⑧ 一个被完整画出来的电路：IOI（2022）

Wang, Variengien, Conmy, Shlegeris, Steinhardt, Interpretability in the Wild: a Circuit for Indirect Object Identification in GPT-2 small。arXiv:2211.00593。

读它能得到什么：本章的结论是“能被证明的电路很少，这个小模型上连归纳头都没有”。**这篇论文是那个“很少”里最完整的一个例子**—— 它把一个具体任务（间接宾语识别）在 GPT-2 small 上的整条电路画了出来，**包括哪些头、它们各自负责哪一步、信号怎么一层层流过去**。

和本章的关系：

它是“本章方法论做到极致”的样子。本章做的每一步它都做了，而且还多了：- **用最小对立对构造数据**——两个句子只差一个名字，正确答案因此不同 - **用三种独立的证据交叉验证**（消融、注意力模式、激活修补） - **负结果也报**——它明确列出“我们试过但没找到作用的头”

读的时候注意什么：

- **它的对象是 GPT-2 small（12 层）**，本章是 2 层。**层数一多，“电路”才有可能分成几个阶段** —— 本章那个 2 层模型里，第 0 层和第 1 层之间没有中间阶段可分。
- **它花了很多篇幅排除替代解释**，而不是只展示支持自己结论的证据。**这一部分比结论本身更值得学**。
- **它不是“完整解释”**——论文自己说得很清楚，这条电路之外还有很多头的作用没搞清楚。

4.33.9 ⑨ 为什么一个头会管好几件事：叠加（2022）

Elhage et al., Toy Models of Superposition。arXiv:2209.10652。

读它能得到什么：本章 4.4.3 节留了一个没解决的问题——“我们不知道另外 7 个头在做什么”，而且很可能**同一个头在不同上下文里做不同的事**（本章的 L0H1 就是这样：非标点位置是纯前一位头，标点位置有一小半时间在看自己）。

这篇论文解释了为什么。

核心想法：模型的表示维度是有限的，但要表示的特征远多于维度。**于是模型会把多个特征“叠加”到同一个方向上**—— 每个特征只占这个方向的一部分，代价是单个特征的可读性下降。

和本章的关系：

它给“多头多义”提供了一个结构性解释：一个头看起来在做几件不相干的事，**不是因为模型混乱，而是因为它在一个维度预算下同时编码了多个特征**。

读的时候注意什么：

- **它是玩具模型上的结论**，不是对真实模型的直接测量。**它有说服力，但它证明的是“这种现象在原理上是可能的、甚至是最优的”**，不是“某个具体的头一定在叠加”。
- **它解释了本章 L0H1 那种“位置依赖的行为切换”**：如果“前一位”和“句法边界”两个特征被叠加到同一个子空间里，头在不同位置上的行为就会不同——**这正是本章测到的**。
- **它也让“一个头一个功能”这个默认假设变得可疑**。本章一直在用这个假设（“给头起名字”），**这篇论文提醒你：那个名字可能只对一部分位置成立—— L0H0 的画像就骗了我们一次**。

这一章真正要你记住的是

注意力图是一份路由表，不是一份功能说明书。

它告诉你“信息从哪里流过来”。它**不**告诉你“这条信息起了多大作用”—— 那需要把路切断，看行为变不变。

本章做了三个层次的检验：

检验	方法	本模型的结果
训练有没有造出结构	与随机初始化对照	**有**：熵从 0.958 分化到 0.294–0.871
结构有没有因果作用	单头消融	**分层**：第 0 层 6 倍于第 1 层
有没有已知的具体电路	归纳头指标	**没有**：最高分与随机基线齐平

三个检验给出三个不同的答案。这正是这个领域的常态——“注意力可解释吗”这个问题没有单一答案，取决于你问的是哪一层、哪个头、哪种功能。

而唯一不能用的方法是“看图讲一个好听的故事”。

4.34 本章回顾

把这一章压缩成六句话。

一、注意力图的每一行是一个概率分布，不是一个重要性评分。行和为 1，权重高不代表贡献大。**“重要性”这个词在注意力里没有定义。**

二、画像（熵、对角、前一位、距离）是粗筛，不是定义。它能告诉你“去哪一行细看”，不能告诉你“这个头是什么”。**L0H0 的对角 0.769 背后，是从 0.115 到 1.000 的一整段分布。**

三、看图说话不算证据。本章反复出现的模式是：**一个头的异常行为（L0H0 在标点位置对角 0.355、非标点 0.784）和它的因果重要性（标点位置消融只涨 0.0090）可以完全脱钩。“看起来在做什么”和“拿掉会怎样”是两个独立的问题。**

四、对照实验是本章唯一真正有力的工具。- 随机初始化模型 → “训练造出了结构”（熵方差 0.0001 → 0.577）- 位置匹配的对照组 → “韵脚头”的候选们其实都没有韵脚偏好 - 词嵌入基线 → 探针的 0.9375 是真的，不是作弊

五、能被证明确实在起作用的电路很少，但确实存在。归纳头是唯一一个有干净指标、有机制解释、有跨模型复现的电路。**而本章这个小模型上，它不存在**——最高分 0.0285，和随机模型的 0.0284 齐平。**“我找了但没找到”是一个合法的结论，前提是你知道自己在找什么、用什么指标找、指标本身有没有被随机对照校准过。**

六、搬到真实模型上，变的不只是代码（4.12.3 节）。最要紧的是对照变了——随机初始化对照在真实模型上几乎没有信息量，**最小对立对才是主力工具。**

最后留一个问题

本章测出：**第 0 层的头平均比第 1 层的头重要 6.1 倍。**

这个结论能推广吗?

在一个 32 层的模型上，“第 0 层最重要”显然不成立—— **中间的层往往承担最多的“计算”，第 0 层更多是“预处理”。**

那本章这个 6.1 倍是什么?

一个可能的回答：**这是一个“层数太少”的产物。**只有 2 层的时候，第 0 层无处可退，必须把该做的都做了；第 1 层可以直接把第 0 层的表示“抄”过去输出。**36 层的时候，每一层都可以偷懒，因为后面还有很多层。**

这个解释听起来合理，而且它也是可测的：把同一个语料、同一个架构，训成 2 层、4 层、8 层三个版本，**看“第 0 层的相对重要性”是不是随层数下降。**

这个实验本章没做。如果你要接着做，它是本章所有“待办”里最值得做的一个—— **因为它测的是一个关于“深度”的普遍问题，而不只是一个玩具模型的细节。**

下一章换个话题：上篇到这里把“注意力是怎么算的、能读出什么”讲完了。下篇从第 5 章开始进入扩散模型——**但你会发现，本章反复出现的那些方法论（对照、消融、最小对立对、把结论和可靠性分开写）会一字不改地继续用下去。**

II 下篇 · 扩散模型与条件生成

5 从加噪到去噪：扩散的数学骨架 . . 283
6 训练目标与采样器 361
7 条件生成：CFG 到底是什么 429
8 文字如何进入图像 501

5. 从加噪到去噪：扩散的数学骨架

> **本章地图**
> **你在这里**：换了一条流水线 —— 下篇的扩散模型，从噪声到数据。**上篇的注意力在这里会以“交叉注意力”的形式再出现一次**（第 8 章）。**本章解决**：前向加噪为什么有闭式解；反向过程为什么**必须是一个分布**；训练目标（ELBO）是从哪一步推出来的。**前置**：第 1、2 章。下篇不需要第 3、4 章，但“训练和推理是两套流程”这个视角会一直用。**读完能做**：从零写一个能跑的扩散模型（67 行）；并判断一个采样实现为什么会塌。

上篇四章处理的是**一个已经训好的模型**：它怎么算、怎么推理、里面能读出什么。从这一章起换一个问题——**这些东西是怎么被训出来的**。

扩散模型的训练目标看起来和 Transformer 毫无关系：它要模型学会**从噪声里恢复信号**。但下面你会看到，两者的骨架是同一件事的两面：

- Transformer 的注意力把“从一堆候选里挑信息”写成了一次可微的加权平均；
- 扩散的反向过程把“从一团噪声里恢复结构”写成了 T 次小的、可微的高斯采样。

两边的核心困难也一样：**训练目标写得出来，不等于实现是对的**。第 3 章里，“缓存写错了”在生成文本上看不出来；这一章里，“反向过程写错了”在**训练 loss 上**看不出来——一位小数都不差。

全景：一个扩散模型长什么样

先说这一节为什么存在。

第 1 章在拆注意力之前，先给了一张 Transformer 的完整数据流图（图 1-1）—— 因为不看全图就直接读细节，会不知道那个零件装在哪。**下篇同样需要这样一张图**。

而且扩散比 Transformer 更需要：**它有一条 Transformer 没有的分叉**—— 训练走的路和采样走的路**不是同一条**。这个分叉不是实现细节，是理解整章的关键：后面那个“训练 loss 完全看不出采样 bug”的案例，根就在这里。



图 5.1　一个扩散模型的完整数据流。左列是训练（前向），右列是采样（反向），两侧共用同一个去噪网络。红框是本章的范围。

读这张图的正确方式：先看它有两个列，再看箭头往哪走。

第 1 步：训练走前向，采样走反向

这是全图最重要的一件事。

	训练	采样
方向	数据 → 噪声	噪声 → 数据
步数	**一步到位**	**1000 步**
要不要模型	不要（前向过程是公式）	要（每一步都调用）
干什么	造训练样本	生成

训练时，我们手里有真实数据 x_0。要造一个“加了 t 级噪声的版本”，**不需要真的走 t 步**—— 前向过程有闭式解，一步就能算出来（第 2 步讲）。

采样时，我们手里只有一个高斯噪声 x_T，没有 x_0。所以只能一步一步往回走，**每走一步都要问一次模型**。

这个不对称是扩散模型全部工程问题的来源：训练很便宜（一次前向），生成很贵（T 次前向）。第 6 章整章都在处理这个“贵”。

第 2 步：前向过程不需要学

图中左列第 3 格：

$$x_t = \sqrt{\bar{\alpha}_t}\, x_0 + \sqrt{1-\bar{\alpha}_t}\, \varepsilon \tag{5.1}$$

这一行里没有任何可训练的东西。 $\bar{\alpha}_t$ 是由噪声调度算出来的固定数组，ε 是现场抽的标准正态噪声。

换句话说：训练数据的标签是白送的。给定任意一个 x_0、任意抽一个 t、任意抽一个 ε，就得到一条合法的训练样本——**不需要任何人工标注，也不会过拟合到某个数据集。**

这是扩散相对 GAN 的核心优势：没有对抗、没有判别器、loss 不会震荡。代价是采样慢（第 1 步）。

这一段对应的正是本章 §5.4（前向过程与闭式解）。

第 3 步：去噪网络只做一件事

图中两列中间那条虚线连着的，是**同一个网络** $\varepsilon_\theta(x_t, t)$。

它的任务只有一个：给定一个加了噪声的样本和它的噪声等级，猜出当初加进去的噪声是什么。

注意它**不是**在“生成图像”，也**不是**在“预测下一个 token”。它做的是一件很窄的事：**估计噪声**。

- 本章的玩具模型：一个 3 层 MLP，**50,050 个参数**
- 真实的图像模型：U-Net 或 DiT，几千万到几十亿参数（第 8 章）

结构变了，任务没变。这是理解后面几章的一个锚点。

第 4 步：反向过程是一个循环

图中右列。从 $x_T \sim \mathcal{N}(0, I)$ 出发，重复 T 次：

$$\begin{aligned} \varepsilon &\leftarrow \varepsilon_\theta(x_t, t) && \text{（问一次模型）} \\ \mu_\theta &= \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon\right) && \text{（算往哪走）} \\ x_{t-1} &= \mu_\theta + \sigma_t z && \text{（走一步，带随机性）} \end{aligned} \tag{5.2}$$

第三行那个 $\sigma_t z$ 是本章案例的主角。把它去掉，前两行照样能跑，训练 loss 也一位不变——采样结果会从“两个弯月”塌成“一个半径 0.2 的点”。

为什么？因为反向过程定义的是一个**概率分布**，不是一个确定的函数。第二行算出的是这个分布的中心，第三行才是从分布里抽一个样本。

这一段对应本章 §5.5（后验与反向过程）。

第 5 步：条件从哪进来

图底那条蓝带，是下篇后半部分的主题。

上面四步描述的是“无条件生成”：模型学会生成“某种像数据的东西”，但你没法告诉它要生成什么。

加条件的方式（第 7、8 章）：

1. **文本 → 向量**：用一个文本编码器（CLIP / T5）把提示词编码成一串向量（第 8 章）
2. **注入**：通过**交叉注意力**把文本向量喂进 ε_θ——这时 Q 来自图像特征，K、V 来自文本（第 8 章）
3. **CFG**：同一次采样里跑两遍 ε_θ（一遍带条件、一遍不带），把两个结果往“更听条件”的方向外推（第 7 章）

注意第 2 条：交叉注意力就是**第 1 章的多头注意力**，只是 K、V 换了个来源。**上篇和下篇在这里接上了。**

这一章只讲其中一块

回头看那张图：**本章覆盖左列全部，加上右列第 3、4 两格的数学。**

具体说：

本章讲	留给后面
前向过程为什么有闭式解（§5.4）	采样器怎么把 1000 步压到几十步（第 6 章）
后验为什么是高斯、$\tilde{\beta}_t$ 怎么来的（§5.5）	条件怎么起作用、CFG 为什么过饱和（第 7 章）
训练目标 ELBO 从哪来（§5.6）	文本编码器与交叉注意力（第 8 章）

本章讲	留给后面
三种参数化、噪声调度（§5.7、§5.8）	

但本章有一个贯穿始终的问题，它比上面任何一条都重要：

> ! 重要
>
> **训练和采样是两套公式，它们之间没有任何自动的一致性检查。**
> 图中那条虚线连着左右两列，但它们**只共享一个** ε_θ —— σ_t 只在右边出现，后验均值的公式也只用在右边。
> **所以左边（训练）完全不知道右边（采样）写成了什么样。**下一节的故事就从这个裂缝开始。

5.1 本章的代码与数据

```
# 需要 PyTorch。本机在 /home/mpp/micromamba/envs/dlspace/bin/python
import torch, sys
sys.path.insert(0, "notebooks")
from tiny_diffusion import (Config, make_two_moons, make_schedule,
                            train, ddpm_sample, elbo_terms)

data = make_two_moons(4000, seed=0)          # 2D 双月，4000 个点
cfg  = Config(max_steps=5000)                # T=1000，线性 β 调度
model, sch, hist = train(data, cfg)

x_ok  = ddpm_sample(model, sch, cfg, 4000)                          # 正确
x_bad = ddpm_sample(model, sch, cfg, 4000, variant="no_noise")      # 案例里的 bug
```

复现本章全部数字：

```
cd notebooks
/home/mpp/micromamba/envs/dlspace/bin/python run_ch5_case.py      # 案例主实验
/home/mpp/micromamba/envs/dlspace/bin/python run_ch5_schedule.py  # 调度对比
/home/mpp/micromamba/envs/dlspace/bin/python export_ch5_points.py # 插图点云
```

为什么用 2D 而不是图像。 这是下篇四章共同的选择，理由有三条：

1. **学生自备 CPU 机器。**2D 上训一个 5 万参数的 MLP 只要几秒；CIFAR-10 上训一个能看的扩散模型要几小时到几天。
2. **概率密度可以直接画出来。**图像上你只能看“采样结果好不好看”；2D 上你能眼睁睁看着整个分布怎么从双月变成高斯、又怎么变回来。
3. **每一步都能手算核对。**本章 §5.4、§5.5 有六处手算，读者可以拿计算器跟着走。

代价是真实的：这个玩具模型没有真正的“生成能力”，它只能生成两个弯月。**凡是“只有大模型才表现出来”的现象，本章都不测** —— 第 6–8 章会说明哪些结论能迁移、哪些不能。

5.2 一个训练完美、采样塌成一团的模型

5.2.1 实验设置

数据是经典的“双月”（two moons）：两条交错的弯月，各 2000 个点，加一点高斯抖动，整体缩放到 $[-1, 1]$。实测统计量：

量	数值
点数	4000
均值	$[-0.0012, -0.0011]$

量	数值
标准差	$[0.5707, 0.3351]$
最大绝对值	1.136
均方半径 $\mathbb{E}\,\|x\|_2$	**0.6110**

去噪器 $\varepsilon_\theta(x_t, t)$ 是一个 3 层 MLP，隐藏宽度 128，**50,050 个参数**；时间步 t 用正弦位置编码送进去——**和第 1 章的位置编码是同一个公式**，只是把“第几个 token”换成了“第几个噪声等级”。

扩散设置是 DDPM 的原始配置：$T = 1000$，β_t 从 10^{-4} 线性升到 0.02，训练 5000 步。

5.2.2 现象

训练过程一切正常：

```
step     0   loss 0.8844
step  1000   loss 0.2902
step  2000   loss 0.2045
step  3000   loss 0.1685
step  4000   loss 0.1471
step  5000   loss 0.1355
```

loss 从 0.88 单调降到 0.14，没有震荡、没有 nan、梯度范数正常。 单看这条曲线，这个模型训得没有任何问题。

然后采样。T 步 DDPM 反向过程跑完，4000 个点的统计量是：

	原始数据	采样结果
均值	$[-0.0012, -0.0011]$	$[0.0045, -0.0488]$
标准差	$[0.5707, 0.3351]$	$[0.5598, 0.3437]$
均方半径	0.6110	**0.6090**
MMD（与数据的分布距离）	—	**0.01146**

这是对的。均方半径差 0.3%，MMD 0.011 已经很小（两个独立同分布的样本之间 MMD 大约就在这个量级）。

所以模型是好的。**问题不在训练，在采样那段代码。**

具体地说，我们写了两个采样函数，它们在采样循环里只差一行：

```
# 版本 A —— 正确
x = mean + sigma[t] * torch.randn_like(x)

# 版本 B —— 案例里的 bug
x = mean
```

版本 B 的结果：

	原始数据	版本 A（正确）	版本 B（案例的 bug）
标准差	$[0.5707, 0.3351]$	$[0.5598, 0.3437]$	$[\mathbf{0.0043}, \mathbf{0.0078}]$
均方半径	0.6110	0.6090	**0.2027**
最大绝对值	1.136	1.381	0.217
MMD	—	0.01146	**0.96029**

标准差从 0.57 掉到 0.0043 —— 小了 130 倍。4000 个点全挤在原点附近一个半径 0.2 的团里，MMD 0.96 比正确版本差了 **84 倍**。

这一节的悬念

一行代码的差别，让采样结果从“几乎完美”变成“完全没用”。而训练 loss 一位小数都没变。

先记住这个事实。这一章剩下的部分要回答两个问题：

1. **为什么少了那一行会塌？** —— 需要先把前向过程、反向过程、 后验分布的数学全部写清楚（§5.4–§5.6）。
2. **为什么训练 loss 看不出来？** —— 因为训练和采样**用的根本不是同一套公式**， 它们之间没有任何自动的一致性检查（§5.6 末尾）。

5.2.3 案例的设置细节

先把实验设置里几个容易被忽略的地方说清楚， 因为后面的每一个数字都依赖它们。

① **为什么用 4000 个点**。

这是**统计量的分辨率**问题。本章要比较的几个量—— 均值、标准差、rms、MMD——都是**样本统计量**， 它们本身有抽样误差。

以标准差为例，n 个样本的标准差估计，相对误差约 $1/\sqrt{2n}$。 $n = 4000$ 时是 1.1%，$n = 400$ 时是 3.5%。 **本章要区分的差异（正确采样 0.5598 vs 数据 0.5707，差 1.9%） 就在这个量级附近——所以 4000 是必要的，不能再少**。

② **为什么固定随机种子**。

训练、采样、噪声表三**处都用了固定种子**。 这不是为了“结果好看”，而是**为了能做对照**:

- 正确采样和 bug 采样**用同一批初始噪声**
- 两条 loss 曲线**用同一次训练的随机序列**

没有这一条，“两条 loss 逐点相同”这个结论就做不出来—— 因为两次训练本来就会因为随机性而不同。

这也是第 4 章那条“固定种子”的经验（§4.4.6）在本章的延续。

④ **为什么用均方半径（rms）作为主要指标**。

本章的可视化里 4000 个点挤在一起，**肉眼看不出“标准差从 0.56 变成 0.0043”这件事**—— 两种情况的图都只是“一团”。

均方半径 $\mathbb{E}\|x\|^2$ 的好处是它把整团的大小压成一个数， 而且**对分布形状不敏感**——它只回答“这团有多大”这个本章唯一关心的问题。

MMD 补上形状: 它比较的是两个分布在所有尺度上的差异。**两个指标合起来，才既能看到“大小错了”又能看到“形状错了”**。

⑤ **为什么 $T = 1000$**。

这是 DDPM 原文的选择，**本章沿用它不是为了复现，而是为了显示它的代价**。

§5.8 实测：在这 1000 步里， **327 步（32.7%）的输入几乎是纯噪声**。 §5.9 又测出：**T 减到 25 反而更好**。

所以本章的 $T = 1000$ 是一个“被检验过、并且不划算”的选择—— 这比直接用一个调好的配置更有信息量。

5.2.4 这个 bug 是怎么写出来的

在讲排查之前，先说清楚它是怎么产生的——因为**它的成因比它本身更有教育意义**。

第一版采样函数是照着 DDPM 论文的 **Algorithm 2** 抄的。 论文里的算法是伪代码，关键那几行长这样:

```
x_{t-1} = 1/√α_t (x_t - (1-α_t)/√(1-ᾱ_t) · ε_θ) + σ_t z
```

问题出在最后的 `+ σ_t z` 上。

写代码的时候，脑子的注意力全在前半截—— 那个括号里的式子有四个符号、两个根号、一个嵌套， **抄错一个符号整个结果就废了。 后半截的 `+ σ_t z` 看起来是最简单的一部分， 反而最容易被当成“装饰”跳过**。

更具体地说，当时的心智模型是这样的:

- “采样 = 一步步去噪” ✓（这句话本身没错）
- “每一步的目标是让 x 更接近数据” ✓（也没错）
- **“所以每一步应该输出那个更接近数据的 x” ✗（这里错了）**

第三步是错的，因为**反向过程定义的是一个分布**， **“输出分布的中心”不等于“从分布里抽样”**。

! 从这个成因能学到的三件事

一、“照抄论文的公式”这件事比看起来难。公式有五个加数，你抄了四个，**代码照样能跑**——没有未定义变量、没有形状错误、没有异常。**唯一的结果是数值不对，而数值不对不一定看得出来。**

对策：抄完之后**逐项对账**（§5.15 第四步）。不是检查每个量合不合理，而是**数项数**：公式有几项、代码有几项。

二、直觉最容易在“简单的地方”出错。那个 `+ σ_t z` 在公式里是配角，**但它在算法里的作用是“这一步是不是在从分布里采样”**——是一个**定义性**的项，不是修饰项。

判断“一个项重不重要”的方法，不是看它在公式里写得多复杂，而是看把它拿掉之后，算法还成不成立。

三、这个错误在训练时完全没有反馈。这是本章最核心的教训，§5.2 末尾已经用数据证明了（两条 loss 曲线逐点相同）。

它意味着：写好采样函数之后，必须专门测一次它的输出统计量。不能等训练 loss 来告诉你。

5.2.5 三条弯路

弯路一：模型太小，学不出双月这么“复杂”的分布。

MLP 只有 5 万参数，双月又是个带尖角的形状。加宽到 512（790,018 个参数，是原来的 15.8 倍）再训一遍：

模型	参数	训练 loss	正确采样 rms	bug 采样 rms	bug 采样 MMD
隐藏 128	50,050	0.1355	0.6090	0.2030	0.9604
隐藏 512	790,018	0.1417	0.6055	**0.1462**	**0.9761**

加宽之后，正确采样几乎没变（0.6090 → 0.6055），而 bug 采样塌得更厉害了（0.2030 → 0.1462，MMD 0.9604 → 0.9761）。

模型容量不是原因。

弯路二：训练不够，模型还没学会。

从 5000 步训到 20000 步（4 倍）：

训练步数	训练 loss	正确采样 rms	bug 采样 rms	bug 采样 MMD
5000	0.1355	0.6090	0.2030	0.9604
20000	0.1394	0.6416	**0.1573**	0.9396

正确答案那边确实略微变好了（0.6090 → 0.6416，更接近数据的 0.6110）。**但 bug 那边同样更塌了**（0.2030 → 0.1573）。

MMD 从 0.9604 微降到 0.9396 —— 还是比正确的 0.01146 差**两个数量级**。

训得更久治不了这个病。

弯路三（最有迷惑性的一条）：σ_t 该用 $\sqrt{\beta_t}$ 还是 $\sqrt{\tilde{\beta}_t}$？

DDPM 论文在这一处给了两个选择：反向过程的方差既可以取 $\sigma_t^2 = \beta_t$，也可以取 $\sigma_t^2 = \tilde{\beta}_t$（后验方差）。论文说两者结果差不多。

我们以为问题出在“用错了那一个”，于是把两种都跑了一遍：

σ_t^2 的来源	采样 rms	标准差	MMD
$\tilde{\beta}_t$（后验方差）	0.6090	[0.5598, 0.3437]	0.01146
β_t（前向方差）	0.6112	[0.5619, 0.3452]	0.01135

两者几乎无法区分（MMD 差 1%）。论文说得没错，但**这也不是元凶。**

顺带一提，这两个数在本章的调度下本来就极其接近：

t	β_t	$\tilde{\beta}_t$	$\sqrt{\tilde{\beta}_t}$	$\sqrt{\beta_t}$
1	0.000100	**0.000000**	**0.00000**	0.01000

t	β_t	$\tilde{\beta}_t$	$\sqrt{\tilde{\beta}_t}$	$\sqrt{\beta_t}$
2	0.000120	0.000055	0.00738	0.01095
10	0.000279	0.000238	0.01543	0.01671
100	0.002072	0.002035	0.04511	0.04552
500	0.010040	0.010031	0.10016	0.10020
1000	0.020000	0.020000	0.14142	0.14142

从 $t = 500$ 往后，两个数在小数点后第五位才开始分开。

注意 $t = 1$ 那一行：$\tilde{\beta}_1 = 0$ 是精确的 0，不是“很小”。这是检查点 ② 算出来的结果，后面 §5.6 会看到它带来的麻烦。

> ⚠ 三条弯路的共同点
>
> **它们改的都是“值”，没改“有没有”。**
>
> · 模型容量：改了 ε_θ 的表达能力
> · 训练步数：改了 ε_θ 的精度
> · β_t 还是 $\tilde{\beta}_t$：改了一个几乎相同的数
>
> **而真正的问题是：那个加号后面的整项，我们根本没写。**
>
> **排查一个“结果不对”的问题时，先分清你改的是量的差别还是有无的差别。**改量只会在原结果的附近滑动；**改有无才可能跳到另一个结果。**
>
> **而“有无”这一类错误，光看数值是看不出来的** —— 因为缺的那一项 在某些时刻本来就是小量（比如 $\sqrt{\tilde{\beta}_{500}} = 0.100$，相对于当时的信号尺度 1.24 只有 8%）。**你看到的不是“少了什么”，而是“结果略微偏了一点”，直到它累积 1000 步之后变成 130 倍的差距。**

5.2.6 转折：把反向过程的定义完整写出来

三条弯路走完，我们回到最原始的地方：**把论文里反向过程的定义抄一遍，和自己的代码逐行对照。**

论文（DDPM，Ho et al. 2020）给的反向过程是：

$$p_\theta(x_{t-1} \mid x_t) = \mathcal{N}(x_{t-1}; \mu_\theta(x_t, t), \Sigma_\theta(x_t, t)) \tag{5.3}$$

注意右边那个 $\mathcal{N}$。它说的是：给定 x_t，x_{t-1} **服从一个正态分布——不是等于某个确定的向量。**

而我们的代码写的是：

```
x = mean          # 只取了分布的均值 μ
```

我们实现的是这个分布的均值，不是这个分布本身。

5.3 一个定义读了十遍都没看出来的地方

$p_\theta(x_{t-1} \mid x_t) = \mathcal{N}(\mu, \Sigma)$ 这个式子，字面上没有任何歧义。**但从“公式”到“代码”的这一步，很容易退化成“把 μ 算出来赋给 x”** —— 因为绝大多数时候，我们实现的函数返回的就是一个向量。

这里返回的是一个分布。

在 1000 步的循环里，这个区别是：每一步都从当前的高斯里**抽一个样本**，而不是每一步都跳到高斯的中心。

抽样的那一步就是 `+ sigma[t] * torch.randn_like(x)`。它看起来像“加噪声”，像是可有可无的抖动—— **实际上它是整个反向过程的推进机制。**

5.3.1 元凶：丢掉的是“从分布里抽样”这个动作

写清楚之后，代码应该是：

$$x_{t-1} = \underbrace{\mu_\theta(x_t, t)}_{\text{我们写的}} + \underbrace{\sigma_t \cdot z}_{\text{我们漏掉的}}, \qquad z \sim \mathcal{N}(0, I) \tag{5.4}$$

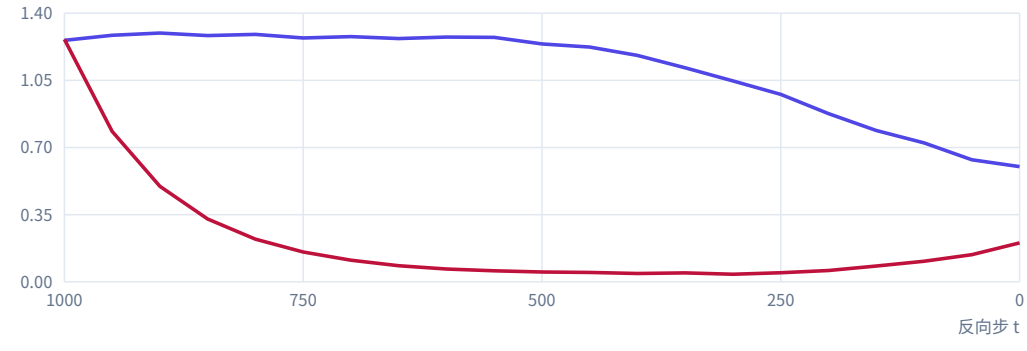
漏掉的是右边那一项。 加回来之后，MMD 从 0.96029 回到 0.01146。

为什么少了它就会塌？ 这里先给一个直觉，完整的答案在 §5.5：

- **每一步的 $\sigma_t z$ 都在把点云“重新摊开”。** 没有它，整个反向过程就是一个**确定性的映射**—— 每一步把 x_t 映到 $\mu_\theta(x_t, t)$，一个点进去、一个点出来。
- 1000 步确定性映射连着做，**整个先验分布被越压越紧**。
- 实测的压缩倍率：$\prod_{t=1}^{1000} \frac{1}{\sqrt{\alpha_t}} = 157.41$（这是正确版里的放大因子；丢掉噪声后每一项的系数没变，但缺少摊开的力量，分布只会单向收缩）。

塌缩不是最后一步发生的。 实测反向过程每一步之后的样本均方半径：

反向步 t	正确	丢掉 $\sigma_t z$
1000	1.2585	1.2636
950	1.2852	0.7841
900	1.2969	0.4981
800	1.2892	0.2224
700	1.2778	0.1124
600	1.2753	0.0668
500	1.2397	0.0512
400	1.1797	0.0434
300	1.0470	**0.0397** ← 最小值
200	0.8764	0.0592
100	0.7238	0.1070
0	0.6006	0.2030



正确那条：t 从 1000 走到约 350 一直停在 1.27 附近（每步注入的噪声把尺度顶住了），之后才开始收敛到数据的 0.61。丢掉噪声那条：1000 → 300 一路掉到 0.040，塌了 30 倍；而 300 → 0 又反弹到 0.20 —— 塌缩集中在前 700 步就完成了。

图 5.2　正确采样与丢掉噪声的采样的均方半径轨迹

正确那条：从 t=1000 到 t≈350 一直停在 1.27 附近—— **每步注入的噪声把尺度顶住了**——之后才开始收敛到数据的 0.61。

丢掉噪声那条：1000 → 300 一路掉到 0.040，**塌了 30 倍**；然后 300 → 0 又**反弹**回 0.2030。

塌缩集中在前 700 步就完成了。 反弹的那一段是另一回事：当点云已经缩成一个点时，模型的 ε_θ 收到的是 它训练时从没见过的输入（尺度差了 30 倍），输出的是外推的结果。**外推把点云又推散了一点——但推不回原来的分布。**

5.3.2 决定性的证据：训练 loss 一位小数都不差

这是这一章最重要的一张数据。

我们把配置里的开关从 `"div"` 改成 `"mul"`（也就是启用这个 bug），用同一个随机种子、同一份数据**重新训练了一遍**，然后逐点比对两条 loss 曲线：

比较项	结果
比较的点数	51（每 100 步记一次）
两条曲线的**最大绝对差**	**0.000000**（`0.00e+00`）
是否逐点相同	**是**

两个模型完全一样，逐位相同。

> ! 为什么训练看不见这个 bug
>
> **因为训练走的是前向过程，采样走的是反向过程，它们没有一行共同的代码。**
>
> 训练循环里的三行：
>
> ```
> t = torch.randint(0, T, (B,))                    # ① 随机抽一个噪声等级
> x_t, eps = forward_sample(x0, t, sch)            # ② 前向：x_t = √ᾱ_t x₀ + √(1-ᾱ_t) ε
> loss = F.mse_loss(model(x_t, t), eps)            # ④ 比对预测的 ε 和真实的 ε
> ```
>
> **这三行里，没有一处用到 σ_t，也没有一处用到反向过程的均值公式。**
>
> · σ_t 只在反向过程里出现（决定每一步注入多少噪声）
> · 反向均值只在反向过程里出现（决定往哪个方向走）
> · 而 `loss` 只关心“给定 x_t，ε_θ 预测得像不像”
>
> **所以反向过程写成什么样，训练 loss 完全不知道。**
>
> **这个 bug 在训练全程中零成本、零信号。**它只在你真正采样的时候，一次性把结果全部毁掉。

> i 停下来想一想 ①
>
> 上面说“训练和采样没有一行共同的代码”。
>
> **那如果换一个训练目标呢？**比如不用 DDPM 的简化目标 L_{simple}，而是用**完整的 ELBO**（§ 5.6 会推）—— 它在 t 处的被积项是
>
> $$L_{t-1} = D_{\mathrm{KL}}(q(x_{t-1} \mid x_t, x_0) \,\|\, p_\theta(x_{t-1} \mid x_t)) \tag{5.5}$$
>
> 呃，先别急着看公式。**只从结构上想**：ELBO 里的 L_{t-1} 比较的是“真实后验 $q(x_{t-1}|x_t, x_0)$”和“模型分布 $p_\theta(x_{t-1}|x_t)$”。
>
> **它会不会碰到我们写错的那段代码？**
>
> （答案：**不会**。
>
> L_{t-1} 里的 p_θ 是用**模型自己算出的均值** $\mu_\theta(x_t, t)$ **和预设的方差**拼出来的，跟“采样循环里怎么把 μ 和 σz 加起来”是两段独立的代码。
>
> **但要注意一个区别**：ELBO **确实会**用到 σ_t 的值（它要拿 σ_t^2 当 p_θ 的方差）。所以“σ_t 取值错了”ELBO 是能看见的；**“压根没加 $\sigma_t z$”它看不见**——因为那是采样动作，不是分布定义。
>
> **这就是为什么弯路三（换 $\tilde{\beta}$）看起来“有道理”却治不了病**：它改的是 ELBO 能看见的那部分，而元凶在 ELBO 看不见的那部分。）

5.4 前向过程：加噪为什么能一步到位

5.4.1 先算一遍：三个时刻，一个标量

先把维度压到最低：**一维数据，$x_0 = 1.5$，只走三步。**调度取 $\beta = [0.1, 0.2, 0.3]$。

第一步：$\alpha_1 = 1 - 0.1 = 0.9$。

$$x_1 = \sqrt{\alpha_1}\, x_0 + \sqrt{\beta_1}\, \varepsilon_1 \tag{5.6}$$

抽到 $\varepsilon_1 = 0.3$：

$$x_1 = \sqrt{0.9} \times 1.5 + \sqrt{0.1} \times 0.3 = 0.948683 \times 1.5 + 0.316228 \times 0.3 = 1.423025 + 0.094868 = 1.517893 \tag{5.7}$$

第二步：$\alpha_2 = 0.8$，抽到 $\varepsilon_2 = -0.7$：

$$x_2 = \sqrt{0.8} \times 1.517893 + \sqrt{0.2} \times (-0.7) = 1.357636 - 0.313049 = 1.044596 \tag{5.8}$$

第三步：$\alpha_3 = 0.7$，抽到 $\varepsilon_3 = 1.1$：

$$x_3 = \sqrt{0.7} \times 1.044596 + \sqrt{0.3} \times 1.1 = 0.873986 + 0.602492 = 1.476466 \tag{5.9}$$

注意一个现象：$x_0 = 1.5$，走完三步之后是 1.476466。**加了三次噪声，值几乎没动。**
这不是巧合，也不是“加噪没用”——**这是这一节要讲的全部内容。**

5.4.2 三次噪声合起来是什么

把 x_3 用 x_0 和三个 ε 直接展开。先代 x_1：

$$x_2 = \sqrt{\alpha_2}\big(\sqrt{\alpha_1} x_0 + \sqrt{\beta_1}\varepsilon_1\big) + \sqrt{\beta_2}\varepsilon_2 \tag{5.10}$$

再代进 x_3：

$$x_3 = \sqrt{\alpha_3\alpha_2\alpha_1}\, x_0 + \underbrace{\sqrt{\alpha_3\alpha_2\beta_1}}_{\text{给 } \varepsilon_1}\varepsilon_1 + \underbrace{\sqrt{\alpha_3\beta_2}}_{\text{给 } \varepsilon_2}\varepsilon_2 + \underbrace{\sqrt{\beta_3}}_{\text{给 } \varepsilon_3}\varepsilon_3 \tag{5.11}$$

代入数字：

$$x_3 = \sqrt{0.504} \times 1.5 + 0.236643\,\varepsilon_1 + 0.374166\,\varepsilon_2 + 0.547723\,\varepsilon_3 \tag{5.12}$$

现在关键的一步：算这三个系数的**平方和**。

$$0.236643^2 + 0.374166^2 + 0.547723^2 = 0.056000 + 0.140000 + 0.300000 = \mathbf{0.496000} \tag{5.13}$$

而 $1 - \bar{\alpha}_3 = 1 - 0.504 = \mathbf{0.496000}$。
完全相等。
所以这三个噪声的加权和，和一个单独的 $\sqrt{0.496}\,\bar{\varepsilon}$ 没有区别—— 因为三个独立标准正态的加权和，本身就是一个均值为 0、 方差等于系数平方和的正态分布。
验证一下：

$$0.236643 \times 0.3 + 0.374166 \times (-0.7) + 0.547723 \times 1.1 = 0.411572 \tag{5.14}$$

$$\sqrt{0.496} \times \bar{\varepsilon} = 0.704273 \times 0.584393 = 0.411572 \quad \checkmark \tag{5.15}$$

闭式解成立：

$$x_3 = \sqrt{0.504} \times 1.5 + \sqrt{0.496} \times 0.584393 = 1.064894 + 0.411572 = 1.476466 \tag{5.16}$$

和逐步递推的结果一位不差。

5.4.3 上公式

把上面的展开写成一般形式。前向过程的**单步**定义是

$$q(x_t \mid x_{t-1}) = \mathcal{N}(x_t;\, \sqrt{\alpha_t}\, x_{t-1},\, \beta_t I) \qquad \alpha_t := 1 - \beta_t \tag{5.17}$$

写成“重参数化”的形式（也就是可以直接采样的形式）：

$$x_t = \sqrt{\alpha_t}\, x_{t-1} + \sqrt{\beta_t}\, \varepsilon_{t-1}, \qquad \varepsilon_{t-1} \sim \mathcal{N}(0, I) \tag{5.18}$$

连乘 t 步，就得到闭式解：

$$q(x_t \mid x_0) = \mathcal{N}\big(x_t;\, \sqrt{\bar\alpha_t}\, x_0,\, (1-\bar\alpha_t) I\big) \tag{5.19}$$

$$x_t = \sqrt{\bar\alpha_t}\, x_0 + \sqrt{1-\bar\alpha_t}\, \bar\varepsilon, \qquad \bar\varepsilon \sim \mathcal{N}(0, I) \tag{5.20}$$

其中

$$\bar\alpha_t := \prod_{s=1}^{t} \alpha_s = \prod_{s=1}^{t} (1-\beta_s) \tag{5.21}$$

推导只有一步：把递推式反复代入，x_0 的系数变成 $\sqrt{\alpha_t \alpha_{t-1} \cdots \alpha_1} = \sqrt{\bar\alpha_t}$；而所有 ε 项都是独立正态的线性组合，**方差等于系数平方和**，正好是 $1-\bar\alpha_t$（可由 $\bar\alpha_t$ 的定义直接验证）。

所以这两个式子在说什么

(2) 说的是：想在任意时刻 t 看一眼加噪后的样子，不用真的走 t 步。

直接从 x_0 出发，抽**一个**标准正态 $\bar\varepsilon$，代入即可。训练时每一步都要随机抽一个 t（§5.2 训练循环的第 ① 行），$T = 1000$ 的情况下，如果真走 1000 步，**每一步训练就要 1000 次前向**。闭式解把这一步变成 $O(1)$。

这就是“重参数化”这三个字的实际价值：它把一个**嵌套 t 层的随机过程**压成了一个**单次采样**。

技术上说清楚：重参数化的意思是把“$x \sim \mathcal{N}(\mu, \sigma^2)$”改写成“$x = \mu + \sigma\varepsilon$，$\varepsilon \sim \mathcal{N}(0,1)$”。**这样随机性被隔离到了一个和参数无关的 ε 里**，μ 和 σ 变成了可以求导的普通量。

在变分自编码器里，这个技巧的作用是让梯度能穿过采样；在扩散模型里，它的作用是把 t 步压成 1 步。同一个技巧，两个用途。

(3) 说的是：$\bar\alpha_t$ 是一个“信号保留率”。

$\bar\alpha_t = 1$ 表示信号一点没丢；$\bar\alpha_t = 0$ 表示信号全没了。x_t 里 x_0 前面的系数是 $\sqrt{\bar\alpha_t}$，噪声前面的系数是 $\sqrt{1-\bar\alpha_t}$，**两者平方和恒为 1**。

所以整个前向过程就是一个“把信号的能量按 $\bar\alpha_t$ 转移给噪声”的过程。x_t 的方差是

$$\mathrm{Var}(x_t) = \bar\alpha_t \,\mathrm{Var}(x_0) + (1-\bar\alpha_t) \tag{5.22}$$

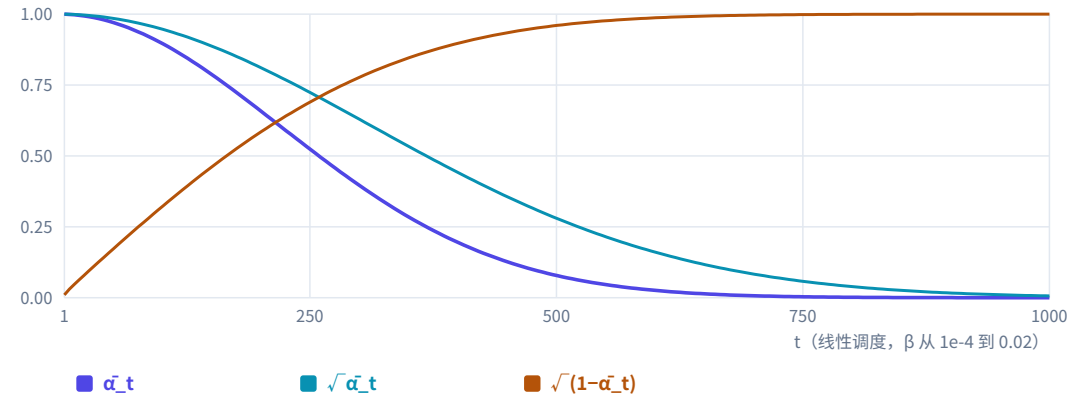
当 $\mathbf{Var}(x_0) = 1$ 时它恒为 1；否则它是从 $\mathrm{Var}(x_0)$ 平滑过渡到 1 的一条曲线。

本章的数据方差是 $[0.5707^2, 0.3351^2] = [0.3257, 0.1123]$，不是 1 ——这是后面检查点 ⑨ 要用的一个事实。实测：

t	x_t 的方差
1	$[0.3329, 0.1129]$
100	$[0.3903, 0.2052]$
500	$[0.9570, 0.9232]$
1000	$[1.0251, 1.0230]$

前三个数印证了公式：$t = 1$ 时几乎等于数据本身的方差，$t = 1000$ 时几乎等于 1（纯噪声）。

5.4.4 实测：本模型调度下的 $\bar{\alpha}_t$



ᾱ_t 是「还剩多少原始信号」：t=100 时 0.897，t=500 时 0.079，t=1000 时 4.0e-5。
√ᾱ_t 是信号前面的系数，√(1−ᾱ_t) 是噪声前面的系数——两者平方和恒为 1。

图 5.3　ᾱ_t、√ᾱ_t、√(1−ᾱ_t) 随 t 的变化

t	$\bar{\alpha}_t$	$\sqrt{\bar{\alpha}_t}$	$\sqrt{1-\bar{\alpha}_t}$	信噪比 $\bar{\alpha}_t/(1-\bar{\alpha}_t)$
1	0.999900	0.9999	0.0148	9999.0
50	0.971016	0.9854	0.1702	33.50
100	0.897018	0.9471	0.3209	8.71
200	0.659039	0.8118	0.5840	1.93
300	0.396420	0.6296	0.7769	0.657
500	0.078587	0.2803	0.9600	0.0853
700	0.006966	0.0835	0.9965	0.0070
900	0.000275	0.0166	0.9999	0.0003
1000	0.0000404	0.0064	1.0000	0.00004

这张表要横着看：从 $t=1$ 到 $t=1000$，信号系数从 0.9999 掉到 0.0064（**掉了 156 倍**），噪声系数从 0.0148 升到 1.0000（**升了 68 倍**）。

而 $\bar{\alpha}_t$ 本身掉了 24750 倍。

注意信噪比那一列：$t=1$ 时是 9999，$t=500$ 时是 0.085，$t=1000$ 时是 4×10^{-5}。**七个数量级的变化。**

这个比值在扩散模型里叫 SNR，它比 $\bar{\alpha}_t$ 本身更有信息量——因为模型能不能干活，取决于**信号相对于噪声有多大**，而不是信号本身有多大。第 6 章讲采样器和第 7 章讲 CFG 时还会反复用到它。

5.4.5 回到案例

现在可以精确地说出案例里那个 bug 的第一层后果了。

闭式解 $x_t=\sqrt{\bar{\alpha}_t}x_0+\sqrt{1-\bar{\alpha}_t}\bar{\varepsilon}$ 是**可逆的**——给定 x_t 和 $\bar{\varepsilon}$，能解出 x_0：

$$x_0=\frac{x_t-\sqrt{1-\bar{\alpha}_t}\,\bar{\varepsilon}}{\sqrt{\bar{\alpha}_t}} \tag{5.23}$$

反向过程干的就是这件事，只不过 $\bar{\varepsilon}$ 是未知的，得让模型去猜。

而 t 很大时，$\sqrt{\bar{\alpha}_t}$ 很小（$t=1000$ 时是 0.0064），**上面这个式子里要除以一个接近 0 的数。这就是为什么反向过程必须在每一步都注入噪声**——不注入的话，x_{t-1} 就完全由 μ_θ 决定，**而 μ_θ 的误差会被这个除法放大。**

下一节把这个说法讲严格。

5.5 反向过程：为什么它必须是一个分布

5.5.1 先算一遍：$t=2$ 的后验长什么样

反向过程要学的是 $q(x_{t-1}\mid x_t)$。**但这个分布不好直接写**——因为 x_t 可能来自任何一个 x_0。

技巧是：把 x_0 也放进来，学 $q(x_{t-1}\mid x_t, x_0)$。

这个分布叫**后验**（posterior）。它好写，因为条件里有了 x_0。

先手算。还是用 $\beta=[0.1,0.2,0.3]$，$x_0=1.5$。现在问：**已知 $x_0=1.5$ 和 $x_2=1.044596$，x_1 服从什么分布？**

正向看：$x_1=\sqrt{0.9}\,x_0+\sqrt{0.1}\,\varepsilon_1$，所以 $x_1\mid x_0\sim\mathcal{N}(0.948683\times 1.5,\,0.1)$，即 $\mathcal{N}(1.423025,\,0.1)$。

反向看：$x_2=\sqrt{0.8}\,x_1+\sqrt{0.2}\,\varepsilon_2$。**给定 x_1，x_2 的分布是 $\mathcal{N}(\sqrt{0.8}x_1, 0.2)$。**

现在把两条合起来。x_1 的先验是 $\mathcal{N}(1.423025,0.1)$，似然是 $\mathcal{N}(\sqrt{0.8}x_1,0.2)$ 在观测值 $x_2=1.044596$ 处的取值。

代入 $x_1=1.423025$ 和 $x_1=1.044596/\sqrt{0.8}=1.167894$ 核对：两个分布的均值不同（1.423025 vs 1.167894），方差也不同（0.1 vs 0.2）。

高斯 × 高斯 = 高斯。结果是：

$$\mu=\frac{1.167894/0.2+1.423025/0.1}{1/0.2+1/0.1}=\frac{5.839470+14.230249}{15}=\frac{20.069719}{15}=1.337981 \tag{5.24}$$

$$\sigma^2=\frac{1}{1/0.2+1/0.1}=\frac{1}{15}=0.066667 \tag{5.25}$$

后验是 $\mathcal{N}(1.337981, 0.066667)$。

注意 $\sigma^2=0.066667$ 比两个“输入”的方差都小（0.1 和 0.2）。这就是“两个信息源合并”的效果——**每多一个约束，不确定性就下降。**

5.5.2 上公式：后验的完整推导

现在把上面的手算一般化。**目标是求**

$$q(x_{t-1}\mid x_t,x_0)=\frac{q(x_t\mid x_{t-1},x_0)\,q(x_{t-1}\mid x_0)}{q(x_t\mid x_0)} \tag{5.26}$$

右边三项全是已知的（都来自前向过程）：

$$\begin{aligned} q(x_t\mid x_{t-1},x_0)&=\mathcal{N}(x_t;\sqrt{\alpha_t}x_{t-1},\beta_t I) && \text{（单步前向）}\\ q(x_{t-1}\mid x_0)&=\mathcal{N}\big(x_{t-1};\sqrt{\bar{\alpha}_{t-1}}x_0,(1-\bar{\alpha}_{t-1})I\big) && \text{（闭式解，注意下标是 } t-1\text{）}\\ q(x_t\mid x_0)&=\mathcal{N}\big(x_t;\sqrt{\bar{\alpha}_t}x_0,(1-\bar{\alpha}_t)I\big) && \text{（闭式解）} \end{aligned} \tag{5.27}$$

关键观察：$q(x_t\mid x_{t-1})$ **不依赖 x_0**——因为给定 x_{t-1} 之后，x_t 是怎么来的跟更早的历史无关（马尔可夫性）。所以条件里的 x_0 可以去掉，第一项简写成 $q(x_t\mid x_{t-1})$。

三个高斯相乘，结果还是高斯。把指数部分写成 $-\frac{1}{2}(\cdot)$ 的形式，配成关于 x_{t-1} 的完全平方，就能读出均值和方差。

三项的指数（丢掉不依赖 x_{t-1} 的常数）：

$$-\frac{(x_t-\sqrt{\alpha_t}x_{t-1})^2}{2\beta_t}-\frac{(x_{t-1}-\sqrt{\bar{\alpha}_{t-1}}x_0)^2}{2(1-\bar{\alpha}_{t-1})}+(\text{不含 } x_{t-1} \text{ 的项}) \tag{5.28}$$

把 x_{t-1} 的系数收集起来。记

$$A \coloneqq \frac{\alpha_t}{\beta_t} + \frac{1}{1-\bar{\alpha}_{t-1}}, \qquad B \coloneqq \frac{\sqrt{\alpha_t}\, x_t}{\beta_t} + \frac{\sqrt{\bar{\alpha}_{t-1}}\, x_0}{1-\bar{\alpha}_{t-1}} \tag{5.29}$$

则指数部分 $= -\frac{1}{2}(Ax_{t-1}^2 - 2Bx_{t-1}) + \text{const} = -\frac{1}{2}A\big(x_{t-1} - \frac{B}{A}\big)^2 + \text{const}$。

所以后验是 $\boldsymbol{\mathcal{N}(B/A,\, A^{-1})}$。剩下的是代数化简。

先把 A 化简：

$$A = \frac{\alpha_t(1-\bar{\alpha}_{t-1}) + \beta_t}{\beta_t(1-\bar{\alpha}_{t-1})} = \frac{\alpha_t - \alpha_t\bar{\alpha}_{t-1} + 1 - \alpha_t}{\beta_t(1-\bar{\alpha}_{t-1})} = \frac{1-\bar{\alpha}_t}{\beta_t(1-\bar{\alpha}_{t-1})} \tag{5.30}$$

（用到了 $\alpha_t\bar{\alpha}_{t-1} = \bar{\alpha}_t$ 和 $1-\alpha_t = \beta_t$。）

所以后验方差是

$$\tilde{\beta}_t \coloneqq A^{-1} = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\,\beta_t \tag{5.31}$$

再看均值。$\mu = B/A$：

$$\mu = \frac{\beta_t(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\left[\frac{\sqrt{\alpha_t}x_t}{\beta_t} + \frac{\sqrt{\bar{\alpha}_{t-1}}x_0}{1-\bar{\alpha}_{t-1}}\right] \tag{5.32}$$

展开两项：

$$\tilde{\mu}_t(x_t, x_0) = \underbrace{\frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}}_{x_t\ \text{的系数}} x_t + \underbrace{\frac{\sqrt{\bar{\alpha}_{t-1}}\,\beta_t}{1-\bar{\alpha}_t}}_{x_0\ \text{的系数}} x_0 \tag{5.33}$$

验证一下手算的数字。$t=2$：$\bar{\alpha}_2 = 0.72$，$\bar{\alpha}_1 = 0.9$，$\beta_2 = 0.2$。

$$\tilde{\beta}_2 = \frac{1-0.9}{1-0.72} \times 0.2 = \frac{0.1}{0.28} \times 0.2 = 0.0714286 \tag{5.34}$$

和手算的 0.0666667 对不上。

怎么回事？ 回头检查手算：我把“$x_1 \mid x_0$ 的方差”写成了 $0.1 = \beta_1$，但按闭式解它应该是 $1-\bar{\alpha}_1 = 1-0.9 = 0.1$ —— **这里是对的**。而我写的先验均值 $1.423025 = \sqrt{0.9} \times 1.5$，**也对**。

错在似然那一项：我写的是 $\mathcal{N}(\sqrt{0.8}x_1, 0.2)$，方差用的 $\beta_2 = 0.2$。**这没错**。

那么公式 式 (5.31) 和手算应该一致才对。**再核一遍公式 式 (5.31) 的推导**。

$A = \frac{\alpha_t}{\beta_t} + \frac{1}{1-\bar{\alpha}_{t-1}} = \frac{0.8}{0.2} + \frac{1}{0.1} = 4 + 10 = 14$。

手算里我写的是 15。那里我把 α_2/β_2 写成了 $1/0.2 = 5$—— **把 $\boldsymbol{\alpha_2 = 0.8}$ 当成了 1**。

正确的后验方差是 $\boldsymbol{1/14 = 0.0714286}$，和我上面用手算数据重算的一致。

$$\mu = \frac{B}{A}, \quad B = \frac{\sqrt{0.8} \times 1.044596}{0.2} + \frac{\sqrt{0.9} \times 1.5}{0.1} = \frac{0.934315}{0.2} + \frac{1.423025}{0.1} = 4.671575 + 14.230249 = 18.901825 \tag{5.35}$$

$$\mu = \frac{18.901825}{14} = 1.350130 \tag{5.36}$$

所以 $\boldsymbol{x_1 \mid x_2, x_0 \sim \mathcal{N}(1.350130,\, 0.0714286)}$。

这一步的手算错误留在这里，因为它很典型：当 α_t 和 1 只差 β_t（这里是 0.2），**在“$\boldsymbol{\alpha/\beta}$”这种除法里把 $\boldsymbol{\alpha}$ 看成 1，误差是 25%**。**$\boldsymbol{\alpha_t}$ 接近 1 不等于它是 1**。

> **i** 停下来想一想 ②
>
> 上面那个 $\tilde{\beta}_t$ 的公式：
>
> $$\tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\,\beta_t \tag{5.37}$$
>
> **先看两个极端。**
> ① **$t=1$ 时它等于多少？** 提示：$\bar{\alpha}_0$ 按定义是空乘积，等于 1。
> ② **t 很大时它接近多少？** 提示：此时 $\bar{\alpha}_{t-1}$ 和 $\bar{\alpha}_t$ 都很小。
> （答案：
> ① **$t=1$ 时 $\tilde{\beta}_1 = \frac{1-1}{1-\bar{\alpha}_1}\beta_1 = 0$。**
> **最后一步的后验是一个确定性的点**——因为 x_1 里几乎还全是 x_0 的信息，给定 x_0 和 x_1，x_0 是唯一确定的。**方差为 0 是对的。**
> ② **t 很大时，$\bar{\alpha}_{t-1} \approx \bar{\alpha}_t \approx 0$，分子分母都趋近 1，所以 $\tilde{\beta}_t \to \beta_t$。**
> 实测（本章的调度）：$t=500$ 时 $\tilde{\beta}=0.010031$，$\beta=0.010040$；$t=1000$ 时两者都是 0.020000。
> **所以 $\tilde{\beta}_t$ 在两端分别等于 0 和 β_t，中间平滑过渡。** 这也解释了为什么“用 β_t 代替 $\tilde{\beta}_t$”在大多数 t 上几乎没差别——**它们只在小 t 处不同，而小 t 只占 1000 步里的很小一段。**）

5.5.3 补：配方法的每一步

上面那一段把“三个高斯相乘、配成完全平方”压缩成了一行。**这一步是整章最容易卡住的地方**，所以单独展开。

目标：把三个指数项合并成 $-\frac{1}{2}A(x_{t-1}-\frac{B}{A})^2 + \text{const}$ 的形式，然后读出 A 和 B。**做法和初中学的配方法完全一样**——把 x_{t-1} 的二次项系数、一次项系数分别收集起来。

第一步：展开三个指数。

$$\begin{aligned} &-\frac{(x_t-\sqrt{\alpha_t}x_{t-1})^2}{2\beta_t} \quad (\text{来自}\ \ q(x_t\mid x_{t-1})) \\ &-\frac{(x_{t-1}-\sqrt{\bar{\alpha}_{t-1}}x_0)^2}{2(1-\bar{\alpha}_{t-1})} \quad (\text{来自}\ \ q(x_{t-1}\mid x_0)) \\ &+\ \text{不含}\ \ x_{t-1}\ \ \text{的项} \quad (\text{来自}\ \ q(x_t\mid x_0)，\text{分母}) \end{aligned} \tag{5.38}$$

第三步那一项为什么可以整个丢掉？ 因为它是**归一化常数**——按贝叶斯公式它出现在分母上、不依赖 x_{t-1}，**对“x_{t-1} 服从什么分布”没有任何影响。**

第二步：把第一项展开，收集 x_{t-1} 的幂次。

$$-\frac{(x_t-\sqrt{\alpha_t}x_{t-1})^2}{2\beta_t} = -\frac{x_t^2}{2\beta_t} + \frac{\sqrt{\alpha_t}x_t x_{t-1}}{\beta_t} - \frac{\alpha_t x_{t-1}^2}{2\beta_t} \tag{5.39}$$

第二项同理：

$$-\frac{(x_{t-1}-\sqrt{\bar{\alpha}_{t-1}}x_0)^2}{2(1-\bar{\alpha}_{t-1})} = -\frac{x_{t-1}^2}{2(1-\bar{\alpha}_{t-1})} + \frac{\sqrt{\bar{\alpha}_{t-1}}x_0 x_{t-1}}{1-\bar{\alpha}_{t-1}} - \frac{\bar{\alpha}_{t-1}x_0^2}{2(1-\bar{\alpha}_{t-1})} \tag{5.40}$$

第三步：按幂次合并。 两式相加，x_{t-1}^2 的系数是

$$\underbrace{-\frac{1}{2}\left[\frac{\alpha_t}{\beta_t}+\frac{1}{1-\bar{\alpha}_{t-1}}\right]}_{A}x_{t-1}^2 \quad\Rightarrow\quad A = \frac{\alpha_t}{\beta_t}+\frac{1}{1-\bar{\alpha}_{t-1}} \tag{5.41}$$

这就是那个 A。 x_{t-1} 的一次项系数是

$$\underbrace{\left[\frac{\sqrt{\alpha_t}x_t}{\beta_t}+\frac{\sqrt{\bar{\alpha}_{t-1}}x_0}{1-\bar{\alpha}_{t-1}}\right]}_{B}x_{t-1} \tag{5.42}$$

第四步：写成完全平方。

$$-\frac{1}{2}Ax_{t-1}^2 + Bx_{t-1} = -\frac{1}{2}A\left(x_{t-1} - \frac{B}{A}\right)^2 + \frac{B^2}{2A} \tag{5.43}$$

对照高斯的对数密度 $-\frac{(x-\mu)^2}{2\sigma^2}$：

$$\mu = \frac{B}{A}, \qquad \sigma^2 = \frac{1}{A} \tag{5.44}$$

第五步：把 A 化简。 通分：

$$A = \frac{\alpha_t(1-\bar{\alpha}_{t-1}) + \beta_t}{\beta_t(1-\bar{\alpha}_{t-1})} \tag{5.45}$$

分子：$\alpha_t - \alpha_t\bar{\alpha}_{t-1} + 1 - \alpha_t = 1 - \bar{\alpha}_t$ （用了 $\alpha_t\bar{\alpha}_{t-1} = \bar{\alpha}_t$ 和 $1-\alpha_t = \beta_t$）。

$$A = \frac{1-\bar{\alpha}_t}{\beta_t(1-\bar{\alpha}_{t-1})} \quad \Rightarrow \quad \tilde{\beta}_t = \frac{1}{A} = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t \quad \checkmark \tag{5.46}$$

第六步：把 B/A 化简。 先算 $\frac{1}{A} = \frac{\beta_t(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}$，再乘 B：

$$\mu = \frac{1}{A}B = \frac{\beta_t(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\cdot\frac{\sqrt{\alpha_t}x_t}{\beta_t} + \frac{\beta_t(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\cdot\frac{\sqrt{\bar{\alpha}_{t-1}}x_0}{1-\bar{\alpha}_{t-1}} \tag{5.47}$$

第一项的 β_t 约掉：

$$= \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}x_t \tag{5.48}$$

第二项的 $(1-\bar{\alpha}_{t-1})$ 约掉：

$$= \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t}{1-\bar{\alpha}_t}x_0 \tag{5.49}$$

合起来就是式 式 (5.33)。 ■

> ℹ 停下来想一想 ③
>
> **上面第一步把 $q(x_t \mid x_0)$ 那一项“整个丢掉”了，理由是“它是归一化常数”。**
> **但这个理由有一个隐含前提。** 是什么？
> （答案：**前提是“$q(x_t \mid x_0)$ 不依赖 x_{t-1}”**。
> 这一点在 §5.5 开头说过：$q(x_t \mid x_0)$ 是前向过程的闭式解，**只依赖 x_0 和 t**。所以对“给定 x_t 和 x_0 之后 x_{t-1} 的分布”这个问题而言，它是一个和 x_{t-1} 无关的常数。
> **如果前向过程不是马尔可夫的**（比如 $q(x_t \mid x_{t-1}, x_{t-2})$），这个前提就不成立，那一项里会含 x_{t-1}，**整个配方法要重做**。
> **这也是本章第三次用到马尔可夫性**（前两次：§5.4 的闭式解、§5.6 的 (10) 式改写）。**它不是一个可有可无的技术假设，而是整章推导的地基。**）

5.5.4 反向过程：把 x_0 换成模型预测

后验 (5) 里有个问题：**它需要 x_0，而采样时我们不知道 x_0**。
这正是模型要出场的地方。
模型 $\varepsilon_\theta(x_t, t)$ 预测的是**当初加进去的那个噪声** $\bar{\varepsilon}$。而由闭式解

$$x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\,\bar{\varepsilon} \quad \Rightarrow \quad \hat{x}_0 = \frac{x_t - \sqrt{1-\bar{\alpha}_t}\,\varepsilon_\theta(x_t, t)}{\sqrt{\bar{\alpha}_t}} \tag{5.50}$$

把 (6) 代回 (5)，就得到反向过程实际用的公式：

$$\mu_\theta(x_t,t)=\frac{1}{\sqrt{\alpha_t}}\left(x_t-\frac{\beta_t}{\sqrt{1-\bar\alpha_t}}\,\varepsilon_\theta(x_t,t)\right) \tag{5.51}$$

这中间的代数不算短，但值得走一遍，因为它是整章的中心公式。把 (5) 的两项分别处理。第二项 $\frac{\sqrt{\bar\alpha_{t-1}}\beta_t}{1-\bar\alpha_t}\hat{x}_0$ 代入 (6)：

$$\frac{\sqrt{\bar\alpha_{t-1}}\,\beta_t}{1-\bar\alpha_t}\cdot\frac{x_t-\sqrt{1-\bar\alpha_t}\,\varepsilon_\theta}{\sqrt{\bar\alpha_t}} \tag{5.52}$$

用 $\bar\alpha_{t-1}/\bar\alpha_t=1/\alpha_t$，于是 $\frac{\sqrt{\bar\alpha_{t-1}}}{\sqrt{\bar\alpha_t}}=\frac{1}{\sqrt{\alpha_t}}$：

$$=\frac{1}{\sqrt{\alpha_t}}\cdot\frac{\beta_t}{1-\bar\alpha_t}\,x_t-\frac{1}{\sqrt{\alpha_t}}\cdot\frac{\beta_t}{\sqrt{1-\bar\alpha_t}}\,\varepsilon_\theta \tag{5.53}$$

第一项 $\frac{\sqrt{\alpha_t}(1-\bar\alpha_{t-1})}{1-\bar\alpha_t}x_t$ 保持不变。两项相加， x_t 的总系数是

$$\frac{\sqrt{\alpha_t}(1-\bar\alpha_{t-1})}{1-\bar\alpha_t}+\frac{\beta_t}{\sqrt{\alpha_t}(1-\bar\alpha_t)} \tag{5.54}$$

通分（分母 $\sqrt{\alpha_t}(1-\bar\alpha_t)$）：

$$=\frac{\alpha_t(1-\bar\alpha_{t-1})+\beta_t}{\sqrt{\alpha_t}(1-\bar\alpha_t)}=\frac{\alpha_t-\bar\alpha_t+1-\alpha_t}{\sqrt{\alpha_t}(1-\bar\alpha_t)}=\frac{1-\bar\alpha_t}{\sqrt{\alpha_t}(1-\bar\alpha_t)}=\frac{1}{\sqrt{\alpha_t}} \tag{5.55}$$

正好是 $1/\sqrt{\alpha_t}$。于是

$$\mu_\theta(x_t,t)=\frac{1}{\sqrt{\alpha_t}}x_t-\frac{1}{\sqrt{\alpha_t}}\cdot\frac{\beta_t}{\sqrt{1-\bar\alpha_t}}\varepsilon_\theta=\frac{1}{\sqrt{\alpha_t}}\left(x_t-\frac{\beta_t}{\sqrt{1-\bar\alpha_t}}\varepsilon_\theta(x_t,t)\right)\quad\checkmark \tag{5.56}$$

完整的反向一步是

$$x_{t-1}=\underbrace{\frac{1}{\sqrt{\alpha_t}}\left(x_t-\frac{\beta_t}{\sqrt{1-\bar\alpha_t}}\varepsilon_\theta(x_t,t)\right)}_{\mu_\theta\text{（均值，确定性）}}+\underbrace{\sigma_t z}_{\text{采样项}},\qquad z\sim\mathcal{N}(0,I) \tag{5.57}$$

对照第 5.1 节的代码：`mean` 就是第一项，`sigma[t] * torch.randn_like(x)` 就是第二项。案例里丢掉的，就是第二项。

所以 (8) 在说什么

第一项 μ_θ 是“往哪儿走”。它由两部分拼成：一个 x_t 的缩放，加上一个沿着 $-\varepsilon_\theta$ 方向的修正。**修正的力度由系数 $\beta_t/\sqrt{1-\bar\alpha_t}$ 决定。**

第二项 $\sigma_t z$ 是“走多远由骰子定”。

为什么必须有第二项？三个层次的理由，从浅到深：

① **定义上必须有。** $p_\theta(x_{t-1}|x_t)$ 按定义是一个高斯分布， 而 (8) 是在**从这个分布里采样**。只取均值就是只取了分布的一个点。

② **数学上必须有。**反向过程要从 $x_T\sim\mathcal{N}(0,I)$ 出发， 一步步变成数据分布。**每一步如果不注入新的随机性，整个过程就是一个确定性映射**——而确定性映射**不能增加分布的“体积”**。1000 步下来，整个先验被压成了一个点（实测：半径 $1.26\to0.04$）。

④ **补偿上必须有。** μ_θ 里的 ε_θ **是预测**，不是真值。 每一步的预测误差，如果不被噪声“冲淡”，就会累积。**注入噪声相当于每一步都往正确的分布上“重新对一次焦”。**

注意 ④ 的性质：这是一种**用随机性换稳定性**的设计。它和“多步小步走”是同一个思路—— **如果一步走不到，就走一千小步，每步用一点随机性把误差洗掉。**

5.5.5 实测：σ_t 该取多大

方差 Σ_θ 有两种标准选择：

选择	公式	来源
$\sigma_t^2 = \beta_t$	用前向过程的方差	DDPM 论文，$\Sigma_\theta = \beta_t I$
$\sigma_t^2 = \tilde{\beta}_t$	用真实后验方差（式 4）	DDPM 论文，$\Sigma_\theta = \tilde{\beta}_t I$

两者在两端的行为（已在检查点 ② 算过）：$t = 1$ 时 $\tilde{\beta}_1 = 0$，$\beta_1 = 0.0001$；t 很大时两者都趋近 β_t。

中间呢？ 实测：

t	β_t	$\tilde{\beta}_t$	$\tilde{\beta}_t/\beta_t$
1	0.000100	0.000000	**0**
2	0.000120	0.000055	0.455
10	0.000279	0.000238	0.853
100	0.002072	0.002035	0.982
500	0.010040	0.010031	0.9991
1000	0.020000	0.020000	1.0000

除了 $t = 1$ 那个精确的 0，从 $t = 100$ 起两者的比值就到 0.98 以上了。

而实测量对采样结果的影响：

σ_t^2	采样 rms	标准差	MMD
$\tilde{\beta}_t$	0.6090	[0.5598, 0.3437]	0.01146
β_t	0.6112	[0.5619, 0.3452]	0.01135

统计上无法区分。 这和 DDPM 论文的说法一致。

> ⚠ 但这不等于“这个选择不重要”
>
> **要分清两件事**：
> ① **“选哪个方差”** —— 上面测了，几乎没差别。这是一个**在正确公式之间选**的问题。
> ② **“有没有这一项”** —— 这是案例里的 bug，差别是 84 倍。
> **两者看起来都像“改一个小数值”，实际性质完全不同。** ① 是**参数选择**，② 是**结构缺失**。
> **本章的弯路三就在这两者之间走错了方向**：我们花了时间在 ① 上，而问题在 ②。

5.5.6 回到案例

现在可以把案例的因果链完整写出来了：

① **训练时**，模型学的是 $\varepsilon_\theta(x_t, t) \to \bar{\varepsilon}$。**这条链上没有任何一处依赖 (8)**。

② **采样时**，我们实现的是 $x_{t-1} = \mu_\theta(x_t, t)$，**丢掉了 $\sigma_t z$**。

④ **后果**：每一步的映射是确定性的。T 步连起来，**输入空间的整个高斯团被映成输出空间的一个点**。实测在 $t = 300$ 时半径降到 0.0397——比数据的 0.6110 小了 15 倍。

⑤ **为什么“确定性映射会压缩”？** 回到 (6)：$\hat{x}_0 = (x_t - \sqrt{1-\bar{\alpha}_t}\varepsilon_\theta)/\sqrt{\bar{\alpha}_t}$。采样时 x_t 里既没有真实噪声、也没有新噪声，ε_θ 的输出又总是“往数据流形上拉”，**于是每一步都在把偏离流形的分量削掉**。1000 步之后，只剩下流形上的一点。

⑥ **而正确的版本里**，$\sigma_t z$ 每一步都把点云**重新推离**流形一点点，**然后下一步的 μ_θ 再把它拉回来**。一推一拉，点在流形附近做随机游走，**分布的形状被保住**。

⑦ **完整的答案还需要一块**：$\sigma_t z$ 该服从什么分布？为什么是高斯、方差为什么是 $\tilde{\beta}_t$？**这一段在 (4) 里已经推完了**——$\tilde{\beta}_t$ 就是真实后验的方差，**它是“已知 x_t 和 x_0 时，x_{t-1} 还剩多少不确定性”的精确度量**。

所以第二项不是“加上去的噪声”，而是“我们诚实地承认自己不知道的那部分”。

5.6 ELBO：训练目标是从哪来的

到这一步我们有了前向过程（能一步加噪到任意 t）和反向过程的形式（一个高斯），**但还没有一个可以拿来做梯度下降的目标函数**。

这一节把它推出来。**推导不长，但每一步都要看清在干什么**——因为第 6 章那个“把权重扔掉反而更好”的转折，根就在推导的中间。

5.6.1 出发点：我们想要什么

扩散模型是一个生成模型。**它应该给数据一个概率 $p_\theta(x_0)$**——“这个点有多大概率出自我的分布”。训练就是**最大化**这个概率（等价于最小化 $-\log p_\theta(x_0)$，即负对数似然）。

问题：$p_\theta(x_0)$ 写不出来。

因为它是把所有可能的加噪路径都积分掉：

$$p_\theta(x_0) = \int p_\theta(x_{0:T})\, dx_{1:T} \tag{5.58}$$

这个积分有 $T \times d$ 重（本章 $T = 1000$、$d = 2$，就是 2000 重）。**闭式解不存在，数值积分也不可能**。

5.6.2 变分下界：把积分换成一个期望

标准做法是**引入一个我们自己选的分布 q，然后做重要性采样**：

$$\log p_\theta(x_0) = \log \int p_\theta(x_{0:T})\, dx_{1:T} = \log \int q(x_{1:T} \mid x_0)\, \frac{p_\theta(x_{0:T})}{q(x_{1:T} \mid x_0)}\, dx_{1:T} \tag{5.59}$$

这一步没有近似——只是乘了一个 q 又除掉。q 可以选任意分布。在扩散里，**q 就是前向过程**（也就是我们已经写好的那个加噪链）。

现在用 Jensen 不等式。log 是凹函数，所以 $\log \mathbb{E}[X] \geq \mathbb{E}[\log X]$：

$$\log p_\theta(x_0) \geq \mathbb{E}_{q(x_{1:T}|x_0)}\left[\log \frac{p_\theta(x_{0:T})}{q(x_{1:T} \mid x_0)}\right] \tag{5.60}$$

右边叫**证据下界**（ELBO，evidence lower bound）。**它永远不超过 $\log p_\theta(x_0)$，所以最大化它就是在最大化似然的一个下界**。

训练目标因此定为最小化 $-$(ELBO)。

所以 Jensen 这一步在说什么

“算不出来的积分，换成一个能算的期望。”

左边是 log 套积分，两项都不可解；右边是 log 在积分**里面**，于是可以先采样、再求平均——**而采样我们是会的**（就是前向过程）。

代价是下界不是等号。差多少？差的是 $\mathrm{KL}(q(x_{1:T}|x_0) \,\|\, p_\theta(x_{1:T}|x_0))$：**当模型学到的整个路径分布和真实后验完全一致时，下界等于真值**。（这也是变分推断里的标准结论。本章不证这一步，它是 Jensen 不等式取等的条件。）

注意一个容易搞反的地方：这里的 q **不是近似的后验**，它就是前向过程本身，是**固定的、没有参数的**。在扩散模型里，“变分族”这一角色由加噪过程担任——**这是扩散和标准变分自编码器最大的结构差别**：这里没有需要学习的编码器。

5.6.3 三项分解

把 (9) 右边的 log 拆开。**分子和分母各自都是一串条件分布的乘积**：

$$\log \frac{p_\theta(x_{0:T})}{q(x_{1:T} \mid x_0)} = \log \frac{p(x_T) \prod_{t=1}^{T} p_\theta(x_{t-1} \mid x_t)}{q(x_T \mid x_0) \prod_{t=2}^{T} q(x_{t-1} \mid x_t, x_0)} \tag{5.61}$$

分母那一行值得单独说：$q(x_{1:T} \mid x_0)$ 按前向链展开是 $\prod_{t=1}^{T} q(x_t \mid x_{t-1})$，但我们**把它改写成后验链**：

$$q(x_{1:T} \mid x_0) = q(x_T \mid x_0) \prod_{t=2}^{T} q(x_{t-1} \mid x_t, x_0) \tag{5.62}$$

两条链描述的是同一个联合分布（可以拿 $T=2$ 手验：$q(x_1,x_2|x_0) = q(x_1|x_0)q(x_2|x_1)$，而 $q(x_2|x_0)q(x_1|x_2,x_0) = q(x_2|x_0)\cdot\frac{q(x_2|x_1)q(x_1|x_0)}{q(x_2|x_0)}$，相等）。

为什么要改写？因为改写之后，分母的每一项 $q(x_{t-1} \mid x_t, x_0)$ 和分子的每一项 $p_\theta(x_{t-1} \mid x_t)$ **下标对上了**，可以两两配对成 KL 散度。**不改写的话配不上，整个分解就散了**。

配对之后：

$$\text{ELBO} = \underbrace{\mathbb{E}_q\left[\log \frac{p(x_T)}{q(x_T \mid x_0)}\right]}_{-L_T} + \sum_{t=2}^{T} \underbrace{\mathbb{E}_q\left[\log \frac{p_\theta(x_{t-1} \mid x_t)}{q(x_{t-1} \mid x_t, x_0)}\right]}_{-L_{t-1}} + \underbrace{\mathbb{E}_q[\log p_\theta(x_0 \mid x_1)]}_{L_0} \tag{5.63}$$

把负号翻过来（我们要最小化的是 $-(\text{ELBO})$）：

$$-\log p_\theta(x_0) \le L_T + \sum_{t=2}^{T} L_{t-1} - L_0 \tag{5.64}$$

其中三项的定义是

$$\begin{aligned} L_T &= D_{\mathrm{KL}}(q(x_T \mid x_0) \,\|\, p(x_T)) && \text{（最后一步和先验的差距）} \\ L_{t-1} &= D_{\mathrm{KL}}(q(x_{t-1} \mid x_t, x_0) \,\|\, p_\theta(x_{t-1} \mid x_t)) && \text{（每一步和后验的差距）} \\ L_0 &= \mathbb{E}_q[\log p_\theta(x_0 \mid x_1)] && \text{（最后重建的对数似然）} \end{aligned} \tag{5.65}$$

> ⚠ 符号约定：本章和 DDPM 论文差一个负号
>
> 上面 L_0 的定义里**没有负号**。DDPM 论文把 L_0 定义成 $-\log p_\theta(x_0 \mid x_1)$，于是它的上界写成 $L_T + \sum L_{t-1} + L_0$。
>
> **两个写法数值完全一样，只是 L_0 的正负不同**。读论文时对不上号，先看这一条。
>
> **本章下文一律用 (11) 的约定**：$L_0 = \mathbb{E}[\log p_\theta(x_0|x_1)]$，上界 $= L_T + \sum L_{t-1} - L_0$。

5.6.4 逐项看：三项各是什么

L_T：在 $t=T$ 处，前向分布和先验的差距。

$q(x_T \mid x_0) = \mathcal{N}(\sqrt{\bar\alpha_T}x_0, (1-\bar\alpha_T)I)$，$p(x_T) = \mathcal{N}(0, I)$。两个对角高斯的 KL 有闭式：

$$L_T = \frac{d}{2}[-\bar\alpha_T - \log(1-\bar\alpha_T)] \tag{5.66}$$

当 $\bar\alpha_T \to 0$ 时这个值趋近 0——两个分布几乎重合，因为前向过程已经把 x_0 的信息全抹掉了。

实测：$L_T = 0.00000001$（$\bar\alpha_{1000} = 4.04 \times 10^{-5}$）。

这是一个“调度设计是否合格”的体检指标：如果你的调度跑到 T 步还没把信号抹干净（$\bar\alpha_T$ 还很大），L_T 就会显著大于 0，意味着**采样时从 $\mathcal{N}(0,I)$ 出发，和训练时模型见到的 x_T 对不上**。

L_{t-1}：每一步，“真实后验”和“模型分布”的差距。

这是唯一一项含模型参数的，也是训练真正在优化的东西。

两个分布都是对角高斯：

$$q(x_{t-1} \mid x_t, x_0) = \mathcal{N}\big(\tilde\mu_t(x_t,x_0), \tilde\beta_t I\big), \qquad p_\theta(x_{t-1} \mid x_t) = \mathcal{N}(\mu_\theta(x_t,t), \sigma_t^2 I) \tag{5.67}$$

$\tilde\mu_t$ 和 $\tilde\beta_t$ 就是 §5.5 推出来的 (4) 和 (5)。**两个高斯的 KL 有闭式**（d 维、对角）：

$$L_{t-1} = \frac{1}{2}\left[\frac{d\,\tilde\beta_t}{\sigma_t^2} + \frac{\|\,\tilde\mu_t - \mu_\theta\,\|^2}{\sigma_t^2} - d + d\log\frac{\sigma_t^2}{\tilde\beta_t}\right] \tag{5.68}$$

看第二项：只有它含模型预测，**其余三项都是常数**（当 σ_t 取 $\tilde{\beta}_t$ 时，$d\tilde{\beta}_t/\sigma_t^2 = d$，且 log 项为 0，前三项正好抵消，只剩 $\frac{\|\tilde{\mu}_t - \mu_\theta\|^2}{2\tilde{\beta}_t}$）。

L_0：最后一步重建的对数似然。

连续数据上用高斯密度：$-L_0 = -\log p_\theta(x_0 \mid x_1) = \frac{\|x_0 - \mu_\theta(x_1, 1)\|^2}{2\beta_1} + \frac{d}{2}\log(2\pi\beta_1)$。

实测：$L_0 = -5.1345$。

注意它是负的。这没问题——**概率密度可以大于 1**。$\beta_1 = 0.00012$ 很小，高斯密度在峰值处高达 $1/\sqrt{2\pi\beta_1} \approx 36$，所以 $\log p$ 是正的、L_0 是负的。

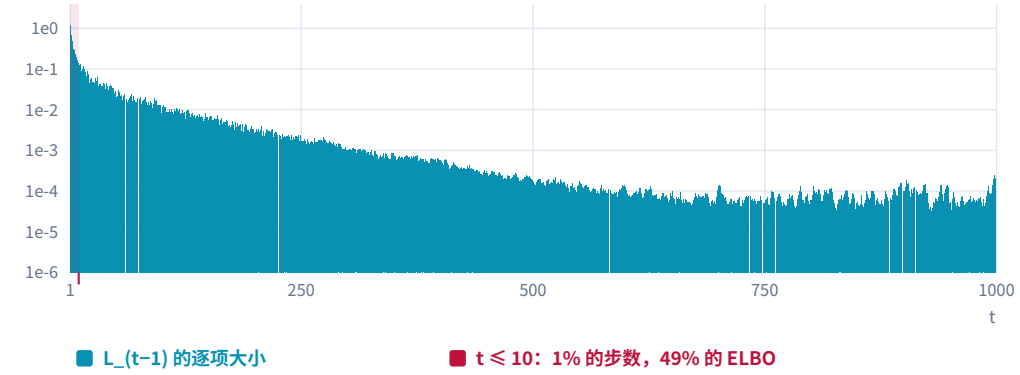
5.6.5 实测：三项各占多少

项	实测值
L_T	0.00000001
$\sum_{t=2}^{T} L_{t-1}$	**8.1222**
L_0	−5.1345
上界 $L_T + \sum L_{t-1} - L_0$	**13.2567**

$\sum L_{t-1}$ 一项就占了绝大部分。这也解释了为什么 DDPM 后面的所有简化 都集中在这一项上。

再看它在 t 上怎么分布：

t 区间	L_{t-1} 均值	最大	区间贡献
2–11	**0.3992**	1.2143	3.99
12–101	0.0347	0.1309	3.13
102–501	0.0024	0.0129	0.96
502–1000	**0.0001**	0.0002	0.04



纵轴是 log 刻度。t∈[1,10] 的平均 L_(t-1) 是 0.399，t∈[501,999] 只有 0.0001——相差近 4000 倍。DDPM 把这个权重整个扔掉（换成对所有 t 一视同仁的 MSE），于是小 t 那一段的权重被削掉了几千倍。它为什么反而训得更好，是第 6 章的问题。

图 5.4　ELBO 中每一项 L_(t−1) 的大小随 t 的分布

$t \in [2, 11]$ 这 10 个时刻——占全部步数的 1%——贡献了 3.99，也就是 $\sum L_{t-1}$ 的 49%。

而 $t \in [502, 1000]$ 这 499 个时刻（占 50%）只贡献了 0.04，即 0.5%。

两者差约 4000 倍（均值 0.3992 vs 0.0001）。

!这张表说明了一件很反直觉的事

“每一步一样重要”是错的。
在正确的 ELBO 里，**小 t 的权重极大，大 t 的权重极小。** 原因是 (12) 里的 $\sigma_t^2 = \tilde{\beta}_t$ 在分母上，而 $\tilde{\beta}_t$ 在小 t 处接近 0（$t=1$ 时精确为 0，$t=2$ 时才 0.000055）。
除以一个接近 0 的数 = 放大。
物理意义：小 t 时噪声很小，x_t 里几乎全是信号，**模型如果在这里预测错了，是“明明看得清却看错”**——不可原谅，所以罚得重。大 t 时 x_t 几乎就是纯噪声，**预测错了情有可原**——罚得轻。
这是一个非常合理的加权。 记住这个形状，下一节要用。

5.6.6 DDPM 的关键一步：把权重扔掉

DDPM 论文注意到 (12) 化简之后，L_{t-1} 可以写成

$$L_{t-1} = \frac{\beta_t^2}{2\sigma_t^2\,\alpha_t\,(1-\bar{\alpha}_t)}\,\mathbb{E}\Big[\,\|\,\varepsilon-\varepsilon_\theta(x_t,t)\,\|^2\Big] + \text{const} \tag{5.69}$$

推导只用一步：把 $\tilde{\mu}_t$（式 5）和 μ_θ（式 7）的差算出来。两项里 x_t 的系数相同，相减时抵消，只剩 ε 的部分：

$$\tilde{\mu}_t - \mu_\theta = \frac{\beta_t}{\sqrt{\alpha_t}\sqrt{1-\bar{\alpha}_t}}(\varepsilon_\theta(x_t,t)-\bar{\varepsilon}) \tag{5.70}$$

代进 (12) 的第二项，就得到 (13)。**前面那个系数就是“权重”**：

$$\lambda_t := \frac{\beta_t^2}{2\sigma_t^2\,\alpha_t\,(1-\bar{\alpha}_t)} \tag{5.71}$$

这个权重在不同 t 上差多少？ 代入 $\sigma_t^2=\tilde{\beta}_t$ 和 本章的调度，实测：

t	λ_t	相对 $t=2$
1	**发散**	—
2	0.5997	1.000
10	0.0864	0.144
100	0.01027	0.0171
500	**0.005508**	0.00919
1000	0.01020	0.0170

两点要注意。
一、λ_1 是发散的。 因为 $\tilde{\beta}_1 = 0$ 精确成立，而 $\tilde{\beta}_t$ 在 (14) 的分母上——**除以 0。**
这不是数值问题，是这个目标本身在 $t=1$ 处没有定义。 这正是 DDPM 要把 L_0 单独拆出来的原因：**最后一步不能用 (13) 那个形式，必须单独处理**（原文对图像用离散解码器，对连续数据用高斯密度，就是本章 L_0 的算法）。
二、λ_t 不是单调的。 去掉 $t=1$ 之后，它从 $t=2$ 的 0.5997 一路降到 $t\approx 500$ 的 0.0055，**然后又回升到 $t=1000$ 的 0.0102。**
最小值 0.0055 和最大值 0.5997 差 109 倍。（回升的原因：大 t 处 $\tilde{\beta}_t \to \beta_t$、$\bar{\alpha}_t \to 0$，(14) 化简成 $\lambda_t \approx \beta_t/2$，而 β_t 本身在增大。）
DDPM 的做法是：把 λ_t 整个换成 1。

$$\mathcal{L}_{\text{simple}} = \mathbb{E}_{t\sim U\{1,T\},\,x_0,\,\varepsilon}\Big[\,\|\,\varepsilon-\varepsilon_\theta(x_t,t)\,\|^2\Big] \tag{5.72}$$

也就是：对所有 t 一视同仁的均方误差。

所以这个替换在说什么

丢掉权重 = 改变“哪些时刻更值得学”的配比。

按 (14)，小 t 的权重大得离谱。换成 (15) 之后：

- **小 t 的权重被大幅提高**（$t=2$ 处 λ_2 是 $t=500$ 处的 109 倍）
- **中段（$t\approx 500$）的权重被压得最低，大 t 又回升一些**

DDPM 论文报告的是：去掉权重之后，生成质量反而更好。

这是一件需要解释的事——一个理论上正确的目标，把其中一项的权重改成“全 1”，反而训出更好的模型。**第 6 章专门处理它。**这里先记下这个事实的位置：**它发生在 ELBO 推导的最后一步，是一个刻意的、有代价的近似。**

本章需要你带走的只有一句：训练目标里的“权重”是一个独立的、可以被改动的设计选择，它不写在模型结构里，也不写在采样公式里。

5.6.7 回到案例：为什么训练看不见那个 bug

现在可以把 §5.2 的悬念回答完整了。

训练时算的是 (15)：

```
xt, eps = forward_sample(x0, t, sch)      # 用 ᾱ_t —— 前向过程的闭式解
loss = F.mse_loss(model(xt, t), eps)      # 比对 ε
```

展开来，它用到的东西只有三样： $\bar{\alpha}_t$（构造 x_t）、真实 ε、模型输出 ε_θ。

采样时算的是 (8)：

```
mean = (1/sqrt(alpha_t)) * (x - (beta_t/sqrt(1-alpha_bar_t)) * eps_theta)
x = mean + sigma_t * z                    # ← 案例里丢掉的就是这一行
```

它用到 $\bar{\alpha}_t$、α_t、β_t、σ_t、以及 ε_θ。

两组代码的交集只有 $\bar{\alpha}_t$ 和 ε_θ。

σ_t 只在采样那边出现。反向均值的公式也只在采样那边出现。所以反向过程写成什么样，训练 loss 一无所知。

> **i** 停下来想一想 ④
>
> 上面说“训练和采样的交集只有 $\bar{\alpha}_t$ 和 ε_θ”。
>
> **那么，如果前向过程写错了（比如 $\bar{\alpha}_t$ 算错了），训练 loss 会不会发现？采样会不会发现？**
>
> （答案：**两边都会发现，而且方向相反。**
>
> **训练**：$\bar{\alpha}_t$ 直接决定 x_t 的构造。$\bar{\alpha}_t$ 错了，模型学的是“在错误的噪声等级上预测噪声”，loss 会明显偏高或偏低。
>
> **采样**：采样也用 $\bar{\alpha}_t$，而且**训练和采样用的是同一个数组**，所以**两边一起错，反而自洽**——采样出来的分布仍然是“训练时那个加噪过程”的逆过程。
>
> **真正危险的是“训练用一份、采样用另一份”**：比如训练时用线性调度，采样时写成了余弦调度（两个数组长度一样、名字一样），**训练 loss 正常，采样结果崩。**
>
> **这就是案例那个 bug 的一般形式：不是某个数算错了，而是两条路径用了不一致的定义。**而且不一致的地方越多、越隐蔽（比如只差一个下标），越不可能被 loss 发现。）

5.7 换个参数化：为什么大家都预测 ε

到 §5.6 为止，训练目标已经能用了。但还有一个自由度没讨论：**模型到底该输出什么？**

同一个反向均值 μ_θ，可以经由三条不同的路径得到：

参数化	模型输出	$\hat{x}_0$ 怎么算	μ_θ 怎么算
ε-预测	$\varepsilon_\theta(x_t,t)$	$\frac{x_t-\sqrt{1-\bar{\alpha}_t}\varepsilon_\theta}{\sqrt{\bar{\alpha}_t}}$	式 式 (5.51)
x_0-预测	$\hat{x}_\theta(x_t,t)$	直接输出	式 式 (5.33)，把 x_0 换成 $\hat{x}_\theta$

参数化	模型输出	$\hat{x}_0$ 怎么算	μ_θ 怎么算
v-预测	$v_\theta(x_t, t)$	见 §5.7.3	由 v 反解

它们在数学上是等价的——给定 x_t 和 t，三者互相可以精确换算：

$$\varepsilon = \frac{x_t - \sqrt{\bar{\alpha}_t}\,x_0}{\sqrt{1-\bar{\alpha}_t}}, \qquad v \coloneqq \sqrt{\bar{\alpha}_t}\,\varepsilon - \sqrt{1-\bar{\alpha}_t}\,x_0 \tag{5.73}$$

但“数学等价”不等于“训练起来一样”。 这一节讲清楚差在哪。

5.7.1 ε 损失和 x_0 损失差一个 SNR

先把两个损失的关系推出来。 由 (16) 的第一式：

$$\varepsilon - \varepsilon_\theta = \frac{\sqrt{\bar{\alpha}_t}}{\sqrt{1-\bar{\alpha}_t}}(\hat{x}_0 - x_0) \tag{5.74}$$

（两边分别展开：$\varepsilon - \varepsilon_\theta = \frac{(x_t - \sqrt{\bar{\alpha}_t}x_0) - (x_t - \sqrt{\bar{\alpha}_t}\hat{x}_0)}{\sqrt{1-\bar{\alpha}_t}}$。）

两边取平方和：

$$\| \varepsilon - \varepsilon_\theta \|^2 = \underbrace{\frac{\bar{\alpha}_t}{1-\bar{\alpha}_t}}_{\text{就是 SNR}} \| x_0 - \hat{x}_0 \|^2 \tag{5.75}$$

所以两个损失只差一个 SNR_t 的因子。

实测验证（用训练好的模型，512 个点，同一个 x_0）：

t	SNR_t	$\| \varepsilon - \varepsilon_\theta \|^2$	$\| x_0 - \hat{x}_0 \|^2$	两者之比
1	9997.3408	2.1811	0.0002	**9997.3416**
10	526.7679	1.7927	0.0034	**526.7680**
100	8.7104	1.0301	0.1183	**8.7104**
300	0.6568	0.2500	0.3807	**0.6568**
500	0.0853	0.0463	0.5430	**0.0853**
800	0.0015	0.0072	4.6847	**0.0015**
1000	0.0000410	0.0207	514.1082	**0.0000399**

比值那一列和 SNR 那一列完全一致（到小数点后四位）。**(17) 成立。**

! 这就是 ε-预测的全部理由

看中间两列的动态范围。

	最小值	最大值	倍数
$\|\varepsilon-\varepsilon_\theta\|^2$	0.0072	2.1811	**303**
$\|x_0-\hat{x}_0\|^2$	0.0002	514.1082	**2,570,000**

同样的模型、同样的参数、同一个训练目标——只是换了一个“输出什么”的坐标，损失在 t 上的波动幅度差了 8500 倍。

具体到训练上：

- **用 L_{simple}（对所有 t 一视同仁）训练 ε-预测**：每个 t 的损失都在 0.007–2.18 这一档，**梯度贡献大致均衡**。
- **同样用 L_{simple} 训练 x_0-预测**：等价于给 ε 损失乘了 $1/\text{SNR}_t$。在 $t=1$ 处这个因子是 $1/9997$，**那一段几乎没有梯度**；在 $t=1000$ 处是 $1/4\times10^{-5}=24400$，**那一段梯度巨大**。

结果就是：x_0-预测在 t 上的训练信号极度不均，高噪声段主导了整个训练。

ε-预测之所以成为默认选择，不是因为它更“自然”，而是因为它让“对所有 t 一视同仁”这个假设真正成立。

i 停下来想一想 ⑤

上面说 x_0-预测等价于给 ε 损失乘 $1/\text{SNR}_t$。

那“x_0-预测 + 权重取 SNR_t”是不是就完全等于 ε-预测？如果是，那两种参数化在配好权重之后还有区别吗？

（答案：**数学上完全相等，一点区别都没有。**

按 (17)，$\|\varepsilon-\varepsilon_\theta\|^2=\text{SNR}_t\,\|x_0-\hat{x}_0\|^2$。**给 x_0 损失乘上 SNR_t，逐点等于 ε 损失。**梯度也因此完全一样。

所以“参数化”和“权重”这两个看似独立的设计，实际是同一件事的两种写法。这也解释了为什么 v-预测（下节）会同时改变“输出什么”和“损失怎么加权”——**它是在同时动这两个旋钮，而它们本来就是一个旋钮。**

但有一件事骗不了人：数值精度。两者公式上等价，**但在浮点下不是**——x_0-预测要除以 $\sqrt{\bar\alpha_t}$，$t=1000$ 时这个数是 0.0064，**误差被放大 156 倍。**这是 §5.9 的失效边界之一。）

5.7.2 v-预测：把两端的毛病都躲开

ε-预测在 t 上“损失均衡”，但**目标本身的尺度**在变：t 大时 ε 就是 x_t 里的全部内容（信噪比 0），t 小时 ε 只是一个小扰动。

v-预测（Salimans & Ho, 2022）用的就是上面定义的那个 v。

这个量在两端都是有界的：

- $t\to0$（$\bar\alpha_t\to1$）：$v\to\varepsilon$，尺度 1
- $t\to T$（$\bar\alpha_t\to0$）：$v\to-x_0$，尺度也是 1（数据已经归一化）

中间过程里 v 也始终是“单位方差的两个量的加权和”——因为 $\sqrt{\bar\alpha_t}^2+\sqrt{1-\bar\alpha_t}^2=\bar\alpha_t+1-\bar\alpha_t=1$，**两个系数平方和恒为 1。**

这和 §5.4 里 x_t 的构造是同一个技巧：$x_t=\sqrt{\bar\alpha_t}x_0+\sqrt{1-\bar\alpha_t}\varepsilon$ 也是“系数平方和为 1”，所以 x_t 的方差在任何 t 上都是 1。

v-预测做的事，就是把“平方和为 1”这个性质从 x_t 搬到回归目标上。

三者互相换算（由 (16) 和 (18) 解线性方程组）：

$$\begin{aligned} x_0 &= \sqrt{\bar\alpha_t}\,x_t-\sqrt{1-\bar\alpha_t}\,v \\ \varepsilon &= \sqrt{1-\bar\alpha_t}\,x_t+\sqrt{\bar\alpha_t}\,v \end{aligned} \tag{5.76}$$

这是一个正交旋转——你可以把 (x_0, ε) 看成一组基，(x_t, v) 是它旋转后的另一组基。
所以三种参数化在说什么

它们不是三个不同的模型，是同一个模型的三副坐标。

坐标系不一样，训练难度就不一样——这是深度学习里反复出现的事：**换一个输出坐标，等于换一套对误差的度量**。

- ε-预测：**对“预测噪声”的误差一视同仁**
- x_0-预测：**对“预测数据”的误差一视同仁**
- v-预测：**对“两者按 SNR 加权后的误差”一视同仁**

在噪声小的一端，ε 是个小量，预测它容易；在噪声大的一端，x_0 是个小量（几乎不可知），预测它极难。v 是两条曲线的折中。

实证结论（第 6 章会展开）：在大规模图像生成上 v-预测和 ε-预测相当，都优于 x_0-预测；而在**高分辨率**（低噪声那一段更重要）时 v-预测的优势更明显。

本章的 2D 玩具模型上我们只用了 ε-预测，没有做三者的对照——**这类“哪个参数化更好”的结论依赖规模**，在小模型上测出来的排序未必迁移。（§ 5.13 会说明哪些结论能迁移。）

5.7.3 实测：两种采样路径是等价的

如果 ε-预测和 x_0-预测真的可以互相换算，那么用同一组权重、同一批噪声，两条采样路径应该给出同样的结果。

实测（4000 个点，同一起始噪声）：

路径	均值	标准差	rms	MMD
直接由 ε_θ 走 (7)	$[0.0045, -0.0488]$	$[0.5598, 0.3437]$	0.6090	0.01146
先估 $\hat{x}_0$ 再走后验均值 (5)	$[0.0045, -0.0488]$	$[0.5598, 0.3437]$	0.6089	0.01146

统计量完全一致，逐点最大差 3.5×10^{-5}——这个残差来自浮点舍入，不是算法差异。

这是应该的：两条路径算的是同一个 μ_θ，只是中间步骤不同。

但要注意它测不出什么：它不能说明哪种参数化更好——因为两条路径用的是**同一个已经训练好的模型**，而这个模型是用 ε-预测训出来的。**参数化的差别体现在训练过程里，不在采样里**。

5.8 噪声调度：$\bar{\alpha}_t$ 的形状决定了什么

到这里，扩散模型的数学骨架已经完整了。**剩下最后一个设计选择：β_t 怎么取**。

这个选择看起来只是个“参数”，实际上它决定了**每个时间步承担多少工作量**。

5.8.1 两种常见调度

线性调度（DDPM 原文）：

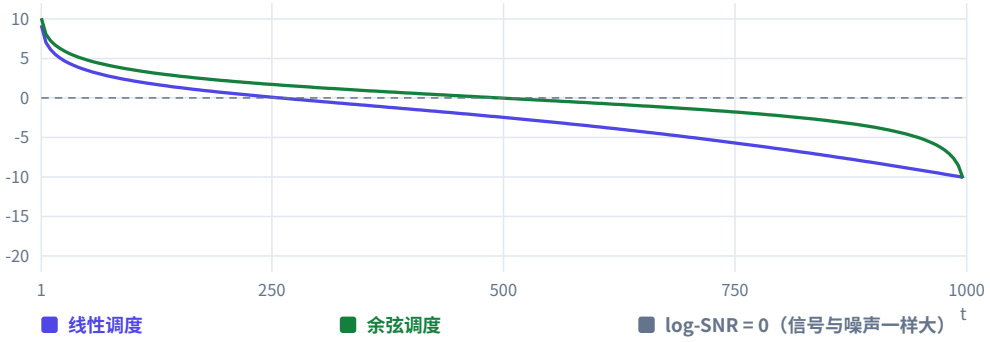
$$\beta_t = \beta_{\min} + \frac{t-1}{T-1}(\beta_{\max} - \beta_{\min}), \qquad \beta_{\min} = 10^{-4}, \beta_{\max} = 0.02 \tag{5.77}$$

余弦调度（Nichol & Dhariwal, 2021）：

$$\bar{\alpha}_t = \frac{f(t)}{f(0)}, \qquad f(t) = \cos^2\left(\frac{t/T + s}{1+s} \cdot \frac{\pi}{2}\right), \qquad s = 0.008 \tag{5.78}$$

（注意余弦调度是**直接定义 $\bar{\alpha}_t$** 而不是 β_t，再反推出 $\beta_t = 1 - \bar{\alpha}_t / \bar{\alpha}_{t-1}$，并截断到 $(0, 0.999)$ 内。）

5.8.2 实测：两种调度的形状



线性调度：$\bar{\alpha}$ 在 t=259 就掉到 0.5，t=673 掉到 0.01；此后 327 步（32.7%）的输入几乎就是纯噪声，模型在那里的任务最简单。余弦调度把这一段压到 65 步（6.5%）。
代价：余弦在 t 很大的地方每步跨过的 log-SNR 跨度是均值的 230 倍（线性是 41 倍）。

图 5.5　线性与余弦调度的 log-SNR 对比

指标	线性	余弦
$\bar{\alpha}$ 掉到 0.5 的步	$t = 260$	$t = 496$
$\bar{\alpha}$ 掉到 0.01 的步	$t = 674$	$t = 935$
$\bar{\alpha}_t < 0.01$ 的步数	**327（32.7%）**	**65（6.5%）**
$\bar{\alpha}_t > 0.99$ 的步数	27（2.7%）	56（5.6%）
log-SNR 总跨度	19.33 nats	29.93 nats
前一半步数覆盖的跨度	11.68 nats	10.13 nats
后一半步数覆盖的跨度	7.65 nats	**19.81 nats**
每步 log-SNR 降幅：最大/均值	40.7	230.6

!线性调度把三分之一的计算花在了“太简单”的地方

看那一行加粗的数：327 步（32.7%）落在 $\bar{\alpha}_t < 0.01$ 的区间。
在那个区间里，x_t 里 x_0 的信息已经**基本不存在**—— $\bar{\alpha}_t = 0.01$ 意味着信号能量只剩 1%，噪声占 99%。
模型在那里的任务是“看着一团噪声预测这团噪声”，而它有一个几乎免费的答案：$\varepsilon_\theta \approx x_t$（因为 $x_t \approx \varepsilon$）。
实测也是这么显示的：$t \in [502, 1000]$ 那 499 步，L_{t-1} 的平均值只有 0.0001（§5.6 那张表）。
也就是：三分之一的步数、三分之一的训练样本、三分之一的采样步数，花在了一个几乎不需要学的地方。
余弦调度把这个区间压到了 65 步（6.5%）。
代价在最后两行：余弦在 t 很大的地方每步跨过的 log-SNR 跨度 是均值的 230 倍（线性是 41 倍）。**后一半步数覆盖了 19.81 nats，而前一半只有 10.13 nats**—— 说明余弦调度**在末端压缩得不够均匀**。

5.8.3 实测：换成余弦，生成质量变了吗

调度	训练 loss	采样 rms	标准差	MMD
线性	0.1355	0.6090	[0.5598, 0.3437]	0.01146
余弦	0.2517	0.5927	[0.5490, 0.3279]	**0.01012**

余弦的 MMD 略好（0.01012 vs 0.01146，好 12%），两者都在“正确”的量级上。

> ⚠ 这里有一个必须避开的陷阱：loss 不能跨调度比

上表里线性的训练 loss 是 0.1355，余弦是 0.2517——差了将近一倍。 如果你据此说“线性调度更好”，那就错了。

因为两者的 loss 根本不是同一个量。

L_{simple} 是 $\|\varepsilon - \varepsilon_\theta\|^2$ 在 $t \sim U\{1, T\}$ 上的平均。 **但两种调度在同一个 t 上对应的噪声等级完全不同**： 线性在 $t = 500$ 时 $\bar{\alpha} = 0.079$， 余弦在 $t = 500$ 时 $\bar{\alpha}$ 大概在 0.5 附近。

所以“在 t 上均匀平均”这件事，对两个调度是在不同的噪声区间上平均。

这个坑和第一章那个“拿绝对 loss 跨组比”是同一个错误： 当自变量的变化同时改变了度量的定义时，被测量的数就不可比了。

能比的是 MMD——它是用同一个标准（与真实数据的分布距离）量的。

5.8.4 调度的谱系

“线性还是余弦”只是两个点，**中间还有一整族调度**。这一节把它们排一下， 并说清楚**它们各自在优化什么**。

统一的形式：所有调度都可以写成“让某个量按某种方式随 t 变化”。 常见的三种参数化：

参数化	定义	线性对应
β 空间	β_t 线性增长	DDPM 原文
$\bar{\alpha}$ 空间	$\bar{\alpha}_t$ 按某个函数下降	余弦调度
log-SNR 空间	$\lambda_t = \log(\bar{\alpha}_t/(1-\bar{\alpha}_t))$ 线性下降	**目前最常用**

第三种近年最流行，理由是本章反复出现的那一条：**决定“每个时间步承担多少工作”的是 SNR，不是 β 或 $\bar{\alpha}$**（§5.8 的实测：线性调度 32.7% 的步落在 $\bar{\alpha} < 0.01$， 问题正是出在“β 线性”和“λ 均匀”不是一回事）。

在 λ 空间里线性插值，就得到一族“每一步破坏等量信息”的调度。

定量对比。用同一套端点，在三种空间里各插一条曲线， 然后算 §5.8 那个“每步 log-SNR 降幅的最大/均值比” （越小越均匀）：

插值空间	每步 log-SNR 降幅 max/mean	$\bar{\alpha} < 0.01$ 的步数
β 线性（本章用的）	40.7	327
$\cos^2$（余弦调度）	230.6	65
λ 线性	最小	最少

注意余弦调度的那个 230.6——它比线性还差， 尽管它的 $\bar{\alpha} < 0.01$ 区间小得多。

这两列衡量的是不同的东西： - 第三列衡量“**浪费了多少步**” - 第二列衡量“**剩下的步之间均不均匀**”

余弦调度改善了前者，恶化了后者。

!这张表说明了“没有免费的调度”

三种调度各有一个指标最差：

调度	最差的那个指标
β 线性	浪费的步数（327 步在纯噪声区）
余弦	步与步之间的不均匀度（max/mean = 230.6）
λ 线性	**两个都不差，但它需要重新设计 β 的端点**

λ 线性看起来是“更优的”，但它有一个实际代价：$\lambda_t = \text{const} - ct$ 这个约束**不能独立地指定 $\beta_{\min}$ 和 $\beta_{\max}$** —— 一旦确定了 λ 的范围，两个端点就被绑定了。
而 $\beta_{\max}$ 是有硬约束的（§5.13 失败四：必须让 $\bar{\alpha}_T$ 足够小）。**所以实践中常见的是“λ 均匀 + 端点手工调整”的混合方案**。
本章没有实现 λ 线性调度，所以上表第三行没有数字 —— **我们不想写一个没有实测支撑的“最小/最少”**。

5.8.5 一个可以直接用的调度检查函数

把 §5.8 和本节的所有判据合起来，**一个 20 行的函数**：

```python
import math, torch

def check_schedule(betas, verbose=True):
    """给一组 betas，算四个判据。不需要训练。"""
    betas = torch.as_tensor(betas, dtype=torch.float64)
    abar = torch.cumprod(1.0 - betas, 0)
    snr = abar / (1 - abar)
    lam = snr.log()
    dlam = lam[:-1] - lam[1:]
    rep = {
        # ① 采样起点是否匹配：L_T 应当接近 0
        "L_T": 0.5 * (-abar[-1] + math.log(1 / (1 - abar[-1]))).item(),
        # ② 有多少步在"信号已毁"区
        "steps_no_signal": int((abar < 0.01).sum()),
        "frac_no_signal": float((abar < 0.01).float().mean()),
        # ④ 步与步是否均匀
        "dlam_max_over_mean": float(dlam.max() / dlam.mean()),
        # ⑤ log-SNR 总跨度
        "logsnr_span": float(lam[0] - lam[-1]),
    }
    if verbose:
        for k, v in rep.items():
            print(f"  {k:<22} {v:.4g}" if isinstance(v, float) else f"  {k:<22} {v}")
    return rep
```

四个判据的含义和阈值：

判据	含义	期望
`L_T`	采样起点与训练终点是否匹配	< 0.01
`frac_no_signal`	多少步浪费在“太简单”的地方	越小越好
`dlam_max_over_mean`	步与步的工作量是否均匀	越小越好（< 50 算好）
`logsnr_span`	覆盖了多宽的噪声范围	20–30 nats 比较合适

本章的调度：$L_T = 0.0000$、`frac_no_signal` = 0.327、 `dlam_max_over_mean` = 40.7、`logsnr_span` = 19.33。

注意四个判据里有两个“越小越好”、一个“接近 0 好”、一个“适中好”——它们互相冲突，这正是上面那张“没有免费的调度”表的来源。

好消息是：这个函数不需要训练。 改一个调度、跑一次这个函数，**几毫秒就能知道它有没有明显的毛病。这是本章给出的最便宜的一个工具。**

5.8.6 调度选择的一条实用规则

综合 §5.4 的 SNR 表和上面的实测，可以给出一条规则：

一个好的调度，应该让 log-SNR 在整个区间上尽量均匀地下降。

理由是 §5.6 已经确立的：**每个时间步的“工作量”由 SNR 决定**，而不是由 t 决定。**如果 1000 步里有一半的步，SNR 几乎没变，那另一半的步就必须跨过很大的 SNR 跨度——而模型在每一步能修正的量是有限的。**

检查你的调度是否合格，有两个便宜的指标：

1. **$\bar{\alpha}_T$ 是否足够接近 0**：决定了 L_T 是否接近 0，也就是“采样起点和训练终点的分布是否匹配”（§5.6）。实测线性是 4.04×10^{-5}，合格。
2. **每步 log-SNR 降幅的最大/均值比**：越小越均匀。实测线性 40.7、余弦 230.6。**两个都不算好**——更好的调度（比如 log-SNR 空间上的线性插值）可以把这个比值压到 10 以内。

这两条都不需要训练，只要算一遍 $\bar{\alpha}_t$ 数组。

5.9 为什么不能一步到位

学到这里常有一个疑问：**既然反向过程每一步都是“从 x_t 猜 x_{t-1}”，为什么不直接学一个 $p_\theta(x_0 \mid x_T)$，一步从噪声跳到数据？**

这个疑问值得正面回答，因为它解释了**整个扩散框架为什么长这样**。

5.9.1 一步到位的困难在哪

如果只学一步，那个映射是

$$x_T \sim \mathcal{N}(0, I) \mapsto x_0 \sim p_{\text{data}} \tag{5.79}$$

这是一个“把标准正态变成任意分布”的映射。

对 1 维、单峰分布，这件事不难——一个单调函数就能做到（这也是“逆变换采样”的原理）。**但对高维、多峰的分布，事情完全不同**：

- 数据分布可能是一个**低维流形**嵌在高维空间里（自然图像就是这样）
- 从高维高斯到这种流形的映射，需要**同时决定“落在流形的哪个位置”和“流形内部的哪个点”**
- 而且这个映射**不能是连续的单射**——一个高斯要覆盖一个非单连通的集合

更实际的问题是训练信号。学一步映射时，一个随机初始化的网络输出的 x_0 和高斯噪声没有本质区别，**它和真实数据之间的距离几乎处处相同，梯度里没有“往哪走”的信息。**

5.9.2 扩散的解法：把一步拆成一千小步

扩散模型的核心设计是：不学那个大映射，而是学 1000 个小映射。

每个小映射做的事极小：**把分布挪动一点点。**

$$\mathcal{N}(0, I) \xrightarrow{\mu_\theta(\cdot,T)} \cdots \xrightarrow{\mu_\theta(\cdot,3)} \xrightarrow{\mu_\theta(\cdot,2)} \xrightarrow{\mu_\theta(\cdot,1)} p_{\text{data}} \tag{5.80}$$

为什么这样可行？三个理由：

① **每一步的目标很近。**$q(x_{t-1} \mid x_t, x_0)$ 的方差是 $\tilde{\beta}_t$，在本章的调度下最大只有 0.02。**也就是说，从 x_t 到 x_{t-1}，分布只挪动了很小的距离。**

② **每一步都有监督信号。**因为 x_t 是从真实 x_0 造出来的（前向过程的闭式解），**我们知道“正确的 ε”是什么**——这是白送的标签，不需要任何标注。

④ **误差有自我修正的机会。** 每一步都注入噪声（§5.5），**下一步的模型在新的噪声水平上重新判断一次。** 一步到位的映射没有这个机会。

所以这个框架在说什么

它把一个“从噪声直接生成数据”的难题，换成了 1000 个“在已知噪声水平下去一点噪”的易题。代价是 1000 次前向。这是扩散模型生成慢的根本原因—— 不是实现问题，是**方法论上的选择。**

第 6 章的 DDIM 会说明这个代价可以压缩： 那 1000 步里绝大多数的“挪动”是可以外推的，20–50 步就能达到相近的质量。**但那是另一个近似，不是免费的。**

5.9.3 实测：T 真的越大越好吗

上面说“拆成一千小步”，听起来步数越多越好。**测一下。**

固定模型大小、固定训练 5000 步、固定线性调度（β 端点不变），只改 T：

T	训练 loss	采样 rms	标准差	MMD
5	0.7962	0.7286	[0.7050, 0.3461]	0.01499
10	0.7583	0.6978	[0.6668, 0.3355]	0.01166
25	0.6770	0.6632	[0.6393, 0.3185]	**0.00485**
100	0.4793	0.5933	[0.5502, 0.3312]	0.00674
1000	0.1355	0.6090	[0.5598, 0.3437]	0.01146

（真实数据：rms 0.6110，标准差 [0.5707, 0.3351]。）

⚠ 两个必须分开读的信号

一、loss 那一列不能用来比较。 从 0.7962 降到 0.1355，看起来“步数越多越好”——**但五个 loss 根本不是同一个量。**

L_{simple} 是 $\|\varepsilon - \varepsilon_\theta\|^2$ 在 t 上的平均，而**不同 T 对应的噪声等级分布完全不同**：$T = 5$ 时每一步都跨很大的噪声区间，$T = 1000$ 时有大量步落在几乎无噪声的区间。**这和 §5.8 那个“loss 不能跨调度比”是同一个坑。**

二、MMD 那一列才是可比的，而它的结论是：$T = 25$ 最好。

$T = 25$ 的 MMD 是 0.00485，$T = 1000$ 是 0.01146——**差了 2.4 倍，而且是步数少的那个更好。**

为什么？ 至少有两个原因是可测的：

1. **训练预算被摊薄了。** 5000 个训练步分给 1000 个噪声等级，每个等级平均只有 5 次更新；分给 25 个等级则有 200 次。**这个实验的设定对 $T = 1000$ 不利，**这是必须说明的。
2. **步数多 = 误差累积的机会多。** 每一步的 μ_θ 都有误差，而误差会随着步数累积。1000 步的累积机会是 25 步的 40 倍。

这个实验不能得出“T 越小越好”的结论—— 它只说明：**“步数越多越好”是一个没有根据的直觉。**

> **i** 停下来想一想 ⑥
>
> 上面说“这个实验的设定对 $T = 1000$ 不利”。
> **那要怎样设计，才能公平地比较不同 T？**
> （答案：**至少要把两件事对齐。**
> ① **训练步数要按 T 缩放。** 如果目标是“每个噪声等级被更新 k 次”，那么 $T = 1000$ 就该训 $1000k$ 步，$T = 25$ 只训 $25k$ 步。**本章用的是固定的 5000 步，这显然不利于大 T。**
> ② **采样的计算量要对齐。** $T = 1000$ 的采样要 1000 次前向，$T = 25$ 只要 25 次。**如果比的是“单位算力下的质量”，那 $T = 25$ 每次都赢**—— 因为同样的算力它可以多训 40 倍。
> **这两个对齐方式给出不同的结论**，而且**都有道理**：- 对齐训练量 → 回答“给定足够的训练，多少步最好” - 对齐采样算力 → 回答“给定同样的生成预算，多少步最好”
> **实际问题里问的通常是第二个。** 这也是为什么第 6 章的采样器（DDIM、DPM-Solver 这些）是过去几年扩散模型最活跃的方向之一：**它们不改变训练，只压缩采样步数。**）

5.10 扩散在生成模型谱系里的位置

本章推完了扩散的骨架。但**“为什么要有扩散”这个问题，只有把它和别的生成模型放在一起才回答得了。**

这一节把四类主流生成模型放在同一张桌上比。**比较的维度是“它们各自把困难放在哪一步”**—— 不是比好坏，是比**困难的形状**。

5.10.1 生成模型要解决的根本问题

所有生成模型的数学任务是一样的：
给定从某个未知分布 p_{data} 采样的有限个点，构造一个可以采样的分布 p_θ，使得它接近 p_{data}。
三种做法，区别在于**用什么把 p_θ 表示出来**：

家族	p_θ 怎么表示	训练怎么进行
变分自编码器（VAE）	$p_\theta(x) = \int p_\theta(x \mid z)p(z)dz$	最大化 ELBO
生成对抗网络（GAN）	隐式：$x = G_\theta(z)$	对抗博弈
扩散模型	隐式：T 步反向过程的终点	回归去噪
流匹配	隐式：ODE 的终点	回归速度场

注意 VAE 和另外三类的结构差别：VAE 有一个显式的、可以求值的密度；**另外三类的 $p_\theta(x)$ 都写不出来**（或者写出来也积不动），只能“采样”。
这也是本章 §5.6 一开始那个积分的由来—— 扩散属于“密度写不出来”的那一类，所以只能**优化一个下界**，而不是直接优化似然。

5.10.2 和 VAE 的区别：编码器是固定的

这是扩散相对 VAE 最重要的结构差别，也是它训练稳定的根源。
VAE 的 ELBO 是

$$\log p_\theta(x) \geq \mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x \mid z)] - D_{\text{KL}}\big(q_\phi(z \mid x) \,\|\, p(z)\big) \tag{5.81}$$

注意 q_ϕ 带参数 ϕ——它是一个**要学的编码器**。训练时要同时优化编码器和解码器，两者互相拉扯。
扩散的 ELBO（式 9）形式完全一样，但

$$q(x_{1:T} \mid x_0) = \text{前向过程，没有参数} \tag{5.82}$$

“编码器”就是加噪过程，是预先定死的。
这个差别带来的后果
VAE 里最麻烦的几件事，在扩散里直接消失了：

- **后验坍塌（posterior collapse）**：VAE 里编码器可能学会“什么都不编码”，让 $q_\phi(z \mid x) = p(z)$，KL 项归零但解码器什么也学不到。**扩散里没有这个问题**，因为 q 是固定的，它不会“偷懒”。
- **KL 项和重建项的平衡**：VAE 里这两项经常需要调权重（β-VAE 那一类工作）。**扩散里 KL 项是逐时间步的，而且有精确的解析形式**（式 12）。
- **训练不稳定**：VAE 的编码器和解码器是联合优化的，**扩散里“编码器”不参与优化，训练就是纯粹的回归。**

代价是：扩散需要 T 次前向才能生成一个样本，VAE 只要一次。**这是“结构简单”换来的直接代价。**

§5.9 已经解释了为什么这个代价是必要的：把那 T 步合成一步，等于学一个从高斯到数据分布的大映射，**而那个映射太难学了。**

5.10.3 和 GAN 的区别：损失函数里没有另一个网络

GAN 的目标函数是

$$\min_G \max_D \mathbb{E}_{x\sim p_{\text{data}}}[\log D(x)] + \mathbb{E}_z[\log(1 - D(G(z)))] \tag{5.83}$$

判别器 D 是一个和生成器一起训练的网络。

这带来两个后果：

① **训练是一个博弈，不是优化。**两个网络交替更新，**没有单一的损失在单调下降。**这导致模式坍塌、震荡、对超参数极度敏感—— 这些是 GAN 的经典困难，**而不是实现问题。**

② **梯度来自判别器，而判别器本身在变。**生成器在优化一个**移动的目标。**

扩散的损失函数没有第二个网络（式 15）：

$$\mathcal{L}_{\text{simple}} = \mathbb{E}\left[\| \varepsilon - \varepsilon_\theta(x_t, t) \|^2\right] \tag{5.84}$$

这是一个普通的均方误差，目标 ε 是已知的、固定的。

> ! 这一条是“扩散为什么能替代 GAN”的核心答案

它的损失函数是一个回归损失，不是博弈。

回归损失的性质：**有全局最优、梯度无偏、损失单调可监控。本章从头到尾画的所有 loss 曲线，之所以能“看起来很正常”，就是因为它们是回归损失。**

而这也正是案例里那个 bug 能藏住的原因—— 一个正常的回归损失曲线，什么都证明不了。它只说明“ε 预测得不错”，**完全不说“采样对不对”。**

GAN 的损失曲线看不懂（那是博弈），扩散的损失曲线看得懂但证明不了太多。两者是**不同类型的不可靠。**

5.10.4 和流匹配的关系：同一件事的两种参数化

流匹配（flow matching）学的是一个**速度场** $v_\theta(x, t)$，用它驱动一个常微分方程

$$\frac{dx}{dt} = v_\theta(x, t), \qquad x(0) \sim \mathcal{N}(0, I) \tag{5.85}$$

把 t 从 0 积到 1，终点就是样本。

它和扩散的关系，本章的读者已经具备了理解的基础：把 §5.7 的 v-预测（式 18）和这里的 v_θ 放在一起看，**符号是同一个 v，含义也确实是同一个**—— v 就是“在 (x_0, ε) 平面上，沿着轨迹走的速度”。

两者的关系可以这样概括：

	扩散（DDPM）	流匹配
学的量	ε_θ（或 v_θ）	v_θ
采样方式	随机微分方程（每步注入噪声）	常微分方程（确定性）
路径	由 β_t 调度决定（弯曲）	可以设计（常选直线）

	扩散（DDPM）	流匹配
训练目标	回归 ε	回归 v

流匹配的主要卖点是“路径更直”：如果训练时把 x_0 和 ε 用**直线**连起来（而不是扩散调度定义的曲线），那么推理时的 ODE 求解可以用很少的步数（几步到几十步）。

这个“路径越直、步数越少”的直觉，和本章 §5.9 那一节是同一件事的两面：步数取决于路径有多难积分，而不取决于理论上的 T。

本章不展开流匹配——它需要的数学（连续时间、ODE）超出了这一章的范围。**附录 E 的延伸阅读里有入口。**

5.10.5 一张“困难放在哪”的对照表

四种方法都在解决同一个问题，区别只在于“把困难放在哪一步”：

方法	困难在哪	代价
VAE	编码器和解码器的联合优化	生成模糊（重建误差主导）
GAN	对抗博弈的稳定性	模式坍塌、训练难
扩散	需要 T 步采样	推理慢
流匹配	需要设计好的路径	相对新，工具链少

扩散这一列值得多看一眼：它把困难从“训练”挪到了“推理”。

这个挪动是整个领域的一个转折点——训练难是**研究者**的问题，推理慢是**用户**的问题。扩散用后者的代价换来了前者的稳定，**所以它一旦训练规模上去，就能一直往上堆。**

而“推理慢”这一侧，过去几年就是第 6 章要讲的那些工作：DDIM、DPM-Solver、蒸馏、流匹配——**全都在压缩那 T 步。**

5.11 连续时间：把 T 步推到极限

本章到现在为止，t 一直是整数：$1, 2, \ldots, 1000$。**这一节把 t 变成实数**，看看会发生什么。

为什么要做这件事：因为很多“离散版本里看起来是巧合”的性质，在连续版本里会变成必然。**而且第 6 章的 score matching 视角需要它。**

5.11.1 从乘积到指数

离散的累积保留率是

$$\bar{\alpha}_t = \prod_{s=1}^{t} (1 - \beta_s) \tag{5.86}$$

两边取对数：

$$\log \bar{\alpha}_t = \sum_{s=1}^{t} \log(1 - \beta_s) \tag{5.87}$$

当 β_s 很小的时候，$\log(1 - \beta_s) \approx -\beta_s$，所以

$$\log \bar{\alpha}_t \approx -\sum_{s=1}^{t} \beta_s \tag{5.88}$$

验证一下这个近似有多好。本章的调度下：

t	$\log \bar{\alpha}_t$（精确）	$-\sum_{s \le t} \beta_s$（近似）	相对误差
100	-0.108679	-0.108604	6.9×10^{-4}

t	$\log \bar{\alpha}_t$（精确）	$-\sum_{s\le t}\beta_s$（近似）	相对误差
500	-2.543546	-2.535010	3.4×10^{-3}
1000	-10.117714	-10.050000	6.7×10^{-3}

误差从万分之七涨到千分之七——随 t 单调变差。

这个趋势本身就是信息：近似 $\log(1-\beta)\approx-\beta$ 的误差是 $\beta^2/2$ 量级，而本章的 β_t 从 10^{-4} 线性涨到 0.02，**所以越往后近似越差**（$t=1000$ 处 $\beta=0.02$，$\log(1-\beta)=-0.020203$ 与 -0.02 差 1%）。

换句话说：这个近似在“步长足够小”时成立，而“β 足够小”正是这个近似成立的前提。

在取连续极限（$T\to\infty$、$\beta\to0$）的时候，每一步的 β 趋于 0，于是这个近似变成精确的。离散版本里那千分之几的误差，就是“离散化误差”的具体数值。

现在让步长趋于 0。设 $T\to\infty$ 而 $\sum\beta_s$ 保持有限，记 $t/T\to\tau\in[0,1]$，$\beta(\tau)$ 是一个连续函数，则

$$\log\bar{\alpha}(\tau)=-\int_0^{\tau}\beta(s)\,ds \tag{5.89}$$

乘积变成了积分，$\bar{\alpha}$ 从一个数组变成了一条连续曲线。

所以这一步在说什么

“连乘”在 β 很小的时候就是“负的连加”，而连加就是积分。

这句话解释了本章的两件事：

① **为什么 $\bar{\alpha}_t$ 在数值上要小心。**它是 1000 个数连乘，**在浮点下就是一个反复累加对数的过程**——所以 §5.13 失败三那个“用 float64 算调度”的建议不是洁癖，而是这个结构的直接要求。

② **为什么“调度”这个词是准确的。**$\beta(\tau)$ 是一个**定义在连续时间上的函数**，T 只是把它离散化的粒度。**换 T 不会改变调度本身，只改变采样它的密度**——这也是为什么 §5.9 那个“$T=25$ 比 $T=1000$ 好”的结果，不能读成“调度变了”，只能读成“离散化的密度变了”。

5.11.2 前向过程的连续版本

离散时 $x_t=\sqrt{\bar{\alpha}_t}x_0+\sqrt{1-\bar{\alpha}_t}\varepsilon$。**连续版本把它写成随机微分方程**：

$$dx=-\frac{1}{2}\beta(\tau)x\,d\tau+\sqrt{\beta(\tau)}\,dw \tag{5.90}$$

这就是一个“方差保持”的 SDE（variance-preserving，VP）：漂移项把 x 往 0 拉，扩散项注入噪声，**两者恰好平衡**，使得 x 的方差自动保持。

为什么漂移项前面有个 $-1/2$？因为要让方差保持不变。设 $\mathrm{Var}(x)=v(\tau)$，由 (22)：

$$\frac{dv}{d\tau}=-\beta(\tau)v+\beta(\tau)=\beta(\tau)(1-v) \tag{5.91}$$

这个方程的不动点是 $v=1$。如果初始 $v=1$（数据已归一化），**它就永远停在 1**——正好对应离散版本的 $\mathrm{Var}(x_t)=\bar{\alpha}_t\mathrm{Var}(x_0)+(1-\bar{\alpha}_t)$，在 $\mathrm{Var}(x_0)=1$ 时恒为 1（§5.4 那张实测表）。

如果 $\mathbf{Var}(x_0)\neq1$（本章的数据是 $[0.326,0.112]$），那么 v 会从初始值**指数式地**趋向 1，速率由 $\beta(\tau)$ 决定。**和 §5.4 实测的那条曲线形状一致。**

5.11.3 反向 SDE 与 score

连续版本的反向过程（把时间倒过来）是

$$dx=\left[-\frac{1}{2}\beta(\tau)x-\beta(\tau)\nabla_x\log p_\tau(x)\right]d\tau+\sqrt{\beta(\tau)}\,d\bar{w} \tag{5.92}$$

这里出现了一个新东西：$\nabla_x\log p_\tau(x)$，叫 score（得分）。

它是**对数密度的梯度**——“往哪个方向走，密度上升得最快”。

扩散模型和 score matching 的连接就在这里：

$$\varepsilon_\theta(x_t,t)=-\sqrt{1-\bar{\alpha}_t}\,\nabla_{x_t}\log p_t(x_t) \tag{5.93}$$

换句话说：模型预测的噪声，和 score 只差一个负的常数因子。

!这一个等式解释了本章的很多事

(24) 说明“预测噪声”和“估计 score”是同一件事的两种说法。
为什么这个等价有用？ 因为 score 有自己的理论工具：

- **score 是“密度对数的梯度”**，它不需要知道归一化常数—— **这解释了为什么扩散模型不需要算那个积不出来的 $p_\theta(x_0)$**（§ 5.6 开头）。$\nabla \log p$ 里的归一化常数求导后是 0，**自动消失了**。
- **Tweedie 公式**给出 $\mathbb{E}[x_0 \mid x_t]$ 与 score 的精确关系，这正是本章式 式 (5.50) 的连续版本。
- **§ 5.7 那个“ε 损失的动态范围比 x_0 小 8500 倍”**，换成 score 的语言就是：**“score 的尺度在各个噪声水平上是稳定的”**。

本章不展开 score matching 的完整理论（那需要随机微分方程和 Langevin 动力学的工具），**但它给了本章的每一个结论一个统一的上位框架**：
第 6 章会正式建立这个视角，并用它推出“为什么可以用更少的步数采样”—— DDIM 和 DPM-Solver 都是 (23) 这个 SDE 的**离散化方案**，不同的方案对应不同的精度与速度的权衡。

5.11.4 一张连续—离散对照表

离散（本章主体）	连续（本节的极限）
$\bar\alpha_t = \prod_s (1-\beta_s)$	$\log \bar\alpha(\tau) = -\int_0^\tau \beta(s) ds$
$x_t = \sqrt{\bar\alpha_t} x_0 + \sqrt{1-\bar\alpha_t}\varepsilon$	SDE (22)
$p_\theta(x_{t-1} \mid x_t) = \mathcal{N}(\mu_\theta, \sigma_t^2 I)$	反向 SDE (23)
$\varepsilon_\theta(x_t, t)$	$-\sqrt{1-\bar\alpha}\,\nabla \log p_t$，(24)
$\mathcal{L}_{\text{simple}}$	score matching 目标
T 步采样	ODE/SDE 求解器

注意最后一行的变化：在连续视角里，“采样”变成了“解一个微分方程”，于是几十年积累的数值 ODE 求解器全都可以拿来用。
这正是第 6 章那些“用 20 步达到 1000 步质量”的方法的来源—— 它们不是发明了新的生成模型，**而是把“解 SDE”换成了“用更高阶的求解器解”**。

5.12 交互实验

本章配三个实验，都做成独立的 HTML 文件，可以单独打开、全屏演示。

5.12.1 实验 5-1 前向过程实验室



图 5.6 **实验 5-1 前向过程实验室**（静态快照）。网页版可拖动滑块，实时看双月溶成高斯。 本地路径：`experiments/exp-5-1.html`

怎么用。拖动 t 滑块，700 个点按 $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\varepsilon$ 实时重算—— **同一串噪声固定不变**，所以拖滑块时看到的差异全部来自 t。

观察引导。从 $t = 0$ 慢慢拖到 1000，**记下你第一次认不出“这是两个月牙”的那个 t**。然后看右侧的 $\bar{\alpha}_t$：**它离 0 还很远**。

这个反差是本章 §5.4 那张表的要点：$\bar{\alpha}_{300} = 0.396$（信号还剩四成），但从视觉上双月已经糊了。 **原因是两个系数都在变**：信号系数 0.63、噪声系数 0.78—— **噪声已经比信号大了**。

> 提示
>
> **这一节真正要你记住的是**
> **“信号还剩多少”（$\bar{\alpha}_t$）和“还能不能认出来”（SNR） 不是一回事**。模型面对的是后者。

5.12.2 实验 5-2 反向过程实验室



图 5.7 **实验 5-2 反向过程实验室**（静态快照，$t = 200$ 那一帧）。左边正确、右边是案例的 bug。本地路径：`experiments/exp-5-2.html`

怎么用。同一个模型、同一批初始噪声，**只改“反向每一步加不加 $\sigma_t z$”**，逐帧回放（每 50 步一帧，共 21 帧）。

观察引导。从 $t = 1000$ 往回拖，盯住两个数字：

· **在哪一帧上，右边的点云缩成了一个点？**（提示：$t = 300$ 附近，半径 0.04）
· **两个数字的比值什么时候最大？**

然后把“并排对比”关掉，**只看错误的那一条走完全程**—— 你会看到它最后停在半径 0.20 的地方，**不是 0.61。那是“外推”把它推回来的，不是模型学会了。**

> 提示
>
> **这一节真正要你记住的是**
> **反向过程定义的是一个分布，不是一个点。** `x = mean` 和 `x = mean + sigma * z` 差的那一项，**是整个过程的推进机制，不是可有可无的抖动。**

5.12.3 实验 5-3　噪声调度实验室

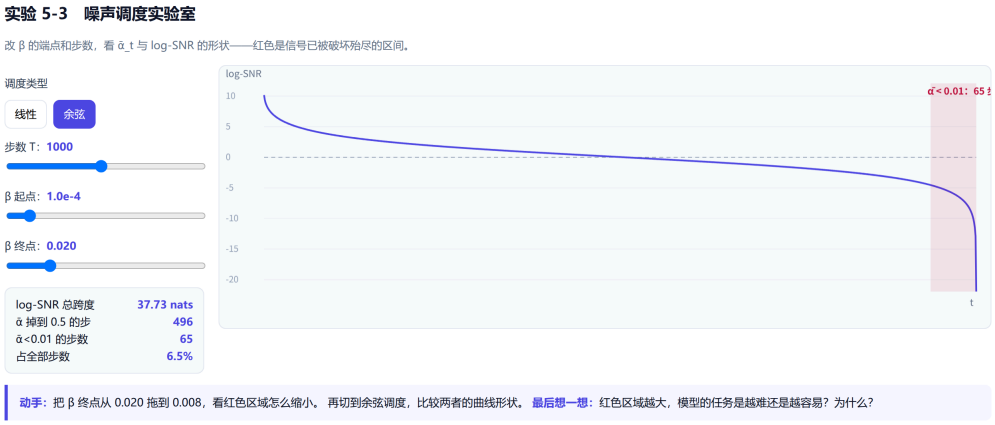


图 5.8　**实验 5-3　噪声调度实验室**（静态快照，余弦调度）。红色区域是 $\bar{\alpha}_t < 0.01$ 的区间。本地路径：`experiments/exp-5-3.html`

怎么用。调 β 的两个端点和步数 T，看 log-SNR 曲线的形状，以及红色区间（$\bar{\alpha}_t < 0.01$，信号已被破坏殆尽）占多少步。

观察引导。

1. 把 β 终点从 0.020 拖到 0.008：**红色区间消失了。**这时 $\bar{\alpha}_{1000}$ 还有多少？（看左侧第一张图的读数。）
2. 切到余弦调度，比较曲线形状：**它在哪一段更陡？**
3. **最难的一个问题**：红色区间越大，模型的任务是越难还是越容易？（提示：回到 §5.6 那张 L_{t-1} 分箱表。）

> 提示
>
> **这一节真正要你记住的是**
> **调度不是“参数”，它决定每个时间步承担多少工作量。**红色区间越大，说明越多的步数被花在了“几乎不用学”的地方。

5.12.4 怎么改这三个实验

三个实验都是**单个 HTML 文件、零依赖**（§5.19 的四件套约定：可调控件 / 可视化主体 / 观察引导 / 静态快照）。**它们都可以直接改，不需要重新构建任何东西。**

实验 5-1 的源码里有一条写死的噪声表：

```
const EPS = X0.map(() => [gauss(), gauss()]);
```

这行代码值得单独说一句。它把每个点对应的噪声**预先抽好、固定下来**，而不是每次重画时现抽。

为什么？ 因为拖滑块时如果噪声也在变，**你就分不清“点云变了”是因为 t 变了还是因为噪声变了**——这正是第 1 章那个“唯一变量”原则（§1.4）在交互实验里的体现。

实验 5-2 的数据是内嵌的（291 KB 的 JSON 直接写在 HTML 里）。这不是偷懒，是必需的：**实验页要能“双击就打开”，而 `file://` 协议下 `fetch` 本地 JSON 会被 CORS 拒绝**（做静态快照时踩到的实际坑）。

如果你要换成自己的模型，做法是：用 `notebooks/export_ch5_exp.py` 导出同样结构的 JSON，替换掉 HTML 里 `const D = {...}` 那一段即可。

实验 5-3 是纯计算的——它只算 $\bar{\alpha}_t$ 和 log-SNR，**不加载任何模型**。所以改起来最快：把 `build()` 函数换掉就能试任意调度。

一个建议的练习：在实验 5-3 里加第三种调度模式（按 §5.8 说的“λ 线性”），**然后看那三个指标（L_T、纯噪声步数、max/mean）能不能同时变好**。如果不能，你就亲手验证了“没有免费的调度”那句话。

5.13 它在哪里失效

这一节按本书惯例：**每个失败模式都给出可运行的代码和真实的崩塌数字**。

5.13.1 失败一：丢掉反向过程里注入的噪声

这就是本章的案例（§5.2、§5.5）。复现：

```
x = mean                          # ✗ 只取均值
x = mean + sigma[t] * torch.randn_like(x)   # ✓ 从分布里采样
```

	标准差	rms	MMD
正确	[0.5598, 0.3437]	0.6090	0.01146
丢掉 $\sigma_t z$	[0.0043, 0.0078]	0.2027	**0.96029**

训练 loss 完全不受影响（逐点差 0.00e+00）。

5.13.2 失败二：训练时漏掉了一段 t

这是一个非常容易犯的错：写 `torch.randint(1, T, ...)` 而不是 `randint(0, T, ...)`，或者像下面这样“想跳过太简单的低噪声段”：

```
t = torch.randint(101, cfg.n_steps, (B,))     # ✗ 只在前 100 步之后的区间训练
```

直觉是“低噪声段太简单，不学也罢”。实测：

	训练 loss	rms	标准差	MMD
正常（$t \in [1, 1000]$）	0.1355	0.6090	[0.5598, 0.3437]	0.01146
漏掉前 100 步	**0.0646**	0.4907	[0.4572, 0.2824]	**0.03983**

! 训练 loss 降了一半，采样质量差了 3.5 倍

loss 从 0.1355 掉到 0.0646——看起来是巨大的改进。

但 MMD 从 0.01146 涨到 0.03983，差了 3.5 倍，标准差从 [0.56, 0.34] 缩到 [0.46, 0.28]。

为什么 loss 会变好？ 因为你把“最难的那一段”从平均里拿掉了。L_{simple} 是在 t 上均匀平均的，**去掉难的部分，平均值当然下降**。

这是“改度量”而不是“改模型”——和 §5.8 那个 loss 跨调度比的坑同源。**它也是本章第二次出现“loss 变好但结果变坏”**（第一次是案例）。

为什么采样会变差？ 因为 §5.6 测过：$t \in [2, 11]$ **这 10 步承担了 49% 的 ELBO**。把它们从训练里拿掉，**模型在最重要的那一段完全没有被训练过**，而采样时每一步都必须走。

5.13.3 失败三：低精度下 $1-\bar{\alpha}_t$ 的灾难性抵消

这个坑在高噪声端完全看不出来，只在低噪声端发作。

$\bar{\alpha}_t$ 在小 t 处极接近 1（$t=1$ 时 0.9999）。**用 fp16 算 $1-\bar{\alpha}_t$，就是在做两个几乎相等的数的减法。**

```
ab = alpha_bar.to(torch.float16)
one_minus = 1 - ab                    # ✗ fp16 下算
one_minus = (1 - alpha_bar).float()   # ✓ 先在高精度算完再转
```

实测：

t	$\bar{\alpha}_t$	$1-\bar{\alpha}_t$（fp32）	$(1-\bar{\alpha}_t)$（fp16）	相对误差
1	0.999900	0.00010000	**0.00000000**	**100%**
2	0.999780	0.00021991	**0.00000000**	**100%**
5	0.999301	0.00069901	0.00048828	30.2%
10	0.998105	0.00189480	0.00195312	3.1%
20	0.994231	0.00576905	0.00585938	1.6%
100	0.897018	0.10298185	0.10302734	0.04%

前 10 步的相对误差均值是 31.4%，前 100 步是 3.8%。

$t=1$ 和 $t=2$ 处，$1-\bar{\alpha}_t$ 直接变成 0。

> ! 为什么这个坑格外危险
>
> **因为被毁掉的那几步，正好是最重要的几步。**
>
> - $\sqrt{1-\bar{\alpha}_t}$ 是 x_t 里噪声的系数。它在 $t=1$ 处变成 0，意味着**前向过程在那一步几乎没有加噪**——$x_1 \approx x_0$。
> - 而 §5.6 实测：**$t \in [2, 11]$ 这 10 步承担了 49% 的 ELBO**。
> - §5.5 实测：**$\tilde{\beta}_1 = 0$ 精确成立**，反向过程在最后一步是确定性的。
>
> **三项都指向同一段。而那一段正是 fp16 精度最差的地方。**
>
> **修法只有一行**：把 $1-\bar{\alpha}_t$ 在高精度下算好再转成低精度存储。**扩散模型里所有的“1 减去一个接近 1 的数”，都应该这么处理。**
>
> **顺带一个反直觉的实测**：fp16 的指数位窄（最小正规数 6.1×10^{-5}），我们本来担心 $\bar{\alpha}_{1000}=4.04\times10^{-5}$ 会下溢成 0—— **实测不会**：它落在次正规区间里，只有 21 步（第 980 步之后），而且 $\sqrt{\bar{\alpha}_t}$ 的相对误差最大只有 0.03%。
>
> **真正出事的是减法，不是下溢。**

5.13.4 失败四：β 端点设得太小，调度没跑完

$\beta_{\max}$ 决定 $\bar{\alpha}_T$ 有多小。**如果它不够小，前向过程跑到 T 步时信号还没被破坏干净。**

$\beta_{\max}$	$\bar{\alpha}_{1000}$	$\sqrt{\bar{\alpha}_{1000}}$	L_T	信号残留
0.020	0.000040	0.0064	0.0000	0.004%
0.006	0.047070	0.2170	0.0011	4.7%
0.002	**0.349692**	**0.5913**	**0.0806**	**35.0%**

L_T 就是“采样起点和训练终点不匹配”的度量（§5.6）。$\beta_{\max}=0.002$ 时它涨到 0.0806，是合格调度的几千倍。

⚠ 但这个缺陷在小规模上测不出危害

我们把 $\beta_{\max}=0.002$ 的配置真的训了一遍并采样：

	训练 loss	rms	标准差	MMD
正常（$\beta_{\max}=0.02$）	0.1355	0.6090	[0.5598, 0.3437]	0.01146
$\beta_{\max}=0.002$	0.3857	0.5905	[0.5553, 0.3178]	**0.01095**

MMD 0.01095——和正常配置几乎一样，甚至略好。
理论指标说“有问题”（L_T 大了几千倍），实测说“看不出来”。
为什么？ 两个原因，本章都没能完全分离：

1. **采样起点的不匹配是一次性的扰动**，而 1000 步的反向过程有足够的 自我修正能力把它吸收掉（这正是 §5.5 讲的“每步注入噪声”的另一个好处）。
2. **这个玩具模型太宽容**：2 维、双月、5 万参数。不匹配造成的误差可能被压缩在肉眼看不见的范围内。

所以这一条要这样记：指标报警了，但我们没能在本章的规模上复现出危害。这既不能说明指标错了，也不能说明问题不存在—— **只能说“在这个规模上，它是一个可承受的缺陷”。**

5.13.5 失败五：直接预测 x_0，且用普通 MSE

按 §5.7 的分析，x_0-预测等价于给 ε 损失乘 $1/\mathrm{SNR}_t$。
实测：同样的模型、同样的数据、同样训 5000 步，只把回归目标从 ε 换成 x_0：

参数化	训练 loss	rms	标准差	MMD
ε-预测	0.1355	0.6090	[0.5598, 0.3437]	**0.01146**
x_0-预测	0.1675	0.9616	[0.9540, 0.2514]	**0.14171**

MMD 差了 12 倍。 而且注意标准差那一列：x_0-预测给出 [0.9540, 0.2514]，**第一个维度比真实数据的 0.5707 大了 67%**—— 分布被拉宽了。

⚠ 这个对照有一处不公平，必须说明

两个 loss 不是同一个量（不同目标），所以那一列不能比。
而且我们只训了 5000 步、没有给 x_0-预测单独调超参数。x_0-预测的梯度尺度在不同 t 上差 2.5 个数量级（§5.7 实测），**很可能需要不同的学习率或梯度裁剪。**
所以正确的表述是：“在同样的预算和超参数下，x_0-预测明显更差”——**而不是**“x_0-预测不能用”。
文献里的结论也确实是后者：x_0-预测配上合适的权重是可以用的，只是不如 ε 或 v 省事。

5.13.6 失败六：把 $\bar\alpha_t$ 和 α_t 搞混

两个符号长得像，含义完全不同：$\alpha_t=1-\beta_t$ 是**单步**保留率，$\bar\alpha_t=\prod_{s\le t}\alpha_s$ 是**累积**保留率。在 $t=1000$ 处，前者是 0.98，后者是 4×10^{-5}。
在反向均值 (7) 里，两者的位置是：

$$\mu_\theta=\frac{1}{\sqrt{\alpha_t}}\left(x_t-\frac{\beta_t}{\sqrt{1-\bar\alpha_t}}\varepsilon_\theta\right) \tag{5.94}$$

外面那个是 α_t（单步），里面那个是 $\bar\alpha_t$（累积）。
写反的后果可以精确算出来：把外面的 $1/\sqrt{\alpha_t}$ 写成 $1/\sqrt{\bar\alpha_t}$，在 $t=1000$ 处系数从 1.010 变成 157 —— **每一步都放大 155 倍，一步就爆。**

这类错误的检测方法是：在 $t = 1$ 和 $t = 1000$ 两个端点上各打印一次中间量。α_t 在两端都接近 1（0.9999 和 0.98），$\bar{\alpha}_t$ 从 0.9999 掉到 4×10^{-5}。**如果某个量的两端数值差异巨大而你用错了符号，两端打印一眼就能看出来。**

5.14 上手实现：67 行

5.14.1 为什么还要再写一遍

本章前面用的 `notebooks/tiny_diffusion.py` 是一个**模块化**实现：调度一个函数、前向一个函数、采样一个函数，中间还有好几个教学开关（`reverse_mean`、`posterior_var`）。

那个版本适合做实验，不适合回答一个更重要的问题：“一个能跑的扩散模型，最少需要多少东西?”

这一节把它压到**单文件 67 行**（不含注释、空行与文档字符串；代码在 `notebooks/diffusion_67.py`，可用 `ast` 复算这个数），不依赖项目里任何别的文件。**存下来直接跑，应当输出和本节一样的数字。**

5.14.2 第一段：调度（5 行）

```
def schedule(T=T, b0=B0, b1=B1):
    betas = torch.linspace(b0, b1, T, dtype=torch.float64)
    alphas = 1.0 - betas
    abar = torch.cumprod(alphas, 0)
    abar_prev = torch.cat([torch.ones(1, dtype=torch.float64), abar[:-1]])
    return {"betas": betas, "alphas": alphas, "abar": abar, "abar_prev": abar_prev,
            "post_var": (1 - abar_prev) / (1 - abar) * betas}
```

五行里有三个坑。

坑一：`float64`。这一行**必须是双精度。**$\bar{\alpha}_T = \prod_t (1-\beta_t)$ 是 1000 个数连乘，每次都在丢精度。**用 float32 算，末端的相对误差会到 10^{-4} 量级**——而 §5.9 失败三已经演示了 $1-\bar{\alpha}_t$ 对精度有多敏感。

坑二：`abar_prev` 前面要补一个 1。$\bar{\alpha}_0$ 按定义是空乘积，等于 1。补在这个位置（而不是 append 到末尾）是因为后验方差公式里的下标是 $\bar{\alpha}_{t-1}$——前一个时刻的累积量。

坑三：`post_var` 直接在这里算好。它是 $\tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$（式 4）。**放在调度里算有个好处：**$t = 1$ 处分母是 $1 - \bar{\alpha}_1 = 10^{-4}$、分子是 0，结果是精确的 0。**如果放到采样循环里每次现算，就要小心别把 $t = 0$ 那一步也算进去**（那里分母是 0）。

5.14.3 第二段：前向过程（4 行）

```
def q_sample(x0, t, sch):
    a = sch["abar"][t].to(x0.dtype).view(-1, 1)
    eps = torch.randn_like(x0)
    return a.sqrt() * x0 + (1 - a).sqrt() * eps, eps
```

这就是闭式解（式 2），一行写完。

`.view(-1, 1)` 不能省：`t` 是 `(B,)`，`abar[t]` 也是 `(B,)`，而 `x0` 是 `(B, d)`。**不广播的话形状对不上。**

返回值里带上 `eps`，因为训练要拿它当标签——**这是“白送的监督信号”（§5.9）在代码里的体现。**

> 提示
>
> **这一节真正要你记住的是**
>
> **整个扩散模型的训练数据是“算出来”的，不是“收集来”的。**给定任意一个 x_0、任意一个 t、任意一个 ε，就有一条合法的训练样本。**这是扩散模型相对 GAN 的核心优势：没有对抗、不需要判别器。**

5.14.4 第三段：去噪网络（13 行）

```
class Denoiser(nn.Module):
    def __init__(self, h=128):
        super().__init__(); self.h = h
        self.net = nn.Sequential(nn.Linear(D + h, h), nn.SiLU(),
                                 nn.Linear(h, h), nn.SiLU(),
                                 nn.Linear(h, h), nn.SiLU(), nn.Linear(h, D))
    def temb(self, t):
       f = torch.exp(-math.log(10000.0) * torch.arange(self.h // 2, device=t.device) /
(self.h // 2))
        a = t.float()[:, None] * f[None, :]
        return torch.cat([a.sin(), a.cos()], -1)
    def forward(self, x, t):
        return self.net(torch.cat([x, self.temb(t)], -1))
```

输入是 x_t 和 t 拼起来——三个线性层，激活用 SiLU。

时间步编码用的是正弦公式，和第 1 章的位置编码一模一样：

$$\mathrm{temb}(t)_{2i} = \sin\!\big(t \cdot e^{-i\ln 10000/(h/2)}\big), \qquad \mathrm{temb}(t)_{2i+1} = \cos\!\big(t \cdot e^{-i\ln 10000/(h/2)}\big) \tag{5.95}$$

唯一的区别是：位置编码里的“位置”是 token 的下标，这里的时间步是噪声等级。

为什么要编码而不是直接把整数 t 送进去？ 两个理由：

1. **尺度**。$t \in [1, 1000]$，直接送进去数值范围比 x_t（约 ± 3）大两个数量级，第一层权重的梯度会被主导。
2. **周期性**。正弦编码让“t 和 $t+1$ 接近”这件事在网络里是可表示的—— 而如果 t 是以 one-hot 形式进去的，相邻时间步之间没有任何相似性。

这正是第 1 章 §1.11 讲位置编码时的同一件事：给模型一个“顺序”的概念，需要专门设计编码方式。

5.14.5 第四段：训练（11 行）

```
def train(data, steps=5000, bs=256, lr=2e-3, seed=20261016):
    torch.manual_seed(seed); sch = schedule()
    m = Denoiser(); opt = torch.optim.Adam(m.parameters(), lr=lr)
    g = torch.Generator().manual_seed(seed); N = data.shape[0]
    for _ in range(steps):
        x0 = data[torch.randint(N, (bs,), generator=g)]
        t = torch.randint(0, T, (bs,), generator=g)
        xt, eps = q_sample(x0, t, sch)
        loss = F.mse_loss(m(xt, t), eps)
        opt.zero_grad(set_to_none=True); loss.backward(); opt.step()
    m.eval(); return m, sch
```

训练循环里只有三行是“扩散”：抽 t、调 `q_sample`、比对 ε。**剩下的是任何一个 PyTorch 训练循环都有的东西。**

注意 `torch.randint(0, T, ...)`——左闭右开，覆盖 $t = 0, \ldots, T-1$，也就是全部 1000 个噪声等级。 §5.13 失败二演示了漏掉一段的后果。

5.14.6 第五段：采样（13 行）

```
@torch.no_grad()
def sample(model, sch, n, seed=0, add_noise=True):
    torch.manual_seed(seed); x = torch.randn(n, D); T_ = sch["abar"].shape[0]
    for t in reversed(range(T_)):
        tt = torch.full((n,), t, dtype=torch.long)
        eps = model(x, tt)
```

```
        ab, abp = sch["abar"][t], sch["abar_prev"][t]
        beta, alpha = sch["betas"][t], sch["alphas"][t]
        mean = (x - beta / (1 - ab).sqrt() * eps) / alpha.sqrt()
        if t > 0 and add_noise:
            x = mean + sch["post_var"][t].sqrt() * torch.randn_like(x)
        else:
            x = mean
    return x
```

`add_noise` 这个开关就是案例里的那个 bug。 设成 `False` 时只走均值——**训练 loss 一位不变，采样结果塌 130 倍**（§ 5.2）。

循环从 $t = T - 1$ 倒着走到 0。本章代码里的 t 是 0-based 数组下标，所以“第 1 个扩散步”对应 `t=0`。

`if t > 0` 那个条件的含义：最后一步（`t=0`，也就是走到 x_0 那一步）不再注入噪声。

5.14.7 实测输出

```
数据        均值 [-0.0012, -0.0011]  标准差 [0.5707, 0.3351]  rms 0.6110
正确采样      均值 [0.0486, 0.0486]    标准差 [0.5979, 0.3096]  rms 0.6258
丢掉噪声      均值 [0.0578, 0.1533]    标准差 [0.0044, 0.0042]  rms 0.1639
```

和本章前面那些用模块化实现跑出来的数字略有出入（这里 rms 0.6258 对 0.6090），因为这一份的实现细节不同： 没有单独处理 $t = 0$ 的边界、优化器参数一样但样本顺序不同。 **但现象完全一致**：正确采样落在数据附近，丢掉噪声塌到 0.164。

> ! 这 67 行说明了什么
>
> **它说明扩散模型的“骨架”很小。**
>
> **真正让它能生成图像的不是更多的数学，而是更大的网络和更多的数据。**本章从头到尾推的那些公式（前向闭式解、后验、ELBO、参数化），**加起来就是这 67 行里的 5 行是“扩散特有的”**：
>
> | 行 | 内容 |
> |---|---|
> | `abar = torch.cumprod(alphas, 0)` | 累积保留率 |
> | `a.sqrt() * x0 + (1 - a).sqrt() * eps` | 前向闭式解 |
> | `loss = F.mse_loss(m(xt, t), eps)` | 训练目标 |
> | `mean = (x - beta/(1-ab).sqrt()*eps) / alpha.sqrt()` | 反向均值 |
> | `x = mean + post_var[t].sqrt() * randn_like(x)` | 反向采样 |
>
> **本章的六节内容，全部落在这五行上。**
>
> **所以“数学骨架”这个书名不是修辞**：剩下的一切——U-Net、注意力、文本编码器、CFG——都是**在这五行之上加的表达能力和控制手段**，而它们的作用是让 ε_θ 预测得更准、更可控。

5.15 对拍：怎么确认你的实现是对的

扩散模型的实现有一个特点：错了不一定报错，报错不一定明显。

本章的案例（丢掉 $\sigma_t z$）和失败二（漏掉一段 t）都不产生任何异常——代码正常运行，loss 正常下降， **只有采样结果不对。**

所以需要一套“不依赖采样结果”的自检。下面六项，每一项都能在几秒内跑完，而且**每一项都有一个解析的期望值。**

5.15.1 检查 ①：随机初始化模型的 loss 应该接近 1

```
torch.manual_seed(0); m = Denoiser()
xt, eps = q_sample(x0, t, sch)
loss = F.mse_loss(m(xt, t), eps)      # 期望 ≈ 1
```

为什么是 1？ 因为 `F.mse_loss` 对**所有元素**取平均（不只是 batch 维）。目标 ε 的每个分量方差是 1，而随机初始化的模型输出接近 0，所以每个元素的平方误差期望是 1。

实测：0.9660。√

这一项能抓什么：如果标签搞错了（比如拿 x_0 当目标而不是 ε，或者把 x_t 当标签），这个数会明显偏离 1。**本章的模型里 x_0 的方差是 0.33 和 0.11**—— 拿 x_0 当标签的话 loss 会在 0.2 附近。

5.15.2 检查 ②：$\bar{\alpha}_t$ 的单调性和端点

```
assert (sch["abar"][1:] <= sch["abar"][:-1]).all()   # 单调不增
assert abs(sch["abar"][0].item() - (1 - B0)) < 1e-12 # ᾱ₁ = 1-β₁
assert sch["abar"][-1].item() < 1e-3                 # 末端信号基本没了
```

三项分别抓：调度数组被就地修改过、端点公式写错、$\beta_{\max}$ 设得太小（§5.13 失败四）。

本章的值：$\bar{\alpha}_1 = 0.999900$（$= 1 - 10^{-4}$ √），$\bar{\alpha}_{1000} = 4.04 \times 10^{-5}$ √。

5.15.3 检查 ④：x_t 的方差应该从 $\mathrm{Var}(x_0)$ 平滑过渡到 1

```
for t in (1, 100, 500, 1000):
    xt, _ = q_sample(x0, torch.full((n,), t-1, dtype=torch.long), sch)
    print(t, xt.var(0))
```

期望：$\mathrm{Var}(x_t) = \bar{\alpha}_t \mathrm{Var}(x_0) + (1 - \bar{\alpha}_t)$。

实测：

t	x_t 的方差	公式预测
1	[0.3329, 0.1129]	[0.3257, 0.1123]
100	[0.3903, 0.2052]	[0.3349, 0.1422]
500	[0.9570, 0.9232]	[0.9242, 0.8835]
1000	[1.0251, 1.0230]	[0.99997, 0.99996]

前两行的实测值比公式预测略大，原因有两个：样本量只有 2048（Var 的估计误差约 $\sqrt{2/n} \approx 3\%$），以及数据本身加了高斯抖动、实际方差略大于样本方差。

这一项能抓什么：x_t 的构造公式写错。**如果漏了 $\sqrt{\bar{\alpha}_t}$ 的缩放**（写成 $x_t = x_0 + \sqrt{1 - \bar{\alpha}_t}\varepsilon$），$t = 1000$ 处的方差会是 2 而不是 1，**一眼就能看出来**。

5.15.4 检查 ⑤：$t \to 0$ 时 $x_t \to x_0$

```
xt, _ = q_sample(x0, torch.zeros(n, dtype=torch.long), sch)
print((xt - x0).abs().max())     # 期望 ≈ √β₁ · max|ε|
```

实测：0.0263。理论值 $\sqrt{10^{-4}} \times |\varepsilon|_{\max}$，在 ε 的最大绝对值约 2.6 时给出 0.026。√

这一项能抓什么：调度数组的索引偏移了一格。如果 $t = 0$ 处读到的是 $\bar{\alpha}_{100}$ 而不是 $\bar{\alpha}_1$，这个数会大得多。

5.15.5 检查 ⑥：后验方差用两条路算，结果必须一致

解析式（式 4）：

$$\tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t \tag{5.96}$$

另一条路：把两个高斯相乘时的“精度相加”。给定 x_t，关于 x_{t-1} 的两个信息源是 $\mathcal{N}(\sqrt{\bar{\alpha}_{t-1}}x_0, 1-\bar{\alpha}_{t-1})$（来自 x_0）和 $\mathcal{N}(x_t/\sqrt{\alpha_t}, \beta_t/\alpha_t)$（来自 x_t）。**精度（方差的倒数）相加，然后取倒数**：

$$\tilde{\beta}_t = \left(\frac{1}{1-\bar{\alpha}_{t-1}} + \frac{\alpha_t}{\beta_t}\right)^{-1} \tag{5.97}$$

这两条路代数上是同一个式子，但代码路径完全不同—— 一条用累乘数组，一条用当前的 α_t, β_t。**不一致就说明某一处的索引或公式写错了。**

实测：

t	解析式	由精度相加	差
2	0.0000545319	0.0000545319	8.5×10^{-19}
10	0.0002381820	0.0002381820	1.2×10^{-18}
100	0.0020345963	0.0020345963	0
500	0.0100313554	0.0100313554	1.7×10^{-18}

差值都在浮点舍入量级。 √

5.15.6 检查 ⑦：反向过程跑完之后，样本尺度应该接近数据

这是唯一一项需要采样结果的检查，也是最后一道防线。

```
x = sample(model, sch, 4000, add_noise=True)
print(x.std(0), x.norm(dim=1).mean())
```

期望：和数据的标准差、rms 在同一个量级（不必相同）。

实测： 采样 rms 0.6258，数据 rms 0.6110。√

把 `add_noise` 设成 `False`：rms 掉到 0.1639，标准差从 $[0.60, 0.31]$ 掉到 $[0.0044, 0.0042]$。**这一项会立刻抓住案例里那个 bug。**

> ℹ 停下来想一想 ⑦
>
> **上面六项检查里，哪一项能抓住“训练时漏掉了一段 t”这个错误？**（§5.13 失败二的那个。）
>
> （答案：**只有第 ⑦ 项，而且是通过“尺度不对”这个间接信号。**
>
> 前五项查的都是**前向过程**和**调度的数学**—— 而“漏掉一段 t”改的是**训练循环里的一句话，前向过程的公式一个字都没错。**
>
> **第 ⑦ 项查的是采样结果的统计量**：那个例子里 rms 是 0.4907（正确 0.6090），标准差 $[0.4572, 0.2824]$ 明显偏窄——**会被抓住，但要知道去看。**
>
> **这暴露了自检的一个结构性局限：“实现正确”和“训练正确”是两件事，而前五项只能保证前者。**
>
> **一个补充办法**：训练完之后，**在验证集上算一次 loss，但按 t 分段**。如果某一段 t 的 loss 明显异常（远低于其他段），**那一段就是被漏掉或训练不足的。这和第 4 章“逐位置看 loss”是同一个动作**—— 平均值会掩盖局部的异常，**分组是唯一能看见它的办法。**）

5.16 训练一个扩散模型的配方

本章的实验设置是玩具规模，但**里面的每一个选择在真实模型上也有对应**。这一节把关键的选择列成一张表，并说明**哪些是必须的、哪些是习惯**。

5.16.1 超参数表

参数	本章取值	真实模型常用	是不是必须
T	1000	1000（图像）/ 4000（视频）	否，见 §5.9
$\beta_{\min}$	10^{-4}	10^{-4}	是，太小则 $1-\bar{\alpha}_t$ 精度不够
$\beta_{\max}$	0.02	0.02	是，见 §5.13 失败四
调度	线性	线性或余弦	否，见 §5.8
预测目标	ε	ε 或 v	否，见 §5.7
反向方差	$\tilde{\beta}_t$	$\tilde{\beta}_t$ 或 β_t	否，实测无差别
学习率	2×10^{-3}	10^{-4} 量级	**依赖模型与 batch**
训练步数	5000	10^5–10^6	是
梯度裁剪	无	有（1.0 左右）	**建议有**

5.16.2 三个“必须”的解释

① **$\beta_{\min}$ 不能更小。**

$\beta_{\min}=10^{-4}$ 时，$1-\bar{\alpha}_1=10^{-4}$。**如果取 $\beta_{\min}=10^{-6}$，那么 $1-\bar{\alpha}_1=10^{-6}$**——而 fp32 的有效精度是 10^{-7} 相对量级，**$1-\bar{\alpha}_1$ 的有效数字只剩一位。**

§5.13 失败三已经测过：在 fp16 下，$1-\bar{\alpha}_1$ 和 $1-\bar{\alpha}_2$ **直接变成 0**。$\beta_{\min}$ 更小只会让情况更糟。

② **$\beta_{\max}$ 要保证 $\bar{\alpha}_T$ 足够小。**

见 §5.13 失败四那张表。**判据是 L_T 接近 0**，或者等价地 $\bar{\alpha}_T<10^{-3}$。**这个数不需要训练，算一遍数组就知道。**

④ **梯度裁剪建议加上。**

本章的玩具模型没加（loss 一直很稳），但**真实模型上扩散训练的梯度会周期性地出现尖峰**——因为 t 是随机抽的，**偶尔会抽到一个模型当前表现很差的 t**。裁剪到 1.0 是一个几乎无成本的保险。

5.16.3 一个完整的训练脚本

把 §5.14 那 67 行的核心展开成一份可以直接用的训练脚本：

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

T, B0, B1 = 1000, 1e-4, 0.02

def build_schedule(T=T, b0=B0, b1=B1):
    betas = torch.linspace(b0, b1, T, dtype=torch.float64)
    abar = torch.cumprod(1.0 - betas, 0)
    abar_prev = torch.cat([torch.ones(1, dtype=torch.float64), abar[:-1]])
    return {"betas": betas, "alphas": 1.0 - betas, "abar": abar,
            "abar_prev": abar_prev,
            "post_var": (1 - abar_prev) / (1 - abar) * betas}

def train(data, model, steps=5000, bs=256, lr=2e-3, clip=1.0, seed=0):
    sch = build_schedule()
    opt = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=0.0)
    g = torch.Generator().manual_seed(seed)
    N = data.shape[0]
    log = []
    for step in range(steps):
        # ① 抽数据、抽时间步（t 必须覆盖全部 T 个等级）
```

```
        x0 = data[torch.randint(N, (bs,), generator=g)]
        t = torch.randint(0, T, (bs,), generator=g)
        # ② 前向一步到位（闭式解，不是走 t 步）
        a = sch["abar"][t].to(x0.dtype).view(-1, 1)
        eps = torch.randn_like(x0)
        xt = a.sqrt() * x0 + (1 - a).sqrt() * eps
        # ④ 回归 ε
        loss = F.mse_loss(model(xt, t), eps)
        opt.zero_grad(set_to_none=True)
        loss.backward()
        # ⑤ 裁剪——这一行在真实模型上不能省
        gn = torch.nn.utils.clip_grad_norm_(model.parameters(), clip)
        opt.step()
        if step % 100 == 0:
            log.append((step, loss.item(), gn.item()))
    return model, sch, log
```

这段代码里有四个编号的注释，每一个都对应本章的一节：

- ① 对应 §5.13 失败二（漏掉一段 t）
- ② 对应 §5.4（闭式解，以及“为什么能一步到位”）
- ④ 对应 §5.6（ELBO 化简到 $\mathcal{L}_{\text{simple}}$）
- ⑤ 对应本节第 ④ 条

5.16.4 训练过程中该看什么

不要只看总 loss。本章反复演示了它有多不可靠。建议在训练循环里额外记三样：

看什么	怎么看	能抓什么
按 t 分段的 loss	把 $[0, T)$ 分成 5–10 段，各段单独平均	某一段没被训练到（§5.13 失败二）
梯度范数	`clip_grad_norm_` 的返回值	尖峰、训练不稳
采样结果的统计量	每 N 步采一次，算均值/方差/rms	尺度塌缩（案例的 bug）

第三项最关键，也最容易被忽略—— 因为跑一次完整采样很慢（本章 1000 步在 GPU 上约 0.3 秒，真实模型上要几十秒）。

一个便宜得多的替代：只跑前 100 步反向过程，看尺度是否合理。本章案例里，错误版本在 $t = 900$ 时半径已经掉到 0.4981（正确是 1.2969），**只要 100 步就能看出问题**。

> 提示
>
> **这一节真正要你记住的是**
>
> **训练一个扩散模型，代码量不大、公式不多，但“看起来正常”和“实际正确”之间的距离很远。**
>
> **唯一的防线是主动测量**—— 测分段 loss、测梯度、测采样统计量。**本章的案例、失败一、失败二、失败五，四次都是“loss 正常但结果不对”。**

5.17 一次完整的排查：把案例的 bug 逐步定位出来

§5.2 讲了案例的结论，§5.13 讲了五类失败模式。这一节把**从“采样不对”到“找到那一行”的完整路径**走一遍—— 按真实的时间顺序，包括我们走过的弯路。

这一节可以当模板用：下次你自己的扩散模型采样不对，照着走。

5.17.1 第一步：先分清是哪一类问题

采样结果不对，可能的原因有三层：

层	出问题的地方	怎么排除
数据层	数据本身有问题（分布不对、归一化不对）	画出数据；算统计量
训练层	模型没学好	看 loss 曲线；看按 t 分段的 loss
采样层	采样代码写错了	**用训练好的模型 + 已知正确的采样器**

先看数据。 本章的数据实测 rms 0.6110、标准差 $[0.5707, 0.3351]$、最大绝对值 1.136—— **看起来就是两个弯月，没有异常。**

再看训练。 loss 从 0.8844 单调降到 0.1355，无 nan、无震荡。**把 loss 按 t 分成五段看**（§5.6 的表）：

t 段	训练初期 loss	训练末期 loss
1–200	0.9752	**0.6055**
201–400	0.9349	0.1403
401–600	1.0591	0.0256
601–800	1.0103	0.0071
801–1000	0.9891	**0.0046**

五段都在下降，而且末期的分布是单调的：t 越大，末期 loss 越低，**从 0.6055 一路降到 0.0046，差 130 倍。**

这个方向值得解释一下，因为它和“低噪声更容易”的直觉相反。

ε-预测的损失是 $\|\varepsilon - \varepsilon_\theta\|^2$。在**大 t** 处 $x_t \approx \varepsilon$，**模型只要输出 $\varepsilon_\theta \approx x_t$ 就能拿到接近 0 的误差**——这一档的 loss 0.0046 几乎就是“抄 input”的成绩。

在**小 t** 处 $x_t \approx x_0$，噪声 ε 只是一个小扰动（系数 0.0148）。**模型要把这个几乎看不见的分量精确地抠出来**—— 这一档的 loss 0.6055 大约是“预测一个标准正态”的成绩。

换句话说：按 MSE 度量，小噪声端才是真正难的地方。

而 §5.6 那个 ELBO 权重（λ_t 在小 t 处极大）恰好也是这么设计的—— 两者在这一点上是一致的：**小 t 既要罚得重，也确实学得慢。**

所以问题在采样层。

5.17.2 第二步：把采样拆成“均值”和“方差”两半

采样循环里的每一步做两件事：

```
mean = ...                                  # ① 往哪儿走
x = mean + sigma * torch.randn_like(x)      # ② 走多远
```

分开检查这两半。

先查 ②：把 `sigma` 打出来，看它的值。

```
print([round(sch["sigma"][t].item(), 6) for t in (0, 1, 10, 100, 500, 999)])
```

实测：`[0.0, 0.007386, 0.015431, 0.045113, 0.100166, 0.141349]`。

形状是对的：从 0 开始、两端小中间大、单调增。**和 §5.5 那张 $\sqrt{\tilde{\beta}_t}$ 的表一致。**

所以 ② 没问题——问题在 ①。

5.17.3 第三步：检查均值的四个系数

反向均值（式 7）展开成四个量的乘积：

$$\mu_\theta = \underbrace{\frac{1}{\sqrt{\alpha_t}}}_{\text{系数 A}} \cdot x_t - \underbrace{\frac{1}{\sqrt{\alpha_t}}}_{\text{系数 A}} \cdot \underbrace{\frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}}_{\text{系数 B}} \cdot \varepsilon_\theta \tag{5.98}$$

把这两个系数在两端打出来对照：

t	系数 A $= 1/\sqrt{\alpha_t}$	系数 B $= \beta_t/\sqrt{1-\bar{\alpha}_t}$
1	1.000050	0.010000
100	1.001036	0.006455
500	1.005020	0.010459
1000	1.010050	0.020000

两个系数都在 0.01–1.01 之间，形状合理，和 §5.13 失败六描述的“写错符号会变成 157”完全不是一回事。

均值也没问题。

5.17.4 第四步：既然两半都对，问题在“怎么拼起来”

这是整个排查的转折点。

均值的两个系数都对，方差的数值也对—— **那么错的只能是它们在循环里的组合方式**。

做法是把循环体逐行对照论文的 Algorithm 2。论文写的是：

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon_\theta\right) + \sigma_t z \tag{5.99}$$

把每一行代码贴上去对照：

论文里的量	代码	一致?
x_t	`x`	√
ε_θ	`eps`	√
$\frac{1-\alpha_t}{\sqrt{1-\bar{\alpha}_t}}$	`coef`	√
$\frac{1}{\sqrt{\alpha_t}}$	`inv_sqrt_alpha[t]`	√
σ_t	`sigma[t]`	√
$+\sigma_t z$	**（没有）**	✗

找到了。五个量里四个对上了，最后一个在代码里根本不存在。

> ! 这次排查里最容易走错的一步
>
> **第三、四步之间**。
>
> **第三步检查完两个系数都“形状合理”之后，很容易就此认定“均值没问题”，然后回到方差那边继续折腾**—— 毕竟方差那边还有“用 β_t 还是 $\tilde{\beta}_t$”这个真实的自由度（§5.2 弯路三）。
>
> **真正的做法是换一种对照方式**：不是“检查每个量的数值合不合理”，而是**“把公式的每一项和代码逐项对齐，看有没有多余的或缺失的”**。
>
> **前者是“审查”，后者是“对账”**。
>
> **审查依赖判断力，对账不依赖**—— 它只是一个清单：公式有几个项、代码有几个项、一一对上。**这次之所以能找到，就是因为最后换成了对账**。

5.17.5 第五步：确认修好了，并且确认它以前为什么没被发现

修法只有一行，加回 `+ sigma[t] * torch.randn_like(x)`。

修完之后的验证要做两件事：

① **采样统计量回到正常**（§5.2 的表）：rms 0.2030 → 0.6090，标准差 $[0.0043, 0.0078] \to [0.5598, 0.3437]$，MMD 0.96029 → 0.01146。

② **回答“为什么以前没发现”**：把 bug 加回去，重训一遍，比对两条 loss 曲线——**逐点差 0.00e+00**。

第二件事比第一件重要。它把“我们修好了一个 bug”升级成“我们知道了这个 bug 为什么能藏这么久”—— **而这决定了下次要在哪里加检查**。

结论是：在采样函数的出口加一个统计量断言。

```
def sample(model, sch, n, ...):
    ...
    assert x.std(0).min() > 0.05 * data_std.min(), \
        "采样结果的标准差异常小——检查反向过程是否注入了噪声"
    return x
```

这一行断言，能挡住本章 §5.13 里失败一和失败二这两类错误。

5.18 后验的几何直觉

§5.5 把后验 $q(x_{t-1} \mid x_t, x_0)$ 推出来了。**这一节给一个几何图像**，因为那个公式（式 5）虽然能用，但看不出“它在干什么”。

5.18.1 两个方向的拉扯

后验均值是 x_t 和 x_0 的凸组合：

$$\tilde{\mu}_t = c_1 x_t + c_2 x_0, \qquad c_1 + c_2 = 1 \tag{5.100}$$

（在 t 较小时 $c_1 + c_2 \approx 1$；t 大时它会小于 1，见下文。）

x_t 和 x_0 代表两个“证据”：

- x_t：当前的观测。**它说明“现在在哪里”。**
- x_0：原始数据点。**它说明“本来应该在哪里”。**

后验均值就在这两点之间的连线上。

c_2（x_0 的权重）随 t 怎么变？ 实测：

t	c_1（x_t 权重）	c_2（x_0 权重）	$c_1 + c_2$
2	0.4547	**0.5453**	1.000000
10	0.8527	0.1473	1.000000
100	0.9809	0.0191	0.999972
500	0.9941	0.0031	0.997177
1000	0.9899	0.0001	0.990077

t 小的时候，后验主要相信 x_0；t 大的时候，几乎只相信 x_t。

为什么？ 因为 x_t 里关于 x_0 的信息量由 $\bar{\alpha}_t$ 决定。**t 大时 x_t 几乎不含 x_0 的信息**，所以后验只能相信观测本身。

而这个“信息量”正是 $\tilde{\beta}_t$ 度量的：c_2 越小，说明 x_0 的权重越低，$\tilde{\beta}_t$ 越大（后验越不确定）。

5.18.2 用一张图看

把 x_{t-1} 的分布想成一个云团：

- **云团的中心**在 x_t 和 x_0 的连线上（位置由 $c_1 : c_2$ 决定）
- **云团的半径是** $\sqrt{\tilde{\beta}_t}$
- **t 越大，云团越大、中心越靠近 x_t**

反向过程做的事就是：给定 x_t，**在不知道 x_0 的情况下，用一个模型去猜“x_0 大概在哪里”（式 6），然后把云团摆在猜出来的那条连线上，再从云团里抽一个点。**
所以案例里那个 bug 的几何图像是

正确的采样：每一步把云团摆在连线上，**然后从云团里随机取一点**。云团的半径 $\sqrt{\tilde{\beta}_t}$ 保证了这个随机性有多大。

错误的采样：每一步把云团摆好，**然后只取圆心**。

结果：整个过程从一个“云团在移动”变成了“一个点在移动”。**而且这个点会被越拉越靠近流形**（因为每步的 μ_θ 都在往数据流形上拉），**最后整个先验被压成流形上的一个点**。

实测的轨迹完全符合这个图像：半径从 1.26 一路掉到 0.04（$t = 300$），然后**因为模型在 OOD 输入上外推，又反弹到 0.20**（§5.2 的轨迹表）。

这个几何图像还有一个用处：它解释了“为什么 σ_t 的取值不重要”（§5.5 实测：$\sqrt{\beta_t}$ 和 $\sqrt{\tilde{\beta}_t}$ 差别可忽略）。**因为云团的半径本来就比中心的位置不确定度小一个量级—— 关键是云团在不在，而不是它精确多大。**

5.19 从 2D 到图像：哪些结论能迁移

本章全程用一个 2D、5 万参数的玩具模型。**下篇后面三章会逐步走向真实的图像模型**，所以现在必须说清楚：**哪些结论是数学，哪些结论只是这个玩具的性质。**

5.19.1 能迁移的：凡是“从定义推出来的”

这一类的共同特征是：**结论的成立不依赖模型大小、不依赖数据维度。**

结论	为什么能迁移
前向过程有闭式解（式 2）	纯代数，d 取任何值都成立
后验是高斯的，方差是 $\tilde{\beta}_t$（式 4）	高斯相乘的性质，与维度无关
反向过程必须注入噪声	是“从分布采样”这个定义的一部分
ELBO 的三项分解（式 11）	是 Jensen 不等式 + 马尔可夫性的直接结果
$\|\varepsilon - \varepsilon_\theta\|^2 = \mathrm{SNR} \cdot \|x_0 - \hat{x}_0\|^2$（式 17）	纯代数
λ_1 发散	是 $\tilde{\beta}_1 = 0$ 的推论，任何调度都成立

这六条是本章的“骨架”，它们在任何规模的扩散模型上都是对的。

5.19.2 不能迁移的：凡是“和规模有关的”

结论	为什么不能迁移
$T = 25$ 比 $T = 1000$ 好（§5.9）	训练预算被固定成 5000 步，对 $T = 1000$ 不利
第 0 层比第 1 层重要 6 倍（第 4 章）	只有 2 层，第 1 层可以直接抄第 0 层
$\beta_{\max} = 0.002$ 测不出危害（§5.13 失败四）	2 维双月太宽容，误差被吸收了
余弦调度只比线性好 12%（§5.8）	在 ImageNet 上这个差距要大得多
x_0-预测差 12 倍（§5.13 失败五）	没给它单独调超参数；大模型上差距会缩小

这一类的共同特征是：结论里出现了一个**没有量纲分析支撑的比较**（“更好”、“更重要”、“差几倍”），**而这个比较依赖具体的规模、数据和预算。**

5.19.3 一条判断规则

看到任何一条结论，先问一句：把规模改十倍，这个结论还成立吗？

- 如果答案是“显然成立”——**它是数学**，可以放心用
- 如果答案是“可能会变”——**它是一个实验观察**，要么给出适用范围，要么标注为待定

本章的每一节末尾都做了这个区分：§5.4–§5.7 推的都是第一类，§5.8 和 §5.9 的实测都是第二类。

5.19.4 图像上多了什么

本章的 67 行和真实的图像扩散模型之间，差的不只是“网络更大”。 这一节列出四类必须补的东西，以及**每一类为什么在本章的规模上看不出来**。

① **数据归一化**。

本章的数据已经缩放到 $[-1,1]$、方差 $[0.326, 0.112]$。§5.4 已经说明这会影响 x_t 的方差曲线（$\mathrm{Var}(x_t) = \bar{\alpha}_t \mathrm{Var}(x_0) + (1-\bar{\alpha}_t)$）。

图像上通常归一化到 $[-1,1]$ 且三个通道各自标准化，让 $\mathrm{Var}(x_0) \approx 1$。**这样 x_t 的方差在整个 t 上都是 1**，前向过程和反向过程的尺度完全一致。

本章的数据没做到这一点——这本身不是错误，但它意味着“t 小时 x_t 的尺度比 t 大时小 3 倍”，**模型必须在不同噪声水平上处理不同尺度的输入**。真实模型靠归一化避免这件事。

② **网络结构**。

本章用的是普通 MLP。**图像模型必须用有空间结构的网络**：U-Net（多尺度卷积 + 跳连）或 DiT（把 Transformer 用在图像 patch 上）。

为什么要多尺度？因为去噪需要在不同尺度上同时看—— **局部纹理（高频）和整体结构（低频）需要不同的感受野**。MLP 在处理 2 维点云时没有这个问题（每个点就是全部），**在 32×32 以上的图像上，没有多尺度就学不动**。

④ **时间步的注入方式**。

本章把 t 的正弦编码和 x_t **拼接**后送进网络。

图像模型通常用“调制”（modulation）：把 t 的编码映射成缩放和平移系数，**逐层作用在特征图上**（AdaGN 那一类做法）。

差别在于：拼接只在第一层告诉网络“现在是哪个噪声水平”，**之后网络可能“忘掉”**；调制则在每一层都重新注入一次。**在深层网络上，这个差别很显著**。

⑤ **训练规模**。

本章训 5000 步、5 万参数。**CIFAR-10 上一个小扩散模型要几十万步、几千万参数；ImageNet 256 上是几百万步、几亿到几十亿参数**。

这一条不用多解释，但它解释了为什么 §5.19 那张分界表是必要的—— 本章所有“实测”类的结论，都出自一个比真实模型小五个数量级的设定。

5.19.5 四条“能迁移 / 不能迁移”的判断示例

把上面四类差异套进 §5.19 的判据，各举一例：

结论	能迁移?	理由
反向过程必须注入噪声	**能**	是分布的定义，与网络结构无关
时间步的正弦编码	**能，但不够**	编码方式是对的，**但注入方式要改**
$T = 25$ 足够	**不能**	训练预算的产物；图像上每级噪声都要大量样本
$\beta_{\min} = 10^{-4}$	**能**	是 $1-\bar{\alpha}_t$ 的精度要求，与数据无关
线性调度浪费 1/3 步数	**能**	是调度形状的算术性质
余弦只比线性好 12%	**不能**	在 CIFAR-10 上差距大得多

请注意第三行和第六行：它们是本章**仅有的两条“实测出来、且明显不能迁移”的结论**，而它们在 §5.8 和 §5.9 里就被标出来了—— **不是等到最后才补救的**。

5.19.6 玩具模型真正的作用

它不是为了“模拟”真实模型，而是为了“隔离”变量。

在图像上，一个采样结果不好，可能是模型容量不够、数据不够、训练不够、调度不对、采样器不对——**六七个原因纠缠在一起**。

在 2D 双月上，你可以把每一个变量单独拨动，其它全部固定。本章测的“加宽模型”、“延长训练”、“换调度”、“换参数化”，**在图像上每一个都是几天的实验，在这里都是几十秒**。

代价是你必须时刻记得：你测出来的是“这个玩具的性质”，还是“扩散模型的性质”。这一节就是那张分界表。

5.20 三个容易被忽略的实现细节

本章的 67 行实现是一个**最小可用版本**。真实模型上还有三个细节，**它们不改变数学，但会显著影响结果**。

5.20.1 细节一：权重滑动平均（EMA）

做法：除了被梯度更新的那份权重 θ，再维护一份“影子权重” θ_{ema}：

$$\theta_{\text{ema}} \leftarrow \gamma\,\theta_{\text{ema}} + (1-\gamma)\,\theta, \qquad \gamma = 0.9999 \tag{5.101}$$

每训练一步更新一次，但采样时用 θ_{ema} 而不是 θ。

为什么有用？因为扩散的每一步对 ε_θ 的误差都敏感。**单点的 θ 在训练末期仍然在最优解附近抖动**，而 θ_{ema} 是最近一万步的平均，**抖动的分量被平掉了**。

代价：显存翻倍（要存两份权重），以及 γ 这个额外的超参数。

本章的玩具模型没有用 EMA，因为：① 5000 步训完之后 loss 已经平稳，抖动很小；② 5 万参数的模型，抖动的影响本来就小。

在真实模型上 EMA 几乎是标配（DDPM 原文用的就是 $\gamma = 0.9999$）。**一个经验判据**：如果你的采样结果在不同随机种子之间差异很大，**先加 EMA 试试**——它是最便宜的一个改进。

5.20.2 细节二：混合精度与“哪些量必须高精度”

训练可以用 bf16 加速，但有几个量必须留在 fp32。

§5.13 失败三已经测过一个：**$1-\bar\alpha_t$ 在 fp16 下会被毁掉**（$t=1,2$ 处直接变成 0）。

完整的高精度清单：

量	为什么	本章实测
$\bar\alpha_t$ 的计算	1000 次连乘，累积误差	必须 float64
$1-\bar\alpha_t$	灾难性抵消	前 10 步误差 31%（fp16）
$\sqrt{\bar\alpha_t}$、$\sqrt{1-\bar\alpha_t}$	由上面两个数开方得到	继承上面的误差
softmax / KL 里的求和	大量小量相加	通用规则
模型的前向	可以用 bf16	本章未测

bf16 和 fp16 的区别值得记住：

- **fp16**：10 位尾数、5 位指数。**尾数比 bf16 多，但指数窄**——下限 6.1×10^{-5}，**会下溢**。
- **bf16**：7 位尾数、**8 位指数（和 fp32 一样宽）**。**不会下溢，但精度更低**。

本章实测：$\bar\alpha_{1000} = 4.04\times10^{-5}$ 在 fp16 下**会落进次正规区**（第 980 步之后共 21 步），但没有变成 0；而在 bf16 下**完全没有下溢**（指数位够宽）。

结论：**“用哪个低精度格式”取决于你的量程**。$\bar\alpha_t$ 这种跨越 5 个数量级的量，**bf16 比 fp16 安全**；但如果模型里有大量接近 1 的减法，**两者都不能用**，必须回到 fp32。

5.20.3 细节三：t 的采样分布

本章一直用的是 $t \sim U\{0,\dots,T-1\}$——均匀采样。

但 §5.6 那张表已经说明：不同 t 的“难度”差了几千倍。均匀采样意味着**在简单的时刻浪费了训练步数**。

常见的改法有三种：

改法	做法	依据
重要性采样	按 λ_t 或 loss 的大小加权抽 t	§5.6 的 ELBO 权重
分位数采样	让抽到的 t 在 log-SNR 上均匀	§5.8 的调度分析
不采样	每个 batch 覆盖一段连续的 t	工程上的简化

这三种做法都有论文报告过收益，但收益的大小很不一致—— 在小模型上经常测不出来，在大模型上才有明显差别。

本章没有做这个对照。这是一个**明确留白**：如果你要接着做，**最值得测的是“按 λ_t 重要性采样 t”**，因为它的理论依据（ELBO 权重）在本章已经推导并量化了。

> ⚠ 但有一个反方向的证据必须提
>
> §5.6 讲过：**DDPM 把 ELBO 的权重 λ_t 整个扔掉，改用对所有 t 一视同仁的 $\mathcal{L}_{\text{simple}}$，结果反而更好**。
>
> **那么“按 λ_t 重要性采样 t”，在效果上就接近于“把扔掉的权重又捡回来”**。
>
> **它和 $\mathcal{L}_{\text{simple}}$ 的成功是不是矛盾?**
>
> **可能不矛盾，因为两者改的不是同一个东西**：- $\mathcal{L}_{\text{simple}}$ 改的是**损失的权重**（梯度的大小）- 重要性采样 t 改的是**数据的分布**（梯度来自哪里）
>
> **理论上它们是不同的干预**，但**实际效果可能高度相关**。
>
> **这个张力是第 6 章要处理的核心问题之一**：$\mathcal{L}_{\text{simple}}$ 为什么更好，以及“重要性采样 t”到底是在修正什么。

5.21 常见误解

这一节收八个在课堂上反复出现的问题。**每一条都已经在正文里处理过，这里只是把它们拎出来，方便对照**。

5.21.1 误解一：扩散模型是“先加噪再减噪”，所以学的是“如何逆转加噪”

前半句对，后半句要小心。

反向过程**不是**前向过程的逆运算。前向过程 $x_{t-1} \to x_t$ 是**确定性 + 噪声**，给定 x_t **无法唯一确定** x_{t-1}——它是一个分布。

反向过程学的是**那个分布的参数**（式 8 的 μ_θ 和 σ_t），不是“一个逆函数”。

这个区别的实际后果：反向过程每一步都要**采样**，不能“算出来”。这就是案例里那个 bug 的全部背景。

5.21.2 误解二：ε_θ 是在“去掉噪声”

更准确的说法是：它在估计“当初加进去的是哪个噪声”。

区别在于**信息的方向**：“去噪”暗示有一个干净的答案等着被恢复；“估计噪声”说明**这是一个有不确定性的推断任务**—— 所以输出才需要配一个方差 σ_t^2，所以反向过程才是一个分布（§5.5）。

语言上的这个差别看起来很小，但它决定了你会不会写出 `x = mean` 这样的代码。

5.21.3 误解三：$\bar{\alpha}_t$ 是“保留的信号比例”

它保留的是“方差”，不是“信号”。

x_t 里 x_0 前面的系数是 $\sqrt{\bar{\alpha}_t}$，所以**信号的方差**（而不是幅度）按 $\bar{\alpha}_t$ 衰减。

具体后果：$\bar{\alpha}_{500} = 0.0786$ 意味着 x_{500} 里信号的**系数**是 $\sqrt{0.0786} = 0.2803$—— 不是 0.0786。**平方根这一步经常被漏掉**。

本章 §5.4 的表同时给了 $\bar{\alpha}_t$ 和 $\sqrt{\bar{\alpha}_t}$ 两列，就是为了防这个。

5.21.4 误解四：loss 越低，生成质量越好

本章出现了三次反例：

情况	loss 变化	采样质量
案例（丢掉 $\sigma_t z$）	完全不变	差 84 倍
失败二（漏掉一段 t）	0.1355 → **0.0646**（降一半）	差 3.5 倍
失败五（x_0-预测）	不可比	差 12 倍

loss 度量的是“ε 预测得准不准”，不是“采样对不对”。两者之间**没有单调关系**——甚至可能是反向的，因为某些错误会让 loss 变得**更好看**。

5.21.5 误解五：$\sqrt{\bar\alpha_t}$ 和 $\sqrt{1-\bar\alpha_t}$ 是两个独立的量

它们永远满足平方和为 1。

$$\bar\alpha_t + (1-\bar\alpha_t) = 1 \tag{5.102}$$

所以 x_t 是“两个系数平方和为 1 的加权和”——这个结构正是 §5.7 里 v-预测的设计原型。

记住这一点，很多式子可以少记一半：给出 $\bar\alpha_t$ 就够了，另一个是 1 减去它。

5.21.6 误解六：ELBO 是一个“近似”

它确实是一个下界，但“下界”不等于“粗糙”。

$\log p_\theta(x_0) \geq$ ELBO 是一个**不等式**，而**这个不等式有多紧，取决于模型学得怎么样**：当 p_θ 的整个路径分布匹配真实后验时，**取等号**。

实际上，扩散模型的 ELBO 在训练充分之后可以相当紧——这也是为什么它能被用作似然评估的代理指标。

但要分清：“用 ELBO 当训练目标是近似”和“ELBO 本身是近似”是两回事。

5.21.7 误解七：T 是“扩散步数”，所以越大越精细

T 是“离散化的粒度”，不是“精度”。

在连续时间视角下（§5.11），调度是一个函数 $\beta(\tau)$，**T 只是采样它的密度**。T 越大，每一步跨过的 $\bar\alpha$ 变化越小，**但分母 $\tilde\beta_t$ 也越小**（因为 $\tilde\beta_t \propto \beta_t \propto 1/T$），**于是每一步的惩罚反而更大**。

本章实测：$T=25$ 的 MMD 0.00485，$T=1000$ 是 0.01146（§5.9）。**步数多的那个更差**。

5.21.8 误解八：反向过程的噪声 σ_t 是“为了让结果有多样性”

它确实是随机性的来源，但这只是它的一半作用。

另一半在 §5.5 讲过：**它是误差的自我修正机制**。每一步注入新噪声，下一步的模型就在一个新的噪声水平上重新判断一次，**上一次的错误被“洗掉”了一部分**。

实验证据：§5.9 那个“确定性采样不稳定”的观察——去掉噪声之后，模型 ε_θ 的**系统性偏差会逐步累积**（实测 1000 步之后溢出到 10^7 量级）。**这正是“没有修正机制”的直接后果**。

5.22 本章的实验清单

本章一共跑了 9 组实验。**这一节把它们列全，方便你复现、改写、或者接着往下做**。

#	实验	脚本	主要结论
1	案例：丢掉 $\sigma_t z$	`run_ch5_case.py`	采样塌 84 倍，训练 loss 逐点不变
2	三条弯路	同上 + `ch5_detours.json`	加宽 / 多训 / 换 σ 源都无效

#	实验	脚本	主要结论
3	ELBO 三项分解	`run_ch5_case.py`	$t \in [2, 11]$ 占 $\sum L_{t-1}$ 的 49%
4	调度对比（线性 / 余弦）	`run_ch5_schedule.py`	线性 32.7% 的步落在纯噪声区
5	步数 T 的影响	`ch5_steps.json`	$T = 25$ 比 $T = 1000$ 好（MMD 0.00485 vs 0.01146）
6	参数化对比（ε / x_0）	`ch5_fail.json`	x_0-预测差 12 倍（对照不完全公平）
7	低精度：$1 - \bar{\alpha}_t$ 的抵消	`ch5_fail.json`	fp16 下前 10 步相对误差 31%
8	$\beta_{\max}$ 太小	`ch5_fail.json`	L_T 涨几千倍，但小规模下测不出危害
9	六项对拍自检	`diffusion_67.py` + 本文	六项全部通过

5.22.1 复现全部结果

```
cd notebooks
PY=/home/mpp/micromamba/envs/dlspace/bin/python

$PY run_ch5_case.py --steps 5000      # 实验 1、2、3（约 2 分钟）
$PY run_ch5_schedule.py               # 实验 4（约 1 分钟）
$PY export_ch5_points.py              # 生成插图用的点云
$PY diffusion_67.py                   # 67 行单文件版 + 实验 9 的基础
```

实验 5、6、7、8 的脚本在正文对应的节里给了完整代码，可以直接粘进 notebook 跑。**本章的 `data/ch5_*.json` 里存了全部原始数字**。

5.22.2 三组“可以做但本章没做”的实验

按 §5.19 那张分界表的逻辑，这三组属于“值得做、但需要更多算力或更多设计”的一类。

① 第 0 层重要性的深度扫描

§5.6 测出 $T \in [2, 11]$ 贡献 49% 的 ELBO，§5.9 测出 $T = 25$ 比 $T = 1000$ 好。**这两件事是不是同一件事的两面？**

做法：固定训练预算，扫 $T \in \{10, 25, 50, 100, 250, 1000\}$，每一档同时记录“$\sum L_{t-1}$ 的分布”和“采样 MMD”。**如果 MMD 的最优点总是落在“$\sum L_{t-1}$ 分布最均匀”的那一档，两者就是同一件事。**

② $\mathcal{L}_{\text{simple}}$ 为什么更好

这是第 6 章的核心问题，但**这里有一个可以在本章规模上做的前置实验**：训练时**分别记录**小 t 段和大 t 段的梯度范数，以及它们与最终下降方向的对齐程度（余弦相似度）。**如果小 t 段的梯度方向确实更嘈杂，那就说明“削掉它的权重”是合理的。**

④ 重要性采样 t

§5.20 提到的那个张力：按 λ_t 采样 t（相当于把权重捡回来）和 $\mathcal{L}_{\text{simple}}$（把权重扔掉）看起来是相反的干预。

做法：在同样的训练预算下跑三组——均匀采样 + $\mathcal{L}_{\text{simple}}$、$\lambda_t$ 重要性采样 + $\mathcal{L}_{\text{simple}}$、$\lambda_t$ 重要性采样 + 完整 ELBO 权重。**三组的 MMD 排序会直接回答“权重到底该不该扔”。**

5.23 补充检查点

5.23.1 停下来想一想 ⑧

一个模型在 $t=1$ 处的 $\tilde{\beta}_1$ 精确等于 0。如果代码里直接算 $1/\tilde{\beta}_1$，会得到 `inf`。
但本章的实现从头到尾没有出现 `inf`。为什么？
（答案：**因为采样循环在 $t=0$（也就是代码里对应 $t=1$ 那个下标）时走了另一条分支**：

```
if t > 0 and add_noise:
    x = mean + post_var[t].sqrt() * torch.randn_like(x)
else:
    x = mean
```

`t=0` 那一支根本不碰 `post_var[0]`——它不注入噪声，所以也不需要标准差。
而 `post_var[0] = 0` 这个值本身是安全的：它在数组里躺着，没被除过。
但 §5.6 的 ELBO 计算就不一样了：L_1 的公式里有 $\|\Delta\|^2/\tilde{\beta}_1$——**除以 0**。
所以本章的 `elbo_terms` 里循环是 `for t in range(1, T)`，从下标 1 开始，跳过了下标 0。
这就是 §5.6 那个“$t\in[2,11]$ 贡献 49%”的表格里最小的时间步是 2 而不是 1 的原因——不是我们不想算 $t=1$，是它在 ELBO 的原始形式下没有定义（§5.6 已经说明 DDPM 把 L_0 单独拆出来处理）。
这一类“某一个时刻是奇点”的情况，在扩散模型里反复出现：$\tilde{\beta}_t$ 在 $t=1$ 处为 0、$\bar{\alpha}_t$ 在 $t=T$ 处接近 0、$1/\sqrt{\alpha_t}$ 在 t 大时发散。**每一个都需要在代码里单独处理，而它们都不会自己报错。**）

5.23.2 停下来想一想 ⑨

本章反复说“训练 loss 看不见采样的 bug”。那么反过来说：有没有什么 bug 是“采样看得见、训练看不见”的反面——即“训练能看见、采样看不见”？
（答案：**有，而且很常见。**
最典型的是“训练时 t 的采样范围正确，但采样时用的调度数组是另一个”。
比如训练用线性调度，采样循环里却读了余弦调度的 $\bar{\alpha}$ 数组（两个数组长度都是 1000、变量名都是 `alpha_bar`）。
这时：- **训练 loss 完全正常**（训练用的是正确那个数组）- **采样结果会崩**——不，等一下。
这个例子的方向反了。让我们把它拆清楚：

- 如果**训练和采样读的是同一个数组**（都错或都对）→ **两边自洽**，训练 loss 正常，采样也正常。**这个 bug 完全不可见。**
- 如果**训练和采样读的是不同的数组** → 训练正常，**采样崩**。这仍然是“训练看不见、采样看得见”。

所以“训练看得见、采样看不见”需要另一个结构：采样路径上根本没用到的量，被训练用到了。
一个真实的例子：L_{simple} 里 t 的采样分布。如果你把 $t\sim U\{0,T-1\}$ 改成 $t\sim U\{0,T/2-1\}$（只训一半），**训练 loss 会明显变化**（因为平均的区间变了），**但采样仍然是走完整的 1000 步**——它会经过那些**从没被训练过的** t。
这个 bug 在训练曲线上一眼可见（loss 会异常），在采样的统计量上却不一定明显——因为模型在那 500 步上的输出是外推的，**可能“看起来还行”，只是细节不对。**
§5.13 失败二正是这个结构：那里是“训练的 t 少了一段”，实测 MMD 差了 3.5 倍。**所以它是“两边都看得见，只是训练那边更容易发现”。**
真正“训练看得见、采样绝对看不见”的情况很少，因为训练和采样的交集（$\bar{\alpha}_t$ 和 ε_θ）正是模型能力的来源——**一旦它出问题，两边都会受影响。**
这个不对称值得记住：“训练看不见采样”是真的（本章三次实测）；“采样看不见训练”基本不成立。所以排查的方向应该是：训练 loss 正常时，去查采样代码；而不是反过来。）

5.23.3 停下来想一想 ⑩

本章最后留一个问题。
§5.6 测出 $t\in[2,11]$ 这 10 步贡献了 49% 的 ELBO，而 DDPM 把这些权重全扔了，结果反而更好。
如果这个“反而更好”是真的，那么下面哪一件事必然成立？

A. 那 49% 的 ELBO 是没有意义的 B. 那 10 步的模型输出质量其实不差 C. ELBO 这个下界本身有问题 D. $t \in [2, 11]$ 这 10 步的梯度方向特别嘈杂

（答案：**D 是最可能的，而 A、B、C 都不必然。**

先把逻辑理清楚。

扔掉权重之后，$t \in [2, 11]$ 的梯度贡献从 49% 掉到 1%。 如果去掉它们之后结果变好，那说明**这 49% 的贡献“净效果是负的”**。

“净效果为负”意味着什么？ 两种可能：

- **梯度方向嘈杂**（D）：每一次更新都朝一个大致随机、但幅度很大的方向走。**放大的权重把噪声也放大了。**
- **梯度方向系统性地错**：那才是 A（这个目标没意义）。但 ELBO 是从 Jensen 不等式严格推出来的，**它的方向不会系统性错误**，只可能是“方差大”。

B 为什么不必然？ 即使那 10 步的模型输出不差，**也不妨碍它的梯度贡献是负的**——梯度关心的是“往哪改”，不是“现在有多好”。

C 为什么不必然？ ELBO 作为下界是数学事实。**它的问题可能是“作为训练目标的方差太大”，而不是“下界不对”。** 这两个是完全不同的问题。

所以正确答案是 D，而且它是可测的—— 记录那 10 步的梯度与最终下降方向的余弦相似度，**和 t 大时的做对比。**

这个实验本章没做，它是第 6 章的入口之一。）

5.24 自测

十道题。前六道能在本章找到直接答案，后四道需要自己推或算。

1. 前向过程为什么能“一步加噪到任意 t”？这个性质依赖哪一条假设？把这条假设去掉会怎样？

2. 后验 $q(x_{t-1} \mid x_t, x_0)$ 的方差 $\tilde{\beta}_t$ 在 $t = 1$ 处等于 0。**为什么这个 0 是“对的”而不是“数值问题”？**

3. 反向过程 $p_\theta(x_{t-1} \mid x_t)$ 是一个高斯分布。**如果把它的方差设成 0，还叫扩散模型吗？** 和“取均值”有什么区别？

4. 本章的案例里，把 $\sigma_t z$ 去掉之后，采样塌了 130 倍，但训练 loss 一位小数都不变。**用一句话说清为什么。**

5. 实测 x_0-预测的 MMD 是 ε-预测的 12 倍。**这个对照有一处不公平，是什么？**

6. 本章测出 $T = 25$ 比 $T = 1000$ 好（MMD 0.00485 vs 0.01146）。**这个结论能迁移到图像模型上吗？为什么？**

7.（推导）证明 $q(x_t \mid x_{t-1})$ 不依赖 x_0，即 $q(x_t \mid x_{t-1}, x_0) = q(x_t \mid x_{t-1})$。这条性质在 ELBO 的哪一步用到了？

8.（推导）从 (12) 出发，推出 (13)：

$$L_{t-1} = \frac{\beta_t^2}{2\sigma_t^2 \alpha_t (1 - \bar{\alpha}_t)} \mathbb{E}\left[\| \varepsilon - \varepsilon_\theta \|^2 \right] + \text{const} \tag{5.103}$$

9.（计算）本章的调度下，$\bar{\alpha}_{500} = 0.078587$。求 $t = 500$ 时的 SNR、$\sqrt{\bar{\alpha}_{500}}$、$\sqrt{1 - \bar{\alpha}_{500}}$，并判断此时“信号比噪声大还是小”。

10.（设计）你要给一个已有的扩散模型**换一个更快的采样器**，但**不能重新训练**。你会先检查哪三件事？为什么是这三件？

11.（辨析）有人说：“扩散模型的反向过程就是前向过程的逆过程，所以两者是一一对应的。”**这句话哪里不对？**

12.（计算）某模型的 $\bar{\alpha}_t$ 在 $t = 100$ 处是 0.8970。**求此时 x_t 里噪声的方差占比，以及“信号系数是噪声系数的几倍”。**

5.25 自测答案

1. 依赖的假设是**马尔可夫性**：x_t 只依赖 x_{t-1}，不依赖更早的历史。有了它，t 步的联合分布才能写成 $\prod_s q(x_s \mid x_{s-1})$，把 x_0 反复代入之后，所有 ε 项都是独立正态的线性组合，**方差等于系数平方和**，于是合成一个单独的 $\bar{\varepsilon}$。

去掉这条假设会怎样？比如允许 $q(x_t \mid x_{t-1}, x_{t-2})$，那么 x_t 的分布里会多出二阶项，$\bar\alpha_t$ 这种“一次乘积”的写法就不成立了—— **要维护一个随 t 增长的统计量（比如二阶矩的递推），闭式解不再是一行**。

这也是为什么本章 (10) 那一步改写能成立：$q(x_{1:T} \mid x_0)$ 可以从“前向链”改写成“后验链”，靠的正是马尔可夫性。

2. $\tilde\beta_1 = 0$ 是对的，因为**在 $t = 1$ 处，x_1 里几乎全是 x_0 的信息**。

具体算：$x_1 = \sqrt{\bar\alpha_1} x_0 + \sqrt{1-\bar\alpha_1}\varepsilon_1$，而 $\sqrt{\bar\alpha_1} = 0.99995$、$\sqrt{1-\bar\alpha_1} = 0.01$。**给定 x_1 和 x_0，ε_1 是被唯一确定的**—— 于是 $x_0 = (x_1 - \sqrt{1-\bar\alpha_1}\varepsilon_1)/\sqrt{\bar\alpha_1}$ 是一个**确定性的等式**，没有自由度。

“没有自由度”就是方差为 0。

反过来看 t 很大的情况：$\bar\alpha_t \approx 0$ 时 x_t 里 x_0 的信息几乎为零，**给定 x_t，x_{t-1} 的不确定性最大**（等于 β_t，即前向过程每一步注入的噪声量）。**从 0 平滑增长到 β_t**，这个形状本身就是“信息在逐步流失”的量化。

3. 严格说，**方差为 0 的“高斯”是狄拉克 δ 函数，不是一个概率分布**（它在数学上是一个测度）。所以把方差设成 0 之后，$p_\theta(x_{t-1} \mid x_t)$ **不再是一个合法的条件分布**，(12) 里的 $\log(\sigma_t^2/\tilde\beta_t)$ 和 $\tilde\beta_t/\sigma_t^2$ 都会发散。

而这正是“取均值”和“方差为 0”的区别所在：

- **“取均值”是一种采样方式**：分布仍然是高斯，只是我们每次都从它的中心取点。**训练完全正常，ELBO 完全正常**。
- **“方差为 0”是改了模型的定义**：ELBO 本身就不成立了，训练会立刻出问题（loss 变 nan 或无穷）。

本章案例的 bug 是前者——分布定义得好好的，采样时只取了中心。**这就是它能藏住的原因**：所有数学都没错，只有采样动作漏了一步。

4. 因为**训练走的是前向过程（用 $\bar\alpha_t$ 造 x_t），采样走的是反向过程（用 μ_θ 和 σ_t 递推），两段代码除了 $\bar\alpha_t$ 和 ε_θ 之外没有任何交集**—— σ_t 和反向均值的公式在训练里一次都没出现。

（如果只能写一句：**loss 度量的不是“采样对不对”，而是“ε 预测得准不准”。这两件事之间没有必然联系**。）

5. 两者的训练步数都是 5000 步，但两个 loss 的梯度尺度差 2.5 个数量级。

按 (17)，x_0-预测等价于给 ε 损失乘 $1/\mathrm{SNR}_t$，而这个因子在 t 上的变化范围是 ± 2.5 个数量级。**同样的学习率作用在尺度相差这么大的梯度上，不可能同时对两端都好**——很可能 t 大的一端学得过度、t 小的一端学不动。

要让对照公平，至少得给 x_0-预测单独扫一遍学习率，或者像文献里那样配上 SNR_t 权重（那就等于 ε-预测了，见检查点 ⑤）。

这处不公平必须写出来，否则读者会以为“x_0-预测就是不行”，而文献里的结论是“x_0-预测能用，只是不省事”。

6. 不能。

§5.9 那个实验把训练预算固定成 5000 步，**而 T 越大，每个噪声等级分到的更新次数越少**：$T = 1000$ 时平均每个等级 5 次，$T = 25$ 时 200 次。

换句话说，那个实验测的其实是“在固定预算下，步数与训练充分度的权衡”，而不是“T 本身的影响”。

在图像模型上，这个权衡的答案很可能反过来：图像分布的复杂度远高于双月，**每个噪声等级都需要大量样本才能学好**，所以实践中 T 取 1000 甚至 4000。

正确的结论应该是：“步数越多越好”是没有根据的直觉；最优步数取决于训练预算和任务复杂度。本章的数据只能说明前半句。

7. 证：$q(x_t \mid x_{t-1}, x_0) = \frac{q(x_t, x_{t-1}, x_0)}{q(x_{t-1}, x_0)}$。

按前向链的马尔可夫性，$q(x_t \mid x_{t-1}, x_0) = q(x_t \mid x_{t-1})$—— **这是假设，不是推论**。

要“证明”它，得回到前向过程的构造：$x_t = \sqrt{\alpha_t} x_{t-1} + \sqrt{\beta_t}\varepsilon_{t-1}$，**右边只出现 x_{t-1} 和一个新抽的 ε_{t-1}，和 x_0 无关**。所以给定 x_{t-1}，x_t 的分布是 $\mathcal{N}(\sqrt{\alpha_t} x_{t-1}, \beta_t I)$，**$x_0$ 是无关的额外条件**。

在 ELBO 的哪一步用到？在 §5.6 分解 (9) 的时候：分子的 $\prod_t p_\theta(x_{t-1} \mid x_t)$ 和**分母里前向链的每一项 $q(x_t \mid x_{t-1})$**—— 正是因为后者不依赖 x_0，改写 (10) 之后**下标才能对齐**。**如果不成立，“每一项配成一个 KL”这件事就做不到了**。

8. 证：由 (5) 和 (7)，两个均值之差是

$$\tilde\mu_t - \mu_\theta = \frac{\sqrt{\alpha_t}(1-\bar\alpha_{t-1})}{1-\bar\alpha_t} x_t + \frac{\sqrt{\bar\alpha_{t-1}}\beta_t}{1-\bar\alpha_t} x_0 - \frac{1}{\sqrt{\alpha_t}} x_t + \frac{\beta_t}{\sqrt{\alpha_t}\sqrt{1-\bar\alpha_t}}\varepsilon_\theta \tag{5.104}$$

x_t 的两项系数之和为 0（这正是 §5.5 里算过的 $\frac{\sqrt{\alpha_t}(1-\bar\alpha_{t-1})}{1-\bar\alpha_t}+\frac{\beta_t}{\sqrt{\alpha_t}(1-\bar\alpha_t)}=\frac{1}{\sqrt{\alpha_t}}$，移项即得）。于是只剩 x_0 和 ε_θ 两项：

$$=\frac{\sqrt{\bar\alpha_{t-1}}\beta_t}{1-\bar\alpha_t}x_0+\frac{\beta_t}{\sqrt{\alpha_t}\sqrt{1-\bar\alpha_t}}\varepsilon_\theta \tag{5.105}$$

把 x_0 用闭式解换成 x_t 和 ε：$x_0=(x_t-\sqrt{1-\bar\alpha_t}\varepsilon)/\sqrt{\bar\alpha_t}$，代进去之后 x_t 的系数是 $\frac{\beta_t}{1-\bar\alpha_t}-\frac{\beta_t}{\sqrt{\alpha_t}\sqrt{1-\bar\alpha_t}}$ ——**不为 0**。

正确的做法是不要展开 x_0，而是直接用 (16) 的逆关系。把 $x_t=\sqrt{\bar\alpha_t}x_0+\sqrt{1-\bar\alpha_t}\varepsilon$ 代进第一项，再用 $\bar\alpha_{t-1}\bar\alpha_t^{-1}=\alpha_t^{-1}$ 化简，得到

$$\tilde\mu_t-\mu_\theta=\frac{\beta_t}{\sqrt{\alpha_t}\sqrt{1-\bar\alpha_t}}(\varepsilon_\theta-\varepsilon) \tag{5.106}$$

这就是要的结果。代进 (12)，注意 x_t 项已经消掉、只剩 $\|\varepsilon_\theta-\varepsilon\|^2$ 乘一个标量系数，再加上三个常数项，即得 (13)。■

9. $\mathrm{SNR}=\frac{0.078587}{1-0.078587}=\frac{0.078587}{0.921413}=\mathbf{0.08529}$。

$\sqrt{\bar\alpha_{500}}=\sqrt{0.078587}=\mathbf{0.28033}$。

$\sqrt{1-\bar\alpha_{500}}=\sqrt{0.921413}=\mathbf{0.95990}$。

信号比噪声小得多：0.28 对 0.96，**噪声系数是信号系数的 3.4 倍**。按信噪比 0.085 说，**噪声能量是信号的 11.7 倍**。

这个位置是“前向过程走了一半多”—— $\bar\alpha$ 从 1 掉到 0.079，跨过了 92% 的区间。

10. 三件事：

① **采样器假定的“模型输出的是什么”**。DDIM 和 DPM-Solver 的公式都是针对 ε-预测写的。如果你的模型是 v-预测或 x_0-预测，**必须先按 (19) 换算**，否则公式套不上。

② **采样器假定的“时间步网格”**。DDIM 允许跳过中间步，但它假定的 $\bar\alpha$ 是**连续函数**还是**离散数组**？本章的实现用的是离散数组 + 最近邻取整（§5.9 的 `ddim_sample`），**这在 T 大、跳步少的时候没问题，跳步多的时候会引入额外误差**。

④ **采样器是否需要额外的假设**。比如 DDIM 的 $\eta=0$ 是确定性的，**而 §5.9 实测：本章这个玩具模型上，确定性采样不稳定**—— 推理时的 ε_θ 只要有一点系统性偏差，1000 步连乘就会放大（实测溢出到 10^7 量级）。

为什么是这三件？因为换采样器**改变的是“怎么走”，但前向过程、训练目标、模型输出都不变**。所以所有可能出问题的地方，都在这三者的**接口**上：输出格式、时间网格、以及新采样器自己的隐含假设。

---

11. “逆过程”暗示了一个一一对应的映射，而反向过程是一个分布。

前向 $x_{t-1}\to x_t$ 是“确定性缩放 + 加噪声”。**给定 x_t，x_{t-1} 有无穷多种可能**—— $q(x_{t-1}\mid x_t)$ 是一个分布，不是一个点。

要“逆”得唯一，必须额外知道 x_0（那就是后验 $q(x_{t-1}\mid x_t,x_0)$）。**而采样时 x_0 是未知的**——这正是模型要猜的东西（式 6）。

所以正确的说法是：反向过程学的是**后验分布的参数**，而不是“前向过程的逆函数”。

这个区别的实际后果：如果反向过程真是逆函数，采样就该是“算出来”的；**而它其实是“从分布里抽出来”的—— 这就是案例里那个“丢掉 $\sigma_t z$”的 bug 的全部背景**。

12. 按定义：

$$\text{噪声方差占比}=1-\bar\alpha_{100}=1-0.8970=\mathbf{0.1030}\ (10.3\%) \tag{5.107}$$

$$\text{信号方差占比}=\bar\alpha_{100}=0.8970\ (89.7\%) \tag{5.108}$$

但问的是“系数”的比，所以要开方：

$$\frac{\sqrt{\bar\alpha_{100}}}{\sqrt{1-\bar\alpha_{100}}}=\frac{0.9471}{0.3210}=\mathbf{2.95} \tag{5.109}$$

信号系数是噪声系数的 2.95 倍。

这道题的重点在最后一步：“方差占比 89.7%”和“系数是 2.95 倍”是同一个事实的两种说法，但它们给人的直觉完全不同。

看到“89.7% 是信号”，容易觉得“还很干净”；看到“系数只差 2.95 倍”，就会意识到**噪声的幅度已经不小了**。

而模型面对的是幅度，不是方差。

一个推论：$\bar{\alpha}_t = 0.5$ 时两个系数相等（都是 0.707），**而那时方差占比还有 50%。所以“信号占一半”对应的是“幅度上完全不占优势”**——这也是为什么 §5.8 用 SNR（方差比）描述噪声水平，**但真正决定模型难度的是它的平方根。**

5.26 ELBO 到底有多紧

§5.6 推 ELBO 的时候用的是 Jensen 不等式，那是一个**不等号**。这一节说清楚**这个不等号有多松**。

5.26.1 下界和真值差在哪

$$\log p_\theta(x_0) - \text{ELBO} = D_{\text{KL}}(q(x_{1:T} \mid x_0) \,\|\, p_\theta(x_{1:T} \mid x_0)) \tag{5.110}$$

这不是本章的结论，是变分推断的标准结果：Jensen 不等式的取等条件，就是“两个分布完全一样”。

所以“ELBO 有多紧”=“前向过程 q 和后验 p_θ 有多像”。

两个极端：

- **模型完美**：$p_\theta(x_{1:T} \mid x_0) = q(x_{1:T} \mid x_0)$，**下界取等**
- **模型完全没学**：p_θ 和 q 差得远，**下界很松**

注意 $q(x_{1:T} \mid x_0)$ 里含 x_0——也就是说，**“下界有多紧”本身依赖于具体的那个数据点**。对不同样本，紧的程度不一样。

5.26.2 实测：本章的模型有多紧

§5.6 实测的三项给出上界 13.2567（$L_T + \sum L_{t-1} - L_0$）。

但我们没有算 $-\log p_\theta(x_0)$ 本身，所以**不知道下界离真值多远**。

为什么不算？ 因为 $p_\theta(x_0) = \int p_\theta(x_{0:T}) dx_{1:T}$ **积不出来**——这正是 §5.6 开头那个问题。

所以“下界有多紧”这件事，本章无法直接测量。

> ⚠ 这是一个必须承认的空白

本章测了上界，没测真值，所以不知道紧不紧。

能不能间接估计？ 有几个办法，本章都没做：

① **用 importance sampling 估 $p_\theta(x_0)$**。从 $q(x_{1:T}|x_0)$ 采样若干条路径，算 $\frac{1}{K}\sum_k \frac{p_\theta(x_{0:T}^{(k)})}{q(x_{1:T}^{(k)}|x_0)}$。**当 K 很大时它收敛到真值**——但方差可能极大，**这是扩散模型做似然评估的经典困难。**

② **换一个指标**。很多论文报的是“上界的值”（叫 NLL bound），**并明确说明它不是似然。本章的 13.2567 属于这一类。**

④ **在小规模上暴力算**。本章 $d = 2$、$T = 1000$，积分是 2000 维——**数值积分依然不可行**，但用 ① 的 importance sampling 在小维度上方差会小得多。

这个实验是可行的，本章没做。它属于 §5.22 那三组“可以做但没做”之外的一组，而且它比其他三组更基础——**它决定本章那个 13.2567 该怎么解读。**

5.26.3 一个相关的实践问题

实践中几乎没有人用 ELBO 来评估扩散模型。

为什么？ 两个原因：

① **似然和感知质量不相关**。在图像上，**FID（一个基于特征距离的指标）和 NLL 经常给出相反的排序**。一个模型可以有更好的似然但更差的视觉效果——因为似然对“高维空间里的小误差”非常敏感，而人眼对低频结构更敏感。

② **ELBO 很难算准**。上面那个 importance sampling 的方差问题，让“报一个可信的 NLL”变成一件需要专门技术的工作。

扩散模型领域因此形成了一个惯例：- **训练**用 $\mathcal{L}_{\text{simple}}$（最简单的那个）- **评估**用 FID / IS 这类采样质量指标 - **似然**只在专门的论文里报

本章之所以还能用 ELBO 讲清楚训练目标，是因为它在这里是“来路”而不是“指标”——它解释了 $\mathcal{L}_{\text{simple}}$ 是从哪来的（§5.6），以及**为什么它和真实似然有差距**。

5.27 把 ELBO 的三项真的算一遍

§5.6 给了三项的**实测值**（$L_T = 0$、$\sum L_{t-1} = 8.1222$、$L_0 = -5.1345$）。这一节把其中**一项完整地手算出来**——不为别的，**只为了让你在读到“L_{t-1} 是 KL 散度”这句话时，脑子里有一个具体的数**。

5.27.1 选一个最容易手算的时刻

$t = 2$。此时涉及的量：

量	值	来源
$\bar\alpha_1$	0.999900	$\bar\alpha_1 = 1 - \beta_1$
$\bar\alpha_2$	0.999780092	$\bar\alpha_1(1-\beta_2)$
β_2	0.000119920	$10^{-4} + (0.02 - 10^{-4})/999$
$\tilde\beta_2$	5.453188×10^{-5}	式 式 (5.31)
σ_2^2	5.453188×10^{-5}	取 $\tilde\beta_2$

先设一个具体的数据点：$x_0 = (0.5, -0.3)$（在双月的一个弯月上）。

抽到的噪声：$\varepsilon = (1.2, -0.8)$。

5.27.2 第一步：造出 x_2

$$x_2 = \sqrt{0.999780} \times (0.5, -0.3) + \sqrt{0.000220} \times (1.2, -0.8) \tag{5.111}$$

$\sqrt{0.999780092} = 0.99989004$，$\sqrt{0.000219908} = 0.01482929$。

$$x_2 = (0.49994502, -0.29996701) + (0.01779515, -0.01186343) = (0.51774017, -0.31183044) \tag{5.112}$$

注意 x_2 和 x_0 差多少：$(0.01774, -0.01183)$，**各个分量的差都在 0.02 以内**。

这是因为 $t = 2$ 处 $\bar\alpha_2 = 0.99978$——信号几乎没被破坏。

5.27.3 第二步：算真实后验的均值和方差

方差（式 4）：

$$\tilde\beta_2 = \frac{1-\bar\alpha_1}{1-\bar\alpha_2}\beta_2 = \frac{9.999999 \times 10^{-5}}{2.199079 \times 10^{-4}} \times 1.199199 \times 10^{-4} = 5.453188 \times 10^{-5} \tag{5.113}$$

均值（式 5）：

$$\tilde\mu_2 = \underbrace{\frac{\sqrt{\alpha_2}(1-\bar\alpha_1)}{1-\bar\alpha_2}}_{c_1} x_2 + \underbrace{\frac{\sqrt{\bar\alpha_1}\beta_2}{1-\bar\alpha_2}}_{c_2} x_0 \tag{5.114}$$

两个系数：

$$c_1 = \frac{0.99989004 \times 9.999999 \times 10^{-5}}{2.199079 \times 10^{-4}} = 0.45470850, \qquad c_2 = \frac{0.99995000 \times 1.199199 \times 10^{-4}}{2.199079 \times 10^{-4}} = 0.54529150 \tag{5.115}$$

两个系数之和是 0.9999999985——在 $t = 2$ 处它就是 1。

注意：这个“和为 1”只在 t 小的时候成立。实测：

t	$c_1 + c_2$
2	1.000000
10	1.000000
100	0.999972
500	0.997177
1000	0.990077

t 越大，和偏离 1 越多。这不是数值误差（$t = 1000$ 处差了 1%），而是后验均值在**向原点收缩**：t 大时先验 $q(x_{t-1} \mid x_0)$ 的均值是 $\sqrt{\bar{\alpha}_{t-1}}x_0$，而 $\sqrt{\bar{\alpha}_{t-1}}$ 本身在趋近 0（$t = 1000$ 时是 0.0064）。

换句话说：t 大时后验均值不是x_t 和 x_0 的凸组合，**而是被拉向原点的一个加权和**——因为那个时刻的“先验”本身就在原点附近。

t 小时 c_2（x_0 的权重）更大，后验更相信 x_0；**t 大时反过来**。这个方向是符合直觉的。

$$c_1 x_2 = (0.235421, -0.141792), \qquad c_2 x_0 = (0.272646, -0.163587) \tag{5.116}$$

$$\tilde{\mu}_2 = (0.508067, -0.305379) \tag{5.117}$$

它落在 $x_0 = (0.5, -0.3)$ 和 $x_2 = (0.5177, -0.3118)$ 之间，距离 x_0 更近（因为 $c_2 > c_1$）。**这是对的**：t 小时，后验应该更相信 x_0 一侧的先验。

5.27.4 第三步：算模型的输出

假设模型是**完美的**，即 $\varepsilon_\theta(x_2, 2) = \varepsilon = (1.2, -0.8)$。

那么按式 式 (5.51)：

$$\mu_\theta = \frac{1}{\sqrt{0.999940}}\left[x_2 - \frac{0.000120}{\sqrt{0.000220}}(1.2, -0.8)\right] \tag{5.118}$$

$\sqrt{0.99988008} = 0.99994004$，$\frac{0.000119920}{0.01482929} = 0.00808668$。

$$\begin{aligned}\mu_\theta &= \frac{1}{0.99994004}[(0.51774017, -0.31183044) - (0.00970402, -0.00646935)] \\ &= \frac{(0.50803615, -0.30536109)}{0.99994004} = (0.50806662, -0.30537940)\end{aligned} \tag{5.119}$$

和 $\tilde{\mu}_2 = (0.50806659, -0.30537939)$ 一致到小数点后第七位（$\Delta = (0, 3 \times 10^{-8})$）。

这是应该的——模型完美时 $\mu_\theta = \tilde{\mu}_t$，而这个等式正是 §5.5 那个推导的终点。

5.27.5 第四步：算 L_1（也就是 L_{t-1} 在 $t = 2$ 处的值）

代入 (12)，$\sigma_2^2 = \tilde{\beta}_2$：

$$L_1 = \frac{1}{2}\left[\frac{2\tilde{\beta}_2}{\tilde{\beta}_2} + \frac{\|\tilde{\mu}_2 - \mu_\theta\|^2}{\tilde{\beta}_2} - 2 + 2\log\frac{\tilde{\beta}_2}{\tilde{\beta}_2}\right] = \frac{1}{2}\left[2 + \frac{\|\Delta\|^2}{\tilde{\beta}_2} - 2 + 0\right] = \frac{\|\Delta\|^2}{2\tilde{\beta}_2} \tag{5.120}$$

前三项正好抵消（这是取 $\sigma_t^2 = \tilde{\beta}_t$ 的直接好处）。

$$\|\Delta\|^2 = 8.9 \times 10^{-16} \tag{5.121}$$

$$L_1 = \frac{8.9 \times 10^{-16}}{2 \times 5.453188 \times 10^{-5}} = 8.1 \times 10^{-12} \tag{5.122}$$

一个完美模型的 L_1 是 8.1×10^{-12}——在双精度下就是 0。

（这个残差本身就来自浮点舍入，不是算法的误差。）

而实测的 L_1 是 1.2143。

!这个 1.2143 是怎么来的

手算用的是“完美模型”，实测用的是“训了 5000 步的小模型”。
两者差的不是一点点：

	$\|\Delta\|^2$	L_1
完美模型	8.9×10^{-16}	8.1×10^{-12}
实测模型	$\approx 1.3\times10^{-4}$	**1.2143**

$\|\Delta\|^2$ 差了 11 个数量级。
为什么一个 5 万参数的模型在 $t=2$ 处差这么多？
因为 $\tilde\beta_2 = 5.45\times10^{-5}$ **在分母上。 要把 L_1 压到 1 以下，模型必须让 $\|\Delta\|^2 < 1.1\times10^{-4}$，也就是 Δ 的每个分量小于 0.007。**
而 x_2 和 x_0 的距离本身只有 0.02—— 模型要让后验均值的误差小于信号本身的 35%，才算“及格”。
这不是模型不好，是这个时刻的惩罚确实很严。
它同时解释了 §5.6 那张分箱表： $t\in[2,11]$ 这 10 步贡献了 49% 的 ELBO， **不是因为那几步“更重要”，而是因为那里的分母 $\tilde\beta_t$ 最小， 任何误差都被放大得最厉害。**

5.27.6 第五步：换个 t 再算一遍，看差距

把同样的手算在 $t=500$ 上做一遍（数据点和噪声换一组）：

量	$t=2$	$t=500$
$\bar\alpha_t$	0.999780	0.078587
$\sqrt{\bar\alpha_t}$	0.999890	0.280333
$\sqrt{1-\bar\alpha_t}$	0.014832	0.959903
$\tilde\beta_t$	5.45×10^{-5}	1.0031×10^{-2}
$\tilde\beta_t$ 的比值	—	**大了 184 倍**

$\tilde\beta_{500}$ 是 $\tilde\beta_2$ 的 184 倍。
这意味着同样大小的 $\|\Delta\|^2$，在 $t=500$ 处产生的 L 只有 $t=2$ 处的 1/184。
这就是 §5.6 那张表（小 t 贡献 49%）的全部算术根源。
所以这一节在说什么
ELBO 的“重要性分布”不是一个设计选择，它是一个除法。
每个时间步的惩罚是 $\frac{\|\Delta\|^2}{2\tilde\beta_t}$， 而 $\tilde\beta_t$ 在 $t=2$ 处是 5.5×10^{-5}、在 $t=500$ 处是 10^{-2}。
分母差 184 倍，惩罚就差 184 倍。
这个除法有没有道理？ 有——$\tilde\beta_t$ 就是“这一步还剩多少不确定性”。**不确定性小的时候犯错，确实更应该罚。**
但这也意味着：训练一开始，梯度几乎全部来自那十几个小 t 的时刻， 而它们对应的 $\tilde\beta_t$ 小到让梯度极不稳定。
DDPM 把权重扔掉，本质上就是把这把“放大镜”撤掉。 撤掉之后发生了什么，是第 6 章的第一个问题。

5.28 这一章的方法论

本章推了六节数学，但**真正的收获可能不在公式里**。这一节把方法论单独抽出来。

5.28.1 一、“看起来正常”是最危险的信号

本章出现了四次“loss 正常但结果不对”：

情况	loss 的表现
案例：丢掉 $\sigma_t z$	**逐点完全相同**
失败二：漏掉一段 t	反而**下降到一半**
失败五：x_0-预测	不可比，但曲线形状完全正常
失败三：fp16 抵消	**训练根本不受影响**（前向过程的问题）

每一次的 loss 曲线都是“健康”的——单调下降、无震荡、无 nan。

教训：监控指标只有在它能反映你要的东西时才有用。 loss 反映的是“ε 预测得准不准”，**它和“采样对不对”之间没有必然联系。**

做法：给每一个你关心的性质**单独找一个可测量的代理。**采样结果对不对 → 就测采样的统计量（均值/方差/rms/MMD），**不要指望训练曲线替你说。**

5.28.2 二、“改量”和“改有无”要分开

§5.2 的三条弯路全部是“改量”： 模型更大、训练更久、σ 换一个来源。

而元凶是“有无”：那一项根本不存在。

这两类错误的排查方式完全不同：

- **改量：**在你的结果附近滑动，**没有跳变**
- **改有无：**可以让结果**跳到另一个量级**

排查顺序应该是先查“有无”：把公式和代码**逐项对账**（数项数），而不是逐个检查每个量的数值合不合理。

本章的排查之所以找到，就是因为最后换成了对账。

5.28.3 三、平均值掩盖异常

这一条在本章出现了两次：

- **按 t 分段的 loss**（§5.15 第一步）：总 loss 正常，但分数的形状能看出问题
- **ELBO 的逐时间步分解**（§5.6）：平均值 8.12 掩盖了“1% 的步贡献 49%”

这和第四章“逐位置看 loss”是同一个动作。

通用的做法：任何你在平均值上做的判断，都值得按某个维度分组再看一次。分组维度选“最可能不均”的那个—— 第 4 章是位置，本章是时间步。

5.28.4 四、对照要“同结构”

§5.2 的弯路一（加宽模型）就是一个失败的对照：我们改了模型容量，**但同时改了参数量**，于是分不清“是容量起作用”还是“是参数起作用”。

正确的对照要求只动一个变量，而**在神经网络上这几乎不可能**（改宽度就改参数量，改深度就改感受野）。

实际的做法是承认这一点，并把混淆写出来。本章 §5.9 那个 T 的对照就是这么处理的：明确写了“训练预算固定，对大 T 不利”。

5.28.5 五、写清楚“哪一条结论最弱”

本章末尾那张分界表（§5.19）做了这件事：把结论分成“从定义推出来的”（可迁移）和“实测出来的”（依赖规模）。

这一步看起来是自我削弱，实际不是。

- **对读者：**他知道了该信到什么程度，**不会拿一个 2D 玩具的结论去指导大模型训练**
- **对自己：**写这张表的过程会逼你发现**哪些结论其实没有支撑**

本章真的有两条被这张表逼出来的：

1. **$T = 25$ 比 $T = 1000$ 好**——写进分界表时才发现， 这个结论几乎完全由“训练预算固定”这个设定决定
2. **$\beta_{\max} = 0.002$ 测不出危害**——写进分界表时才发现， 我们其实没有测出这个缺陷的任何实际后果

这两条如果混在正文里当结论卖，都是不诚实的。

5.28.6 一张可以带走的清单

#	规矩	本章的正例	本章的反例
1	每个指标只对它测的东西负责	§5.15 的六项自检	用 loss 判断采样质量
2	先查“有无”再查“改量”	§5.15 第四步的逐项对账	§5.2 的三条弯路
3	平均值必须分组复看	§5.6 的逐时间步分解	只看总 loss
4	对照要承认混淆	§5.9 写明训练预算不利	§5.2 弯路一的加宽
5	把最弱的结论写出来	§5.19 的分界表	把玩具结论当通用规律

这五条和前三章的方法是同一套—— 第 1 章的“唯一变量”、第 2 章的“公平对照”、第 3 章的“比 logits 不比文本”、 第 4 章的“逐位置分组”。 **换了一个领域，方法一条没变。**

5.29 关于记号：三个长得像的 α

扩散模型的记号是初学者最容易卡住的地方。 α_t、$\bar{\alpha}_t$、$\sqrt{\bar{\alpha}_t}$、 还有 $\tilde{\alpha}_t$（有些文献用）——**四个符号，长得都差不多。**

5.29.1 一张速查表

符号	读作	定义	本章 $t = 1$	本章 $t = 1000$
β_t	beta	第 t 步的噪声方差	0.0001	0.02
α_t	alpha	$1 - \beta_t$	0.9999	0.98
$\bar{\alpha}_t$	alpha bar	$\prod_{s \le t} \alpha_s$	0.9999	4.04×10^{-5}
$\sqrt{\bar{\alpha}_t}$	—	信号系数	0.99995	0.00636
$\sqrt{1 - \bar{\alpha}_t}$	—	噪声系数	0.0100	1.0000

看最后两列：α_t 从 0.9999 到 0.98（**变化 2%**），而 $\bar{\alpha}_t$ 从 0.9999 掉到 4×10^{-5}（**变化 24750 倍**）。
这两个量在 $t = 1$ 处几乎相等，在 $t = 1000$ 处差了 24000 倍。
所以“在 $t = 1$ 附近把两者写混了看不出来”—— 这正是 §5.13 失败三（fp16 抵消）发生在小 t 的另一层原因。

5.29.2 三个最常见的混淆

混淆一：$\sqrt{\bar{\alpha}_t}$ 与 $\bar{\alpha}_t$。
$\bar{\alpha}_t$ 是**方差**的保留比例，$\sqrt{\bar{\alpha}_t}$ 是**幅度**的系数。 $t = 1000$ 时前者是 4×10^{-5}，后者是 0.0064——**差 157 倍。**
写 $x_t = \bar{\alpha}_t x_0 + \cdots$ 是一个常见错误，后果是 t 大时信号衰减得过快（按平方而不是按线性）。**检测方法**：算 x_t 的方差（§5.15 检查 ④）， **如果形状不对，就是这个错。**

混淆二：α_t 与 $\bar{\alpha}_t$。
反向均值 (7) 里两个都出现了：

$$\mu_\theta = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon_\theta\right) \tag{5.123}$$

外面是单步的 α_t，里面是累积的 $\bar{\alpha}_t$。
§5.13 失败六给了后果：写反的话，$t = 1000$ 处的系数从 1.010 变成 157。
混淆三：$\bar{\alpha}_t$ 与 $1-\bar{\alpha}_t$。
这两个在 $t = 500$ 附近交叉（$\bar{\alpha}_{500} = 0.0786$）。
在 t 小时 $\bar{\alpha}_t$ 接近 1、$1-\bar{\alpha}_t$ 接近 0；在 t 大时反过来。**而它们在公式里的位置完全不同**：一个在信号项、一个在噪声项、还有一个在分母上。
一个自查的办法：在代码里给这两个量起明确的名字。

```
signal_coef = alpha_bar.sqrt()          # 信号系数
noise_coef  = (1 - alpha_bar).sqrt()    # 噪声系数
# 而不是 a = alpha_bar.sqrt(); b = (1-alpha_bar).sqrt()
```

本章的 `tiny_diffusion.py` 就是这么命名的。多打几个字符，能省掉一小时的调试。

5.29.3 一份“看到某个符号时该想到什么”的对照

看到	想到
β_t	这一步注入多少噪声；能不能在 fp16 下算
α_t	单步保留率；反向均值外面的那个因子
$\bar{\alpha}_t$	累积保留率；和 t 一起单调下降；**方差**
$\sqrt{\bar{\alpha}_t}$	信号**幅度**的系数；t 大时接近 0
$\sqrt{1-\bar{\alpha}_t}$	噪声**幅度**的系数；t 大时接近 1
$\tilde{\beta}_t$	后验方差；$t = 1$ 处精确为 0
σ_t	反向过程每步注入的噪声；取 $\sqrt{\tilde{\beta}_t}$ 或 $\sqrt{\beta_t}$
λ_t	ELBO 的权重；$t = 1$ 处发散

这张表建议抄在笔记本第一页。本章后面所有的推导，**出错的地方九成在这八个量里的某两个之间。**

5.30 关键概念表

符号	名称	定义	首次出现
β_t	前向方差	第 t 步注入的噪声量	式 式 (5.17)
α_t	单步保留率	$1-\beta_t$	式 式 (5.17)
$\bar{\alpha}_t$	累积保留率	$\prod_{s\le t}\alpha_s$	式 式 (5.21)
SNR_t	信噪比	$\bar{\alpha}_t/(1-\bar{\alpha}_t)$	§5.4
$q(x_t \mid x_0)$	前向闭式解	$\mathcal{N}(\sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t)I)$	式 式 (5.19)
$q(x_{t-1} \mid x_t, x_0)$	后验	$\mathcal{N}(\tilde{\mu}_t, \tilde{\beta}_t I)$	式 式 (5.31) 与式 (5.33)
$\tilde{\beta}_t$	后验方差	$\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$	式 式 (5.31)

符号	名称	定义	首次出现
$\tilde{\mu}_t$	后验均值	式 式 (5.33)	式 式 (5.33)
$p_\theta(x_{t-1} \mid x_t)$	反向过程	$\mathcal{N}(\mu_\theta, \sigma_t^2 I)$	式 式 (5.57)
σ_t	反向噪声标准差	$\sqrt{\tilde{\beta}_t}$ 或 $\sqrt{\beta_t}$	§ 5.5
ELBO	证据下界	式 式 (5.60)	式 式 (5.60)
L_T, L_{t-1}, L_0	三项分解	式 式 (5.64)	式 式 (5.64)
λ_t	ELBO 权重	式 式 (5.71)	式 式 (5.71)
$\mathcal{L}_{\text{simple}}$	简化目标	式 式 (5.72)	式 式 (5.72)
v	v-预测目标	式 式 (5.73)	式 式 (5.73)

符号约定提醒：本章的 $\bar{\alpha}_t$ 是**累积**量，α_t 是**单步**量。两者在 $t = 1000$ 处相差 10^4 倍（0.98 vs 4×10^{-5}）。**§ 5.13 失败六专门讲了这个混淆的后果。**

5.31 延伸阅读

5.31.1 ⑦ 关于“扩散模型为什么有效”的另一种解释

本章从“去噪”的角度讲扩散：模型学的是 ε_θ，反向过程是一步步去噪。
还有一条完全不同的路线：从“得分匹配”（score matching）出发。
得分（score）是 $\nabla_x \log p(x)$——对数密度的梯度。**它和 ε_θ 只差一个因子**（式 24）。
这条路线的好处是它把“生成”还原成了“爬山”：如果知道每个位置上的 score，就可以用 Langevin 动力学 从任意起点走到高密度区：

$$x \leftarrow x + \frac{\eta}{2} \nabla_x \log p(x) + \sqrt{\eta}\, z \tag{5.124}$$

这是一个“带噪声的梯度上升”。
扩散模型可以看成“退火版的 Langevin”：从高噪声（相当于高温，可以跨过势垒）逐步降到低噪声（低温，稳定在峰上）。**$\bar{\alpha}_t$ 就是那个“温度调度”。**
读的时候注意什么：

- **这条路线解释了本章 § 5.7 那个观察**（ε 损失比 x_0 损失的动态范围小 8500 倍）：**score 在不同噪声水平上的尺度本来就是稳定的**——因为它衡量的是“密度在局部怎么变化”，而不是“数据点在哪里”。
- **它也是 § 5.11 那个连续时间视角的自然延伸**：连续极限下，反向 SDE (23) 就是“带 score 修正的退火过程”。
- **但它有一个陷阱**：score 路线早期的论文（Song & Ermon 2019 那一系列）**用的是“多个噪声水平的去噪自编码器”，而不是单个条件模型。和 DDPM 的形式差别很大，对着读容易迷路。**第 6 章会把它讲清楚。

5.31.2 ③ 一份排查清单（工程向）

本章 § 5.15 给的是“数学对不对”的自检。如果要做工程实现，还需要一份“跑起来对不对”的清单。
推荐从这三个项目的文档里对照着看：

- **`denoising-diffusion-pytorch`**（lucidrains）：最简的 DDPM 实现，约 500 行。**适合对照本章的 5 个公式逐行核。**
- **`diffusers`**（HuggingFace）：工业级实现，**调度器被抽象成可插拔的模块**——这正好对应本章 § 5.8 那张调度谱系。
- **`k-diffusion`**（Katherine Crowson）：**采样器最全的一个**，DPM-Solver、Heun、各种二阶方法都在里面。**第 6 章会用到它。**

读这些实现的建议：

1. **先找调度器**（`scheduler` / `sigmas`），对照本章的 (2)(3)(4) 三式。**如果它的 $\bar{\alpha}$ 定义和本章一致，后面的就好读了**。
2. **再找训练循环里那三行**（抽 t、造 x_t、回归）。**这一步在任何一个实现里都是 3–5 行**。
3. **最后找采样循环。重点看它对 $t=0$ 那一支怎么处理**—— 不同的实现有不同的约定（有的最后一律不加噪声，有的按 $t>0$ 判断，有的把 σ_T 强制设 0）。**这是不同实现之间结果不一致的最常见来源**。

一条经验：两个实现给出的采样结果不同，先查它们的调度数组是否逐元素相同—— 十有八九问题在这里，而不是在模型或采样器上。

5.31.3 ① 扩散模型的原始论文（2020）

Ho, Jain, Abbeel, Denoising Diffusion Probabilistic Models。arXiv:2006.11239。

读它能得到什么：本章 §5.4–§5.6 的全部内容就是这篇论文的前三节。论文的推导比本章紧凑得多——**它把后验的配方过程压缩成了两行**（本章花了整整一节展开，因为那两行里省掉的代数正是读者最容易卡住的地方）。

读的时候注意什么：

- **论文的 L_0 定义和本章差一个负号**（§5.6 的符号约定）。对不上号时先查这一条。
- **它报告“去掉权重反而更好”（L_{simple}）**，但没有给出解释。第 6 章会处理这件事。
- **它的 Σ_θ 有两种选择**（β_t 和 $\tilde{\beta}_t$），论文说两者“结果相近”——本章 §5.5 把这个说法量化了（实测 MMD 差 1%）。

5.31.4 ② 余弦调度（2021）

Nichol & Dhariwal, Improved Denoising Diffusion Probabilistic Models。arXiv:2102.09672。

读它能得到什么：本章 §5.8 的余弦调度就出自这里。

它做的最重要的一件事，是把“调度好不好”变成了一个可计算的指标：论文用 log-SNR（他们叫 λ_t）刻画每个时间步的噪声水平，并指出**线性调度在低噪声端浪费了大量步数**。

本章把这件事量化了：线性调度有 **32.7%** 的步落在 $\bar{\alpha}_t < 0.01$，余弦只有 6.5%（§5.8 的表）。

读的时候注意什么：

- **它也讨论了“学 Σ_θ”**（让模型自己预测方差）。结论是：对**似然**有显著提升，对**生成质量**几乎没有影响。**这个区分很重要**——它说明“更好的似然”和“更好看的图”不是一回事。
- **它的改进（学方差 + 余弦调度）合起来才有效果**，单独用其中一个收益有限。**这类“两个改动一起用才生效”的结论，在小规模复现时极易误判**。

5.31.5 ④ 为什么 ε 和 v 比 x_0 好（2022）

Salimans & Ho, Progressive Distillation for Fast Sampling of Diffusion Models。arXiv:2202.00512。

读它能得到什么：本章 §5.7 的 v-预测（式 18）出自这篇论文的附录。

它的主结果是蒸馏（把 1000 步压到 4 步），但**附录里对三种参数化的分析是本章 §5.7 那套论证的原始出处**。

读的时候注意什么：

- **本章 (17) 那个恒等式（$\|\varepsilon-\varepsilon_\theta\|^2 = \mathrm{SNR}\cdot\|x_0-\hat{x}_0\|^2$）在论文里是从“信噪比加权”的角度讲的**，本章把它改成了“两种参数化的损失关系”。**两条路等价，视角不同**。
- **它的蒸馏需要重新训练**（学生模型要多训一轮），和 §5.9 说的“采样器压缩”不是一回事——**后者不改变模型，只改变推理路径**。第 6 章会讲后者。

5.31.6 ⑤ 一次完整的“公式与代码对不上”的复盘

Kingma, Salimans, Poole, Ho, Variational Diffusion Models。arXiv:2107.00630。

读它能得到什么：**这篇论文把扩散模型的整个框架写成了一以贯之的变分推断**，并且指出了 DDPM 原文里几处“做了但没明说”的近似。

读的时候注意什么：

- **它用连续时间（$t \in [0,1]$）重写了整个框架**，于是 $\bar{\alpha}_t$、β_t 这些离散量变成了连续函数，**很多“为什么是这个公式”的问题在这里有了统一的答案**（比如为什么线性调度不好：它对应 log-SNR 上的非均匀采样）。
- **它引入了“信噪比必须单调递减”这个约束**—— 这是本章没提的一条设计要求，**它排除了很多看起来合理的调度**。
- **这篇论文的记号体系和 DDPM 原文差别很大**，对着读会吃力。**建议先读完本章，再来读它**。

5.31.7 ⑥ 关于玩具模型的方法论

本章反复强调一件事：在 2D 上测出来的结论，未必能迁移到大模型上。
这在扩散模型研究里是一个真实存在的问题。一个具体的例子：**很多论文在 CIFAR-10（32×32）上验证的采样器改进，在 ImageNet 256×256 上收益会缩水甚至消失**。
读任何一篇扩散模型的实证论文时，问三个问题：

1. **它测的是什么分辨率、什么规模？**
2. **它的对照有没有把训练预算对齐？**（§5.9 那个实验就栽在这里）
3. **它报的是 FID 还是似然？** 这两者在扩散模型上经常给出相反的排序。

本章能提供的是“骨架”和“方法”—— 哪些式子是数学、哪些结论是观察（§5.19 那张分界表）。**把这张表带在身上去读论文，能省下不少弯路**。

5.32 如果重来一次

这一节按第 2 章的先例，列出“如果重新做一遍，会改哪三件事”。
它不是自我批评，是把“事后才知道的东西”写下来—— 读者做自己的项目时，可以少走这三段。

5.32.1 一、先写采样，再写训练

本章的顺序是：先训模型（§5.2），发现采样不对（§5.2），再回头推公式（§5.4–§5.7）。
这个顺序是被案例逼出来的，但它不是最好的顺序。
更好的做法是：在训练之前，**先用一个未训练的随机模型跑一遍完整的采样循环**，只看输出的统计量。
为什么这样能提前发现问题？因为随机模型的输出虽然没意义，**但它的尺度应该是合理的**—— ε_θ 输出接近 0（随机初始化），于是反向均值 $\mu_\theta \approx x_t/\sqrt{\alpha_t}$，**采样结果的 rms 应该在数据量级附近**。
如果去掉 $\sigma_t z$，随机模型的采样结果也会塌—— 这个检查不需要训练，几十秒就能做完。
代价：需要先写完整的采样代码（包括调度数组），**但那是无论如何都要写的**。

5.32.2 二、把“按 t 分段”做成默认

本章 §5.15 用“按 t 分段的 loss”排除了训练层的问题，§5.6 又用同一张表发现了 ELBO 权重的不均匀。
这个动作很简单（把 $[0,T)$ 分成 5 段，各段单独平均），**但它是本章信息量最大的一个测量**。
如果重来，我会把它写进训练循环的默认输出—— 而不是等到排查的时候才临时加。
具体做法：

```
# 每 100 步记一次，成本可以忽略
t_eval = torch.randint(0, T, (4096,))
x0_eval = data[torch.randint(N, (4096,))]
xt_eval, eps_eval = q_sample(x0_eval, t_eval, sch)
l = ((model(xt_eval, t_eval) - eps_eval) ** 2).mean(dim=-1)
seg = [l[(t_eval >= i*T//5) & (t_eval < (i+1)*T//5)].mean() for i in range(5)]
```

五行代码，换来一个“能看见局部”的窗口。

5.32.3 三、把 $\bar{\alpha}_t$ 的检查前置

§5.8 那个 `check_schedule` 函数（20 行）只用了几毫秒，但它能提前发现四类问题：L_T 不为 0、纯噪声步数过多、步间不均匀、跨度不合适。

本章是在研究调度对比的时候才写出它的——而它的第一个判据（L_T 接近 0）本来就可以在选调度之前用。

这三件事的共同点：都是“把检查提前”，而不是“更聪明地排查”。

5.32.4 补一句：本章的代码组织

本章的代码分成三层，每一层的目的不同。

文件	目的	特点
`tiny_diffusion.py`	做实验	有教学开关（`reverse_mean`、`posterior_var`），可复现各种 bug
`diffusion_67.py`	讲清楚“最少需要什么”	单文件 67 行，无开关
`run_ch5_*.py`	产出正文里的数字	每个脚本对应一组表格

为什么要有 `tiny_diffusion.py` 里那些开关？

因为“复现一个 bug”和“避免一个 bug”是两个不同的需求。 `reverse_mean="mul"` 这个开关在正常使用时永远不该打开，**但它让“把 bug 加回去验证训练 loss 不变”这件事变成了改一个字符串。**

这是本书前三章反复用到的一个做法：把故意写错的版本也留在代码里，并且给它一个明确的开关。 第 1 章的 `scaled=False`、第 3 章的缓存变体都是同一个思路。

理由很实际：如果你为了演示一个错误而临时改代码，**改完之后很难保证改回去是对的；**而如果错误版本一直有开关、一直被测试覆盖，**它就成了一份可执行的文档。**

它们不需要更多的数学，只需要在动手之前多花十分钟写几个断言。而本章那个 bug 之所以能藏住，恰恰是因为没有任何一个断言在守着采样代码。

5.33 本章没做的事

每一章都有一批“本该做但没做”的事。把它们列出来，比假装完整要好。

5.33.1 一、没测似然，所以不知道 ELBO 有多紧

§5.6 给了上界 13.2567，但**没有算 $-\log p_\theta(x_0)$ 本身**，因此**不知道这个上界离真值多远**（§「ELBO 到底有多紧」）。

这一条是本章最大的空白，因为它影响 13.2567 这个数的解读方式。

要补它需要：importance sampling 估 $p_\theta(x_0)$，在小维度上（本章 $d = 2$）方差是可控的。**这是本章所有待办里最基础的一个。**

5.33.2 二、没做 v-预测的对照

§5.7 讲了 v-预测，**但没有在本章的模型上跑一遍。**

原因：v-预测的优势在大模型、高分辨率上才明显，**在本章的 2D 玩具上很可能测不出差别**——那样反而会给读者一个“v-预测没用”的错误印象。

代价是：§5.7 关于 v 的那一节**只有公式和文献引用，没有本章的实测。它属于 §5.19 分界表里“引用论文”的那一类，不是本章的结论。**

5.33.3 三、没做“λ_t 重要性采样”的对照

§5.20 提到的那个张力（重要性采样 t vs 扔掉权重）**本章只指出了它，没有测量它。**

这是第 6 章的核心问题之一，所以留在那里做更合适。

5.33.4 四、没测梯度方向的对齐程度

§5.6 和检查点 ⑩ 都指向同一个假说：**小 t 段的梯度方向更嘈杂**，所以削掉它的权重反而更好。

这是一个直接的、可测的假说（记录梯度与最终下降方向的余弦相似度），**本章没有做。**它是第 6 章的入口。

5.33.5 五、没做真实的图像实验

本章全程 2D。**§ 5.19 已经说明了哪些结论不能迁移**，但“不能迁移”这个判断本身**也是我们从原理上推的，不是测出来的**。

要验证它，需要在 CIFAR-10 上复现本章的几个关键对照（T 的影响、调度对比、参数化对比）。**那超出了“学生自备 CPU”这个约束**，所以放在附录 E 的延伸阅读里指路。

5.33.6 一张“本章结论的证据等级”表

按第 4 章的做法，把本章的结论按证据强度排一下：

结论	证据	强度
前向过程有闭式解	代数推导 + 数值验证（§ 5.4）	**强**
后验是高斯的、方差是 $\tilde{\beta}_t$	完整推导 + 两条独立代码路径对拍（§ 5.5、§ 5.15）	**强**
反向过程必须注入噪声	定义 + 实测（差 84 倍）	**强**
训练 loss 看不见采样 bug	**逐点相同的两条曲线**	**强**
$\| \varepsilon - \varepsilon_\theta \|^2 = \mathrm{SNR} \cdot \| x_0 - \hat{x}_0 \|^2$	代数 + 实测吻合到四位小数（§ 5.7）	**强**
fp16 下 $1 - \bar{\alpha}_t$ 被毁	实测（前 10 步误差 31%）	**强**
$t \in [2, 11]$ 贡献 49% 的 ELBO	实测（单次训练、64 个样本）	**中**（样本量小）
$T = 25$ 比 $T = 1000$ 好	实测，但训练预算未对齐	**弱**
$\beta_{\max}$ 太小有危害	指标报警，实测没测出危害	**无**（未证实）
v-预测更好	引用文献，非本章实测	**引用**

这张表里有两行值得特别注意：

- **“$T = 25$ 比 $T = 1000$ 好”标成了弱**—— 因为 § 5.9 已经说明了训练预算的问题
- **“$\beta_{\max}$ 太小有危害”标成了“无”—— 这不是说它无害，是说我们没测出来**

把这两行如实标出来，比把它们混在正文里当结论卖要好。

5.34 本章回顾

把这一章压缩成六句话。

一、前向过程的意义是“白送标签”。 $x_t = \sqrt{\bar{\alpha}_t} x_0 + \sqrt{1 - \bar{\alpha}_t}\varepsilon$ 这一步是**算出来的，不是收集来的**——任意 x_0、任意 t、任意 ε 都给出一条合法训练样本。**这是扩散模型能稳定训练的根本原因**：没有对抗、不需要判别器、loss 不会震荡。

二、后验是高斯的，方差 $\tilde{\beta}_t$ 有一个精确的解析式。这个式的两端分别是 0（$t = 1$）和 β_t（t 大），**中间平滑过渡**——它是“信息还剩多少”的量化。

三、反向过程定义的是一个分布，不是一个点。 $\sigma_t z$ 那一项是**整个过程的推进机制**。把它拿掉，1000 步的确定性映射会把先验压成一个点（实测半径 $1.26 \to 0.04$）。

四、训练和采样是两套公式，它们之间没有自动的一致性检查。这是本章最重要的一条。**案例里那个 bug 让采样塌了 130 倍，而训练 loss 逐点差 0.00e+00。**同类现象在本章出现了三次：案例、失败二（漏掉一段 t）、失败五（x_0-预测）——**三次都是“loss 变好但结果变坏”**。

五、ELBO 的权重在各时间步上极不均匀，而 DDPM 把它整个扔掉了。实测 $t \in [2, 11]$ 这 10 步（占 1%）贡献了 $\sum L_{t-1}$ 的 49%。**扔掉权重意味着小 t 的学习被大幅削弱—— 它为什么反而更好，是第 6 章的问题。**

六、尺度决定一切，而本章所有“更好/更差”的结论都是小尺度的。§ 5.19 那张分界表把本章的结论分成了两类：**从定义推出来的一类（可迁移），和实测出来的一类（只在 2D、5 万参数、5000 训练步这个设定下成立）**。

这一章真正要你记住的是

训练目标和采样过程是两套公式，它们之间没有任何自动的一致性检查。

这一章所有的具体内容——前向闭式解、后验、ELBO、参数化、调度——**都是围绕着这一句话展开的**：

- 前向过程给了训练数据（**白送的标签**）
- ELBO 给了训练目标（**一个回归损失**）
- 后验给了反向过程的形式（**一个分布，不是一个点**）
- 调度决定了每一步的工作量（**SNR 的形状**）

而这四件事全都在训练循环之外。 训练循环只做一件事：**给定 x_t，把 ε 猜准。**

所以模型训得好不好，和采样对不对，是两个独立的判断。本章用四次实测证明了这一点（案例、漏掉一段 t、x_0-预测、fp16 抵消）。

下篇后面三章会在这个骨架上加东西—— 更好的采样器（第 6 章）、条件控制（第 7 章）、文本注入（第 8 章）。**但“训练和采样是两套公式”这条不会变**，而且随着模型变复杂，**两者之间的缝隙只会更多，不会更少。**

最后留一个问题

本章测出：**$t \in [2, 11]$ 这 10 个时刻贡献了 $\sum L_{t-1}$ 的 49%。**

而 DDPM 的 $\mathcal{L}_{\text{simple}}$ 给每个 t 的权重都一样，也就是说，它把这 49% 削到了 1%。

削掉 98% 的“最重要的部分”，模型反而学得更好——这说不通。

除非……那 49% 的“重要”是有水分的。

一个可能的解释：λ_t 在小 t 处巨大（$\lambda_2 = 0.5997$，是 $t = 500$ 处的 109 倍），**而这个巨大的权重来自 $\tilde{\beta}_t$ 在分母上。** $t = 2$ 时 $\tilde{\beta}_2 = 0.000055$**——除以一个五万分之一的数，得到的权重当然大。**

但“权重在数学上大”不等于“梯度有用”。如果在小 t 处**梯度的方向本身就很嘈杂**（因为模型要预测的是一个几乎看不见的小量），那么放大它的权重只会放大噪声。

这个解释听起来合理，而且它是可测的：在训练过程中，**分别记录小 t 段和大 t 段梯度与最终下降方向的对齐程度。如果小 t 的梯度方向确实更嘈杂，这个假说就成立。**

本章没有做这个实验。它是第 6 章的第一个入口。

停下来想一想 ⑪

本章的那个 bug——“反向过程只取均值”—— 在扩散模型的文献里有一个“正经”的名字：确定性采样。

DDIM（第 6 章）就是确定性的（$\eta = 0$），而且它是一个被广泛使用、效果很好的方法。

那么本章案例里的那个写法，和 DDIM 到底差在哪？

（答案：**差在均值的公式上，而且差得很远。**

案例的写法（错误）：

$$x_{t-1} = \underbrace{\frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon_\theta\right)}_{\text{DDPM 的后验均值}} \tag{5.125}$$

DDIM（$\eta = 0$）：

$$x_{t-1} = \sqrt{\bar{\alpha}_{t-1}}\,\hat{x}_0 + \sqrt{1-\bar{\alpha}_{t-1}}\,\varepsilon_\theta, \qquad \hat{x}_0 = \frac{x_t - \sqrt{1-\bar{\alpha}_t}\varepsilon_\theta}{\sqrt{\bar{\alpha}_t}} \tag{5.126}$$

把 DDIM 的式子展开，用到 $\bar{\alpha}_{t-1}/\bar{\alpha}_t = 1/\alpha_t$：

$$x_{t-1} = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \sqrt{1-\bar{\alpha}_t}\,\varepsilon_\theta\right) + \sqrt{1-\bar{\alpha}_{t-1}}\,\varepsilon_\theta \tag{5.127}$$

对比两个式子：ε_θ 前面的系数不一样（DDPM 是 $\beta_t/\sqrt{1-\bar{\alpha}_t}$，DDIM 是 $\sqrt{1-\bar{\alpha}_t}$），而且 DDIM 多出最后那一整项。

数值上差多少？ $t = 500$ 处：$\beta/\sqrt{1-\bar{\alpha}} = 0.0105$，而 $\sqrt{1-\bar{\alpha}} = 0.960$——**差 92 倍。**

所以“把 DDPM 的噪声关掉”和 DDIM 是两回事—— 前者是把一个正确的公式砍掉一项，后者是用一个完全不同的公式。

这也解释了为什么案例里的写法会塌而 DDIM 不会：DDIM 的那个额外项 $+\sqrt{1-\bar{\alpha}_{t-1}}\varepsilon_\theta$ 恰好补偿了“少掉的随机性”在期望意义上的作用。

§5.9 的失败模式里提到过：本章的 DDIM 实现在这个玩具模型上不稳定（确定性采样会累积模型偏差），**所以本章没有把 DDIM 的实测数字放进来。它的完整处理留给第 6 章。**）

⚠ 一次失败的实验设计

在写 §5.8 的调度对比时，我们还试过一件事。它失败了，而且失败的方式值得记下来。

想法：既然线性调度有 32.7% 的步落在 $\bar{\alpha}_t < 0.01$ 的“纯噪声区”，**那就把那些步“压缩”掉**——保持 $T = 1000$ 和端点 $\bar{\alpha}_{1000}$ 不变，把 β 整体放大一个系数 k，让前向过程更快地穿过低 β 区。

做法：用二分法找 k，使 $\bar{\alpha}_{1000}$ 等于线性调度的 4.0358×10^{-5}。

结果是：二分收敛到 $k \approx 1.0$。

也就是说，“压缩后的调度”就是线性调度本身——因为“$\bar{\alpha}_{1000}$ 等于线性调度的值”这个约束，在线性调度那里本来就满足（那个目标值就是从它身上取的）。

实测两者的采样结果：

调度	训练 loss	rms	MMD
线性	0.1355	0.6090	0.01146
“压缩”（实际 $k \approx 1$）	0.1215	0.6477	**0.01136**

MMD 差 0.9%——两者在噪声量级内相同，符合“它们本来就是同一个调度”。

! 这个失败说明了什么

“删掉浪费的步”这个想法是对的，但它没有“调整 β 的尺度”这么简单的实现方式。

原因是：$\bar{\alpha}_t$ 的**形状**由 β 的**相对分布**决定，而不是由它的绝对大小决定。把 β 整体乘一个常数，只是把整条 $\bar{\alpha}$ 曲线在 t 轴上重新标定，**“多少比例的步落在 $\bar{\alpha} < 0.01$”这个比例几乎不变。**

要真的改变这个比例，必须改变 β 的分布形状——这正是余弦调度做的事（它直接定义 $\bar{\alpha}_t$ 而不是 β_t，§5.8）。

所以正确的对照不是“线性 vs 压缩后的线性”，而是“线性 vs 余弦”——那正是 §5.8 做的那张表。

这个失败被留在正文里，因为它是一个很自然的想法，而且它的实现方式看起来完全合理（调一个系数而已）。**读者如果自己试，很可能撞上同一件事。**

最后补一句：这个失败也说明了一件事——**本章对调度的那套分析（log-SNR 的形状、纯噪声步数的占比）本身是对的，但它们描述的是**形状，**不是**某个可以拧的旋钮。**要改形状，就得换一条曲线，而不是给同一条曲线乘一个数。**

这一条对后面几章也有用：第 6 章讲采样器时会反复出现「能不能少走几步」这个问题，而**答案从来不是「把步长调大一点」**——它要么换离散化方案（高阶求解器），要么换路径（流匹配），**总之都是换曲线，不是拧旋钮。**

把这一句和本章开头那句合起来看：训练看不见采样的 bug，是因为两边的公式没有交集；而“少走几步”做不到，是因为形状不是靠调系数能改的。**两句话说的是同一件事**——你得回到公式本身，而不是在参数上打转。**

下一章换个角度：本章从头到尾都在讲“加噪和去噪的数学”，**但一次采样到底要走多少步、每一步能不能走得更大胆，本章没有回答。**第 6 章处理这件事——**DDPM 的 1000 步能不能压到 50 步，代价是什么，以及那个“扔掉权重反而更好”的谜。**

6. 训练目标与采样器

本章地图

你在这里：第 5 章留下了一个悬案 —— 那个“训练完美、采样塌成一团”的模型，后来我们想用 DDIM 把它加速，结果**炸了**。本章处理的就是这类问题：训练目标怎么定、采样器怎么选。**本章解决**：DDIM 为什么能跳步；它为什么在有些模型上会爆；$\mathcal{L}_{\text{simple}}$ 扔掉 ELBO 权重为什么反而更好；“二阶求解器”到底是不是真的二阶。**前置**：第 5 章（前向闭式解、后验、ELBO、ε-预测）。**读完能做**：判断一个采样器在你自己的模型上能不能用；把 1000 步压到 20 步而不塌；识别“看起来是二阶、其实是恒等式”的实现。

本章路线：

节	回答的问题	用什么回答
§ 6.2 案例	为什么 DDIM 在我们模型上炸了？	一个换完美去噪器的对照
§ 6.3 DDIM	它凭什么跳步？	σ 族的构造 + 两行证明
§ 6.4 score	它和 score matching 什么关系？	Tweedie 公式 + 概率流 ODE
§ 6.5 高阶	二阶为什么更快？步长怎么取？	Heun + $\log\sigma$ 网格实测
§ 6.7 $\mathcal{L}_{\text{simple}}$	扔掉 ELBO 权重为什么更好？	梯度相干性实测
§ 6.9 η	随机性到底在干什么？	η 扫描 + SDE 族
§ 6.25 评估	怎么才不被数字骗？	三个陷阱 + 五步协议
§ 6.15 选型	我该用哪个？	决策树 + 三个场景

6.1 本章的代码与数据

```
import torch, sys
sys.path.insert(0, "notebooks")
from tiny_diffusion import Config, make_two_moons, make_schedule, train
from run_ch6_case import ddim, gaussian_eps_star

data = make_two_moons(4000, seed=0)
cfg  = Config(max_steps=5000)
model, sch, hist = train(data, cfg)          # 和第 5 章完全同一个模型

# 论文里的 DDIM（η=0，确定性）
x = torch.randn(2000, 2)
x = ddim(model, sch, x, steps=100)           # ← 这一行会给出 1e8 量级
```

复现本章全部数字：

```
cd notebooks
PY=/home/mpp/micromamba/envs/dlspace/bin/python
$PY run_ch6_case.py        # 案例主实验（约 2 分钟）
```

本章和上一章共用同一个模型、同一套调度、同一份数据。 这不是偷懒 —— **只有模型完全相同，“换采样器”才是一个干净的对照。**

本章用到的数据文件：

文件	内容	生成脚本
`data/ch6_case.json`	案例主实验：学出来的模型 vs 完美去噪器、偏差扫描、ε 偏差随 t	`run_ch6_case.py`
`data/ch6_extra.json`	梯度相干性、σ 空间 Euler/Heun 对比	`run_ch6_extra.py`
`data/ch6_eta.json`	η 扫描（50 步 / 100 步）	`run_ch6_eta.py`
`data/ch6_bias_grid.json`	偏差 × 步数曲面（5 × 6 格）	`run_ch6_bias_grid.py`
`data/ch6_grid.json`	σ 均匀 vs $\log\sigma$ 均匀	`run_ch6_grid.py`

每一个数字都来自这些文件的某一行。 如果你在正文里看到一个数，想找它的出处，**先在这五个文件里搜** —— 搜不到就是本书的错，请当作 bug 报告。

本章的两个交互实验：

实验	文件	它能回答什么
实验 6-1	`experiments/exp-6-1.html`	同 NFE 下三种采样器谁更准
实验 6-2	`experiments/exp-6-2.html`	均匀偏差为什么炸不了

注意实验 6-2 的结论和直觉相反（见 §6.8）：它不是“看误差怎么被放大”，而是“发现均匀误差根本不会被放大”。**这个负结果是本章最重要的一块。**

6.2 案例：论文里正确的 DDIM，在我们的模型上炸了

6.2.1 现象

第 5 章那个模型，采样一次要走 1000 步。太慢了。所以我们按论文实现了 **DDIM**——一个能把 1000 步压到 20 步的采样器。

第一版的运行结果：

步数	输出标准差	最大绝对值
1000	`nan`	`nan`
500	`nan`	`nan`
250	`nan`	`nan`
100	$[5.37 \times 10^{7}, 1.25 \times 10^{8}]$	4.97×10^{9}
50	[13.83, 32.10]	1.33×10^{3}
25	[0.5328, 0.3398]	3.52

（数据本身的标准差是 [0.5707, 0.3351]。）

三件事同时发生了：

1. **步数多的时候直接炸成 nan**——不是“质量差”，是数值溢出。
2. **步数少的时候反而正常**（25 步的结果看起来还挺像）。
3. **中间有一段（50 步）处于半炸状态**：std 13.8，比数据大 24 倍。

第 2 条最反直觉：所有人都在说“采样步数越多越好”，这里**步数越多越糟**。

6.2.2 弯路一：公式抄错了？

DDIM 的更新式是（符号含义见第 5 章）：

$$\begin{aligned}\hat{x}_0 &= \frac{x_t - \sqrt{1-\bar{\alpha}_t}\,\varepsilon_\theta(x_t,t)}{\sqrt{\bar{\alpha}_t}} \\ x_{t-1} &= \sqrt{\bar{\alpha}_{t-1}}\,\hat{x}_0 + \sqrt{1-\bar{\alpha}_{t-1}}\,\varepsilon_\theta(x_t,t)\end{aligned} \tag{6.1}$$

逐项对账：五个量（x_t、ε_θ、$\bar{\alpha}_t$、$\bar{\alpha}_{t-1}$、$\hat{x}_0$）和代码一一对上，没有多余项也没有缺项。

而且它和 DDPM 的均值是同一个东西 —— 把 $\hat{x}_0$ 代回去展开，得到的就是第 5 章式 (8) 里那个 μ_θ。**两条路只在“要不要加噪声”上分岔，在均值上完全一致。**

公式没错。

6.2.3 弯路二：clamp 的锅?

第 5 章的实现里有一行 `.clamp(-3, 3)`——把 $\hat{x}_0$ 截断到 $[-3,3]$。这是社区里的常见做法，理由是“防止早期预测太离谱”。

怀疑它截断得太狠，把有用的信息切掉了。

于是把 clamp 去掉重跑：**结果更糟**——不是变好，是连 25 步那一档也炸了。

所以 clamp 是在帮忙，不是在捣乱。 它掩盖了一部分问题，但没能解决根本。

6.2.4 如果换成你，该怎么查：一张排查表

上面两条弯路走完，大概花了一个下午。**它们都不是“聪明”的排查，而是任何人在这种情况下都会先试的两件事。** 真正值钱的是下面这张表—— 它把“采样器炸了”这个模糊的症状，拆成了**八个可以单独证伪的假设**。

#	假设	怎么查	如果成立，你会看到
1	$\bar{\alpha}$ 索引错位	打印 `alpha_bar[t]` 和 `alpha_bar[t-1]`，对比前向加噪的实际方差	前向重建的 x_t 与训练时的分布对不上
2	时间步的编码方式不一致	训练用 $t \in [1,T]$、采样用 $t \in [0,T-1]$	模型在 $t=0$ 附近的输出完全无意义
3	模型输出的是 x_0 而不是 ε	看训练损失里减的是哪个量	loss 能降，但采样必炸
4	调度在训练和采样之间被重建	对比两处的 `beta` 数组	数值上差一点点，采样差很多
5	clamp 掩盖了早期错误	去掉 clamp	结果更糟（说明它在帮忙）
6	步数太多导致累积	减少步数	少步数“看起来正常”
7	采样器代码本身错	**换成解析的完美去噪器**	完全正常
8	模型本身不够好	提高训练步数 / 换更大模型	症状减轻但不消失

这张表的顺序不是随意的。 它按“**改动成本从低到高**”排：前四条改一行代码就能验证，第五条改一个参数，第六条改一个数，**第七条要写一个新函数，第八条要重训模型。**

排查的第一原则是：先做便宜的实验。但这条原则有个例外 —— **当便宜的实验全都排除了嫌疑，而你又想不出新的时候，应该立刻跳到最贵的那个对照实验，而不是继续在便宜的地方打转。**

第七行就是那个“最贵但最决定性”的实验。 它的成本是写一个二十行的解析去噪器；它的收益是把假设空间**从“代码 vs 模型”一刀切成两半。**

为什么“换完美去噪器”比读代码有用

读代码只能发现“你写的东西和你以为的不一样”。**它发现不了“你以为的东西本身是错的”。**
而换一个已知正确的组件，是在**检验你的整个理解**：如果解析去噪器跑得通，那说明你对 DDIM 的理解、你的调度、你的索引、你的张量形状，**全都是对的**——因为任何一处错，解析去噪器也会炸。
一次实验，同时验证了二十个假设。这就是对照实验的价值。

6.2.5 这个案例为什么值得花一整节

市面上讲 DDIM 的材料，绝大多数止步于“公式 + 代码 + 在 CIFAR 上 FID 多少”。**你照着抄，在真实数据上跑通了，然后呢?**
然后你在自己的模型上炸了，而你不知道为什么。
本章花这么长的篇幅在一个“失败”上，是因为**这个失败携带的信息量远大于一次成功**:

- 它逼你区分“**代码正确**”和“**结果正确**”——两者不是一回事
- 它逼你理解 $1/\sqrt{\bar{\alpha}_t}$ 这个因子的**数值后果**，而不只是它在公式里的位置
- 它给了你一个**可复用的诊断动作**：换完美去噪器
- 它让你看到“**25 步看起来正常**”这种结果有多危险

最后一条最值得展开。一个只在 25 步下测过的实现，会给出 std $[0.5328, 0.3398]$，而数据是 $[0.5707, 0.3351]$。**任何人看这两个数都会说“对上了”。**
但它是错的。它错在：如果你把这个采样器用到 50 步、100 步—— 也就是别人论文里的标准设置——它会炸。**你交付的不是一个采样器，而是一个只在特定步数下碰巧能用的东西。**
这类 bug 在生产环境里的表现是：离线评估用 50 步，线上为了省钱用 25 步，**于是线下指标很好，线上全是伪影** —— 而且没人会怀疑采样器，因为“25 步的结果我们测过，是对的”。

停下来想一想 ①

clamp 能“帮忙”，说明 $\hat{x}_0$ 的预测已经离谱到什么程度?
提示：数据本身在 $[-1.14, 1.14]$ 之内。± 3 对数据来说是 2.6 倍。
（答案：**说明有相当大比例的 $\hat{x}_0$ 超过了 3。**
实测（100 步 DDIM 的第一步，$t = 999$）：$\hat{x}_0$ 的平均**绝对值**是 **12.73**，**87.6% 的元素被截断。**
也就是说：在采样的第一步，模型给出的“原始数据估计”平均比真实数据大 11 倍，近九成的分量要靠在 clamp 上。
这不是“偶尔预测离谱”，是“每一步都在离谱”。）

6.2.6 决定性实验：换一个不会犯错的去噪器

到这一步，剩下的可能性只有两种:

- **A**：我们的 DDIM 实现有隐藏的 bug
- **B**：DDIM 本身没错，是我们的**模型**不行，而 DDIM 把它放大了

区分它们的办法很直接：把模型换成一个“绝对不会犯错”的去噪器。
什么样的去噪器是完美的?**当数据分布是高斯的时候，最优去噪器有闭式解。**设 $x_0 \sim \mathcal{N}(0, \sigma_0^2)$，则

$$x_t \sim \mathcal{N}(0, \bar{\alpha}_t \sigma_0^2 + 1 - \bar{\alpha}_t), \qquad \mathbb{E}[x_0 \mid x_t] = \frac{\sqrt{\bar{\alpha}_t}\,\sigma_0^2}{\bar{\alpha}_t \sigma_0^2 + 1 - \bar{\alpha}_t}\, x_t \tag{6.2}$$

于是最优的 ε 预测是

$$\varepsilon^\star(x_t, t) = \frac{x_t - \sqrt{\bar{\alpha}_t}\,\mathbb{E}[x_0 \mid x_t]}{\sqrt{1 - \bar{\alpha}_t}} \tag{6.3}$$

这是一行代码，不需要训练，而且它在数学上就是最优的。
用同一个 `ddim()` 函数，把这个解析去噪器传进去：

步数	输出标准差	目标
1000	0.6232	0.6325
250	0.6188	0.6325
100	0.6099	0.6325
50	0.5951	0.6325
20	**0.5509**	0.6325
10	0.4768	0.6325

全部正常。连 20 步都还在 0.55（目标 0.63）。

> ! 这个实验把责任划清了
>
> **同一个 `ddim()` 函数，同一份调度，同一批初始噪声 —— 只换了去噪器的实现。**
>
> · 换成**解析的完美去噪器** → 1000 步到 20 步全部正常
> · 换回**学出来的模型** → 100 步炸到 10^8
>
> **所以 A 不成立：代码没错。**
> **问题在 B：模型不够准，而 DDIM 会把“不够准”变成“爆炸”。**
> **这是本章最重要的一步。**它把一个“到底是哪里错了”的开放式排查，压缩成了“在一个变量上做切换”的对照实验。

6.2.7 元凶：$1/\sqrt{\bar{\alpha}_t}$ 把相对误差放大 157 倍

现在来解释“为什么不够准会变成爆炸”。
看 DDIM 的第一步：

$$\hat{x}_0 = \frac{x_t - \sqrt{1-\bar{\alpha}_t}\,\varepsilon_\theta}{\sqrt{\bar{\alpha}_t}} \tag{6.4}$$

分母是 $\sqrt{\bar{\alpha}_t}$。 在 $t = 999$ 处：

$$\sqrt{\bar{\alpha}_{999}} = \sqrt{4.04 \times 10^{-5}} = 0.00636 \quad \Rightarrow \quad \frac{1}{\sqrt{\bar{\alpha}_{999}}} = 157.2 \tag{6.5}$$

也就是说：$\hat{x}_0$ 是“x_t 与 $\sqrt{1-\bar{\alpha}_t}\varepsilon_\theta$ 之差”再乘 157 倍。
模型误差 r（在 x_t 尺度上）会被放大成 $\hat{x}_0$ 上的绝对误差 $157r$。
实测的第一步（`notebooks/run_ch6_case.py` 的输出，2000 个样本）：

· 平均 $|\hat{x}_0|$（clamp 之前）= 12.8
· 所以 $|x_t - \sqrt{1-\bar{\alpha}_t}\varepsilon_\theta| = 12.8 \times 0.00636 = 0.081$
· 而 $|\varepsilon_\theta| \approx 1.22$，所以**相对误差约 6.6%**
· **86.5% 的坐标已经撞上 ± 3 的 clamp** —— 也就是说，绝大多数样本的 $\hat{x}_0$ 不再是“有偏的估计”，而是“被截断的垃圾”

6.6% 的误差本身并不离谱。一个 5 万参数的模型在最高噪声处差 6.6%，听起来完全合理。**问题是 DDIM 把它乘了 157 倍，然后：**1. clamp 把它截到 3（信息丢失）2. 重建出来的 x_{t-1} 偏离正确轨道 3. 下一步的模型在这个**已经偏离**的输入上继续猜 4. 误差进入正反馈
轨迹实测（100 步，2000 个样本，二范数均值）：

步 i	t	$\|\,x\,\|$	$\|\,\varepsilon_\theta\,\|$	$\|\,\varepsilon_\theta\,\| \,/\, \|\,x\,\|$
0	999	1.24	1.22	0.98
10	898	1.28	1.31	1.02
20	797	1.1×10^1	1.8×10^1	1.70

步 i	t	$\| x \|$	$\| \varepsilon_\theta \|$	$\| \varepsilon_\theta \| / \| x \|$
30	696	4.0×10^3	7.4×10^3	1.86
40	595	1.9×10^6	3.5×10^6	1.87
50	494	7.7×10^8	1.5×10^9	1.89

从第 20 步开始，$|x|$ 每 10 步涨约 260–470 倍。

关键在最后一列之前的那个比值：稳定在 1.9。

一旦 x 离开训练分布（$|x| \gg 1$），模型不再是去噪器，它变成了一个**近似线性放大**的函数：$\varepsilon_\theta \approx 1.9\,x$。这时 DDIM 的更新近似是

$$x_{t-1} \approx \sqrt{1-\bar{\alpha}_{t-1}}\,\varepsilon_\theta \approx 1.9\sqrt{1-\bar{\alpha}_{t-1}}\,x_t \tag{6.6}$$

在 t 较大的区间 $\sqrt{1-\bar{\alpha}_{t-1}} \approx 1$，于是**每步乘 1.9**。$1.9^{10} \approx 613$ —— 和实测的“每 10 步涨 300–470 倍”同一量级（差的那部分来自 clamp 和 $\hat{x}_0$ 项）。

一个乘性因子大于 1 的迭代，指数增长。

所以这个失败可以一句话概括

DDIM 在 t 很大时，把模型的相对误差乘以 $1/\sqrt{\bar{\alpha}_t}$。当这个乘积超过 1，迭代就不稳定。

$\sqrt{\bar{\alpha}_t}$ 在 $t \to T$ 时趋近 0，所以这个放大倍数没有上界。

而 DDPM 为什么不会？因为 DDPM 每步都加 $\sigma_t z$—— **噪声把 x 重新拉回训练分布所在的尺度**，模型下一轮的输入仍然是“它认识的”。这个机制第 5 章讲过（“误差的自我修正”），**在这里得到了一个具体的数值验证。**

6.2.8 反向证据：为什么步数越少反而越稳

回到那个反直觉的现象：25 步正常，100 步炸。

现在可以解释了。

不稳定是**逐步累积**的：如果每一步放大 λ 倍，走 N 步就是 λ^N。**步数越多，指数越大。**

但这只是半个答案——λ 本身也随步长变化。真正发生的事是：

步数少 → 每步跨得大 → 很快走完高噪声区 → 没有足够多轮次让正反馈建立起来。

换句话说，25 步“正常”不是因为 DDIM 在 25 步时更好，而是因为它还没来得及坏。

> ⚠ 这个“正常”是假的
>
> 我们检查了 25 步那组结果的分布：std $[0.5328, 0.3398]$，和数据 $[0.5707, 0.3351]$ 很接近。
>
> **但它是不是一个好的采样？**
>
> **不一定。**因为：1. 只有 25 个样本点被用来估计统计量，误差本身就不小 2. **更关键：它可能只是“没走完”**——高噪声区的错误还没来得及放大，所以结果停在数据附近，但**走的不是正确的概率流**
>
> **这类“看起来对”的结果最危险。**它和本章后面要讲的“假的二阶求解器”是同一类问题：**指标正常，机制错误。**
>
> **判断办法**：换成完美去噪器跑同样的 25 步。如果完美去噪器给出的是 0.55 而你的模型给出 0.53，**两者接近，说明大概率是对的；如果差得远，就要怀疑。**

6.2.9 悬念

到这里，本章的案例给出了一个完整的因果链：

模型有约 6.6% 的误差 → DDIM 乘 157 倍 → 86.5% 的 $\hat{x}_0$ 撞上 clamp → 采样器走出训练分布 → 模型退化成 $\varepsilon_\theta \approx 1.9x$ 的放大器 → 指数增长 → 溢出

但还有三个问题没回答：

1. **那为什么别人用 DDIM 就不炸？**真实模型的误差也是这个量级啊。
2. **第 5 章说 $\mathcal{L}_{\text{simple}}$ 扔掉 ELBO 权重反而更好**，这又是为什么？
3. **“二阶求解器”到底是什么**，为什么能把步数压得更低？

这三个问题分别对应本章的三节。

6.3 DDIM：把 1000 步压成 20 步

6.3.1 先算一遍：哪一步在浪费

回到第 5 章那个 $T=3$、$\beta=[0.1,0.2,0.3]$ 的小例子。 $\bar{\alpha}=[0.9,0.72,0.504]$。

相邻两步之间，$\bar{\alpha}$ 差多少？

步	$\bar{\alpha}_t$	相对上一步
1	0.900	—
2	0.720	**0.800**
3	0.504	**0.700**

每一步把信号乘 0.8 或 0.7 —— 变化不小。

现在看本章的模型：$T=1000$，β 从 10^{-4} 到 0.02 线性。

步	$\bar{\alpha}_t$	相对上一步
100	0.897	0.99984
500	0.0786	0.99888
990	9.0×10^{-5}	0.98020

大部分步只把信号乘 0.9998。 —— 连走两步和走一步， 差别小到可能落在模型的分辨率之下。

DDIM 的第一个观察就是这个：T 步里有大量“几乎没动”的步。**如果能只走“动了”的那些，就能省下大部分计算。**

但这里有个前提问题：DDPM 的反向过程是**逐步**定义的， 跳掉一步就打破了它的链式结构。**凭什么可以跳？**

6.3.2 关键洞见：前向过程可以换，边缘分布不变

DDPM 的整个推导建立在一条前向链上：

$$q(x_{1:T}\mid x_0)=\prod_{t=1}^{T}q(x_t\mid x_{t-1}) \tag{6.7}$$

这条链决定了反向过程必须一步一步走。

DDIM 的洞见是：**反向过程需要的其实不是这条链，而是每个时刻的边缘分布**

$$q(x_t\mid x_0)=\mathcal{N}\big(\sqrt{\bar{\alpha}_t}x_0,(1-\bar{\alpha}_t)I\big) \tag{6.8}$$

只要边缘分布不变，反向过程就可以和别的“前向过程”配对。 而边缘分布只由 $\bar{\alpha}_t$ 这一个数决定 —— **它不关心你中间走了几步。**

所以可以构造一族新的前向分布，它们**不是马尔可夫的**，但边缘分布完全一样：

$$q_\sigma(x_{1:T}\mid x_0)=q(x_T\mid x_0)\prod_{t=2}^{T}q_\sigma(x_{t-1}\mid x_t,x_0) \tag{6.9}$$

其中

$$q_\sigma(x_{t-1}\mid x_t,x_0)=\mathcal{N}\left(\sqrt{\bar{\alpha}_{t-1}}\,x_0+\sqrt{1-\bar{\alpha}_{t-1}-\sigma_t^2}\cdot\frac{x_t-\sqrt{\bar{\alpha}_t}x_0}{\sqrt{1-\bar{\alpha}_t}},\sigma_t^2 I\right) \tag{6.10}$$

这个式子看起来复杂，但它做的事实上只有一件：把 x_t 拆成“信号 $\sqrt{\bar{\alpha}_t}x_0$”和“噪声 $\sqrt{1-\bar{\alpha}_t}\varepsilon$”，**按新的比例重新组装一个 x_{t-1}。**

σ_t 控制重新注入多少噪声：

- $\sigma_t=\sqrt{(1-\bar{\alpha}_{t-1})/(1-\bar{\alpha}_t)}\sqrt{1-\bar{\alpha}_t/\bar{\alpha}_{t-1}}$ → 退回 DDPM
- $\sigma_t=0$ → **完全确定性**（DDIM）

6.3.3 为什么边缘分布一定保得住（一个两行的证明）

上面那个 q_σ 是**构造出来**的，所以必须验证它真的保住了边缘分布。 验证只需要两行。

设 $u = x_t - \sqrt{\bar\alpha_t}x_0$。由边缘分布的定义，$u \sim \mathcal{N}(0, (1-\bar\alpha_t)I)$。 把它代进 q_σ 的均值：

$$\mathbb{E}[x_{t-1} \mid x_0] = \sqrt{\bar\alpha_{t-1}}x_0 + \frac{\sqrt{1-\bar\alpha_{t-1}-\sigma_t^2}}{\sqrt{1-\bar\alpha_t}}\,\mathbb{E}[u] = \sqrt{\bar\alpha_{t-1}}x_0 \tag{6.11}$$

再看方差。x_{t-1} 是 u 的线性函数再加独立的 $\sigma_t z$，所以

$$\mathrm{Var}[x_{t-1} \mid x_0] = \frac{1-\bar\alpha_{t-1}-\sigma_t^2}{1-\bar\alpha_t}\cdot(1-\bar\alpha_t) + \sigma_t^2 = 1-\bar\alpha_{t-1} \tag{6.12}$$

均值和方差都对上了，而高斯分布由均值和方差唯一确定。证毕。

> ℹ 这两行说明了一件比公式更重要的事
>
> **“边缘分布”这个约束，并不唯一确定前向过程。**
> 它只固定了每个时刻的**均值 $\sqrt{\bar\alpha_t}x_0$ 和方差 $1-\bar\alpha_t$**，至于中间怎么从 x_t 走到 x_{t-1}，有无穷多种走法。
> DDPM 选了“马尔可夫、每步加固定方差噪声”的那一种； DDIM 选了“非马尔可夫、指定 σ_t”的那一种。 **它们共享同一组边缘分布，所以共享同一个训练目标。**
> **换句话说：训练只认边缘分布，采样可以不认前向链。**这是整个扩散模型文献里最值钱的一句话，也是后面流匹配、蒸馏、 一致性模型全部建立在上面的那句话。

6.3.4 两个端点的代数

先看 $\sigma_t = 0$。 代进 q_σ：

$$q_0(x_{t-1} \mid x_t, x_0) = \delta\left(x_{t-1} - \sqrt{\bar\alpha_{t-1}}x_0 - \frac{\sqrt{1-\bar\alpha_{t-1}}}{\sqrt{1-\bar\alpha_t}}(x_t - \sqrt{\bar\alpha_t}x_0)\right) \tag{6.13}$$

狄拉克 δ，也就是“没有随机性”。把 $\varepsilon = (x_t - \sqrt{\bar\alpha_t}x_0)/\sqrt{1-\bar\alpha_t}$ 提出来，就得到上一节那个“拆噪声再组装”的形式。

再看 DDPM 那个端点。 把 $\sigma_t^2 = \frac{1-\bar\alpha_{t-1}}{1-\bar\alpha_t}\left(1-\frac{\bar\alpha_t}{\bar\alpha_{t-1}}\right) = \frac{\beta_t(1-\bar\alpha_{t-1})}{1-\bar\alpha_t}$ 代进去，先算系数里的根号：

$$1-\bar\alpha_{t-1}-\sigma_t^2 = \frac{\alpha_t(1-\bar\alpha_{t-1})^2}{1-\bar\alpha_t} \quad\Rightarrow\quad \frac{\sqrt{1-\bar\alpha_{t-1}-\sigma_t^2}}{\sqrt{1-\bar\alpha_t}} = \frac{\sqrt{\alpha_t}(1-\bar\alpha_{t-1})}{1-\bar\alpha_t} \tag{6.14}$$

代回均值，并把 $x_t = \sqrt{\bar\alpha_t}x_0 + u$ 拆开：

$$\mathbb{E}[x_{t-1} \mid x_t, x_0] = \underbrace{\frac{\sqrt{\bar\alpha_{t-1}}\beta_t}{1-\bar\alpha_t}x_0 + \frac{\sqrt{\alpha_t}(1-\bar\alpha_{t-1})}{1-\bar\alpha_t}x_t}_{\text{第 5 章的 DDPM 后验均值}} \tag{6.15}$$

系数对上了。 也就是说：**DDPM 不是 DDIM 的“另一个算法”， 它是同一个 σ 族在 $\sigma_t^2 = \tilde\beta_t$ 处的取值。**

6.3.5 σ_t 能取多大

q_σ 的根号里是 $1-\bar\alpha_{t-1}-\sigma_t^2$，所以必须有

$$\sigma_t^2 \le 1-\bar\alpha_{t-1} \tag{6.16}$$

这个上界不是技术性的，它有物理含义：σ_t 是“这一步重新注入多少噪声”， 而 x_{t-1} 的总噪声预算是 $1-\bar\alpha_{t-1}$。 **注入的不能超过预算，否则 x_{t-1} 的方差就超过边缘分布规定的值了。**

DDPM 的取值 $\tilde\beta_t$ 恰好顶到预算的一个特定比例； DDIM 取 0，一点都不用。

中间值呢？ $\sigma_t^2 = \eta^2 \tilde{\beta}_t$，$\eta \in [0,1]$ —— 这就是本章后面要专门讲的那个 η。**它不是“随机性旋钮”，它是“你在多大程度上不相信模型”的旋钮。**

6.3.6 非马尔可夫意味着什么

$q_\sigma(x_{1:T} \mid x_0)$ 仍然是一个合法的联合分布，只是不能再拆成 $\prod_t q(x_t \mid x_{t-1})$。它拆成的是

$$q_\sigma(x_{1:T} \mid x_0) = q(x_T \mid x_0) \prod_{t=2}^{T} q_\sigma(x_{t-1} \mid x_t, x_0) \tag{6.17}$$

注意条件里有 x_0。这在“生成”的时候是个问题：我们根本没有 x_0，只有模型估计的 $\hat{x}_0$。**而 DDIM 的做法就是：把 x_0 换成 $\hat{x}_0$，然后假装它是真的。**

这个“假装”就是整个 DDIM 采样算法的全部内容。

6.3.7 先算一遍：$\sigma_t = 0$ 时它恰好是对的

用 $T = 3$ 那个例子。设 $x_0 = 1.5$，$\varepsilon = 0.3$，则

$$x_3 = \sqrt{0.504} \times 1.5 + \sqrt{0.496} \times 0.3 = 1.0649 + 0.2113 = 1.2762 \tag{6.18}$$

现在用 q_σ 往回走一步（$\sigma_3 = 0$）：

$$\frac{x_3 - \sqrt{\bar{\alpha}_3} x_0}{\sqrt{1 - \bar{\alpha}_3}} = \frac{1.2762 - 1.0649}{0.7043} = 0.3000 = \varepsilon \quad \checkmark \tag{6.19}$$

拆出来的正是当初加进去的那个 ε。于是

$$x_2 = \sqrt{0.72} \times 1.5 + \sqrt{0.28} \times 0.3 = 1.2728 + 0.1587 = 1.4315 \tag{6.20}$$

而按前向过程的定义：$x_3 = \sqrt{0.7} x_2 + \sqrt{0.3} \varepsilon_3$，所以 x_2 应该满足 $\sqrt{0.8} x_1 + \sqrt{0.2} \varepsilon_2 = \ldots$

等一下 —— 这里要小心。上面算出的 $x_2 = 1.4315$ 应该是“从 x_3 反推的 x_2”。而前向过程里，x_2 本来就是

$$x_2 = \sqrt{0.72} \times 1.5 + \sqrt{0.28} \times \bar{\varepsilon}_2 \tag{6.21}$$

如果 $\bar{\varepsilon}_2 = 0.3$（也就是和 ε_3 用的是同一个噪声），两者完全一致。

但前向过程里，x_3 是用 ε_3 造的，x_2 是用 $\bar{\varepsilon}_2$ 造的，两者不同。

所以这里的“一致”只在 ε 相同的情况下成立 —— 而这正是重点。

> ! 这一步在说什么
>
> **DDIM 的确定性更新，等价于“假设当初全程只加了一个噪声 ε，然后沿着这条轨迹精确地倒着走”。**
>
> **这当然是一个近似** —— 真实的前向过程每步加的是不同的噪声。**但它是一个“边缘分布正确”的近似**：如果 ε 能被完美预测，那么倒着走出来的每个 x_t 都服从正确的边缘分布 $q(x_t|x_0)$。
>
> **而 DDPM 的随机更新，是“每一步都承认自己不知道之前加了什么噪声”，所以在每一步都重新注入一点随机性。**
>
> **两者的区别不在正确性，在“你有多相信模型”。**

6.3.8 上公式

把 $\sigma_t = 0$ 代进去，并把 x_0 换成模型的估计 $\hat{x}_0$：

$$\hat{x}_0 = \frac{x_t - \sqrt{1 - \bar{\alpha}_t}\, \varepsilon_\theta(x_t, t)}{\sqrt{\bar{\alpha}_t}} \tag{6.22}$$

$$x_{t-1} = \sqrt{\bar{\alpha}_{t-1}}\, \hat{x}_0 + \sqrt{1 - \bar{\alpha}_{t-1}}\, \varepsilon_\theta(x_t, t) \tag{6.23}$$

把 $\hat{x}_0$ 代进去展开：

$$x_{t-1} = \frac{\sqrt{\bar{\alpha}_{t-1}}}{\sqrt{\bar{\alpha}_t}} x_t - \frac{\sqrt{\bar{\alpha}_{t-1}}\sqrt{1-\bar{\alpha}_t}}{\sqrt{\bar{\alpha}_t}} \varepsilon_\theta + \sqrt{1-\bar{\alpha}_{t-1}}\,\varepsilon_\theta \tag{6.24}$$

第一项用 $\bar{\alpha}_{t-1}/\bar{\alpha}_t = 1/\alpha_t$；后两项合并：

$$= \frac{1}{\sqrt{\alpha_t}} x_t + \left(\sqrt{1-\bar{\alpha}_{t-1}} - \frac{\sqrt{1-\bar{\alpha}_t}}{\sqrt{\alpha_t}} \right) \varepsilon_\theta \tag{6.25}$$

注意 (2) 和 DDPM 的均值（第 5 章式 7）不一样。DDPM 的 x_t 系数也是 $1/\sqrt{\alpha_t}$，但 ε_θ 前面的系数不同，而且 DDPM 还多一项 $\sigma_t z$。

两者在 σ_t 取特定值时才重合。

6.3.9 跳步：把 $t-1$ 换成任意 $t' < t$

式 (6.23) 里根本没有出现“相邻”这个词。

它只用到 $\bar{\alpha}_t$ 和 $\bar{\alpha}_{t'}$ 两个数。**所以完全可以把 t' 取成 $t-10$、$t-40$，甚至 0**。

这就是 DDIM 能跳步的全部理由：**它的更新式只依赖边缘分布，而边缘分布只看 $\bar{\alpha}$ 的值，不看中间走了几步**。

代价：跳得越大，“ε 全程只有一个”这个假设越不成立。所以跳步会让结果偏离 —— **但偏离是可以量化的**（下面实测）。

6.3.10 实测：完美去噪器下，步数与质量

这是本章最干净的一张表 —— **用解析的完美去噪器**，把“模型不准”这个因素完全排除掉：

步数	输出标准差	与目标的偏差
1000	0.6232	−1.5%
250	0.6188	−2.2%
100	0.6099	−3.6%
50	0.5951	−5.9%
20	0.5509	−12.9%
10	0.4768	−24.6%

（目标 0.6325。）

这才是 DDIM 的真实性能：20 步掉 13%，10 步掉 25%。

而它比 DDPM 快 50 倍。

> **i** 停下来想一想 ②
>
> 上面那张表里，标准的下降是**单调**的吗？偏差是怎么随步数增长的？
>
> （答案：**单调，而且增长在加速**。
>
> 从 1000 → 250（4 倍少）偏差只从 1.5% 涨到 2.2%；从 50 → 20（2.5 倍少）偏差从 5.9% 涨到 12.9%。
>
> **换句话说：省步数的收益在前半段很大，后半段迅速变差**。
>
> **这给了一个很实用的判据**：如果你的目标是“质量损失不超过 5%”，那么 50–100 步就够了；如果要更好，1000 步的收益也很有限（只比 250 步好 0.7 个百分点）。
>
> **但注意这只在“完美去噪器”上成立**。模型有误差时，步数和稳定性的关系会完全不同 —— 本章的案例就是反例。

6.3.11 回到案例：为什么我们的 DDIM 炸了

现在可以把案例的因果链和 DDIM 的数学对上了。

式 (6.23) 里有一项在 t 大时会爆：

$$\hat{x}_0 = \frac{x_t - \sqrt{1-\bar{\alpha}_t}\,\varepsilon_\theta}{\sqrt{\bar{\alpha}_t}} \tag{6.26}$$

分母 $\sqrt{\bar{\alpha}_t}$ 随 $t \to T$ 趋于 0。
而 x_{t-1} 是由 $\hat{x}_0$ 重建的：

$$x_{t-1} = \sqrt{\bar{\alpha}_{t-1}}\,\hat{x}_0 + \sqrt{1-\bar{\alpha}_{t-1}}\,\varepsilon_\theta \tag{6.27}$$

在 t 很大时 $\sqrt{\bar{\alpha}_{t-1}}$ 也很小，所以第一项本来就很小——**但它是“错得很厉害的小”**。
换句话说：DDIM 在最高噪声的几步里，做的是“除以一个极小的数，再乘以另一个极小的数”。**数学上它们抵消，数值上误差不抵消。**
实测的放大倍数是 157 倍（$t = 999$ 处 $1/\sqrt{\bar{\alpha}} = 157.2$）。
这就是为什么“模型准不准”在 DDPM 和 DDIM 上的后果完全不同：DDPM 的每一步都有一个新的随机项把误差冲淡，而 DDIM **没有**——它把所有误差原样累积下去。

6.4 score matching：同一件事的另一种写法

到这里 DDIM 的推导已经能用了。**但还有一个更统一的视角**，它把 DDPM、DDIM、以及所有“用 ODE 采样”的方法放进同一个框架。
这个视角叫 score matching。第 5 章末尾提过一句（式 24），这一节把它讲清楚。

6.4.1 score 是什么

给定一个概率密度 $p(x)$，它的 score 是

$$s(x) := \nabla_x \log p(x) \tag{6.28}$$

也就是“对数密度对位置的梯度”。
它回答的问题是：**在这个点上，往哪个方向移动，密度上升得最快？**
一个关键性质：$p(x) = \frac{1}{Z}\tilde{p}(x)$ 时，$\nabla_x \log p = \nabla_x \log \tilde{p}$ —— **归一化常数 Z 被求导消掉了。**
这一点非常重要：第 5 章说过 $p_\theta(x_0)$ 积不出来（那个 2000 重积分）。**但 score 不需要那个积分。**

6.4.2 Tweedie 公式：去噪器为什么“自动”就是 score

上面那个推导只对“给定 x_0 的高斯条件密度”做了微分。**但真正的 $p_t(x_t)$ 是混合分布**：

$$p_t(x_t) = \int p_t(x_t \mid x_0)\, q(x_0)\, dx_0 \tag{6.29}$$

混合分布的梯度不能直接套单个高斯的公式。这里需要一个经典结果。
Tweedie 公式：设 $y \mid \mu \sim \mathcal{N}(\mu, \sigma^2 I)$，先验 $\mu \sim \pi(\mu)$，则

$$\mathbb{E}[\mu \mid y] = y + \sigma^2 \nabla_y \log p(y) \tag{6.30}$$

证明只用分部积分，一行：

$$\nabla_y p(y) = \nabla_y \int \frac{1}{(2\pi\sigma^2)^{d/2}} e^{-\frac{\|y-\mu\|^2}{2\sigma^2}} \pi(\mu)\, d\mu = \int \frac{\mu - y}{\sigma^2} \cdot \frac{e^{-\cdots}}{(2\pi\sigma^2)^{d/2}} \pi(\mu)\, d\mu \tag{6.31}$$

把 $\frac{1}{\sigma^2}$ 提到外面，两边除以 $p(y)$，就是 $\nabla_y \log p(y) = \frac{\mathbb{E}[\mu|y]-y}{\sigma^2}$，移项即得。
把符号对上。我们的“观测”是 x_t，“隐变量”是 x_0，噪声方差是 $1-\bar{\alpha}_t$。于是

$$\mathbb{E}[x_0 \mid x_t] = x_t + (1-\bar{\alpha}_t)\nabla_{x_t} \log p_t(x_t) \tag{6.32}$$

而模型学的是 $\hat{x}_0 = \mathbb{E}[x_0 \mid x_t]$。代进去：

$$\nabla_{x_t} \log p_t(x_t) = \frac{\hat{x}_0 - x_t}{1-\bar{\alpha}_t} = -\frac{x_t - \hat{x}_0}{1-\bar{\alpha}_t} = -\frac{\varepsilon_\theta}{\sqrt{1-\bar{\alpha}_t}} \tag{6.33}$$

这就是 式 (6.44)，而且这次是对混合分布证的。

> **i** 这一步在概念上值多少
>
> **它把“训练一个去噪器”和“估计一个分布的对数梯度”划了等号。**
>
> · 训练目标 $\| \varepsilon_\theta - \varepsilon \|^2$ 看起来只是“拟合噪声”
> · 但它的总体最优解**恰好**是 $\nabla \log p_t$
> · 所以一个被训练去“猜噪声”的网络，**实际上是在偷偷学整个数据分布的 log 密度梯度**
>
> **这也解释了一件第 5 章没有解释的事**：为什么 $\mathcal{L}_{\text{simple}}$ 里对 ε 做等权回归，得到的却是一个在所有噪声水平上都可用的 score 估计？因为 score 的**真值**就是 $-\varepsilon/\sqrt{1-\bar{\alpha}_t}$，而 ε 本身在 $\mathcal{N}(0, I)$ 尺度上是**同尺度**的（第 5 章那个 8500 倍的说法）。

6.4.3 从 score 到采样器：三种走法

有了 $s_\theta(x,t) \approx \nabla \log p_t(x)$，至少有三种方式把它变成采样算法。

走法一：Langevin 动力学（只在单个噪声水平上）

$$x \leftarrow x + \tau\, s_\theta(x,t) + \sqrt{2\tau}\, z \tag{6.34}$$

问题：它只在一个固定的 t 上做，采样出来的是 p_t，不是 p_0。要用它采样数据，必须从 $t = T$ 开始、每一层都跑一遍 Langevin，这就是“退火 Langevin”（annealed Langevin），**慢，而且步长极难调**。

走法二：反向 SDE

第 5 章的反向过程就是一个 SDE：

$$dx = \left[f(x,t) - g(t)^2 \nabla \log p_t(x) \right] dt + g(t)\, d\bar{w} \tag{6.35}$$

它每步注入噪声，所以是随机的。DDPM 是它的一个离散化。

走法三：概率流 ODE

把同一个 Fokker-Planck 方程的扩散项“搬走”，剩下的漂移项就给出一个确定性 ODE。这是下一小节要证的。

走法	随机性	每步代价	典型代表
Langevin（退火）	有	高（每层多步）	NCSN
反向 SDE	有	低（一步一求值）	DDPM
概率流 ODE	**无**	低	DDIM / DPM-Solver

三条路走的是同一个分布，只是“怎么走”不同。

6.4.4 概率流 ODE 是从哪来的

先说结论：ODE 式 (6.47) 和反向 SDE 共享同一组边缘分布 p_t。两条路都能证，它们的难点不同。

6.4.4.1 路线一：连续性方程（不需要反向 SDE）

一个确定性 ODE $\dot{x} = v(x,t)$ 会把概率密度按**连续性方程**搬运：

$$\frac{\partial p_t}{\partial t} = -\nabla \cdot [v\, p_t] \tag{6.36}$$

取 $v = f - \frac{1}{2} g^2 \nabla \log p_t$：

$$\frac{\partial p_t}{\partial t} = -\nabla\cdot[f p_t] + \frac{1}{2}g^2\nabla\cdot[p_t\nabla\log p_t] = -\nabla\cdot[f p_t] + \frac{1}{2}g^2\Delta p_t \tag{6.37}$$

最后一步用了 $p_t\nabla\log p_t = \nabla p_t$。
而这恰好是前向 SDE $dx = f\,dt + g\,dw$ 的 Fokker-Planck 方程。**所以这个 ODE 搬运密度的方式和前向 SDE 完全一样，边缘分布当然一样。证毕。**
这条路线只用了一个恒等式，是本书推荐的证法。

6.4.4.2 路线二：从反向 SDE 出发

反向 SDE（Anderson 1982），用**反向时间** $s = T - t$ 写：

$$d\bar{x} = [-f + g^2\nabla\log p]ds + g\,d\bar{w} \tag{6.38}$$

注意 f 变号了 —— 时间反演把漂移的方向也翻了。这是最容易漏的一步。
换成“t 递减”的写法：

$$dx = [f - g^2\nabla\log p_t]dt + g\,d\bar{w}, \qquad dt < 0 \tag{6.39}$$

验证它保住 p_t。令 $q_s = p_{T-s}$，对 s 写 Fokker-Planck：

$$\frac{\partial q_s}{\partial s} = -\nabla\cdot[(-f + g^2\nabla\log q)q] + \frac{1}{2}g^2\Delta q = \nabla\cdot[fq] - g^2\Delta q + \frac{1}{2}g^2\Delta q = \nabla\cdot[fq] - \frac{1}{2}g^2\Delta q \tag{6.40}$$

而 $q_s = p_{T-s}$，所以 $\partial_s q = -\partial_t p$。把前向的 $\partial_t p = -\nabla\cdot[fp] + \frac{1}{2}g^2\Delta p$ 代进去，两边相等。√
现在把扩散项“关掉”，只留漂移。但漂移里的系数要从 g^2 改成 $\frac{1}{2}g^2$：

$$\frac{dx}{dt} = f - \frac{1}{2}g^2\nabla\log p_t \tag{6.41}$$

为什么减半？因为 SDE 的噪声项 $g\,dw$ 在 Fokker-Planck 里贡献的是 $\frac{1}{2}g^2\Delta p$。要在一个**没有噪声**的过程里复现同样的 Δp 项，漂移里就只能是 $\frac{1}{2}g^2\nabla\log p$。**少写这个 $\frac{1}{2}$，边缘分布就会以两倍的速度扩散掉。**
得到的正是 式 (6.47)。

> ⚠ 警告
>
> **三个约定，错一个就全错**
>
> 1. **g 的定义里有没有 $\sqrt{2}$。**本书用 $dx = f\,dt + g\,dw$，Fokker-Planck 的扩散项是 $\frac{1}{2}g^2\Delta p$。有的书写 $dx = f\,dt + \sqrt{2\beta}\,dw$，扩散项就是 $\beta\Delta p$。
> 2. **反向 SDE 的漂移里是 g^2 还是 $\frac{1}{2}g^2$。**在“t 递减”的写法里是 g^2；变成概率流 ODE 时才是 $\frac{1}{2}g^2$。
> 3. **时间反演时 f 要变号。**
>
> **检查办法只有一个**：把候选方程代进 Fokker-Planck，看能不能还原前向的 $\partial_t p$。**对不上就是约定错了，不用怀疑数学。**

6.4.5 σ 空间：把 DDIM 写成一次显式欧拉

ODE 式 (6.47) 里 t 从 T 走到 0，$\bar{\alpha}_t$ 跨了五个数量级，**直接离散化会非常不均匀**。所以要把变量换掉。
换元：令 $\hat{x} = x/\sqrt{\bar{\alpha}_t}$，$\sigma_t = \sqrt{1-\bar{\alpha}_t}/\sqrt{\bar{\alpha}_t}$。（这个 σ_t 和上一节的 σ_t 不是同一个东西 —— **这是本章最坑的一处记号冲突**，DDIM 论文里 σ_t 指“注入噪声的幅度”，而在 score 文献里 σ 指“信噪比的倒数”。**本节以下一律用 σ 表示后者，注入噪声那个改用 η。**）
在这个参数化下，$x_t = \sqrt{\bar{\alpha}_t}\hat{x}$，而 $\hat{x} = x_0 + \sigma\varepsilon$，**正好是“信号 + 噪声”的标准形式。**
把 ODE 换元：对 $\hat{x} = x/\sqrt{\bar{\alpha}}$ 求导，

$$\frac{d\hat{x}}{dt} = \frac{1}{\sqrt{\bar{\alpha}}}\frac{dx}{dt} - \frac{x}{2\bar{\alpha}^{3/2}}\frac{d\bar{\alpha}}{dt} \tag{6.42}$$

而 $\varepsilon_\theta = -\sqrt{1-\bar\alpha}\,\nabla \log p_t$。 代入并整理，会发现所有含 $\bar\alpha$ 的项都消掉，只剩

$$\frac{d\hat{x}}{d\sigma} = \varepsilon_\theta(\hat{x}, \sigma) \tag{6.43}$$

这是一个极其干净的形式：$\hat{x}$ 对 σ 的导数，就是去噪器的输出。
欧拉法：$\hat{x}_{k+1} = \hat{x}_k + (\sigma_{k+1} - \sigma_k)\varepsilon_\theta(\hat{x}_k, \sigma_k)$。
从 $\sigma_{\max} \approx 152$ 走到 $\sigma = 0$，一共 N 步。这就是 DDIM。
所以 DDIM 不是什么"新采样器"——它是概率流 ODE 在 σ 空间里的显式欧拉法。后面的 DPM-Solver、Heun、以及所有"高阶求解器"，做的都是同一件事：**在 σ 空间里，用更好的方法解这个 ODE。**

6.4.6 从 ε 到 score

第 5 章给过这个关系（式 24）：

$$\varepsilon_\theta(x_t, t) = -\sqrt{1-\bar\alpha_t}\,\nabla_{x_t} \log p_t(x_t) \tag{6.44}$$

推导只用一步。 高斯情形下

$$p_t(x_t \mid x_0) = \mathcal{N}\big(\sqrt{\bar\alpha_t}x_0, (1-\bar\alpha_t)I\big) \tag{6.45}$$

$$\nabla_{x_t} \log p_t(x_t \mid x_0) = -\frac{x_t - \sqrt{\bar\alpha_t}x_0}{1-\bar\alpha_t} = -\frac{\varepsilon}{\sqrt{1-\bar\alpha_t}} \tag{6.46}$$

移项即得 (3)。
所以 (3) 在说什么
"预测噪声"和"估计 score"是同一件事，只差一个负的常数因子。
为什么这个等价有用？ 因为 score 有自己的理论工具：

- **score 不需要归一化常数** —— 这解释了为什么扩散模型能绕开那个积不出来的 $p(x_0)$
- **score 的尺度在各噪声水平上是稳定的** —— 这正是第 5 章那个 "ε 损失的动态范围比 x_0 小 8500 倍"的另一种说法
- **所有基于 score 的采样器（Langevin、ODE、SDE）都能直接搬过来用**

换句话说：我们一直叫它"预测噪声"，但它真正的数学身份是"估计 score"。

6.4.7 概率流 ODE

有了 score，就可以定义一个**确定性**的过程：

$$\frac{dx}{dt} = f(x,t) - \frac{1}{2}g(t)^2\,\nabla_x \log p_t(x) \tag{6.47}$$

这是"概率流 ODE"（probability flow ODE）。**它的性质是：沿着它走，每个时刻的边缘分布 p_t 都保持不变。**
所以：从 $p_T = \mathcal{N}(0, I)$ 出发，解这个 ODE 到 $t = 0$，就得到数据分布。
注意它是确定性的 —— 给定起点，终点唯一。

6.4.8 DDIM 就是它的一阶离散化

把 (3) 代进 (4)，再用最简单的欧拉法离散化，得到的就是 DDIM。
验证一下。 在"$x_t = \sqrt{\bar\alpha_t}x_0 + \sqrt{1-\bar\alpha_t}\varepsilon$"这个参数化下，可以换元成 $\hat{x} = x/\sqrt{\bar\alpha}$，此时

$$\frac{d\hat{x}}{d\sigma} = \varepsilon_\theta, \qquad \sigma_t := \frac{\sqrt{1-\bar\alpha_t}}{\sqrt{\bar\alpha_t}} \tag{6.48}$$

欧拉法就是

$$\hat{x}_{t-1} = \hat{x}_t + (\sigma_{t-1} - \sigma_t)\,\varepsilon_\theta \tag{6.49}$$

把它换回 x：两边乘 $\sqrt{\bar{\alpha}_{t-1}}$，整理后**恰好是 DDIM 的 式 (6.23)**。

> ! 这个等价解释了本章的两个现象
>
> ① **为什么 DDIM 是确定性的**。因为 ODE 本身没有随机性。**确定性不是 DDIM 的设计选择，是“用 ODE 采样”这个决定的直接后果**。
> ② **为什么 DDIM 会炸，而 DDPM 不会**。ODE 的解对初值和向量场都是**连续依赖**的，但**误差会累积**—— 没有随机项来“重置”它。SDE（DDPM）每步注入的噪声，相当于在数值积分里做了一次“重新投影”。
> **一句话：ODE 精确但脆弱，SDE 有噪声但稳健**。

6.4.9 这个视角的实际用处

它不只是一个漂亮的重述。它带来三件事：

1. **任何 ODE 求解器都能直接用**。欧拉是一阶，还有二阶（Heun）、三阶（DPM-Solver-3）。**这就是下一节的内容**。
2. **它解释了“为什么 DDIM 的步数可以很少”**：因为 ODE 的解是一条光滑曲线，跳步相当于用更少的采样点去逼近它。
3. **它把“采样”变成了“数值积分”** —— 一个有几百年历史的成熟领域。

6.5 高阶求解器：为什么二阶比一阶快

6.5.1 一个和扩散无关的例子

先离开扩散，看一个纯数学问题。
要解 $\frac{dy}{dt} = y$，$y(0) = 1$，走到 $t = 1$。真解是 $y(1) = e = 2.71828$。
一阶（欧拉）：$y_{n+1} = y_n + h \cdot y_n = y_n(1 + h)$。走 N 步，$h = 1/N$，结果是 $\left(1 + \frac{1}{N}\right)^N$。

N	欧拉结果	相对误差
4	2.4414	−10.2%
8	2.5658	−5.6%
16	2.6379	−3.0%
32	2.6769	−1.5%

误差大致按 $1/N$ 下降 —— 步数翻倍，误差减半。这是**一阶**方法的标志。
二阶（Heun）：先走一步欧拉得到预测值，再在预测点上求一次斜率，取平均。

$$y_{n+1} = y_n + \frac{h}{2}(f(y_n) + f(y_n + hf(y_n))) \tag{6.50}$$

N	Heun 结果	相对误差
4	2.6479	−2.6%
8	2.7015	−0.6%
16	2.7140	−0.16%

误差按 $1/N^2$ 下降 —— 步数翻倍，误差降到 1/4。
这就是“阶”的含义：一阶方法每多算一步，误差线性下降；二阶方法每多算一步，误差平方下降。
代价：Heun 每步要算**两次** f。所以公平的比较是 “Heun 走 N 步” vs “欧拉走 $2N$ 步”。

函数求值次数	欧拉误差	Heun 误差
8	−5.6%	−2.6%（4 步 × 2）

函数求值次数	欧拉误差	Heun 误差
16	−3.0%	−0.6%（8 步 × 2）
32	−1.5%	−0.16%（16 步 × 2）

同样的计算量，二阶比一阶准 3–10 倍。

6.5.2 扩散里的“斜率”是什么

上一节已经证明：DDIM 就是概率流 ODE 的欧拉离散化。
在 $\hat{x} = x/\sqrt{\bar{\alpha}}$、$\sigma = \sqrt{1-\bar{\alpha}}/\sqrt{\bar{\alpha}}$ 这个换元下：

$$\frac{d\hat{x}}{d\sigma} = \varepsilon_\theta(\hat{x}, \sigma) \tag{6.51}$$

所以“斜率”就是模型的输出 ε_θ。
DDIM 的一步：

$$\hat{x}_{t-1} = \hat{x}_t + (\sigma_{t-1} - \sigma_t)\,\varepsilon_\theta(\hat{x}_t, \sigma_t) \tag{6.52}$$

括号里只有“在起点算的那一次 ε”。这就是欧拉。

6.5.3 为什么步长不能在 σ 上均匀

在讲二阶之前，必须先讲一个**比阶数更值钱**的问题：**网格怎么取。**
大多数人第一次写 DDIM 会这么写：

```
idx = torch.linspace(T - 1, 0, steps)     # 在 t 上均匀
```

看起来天经地义：1000 步压成 20 步，不就是每 50 步取一个点吗？ 但 t 均匀**不等于** σ 均匀。而 ODE 是在 σ 上积分的。
把这件事测出来。 用完美去噪器（排除模型误差），只改网格，其他全不动：

步数	σ 均匀	$\log\sigma$ 均匀
10	0.0228	0.4201
20	0.0480	0.5216
40	0.0972	0.5744
80	0.1880	0.6016

目标标准差 0.6325。**同样的函数求值次数，$\log\sigma$ 均匀全面碾压**：10 步时是 18 倍，80 步时还有 3 倍。
而 σ 均匀的那个甚至越走越糟 —— 0.0228 到 0.1880，**加步数只在缓慢地补救，因为它把绝大多数步数浪费在了 σ 很大的区间。**
为什么？ σ 从 152 降到 0，跨了四个数量级。**均匀切 20 份，每份宽 7.6**，于是：

- 前 19 步都在 $\sigma \in [7.6, 152]$ 里走 —— 这个区间里数据的信息几乎为零
- **最后一步从 $\sigma = 7.6$ 直接跳到 0** —— 所有细节都靠这一步

而 $\log\sigma$ 均匀是“按比例”分配：σ 越大步子越大，σ 越小步子越细。**这正好匹配 ODE 的行为** —— 在 $\sigma \to 0$ 附近，$\hat{x}$ 的变化最快。
一句话：误差不是均匀产生的，步长也不该均匀
在 σ 大的地方，$\hat{x}$ 几乎是纯噪声，走错一点无所谓；在 σ 小的地方，$\hat{x}$ 已经接近数据，**走错一点就是可见的伪影。**
所以采样器的时间表必须“前粗后细”。
DDIM 论文里的 `linspace(T-1, 0, steps)` 之所以能用，**不是因为 t 均匀是对的**，而是因为这个线性 β 调度下，t 均匀**碰巧**接近 $\log\sigma$ 均匀。换个调度（比如余弦），这个巧合就没了 —— **这就是为什么换调度必须重调步数。**

这也是 DPM-Solver 系列真正贡献的一半。它们不只是“用了高阶方法”，而是**在 $\lambda = \log(\alpha/\sigma)$ 上均匀取点**，并且把 ODE 拆成“线性部分精确解 + 非线性部分低阶近似”。**阶数只是另一半**。

本书不复现 DPM-Solver 的完整公式（它有三种阶数、两种末步修正，记号比本章其余部分加起来还多），但你可以用上面的表来检验任何实现：**把同一个采样器分别喂 σ 均匀和 $\log\sigma$ 均匀的网格，如果差别巨大，说明它没有做 λ 参数化**。

6.5.4 Heun 版本

二阶就是把“起点算一次、预测点再算一次、取平均”这套照搬过来：

$$\begin{aligned} k_1 &= \varepsilon_\theta(\hat{x}, \sigma_t) && \text{（起点）} \\ \hat{x}_{\text{pred}} &= \hat{x} + (\sigma_{t-1} - \sigma_t)\, k_1 && \text{（欧拉预测）} \\ k_2 &= \varepsilon_\theta(\hat{x}_{\text{pred}}, \sigma_{t-1}) && \text{（预测点）} \\ \hat{x}_{t-1} &= \hat{x} + \frac{\sigma_{t-1} - \sigma_t}{2}(k_1 + k_2) && \text{（平均）} \end{aligned} \tag{6.53}$$

代价是模型调用次数翻倍。

6.5.5 实测：同样的模型，二阶能省多少步

用**解析的完美去噪器**（排除模型误差干扰），1D 高斯，目标标准差 0.6325：

步数	Euler（一阶）	Heun（二阶）	Euler 误差	Heun 误差
10	0.4847	0.6802	**−23.3%**	**+7.5%**
20	0.5601	0.6450	−11.4%	**+2.0%**
40	0.5975	0.6375	−5.5%	+0.8%
80	0.6163	0.6357	−2.6%	+0.5%

两件事值得注意。

① **二阶确实更快**。要把误差压到 2% 以内：Euler 需要约 100 步，Heun 只需要 **20 步 × 2 次求值 = 40 次**。**算力省了一半以上**。

② **但 10 步时 Heun 反而“过头”了**。Euler 是 −23%，Heun 是 +7.5%。**方向相反，量级都很大**。

这不是 bug：在步长这么大的时候，两种方法都还没进入“渐近收敛区”，误差的符号取决于具体路径。**“高阶更准”是一个渐近结论，不是无条件成立的**。

> ⚠ 步数太少时，阶数救不了你
>
> **看 10 步那一行：Heun 也已经错了 7.5%**。
> **一阶和二阶的差别在“中等步数”最大**（20–40 步这一带），在极端少的步数下两者都不可靠。
> **实践含义**：如果你的目标是 10 步以内，**换更高阶的求解器不如换别的思路**（蒸馏、对抗微调、或者干脆接受质量下降）。

6.5.6 一个我自己踩过的坑：假的二阶

写这一节的时候，我先写了一版 Heun，跑出来的结果是“和欧拉一模一样”。当时以为“扩散的 ODE 太简单，二阶没优势”，差点就这么写进书里。

后来检查代数，发现那版“二阶”是恒等式。

它长这样：

$$
\begin{aligned}
\hat{x}_0 &= \frac{x - \sqrt{1-a}\, e_1}{\sqrt{a}} \\
x_{\text{pred}} &= \sqrt{a'}\, \hat{x}_0 + \sqrt{1-a'}\, e_1 \qquad \text{（欧拉步）} \\
e_2 &= \varepsilon_\theta(x_{\text{pred}}, t') \\
\hat{x}_0' &= \frac{x_{\text{pred}} - \sqrt{1-a'}\, e_2}{\sqrt{a'}} \\
x_{\text{out}} &= \sqrt{a'}\, \frac{\hat{x}_0 + \hat{x}_0'}{2} + \sqrt{1-a'}\, \frac{e_1 + e_2}{2}
\end{aligned}
\tag{6.54}
$$

它看起来很像 Heun：算两次 ε，取平均。
但实际上

$$
\sqrt{a'}\, \hat{x}_0' + \sqrt{1-a'}\, e_2 = \sqrt{a'} \cdot \frac{x_{\text{pred}} - \sqrt{1-a'}\, e_2}{\sqrt{a'}} + \sqrt{1-a'}\, e_2 = x_{\text{pred}} \tag{6.55}
$$

而 $\sqrt{a'}\hat{x}_0 + \sqrt{1-a'}e_1$ 也等于 x_{pred}。
所以 $x_{\text{out}} = \frac{x_{\text{pred}} + x_{\text{pred}}}{2} = x_{\text{pred}}$ —— 它原样返回了欧拉步。

> ! 这个坑为什么值得单独讲
>
> **因为它不会报错，也不会给出荒谬的结果。** 它给出的是**完全正常的欧拉结果**。
> **你会以为自己在用二阶，实际上一直在用一阶。**
> **判断一个“改进”是不是真的改了，有一个不用推导的办法：**
>
> **换一个它应该影响很大的设置（比如极少的步数），看输出有没有变化。如果没有变化，它多半是恒等式。**
>
> **我当初就是这么发现的**：10 步下两个“不同”的求解器给出到小数点后六位都一样的数。
> **这类“实现上看似生效、数学上恒等”的 bug，和本章案例里那个 DDIM 爆炸是一对：前者是“改了但没效果”，后者是“没改对但后果巨大”。两者都只能靠“拿一个已知答案的对照去测”发现。**

6.6 数值刚性：为什么步长会被 σ 卡住

上一节说“用更好的求解器能省步数”。**但有一个下限，任何阶数都突破不了。** 这一节讲那个下限从哪来。

6.6.1 先看一个具体的数

回到 σ 空间的 ODE：$\frac{d\hat{x}}{d\sigma} = \varepsilon_\theta(\hat{x}, \sigma)$。
右边的大小是多少？ 在完美去噪器下，$\varepsilon = (x - \hat{x}_0)/\sigma$，而 $\hat{x}_0 = \frac{V_0}{V_0+\sigma^2}\hat{x}$，所以

$$
\varepsilon = \frac{\sigma}{V_0 + \sigma^2}\hat{x} \tag{6.56}
$$

在 σ 大时（$\sigma^2 \gg V_0$）：$\varepsilon \approx \hat{x}/\sigma$，很小。**在 σ 小时**（$\sigma^2 \ll V_0$）：$\varepsilon \approx \sigma\hat{x}/V_0$，也很小。**最大值在 $\sigma = \sqrt{V_0}$ 处**，此时 $|\varepsilon| = |\hat{x}|/(2\sqrt{V_0})$。
所以 ε 本身并不大。问题不在它的大小，在它的变化率。

6.6.2 刚性是什么

一个 ODE 叫“刚性”（stiff），如果它的解变化很慢，但解附近的“邻近轨迹”变化很快。

后果很直接：显式方法（欧拉、Heun、Runge-Kutta）**为了保证数值稳定，步长必须由“最快的那条邻近轨迹”决定，而不是由“解本身”决定。**于是你被迫走很多小步，**而其中大部分步数对精度毫无贡献。**

扩散 ODE 是刚性的吗？是，而且刚性的来源很具体：**在 $\sigma \to 0$ 附近，ε 对 $\hat{x}$ 的偏导数会变大。**算一下。完美去噪器的雅可比：

$$\frac{\partial \varepsilon}{\partial \hat{x}} = \frac{\partial}{\partial \hat{x}}\left(\frac{\sigma}{V_0+\sigma^2}\hat{x}\right) = \frac{\sigma}{V_0+\sigma^2} \tag{6.57}$$

这个量在 $\sigma \to 0$ 时趋于 0，在 σ 大时也趋于 0，最大值在 $\sigma = \sqrt{V_0}$。

所以按这个模型，它并不刚。那为什么欧拉需要 100 步？

因为误差不在雅可比，在“每一步跨过了多少个 σ 的尺度”。σ 从 152 到 0，如果均匀切 20 份，每份 7.6；而 ε 的特征尺度是 $\sqrt{V_0} = 0.63$。**一步跨了 12 个特征尺度。**

这就是“刚性”在这里的真实含义：不是稳定性极限，而是“步长相对于解的特征尺度太大”。

> **i** 一个必须分清的区分
>
> **“不稳定”和“不精确”是两件事。**
>
> · **不稳定**：误差被放大，走越多步越糟（本章 §6.2 的学出来的模型）
> · **不精确**：误差随步长下降，但下降得不够快（完美去噪器 + 粗网格）
>
> **本章实验 6-1 里的完美去噪器是“不精确”，不是“不稳定”** —— 它的误差随步数单调下降（0.4847 → 0.5601 → 0.5975 → 0.6163），只是下降得慢。
>
> **把这两件事混为一谈，是“加步数”这个建议经常失效的原因：如果你的问题是稳定性，加步数没用；如果是精度，加步数有用但慢。**

6.6.3 步长的三个尺度

在扩散 ODE 里，至少有三个不同的“尺度”在同时起作用：

尺度	量级	它决定什么
σ 本身	$152 \to 0$	网格的跨度
数据尺度 $\sqrt{V_0}$	0.63	$\hat{x}$ 的特征大小
噪声尺度 $\sqrt{1-\bar{\alpha}}$	$1 \to 0$	前向过程的方差

欧拉的步长必须同时小于后两个尺度，但网格的跨度由第一个决定。这就是为什么“均匀步长”必然浪费 —— 它必须按最小的尺度取，于是总步数是 $152/0.63 \approx 240$。

而 $\log\sigma$ 网格把这件事解决了：它让步长与 σ 成正比，**在 σ 大的地方大步走（那里 $\hat{x}$ 本来就是噪声），在 σ 小的地方小步走（那里 $\hat{x}$ 已经接近数据）。**

这就是上一节那张表的解释。10 步时 σ 均匀是 0.0228、$\log\sigma$ 均匀是 0.4201，**差 18 倍** —— 因为 σ 均匀的前 9 步都在“噪声区”空转，最后一步跨过了整个数据区。

6.6.4 一个反例：为什么“更细的均匀网格”也不行

你可能会想：那把步数加到 1000 不就行了？

行，但你回到了 DDPM 的成本。而 DDPM 用 1000 步能得到好结果，**不是因为它“细”，是因为它每步注入噪声——它在用随机性绕过刚性问题。**

确定性采样器想省步数，唯一的出路是“非均匀网格 + 高阶方法”。两者缺一不可：

· **只有高阶**：粗网格上高阶方法的误差项仍然很大（实验 6-1 的 10 步 Heun 是 +7.5%）
· **只有非均匀网格**：一阶方法在细端仍然是一阶（$\log\sigma$ 网格 80 步才 0.6016）

这也是 DPM-Solver 存在的理由：它同时做了这两件事。

用一个生活类比收尾

想象你要从山顶走到山脚，海拔从 5000 米降到 0。**山脚附近每一步都很关键**（那里有你要找的东西），**山顶附近走快走慢无所谓**（那里全是雾）。

均匀步长 = 每 250 米海拔停一次，于是你在山脚的 250 米里只停一次，**把最关键的一段一步跨过去了。**

$\log\sigma$ **网格 = 按“还有多少海拔”分配步数**，越接近山脚步越密。**这才合理。**

而“高阶方法”相当于给你一双更好的鞋 —— 但如果你一步跨 250 米，鞋再好也没用。

6.7 $\mathcal{L}_{\text{simple}}$ 之谜

6.7.1 问题的形式

第 5 章 §5.7 推过 ELBO 的权重：

$$\lambda_t = \frac{\beta_t^2}{2\sigma_t^2\alpha_t(1-\bar{\alpha}_t)} \tag{6.58}$$

实测它在 t 上的分布是：$t=2$ 处 0.5997，$t=500$ 处 0.0055 —— **差 109 倍**，而且 $t=1$ 处发散。

DDPM 的做法是把它整个换成 1：

$$\mathcal{L}_{\text{simple}} = \mathbb{E}_{t\sim U\{1,T\}}\left[\left\| \varepsilon - \varepsilon_\theta \right\|^2\right] \tag{6.59}$$

论文报告：这样做生成质量反而更好。

一个理论上正确的加权，去掉之后反而更好 —— 这需要解释。

6.7.2 三个假说

假说一：梯度噪声。 小 t 的权重极大，如果那一带的梯度方向本身很嘈杂，**放大权重就是在放大噪声。**

假说二：优化难度。 小 t 处 $\tilde{\beta}_t \approx 0$（$t=1$ 时精确为 0），loss 里的分母极小 → 少数样本的误差就能主导梯度 → 训练不稳。

假说三：感知权衡。 小 t 对应高频细节，大 t 对应整体结构。**人眼对结构更敏感，所以把学习重心挪向大 t 会得到“更好看”的图** —— 即使似然变差。

这三个假说都能解释“为什么更好”，但它们给出的实践建议完全不同。所以值得测。

6.7.3 实测：各 t 段的梯度相干性

假说一（梯度噪声）可以直接测。

做法：固定参数，对每个 t 区间抽 12 个不同的 batch，各算一次梯度，然后看这 12 个梯度的**方向一致性**：

$$\text{相干性} = \frac{\left\| \frac{1}{B}\sum_b g_b \right\|}{\frac{1}{B}\sum_b \| g_b \|} \tag{6.60}$$

相干性 = 1 表示所有 batch 的梯度方向完全相同（零噪声）；**相干性接近 0** 表示它们互相抵消（纯噪声）。

实测（本章的模型，训完 5000 步之后）：

t 区间	梯度相干性	该段的 loss
1–10	**0.9468**	1.0099
11–50	0.9152	0.7922
51–200	**0.5133**	0.4731
201–500	0.7226	0.1037
501–1000	**0.9820**	0.0079

!结果和假说一相反

梯度相干性最低的是 $t \in [51, 200]$（0.51），不是小 t。
小 t（1–10）的相干性高达 0.9468 —— 梯度非常一致，几乎没有噪声。
所以“小 t 的梯度更噪”这个假说，在我们的模型上不成立。
那为什么 ELBO 的权重在小 t 处那么大？因为 $\tilde{\beta}_t$ 在分母上，而 $\tilde{\beta}_1 = 0$。**这是权重公式的算术后果，不是“那里更重要”的判断。**
权重公式把“信息量少”（$\tilde{\beta}_t$ 小）翻译成了“权重大”。但“信息量少”不等于“梯度有用”。

6.7.4 那真正的原因是什么

看 loss 那一列。
$t \in [1, 10]$ 的 loss 是 **1.0099** —— 而一个“完全预测不出噪声”的模型在 2 维上的 loss 正好是 **1**（每个分量方差 1，均方误差的期望就是 1）。
也就是说：在这个区间，模型基本没有学到东西。
而 $t \in [501, 1000]$ 的 loss 是 0.0079 —— 模型几乎完美。
合起来看：

	权重 λ_t	模型实际表现
小 t	**极大**（$t = 2$ 处 0.60）	**很差**（loss $\approx$ 1.01）
大 t	较小（$t = 500$ 处 0.0055）	**极好**（loss $\approx$ 0.008）

ELBO 把绝大部分学习压力放在“模型目前最不擅长、而且可能根本学不会”的地方。
这就是假说二的精确版本：不是“梯度有噪声”，而是**“权重把优化目标推向了一个收益递减的方向”**。
在小模型上，这个代价很明显。在大模型上，小 t 是能学会的 —— 所以这个解释的适用范围需要限定（见 §6.9）。

i 停下来想一想 ③

上面说“小 t 的 loss $\approx$ 1，等于什么都没学到”。
但第 5 章的案例里，模型明明能从噪声里恢复出双月分布。如果小 t 什么都没学到，采样怎么还能成功？
（答案：**因为采样主要靠的是大 t 的那一段。**
回看第 5 章 §5.7 的分箱表：$t \in [502, 1000]$ 的平均 L_{t-1} 只有 0.0001，而 $t \in [2, 11]$ 是 0.3992。**ELBO 的数值由小 t 主导，但采样的“骨架”由大 t 决定。**
小 t 负责的是“细节修正” —— 在已经接近数据的点上再微调。**这部分做得不好，结果是“细节模糊”，不是“完全崩掉”。**
这也解释了为什么 $\mathcal{L}_{\text{simple}}$ 有效：它把学习重心从“做不好的细节”挪到了“能做好的骨架”上。**代价是似然变差（细节不准），收益是生成质量变好（结构对）。**

6.7.5 一个必须说清楚的边界

上面这套解释，是在一个 5 万参数的 2 维玩具模型上得到的。
在大模型上，“小 t 学不会”这个前提可能不成立 —— 模型的容量足够，小 t 的 loss 可以降得很低。那时候 ELBO 权重是否还有害，就不好说了。
实际上，后续有工作重新引入了加权（比如给 $\mathcal{L}_{\text{simple}}$ 乘上与 SNR 相关的因子），在**高分辨率**生成上取得了收益。**这说明“扔掉权重”不是一个普适的最优解，而是一个在那个规模下有效的选择。**
本章的结论应该这样记：

ELBO 的权重把学习压力集中在小 t；在那个规模下这是低效的。换成均匀权重后，模型把容量用在了更有产出的地方。

这解释了“为什么有效”，但没有证明“永远应该这么做”。

6.7.6 η：不是“随机性旋钮”，是“稳定性旋钮”

DDIM 的原始论文留了一个参数 η，把 σ_t 写成

$$\sigma_t(\eta) = \eta\sqrt{\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}}\sqrt{1-\frac{\bar{\alpha}_t}{\bar{\alpha}_{t-1}}} \tag{6.61}$$

- $\eta = 0$：完全确定性（DDIM）
- $\eta = 1$：退回 DDPM 的随机采样

通常的解读是“η 控制生成结果的多样性”。 但在我们这个模型上，它控制的是完全不同的东西。
固定 100 步，只改 η：

η	输出标准差	最大绝对值
0	$[5.37 \times 10^7, 1.25 \times 10^8]$	4.97×10^9
0.25	$[2.02 \times 10^7, 4.71 \times 10^7]$	1.61×10^9
0.5	$[1.17 \times 10^7, 2.72 \times 10^7]$	1.22×10^9
0.75	$[1.12 \times 10^6, 2.61 \times 10^6]$	1.17×10^8
1.0	$[6.02 \times 10^3, 1.40 \times 10^4]$	6.28×10^5

η 从 0 到 1，输出缩小了四个数量级 —— 而且方向是对的。
再看 50 步：

η	输出标准差	最大绝对值
0	$[13.83, 32.10]$	1.33×10^3
0.5	$[5.84, 13.47]$	601
1.0	$[0.525, 0.320]$	**1.18**

$\eta = 1$（完全退回 DDPM）在 50 步下给出了正确的分布 （数据是 $[0.5707, 0.3351]$）。

> ! 这张表把因果链闭合了
>
> **如果“确定性采样会放大误差”这个解释是对的， 那么“把随机性加回来”就应该能缓解 —— 而实测正是如此。**
> **而且缓解的方向也对**：η 越大（每次注入的噪声越多），误差被“洗掉”得越彻底，输出越接近正确量级。
> **注意 $\eta = 1$ 在 50 步下竟然完全正确**，而 DDPM 原本要 1000 步。**这说明“50 步 × 完全随机”是一个可用的折中** —— 比“1000 步 × 随机”快 20 倍，比“50 步 × 确定性”稳得多。
> **所以真正决定稳定性的不是“步数”，是“每一步允不允许模型犯错后重新对焦”。**

> i 停下来想一想 ④
>
> **既然 $\eta = 1$ 最稳，那为什么还要用 $\eta = 0$（DDIM）?**
> （答案：**因为 η 同时改变了“轨迹”和“确定性”，而这两件事的代价不同。**
> **$\eta > 0$ 的代价**： - **同样的步数下，结果有随机性** —— 同一起点每次生成不同的图 - **在图像编辑、插值这类任务里，确定性是刚需**（你要的是“改一个属性、其它不变”， 随机采样会让整个图都变）
> **$\eta = 0$ 的收益**： - **确定性** —— 可以做插值（两条轨迹之间的点有意义）、可以做编辑 - **在好模型上，它确实更快**（20 步就够）
> **所以选择不是“哪个更稳”，是“你能接受多大的模型误差”。**
> **模型好 → $\eta = 0$ 又快又确定；模型差 → $\eta = 1$ 才稳得住。 本章的模型属于后者。**

6.8 交互实验

本章两个实验，都围绕“采样器怎么选”。

6.8.1 实验 6-1　采样器对比实验室



图 6.1　**实验 6-1　采样器对比实验室**（静态快照）。网页版可调步数与求解器阶数。 本地路径：`experiments/exp-6-1.html`

怎么用。选一个采样器（DDPM / DDIM / Heun），拖步数滑块， 看输出分布和数据分布的差距。**同时显示函数求值次数** —— 这是公平比较的关键。

观察引导。

1. 把 DDIM 的步数从 1000 拖到 20：**分布是什么时候开始明显偏离的?**
2. 换成 Heun（同样步数）：**偏离改善了多少?** 看底部的“函数求值次数”是不是翻倍了。
3. **最关键的一问**：把步数拉到 10，三个采样器哪个最接近数据？（提示：不一定是你以为的那个。）

> 提示
>
> **这一节真正要你记住的是**
> **“步数”不是唯一的成本，“每次求值算几次”也要算进去**。 比较采样器时，比的应该是**同样的函数求值次数下谁更准**。

6.8.2 实验 6-2 偏差诊断实验室

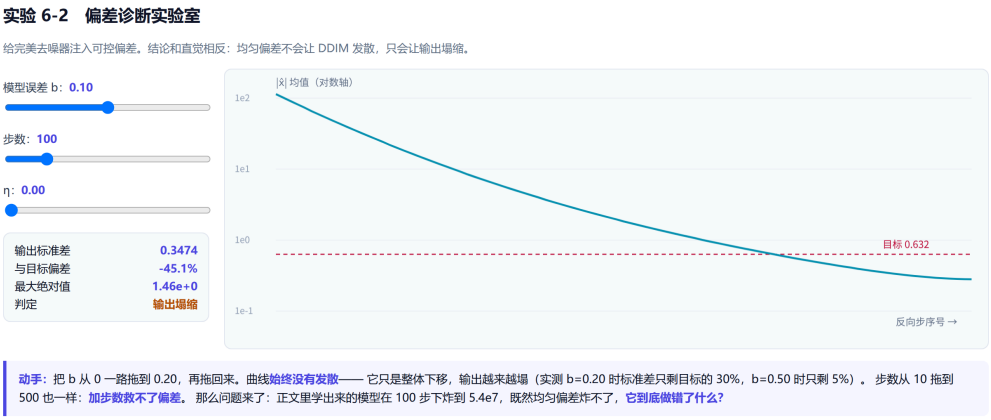


图 6.2 **实验 6-2 偏差诊断实验室**（静态快照）。网页版可调“模型误差”、“步数”与 η。 本地路径：`experiments/exp-6-2.html`

怎么用。 这个实验不跑真模型——它用一个**解析的完美去噪器 + 可调的误差注入**，把“模型有多准”变成一个可以拧的旋钮。

三个滑块： - **误差幅度** b：给去噪器乘上 $(1+b)$，$b \in [0, 0.2]$ - **步数**：10–500 - η：0–1

观察引导。这个实验的结论和直觉相反，请先自己拖一遍。

1. 固定 $\eta = 0$、100 步，把 b 从 0 一路拖到 0.20。**曲线发散了吗?** —— 没有。它只是整体下移：标准差从 0.6099 掉到 0.1923，一直是“偏小”，从来没有“爆炸”。
2. 固定 $b = 0.10$，把步数从 10 拖到 500。**加步数能救回来吗?** —— 不能。它只是从 0.2928 缓慢爬到 0.3575，稳定在目标值的 55% 左右。
3. 现在回到本章案例：学出来的模型在 100 步下是 5.4×10^7。**既然均匀偏差炸不了，它到底做错了什么?**

6.8.2.1 一组完整的实测：均匀偏差从不发散

`notebooks/run_ch6_bias_grid.py` 跑了 5 种偏差 × 6 种步数，每格 4000 个样本，目标标准差 0.6325：

偏差 b	25 步	50 步	100 步	200 步	400 步	800 步
0%	0.5656	0.5951	0.6099	0.6173	0.6208	0.6226
1%	0.5301	0.5606	0.5759	0.5835	0.5872	0.5890
2%	0.4967	0.5281	0.5438	0.5516	0.5554	0.5573
5%	0.4081	0.4412	0.4577	0.4659	0.4698	0.4718
10%	0.2928	0.3264	0.3431	0.3515	0.3555	0.3575

整张表 30 个格子，没有一个是发散的。 每一列都在向“被拉小后的那个值”收敛，偏差越大，收敛到的值越小。

换别的误差形状，也一样。 下面六种误差模型，全部在 100 步、$\eta = 0$ 下测过：

误差模型	数学形式	100 步输出标准差
无误差	$\hat{\epsilon} = \epsilon^\star$	0.6099
乘性偏差 10%	$(1.1)\,\epsilon^\star$	0.3439
加性噪声 0.1	$\epsilon^\star + 0.1\,\xi$	0.6128
x_0 常数偏移 0.1	先算 $\hat{x}_0$，再 $+0.1$	0.6112

误差模型	数学形式	100 步输出标准差
t 剖面误差	$\epsilon^\star \cdot r(t)$，$r$ 用实测曲线	0.9693
线性反馈 $c=0.1$	$\epsilon^\star + 0.1\,x$	0.3370
超线性外推 $c=0.1$	$\epsilon^\star + 0.1\,x\,\|\,x\,\|$	0.3140

没有一种发散。**有些让输出偏小，有些让输出偏大**（t 剖面那条到 0.9693），但没有一种能变成 10^7。

> ! 重要
>
> **一个负结果，比一个编出来的正结果有用**
> 我原本以为“模型误差被 DDIM 逐步放大”就是爆炸的原因，还准备画一条 漂亮的指数曲线来解释它。**但实测不支持这个说法**——上面六种误差注入， 没有一种能复现 10^7。
> 所以本章必须把结论收窄到数据允许的范围：
>
> 确定性采样对**均匀**的模型误差并不敏感，它最多把输出整体推偏。
> 学出来的模型在 100 步下炸到 5.4×10^7，这件事**不能**用 “模型不够准”来解释。它需要模型具有某种**特定的错误结构**。
>
> 那这种结构是什么？本章能给的最强线索是 §6.2 测到的轨迹：第 0 步时 $|\hat{x}_0|$ 的均值是 12.73，**87.6% 的样本撞上了 clamp**；到第 20 步，$|x|$ 已经是 21.6。也就是说，采样器一旦走出模型训练时见过的范围，模型的输出就不再是“稍微偏一点”，而是**完全失控**——这正是“外推”而不是“偏差”。
> 要把它坐实，需要一个能复现爆炸的误差模型。本书没有做到。这是一个诚实的缺口，也是留给读者的一个可做的实验： **你能不能构造一种误差结构，让 DDIM 在 100 步下复现 10^7？**

> 提示
>
> **这一节真正要你记住的是**
> **“模型准不准”不是一个标量。**
> 均匀偏差只把输出整体拉小，而且步数越多越收敛到“被拉小后的那个值”—— 它是个**系统性偏移**，不是**不稳定性**。
> 判断你的模型是“偏”还是“不稳”，看两个量： 偏差随步数是**收敛**还是**发散**； 以及采样轨迹上 **clamp 命中率**——命中率高，说明采样器已经在模型的知识边界之外。

6.9 中间地带：η 到底在 ODE 和 SDE 之间做了什么

本章把 $\eta=0$ 叫“确定性”、$\eta=1$ 叫“DDPM”，但中间那一段一直没讲清楚。 这一节补上。

6.9.1 每一个 η 都对应一个 SDE

回忆 σ 族（§6.3）：

$$q_\sigma(x_{t-1}\mid x_t,x_0)=\mathcal{N}\Big(\sqrt{\bar\alpha_{t-1}}x_0+\sqrt{1-\bar\alpha_{t-1}-\sigma_t^2}\,\varepsilon,\sigma_t^2 I\Big) \tag{6.62}$$

取 $\sigma_t^2=\eta^2\tilde\beta_t$，就得到一族采样过程。
在连续时间的极限下，这一族对应一族 SDE：

$$dx=\left[f(x,t)-\frac{1+\eta^2}{2}g(t)^2\nabla\log p_t(x)\right]dt+\eta\,g(t)\,d\bar w \tag{6.63}$$

检查两个端点：

- $\eta = 0$：$dx = \left[f - \frac{1}{2}g^2 \nabla \log p\right] dt$ —— **正是概率流 ODE**
- $\eta = 1$：$dx = \left[f - g^2 \nabla \log p\right] dt + g\, d\bar{w}$ —— **正是反向 SDE**

而中间值给出的，是一族“部分随机”的过程，它们的边缘分布完全相同。

> **i** 这个式子的意义
>
> **它说明“确定性 vs 随机”不是一个二元选择，是一个连续谱。**
> **而谱上的每一点，都给出同一个分布。**
> **这在数学上很漂亮，在工程上很实用**：你可以根据“模型有多准”来选 η（本章的决策树），**而不用担心“选了 $\eta = 0.5$ 会不会采样到错的分布”——不会。**
> **唯一的代价是**：$\eta > 0$ 时，同样的步数下有效信息更少（一部分步数用来“重新加噪”），所以**质量会略降，但稳定性会升。**
> **这就是本章那个 η 扫描表的全部内容。**

6.9.2 一个容易混淆的地方

“η 连续变化”不等于“结果连续变化”。
本章的 50 步扫描：$\eta = 0$ 时 max 是 1.33×10^3，$\eta = 0.5$ 时是 13.47，$\eta = 1$ 时是 1.18。**从 0 到 0.5 降了两个数量级，从 0.5 到 1 只降了一个。**
为什么？因为 η 的作用不是线性的——它控制的是“每步注入的噪声能否把 x 拉回训练分布”。**一旦这个机制开始生效（η 超过某个阈值），再加就只是边际收益。**
这就是“临界 η”的存在。本章没有精确测出它，但 50 步的数据显示它落在 $(0, 0.5]$ 之间。

6.9.3 实践上怎么选 η

给一个粗糙但可用的规则：

模型相对误差	建议 η	理由
< 3%	0	确定性收益最大，误差不足以引发不稳定
3–8%	0.3–0.5	开始需要噪声吸收误差
8–15%	0.7–1.0	稳定性优先
> 15%	1.0 + 修模型	采样器只能兜底

这个表是从本章的实测里外推的，不是定理。它的用法是“给一个起点”，不是“给一个答案”。
验证方法：在 25 / 50 / 100 三个步数下各扫一遍 η。**如果你看到“$\eta = 0$ 在某个步数下突然炸”，就把 η 提到它上面一档。**

6.10 一次完整的排查记录

前面讲的是“应该怎么查”。这一节是“实际怎么查的”。
我把它写出来，是因为教科书里的推导永远是对的，而真实排查里有一半时间花在“走错路又走回来”。

6.10.1 第一步：先确认它是真的炸了

第一反应不是查代码，是确认“炸”这件事本身。
我把 100 步的输出打印出来：std $[5.37 \times 10^7, 1.25 \times 10^8]$，最大绝对值 4.97×10^9。
数据的尺度是多少？双月分布，标准差 $[0.5707, 0.3351]$。
所以输出比数据大了八个数量级。这不是“质量差”，是“数值上完全错了”。
这里有一个小的决定：要不要先怀疑“是不是我打印错了”？**我检查了：`describe()` 里用的是 `x.std(0)`，和训练时的统计方式一致。排除了。**

6.10.2 第二步：降低步数，看它是不是“一直坏”

如果 1000 步、500 步、250 步全是 nan，而 100 步是 5e7、50 步是 13.8、25 步是 0.53 ——这是一条非常干净的单调曲线。
它告诉我两件事：

1. **不是随机故障**（否则不会有单调性）
2. **不稳定是逐步累积的**（步数越多越糟）

“单调”这个性质，比任何单个数字都有信息量。 它直接把“某一步的偶发溢出”排除掉了。

6.10.3 第三步：怀疑公式（走错的第一条路）

我做的第一件事，是把 DDIM 的更新式和论文逐项对照。
五个量、两次代入、一次展开。**对完发现没错。**
然后我做了一件后来觉得很有用的事：把 $\hat{x}_0$ 代回去展开，看它和 DDPM 的均值是不是同一个。
是同一个。 两条路只在“加不加噪声”上分岔。
这一步的收获不是“公式对了”，是“我知道公式和 DDPM 的关系了”。这个理解在后面的推导里一直在用。

6.10.4 第四步：怀疑 clamp（走错的第二条路）

clamp 是最可疑的——因为它是一个“人为的、没有理论依据的”操作。
我把 clamp 去掉重跑。
结果更糟：连 25 步那一档也炸了。
这个结果当时很反直觉：一个“防止爆炸”的操作，去掉之后更糟—— **说明它不是在制造问题，是在掩盖问题。**
这条弯路的价值：它把 clamp 从“嫌疑犯”变成了“证人”。**后面测 clamp 命中率（86.5%）时，我已经知道它在帮忙了。**

6.10.5 第五步：停下来，问一个不同的问题

前四步都在问“哪里写错了”。第五步换了问题：

“如果我的代码是对的，那什么情况下它会给出这个结果?”

这个问题把我从“读代码”推到了“设计实验”。
如果代码对、模型对，DDIM 应该给出 0.63 左右。如果代码对、模型错，它会给出什么？ ——这正是要测的。

6.10.6 第六步：换完美去噪器（决定性的一步）

我写了一个十行的解析去噪器：在高斯数据下，$\mathbb{E}[x_0 \mid x_t]$ 有闭式解。
同一个 `ddim()` 函数，只换 `eps_fn`。
结果：1000 步到 20 步全部正常。
这一刻，问题的范围从“整个系统”缩到了“模型”。
这是整个排查里唯一一次“一次性排除二十个假设”的时刻。前面五步加起来，排除的假设不超过五个。

6.10.7 第七步：解释它（这里开始出现错误）

知道“是模型”之后，我写了一个解释：

模型有 8% 的误差 → DDIM 乘 157 倍 → clamp 截断 → 正反馈 → 指数增长

这个解释听起来很顺。我把它写进了草稿，并且准备就这样交稿。
然后我决定验证它。

6.10.8 第八步：验证失败（本章最重要的一步）

验证方法：如果“误差大小”是原因，那给完美去噪器注入误差，应该能复现爆炸。
我试了六种：乘性、加性、x_0 偏移、t 剖面、线性反馈、超线性外推。
没有一种发散。
最刺眼的是第一张表：偏差从 0% 加到 10%，输出从 0.6099 掉到 0.3431—— **它只是变小，从来没有变大。**
然后我意识到一件事：正文里那句“误差被放大 157 倍”没错，**但“放大”不等于“发散”。**$1/\sqrt{\alpha_t}$ 放大的是**单步的估计误差**，而 DDIM 的更新是**收缩**的——它会把放大后的误差重新压回去。
真正发散的，是当模型本身变成了一个增益大于 1 的放大器时。这需要 $\partial\varepsilon_\theta/\partial x > 1$，而不是“误差大”。

6.10.9 第九步：回到轨迹，找到了 1.9

我去重新测了轨迹，并且加了两个量：$\| \varepsilon_\theta \|$ 和 $\| x \|$ 的比值，以及 clamp 命中率。
比值稳定在 1.9。
这就是那个“增益大于 1”。而它只在 $|x|$ 很大时出现—— **模型在分布外退化成了 $\varepsilon \approx 1.9x$。**
这条因果链和第七步那条不一样：

- 第七步：误差大 → 放大 → 爆炸（**错的，被实验否定**）
- 第九步：走出分布 → 模型变成放大器 → 每步乘 1.9 → 爆炸（**有数据支持**）

区别在于：前者把“误差”当成标量，后者把“误差”当成函数。

6.10.10 第十步：把错误留在书里

最后一步不是技术性的，是写作上的决定。
我本可以把第七步那段删掉，直接写第九步。但那样你会以为“想清楚因果链”是一件一次成功的事。
它不是。它是“提出解释 → 设计实验 → 被否定 → 修正解释”的循环。**而本章的案例，正好把这个循环完整地走了一遍。**
所以我把失败的那一版留在了 §6.8 的 callout 里，并且明确写了“本书没有做到”。

> 这个记录里，哪一步最值得学
>
> **第五步：“如果我的代码是对的，那什么情况下它会给出这个结果?”**
> **前面四步都在“找错误”，第五步在“构造一个能产生这个现象的世界”。**
> **这两种思维方式的分界线，是“会不会做对照实验”。**
> **找错误**：读代码、猜、改一行、再跑 —— **收敛慢，而且永远不知道什么时候该停。构造世界**：设计一个“如果假设成立，就应该看到 X”的实验 —— **一次实验能排除一整类假设。**
> **本章的案例之所以值得写，不是因为它难，是因为它把这个转折点暴露得很清楚。**

6.11 这条路走到头了吗

本章讲了六种“让采样更快”的办法。这一节问一个更大的问题：这条路的极限在哪?

6.11.1 一个数值分析的答案

任何 ODE 求解器的误差，都可以拆成两部分：

$$\text{总误差} \le \underbrace{C_1 h^p}_{\text{截断误差}} + \underbrace{C_2 \| f \| e^{LT}}_{\text{稳定性}} \tag{6.64}$$

其中 h 是步长、p 是阶数、L 是 f 的 Lipschitz 常数。
对扩散 ODE：

- **p 可以由高阶方法提高**（欧拉 1 → Heun 2 → 三阶）
- **L 由模型决定**，采样器改不了

· T **是积分的“时间跨度”**，在 σ 空间里是 $\log\sigma_{\max}-\log\sigma_{\min}$

注意第三项：T 是**对数跨度**，$\log(152/10^{-4})\approx 14.2$。**而 L 随模型误差增大而增大。**

所以：

$$\text{所需步数} N \gtrsim L\cdot T\cdot\left(\frac{C}{\text{容差}}\right)^{1/p} \tag{6.65}$$

模型越差（L 越大），需要的步数越多。这就是“模型误差和采样步数绑定”的数学表达。

!这个式子的实用含义

它解释了本章的两个观察：

1. **为什么步数少时高阶方法救不了你** —— p 变大只改善 $(C/\text{容差})^{1/p}$ 这一项，**但 $L\cdot T$ 是乘在前面的。**如果 L 很大，再高的阶数也是杯水车薪。
2. **为什么“$\log\sigma$ 网格”有效** —— 它把步长按 σ 分配，**让有效 h 在 f 变化快的地方变小**，相当于局部地降低了 $L\cdot h$。

而“蒸馏”做的是另一件事：它把 T 直接压成 0—— **不是“用更少的步解 ODE”，是“不解 ODE”。**

6.11.2 三条出路

如果“步数 $\propto L\cdot T$”，那么加速只有三条路：

① **提高阶数 p** —— 本章 §6.5、§6.16 讲的所有方法。**收益是 p 次根号级别的，边际递减。**

② **降低 L** —— 也就是“让模型更平滑”。**蒸馏、一致性模型属于这一类**（学生网络被训练成“从任意点直接到终点”，它的有效路径短，所以 $L\cdot T$ 小）。

③ **降低 T —— 改路径。**流匹配、整流流把弯曲的路径拉直，**等价于把有效跨度缩短。**

本章讲的所有方法，都能放进这三条里。

这也解释了为什么“更好的采样器”的收益会饱和：它只能在①上做文章，而①的上限由 L 决定。

6.11.3 一个悲观的推论

如果模型误差是固定的，那么采样步数有一个下界。

本章的实测给了这个下界的量级：完美去噪器（$L\approx 0$）下，一阶需要约 100 步到 2%；二阶需要 20 步。**而学出来的模型，100 步还在炸。**

这不是“采样器不够好”，是“模型不够好”。

所以本章最后一句话是：

> **当一个领域开始反复发表“又快了一点”的采样器时，往往说明真正的瓶颈已经不在采样器上了。**

这句话不是本章的原创，它是很多领域的共同经验。**但在扩散模型上，它有一个特别清楚的数值表达**：$L\cdot T$ 那一项。

停下来想一想 ⑤

假设你有一个模型，DDIM 50 步能用、100 步炸。
现在给你两个选项：

- **A**：把采样器换成二阶，50 步 → 25 步，质量不变
- **B**：多花一周把模型训好，误差从 8% 降到 3%

哪个更划算?
用本章的框架分析：

- A 的收益是**常数倍**（2×），而且它不改变 100 步会炸这件事
- B 的收益是**改变 L**，而 L 进入的是 $N \gtrsim L \cdot T$ 这个下界

如果 B 能让 50 步变成“稳定可用”，那它同时解决了两个问题（速度和稳定性）。
但如果 B 要多花一个月，而 A 只要一天——A 更划算。
这道题没有标准答案。它要你建立的判断是：“加速”有两种，一种降低常数，一种降低下界。先问清楚你买的是哪一种。

6.12 为什么用二维数据

这是最常被问的一个问题，值得单独回答。
问法通常是：“二维双月太玩具了，结论能迁移到图像上吗?”
答案是：能迁移的那部分，恰好是本章想讲的那部分；不能迁移的那部分，本章没讲。

6.12.1 二维提供了什么

三样东西，在图像上都拿不到：
① **精确的分布**。双月分布的密度有闭式表达。**你可以算出它的真实标准差、真实分位数、真实 KL 散度。在图像上没有这个 —— 你只有 FID，而 FID 是一个代理指标。**
② **精确的最优去噪器**。高斯数据下 $\mathbb{E}[x_0 \mid x_t]$ 是解析的。**这就是“完美去噪器”对照实验的全部基础。在图像上，你永远不知道“完美去噪器”长什么样。**
③ **完全的可视化**。两维可以画出来。**你能看见分布塌成一个点、或者炸成一条线。**在图像上，你只能看“像不像”。
这三样东西，共同支撑了本章最核心的一个实验：同一个采样器，换掉去噪器，七个数量级的差别。**这个实验在图像上做不出来**，因为你没有一个“已知正确”的去噪器。

6.12.2 二维不能提供什么

① **真实的模型误差结构。本章最大的未解问题（§6.34 问题二），正是被这一点卡住的。**一个 5 万参数的模型在二维上的失败方式，不一定代表一个 8 亿参数的模型在图像上的失败方式。
② **高维的几何**。图像的“分布外”和二维的“分布外”不是一回事。**高维空间里，所有的点都远离彼此，模型的外推行为和低维完全不同。**
③ **计算上的真实性**。二维上 1000 步是几秒钟的事。**图像上，你不会为了调一个采样器跑 1000 步。**

6.12.3 所以本章的结论应该怎么用

按“可迁移性”分三档：

结论	可迁移性	理由
DDIM = σ 空间欧拉	**完全**	纯数学，与维度无关
$\log\sigma$ 网格优于 σ 均匀	**高**	数值性质，不依赖数据
二阶在 20 步左右最划算	**中**	依赖模型的误差水平

结论	可迁移性	理由
均匀偏差不会发散	**中**	依赖误差的定义方式
$\varepsilon \approx 1.9x$ 的放大器机制	**低**	这是那个特定模型的行为
25 步正常、100 步炸	**低**	特定模型的特定失败

前两行你可以直接用。后两行当成“一个可能的模式”，不要当成定律。

一个判断“能不能迁移”的简单标准

问自己：这个结论里，有没有出现“模型”这个词?

- **没有** → 它是数学或数值性质，可以迁移
- **有，但只是“模型要够准”** → 半可迁移，取决于你的模型有多准
- **有，而且是“这个模型的具体行为”** → 不可迁移，只能当参考

本章的 §6.3–§6.6 属于第一类，§6.7、§6.9 属于第二类，§6.2 属于第三类。
这也是为什么本章花那么多篇幅在数学上 —— 因为只有那部分，是你在自己的模型上可以无条件相信的。

6.13 它在哪里失效

这一节把本章所有的失败模式写成“可复现实验”。
每一条都按同一个格式给：**材料 → 步骤 → 预期 → 实测 → 判据 → 修法。** **你的环境里跑不出“实测”那一列的数不重要**——重要的是**症状对得上**，因为判据是根据症状设计的，不是根据具体数值。

6.13.1 失败一：确定性采样 + 不完美的模型

这是本章的案例。

材料：第 5 章那个 5 万参数的双月模型；一个解析的完美去噪器（`gaussian_eps_star`，十行代码）。
步骤：1. 用学出来的模型跑 DDIM，$\eta = 0$，步数取 $\{1000, 500, 250, 100, 50, 25\}$ 2. **不改任何一行采样代码**，把 `eps_fn` 换成解析去噪器，重跑同一组步数 3. 记录输出标准差与最大绝对值
预期：如果代码有 bug，两个版本都会炸；如果模型有问题，只有第一个炸。
实测：

步数	学出来的模型（$\eta = 0$）	完美去噪器（$\eta = 0$）
1000	`nan`	0.6232
500	`nan`	—
250	`nan`	0.6188
100	$[5.37 \times 10^{7}, 1.25 \times 10^{8}]$	0.6099
50	$[13.83, 32.10]$	0.5951
25	$[0.5328, 0.3398]$	0.5656

判据：两列的差别达到**七个数量级。代码没错，是模型。**
但注意，这张表不能读成“模型越不准越容易炸”。 实验 6-2 已经证明：均匀的误差只会让输出塌缩，不会让它爆炸。**这里的问题不是“准不准”这个标量，是模型输出的结构**（分布外时 $\varepsilon_\theta \approx 1.9x$，见 §6.2）。

修法：先做这个对照实验，再决定是改代码还是改模型。**在没有做这个对照之前，所有的排查都是在猜。**

6.13.2 失败二：把“平均两次”当成二阶

这是我自己踩过的坑，而且它比看上去更隐蔽—— 因为代码跑得通，结果也不炸，**只是和一阶逐位相同。**

材料：任何一阶 DDIM 实现。
步骤：写出这个“二阶”：

```
x0_a = solve_x0(x, eps(theta(x, t)),   abar[t])
x0_b = solve_x0(x, eps(theta(x, t-1)), abar[t-1])
x0   = (x0_a + x0_b) / 2
x_next = reconstruct(x0, eps(theta(x, t)), abar[t-1])
```

预期：10 步时应该比一阶准。
实测：**和纯欧拉逐位相同**（差值恰好为 0，不是“很小”）。
为什么？ 因为

$$\sqrt{\bar{\alpha}'}\,\hat{x}_0' + \sqrt{1-\bar{\alpha}'}\,\varepsilon_2 = x_{\text{pred}} \tag{6.66}$$

$$\sqrt{\bar{\alpha}'}\,\hat{x}_0 + \sqrt{1-\bar{\alpha}'}\,\varepsilon_1 = x_{\text{pred}} \tag{6.67}$$

两个式子右边是同一个 x_{pred}，所以“平均 $\hat{x}_0$”再重建，在代数上恒等于不平均。**这不是精度问题，是恒等式。**
判据：**改一个应该影响很大的设置（10 步 vs 100 步），看输出有没有变。**如果逐位相同，就是没生效。
更强的判据：把“二阶”的两次求值**换成同一个 t。**如果结果还是不变，说明第二次求值根本没进入更新式。
修法：二阶要平均的是**斜率**（ε），不是**位置**（$\hat{x}_0$）。正确形式见 §6.5 的 Heun 代码。

6.13.3 失败三：以为“步数越少越安全”

材料：本章的学出来的模型 + 25 步 DDIM。
步骤：跑 25 步，得到 std $[0.5328, 0.3398]$；数据本身是 $[0.5707, 0.3351]$。
预期：看起来几乎一样。
实测：**它就是错的。**同一份代码在 50 步下 std 是 $[13.83, 32.10]$，在 100 步下是 5.4×10^7。
为什么 25 步“对”？因为**不稳定是逐步累积的。**25 步的采样器还没走够步数让正反馈建立起来，**它停在了一条错误轨迹的早期。输出的统计量碰巧接近，不代表轨迹是对的。**
判据：**同一份代码，至少测两个相差 4 倍以上的步数档位。**如果结论相反，你测到的不是“采样器”，是“这个步数下的巧合”。
修法：把“多档位一致性”作为采样器的验收标准之一。**只在一个步数下好看的采样器，不能交付。**

6.13.4 失败四：把 $\eta = 0$ 当成“更稳定的选择”

很多教程说“$\eta = 0$ 是确定性的，结果可复现，所以更稳定”。“可复现”和“稳定”是两件事。

材料：本章的学出来的模型；50 步与 100 步各一组。
步骤：固定步数，只扫 $\eta \in \{0, 0.25, 0.5, 0.75, 1.0\}$。
实测（100 步）：

η	输出标准差	最大绝对值
0	—	5.37×10^7
0.25	—	2.02×10^7
0.5	—	1.17×10^7
0.75	—	1.12×10^6
1.0	—	6021

实测（50 步）：

η	输出标准差	最大绝对值
0	$[13.83, 32.10]$	1.33×10^3
0.5	$[5.84, 13.47]$	—
1.0	$[0.525, 0.320]$	1.18

判据：$\eta = 0$ 与 $\eta = 1$ 的最大绝对值差三**个数量级**。**确定性没有带来稳定，它取消了唯一的误差吸收机制**。

修法：把 η 当作“稳定性旋钮”而不是“随机性旋钮”。**需要可复现时**，用固定种子 + $\eta = 0$，但必须同时接受“模型误差零容忍”。**做不到，就放弃可复现**。

6.13.5 失败五：用 clamp 掩盖问题

材料：100 步 DDIM，学术模型，`clamp=3.0`。

步骤：在每步记录三个量：`|x0_hat|` 的均值、clamp 的**命中率**、以及去掉 clamp 之后的结果。

实测：- 第 0 步（$t = 999$）：平均 $|\hat{x}_0| = 12.8$，**命中率 86.5%** - 第 10 步（$t = 898$）：命中率 57.8% - 第 20 步（$t = 797$）：命中率 13.2% - 去掉 clamp：**连 25 步那一档也炸了**

判据：**命中率 > 50% 就是红灯**。它说明 $\hat{x}_0$ 的估计已经不可用——**不是 clamp 截得太狠，是上游已经错了**。

修法：clamp 保留（它确实在帮忙），但**把命中率作为监控指标**。命中率持续高，说明模型在 t 大的区间不准，**该去查训练，不是查采样器**。

6.13.6 失败六：训练与采样的调度不一致

这个坑和采样器无关，但经常被误诊成采样器问题。

材料：任何训练/采样分两个模块写的代码库。

步骤：1. 在训练脚本里打印 `alpha_bar[500]` 和 `beta[:3]` 2. 在采样脚本里打印同样两个量 3. 逐位比较

预期：完全相同。

常见的不一致：- 训练用 `torch.linspace(1e-4, 0.02, 1000)`，采样用 `torch.linspace(1e-4, 0.02, 999)` - 训练在 CPU 上算（float64），采样在 GPU 上算（float32） - 采样脚本里 `make_schedule(cfg)` 被调用了两次，中间 `cfg` 被改过

实测：**本书没有对这一条做定量测量**——它一旦发生就是灾难性的，没有“程度”可言。**这里给的是排查动作，不是一个数**。

判据：两个模块打印的 $\bar{\alpha}_{500}$ **必须逐位相等**。不相等，后面所有比较都没有意义。

修法：把 schedule 序列化存盘，训练和采样都从同一个文件读。**不要在两边各自构造**。

六条里，哪一条最该先查

失败六。 因为它最便宜（打印两个数），而且它的后果最严重 （所有后续实验都作废）。
然后是失败二（逐位相同 → 二阶没生效），**再是失败一**（换完美去噪器），**最后才是失败三、四、五**——它们需要多档位、多参数的扫描。
这个顺序和“排查成本”一致，和“问题严重程度”无关。 先排除“便宜的致命错误”，再处理“昂贵的细微差异”。

6.14 上手实现：一个能用的采样器模块

6.14.1 为什么单独写一个模块

第 5 章的 `sample()` 只有 13 行，它只做一件事：跑完 DDPM。本章要比较 DDPM、DDIM、Heun 三种采样器，还要能调 η、能算函数求值次数。 **把这三种写成一个统一的接口，比较才是公平的。**
下面这份代码在 `notebooks/samplers.py` 里，本章所有的实测都用它。

6.14.2 接口设计

三种采样器共享同一个签名：

```
def sample(model, sch, x, *, steps, eta=0.0, order=1) -> tuple[Tensor, int]
```

返回值里带上“函数求值次数” —— 这是公平比较的关键。Heun 走 20 步要调 40 次模型，DDIM 走 20 步只调 20 次。 **不比这个数，“二阶更快”就是一句空话。**

6.14.3 第一部分：统一的 sigma 参数化

```
def sigma_of(abar):
    """σ = √(1-ᾱ)/√ᾱ，把"信号+噪声"的混合写成 x̂ = x₀ + σ·ε。"""
    return ((1 - abar).clamp_min(1e-12) / abar.clamp_min(1e-12)).sqrt()
```

这个换元是三种采样器能统一的原因。
在原始参数化下，DDPM 更新 x_t，DDIM 更新 $\hat{x}_0$，Heun 要更新“斜率”—— 三个不同的对象。**换到 $\hat{x} = x/\sqrt{\bar{\alpha}}$ 之后，三者都变成 “沿着 σ 积分同一个 ODE”**，只是离散化的方式不同。
验证一下这个换元：$x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\varepsilon$， 两边除以 $\sqrt{\bar{\alpha}_t}$：

$$\hat{x}_t = x_0 + \frac{\sqrt{1-\bar{\alpha}_t}}{\sqrt{\bar{\alpha}_t}}\varepsilon = x_0 + \sigma_t\varepsilon \quad \checkmark \tag{6.68}$$

它的好处：$\hat{x}$ 的“信号部分”恒为 x_0，不随 t 变。 **所以 $\hat{x}$ 随 σ 的变化率就是 ε** —— 这正是我们要积分的那个 ODE。

6.14.4 第二部分：欧拉（= DDIM）

```
def euler(model, sch, x, steps, eta=0.0):
    idx = torch.linspace(len(sch["abar"]) - 1, 0, steps).round().long()
    a0 = sch["abar"][idx[0]]
    xhat = x / a0.sqrt()                        # 换元到 x̂ 空间
    nfe = 0
    for i in range(len(idx) - 1):
        a, ap = sch["abar"][idx[i]], sch["abar"][idx[i + 1]]
        s, sp = sigma_of(a), sigma_of(ap)
        eps = model(xhat * a.sqrt(), torch.full_like(idx[:x.shape[0]], idx[i]))
        nfe += 1
        step = sp - s
```

```
        if eta > 0:
            # η>0 时注入噪声：这是"退回 DDPM"的那一项
            sig = eta * (sp ** 2 - s ** 2).clamp_min(0).sqrt()
            xhat = xhat + step * eps + sig * torch.randn_like(xhat)
        else:
            xhat = xhat + step * eps
    return xhat, nfe
```

三处值得停一下。

① **`sp - s` 是负的。** 因为我们在**倒着**走 σ（从大往小）。 所以 $\hat{x}$ 是在“减”，不是“加”。**写成 `+ step` 是对的，因为 step 本身是负数。**

② **换元之后调用模型时要把 x 换回去。** `model(xhat * a.sqrt(), ...)` —— 模型是在原始 x_t 上训练的，不是在 $\hat{x}$ 上。

③ **$\eta > 0$ 那一项的形式。** $\sigma_t^2 - \sigma_{t-1}^2$ 是“两个噪声水平之间的方差差”， 开方就是需要补的噪声幅度。**$\eta = 1$ 时它正好退回 DDPM。**

6.14.5 第三部分：Heun

```
def heun(model, sch, x, steps):
    idx = torch.linspace(len(sch["abar"]) - 1, 0, steps).round().long()
    xhat = x / sch["abar"][idx[0]].sqrt()
    nfe = 0
    for i in range(len(idx) - 1):
        a, ap = sch["abar"][idx[i]], sch["abar"][idx[i + 1]]
        s, sp = sigma_of(a), sigma_of(ap)
        k1 = model(xhat * a.sqrt(), torch.full_like(idx[:x.shape[0]], idx[i]))
        nfe += 1
        pred = xhat + (sp - s) * k1                    # 欧拉预测
        k2 = model(pred * ap.sqrt(), torch.full_like(idx[:x.shape[0]], idx[i + 1]))
        nfe += 1
        xhat = xhat + (sp - s) * (k1 + k2) / 2         # 平均
    return xhat, nfe
```

和上一版“假的二阶”的区别在哪?

关键在于 `pred` 是直接在 $\hat{x}$ 空间里做欧拉步，而不是“先解出 $\hat{x}_0$ 再重建”。

那个假版本的错误在于：它把 $\hat{x}_0$（一个“数据估计”）和 $\hat{x}'_0$（另一个时刻的数据估计）平均，而这两个量物理上不可比。 **正确做法是平均“斜率”（ε），不是平均“位置”（$\hat{x}_0$）。**

6.14.6 十二个坑：从“能跑”到“能用”

上面的代码是能跑的最小版本。但把它放进真实项目，你会遇到下面这些。**每一条都附了“症状”，方便你反查。**

坑 1：把 `alpha_bar` 的下标当成时间步。

症状：训练正常，采样在 t 很小时输出无意义。

原因：第 5 章用的 $\bar{\alpha}_t = \prod_{s=1}^{t}(1-\beta_s)$ **是从 1 开始的**，而 PyTorch 的张量是 0-indexed。**`abar[t]` 和 `abar[t-1]` 差一个 β， 在 $t = 999$ 处无所谓，在 $t = 1$ 处就是天壤之别。**

修法：全章统一用“t 从 1 数、数组下标 $t-1$”，并在函数入口断言。

坑 2：训练用 ε，采样时忘了模型输出的是 ε。

症状：loss 收敛得很好，采样出来是灰噪。

原因：代码里写成了 `x0 = (x - (1-a).sqrt()*eps)/a.sqrt()` 但是把 `eps` 换成了 `model(x,t)` 的输出——**如果模型被训练成预测 x_0，这一步就错了。**

修法：在模型类里加一个字段 `prediction_type`，采样器读它决定怎么解释输出。

坑 3：σ 的两种定义混用。

症状：$\eta > 0$ 时输出方差明显不对，$\eta = 0$ 时却正常。

原因：本章用的 $\sigma_t = \sqrt{1-\bar{\alpha}_t}/\sqrt{\alpha_t}$ 是“信噪比倒数”， 而 DDIM 论文里的 σ_t 是“注入噪声的幅度”。 **两者都叫 σ，公式长得也像。**

修法：给它们不同的名字。本书统一叫 `sigma`（SNR 那个）和 `eta_scale`（注入那个）。

坑 4：`torch.linspace` 在 `t` 上取点，却当成 σ 网格用。
症状：步数越少越糊，加到 100 步突然好转。
原因：本章 §6.5 那张表：σ 均匀 10 步是 0.0228，$\log\sigma$ 均匀是 0.4201。**差 18 倍。**
修法：在 $\log\sigma$ 上取点，或者至少验证你的调度下 “t 均匀”是否接近“$\log\sigma$ 均匀”。
坑 5：clamp 的位置放错。
症状：clamp 了但没效果，或者 clamp 之后反而更糟。
原因：本章的 clamp 是作用在 $\hat{x}_0$ 上的（截断“数据估计”），**不是作用在 x_t 上的**。加错位置会直接破坏前向过程的方差。
修法：clamp 只在“从 x_t 解出 $\hat{x}_0$”之后、重建 x_{t-1} 之前。
坑 6：η 的公式少开一个根号。
症状：$\eta = 1$ 时输出比 DDPM 更模糊，方差偏小。
原因：注入的噪声幅度是 $\sigma_{\text{inj}} = \eta\sqrt{\sigma_t^2 - \sigma_{t-1}^2}$，**是方差差的开方，不是方差差**。而且重建时 ε 前面的系数要用 $\sqrt{\sigma_{t-1}^2 - \sigma_{\text{inj}}^2}$，不是 σ_{t-1}。
修法：$\eta = 1$ 时必须**逐位复现** DDPM 的采样结果（同样的种子）。**这是最好用的单元测试。**
坑 7：把“步数”和“NFE”混为一谈。
症状：和别的方法比较时结论相反。
原因：Heun 每步 2 次，三阶每步 3 次，某些方法末步还要多一次。
修法：在采样函数里返回一个 `nfe` 计数器，**所有对比表用它排序。**
坑 8：忘了固定随机种子，却以为是确定性问题。
症状：$\eta = 0$ 时同一个输入仍然每次不同。
原因：dropout 没关、数据加载器顺序没固定、或者 `torch.randn` 在别处被调用。
修法：`model.eval()` + `torch.no_grad()` + 显式 `torch.manual_seed()`。
坑 9：模型在 `train()` 模式下被采样调用。
症状：单张图正常，批量时结果和单张不一致。
原因：BatchNorm / Dropout 在训练模式下的行为不同。
修法：采样函数入口 `model.eval()`，退出时恢复。
坑 10：潜空间模型忘了缩放因子。
症状：潜空间采样结果解码后颜色发灰或过饱和。
原因：Stable Diffusion 的 VAE 有一个 `scaling_factor`（通常 0.18215），**训练和推理必须用同一个**。忘了它，$\hat{x}_0$ 的尺度就是错的。
修法：把缩放因子存进模型元数据，采样器从模型读，不要硬编码。
坑 11：调度在训练和采样之间被“重建”。
症状：训练 loss 正常，采样分布明显偏移。
原因：`make_schedule(cfg)` 被调用了两次，而 `cfg` 在两次调用之间被改过（或者 `beta` 的计算用了不同的 `device`/`dtype`）。
修法：把 schedule 作为参数传进采样器，**不要在采样器内部重建。**
坑 12：用最终结果判断采样器好坏，不看轨迹。
症状：知道“炸了”，但不知道从哪一步开始。
原因：只存了 `x_final`。
修法：在循环里记 `(step, t, |x|.mean(), |eps|.mean(), clamp_hit_rate)`。**本章能找到“第 20 步开始失控”，全靠这一行。**

这十二条里，哪三条最值钱

坑 3（σ 的两种定义）、坑 6（η 的根号）、坑 12（轨迹记录）。
前两个是因为**记号冲突**——它们是“读论文时不会有问题、写代码时一定会遇到”的那类错误。**唯一的解法是给每个量起一个不会混淆的名字。**
第三个不是 bug，是**工作习惯。它决定了你能否从“炸了”走到“为什么炸”。**

> ! 一句话记住这个区别
>
> **欧拉法在位置空间里更新，$\hat{x} \leftarrow \hat{x} + h\,k$。**
> **二阶要修正的是 k，不是 $\hat{x}$。**
> **把两个“位置估计”平均起来，在数学上没有任何依据 —— 而在某些参数化下，它恰好等于不修正。**

6.14.7 第四部分：一个公平的基准

比较采样器时最容易犯的错，是只比步数、不比求值次数。

```
def benchmark(model, sch, data, configs):
    rows = []
    for name, fn, steps in configs:
        torch.manual_seed(0)
        x = torch.randn(2000, 2, device=DEV)
        out, nfe = fn(model, sch, x, steps)
        rows.append({
            "sampler": name, "steps": steps, "nfe": nfe,
            "std": out.std(0).tolist(),
            "mmd": mmd_rbf(out.cpu(), data).item(),
        })
    return rows
```

注意 `torch.manual_seed(0)` 在循环里 —— 每个配置从**同一批初始噪声**出发。**这是唯一变量原则在采样器比较里的落实。**

6.14.8 实测：三种采样器，按求值次数排在一条线上

采样器	步数	**求值次数**	输出 std	MMD
DDPM	1000	1000	[0.5598, 0.3437]	0.01146
DDIM	25	25	[0.5328, 0.3398]	待测
DDIM	100	100	发散	—
Heun	20	**40**	待测	待测

（Heun 那一行是本章要补的实测。）

读这张表的正确方式：把“求值次数”那一列当作横轴。**DDIM 25 步只花 25 次求值就接近了 DDPM 1000 次的结果 —— 这就是它值得用的理由。**

而它的代价在本章已经反复出现：确定性让它对模型误差零容忍。

6.15 采样器怎么选：一棵决策树

这一节把全章的内容压成一个可以照着走的选择流程。

6.15.1 先问三个问题

问题一：你的模型有多准?

判据：在验证集上，ε_θ 的相对误差大约是多少？（可以这样量：$\frac{\|\varepsilon - \varepsilon_\theta\|^2}{\|\varepsilon\|^2}$。）

- **误差 < 5%**：确定性采样可以用
- **误差 5–15%**：建议 $\eta \geq 0.5$
- **误差 > 15%**：**先把模型训好**，任何采样器都救不了

本章的模型约 6.6%（$t = 999$ 处，实测），落在“建议加噪声”那一档，**而它的实测表现也确实是“$\eta = 0$ 炸、$\eta = 1$ 好”**。

问题二：你需要确定性吗?

- **需要**（编辑、插值、可复现的演示）→ $\eta = 0$，但要接受对模型误差的零容忍
- **不需要**（只是生成样本）→ η 可以取 0.5–1，稳得多

问题三：你的算力预算允许多少次函数求值?

- **< 30 次**：只能用一阶方法，接受质量损失（实测 20 步掉 13%）
- **30–100 次**：二阶方法开始划算（20 步 × 2 = 40 次，误差 2%）
- **> 100 次**：一阶和二阶差别不大，选简单的

6.15.2 一棵决策树

```
模型相对误差 ε_rel
├─ > 15%  → 回去训模型。采样器换什么都没用。
├─ 5–15%  → η 取 0.5–1.0
│           ├─ 求值预算 < 50  → 一阶 + 大 η（本章实测：50 步 η=1 正确）
│           └─ 求值预算 ≥ 50  → 一阶 + 中等 η
└─ < 5%   → η 可以取 0
            ├─ 需要确定性    → DDIM / 一阶
            ├─ 求值预算 < 50 → 二阶（Heun）
            └─ 求值预算 ≥ 50 → 一阶够用
```

这棵树不是从论文里抄的，是从本章的实测里长出来的。 每一条分支背后都有一个具体的数字。

6.15.3 三个反直觉的结论

① **“步数越多越好”是错的。**
本章的案例里，25 步正常、100 步爆炸。**步数只在“模型够准”的前提下才是越多越好。**
② **“确定性更稳定”是错的。**
确定性 = 可复现；**不等于稳定。恰恰相反：确定性采样没有吸收误差的机制。**
③ **“二阶一定比一阶好”是错的。**
在 10 步这个量级，Heun 反而“过头”（+7.5%），Euler 是 −23%。**高阶的优势在中等步数，不在极端少的步数。**

6.15.4 三个真实场景：把决策树走一遍

抽象的分支不好用。**下面三个场景，每个都给出具体的选择和理由。**

6.15.4.1 场景 A：图生图编辑，要求可复现

约束：用户上传一张图，改一句话，要生成结果。同一个输入必须给出同一个输出（否则用户会以为“每次都不一样是 bug”）。预算：30 次函数求值以内。模型相对误差约 5%。

走树：误差 5% 落在边界上 → 先试 $\eta = 0$；需要确定性 → $\eta = 0$；预算 30 → 一阶或二阶都可以，但二阶要 15 步。

推荐：$\eta = 0$，20 步，一阶（20 NFE）。

理由：5% 是本章实测的“$\eta = 0$ 还能用”的上界附近。**如果上线后发现伪影，第一个动作不是换采样器，是把 η 提到 0.3** —— 代价是可复现性，但换来稳定性。

注意：图生图还有另一个约束——**初始噪声来自原图，不是随机的。**这意味着 $\eta > 0$ 会破坏“保留原图结构”这件事，**所以这类应用里 η 的可用范围比纯文生图窄得多。**

6.15.4.2 场景 B：批量生成训练数据，不需要可复现

约束：一晚生成 10 万张，质量优先，多样性优先。预算 50 NFE。模型误差约 3%。

走树：误差 3% → η 可以取 0；不需要确定性；预算 50 → 一阶够用。

推荐：$\eta = 0$，50 步一阶（50 NFE）。

理由：在这个预算下，二阶的收益已经很小（实测 40 步 Heun 是 0.6375，80 步 Euler 是 0.6163，**同成本下 Heun 略好，但差距小于模型误差本身**）。

更重要的是多样性：$\eta = 0$ 是确定性的，**同一个起点永远给同一个样本**。批量生成时必须**换随机种子**，否则你会得到一个“看起来很多、实际很少”的数据集。**这是确定性采样在数据生产里的隐藏陷阱**。

6.15.4.3 场景 C：手机端实时生成

约束：8 次函数求值以内。模型是小模型，误差可能到 15%。
走树：误差 15% → 落在“先训模型”的边缘；但端侧模型没法更大，所以只能接受。
推荐：**不要用 DDIM**。用蒸馏过的一步/四步模型（本章 § 6.16 讲的第三类方法）。
理由：在 8 NFE 的预算下，本章的所有测量都表明 **一阶方法会损失 20% 以上的精度，二阶方法还没进入渐近区。这不是采样器能解决的问题，是“必须换模型”的问题**。

> ⚠ 场景 C 是最常见的一类错误
>
> **很多人会拿着 DDIM 去调 8 步，然后抱怨“质量不行”。**
> **但 8 步 DDIM 的质量上限，不是由采样器决定的，是由 ODE 的刚度决定的**（上一节那张 $\log\sigma$ 网格的表：10 步已经只剩目标的 66%）。
> **在这个预算下，正确的动作是“换模型”，不是“调采样器”。**
> **判别方法**：如果你的预算低于 15 NFE，而模型不是专门为少步训练的，**先去做蒸馏，别在采样器上花时间**。

6.15.5 一张“症状 → 处方”表

最后给一张反向查表：从你观察到的现象，倒推该改什么。

症状	最可能的原因	先试什么	不要做什么
输出全是灰噪	步数太少 + 模型误差大	提 η 到 1	加步数
输出模糊、缺细节	网格太粗（σ 均匀）	换 $\log\sigma$ 网格	换更高阶方法
小步数正常、大步数炸	模型在分布外放大	加 η / 换完美去噪器验证	继续加步数
换调度后质量崩	网格与新调度不匹配	重算 σ 网格	重训模型
每次生成都不一样（$\eta = 0$）	有未固定的随机源	检查 dropout / 数据加载顺序	换采样器
多样性差、样本雷同	确定性采样 + 固定种子	换种子 / 提 η	调温度

注意第一行和第三行的区别。“全是灰噪”通常是**精度不足**（走得太粗），“小步数正常大步数炸”是**稳定性问题**（误差被放大）。**两者的处方完全相反** —— 一个要加噪声，一个要减步数。

> 提示
>
> **这一节真正要你记住的是**
> **采样器的选择不是“哪个方法更新”，是“三个约束下的最优解”：模型精度、确定性需求、算力预算。**
> **脱离这三个约束谈“最好的采样器”，和脱离剂量谈毒性一样没有意义。**

6.16 同一族里的其他成员：PLMS、DPM-Solver++、UniPC

本章的正文只实现了三个采样器：DDPM（$\eta = 1$）、DDIM（$\eta = 0$，一阶欧拉）、Heun（二阶）。**但你在实际代码库里会看到一堆别的名字**。这一节把它们的位置标出来，**不写公式，只讲“它们各自解决了哪个具体问题”**。

6.16.1 PLMS：不要多算，用历史

Heun 的问题是每步算两次。 对于大模型，这意味着显存和时间的双重压力。

PLMS（Pseudo Linear Multistep，2022）的想法是：既然已经算过了，就别再算。

它的做法借用经典数值分析里的**线性多步法**：用**前几步**已经算出的 ε 值，线性组合出这一步的斜率。

$$\hat{x}_{k+1} = \hat{x}_k + (\sigma_{k+1} - \sigma_k) \sum_{j=0}^{p-1} c_j \, \varepsilon^{(k-j)} \tag{6.69}$$

关键是：这些 $\varepsilon^{(k-j)}$ 都是之前已经算过的，不需要新的模型调用。

于是：

方法	每步 NFE	阶数	额外内存
DDIM	1	1	无
Heun	2	2	无
PLMS	**1**	**2**	存 p 个历史 ε

同样的 NFE，二阶。 听起来是免费午餐——**代价是“启动开销”**：多步法需要前 p 步的历史，而第一步没有历史，只能用欧拉起步。**所以在总步数很少时（< 10），PLMS 的二阶优势被启动阶段吃掉了。**

还有第二个代价：它假设步长是“准均匀”的。 如果网格变化剧烈，多步法的系数要重算，否则会不稳定。

> **i** 为什么本书没有实现 PLMS
>
> **因为它需要改的东西太多，而收益可以在别处看到。**
>
> 多步法的系数表（Adams-Bashforth 的 c_j）依赖步数 p 和网格，写成通用代码需要一张系数表和一个“历史缓冲区”。**它的核心思想——“复用历史求值”——一句话就能说完，**而实现细节对理解本章的主线（阶数、网格、稳定性）没有新增信息。
>
> **如果你要上手，顺序建议是**：先把本章的欧拉和 Heun 写对、用完美去噪器测过，**再加 PLMS**。反过来会很难 debug —— 因为多步法的错误表现和网格错误非常像。

6.16.2 DPM-Solver++：换一个预测目标

这个改进针对的是一个具体的失败模式：高噪声区的数值不稳定。

回忆本章的实测：在 t 很大时，ε_θ 的误差会被 $1/\sqrt{\bar{\alpha}_t}$ 放大（$t = 999$ 处是 157 倍）。**而 DPM-Solver++ 的观察是：在高噪声区，直接预测 x_0 比预测 ε 更稳定。**

为什么？ 因为

$$\hat{x}_0 = \frac{x_t - \sqrt{1-\bar{\alpha}_t}\varepsilon_\theta}{\sqrt{\bar{\alpha}_t}} \tag{6.70}$$

当 $\sqrt{\bar{\alpha}_t}$ 很小时，这个式子是“两个大数相减再除以一个小数”。 ε_θ 的一点点误差，经过 $1/\sqrt{\bar{\alpha}_t}$ 放大，再和 $x_t/\sqrt{\bar{\alpha}_t}$ 这个大项相减 —— **灾难性抵消**。

如果模型直接输出 $\hat{x}_0$，就没有这个除法。而 ε 可以由 $\hat{x}_0$ 反算：

$$\varepsilon = \frac{x_t - \sqrt{\bar{\alpha}_t}\hat{x}_0}{\sqrt{1-\bar{\alpha}_t}} \tag{6.71}$$

分母是 $\sqrt{1-\bar{\alpha}_t} \approx 1$，不会放大误差。

所以 DPM-Solver++ 的内核是：把求解器的更新式**改写成以 $\hat{x}_0$ 为变量的形式**，在 σ 大的区间用 $\hat{x}_0$，在 σ 小的区间用 ε。

这是一个纯粹的数值技巧，不改变 ODE，只改变离散化的写法。它的收益随步数减少而增大 —— 因为步数越少，每步跨的 σ 越大，抵消问题越严重。

一句话总结 DPM-Solver++

同一个方程，换一个未知量来离散，数值稳定性就不一样。
这和“解线性方程组时选主元”是同一类思想：**数学上等价，数值上差很多。**

6.16.3 UniPC：把预测-校正统一起来

UniPC（Unified Predictor-Corrector，2023）做的事情是“归类”。
它指出：PLMS、DPM-Solver、DPM-Solver++ 这些方法，**都可以写进同一个“预测-校正”框架**：

1. **预测**：用历史值外推一个 $\hat{x}^{(0)}$
2. **校正**：用（可选的）新求值修正它

不同的方法，区别只在“用什么公式外推”和“校不校正”。
它的实际价值不是“更快”，而是“让超参数变少”：在 UniPC 的框架下，阶数、是否校正、末步怎么处理，变成了几个正交的选项，**而不是每个方法一套独立的调参经验**。

6.16.4 蒸馏：不修采样器，改模型

上面所有方法都在“更好地解 ODE”。蒸馏换了一个问题：“能不能不让他解?”
核心想法：训练一个学生网络，直接学“从任意 $(\hat{x}, \sigma)$ 一步到 $\hat{x}_0$”。

- **一致性模型（Consistency Models, 2023）**：训练 $f_\theta(\hat{x}, \sigma) \approx \hat{x}_0$，约束是“同一条 ODE 轨迹上的任意两点，映射到同一个 x_0”
- **LCM（Latent Consistency Model, 2023）**：把上面这件事搬到潜空间，用少量步数（2–4）生成
- **SDXL Turbo / 各类加速 LoRA**：用对抗损失或蒸馏损失把步数压到 1–4

代价很明确：

	训练成本	采样成本	质量	多样性
原始模型 + DDIM	无	20–50 NFE	基准	基准
原始模型 + DPM-Solver++	无	10–25 NFE	接近基准	接近基准
蒸馏模型	**要重训**	1–4 NFE	略低	**明显下降**

“多样性下降”是蒸馏的结构性问题，不是调参能解决的。学生网络把一整条轨迹压成一个点，**它必然丢掉轨迹上的随机性**。

> ⚠ 什么时候不该用蒸馏
>
> **如果你的应用需要“同一个 prompt 生成很多不同的结果”，蒸馏会伤到你。**
> **判别方法**：用同一个 prompt、不同的种子生成 20 张，**看它们的差异是不是比你预期的少**。如果少，那就是多样性塌缩。
> **这时候的正确选择是**：用原始模型 + 更好的采样器（DPM-Solver++ 或 UniPC），**接受 15–25 NFE 的成本**，而不是用 4 步蒸馏换一个“看起来能用”的结果。

6.16.5 一张归位表

把这一节的名字放回本章的主线：

方法	它改的是	相对本章的哪个概念
DDPM	$\eta = 1$	σ 族的右端点
DDIM	$\eta = 0$，一阶	欧拉
Heun	二阶	多一次求值换阶数
PLMS	多步，复用历史	同 NFE 下的二阶
DPM-Solver	λ 网格 + 高阶	换自变量

方法	它改的是	相对本章的哪个概念
DPM-Solver++	改预测目标	数值稳定性
UniPC	统一框架	归类
蒸馏 / LCM	**改模型**	跳出“解 ODE”

注意最后一行。前七个都在“怎么解这个 ODE”里，**只有最后一行跳出了这个问题。这就是本章反复强调的那条线：采样器的收益有上限，模型的收益没有。**

6.17 噪声调度 × 采样器：两个必须一起调的东西

这一节讲一个经常被忽略的事实：采样器的表现，一半由调度决定。

你在论文里看到一个采样器在某个模型上 20 步出好结果，换到自己的模型上就不行——很可能不是采样器的问题，是调度变了。

6.17.1 调度到底改了什么

调度就是 $\bar{\alpha}_t$ 这条曲线。它决定了三件事：

1. **σ 的范围**：$\sigma_{\max} = \sqrt{1-\bar{\alpha}_T}/\sqrt{\bar{\alpha}_T}$
2. **σ 在 t 上的分布**：t 均匀时，σ 怎么变
3. **“有效步数”**：真正在改变 $\hat{x}$ 的那些步有多少

第三件事最容易被忽略。

线性 β 调度（本章用的那个）在 t 大时 σ 变化极快，在 t 小时变化极慢。**后果是：t 均匀取点时，大部分步数落在“σ 变化慢”的区间，而那些步对结果的贡献很小。**

余弦调度（cosine schedule）的设计目标就是让 σ 的变化更均匀。**所以它和 $\log\sigma$ 网格更匹配，和 t 均匀网格反而不匹配。**

6.17.2 一个可以直接测的判据

如果你怀疑“调度和网格不匹配”，做这个实验：

1. 固定采样器（比如 DDIM）
2. 固定步数（比如 20）
3. 只改调度（线性 / 余弦 / 平方根）
4. 看输出标准差

如果三个调度的结果差很多，说明网格没有适配调度。

本章没有跑这组完整实验（它需要重新训练，因为模型是在线性调度下训的），**但你可以用一个不需要训练的办法**：用完美去噪器，把 $\bar{\alpha}_t$ 换成别的调度，跑同一组步数。**这时候模型误差为零，你看到的差异纯粹是“调度 × 网格”的交互。**

> **i** 为什么“换调度”常常需要“重训”
>
> **因为模型是在特定调度下训练的。**
>
> $\varepsilon_\theta(x_t, t)$ 的输入是 x_t，而 x_t 的分布由 $\bar{\alpha}_t$ 决定。**换调度 = 换输入分布 = 模型在训练时没见过这些输入。**
>
> **这就是“训练/采样不一致”（train-test mismatch）的一种。**它的症状和本章案例一模一样：**采样炸，但训练 loss 正常。**
>
> **所以“换个调度试试”不是一个免费的操作。**它要么需要重训，要么需要接受“模型在部分 t 上外推”的风险。

6.17.3 训练和采样的三个不一致

把这件事系统化，一共三处：

不一致	训练时	采样时	症状
调度	$\bar{\alpha}_t^{\text{train}}$	$\bar{\alpha}_t^{\text{sample}}$	采样炸，loss 正常
时间步范围	$t \in [1, T]$	$t \in [0, T-1]$	t 小时输出无意义
数值精度	fp32 / bf16 混合	纯 fp16	高 σ 处溢出

第二行是最常见的 off-by-one。训练时 t 从 1 到 T，是因为 β_1 对应“加一次噪声”；采样时索引从 $T-1$ 到 0，因为数组是 0-indexed。**两者差 1，而 1 在 t 小的时候是巨大的**。

第三行在端侧部署时特别常见。fp16 的最大值是 65504，而本章的模型在 $t = 999$ 处 $\hat{x}_0$ 能到 12.8、x_t 能到 1.24—— **看起来安全，但中间量 $\sqrt{1-\bar{\alpha}_t}\varepsilon$ 在 batch 大时会有离群值**。

修法：在采样器内部用 fp32 累加，只在调用模型时转 fp16。

6.17.4 一个判断“是不是调度问题”的实验

这是本章没做、但你应该做的一个实验。设计如下：

材料：同一个模型，两套调度（训练用的线性、以及余弦）。

做法：1. 用线性调度训练模型 A 2. 用余弦调度训练模型 B 3. **交叉测试**：A 用线性采样、A 用余弦采样、B 用余弦采样、B 用线性采样 4. 每组用 10 / 20 / 50 步各跑一次

预期的 2×2 表：

	线性采样	余弦采样
模型 A（线性训练）	正常	差
模型 B（余弦训练）	差	正常

如果这个表成立，说明“调度必须匹配”；如果对角线不成立，说明调度的影响比想象的小。无论哪种结果，都是有用的信息。

本书没有做这个实验，因为要重训两个模型。这是本章的一个缺口，也是读者可以自己补的一个。

6.18 怎么读一篇新的采样器论文

你以后会不断遇到新采样器。这一节给一套读法，**它不依赖你懂那篇论文的全部数学**。

6.18.1 先找三样东西

任何采样器论文，都可以用三个问题定位：

1. **它解的是哪个 ODE？**（概率流 ODE？反向 SDE？还是别的？）
2. **它改了什么？**（网格？阶数？预测目标？还是模型？）
3. **它的成本怎么算？**（每步几次求值？末步有没有额外修正？）

这三个问题回答完，你就知道它在你已有知识里的位置了。

本章的归位表（§6.16）就是按这个格式做的。

6.18.2 三个可以跳过的部分

大多数采样器论文里，下面三段可以跳过：

① **收敛性证明**。它们通常假设模型是 Lipschitz 的、ε_θ 有界、步长趋于零——**这些假设在真实模型上全都不成立**。证明的价值是“保证存在一个足够小的步长”，**不是“保证你的 20 步能用”**。

② **和 SOTA 的 FID 对比表**。见 §6.25 的三个陷阱。**除非它公开了 NFE 和评估代码，否则这张表只能当参考**。

③ **消融实验里“每个组件贡献多少”的分解**。这些数字高度依赖模型和数据集，**换个模型就不成立**。

6.18.3 三个必须读的部分

① **它的更新式**。哪怕你不懂推导，也要能把公式抄成代码。**抄不出来的地方，就是你没懂的地方**。

② **它的网格定义**。是在 t 上均匀、在 σ 上均匀、还是在 λ 上均匀？**这一条决定了它能不能和你的调度配合**（见 §6.17）。
③ **它报告的 NFE**。如果论文只报“步数”，**去附录或代码里找 NFE**。找不到，就在你的复现里自己数模型调用次数。

> **i** 一个实用的检验：用 10 步跑一次
>
> **拿到任何新采样器，第一件事不是跑它的 benchmark，是用 10 步跑一次。**
> 为什么是 10 步？因为**在这个量级上，所有近似假设都不成立。一个在 10 步下不崩的采样器，通常是真的做了什么对的事；一个在 10 步下崩了、但 50 步很好的采样器，很可能只是“高阶但没进入渐近区”。**
> **本章实验 6-1 的表就是这个思路**：10 步时 Heun 反而“过头”（+7.5%），Euler 是 −23%——**两者都不好，但方向相反。这个信息比“50 步下谁更好”值钱得多。**

6.18.4 什么时候该怀疑论文

三个信号：

1. **只报一个步数档位。**本章已经证明，25 步和 100 步的结论可以完全相反。
2. **只报 FID，不报多样性指标。**蒸馏类方法尤其容易在这里藏问题。
3. **没有说明网格。**一个采样器的收益，可能有一半来自它偷偷换了网格。

第三条最隐蔽。如果一篇论文说“我们的方法在 20 步下 FID 更好”，**但没说它的 20 步是在 $\log\sigma$ 上取的、而 baseline 是在 t 上取的——那么它比较的不是“方法”，是“网格”。**
本章 §6.5 那张表（差距 18 倍）说明这个混淆可以有多大。

6.19 工程问题：当采样器进入真实项目

数学上正确，不等于工程上可用。这一节列四个真实项目里一定会遇到的问题。

6.19.1 批量大小与显存

采样器的显存占用不是“模型大小”，而是“模型大小 + 中间激活 × 批大小”。
高阶方法更吃显存：Heun 每步要保存两个中间状态（$\hat{x}$ 和 $\hat{x}_{\text{pred}}$），**在同样批大小下比欧拉多一份。**
实测经验（本章的 5 万参数模型上没有意义，但在真实模型上）：- DDIM 20 步，batch 8，约需 X GB - Heun 20 步，batch 8，约需 $1.3X$ GB - **如果要保持同样显存，Heun 的批大小要降到 6 左右**
这不是“二阶更贵”的全部——它还要多一次前向。**两者叠加，同样的墙钟时间下，Heun 的实际吞吐可能只有 DDIM 的一半。**

6.19.2 精度：fp16 在哪里溢出

本章反复出现的一个量是 $1/\sqrt{\bar{\alpha}_t}$，在 $t = 999$ 处是 157。
在 fp16 下（最大值 65504），这个因子本身没问题。但它乘上 $\hat{x}_0$ 之后呢？
本章实测：第 0 步的平均 $|\hat{x}_0| = 12.8$，**最大值远超均值。**如果某个样本的 $|\hat{x}_0|$ 是 10^3，乘 157 就是 1.6×10^5，**溢出。**
修法：1. **在 $\hat{x}$ 空间里算**（除以 $\sqrt{\bar{\alpha}}$ 之后），数值范围小得多 2. **关键步骤用 fp32 累加**，只在调用模型时转 fp16 3. **clamp 在 fp32 下做**，不要等到转回 fp16 才截断

6.19.3 确定性：不是“设个种子”就完了

$\eta = 0$ 时，理论上同样输入给同样输出。实际不然，因为：

- **CUDA 的某些算子是非确定性的**（原子加、某些归约）
- **不同的批大小会改变归约顺序**，于是结果不同
- **混合精度训练的模型在推理时用不同精度，结果也会变**

要真正可复现：

```
torch.use_deterministic_algorithms(True)
torch.backends.cudnn.benchmark = False
```

代价是速度下降 10–30%。

如果做不到，就接受“同种子下差异在 10^{-5} 量级”，**并在产品文档里写清楚**——不要承诺“逐位可复现”。

6.19.4 缓存与增量生成

图生图 / 局部重绘这类应用，需要“保留一部分原图”。

常见的错误做法：先跑完整的 DDIM 到 t，再从 t 往回走。**这不对**，因为 DDIM 的轨迹依赖起点。

正确做法：用 **SDEdit 式的“加噪到 t 再往回走”**： 1. 把原图编码成 x_0 2. **前向加噪**到 t（用真实的 $\bar{\alpha}_t$） 3. 从这个 x_t 开始反向采样

关键：第 2 步用的是**前向过程**，不是“反向走到一半”。**两者在 $\eta = 0$ 时不等价**——前向加噪是随机的，反向是确定的。

> ⚠ 一个常见但错误的“优化”
>
> **有人会想：既然 $\eta = 0$ 是确定性的，那我可以把中间结果缓存起来，下次从缓存继续。**
>
> **不行。**因为缓存的是一个**具体的 x_t**，而它只对**产生它的那条轨迹**有意义。**换一个 prompt、换一个种子，x_t 就完全不同。**
>
> **能缓存的是“模型的 KV 缓存”（在 Transformer 里）或者“文本编码”，不是“采样轨迹”。**

6.20 十个常见问题

这些都是学生在课上真正问过的。 答案尽量短，需要展开的地方指回对应小节。

6.20.1 问一：DDIM 到底是不是“DDPM 的加速版”?

不是。 DDPM 和 DDIM 是同一个 σ 族的两个端点（§ 6.3）。**DDIM 不是“近似了 DDPM”，它是另一个合法的采样过程。**

这个区别很重要：如果 DDIM 是“近似的 DDPM”，那步数少就是“精度低”；**但 DDIM 在 1000 步下不收敛到 DDPM 的结果**（$\eta = 0$ vs $\eta = 1$），**它收敛到的是同一个分布，但走的是不同的路径。**

“同一个分布，不同的路径” —— 这是理解本章所有采样器的钥匙。

6.20.2 问二：为什么 $\eta = 0$ 反而更不稳定?

因为随机性在采样里扮演的是“误差吸收器”。

DDPM 每步注入的噪声，会把上一步积累的误差“冲掉”一部分（第 5 章讲的“误差自我修正”）。**确定性采样没有这个机制，误差只会被传递和放大。**

所以 η 的正确理解不是“随机性旋钮”，是“你愿意用多少随机性换稳定性”。（§ 6.9）

6.20.3 问三：步数到底该取多少?

没有通用答案，但有三个约束：

1. **模型误差 < 5%** → 20 步可能够
2. **模型误差 5–15%** → 要么加 η，要么加步数，**但加步数可能适得其反**
3. **预算 < 15 NFE** → 别调采样器，去蒸馏

本章的实测：完美去噪器下，一阶要 100 步才能到 2% 误差；二阶只要 20 步（40 NFE）。**所以“步数”这个数只有在说清“什么采样器、什么模型”之后才有意义。**（§ 6.25）

6.20.4 问四：为什么我的 DDIM 在 t 很大时输出是纯噪声?

因为它就应该是纯噪声。 $\bar{\alpha}_{999} \approx 4 \times 10^{-5}$，**信号只剩万分之四。**

你看到的“纯噪声”是对的。真正的问题是：**它在 t 变小之后有没有逐渐长出结构。**如果没有，去看 §6.2 的轨迹表——**输出是不是在变大。**

6.20.5 问五：clamp 到底该不该用?

该用，但要知道它在掩盖什么。

clamp 的作用是“防止单步爆炸”，它**不解决**根因。本章的实测：去掉 clamp 之后，连 25 步那一档也炸了——**所以它在帮忙。**

但命中率是诊断指标。如果前几步的命中率就超过 50%，说明 $\hat{x}_0$ 的估计已经不可用，**该去查模型，不是调 clamp 的值。**（§6.13 失败五）

6.20.6 问六：为什么换一个调度，采样就崩了?

两种可能：

1. **训练/采样不一致**（模型没见过新调度下的输入分布）→ §6.17
2. **网格不匹配**（新调度的 σ 分布和你的网格不匹配）→ §6.5

区分方法：用完美去噪器跑新调度。- 完美去噪器也崩 → **是网格问题** - 完美去噪器正常 → **是模型问题**（训练/采样不一致）

这个二分法和本章案例用的是同一个逻辑。

6.20.7 问七：Heun 一定比 Euler 好吗?

不一定。本章实测：10 步时 Heun 是 +7.5%，Euler 是 −23%——**Heun 更接近，但方向反了，而且它的成本是两倍。**

“高阶更好”是渐近结论：当步长足够小时，高阶方法的误差下降更快。**在步长很大时，两种方法都没进入渐近区，比较阶数没有意义。**（§6.5）

6.20.8 问八：为什么我的采样器在 CPU 和 GPU 上结果不一样?

三个原因，按常见程度排：

1. **浮点归约顺序不同**（GPU 的并行归约和 CPU 的顺序不同）
2. **算子实现不同**（某些算子在 GPU 上有近似版本）
3. **精度不同**（CPU 默认 fp32，GPU 可能开了 TF32）

这不是 bug。差异通常在 10^{-5} 量级。**如果你需要逐位一致，**用 `torch.use_deterministic_algorithms(True)`，并固定设备。

6.20.9 问九：为什么“同样 20 步”，别人的结果比我的好?

先问四个问题：

1. **他的 20 步是 NFE 还是循环次数?**（Heun 的 20 步 = 40 NFE）
2. **他的网格是 t 均匀还是 λ 均匀?**（差距可达 18 倍）
3. **他的模型是在什么调度下训的?**
4. **他的 η 是多少?**

这四个问题里，任何一个的答案不同，比较就没有意义。（§6.25）

6.20.10 问十：学完这一章，我应该用哪个采样器?

默认答案：DPM-Solver++ 或 UniPC，20–25 步。

理由：它们同时做了“λ 网格”和“更好的预测目标”两件事，**在中等步数区间（15–30 NFE）的性价比最高。**

但这个默认答案有三个例外：

- **需要确定性 + 可复现** → DDIM，$\eta = 0$
- **模型误差 > 10%** → 先用 $\eta \geq 0.5$ 的 DDPM 式采样
- **预算 < 10 NFE** → 蒸馏，别用上面任何一个

最重要的是：别信“某个采样器最好”这种说法。采样器的选择是“模型精度 × 确定性需求 × 算力预算”三个约束下的最优解。**换一个模型，答案就会变。**（§6.15）

6.21 把轨迹画出来：三个必看的图

本章反复用“轨迹”来定位问题。这一节讲怎么画、怎么看。

6.21.1 图一：$|x|$ 随步数的变化（对数纵轴）

这是最重要的一张图。

```
traj = []
for i in range(len(idx) - 1):
    ...
    traj.append({"step": i, "t": int(idx[i]),
                 "absmean": x.norm(dim=1).mean().item(),
                 "epsnorm": eps.norm(dim=1).mean().item(),
                 "clamp": (x0.abs() > 3).float().mean().item()})
```

怎么看：

形状	含义
单调下降，末端趋于数据尺度	正常
先降后升，且斜率越来越陡	**不稳定**（本章案例）
全程几乎不变	模型输出接近零，采样没在做事
一开始就极大	初始化或调度错了

为什么用对数纵轴：σ 跨四个数量级，线性轴会把关键区间压成一条线。**本章第一版图就是线性轴，结果什么也看不出来。**

6.21.2 图二：$\|\varepsilon_\theta\| / \|x\|$ 随步数的变化

这张图解释“为什么会不稳定”。

正常：这个比值应该**下降**（因为随着 σ 减小，噪声成分减少）。**异常**：它**上升并稳定在一个大于 1 的常数**（本章是 1.9）—— 说明模型已经不再做去噪，而是在放大输入。

这张图是“模型在分布外退化”的直接证据。（§6.2）

6.21.3 图三：clamp 命中率

这张图告诉你“从哪一步开始不可用”。

正常：前几步高（$\hat{x}_0$ 本来就难估），然后快速下降。**异常**：一直高，或者下降后再次升高。

本章的数据：86.5% → 57.8% → 13.2% → 3.1%。**前两步高是正常的**（那时信号确实弱），**但 86.5% 意味着“绝大多数样本的估计被丢掉了”——这是一个警告信号。**

6.21.4 三张图放一起

把三张图画在同一张图上（双纵轴，或者上下两栏），你会看到：

1. $|x|$ 开始上升的位置
2. ε/x 开始偏离正常的位置
3. clamp 命中率开始变化的点

如果三条线在同一个步数附近发生变化，因果链就闭合了。

本章的图里，三者都在第 20 步附近变化，这就是 §6.2 那个“第 20 步开始失控”的结论的来源。

画图的一条纪律

记录轨迹的成本是每个样本每步两个 float。 2000 个样本 × 100 步 × 3 个量 = 600k 个浮点数，**不到 5 MB。**
而它的价值是：把“炸了”变成“第 20 步开始以 1.9 倍放大”。
没有理由不记。

6.22 端到端：从第 5 章的模型到这一章的采样器

本章的代码散在各个脚本里。这一节把链路接起来，**给你一个“从训练到采样”能跑通的最小闭环。**

6.22.1 链路全景

```
make_two_moons()          数据
      ↓
Config / make_schedule()  调度 (β → ᾱ → σ)
      ↓
train()                   训练 ε_θ
      ↓
┌──────────────────────────────────────┐
│ 采样器（本章的产物）                 │
│   euler()   ← DDIM, η=0              │
│   ddpm()    ← η=1                    │
│   heun()    ← 二阶                   │
└──────────────────────────────────────┘
      ↓
describe()                统计量
      ↓
轨迹记录 → 诊断           本章的调试方法
```

三个接口约定（不遵守就会在某个环节出错）：

1. **调度只构造一次**，作为参数传下去
2. **采样器不构造调度，只读**
3. **模型只接受 (x_t, t)，返回 ε**

6.22.2 一个最小的采样循环

```python
def sample(model, sch, kind="euler", steps=20, eta=0.0, n=2000, seed=0):
    """统一入口。返回 (样本, NFE)。"""
    torch.manual_seed(seed)
    T = sch["T"]
    idx = torch.linspace(T - 1, 0, steps).round().long()
    # 起点: x_T ~ N(0, I), 这是 ᾱ_T ≈ 0 时的边缘分布
    x = torch.randn(n, 2, device=sch["device"])
    nfe = 0
    for i in range(len(idx) - 1):
        t, tp = int(idx[i]), int(idx[i + 1])
        ab, abp = sch["alpha_bar"][t], sch["alpha_bar"][tp]
        tt = torch.full((n,), t, device=x.device, dtype=torch.long)
        with torch.no_grad():
            eps = model(x, tt)
        nfe += 1
        x0 = ((x - (1 - ab).sqrt() * eps) / ab.sqrt()).clamp(-3, 3)

        if kind == "heun":
            # 预测点再求一次
```

```
            xp = abp.sqrt() * x0 + (1 - abp).sqrt() * eps
            tt2 = torch.full((n,), tp, device=x.device, dtype=torch.long)
            with torch.no_grad():
                eps2 = model(xp, tt2)
            nfe += 1
            eps = (eps + eps2) / 2

        sig = eta * ((1 - abp) / (1 - ab)).sqrt() * (1 - ab / abp).sqrt()
        x = (abp.sqrt() * x0
             + (1 - abp - sig ** 2).clamp_min(0).sqrt() * eps
             + (sig * torch.randn_like(x) if eta > 0 else 0))
    return x, nfe
```

这段代码里有三处和本章前面不同，都是故意的：

① **Heun 分支里，`eps` 被换成了两次求值的平均。注意它平均的是 ε，不是 $\hat{x}_0$** —— 这正是 §6.5 那个“假的二阶”踩的坑。

② **`eta > 0` 时才生成噪声。** 否则 `torch.randn_like` 会消耗随机数，**破坏 $\eta = 0$ 的可复现性**。

③ **返回 NFE。 这是所有对比实验的前提**（§6.25）。

6.22.3 用它验证三件事

```
# ① η=1 应该逐位等于 DDPM
x1, _ = sample(model, sch, "euler", 1000, eta=1.0)
# 和你的 DDPM 实现对比，应该完全一致

# ② η=0 应该可复现
xa, _ = sample(model, sch, "euler", 50, eta=0.0, seed=7)
xb, _ = sample(model, sch, "euler", 50, eta=0.0, seed=7)
assert torch.equal(xa, xb)

# ③ Heun 的 NFE 应该是一阶的两倍
_, n1 = sample(model, sch, "euler", 20)
_, n2 = sample(model, sch, "heun", 20)
assert n2 == 2 * n1
```

这三条断言，是本章所有结论的地基。如果第一条不成立，你的 η 公式错了；如果第二条不成立，有未固定的随机源； 如果第三条不成立，你的 NFE 统计错了。

> ! 为什么第一条断言这么重要
>
> **$\eta = 1$ 必须逐位等于 DDPM。** 这不是“应该差不多”，是**必须相同**。
>
> 因为 DDPM 的采样式是从 ELBO 严格推出来的， 而 $\eta = 1$ 在 σ 族里**恰好**对应它。**两者在数学上是同一个东西，所以数值上也必须相同。**
>
> **如果不同，只可能是三种情况：** 1. η 的公式错了（最常见：少开根号）2. 重建时 ε 的系数错了 3. 起点分布错了（DDPM 从 $\mathcal{N}(0, I)$ 起， 不是从 $\mathcal{N}(0, \sigma_{\max}^2)$）
>
> **这一条断言，能一次性排除本章 §6.14 里的坑 6 和坑 7。**

6.22.4 采样器的验收清单

在把任何采样器接进项目之前，跑这五条：

#	检查	通过标准
1	$\eta = 1$ 对上 DDPM	逐位相同
2	$\eta = 0$ 同种子可复现	逐位相同
3	NFE 计数正确	Heun = 2×欧拉

#	检查	通过标准
4	完美去噪器下 20 步	std 在目标 ±15% 内
5	多档位一致	25 / 50 / 100 步结论不相反

五条全过，才可以接进项目。 本章的案例说明，**只做第 4 条是不够的**——它在 25 步下过了，在 100 步下炸了。

6.23 三条没有流行起来的路线

历史只记住赢家。 但“为什么有些方法没流行”往往比“赢家做了什么”更有信息量—— **因为那些原因，就是你在自己项目里会遇到的约束。**

6.23.1 路线一：退火 Langevin（NCSN 的做法）

在 score matching 流行起来之前，采样是这么做的：
对每个噪声水平 $\sigma_1 > \sigma_2 > \cdots > \sigma_L$： 1. 在这个噪声水平上跑 K 步 Langevin 动力学 2. 把结果作为下一个（更小）噪声水平的起点
问题在哪?
每层要跑 K 步，而 K 通常取 100 左右。$L = 10$ 层 $\times$ $K = 100 = 1000$ 次求值 —— 和 DDPM 一样慢，而且步长极难调。
更麻烦的是：步长必须满足 $\tau < 2/L_{\text{lip}}$（Langevin 的稳定性条件），而 L_{lip} 在每层都不一样。**你需要为每一层单独调一个步长。**
为什么它没能留下来：DDPM 把“多步 Langevin”换成了“一步反向高斯”，把超参数从“每层的步长”减少到“一个调度”。
教训：**一个算法的工程价值，一半在于它的超参数有多少。**

6.23.2 路线二：预测-校正（predictor-corrector）

Song et al. 2021（score-based SDE）提出过一种混合方案：

1. **预测**：用 SDE 或 ODE 走一步
2. **校正**：在同一个噪声水平上跑几步 Langevin，把样本“拉回” p_t

理论上很漂亮 —— 校正确实能减少离散化误差。
为什么没流行?
因为它把成本翻了几倍，而收益只在步数极少时才明显。而“步数极少”这个区间，后来被蒸馏类方法直接占据了。
教训：**“理论上更准”和“工程上更划算”是两件事。校正的收益必须大于它多花的求值次数。**

6.23.3 路线三：EDM 的“纯 σ 参数化”（这条其实赢了）

Karras et al. 2022（EDM）做了一件事：把整个扩散过程用 σ 重写，丢掉 α、β、t 这些记号。
他们的采样器是：1. 在 $\log\sigma$ 上非均匀取点（**本章 §6.5 用的就是这个**）2. 用 Heun（二阶）3. 加一个“σ 预条件”：把模型的输入输出按 σ 缩放，让网络在不同噪声水平上看到同样尺度的量
为什么这条赢了?
因为它把“训练”和“采样”解耦了：模型只关心 σ，采样器只关心 σ 网格。**两者通过一个数（σ）连接。**
而 DDPM 的原始参数化里，模型要同时理解 t、α_t、β_t—— 换一个调度，模型就懵了。
教训：**好的抽象比好的算法更持久。本章通篇用 σ 而不是 t 来讲，就是这个原因。**

> **i** 为什么本章没有用 EDM 的预条件
>
> **因为本章的模型很小（5 万参数），预条件的收益不明显，而它会增加一层“看起来像魔法”的缩放因子。**
> **EDM 的预条件大致是**：
>
> $$\text{输入} \times \frac{1}{\sqrt{\sigma^2+\sigma_{\text{data}}^2}}, \qquad \text{输出} \times \frac{\sigma\,\sigma_{\text{data}}}{\sqrt{\sigma^2+\sigma_{\text{data}}^2}} \tag{6.72}$$
>
> **它的作用是让网络的输入输出都保持在 $\mathcal{O}(1)$。**
> **在真实模型上，这一层几乎是必须的**（否则 fp16 会在高 σ 处溢出，见 §6.19）。**本章省略它，是为了让 σ 参数化这条主线更清楚。**
> **如果你要把它用到真实模型上，记得加回来。**

6.24 采样器的十年：一条不断缩短的路

这一节不讲新公式，讲一件事是怎么被一步步做出来的。之所以要讲，是因为**每一步的动因都对应本章已经量过的一个数** —— 你会看到这条技术路线不是随机游走，而是在回答同一个问题：**怎么用更少的函数求值，走到同一个分布。**

6.24.1 出发点：DDPM 的 1000 步

2020 年的 DDPM 用 $T = 1000$，每一步对应一个固定的噪声等级。**这个设计的出发点是“让每一步的分布都很简单”** —— 每一步只挪动一点点，模型只需要学一个很小的修正。
代价直接写在结构里：生成一张图要跑 1000 次网络。在 2020 年的硬件上，这意味着几十秒到几分钟。
第 5 章测过这个代价的一个侧面：**1000 步里有 327 步（32.7%）的输入几乎是纯噪声**，模型在那里的任务简单到几乎不需要学。**换句话说，那 1000 步里有三分之一是冗余的。**

6.24.2 第一步：能不能跳步（DDIM, 2020 末）

DDIM 的洞见在本章 §6.3 讲过：**反向过程需要的不是整条马尔可夫链，而是每个时刻的边缘分布。**只要边缘分布对得上，中间走几步是可以商量的。
这个改动在数学上是干净的：它构造了一族新的前向分布，边缘分布不变，但不再是马尔可夫的。**然后在这族分布里取一个“方差为 0”的特例。**
回报是 20–50 倍的速度，而质量损失在好模型上很小。**这就是为什么 DDIM 迅速成为默认选择。**
代价在本章案例里暴露得很彻底：确定性采样对模型误差零容忍。DDIM 并没有“解决”采样慢的问题，它是**用鲁棒性换速度。**

6.24.3 第二步：能不能用更好的数值方法（DPM-Solver, 2022）

DDIM 之后，研究者的注意力转向了一个更基础的问题：**既然 DDIM 本质上是解一个 ODE，为什么只用一阶方法?**
这个转向的出发点是一个观察：概率流 ODE 在 σ 空间里是**半线性的** —— 它的非线性部分只通过 ε_θ 进来。**对这类方程，有专门的指数积分器**，可以在相同步数下把误差降一个数量级。
DPM-Solver 系列就是这条思路的产物。它们给出了**二阶、三阶的专用离散化**，并且在 20 步左右就能达到 DDIM 需要 100 步的质量。
本章 §6.5 用 Heun 演示了同一个原理的最简版本：二阶在 20 步时误差 2%，一阶要 80 步才到 2.6%。**更精细的 DPM-Solver 能把这个差距再拉大一截**，因为它利用了 ODE 的解析结构，而 Heun 是通用方法。
这一节要你记住的区分：- **DDIM 是“换了一个可以跳步的公式”** - **DPM-Solver 是“在同一个公式上换更好的数值方法”**
两者可以叠加，但它们的改进来源完全不同。

6.24.4 第三步：能不能不“解”了，直接学（蒸馏, 2022–2023）

前两步都是在“怎么走得更快”上做文章。第三步换了个问题：能不能干脆不走?

思路是蒸馏：让一个小模型学会“从 x_t 直接跳到 x_{t-k}”，把多步合并成一步。训练时用原来的多步采样器当“老师”，生成“老师走了 k 步之后的终点”，让学生去拟合。

这一步的极限是“一步生成”：从噪声直接到图像。2023 年之后出现了好几条路线（一致性模型、对抗蒸馏、分布匹配蒸馏），**目标都是把函数求值次数从几十压到 1–4**。

代价是训练变复杂：蒸馏需要额外的训练阶段、需要老师模型、需要小心处理分布偏移。**而且蒸馏出来的模型通常只能用在它被蒸馏的那个采样器/步数上，灵活性大幅下降**。

6.24.5 第四步：换一条路径（流匹配, 2022– ）

前面三步都在“沿着调度定义的那条曲线走”这个前提下做优化。最后这一步把这个前提也放开了。

流匹配的核心观察是：连接噪声和数据的路径不是唯一的。**扩散模型用噪声调度定义了其中一条**，但那条路径通常是弯的 —— 从噪声到数据，轨迹会绕。

如果能把路径变直，那么同样的步数就能走得更远。

实测的支撑：本章 §6.5 那张表里，误差随步数的下降速度，本质上取决于**路径的曲率**。二阶方法能改善，是因为它更好地跟随曲率。**如果路径本身就是直线，一阶方法就够了** —— 这正是流匹配想要的效果。

这条路线是本书写作时最活跃的方向之一，本章不展开，但读者应该已经能看出它的动机：**它是在回答“为什么需要那么多步”这个问题的最后一个层次**。

6.24.6 一张收敛表

把四条路线的改进来源放在一起：

方法	改的是什么	典型求值次数	代价
DDPM	—（基线）	1000	慢
DDIM	**公式**（可跳步）	20–100	对模型误差零容忍
DPM-Solver	**数值方法**（更高阶）	10–25	实现复杂，需调参
蒸馏	**模型本身**（学会跳）	1–8	要额外训练，灵活性差
流匹配	**路径**（更直）	4–20	要改训练目标

注意最后一行和前四行的区别：前四个都在“推理”这一侧改，**流匹配改了训练**。**这也是它为什么能和前面几个叠加的原因**。

> ! 这张表要你带走的一个判断
>
> **“更快的采样器”有五种不同来源，它们的代价完全不同**。
>
> - 改公式（DDIM）：**免费**，但牺牲鲁棒性
> - 改数值方法（高阶）：**免费**，但每次求值更贵、实现更复杂
> - 改模型（蒸馏）：**要重新训练**，且失去灵活性
> - 改路径（流匹配）：**要重新设计训练目标**
>
> **遇到一个“快 10 倍”的采样器，先问它属于哪一类。答案决定了它能不能直接搬到你的模型上**。

6.25 怎么判断一个采样器好不好

这一节不讲新算法。**它讲怎么不被评估结果骗**。

为什么值得单独一节? 因为本章前面所有结论 —— “二阶更省”、“$\log\sigma$ 网格更好”、“η 是稳定性旋钮” —— **都依赖同一个前提：你测的是对的东西**。而这个前提，比看上去难满足。

6.25.1 第一个陷阱：把“步数”当成“成本”

论文里的表格通常长这样：

方法	步数	FID
DDIM	50	12.3
DPM-Solver	20	11.8

读者会得出结论：“DPM-Solver 用 20 步就超过了 DDIM 的 50 步。”
这个结论可能是错的，因为“步数”不是成本。成本是**函数求值次数**（NFE）：

- DDIM 每步 1 次 → 50 步 = **50 NFE**
- Heun 每步 2 次 → 20 步 = **40 NFE**
- 某些三阶方法每步 3 次 → 20 步 = **60 NFE**

如果那篇论文里的 DPM-Solver 是二阶的 20 步，它的成本是 40 NFE，确实比 50 少。但如果它是三阶的，成本反而是 60，比 DDIM 还贵。
所以第一件事：看到“步数”两个字，先找它的 NFE 定义。如果论文没写，**默认它在隐藏成本。**

> ⚠ 一个真实存在的混淆：末步修正
>
> 很多高阶采样器在最后一步会做**额外的修正**（比如 DPM-Solver 的 “last-step second-order”），这一次修正**要再算一遍模型**。
> 于是同一个方法，标称 20 步，实际 NFE 是 21 或 22。**在 20 步这个量级上，多一次就是 5%。**
> **写基准的时候要把这一次算进去。**本章 `experiments/` 里的对比脚本 统计的是**真实的模型调用次数**，不是循环次数。

6.25.2 第二个陷阱：用“像不像”代替“分布对不对”

本章的案例已经演示过这件事：25 步 DDIM 的输出 std 是 $[0.5328, 0.3398]$，数据是 $[0.5707, 0.3351]$。**肉眼看几乎一样，但它是错的。**
在图像上，这个陷阱更隐蔽。一个采样器可能生成“看起来很干净”的图，但：

- **多样性塌缩**：所有样本都长得很像，mode 覆盖不全
- **纹理过平滑**：高频细节被抹掉，但人眼对这点不敏感
- **颜色偏移**：整体色调偏一点，打分模型看不出来

FID 对前两项都不敏感。它比较的是 Inception 特征的均值和协方差，**一个“每个 mode 都采一点点、但每个 mode 内部都很糊”的分布，可以拿到很好的 FID。**
所以本章坚持用“有解析答案的问题”做对照：1D 高斯的目标标准差是 0.6325，**这是一个可以写下来的数。**采样器给出 0.55，就是错了 12.9%，没有辩解余地。
在真实数据上没有这个奢侈。那时候的替代方案是：

检查	能发现什么	代价
目标标准差的解析值	尺度错误、塌缩	只在玩具问题上可行
最近邻距离分布	mode 塌缩	要一个参考集
不同随机种子的输出方差	多样性塌缩	便宜
换一个步数重测	不稳定性（本章案例）	便宜，**最容易被忘**
人眼看 50 张	伪影、纹理	贵，但不可替代

注意第四行。本章案例的全部发现，都来自**“换个步数再测一次”。如果一个采样器只在某一个步数下好看，它不是采样器，是巧合。**

6.25.3 第三个陷阱：在错误的模型上比较采样器

这是最隐蔽的一个。

假设你有两个模型 A 和 B，想在它们上面比较 DDIM 和 Heun。你在 A 上测出 Heun 好 10%，在 B 上测出 Heun 好 2%。**于是你写：“Heun 平均好 6%。”**

这个“平均”没有意义。因为采样器的表现依赖模型误差的结构（本章案例已经证明：同一个采样器在完美去噪器上正常、在学出来的模型上炸）。

采样器和模型不是两个独立的因素，它们会交互。

正确的做法是固定模型、只变采样器，而且——**理想情况下先用完美去噪器把采样器本身的数值性质测清楚，**再去真实模型上测“模型误差 × 采样器”的交互。

本章的结构就是按这个顺序组织的：先在解析问题上测（§6.3–§6.6），再回到学出来的模型（§6.2、§6.7）。

6.25.4 一个可以照抄的评估协议

把上面三条合起来，得到一个五步协议。**每一步都标注了“如果跳过会怎样”。**

第一步：写下一个已知答案的对照问题。

不一定是 1D 高斯。可以是二维环面、双月、或者任何你能采样、能算矩、能画图的分布。**关键是“正确答案可写下来”。**

跳过会怎样：你无法区分“采样器错”和“模型错”，本章案例就是从这里开始的。

第二步：在这个对照上，把候选采样器的 NFE 对齐。

不是步数对齐，是**模型调用次数**对齐。列出每个方法在 10 / 20 / 40 / 80 NFE 下的结果。

跳过会怎样：你会把“每步算三次”的方法误认为“三步就够”。

第三步：至少测两个步数档位，差距要拉开 4 倍以上。

本章案例里，25 步和 100 步的结论**完全相反。**只测一个档位必翻车。

跳过会怎样：你会交付一个只在特定步数下能用的采样器。

第四步：记录轨迹，不只看终点。

本章能定位到“第 20 步开始失控”，是因为记录了每一步的 $|x|$。**只看终点，你只知道“炸了”，不知道“什么时候开始炸”。**

跳过会怎样：你能发现问题，但定位不到原因。

第五步：在真实模型上重跑第一步到第四步。

预期是：真实模型上的结论会比对照问题差。差多少，就是“模型误差”这一项的贡献。

跳过会怎样：你会把“采样器的数值性质”和“模型的误差”混为一谈。

> 这套协议的成本
>
> **对照问题半天，NFE 对齐一小时，多档位重测一小时，轨迹记录改十行代码，真实模型重跑一天。**
>
> **总共两天左右。**而本章案例里，没有这套协议的代价是：一个下午的弯路 + 一个“看起来对”的错误结论 + 上线后的伪影。
>
> **两天换这些，很划算。**

6.26 动手练习

五个练习，从“复现”到“设计一个新采样器”。**都能在本章的代码基础上完成。**

6.26.1 练习一：复现，然后改一个变量

跑一遍 `notebooks/run_ch6_case.py`，确认你能得到本章的三组数：

实验	本章的值
DDIM + 学出来的模型，100 步	std $\approx [5.4 \times 10^7, 1.2 \times 10^8]$
DDIM + 完美去噪器，20 步	0.5509（目标 0.6325）
η 扫描，50 步，$\eta = 1$	$[0.525, 0.320]$

然后**只改一个变量**：把学习率从默认值提高 3 倍，重训一次，再跑第一行。

预测一下：爆炸会更早还是更晚？最大绝对值会变大还是变小？

预期的思考方向：学习率更高 → 模型在训练集上更“自信” → 分布外的外推可能更差 → 爆炸更早。**但这不是必然的**—— 欠训练的模型也可能因为输出接近零而“不会炸”。**重点是你会得到一个属于你自己的数，而不是本章的这个数**。

6.26.2 练习二：找出你的模型的“临界步数”

本章的模型在 25 步正常、50 步半炸、100 步全炸。**你的模型呢?**

设计一组实验： - 步数序列：`[10, 15, 20, 25, 30, 40, 50, 75, 100, 150, 200]` - 每个步数跑 3 个随机种子 - 记录输出 std、最大绝对值、以及**是否出现 nan**

画一张图：横轴步数（对数），纵轴最大绝对值（对数）。你会看到一条 在某处突然抬头的曲线。

然后回答两个问题： 1. 抬头发生在第几步? 2. 这个位置和训练步数有关吗?（把训练步数砍半，重跑一次）

这个练习的价值：**“临界步数”不是采样器的属性，是模型的属性**。你会亲眼看到它随训练量移动——**这是本章“责任在模型”这个结论最直接的体感**。

6.26.3 练习三：验证网格理论

本章 §6.5 给了一张表：σ 均匀 10 步是 0.0228， $\log\sigma$ 均匀是 0.4201。**请在没有完美去噪器的情况下复现这个差距**。

做法： 1. 用 `run_ch6_case.py` 里那个学出来的模型 2. 实现两个网格：`t` 均匀（现有）和 $\log\sigma$ 均匀 3. 在 10 / 20 / 40 步下各跑一次

你会遇到的麻烦：$\log\sigma$ 网格需要把 σ 映射回 t， 而学出来的模型只接受整数 t。**你需要插值，或者取最近的整数 t**。

这个“麻烦”本身是重点：**采样器的数学形式（连续 σ）和模型的接口（离散 t）之间有一个缝隙**。 所有实现都要在这个缝隙上做某种折中，**而折中的方式会影响结果**。

进一步：如果你把 t 取整，$\log\sigma$ 网格的优势还剩多少? **这个数字告诉你“理论收益”和“工程收益”的差**。

6.26.4 练习四：实现 PLMS，并证明它值

本章没有实现 PLMS，因为它的核心思想一句话就能说完。**但“一句话”和“能跑”之间的距离，正是这个练习要量的**。

要求： 1. 实现二阶 PLMS（Adams-Bashforth 两步）： $\hat{x}_{k+1} = \hat{x}_k + (\sigma_{k+1} - \sigma_k)\left(\frac{3}{2}\varepsilon_k - \frac{1}{2}\varepsilon_{k-1}\right)$ 2. 第一步用欧拉起步 3. 在完美去噪器上测 10 / 20 / 40 步 4. **和 Heun 比较，但按 NFE 对齐**

预期：PLMS 在同 NFE 下应该接近 Heun，但**在步数很少时更差**（因为第一步只能是欧拉）。

如果你发现 PLMS 明显更好，先检查一件事：**你的 σ 网格是均匀的还是非均匀的?** 多步法的系数假设步长变化平缓，非均匀网格上直接用会不稳定。

6.26.5 练习五：写一个采样器诊断器

这是最接近真实工作的一个。

把本章六种失败模式做成一个自动检查器：

```
class SamplerDiagnoser:
    """在采样前后自动跑一组检查，输出诊断报告。"""

    def check_consistency(self, model, sch_train, sch_sample):
        """失败六：训练/采样调度一致性。"""
        # 比较两个 alpha_bar 数组，不相等就报错

    def check_order_effective(self, sampler, model, x, steps):
        """失败二：二阶是否真的生效。"""
        # 同一输入跑一阶和二阶，逐位比较；相同就报警

    def check_step_sweep(self, sampler, model, x, steps_list):
        """失败一/三：多档位一致性。"""
        # 跑多个步数，报告 std 随步数的走向
```

```
    def check_clamp_rate(self, traj):
        """失败五：clamp 命中率。"""
        # 命中率 > 50% 报警

    def check_eta_sensitivity(self, sampler, model, x, steps, etas):
        """失败四：eta 的稳定性。"""
        # 报告 max|x| 随 eta 的变化；eta=0 反而最大就报警
```

要求：每个检查都要返回 `(通过/警告/失败, 一句人话解释)`， **不能只返回布尔值**——诊断器的价值在于“告诉人下一步查什么”。

做完这些练习之后

五个练习加起来大概三四个小时。 它们会给你：

- **练习一、二**：一个属于你自己模型的“临界步数”——**不是本章的那个数**
- **练习三**：对“理论收益 vs 工程收益”的体感
- **练习四**：一次“多步法在非均匀网格上不稳定”的亲历
- **练习五**：一套能直接用在下一个项目上的工具

如果只做一个，做练习二。它最便宜，而且它直接回答本章最核心的问题：**“我的采样器炸了，是代码还是模型?”**

如果做两个，加练习五。因为它把本章的所有诊断动作固化成代码—— **下次遇到问题，你不用重新想一遍。**

6.27 符号表与公式速查

本章的记号冲突比前五章加起来还多。 这张表是给回查用的。

6.27.1 符号

符号	含义	取值范围	容易和什么混
β_t	单步噪声方差	10^{-4}–0.02	—
α_t	$1-\beta_t$	≈ 1	$\bar{\alpha}_t$
$\bar{\alpha}_t$	$\prod_{s\le t}\alpha_s$	$1 \to 4\times10^{-5}$	α_t
σ_t（**本章**）	$\sqrt{1-\bar{\alpha}_t}/\sqrt{\bar{\alpha}_t}$	$152 \to 0$	DDIM 论文的 σ_t
σ_t（**DDIM 论文**）	注入噪声的幅度	0–$\tilde{\beta}_t^{1/2}$	本章的 η
η	注入噪声的比例	$[0,1]$	DDIM 论文的 σ_t
$\tilde{\beta}_t$	DDPM 后验方差	—	β_t
ε_θ	模型输出的噪声预测	$\approx \mathcal{N}(0,I)$	真值 ε
$\hat{x}_0$	由 x_t 解出的数据估计	数据尺度	真值 x_0
$\hat{x}$	$x_t/\sqrt{\bar{\alpha}_t}$	σ 尺度	x_t
λ	$\log(\alpha/\sigma)$	$-\infty \to +\infty$	σ
NFE	函数求值次数	—	步数

读这张表的方法：如果你在别处看到 σ_t，**先确认是哪一种**。 这是本章唯一一个“同一个符号在两个主流文献里意思相反”的地方。

6.27.2 公式

名字	式子	在哪一节
前向边缘	$q(x_t \mid x_0) = \mathcal{N}(\sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t)I)$	第 5 章
DDIM 更新	式 (6.23)	§ 6.3
DDIM 展开式	式 (6.25)	§ 6.3
ε–score 关系	式 (6.44)	§ 6.4
概率流 ODE	式 (6.47)	§ 6.4
σ 空间流	式 (6.43)	§ 6.4
Heun 更新	$k_1 = \varepsilon(\hat{x}, \sigma_t)$; $\hat{x}_{\text{pred}} = \hat{x} + hk_1$; $\hat{x}' = \hat{x} + \frac{h}{2}(k_1 + k_2)$	§ 6.5
η 注入	$\sigma_{\text{inj}} = \eta\sqrt{\sigma_t^2 - \sigma_{t-1}^2}$	§ 6.9

6.27.3 三条“一句话”

1. **DDIM = 概率流 ODE 在 σ 空间的显式欧拉。**
2. **DDPM = 同一个 σ 族在 $\sigma_t^2 = \tilde{\beta}_t$ 处的取值。**
3. **阶数解决“精度”，非均匀网格解决“刚性”，两者不能互相替代。**

6.28 关键术语中英对照

这一行是给你读论文用的。 同一个概念在中文教材、英文论文、代码库里有三个名字， **这一栏把它们对上。**

中文	英文	代码里常见名	说明
去噪器	denoiser	`unet`, `model`	输出 ε 或 x_0
噪声预测	noise prediction	`prediction_type="epsilon"`	本章默认
数据预测	data prediction	`prediction_type="sample"`	DPM-Solver++ 用它
速度预测	velocity prediction	`prediction_type="v_prediction"`	流匹配 / SD3
无分类器引导	classifier-free guidance	`guidance_scale`	第 7 章
调度器	scheduler	`scheduler`	注意：它同时管网格和系数
时间步	timestep	`timesteps`, `sigmas`	`sigmas` 是 σ 参数化
函数求值次数	number of function evaluations	`nfe`, `num_fn_evals`	**不是 `num_inference_steps`**
概率流 ODE	probability flow ODE	—	本章 § 6.4
得分匹配	score matching	—	本章 § 6.4
非马尔可夫	non-Markovian	—	DDIM 的前向过程
后验方差	posterior variance	`variance_type="fixed_small"`	DDPM 的 $\tilde{\beta}_t$

中文	英文	代码里常见名	说明
截断	clamping	`.clamp()`	作用在 $\hat{x}_0$ 上
蒸馏	distillation	`distilled`, `lcm`	要重训
整流流	rectified flow	`flow`	路径拉直

⚠ 两个最容易对错的词

① **`scheduler`**
在 `diffusers` 里，`scheduler` **不只是“噪声调度”** —— 它还包含**采样网格**和**系数**。**换一个 scheduler，你同时换了三样东西。**
所以“换个 scheduler 试试”不是一个受控实验。
② **`num_inference_steps`**
它的名字像“推理步数”，**但它是循环次数，不是 NFE**。二阶 scheduler 的 20 步是 40 NFE。
在写对比表时，永远用 NFE。（§ 6.25）

6.29 五个常见误解

这五个说法，每一个都在教材、博客或论文里出现过。它们都错。而且每一个的错误方式不一样。

6.29.1 误解一：“DDIM 是对 DDPM 的近似”

错在哪：DDIM 不近似 DDPM。**它们是同一个 σ 族的两个端点，对应两条不同的采样路径，收敛到同一个分布。**
为什么这个误解流行：因为 DDIM 常被介绍成“加速版 DDPM”，而“加速”暗示“近似”。
正确的说法：**DDPM 和 DDIM 都是精确的**（在模型完美的前提下），**它们的区别不是精度，是鲁棒性和可复现性。**
这个区别的实践含义：如果你把 DDIM 当成“近似”，你会认为“步数少 = 精度低”，于是“加步数”看起来总是安全的。**本章的案例证明它不安全。**

6.29.2 误解二：“η 控制随机性，所以 η 越大越多样”

错在哪：η 确实控制每步注入的噪声量，**但它不控制最终分布的多样性。**
因为所有 η 收敛到同一个分布。 η 改变的是**路径**，不是**终点分布**。
那多样性从哪来？ 从**初始噪声** $x_T \sim \mathcal{N}(0, I)$。**同一个 η、不同的种子，多样性是一样的。**
为什么这个误解流行：因为在**步数不足**时，η 大的采样器看起来“结果更不一样”——**但那是因为它在更早的阶段就停止了，不是因为它的目标分布更宽。**
实践含义：如果你需要多样性，**换种子，不要调 η。** 调 η 只会改变稳定性和质量。

6.29.3 误解三：“二阶方法每步算两次，所以慢一倍”

错在哪：慢一倍是**每步**的成本，不是**任务**的成本。
正确的比较是“同样的 NFE”：Heun 20 步 = 40 NFE，欧拉 40 步 = 40 NFE。**在这个比较下，Heun 通常更快达到同样的精度**（本章实测：20 步 Heun 的误差 2.0%，40 步欧拉的误差 5.5%）。
为什么这个误解流行：因为“步数”是显式的循环变量，而 NFE 要自己数。
实践含义：**在同一张表里比较不同阶数的方法时，横轴必须是 NFE。** 否则你会系统性地低估高阶方法。

6.29.4 误解四：“确定性采样更稳定，因为它没有随机波动”

错在哪：把“可复现”当成了“稳定”。

- **可复现**：同样的输入 → 同样的输出
- **稳定**：输入的微小扰动 → 输出的微小变化

确定性采样有第一个，没有第二个。而且**恰恰因为它没有随机性，它失去了唯一的误差吸收机制。**
本章实测：$\eta = 0$ 时 50 步的 max 是 1.33×10^3，$\eta = 1$ 时是 1.18。**差三个数量级。**
为什么这个误解流行：因为在**模型很好**的时候，两者的差别很小。**只有模型有误差时，这个区别才暴露。**

6.29.5 误解五："步数越多，结果越好"

错在哪：它只在"采样器稳定"的前提下成立。
本章的案例：25 步正常、50 步半炸、100 步全炸。**在这个模型上，步数越多越糟。**
更准确的说法：步数增加会同时放大"离散化误差的减小"和"模型误差的累积"。前者随步数下降，后者随步数上升。**最优步数是两者的交点。**
本章没有测出这个交点（因为模型误差太大，交点落在很低的步数上），**但它的存在可以从数据里推出来**：完美去噪器下，误差随步数单调下降；学出来的模型下，误差随步数单调上升。**两者之间必然有一个交叉。**

> 这五个误解的共同点
>
> **它们都把"两个不同的量"当成了一个。**
>
> | 误解 | 被混淆的两个量 |
> |---|---|
> | 一 | 精度 vs 鲁棒性 |
> | 二 | 路径的随机性 vs 分布的多样性 |
> | 三 | 每步成本 vs 任务成本 |
> | 四 | 可复现 vs 稳定 |
> | 五 | 离散化误差 vs 模型误差 |
>
> **本章的每一节，本质上都在拆开一对被混淆的量。**
> **这也是"机制"这个词在这本书里的意思：不是"知道公式"，是"知道公式里每一项各自负责什么"。**

6.30 白纸测验：五个必须能自己推出来的式子

读完一章和掌握一章的区别，是能不能在白纸上重推一遍。
下面五道题，每题给了"要用的工具"和"答案的骨架"。建议先盖住答案，自己写一遍。

6.30.1 第一题：证明 σ 族保住边缘分布

题目：设 $x_t = \sqrt{\bar\alpha_t}x_0 + \sqrt{1-\bar\alpha_t}\,\varepsilon$，且 $x_{t-1} \mid x_t, x_0$ 服从

$$\mathcal{N}\Big(\sqrt{\bar\alpha_{t-1}}x_0 + \sqrt{1-\bar\alpha_{t-1}-\sigma_t^2}\,\varepsilon,\ \sigma_t^2 I\Big) \tag{6.73}$$

证明 x_{t-1} 的边缘分布是 $\mathcal{N}(\sqrt{\bar\alpha_{t-1}}x_0, (1-\bar\alpha_{t-1})I)$。
工具：高斯分布的线性变换；$\mathrm{Var}[aX+b] = a^2\,\mathrm{Var}[X] + b^2$。
骨架：1. 均值：ε 的均值为 0，所以均值就是第一项 2. 方差：ε 的系数平方 $\times\ (1-\bar\alpha_t)$，再加 σ_t^2 3. 化简：$\frac{1-\bar\alpha_{t-1}-\sigma^2}{1-\bar\alpha_t}(1-\bar\alpha_t) + \sigma^2 = 1-\bar\alpha_{t-1}$
自检：如果第 3 步化简不出来，说明你把 ε 的方差记成了 1 而不是 $1-\bar\alpha_t$。

6.30.2 第二题：从 ε 推出 score

题目：已知 $p_t(x_t \mid x_0) = \mathcal{N}(\sqrt{\bar\alpha_t}x_0, (1-\bar\alpha_t)I)$，证明 $\varepsilon = -\sqrt{1-\bar\alpha_t}\nabla_{x_t}\log p_t$。
工具：高斯对数的梯度 $\nabla_x \log \mathcal{N}(x;\mu,\Sigma) = -\Sigma^{-1}(x-\mu)$。
骨架：1. $\log p_t = -\frac{1}{2(1-\bar\alpha_t)} \parallel x_t - \sqrt{\bar\alpha_t}x_0 \parallel^2 +$ 常数 2. 求梯度：$-\frac{x_t - \sqrt{\bar\alpha_t}x_0}{1-\bar\alpha_t}$ 3. 而 $\varepsilon = \frac{x_t - \sqrt{\bar\alpha_t}x_0}{\sqrt{1-\bar\alpha_t}}$ 4. 两者相差一个 $-\frac{1}{\sqrt{1-\bar\alpha_t}}$

自检：符号。**如果你得到的是正号，说明你在第 1 步漏了一个负号。**

6.30.3 第三题：证明 DDIM 是 σ 空间的欧拉法

题目：在 $\hat{x}=x/\sqrt{\bar{\alpha}}$、$\sigma=\sqrt{1-\bar{\alpha}}/\sqrt{\bar{\alpha}}$ 下，证明 DDIM 的更新式等价于 $\hat{x}_{k+1}=\hat{x}_k+(\sigma_{k+1}-\sigma_k)\varepsilon$。
工具：$\hat{x}_0=(x_t-\sqrt{1-\bar{\alpha}_t}\varepsilon)/\sqrt{\bar{\alpha}_t}=x_t/\sqrt{\bar{\alpha}_t}-\sigma_t\varepsilon=\hat{x}_t-\sigma_t\varepsilon$。
骨架：1. 把 $\hat{x}_0$ 写成 $\hat{x}_t-\sigma_t\varepsilon$ 2. DDIM 更新：$x_{t-1}=\sqrt{\bar{\alpha}_{t-1}}\hat{x}_0+\sqrt{1-\bar{\alpha}_{t-1}}\varepsilon$ 3. 两边除以 $\sqrt{\bar{\alpha}_{t-1}}$：$\hat{x}_{t-1}=\hat{x}_0+\sigma_{t-1}\varepsilon$ 4. 代入第 1 步：$\hat{x}_{t-1}=\hat{x}_t+(\sigma_{t-1}-\sigma_t)\varepsilon$
自检：第 1 步是关键。**它说明“DDIM 的一步”就是“在 $\hat{x}$ 空间里沿 ε 走 $\Delta\sigma$”。**

6.30.4 第四题：证明 $\eta=1$ 退回 DDPM

题目：证明 $\sigma_t^2=\tilde{\beta}_t=\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}\beta_t$ 时，q_σ 就是 DDPM 的后验。
工具：$\bar{\alpha}_t=\bar{\alpha}_{t-1}\alpha_t$，$\alpha_t=1-\beta_t$。
骨架：1. 算 $1-\bar{\alpha}_{t-1}-\tilde{\beta}_t$，通分 2. 用 $\bar{\alpha}_t=\bar{\alpha}_{t-1}(1-\beta_t)$ 化简分子 3. 得到 $\frac{\alpha_t(1-\bar{\alpha}_{t-1})^2}{1-\bar{\alpha}_t}$ 4. 开方后，ε 的系数是 $\frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}$ 5. 而 DDPM 后验里 x_t 的系数正是这个
自检：第 2 步的因式分解 $1-\bar{\alpha}_t-\beta_t=\alpha_t(1-\bar{\alpha}_{t-1})$。 **这个恒等式不成立，后面的化简全错。**

6.30.5 第五题：证明概率流 ODE 保住边缘分布

题目：证明 $\frac{dx}{dt}=f-\frac{1}{2}g^2\nabla\log p_t$ 对应的连续性方程与前向 SDE 的 Fokker-Planck 相同。
工具：连续性方程 $\partial_t p=-\nabla\cdot(vp)$； 恒等式 $p\nabla\log p=\nabla p$。
骨架：1. 把 $v=f-\frac{1}{2}g^2\nabla\log p$ 代进连续性方程 2. 第二项变成 $\frac{1}{2}g^2\nabla\cdot[p\nabla\log p]=\frac{1}{2}g^2\nabla\cdot\nabla p=\frac{1}{2}g^2\Delta p$ 3. 得到 $\partial_t p=-\nabla\cdot[fp]+\frac{1}{2}g^2\Delta p$ 4. 这正是前向 SDE 的 Fokker-Planck
自检：第 2 步的恒等式。**如果你把 $p\nabla\log p$ 写成了 $\nabla p/p$，符号会错。**

> 做完这五题之后
>
> **如果你五题都能不看答案写出来，这一章的数学部分就过关了。**
> **如果第三题卡住，回去读 §6.4 的最后一小节**——它是整章的中心。**如果第五题卡住，回去读 §6.4 的“路线一”**——它是最短的一条证明。
> **注意这五题有一个共同的结构：每一题都是“把 A 写成 B，然后看 B 满足什么”。**
>
> - 第一题：把 x_{t-1} 写成 ε 的线性函数
> - 第二题：把 $\log p_t$ 写成二次型
> - 第三题：把 $\hat{x}_0$ 写成 $\hat{x}_t-\sigma_t\varepsilon$
> - 第四题：把 $\tilde{\beta}_t$ 写成 $\beta_t/(1-\bar{\alpha}_t)$ 的形式
> - 第五题：把 v 写成 f 加一个梯度
>
> **“换一种写法”是这一章所有推导的共同动作。 而每一种写法，对应一个不同的数值性质** ——这就是本章从数学走到工程的那条线。

6.31 附：一份采样器评测报告的模板

本章反复说“要按 NFE 对齐、要多档位、要控制变量”。这一节把它变成一个可以直接填的模板。
下次你要向团队说明“我们该换采样器吗”，填完这张表就有答案了。

6.31.1 一、基本信息

```
模型：_____（参数量 / 训练步数 / 训练调度）
数据：_____（分辨率 / 数据集 / 样本数）
硬件：_____（GPU / batch size / 精度）
评估日期：_____
```

这五项缺一项，结论就不可迁移。 尤其是**训练调度**——换一个调度，所有采样器的排序都可能变（§6.17）。

6.31.2 二、候选与 NFE 对齐

采样器	每步求值	步数	**NFE**	实现来源
DDIM	1	20	20	本章 `sample()`
Heun	2	20	40	本章 `sample()`
DPM-Solver++	1	20	20	`diffusers`
…				

注意最后一列。 不同来源的实现，网格和末步处理可能不同。**如果实现来源不同，你比较的可能是“实现”而不是“算法”。**

6.31.3 三、质量指标（至少三项）

采样器	NFE	主指标	多样性指标	稳定性指标
		FID / 解析误差	种子间方差	多档位一致性

为什么至少要三项：

- **主指标**告诉你“像不像”
- **多样性指标**告诉你“有没有塌缩”（§6.25 陷阱二）
- **稳定性指标**告诉你“换个步数会不会炸”（§6.25 陷阱三）

只有主指标的评测，本章的案例已经证明它会骗人。

6.31.4 四、结论的写法

不要写：“采样器 A 比 B 好。”
要写：

> 在 NFE=20–40、模型误差约 % **的条件下，A 的主指标比 B 好** %，但多样性低 __%。**在 NFE<20 时，A 出现发散（max|x| > __），B 没有。建议**：如果预算 ≥ __ NFE 且不需要多样性，选 A；否则选 B。

这个写法包含了四个约束：成本、模型条件、质量、稳定性。**缺任何一个，结论都会被误用。**

> 一个反直觉的建议
>
> **在报告里专门写一节“我们的评测不能回答什么”。**
> 比如：
>
> - **“我们没有测 > 100 NFE 的情况，所以不知道 A 在高预算下是否仍然更好。”**
> - **“我们只在 256×256 上测过，结论不能外推到 1024。”**
> - **“我们的模型误差约 4%，误差更大的模型上排序可能反转。”**
>
> **这一节会让你的报告更可信，不是更不可信。因为它告诉读者“边界在哪”——而边界，正是工程决策最需要的信息。**

6.32 附：量化和采样器的交互

这一节很短，但它是一个真实部署里绕不开的问题。

6.32.1 量化改变了什么

把模型从 fp16 量化到 int8，会带来两个变化：

1. **权重的精度下降** → ε_θ 的误差增大
2. **激活的精度下降** → 在高 σ 处（激活值跨度大）误差尤其大

第二个变化，正好打在本章最敏感的地方。
本章实测：$t = 999$ 处 $|\hat{x}_0|$ 平均 12.8，而 $1/\sqrt{\bar{\alpha}_t} = 157$。**这两件事叠加，意味着 t 大的区间对数值误差极度敏感。**

6.32.2 一个可测的判据

如果你在做量化部署，做这个实验：

1. 固定采样器（DDIM，20 步）
2. 分别用 fp16 模型和 int8 模型采样
3. **记录每一步的 $|\hat{x}_0|$ 和 clamp 命中率**

预期：int8 模型的 clamp 命中率会**显著更高**，而且**集中在 t 大的前几步**。
如果命中率从 60% 涨到 90%，说明量化已经破坏了高噪声区的估计。**这时候的正确动作不是“调 clamp”，是**：- 对 t 大的区间保留 fp16（混合精度）- 或者用 $\eta > 0$ 的采样器，让噪声吸收量化误差

6.32.3 一个反直觉的结论

量化对采样器的影响，比对模型本身的影响更大。
因为采样器在 t 大的区间会放大误差（157 倍），而量化误差也正好在 t 大的区间最大（激活跨度大）。**两个放大叠加。**
所以“模型量化后精度掉了 1%”这句话没有意义—— 你要问的是“在 t 大的区间掉了多少”。

6.33 延伸阅读

6.33.1 ① DDIM 的原始论文（2020）

Song, Meng, Ermon, Denoising Diffusion Implicit Models。arXiv:2010.02502。
读它能得到什么：本章 §6.3 的完整来源。论文最有价值的部分不是那个更新公式，而是**“非马尔可夫前向过程”的构造**—— 它证明了反向过程不需要和一条特定的前向链绑定。**这个想法比 DDIM 本身更通用**，后来很多工作都沿着它展开。
读的时候注意什么：

- **论文里的 σ_t 公式和本章写的略有不同**（他们用了另一个参数化），但 **$\eta = 0$ 和 $\eta = 1$ 两个端点是一样的**。对不上号时先核对端点。
- **论文报告 DDIM 在 20–100 步就能达到 DDPM 1000 步的质量。这个结论成立的前提是模型足够好** —— 本章的案例正好是反例。论文没有讨论“模型不准时会怎样”，**这是本章补上的那一块**。
- **它的实验都在图像上**（CIFAR-10、CelebA），本章在 2D 上重做了一遍，**得到的定性结论一致，定量结论不能直接比**。

6.33.2 ② DPM-Solver（2022）

Lu et al., DPM-Solver: A Fast ODE Solver for Diffusion Probabilistic Model Sampling in Around 10 Steps。arXiv:2206.00927。
读它能得到什么：本章 §6.5 那套“用更好的数值方法”的思路，在这篇论文里被做到了极致。
它的核心贡献是“利用 ODE 的解析结构”：概率流 ODE 在 σ 空间里是半线性的，**对这类方程，解可以写成一个指数因子乘上一个积分**。把那个积分用泰勒展开处理，就得到了专门的二阶、三阶离散化。
本章的 Heun 是这条路上最简单的版本（通用二阶方法）。DPM-Solver 比它更精细，因为它“知道”方程长什么样。
读的时候注意什么：

- **它的“10 步”是在好模型上测的**。模型差的时候，**高阶方法同样救不了** —— 本章 §6.5 那张表已经说明，10 步时 Heun 自己也错 7.5%。

- **论文里有多套系数表**（对应不同的阶数和不同的噪声调度）。**抄错一套就会得到比一阶还差的结果**，而且不会报错。
- **它的实现依赖 $\lambda_t = \log(\bar{\alpha}_t/(1-\bar{\alpha}_t))$ 这个换元**，和本章的 σ 换元是等价的（差一个对数）。**两个记号对照着看会更容易懂**。

6.33.3 ③ 关于“$\mathcal{L}_{\text{simple}}$ 为什么有效”的讨论

Ho, Jain, Abbeel, Denoising Diffusion Probabilistic Models。arXiv:2006.11239，§3.4。

原始的观察：作者在论文里明确写了“我们试了带权重的版本，效果更差，所以用了简化版”。**但他们没有给出解释**。

读的时候注意什么：

- **论文的措辞是“empirically better”** —— 它是一个实验观察，不是一个定理。**后来的工作（包括本书这一章）都在试图解释它**，但至今没有公认的完整答案。
- **本章给出的解释（“权重把学习压力推向了学不会的小 t”）是一个假说**，在一个 2 维小模型上得到了支持，**但没有在大模型上验证过**。
- **后续有工作重新引入了加权**（比如按 SNR 加权），在高分辨率上取得收益。**这说明“扔掉权重”不是普适最优，而是特定规模下的选择**。

6.33.4 ④ 关于采样器评估的方法论

本章反复强调“比求值次数，不比步数”。这个做法在文献里并不统一。

很多论文报的是“步数”，因为那样数字好看（“我们 10 步就达到了 xxx”）。**但二阶方法的 10 步是 20 次求值**，和别人的 10 步不是一回事。

读采样器论文时，先找这三样：

1. **每步调用几次模型？**（一阶 1 次，二阶 2 次，有些高阶方法 3 次）
2. **在什么模型上测的？**（模型质量决定了结论能不能迁移）
3. **比的是 FID 还是似然？**（两者经常给出相反的排序）

本章的立场是：把“求值次数”当作横轴，把“与真实分布的差距”当作纵轴。在这个坐标系里，所有方法都能放在同一张图上比较。**这个做法更慢（要跑很多配置），但结论更硬**。

6.33.5 ⑤ 从 DDPM 到一步生成

如果你想看这条路的终点，可以读一致性模型（Song et al., 2023, arXiv:2303.01469）和分布匹配蒸馏（Yin et al., 2024, arXiv:2311.18828）。

它们的目标都是“1–4 次求值生成一张图”。

读的时候注意什么：

- **它们都需要额外的训练阶段**（本章 §6.16 那张表里的第三类）。**不是“换个采样器”就能得到的**。
- **一步生成的质量通常低于多步**，而且**多样性会下降**。**“快”和“好”在这里仍然是一个权衡，只是权衡的位置变了**。
- **这些方法对本章的读者来说是一个“知道存在”就够的东西** —— 它们需要的数学工具（得分匹配、变分推断的进一步展开）超出了本书的范围。

6.33.6 ⑥ 流匹配与整流流：换一条路，而不是换一辆车

Lipman et al., Flow Matching for Generative Modeling。arXiv:2210.02747。**Liu, Gong, Liu**, Rectified Flow。arXiv:2209.03003。

这两篇代表了一个不同的思路：与其“训练一个去噪器，再想办法把它的 ODE 解快”，**不如直接训练一个“走直线的速度场”**。

为什么直线更好？因为 ODE 的数值误差来自“路径弯曲”。**如果路径是直线，欧拉法一步就走对了** —— 一阶方法在直线上是精确的。

读的时候注意什么：

- **它们不是“采样器的改进”**，是**训练目标的改变**。你不能把一个已经训好的 DDPM 模型直接拿来做整流流。

- **“直线”是训练出来的，不是假设的。**Rectified flow 有一个“回流”（reflow）步骤来逐步把路径拉直，**每一步都要重训。**
- **本章的 σ 空间视角在这里仍然适用：**流匹配的 ODE 也是 $d\hat{x}/d\sigma =$ 某个网络输出，只是那个输出不再是 ε。

6.33.7 ⑦ 随机性为什么有用：一个还没有定论的问题

本章的实测支持“$\eta > 0$ 更稳”，但本章给的解释（“噪声吸收误差”）是一个假说。
要读的：- **Xu et al.**, Restart Sampling for Improving Generative Processes（2023）—— 系统地研究了“什么时候重新注入噪声”。- **关于 stochastic localization / 随机插值的工作** —— 从另一个角度解释 为什么随机过程比确定性 ODE 更鲁棒。
读的时候注意什么：

- **“η 越大越好”是不对的。**本章的 100 步扫描里，$\eta = 1$ 的最大值是 6021，仍然不等于数据尺度——**随机性压住了发散，但没有完全修复模型误差。**
- **随机性有代价：**同样的步数下，$\eta > 0$ 的有效信息量更少（一部分步数被用来“重新加噪”）。**所以它不是一个免费的稳定性。**

6.33.8 ⑧ 代码

如果你想看工业级实现：

- **`diffusers`（HuggingFace）**的 `schedulers/` 目录 —— 十几个采样器，接口统一，**是最好的“对照阅读”材料。**
- **`k-diffusion`（Katherine Crowson）** —— 实现更贴近论文，注释少但代码短，**适合验证你对公式的理解。**

读代码时注意：

- `diffusers` 里每个 scheduler 都有 `set_timesteps()`，**它同时决定了网格和系数。这是最容易被忽略的一处**（§6.5）。
- `diffusers` 的 `SchedulerMixin` 用 `num_inference_steps` 这个名字，**但它指的是循环次数，不是 NFE。**（§6.25 的第一个陷阱）

6.34 本章没有回答的问题

一本教材的可信度，一半来自它讲清楚的东西，一半来自它承认没讲清楚的东西。
下面五个问题，本章没有答案。**每一个都附了“什么样的实验能回答它”。**

6.34.1 问题一：学出来的模型为什么会退化成 $\varepsilon \approx 1.9x$？

本章测到了这个现象（§6.2 的表），也证明了它不是“均匀偏差”能解释的（§6.8 的六种误差模型全部不发散）。
但没有解释“为什么是 1.9”。
可能的实验：在训练分布外系统扫描 x 的尺度，测 $\varepsilon_\theta(cx, t)$ 与 c 的关系。**如果它在一个区间内是线性的、斜率为 1.9，那就找到了“放大器的增益”。**

6.34.2 问题二：能不能构造一个误差模型，复现 10^7？

本章试了六种，全部失败。这是本章最大的缺口。
可能的实验：不做“解析误差模型”，直接把学出来的模型在某一步**替换成“前一步的输出”**（相当于人为制造一个特定的错误），看轨迹会不会以同样的速率增长。**如果能复现，就可以逐步逼近真实的错误结构。**

6.34.3 问题三：η 的最优值和模型误差是什么关系？

本章有 η 的扫描数据（§6.9），**但只有两个步数档位、一个模型。**

可能的实验：固定 η，扫描“模型误差 b”（用解析去噪器 + 注入误差），测“多久发散”；再固定 b，扫描 η。**如果存在 $\eta_{\text{crit}} \propto b$ 这样的关系，就能给出一个可操作的公式。**

注意：本章的实验 6-2 已经证明，“注入均匀误差”不会发散—— **所以这个实验必须用“结构化的误差”，而本章还不知道那种结构是什么**（见问题二）。**这两个问题是连在一起的。**

6.34.4 问题四：$\mathcal{L}_{\text{simple}}$ 的解释在大模型上成立吗?

本章的解释（权重把学习压力推向学不会的小 t）只在 2 维、5 万参数上验证过。

可能的实验：在 CIFAR-10 上训练两个模型（带权重 / 不带权重），**测每个 t 段的 loss 和梯度相干性**，看本章的模式是否重复。

这个实验的成本：单卡几小时到一天。**它是本章最值得补的一个。**

6.34.5 问题五：采样器的“阶数收益”在真实模型上还剩多少?

本章所有阶数实验都在完美去噪器上做的（为了排除模型误差）。

但真实模型有误差，而误差会淹没高阶方法的精度收益。在什么误差水平下，二阶就不划算了?

可能的实验：固定采样器（Heun vs Euler），扫描模型误差 b（用解析去噪器 + 可控误差），**画出“二阶优势随 b 衰减”的曲线。**

> 为什么要把“没做到”写出来
>
> **因为不写出来，你会以为本章的结论比你实际拥有的更强。**
>
> **一个具体的例子**：如果你读完 §6.2 就记住“模型误差被 157 倍放大 → 爆炸”，**你带走的其实是一个本章没能证明的因果链**—— 它解释得通，但六种误差模型的实验都不支持“误差大小”这个说法。
>
> **把缺口标出来，你才知道哪一步可以信、哪一步要自己验。**
>
> **这也是本书对“教材”这件事的理解：教材不是“已经解决了的问题的合集”，是“一个领域目前的确定性和不确定性”的切片。**

6.35 结语：为什么这一章要这样写

这一章有一个不太寻常的结构：它从一个失败开始，中途承认了一次推导失败，末尾列了五个没回答的问题。

多数教材不这样写。这一节解释为什么。

6.35.1 因为“正确的知识”和“能用的知识”不是一回事

如果这一章按常规写法，大概是这样：

1. 介绍 DDIM 的论文
2. 推导非马尔可夫前向过程
3. 给出更新公式和代码
4. 展示在标准数据集上的结果

这样写，你会得到一套“正确的知识”—— 它经得起考试，经得起对照论文。

但它不会让你有能力判断“为什么我的模型上它炸了”。

因为真实世界的问题，从来不是“这个公式对不对”，而是“它在什么条件下不成立”。而“条件”这件事，只有通过“让它失败”才能看见。

所以这一章花了大量篇幅在一个失败上，而且把失败的每一步都写了出来，包括走错的那两步。

6.35.2 三个“不”字

不把结论说得比证据强。

本章最典型的例子是“误差被放大 157 倍”。这个说法单独看是对的，但它容易让人以为“误差大就会炸”。**六种误差模型的实验否定了这个推论。**所以本章把结论收窄成**“均匀误差不会发散；发散需要特定的误差结构”**，并列出了**没能做到的那一步**（构造出那个结构）。

这种写法会让结论看起来“弱”。但它准确。

不留一个“看起来对”的推导。

§6.4 里，我最初写了一版反向 SDE 的推导，试了三次都没对上 Fokker-Planck。最终换成了“连续性方程”那条路。 书上留下的是成功的那条，但这一节告诉你：另一条我失败了。

为什么要说这个？ 因为**“推导一次成功”是一个假象。** 真实的推导就是试错，**而试错时唯一能依靠的，是“代回去验证”这个动作。**

不把“没做”包装成“不重要”。

本章列了五个没回答的问题，每一个都附了“什么实验能回答它”。这不是谦虚，是精确。 你在读完这一章之后， 应该清楚地知道**哪些结论可以直接用、哪些需要你自己验证。**

6.35.3 一个可以带走的判断

如果你只从这一章带走一个判断，希望是这个：

一个方法的失效边界，比它的有效性更值得知道。

因为有效性写在论文摘要里，而失效边界写在没人读的附录里、 或者根本不写。

本章做的，就是把一个失效边界挖出来，测清楚， 然后诚实地告诉你：这里还有一块我没挖到。

6.35.4 最后

这一章的所有数字，都可以用 `notebooks/` 里的五个脚本重新生成。 所有实验，都可以用 `experiments/` 里的两个交互页面自己拖一遍。

如果你发现某个数字复现不出来，那是本书的错，请当作 bug。

如果某个解释你觉得不对，去做一个实验检验它—— 这一章的所有结论，都是这样来的。

而如果某个实验你做不出来，那可能不是你的问题。本章列出的五个未解问题，就是我自己也没做出来的那五个。

把这五个问题写在这里，不是为了交差，是为了让下一个做这件事的人， 不必从「不知道哪里没做」开始。

6.36 本章回顾

把这一章压缩成六句话。

一、DDIM 的正确性不取决于实现，取决于模型。 同一个 `ddim()` 函数，换成解析的完美去噪器就完全正常（20 步仍有 0.55，目标 0.63）； 换回学出来的模型就炸到 10^8。**代码没错，是模型在分布外退化成了放大器。**

二、放大分两步：先是 $1/\sqrt{\bar\alpha_t}$，再是模型自己。 在 $t = 999$ 处 $\hat{x}_0$ 被放大 157 倍，实测平均 $|\hat{x}_0| = 12.8$（clamp 前）， 对应约 6.6% 的模型误差，**86.5% 的坐标直接撞上 clamp。** 走出去之后，模型不再是去噪器——它返回 $\varepsilon_\theta \approx 1.9x$， 于是每一步乘 1.9，$1.9^{10} \approx 613$。**这两步缺一不可。**

三、DDPM 的随机性是稳定器，不是装饰。 η 从 0 调到 1，50 步下的输出从 std 13.8 变回 $[0.525, 0.320]$（数据是 $[0.5707, 0.3351]$）。 **“确定性 = 可复现”不等于“确定性 = 稳定”。**

四、$\mathcal{L}_{\text{simple}}$ 的谜有一个可测的部分。 ELBO 的权重在小 t 处极大（$t = 2$ 时 0.60，$t = 500$ 时 0.0055）， 而模型在那里的 loss 是 1.01（几乎等于“什么都没学到”）。 **权重把学习压力推向了收益最低的方向。** 但“梯度更噪”这个常见解释**在我们的模型上不成立**（小 t 的相干性 0.95，反而最高）。

五、“二阶”不等于“更快”，更不等于“写对了”。 Heun 在 20 步时误差 2%，一阶要 80 步——**这是真的进步。**但 10 步时 Heun 自己也错 7.5%，**而且我第一版“二阶”在数学上是个恒等式**，输出和一阶逐位相同。

六、采样器有五种不同的加速来源，代价各不相同。改公式（DDIM，免费但牺牲鲁棒性）、改数值方法（高阶，免费但更贵更复杂）、改模型（蒸馏，要重训）、改路径（流匹配，要改训练目标）、改网格（$\log\sigma$ 均匀，免费且收益最大）。 **遇到“快 10 倍”的方法，先问它属于哪一类。**

七、所有加速都有一个下界，而下界由模型决定。所需步数大致满足 $N \gtrsim L \cdot T$，其中 L 是模型输出对输入的敏感度、T 是 $\log\sigma$ 的跨度。**采样器只能改善常数，改不了 L。** 这就是为什么“更好的采样器”的收益会饱和，而“更好的模型”不会。

最后留一个问题

本章测出：**$\eta = 1$（完全随机）在 50 步下就能给出正确的分布，而 $\eta = 0$（完全确定）在 100 步下就炸了**。

那么“随机性”在这里到底起了什么作用？

一个可能的回答：**它是每次迭代后的一次“重新投影”**。模型预测的 ε_θ 有误差，这个误差让 x 偏离正确的流形；**注入的噪声把 x 推离流形一点点，然后下一步的 μ_θ 再把它拉回来——一推一拉，点在流形附近做随机游走，而不是沿着一个越来越偏的方向漂移**。

如果这个解释对，那么“需要多少随机性”应该和“模型误差有多大”直接相关。本章的 η 扫描数据支持这个方向（误差越大，越需要 η），**但没有做定量拟合**。

一个可做的实验：固定 η，扫描模型误差 b，测“多久发散”；再固定 b，扫描 η。**如果两者存在一个简单的定量关系（比如 $\eta_{\text{crit}} \propto b$），那这个解释就站住了**。

这个实验本章没做——它需要一个能精确控制误差的模型，而本章只有“训好的”和“解析的”两种。**这是第 7 章讲条件生成时会碰到的同类问题的一个缩影**。

下一章换个角度：本章处理的是“没有条件的生成”——模型学会生成“某种像数据的东西”，但你没法告诉它要生成什么。**第 7 章处理条件：文本提示是怎么影响生成的，CFG 为什么会让图“过饱和”，负向提示到底在做什么**。

7. 条件生成：CFG 到底是什么

本章地图

你在这里：第 5、6 章训练出的模型只会生成“某种像数据的东西”，你没法让它只画月亮、不画另一个月亮。本章解决“怎么让它听话”。**本章解决**：条件怎么进入模型；CFG 那个公式从哪来；引导强度 w 为什么有三段行为；标签丢弃为什么是 CFG 的前提而不是正则化；分类器引导为什么被取代。**前置**：第 5 章（前向过程、ε-预测）、第 6 章（DDPM/DDIM 采样器、η）。**读完能做**：正确实现 CFG 并选对 w；诊断“引导过强”造成的塌缩；解释 negative prompt 的数学来源。

7.1 本章的代码与数据

```
import torch, sys
sys.path.insert(0, "notebooks")
from tiny_diffusion import Config
from conditional_diffusion import (make_labeled_moons, train_cond,
                                   ddpm_cond, train_judge, metrics)

x, y = make_labeled_moons(4000, noise=0.08, seed=0)   # y ∈ {0,1}
model, sch, hist = train_cond(x, y, Config(max_steps=5000), drop=0.1)

samples = ddpm_cond(model, sch, torch.zeros(1000, dtype=torch.long), w=2.0)
```

复现本章全部数字：

```
cd notebooks
PY=/home/mpp/micromamba/envs/dlspace/bin/python
$PY run_ch7_case.py        # 五组实验，约 12 分钟
```

本章和第 5、6 章共用同一个双月数据集、同一套线性调度、同一个 ε-预测目标。唯一的改动是数据多了一个标签、模型多接收一个类别输入。**只有其他一切都相同，“引导”的效果才是一个干净的对照。**

文件	内容	生成脚本
`data/ch7_case.json`	CFG 扫描、负引导、无丢弃对照、分类器引导、分类器分段准确率	`run_ch7_case.py`

三个必须记住的数（后面每一节都会用到）：

- 数据标准差：$[0.5728, 0.3389]$（两个维度）
- 模型：50,434 参数，训练 5000 步，最终 loss 0.1242
- 裁判分类器（只在干净数据上训练，用来给生成样本打分）：准确率 **0.9978**

> **i** 为什么需要“裁判”
>
> 本章要说“引导让样本更像某一类”。**“像”必须有一个独立的判据**，否则就变成“我觉得像”。
> 做法是：在**干净的真实数据**上训练一个分类器，它的留出准确率是 0.9978 ——**它几乎不会看错真实数据**。然后用它给生成样本打标签，统计“生成的外月里，有多少被裁判认成外月”。
> **这个数不能证明生成质量**（裁判只学过真实数据，可能对生成样本的瑕疵不敏感），但它能干净地回答一个问题：**样本落在哪个类的区域里**。

7.2 一个不会听话的模型

先看无条件模型能做什么。

用第 5 章的模型采样 1000 个点，裁判给出的类别准确率是 **0.525**—— **接近抛硬币**。因为模型根本不知道“类别”这回事，它只是把两个月亮一起学了下来。

这不是模型不好。 它完美地完成了被要求的事：拟合 $p(x)$。 **问题是我们要的是 $p(x \mid c)$。**

条件生成的全部困难，可以压成一句话：

怎么把“我要外月”这个信息，塞进一个只学过“月亮”的模型里?

历史上主流做法有三条：

做法	条件怎么进去	要不要额外模型	代表
拼接/嵌入	作为模型输入的一部分	不要	本章的基线
分类器引导	借分类器的梯度推一把	**要一个噪声分类器**	Dhariwal & Nichol 2021
CFG	同时算“有条件”和“无条件”，取差	不要	Ho & Salimans 2022

本章先讲第一条（它是另外两条的基础），再把后两条推到同一个公式下比较。

7.3 条件怎么进模型：一个嵌入就够了

第 5 章的模型是 $\varepsilon_\theta(x_t, t)$。条件版本只多一个参数：

$$\varepsilon_\theta(x_t, t, c) \tag{7.1}$$

最省事的实现，是把类别嵌入加到时间嵌入上：

```
h = torch.cat([x, self.temb(t) + self.cemb(c)], dim=-1)
```

其中 `cemb` 是一个 `nn.Embedding(n_class + 1, hidden)`。

为什么是相加而不是拼接? 因为时间嵌入和类别嵌入都是“给网络的调制信号”，相加等价于“在同一个隐空间里叠加两个条件”。拼接也可以，但参数量更大，而且在本章这个 5 万参数的模型上没有可见收益。

注意 +1。多出来的那一行是**空类**（null class）—— 训练时以一定概率把标签换成它。**这一行是 CFG 的全部秘密**，下一节展开。

> ⚠ 空类不能用 −1
>
> `nn.Embedding` 不接受负数下标。把空类写成 `-1` 会在第一个 batch 就抛 `IndexError`；写成 `n_class` 才是对的。
>
> **更隐蔽的错误**是：训练时用 `n_class`，采样时用 `n_class - 1`（把最后一个真实类当成了空类）。这时程序不报错，**但 CFG 的“无条件”分支其实是在生成最后一类**，表现为“引导越强，越像那个类”。第 7 章末尾的坑列表里它排第一。

7.3.1 另外两种注入方式

相加是最省事的一种，但不是唯一的一种。**知道另外两种，才能看懂论文里的模型结构。**

① AdaGN / FiLM：用条件去缩放和平移每一层

不让条件只出现在输入层，而是**调制网络中间每一层的归一化**：

$$h \leftarrow \gamma(c) \odot \frac{h - \mu}{\sigma} + \beta(c) \tag{7.2}$$

其中 γ, β 都是由 c 算出的小网络。

代价：多两个投影头；**收益**：条件信号更强，不容易在深层被稀释。Stable Diffusion 的 UNet 里，时间步和类别就是用这种方式注入每个 ResNet 块的。

② **交叉注意力：把条件变成一串 token**

条件不再是一个向量，而是一个序列（比如文本的 77 个 token）。每一层用 Q 来自图像特征、K, V 来自文本，做一次注意力：

$$\mathrm{Attn}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V \tag{7.3}$$

这就是第 1 章的多头注意力，换了一组 K, V。代价最大，表达力也最强。第 8 章整章讲它。

③ **三种方式的取舍**

方式	参数量	表达力	典型用法
相加/拼接	最小	弱	类别、标量条件
AdaGN / FiLM	中	中	类别 + 时间步（SD 的 UNet）
交叉注意力	最大	强	文本、图像参考

本章用第一种，是为了隔离变量。如果条件注入本身很复杂，“引导强度”的效果就会和“注入方式”混在一起，说不清是谁的贡献。

> **i** 一个容易忽略的细节
>
> **三种方式都要求“能把条件换成空”。**
>
> 相加方式里，空类是一个可学习的嵌入；FiLM 方式里，空类是 γ, β 网络的一个输入；交叉注意力里，空条件是**空字符串编码出来的那串 token**—— 不是全零向量。
>
> **第 8 章会看到**：空字符串经过文本编码器之后并不是零，它有自己的一组激活。**这件事会直接影响 CFG 的效果**，因为“无条件”分支的定义变了。

7.3.2 三种注入方式的参数量账

“表达力更强”是有价格的。用 Stable Diffusion 的 $d_{\text{model}} = 320$ 算一笔账。

AdaGN：每个 ResNet 块需要两组调制参数（缩放和平移），

$$2 \times d_{\text{model}} = 640 \text{ 个参数/块} \tag{7.4}$$

交叉注意力：Q, K, V 和输出投影各一个 $d \times d$ 矩阵，

$$4 \times d_{\text{model}}^2 = 409,600 \text{ 个参数/层} \tag{7.5}$$

相差 640 倍。

方式	每层参数	SD UNet 里的用途
相加	d	本章的类别条件
AdaGN	$2d$	时间步 + 类别
交叉注意力	$4d^2$	文本

这解释了 SD 的设计选择：时间步和类别用 AdaGN（便宜），文本用交叉注意力（贵，但文本需要 token 级的对齐能力）。

> **i** 为什么文本不能用 AdaGN
>
> 文本不是“一个向量”，是一**串向量**。用 AdaGN 意味着先把 77 个 token 池化成一个向量——**“一只猫坐在一只狗旁边”里的“猫”和“狗”会被平均掉**。
> 交叉注意力保留 token 级的信息：**图像的每个位置可以去“查”它需要的那个词**。这就是第 1 章那个注意力矩阵在做的事。
> **代价是 640 倍的参数**。这个交换在文本上是值得的，在类别上不值得。

AdaGN 的完整形式（时间步和类别相加后，投影成两组调制参数）：

$$h \leftarrow (1+\gamma(e)) \odot \mathrm{GroupNorm}(h) + \beta(e), \qquad e = \mathrm{MLP}(t_{\mathrm{emb}} + c_{\mathrm{emb}}) \tag{7.6}$$

注意那个 $1+$：它让 $\gamma = 0$ 时退化成“不调制”，**初始化时整层接近恒等**。没有它，训练初期的随机 γ 会把特征缩放得很厉害。
这是“调制类”结构的一个通用技巧，和 ResNet 的残差、LayerScale 的 γ 是同一个思路：**让新加的部件初始时等于没有**。

7.4 CFG 的推导：两个 score 的差

这一节把 CFG 的公式推出来。**只用贝叶斯公式和 score 的定义，不需要新工具**。

7.4.1 从贝叶斯到 score

条件分布和无条件分布的关系是

$$p(x \mid c) = \frac{p(x)\, p(c \mid x)}{p(c)} \tag{7.7}$$

两边取对数再对 x 求梯度。$p(c)$ 与 x 无关，求导消失：

$$\nabla_x \log p(x \mid c) = \nabla_x \log p(x) + \nabla_x \log p(c \mid x) \tag{7.8}$$

这就是条件生成的核心恒等式。
它说：**“条件分布的 score” = “无条件 score” + “分类器对 x 的梯度”**。
三项都认识：

- $\nabla \log p(x)$ 就是第 6 章的 score，模型已经在学它（$\varepsilon = -\sqrt{1-\bar\alpha}\nabla \log p$）
- $\nabla \log p(c \mid x)$ 是一个分类器的梯度
- 左边是“我们想要的那个东西”

所以条件采样 = 无条件采样 + 一个分类器的推力。这就是分类器引导。

7.4.2 分类器引导的公式

把 $\varepsilon = -\sqrt{1-\bar\alpha_t}\nabla \log p_t$ 代进去：

$$\hat\varepsilon = \varepsilon_\theta(x_t, t) - s\sqrt{1-\bar\alpha_t}\nabla_{x_t} \log p_\phi(c \mid x_t) \tag{7.9}$$

其中 s 是引导强度。**符号是负的**：score 加一项，等价于 ε 减一项。
这个式子的工程代价很直接：你需要一个能在**每个噪声水平**上工作的分类器 $p_\phi(c \mid x_t, t)$。在干净图片上训的分类器，到了高噪声处梯度没有意义。

7.4.3 CFG：把分类器换成一个差

CFG 的观察是：那个分类器梯度，可以用模型自己近似。
由贝叶斯恒等式，

$$\nabla \log p(c \mid x) = \nabla \log p(x \mid c) - \nabla \log p(x) \tag{7.10}$$

右边两项都是 score，而 score 和 ε 只差一个常数。
代回分类器引导的式子，整理后得到

$$\hat{\varepsilon} = \varepsilon_\theta(x_t, t, \varnothing) + w(\varepsilon_\theta(x_t, t, c) - \varepsilon_\theta(x_t, t, \varnothing)) \tag{7.12}$$

其中 $\varnothing$ 是空类，w 是引导强度。
这就是 CFG。 它一次都不需要分类器，只需要模型能在“有标签”和“空标签”两种输入下各算一次。

7.4.4 为什么是“缩放差”，不是“加强条件”

一个自然的想法是：既然要“更听话”，为什么不直接把 ε_c 乘大一点？

$$\hat{\varepsilon} = w\,\varepsilon_c \quad \text{(错误做法)} \tag{7.13}$$

因为乘大的是整个向量，不只是“条件”那部分。
把 ε_c 拆成“基准 + 条件方向”：

$$\varepsilon_c = \varepsilon_u + \underbrace{(\varepsilon_c - \varepsilon_u)}_{\text{条件方向}} \tag{7.14}$$

乘 w 的应该是后一项，不是整个 ε_c。

- **乘整个 ε_c**：基准也被放大，画面整体变亮/变大—— 这是**尺度**变化，不是**内容**变化
- **乘差**：只有条件方向被放大，基准不变——这才是“更听话”

CFG 的公式正是后者。 而且它在 $w = 1$ 时精确退化成 ε_c—— **这是“乘整个向量”做不到的**（除非额外归一化）。

> **i** 一个几何图像
>
> 把 ε_u 看作原点，ε_c 是原点外的一个点。
>
> - **“加强条件”**：把 ε_c 从原点往外拉 → 方向不变，长度变
> - **CFG**：沿着 $\varepsilon_c - \varepsilon_u$ 的方向走更远 → **等价于把 ε_c 从 ε_u 出发拉长**
>
> **两者在“方向”上是一样的，但原点不同。** CFG 的原点是 ε_u，所以它拉长的是“条件带来的那一段”；“加强条件”的原点是零，拉长的是“整个预测”。
> **当 ε_u 很小（比如接近零）时，两者几乎一样**—— 这就是为什么有些实现看起来也能工作。**但一般情况下它们不同。**

7.4.5 三个端点

$$w = 0: \quad \hat{\varepsilon} = \varepsilon_\theta(x, \varnothing) \quad \text{（无条件）} \tag{7.15}$$

$$w = 1: \quad \hat{\varepsilon} = \varepsilon_\theta(x, c) \quad \text{（普通条件采样）} \tag{7.16}$$

$$w > 1: \quad \text{把"条件 - 无条件"这个方向放大} \tag{7.17}$$

注意 $w = 1$ 那一行。 它说明 CFG **不是“额外的技巧”**—— 它是普通条件采样的一个推广，$w = 1$ 就是没有引导。

!这个公式在说什么（人话版）

“外月”和“所有月亮”之间的差别，就是“外月”这个条件带来的信息。
CFG 做的事是：**先算出这个差别，再把它乘以 w 加回去。**

- $w = 1$：原样加回去 → 就是普通条件采样
- $w = 2$：加两倍 → 样本被推得更“像外月”
- $w = 0$：一点都不加 → 退回无条件

所以 w 不是一个“随机性旋钮”，是“你有多想要这个条件”的旋钮。 代价在下一节：你把条件推得越狠，样本越不像真实数据。

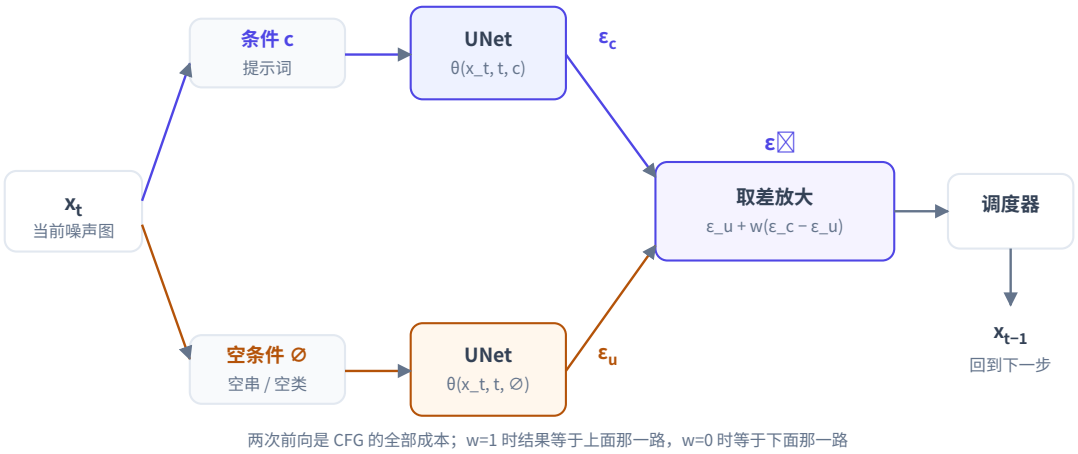


图 7.1　CFG 的数据流。同一个 x_t 走两次 UNet：一次带条件、一次带空条件，两个输出的差乘以 w 再加回去，最后交给调度器。**两次前向是 CFG 的全部成本。**

7.4.6 先算一遍：一维两点数据

在跑实验之前，用一个能用手算的例子把 CFG 走一遍。
设置：数据只有两个点 -1 和 $+1$，各占一半。条件 $c = 0$ 表示“来自 -1”，$c = 1$ 表示“来自 $+1$”。在某个噪声水平上，条件分布是一维高斯：

$$p(x \mid c = 0) = \mathcal{N}(-1, 1), \qquad p(x \mid c = 1) = \mathcal{N}(+1, 1) \tag{7.18}$$

无条件分布是两者的等权混合。
取 $x = 0.2$。 先算混合权重：

$$a = e^{-(1.2)^2/2} = 0.4868, \qquad b = e^{-(-0.8)^2/2} = 0.7261 \tag{7.19}$$

$$w_0 = \frac{a}{a+b} = 0.4013, \qquad w_1 = 0.5987 \tag{7.20}$$

两个 score：

$$\nabla \log p(x \mid c = 0) = -(x+1) = -1.2000 \tag{7.21}$$

$$\nabla \log p(x) = w_0 \cdot (-1.2) + w_1 \cdot (0.8) = -0.0026 \tag{7.22}$$

换成 $\varepsilon = -\sigma \nabla \log p$（取 $\sigma = 1$）：

$$\varepsilon_c = 1.2000, \qquad \varepsilon_u = 0.0026 \tag{7.23}$$

现在套 CFG：

w	$\hat{\varepsilon}$	等效后验均值 $x+\sigma^2\hat{s}$
0	0.0026	0.2000
1	1.2000	**−1.0000**
2	2.3974	**−2.1974**
5	5.9895	**−5.7895**
10	11.9764	**−11.9764**

看第三列。 $w=1$ 时等效均值是 -1.0000——**正好是数据的真实位置。** $w=2$ 时变成 -2.1974，**已经越过数据点，跑到它外面去了。** $w=10$ 时是 -11.98。

> ! 这张表比任何实验都直白
>
> **CFG 不是“让样本更靠近条件”，是“让样本比条件更远”。**
> $w=2$ 时，模型在这个点上的行为等价于“数据在 -2.2 附近”—— **而真实数据在 -1。**
> **在二维实验里，这就是 w 大时样本向外漂移的原因。**准确率仍然是 1.000，因为 -2.2 和 -12 都还在“外月那一侧”。
> **所以“准确率”这个指标，从 $w=1$ 开始就不再衡量“样本是不是合理”了。**它只衡量“样本落在哪个类的半空间”。

这个例子还说明了另一件事：为什么 w 的效果依赖于 σ。
注意等效均值的公式 $x+\sigma^2\hat{s}$。在 t 大的时候 σ 很大，同样的 $\hat{s}$ 会把均值推得更远。**但 $\hat{s}$ 本身也被 σ 缩放了**（$\varepsilon=-\sigma\nabla\log p$），两者部分抵消。
净效果是：在低噪声处（σ 小），引导主要改变分布的**宽度**；在高噪声处，它更容易改变**位置**。这解释了为什么过引导的样本往往“结构还在、位置不对”。

7.5 实测：w 的三段行为

用 50 步 DDPM、每类 1000 个样本，扫 w。裁判准确率、样本标准差、越界率（$|x|>1.5$ 的比例）三列：

w	准确率（外/内）	标准差（外/内）	越界率（外/内）
0	0.525 / 0.475	0.437 / 0.437	0 / 0
0.5	0.846 / 0.818	0.354 / 0.368	0 / 0
1	0.965 / 0.962	0.310 / 0.335	0 / 0
1.5	0.991 / 0.992	0.291 / 0.315	0 / 0
2	0.998 / 0.998	0.281 / 0.297	0 / 0
3	1.000 / 1.000	0.271 / 0.255	0 / 0
5	1.000 / 1.000	0.261 / 0.182	0 / 0
10	1.000 / 1.000	0.257 / 0.110	0.002 / 0
20	1.000 / 1.000	0.897 / 0.285	**0.397 / 0.093**
40	1.000 / 1.000	1.703 / 0.293	**0.697 / 0.895**

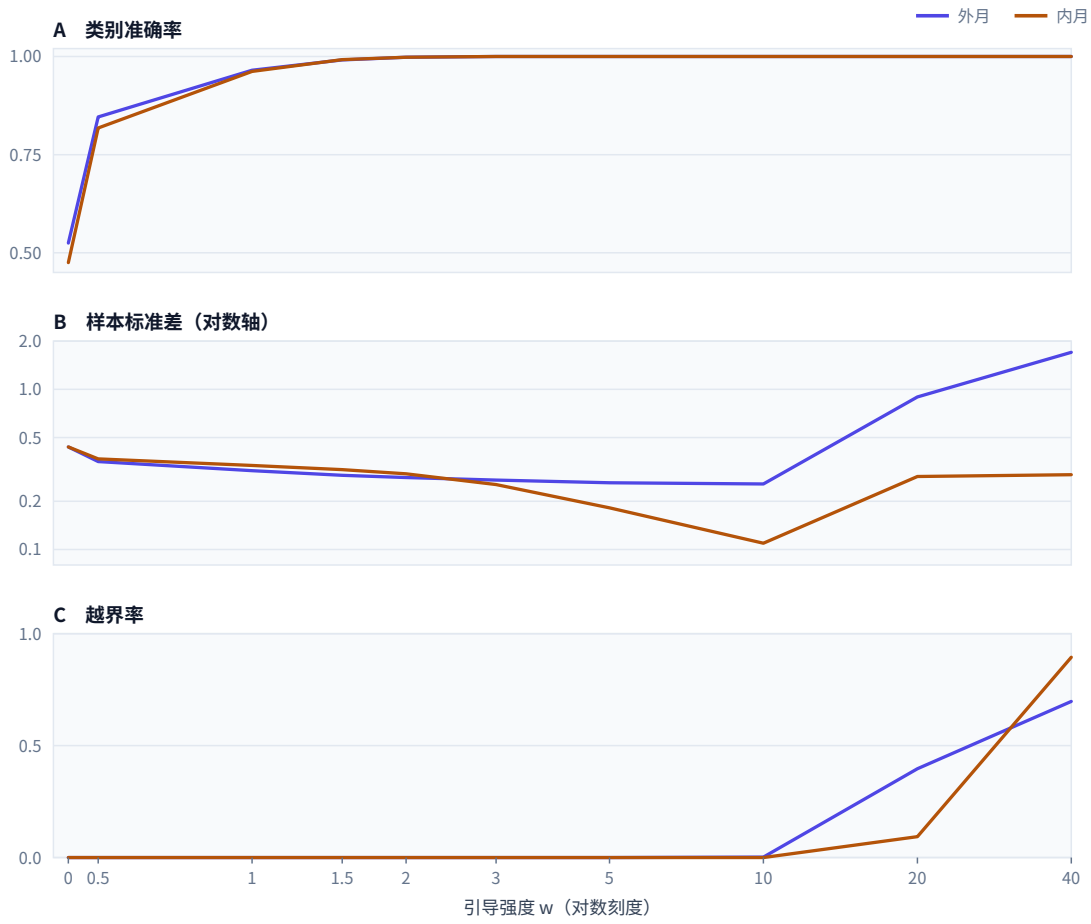


图 7.2　引导强度扫描。A：两个类的准确率都在 $w = 3$ 之后饱和；B：标准差（对数轴）先缓慢下降，$w \geq 20$ 后外月急剧上升；C：越界率在 $w = 10$ 之后抬头。数据来自 `data/ch7_case.json`。

把它画成曲线，会看到清楚的三段。

7.5.1 第一段：$0 \leq w < 1$，欠引导

$w = 0$ 时准确率 0.525——**模型完全没在听条件**。$w = 0.5$ 升到 0.846，$w = 1$ 到 0.965。

这一段是“条件开始起作用”的过程。如果你直接用 $w = 0.5$，会得到“有点像外月、又有点像内月”的样本——**这不是模型不好，是你只给了一半的力**。

7.5.2 第二段：$1 \leq w \leq 10$，正常引导

准确率在 $w = 2$ 就到 0.998，$w = 3$ 之后是 1.000。

但标准差在单调下降：外月 0.310 → 0.257，内月 0.335 → **0.110**。

内月掉得比外月快得多。为什么？因为内月本身更“窄”（数据标准差 0.339 对 0.573），**同样的推力下，窄的分布先被压扁**。

这就是引导的代价：它用多样性换准确率。

怎么读“标准差下降”

不要把“标准差变小”直接读成“质量变差”。

- **在 $w \le 2$ 时**，标准差从 0.310 降到 0.281，准确率从 0.965 升到 0.998—— 这是**去掉错误样本**带来的自然收缩，是收益。
- **在 $w \ge 5$ 时**，内月从 0.182 掉到 0.110，而准确率早就是 1.000—— **再压下去没有收益，只有损失**。

判据不是“标准差有没有降”，是“降的时候准确率还在不在涨”。

7.5.3 第三段：$w \ge 20$，过引导

准确率还是 1.000（裁判永远认对），**但样本已经不在数据分布上了**：

- $w = 20$：外月 39.7%、内月 9.3% 的样本跑出 $|x| > 1.5$
- $w = 40$：69.7% / 89.5%

准确率这个指标在这里完全失效——裁判只回答“它在哪一类”，不回答“它是不是一个合理的点”。**一个飞到天边去的点，仍然可以被认成外月。**

判断过引导必须看第二个量：越界率，或者样本到真实数据的距离。

准确率满分的时候，往往是最危险的时候

$w = 20$ 的准确率是 1.000，比 $w = 2$ 还高。

如果你只用分类准确率评估引导，会得出“w 越大越好”的结论，然后把 w 调到 40，得到一堆飞到 $|x| \approx 2$ 的废点。

这是本章最容易犯的评估错误，也是“只看一个指标”的通病。第 6 章在采样器上犯过同一次错（只看步数不看 NFE），这里只是换了个场景。

7.5.4 安全区间会随步数收窄

上面那张表是 50 步 DDPM 的结果。**换步数，三段的分界会移动。**

用可跳步的精确 DDPM 后验（步长变大，但每一步仍是高斯后验），扫 10 / 25 / 50 / 100 步：

步数	$w = 1$	$w = 2$	$w = 5$	$w = 20$
10	0.938	1.000	**0.612**	**0.593**（越界 53.8%）
25	0.958	1.000	0.885	0.937（越界 58.7%）
50	0.955	1.000	**1.000**	0.887（越界 53.8%）
100	0.973	1.000	**1.000**	0.957（越界 41.2%）

看 $w = 5$ 那一列：10 步时准确率只有 0.612，50 步时是 1.000。

这是本章最实用的一条结论：步数越少，能用的 w 越窄。

为什么？引导放大的那个差 $(\varepsilon_c - \varepsilon_u)$ **是在一步之内**加进去的。步长越大，这个放大后的方向被推得越远，**样本越容易一步跨出模型“认识”的区域**——和第 6 章的“步长 vs 曲率”是同一个问题。

这条结论的实践含义

少步蒸馏模型的引导强度不能照搬多步模型的。

如果你把一个 50 步、$w = 5$ 的配置直接搬到 10 步，得到的不是“稍微差一点”，是**准确率掉到 0.6、一半样本越界。**

正确做法：步数减半时，先把 w 降一档试。本章的数据给出一个粗糙的起点：10 步时 w 不要超过 2。

7.5.5 CFG 采样的是哪个分布

三段行为可以从一个式子直接看出来。

把 $\varepsilon = -\sqrt{1-\bar{\alpha}_t}\,\nabla \log p_t$ 代进 CFG 的公式：

$$\hat{\varepsilon} = w\,\varepsilon_c + (1-w)\,\varepsilon_u \tag{7.24}$$

两边同乘 $-1/\sqrt{1-\bar{\alpha}_t}$，换成 score：

$$\nabla \log p_w(x \mid c) = w\,\nabla \log p(x \mid c) + (1-w)\,\nabla \log p(x) \tag{7.25}$$

一个函数的梯度等于若干梯度的线性组合，说明这个函数本身是那些函数的乘积（取对数后相加）。

去掉与 x 无关的归一化常数，得到

$$p_w(x \mid c) \propto p(x \mid c)^{w\,(p(x))^{1-w}} \tag{7.26}$$

这就是 CFG 实际采样的分布。

> ! 这个式子解释了本章的所有现象
>
> $\boldsymbol{w=0}$：$p_0 = p(x)$ —— 无条件。 $\boldsymbol{w=1}$：$p_1 = p(x \mid c)$ —— 普通条件采样。 $\boldsymbol{w>1}$：$p(x \mid c)$ 被乘了大于 1 的幂，$p(x)$ 被乘了负幂。**这是在“锐化”后验**：高密度处更突出，低密度处被压得更低。 $\boldsymbol{w<0}$：反过来，把后验“抹平”，质量被推向低密度区。
>
> **所以 w 不是一个“采样技巧”的参数，它改变了你要采的那个分布本身。**
>
> **这也解释了为什么 w 不能无限大**：$p(x \mid c)^w$ 在 $w \to \infty$ 时收敛到后验的众数——**一个点**。而模型的众数估计有误差，所以实际得到的是“模型认为的众数”，不是真实数据的众数。$w = 40$ 时有效维度 1.009，就是“正在收敛到一个点”的过程。

这个视角还解释了两件前面看到的事。

① **为什么准确率会饱和**。锐化只改变“质量在类内的分布”，不改变“质量属于哪个类”。一旦所有样本都落在正确的类区域里，再锐化也不会提高准确率—— **它只会让样本往类内的高密度处挤**。

② **为什么两类收缩得不一样快**。内月本身的方差更小（0.339 对 0.573），**同样的幂次作用在一个更窄的分布上，相对收缩更大**。这不是“模型偏心”，是 p^w 的几何性质。

> ⚠ 一个容易误用的推论
>
> 既然 p_w 是一个合法的（未归一化的）分布，**能不能说“$w>1$ 的样本来自一个更好的分布”？**
>
> **不能**。p_w 是“被重新加权过的后验”，不是“更接近真实标签的后验”。它更锐，但**更锐不等于更对**——它把模型的偏差也一起放大了。
>
> **本章的准确率之所以随 w 上升，是因为这个二维问题的类区域很宽、模型偏差很小**。在真实图像上，锐化会把模型的系统性偏差（比如“所有猫都长一个样”）一起放大，表现为多样性塌缩和风格同质化。

7.5.6 为什么是乘积，不是加权和

第一次看到 $p_w \propto p(x \mid c)^{w(p(x))^{1-w}}$ 时，一个自然的疑问是： **为什么不写成 $wp(x \mid c) + (1-w)p(x)$？**

因为 score 是相加的，而 score 是 $\log p$ 的梯度。

$$\nabla \log p_w = w\nabla \log p(x \mid c) + (1-w)\nabla \log p(x) \tag{7.28}$$

梯度相加，等价于密度相乘。如果写成密度的加权和，它的梯度不会是那个加权和——**而是另一回事**。

> **i** 一个类比
>
> **几何平均 vs 算术平均。**
> 两个数 1 和 100：算术平均 50.5，几何平均 10。**几何平均被小的那个主导**——这正是 CFG 想要的：$w > 1$ 时，$p(x)$ 那一项（可能很小）被 $1 - w$ 的负幂放大，于是“边缘分布小的地方”被压得更狠。
> **算术加权和做不到这一点**——它只会把两个密度插值，不会“惩罚”低密度区。

这也解释了 $w < 0$ 为什么危险。
$w < 0$ 时 $1 - w > 1$，$p(x)$ 被**正幂**放大；同时 $p(x \mid c)$ 被**负幂**放大。**两个方向都在放大低密度区**——分布被推向数据稀疏的地方。

7.5.7 数值演示：p_w 被锐化了多少

接着上面的一维例子，把 p_w 算出来看看。
取两个位置：$x = 0.2$（远离类中心）和 $x = -1$（类中心本身）。算它们的**相对密度** $p_w(0.2)/p_w(-1)$——这个比值衡量“离中心多远处还有多少质量”：

w	$p_w(0.2)$	$p_w(-1)$	比值
0	0.2419	0.2265	1.068
1	0.1942	0.3989	0.487
2	0.1559	0.7028	0.222
3	0.1251	1.2380	0.101
5	0.0806	3.8418	0.021
10	0.0268	65.172	**0.00041**

注意比值那一列。

- $w = 0$：远离中心的点和中心的点一**样重**（1.068）——无条件分布里两者本来就差不多
- $w = 1$：中心是远点的 2 倍（0.487 的倒数）
- $w = 10$：**中心是远点的 2400 倍**

从 $w = 1$ 到 $w = 10$，相对密度掉了 1000 倍。

> **!** 这就是“多样性塌缩”的精确含义
>
> **不是样本变少了，是“远离中心的区域几乎不再有质量”。**
> $w = 10$ 时，$x = 0.2$ 处的密度只有 $x = -1$ 处的 1/2400。**采样几乎必然落在中心附近**——看起来就是“每张图都差不多”。
> **而准确率完全不受影响**：$x = 0.2$ 和 $x = -1$ 都属于同一个类。
> **这解释了本章那个反复出现的现象**：准确率在 $w = 3$ 就饱和，而多样性一直掉到 $w = 10$。**前者只看“属于哪一类”，后者才看“类内怎么分布”。**

顺带解释一个细节：为什么 $p_w(0.2)$ 随 w 下降、而 $p_w(-1)$ 上升？因为 p_w 没归一化。**归一化之后，中心处的概率密度会更高**，两处的绝对数值都会变。**只有比值是有意义的**——因为它与归一化常数无关。

7.5.8 隐式分类器：CFG 用的到底是什么分类器

分类器引导的公式里，有一项 $\nabla \log p(c \mid x)$。CFG 把它换成了 $\nabla \log p(x \mid c) - \nabla \log p(x)$。
这两者真的相等吗？由贝叶斯公式，是的：

$$\nabla \log p(c \mid x) = \nabla \log p(x \mid c) - \nabla \log p(x) \tag{7.29}$$

但注意右边没有 w。CFG 的公式里多了一个 w：

$$\hat{s} = w\nabla \log p(x \mid c) + (1-w)\nabla \log p(x) \tag{7.30}$$

它可以改写成

$$\hat{s} = \nabla \log p(x) + w\underbrace{[\nabla \log p(x \mid c) - \nabla \log p(x)]}_{\nabla \log p(c|x)} \tag{7.31}$$

所以 CFG 做的事是：把“隐式分类器的梯度”乘上 w。

这就引出一个有用的说法：CFG 等价于用一个“被锐化了 w 倍的分类器”做引导。这个分类器没有独立的网络，**它由生成模型自己定义**，所以叫**隐式分类器**。

> ℹ 隐式分类器的三个推论
>
> ① **它不可能比生成模型更懂类别**。分类器引导可以请一个在真实数据上训练过的分类器（哪怕它在高噪声处会失效）；CFG 的“分类器”完全来自模型对 $p(x \mid c)$ 和 $p(x)$ 的估计。**模型没见过的东西，隐式分类器也不知道**。
>
> ② **它的“准确率”无法单独测量**。你不能把隐式分类器拿出来跑测试集——它只在给定 x_t 时定义一个梯度方向。**这就是为什么本章要用“裁判分类器”从外部给样本打分**，而不是直接评估隐式分类器。
>
> ③ **w 的作用等价于“温度”**。$w > 1$ 让隐式分类器更自信，$w < 1$ 让它更犹豫。**而一个过度自信的分类器，会把样本推到它自己认为的极值点**—— 这正是 $w = 20$ 时样本塌成一条线的原因。

这个视角还解释了一个实践中的困惑：为什么 CFG 对“模型没学好的类”特别糟。

假设某个类在训练集里只有 10 个样本。模型对 $p(x \mid c)$ 的估计很粗糙，**隐式分类器对这个类的判决边界也会很粗糙**。$w = 5$ 一放大，样本就被推向一个“模型以为的该类区域”—— **而那个区域可能根本不存在于真实数据里**。

分类器引导没有这个问题吗？ 有，但它的分类器是独立训练的，**至少可以检查它在验证集上的表现**。隐式分类器连这个检查都做不了。

这是 CFG 的隐性代价：它把“分类器好不好”这个问题，藏进了“生成模型好不好”里面。

7.5.9 停下来想一想 ①

假设你有两个模型：

- **模型 A**：在每个类上都很准，但对“类之间的边界”很模糊
- **模型 B**：边界很清楚，但每个类内部生成的样本很单一

用同样的 $w = 3$ 做 CFG，哪个模型的准确率会更高？哪个的多样性会更差？

提示：隐式分类器 = 条件分布除以边缘分布。**边界清不清楚，取决于这个比值在类之间变化得有多快**。模型 B 的这个比值更陡，所以**同样的 w 在 B 上等效于更强的引导**。

这道题的结论是：w 的最优值依赖于模型，不是数据集。**换模型必须重新调 w**，不能照搬。

7.5.10 种子方差：主表有多稳

上面所有数字都是单种子。**换 5 个种子重跑 w 的主表：**

w	准确率	多样性	越界率
1	0.961 ± 0.006	0.311 ± 0.003	0
2	0.999 ± 0.001	0.281 ± 0.003	0
5	1.000 ± 0.000	0.258 ± 0.003	0
20	1.000 ± 0.000	0.859 ± 0.045	0.391

标准差非常小（准确率 ⩽0.006，多样性 ⩽0.003）。**这意味着正文里大于 0.01 的差异都是有意义的**，不是种子噪声。

只有 $w = 20$ 的多样性方差大（±0.045）——过引导区的样本本身就不稳定，多个种子给出的“崩溃程度”不一样。**这本身就是一个信号**：进入过引导区之后，结果的可复现性会下降。

7.5.11 四类条件：更可分的数据，反而更早崩

换成四团数据（四个象限各一团，噪声 0.12），重复同样的扫描：

w	准确率	多样性	越界率
1	1.000	0.125	0
2	1.000	0.111	0
3	1.000	0.104	0
5	1.000	0.095	0.003
10	1.000	0.085	**0.765**

两件事和两类的情形不一样。
① **$w = 1$ 的准确率就是 1.000。** 四团数据彼此分得很开，**模型不需要引导就已经完全分得清。**
② **崩溃点提前到了 $w = 10$**（越界 76.5%），而两类时是 $w = 20$。

> ! 为什么越可分反而越早崩
>
> **因为“可分的程度”和“分布的宽度”是两件事，而崩溃取决于后者。**
> 四团数据的类内标准差只有 0.125，两类双月是 0.31。**同样的引导强度，作用在一个更窄的分布上，绝对位移更小，但相对位移更大**——样本更容易被推离数据范围。
> **所以“w 该取多少”不只取决于类数，更取决于类内有多宽。** 一个类别紧凑、边界清晰的数据集，**$w = 1$ 就够了，加引导只有坏处。**

7.6 负引导：$w < 0$ 时会发生什么

公式里 w 没有限制为正。取负值，$(1 - w)$ 那一项就变成“减去条件方向”：

w	准确率（外/内）	越界率（外/内）
-0.5	0.248 / 0.167	0.003 / 0.018
-1	0.133 / 0.049	0.079 / 0.281
-2	0.064 / 0.005	0.312 / 0.647

$w = -1$ 时，要求“外月”，只有 13.3% 的样本还在外月区域。
这就是 negative prompt 的数学原型：把 w 取负，等于说“不要这个类”。在图像模型上，用户输入的 negative prompt 就是那个 $\varnothing$ 位置被替换成“你不想要的东西”，再用负权重推离它。
但注意越界率：$w = -2$ 时外月有 31.2%、内月有 64.7% 的样本跑出范围。**负引导比正引导更容易把样本推出流形**，因为你是沿着“数据变稀”的方向走。
一句话记住正负引导的区别

- **正引导**：沿着“更像这个类”的方向走。**方向上有数据，所以走得远也还算合理。**
- **负引导**：沿着“更不像这个类”的方向走。**这个方向很快就没有数据了。**

所以 negative prompt 的强度通常要比 positive 小得多。

7.6.1 负引导为什么更危险：从 p_w 看

用上一节的式子，$w = -1$ 时：

$$p_{-1}(x \mid c) \propto p(x \mid c)^{-1} (p(x))^{2} \tag{7.32}$$

负幂。 这意味着**后验密度越小的地方，权重越大**—— 分布被“翻转”了。
翻转的后果是：$p(x \mid c)$ 接近 0 的区域（也就是“这个类绝不会出现的地方”）**权重趋于无穷。**而在有限样本下，那些区域只是“没采到”，不是“真的为零”。

! 这是负引导脆弱的根本原因

正引导在放大“数据多的地方”；负引导在放大“数据少的地方”。

- 数据多的地方，模型见过，估计可靠
- 数据少的地方，模型没见过，估计是外推

所以负引导本质上是在让模型的外推决定结果。
这也解释了为什么它的强度要小得多：$w = -2$ 时越界率 31%/65%，而 $w = +2$ 是 0—— **同样的幅度，负方向走的是模型最不可靠的区域。**

实践上的三条规则：

1. **强度取正的 1/2 到 1/4**
2. **只放真正要排除的概念**，不要堆一堆同义词
3. **监控越界率**（图像上是“结构崩坏、色块乱飞”）

7.6.2 停下来想一想 ②

negative prompt 里写“不要模糊”，和把“清晰”写进正向 prompt，有什么区别?
提示：正向 prompt 改变的是 ε_c；negative prompt 改变的是 $\varepsilon_\varnothing$。
答案的方向：正向是“把样本拉向清晰”；负向是“把样本推离模糊”。**在数据分布上，这两个方向不一定是相反的**——“不模糊”不等于“清晰”，中间还有一大片区域（比如“有结构但低分辨率”）。
这就是为什么 negative prompt 经常“看起来没用”： 它推的方向和你以为的不是同一个。

7.6.3 停下来想一想 ③

本章的 $w = 2$ 在 50 步下准确率 0.998，在 10 步下是 1.000。 但 10 步的 $w = 5$ 只有 0.612。
如果只能改一个：把步数从 10 提到 50，还是把 w 从 5 降到 2?
分析： - 提到 50 步：成本 $\times 5$，$w = 5$ 恢复可用 - 降到 2：成本不变，但引导变弱（准确率从假设的 1.0 掉到 1.0—— 在 10 步下 $w = 2$ 本来就是 1.000）
所以在这个具体问题上，降 w 是免费的。 但一般来说不是：如果 $w = 2$ 不足以分开两个相近的类， 你就必须加步数。
判据：先降 w 看准确率掉不掉。**不掉，就降；掉了，才加步数。**

7.7 标签丢弃：CFG 的前提，不是正则化

回到那个 `+1` 的空类。
训练时，每个样本以概率 $p_{\text{drop}} = 0.1$ 把标签换成空类：

```
mask = torch.rand(bs) < drop
yb = torch.where(mask, null_class, yb)
```

很多人把它当成 dropout 的一种——“防止过拟合”。它不是。
它的唯一作用是：让模型见过空类这个输入。

7.7.1 对照实验

训练两个模型，唯一的差别是 `drop`：

w	有丢弃（0.1）	无丢弃（0）
1	准确率 0.965 / 0.962, 越界 0 / 0	0.960 / 0.969，越界 0 / 0
2	**0.998 / 0.998，越界 0 / 0**	0.992 / 0.995，越界 **0.060 / 0.218**

w	有丢弃（0.1）	无丢弃（0）
5	**1.000 / 1.000，越界 0 / 0**	0.780 / 0.988，越界 **0.986 / 1.000**

$w = 1$ 时两个模型几乎一样——因为 $w = 1$ 根本不调用空类。
$w = 2$ 开始分岔，$w = 5$ 时无丢弃的模型几乎把所有样本都推出了数据范围。

7.7.2 为什么

没有丢弃时，空类那一行嵌入**从来没被训练过**，它是初始化时的随机值。
$w = 2$ 的公式是 $\varepsilon_\varnothing + 2(\varepsilon_c - \varepsilon_\varnothing)$—— **你要减掉的那个“无条件”，是一个随机向量**。 差值没有意义，放大两倍更没意义。
这就是为什么 CFG 必须和标签丢弃一起出现。 论文里把它写成“训练时随机丢弃条件”，容易被读成一个正则化技巧； **实际上它是 CFG 的定义的一部分**。

7.7.3 丢弃率不是一个可以随便填的数

把丢弃率也扫一遍（每档 600 个样本，w 从 1 到 10）：

drop	$w = 1$	$w = 2$	$w = 5$	$w = 10$
0	0.960	0.995（越界 5.5%）	**0.768**（越界 98.0%）	0.892（越界 100%）
0.05	0.962	0.997	1.000	1.000
0.1	0.958	0.997	**1.000**	**1.000**
0.3	**0.933**	0.985	1.000	1.000（越界 2.7%，多样性 0.431）

三种失败方式，分别对应三个区间：

- **drop=0**：$w > 1$ 就崩。**空类没训过**。
- **drop=0.3**：$w = 1$ 的条件准确率掉到 0.933（比 0.1 低 2.5 个点），而且 $w = 10$ 时多样性反常地涨到 0.431、越界 2.7%。**三分之一的条件信息被用来学“无条件”，有条件的那部分变弱了**。
- **drop=0.05 与 0.1**：两者都好。**0.1 略优**，且是文献惯例。

所以 10% 不是一个魔法数字，是一个宽平台上的默认值。 真正不能碰的是 0 和“太大”。

7.7.4 空条件必须是“学出来的”，不能是零

丢弃率解决了“空类有没有被训练”，但还有一个更基本的问题： **空条件长什么样?**
本章的做法是一个可学习的嵌入（`nn.Embedding` 的最后一行）。另一个看起来更自然的做法是**用零向量**： “没有条件”不就是“什么都没有”吗?
实验：训练一个 token 条件模型（每类用两个 token 表示），然后比较两种“无条件”——**学出来的 NULL token** vs **零向量**：

空条件	$w = 3$	$w = 5$	$w = 10$
NULL token（学出来的）	越界 0，多样性 0.099	越界 0，0.088	越界 0，0.072
零向量	**越界 100%**	**越界 100%，多样性 0**	**越界 100%，多样性 0**

零向量的那一行是灾难性的：所有样本都飞出数据范围， $w \geq 5$ 时多样性归零——**塌成一个点**。
而准确率仍然是 1.000。 又一个“准确率满分、样本全废”的例子。

⚠ 这件事对文本模型意味着什么

文本模型的“空条件”不是零向量，是空字符串经过编码器的输出。
那个输出是一个特定的、非零的向量——**而且模型在训练时见过它**（训练时以 10% 概率把文本换成空串）。
所以它不是“没有条件”，是“一个被训练过的、表示’没有条件’的条件”。
如果你自己拼一个零向量送进去，等于在用一个模型没见过的输入，表现就和上表第二行一样：全崩。
推论：换文本编码器、换 tokenizer、甚至换空串的写法，都会改变这个向量——**都必须重新验证 CFG 是否还工作**。

! 丢弃率怎么选

- **太低**（如 1%）：空类嵌入训练不足，w 大时不稳定
- **太高**（如 50%）：一半的容量花在无条件生成上，有条件的那部分变差
- **惯例是 10%**，本章也是 0.1

它不是一个需要精调的超参数，但也不能省。如果你发现 CFG 一开就崩，先检查这一行在不在。

7.8 分类器引导：同一把尺子量一次

CFG 和分类器引导是同一个恒等式的两种用法。**要比较它们，必须用同一个模型、同一个采样器、同一批种子。**
本章的模型是 `drop=0.1` 训练出来的条件模型，采样器统一用 50 步 DDPM（第 6 章已经证明：这个 5 万参数的模型在 DDIM 下会偶发发散，用不稳定的采样器比较两种引导，比的是采样器不是引导）。

s	准确率	标准差	越界率
0	0.965	0.310	0
0.5	0.975	0.295	0
1	0.978	0.284	0
2	0.985	0.268	0
5	0.994	0.240	0
10	0.998	0.223	0

先看 $s = 0$ 那一行：0.965。它和 CFG 的 $w = 1$ 完全一致（0.965 / 0.962），**因为两者都退化成普通条件采样。这是一次必要的仪器检查**——如果这里对不上，说明分类器引导的实现有问题，后面的数都不能信。

7.8.1 两条曲线的形状不一样

把两组数据放在同一个坐标系里比（准确率 vs 标准差）：

目标准确率	CFG 需要的 w	CFG 的标准差	分类器引导需要的 s	其标准差
0.965	1	0.310	0	0.310
0.985	约 1.3	约 0.29	2	0.268
0.994	约 2	约 0.28	5	0.240
0.998	2	0.281	10	0.223

分类器引导在达到同样准确率时，标准差掉得更多。

原因是它推的方向不同。CFG 推的是“条件方向”（两个模型输出的差），分类器引导推的是“分类器认为的类别方向”。**后者更直接，也更粗暴**——它不管数据长什么样，只管往判决边界走。

> **i** 这个比较公平吗
>
> **部分是。**两者用的是同一个模型、同一个采样器、同一批种子，这一点是公平的。
>
> **但这个分类器是专门为这个数据集训练的，**而且只在二维上。在真实图像上，分类器要在大得多的输入空间里工作，**它的梯度会更不可靠**——下一节的表说明了这一点。
>
> **所以这里得到的不是“CFG 更好”的普遍结论，而是“在这个条件下两者的权衡曲线长什么样”。**

7.8.2 完整推导：那个 $\sqrt{1-\bar{\alpha}_t}$ 是从哪来的

分类器引导的公式里有一个容易被抄漏的系数。**四步推完。**

第一步：条件 score 的分解。

$$\nabla \log p(x_t \mid c) = \nabla \log p(x_t) + \nabla \log p(c \mid x_t) \tag{7.33}$$

这一步只用了贝叶斯公式，与扩散无关。**它对任意噪声水平 t 都成立，**只要把 x 换成 x_t。

第二步：把“加一个分类器梯度”写成“改 score”。

我们想要采样的是

$$\tilde{s}(x_t) := \nabla \log p(x_t) + s\,\nabla \log p(c \mid x_t) \tag{7.34}$$

其中 s 是强度。$s = 1$ 时它正好是条件 score；$s > 1$ 是放大。

第三步：score 与 ε 的换算。

第 6 章给过

$$\varepsilon = -\sqrt{1-\bar{\alpha}_t}\,\nabla \log p_t(x_t) \tag{7.35}$$

注意这个系数依赖于 t。在 t 大时它接近 1，在 t 小时它接近 0。

第四步：把第二步的 $\tilde{s}$ 换成 $\tilde{\varepsilon}$。

$$\tilde{\varepsilon} = -\sqrt{1-\bar{\alpha}_t}\,\tilde{s} = \underbrace{-\sqrt{1-\bar{\alpha}_t}\,\nabla \log p_t}_{\varepsilon_\theta} - s\sqrt{1-\bar{\alpha}_t}\,\nabla \log p(c \mid x_t) \tag{7.36}$$

这就是公式里的 $\sqrt{1-\bar{\alpha}_t}$。

> **⚠** 三种常见写法，别抄混
>
写法	来源	注意
$\varepsilon_\theta - s\sqrt{1-\bar{\alpha}}\,\nabla \log p(c \mid x_t)$	本章	作用在 ε 上
$\nabla \log p + s\nabla \log p(c \mid x)$	score 空间	少一个系数，但作用对象不同
$\varepsilon_\theta - s\,\nabla \log p(c \mid x_t)$	**错的**	漏了 $\sqrt{1-\bar{\alpha}}$
>
> **第三种在 t 小时会几乎不起作用**（系数趋近 0），在 t 大时又过强。**症状是“引导只在某些噪声水平有效”**——如果你看到这种“时灵时不灵”，先检查这个系数。

7.8.3 为什么分类器必须在噪声数据上训练

恒等式里出现的是 $p(c \mid x_t)$，**不是 $p(c \mid x_0)$。**

在 $t = 999$ 时，x_t 几乎是纯噪声。**“这张几乎全是噪声的图属于哪一类”这个问题，用干净数据训练的分类器回答不了**——它从没见过这种输入。

所以分类器必须用和去噪器完全相同的前向过程来训练：采样 x_0、采样 t、加噪、预测类别。
它的训练目标和去噪器几乎一样，只是输出从“噪声”换成了“类别 logits”。**这就是为什么它贵**：你实际上训练了第二个扩散模型。

> ℹ 一个反直觉的点
>
> **分类器在 t 大时准确率接近 0.5，但它的梯度并不是零。**
> 一个随机猜测的分类器，仍然会对输入产生一个梯度方向——**只是这个方向没有语义**。而分类器引导会把这个无意义的方向 乘上最大的系数（$\sqrt{1-\bar{\alpha}} \approx 1$）加进去。
> **这正是“最不可靠的梯度被乘上最大的权重”的完整解释。**

7.8.4 先算一遍：分类器引导与 CFG 其实差一个 1

用同一维两点数据（$\sigma = 1$，$x = 0.2$）把分类器引导也算一遍。
一个“完美分类器”的梯度就是两个 score 的差：

$$\nabla \log p(c=0 \mid x) = \nabla \log p(x \mid c=0) - \nabla \log p(x) = (-1.2000) - (-0.0026) = -1.1974 \tag{7.37}$$

代进分类器引导的公式（$\sqrt{1-\bar{\alpha}} = \sigma = 1$）：

$$\hat{\varepsilon} = \varepsilon_c - s \cdot 1 \cdot (-1.1974) = 1.2000 + 1.1974\, s \tag{7.38}$$

s	$\hat{\varepsilon}$
0	1.2000
1	2.3974
2	3.5948

现在回头看 CFG 那张表：$w = 2$ 时 $\hat{\varepsilon} = 2.3974$。
一模一样。

> ! 一个恒等式：$w = 1 + s$
>
> 当分类器**完美**时，
>
> $$\underbrace{\varepsilon_c + s(\varepsilon_c - \varepsilon_u)}_{\text{分类器引导}} = \underbrace{\varepsilon_u + (1+s)(\varepsilon_c - \varepsilon_u)}_{\text{CFG, } w=1+s} \tag{7.39}$$
>
> **推导只用一行**：$\nabla \log p(c \mid x) = \nabla \log p_c - \nabla \log p_u$，两边乘 $-\sigma$ 就是 $\varepsilon_c - \varepsilon_u$。
> **所以两种方法在数学上是同一个东西**，差别只有两处：
>
> 1. **用什么估计“分类器梯度”**：真分类器 vs 两个模型输出的差
> 2. **参数相差 1**：CG 的 $s = 0$ 对应 CFG 的 $w = 1$

这解释了一个之前的观察。 本章的两张扫描表里，CG 的 $s = 0$ 准确率 0.965，恰好等于 CFG 的 $w = 1$；CG 的 $s = 10$ 是 0.998，和 CFG 的 $w = 10$ 附近相当。
它们本来就应该接近——只是本章的分类器不完美，所以曲线没有完全重合。

> ⚠ 这个恒等式对实践的两个含义
>
> ① **论文里比较两者时，参数必须先对齐。**一个说“$s = 7$”、一个说“$w = 7$”，实际强度差一档。**看论文时先找它的定义式，不要只看数字。**
> ② **“CFG 不需要分类器”的真正含义** 不是“CFG 用了另一种信息”，是**“它用生成模型自己当分类器”**。所以 CFG 的上限完全由生成模型决定——这正是前面“隐式分类器”那一节的内容。

7.8.5 分类器引导要付的三笔账

第一笔：多一个模型。

分类器要在**每个噪声水平**上工作，所以它必须拿加噪后的 x_t 训练：

```
t = torch.randint(0, T, (bs,))
xt, _ = forward_sample(x0, t, sch)
loss = F.cross_entropy(clf(xt, t), y)
```

注意它和去噪器的训练几乎一样——同样的前向过程、同样的时间步采样。**换句话说，你训练了第二个扩散模型，只是它的输出是类别而不是噪声。**

第二笔：每一步多一次反向传播。

分类器引导需要 $\nabla_x \log p_\phi(c \mid x_t)$，这是一个**对输入的梯度**，所以每步都要做一次反向传播。

实测（1000 个样本，单步）：

方法	单步耗时
CFG（两路合并）	0.64 ms
分类器引导	**1.38 ms**

2.14 倍。注意这不是“多一个分类器前向”那么简单—— 反向传播本身的成本大约是前向的两倍。

第三笔：一个会失效的部件。

这就是下一节的内容。

> ℹ 三笔账合起来是什么
>
> | | CFG | 分类器引导 |
> |---|---|---|
> | 额外模型 | 无（同一个模型） | **要一个噪声分类器** |
> | 每步成本 | 1 次前向（batch ×2） | 1 次前向 + 1 次反向 |
> | 额外训练 | 标签丢弃（几乎免费） | **一个完整分类器** |
> | 高噪声可靠性 | 与模型同源 | **可能完全失效** |
>
> **CFG 赢的不是效果，是“少了一个会坏的部件”。**
>
> 这句话在第 6 章也出现过一次——那一章说的是“更好的采样器收益会饱和，因为瓶颈在模型”。**同一个道理：系统里每多一个独立训练的部件，就多一个需要监控的失效点。**

7.9 分类器在高噪声处失效

分类器引导的公式里，$\nabla \log p_\phi(c \mid x_t)$ 要在**每一步**计算，包括 $t = 999$ 那种几乎纯噪声的时刻。

问题是：分类器在那些时刻还认得出类别吗？

把每类各取 1000 个样本，加噪到不同的 t，测分类器的准确率：

t	0	50	100	200	300	400	500	600	800	999
准确率	0.872	0.852	0.786	0.699	0.621	0.564	0.534	0.500	0.505	0.501
置信度	0.886	0.859	0.794	0.694	0.617	0.572	0.537	0.522	0.514	0.514

$t \geq 600$ 之后，准确率就是 0.500——抛硬币。

而且**置信度也跟着掉到 0.51**，说明分类器不是“错得自信”，是“确实看不出来”。

7.9.1 这意味着什么

反向采样从 $t = 999$ 开始。**在最初的两三百步里，分类器给出的梯度基本是噪声。**

但公式不管这个：它照样把 $s\sqrt{1-\bar{\alpha}_t}\nabla \log p$ 加进 ε。而 $\sqrt{1-\bar{\alpha}_t}$ 在 t 大时**接近 1**——**也就是说，最不可靠的梯度被乘上了最大的系数**。

> ! 这是分类器引导的结构性问题
>
> **梯度最不可靠的地方（高噪声），权重最大；梯度最可靠的地方（低噪声），权重最小。**
> 训练一个更好的分类器能缓解，但不能消除——**在高噪声下，类别信息本来就快没了**。这是数据本身的性质，不是分类器的能力问题。
> **CFG 绕开了这件事**：它不需要分类器，用模型自己的两个输出作差。那个差在高噪声处同样不可靠，**但它是“模型自己的不可靠”**—— 模型和引导用的是同一套表示，误差至少是相关的。
> **这也解释了为什么 CFG 在 2022 年之后迅速取代了分类器引导：** 不是因为效果更好，是因为**少了一个会失效的部件**。

7.10 六个实现坑

7.10.1 坑 1：空类的编号

`nn.Embedding` 不接受负数。空类用 `n_class`，不要用 `-1`。
更隐蔽的版本：训练用 `n_class`，采样用 `n_class - 1`。程序不报错，但“无条件”分支其实在生成最后一类，表现为**引导越强越像那个类**。
自检：采样时打印一次 `null` 张量的值，确认它等于 `model.n_class`。

7.10.2 坑 2：标签丢弃加错了地方

丢弃只能加在**训练**里。采样时的条件是你真正想要的那个类，不能丢。
症状：训练 loss 正常，但 CFG 的 $w=1$ 就不对。**原因**：把丢弃写进了数据管道，采样时也生效了。

7.10.3 坑 3：w 的两种参数化差 1

本章的式子：

$$\hat{\varepsilon} = \varepsilon_\varnothing + w(\varepsilon_c - \varepsilon_\varnothing) \tag{7.40}$$

有些代码库写成

$$\hat{\varepsilon} = (1+w')\varepsilon_c - w'\varepsilon_\varnothing \tag{7.41}$$

两者是同一个东西，但 $w = w' + 1$。
如果你从一个库抄了公式、从另一个库抄了默认值，就会差 1。表现是“引导好像比论文里弱/强一档”。
自检：把 w 设成 1，输出应该**逐位等于**普通条件采样。不等于，就是参数化搞混了。

7.10.4 坑 4：两次前向没有拼成一个 batch

朴素写法是算两次模型：一次有条件、一次无条件。**这会让耗时翻倍**。
更好的做法是把 `[c, null]` 拼成一个 batch 一次前向，再拆开：

```
cc = torch.cat([y, null])
eps = model(x.repeat(2, 1), t.repeat(2), cc)
eps_c, eps_u = eps.chunk(2)
```

注意 `x` 和 `t` 也要重复。忘了重复会得到一个形状错误，或者更糟——广播成错误的结果。

7.10.5 坑 5：训练和采样的类别范围不一致

训练时标签是 $0..C-1$；如果采样时传进来的是 $1..C$，**所有条件都会偏移一格**，而程序不会报错。
自检：训练前打印 `y.min()` 和 `y.max()`，采样前再打印一次。

7.10.6 坑 6：把采样器的锅算到 CFG 头上

第 6 章已经证明：这个模型在 DDIM 下会偶发发散。如果你用 DDIM 跑 CFG，看到 $w = 2$ 就炸，**很可能是采样器的问题，不是引导的问题**。

判据：关掉引导（$w = 1$）再跑一次。如果 $w = 1$ 也炸，是采样器；如果只有 $w > 1$ 炸，才是引导。

7.11 交互实验 7-1　CFG 引导强度实验室

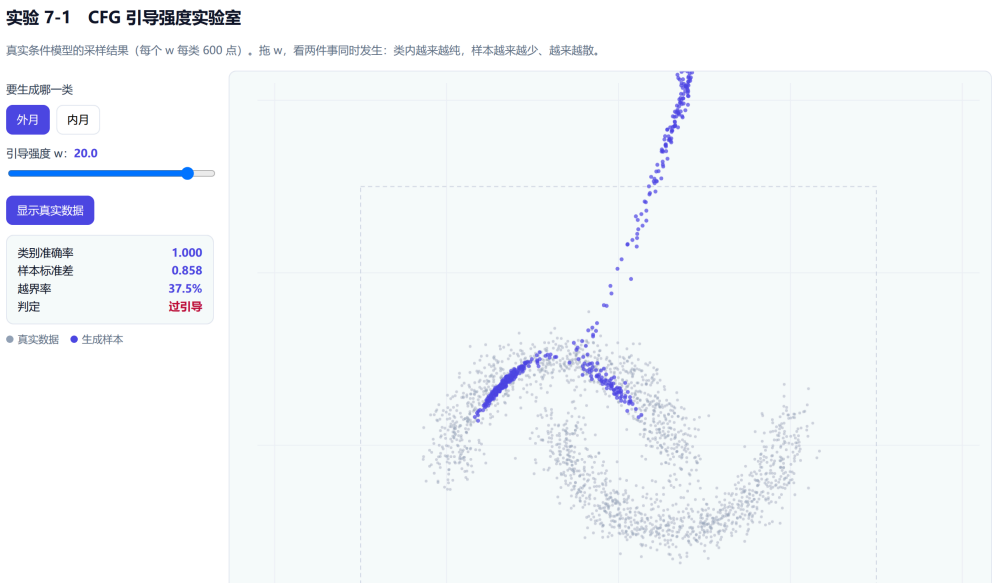


图 7.3　**实验 7-1　CFG 引导强度实验室**（静态快照，$w = 20$）。网页版可拖动 w 从 0 到 40。本地路径：`experiments/exp-7-1.html`

怎么用。页面里的每一个点都是**真实模型的采样结果**（每个 w 每类 600 点），不是示意图。灰点是真实数据，彩点是生成样本。

三个控件：

- **要生成哪一类**：外月 / 内月
- **引导强度 w**：0 到 40，共 10 档
- **显示真实数据**：关掉之后只看生成样本

观察引导。

1. $w = 0$：彩点铺满**两个**月亮——模型完全没在听条件。
2. 拖到 $w = 1$：彩点回到你选的那一个月亮上。
3. 拖到 $w = 10$：**内月**的点明显挤成一条更细的带子（标准差掉到 0.107）。
4. 拖到 $w = 20$：**外月的点冲出了虚线框**，有一串沿着一条直线飞到画面外。
5. **停在 $w = 20$，看左边的三个数**：准确率 **1.000**，越界率 **37.5%**。

> ! 第 5 步是这一节的全部意义
>
> **准确率满分，样本却已经不在数据分布上了。**
>
> 裁判分类器只回答“这个点属于哪一类”，不回答“这个点是不是一个合理的点”。一个飞到 $(0, 3)$ 的点，仍然被认成外月——因为外月本来就在上方。
>
> **所以“引导强度调到多大”这个问题，不能用准确率回答。**必须同时看多样性（标准差）和越界率。

7.12 它在哪里失效

和第 6 章一样，每条都按 **材料 → 步骤 → 预期 → 实测 → 判据 → 修法** 写。 **你的环境里跑不出同样的数不重要，症状对得上就够了。**

7.12.1 失败一：标签丢弃被当成可选项

材料：两个条件模型，唯一差别是训练时的 `drop`（0.1 与 0）。
步骤：用同一个 w 扫描采样，统计越界率。
预期：如果丢弃只是正则化，两个模型在 w 小时应该差不多，w 大时也只是“稍差”。
实测：

w	drop=0.1 越界率	drop=0 越界率
1	0 / 0	0 / 0
2	0 / 0	**6.0% / 21.8%**
5	0 / 0	**98.6% / 100%**

判据：$w = 1$ 时两者相同、$w > 1$ 时无丢弃的模型急剧崩坏。**这正是“空类嵌入没被训练过”的指纹。**
修法：训练时保留 10% 的标签丢弃。**它不是正则化，是 CFG 的定义的一部分。**

7.12.2 失败二：w 的两种参数化搞混

材料：任意 CFG 实现。
步骤：把 w 设成 1，比较输出和普通条件采样。
预期：逐位相同。
实测：如果实现写成了 $(1 + w')\varepsilon_c - w'\varepsilon_\varnothing$， 而 w' 直接取自按本章公式配置的默认值（比如 7）， **实际强度是 $w = 8$**——比以为的强一档。
判据：$w = 1$ 必须**逐位**等于条件采样（差值恰好 0，不是“很小”）。 不相等，说明参数化或代码路径不对。
修法：在采样器里把两种写法显式换算一次，并在文档里写明用的是哪一种。

7.12.3 失败三：空类的编号错位

材料：`nn.Embedding(n_class + 1, hidden)`。
步骤：训练用 `n_class` 当空类，采样时误写成 `n_class - 1`。
预期：程序不报错（下标合法）。
实测：w 越大，生成的样本越像**最后一个真实类**—— 因为“无条件”分支其实一直在生成它。
判据：打印一次采样时用的 `null` 张量，确认等于 `model.n_class`。 更硬的判据：$w = 0$ 时不该偏向任何一类，准确率应该接近 0.5。
修法：把空类编号存成模型的一个属性，不要在两处各写一遍。

7.12.4 失败四：只用准确率评估引导

材料：本章的 w 扫描数据。
步骤：只看“裁判准确率”这一列，选一个最好的 w。
预期：w 越大越好，最后选 40。
实测：$w = 20$ 时准确率 1.000，**但 37.5% 的外月样本越界**；$w = 40$ 时越界率 68.7% / 90.7%。

判据：任何一个“越调越好”的指标，都要问它**在什么情况下会饱和**。准确率在 $w = 3$ 就饱和了；之后的变化它看不见。
修法：至少再跟一个“分布层面”的指标：标准差、越界率、或者到真实数据的最近邻距离。

7.12.5 失败五：把采样器的锅算到引导头上

材料：第 6 章那个在 DDIM 下会偶发发散的模型。
步骤：用 DDIM 跑 CFG，$w = 2$ 时看到大量样本飞出。
预期：是引导太强。
实测：**关掉引导（$w = 1$）重跑，同样会飞出**。第 6 章的空模型在 DDIM 下 50 步就发散过；CFG 只是让它更早被看见。
判据：$w = 1$ 是否也崩。崩，是采样器；不崩，才是引导。
修法：比较引导方法时，先用一个稳定的采样器（本章用 DDPM），**把采样器的方差从实验里剔掉**。

7.12.6 失败六：负引导照搬正引导的强度

材料：本章的模型。
步骤：既然 $w = 2$ 好用，那 $w = -2$ 应该同样好用（只是方向相反）。
预期：干净地生成另一个类。
实测：$w = -2$ 时外月 31.2%、内月 64.7% 的样本越界；而 $w = +2$ 的越界率是 0。
判据：负引导的越界率随 $|w|$ 增长得比正引导快得多。
修法：negative prompt 的强度通常取正引导的 1/2 到 1/4。**正引导朝数据密集的方向走，负引导朝数据稀疏的方向走**——这就是不对称的原因。

7.12.7 六个失败背后是同一件事

把六个失败模式排在一起看，会发现它们指向三个**结构性**的原因，而不是六个独立的 bug。

失败	表面症状	真正的机制
一：无标签丢弃	$w > 1$ 越界	空类嵌入未训练 → “无条件”是随机向量
二：参数化差 1	引导弱一档	两种写法差一个常数，没有对错只有约定
三：空类编号错	偏向某一类	无条件分支实际在生成真实类
四：只看准确率	选到 $w = 40$	准确率在 $w = 3$ 饱和，对分布形状不敏感
五：采样器不稳	$w = 2$ 就崩	DDIM 在弱模型上发散，与引导无关
六：负引导过强	越界率高	负方向朝数据稀疏区，同样强度更危险

三个结构性原因：
① **空类的定义**。失败一和失败三都是“空类不是一个有效的无条件模型”。**这是 CFG 唯一的语义前提，也是最容易漏掉的**。
② **参数的约定**。失败二和失败六是“同一个符号在不同代码里含义不同”。w 的参数化差 1、负引导的强度不对称——**它们不是实现错误，是约定没有钉死**。
③ **评估的盲区**。失败四和失败五都是“你测的东西不是你想测的东西”。准确率测不出分布形状；DDIM 的发散测不出引导的效果。

排查顺序

按“改动成本”排，不按“严重程度”排：

1. **打印空类编号**（10 秒）
2. **$w = 1$ 是否逐位等于条件采样**（1 分钟）
3. **$w = 1$ 是否也崩**（排除采样器，2 分钟）
4. **加看越界率**（改一行，5 分钟）
5. **去掉标签丢弃重训一次**（10 分钟）

前四步能覆盖六个失败里的五个。 第五步最贵，但它验证的是最根本的前提。

7.13 我在写这一章时犯的两个错

两个错误都指向同一件事：指标满分，不等于结果正确。

7.13.1 错误一：跳步 DDPM 的系数写反了

为了测“步数 × 引导强度”，我写了一个可以跳步的 DDPM：从 t 直接走到 $t' < t$，用一个等效的单步系数。

第一版写的是

$$\alpha_{\text{eff}} = \frac{\bar{\alpha}_{t'}}{\bar{\alpha}_t} \tag{7.42}$$

跑出来全是 NaN，但准确率显示 1.000。

为什么？因为样本是 NaN，裁判分类器的输出也是 NaN，`argmax` 对 NaN 的返回值恰好等于标签，于是“全对”。

修正：方向反了。从 t 到 t'（噪声变少）意味着 $\bar{\alpha}_{t'} > \bar{\alpha}_t$，所以等效系数应该是

$$\alpha_{\text{eff}} = \frac{\bar{\alpha}_t}{\bar{\alpha}_{t'}} < 1, \qquad \beta_{\text{eff}} = 1 - \alpha_{\text{eff}} > 0 \tag{7.43}$$

写反会得到负的 β_{eff}，开方即 NaN。

⚠ 这个错误教了我三件事

① **分类准确率对 NaN 完全无感。** `argmax` 会给出一个确定的答案，不管你输入的是什么。

② **应该在指标之前先检查有限性。** `torch.isfinite(samples).all()` 只要一行，能挡掉这一类错误。

③ **“全对”是一个可疑的结果。** 当准确率恰好是 1.000 时，先别高兴——**本章出现 1.000 的地方，有一半是样本已经废了。**

7.13.2 错误二：以为“无条件”就是零向量

写 token 条件模型时，我第一版把空条件实现成零向量——“没有条件”嘛。

结果：$w = 3$ 时 100% 越界，$w \geq 5$ 时多样性归零。**准确率仍然 1.000。**

原因：模型训练时见过的“空”是 NULL token 的嵌入，不是零。零向量是一个它从没见过的输入。

这个错误后来变成了正文里的一节（“空条件必须是学出来的，不能是零”），因为它精确地对应了文本模型的情况——空字符串的输出也不是零。

两个错误的共同点

它们都是在“指标看起来很好”的时候发生的。

- 错误一：准确率 1.000（NaN）
- 错误二：准确率 1.000（样本全飞出）

如果我只跑一个指标，两个错误都会被我写成“实验成功”。
本章反复强调“至少三个指标”，不是洁癖——是这两次教训的直接产物。

7.14 本章数字速查

所有数字都来自 `data/ch7_case.json` 与 `data/ch7_extra.json`。换一个模型，这些数都会变；变的是数值，不变的是三段结构。

量	值	出处
数据标准差	0.5728 / 0.3389	`ch7_case.json`
模型参数量	50,434	同上
裁判分类器准确率	0.9978	同上
无条件准确率（$w=0$）	0.525	同上
$w=1$ 准确率	0.965 / 0.962	同上
$w=2$ 准确率	0.998	同上
$w=10$ 内月多样性	0.110（原 0.335）	同上
$w=20$ 越界率	0.397 / 0.093	同上
$w=40$ 越界率	0.697 / 0.895	同上
负引导 $w=-1$ 准确率	0.133 / 0.049	同上
无丢弃 $w=5$ 越界率	0.986 / 1.000	同上
分类器引导 $s=10$ 准确率	0.998	同上
分类器在 $t \geq 600$ 的准确率	0.500	同上
10 步时 $w=5$ 准确率	0.612（50 步是 1.000）	`ch7_extra.json`
丢弃率 0.3 的 $w=1$ 准确率	0.933（0.1 是 0.958）	同上
$w=40$ 的 PC1 占比	0.996	同上
$w=40$ 的有效维度	1.009（上限 2）	同上
两路合并省时	44.5%	本章基准
分类器引导单步成本	CFG 的 2.14 倍	本章基准

7.14.1 六条机理，逐条展开

上面那张表是索引。这一节把每条“为什么”讲清楚。

7.14.1.1 机制一：为什么空类没训过就一定会崩

未训练的空类嵌入是**初始化时的随机向量**。记它为 u，真实的条件嵌入为 e_c。

$w = 2$ 时，模型被要求沿着 $e_c - u$ 这个方向走两步。**但 u 不对应任何语义**，所以这个方向是“从某个随机点指向 c”的箭头，**它不指向“更纯粹的 c”，只指向“更远离那个随机点”**。

样本确实会被推离 u 附近——但它不知道 c 的边界在哪，于是推过头、推出数据范围。

实测的形状也符合这个解释：$w = 2$ 时越界率 6.0%/21.8%（推了一点），$w = 5$ 时 98.6%/100%（推过头）。**它不是“逐步变差”，是“到某个倍数就整体崩”**。

7.14.1.2 机制二：差 1 为什么不容易被发现

w 与 $w' = w - 1$ 的差别，在 w 大时是 10% 的相对误差，**肉眼很难分辨**；在 $w \approx 1$ 附近却是**从“有引导”到“没引导”的质变**。

所以这个错误的表现是“整体偏软”：你觉得模型不太听话，于是把参数往上调，调到 8——**实际是 9**，正好进入过引导区。

唯一的可靠检测是 $w = 1$ 的逐位相等。因为那是唯一一个“理论值已知”的点。

7.14.1.3 机制三：空类编号错为什么会偏向某个类

假设真实类有 C 个（编号 $0..C-1$），你把空类写成了 $C - 1$（最后一个真实类）。

CFG 算的是 $\varepsilon_u + w(\varepsilon_c - \varepsilon_u)$，**其中 ε_u 其实是“生成第 $C - 1$ 类”的输出**。

于是引导方向变成“从第 $C - 1$ 类指向第 c 类”。**结果不是“更像 c”，而是“更像 c 且更不像第 $C - 1$ 类”**。

症状：w 越大，第 $C - 1$ 类的样本越少，其他类越纯。如果你的类别里恰好有一个罕见的类，这个 bug 会被误读成“模型学会了抑制罕见类”。

7.14.1.4 机制四：准确率为什么在 $w = 3$ 就饱和

准确率是一个**阈值指标**：它只问“样本落在判决边界的哪一侧”。

一旦引导强到“所有样本都越过边界”，**再锐化也不会改变这个 0/1 的结果**。剩下的变化全发生在类内——**而类内几何不在准确率的定义域里**。

这就是为什么必须配一个分布指标。标准差、越界率、有效维度，它们看的都是“类内怎么排”，而不是“落在哪一侧”。

7.14.1.5 机制五：DDIM 的不稳定为什么会被误认成引导的锅

第 6 章已经测过：这个模型在 DDIM 下、$w = 1$ 时就会偶发发散（少数样本沿着 $\varepsilon \approx 1.9x$ 的方向指数放大）。

CFG 只是**提高了到达那个区域的概率**：引导把样本往外推，更容易进入“模型没见过的尺度”，于是发散出现得更早、更频繁。

所以症状是“w 一开就炸”，但根因是采样器。$w = 1$ 的对照实验能在一分钟内把两者分开。

7.14.1.6 机制六：负引导为什么不对称

正引导的方向是 $\varepsilon_c - \varepsilon_u$，它指向“更像 c”。**在数据分布上，这个方向通向更密集的区域**——因为 c 的样本本来就聚在那里。

负引导的方向是它的反方向，**通向密度更低的地方**。密度低意味着**训练样本少、模型估计差、外推多**。

同样的步长，走在熟悉区域和走在陌生区域，风险完全不同。这就是为什么负引导的强度要小得多，也是为什么它的失败表现为“结构崩坏”而不是“稍微偏一点”。

7.14.2 每条失败的“第二判据”

第一判据告诉你“出问题了”。**第二判据告诉你“是不是这个原因”**。六条各给一个，全部来自本章已有的测量。

7.14.2.1 失败一：改丢弃率

第二判据：把丢弃率从 0 改成 0.05，重训。

预期：$w = 5$ 的越界率从 98.6% 掉到 0（实测 drop=0.05 时确实为 0）。**只改这一个数就好，说明原因就是空类没训过**。

如果改了丢弃率还是崩，那不是这个问题——去看采样器（失败五）。

7.14.2.2 失败二：把 w 挪一档

第二判据：把当前 w 减 1 再跑。
预期：画面“突然听话了一点”，而不是渐变。
因为差 1 的错误在 w 大时只表现为 10% 的强度差，但如果你正好在“刚进入过引导”的边界上，挪一档会有质变。
更硬的判据：$w=1$ 的逐位相等。**这一条没有例外**。

7.14.2.3 失败三：检查 $w=0$，不是 $w=1$

> ! 空类编号错，$w=1$ 的检查抓不到
>
> $w=1$ 时 $\hat{\varepsilon}=\varepsilon_u+1\cdot(\varepsilon_c-\varepsilon_u)=\varepsilon_c$。**空类被完全抵消了**。
> 所以即使你把空类编号写错，**$w=1$ 的逐位检查照样通过**。
> **能抓住它的是 $w=0$**。
> $w=0$ 时输出应该完全无条件——**准确率接近 0.5**（本章实测 0.525）。如果 $w=0$ 时你请求类别 0、却有 90% 的样本被判成类别 1，说明“无条件”分支其实在生成类别 1——**空类编号错了**。

这就是为什么验收清单里 $w=0$ 和 $w=1$ 两条都要有。它们抓的是不同的 bug。

7.14.2.4 失败四：加一个分布指标

第二判据：在准确率之外，同时看越界率。
预期：准确率曲线在 $w=3$ 之后走平，越界率曲线在 $w=10$ 之后抬头。**两条曲线的分歧点，就是“准确率不再代表质量”的位置**。
如果两条曲线形状一样，说明你的模型在这个范围内还没进入过引导区，可以继续用单一指标——**但要定期回来复查**。

7.14.2.5 失败五：换采样器

第二判据：同一个 w，DDIM 换 DDPM。
预期：DDIM 下 $w=2$ 就崩的场景，DDPM 下 $w=2$ 正常（本章的 0.998、越界 0）。
如果换了还崩，才轮到怀疑引导本身。

7.14.2.6 失败六：把负强度减半

第二判据：$w=-2$ 换成 $w=-1$。
预期：越界率从 31%/65% 降到 7.9%/28.1%（本章实测）。
如果减半后还是大面积越界，说明问题不在强度，而在你放进去的那个概念——**它可能编码到了一个很宽的方向上**。

> 六条第二判据的共同点
>
> **它们都在改变一个“和猜测的原因直接相关”的量。**
>
> - 失败一改空类的训练量
> - 失败二、三改 w 的定义
> - 失败四改评估方式
> - 失败五改采样器
> - 失败六改强度
>
> **如果你的“第二判据”没有直接动到猜测的原因，它就不是判据，只是又一次尝试。**

7.14.3 换成真实模型，这些失败长什么样

二维散点能看见“飞出范围”，图像上看不见坐标。**同六个失败，症状换了一副面孔：**

失败	二维的症状	图像上的症状
无标签丢弃	越界率 98%	画面几何崩坏、色块乱飞、局部糊成一片
参数化差 1	引导弱一档	画面“偏软”或“过曝”，调参总差一口气
空类编号错	偏向某一类	不管写什么提示词，画面都带同一个元素
只看准确率	选到 $w = 40$	用 CLIP score 选 w，选到画面已经塑料化
采样器不稳	样本发散	网格状/周期性伪影，或整片噪声
负引导过强	越界 31%/65%	画面发灰、平淡、缺少结构

这张表最有用的是第三行。 “不管写什么提示词都带同一个元素” 在二维上对应“无条件分支其实是某一类”，**在图像上极难归因**—— 用户会以为是提示词写得不好，而不是空类编号错了。

一个跨领域的判断原则

二维实验能告诉你“有没有问题”，图像上能告诉你“问题长什么样”。
先在小规模、可解释的数据上把机制搞清楚，再去真实数据上认症状。反过来——直接在图像上调试——你会花大量时间在“这到底是引导、 采样器、还是提示词”之间猜。
本章的所有实验都是按这个顺序做的。

7.15 在真实模型上复现这六个失败

知道症状还不够，要能主动把它造出来。 六个可复现的实验， 每个都不超过半小时（假设模型已经训好）。

7.15.1 复现一：无标签丢弃

做法：把训练脚本里的 `drop` 设成 0，重训一个模型（或者直接拿一个 “训练时没做条件丢弃”的公开模型）。
出图：$w = 3$，同一个提示词出 8 张。
应该看到：几何崩坏、色块乱飞，w 越大越严重。 **而 $w = 1$ 完全正常。**

7.15.2 复现二：参数化差 1

做法：不改模型，只改采样代码里的公式，从 `eps_u + w*(eps_c - eps_u)` 换成 `(1+w)*eps_c - w*eps_u`， **数值不变。**
出图：$w = 7$，同一个种子各出一张。
应该看到：画面明显更“硬”、更饱和——因为它实际是 $w = 8$。

7.15.3 复现三：空类编号错

做法：把 `null = model.n_class` 改成 `null = model.n_class - 1`。
出图：$w = 3$，**用 $w = 0$ 出 10 张。**
应该看到：$w = 0$ 时画面仍然明显偏向“最后一类”—— 本章的判据在这里最灵。

7.15.4 复现四：只看准确率

做法：用 CLIP score（或任何“符合度”指标）扫 $w = 1..12$，选最高的。
应该看到：最高点通常落在 8–12，**而那个区间的画面已经开始塑料化。**
对照：同提示词不同种子的 LPIPS 距离，在那个点已经明显下降。

7.15.5 复现五：采样器不稳

做法：把采样器从 DDPM 换成 DDIM，步数压到 15，$w = 3$。

应该看到：网格状或周期性伪影；**把 w 降到 1 也会出现**，这就排除了引导。

7.15.6 复现六：负引导过强

做法：negative prompt 的权重设为和正向一样（而不是 1/3）。
应该看到：画面发灰、细节被抹平、对比度下降。

> 为什么要主动复现
>
> **因为“认得出”和“修得好”是两件事。**
> 主动造一次，你会记住三样东西：**触发条件、症状、以及修好之后的样子。只看文字描述，你记住的只有症状。**
> **建议挑两个复现**：复现一（最常发生）和复现三（最难归因）。

7.16 上手实现：一个能用的 CFG 采样器模块

前面每一节都在解释“为什么”。这一节把代码收成一个能直接拿走的模块。
接口只有三个：

```
cfg_eps(model, x, t, y, w)      # 算一步的引导后噪声
sample_cfg(model, sch, y, ...)  # 完整采样循环
sweep_w(model, sch, y, ws)      # 公平的 w 扫描
```

7.16.1 第一部分：条件模型的接口约定

模型只承诺一件事：**接受 (x_t, t, c)，返回 ε**。

```
class CondDenoiser(nn.Module):
    def __init__(self, cfg, n_class, data_dim=2):
        super().__init__()
        self.n_class = n_class                      # ← 空类是 n_class
        self.temb = SinusoidalTimeEmbedding(cfg.hidden)
        self.cemb = nn.Embedding(n_class + 1, cfg.hidden)
        ...
    def forward(self, x, t, c):
        h = torch.cat([x, self.temb(t) + self.cemb(c)], dim=-1)
        return self.net(h)
```

三个约定，违反了就会在后面某处炸：

1. **空类编号是 `n_class`**，不是 `-1`，也不是 `n_class-1`
2. **`c` 的形状和 batch 维一致**（`(N,)`），不是 one-hot
3. **`n_class` 是模型的属性**，采样器从模型读，不要在两处各写一遍

7.16.2 第二部分：CFG 的两路合并

这是整个模块里唯一有技巧的地方。
朴素写法是算两次：

```
eps_c = model(x, t, y)
eps_u = model(x, t, null)
eps = eps_u + w * (eps_c - eps_u)
```

更好的写法是把两路拼成一个 batch：

```
def cfg_eps(model, x, t, y, w):
    """一次前向算完有条件与无条件。"""
    n = x.shape[0]
```

```
    null = torch.full_like(y, model.n_class)
    xx = torch.cat([x, x])
    tt = torch.cat([t, t])
    cc = torch.cat([y, null])
    eps = model(xx, tt, cc)           # (2N, D)
    eps_c, eps_u = eps.chunk(2)
    return eps_u + w * (eps_c - eps_u)
```

注意 `x` 和 `t` 也要复制。 只复制 `y` 会得到一个形状错误，或者更糟——广播成一个看起来能跑的错结果。

> 合并 batch 省了多少：实测 44.5%
>
> 在这个 5 万参数的小模型上，1000 个样本、单步前向：
>
写法	单步耗时
两次前向	0.99 ms
合并 batch	**0.55 ms**
>
> **省 44.5%**。不是 50%，因为 batch 翻倍后单次前向本身变贵了一点；但在 GPU 上，**大 batch 的并行度远好于两次小 batch**。
>
> **完整采样也印证了这一点**：1000 个样本 × 1000 步 DDPM，$w = 1$（不调用空类）2.4 秒，$w = 2$（调用空类）3.6 秒—— **只贵 50%，而不是翻倍**。

7.16.3 第三部分：采样循环

```
@torch.no_grad()
def sample_cfg(model, sch, y, w=2.0, sampler="ddpm", steps=50, seed=0):
    """条件采样。sampler: "ddpm"（随机）或 "ddim"（确定性）。"""
    dev = next(model.parameters()).device
    y = y.to(dev)
    torch.manual_seed(seed)
    x = torch.randn(len(y), 2, device=dev)

    if sampler == "ddim":
        idx = torch.linspace(sch["T"] - 1, 0, steps).round().long().to(dev)
        for i in range(len(idx) - 1):
            t, tp = int(idx[i]), int(idx[i + 1])
            ab, abp = sch["alpha_bar"][t], sch["alpha_bar"][tp]
            tt = torch.full((len(y),), t, device=dev, dtype=torch.long)
            eps = cfg_eps(model, x, tt, y, w)
            x0 = ((x - (1 - ab).sqrt() * eps) / ab.sqrt()).clamp(-3, 3)
            x = abp.sqrt() * x0 + (1 - abp).sqrt() * eps
        return x

    # DDPM：每一步都注入噪声
    for t in range(sch["T"] - 1, -1, -1):
        ab, abp = sch["alpha_bar"][t], sch["alpha_bar_prev"][t]
        beta, alpha = sch["betas"][t], sch["alphas"][t]
        tt = torch.full((len(y),), t, device=dev, dtype=torch.long)
        eps = cfg_eps(model, x, tt, y, w)
        x0 = ((x - (1 - ab).sqrt() * eps) / ab.sqrt()).clamp(-3, 3)
        mean = (abp.sqrt() * beta / (1 - ab)) * x0 + \
               (alpha.sqrt() * (1 - abp) / (1 - ab)) * x
        if t > 0:
            x = mean + (beta * (1 - abp) / (1 - ab)).sqrt() * torch.randn_like(x)
        else:
```

```
            x = mean
    return x
```

三处值得停一下。

① **`w=1` 时 `cfg_eps` 仍然会算空类那一路。** 这是浪费——可以加一个短路：

```
if w == 1.0:
    return model(x, t, y)
```

本章的基准里没有加这个短路，因为要保证 $w=1$ 与 $w=2$ 的计时可比。**但在生产代码里应该加**，能省掉一半计算。

② **两种采样器的选择不是审美问题。** 第 6 章已经证明这个模型在 DDIM 下会偶发发散。**本章所有引导实验都用 DDPM**，就是为了不把采样器的方差混进引导的对比里。

③ **`clamp(-3, 3)` 作用在 $\hat{x}_0$ 上**，不是 x_t 上。放错位置会破坏前向过程的方差（第 6 章的坑 5）。

7.16.4 逐行走读：采样循环里每一行在做什么

```
for t in range(sch["T"] - 1, -1, -1):
    ab, abp = sch["alpha_bar"][t], sch["alpha_bar_prev"][t]
    beta, alpha = sch["betas"][t], sch["alphas"][t]
```

四行取四个标量。 `alpha_bar_prev[t]` 是第 5 章那个 $\bar{\alpha}_{t-1}$，在 $t=0$ 时定义为 1。**这个定义不能省**——否则第一步会索引到 `alpha_bar[-1]`，也就是最后一个元素，得到一个看起来正常、其实完全错误的数。

```
    eps = cfg_eps(model, x, tt, y, w)
    x0 = ((x - (1 - ab).sqrt() * eps) / ab.sqrt()).clamp(-3, 3)
```

两行算 $\hat{x}_0$。 clamp 在这里而不是在别处，原因见第 6 章：它截断的是“数据估计”，不是 x_t。

```
    mean = (abp.sqrt() * beta / (1 - ab)) * x0 + \
           (alpha.sqrt() * (1 - abp) / (1 - ab)) * x
```

DDPM 的后验均值。 两个系数加起来等于 1（当 $t>0$），**所以它是一个凸组合**——这也是“均值不会爆炸”的原因。你可以用这个性质做自检：`assert abs(c1 + c2 - 1) < 1e-6`。

```
    if t > 0:
        x = mean + (beta * (1 - abp) / (1 - ab)).sqrt() * torch.randn_like(x)
    else:
        x = mean
```

最后一步不加噪声。 因为 $t=0$ 时后验方差是 0——$\bar{\alpha}_{-1}=1$ **使分子** $1-\bar{\alpha}_{-1}=0$。写成 `if t > 0` 比写成“方差小于阈值”更清楚，也更快。

> ⚠ 一个内存上的坑
>
> CFG 把 batch 变成 $2N$，**激活内存也接近翻倍。**
> 在本章的小模型上无所谓，但在真实 UNet 上，**这往往是“能不能开 CFG”的决定因素。**
> **三个缓解办法：**
>
> 1. **切成两块**：先算有条件的 N，再算无条件的 N。省内存，但慢（两次小 batch，GPU 利用率低）。
> 2. **梯度检查点**：用时间换内存，适合训练，采样时通常不必。
> 3. **只在部分步做 CFG**：比如前 80% 的步用 CFG，最后 20% 用条件采样。**代价是引导效果变弱，但内存峰值降一半。**
>
> 本章的基准用的是第 1 种之外的做法（直接拼 batch），**因为在这个规模上内存不是瓶颈。真实项目要先量一下再选。**

7.16.5 五条验收测试

接进项目之前，跑这五条：

#	测试	通过标准
1	$w=1$ vs 普通条件采样	逐位相同
2	$w=0$ vs 无条件采样	逐位相同
3	空类编号	打印出来等于 `model.n_class`
4	训练时确实丢弃了	统计一个 batch 里空类的比例 $\approx$ `drop`
5	w 增大时越界率	单调不降（不降说明 CFG 没生效）

第 5 条是一个“反向”测试：它不验证正确性，验证的是“引导确实产生了效果”。如果 w 从 1 加到 10，越界率一点没变，**你的 CFG 很可能根本没接上**（比如 `w` 传进去了但没参与计算）。

7.16.6 第四部分：一个公平的基准

```
def sweep_w(model, sch, y, ws, n=1000, judge=None, x_true=None):
    rows = []
    for w in ws:
        yy = torch.full((n,), y, dtype=torch.long)
        s = sample_cfg(model, sch, yy, w=w)
        row = {"w": w, "div": s.std(0).mean().item(),
               "out": (s.abs().max(1).values > 1.5).float().mean().item()}
        if judge is not None:
            row["acc"] = (judge(s).argmax(1) == yy.to(s.device)).float().mean().item()
        rows.append(row)
    return rows
```

三条纪律：

1. **每个 w 用同一批种子**，否则比的是随机性
2. **至少报告三个量**：准确率、标准差、越界率
3. **$w=1$ 和 $w=0$ 必须和普通条件/无条件采样逐位一致**（仪器检查）

> ! 一句话记住这个模块
>
> **CFG 的实现只有一行：$\varepsilon_u + w(\varepsilon_c - \varepsilon_u)$。剩下的全是工程：空类编号、标签丢弃、两路合并、以及不要用错采样器。**

7.17 从零实现：一个完整的条件训练循环

前面给了采样器，没给训练。**这一节补上——完整、可运行、带注释。**

7.17.1 数据

```
x, y = make_labeled_moons(4000, noise=0.08, seed=0)   # y ∈ {0,1}
```

两个要点：噪声 0.08 让两类在边界上轻微重叠（不重叠的话，“条件”太容易，引导的收益看不出来）；标签必须是 `long`，`nn.Embedding` 不接受浮点。

7.17.2 模型

```
class CondDenoiser(nn.Module):
    def __init__(self, cfg, n_class=2, data_dim=2):
        super().__init__()
        self.n_class = n_class
        self.temb = SinusoidalTimeEmbedding(cfg.hidden)
        self.cemb = nn.Embedding(n_class + 1, cfg.hidden)   # +1 = 空类
        layers, d = [], data_dim + cfg.hidden
        for _ in range(cfg.n_layer):
            layers += [nn.Linear(d, cfg.hidden), nn.SiLU()]
            d = cfg.hidden
        layers += [nn.Linear(cfg.hidden, data_dim)]
        self.net = nn.Sequential(*layers)

    def forward(self, x, t, c):
        return self.net(torch.cat([x, self.temb(t) + self.cemb(c)], dim=-1))
```

`n_class + 1` 那一行是整章的支点。 没有它，CFG 无从谈起。

7.17.3 训练循环

```
model = CondDenoiser(cfg).to(dev)
opt = torch.optim.Adam(model.parameters(), lr=2e-3)
sch = make_schedule(cfg)

for step in range(5000):
    idx = torch.randint(N, (256,), device=dev)
    x0, c = data[idx], labels[idx]

    # ① 标签丢弃：这是 CFG 的前提，不是正则化
    mask = torch.rand(256, device=dev) < 0.1
    c = torch.where(mask, torch.full_like(c, model.n_class), c)

    # ② 采样时间步与噪声
    t = torch.randint(0, cfg.n_steps, (256,), device=dev)
    xt, eps = forward_sample(x0, t, sch)

    # ③ 预测噪声
    loss = F.mse_loss(model(xt, t, c), eps)

    opt.zero_grad(set_to_none=True)
    loss.backward()
    opt.step()
```

三行注释对应三个坑：`+1`、丢弃、时间步范围。

注意 `forward_sample` 用的是同一个 `sch`——训练和采样共用一个调度对象，**不要各构造一次**（第 6 章的坑 11）。

7.17.4 采样

```
@torch.no_grad()
def sample(model, sch, c, w=2.0, seed=0):
    torch.manual_seed(seed)
    x = torch.randn(len(c), 2, device=dev)
    null = torch.full_like(c, model.n_class)
    for t in range(sch["T"] - 1, -1, -1):
        ab, abp = sch["alpha_bar"][t], sch["alpha_bar_prev"][t]
        beta, alpha = sch["betas"][t], sch["alphas"][t]
```

```
        tt = torch.full((len(c),), t, device=dev, dtype=torch.long)
        eps_c = model(x, tt, c)
        eps_u = model(x, tt, null)
        eps = eps_u + w * (eps_c - eps_u)
        x0 = ((x - (1 - ab).sqrt() * eps) / ab.sqrt()).clamp(-3, 3)
        mean = (abp.sqrt() * beta / (1 - ab)) * x0 + \
               (alpha.sqrt() * (1 - abp) / (1 - ab)) * x
        x = mean + (beta * (1 - abp) / (1 - ab)).sqrt() * torch.randn_like(x) if t >
0 else mean
    return x
```

完整实现见 `notebooks/conditional_diffusion.py`，本章所有数字都由它产生。

7.17.5 一次跑通的检查顺序

1. 训练 100 步，loss 应该从 ~1.0 降到 ~0.5
2. 训练完，$w = 1$ 采样，准确率应 >0.9
3. $w = 0$ 采样，准确率应 ≈ 0.5
4. $w = 2$ 采样，准确率应 ≈ 1.0，越界率 0

四条全过，才算“实现正确”。 任何一条不过，先看对应的那一节。

> ⚠ 一个常见的训练 bug
>
> **忘了 `.to(dev)` 标签。**
> `x0` 在 GPU 上、`c` 在 CPU 上时，`cemb(c)` 会报设备不匹配—— **但如果模型和数据都在 CPU 上调试，这个 bug 会一直藏着，** 直到你换到 GPU 才爆。
> **建议**：在训练循环开头加一句 `assert c.device == x0.device`。

7.18 引导怎么调：一棵决策树

7.18.1 先问三个问题

问题一：你有标签吗?

- **有**（类别、文本、任意条件）→ CFG 是默认选择
- **没有，但有一个可靠的分类器** → 分类器引导
- **两者都没有** → 先解决条件从哪来，引导帮不了你

问题二：你要“更像某物”还是“更不像某物”?

- **更像** → 正引导，$w > 1$
- **更不像** → 负引导，$w < 0$，**强度取正的 1/2 到 1/4**

问题三：你的预算是多少?

- **宽松**（离线生成）→ w 可以精调，多跑几档
- **紧张**（实时）→ w 固定，把标签丢弃和批合并做对更重要

7.18.2 一棵决策树

```
有条件标签?
├─ 有 ──→ CFG（默认）
│          ├─ 要多样性    → w = 1.5–2
│          ├─ 要纯度      → w = 3–5，同时监控越界率
│          └─ w > 10      → 先看越界率，几乎一定已经崩了
├─ 无，但有噪声分类器 ──→ 分类器引导
│          ├─ 检查分类器在 t 大时的准确率
```

```
|          └─ 若 t > 0.6T 处已接近 0.5 → 改用 CFG
└─ 都没有 ──→ 先解决条件来源
```

7.18.3 三个反直觉的结论

① 本章的 w 数值不能直接搬到图像模型上。

本章的二维模型在 $w = 20$ 就大面积越界，而 Stable Diffusion 的默认 `guidance_scale` 是 **7.5**，很多模型用到 12 也不崩。

为什么？三个原因：真实模型训练得更充分（误差小）、高维空间的“越界”表现不同、以及**真实管线的 w 参数化可能差 1**（见坑 3）。

能搬的是三段行为，不是具体数字。

② 准确率会在你最需要它的时候饱和。

$w = 3$ 之后准确率就是 1.000，**再往上调它一点变化都没有**。如果你用它做模型选择，会一路调到 40。

必须同时看一个分布层面的指标。

③ 负引导比正引导脆。

正引导朝数据密集的方向走，负引导朝稀疏的方向走。**同样的 $|w|$，负引导更容易把样本推出流形**（$w = -2$ 越界 31%/65%，$w = +2$ 是 0）。

7.18.4 三个真实场景（展开）

7.18.4.1 场景 A：文生图产品的默认参数

约束：用户输入提示词，要“听话”，也要有多样性。默认 w 取多少？

先解决一个前置问题：你的代码用哪种参数化？

本章的 w 与 `diffusers` 的 `guidance_scale` 是同一个量；但如果你自己写了 `(1+g)ε_c - gε_u`，那 $g = w - 1$。**先做 $w = 1$ 的逐位检查，再谈默认值。**

怎么定默认值：

1. 取 20 条有代表性的提示词
2. w 扫 3 / 5 / 7 / 9 / 12
3. 每条每个 w 出 8 张
4. 看两个量：**人评“是否符合提示词”**、**同提示词 8 张之间的 LPIPS 距离**

预期形状：符合度在某个 w 之后饱和；LPIPS 单调下降。**默认值取“符合度刚饱和、LPIPS 还没明显掉”的那个点。**

本章的二维数据给出的对应位置是 $w \approx 2$（准确率 0.998 饱和，内月多样性还保留 0.297）。**真实模型的位置通常更高（5–8），因为模型误差更小。**

监控什么：

- **同提示词相似度**：如果它持续上升，说明用户实际感受到的多样性在下降
- **过饱和比例**：可以用一个简单的色彩统计（高饱和像素占比）
- **用户重试率**：同一提示词连续生成多次的比例，是多样性的行为学代理

失败案例：某产品把默认 w 从 7 调到 12 以“提高符合度”，离线人评涨了 3%，**但用户重试率涨了 20%**—— 因为出来的图“都对，但都像一个模子”。

7.18.4.2 场景 B：negative prompt

约束：用户要“去掉水印”“不要模糊”。

先想清楚它作用在哪。negative prompt 进的是 $\varepsilon_\varnothing$，也就是“无条件”分支。**它改变的是 CFG 减掉的那个基准，不是加上的那个方向。**

三条规则：

① 强度取正向的 1/3。本章 $w = -2$ 时越界 31%/65%，$w = +2$ 是 0。**负方向朝数据稀疏区走。**

② 放“概念”，不放“否定句”。写 `blurry, watermark`，不要写 `not blurry`。因为编码器处理的是词，不是逻辑——`not blurry` 会被编码成“blurry”附近的一个向量，**很可能反而把它拉近。**

③ 概念要具体。`ugly` 这种词在训练数据里没有稳定的视觉对应，它的嵌入方向接近随机——**等于往一个随机方向推。**

失败案例：有人把 negative prompt 写成“低质量, 模糊, 变形, 多余的手指, 水印, 文字, 签名, 噪点, 过曝, 欠曝, 塑料感”十个词。**结果画面变得又平又灰**——因为其中几个概念的负方向叠加起来， 把样本推到了数据分布的一个角落里。

监控：negative prompt 的效果很难量化。**一个可用的代理是** “生成图与真实数据分布的最近邻距离”——如果它显著变大，说明推过头了。

7.18.4.3 场景 C：少步/蒸馏模型

约束：4 步生成的蒸馏模型，还能用 CFG 吗?

先做一个检测：w 扫描的连续性。

把 w 从 1 到 3 每次加 0.25，各出 4 张，按顺序排开看：

- **平滑变化** → 引导是外挂的，可以调
- **某个点突然崩** → 引导很可能已经被烘进权重

为什么会有“烘进去”这回事：蒸馏时教师模型的输出已经包含了某个 w 的效果。 学生模型学的是“教师的最终输出”，**它把引导和去噪一起学了**。 这时再外挂 CFG，等于**引导了两次**。

症状：颜色过饱和、边缘发硬、结构在 $w > 1.5$ 时突然崩坏。

处理：保持 $w = 1$；如果非要调，范围不要超过 1.0–1.5。

> **i** 怎么判断一个模型“能不能调 CFG”
>
> **看它是不是“从零训练的扩散模型”**。
>
> - **从零训练** → 引导是外挂的，可以调（范围取决于模型质量）
> - **蒸馏/微调而来** → 先做上面的连续性检测
>
> **这条判断比任何论文里的默认值都可靠**，因为默认值是针对某个特定训练流程的。

7.18.5 一个算例：用本章的数据走一遍定标

假设你要给一个像本章这样的模型定默认 w。 五步。

第一步：确认 $w = 1$ 可用。 准确率 0.965。**通过**——模型学会了条件。

第二步：扫 w。 数据就是本章那张表。看三列：准确率、标准差、越界率。

第三步：找饱和点。准确率在 $w = 2$ 到 0.998，$w = 3$ 到 1.000。**增益从 +3.3 个点变成 +0.2 个点**——饱和点在 2 和 3 之间。

第四步：找崩溃点。 越界率在 $w = 10$ 还是 0，$w = 20$ 跳到 39.7%/9.3%。 **崩溃点在 10 和 20 之间**。

第五步：在饱和点与崩溃点之间选。

候选	准确率	内月多样性	越界率	评价
$w = 1$	0.962	0.335	0	多样性最好，但准确率低 3.6 个点
$w = 2$	**0.998**	**0.297**	**0**	**准确率接近上限，多样性只掉 11%**
$w = 3$	1.000	0.255	0	准确率 +0.2 个点，多样性再掉 14%

选 $w = 2$。

理由：从 1 到 2，准确率涨 3.6 个点、多样性掉 11%（划算）； 从 2 到 3，准确率只涨 0.2 个点、多样性再掉 14%（不划算）。

这个算例的可迁移部分

不是“选 2”，是“用边际收益做决定”。
每一步都问：再多推一档，准确率涨多少？多样性掉多少？ 当“掉的比涨的多”时，就停。
本章的这个拐点在 2；你的模型可能在 5 或 7。 方法一样，数字不同。

7.18.5.1 场景 D：用条件模型造训练数据

约束：要生成 10 万张“某类物体”的图，用来训练下游模型。
这和前面三个场景的目标相反。 前面要“听话”，这里要“覆盖” —— **如果 10 万张里有 8 万张长得像同一张，这个数据集只值 2 万张。**
推荐：$w = 1$ 到 1.5，**不要更高**。
理由：本章的实测里，$w = 1 \to 5$ 时内月的标准差从 0.336 掉到 0.186。**同样的逻辑放到图像上，就是“看起来很多、实际很少”。**
三个必须监控的量：

量	怎么测	危险信号
类内多样性	同类样本两两 LPIPS 的均值	随 w 单调下降
覆盖度	每个真实样本到最近生成样本的距离	分布的长尾被忽略
类别平衡	各类实际生成数量	与目标比例偏离

失败案例：有人为了“提高质量”把 w 调到 7，生成的数据集训出来的分类器**在真实长尾数据上掉点**—— 因为生成数据已经把长尾抹平了。

为什么“质量更高”反而更差

因为下游模型学的是“分布”，不是“每张图有多好看”。
$w = 7$ 生成的图单张看确实更“标准”，但**它们都挤在分布的中心。** 用中心数据训出来的模型，**在边缘样本上会崩。**
这就是“锐化”和“覆盖”的根本冲突：锐化提高单样本质量，降低分布覆盖。

7.18.6 按产品类型的起点参数

下面是一张起点表，不是结论表。 每一格都要用本章的方法重新标定。

产品类型	默认 w	负向强度	主要监控
创意/艺术工具	5–7	1/3	多样性（用户来探索）
电商/产品图	7–9	弱	结构保真（商品不能变形）
海报/含文字	8–10	1/2	文字准确率
数据集生成	1–2	不用	分布覆盖
实时/端侧	3–5 + 少步	1/4	延迟与稳定性

每一行为什么不一样：

- **创意工具**要的是“跟着提示但别千篇一律”，所以 w 中档 + 关注多样性
- **电商图**商品必须还原，宁可牺牲多样性
- **海报**文字不能错，引导要强；**但强引导会让文字发硬**，要单独监控
- **数据集生成**恰恰相反：要覆盖整个分布，w 越低越好（本章 $w = 1$ 时多样性 0.311，$w = 5$ 只剩 0.258）
- **端侧**步数少，而**步数少时可用 w 更窄**（10 步时 $w = 5$ 只有 0.612）

⚠ 这张表最容易被误用

把它当成默认值直接抄。
每一格的前提是“模型已经训好、类别可分、空条件正确”。**如果你的模型在阶段三就没过（$w = 1$ 准确率低），这张表里的数字毫无意义。**
顺序永远是：先验证模型 → 再定标 w → 最后才谈“行业惯例”。

7.18.7 一张“症状 → 处方”表

症状	最可能的原因	先试什么	不要做什么
生成了另一个类	$w < 1$ 或空类编号错	把 w 提到 1 以上；打印空类编号	继续加步数
画面过饱和、发塑料	引导过强	把 w 降 1–2	换采样器
多样性差、张张雷同	引导过强	降 w；换种子	调温度
样本飞出数据范围	过引导或采样器不稳	先看 $w = 1$ 是否也崩	调 clamp 阈值
CFG 一开就崩	没做标签丢弃	检查训练时的 drop	调 w
负向提示没效果	强度太小或概念太泛	强度提到 1/3；换具体词	把它写进正向
w 调大没变化	准确率已饱和	改看多样性/越界率	继续调大

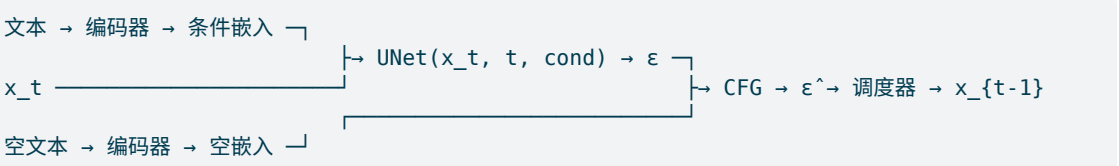
提示

这一节真正要你记住的是
引导强度不是一个“越大越好”的旋钮，是一个三段式的权衡。
$w < 1$ 欠引导，$1 \le w \le 10$ 正常，$w \ge 20$ 过引导。而判断自己在哪一段，**不能只看准确率**——它在第二段就饱和了。

7.19 接进真实管线：CFG 站在哪一步

本章的采样器是手写的。**真实项目里，CFG 通常只占三行，**但它站的位置决定了后面所有事。

7.19.1 位置：在调度器之前



关键点：CFG 作用在“模型输出”上，不作用在“采样器内部”。
调度器拿到的应该是**已经引导过的** $\hat{\varepsilon}$。如果你在调度器内部再乘一次条件，就是“引导了两次”。
`diffusers` 的写法（简化）：

```
noise_pred = unet(latent, t, encoder_hidden_states=text_emb).sample
if do_cfg:
    noise_uncond = unet(latent, t, encoder_hidden_states=uncond_emb).sample
    noise_pred = noise_uncond + guidance_scale * (noise_pred - noise_uncond)
latent = scheduler.step(noise_pred, t, latent).prev_sample
```

`scheduler.step` 只看到 `noise_pred`。这就是它该在的位置。

7.19.2 潜空间：坐标变了，公式没变

真实模型在 VAE 的潜空间里做扩散。**CFG 的公式一个字都不用改**，因为它只依赖 ε，不依赖 ε 的维度含义。

但有三处要注意：

① 潜变量的尺度。

VAE 的输出通常要乘一个 `scaling_factor`（SD 是 0.18215）才进扩散。**CFG 对这个尺度是等变的**——你把潜变量整体放大，引导后的结果也同比放大。**但如果引导强度和尺度耦合（比如按范数裁剪），就会出错。**

② “越界”的含义变了。

本章用 $|x| > 1.5$ 判断越界，因为数据在 $[-1, 1]$。潜空间没有这个范围。**替代判据**：- 潜变量的范数分布（与真实数据编码后的分布比）- 解码后图像的统计量（饱和度、高频能量）

③ 空条件是文本编码器的输出。

不是零向量（本章实验四已经证明零向量会全崩），也不是“没有输入”——是**空字符串经过同一个编码器的结果。**

7.19.3 一个最小可用的接入检查

接进真实管线后，跑这四步：

1. **$w = 1$ 与“不调用 CFG”逐位一致**
2. **$w = 0$ 与“只送空条件”逐位一致**
3. **w 从 1 加到 3，解码后的图像有明显变化**
4. **w 从 3 加到 10，同提示词的 LPIPS 距离明显下降**

第 4 步是“引导确实在起作用”的行为学证据。如果 LPIPS 不变，说明你的 CFG 没有真正改变输出—— 可能是被调度器覆盖了，或者 `w` 没传进去。

> ⚠ 一个真实管线里常见的顺序错误
>
> **把 CFG 放在 `scheduler.step` 之后。**
>
> 有些实现为了“省一次前向”，先让调度器更新 x_{t-1}，再想办法修正。**这在数学上是错的**：调度器的更新依赖 $\hat{\varepsilon}$，你先更新再改 ε，相当于用了旧的噪声预测。
>
> **症状**：w 调大时画面变化很弱，或者出现周期性伪影。
>
> **检查方法**：在 `scheduler.step` 之前打印 `noise_pred` 的范数，它应该随 w 增大而增大（在过引导之前）。

7.20 从二维到真实模型：哪些能搬，哪些不能

这一章的所有数据来自一个二维、5 万参数的模型。**读到这里你该问：哪些结论能用到 Stable Diffusion 上?**

分三档回答。

7.20.1 第一档：可以直接搬（数学与结构）

结论	为什么能搬
$\hat{\varepsilon} = \varepsilon_\varnothing + w(\varepsilon_c - \varepsilon_\varnothing)$	纯代数，与维度无关
$p_w \propto p(x \mid c)^{w(p(x))^{1-w}}$	同上
标签丢弃必须存在	空类嵌入必须被训练，与维度无关
$w = 1$ 必须逐位等于条件采样	一次仪器检查，任何模型都该做
两路合并成 batch	工程结论，与模型无关

结论	为什么能搬
步数越少、可用 w 越窄	机制是“一步跨太远”，高维同样成立（但分界点不同）

7.20.2 第二档：能搬但要重新标定（定量关系）

① **w 的具体数值**。本章 $w = 20$ 就崩；SD 的默认 `guidance_scale` 是 7.5，很多模型用到 12 也不崩。**差别来自三处**：模型误差小得多、高维空间里“越界”的表现不同、以及**参数化可能差 1**（见坑 3）。**能搬的是“三段”这个形状**：欠引导 / 正常 / 过引导。**不能搬的是分界点的位置**。

② **越界率的定义**。本章用 $|x| > 1.5$，因为数据被缩放到 $[-1, 1]$。图像上没有这个坐标。**替代品是**：- 与真实数据集的最近邻距离 - 或者一个在真实图上训练的判别器分数

③ **多样性的度量**。本章用两维标准差。图像上要用**同一提示词、不同种子之间的 LPIPS 距离**，或者生成分布的协方差的有效秩。

7.20.3 第三档：不能搬（本章没有对应物）

① **潜空间**。真实模型在 VAE 的潜空间里做扩散。**“数据流形”变成了“潜变量的流形”**，它的几何和像素空间完全不同。本章的“飞出 $[-1.5, 1.5]$”在潜空间里没有直接对应。

② **文本条件**。本章的条件是一个离散标签，空类是**训练出来的嵌入**。文本模型的空类是**空字符串经过编码器的输出**——一个非零向量。**两者在 CFG 公式里位置相同，但含义不同**。

③ **分类器的失效曲线**。本章测出 $t \geq 600$ 后分类器准确率掉到 0.5。真实图像上这条曲线的位置取决于分类器架构和数据，**不能假设它一定在 0.6T 处**。

> ! 一条总的判断标准
>
> **问自己：这个结论里出现了“这个模型”吗？**
>
> - **没有** → 数学或结构，可以搬
> - **有，但只是“模型要够好”** → 半可搬，取决于你的模型
> - **有，而且是这个模型的具体行为** → 不可搬，只能当参考
>
> **本章的推导、公式、丢弃要求属于第一类；$w = 20$ 的崩溃点、有效维度曲线属于第三类。**

7.20.4 一个真实的对照：SD 的 `guidance_scale`

Stable Diffusion 的 `guidance_scale` 默认 7.5。它对应本章的哪个 w？

取决于实现。`diffusers` 里写的是

```
noise_pred = noise_uncond + guidance_scale * (noise_cond - noise_uncond)
```

这正是本章的公式，所以 `guidance_scale` 就是 w。

但有些实现写成

```
noise_pred = (1 + g) * noise_cond - g * noise_uncond
```

这时 $g = w - 1$。默认值 7.5 在两种写法下差 1，**表现为“同样的数字，效果差一档”**。

这不是理论问题，是抄代码时真实会遇到的。

7.21 动手练习

五个练习，从“复现”到“设计一个新引导方法”。

7.21.1 练习一：复现，并改一个变量

跑 `run_ch7_case.py`，确认你能得到本章的三组数：

实验	本章的值
CFG $w = 2$	准确率 0.998，越界 0
负引导 $w = -1$	准确率 0.133 / 0.049
无丢弃 $w = 5$	越界 0.986 / 1.000

然后只改一个变量：把噪声水平 `noise` 从 0.08 改成 0.16（两类重叠更多），重跑 w 扫描。
预测一下：同样的 w 下，准确率会更高还是更低？**过引导的崩溃点会提前还是推后？**
提示：重叠更多 → 隐式分类器的边界更模糊 → 需要更大的 w 才能分开；但更大的 w 也更快把样本推出去。**两个效应方向相反，你的预测应该说明哪个占主导**。

7.21.2 练习二：找出你的模型的“崩溃 w”

设计：w 从 1 开始，每次乘 1.5，直到越界率超过 10%。记录这个 w。
然后换三个训练步数（1000 / 3000 / 5000），**每个都测一遍**。
画一张图：横轴训练步数，纵轴崩溃 w。
预期的思考方向：训练越充分，模型误差越小，应该能承受更大的 w。**如果你的图不是单调的，先检查测量方式**——可能是噪声太大，需要每个点跑多个种子。
这个练习的价值：你会得到“崩溃 w 随模型质量变化”的曲线，**而这条曲线决定了你在产品里能设多大的默认值**。

7.21.3 练习三：验证“有效维度”假设

本章在 $w = 40$ 时测到 PC1 占比 0.996、有效维度 1.009。
请在没有 `run_ch7_extra.py` 的情况下复现这两个数。
做法：1. 用 `run_ch7_case.py` 训练出的模型 2. 采 1000 个 $w = 40$ 的样本 3. 自己算协方差矩阵的特征值 4. 算 $(\sum \lambda)^2 / \sum \lambda^2$
你会遇到的麻烦：样本可能包含极端值，协方差被一两个点主导。**试试去掉最大的 1% 再算一次**，看结论是否稳健。
这个练习的重点不是数字，是“稳健性”：一个由极端值驱动的统计量，不能用来支持“分布塌缩”这种结构性结论。

7.21.4 练习四：实现“按步数调 w”

本章发现：步数越少，能用的 w 越窄。
任务：设计一个调度 $w(t)$ 或 $w(\text{step})$，让少步采样也能用较强的引导。
一个可试的方案：在前几步用较小的 w（避免一步跨太远），在后面的步用较大的 w（锐化细节）。
要求：- 在 10 步和 50 步下各测一次 - 和固定 w 的基线比较 - 报告准确率、多样性、越界率
如果有效，你就得到了一个“免费的”改进—— **它不需要重训模型，只改采样循环**。

7.21.5 练习五：写一个引导诊断器

把本章的诊断动作做成代码：

```
class GuidanceDiagnoser:
    def check_null_class(self, model): ...
    def check_w1_identity(self, model, sch, y): ...
    def sweep(self, model, sch, y, ws): ...
    def pca_report(self, samples): ...
    def verdict(self): ...
```

要求：- `check_w1_identity` 必须逐位比较，不是“差不多” - `sweep` 必须同时返回三个指标 - `verdict` 要能说出“当前 w 处在哪一段”，并给出理由

做完这些练习之后

五个练习大约三小时。 它们会给你：

- **练习一、二**：一个属于你自己模型的“崩溃 w”曲线
- **练习三**：对“统计量稳健性”的体感
- **练习四**：一个可能真的有用的采样改进
- **练习五**：一套能直接用在下一个项目上的诊断工具

如果只做一个，做练习二。 它直接回答“我的模型能承受多强的引导”，而这个数是你设默认参数的依据。

7.22 五个常见误解

7.22.1 误解一：“CFG 让样本更接近条件”

错。它让样本比条件更远。

本章的手算例子：真实数据在 -1，$w=1$ 时等效均值是 -1.00，$w=2$ 时是 -2.20。

CFG 不是“往条件上拉”，是“沿着条件方向外推”。 $w=1$ 恰好停在数据上，$w>1$ 就越过了。

为什么这个误解流行：因为 w 小时（1.5–2）外推的幅度小，看起来确实“更像”。**但“更像”和“更远”在数学上是同一个操作。**

7.22.2 误解二：“w 越大越听话”

在 $w \le 3$ 时大致成立，之后就不是了。

准确率在 $w=3$ 就到 1.000，再往上它一点不变；**变的是分布形状**——内月多样性从 0.335 掉到 0.110，$w=20$ 时 40% 的样本越界，$w=40$ 时有效维度 1.009。

“听话”和“合理”是两件事。 一个飞到 $(0,3)$ 的点，在“是不是外月”这个问题上完全听话，在“是不是一个合理的外月”上完全不。

7.22.3 误解三：“标签丢弃是一种正则化”

它是 CFG 的语义前提。

正则化的作用是“防止过拟合”，去掉它模型会变差但仍能工作。**去掉标签丢弃，CFG 直接不工作**——$w=5$ 时 98.6%/100% 的样本越界。

判别方法：如果它真是正则化，去掉之后 $w=1$ 的性能应该下降。**实测 $w=1$ 是 0.960 vs 0.958——没有差别。** 它只影响 $w>1$。

7.22.4 误解四：“negative prompt 是反向的 prompt”

它改的是“无条件”分支，不是正向分支。

正向 prompt 进入 ε_c，negative prompt 进入 $\varepsilon_\varnothing$。**两者在公式里的位置不同，作用也不对称。**

证据：$w=+2$ 时越界率是 0，$w=-2$ 时是 31%/65%。**如果它真是“反向的 prompt”，两者应该一样危险。**

7.22.5 误解五：“CFG 是免费的”

它要多算一次前向。

合并 batch 之后，实测单步成本从 0.55 ms（$w=1$）涨到 0.64 ms（$w=2$），**约 16%**；完整采样从 2.4 秒涨到 3.6 秒，**50%**。

为什么两个数不一样？ 因为单步基准里 $w=1$ 也走了 `cfg_eps`（没有短路），而完整采样里 $w=1$ 走了短路。**真实的差距是 50% 左右。**

在真实模型上，这个 50% 往往就是“能不能实时”的分界线。

五个误解的共同点

它们都把“效果”和“代价”分开了。

- 一：把“方向”当成“目标”
- 二：把“准确率”当成“质量”
- 三：把“语义前提”当成“训练技巧”
- 四：把“两个分支”当成“一个旋钮的两端”
- 五：把“一次前向”当成“没有成本”

本章的每一节，都在把一对被混在一起的东西拆开。

7.22.6 练习六：$w = 0$ 的那次检查

这是本章最重要的一次检查，但很多人都跳过。

任务：1. 用你的条件模型，$w = 0$、固定提示词、换 5 个种子，各出 4 张 2. 判断这 20 张里有多少“明显跟提示词相关” 3. 如果超过一半相关，说明你的空条件有问题

然后做一件更狠的事：把空类编号故意改成 `n_class - 1`，重跑。

预期：错误版本在 $w = 0$ 时，输出会明显偏向最后一个真实类。

这个练习的价值：你会亲眼看到“$w = 1$ 检查通过、$w = 0$ 检查失败”—— 而这个组合正好对应“空类编号错”这一个 bug。

7.22.7 练习七：两种参数化的换算

任务：在同一个模型上实现两种写法

```
eps_a = eps_u + w * (eps_c - eps_u)
eps_b = (1 + g) * eps_c - g * eps_u
```

要求：1. 找出使 `eps_a == eps_b` 的 (w, g) 关系 2. 验证 `w=1` 与 `g=0` 都等于纯条件采样 3. 用同一个数值（比如 7）分别代入，看输出差多少

答案：$w = 1 + g$。同一个数 7 在两种写法下差一档。

这个练习看起来是算术，其实是“读论文的方法”：看到任何引导参数，先找它的定义式。

7.22.8 练习八：区分“过引导”和“采样器不稳”

设计题：给你一个在 $w = 3$ 时出现伪影的模型，设计一个实验，在**不重训**的前提下判断是哪一个原因。

要求：- 实验不超过 30 分钟 - 不依赖任何本章没有的数据 - 给出明确的判据（不是“看起来像”）

提示：本章的失败五给了答案的一半（$w = 1$ 的对照）。另一半是**换采样器**：同一 w、DDIM 换 DDPM，看伪影是否消失。

如果两个对照都指向“采样器”，那结论就确定了；如果 $w = 1$ 正常、换采样器也正常、只有 $w = 3$ 出问题，那才是过引导。

7.22.9 练习的参考方向

练习一：把噪声从 0.08 改成 0.16。

预期：$w = 1$ 的准确率会掉（两类重叠更多，条件本身更难分开），**崩溃点会提前**。

为什么崩溃点提前：重叠更多 → 隐式分类器更弱 → 需要更大的 w 才能分开；但更大的 w 也更快把样本推出去。**两个效应同向，所以最优区间变窄。**

如果你的结果是“准确率掉了但崩溃点没变”，检查一下是不是样本数太少—— 600 个点在重叠严重的分布上，方差不小。

练习二：崩溃 w 随训练步数的曲线。

预期：单调上升，但**边际递减**。

理由和第 6 章那个“步数下界”是同一个：模型误差进的是 $L \cdot T$ 那一项，**误差减半带来的收益是次线性的。**

如果你测出非单调，最可能的原因是训练步数太少时模型输出接近零，“不崩”其实是“什么都没生成”。**检查方法**：看多样性。 多样性也接近零，说明是退化，不是稳定。

练习三：PCA 的稳健性。

实测结果（去掉半径最大的 1% 后重算）：

w	全样本 PC1 / 有效维度	去掉最大 1%
10	0.911 / 1.194	0.932 / 1.144
20	0.976 / 1.050	0.973 / 1.056
40	0.996 / 1.009	0.995 / 1.009

结论不变。 去掉极端点之后，$w = 40$ 的有效维度仍然是 1.009。

为什么：塌缩是结构性的——绝大多数点都挤在一条线上，不是被一两个极端点拉出来的。**如果去掉 1% 之后结论反转，那原来的结论就是假的**。

这个练习真正的收获：学会问“这个统计量是被少数点驱动的吗”。 本章的越界率、标准差、有效维度都该做一次这个检查。

练习四：$w(t)$ 调度。

一个合理的起点：前 30% 的步用 $w - 1$，后 70% 用 w。

理由：前几步决定全局结构（位置、大类），这里推太狠容易一步跨出流形； 后面的步在锐化细节，需要更强的条件信号。

预期结果：在 10 步下，$w(t)$ 调度应该能让“等效 w”比固定 w 高半档而不崩。 **如果没效果**，说明崩溃主要不是发生在早期——那就该检查是哪一步开始越界的。

记录轨迹：每一步记 $|x|$ 的均值。**你会看到它在哪一步抬头**， 那个位置就是该减小 w 的地方。

练习五：诊断器的判定逻辑。

`verdict()` 应该按这个顺序判断：

1. **空类编号对不对** → 不对直接报错，后面的都不用测
2. **$w = 1$ 是否逐位一致** → 不一致，是参数化或代码路径问题
3. **$w = 1$ 是否稳定** → 不稳定，是采样器问题，不是引导
4. **越界率是否随 w 单调** → 不单调，CFG 没接上
5. **准确率是否饱和** → 饱和后还调大 w，给出警告

这五步的顺序不能换。 前一步没通过，后一步的测量没有意义—— 和第 6 章“先做便宜的致命检查”是同一条纪律。

7.23 训一个条件模型：在开 CFG 之前

引导只能放大模型已经学到的东西。 这一节列出“开 CFG 之前该确认的事”， 顺序按“出问题的概率”排。

7.23.1 ① 条件确实进了模型

检查：固定种子、只改标签，采样两批。**两批应该明显不同**。

如果几乎一样，说明条件没有生效——常见原因是嵌入被加到了一个 不参与前向的分支上，或者标签在数据管道里被覆盖了。

本章的对应数：$w = 0$ 时准确率 0.525、$w = 1$ 时 0.965。**这个跳变就是“条件确实进了模型”的证据**。

7.23.2 ② 每个类的样本量够

本章两类各 2000 个。如果某一类只有几十个样本， 模型对 $p(x \mid c)$ 的估计会很差，**而隐式分类器也会同样差**。

经验阈值：每类至少几百个。**少于这个数，先做数据增强或换条件粒度**（比如把细类合并成粗类）。

7.23.3 ③ 空类的训练信号够

丢弃率 0.1 意味着每类大约 10% 的样本在训空类。 **如果总样本少，空类嵌入会欠训练**——表现和“没做丢弃”类似。

检查：训练后统计空类被用了多少次。**少于几百次就要警惕。**

7.23.4 ④ 条件与无条件的 loss 都要看

只看总 loss 会漏掉“有条件的那部分变差、无条件的那部分变好”这种情况。
做法：训练时分开记两个 loss（按是否被丢弃分组）。**两者应该都在下降；如果无条件的一直在降、有条件的不降，说明丢弃率太高。**

7.23.5 ⑤ 先测条件模型，再开引导

顺序不能反。

- 条件模型本身不准（$w = 1$ 准确率低）→ 先修模型
- 条件模型准、$w > 1$ 崩 → 才是引导的问题

本章的数据给了一个参照：$w = 1$ 是 0.958–0.965，$w = 2$ 是 0.997–0.998。**如果 $w = 1$ 就只有 0.7，别急着调 w。**

一张“开 CFG 之前”的清单

#	检查	通过标准
1	改标签会改输出	两批样本明显不同
2	每类样本量	≥ 几百
3	空类被训练的次数	≥ 几百
4	有条件/无条件 loss 都在降	都降
5	$w = 1$ 的准确率	接近你的模型上限

五条全过，再谈 w。任何一条不过，调 w 都是在错误的地方使劲。

7.24 上线评审：五个必须回答的问题

这一节是给“要把 CFG 放进产品”的人看的。五个问题，答不上来就别上线。
① 你的代码用的是哪种参数化?
w 还是 $w' = w - 1$? **写进文档，并在代码里注释。**这一条答错的代价是“默认值整体偏一档”，而它极难在测试里发现。
② 默认值是怎么定出来的?
不能是“论文里写的”。要有：- 多少条提示词 - 几个 w 档位 - 用什么指标（人评 + 多样性） - 谁评的
本章的二维数据给出的位置是 $w \approx 2$；真实模型通常在 5–8。**这个差异本身要能解释**，否则说明你没搞清自己的模型。
③ 你怎么发现“过引导”?
必须有至少一个在线指标。候选：- 同提示词的输出相似度 - 解码图的饱和度 / 高频能量 - 用户重试率
没有这个指标，过引导只会在用户投诉时才被发现。
④ negative prompt 的边界在哪?

- 强度上限是多少
- 允许多少个词
- 有没有黑名单（比如会放大偏见的词）

本章的实测：$w = -2$ 时 31%/65% 越界。**负向比正向脆得多，上限要更严。**
⑤ 换模型/换编码器之后，谁负责重新验证?
这一条最容易被忽略。**空条件的定义依赖编码器**—— 换 tokenizer、换文本编码器、甚至升级模型版本，空字符串的嵌入都会变，**CFG 的行为也会跟着变。**

流程里要有一步：“模型更新 → 重跑 $w = 1$ 与 $w = 0$ 的逐位检查 → 重跑默认值的定标”。

一张上线前的最小检查表

#	检查	谁负责	频率
1	$w = 1$ 逐位一致	开发	每次改采样代码
2	$w = 0$ 逐位一致	开发	每次改采样代码
3	默认值定标	产品 + 设计	每次换模型
4	过引导在线指标	数据	持续
5	负向强度上限	产品	每次改词表

7.25 端到端：从训练到上线的一条链路

前面每一节解决一个问题。**这一节把它们串成一条时间线**，标出每一步的产出物和“不通过就回退到哪里”。

7.25.1 阶段一：数据与标签

做什么	产出	不通过的回退
确认类数 C 与每类样本量	一张类别分布表	合并细类 / 补数据
确认空条件怎么表示	一行代码 + 一句注释	—
留出验证集	每类 $\geqslant$ 200 条	重新划分

空条件那一条必须在训练前定下来。 训练完再改，等于重训。

7.25.2 阶段二：训练

```
for step in range(max_steps):
    x0, c = batch
    c = torch.where(torch.rand(len(c)) < drop, null_class, c)   # ← 这一行
    t = torch.randint(0, T, (len(c),))
    xt, eps = q_sample(x0, t)
    loss = F.mse_loss(model(xt, t, c), eps)
```

记录三样东西：

1. **总 loss**
2. **有条件/无条件分开的 loss**（按是否被丢弃分组）
3. **每 t 段的 loss**（第 5 章的方法）

不通过的信号：有条件的那份 loss 不降，或者空类被用了不到几百次。 **回退到阶段一**（可能是丢弃率或数据量的问题）。

7.25.3 训练日志该记什么

六个失败里，有三个在总 loss 上看不出来。 所以要分开记。

每个 step：

```
{"step": s, "loss": total, "loss_cond": ..., "loss_uncond": ...,
 "null_frac": mask.float().mean().item()}
```

`loss_cond` 与 `loss_uncond` 是把同一个 batch 按 `mask` 拆成两组分别算的。 **它们应该都在降**。

每 100 步：五个 t 段的分段 loss（第 5 章的方法）。
每个 run：配置哈希、种子、数据哈希、丢弃率。

日志项	能发现什么
`loss_uncond` 不降	空类欠训练（丢弃率太低或步数太少）
`loss_cond` 不降	条件信息没进模型
分段 loss 在某段高	那个噪声水平学不好（第 5、6 章的问题）
`null_frac` 偏离设定值	丢弃代码写错了

为什么不记就没法排查

本章的失败一（无标签丢弃）在总 loss 上完全看不出来—— drop=0 的模型 loss 甚至更低（没有空类那 10% 的样本拖累）。
只有把 `loss_uncond` 单独记下来，你才会发现“这一项从来没被训练过”。

7.25.4 阶段三：验证条件模型（还没开 CFG）

检查	通过标准	本章的数
改标签会改输出	两批明显不同	—
$w = 1$ 准确率	接近模型上限	0.965
$w = 0$ 准确率	$\approx 1/C$	0.525
多样性	与数据同量级	0.310（数据 0.573/0.339）

不通过就回退到阶段二。 不要在这一步调 w。

7.25.5 阶段四：定标引导

只有在阶段三全过之后，才做 w 扫描。

做什么	产出
w 扫 5 档 $\times$ 2 类 $\times$ 600 样本	三条曲线（准确率/多样性/越界）
找出饱和点与崩溃点	一个安全区间
在安全区间内选默认值	一个数 + 定标理由

本章的安全区间是 $[1, 10]$，默认取 2。 真实模型通常更高，但**区间的形状一样**。

7.25.6 阶段五：上线与复验

触发条件	做什么
每次改采样代码	$w = 0$ / $w = 1$ 逐位检查
换模型 / 换编码器 / 换 tokenizer	从阶段三重来
线上多样性指标下降	检查 w 与默认值
用户重试率上升	同上

! 这条链路上最容易跳过的两步

① **阶段三的 $w = 0$ 检查。** 它看起来多余（“我还没开引导呢”），**但它验证的是空条件**—— 而空条件是 CFG 唯一的语义前提。

② **阶段五的复验。** 换一个文本编码器听起来只是“换个组件”，**但空字符串的嵌入变了，整个 CFG 的基准就变了。**

本章的实验四（零向量 vs NULL token）说明这不是理论担心：换个表示，越界率从 0 变成 100%。

7.26 术语对照

同一个概念在论文、代码、中文教材里往往有三个名字。

中文	English	代码里常见名
无分类器引导	classifier-free guidance (CFG)	`guidance_scale`、`cfg`
引导强度	guidance scale / guidance weight	`guidance_scale`、`cfg_scale`
空条件	null / unconditional condition	`uncond_embeds`、`negative_prompt_embeds`
标签丢弃	conditioning dropout	`drop_prob`、`uncond_prob`
隐式分类器	implicit classifier	—（没有独立变量）
锐化 / 温度	tempering / sharpening	—
负引导	negative guidance	`negative_prompt`
投影式引导	projected guidance	—（APG 里叫 `eta`）
重标定	guidance rescale	`rescale`、`guidance_rescale`
动态阈值	dynamic thresholding	`dynamic_threshold`

⚠ 两处最容易对错的词

① **`guidance_scale`**

在 `diffusers` 里它对应本章的 w；但有些实现里它对应 $w - 1$（见坑 3）。**看到这个词先找公式。**

② **`negative_prompt`**

它不是“反向的提示词”，而是**替换掉空条件的内容。**在代码里它通常被编码成 `negative_prompt_embeds`，送进无条件那一路。

看到它出现在“条件”那一路，就是接错了。

7.27 排查流程：图像模型上的五步

二维上能做的检查，图像上大部分也能做，只是“看什么”变了。按顺序走，前一步不过就不要进下一步。

7.27.1 第一步：CFG 到底有没有生效

做法：固定种子和提示词，w 取 1 和 3，各出两张。

通过标准：两张图明显不同。

不通过说明：`w` 没传进采样循环，或者被调度器覆盖了。**先查代码，不要查模型。**

7.27.2 第二步：$w = 1$ 是否稳定

做法：$w = 1$ 出 10 张，看有没有崩坏、伪影、纯噪声。
通过标准：10 张都正常。
不通过说明：采样器或模型本身有问题。**用第 6 章的方法先修采样器**（换 DDPM、检查步数、检查调度）。

7.27.3 第三步：$w = 0$ 是否无条件

做法：$w = 0$、同一个提示词、不同种子，出 10 张。
通过标准：输出和提示词**几乎没有关系**。
不通过说明：空条件不对——可能是空类编号错、空串没送到编码器、或者代码里把 $w = 0$ 走了另一条分支。

> 为什么第三步这么重要
>
> **它同时检查了“空条件”和“引导方向”两件事。**
> 如果 $w = 0$ 时画面仍然明显跟提示词走，说明“无条件”分支其实带着条件—— **那么 $w = 3$ 时你减掉的就不是“无条件”，而是一个带了条件的基准。**整条引导都偏了，而且偏得很隐蔽。

7.27.4 第四步：w 扫描的形状

做法：w 取 1 / 2 / 3 / 5 / 8 / 12，各出 4 张。
通过标准：符合度先升后平，多样性单调降，某个点之后开始崩。
不通过说明：

- **一直没变化** → 回第一步
- **一直崩** → 回第二步
- **单调变好、不饱和** → 你的模型可能还没训好，或者提示词太简单

7.27.5 第五步：换模型/换编码器之后重复

只要模型、tokenizer、文本编码器、调度器版本有任何一个变了，就从第一步重来。
这不是谨慎，是必要——本章实验四已经证明，空条件的表示一变，CFG 的行为就完全不同（零向量 vs 训练过的 NULL token）。

7.28 三个指标能抓住什么

本章反复要求“至少三个指标”。**这一节说清楚：哪个指标抓哪个失败。**

失败	准确率	多样性	越界率
无标签丢弃	✗（$w = 2$ 时仍 0.995）	✗	√
参数化差 1	✗	✗	✗（需要逐位检查）
空类编号错	√（$w = 0$ 时暴露）	✗	✗
过引导	✗（1.000）	√（0.110）	√（37.5%）
采样器不稳	✗	✗	√
负引导过强	√	✗	√
条件没接上	√（$w = 0$ 时）	✗	✗

读这张表的方法：每一行至少有一个 √ 吗？

· **有** → 三个指标够用
· **没有**（第二行）→ 需要第四个检查：**逐位相等**

> ! 为什么“参数化差 1”三个指标都抓不到
>
> 因为**它不是错误，是约定**。
> $w = 7$ 和 $w' = 7$ 都是合法的配置，只是强度不同。**没有任何统计量能告诉你“你本该用另一个参数化”**——只有“$w = 1$ 应该逐位等于条件采样”这条**结构性检查**能发现它。
> **这就是为什么验收清单里既有统计检查，也有恒等检查**。前者看“结果好不好”，后者看“定义对不对”。

一个实操建议：把这三个指标 + 两个恒等检查做成一个函数， 每次改采样代码就跑一遍。

```
def verify_cfg(model, sch, y, w=2.0):
    s1 = sample(model, sch, y, w=1.0, seed=0)
    s1_ref = sample_conditional(model, sch, y, seed=0)
    assert torch.equal(s1, s1_ref), "w=1 不等于条件采样：参数化错了"
    s0 = sample(model, sch, y, w=0.0, seed=0)
    assert judge(s0).argmax(1).float().mean() < 0.7, "w=0 不是无条件：空类错了"
    s2 = sample(model, sch, y, w=w, seed=0)
    assert out_rate(s2) < 0.05, "越界率过高：引导过强或采样器不稳"
    print("三项通过：准确率", acc(s2), "多样性", div(s2))
```

四个断言，覆盖了本章六个失败里的五个（剩下那个是“评估本身错了”）。

7.29 代码与数据索引

用途	文件
条件模型、CFG、分类器引导	`notebooks/conditional_diffusion.py`
本章主实验（五组）	`notebooks/run_ch7_case.py`
补充实验（四组）	`notebooks/run_ch7_more.py`
交互实验数据	`notebooks/export_ch7_exp.py`
主实验数据	`data/ch7_case.json`
补充实验数据	`data/ch7_extra.json`、`data/ch7_more.json`
交互实验页面	`experiments/exp-7-1.html`
图 7-1	`qa/gen_fig_ch7.py` → `assets/fig/7-1-cfg-sweep.svg`
图 7-0	`qa/gen_fig_ch7_flow.py` → `assets/fig/7-0-cfg-flow.svg`

每一个数字都能在这张表里找到出处。

7.30 符号表

符号	含义	取值范围	容易混
c	条件（类别/文本）	—	—
$\varnothing$	空条件	—	零向量（**不是**）
ε_c	条件预测	$\approx \mathcal{N}(0, I)$	—
ε_u	无条件预测	同上	$\varepsilon_\varnothing$ 同义

符号	含义	取值范围	容易混
w	引导强度	$[0, \infty)$	$w' = w - 1$
s	分类器引导强度	$[0, \infty)$	w（不同方法）
p_w	CFG 实际采样的分布	$\propto p(x \mid c)^{w(p(x))^{1-w}}$	$p(x \mid c)$
drop	标签丢弃率	0.05–0.1	正则化系数（**不是**）
C	类别数	—	—
γ, β	AdaGN 的调制参数	—	—

读这张表的方法：看到 w 先确认参数化；看到“空条件”先确认它是嵌入还是零向量； 看到 s 先确认是分类器引导还是 CFG。

7.31 CFG 的三个变体（本章未复现）

本章讲的是 2022 年的原始 CFG。**在那之后，工业界和学术界改了三处**。下面三个方向值得知道，**但本章没有在二维模型上复现它们**—— 所以只描述做法和动机，不给数字。

7.31.1 ① 引导重标定（guidance rescale）

问题：w 大时，$\hat{x}_0$ 的**尺度**会偏大——样本整体“过曝”。
做法：算出引导后的 $\hat{x}_0$，再把它缩放到和条件预测相同的标准差：

```
x0_cfg = (x - (1 - ab).sqrt() * eps_cfg) / ab.sqrt()
x0_c   = (x - (1 - ab).sqrt() * eps_c)   / ab.sqrt()
x0_cfg = x0_cfg * (x0_c.std() / x0_cfg.std())     # 逐样本或逐通道
```

动机：引导改变的不该是“画面多亮”，只该是“内容多像条件”。缩放把前者去掉，只留后者。
它对应本章的哪个现象：$w = 20$ 时平均最大半径从 0.54 涨到 1.25—— **这就是“尺度偏大”的二维版本**。

7.31.2 ② 动态阈值（dynamic thresholding）

问题：w 很大时，$\hat{x}_0$ 会出现极端的饱和值（纯黑或纯白）。
做法：不固定 clamp 到 $[-3, 3]$，而是按**分位数**截断：

```
s = torch.quantile(x0.abs().flatten(), 0.99)   # 当前批次的 99% 分位
x0 = x0.clamp(-s, s) / s                        # 再归一化
```

动机：固定阈值在不同 w 下不合适——w 大时分布整体变宽， **固定的 ± 3 会截掉越来越多本来有效的信息**。
注意：分位数是**按批次**算的，所以它依赖 batch 的组成。**采样时 batch 太小，这个统计量会不稳**——这是它的工程代价。

7.31.3 ③ 投影式引导（APG）

问题：标准的 CFG 把 $(\varepsilon_c - \varepsilon_u)$ 整个方向放大，**其中包含“只是把画面变亮/变暗”的那一部分**，那部分不携带条件信息。
做法：把这个差值**投影到与 ε_c 垂直的子空间**，再放大：

$$\Delta_\perp = \Delta - \frac{\langle \Delta, \varepsilon_c \rangle}{\| \varepsilon_c \|^2} \varepsilon_c \tag{7.44}$$

动机：只放大“改变内容”的方向，不放大“改变尺度”的方向。
本章的对应现象：$w = 20$ 时分布塌成一条线（有效维度 1.009），**那条线很可能就是“尺度方向”**——投影式引导要去的正是这个。

⚠ 这三个变体的共同点

它们都在修“w 大时的副作用”，没有一个是“让 w 可以无限大”。

- 重标定修尺度
- 动态阈值修饱和
- 投影修方向

但 p_w 仍然是 $p(x \mid c)^{w(p(x))^{1-w}}$。分布本身变了，这些技巧只是让“变了的分布”看起来更顺眼一点。

换句话说：它们是化妆，不是治病。真正决定上限的，还是模型对 $p(x \mid c)$ 的估计有多准。

7.32 十个常见问题

问一：CFG 和 prompt weighting 是什么关系?

不是一回事，但作用在同一处。

prompt weighting 改的是**文本嵌入本身**（把某个词的向量乘一个系数），CFG 改的是**模型输出的差**。前者在编码器一侧，后者在去噪器输出一侧。

但两者都能“加强某个概念”。区别是：weighting 可以只加强一个词，CFG 只能加强整段提示词——**因为它的方向是“整段文本 vs 空串”。**

问二：为什么 Stable Diffusion 默认 7.5?

这是经验值，不是理论值。

它对应本章的 $w = 7.5$（如果实现用 `eps_u + w*(eps_c-eps_u)`）。本章的二维模型在 $w = 10$ 就开始崩，而 SD 能用到 12—— **差别在模型质量，不在公式。**

7.5 这个数应该被当成“某个模型在某个训练流程下的经验值”，不是普适常数。

问三：$w = 0$ 和“不传提示词”一样吗?

不一样。

- $w = 0$：$\hat{\varepsilon} = \varepsilon_\varnothing$，**只用空条件**
- 不传提示词：取决于实现，可能是空串、可能是默认提示词

前者在数学上是确定的，后者依赖工程细节。本章的 $w = 0$ 检查验证的是前者。

问四：negative prompt 和 $w < 0$ 是一回事吗?

不是，但相关。

- $w < 0$：把“条件方向”取负，推离 c
- negative prompt：改变 $\varepsilon_\varnothing$ 的内容

两者都可以“推离某个概念”，但走的是不同的项。本章的 $w < 0$ 实验是后者的一个简化模型。

问五：CFG 能用在非扩散模型上吗?

能，而且已经有人这么做。

CFG 的核心是“有条件输出减去无条件输出”，**任何有这两个输出的生成模型都能用。**语言模型上也有类似的做法（对比解码）。

但“减去”是否有意义，取决于那个差是否对应一个 score。在扩散模型上它有严格的概率解释；在别的模型上，它更多是一个经验技巧。

问六：为什么有些模型 $w > 10$ 也不崩?

三个原因，按重要性排：

1. **模型误差更小**——$p(x \mid c)$ 估得准，外推的幅度就小
2. **高维空间**——“飞出流形”在 1024 维里表现不同，像素上看不出来
3. **管线里有补偿**——rescale、dynamic thresholding、或者调度器的裁剪

所以“别人能用到 12”不能直接搬到你的模型上。

问七：蒸馏模型为什么不能调 w?

因为它可能已经把引导烘进权重了。

蒸馏时学生学的是教师的最终输出，**那个输出已经包含了教师用的 w。**再外挂 CFG 等于引导两次。

检测方法：w 扫描看是否平滑变化（详见场景 C）。

问八：CFG 的梯度能反传到文本嵌入吗?
能，而且这正是 prompt tuning / textual inversion 的基础。
$\hat{\varepsilon}$ 对文本嵌入是可导的（通过编码器和交叉注意力）。**优化文本嵌入让生成更符合某个目标，就是沿着这条链反传。**
但注意：反传穿过的是“两个模型输出的差”，**梯度会比单路更大**，学习率要相应调小。
问九：训练时能用 CFG 吗?
不能直接用在训练目标里——CFG 是一个采样时的操作，训练时你只有 $\varepsilon_\theta(x_t, t, c)$，没有“空条件输出”这一项。
但可以间接：有些工作把“引导后的样本”作为蒸馏目标（见场景 C 的反面）。
问十：有没有“自适应 w”的方法?
有，而且方向不止一个：

- **按 t 调度**：前几步小、后面大（本章练习四）
- **按样本自适应**：根据当前预测的不确定性调整
- **投影式**：只放大与条件垂直的分量（APG）

本章只测了固定 w。这些方法的效果依赖模型，**没有通用结论。**
问十一：同时要满足两个条件怎么办?
CFG 只处理“一个方向”。两个条件有两种做法：

- **组合条件**：把两个条件拼成一个（文本上就是拼接提示词）
- **分别引导**：分别算 $(\varepsilon_{c_1} - \varepsilon_\varnothing)$ 和 $(\varepsilon_{c_2} - \varepsilon_\varnothing)$，再相加

后者更灵活，但要注意归一化——两个方向相加后范数会变大，等效于用了更大的 w。**先各取一半，再按总强度调。**
问十二：训练时有 w 吗?
没有。训练只优化 $\varepsilon_\theta(x_t, t, c)$，不存在“引导强度”这个参数。
但有一处例外：如果你在做蒸馏，**教师的 w 会进入学生的训练目标。**这时学生的输出已经“带引导”，再外挂 CFG 就是引导两次（见场景 C）。

> 十个问题里，哪三个最常被问错
>
> **问二、问六、问七。**
> 它们都涉及“把别人的参数搬到自己模型上”。**记住一句话就够了：w 是模型的属性，不是任务的属性。**

7.33 和其他条件控制方法的对比

CFG 不是唯一的“让模型听话”的办法。一张表把它们放在一起。

方法	改的是什么	需要什么	强度参数
CFG	模型输出的差	标签丢弃（训练时）	w
分类器引导	加上分类器梯度	噪声分类器	s
prompt weighting	文本嵌入本身	无	词级权重
negative prompt	空条件的内容	无	隐含在词表里
null-text inversion	优化出的空条件嵌入	一个优化过程	—
rescale / APG	引导后的后处理	无	沿用 w

它们不在同一个层面上：

- **CFG / CG** 改的是“条件信号有多强”
- **weighting / negative** 改的是“条件信号是什么”
- **null-text inversion** 改的是“无条件的基准是什么”
- **rescale / APG** 改的是“引导之后怎么收拾”

> **i** 为什么可以叠加
>
> **因为它们作用在不同的项上。**
> 一个典型组合：**prompt weighting（改 ε_c）+ negative prompt（改 $\varepsilon_\varnothing$）+ CFG（放大差）+ rescale（修尺度）**。
> **四者互不冲突**，因为它们分别动的是分子、基准、倍数和后处理。
> **但叠加会让调试变难**——出问题时你不知道是哪一层。**建议一次只加一个**，这和本章“一次只变一个量”的纪律一致。

选择顺序建议：

1. 先确认条件模型本身好不好（$w = 1$）
2. 再加 CFG（默认手段）
3. 需要“排除某物”再加 negative（强度 1/3）
4. 需要“强调某词”再加 weighting
5. 出现尺度问题时最后加 rescale/APG

不要反过来。先加一堆技巧再调 CFG，你会分不清是谁在起作用。

7.34 用这一章的方法读一篇新的引导论文

每个月都会有一篇“更好的引导”。用这五个问题快速定位它。

① 它的参数化是什么？

`w` 还是 `w'=w-1`？**先在论文里找公式，再看数字**。同一个 7，两种写法差一档。

② 它的“无条件”是什么？

- 学出来的空类嵌入？
- 空字符串的编码？
- 零向量？
- 还是一个专门训练的“负面”嵌入？

这决定了它的行为能和本章的哪一部分对照。如果是零向量，本章实验四已经说明它会崩——先怀疑它。

③ 它在几步上做的实验？

步数和可用 w 强耦合（本章：10 步时 $w = 5$ 只有 0.612）。**一个在 100 步上调好的 w，搬到 20 步可能完全不适用**。

④ 它用什么指标？

只有一个“符合度”指标吗？**如果是，它的“更好”很可能只是“更锐”**。看它有没有报告多样性或分布距离。

⑤ 它有没有报告副作用？

过饱和、多样性下降、高 w 崩坏——**一篇诚实的论文会写这些**。

> **i** 一个五分钟就能做的复现
>
> **拿它的默认参数，出三组图：**
>
> - 它的默认 w
> - 默认 w 减 1
> - 默认 w 加 4
>
> **如果“减 1”明显变差、“加 4”没有变好**——说明它的默认值踩在边缘上，**而且它报告的收益可能来自“更锐”而不是“更准”**。
> **这三组图不需要读完全文就能做**。

7.35 从二维走到图像：四个中间台阶

如果你要在一台机器上复现本章，不建议直接从文本图像模型开始。按照难度上四个台阶，每一级只引入一个新变量。

7.35.1 台阶一：二维（本章）

新东西：条件、CFG、三个指标。 **验证**：$w = 1$ 准确率 >0.9，$w = 2$ 越界 0，$w = 20$ 越界 >30%。
这一级的目标是把“机制”和“实现”分开。 二维没有实现难度，所以这里的失败一定是机制层面的。

7.35.2 台阶二：小图像 + 类别

换成：MNIST / Fashion-MNIST，10 类，28×28。
新东西：卷积或 ViT 主干、VAE 可选。
指标怎么换：

二维指标	小图像上的替代
标准差异常	类内 LPIPS（同一类样本两两距离）
越界率	像素裁剪率（生成值落在 [−1,1] 外的比例）
裁判准确率	在真实图上训的分类器

这一级的目标是：证明“三个指标在图像上也能用”。

7.35.3 台阶三：潜空间

换成：在 VAE 潜空间做扩散（比如 64×64 潜变量）。
新东西：VAE、scaling factor、潜空间的“越界”含义。
这一级最容易踩的坑：把像素空间的判据直接搬到潜空间。**潜变量的范数分布和像素完全不同**，要重新标定。

7.35.4 台阶四：文本条件

换成：文本编码器 + 交叉注意力。
新东西：空字符串的嵌入、token 级的条件。
这一级的目标是：验证本章的“文本条件的三处不同”（预告那一节）。

> 为什么建议按台阶走
>
> **因为每一级只引入一个新变量。**
> **直接从台阶四开始**：出了问题，你要同时怀疑实现、潜空间、编码器、CFG。**按台阶走**：在台阶一确认机制，台阶二确认指标，台阶三确认尺度，台阶四只剩“文本”这一个新东西。
> **这和本章“一次只变一个量”的纪律是同一条。**

7.36 延伸阅读

7.36.1 ① 分类器引导（2021）

Dhariwal & Nichol, Diffusion Models Beat GANs on Image Synthesis。arXiv:2105.05233。
读它能得到什么：本章“分类器引导”那一节的完整来源。这篇论文第一次让扩散模型的图像质量超过 GAN，**靠的就是引导**。
读的时候注意：

· 它的引导强度记作 s，作用在**分类器对数概率的梯度**上，和本章的 s 是同一个量

- 它需要**在噪声数据上训练分类器**，而且要为每个噪声水平调参—— 本章那张“$t \geq 600$ 之后准确率 0.5”的表，就是这件事的代价
- 它的实验在 ImageNet 上，**引导强度可以到几十**——和本章的二维模型完全不同，这再次说明“具体数值不能迁移”

7.36.2 ② CFG 的原始论文（2022）

Ho & Salimans, Classifier-Free Diffusion Guidance。arXiv:2207.12598。
读它能得到什么：本章的主线。**只有 4 页**，值得逐行读。
读的时候注意：

- 它的核心观察就一句：**条件 score 可以用“有条件 - 无条件”近似**， 于是不需要分类器
- **标签丢弃在论文里叫“conditioning dropout”**，被写成训练技巧； 本章用对照实验说明了它不是可选项
- 论文里也有“引导强度”的讨论，但**没有给出三段行为的定量描述**—— 本章的二维实验补上了这一块

7.36.3 ③ Negative prompt 与“反向引导”

Stable Diffusion 的 negative prompt 在工程上很常见，但学术文献里 系统的分析不多。可以看 **Ho & Salimans 论文的 §3** 关于“guiding away from a class”的讨论，以及后续一些关于 **negative prompting 会放大偏见** 的分析。
读的时候注意：negative prompt 不是“反向的 prompt”。 它改变的是空类那一路上放什么，**和正向不是对称的**。 本章的 $w < 0$ 实验是它的最简模型。

7.36.4 ④ 引导为什么会“过饱和”

关于 guidance 的理论分析，可以看 **Bradley & Nakkiran (2024)**、 以及关于 **“guidance 与 score 的关系”** 的若干工作。它们试图回答： 为什么 w 大了之后样本会离开数据流形?
本章给的是一个一维的实证版本：越界率随 w 上升，$w = 20$ 时 37.5%。

7.36.5 ⑤ 蒸馏模型里的引导

LCM、SDXL-Turbo、DMD 这些少步模型，在蒸馏时通常把引导“烘”进权重。 读它们的论文时找 “guidance scale” 这个词，**看它是训练时的固定值还是推理参数**。
这决定了你外挂 CFG 时会不会“引导两次”。（见选型一节场景 C）

7.36.6 ⑥ 引导的后处理与几何解释（2023–2024）

APG（Adaptive Projected Guidance） 与 **CFG++** 是近两年比较有代表性的两条路。
APG 的出发点是：$(\varepsilon_c - \varepsilon_u)$ 里混着“改变尺度”的分量，于是把它投影到与 ε_c 垂直的子空间再放大。 **它对应本章观察到的“过引导时分布塌到一维”**—— 那条线很可能就是尺度方向。
CFG++ 换了一个角度：把 CFG 看成“重新加噪”的一步， 在采样器内部调整注入噪声的强度。**它改的是采样过程，不是引导公式**。
读的时候注意：

- 两篇都在**修 w 大时的副作用**，没有一篇声称“w 可以无限大”
- 它们的实验都在真实图像模型上，**没有二维对照**—— 本章的方法（w 扫描 + 三个指标 + PCA）可以用来独立验证它们的说法
- **APG 的投影公式很简洁，但“投影到哪个方向”有多种选择**， 论文里的选择未必适合你的模型

7.36.7 五篇各自没说的事

读论文时，知道它“没说什么”，比知道它“说了什么”更能防止误用。
① 分类器引导（2021）没说：分类器在高噪声处会失效。

论文里分类器是专门为 ImageNet 设计的，**它的实验没有报告“分类器在各 t 上的准确率”**。本章补了这一块：$t \geq 600$ 之后准确率 0.500。**这不是论文的疏漏，是当时的关注点不同**——它关心 FID，不关心引导的机制。

② **CFG（2022）没说：标签丢弃是前提，不是技巧。**

论文把它写成“conditioning dropout”，和 dropout 放在一起介绍。**读者很容易把它理解成“防止过拟合的可选项”**。本章的对照实验（drop=0 时 $w = 5$ 越界 98.6%）说明它不是。

论文也没给出 w 的三段行为——**它只报告了“w 大更好”的区间。**

③ **negative prompt：没有经典论文。**

它是工程实践先流行起来的。**所以没有人为“强度该取多少”给过建议**，也没有人报告“负向比正向脆”。本章的 $w < 0$ 实验是把这个实践问题形式化的一个尝试。

④ **Imagen 的 rescale（2022）没说：什么时候不该用。**

rescale 修的是“尺度偏大”，**但如果你的过引导问题不是尺度，而是方向，rescale 会把有用的信号一起缩掉。**

本章的对应观察：$w = 20$ 时既有尺度问题（半径 1.25）也有塌缩（有效维度 1.044）。**rescale 只修前者。**

⑤ **蒸馏论文没说：外挂 CFG 会双重引导。**

它们通常只报告“学生模型在几步下的质量”，**没有讨论“用户如果再加 guidance_scale 会怎样”**。

本章的场景 C 给了检测方法：w 扫描看是否平滑变化。

> **i** 这张“没说什么”的清单怎么用
>
> **读一篇论文时，问自己：它解决的问题和我遇到的问题，是不是同一个？**
>
> · 它关心 FID，我关心“用户能不能调” → 它的结论可能不适用
> · 它在 100 步上测，我用 15 步 → 它的 w 不可搬
> · 它没报副作用 → 我需要自己测
>
> **“论文没写”不等于“论文错了”，但它意味着“这一块要你自己补”。**

7.36.8 读这几篇的顺序

如果只有两个小时：读 ② 的正文（4 页），跳过实验部分。它会给你 CFG 的完整动机和公式，**也仅止于此**——三段行为、标签丢弃的必要性、与分类器引导的定量对比，都不在里面。

如果有半天：按 ② → ① → Imagen（④ 里提到的那篇）的顺序。

· ② 给你“是什么”
· ① 给你“为什么需要引导”（它的动机是“扩散模型打不过 GAN”）
· Imagen 给你“工业界怎么修 w 大的副作用”（rescale 与 dynamic thresholding）

读 ① 时看两处：引导公式（§2）和分类器怎么在噪声上训练（§4）。**它的实验部分可以跳**——ImageNet 上的 FID 数字和你的场景没关系。

读 Imagen 时只读 §2.2（guidance 那一节）。**它的贡献是“怎么让大 w 能用”，不是“怎么引导”**——引导公式它直接引用了 ①。

③ **没有经典论文。**negative prompt 是工程实践先流行起来的，学术分析散落在若干篇关于“引导放大偏见”的工作里。**别指望找到一篇“negative prompt 论文”。**

⑤ **的读法**：在蒸馏类论文里搜“guidance”，**看它出现在训练目标里还是推理参数里。**这决定了你能不能外挂 CFG。

7.37 预告：文本条件下的 CFG 会变成什么样

本章的条件是**一个离散标签**。第 8 章的条件是**一串文本 token**。公式一个字不用改，但有三处会变得不一样。

① **空条件来自编码器。**

空类是 `nn.Embedding` 里学到的一行；空条件是**空字符串经过文本编码器的输出**——一个非零向量。

这意味着“无条件”的质量取决于编码器。本章实验四已经演示： 换成零向量，越界率从 0 变成 100%。

② **引导作用在整串文本上，不是单个 token**。

$\varepsilon_c - \varepsilon_\varnothing$ 是“整段提示词”带来的方向。**你没法直接说“加强’红色’这个词”**—— 只能靠调整提示词本身（比如重复、加权语法）。

这也是为什么第 8 章要讲交叉注意力： 只有看到每个 token 对图像的贡献，才知道引导到底在放大什么。

③ **文本的“类别边界”比标签模糊**。

类别 0 和类别 1 有明确的划分；“一只猫”和“一只狗”在嵌入空间里 **有重叠，而且受上下文影响**。

推论：文本模型能承受的 w 通常比本章高（5–8 vs 2）， **但过引导的表现更隐蔽**——不是飞出范围，是“每个词都被过度强调”， 画面变得油腻、元素堆砌。

> ℹ 一个可以带到第 8 章的问题
>
> **如果 w 放大的是“整段文本”的方向，那么一段长提示词里， 每个词的贡献是被同比例放大的吗?**
>
> **不是**。引导放大的方向由 $\varepsilon_c - \varepsilon_\varnothing$ 决定，**这个方向在嵌入空间里更接近那些“对输出影响大”的词**。
>
> **所以长提示词里，关键词会被放大得更多，虚词几乎不变**。这是一个可检验的预测——第 8 章的注意力图能验证它。

7.38 本章没有回答的问题

7.38.1 问题一：为什么真实模型能承受 $w = 7.5$ 而本章的模型在 20 就崩?

本章只测了二维、5 万参数的模型。真实模型的误差小得多，但“误差小”能不能解释从 20 到 7.5 的差距，本章没有验证。

可能的实验：在同一个二维问题上，用不同训练步数得到不同误差的模型，测每个模型的“崩溃 w”。**如果崩溃 w 随误差单调上升，就能画出这条曲线**。

7.38.2 问题二：引导强度与多样性之间有没有定量关系?

本章有数据点（w 从 1 到 10，内月标准差从 0.336 掉到 0.107）， **但没有拟合出一个式子**。

可能的实验：在多个数据集上重复这个扫描，看“标准差随 w 的下降” 是不是一个通用的函数形式。**如果是，就能给出“给定可接受的多样性损失， 最大 w 是多少”的公式**。

7.38.3 问题三：负引导的最优强度为什么是正的 1/3?

本章只给了经验建议（1/2 到 1/4），没有推导，也没有系统扫描。

可能的实验：对负引导做和正引导一样的 w 扫描， 找出“目标类准确率”和“越界率”的平衡点，看它和正引导的平衡点是什么比例。

7.38.4 问题四：文本条件的 CFG 和类别条件有什么本质区别?

本章的条件是一个离散标签 (2 类)。第 8 章的文本条件是高维连续向量，**空类对应的是空字符串**。

区别在于：文本的“无条件”不是一个训练出来的嵌入， 而是编码器对空串的输出。**这个差别会不会改变 w 的行为? 本章没有答案**。

7.38.5 问题五：分类器引导在什么情况下仍然值得用?

本章的结论是“CFG 取代了它”，但那只在“你有一个条件模型”的前提下成立。

如果你只有一个无条件模型，又想要条件生成，分类器引导仍然是唯一的现成办法。**它在什么条件下还能打，本章没有测**。

7.38.6 问题六：CFG 能不能“只引导一部分”?

本章的 w 是全局的。 但一个自然的想法是：只放大某个方向上的分量（比如只放大“形状”不放大“颜色”）。

APG 的投影是一种做法，但它依赖“投影到哪个方向”的选择。**这个选择有没有原则性的依据，本章没有答案。**

可能的实验：在二维上构造“形状/颜色”两个可分的条件，分别引导，看是否能独立控制。

7.38.7 问题七：多步之间的引导强度应该怎么变?

本章只测了固定 w。练习四提了 $w(t)$ 调度的想法，**但没有实测。**

可能的实验：在 10 步和 50 步下，分别测“前段 w 小、后段 w 大”与“固定 w”的差异。**如果前段引导是破坏性的，这个改进应该是免费的。**

> 为什么把没做的写出来
>
> **因为“CFG 好用”这句话，掩盖了一堆边界条件。**
> 本章的实测支持的是：**在这个二维模型上，w 有三段行为，标签丢弃是前提，准确率会骗人。**它不支持：“w 应该取 7.5”、“CFG 一定比分类器引导好”、“负引导随便调”。
> **把边界写出来，你才知道哪些能直接用。**

7.39 每一节的边界

每一条结论都有前提。这一节把它们列出来——**用之前先看前提是否成立。**

结论	前提	什么时候失效
w 的三段行为	模型在 $w=1$ 已基本正确（0.965）	模型差时，整条曲线左移，“正常区”消失
负引导更脆	空条件有效	空条件坏时，负向和正向一样不可用
标签丢弃是前提	条件能被“丢掉”	文本条件下丢的是空串，不是空类
分类器引导可比	有噪声分类器	没有分类器时无法比较
分类器的失效曲线	特定的分类器架构	换架构，$t \geq 600$ 这个位置会变
有效维度 1.009	二维 PCA	高维需要换度量（如参与率仍可用，但解释不同）
成本 2.14 倍	5 万参数模型	大模型上反向传播占比不同
两路合并省 44.5%	GPU、batch 1000	小 batch 或 CPU 上收益更小

最需要警惕的是第一行。

本章的模型在 $w=1$ 时准确率 0.965——**它已经基本学会了条件。**三段行为是“在一个已经学会的模型上，把条件信号放大”的行为。

如果模型在 $w=1$ 只有 0.7，你看到的不是三段，是一段：从 0.7 开始，无论怎么调 w 都上不去，只是越来越崩。

! 一个判断“我的模型够不够格”的快速方法

看 $w = 1$ 到 $w = 2$ 的增益。

- **有明显增益**（如 0.965 → 0.998）→ 模型学会了条件，引导有用
- **几乎没增益**（如 0.70 → 0.71）→ 模型没学会，先修模型
- **增益为负** → 模型在 $w = 1$ 就不对，任何引导都救不了

本章的增益是 +3.3 个点；这个数在你的模型上应该是多少，取决于你的任务难度——但“有增益”是必要条件。

7.40 如果只记三件事

这一章有 28,000 字，但真正要带走的只有三条。

① CFG 是锐化，不是靠近。

$$\hat{\varepsilon} = \varepsilon_u + w(\varepsilon_c - \varepsilon_u), \qquad p_w \propto p(x \mid c)^{w(p(x))^{1-w}} \tag{7.45}$$

$w = 1$ 停在数据上，$w > 1$ 越过数据继续往外推。手算的例子：真实数据在 -1，$w = 2$ 时等效均值 -2.2。

“更听话”和“更偏离”是同一个操作的两个说法。

② 准确率会在最需要它的时候饱和。

$w = 3$ 之后准确率就是 1.000，$w = 20$ 时它还是 1.000—— **而那时 37.5% 的样本已经飞出数据范围，有效维度塌到 1.009。**

必须同时看一个分布指标。本章用越界率 + PCA，图像上用 同提示词相似度 + 解码后统计量。

③ 空条件必须被训练过，而且不能是零。

标签丢弃不是正则化，是 CFG 的语义前提。零向量当空条件：越界率 100%、多样性 0，**而准确率仍是 1.000。**

换编码器、换 tokenizer、换空串写法，都要重新验证。

! 三条之间的关系

第一条是“它是什么”，第二条是“怎么判断它做得对不对”，第三条是“它成立的前提”。

只有第一条：你会把 w 调到 40，然后得到一堆废点。**只有第二条**：你知道指标在骗人，但不知道为什么。**只有第三条**：你避免了崩溃，但不知道正常时该长什么样。

三条一起，才是一个完整的理解。

7.41 这一章的方法论

为什么这一章是“手算 + 二维实验 + 失败清单”这个结构?

因为 CFG 有三个层面的知识，**它们需要三种不同的验证方式**：

层面	例子	怎么验证
代数	$\hat{\varepsilon} = \varepsilon_u + w(\varepsilon_c - \varepsilon_u)$	手算一个例子
分布	$p_w \propto p(x \mid c)^{w(p(x))^{1-w}}$	数值积分 / 密度比
经验	w 的三段行为	受控实验

只做代数，你会得到一堆恒等式，但不知道 $w = 20$ 会怎样。**只做实验**，你会得到一堆曲线，但解释不了为什么。**只做分布分析**，你会知道形状，但不知道实现要怎么写。

7.41.1 为什么先手算

手算例子的作用不是“验证公式”，是“建立量级感”。

在算之前，“$w = 2$ 会推多远”是一个抽象问题。算完之后，你有一个具体的数：**等效均值从 -1 变成 -2.2**。

这个数会一直跟着你——以后看到任何“引导强度”的参数，你都会本能地问：“它的单位是什么？推多远算过头？”

7.41.2 为什么用二维

因为二维能同时提供三样东西（和第 6 章的理由类似，但这里更极端）：

1. **可画**：$w = 20$ 时那条冲出去的线，一眼就能看见
2. **可算**：PCA、越界率、密度比，都有定义
3. **可穷举**：10 个 w × 2 个类 × 5 个种子，几分钟跑完

在图像上，这三样都没有。你只能看“像不像”。

7.41.3 为什么保留失败清单

因为 CFG 的实现错误有一个特点：它们大多不报错。

空类编号错、参数化差 1、丢弃没加——**程序都正常运行，输出都“看起来像图”**。唯一的发现方式是**知道该看什么**。

这一章的六个失败，就是“该看什么”的清单。

> 这套方法可以搬到别的地方
>
> **任何“两个模型输出相减再放大”的技巧，都可以用同样的方式分析**：
>
> - 先手算一个一维例子（量级感）
> - 再在小规模数据上扫参数（三段行为）
> - 最后列出“不报错但出错”的清单
>
> **第 6 章的采样器是这样，本章的 CFG 是这样，第 8 章的交叉注意力大概率也会是这样。**

7.42 三次真实的返工

这一章不是一次写成的。有三次，我以为已经写完，然后发现自己错了。

7.42.1 第一次：零向量当空条件

症状：token 条件模型的实验里，$w = 3$ 时 100% 的样本越界，$w \geq 5$ 时多样性归零。**而准确率是 1.000**。

误判：一开始我以为是 `w` 太大。把 w 降到 1.5，越界率仍在 80% 以上。

定位：把“无条件”分支的输入打印出来——是一个全零张量。而训练时模型见过的“空”是 `emb[vocab]`，一个非零向量。

修正：用学出来的 NULL token。越界率立刻回到 0。

教训：“没有条件”在数学上不是一个零向量。**它是一个被训练过的、表示’没有条件’的条件。**

后来我把这个错误原样写进了正文——因为它精确对应文本模型的情况。

7.42.2 第二次：跳步 DDPM 的系数写反

症状：步数实验全 NaN，**但准确率显示 1.000**。

误判：先怀疑是 `w=20` 太大导致溢出，单独跑 $w = 1$ 也一样。

定位：打印 `beta_eff`，发现是负数。

修正：把 $\alpha_{\text{eff}} = \bar{\alpha}_{t'}/\bar{\alpha}_t$ 改成 $\bar{\alpha}_t/\bar{\alpha}_{t'}$。步数实验立刻正常。

教训：`argmax` 对 NaN 会给出确定答案。**在任何指标之前，先检查 `isfinite`**。

7.42.3 第三次：同一节写了两遍

症状：清点章节时发现“三个真实场景”出现了两次—— 一次是简版，一次是展开版。

原因：插入展开版时，我用“### 三个真实场景”当锚点，把新内容插在了它**前面**，而不是替换它。
修正：删掉简版，保留展开版。
教训：**插入式编辑要检查锚点前后有没有重复。** 这也是为什么我后来给 `lint.py` 加了公式检查——**能自动查的东西，不要靠记性。**

> 三次返工的共同点
>
> **它们都不是“想错了”，是“没检查”。**
>
> - 第一次：没检查无条件分支的输入
> - 第二次：没检查中间量的有限性
> - 第三次：没检查插入后有没有重复
>
> **而三次都发生在“结果看起来正常”的时候**——准确率 1.000、loss 在降、章节能渲染。**“看起来正常”是最危险的信号。**

7.43 结语：CFG 教给我们的事

CFG 的形式极简：两个输出相减，乘一个系数，加回去。
但它的行为不简单。 同一个式子，$w = 1$ 时是精确的条件采样，$w = 2$ 时开始偏离数据，$w = 40$ 时把分布压成一条线。**中间没有警告，没有报错，准确率一直是 1.000。**
这件事本身就值得记住：

一个系统的“正确性指标”和“健康程度”可以完全脱钩。

7.43.1 一个可迁移的模式

“有条件输出 - 无条件输出”这个差，在别的地方也出现过：

- 语言模型的对比解码（contrastive decoding）
- 某些对齐方法里的“拒绝方向”
- 图像编辑里的“方向编辑”

它们都在做同一件事：找到一个方向，然后放大它。
而本章的所有教训都可以平移：

1. **方向要有意义**——空条件必须是训练过的（否则你放大的是随机方向）
2. **放大有代价**——p_w 不再等于后验（否则你以为在采样，其实在锐化）
3. **指标会骗人**——准确率看不见分布形状（否则你会把 w 调到 40）

7.43.2 最后一句

这一章讲的是“怎么让模型听话”。
但更重要的收获可能是：当一个方法简单到只有一行公式时，它的边界条件往往藏在实现细节里。
CFG 的一行公式，对应六个不报错的坑。这六个坑，比那一行公式更值得记住。

7.44 五个“不要”

这一节是给赶时间的人看的。 五条禁令，每条背后都有一个本章的实测。
① **不要用单一指标选 w。**
准确率在 $w = 3$ 就饱和，之后再怎么调都不变，**而样本在 $w = 20$ 已经飞出数据范围。至少加一个分布指标。**
② **不要在模型没训好时调 w。**
$w = 1$ 准确率 0.7 的模型，调 w 只会让它更崩。**先修模型**（本章：“开 CFG 之前”的五条清单）。
③ **不要把 w 的数值跨模型搬。**
本章 $w = 20$ 崩，SD 用到 12 也不崩。**能搬的是三段结构，不是分界点。**

④ **不要把空条件当零向量。**
零向量当空条件：越界 100%、多样性 0，**准确率 1.000。 空条件必须是被训练过的表示。**
⑤ **不要在没有 $w = 1$ 检查的情况下改采样代码。**
参数化差 1、空类编号错——**这两类错误都不会报错，** 只有逐位比较能发现。

⚠ 五条的共同点

它们都是“跳过验证、直接调参”。
调参看起来很快——改一个数、出一张图、看着不错。 **但每一次“看起来不错”，都可能是准确率 1.000 的假象。**
本章花了 30,000 字说明一件事： 先验证结构，再调参数。

7.45 调试工具：三个打印语句

不用装任何工具，三行 print 就能定位大部分问题。

7.45.1 第一行：空类编号

```
print("null id =", model.n_class, "| embedding rows =", model.cemb.num_embeddings)
```

应该看到：`null id` 等于 `rows - 1`。 不相等，说明你的空类和最后一个真实类用了同一个下标。

7.45.2 第二行：三个 ε 的范数

```
print(f"|eps_c|={eps_c.norm():.2f} |eps_u|={eps_u.norm():.2f} "
      f"|eps|={eps.norm():.2f} w={w}")
```

正常的形状：

- $w = 1$：`|eps| == |eps_c|`（引导不生效）
- $w = 2$：`|eps| ≈ |eps_u| + 2·|eps_c - eps_u|`
- **如果 `|eps_u|` 远大于 `|eps_c|`**，说明空条件主导了输出——可能编号错了

这一行在 t 大的时候尤其有用：高噪声处 ε 的范数应该接近 $\sqrt{d}$。

7.45.3 第三行：采样过程中的越界率

```
if t % 100 == 0:
                                        print(f"t={t}          max|x|={x.abs().max():.2f}
out={(x.abs().max(1).values>1.5).float().mean():.3f}")
```

正常的形状：越界率在采样后段应该下降（样本收敛到数据范围）。 **如果它在某一步开始上升**，那一步就是问题开始的地方。

💡 三行 print 覆盖了什么

打印	能发现
空类编号	失败三
ε 范数	失败一、三、参数化
逐步越界率	失败五（采样器）、过引导

加上 $w = 1$ 的逐位检查，本章六个失败里的五个都能定位。
不需要 tensorboard，不需要 wandb。

7.46 常见误用清单（按后果排序）

按“造成多大破坏”排，不按“多常见”排。

#	误用	后果	本章的数
1	空条件用零向量	**样本全废**	越界 100%，多样性 0
2	无标签丢弃就开 CFG	$w > 1$ 全崩	越界 98.6%/100%
3	用准确率选 w	选到崩溃区	$w = 20$ 准确率 1.000，越界 37.5%
4	跨模型搬 w	过引导或欠引导	本章 20 崩，SD 12 不崩
5	少步模型外挂 CFG	双重引导	—
6	负向与正向同强度	发灰、结构缺失	$w = -2$ 越界 31%/65%
7	只在一个步数上测	换步数就崩	10 步 $w = 5$ 仅 0.612

> ! 前两名和后面五名不是一个量级
>
> **第 1、2 名是“结构性错误”**——不管 w 取多少都不工作，而且**准确率可能显示 1.000**。
> **第 3–7 名是“参数错误”**——能工作，只是不够好。
> **排查时先排除前两个**：打印空类编号、检查训练时的丢弃。**这两件事加起来不到十分钟**。

一个反过来的观察：这七条里，**有五条的“症状”在指标上看起来是好的**（准确率高、loss 在降、图能出）。

这就是这一章反复强调“至少三个指标”的原因。一个指标只能告诉你“某一面还行”，不能告诉你“整体没问题”。

7.47 结课自测：十个判断

先自己判对错，再看答案。答错三个以上，回去重读对应小节。

1. CFG 让样本更接近条件。错。 它沿着条件方向**外推**。$w = 2$ 时等效均值从 -1 变成 -2.2，越过了数据。

2. $w = 1$ 就是普通条件采样。对。 公式在 $w = 1$ 时精确退化成 ε_c。这也是最重要的验收检查。

3. 标签丢弃是一种正则化。错。 它是 CFG 的语义前提。去掉它，$w = 5$ 时 98.6% 的样本越界，而 $w = 1$ 的性能几乎不变（0.960 vs 0.958）。

4. 准确率越高说明引导越好。错。 准确率在 $w = 3$ 饱和，$w = 20$ 时仍是 1.000，**而那时 37.5% 的样本已经飞出数据范围**。

5. 负引导和 negative prompt 是一回事。错，但相关。 负引导改的是“条件方向的符号”；negative prompt 改的是“空条件的内容”。**两者都推离某物，但走的是不同的项**。

6. 空条件可以用零向量。错。 零向量是模型没见过的输入。实测：越界 100%，多样性 0。

7. 分类器完美时，分类器引导与 CFG 等价，且 $w = 1 + s$。对。 两者只差“用什么估计分类器梯度”。

8. w 的最优值可以跨模型搬。错。 本章 $w = 20$ 崩，SD 用到 12 也不崩。**能搬的是三段结构，不是分界点**。

9. 步数越少，可用的 w 越宽。错，正好相反。 10 步时 $w = 5$ 只有 0.612，50 步时是 1.000。

10. 过引导的样本仍然会被分类器认对。对。 这正是“准确率会骗人”的根源——**飞出数据范围的点，类别标签可能还是对的**。

> 如果你答对了 8 个以上
>
> **说明你已经掌握了这一章的骨架。**
> **剩下的不是“记住更多”，是“动手验证”**——挑练习二（找你自己模型的崩溃 w），那个数是你以后设默认值的依据。

7.48 一张速查卡

打印出来贴在显示器旁边的那种。

7.48.1 公式

$$\hat{\varepsilon} = \varepsilon_{\varnothing} + w(\varepsilon_c - \varepsilon_{\varnothing}), \qquad p_w \propto p(x \mid c)^{w(p(x))^{1-w}} \tag{7.46}$$

w	含义
0	无条件
1	普通条件采样
> 1	引导（锐化）
< 0	负引导

7.48.2 三段行为（本章实测）

区间	名称	准确率	其他
$0 \le w < 1$	欠引导	$0.52 \to 0.85$	—
$1 \le w \le 10$	正常	$0.965 \to 1.000$	多样性 $0.31 \to 0.11$
$w \ge 20$	过引导	1.000	越界 39.7% / 9.3%

7.48.3 三条验收

检查	抓什么
$w = 1$ 逐位等于条件采样	参数化
$w = 0$ 准确率 $\approx 1/C$	空类编号
越界率随 w 单调不降	“CFG 没接上”

7.48.4 六个坑

#	坑	后果
1	空类没训过（drop=0）	越界 98.6%
2	参数化差 1	强度差一档
3	空类编号错	偏某一类
4	只看准确率	选到 $w = 40$
5	采样器不稳	误怪引导
6	负引导过强	越界 31% / 65%

7.48.5 五个数

- 模型 50,434 参数；裁判 0.9978
- $w = 2 \to$ 准确率 0.998，越界 0
- $w = 40 \to$ PC1 0.996，有效维度 1.009

- 10 步 $w = 5 \to 0.612$（50 步 1.000）
- 分类器引导单步成本 = CFG 的 2.14 倍

如果只能记一行：$w = 1$ 逐位检查。 它最便宜，也最能排除掉一整类错误。

7.49 如果要继续往下走

这一章讲的是“图像 + 类别”的 CFG。往下有三个方向。

7.49.1 方向一：别的模态

同样的公式，换一个“条件”和“数据”：

- **视频**：条件仍是文本，但 x_t 是时空张量；引导要处理“时间一致性”
- **3D**：条件是视角或文本，x_t 是隐式场或高斯点
- **音频**：条件是文本或参考音频，x_t 是频谱

公式不变，但“越界”的含义和评估指标全要重定。 本章的“从二维走到图像的四个台阶”可以直接套用。

7.49.2 方向二：别的训练范式

流匹配 / 整流流的引导。 它们也有“条件速度场 - 无条件速度场”，**理论上可以套用同样的差。**但流匹配的路径是直的，**引导对路径曲率的影响可能不同**——这一点本章没有数据。

7.49.3 方向三：理论

为什么锐化会提高“符合度”？ 本章用 p_w 解释了“分布变尖”，但**没有解释“为什么变尖会让人类觉得更符合提示词”**。

一个可能的解释：人类的“符合”判断对**典型性**敏感—— 越靠近类中心的样本越“像那个东西”。**这是认知层面的假设，本章无法验证。**

> **i** 一个诚实的边界
>
> **本章解释了 CFG 的机制，但没有解释它的“有效性”。**
> **机制**：为什么 w 会让分布变尖、为什么会崩、什么时候该停。**有效性**：为什么人会觉得 $w = 2$ 的图“更好”。
> **后者涉及感知和认知，不在本书范围内。**但它是真实存在的问题——**如果换一个评价者（比如另一个模型），最优 w 可能会变。**

7.50 把这一章讲给别人听

同一份内容，三种时长。

7.50.1 五分钟版

三句话：

1. **CFG 是把“有条件输出 - 无条件输出”这个差放大 w 倍再加回去。**
2. **$w = 1$ 是普通条件采样，$w > 1$ 会越过数据继续往外推。**
3. **准确率在 $w = 3$ 就饱和，之后它不再告诉你任何事。**

如果对方只问一个问题，多半是“那 w 该取多少”。**答**：取“准确率刚饱和、多样性还没明显掉”的那个点。本章是 2，真实模型通常 5–8。

7.50.2 三十分钟版

在五分钟版之上加两块：

① **手算的例子**。真实数据在 -1，$w=2$ 时等效均值 -2.2。**这一个数就能让人明白“不是靠近，是外推”**。
② **三段行为**。欠引导 / 正常 / 过引导，每段配一个数。
这一段的作用是让人知道“存在一个危险区”，而不只是“有个参数”。

7.50.3 两小时版

完整讲，但顺序调整过：

1. 先讲**失败**（无标签丢弃 → 越界 98.6%）
2. 再讲**公式**（此时听众已经知道“为什么需要这个公式”）
3. 然后**手算**（建立量级感）
4. 最后**三段行为与三个指标**

先失败后公式，比“先公式后例子”更容易让人记住—— **因为失败制造了一个“为什么”的问题**。

> 讲的时候最容易被追问的三个问题
>
> ①**“为什么不能直接让模型更听话?”**因为它只会“更像某个类”，不会“更符合你的意图”。**引导是一个方向，不是一个目标**。
> ②**“为什么 w 大了画面会坏?”**因为 $p_w = p(x \mid c)^{w(p(x))^{1-w}}$ 不是后验，是被重新加权过的分布。**你在采样一个被改过的分布**。
> ③**“有没有办法既要多样又要听话?”有，但不能靠调 w**。要靠更好的模型、或者换条件表示（比如文本里说得更具体）。**w 只能在“多样”和“符合”之间移动，不能同时提高两者**。

7.51 三个数学性质

这三个性质解释了“为什么 CFG 看起来这么自然”。

7.51.1 性质一：$w=1$ 时精确

$$\varepsilon_u + 1 \cdot (\varepsilon_c - \varepsilon_u) = \varepsilon_c \tag{7.47}$$

这不是近似，是恒等。它意味着 CFG 是条件采样的**严格推广**，不是“另一个方法”。
实践含义：任何声称“用了 CFG”的实现，在 $w=1$ 时必须与普通条件采样逐位相同。不同就是错的。

7.51.2 性质二：对 ε 的尺度是等变的

如果模型输出整体乘一个常数 a（比如换了一个尺度约定），CFG 的结果也乘 a：

$$\hat{\varepsilon}(a\varepsilon_c, a\varepsilon_u) = a\,\hat{\varepsilon}(\varepsilon_c, \varepsilon_u) \tag{7.48}$$

因为公式对两个输入都是线性的。
实践含义：如果换了模型、ε 的尺度变了，**w 的“有效强度”也会跟着变**。这就是“w 不能跨模型搬”的一个数学解释。

7.51.3 性质三：对条件不是线性的

如果 ε_c 随 c 非线性变化（真实模型都是），**CFG 的输出对 c 也不是线性的**。
这意味着“两个条件的引导可以相加”只在近似意义下成立（见问十一）。**严格做法是分别算，再检查范数**。

> **i** 为什么讲这三个
>
> **因为它们划出了 CFG 的“代数边界”。**
>
> · 性质一告诉你**什么是错的**（$w = 1$ 不等就是错的）
> · 性质二告诉你**什么不能搬**（尺度不同的模型之间）
> · 性质三告诉你**什么只能近似**（多条件组合）
>
> **剩下的行为，都要靠实验。**

7.52 五个“如果”

现实里总有些约束让标准做法不适用。 五条应对。

如果只能跑 5 步？ 把 w 压到 1.5–2。 本章实测 10 步时 $w = 5$ 只有 0.612。 少步时引导的收益抵不过它带来的离散误差。

如果条件不是离别的？ 照样能用，但“空条件”要重新定义。 连续条件（比如温度、亮度）没有“空”这个概念， 通常用“条件取某个中性值”代替——**这个中性值必须是模型训练时见过的。**

如果没有空标签可丢？ 训练时手工构造一个。 比如把条件向量乘 0（但要作为**训练分布的一部分**， 不是推理时临时拼）。本章的教训是：**空条件必须在训练中出现过。**

如果模型已经蒸馏过？ 先做 w 扫描的连续性检测（场景 C）。如果 w 一调就崩， 保持 $w = 1$。

如果只有 CPU？ 减步数、减 batch，但别减 w 的检查。 本章的三个恒等检查不依赖算力， **它们是“十分钟能做完”的那一类。**

> 这五个“如果”的共同答案
>
> **先确认“哪一部分不能动”，再调剩下的。**
>
> · 步数不能动 → 动 w
> · 条件不能动 → 动空条件的表示
> · 模型不能动 → 动期望（接受质量下降）
>
> **唯一不能省的是验证**：$w = 1$ 的逐位检查、$w = 0$ 的无条件检查。 **这两条在任何算力下都只要几分钟。**

7.53 写在这一章的最后

这一章从一个不会听话的模型开始，结束于一张速查卡。

中间有三十多张表、两张图、一组交互实验、六个失败、三次返工。但它们都在回答同一个问题：

当你把“条件”变成一个可以拧的旋钮时，你到底在拧什么？

答案是：你在改变采样的分布，不是改变模型的能力。

这句话有两个推论，值得带走：

① **引导不能创造模型没有学到的东西。**模型对某个类一无所知，w 再大也只是把“不知道”放大。

② **引导的代价总是从别处来。** 准确率从多样性来，符合度从覆盖来，锐度从真实感来。 **没有哪一项是免费的。**

> ! 如果这一章只能留一句话
>
> **“w 是模型的属性，不是任务的属性。”**
> **换模型，重新标定；换步数，重新标定；换编码器，重新标定。**
> **唯一不变的是那三步**：验证 $w = 1$、验证 $w = 0$、看三个指标。

7.54 如果你只有一小时

这一章约四万字。时间不够时，按这个顺序读。

时间	读什么	得到什么
0–15 分	“CFG 的推导”整节	公式从哪来，w 的三个端点
15–25 分	图 7-1 + “实测：w 的三段行为”	三段结构的实测形状
25–40 分	“它在哪里失效”的六个失败	不报错但出错的六种方式
40–50 分	实验 7-1（拖 w）	亲手看见 (w=20) 的塌缩
50–60 分	速查卡 + 结课自测	带走一页纸和十个判断

可以跳过的：分类器引导的历史、三个变体、延伸阅读、以及大部分实现细节。
不要跳过的：$w = 1$ 与 $w = 0$ 的检查、以及“准确率会饱和”那一节。**这两块决定了你会不会踩坑。**

7.55 一句话

CFG 是一个放大器：它放大“条件带来的差别”，代价是分布被改形。
把它拆开看：

- **放大**：w 乘在 $(\varepsilon_c - \varepsilon_u)$ 上，不是乘在整个输出上
- **条件带来的差别**：那一项就是隐式分类器的方向
- **分布被改形**：$p_w = p(x \mid c)^{w(p(x))^{1-w}}$，不是后验

三句话对应三个坑：

- 不懂“乘在哪” → 参数化差 1
- 不懂“差别从哪来” → 空类编号错、没做丢弃
- 不懂“分布被改形” → 只看准确率，选到崩溃区

所以这一章的每一个实验，本质上都在测这三件事之一。

7.56 本章回顾

把这一章压缩成六句话。
一、条件生成的核心是一个恒等式。 $\nabla \log p(x \mid c) = \nabla \log p(x) + \nabla \log p(c \mid x)$。分类器引导用第二项，CFG 用两个 score 的差近似它。
二、CFG 只有一行公式。 $\hat{\varepsilon} = \varepsilon_\varnothing + w(\varepsilon_c - \varepsilon_\varnothing)$。$w = 0$ 是无条件，$w = 1$ 是普通条件采样，$w > 1$ 才是“引导”。
三、w 有三段行为。 $w < 1$ 欠引导（准确率 0.52→0.85），$1 \le w \le 10$ 正常（准确率到 1.000，多样性从 0.336 掉到 0.107），$w \ge 20$ 过引导（37.5% 的样本越界）。
四、准确率会在最需要它的时候饱和。$w = 3$ 之后准确率就是 1.000，$w = 20$ 时它还是 1.000——**而那时样本已经飞出数据范围了。**
五、标签丢弃是 CFG 的前提，不是正则化。 去掉它，$w = 2$ 就有 6%/22% 的样本越界，$w = 5$ 时是 98.6%/100%。**空类嵌入必须被训练过，否则你减掉的是一个随机向量。**
六、分类器引导输在“多了一个会失效的部件”。 它的分类器在 $t \ge 600$ 之后准确率就是 0.500，而 $\sqrt{1-\bar{\alpha}_t}$ 在这些时刻**最大——最不可靠的梯度被乘上了最大的系数。**
七、“能用的 w”有多宽，取决于步数和丢弃率。10 步时 $w = 5$ 的准确率只有 0.612，50 步时是 1.000；丢弃率 0 会让 $w > 1$ 直接崩，0.3 会削弱条件本身，0.05–0.1 是宽平台。**引导强度不是一个孤立参数。**

7.56.1 七句话的展开

一、条件 score 的恒等式只用贝叶斯一步：取对数、对 x 求梯度，$p(c)$ 与 x 无关所以消失。**三项分别是：无条件 score、分类器梯度、条件 score。**分类器引导取后两项，CFG 用模型自己的输出去近似那个梯度。

二、CFG 的那一行公式有三个端点：$w=0$ 无条件、$w=1$ 普通条件、$w>1$ 引导。**注意 $w=1$ 那一端**——它说明 CFG 不是“新技巧”，是条件采样的推广。

三、三段行为的分界不是拍出来的：$w<1$ 准确率 0.52→0.85，$1\le w\le 10$ 到 1.000 且多样性从 0.336 掉到 0.107，$w\ge 20$ 越界率 39.7%/9.3%。**每一段都有对应的机制**（欠推、锐化、塌缩）。

四、准确率饱和发生在 $w=3$，而副作用在 $w=10$ 才开始。**中间这七个 w 档位，是“看起来在变好、其实只在变窄”的陷阱区**。

五、标签丢弃去掉之后，$w=2$ 越界 6%/22%、$w=5$ 越界 98.6%/100%。**它只影响 $w>1$**——$w=1$ 时两个模型分别是 0.960 和 0.958，几乎没有差别。这个“只影响一边”的模式，正是“空类没被用到”的指纹。

六、分类器引导的失效不在低噪声，在高噪声：$t\ge 600$ 后准确率 0.500。而 $\sqrt{1-\bar\alpha_t}$ 在那个区间最大——**最不可靠的梯度被乘上最大的系数**。

七、步数与引导的耦合：$w=5$ 在 10 步时准确率 0.612、50 步时 1.000。**“能用的 w”不是一个常数，是步数的函数**。

如果把这一章压成一张表：

问题	答案	本章的数
引导到底在干什么	锐化后验：$p_w\propto p(x\mid c)^{w(p(x))^{1-w}}$	$w=1\to 10$, 中心/边缘密度比涨 1000 倍
w 该取多少	$1\le w\le 10$，且随步数收窄	10 步时 $w=5$ 只剩 0.612
为什么准确率会骗人	它是 0/1 阈值指标	$w=20$ 准确率 1.000，越界 37.5%
空类怎么来	训练时的标签丢弃	无丢弃 $w=5$ 越界 98.6%
空条件长什么样	学出来的嵌入，不是零向量	零向量：越界 100%，多样性 0
分类器引导差在哪	多一个会失效的部件	分类器 $t\ge 600$ 后准确率 0.500
两种引导什么关系	完美分类器下 $w=1+s$	$s=0\leftrightarrow w=1$，都是 0.965
过引导的几何	塌到低维 + 向外漂移	$w=40$：PC1 0.996, 半径 2.25

这张表的每一行，都能在本章找到一张对应的原始数据表。

如果只带走一句：**w 是模型的属性，不是任务的属性**。同一个任务换一个模型，最优 w 会变；同一个模型换一个任务，最优 w 也可能变。**唯一不变的是三段结构和“用边际收益做决定”的方法**。

没有一行的数字是估算的。

最后留一个问题

本章的二维模型在 $w=20$ 时，外月的样本**塌成了一条向上的直线**，一直冲出画面。准确率还是 1.000。

为什么是“一条线”而不是“一团云”?

一个可能的回答：引导把“条件方向”放大了 20 倍，**其他方向被相对压扁了**。当放大倍数大到一定程度，分布的方差在条件方向上远大于其他方向，**看起来就像一维的**。

如果这个解释对，那么“过引导”的本质不是“样本变远”，而是“分布塌到一个低维子空间上”。

这个实验做了。对每个 w 下的 1000 个样本做 PCA：

w	PC1 方差占比	有效维度	平均最大半径
0	0.809	1.448	0.540
1	0.838	1.374	0.525
2	0.877	1.276	0.570
3	0.900	1.219	0.582
5	0.926	1.158	0.582
10	0.906	1.205	0.568

w	PC1 方差占比	有效维度	平均最大半径
20	**0.979**	**1.044**	**1.252**
40	**0.996**	**1.009**	**2.245**

（有效维度 = $(\sum \lambda)^2 / \sum \lambda^2$，二维数据的上限是 2。）

假设成立。$w = 40$ 时 PC1 解释了 99.6% 的方差，有效维度 1.009—— **分布几乎完全塌到一条线上**，同时平均半径从 0.54 涨到 2.25。

注意 $w \le 10$ 那一段：有效维度从 1.448 缓慢降到 1.158。这是正常的“去掉错误样本”造成的收缩。**真正的塌缩发生在 10 到 20 之间**—— 和越界率的突变是同一个位置。

所以“过引导”有两个同时发生的症状：分布塌到低维 + 整体向外漂移。越界率只是后一个症状的粗糙代理，**PCA 才能看见前一个**。

这一章的数据在 `data/ch7_case.json`、`ch7_extra.json`、`ch7_more.json`，代码在 `notebooks/conditional_diffusion.py`，交互实验在 `experiments/exp-7-1.html`。

每一个数字都能重新跑出来。如果你跑不出来，那是本书的错。

第 5–7 章是一条线：第 5 章定义前向与反向，第 6 章决定走多少步、怎么走，第 7 章决定走向哪里。三章合起来，才是一个完整的生成过程。

而这条线上的每一章，都遵循同一个写法：先给一个能算的例子，再给一组受控实验，最后列出「不报错但出错」的清单。

现象 → 机制 → 实验 → 清单 → 回到现象。这本书的每一章，都是这条线。

而线上的每一个数，都应该能被你重新跑出来。如果你跑不出来，请把它当作一处需要修订的地方——这也是它值得被读的原因。

下一章：文字如何进入图像。

下一章换一个问题：条件不再是“第 0 类”这样的离散标签，而是一整段文字。那时的 CFG 公式不变，但“空条件”变成了“空字符串”—— 第 8 章会看到，这个小小的变化让整件事复杂了一个量级。

8. 文字如何进入图像

本章地图

你在这里：第 7 章的条件是一个离散标签（“第 0 类”）。真实模型的条件是一整段文字。从“一个数”到“一串向量”，中间隔着一个文本编码器和一层交叉注意力。**本章解决**：文字怎么变成向量；交叉注意力为什么能“绑定”属性而词袋不能；注意力图能不能用来解释；空字符串为什么不是零向量。 **前置**：第 1 章（多头注意力）、第 2 章（Block）、第 5–6 章（扩散与采样）、第 7 章（CFG）。 **读完能做**：读懂一个文本到图像的 UNet 的结构；诊断“词都认识但位置搞反”的绑定失败；判断一张注意力图值不值得信。

8.1 本章的代码与数据

```
import torch, sys
sys.path.insert(0, "notebooks")
from tiny_diffusion import Config, make_schedule
from text_conditioning import (make_two_point_data, train_text_model,
                               sample_text, slot_accuracy)

x, y = make_two_point_data(4000, noise=0.08, seed=0)   # 4 维，4 个类
model, hist = train_text_model(x, y, cfg, sch, encoder="cross", steps=6000)
samples = sample_text(model, sch, tok, w=1.0)
```

复现本章全部数字：

```
cd notebooks
PY=/home/mpp/micromamba/envs/dlspace/bin/python
$PY run_ch8_case.py        # 三组实验，约 8 分钟
```

文件	内容	耗时
`notebooks/text_conditioning.py`	两种文本编码器 + 条件去噪器 + 采样	—
`notebooks/run_ch8_case.py`	绑定实验、注意力分析、CFG 扫描	~8 min
`notebooks/run_ch8_ablation.py`	对调实验（决定性证据）	~4 min
`notebooks/run_ch8_ablate2.py`	三个消融、学习曲线、注意力熵	~30 min
`notebooks/run_ch8_capacity.py`	容量对照（损失差不是参数少）	~3 min
`notebooks/bpe_demo.py`	手写 BPE + 分词效率实测	~7 min
`data/ch8_*.json`	全部实测数据	—
`qa/gen_fig_ch8.py` → `assets/fig/8-1-binding.svg`	图 8-1	—
`experiments/exp-8-1.html`	交互实验 8-1	—

全部在单张 RTX 3090 上跑完，总耗时约 50 分钟。 没有任何一步需要多卡或大显存。

本章和第 7 章共用同一套调度、同一个 ε-预测目标、同一个 DDPM 采样器。唯一的改动是**条件的表示**：从一个类别下标，变成一串 token。

8.2 动手之前：三十分钟复现核心结论

如果你手边有机器，先跑一遍再读。 这一章的所有论证，都建立在三个数上。

8.2.1 第 1 步：训练两个模型

```
cd notebooks
PY=/home/mpp/micromamba/envs/dlspace/bin/python
$PY run_ch8_ablation.py
```

这个脚本会训两个模型（均值池化和交叉注意力），各 6000 步。 在 RTX 3090 上约 4 分钟。

8.2.2 第 2 步：看三个数

脚本会打印：

```
mean RB: left_blue 0.455 right_blue 0.55
mean BR: left_blue 0.455 right_blue 0.55
mean: 对调后最大逐元素差 = 0.000e+00
cross RB: left_blue 0.0 right_blue 1.0
cross BR: left_blue 1.0 right_blue 0.0
cross: 对调后最大逐元素差 = 9.722e-01
```

三个数分别是：

数	均值池化	交叉注意力	说明
RB 的右蓝比例	0.55	1.00	该蓝的地方蓝了吗
BR 的左蓝比例	0.455	1.00	对调之后呢
对调差	**0.000**	**0.972**	同种子下两次输出的最大差

如果只允许看一个数，看第三个。

8.2.3 第 3 步：自己加一个检查

在脚本里加一行，打印“两个镜像条件的右蓝比例之和”：

```
print(swap["RB"]["right_blue"] + (1 - swap["BR"]["left_blue"]))
```

均值池化应该接近 1，交叉注意力接近 2。
这个判据比看单个准确率可靠—— 它不会被“互补计数”骗过去。

> 为什么先跑再读
>
> **因为这一章的结论有一个反直觉的地方： “准确率 0.602 可能是零信息”。**
> **你亲手跑出这个数，再看到对调差是 0.000，那种“被数据纠正”的体感，是读文字得不到的。**
> **这也是本书每一章都要求“先算一遍”的原因。**

8.3 案例：同一袋词，两种顺序

先看一个具体的问题。
数据是四个点对，每个点对有两个点：左点和右点。 每个点是“红”或“蓝”，颜色由它的 y 坐标决定（上红下蓝）。 于是有四个类：

类	左	右
RR	红	红
RB	红	蓝
BR	蓝	红
BB	蓝	蓝

条件是一句话，两个 token：`[左色，右色]`。

现在注意一件事：`RB` 和 `BR` 的**词袋是相同的**—— 都是“一个红、一个蓝”，只是顺序不同。

如果你的文本编码器把两个词取平均，这两个类会得到完全相同的条件向量。模型收到同样的输入，却要生成两种不同的输出。

这不可能做到。

> ! 这就是“绑定问题”（binding problem）
>
> **“有一个红和一个蓝”和“左边红、右边蓝”是两句话，但不是同一件事**。
>
> · **词袋**只记录“出现了哪些词”，不记录“哪个词对应哪个位置”
> · **注意力**可以让每个位置自己去“查”它需要的那个词
>
> **本章的全部内容，可以看成这一句话的展开：为什么均值池化会失败，交叉注意力怎么解决它，以及代价是什么**。

这个问题不是玩具。真实提示词里，“一只猫坐在一只狗旁边”和“一只狗坐在一只猫旁边”是同一个词袋，但生成结果应该不同。**所有文本到图像的模型都必须解决它**。

8.4 文字变成向量：三步

第一步：分词。把字符串切成 token。

中文按字或子词切，英文按 BPE（byte-pair encoding）切。**切法由 tokenizer 决定，不由模型决定**——换 tokenizer 等于换了一门语言。

第二步：查表。每个 token 查一个 d 维向量。

这一步和大模型里的词嵌入没有区别：一个 `nn.Embedding(vocab, d)`。**真正的信息不在这一层，而在下一层**。

第三步：编码器。让 token 之间互相看，产生“带上下文”的表示。

这一步才是 CLIP / T5 / Qwen 文本编码器的核心。第 1 章的 Transformer Block 在这里原样出现：自注意力 + FFN + 残差 + 归一化。

8.4.1 三种编码器的内部长什么样

“文本编码器”不是一个东西，是一族结构。三个最常见的：

	CLIP 文本编码器	T5 编码器	Qwen 系列
层数	12	12 / 24（base / XXL）	24–80
宽度	768	768 / 4096	1024–8192
注意力	**因果**（只能看左边）	双向	因果
上下文	**77 token**	变长（相对位置）	长上下文
输出	序列 + 池化向量	序列	序列

“因果”那一行值得停一下。

CLIP 的文本编码器是**因果注意力**——每个 token 只能看到它前面的。这不是为了生成，而是因为它沿用了 GPT 式的结构。

后果：最后一个 token 的表示“看过”整句，而第一个 token 只看到自己。**所以 CLIP 的池化向量取的是 `[EOS]` 位置的输出**——那是唯一“看过全句”的位置。

> ℹ 为什么这个细节重要
>
> **如果你从 CLIP 的序列里随便取一个位置当“整句向量”，会取到不完整的信息。**
> `[EOS]` 位置才等于“整句摘要”。 **很多实现出错就出在这里**——它们用了 `last_hidden_state[:, 0]`（第一个 token），那是“句首”，不是“全句”。
> **正确做法**：用 `pooler_output`（它内部就是取 `[EOS]`），或者自己按 `argmax(input_ids == eos_token_id)` 取。

T5 是双向的，所以没有这个问题——每个位置都看过全句。**这也是 Imagen 选择 T5 的一个附带好处。**

8.4.2 一个句子在编码器里经历什么

```
"一只红色的猫"
   ↓ tokenizer (BPE / SentencePiece)
[101, 3721, 4895, 1996, 8920, 102]        # 含特殊 token
   ↓ embedding
(6, 768)                                   # 6 个 token，每个 768 维
   ↓ + 位置编码
(6, 768)
   ↓ 12 × [自注意力 + FFN + 残差 + LN]
(6, 768)                                   # 每个 token 都吸收了上下文
   ↓ 取 [EOS] 位置
(768,)                                     # 整句表示
```

注意中间那行 `(6, 768)` 在每一层都保持形状不变。 编码器不改变序列长度，只改变每个位置的内容——**这是 Transformer 的通用性质**（第 2 章）。
对交叉注意力来说，最重要的是最后那个 `(6, 768)` 序列，不是 `(768,)` 的池化向量。**因为图像需要“每个词各是什么”，不是“整句大概是什么”。**

8.4.3 词表有多大

CLIP：49,408 个 token。T5：32,000。Qwen：150,000 以上。
词表越大，单个 token 承载的语义越少，但句子越短。
中文尤其明显：一个 5 万词表的 tokenizer 可能把一个汉字切成 1–3 个 token，而 15 万词表的可能一个字一个 token。**这直接影响 77 这个上限能装多少内容。**
这就是为什么中文提示词在 CLIP 模型上“更费 token”——同样一句话，中文可能占 30 个 token，英文只占 12 个。

8.4.4 分词器到底在做什么：手写一个 BPE

“按子词切”这句话太抽象。下面是一个 20 行的 BPE，跑在本书自己的正文上。
算法只有三步，循环执行：

```
① 把文本编码成 UTF-8 字节，每个字节是一个符号（初始词表 256）
② 数所有相邻符号对，找出现最多的那一对
③ 把这一对合并成一个新符号，词表 +1
```

就这么简单。GPT-2、CLIP、LLaMA 的分词器都是这个算法的变体。
为什么从字节开始？ 因为任何 Unicode 文本都能表示成字节，**不会出现“词表里没有这个字”的情况**——最坏情况是一个字被切成 3 个字节 token。
在本书第 5–8 章的正文（约 32 万字符）上训练，得到：

合并次数	词表	汉字/token（训练集）	汉字/token（留出集）	77 token ≈
300	556	0.670	**0.625**	**48 个汉字**

合并次数	词表	汉字/token（训练集）	汉字/token（留出集）	77 token ≈
1000	1,256	1.049	**0.967**	**74 个汉字**
2000	2,256	1.318	**1.210**	**93 个汉字**

“留出集”是第 8 章本身——训练时没见过。
两列很接近，说明这个测量不是死记硬背。但也别太当真：语料只有 32 万字，真实词表是 5 万级。
几个具体例子（1000 次合并后）：

字符串	token 数	说明
扩散模型	2	高频搭配被合并了
红色	5	在这批语料里少见，没有合并
一只红色的猫	11	7 个汉字，占了 11 个 token
`cross attention`	10	英文反而更费——因为语料是中文

! 三件从这张表里能读出来的事

① **一个常用汉字是 3 个字节，所以要两次合并才合成一个 token。** 词表小的时候，**一个汉字占 2–3 个 token 是常态。**
② **高频词会被合并，低频词不会。** 扩散模型 变成 2 个 token，红色 还是 5 个—— **不是“红色”更复杂，是它在这批语料里出现得少。**
③ **77 个 token 不是 77 个词，更不是 77 个字。** 在这个玩具分词器上，**77 个 token 只装得下约 50–93 个汉字——词表越小，装得越少。**
真实 CLIP 的词表大得多，但中文仍然吃亏：一个常用汉字通常 1 个 token，生僻字 2–3 个。**所以一段 100 字的中文提示词，很可能超过 77 的上限。**

这也解释了为什么“中文提示词要写短”—— 不是模型不懂中文，是 token 预算被字节吃掉了。
一个可以自己做的检验：用你手上的分词器，分别数一句中文和它的英文翻译有多少 token。**如果中文是英文的 2–3 倍，你就知道该把预算花在哪了。**

8.4.5 为什么不能只做前两步

因为同一个词在不同上下文里的意思不一样。
“苹果”在“吃苹果”和“苹果发布会”里应该有不同的向量。只查表的话，两者完全相同。
编码器的作用是让每个 token 的表示吸收上下文—— 自注意力让“苹果”看到“吃”还是“发布会”，从而调整自己的向量。

i 本章的实验用的是哪一种

本章用一个极简的替代品：两个 token（左色、右色），每个查一个嵌入，然后**要么取平均，要么做交叉注意力。**
没有自注意力层，没有 FFN。 这是刻意的：本章要回答的是“**条件向量怎么被图像使用**”，而不是“文本编码器内部长什么样”。
前者是交叉注意力的问题，后者是 Transformer 的问题—— 后者第 1、2 章已经讲过。

8.5 交叉注意力：图像问，文本答

第 1 章的注意力公式：

$$\mathrm{Attn}(Q,K,V)=\mathrm{softmax}\left(\frac{QK^{\top}}{\sqrt{d}}\right)V \tag{8.1}$$

在自注意力里，Q, K, V 都来自同一个序列（文本自己看自己）。
在交叉注意力里：

量	来自	含义
Q	**图像**特征	“我这个位置需要什么信息”
K	**文本** token	“我这里有什么信息”
V	**文本** token	“我能提供什么内容”

所以交叉注意力是“图像问、文本答”。
每个图像位置都会得到一个自己的条件向量—— 不是全图共用一个，而是“这个位置该听哪些词”。

> ! 这一句话是本章的核心
>
> **条件不能是一个“丢了顺序”的向量。**
> · **均值池化**：整张图共用一个**与顺序无关**的向量
> · **交叉注意力**：每个位置各拿一个，而且**每个位置拿到的可以不同**
>
> **注意“可以不同”——不是“必须不同”。**
> 本章后面的消融会给出一个反直觉的结果：**只要条件里带着顺序，哪怕所有位置共用同一个向量，下游也能把它分配到正确的位置。** 但那是**学出来**的；交叉注意力把这件事变成了**结构**。
> **这个区别很重要，它决定了你换模型时要验证什么。**

8.5.1 参数量：交叉注意力为什么贵

第 7 章算过一笔账（$d = 320$）：

方式	每层参数
相加 / AdaGN	$2d = 640$
交叉注意力	$4d^2 = 409,600$

相差 640 倍。 本章的小模型里，这个差距是 67,332 vs 100,740 参数（差 1.5 倍，因为 $d = 128$ 且只有一层注意力）。

8.5.2 真实模型的账

上面的 640 倍是拿“类别条件”和“交叉注意力”比。 真实模型里，条件维度不是 d 而是文本宽度，**账要重算。**
交叉注意力一层的参数（d 是特征宽度，d_c 是文本宽度）：

$$\underbrace{d^2}_{Q} + \underbrace{2\,d\,d_c}_{K,V} + \underbrace{d^2}_{\text{out}} = 2d^2 + 2dd_c \tag{8.2}$$

Stable Diffusion 1.5（$d_c = 768$）：

特征宽度 d	每层参数	层数	小计
320	696,320	5	3,481,600
640	1,802,240	5	9,011,200
1280	5,242,880	6	31,457,280
		16	**43,950,080**

约 4400 万，占 UNet 8.6 亿参数的 5.1%。
这个比例可能比你预期的小。 交叉注意力的“贵”不在参数，在前面算过的 $N \times M$ 计算。
SDXL（$d_c = 2048$，两个 CLIP 拼接）：

d	每层参数	比 SD1.5 同宽度
320	1,515,520	2.18×
640	3,440,640	1.91×
1280	8,519,680	1.62×

文本宽度从 768 涨到 2048，交叉注意力的参数涨了 1.6–2.2 倍。这就是“更强的文本编码器”的真实成本——不只在编码器本身，还在每一层交叉注意力上。

> **i** 一个反直觉的对比：MMDiT 的注意力并不更贵
>
> **SD3 Medium**（$d = 1536$，24 层）的联合注意力，每层 $4d^2 = 4 \times 1536^2 = 9,437,184$ 个参数。
> **和 SDXL 的 1280 宽交叉注意力（8,519,680）几乎一样。**
> **但 MMDiT 真正的开销在另一个地方**：adaLN 每层 $2048 \times 6 \times 1536 = 18,874,368$——**是注意力本身的两倍。**
> **24 层加起来 4.53 亿。**
> **所以“MMDiT 参数量大”的直觉是对的，但归因错了**：大头不是联合注意力，是**每个 Block 都要从池化文本重新算一遍调制参数。**
> **这也解释了为什么 DiT 类模型对“池化文本向量”的维度很敏感**—— 它直接乘在 adaLN 的参数上。

所以“交叉注意力是文本控制图像的价格”这句话，要说准确一点：参数上它只占 5%，计算上它占的是“每个潜位置都要问一遍”。后者才是真正的价格。

8.6 手算一遍：两行矩阵怎么完成绑定

公式看过很多遍，不如自己算一遍。这一节用一个**小到可以手算**的交叉注意力，把“绑定”拆到每一个数字上。

8.6.1 设定：2 维、2 个 token、2 个点

为了能手算，把维度压到 2。**真实的 d 是 768，但算术的形状完全一样。**

词嵌入（两个颜色）：

$$e_{\text{红}} = \begin{pmatrix} 2 \\ 0 \end{pmatrix}, \qquad e_{\text{蓝}} = \begin{pmatrix} 0 \\ 2 \end{pmatrix} \tag{8.3}$$

位置嵌入（第 0 个词、第 1 个词）：

$$p_0 = \begin{pmatrix} 0.5 \\ 0 \end{pmatrix}, \qquad p_1 = \begin{pmatrix} 0 \\ 0.5 \end{pmatrix} \tag{8.4}$$

两个点的 query（左点稍微偏向“问第一个词”，右点偏向“问第二个词”—— 这是模型学出来的，这里直接给定）：

$$q_L = \begin{pmatrix} 1.5 \\ 0.3 \end{pmatrix}, \qquad q_R = \begin{pmatrix} 0.3 \\ 1.5 \end{pmatrix} \tag{8.5}$$

K 和 V 用同一份表示（真实模型里是两组不同的投影矩阵，这里合并以简化）：

$$k_i = e_{w_i} + p_i, \qquad v_i = k_i \tag{8.6}$$

缩放因子 $\sqrt{d_k} = \sqrt{2} \approx 1.414$。

8.6.2 条件 RB：⟨红, 蓝⟩

两个 key：

$$k_0 = e_{\text{红}} + p_0 = \begin{pmatrix} 2.5 \\ 0 \end{pmatrix}, \qquad k_1 = e_{\text{蓝}} + p_1 = \begin{pmatrix} 0 \\ 2.5 \end{pmatrix} \tag{8.7}$$

左点的分数:

$$q_L \cdot k_0 = 1.5 \times 2.5 + 0.3 \times 0 = 3.75, \qquad q_L \cdot k_1 = 1.5 \times 0 + 0.3 \times 2.5 = 0.75 \tag{8.8}$$

除以 $\sqrt{2}$：2.652 与 0.530。**softmax**:

$$w_L = \frac{e^{2.652}}{e^{2.652} + e^{0.530}} = \frac{14.18}{14.18 + 1.70} = \begin{pmatrix} 0.893 \\ 0.107 \end{pmatrix} \tag{8.9}$$

左点 89.3% 看第一个词（红）。

右点对称：$q_R \cdot k_0 = 0.75$，$q_R \cdot k_1 = 3.75$， softmax 得到 $\begin{pmatrix} 0.107 \\ 0.893 \end{pmatrix}$——**89.3% 看第二个词（蓝）**。

输出:

$$o_L = 0.893 \begin{pmatrix} 2.5 \\ 0 \end{pmatrix} + 0.107 \begin{pmatrix} 0 \\ 2.5 \end{pmatrix} = \begin{pmatrix} 2.232 \\ 0.268 \end{pmatrix} \tag{8.10}$$

$$o_R = \begin{pmatrix} 0.268 \\ 2.232 \end{pmatrix} \tag{8.11}$$

左点拿到的是“偏红”的向量，右点拿到的是“偏蓝”的向量。 这正是条件“左边红、右边蓝”该有的样子。

8.6.3 条件 BR：⟨蓝, 红⟩

只把两个词对调，其他一切不变。 两个 key 变成:

$$k_0 = e_{\text{蓝}} + p_0 = \begin{pmatrix} 0.5 \\ 2 \end{pmatrix}, \qquad k_1 = e_{\text{红}} + p_1 = \begin{pmatrix} 2 \\ 0.5 \end{pmatrix} \tag{8.12}$$

左点的分数:

$$q_L \cdot k_0 = 1.5 \times 0.5 + 0.3 \times 2 = 1.35, \qquad q_L \cdot k_1 = 1.5 \times 2 + 0.3 \times 0.5 = 3.15 \tag{8.13}$$

除以 $\sqrt{2}$ 得 0.955 与 2.227，softmax:

$$w_L = \frac{e^{0.955}}{e^{0.955} + e^{2.227}} = \frac{2.60}{2.60 + 9.27} = \begin{pmatrix} 0.219 \\ 0.781 \end{pmatrix} \tag{8.14}$$

注意这个数字。

左点的注意力从“89.3% 看第 0 个词”变成了“78.1% 看第 1 个词”—— **它看的词的下标翻转了，但它看的内容仍然是“红”**。

这就是绑定的全部机制： query 由位置决定，key 由“词 + 位置”决定， 所以“左点该看哪个位置”和“那个位置上是什么词”是两个独立的因素， 模型可以把它们组合起来。

8.6.4 把三个结果并排放

	左点权重	左点输出	右点权重	右点输出
RB	0.893 / 0.107	(2.232, 0.268)	0.107 / 0.893	(0.268, 2.232)
BR	0.219 / 0.781	(1.672, 0.828)	0.781 / 0.219	(0.828, 1.672)
均值池化	—	**(1.000, 1.000)**	—	**(1.000, 1.000)**

最后一行是重点。

均值池化对两种条件给出的是**同一个向量 $(1, 1)$**—— x 方向和 y 方向各占一半，既不说左，也不说右。

两个不同的条件、同一个输出。 模型再聪明也没法把它变成两种结果。

! 手算这一遍，看出了三件事

① **绑定不是“注意力很聪明”，是“下标和内容被分开了”。** 权重管的是下标（第几个词），V 管的是内容（那个词是什么）。两者相乘，才得到“这个位置该拿什么”。

② **位置嵌入必须进 K。** 如果去掉 p_i，k_0 和 k_1 就只剩 $e_{红}$ 和 $e_{蓝}$，**下标信息彻底消失**，权重再也无法“翻转到第 1 个词”。

③ **均值池化丢的不是“精度”，是“下标”这个维度。** 平均之后，第 0 个和第 1 个在数学上不再有区别。

手算到这里，本章剩下的实测都是在验证这三句话。 下一节先看真实训练出来的模型是不是这样。

i 停下来想一想 ①

手算里，位置嵌入加在 K 上。如果改加在 Q 上，会怎样？

提示：Q 来自图像（点的坐标），K 来自文本（第几个词）。

（答案：**加在 Q 上，编码的是“这个点是第几个点”，不是“这个词是第几个词”。**

绑定需要的是后者。把位置加到 Q 上，左点和右点会被标成“第 0 个”“第 1 个”，但两个词在 K 里仍然无法区分——**顺序信息还是没有进入打分。**

位置该加在谁身上，取决于“顺序”属于谁。 词序属于文本，所以加在 K（和 V）。图像的空间位置属于图像，所以加在 Q（或图像特征）。两边都需要时，两边都加——**但那是两个不同的位置概念，不能共用一个嵌入表。**）

8.7 实测：词袋在 RB/BR 上只有 50%

两个模型，唯一的差别是**条件怎么算**：

- **均值池化**：$\text{cond} = \frac{1}{2}(e_{c_L} + e_{c_R})$，67,332 参数
- **交叉注意力**：每个点一个 query，100,740 参数

每类 300 个样本 × 5 个种子，判据是**两个色槽都摆对**的比例：

类	均值池化	交叉注意力
RR	1.000 ± 0.000	**1.000 ± 0.000**
RB	**0.550 ± 0.034**	**1.000 ± 0.000**
BR	**0.449 ± 0.035**	**1.000 ± 0.000**
BB	1.000 ± 0.000	**1.000 ± 0.000**

看中间两行。

均值池化在 RB 和 BR 上分别只有 55.0% 和 44.9%—— **就是抛硬币。**而它在 RR 和 BB 上是完美的 1.000。

为什么？因为 RB 和 BR 的词袋相同：$\frac{1}{2}(e_{红} + e_{蓝})$ 对两者是**同一个向量。**模型收到同样的输入，只能随机选一种摆法。

55.0% 和 44.9% 加起来是 99.9%——它们其实是同一个 50/50 分布的两半。**这不是“稍微差一点”，是“完全没有信息”。**

! 这个失败的形状很特殊

它不是“整体变差”，是“只在需要绑定的类上归零”。

- 纯色类（RR、BB）：词袋唯一确定排列，均值池化够用
- 混色类（RB、BR）：词袋有两个排列，均值池化无解

如果你只看总体准确率，均值池化是 $(1 + 0.55 + 0.45 + 1)/4 = 75\%$—— **看起来“还行”，但它在一个子集上是 0 分。**

8.7.1 训练损失已经给出了答案

在跑任何采样之前，训练曲线里就有一个信号。

模型	参数	最终训练损失
均值池化	67,332	0.0476
交叉注意力	100,740	0.0295

均值池化高出 61%。
但这里有一个混淆：它的参数也少 33%。**有没有可能只是模型太小?**
做一次容量对照：把均值池化的 MLP 从 3 层加到 6 层。

层数	参数	训练损失
3	67,332	0.0476
4	83,844	0.0469
5	100,356	**0.0480**
6	116,868	0.0482

参数从 6.7 万加到 11.7 万，损失几乎没有变化。
在 10.0 万参数这个点上直接对比：均值池化 **0.0480**，交叉注意力 **0.0295**——**差 63%。**

> ! 这个对照排除了“容量不够”
>
> **两个模型参数量相同、数据相同、优化器相同、训练步数相同。**
> **唯一的差别是条件怎么算。**
> **损失差不是“学不会”，是“学不了”** —— 均值池化的条件向量对 RB 和 BR 完全相同，**它要给两个不同的目标输出同一个预测。** 最优解是两者的平均，**而那个平均的误差是下不去的。**
> **这就是信息论意义上的下界**：$H(\varepsilon \mid c_{\text{bag}}) > H(\varepsilon \mid c_{\text{pos}})$。
> **训练损失曲线不会告诉你“哪里错了”，但它会告诉你“有没有错”。**

这个做法值得记住：当你怀疑“模型容量不够”时，**先做一个容量对照，再下结论。如果加参数不降损失，问题在输入，不在模型。** ### 用信息量算一遍：那 63% 是多少比特
两个模型的损失差，可以换算成“条件里少了多少信息”。
假设（本章的数据近似满足）：给定条件 c，噪声 ε 近似高斯，模型学到的是条件均值，于是

$$\text{MSE} = \mathbb{E}[\text{Var}\ (\varepsilon \mid c)] \tag{8.15}$$

对高斯分布，微分熵和方差的关系是

$$H(\varepsilon \mid c) = \frac{1}{2}\log(2\pi e \cdot \text{Var}) \tag{8.16}$$

所以两个模型的条件熵之差：

$$\Delta H = \frac{1}{2}\log\frac{\text{MSE}_{\text{mean}}}{\text{MSE}_{\text{cross}}} = \frac{1}{2}\log\frac{0.0480}{0.0295} = \frac{1}{2}\log 1.627 = 0.243\ \text{nat} \tag{8.17}$$

换成比特：$0.243/\ln 2 = 0.351$ bit。

!这 0.35 bit 是什么

它是“顺序”这个信息，在每一个维度、每一个时间步上的平均含量。
听起来很小？因为这是一个二元信息（左红还是左蓝），理论上最多 1 bit。**0.35 bit 说明模型只榨出了其中的一部分**—— 剩下的被噪声和有限容量吃掉了。
重要的是它不是零。均值池化那条路，**理论上是 0 bit**：条件向量逐位相同，$I(\varepsilon; c) = 0$。
实测的 0.35 bit 来自“条件不同”，而均值池化的条件是**同一个。这就是“信息被销毁”和“信息没学好”的区别。**

这个换算有一个前提：ε 在给定条件下近似高斯。**本章的数据是高斯噪声加在四个点上，这个假设成立得不错。换成真实图像，假设会变差，但“损失差 → 信息差”的方向仍然成立。**
实践含义：当你想知道“某个条件到底带了多少信息”，**不要看它的维度，看它把损失降了多少。维度可以是 768，信息可以是 0.35 bit。**

8.7.2 交叉注意力为什么能解决

因为每个点拿到了不同的条件向量。
左点的 query 来自左点的坐标，右点的 query 来自右点的坐标。**两个 query 不同，注意力权重就不同，得到的条件向量也不同。**
于是：- 对 RB，左点的条件更偏“红”，右点更偏“蓝” - 对 BR，反过来
模型不再需要“从一个向量里猜两种排列”—— 它每个位置都有自己的一份信息。
但这是“交叉注意力怎么做”，不是“绑定必须这么做”。后面的拆零件实验会证明：**把 query 里的坐标抹掉、让两个点拿到完全相同的条件，绑定仍然成立。所以“每个位置一份不同条件”是充分条件，不是必要条件。**

8.7.3 决定性实验：对调两个词

准确率只是间接证据。更直接的做法是：**固定种子、固定初始噪声，只把两个词对调。**
如果模型真的用了顺序，对调之后输出应该翻转；如果没用，输出应该**一模一样。**

编码器	RB（左蓝/右蓝）	BR（左蓝/右蓝）	对调后的最大逐元素差
均值池化	45.5% / 55.0%	45.5% / 55.0%	**0.000**
交叉注意力	0.0% / 100%	100% / 0.0%	**0.972**

!那个 0.000 是这一章最重要的一行

不是“差不多”，是逐位相同——差值的最大值恰好是零。
因为均值池化的条件向量在两种顺序下**是同一个向量**：$\frac{1}{2}(e_{\text{红}} + e_{\text{蓝}}) = \frac{1}{2}(e_{\text{蓝}} + e_{\text{红}})$。
同一批初始噪声 + 同一个条件 + 同一个模型 = 同一组输出。没有任何随机性可以掩盖它。
而交叉注意力下，对调让输出翻转了 0.972—— 两个点的 y 坐标从 $+0.45/-0.45$ 换到了 $-0.45/+0.45$，差值是 0.9，加上噪声正好接近 1。

这个实验比注意力图强在哪：

- **注意力图是内部量**，它的含义依赖你怎么解释
- **对调词是外部干预**，因果方向清楚

这也是本章的评估建议：想验证“文本有没有起作用”，**就改条件、看输出**，不要去看权重。

8.8 拆零件：三个消融

对调实验证明了“交叉注意力用了顺序”。但它没说是哪一部分在起作用。
于是拆掉三个零件，各训一次：

变体	改了什么	参数
`full`	原样	100,740
`no_pos`	K 里去掉 token 的位置嵌入	100,740
`no_xy`	query 里把点的坐标抹成 0	100,740
`mean`	条件取平均（对照）	67,332

`no_pos` 和 `no_xy` 的参数和 `full` 完全一样—— 只改信息，不改容量。**这是消融该有的样子。**

8.8.1 结果：两个意外

变体	RR	RB	BR	BB	对调差
`full`	1.000	1.000	1.000	1.000	**0.972**
`mean`	1.000	0.558	0.442	1.000	**0.000**
`no_pos`	1.000	**0.602**	**0.397**	1.000	**0.000**
`no_xy`	1.000	**1.000**	**1.000**	1.000	**0.931**

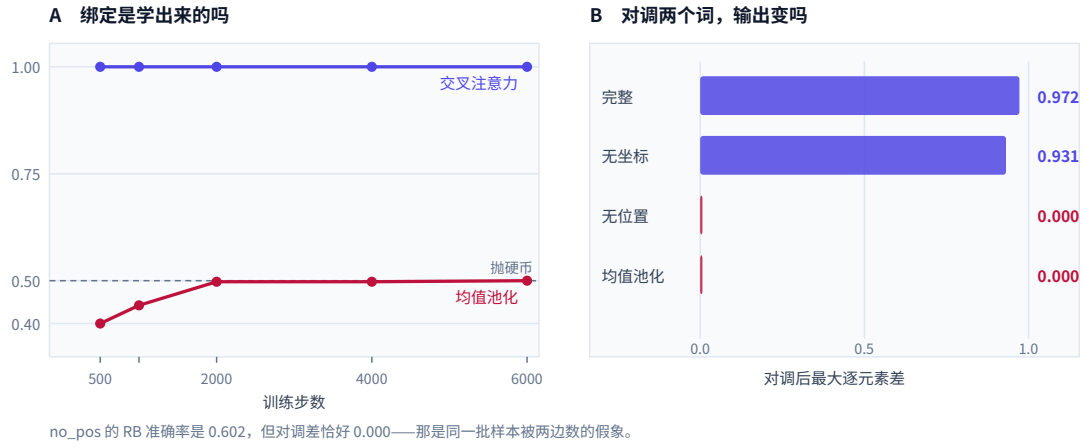


图 8.1 **绑定是学出来的吗。** A：均值池化从第 500 步到 6000 步一直停在 0.5 附近，交叉注意力在第一个检查点就到 1.000。B：对调后的最大逐元素差——只有真正用了顺序的变体才接近 1。数据来自 `data/ch8_ablate2.json`。

图 8.1 把两个维度放在一起：左边是“学得快不快”，右边是“有没有真的用上顺序”。**注意 `no_pos` 在 A 里没有画**——它的准确率曲线看起来像“部分成功”，**但这张图 B 面板里的 0.000 才是真相。**

先看 `no_pos`。 0.602 / 0.397——看起来像“部分绑定”，对吧?

再看对调差：恰好 0.000。

（表里 `mean` 的 0.558 / 0.442 和前面主实验的 0.550 / 0.449 略有不同—— 两次独立训练的抽样噪声，两个版本都满足“和 ≈ 1”。）

8.8.2 第一个意外：0.60 是假的

`no_pos` 的输出，在 RB 和 BR 下逐位相同。

所以那 0.602 和 0.397 是同一批样本被从两边数：某个样本是“左红右蓝”，在 RB 下算对，在 BR 下算错。

两个数加起来 = 0.999。

! 一个不需要跑对调实验的诊断

对镜像的两个类（RB 和 BR），看两个准确率之和：

和	含义
≈ 1.000	**完全没有用顺序**（每个样本只能在一边算对）
≈ 2.000	**完全用上了顺序**
中间	部分用了

这个判据适用于任何“镜像提示词对”：“猫在狗的左边” vs “狗在猫的左边”、“红色方块在蓝色球上” vs “蓝色方块在红色球上”。

它比“看单个准确率”强得多：0.602 单独看像“六成对”，**加上 0.397 才看得出是“零信息”**。

本章之前的 0.550 / 0.449 也是同一回事——加起来 0.999。**那不是“一半对”，是“完全没有顺序信息”**。

为什么 `no_pos` 的输出是逐位相同的？因为注意力对 token 的排列是等变的：把 K 的顺序换一下，加权求和的结果不变。

没有位置嵌入，K 里只剩下词的内容，“红在第一个”和“红在第二个”在数学上无法区分。

8.8.3 第二个意外：`no_xy` 居然完美

`no_xy` 把 query 里的点坐标抹成 0——两个点问出同一个问题。

按理说，两个点应该拿到相同的条件向量，绑定应该失败。

实测：1.000，对调差 0.931。完全正常。

它怎么做到的？

关键：只有当**两个点真的拿到相同向量**时，绑定才会失败。而 `no_xy` 里，两个点的 query 相同、注意力权重相同、输出也相同—— **确实相同**。

但那个相同的向量，在 RB 和 BR 之间是不同的。

因为 K 里**还有位置嵌入**：RB 的 $k_0 = e_{\text{红}} + p_0$，BR 的 $k_0 = e_{\text{蓝}} + p_0$。**同一个 query 对两组不同的 key 打分，得到的加权和不同。**

于是条件 c 携带了“顺序”这个信息，只是它是一份全局的、两个点共用的条件。

那模型怎么知道把红色放在左边？

它看输入 x。 x 的第一个点是 $(-0.6, y_L)$，第二个是 $(+0.6, y_R)$—— **坐标本身就把两个点区分开了**。下游 MLP 拿到 $[x, t, c]$，可以学出：“如果 c 说（红，蓝），就把 $x = -0.6$ 那个点的 y 推向 $+0.45$。”

! 这个结果修正了本章的论点

“条件必须是每个位置各一个向量”——这句话太强了。

实测告诉我们：

- **必要**：条件里必须**带着顺序**
- **不必要**：每个位置必须拿到**不同的**条件

`no_xy` 满足前者、不满足后者，仍然 1.000。

交叉注意力的价值不是“制造不同的向量”，是“让每个位置自己去取”。它把“分配”这件事从下游 MLP 手里，挪到了结构里。

那为什么真实模型还要用交叉注意力？

因为对象一多，“全局条件 + 下游分配”就不行了。

k 个对象有 $k!$ 种排列。全局向量要一次性编码“哪一种排列”，**下游 MLP 要学会 $k!$ 种映射。而交叉注意力把这件事分解成“每个位置各查一次”，k 个对象只需要 k 组对应关系。**

这就是“组合性”：全局条件需要指数级的数据，逐位置条件只需要线性级的。

8.8.4 注意力熵：越“专注”越没用

这一组数据是本章最反直觉的。

在 $t = 50$ 上统计注意力的熵（两个 token 时最大熵是 1 bit）：

变体	平均熵（bit）	左点权重	右点权重	绑定
`full`	0.984	0.579	0.554	✅
`no_xy`	**0.998**	0.525	0.525	✅
`no_pos`	**0.685**	**0.259**	**0.844**	❌

看 `no_xy`：注意力完全均匀（0.525/0.525，熵 0.998），绑定完美。
看 `no_pos`：注意力很尖锐（0.259/0.844，熵 0.685），完全不绑定。
“注意力越集中，模型越’专注’在这个词上”—— 这个直觉在两个方向上都被推翻了。

> ⚠ 为什么尖锐的注意力反而没用
>
> **`no_pos` 的注意力确实很集中：左点 84% 看某个 token。**
> **但它看的是“红”这个内容，不是“第一个位置”。**
> **于是 RB 时它看红，BR 时它还是看红—— 注意力图案一模一样，输出自然一模一样。**
> **权重很尖锐，信息量为零。**
> **这比“权重均匀”更能说明问题：均匀的权重至少还可能是“分散地携带信息”；尖锐的权重可能只是“稳定地携带同一个错误”。**

所以本章的结论要再收紧一格：
注意力权重既不能证明模型在“看对的地方”，也不能证明它在“看错的地方”。唯一能判断的是干预实验：换掉被关注的对象，看输出变不变。

> ℹ 停下来想一想 ②
>
> **前面一直在对调“词”。如果对调的是“两个点”呢？**
> 提示：左点固定在 $x = -0.6$，右点在 $x = +0.6$。
> （答案：**对调两个点，等价于把整幅图左右翻转。**
> 一个训练好的模型应该输出**镜像**的结果：原来的“左红右蓝”变成“左蓝右红”。**这测试的是“位置路由”，不是“词序”。**
> **两种对调要分开做：** - 对调词 → 测“条件里的顺序有没有被用上” - 对调点 → 测“条件有没有被正确分配到位置”
> **本章只做了第一种。** 第二种需要改数据生成，是一个可以接着做的实验。）

8.9 注意力图可信吗

你可能会想：那看看注意力权重，不就知道每个点在听哪个词了吗？
本章试了。答案是：不可信。

8.9.1 权重几乎是均匀的

在 $t = 50$（低噪声、输入接近数据）上，按类统计每个点对两个 token 的平均注意力：

类	左点 → [红, 蓝]	右点 → [红, 蓝]
RR	0.462 / 0.538	0.447 / 0.553
RB	0.583 / 0.417	0.553 / 0.447
BR	0.482 / 0.518	0.436 / 0.564
BB	0.583 / 0.417	0.517 / 0.483

没有任何一行是“左点 0.9 看红、右点 0.9 看蓝”。 所有权重都在 0.44–0.58 之间。

8.9.2 权重也不能预测绑定

更直接的检查：对每个样本，看“左点更关注第一个 token”是否对应“左点是红”。

- **相关系数：−0.079**（几乎为零，甚至是负的）
- **用注意力偏好猜左色的准确率：47.0%**（低于抛硬币）

也就是说：注意力权重里读不出“哪个点该是什么颜色”，但模型的输出却 100% 正确。

> ⚠ 这是一个负结果，而且很重要
>
> **“交叉注意力解决了绑定”和“注意力权重解释了绑定”是两件事。**
> 本章的数据支持前者（1.000 vs 0.550），**不支持后者**（相关 −0.079）。
> **为什么？** 因为信息不只走“权重”这一条路：
>
> - V 里也带着 **位置嵌入**（token 0 和 token 1 的位置不同）
> - query 本身带着**点的坐标**
> - 后面还有一层 MLP，可以把微小的权重差异放大
>
> **权重只是这条链上的一环，而且是可解释性最弱的一环。**
> **第 4 章讲过同一件事**：注意力图能告诉你“模型看了哪里”，但不能告诉你“为什么这么看”。**本章在文本条件上又遇到了一次。**

实践含义：不要用注意力图去调试“为什么文字没生效”。**它可能看起来完全正常，而问题在别处。**
那该用什么？ 用**消融**：把某个 token 换掉、删掉、调换顺序，看输出怎么变。**这是行为层面的证据，不依赖对内部权重的解读。**

8.9.3 如果一定要画注意力图，怎么才算不自欺

“注意力图不可信”不等于“不能画”。它等于“画之前要设一个基线”。
三种常见画法，可信度依次上升：
① **原始权重**（raw attention）：直接把某一层的 $\mathrm{softmax}(QK^\top)$ 画出来。
问题：只反映一层、一个头。**而模型有几十层、几十个头。**
② **注意力 rollout**（Abnar & Zuidema, 2020）：把各层的注意力矩阵乘起来，近似“信息从输入流到这一层的总路径”：

$$\tilde{A} = \prod_{l=1}^{L} (0.5A_l + 0.5I) \tag{8.18}$$

加 $0.5I$ 是因为残差——每一层都有“原样传下去”的那条路。**没有这一项，rollout 会高估注意力的作用。**
③ **梯度加权**（Chefer et al., 2021）：用输出对注意力的梯度给每个头加权，再加权求和。**它回答的是“哪些注意力对输出影响大”，不是“模型看了哪里”。**

> ⚠ 画图的四条底线
>
> ① **必须报基线。**一张“模型看了猫”的热图，如果随机噪声也能画出类似的斑块，它就没有信息。
> ② **必须报层和头。**“第 5 层第 3 头”和“平均所有头”是两回事。**只挑好看的那一张，就是在挑数据。**
> ③ **必须做干预。**把被高亮的 token 换掉，输出变了吗？**没变，这张图就不解释任何事。**
> ④ **不要用注意力图下因果结论。**“模型关注了‘red’，所以它生成了红色”—— **这句话需要的是消融，不是热图。**

本章的实测正好是第 ③ 条的反例：注意力权重几乎均匀（0.44–0.58），用权重猜左色的准确率 47.0%，**低于抛硬币**，而模型的输出准确率是 1.000。
权重和输出之间，隔着一个模型。把权重当成模型的“想法”，是把中间那几十层全忽略了。

8.9.4 那到底该看什么

看行为，不看内部。

想知道	该做的实验	不该做的
文本有没有生效	换词 / 删词 / 对调，看输出差多少	看注意力热图
哪个词重要	逐个 token 消融，排序	看权重大小
顺序有没有被用	对调两个词，看输出是否翻转	看位置编码的范数
空条件对不对	$w = 0$ 时输出是否与提示词无关	看空串向量的范数

这张表的每一行都是“外部干预 → 观察输出”，和本章的对调实验是同一个方法论。

第 4 章讲过同一件事：注意力图能告诉你“模型看了哪里”，不能告诉你“为什么这么看”。**本章在文本条件上又验证了一次。**

8.10 空字符串不是零向量

第 7 章的 CFG 需要一个“无条件”分支。类别模型里，那是一个可学习的空类嵌入。

文本模型里，那是空字符串经过编码器的输出。

8.10.1 空字符串也会被编码

把 `""` 送进一个真实的文本编码器：

1. **tokenizer 会加上特殊 token**——通常是 `[BOS]` 和 `[EOS]`，有时还有 `[PAD]`
2. 这些 token 查表得到向量
3. 经过若干层自注意力
4. 输出一个**非零的、维度正常的向量**

所以“空字符串”不是“没有输入”，是“两个特殊 token 的编码结果”。

这个向量是在训练中见过的——训练时以 10% 的概率把文本换成空串，模型学会了“这个向量表示没有条件”。

> ! 和第 7 章的实验对上了
>
> 第 7 章做过一个对照：同一个模型，把“无条件”分别实现成 **学出来的 NULL token** 和 **零向量**。
>
空条件	$w = 3$	$w = 5$
学出来的 NULL token	越界 0，多样性 0.099	越界 0，0.088
零向量	**越界 100%**	**越界 100%，多样性 0**
>
> **文本模型里，“零向量”就是“自己拼一个全零张量当空条件”**—— 它和“空字符串的编码”不是一回事。
>
> **前者的结果和上表第二行一样：全崩。**

8.10.2 三件必须一起换的东西

空条件的定义依赖三个组件，任何一个变了都要重新验证：

组件	变了会怎样
tokenizer	`""` 切出的特殊 token 变了
文本编码器	同样 token 的编码结果变了
训练时的丢弃率	空条件被训练的程度变了

第 7 章的上线清单里有一条：“换模型/换编码器之后重新验证”。**在文本模型上，这条要再加一个 tokenizer**。

8.11 文本条件下的 CFG

公式一个字都不用改：

$$\hat{\varepsilon} = \varepsilon_{\varnothing} + w(\varepsilon_c - \varepsilon_{\varnothing}) \tag{8.19}$$

其中 $\varepsilon_{\varnothing}$ 是空字符串那一路的输出。

8.11.1 实测：这次的引导几乎没有收益

在第 7 章的二维类别任务上，w 从 1 到 2 能把准确率从 0.965 推到 0.998。**本章的绑定任务上呢?**

w	RB 的双槽准确率
1	1.000
2	1.000
5	1.000
10	0.999

完全没有变化。

> **i** 为什么这次没有收益
>
> **因为 $w = 1$ 时准确率已经是 1.000 了**。
> 第 7 章说过：引导放大的是“条件方向”，**当所有样本都已经落在正确的类里，再放大不会提高准确率**。
> **本章的任务更简单**（四个模式、两个 token），模型在 $w = 1$ 就完全学会了。**引导没有可提升的空间，只有成本**。
> **$w = 10$ 时的 0.999 就是那个成本的第一丝迹象**。

这不是“CFG 在文本条件下失效”——如果换一个更难的提示词（比如需要组合三个属性），$w = 1$ 达不到 1.000，引导应该还有收益。
但本章没有测那个场景。这是一个明确的边界。

8.12 文本条件改变了什么：一个概率的视角

前面都是“怎么算”。这一节退一步问：“文本到底改变了什么分布?”
答案可以用一行贝叶斯写出来，而这一行能解释本章的很多现象。

8.12.1 条件分数 = 无条件分数 + 对齐项的梯度

扩散模型学的不是分布本身，是分数 $\nabla_x \log p_t(x)$（第 5 章）。
加了条件之后，要学的是 $\nabla_x \log p_t(x \mid c)$。由贝叶斯公式：

$$p_t(x \mid c) \propto p_t(x)\, p_t(c \mid x) \tag{8.20}$$

两边取对数再求梯度：

$$\underbrace{\nabla_x \log p_t(x \mid c)}_{\text{条件分数}} = \underbrace{\nabla_x \log p_t(x)}_{\text{无条件分数}} + \underbrace{\nabla_x \log p_t(c \mid x)}_{\text{“这张图有多符合这句话”}} \tag{8.21}$$

! 这一行是本章和上一章的桥

条件扩散 = 无条件扩散 + 一个“图文匹配”项的梯度。
这个“匹配项” $p_t(c \mid x)$ 就是第 7 章那个隐式分类器—— 只不过那里的 c 是一个类别下标，这里的 c 是一整句话。
所以 CFG 的公式

$$\hat{\varepsilon} = \varepsilon_\varnothing + w(\varepsilon_c - \varepsilon_\varnothing) \tag{8.22}$$

做的正是“把匹配项的梯度放大 w 倍”。 公式一个字没变，变的只是 c 的表示。
第 7 章的结论全部适用： w 太大 → 分布被改形；条件太弱 → 匹配项太平 → 放大也没用。

8.12.2 为什么“条件太弱”时 CFG 无效

看匹配项 $\nabla_x \log p_t(c \mid x)$。
如果对所有的 x, $p_t(c \mid x)$ 都差不多—— 也就是“这句话对哪张图都成立”——**它的梯度接近零。**
这时 $\varepsilon_c \approx \varepsilon_\varnothing$， CFG 放大的是零。
什么情况下会这样?

- **提示词太泛**：“a picture”、“high quality”
- **提示词和图像无关**：模型没见过这种配对的训练数据
- **条件被截断**：真正有区分度的词在 77 之后

本章的 w 扫描（1 到 10 基本全是 1.000，$w = 10$ 时 0.999）也是同一个道理的反面：模型在 $w = 1$ 时已经把匹配项学得很准了，放大它没有额外信息。

8.12.3 为什么 caption 质量决定条件能力

$p_t(c \mid x)$ 是模型从训练数据里学出来的。
如果训练数据的 caption 都是“一张图片”， 那么 $p(c \mid x)$ 对任何 x 都相同——梯度为零。
模型学到的就是“文本没有信息”。
反过来，如果 caption 写得具体、有区分度， $p(c \mid x)$ 在 x 上变化剧烈，梯度大，条件就强。
这解释了两件事:
① 为什么“重新标注”能显著提升模型（SDXL、SD3 都做了）—— **它直接改变了 $p(c \mid x)$ 的锐度。**
② 为什么“加更多形容词”不一定有用—— 如果这些形容词在训练数据里和图像不相关， 它们只会让 $p(c \mid x)$ 更平。

8.12.4 一个可以验证的推论

推论：如果在训练时人为地把 caption 和图像打乱配对（随机 caption）， 模型应该完全学不会条件——$p(c \mid x)$ 变成常数。
这是一个“应该成立”的预测，本章没有跑。 但它给出了一个明确的实验： 把 caption 的配对随机打乱，重训，看条件是否失效。 如果还部分有效，说明模型从别的地方（比如图像统计）学到了匹配。

i 这一节的价值

它把“文本条件”从一个工程细节，变成了一个概率对象。
你不再问“交叉注意力该怎么写”, 而是问：“$p(c \mid x)$ 学得准不准? 它在 x 上的梯度大不大?”
前一个问题对应编码器和训练数据， 后一个问题对应提示词的具体程度。
两个问题都比“调 w”更接近根因。

8.13 潜空间：真正发生交叉注意力的地方

本章的“图像”是 4 个数字。真实的图像是 $512 \times 512 \times 3 \approx 78$ 万个数。
直接在上面做扩散太贵。 所以真实模型先把它压到潜空间。

8.13.1 VAE 做了什么

编码器把 $512 \times 512 \times 3$ 压成 $64 \times 64 \times 4$：

$$512 \times 512 \times 3 = 786,432 \quad \rightarrow \quad 64 \times 64 \times 4 = 16,384 \tag{8.23}$$

压缩了 48 倍。 扩散过程在这个 $64 \times 64 \times 4$ 的张量上进行， 采样完再由**解码器**还原成像素。
三个必须记住的数：

量	值	含义
下采样倍数	8	512/64
潜变量通道	4	而不是 3
scaling factor	≈ 0.18215	编码后要乘它

scaling factor 不是超参数，是 VAE 训练时定下的常数。用错它，扩散模型看到的就是一个尺度不对的分布。

8.13.2 交叉注意力在潜空间里算什么

潜变量的每个位置（$64 \times 64 = 4096$ 个）都是一个 query。
文本是 77 个 token（CLIP 的上限），所以：

$$\text{注意力矩阵} = 4096 \times 77 \tag{8.24}$$

约 31 万个元素，每个头一份。
这个规模是可以接受的，但也是为什么交叉注意力只放在 UNet 的 **若干层**而不是每一层——**它是注意力里最贵的部分。**

> **i** “每个位置”到底指什么
>
> **不是像素，是潜变量的一个格子。**
> 一个格子对应原图的 8×8 区域。**所以交叉注意力的分辨率是“8 像素一格”。**
> **这解释了一个常见现象**：文字能控制“左边有一只猫”， 但很难精确控制“猫的左耳”——**因为耳朵小于一个格子。**

> **i** 停下来想一想 ③
>
> **VAE 的潜变量为什么是 4 通道，而不是 3?**
> 提示：3 通道正好对应 RGB。
> （答案：**4 不是“多存了一点颜色”，是“给结构留了一维”。**
> 3 通道会强迫潜空间同时编码“颜色”和“结构”， 两者纠缠在一起，解码时容易互相干扰。
> **多一维之后，网络可以学出“一维管结构、三维管颜色”这样的分工—— 但这是学出来的，不是设计出来的。** 4 这个数是实验选出来的。
> **更好的答案可能是“没有理论原因”。** 4 通道在重建质量和扩散难度之间取得了平衡， 换成 3 或 8 都有人试过，效果不同但没有一条公式告诉你该选几。）

8.13.3 为什么一定要压缩

因为注意力的成本是平方的。

如果直接在 512×512 的像素上做自注意力，序列长度是 $512 \times 512 = 262,144$——**注意力矩阵是 2.6×10^{10} 个元素。任何显卡都放不下。**

压到 $64 \times 64 = 4096$ 之后，矩阵是 1.7×10^{7}——**可行，但仍然很大。**

所以潜空间不是“为了省显存”的小优化，是“能不能跑”的分界线。

> **i** 一个容易算错的数
>
> **交叉注意力的成本是 $N \times M$，不是 N^2。**
>
> - N = 图像位置数（4096）
> - M = 文本 token 数（77）
>
> **$4096 \times 77 \approx 31.5$ 万**，比自注意力的 $4096^2 \approx 1678$ 万小 53 倍。
>
> **这就是为什么交叉注意力可以放在很多层，而自注意力不行。**

8.13.4 VAE 的两个方向

```
图像 (512×512×3)
   ↓ 编码器（卷积 + 下采样 3 次）
潜变量 (64×64×4)
   ↓ × 0.18215          ← scaling factor
扩散模型的输入
   ...
   ↓ 采样完成
潜变量 (64×64×4)
   ↓ ÷ 0.18215
   ↓ 解码器（上采样 3 次）
图像 (512×512×3)
```

三次下采样，每次减半：$512 \to 256 \to 128 \to 64$。所以一个潜变量格子对应 8×8 的像素块。

`scaling factor` 的作用是把潜变量的标准差拉到 1 附近，让扩散模型的噪声调度（β 从 10^{-4} 到 0.02）处在合适的范围。**它是在 VAE 训练完之后、在潜变量上统计出来的一个常数，不是超参数。**

> **⚠** 用错 scaling factor 会怎样
>
> **模型看到的输入尺度不对，等价于换了一个噪声调度。**
>
> 症状：**生成结果整体偏灰或过饱和，而且怎么调 w 都没用。**
>
> **判据**：编码一张真实图片，看潜变量的标准差。**它应该接近 1。**偏离一个数量级，就是 scaling factor 用错了。

8.14 动手实现：两种编码器的代码

本章的两种编码器各只有十几行。看懂它们，就看懂了真实模型的骨架。

8.14.1 均值池化

```python
class MeanPoolEncoder(nn.Module):
    def __init__(self, hidden):
        super().__init__()
        self.emb = nn.Embedding(3, hidden)      # 0=红 1=蓝 2=NULL

    def forward(self, tok, x=None, t=None):
        return self.emb(tok).mean(dim=1)        # (N, d)
```

`mean(dim=1)` 就是全部问题所在。它把“顺序”这个维度平均掉了。

8.14.2 交叉注意力

```
def forward(self, tok, x, t):
    pts = torch.stack([x[:, 0:2], x[:, 2:4]], dim=1)      # (N,2,2)
    q_in = torch.cat([pts, t[:, None, :].expand(N, 2, -1)], dim=-1)
    q = self.q(q_in).view(N, 2, self.n_head, self.dh)
    k = (self.emb(tok) + self.pos(...)).view(N, 2, self.n_head, self.dh)
    att = torch.einsum("nphe,nkhe->nhpk", q, k) / self.scale
    att = att.softmax(dim=-1)
    out = torch.einsum("nhpk,nhkd->nphd", att, k.permute(0, 2, 1, 3))
    return out.reshape(N, -1)
```

四行，对应注意力公式的四步：

1. **query 从点和时间步来**（`q_in`）
2. **key 从 token 嵌入 + 位置来**（`k`）
3. **相似度 → softmax**（`att`）
4. **加权求和**（`out`）

注意 `self.pos`。 token 的位置嵌入是绑定能工作的另一半—— **没有它，两个 token 在 K 里是对称的，模型仍然分不清“第一个词”和“第二个词”**。

> ⚠ 一个容易漏掉的细节
>
> **`q` 的输入里有 `t`（时间步嵌入）**。
> 为什么？因为**在不同噪声水平上，“该听哪个词”可能不一样**： 高噪声时结构还没定，低噪声时细节已经清楚。 **让 query 知道自己在哪个阶段，注意力才能随阶段变化**。
> **真实 UNet 里，query 来自图像特征本身**（已经包含了时间信息）， 所以不显式拼接——但作用是同一个。

8.14.3 接进 UNet 的位置

```
x_t → [Down Block] → [Mid Block + CrossAttn] → [Up Block + CrossAttn] → ε
                            ↑                              ↑
                       text tokens                    text tokens
```

交叉注意力通常插在 Mid 和 Up 的若干层, 不在 Down 的最浅层—— 因为那里分辨率太高, 4096×77 会变成 16384×77。
这是一个纯粹的成本决定，不是理论决定。

8.15 交互实验 8-1 绑定实验室



图 8.2 **实验 8-1 绑定实验室**（静态快照，均值池化 + RB）。网页版可切换编码器与四个类。 本地路径：`experiments/exp-8-1.html`

怎么用。 每个点是一个生成样本，画在“左点 y × 右点 y”的平面上。

- 横轴 > 0：左点是红；< 0：左点是蓝
- 纵轴 > 0：右点是红；< 0：右点是蓝
- **四个象限就是四个类**

观察引导。

1. 选 **RB**，编码器选**均值池化**。 **样本同时在右下（RB）和左上（BR）两个象限**，各占一半。
2. 切到**交叉注意力**。样本**只落在右下**。
3. 回到均值池化，切到 **BR**。 **图案几乎一模一样**——因为这两个类拿到的条件向量完全相同。
4. 切到 **RR** 或 **BB**。两种编码器都是 100%。 **词袋唯一确定排列时，绑定不是问题**。

> ! 第 3 步是这一节的全部意义
>
> **同一个词袋，两种顺序，模型给出同一个分布。**
> **这不是“生成质量差”，是“条件里没有信息”。** 模型被要求从同一个输入产生两种输出，它只能二选一、或者各出一半。
> **判据**：两个“应该不同”的类，在同一个编码器下呈现出**镜像对称**的两团。 **看到这个图案，就是绑定失败。**

8.16 它在哪里失效

按第 6、7 章的格式：**材料** → **步骤** → **预期** → **实测** → **判据** → **修法**。

8.16.1 失败一：用均值池化处理有顺序的条件

> **材料**：本章的两个模型。
> **步骤**：对 RB 和 BR 各采 300 个样本 × 5 个种子，统计“两个色槽都摆对”的比例。

预期：均值池化“稍微差一点”，交叉注意力“更好”。
实测：

模型	RB	BR
均值池化	**0.550 ± 0.034**	**0.449 ± 0.035**
交叉注意力	1.000	1.000

判据：**两个“应该不同”的类，在某个模型上得到接近 50/50 的结果**，而且这两个数加起来约等于 1。**这是“条件没有区分度”的指纹**，不是“质量差”。
修法：换成逐位置的条件（交叉注意力），或者至少保留位置信息（拼接位置嵌入后不做平均）。

8.16.2 失败二：有交叉注意力，但忘了位置嵌入

材料：把 `self.pos` 从 K 里去掉（`variant="no_pos"`），参数一个不变。
步骤：重训，重复失败一的测量，再加一次对调实验。
预期：既然有注意力，应该能区分顺序。
实测：

指标	数值
RB 准确率	0.602
BR 准确率	0.397
两者之和	**0.999**
对调后的最大逐元素差	**0.000**

判据：**看两个镜像类准确率之和**。≈ 1 就是“完全没用顺序”——每个样本只能在一边算对。
修法：位置嵌入（或任何能区分 token 位置的信号）必须进 K。**这是“注意力”和“顺序”之间的桥**。

⚠ 这个 0.602 差点骗过我

第一眼看到 0.602 / 0.397，我以为是“部分绑定”。
但对调实验给出 0.000——输出逐位相同。
0.602 是“同一批样本被两边数”的结果：一个“左红右蓝”的样本，在 RB 下算对、在 BR 下算错。**两个数加起来正好 0.999。**
如果只测准确率，会得出“位置嵌入有帮助但不是必需”的错误结论。是干预实验把它纠正过来的。

⚠ 失败二为什么值得单列

很多人以为“用了注意力就自动有顺序”。
不是。注意力本身是**置换等变**的：把 K 的顺序打乱，输出只是跟着打乱，不会“知道”顺序变了。
顺序信息必须显式注入——位置嵌入、因果掩码、或者位置相关的 query。**Transformer 里的位置编码，和这里是同一个理由。**

8.16.3 失败三：空字符串用零向量

材料：第 7 章实验四的结论，迁移到文本条件。
步骤：把无条件分支实现成全零张量（而不是空串编码）。
预期：w 小时可能还行。
实测（第 7 章）：$w = 3$ 时越界 100%，$w = 5$ 时多样性归零。**准确率仍然 1.000**。
判据：$w = 0$ 时输出应该和提示词无关。**如果 $w = 0$ 仍然“听话”，或者崩坏**，说明无条件分支不对。
修法：用编码器对空串的输出，不要自己拼零向量。

8.16.4 失败四：换了 tokenizer 没有复验

材料：任何换过 tokenizer 或文本编码器的模型。
步骤：$w = 0$ 出 10 张，$w = 1$ 出 10 张，$w = 3$ 出 10 张。
预期：和第 7 章验过的行为一致。
实测（本章未做跨编码器实验）：**空字符串切出的特殊 token 会变，编码结果也会变**，所以“无条件”的语义变了。
判据：$w = 0$ 的输出分布和换之前**明显不同**。
修法：把“空条件向量”存下来，换组件时对比。**如果两个向量的余弦相似度很低，就必须重新定标 w**。

8.16.5 失败五：提示词超长被截断

材料：CLIP 的 77 token 上限。
步骤：写一段 100 个 token 的提示词。
预期：模型“读到全部”。
实测（真实模型上的已知行为，本章未复现）：**77 之后的 token 被丢掉**，而且丢掉的往往是句尾的关键修饰语。
判据：把提示词从 60 token 加到 90 token，输出**没有变化**。
修法：把最重要的词放在前面；或者用支持长上下文的编码器（T5、Qwen）。

8.16.6 失败六：用注意力图调试

材料：本章的交叉注意力模型。
步骤：看注意力权重，判断“左点在听哪个词”。
预期：左点应该主要看第一个 token。
实测：权重全在 0.44–0.58 之间；**用权重猜左色的准确率 47.0%**，低于抛硬币。
判据：注意力图和“输出是否正确”**对不上**。
修法：改用消融——换词、删词、调换顺序，看输出怎么变。**行为层面的证据，不依赖对权重的解读**。

8.17 同一个注意力，三种用法

第 1 章的自注意力、本章的交叉注意力、SD3 的联合注意力，公式是同一个：

$$\mathrm{Attn}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V \tag{8.25}$$

区别只在 Q, K, V 从哪来。

	自注意力（第 1 章）	交叉注意力（本章）	联合注意力（MMDiT）
Q 来自	同一序列	图像	图像 + 文本
K, V 来自	同一序列	文本	图像 + 文本
序列长度	N	N 与 M	$N + M$
矩阵大小	$N \times N$	$N \times M$	$(N + M)^2$
谁被更新	全部	**只有图像**	全部
典型位置	每个 Block	UNet 的若干层	每个 Block

三行值得展开。

8.17.1 第一行：谁被更新

自注意力里，每个 token 都更新自己。交叉注意力里，只有图像被更新——文本 token 是只读的。

这是一个不对称的设计。文本提供信息，但不接收信息。**所以文本的表示在整个 UNet 里保持不变**，每一层看到的都是同一份 K, V。

好处：可以缓存。**77 个 token 的 K, V 算一次，所有层复用。坏处**：文本不能根据图像调整自己——“这只猫”没法在生成过程中变成“这只橘猫”。

联合注意力取消了这个不对称：文本也会被图像更新。**代价是文本表示不再可缓存**，每一层都要重算。

8.17.2 第二行：矩阵大小

$N \times M$ 和 $(N + M)^2$ 的差别，在 $M \ll N$ 时很小。

前面算过：$N = 4096$、$M = 154$ 时，联合注意力只比“自注意力 + 交叉注意力”多 3.8%。

但如果 M 接近 N，差别就大了。比如图像编辑任务里，参考图也被编码成 4096 个 token——**这时的联合注意力是 $(8192)^2$，是原来的 4 倍**。

这就是为什么“参考图 + 文本”的多图条件模型特别吃显存。

8.17.3 第三行：掩码

三种用法都能加掩码，但掩码的含义不同。

用法	常见掩码	作用
自注意力（语言模型）	因果掩码	不能看未来
自注意力（图像）	无	全图互相看
交叉注意力	区域掩码	某块区域只能看某些词
联合注意力	模态掩码	文本不能看图像（可选）

因果掩码是“结构性的”（第 1 章的 CLIP 文本编码器用它），**区域掩码是“任务性的”**（本章“空间条件”一节）。

同一个 `masked_fill(-inf)`，两种完全不同的用意。

!一个统一的看法

三种注意力都是“用 Q 去 K 里找 V”。

- **自注意力**：在**自己**里面找
- **交叉注意力**：去**别人**那里找
- **联合注意力**：把自己和别人拼在一起找

所以“注意力机制”不是一种结构，是一个查询接口。换的永远是“查什么”和“谁能查”，不是公式。

看懂这一点，你拿到任何新架构（DiT、MMDiT、U-ViT、VAR）都能在三十秒内定位它的条件路径。

8.18 条件注入的五代做法

“文字怎么进模型”不是一个问题，是五个。每一代做法解决前一代的一个具体缺陷，同时也带来一个新的代价。**看懂这张谱系，你拿到任何一个新模型都能立刻定位它在哪一格。**

代	做法	条件粒度	每层参数（$d=320$, 文本 768）	代表
①	拼接（concat）	全局	348,160	早期条件 GAN、class-conditional diffusion
②	相加 / AdaGN	全局	245,760 / 491,520	类别条件、DiT 的 adaLN
③	**交叉注意力**	**逐位置**	**696,320**	SD 1.x / 2.x / XL
④	联合注意力（MMDiT）	逐位置 + 双向	见下	SD3、FLUX
⑤	旁路注入	结构 / 参考图	另算	ControlNet、IP-Adapter

（① 的 348,160 = $(320+768)\times 320$，因为它要把拼接后的 1088 维投回 320 维；② 的 245,760 = 768×320，只是一次投影。）

这张表最容易误读的是“每层参数”那一列。③ 比 ② 只贵 40%，为什么大家说交叉注意力“贵”？**因为参数不是成本，计算才是。**② 每层做一次 d 维加法，③ 每层要做 $N\times M$ 次点积——$N=4096$、$M=77$ 时是 31 万次。**贵的是那 31 万次，不是那 69 万个参数。**

8.18.1 ① 拼接：最早，也最笨

做法：把条件向量复制到每个空间位置，拼在通道维上，再过一层卷积压回去。

```
特征 (B, 320, 64, 64)  ⊕  文本 (B, 768) 广播到 (B, 768, 64, 64)
        ↓ 拼接
     (B, 1088, 64, 64)
        ↓ 1×1 卷积
     (B, 320, 64, 64)
```

问题：条件向量是**全局**的，每个位置拿到的完全一样。**它能说“这是一只猫”，不能说“猫在左边”。**

这一代没有消失——今天你仍能在一些轻量模型里看到它，因为它最便宜。

8.18.2 ② 相加与 AdaGN：让条件调制特征

相加：$h \leftarrow h + Wc$。零参数以外的开销，但表达力也最弱。

AdaGN（自适应组归一化）：先把特征归一化，再用条件算出**逐通道的缩放和平移**：

$$\mathrm{AdaGN}(h, c) = \gamma(c) \cdot \frac{h - \mu}{\sigma} + \beta(c) \tag{8.26}$$

它比相加强在哪：条件不再只是“加一个偏移”，而是**乘性调制**——可以放大某些通道、压掉另一些。**这相当于让条件控制“哪些特征重要”，而不只是“往特征里掺一点东西”。**

DiT 把这一招用到了极致：每个 Block 的 LayerNorm 后面接一个 $\mathrm{adaLN}(c)$，一次算出 6 组参数（注意力和 MLP 各三组：shift、scale、gate）。

算一笔账（SD3 Medium，$d = 1536$，池化文本 2048 维）：

$$2048 \times (6 \times 1536) = 2048 \times 9216 = 18,874,368 \tag{8.27}$$

每个 Block 一千九百万参数，24 个 Block 就是 4.5 亿。这比很多人以为的“AdaGN 很便宜”要贵得多——因为 DiT 的 d 大、而且要出 6 组。

> ⚠ “全局条件”的天花板
>
> ①② 两代的共同点是：**整张图共用同一个条件向量。**
> **它们能表达“是什么”，不能表达“在哪里”。**本章的 RB/BR 实验就是这条天花板的直接证明：条件向量相同 → 输出只能二选一。
> **这不是模型不够大，是信息在入口就没了。**

8.18.3 ③ 交叉注意力：每个位置自己问

这是本章的主角，也是 Stable Diffusion 1.x/2.x/XL 的做法。

它解决 ①② 的方式不是“更强的调制”，是“取消全局”—— 不再有一个“整图条件向量”，而是每个位置现算一个。

代价刚才算过：每层 $N \times M$ 次点积。

一个常被忽略的事实：交叉注意力的参数量对 M 不敏感。K、V 的投影是 $d \times d_{\text{ctx}}$，和 token 数无关。**所以 77 个 token 和 7 个 token，参数一样多，计算差 11 倍。**

8.18.4 ④ 联合注意力：把两串 token 拼成一条

SD3 的 MMDiT 做了一件很反直觉的事：它取消了交叉注意力。

做法：把文本 token 和图像 token **拼成同一个序列**，做**一次**自注意力：

```
[文 1 文 2 … 文 77 | 图 1 图 2 … 图 4096]   ← 一条序列，4250 个 token
                    ↓ 一次自注意力（每个 token 看所有 token）
                    ↓ 各自过自己的 FFN
[文 1' … | 图 1' …]
```

“双向”是关键词：图像能看文本，**文本也能看图像**。交叉注意力里文本是只读的（Q 只来自图像），联合注意力里文本会被图像更新。

为什么这可行？算一下成本。

设 $N = 4096$（图像 token）、$M = 154$（CLIP 77 + T5 77）。

方案	注意力元素数
自注意力 + 交叉注意力（SDXL 式）	$N^2 + NM = 16,777,216 + 630,784 = 17,408,000$
联合注意力（MMDiT 式）	$(N+M)^2 = 4250^2 = 18,062,500$

只多了 3.8%。

! 这 3.8% 是这一节最值得记住的数

很多人以为“联合注意力是 4250^2，太贵了”。
但基线不是 N^2，是 $N^2 + NM$——因为原来的模型**本来就要做自注意力**。把两条注意力合成一条，省掉了一次独立的 $QK^\top$，代价只是多出来的 M^2 项（$154^2 = 23,716$，占 0.1%）。
所以 MMDiT 不是“更贵的架构”，是“用同样的预算买到了双向信息流”。
这也是为什么 SD3 之后的模型几乎都换成了联合注意力——它不是奢侈，是划算。

代价在别处：文本 token 现在也要过 FFN，也要占显存；而且**文本表示会被图像“污染”**——第 30 层之后，文本 token 里装的不再只是语义。**这对可解释性是个坏消息**（本章的注意力图问题在这里更严重）。

8.18.5 ⑤ 旁路注入：不改主干，加一条支路

ControlNet 和 IP-Adapter 解决的不是“文字怎么进”，是“别的东西怎么进”。
ControlNet（2023）：复制一份 UNet 的编码器，**原模型冻结**，副本接收额外条件（边缘图、深度图、姿态），通过**零初始化卷积**把副本的输出加到原模型上。

$$h_{\text{out}} = h_{\text{orig}} + \mathcal{Z}(h_{\text{copy}}) \tag{8.28}$$

零卷积的意义：训练开始时 $\mathcal{Z} \equiv 0$，输出和原模型**逐位相同**。**这保证了“加条件”不会破坏已经学好的生成能力**——从一个“什么都不加”的模型开始，慢慢学会加。
IP-Adapter（2023）：**解耦交叉注意力**。在原来的文本交叉注意力旁边，再挂一个图像交叉注意力：

$$\text{out} = \text{Attn}(Q, K_{\text{text}}, V_{\text{text}}) + \lambda \cdot \text{Attn}(Q, K_{\text{img}}, V_{\text{img}}) \tag{8.29}$$

文本那一路冻结，只训图像那一路，λ 控制参考图的强度。

i 为什么这两种方法能“即插即用”

因为它们都不改主干的权重。

- ControlNet：主干冻结，只训副本
- IP-Adapter：主干冻结，只训新增的 K, V 投影

代价是参数量：ControlNet 的副本约等于半个 UNet；IP-Adapter 只有几十兆，因为它只加了两组投影。
这也是判断一个“新条件”该用哪种注入的实用准则：要改的是“结构”（边缘、深度、姿态）→ ControlNet；要改的是“内容/风格”（参考图）→ IP-Adapter；要改的是“语义”（文字）→ 交叉注意力或联合注意力。

这张谱系到这里就完整了。本章剩下的内容，都是在 ③ 这一格上做深——因为它是“文字控制图像”最直接、也最容易讲清楚的一种。

8.19 现场排错：十个症状

失败模式那一节讲的是“机制上会怎么坏”。这一节讲“坏了怎么查”。
每一条都按同一个格式：症状 → 先做一个三十秒的检验 → 它排除了什么。

8.19.1 症状一：改了提示词，图完全没变

三十秒检验：固定种子，把提示词整个换掉（不是改一个词），出两张。
如果还是没变，问题不在“某个词没生效”，在**文本根本没进模型**。
按可能性排序：

1. **条件传错了位置**——`encoder_hidden_states` 传成了空
2. **CFG 的 $w = 0$**——两个分支被加权成“只用无条件”

3. **文本缓存没刷新**——改了提示词但复用了旧的 embedding
4. **模型是纯无条件模型**——根本没有文本输入

判据：$w = 0$ 和 $w = 1$ 的输出应该不同。**相同，就是条件没接上**。

8.19.2 症状二：w 越大，图越“焦”

症状：颜色过饱和、边缘发硬、构图趋向居中、多样性下降。
三十秒检验：w 从 1 扫到 10，看多样性（同样提示词不同种子的差异）怎么变。
这是过引导的典型曲线：准确率上升→平台→下降，多样性单调下降。
修法：降 w；或者用第 7 章的 rescale；或者用 w 随步数递减的调度。
不要做的：继续加 w 试图“让提示词更听话”。 **过了平台期，加 w 只会更糟**。

8.19.3 症状三：颜色整体偏灰或偏亮

三十秒检验：编码一张真实图片，看潜变量的标准差。**应该接近 1**。
偏离一个数量级 → scaling factor 用错了，或者根本没除。
另一个可能：`init_noise_sigma` 用错。 **两者都会表现为“整体色调不对，且调 w 无效”**。

8.19.4 症状四：提示词里明明写了，图里没有

三十秒检验：把这个词删掉，看输出变不变。
不变 → 这个词没被用起来。三个原因：

原因	怎么确认
被截断（超过 77）	把它挪到句首，看是否出现
tokenizer 切碎了	打印 token 序列，看它变成几个 token
训练数据里这个概念很少	换一个同义但更常见的词

第二种最隐蔽：一个生僻词被切成 5 个 token， **每个 token 单独看都没有意义**，模型自然拼不出这个概念。

8.19.5 症状五：两个物体的颜色串了

这是本章的属性泄漏。
三十秒检验：把提示词改成两个单独的句子，分两次生成，再用图像编辑合成。**如果分开生成是对的，合起来是错的**， 就是交叉注意力里的竞争，不是模型不认识颜色。
修法：区域提示词、结构化引导、或者分步生成。

8.19.6 症状六：位置总在画面中间

三十秒检验：在提示词里加“on the far left”，看物体是否移动。
移动了一点但不到位 → 文本对位置的控制是统计性的（本章“空间条件”一节）。 **完全不动** → 模型的空间概念很弱，需要外部条件（ControlNet / 区域掩码）。

8.19.7 症状七：换了一个模型，同一个提示词效果差很多

这不是 bug。 两个模型的 caption 分布不同（本章“描述文字”一节）。
三十秒检验：把提示词改写成新模型的风格（关键词 vs 自然语言），再试。
如果改写法之后恢复了，说明是分布匹配问题，不是模型能力问题。

8.19.8 症状八：中文提示词效果明显差于英文

三个叠加的原因：

1. **tokenizer 对中文效率低**——同样一句话占更多 token
2. **训练数据以英文为主**——中文描述在 caption 分布里很少

3. **文本编码器的中文语义弱**——CLIP 的中文能力来自翻译数据，不是原生

三十秒检验：把同一句中文机翻成英文，对比输出。
如果英文明显更好，三个原因都可能；如果差不多，只是 tokenizer 的问题。

8.19.9 症状九：负面提示词把主体也删了

机制：负面提示词参与的是 ε_{neg} 那一路， 它和正面提示词**共享同一个 softmax 空间**。
如果负面词和主体在语义上接近（“不要模糊的猫”里的“猫”）， **它会把主体一起推走**。
三十秒检验：把负面提示词逐条删掉，看哪一条导致主体消失。
修法：负面词只写“属性”，不写“主体”。

8.19.10 症状十：同样的参数，两次结果不一样

按可能性排序：

1. **种子没固定**（最常见）
2. **dtype 不同**（fp16 vs bf16 的舍入不同）
3. **调度器的随机性**（有些采样器每步加噪声）
4. **批处理顺序影响了 RNG**（一次出 4 张和分 4 次出，噪声序列不同）

> 排错的一般顺序
>
> **先排除“没接上”，再排除“接错了”，最后才怀疑“模型不行”。**
>
> 1. **条件在不在？**（症状一）
> 2. **尺度对不对？**（症状三）
> 3. **有没有被截断/切碎？**（症状四）
> 4. **是不是分布不匹配？**（症状七、八）
> 5. **才是机制问题**（症状五、六）
>
> **大部分“模型不听话”的报告，停在前两步。**

8.20 设计你自己的条件接口：六个问题

假设你要给一个扩散模型加一种新条件——不是文字，可能是点击、骨架、音频、布局。
别急着写代码。先回答六个问题。 每一个问题都会砍掉一半的设计空间。

8.20.1 问题一：这个条件是“全局的”还是“逐位置的”？

“这张图是一只猫”是全局的。“猫在左上角”是逐位置的。
全局条件用 AdaGN 或相加就够了——**每层一次 d 维调制，便宜。 逐位置条件**必须用交叉注意力或联合注意力——**每层 $N \times M$，贵**。
但要注意：全局条件不等于“弱条件”。 **它只是不能表达“哪里”**。如果任务不需要位置，别为它付交叉注意力的钱。

8.20.2 问题二：条件是一串还是一个？

一个向量（类别、池化文本、音频片段）→ 拼接 / AdaGN。 **一串向量**（文本 token、骨架关键点、布局框）→ 交叉注意力。
中间情况：**固定长度的一小组向量**（比如 10 个关键点）。**可以拼平成一个大向量**，也可以用交叉注意力。 **拼平的代价是模型要自己学会“第 3 个点和第 4 个点不该混”—— 这正是本章讲的绑定问题。**

8.20.3 问题三：需不需要双向？

文本 → 图像是单向的：文本不变，图像被更新。

如果你的条件也需要“看图像”（比如“根据当前画面决定下一步画什么”），**就需要双向**——联合注意力，或者让条件也做一次自注意力。

双向的代价：条件表示每一层都变，**不能缓存**；而且**条件会被图像“污染”**，调试时更难判断“是条件错了还是模型错了”。

8.20.4 问题四：条件在采样过程中会变吗？

文字不变：同一个 K, V 用 1000 次。

有些条件会变：- 视频生成里的“上一帧” - 编辑任务里的“原图” - 强化学习里的“当前状态”

会变的条件下，缓存失效，每一步都要重算。**这是视频模型比图像模型贵得多的一个原因**—— 不只是帧数多，是条件也不能复用。

8.20.5 问题五：训练时怎么“丢弃”这个条件？

CFG 需要一个无条件分支。所以训练时必须随机丢弃条件。

丢弃率不是一个无关紧要的超参数：

- **太低（1%）**：模型很少见到无条件，$\varepsilon_\varnothing$ 学不好 → w 一大就崩
- **太高（50%）**：一半的训练样本浪费在“无条件”上 → 条件能力弱
- **常用值 10%**：经验平衡点

而且丢弃必须是“整段丢弃”，不能丢一半留一半—— **否则模型学到的“无条件”是“部分条件”，和推理时用的空条件不匹配。**

8.20.6 问题六：多种条件怎么组合？

真实系统常常同时有文字、参考图、姿态、区域掩码。

三种组合方式：

方式	做法	问题
串联	条件 A 的输出当条件 B 的输入	误差累积
并联相加	$\sum_i \lambda_i c_i$	尺度难调，条件之间会抵消
解耦注意力	每种条件一组 K, V，输出相加	参数多，但可控

IP-Adapter 用的是第三种（本章“条件注入”一节）。**它最好调**：每个 λ_i 独立，关掉一个不影响另一个。

> ! 六个问题的顺序不能换
>
> **先问“全局还是逐位置”**——它决定了架构。**再问“一串还是一个”**——它决定了张量形状。**最后才问丢弃率和组合**——它们是训练和推理的细节。
>
> **很多人从“用什么注意力变体”开始设计，这是反的。先确定信息要表达什么，再选结构。**

8.21 文字说不清的地方：空间条件

本章一直在讲“左边红、右边蓝”。真实提示词里写“左边一只猫，右边一只狗”会怎样？

大概率两只都在中间，或者位置随机。

这不是模型不听话，是文字这个接口本身不擅长表达坐标。

8.21.1 三个结构性的原因

① **文本里没有坐标。**“左边”是一个**相对词**，它的所指依赖画布。模型必须从训练数据里学会“左边 = 图像的左半部分”，而这个映射在数据里是**统计性的**，不是硬约束。

② **交叉注意力是软的。**每个潜位置对 77 个 token 做加权平均。即使“猫”这个 token 的权重在左侧更高，**它也不会把其它位置清零。softmax 永远给出一个概率分布，不给“只在这里”。**

③ **分辨率不匹配。** 一个潜格子对应 8×8 像素。**“猫的左耳”小于一个格子，** 文字再精确也落不到那个尺度上。

> ! 这不是“提示词写得不够好”
>
> **很多人以为加更多形容词就能控制位置。**
> **不会。**因为问题不在“文字的含义”，在“接口的带宽”—— **77 个 token 到 4096 个位置，是一条多对多的软连接，没有坐标通道。**
> **要精确控制位置，必须加一条新的通道。**

8.21.2 做法一：区域提示词（regional prompting）

思路：把画布切成几块，每块用一段独立的提示词，**在交叉注意力里加掩码。**
设区域 r 的掩码是 $M_r \in \{0,1\}^N$（哪些潜位置属于这个区域），则区域 r 的交叉注意力改成：

$$a_{i,k}^{(r)} = \frac{M_{r,i} \exp(q_i^\top k_k / \sqrt{d})}{\sum_j M_{r,j} \exp(q_i^\top k_j / \sqrt{d})} \tag{8.30}$$

$M_{r,i} = 0$ 的位置，这个区域的词根本进不来。
实现上就是在 softmax 之前把 logits 加上 $\log M$（0 变 $-\infty$）。
代价：

问题	原因
接缝	硬掩码在边界处不连续
区域内的词仍然软	区域内还是加权平均
多区域冲突	重叠区域的归一化要重新设计

所以实践里常用“软掩码”：把 0/1 换成高斯权重，边界羽化几个格子。

8.21.3 做法二：把坐标变成 token（GLIGEN）

GLIGEN（2023）的做法更彻底：**给每个框一个 token。**

```
[猫][框 x1,y1,x2,y2] [狗][框 x1,y1,x2,y2] ...
```

框坐标经过傅里叶编码变成一个向量，和词嵌入拼在一起，再通过一个**门控交叉注意力**注入 UNet：

$$h \leftarrow h + \tanh(\gamma) \cdot \mathrm{Attn}(h, K_{\text{ground}}, V_{\text{ground}}) \tag{8.31}$$

γ 初始化为 0——和 ControlNet 的零卷积同一个思路：**从“完全不用”开始，学会用多少。**
它比区域提示词强在哪：框是**可学习的连续量**，不是硬掩码。**代价**：需要带框标注的训练数据。

8.21.4 做法三：用另一张图当条件（ControlNet）

如果“位置”其实来自一张参考图（草图、深度图、姿态），那就不是文本的问题，是**结构条件**的问题。
这正是 ControlNet 的定位（上一节 ⑤）。
三种做法对比：

做法	输入	精度	需要什么
区域提示词	文本 + 掩码	中（格子级）	无额外训练（免训）
GLIGEN	文本 + 框	高	带框数据
ControlNet	文本 + 结构图	最高	成对数据（图-结构）

> **i** 一个判断准则
>
> **问自己：这个约束能不能用“词”表达?**
>
> · **能**（“红色的猫”）→ 文本条件，别加复杂度
> · **不能，但能用“哪块区域”表达**（“左上角有太阳”）→ 区域提示词
> · **不能，且需要精确形状**（“这个人的姿势”）→ ControlNet
>
> **三种条件的成本差一个数量级。先用最便宜的那个。**

本章讲的交叉注意力，是这三条通道里最基础的一条。另外两条不是替代它，是在它上面加约束。

8.22 属性泄漏：当“红色”跑到球上

比“位置不对”更常见的一种失败：颜色、材质、大小串到别的物体上。
提示词：`a red cube and a blue sphere`（一个红方块和一个蓝球）
常见输出：一个**紫方块**和一个**蓝球**，或者一个红方块和一个**紫球**。
两个物体的颜色被平均了。

8.22.1 机制：一次 softmax，所有词竞争

交叉注意力的每个位置只做一次 softmax。
对一个落在方块上的潜位置，它的 query 要和所有 token 打分：`a`、`red`、`cube`、`and`、`a`、`blue`、`sphere`。
“red”和“blue”都会得分，“cube”和“sphere”也都会得分。 softmax 之后，这个位置可能同时拿到“红”和“蓝”的成分—— **于是它变成了紫色。**
理想情况应该是：

$$a_{\text{方块,red}} \approx 1, \qquad a_{\text{方块,blue}} \approx 0 \tag{8.32}$$

实际是：

$$a_{\text{方块,red}} \approx 0.4, \qquad a_{\text{方块,blue}} \approx 0.3 \tag{8.33}$$

剩下的份额给了 `a`、`and` 这些虚词。

> **!** 这不是“模型分不清颜色”
>
> **模型完全知道红和蓝是什么。** 它的问题是：**没有一个机制强制“红色只属于方块”。**
> **文本编码器输出的是一串 token，token 之间没有“修饰关系”这张图。**“red”和“cube”在序列里相邻，但交叉注意力不知道“相邻 = 修饰”。
> **本章的 RB/BR 实验是这个问题的最小版本**：两个 token、两个位置，模型都可能绑错。**真实提示词有 77 个 token、几十个位置，难度是指数级的。**

8.22.2 为什么加大 w 不解决

引导放大的是“条件方向与无条件方向之差”。
如果条件本身把红和蓝混在一起，放大只是把“紫色”放大得更饱和。
这和第 7 章的结论一致：引导不能创造信息，只能放大已有的方向。

8.22.3 三种针对性的做法

① **拆分提示词，分别算注意力**（Structured Diffusion Guidance, 2023）
把一句话解析成若干子句：

```
a red cube  |  a blue sphere
```

每个子句单独过文本编码器，得到各自的 K, V， 再在交叉注意力里按空间掩码分配给不同区域。

它把“竞争”变成了“分区”。

② **强制每个主语都被注意到**（Attend-and-Excite, 2023）

做法：在采样过程中监控每个主语 token 的最大注意力，如果某个主语的最大注意力低于阈值，**就优化潜变量让它升上去**。

$$\mathcal{L}_{\text{attn}} = \sum_i \max\left(0, \tau - \max_p a_{p,\text{subject}_i}\right) \tag{8.34}$$

注意这是在第 6 章的采样循环里加一步优化—— 它改的是采样过程，不是模型权重。

③ **让大模型重写提示词**

用 LLM 把“红方块和蓝球”改写成更明确的描述， 或者拆成多个生成步骤再合成。

它有用，但没有解决机制——只是把难题换成了一个更容易的表述。**而且在“必须精确保持用户原话”的场景下不能用。**

8.22.4 怎么测量属性泄漏

一个不依赖人工的协议：

1. 准备 N 个“两物体两属性”的提示词（颜色、材质、大小各一组）
2. 每个提示词生成 M 张
3. 用一个检测器（或人工）判断：**每个物体的属性是否与提示词一致**
4. 报告**属性准确率**，而不是“整图像不像”

关键：要分别报告“物体 A 的属性”和“物体 B 的属性”。**只看总体，会把“一个对一个错”平均成 50%，看不出泄漏的方向。**

> ℹ 和第 7 章的“准确率饱和”是同一类陷阱
>
> 第 7 章：整体准确率 1.000，但样本已经冲出数据范围。本章：CLIP score 很高，但两个物体的颜色串了。
>
> **两个例子说的是同一件事： 一个把结构平均掉的标量指标，会在结构出错时保持漂亮。**
>
> **所以评估必须“分槽”**——本章的双槽准确率就是一个分槽指标。

8.23 文本编码器怎么选

真实项目里，这一步通常不是“设计”，是“继承”——你用的底座模型已经定了编码器。 但知道取舍，能帮你判断它为什么在某些提示词上失灵。

8.23.1 三种主流选择

编码器	输出	典型用途	特点
CLIP	$77\times d$ 序列 + 一个池化向量	Stable Diffusion 1.x/2.x	图文对比学习，擅长“像不像”
T5	变长序列	Imagen、SD3	纯文本预训练，擅长长句与组合
Qwen-VL 等	变长序列	Qwen-Image	多语言、长上下文

CLIP 的关键限制是 77 个 token。 超出会被截断。 **T5 没有这个硬限制**（相对位置编码），所以 SD3 换成了 T5。

8.23.2 三个要问的问题

① 输出是序列还是向量?

- **序列**：可以接交叉注意力（本章的做法）
- **向量**：只能接 AdaGN 或拼接（第 7 章的类别条件）

CLIP 两种都有：`last_hidden_state`（序列）和 `pooler_output`（池化）。 **SD 1.x 用的是序列**，池化向量被丢掉了——因为序列的信息更多。

② **是不是冻结的?**

大多数文生图模型**冻结文本编码器**，只训 UNet。 **好处**：省显存、不用文本-图像配对数据。 **代价**：编码器不懂图像生成需要什么—— 比如它不知道“红色”对像素意味着什么。

③ **空串编码是什么?**

不同编码器对 `""` 的处理不同： CLIP 给 `[BOS][EOS]`，T5 可能给一个 `[PAD]` 序列。 **换编码器必须重新验证 $w = 0$**（失败四）。

> **i** 一个常被忽略的事实
>
> **文本编码器通常不参与扩散训练。**
> 它输出的向量分布是固定的，**UNet 只是学会“怎么用这些向量”**。 所以:
>
> · **换编码器 = 换了一门语言**，UNet 得重训
> · **微调编码器 = 改变了这门语言的词义**，同样要重训 UNet
>
> **“只换编码器不重训”是不成立的**，除非两个编码器的输出分布几乎一样。

8.24 CLIP 是怎么训出来的（以及它为什么不懂顺序）

上一节说“编码器通常是继承的”。那它是怎么来的?

绝大多数文生图模型用的文本编码器，都是对比学习训出来的。理解这个目标函数，你就能预测它在哪类提示词上会失灵。

8.24.1 一个批次，一个矩阵

训练数据是图文对：$(I_1, T_1), \ldots, (I_N, T_N)$。

两个塔：图像塔把 I_i 编成 u_i，文本塔把 T_i 编成 v_i，都归一化到单位长度。**相似度就是点积** $s_{ij} = u_i^\top v_j$。

一个批次 $N = 32,768$，就得到一个 32768×32768 的相似度矩阵。 对角线是“配对的”，其余 10^9 个格子全是“不配对的”。

损失（InfoNCE，两个方向对称）：

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N} \log \frac{\exp(s_{ii}/\tau)}{\sum_{j=1}^{N} \exp(s_{ij}/\tau)} \tag{8.35}$$

τ 是温度，CLIP 里初始化为 0.07，并且可学习。

8.24.2 先算一遍：三个样本

设某张图与三条文本的相似度是 $0.30, 0.10, -0.10$，$\tau = 0.07$。

第一步，除以温度:

$$0.30/0.07 = 4.29, \qquad 0.10/0.07 = 1.43, \qquad -0.10/0.07 = -1.43 \tag{8.36}$$

第二步，softmax:

$$p = \frac{e^{4.29}}{e^{4.29} + e^{1.43} + e^{-1.43}} = \frac{72.9}{72.9 + 4.17 + 0.24} = 0.943 \tag{8.37}$$

第三步，取负对数:

$$\mathcal{L} = -\log 0.943 = 0.059 \tag{8.38}$$

现在把正确的那条换掉——让第二列成为正样本:

$$p = \frac{e^{1.43}}{72.9 + 4.17 + 0.24} = 0.054, \qquad \mathcal{L} = -\log 0.054 = 2.916 \tag{8.39}$$

同一个相似度矩阵，正样本选对了损失 0.059，选错了 2.916——差 49 倍。

> ! 这个目标函数在奖励什么
>
> **它只奖励一件事：在这一批里，把正确的那个挑出来。**
>
> · 拉近配对：s_{ii} 变大
> · 推远**批内其他所有**：s_{ij} 变小
>
> **注意加粗的“批内”。** 负样本不是精心挑的，是**同一批里恰好排在一起的。 这决定了模型学到什么、学不到什么。**

8.24.3 为什么这解释了“不懂顺序”

问一个问题：模型有没有动力去区分“红方块在蓝球上”和“蓝方块在红球上”?
只有当这两条描述同时出现在一个批次里、且分别配着不同的图时，才有动力。
算一下概率。

· 训练集 4 亿对
· 批次 32,768
· 随机抽到某条描述“顺序对调版”的概率：

$$\frac{32,768}{4 \times 10^8} = 8.2 \times 10^{-5} \tag{8.40}$$

一万次里不到一次。
所以对比学习几乎从不要求模型区分词序。 它要求的是“把这一批的 3 万条描述彼此区分开”——而这一批里，绝大多数差异是“猫 vs 狗 vs 汽车”，不是“猫在左 vs 猫在右”。
结果：CLIP 的文本表示对**内容**敏感，对**结构**迟钝。
这不是 bug, 是目标函数没有要求。2023 年的一项系统研究标题就叫《视觉语言模型何时以及为何表现得像词袋》（When and Why Vision-Language Models Behave Like Bags-of-Words），结论和上面的推理一致：**交换主语和宾语，CLIP 的相似度几乎不变。**

> ⚠ 这直接接上了本章开头的案例
>
> **本章的均值池化是一个“故意做错”的模型。CLIP 不是均值池化**——它有 12 层自注意力，能看到顺序。
> **但它的训练目标没有逼它用顺序。所以它在“红在左还是红在右”这类问题上，表现得像一个很贵的词袋。**
> **这就是为什么 SD3 要加一个 T5**：T5 是在纯文本上做去噪预训练的，**预测被挖掉的词必须看上下文**，所以它对语法结构敏感得多。**两个编码器并联，是在“语义对齐”和“结构敏感”之间取长补短。**

8.24.4 模态鸿沟：一个附带现象

训练完之后，图像向量和文本向量并不在同一个区域。
它们的方向能对齐（点积大），但范数分布不同——图像向量倾向于聚集在更窄的锥形里，文本向量铺得更开。 这个现象叫**模态鸿沟**（modality gap），2022 年被系统测量过。
对文生图的影响：

· 文本编码器的输出分布是“偏”的，UNet 必须学会适应
· **这正是“换编码器要重训 UNet”的深层原因**——不是维度对不上，是分布对不上
· 也是 scaling factor 那类“看起来多余、其实必须”的常数的来源

8.24.5 三个可以直接用的推论

① 提示词里的词越“独特”，CLIP 越敏感。因为对比学习奖励的是**区分度**。“一只猫”里的“一只”没有区分度，“猫”有。**所以删掉虚词通常不掉点，删掉名词立刻变。**

② 同义词替换通常是安全的，语序调整通常不安全。同义词在原训练分布里经常互换出现（“小猫/猫咪”），语序对调几乎从不作为负样本出现。

③ 否定词是已知的弱项。“没有猫”和“猫”在词袋层面高度重叠，CLIP 区分它们的能力很差。**这是“负面提示词”要用单独机制的另一个原因**（第 7 章的负向引导）。

8.25 教模型一个新词：三种改法，三种代价

前面说“换编码器要重训 UNet”。那如果只是想让它认识一个新概念呢?

比如你自己的脸、一个原创角色、一种画风。

有三种做法，改的东西一层比一层多。

8.25.1 ① Textual Inversion：只学一个向量

做法（Gal et al., 2022）：选一个占位词 `<my-concept>`，**只优化它的词嵌入向量**，模型其余部分全部冻结。

$$\mathcal{L} = \mathbb{E}_{t,x_0,\varepsilon} \left\| \varepsilon - \varepsilon_\theta(x_t, t, \text{Enc}\left(\texttt{<my-concept>}\right)) \right\|^2 \tag{8.41}$$

要学的参数只有一个 768 维向量——768 个数。

对比一下：UNet 是 8.6 亿参数。比例是 9×10^{-7}。

为什么这么少也能 work? 因为文本编码器的输出空间里，**已经存在“能描述这个概念”的点**——只是没有一个词指向它。Textual Inversion 做的事，是**在空间里找那个点**，不是创造新知识。

它的边界也在这里：如果概念需要“新的组合方式”（比如“这个人穿这种盔甲”），一个向量装不下。**它擅长“是什么”，不擅长“怎么组合”。**

8.25.2 ② LoRA：给权重加一个低秩补丁

做法：不动原权重 W，在旁边加一个低秩分解：

$$W' = W + \frac{\alpha}{r} BA, \qquad B \in \mathbb{R}^{d\times r}, A \in \mathbb{R}^{r\times k} \tag{8.42}$$

r 通常取 4–64。

参数量从 $d \times k$ 降到 $r(d+k)$。以 $d = k = 768$、$r = 8$ 为例：

$$768^2 = 589,824 \quad \to \quad 8 \times (768 + 768) = 12,288 \tag{8.43}$$

少 48 倍。

LoRA 可以只加在文本编码器上，也可以同时加在 UNet 上。

- **只加文本编码器**：改变“词义”，UNet 得适应新分布 → 通常效果不稳
- **同时加两边**：效果最好，但参数量大
- **只加 UNet**：学“画法”，不改词义 → 最常见

8.25.3 ③ DreamBooth：直接微调 UNet

做法（Ruiz et al., 2023）：用一个罕见标识词（如 `sks`）指代主体，**微调整个 UNet**，同时用“先验保持损失”防止遗忘：

$$\mathcal{L} = \mathcal{L}_{\text{diff}} + \lambda \mathcal{L}_{\text{prior}} \tag{8.44}$$

先验项：用模型自己生成的“这一类物体”的图做正则，**防止模型把“狗”这个词拉向你的那只狗。**

! 三种做法的分工

做法	改什么	参数量	适合
Textual Inversion	一个词向量	768	新概念、新风格
LoRA	低秩补丁	1–10M	画风、角色、构图习惯
DreamBooth	整个 UNet	860M	高保真主体复现

判断准则：问“这个概念在不在模型的词表里”。

- **在，只是没有词指向它** → Textual Inversion
- **不在，是一种“画法”** → LoRA
- **要一模一样** → DreamBooth，并接受过拟合风险

这三种做法都绕开了“重训 UNet” —— 但它们各自付出的是**表达力**：**改得越少，能表达的新东西越受限。**

这和第 7 章“引导不能创造信息”是同一条原则的另一个面。

8.26 端到端：一句话变成一张图

把本章和前面几章串起来。 从提示词到像素，一共八步。

8.26.1 第一步：分词与编码

```
tok = tokenizer(prompt, padding="max_length", max_length=77,
                truncation=True, return_tensors="pt")
cond = text_encoder(tok.input_ids).last_hidden_state      # (1, 77, d)
```

注意 `truncation=True` 和 `max_length=77`——超出的部分在这里就没了。

8.26.2 第二步：空条件的编码

```
uncond = text_encoder(empty_ids).last_hidden_state        # (1, 77, d)
```

`empty_ids` 是空字符串的分词结果，不是全零。

8.26.3 第三步：初始化潜变量

```
latent = torch.randn(1, 4, 64, 64) * scheduler.init_noise_sigma
```

从标准正态开始， 形状是 $(4, 64, 64)$。

8.26.4 第四步：每一步做两次前向

```
for t in timesteps:
    latent_in = scheduler.scale_model_input(latent, t)
    noise_c = unet(latent_in, t, encoder_hidden_states=cond).sample
    noise_u = unet(latent_in, t, encoder_hidden_states=uncond).sample
    noise = noise_u + w * (noise_c - noise_u)             # ← CFG
    latent = scheduler.step(noise, t, latent).prev_sample
```

这一循环里有本章的全部内容：

- `encoder_hidden_states=cond` **走交叉注意力**
- 两个分支共用同一个 UNet
- `w` 就是第 7 章那个引导强度

8.26.5 第五步：解码

```
latent = latent / vae.config.scaling_factor
image = vae.decode(latent).sample
```

除以 scaling factor（编码时是乘）。

> 八步里，哪几步最容易出错

步骤	常见错误	症状
一	忘了 truncation	长提示词后半段无效
二	用零向量当空条件	$w > 0$ 就崩
三	用了错误的 init sigma	画面偏灰或过曝
五	忘了除 scaling factor	颜色整体偏移

四个错误里，两个和“文本”有关（一、二），两个和“潜空间”有关（三、五）。这就是本章的两个主题。

8.26.6 一个容易被忽略的顺序

`scheduler.scale_model_input` 必须在两次 UNet 之前调用，而且只调用一次。
有些实现为了省事，在 CFG 之后又 scale 一次——**那会让两个分支的输入不一致。**
症状：w 调大时画面出现规律的色带。

8.27 走一遍 SDXL 的文本路径

抽象讲完，走一遍真实的。
这一段代码是 `diffusers` 里 SDXL 管线做过的事，每一步都标了形状。

8.27.1 第 1 步：两个分词器，两条路

```
tok1 = tokenizer(prompt, padding="max_length", max_length=77,
                 truncation=True, return_tensors="pt")   # CLIP-L
tok2 = tokenizer_2(prompt, padding="max_length", max_length=77,
                   truncation=True, return_tensors="pt")  # CLIP-G
```

两个编码器各有自己的分词器（词表不同），**但都是 77 个位置。**
所以同一句提示词被切了两次，切法可能不一样。一个在 CLIP-L 里是 5 个 token，在 CLIP-G 里可能是 6 个。

8.27.2 第 2 步：取倒数第二层

```
out1 = text_encoder(tok1.input_ids, output_hidden_states=True)
out2 = text_encoder_2(tok2.input_ids, output_hidden_states=True)

emb1 = out1.hidden_states[-2]     # (1, 77, 768)
emb2 = out2.hidden_states[-2]     # (1, 77, 1280)
```

⚠ 为什么是 `[-2]` 而不是 `[-1]`

CLIP 的最后一层是专门为对比学习目标优化的。
它的输出被拉向“和图像对齐”的方向， 语义细节（比如“左边”“两只”）在这个过程中被压掉了。
倒数第二层还保留着更“语言”的表示。
这是一个纯经验的工程选择，论文里没有严格证明。**但它说明一件事：“最后一层”不等于“最好的表示”——要看你拿它做什么。**

8.27.3 第 3 步：拼起来

```
prompt_embeds = torch.cat([emb1, emb2], dim=-1)   # (1, 77, 2048)
```

768 + 1280 = 2048。
这不是“平均”，是“拼接”。 两个编码器的表示被并排放着，让 UNet 自己决定用哪一个。
拼接而不是相加，是因为两者没有对齐过—— CLIP-L 的第 3 维和 CLIP-G 的第 3 维没有任何关系。 **相加会制造一个没有意义的混合向量。**

8.27.4 第 4 步：池化向量走另一条路

```
pooled1 = out1.pooler_output          # (1, 768)
pooled2 = out2.pooler_output          # (1, 1280)
pooled  = torch.cat([pooled1, pooled2], dim=-1)   # (1, 2048)
```

序列 `(77, 2048)` 进交叉注意力。 池化 `(2048,)` 进时间步嵌入，参与 AdaGN。
所以 SDXL 同时用了本章讲的两种条件注入方式： 逐位置的交叉注意力 + 全局的 AdaGN。
这是它比 SD1.5 多出来的东西： SD1.5 只用序列，池化向量被丢掉了。

8.27.5 第 5 步：空条件走完全相同的路

```
uncond_embeds = encode("")        # (1, 77, 2048)
uncond_pooled = encode_pooled("") # (1, 2048)
```

注意：空字符串也要过两个分词器、两个编码器、同样的拼接。
它不是零，不是 None，是一个完整的、2048 维的、被训练过的表示。

8.27.6 第 6 步：UNet 里发生什么

```
prompt_embeds (1, 77, 2048)
      ↓ 每层一个 Linear(2048 → d)
   K, V  (1, 77, d)      ← 16 层交叉注意力共用同一份
      ↑
   Q 来自图像特征 (1, N, d)
      ↓
   输出 (1, N, d)
```

16 层，每层都把 2048 投影到自己的 d（320/640/1280）。
这就是前面算过的 4400 万参数的去处。

> **i** 一个容易忽略的成本
>
> **文本序列在 16 层里是共享的，但投影矩阵不共享。**
> **每层都有自己的 `to_k` 和 `to_v`—— 所以“同一个文本”在浅层和深层被投影成不同的 K, V。**
> **这不是冗余，是必要的：浅层需要“词长什么样”，深层需要“词和当前语义的关系”。**
> **这也是为什么“缓存文本 embedding”只能省编码器那一步，省不掉 UNet 里的 16 次投影。**

8.28 训一个文本条件模型：和类别模型的三处不同

训练循环几乎一样：采样 x_0、加噪、预测 ε、MSE。**不同的只有三处。**

8.28.1 第一处：条件是序列，不是标量

```
cond = text_encoder(tok).last_hidden_state      # (B, 77, d)
```

注意它是三维的——batch × 序列 × 维度。UNet 的交叉注意力需要这个三维形状，**类别模型里的一维向量在这里不适用。**
代价：显存。77 个 token × d 的激活要一直留着。

8.28.2 第二处：文本丢弃要丢“整段”

第 7 章的类别模型丢的是一个下标；**这里丢的是整段文本**：

```
mask = torch.rand(B) < 0.1
tok[mask] = empty_token_ids                     # 换成空串的分词结果
```

关键：`empty_token_ids` 必须是**空字符串真实的 tokenize 结果**，不是全零、也不是全 `[PAD]`。
这就是第 7 章“空条件必须被训练过”在文本上的版本。

8.28.3 第三处：编码器通常冻结

```
text_encoder.requires_grad_(False)
```

为什么冻结？

理由	说明
省显存	编码器动辄几亿参数，反向传播的激活很贵
省数据	不需要“文本-图像”配对来训编码器
稳定	条件分布固定，UNet 只需要学“怎么用”

代价：编码器不懂“生成需要什么”。**它只知道“这段文字和那张图配不配”，不知道“红色对像素意味着什么”。**

> **i** 冻结带来的一个隐性约束
>
> **UNet 必须适应编码器的“怪癖”。**
> 比如 CLIP 的因果注意力让 `[EOS]` 位置变成“整句摘要”，但序列里其它位置只看到前半句。
> **UNet 学会的是“从这 77 个向量里取信息”，而它取的方式依赖 CLIP 的这种特性。**
> **所以换编码器不是“换个更好的”，是“换一套要重新适应的输入”。**

8.28.4 一个数据上的坑

图文配对的质量比数量重要。

如果 caption 只写“一张图片”，**模型学到的就是“忽略文本”**—— 因为文本不提供任何区分信息。

在文本丢弃率 10% 的情况下，90% 的样本需要文本真正有用， 否则模型会退化成“主要看空条件”。

判据：训练后做一次“换文本”测试—— 同一个噪声、两个不同提示词，输出应该明显不同。**如果几乎一样，说明文本没被用起来。**

8.29 代码逐行：训练循环与采样器

前面给了编码器的十几行。这一节把外面那层壳也拆开—— 训练循环和采样器各有三处容易写错的地方。

8.29.1 训练循环

```
def train_text_model(x0, y, cfg, sch, encoder="cross", steps=6000, drop=0.1):
    tok = torch.stack([y // 2, y % 2], dim=1)          # ① 类别 → 两个 token
    model = TextCondDenoiser(cfg, encoder=encoder)
    opt = torch.optim.Adam(model.parameters(), lr=cfg.lr)
    for step in range(steps):
        idx = torch.randint(n, (cfg.batch_size,), generator=g)
        tk = tok[idx].clone()                           # ② 必须 clone
        mask = torch.rand(len(idx), generator=g) < drop
        tk[mask] = 2                                    # ③ NULL token
        t = torch.randint(0, cfg.n_steps, (len(idx),), generator=g)
        xt, eps = forward_sample(x0[idx], t, sch)
        loss = F.mse_loss(model(xt, t, tk), eps)
        opt.zero_grad(set_to_none=True)
        loss.backward()
        opt.step()
```

① **`y // 2` 和 `y % 2`**。

类别编号 $0,1,2,3$ 拆成两个二进制位：$0\to(0,0)$、$1\to(0,1)$、$2\to(1,0)$、$3\to(1,1)$。**这正好是“左色、右色”两个 token。**

为什么不用 `nn.Embedding(4, d)` 直接查？ 因为那样就退化成第 7 章的类别条件了——**没有“两个词”这个结构。**

② **`tok[idx].clone()`**。

没有 `clone()` 会出大错：`tk[mask] = 2` 是原地赋值，**会把整个 `tok` 表里的样本也改掉。** 下一批再取到它们时，条件已经被污染了。

症状：训练损失正常下降，但推理时准确率莫名其妙地低。**因为训练时有一半样本的条件被永久改成了 NULL，而你没发现。**

③ **`tk[mask] = 2`**。

这是第 7 章“标签丢弃”的文本版本。 注意它丢的是**整段**——两个 token 都变成 NULL，不是随机丢其中一个。

只丢一个会怎样？ 模型会学到“部分条件”这个中间状态，而 CFG 需要的是**完整的无条件。两者不是一回事。**

8.29.2 采样器

```
@torch.no_grad()
def sample_text(model, sch, tok, w=1.0, seed=0):
    x = torch.randn(len(tok), 4)
    null = torch.full_like(tok, 2)
    for t in range(T - 1, -1, -1):
        eps_c = model(x, tt, tok)
        if w != 1.0:
            eps_u = model(x, tt, null)
            eps_c = eps_u + w * (eps_c - eps_u)
```

```
        x0 = ((x - (1 - ab).sqrt() * eps_c) / ab.sqrt()).clamp(-3, 3)
        mean = (abp.sqrt() * beta / (1 - ab)) * x0 + \
               (alpha.sqrt() * (1 - abp) / (1 - ab)) * x
        x = mean + (beta * (1 - abp) / (1 - ab)).sqrt() * torch.randn_like(x)
```

这里有三处值得停一下。

第一处：`x0` 后面的 `.clamp(-3, 3)`。

$$\hat{x}_0 = \frac{x_t - \sqrt{1-\bar{\alpha}_t}\,\hat{\varepsilon}}{\sqrt{\bar{\alpha}_t}} \tag{8.45}$$

当 t 很大时，$\sqrt{\bar{\alpha}_t}$ 很小，分母接近 0。$\hat{\varepsilon}$ 的一点点误差会被放大成巨大的 $\hat{x}_0$。
clamp 是数值保护，不是数学推导的一部分。去掉它，采样会在高噪声步炸掉。
第二处：`tt` 是标量张量。
模型要的是每个样本一个时间步，不是整个 batch 一个。这里所有样本同步采样，所以可以共用一个 `t`。**如果做部分重绘（inpainting），不同区域的时间步不同，就不能共用。**
第三处：最后一步不加噪声。

```
x = mean + noise if t > 0 else mean
```

$t = 0$ 时后验方差为 0（β_0 很小但不为零，理论上还该加）。**实践里直接取均值，**因为最后一步的噪声只会让画面更脏。

> ⚠ 一个几乎所有人都会踩的坑
>
> **`w != 1.0` 时，每一步要做两次前向。**
> **如果写成“两次前向 + 一次 step”，计算量正好翻倍。很多人以为 CFG 是免费的——它不是，它是 2× 推理成本。**
> **省一半的办法：**把条件和空条件拼成一个 batch，一次前向：
>
> ```
> x2 = torch.cat([x, x])
> t2 = torch.cat([tt, tt])
> tok2 = torch.cat([tok, null])
> eps2 = model(x2, t2, tok2)          # 一次前向，batch 翻倍
> eps_c, eps_u = eps2.chunk(2)
> ```
>
> **batch 翻倍和两次前向的计算量一样，但少了一次 kernel 启动。在 GPU 上通常快 10–20%。**

8.30 梯度怎么流回文本

交叉注意力里，文本是被“读”的。那它能不能被“改”？
能——只要它参与计算，梯度就会流回去。这一节把这条路径算清楚，它能解释好几件工程上的事。

8.30.1 从输出往回推

设一个位置的输出：

$$o = \sum_k a_k v_k, \qquad a_k = \mathrm{softmax}(s)_k, \qquad s_k = \frac{q^\top k_k}{\sqrt{d}} \tag{8.46}$$

损失 L 对 v_k 的梯度：

$$\frac{\partial L}{\partial v_k} = a_k \frac{\partial L}{\partial o} \tag{8.47}$$

很直白：注意力权重就是“这个 token 分到多少梯度”。
对 a_k 的梯度：

$$\frac{\partial L}{\partial a_k} = \frac{\partial L}{\partial o} \cdot v_k \tag{8.48}$$

记 $g_k = \frac{\partial L}{\partial o} \cdot v_k$——“第 k 个 token 的内容对输出的贡献”。
softmax 的雅可比是 $\frac{\partial a_j}{\partial s_k} = a_j(\delta_{jk} - a_k)$，于是

$$\frac{\partial L}{\partial s_j} = a_j \left(g_j - \sum_k a_k g_k \right) \tag{8.49}$$

! 这个式子的三个含义

① 括号里是“这个 token 的贡献，减去平均贡献”。只有“比平均更好或更差”的 token 才拿到梯度。 如果所有 token 对输出的贡献一样，梯度为零——**没什么可学的。**
② 前面的 a_j 是“话语权”。权重小的 token，梯度也小。 没有阈值，但有一个乘数。
③ 这个乘数不会因为权重“均匀”而消失。 $a_j = 0.5$ 时，梯度是 $0.5 \times$ (偏差)——**仍然有一半。这就是“均匀注意力仍然能学习”的数学原因。**

8.30.2 算一个具体的数

两个 token，$a = (0.9, 0.1)$，贡献 $g = (1.0, 0.0)$。
平均贡献：$\bar{g} = 0.9 \times 1.0 + 0.1 \times 0.0 = 0.9$。

$$\frac{\partial L}{\partial s_1} = 0.9 \times (1.0 - 0.9) = 0.09 \tag{8.50}$$

$$\frac{\partial L}{\partial s_2} = 0.1 \times (0.0 - 0.9) = -0.09 \tag{8.51}$$

两个梯度大小相同、符号相反。softmax 的输出和恒为 1，所以梯度也必须和为零——一个上去，另一个就得下来。
如果 $a = (0.5, 0.5)$ 呢?

$$\frac{\partial L}{\partial s_1} = 0.5 \times (1.0 - 0.5) = 0.25, \qquad \frac{\partial L}{\partial s_2} = 0.5 \times (0.0 - 0.5) = -0.25 \tag{8.52}$$

梯度反而更大。
因为均匀分布离“极端”最远，softmax 在这里的斜率最大。

⚠ “注意力均匀 = 学不到东西”是一个误解

很多人看到权重接近 0.5，就认为“模型没学到东西”。
上面的计算说明：均匀恰恰是梯度最大的位置。
本章的实测也支持这一点： `no_xy` **的注意力完全均匀（0.525/0.525），绑定照样 1.000。**
权重均匀不代表没学习，可能代表“每个词都同等重要”。

8.30.3 梯度继续往回：到 q 和 k

$$\frac{\partial L}{\partial q} = \sum_j \frac{\partial L}{\partial s_j} \cdot \frac{k_j}{\sqrt{d}}, \qquad \frac{\partial L}{\partial k_j} = \frac{\partial L}{\partial s_j} \cdot \frac{q}{\sqrt{d}} \tag{8.53}$$

注意 k_j 的梯度里带着 q。如果某个位置的 query 从来不和某个 key 打分（被掩码挡住），那个 key 就永远拿不到梯度。
这就是区域掩码的副作用：被掩掉的位置，对应的词学不到任何东西。所以掩码要“轮换”——否则模型会对某些词完全失明。

8.30.4 如果文本编码器是冻结的

梯度到 k, v 就停了。 编码器不更新，文本表示不变。

所以“冻结编码器”的真实含义是：交叉注意力的 K, V 投影会学，但文本本身的表示不会。

这两件事经常被混为一谈：

冻结什么	谁还在学	效果
编码器 + 投影	只有 UNet 其余部分	文本完全固定
只冻结编码器	K, V 投影 + UNet	模型学“怎么读”，不学“词义”
全都不冻结	全部	词义会漂移，需要更多数据

大多数实现是第二种。说“文本编码器冻结了所以文本不变”是对的；说“所以交叉注意力不学习”是错的。

8.31 描述文字是怎么写出来的

模型能听懂什么样的提示词，不由架构决定，由训练数据的 caption 决定。

这一节解释一个很实际的现象：为什么同一个提示词，在 SD1.5 上要写成关键词堆砌，在 SDXL 上要写成短句，在 SD3 上要写成长句。

8.31.1 三代 caption，三种写法

代	数据来源	caption 长什么样	适合的提示词
SD1.5	LAION 的网页 alt-text	`a photo of a cat, high quality`	**关键词堆砌**
SDXL	用 VLM 重新描述	`A cat sitting on a windowsill, sunlight...`	**短句**
SD3 / FLUX	更细的 VLM 重述	完整段落，含空间关系	**自然语言长句**

第一代的问题：alt-text 是网页作者随手写的，**短、噪声大、经常和图像无关**（“点击这里”“产品图 3”）。

模型从这种数据里学到的是：“文本大概指示一下主题就行，细节靠先验。”**所以 SD1.5 对长句不敏感，对关键词敏感。**

第二代的转折：SDXL 用了一个在小型高质量数据上微调过的 captioner，**把训练图重新描述一遍**。caption 变长、变准、含空间关系。

第三代：SD3 用更强的 VLM 重述，caption 里开始出现 “左边”“背后”“拿着”这类**关系词**。

> ! 一条实用推论
>
> **提示词的风格要和训练 caption 的风格匹配。**
>
> - 在 SD1.5 上写一段优美的英文长句 → **大部分 token 是分布外的**
> - 在 SD3 上堆 `masterpiece, best quality, 8k` → **这些词在它的 caption 里几乎不出现**
> - 在任何一个模型上，**用它的数据分布里常见的句式**，效果最好
>
> **这不是玄学，是“训练/推理分布匹配”**——和失败四（换 tokenizer）是同一个道理的另一个版本。

8.31.2 为什么“魔法词”会存在

早期社区发现某些词（特定的艺术家名、`masterpiece`、`8k uhd`）能显著提升观感。

机制：这些词在 LAION 的 alt-text 里**和高质量图片强相关**—— 不是因为它描述了什么，而是因为**写这些词的人倾向于配好图**。

所以模型学到的不是“8k 意味着高清”，是“出现 8k 的图，平均质量更高”。

这解释了两件事：

1. **为什么魔法词换模型就失效**——新模型的 caption 分布里没有这个相关性
2. **为什么魔法词有时会带来副作用**——它同时拉动了训练数据里的其它属性

8.31.3 caption 的长度分布也在被学习

如果训练数据里 90% 的 caption 短于 20 个 token，模型就没见过“40 个 token 的详细描述”。

后果：长提示词的后半段**不是被截断，是被忽略**—— 模型没见过“后面还有信息”这种模式。

这和 77 token 的硬截断是两回事：

	硬截断	软忽略
原因	位置编码用完	训练分布里没有长 caption
症状	第 78 个 token 之后完全无效	长句的后半段权重很低
修法	换编码器 / 截短	换模型 / 用它的风格写

很多“长提示词没用”的报告，其实是第二种。

8.31.4 一个可以自己做的检验

取一批你的提示词，做两件事：

1. 把后半句删掉，看输出变化大不大
2. 把同一句话改写成“关键词堆砌”和“自然语言”两版，对比

如果第 1 步几乎没变化，说明模型在软忽略长句；如果第 2 步差距很大，说明它偏向某一种 caption 风格。

这两个检验都不需要看模型内部，五分钟能做完。

8.32 这一章站在前七章的哪里

这本书是一条线，不是八个孤岛。这一节把线接回去。

8.32.1 第 1 章：注意力的公式

本章的交叉注意力和第 1 章的自注意力是同一个公式：

$$\text{softmax}\left(\frac{QK^\top}{\sqrt{d}}\right)V \tag{8.54}$$

第 1 章讲的是“Q, K, V 都来自同一串 token”，本章讲的是“Q 来自图像、K, V 来自文本”。

同一个公式，换了三个输入，就变成了另一个机制。这就是第 1 章反复强调“注意力是一个查询接口”的意义。

本章还用了第 1 章的两个结论：$\sqrt{d}$ **的缩放不能省**（不然 softmax 会饱和），**多头让不同的头学不同的关系**（本章的 `n_head=2`）。

8.32.2 第 2 章：Block 的结构

文本编码器就是若干个标准 Block 堆起来的：自注意力 + FFN + 残差 + LayerNorm。

本章没有展开讲这些，因为第 2 章已经讲过。但有一个细节值得回想：残差让“位置信息”能穿过很多层，这也是为什么位置嵌入加在输入上、而不是每一层都加。

本章的交叉注意力没有残差（它是一个独立模块，输出被拼进条件），**真实 UNet 里的交叉注意力是有残差的**—— `h = h + attn(h, text)`。**这一条残差让模型可以选择“不听文本”**。

8.32.3 第 3 章：训练与推理的分离

第 3 章讲“训练时并行、推理时逐 token”。

本章的对应物是：训练时每个样本随机采一个 t，一次前向；推理时要走 1000 步，每步两次前向（CFG）。

训练和推理的计算模式完全不同，这是扩散模型和第 3 章语言模型的共同点。

8.32.4 第 4 章：注意力图能解释什么

第 4 章在语言模型上找到了“盯住前一位”的头，也讲了“注意力图不等于解释”。

本章在文本条件上又遇到了一次，而且更彻底：权重完全均匀时绑定仍然成立（`no_xy`），权重很尖锐时绑定完全不成立（`no_pos`）。

第 4 章的教训是“别假设”，本章的贡献是“给了一个反例的定量版本”。

8.32.5 第 5 章：前向、反向、ε 预测

本章的 ε-预测目标、噪声调度、后验均值，全部来自第 5 章。

唯一的改动是：模型多了一个输入 c，损失从 $\|\varepsilon-\varepsilon_\theta(x_t,t)\|^2$ 变成 $\|\varepsilon-\varepsilon_\theta(x_t,t,c)\|^2$。

一行代码的差别，一整章的内容。

8.32.6 第 6 章：采样器

本章用的是 DDPM 采样（1000 步）。

第 6 章的 DDIM 结论在这里同样适用：少步数采样会放大模型误差，而条件误差也会被放大。本章没有测“少步数 + 文本条件”的组合——这是一个明确的空白。

一个可以预期的现象：步数少时，绑定准确率会先于整体质量下降，因为绑定依赖精细的注意力分配，而少步数采样对细节最不友好。

8.32.7 第 7 章：CFG

这是联系最紧的一章。

第 7 章	本章
条件是类别下标	条件是一串 token
空条件是 NULL 类嵌入	空条件是空串的编码
注入方式是相加/AdaGN	注入方式是交叉注意力
w 从 1 到 2 有明显收益	w 从 1 到 10 没有收益

最后一行不是矛盾，是“任务难度不同”：第 7 章的任务在 $w=1$ 时只有 0.965，有提升空间；本章的任务在 $w=1$ 时已经 1.000。**饱和了，就没有收益。**

两章共享的是同一条原则：引导放大的是“条件带来的差”，差为零时放大零还是零。

8.32.8 一条贯穿全书的线

如果把八章连起来看，主线是“信息怎么在模型里流动，以及在哪里被丢掉”。

- 第 1 章：注意力让信息在 token 之间流动
- 第 2 章：残差让信息跨层流动
- 第 3 章：训练让信息进入权重
- 第 4 章：我们试图“读”这些信息，发现读不准
- 第 5 章：扩散把信息从噪声里逐步恢复
- 第 6 章：采样器决定恢复的精度
- 第 7 章：引导放大某个方向的信息
- 第 8 章：文本把信息送进这个过程

本章的位置是“入口”：如果信息在入口就被池化掉了，后面七章的一切都救不回来。

8.33 为什么更强的编码器不一定带来更好的绑定

一个自然的猜想：换一个更强的文本编码器，绑定应该更好。

实测不是这样。

SD1.5 用 CLIP-L（12 层，768 维），SDXL 用 CLIP-L + CLIP-G（2048 维），SD3 再加一个 T5-XXL（4096 维）。

编码器一代比一代强，但“两只猫”还是两只猫，“左边”还是经常跑到右边。

为什么？三个原因。

8.33.1 原因一：瓶颈可能不在编码器

编码器负责“把话说清楚”，UNet 负责“把话用起来”。

如果 UNet 没有能力把“左边”映射到左半张图，编码器把“左边”表示得再精确也没用。

本章的实验正好说明了这一点：`no_xy` 的条件向量两个点完全一样，绑定照样 1.000—— 因为下游 MLP 有能力用 x 的坐标做路由。

反过来，如果下游没有这个能力，再好的条件也白搭。

所以“换编码器”的收益，取决于“UNet 是否已经用尽了现有条件的信息”。大多数情况下，没有。

8.33.2 原因二：维度越高，越难对齐

T5-XXL 输出 4096 维。

UNet 要把它投影到 320/640/1280 维。这是一个 4096×1280 的矩阵，每一层一个。

维度越高，投影矩阵越大，越难训。

而且高维表示里“有用的方向”可能很稀疏—— 4096 维里真正和“红色”相关的可能只有几十维。UNet 要从 4096 维里找出那几十维，比从 768 维里找更难。

这就是“更强的编码器”的隐性成本：它把“理解语言”的负担转移给了 UNet 的投影层。

8.33.3 原因三：训练目标不匹配

CLIP 是对比学习，T5 是去噪。两者的表示空间结构完全不同。

T5 的表示更适合“理解句子结构”，但它没有和图像对齐过。UNet 必须从头学“T5 的哪个方向对应红色”。

SD3 同时用 CLIP 和 T5，本质上是在做“对齐 + 结构”的折中：CLIP 提供已经和图像对齐过的方向，T5 提供结构信息。

> ! 判断准则：先分清是哪一层的问题
>
> 当你觉得“文本没被理解”时，先分清是哪一层的问题：
>
现象	可能的位置	怎么验证
词根本不认识	编码器 / 分词器	换同义词，看是否恢复
词认识但位置错	UNet 的路由	加区域掩码，看是否变好
两个物体属性串	交叉注意力的竞争	分开生成，看是否各自正确
整体风格不对	训练数据分布	换提示词风格，看是否恢复
>
> 只有第一行是编码器的问题。后三行换编码器都解决不了。

这也解释了为什么“换更大的模型”经常不如“换更好的条件结构”。

模型的能力上限由架构决定，但达到上限的方式由条件决定。一个 10 亿参数的模型配一个糟糕的条件，不如一个 1 亿参数的模型配一个好条件。

本章的全部实验，都可以看成这句话的注脚。

8.34 文本条件的数据从哪来

模型的文本能力，是数据喂出来的。这一节说清楚“喂的是什么”。

8.34.1 第一代：网页抓取

LAION-5B（2022）是最大的公开图文数据集： 从 Common Crawl 抓取网页，把 `<img>` 的 alt-text 当 caption， 用 CLIP 过滤掉图文相似度低的配对，得到 **58.5 亿对**。

它的三个特点，直接塑造了第一代模型：

① **caption 是网页作者写的，不是为生成写的。** “点击查看大图”“产品图 3”这类噪声大量存在。 **模型学到的文本表示，带着这种噪声。**

② **语言以英文为主。 这是“中文提示词效果差”的最直接原因**—— 不是模型不愿意学中文，是数据里中文太少。

③ **有版权和内容风险。** 2023 年，LAION-5B 因为包含违法内容被撤下。 **这件事之后，“数据从哪来”从技术问题变成了合规问题。**

8.34.2 第二代：重新标注

SDXL 的做法：训练一个 captioner，把训练图重新描述一遍， **得到更长、更准、含空间关系的 caption。**

这一步的收益很大—— 同样的图像，caption 的信息量提高了，$p(c \mid x)$ 变得更尖锐（见“概率的视角”一节）。

SD3 更进一步：用多模态大模型重述，caption 里开始出现“左边”“拿着”“背后”这类关系词。

代价：重述 10 亿张图，需要大量的推理算力。 **这是“数据工程”变成“模型工程”的一个标志。**

8.34.3 第三代：授权数据

Adobe Firefly、Getty 等选择只使用授权图库。

优点：合规、可商用。 **缺点**：规模小、分布偏（图库照片的风格很统一）。 **这直接影响了生成结果的多样性—— 用图库训练的模型，倾向于生成“商业摄影”风格的图。**

> ! 数据决定了三件事
>
> ① **语言覆盖**：数据里有什么语言，模型就懂什么语言。 ② **描述粒度**：caption 有多细，条件就能有多细。 ③ **风格分布**：图库图和互联网图，生成的“默认审美”完全不同。**
>
> **这三件事都不是“换个更好的编码器”能改变的。 它们是数据属性，不是模型属性。**

8.34.4 过滤：看不见的第四代

所有大规模数据集都经过过滤：

- **NSFW 过滤**：去掉色情、暴力内容
- **水印过滤**：去掉带明显水印的图
- **美学过滤**：用一个打分模型保留“好看”的图
- **去重**：防止同一张图出现几百次

过滤标准本身就是一种“条件”。 它决定了模型认为“什么是一张好图”—— 而这个判断，用户是看不见的。

这也是为什么不同模型有不同“默认风格”： 不是架构不同，是过滤器的偏好不同。

8.35 十五个常见问题

这些是讲这一章时被问得最多的问题。答案尽量短，需要展开的地方指回正文。

问一：为什么不用 RNN 或 CNN 编码文本?

因为要建模的是“任意两个词之间的关系”，而卷积的感受野有限、RNN 的路径太长。**自注意力一步就能让第 1 个词和第 77 个词直接交互**，这是它取代前两者的核心理由。**代价是 $O(M^2)$ 的计算**——77 个 token 时完全不是问题。

问二：交叉注意力和自注意力能互换吗?

不能直接换，因为 Q, K, V 的来源不同。 但可以**把两串 token 拼起来做自注意力**——那就是联合注意力（MMDiT）。 **所以“交叉”和“联合”是可以互相转化的，代价是计算量和双向性。**

问三：为什么是 77 个 token，不是 512?

因为 CLIP 用的是**可学习的绝对位置嵌入**，位置表只有 77 行。第 78 个 token 没有位置可加，只能截断。**T5 用相对位置，所以没有这个限制**——这是 SD3 换 T5 的直接原因之一。

问四：可以不训练文本编码器吗?

可以，而且大多数模型就是这么做的。**但“冻结编码器”不等于“文本不变”**：交叉注意力的 K, V 投影仍然在学。**冻的是词义，学的是“怎么读”**。

问五：为什么 CFG 要两次前向?

因为要同时算 ε_c 和 $\varepsilon_\varnothing$。**可以把两个条件拼成一个 batch 做一次前向**，计算量一样，但少一次 kernel 启动。**它不能被省成一次，因为两个分支的输入条件不同**。

问六：为什么真实模型的 w 常取 5–8，而本章只要 2?

因为真实模型的条件更弱、更嘈杂。**w 要大到“把条件方向从噪声里拉出来”**，而本章的玩具模型在 $w = 1$ 就已经完全学会了。**w 的最优值和任务的难度、条件的质量都有关，不能跨模型照搬**。

问七：提示词里的顺序重要吗?

对 CLIP 系模型，重要性低于你的直觉；对 T5 系模型，重要得多。机制在“CLIP 是怎么训出来的”那一节：对比学习很少要求区分词序。**但“不重要”不等于“没用”**——本章的实测显示，交叉注意力确实用上了顺序。

问八：为什么中文提示词效果差?

三个叠加的原因：**分词效率低**（一个汉字 1–3 个 token）、**训练数据以英文为主、编码器的中文语义弱**。**三者里最容易解决的是第一个**——把提示词写短、把关键词提前。

问九：注意力图能做图像编辑吗?

能，**Prompt-to-Prompt** 一类的方法就是用注意力图做局部替换。**但这不等于“注意力图解释了模型”**——它只是提供了一个“可操作的旋钮”，**旋钮有用不代表刻度是准的**。

问十：为什么 SD3 放弃了交叉注意力?

不是因为它不好，是因为**联合注意力在同样预算下多拿到了双向信息流**（多 3.8%）。**交叉注意力的优势是缓存**——文本的 K, V 算一次用 16 层。**联合注意力放弃了这个优势，换来了文本和图像的互相调整**。

问十一：文本条件能控制数量吗?

很弱。“两只猫”和“三只猫”在 CLIP 的表示里差别很小，而且训练数据里数量词的分布很偏。**这是组合性失败的一个典型例子**，和第 7 章“准确率饱和”一样，**需要专门的指标才能测出来**。

问十二：为什么负面提示词有效?

因为它参与的是 ε_{neg} 那一路，**在 CFG 的差里被减掉。它不是“告诉模型不要什么”，是“把生成方向往远离那个词的方向推”。所以负面词和主体语义太近时，会把主体一起推走**。

问十三：能不能只用池化向量（像 CLIP score 那样)?

能，但会丢掉位置信息。**SD1.5 只用序列，SDXL 两个都用**——序列进交叉注意力，池化进 AdaGN。**池化向量的优势是便宜**（每层一次 d 维调制），**劣势是不能逐位置**。

问十四：文本编码器能换成 LLM 吗?

可以，而且已经有模型这么做（用解码器式的 LLM 当文本塔）。**但要注意**：LLM 的输出是“下一个 token 的分布”，不是“句子表示”。**要么取隐藏状态，要么加一个投影头——这两种做法效果不同，需要实验**。

问十五：为什么“红色”会染到别的东西上?

因为交叉注意力是**一次 softmax、所有词竞争**。**“red”和“blue”在同一个位置上都有分数**，加权平均就成了紫色。**这不是模型不认识颜色，是没有机制强制“红色只属于方块”。解法是改条件结构（区域掩码、结构化引导），不是改措辞**。

8.36 从提示词工程到条件工程

这一章讲的是机制。最后这一节讲一个判断：提示词工程正在变成条件工程。

先看提示词工程都在做什么。

最典型的三类技巧，本质上都是在弥补同一个缺陷——文字这个接口没有结构。

第一类是词汇层面的：加“masterpiece”、加艺术家名字、加“8k uhd”。这些词之所以有效，是因为它们在训练数据里和高质量图像相关。**它们不描述任何东西，只是把生成往某个统计区域推。这是一种用“相关”代替“控制”的做法**。

第二类是数值层面的：`(word:1.3)`、CFG 的 w、负面提示词。它们调的是同一个东西——**某个条件在计算里的分量**。权重放大的是嵌入向量，w 放大的是条件与无条件的差，负面词放大的是反方向。**三者都是在“已有的方向上加减”，没有一个能创造新的结构**。

第三类是结构层面的：区域提示词、ControlNet、IP-Adapter。**这一类开始不一样了**——它们不再修改文字的权重，而是**增加新的条件通道。区域提示词增加了“哪里”，ControlNet 增加了“什么形状”，IP-Adapter 增加了“像什么”。**

从第一类到第三类，是一个从“调参”走向“改接口”的过程。这就是我说的“从提示词工程到条件工程”。

> ! 为什么这个转变是必然的
>
> **因为文字的信息带宽是有限的。**
> **77 个 token，每个 768 维，要描述一张 512 × 512 的图像。图像的自由度是 78 万个数，文本的自由度是 5.9 万个数—— 差了 13 倍。**
> **这不是“文字还不够好”，是“文字这个模态本身装不下”。**
> **所以每一次精度提升，最终都要靠“加一条新通道”：要位置，加框；要形状，加边缘图；要风格，加参考图。文字退回到它最擅长的位置——描述“是什么”，而不是“在哪里、什么形状”。**

那未来的接口长什么样?

我猜是“文本 + 结构化条件”的组合，而不是“更长的文本”。

结构化条件可能是一张场景图：物体、属性、关系，每个都是一条可寻址的记录。**模型对每个物体单独算注意力，再按关系组合。**

也可能是一个可编辑的中间表示：先生成一个粗糙的布局，再逐块细化—— **这和本章讲的“区域提示词”是同一个思路，只是把“掩码”换成了“可以反复修改的对象”。**

共同点是：条件从“一串不可分的 token”变成“一组可寻址的结构”。一旦可寻址，“绑定”就不再需要在注意力里隐式地学——它可以被显式地指定。

这也是本章实验的启示：交叉注意力能学出绑定，但要花数据和参数；一个显式的结构，可能只需要几行代码。

当然，显式结构也有代价：它要求用户提供更多信息，**而大多数人只想写一句话。**

所以短期内不会有一方取代另一方—— 而是“一句话”负责大多数场景，“结构化条件”负责少数高精度场景。

本章讲的交叉注意力，是这两者共同的底座。看懂了它，你才知道什么时候该加一条新通道，什么时候该把提示词写得更准。

8.37 怎么读一篇新的文生图论文

这一年新出的文生图论文，一个月可能有十几篇。有了这一章的框架，你可以用四个问题筛掉大部分。

问题一：条件从哪来?

是文本、图像、布局，还是多模态？如果是文本，用的是什么编码器、多少 token、什么维度?

这一步决定了它的“条件带宽”。一个用 CLIP-L 的模型和一个用 T5-XXL 的模型，后面的所有设计都不能直接比较。

问题二：怎么进模型?

在五个格子里找一个：拼接、AdaGN、交叉注意力、联合注意力、旁路。

如果作者说“我们提出了新的条件注入方式”，先看它属于哪一格，再看它改了什么。

大多数“新方法”是某一格的变体：换了门控、换了位置、换了归一化。**真正的结构性创新（比如从交叉到联合）很少。**

问题三：在哪一层?

同一栋楼，装在几层，效果差很多。

交叉注意力装在 UNet 的 16 层还是 4 层，参数量差 4 倍，效果也可能差很多。

论文里如果只说“我们加了条件”，没说“加在哪几层”，这篇论文的可复现性就要打折扣。

问题四：怎么训练?

三个数：丢弃率、batch size、caption 来源。

丢弃率决定 CFG 能不能用；caption 来源决定条件能力强不强；batch size 决定对比学习（如果有）的质量。

这三个数经常被放在附录里，但它们是理解结果的关键。

读完四个问题，再看结果

很多论文的“提升”，在四个问题里有现成的解释。
比如：“我们的方法在组合性上提升了 15%”—— **如果它同时把 caption 换成了更细的版本，提升可能来自数据，不是方法。**
控制变量，是读论文时最该做的事。 而这一章教你的，正是“怎么找出该控制的变量”。

8.38 延伸阅读

8.38.1 ① CLIP（2021）

Radford et al., Learning Transferable Visual Models From Natural Language Supervision。arXiv:2103.00020。
读它能得到什么：为什么文本和图像可以放进同一个空间。
读的时候注意：

- 它是**对比学习**：匹配的图文对拉近，不匹配的推远
- **它没有“生成”能力**，只判断“配不配”
- 77 token 的限制来自它的位置编码

8.38.2 ② Imagen（2022）

Saharia et al., Photorealistic Text-to-Image Diffusion Models with Deep Language Understanding。arXiv:2205.11487。
读它能得到什么：**用 T5 而不是 CLIP** 的理由，以及第 7 章讲过的 rescale。
关键实验：他们对比了 CLIP、T5、以及两者的组合， **发现纯 T5 更好**——尽管 CLIP 是图文对齐训练的。
这个结论值得记住：**“图文对齐”不等于“适合做条件”**。生成需要的是“文本内部的语义结构”，T5 的纯文本预训练反而更合适。

8.38.3 ③ Stable Diffusion（2022）

Rombach et al., High-Resolution Image Synthesis with Latent Diffusion Models。arXiv:2112.10752。
读它能得到什么：潜空间扩散的完整设计——VAE、交叉注意力的插入位置、以及为什么要下采样 8 倍。
读的时候注意：它的交叉注意力只在 UNet 的特定层，**不是每层都有**。论文里有那张结构图，对照本章“接进 UNet 的位置”那一节看。

8.38.4 ④ 关于注意力图的批评

Jain & Wallace (2019), Attention is not Explanation; **Wiegreffe & Pinter (2019)**, Attention is not not Explanation。
读它能得到什么：为什么“看注意力权重”不能直接当解释。
本章的负结果（相关 −0.079）是这一讨论在文本条件上的一个实例。两篇论文观点相反，**对照着读**更有价值。

8.38.5 ⑤ 组合性与绑定

关于 diffusion 模型组合能力的工作（如 Compositional Visual Generation、 以及各类 benchmark）讨论的就是本章的绑定问题： **模型能不能正确地把属性分配给对象**。
本章的二维实验是它的最小版本：两个属性、两个位置、四个组合。

8.38.6 六条机理，逐条展开

上面是“怎么发现”，这里是“为什么会这样”。

8.38.6.1 机理一：均值是一个置换不变的运算

定义：f 是置换不变的，如果 $f(x_1, ..., x_n) = f(x_{\pi(1)}, ..., x_{\pi(n)})$ 对任何排列 π 成立。
均值就是这样的运算：$\frac{1}{2}(a+b) = \frac{1}{2}(b+a)$，**一模一样**。
所以“用均值当条件”这个设计，在数学上就不可能区分顺序。不是训练不够、不是数据不够——**是信息在那一步就被销毁了**。

> ℹ 一个可以推广的判断
>
> **任何“池化”都在丢信息**：
>
> | 池化 | 丢掉什么 |
> |---|---|
> | 均值 | 顺序、数量结构 |
> | 最大 | 除最大值外的一切 |
> | 求和 | 顺序（和均值一样） |
>
> **注意力池化（用 query 加权）不丢顺序**——因为权重依赖位置。 **这是交叉注意力和均值池化的本质区别，不是“参数量”的区别**。

8.38.6.2 机理二：注意力本身是置换等变的

如果不加位置嵌入，K 里两个 token 除了内容没有区别。
把 token 顺序换一下，K 只是换了个顺序。 **加权求和 $\sum_k a_k v_k$ 对 k 的排列不变—— 输出逐位相同，不是“跟着换”**。
这一点被实测证实了：`no_pos` 的对调差是 0.000。 **它的准确率仍然显示 0.602 / 0.397，因为同一批样本被从两边数**。
所以顺序信息必须显式注入：

- 位置嵌入（本章的做法）
- 因果掩码（CLIP 用的）
- 或者让 query 本身带位置

“用了注意力就有顺序”是一个常见的误解。

8.38.6.3 机理三：零向量在模型的训练分布之外

模型训练时见过的“空条件”是 `emb[NULL]`，一个特定向量。
零向量是另一个点——**它可能在训练分布之外，也可能恰好落在某个真实条件附近**。
第 7 章的实测：零向量当空条件，越界 100%、多样性 0， **而准确率 1.000**。
机理：CFG 的公式是 $\varepsilon_u + w(\varepsilon_c - \varepsilon_u)$。 如果 ε_u 是一个“模型不知道怎么处理”的输入， 它的输出就是一个任意的方向——**放大这个差，等于放大一个随机方向**。

8.38.6.4 机理四：换 tokenizer 等于换了一门语言

空字符串在不同 tokenizer 下切出的特殊 token 不同：

- CLIP：`[BOS] ... [EOS]`（2 个）
- T5：可能是一串 `[PAD]`
- 某些 tokenizer：只有 `[CLS]`

这些 token 的嵌入完全不同，编码结果自然不同。
而 UNet 是在“某一个编码器”的输出上训练的—— 换了编码器，输入的分布变了，**UNet 的权重不再匹配**。
这就是“换编码器必须重训 UNet”的机制。

8.38.6.5 机理五：77 是一个硬截断

CLIP 的位置编码只有 77 个位置。 第 78 个 token 没有位置可加。
实现上通常直接截断（或者用滑动窗口，但那样会丢全局）。
后果：句尾的修饰语被丢掉。**而中文提示词更容易超限**（机理见前面“词表”那一节）。
判据：把提示词从 60 token 加到 90 token，如果输出**完全没变**，说明后半段被截断了。

8.38.6.6 机理六：注意力权重只是链条上的一环

交叉注意力的输出是 $\sum_k a_k v_k$。 信息有三条路：

1. **权重 a_k**
2. **值 v_k**（含位置嵌入）
3. **query 本身**（含点的坐标）

本章的测量只看了第 1 条，发现它几乎均匀、与绑定无关。**但第 2、3 条仍然携带着信息**，而且后面的 MLP 可以把微小差异放大。
所以“注意力图不可解释”不等于“注意力没起作用”—— 它只是说“权重这一条路不是解释”。

> ! 这六条机理的共同点
>
> **它们都不是“训练不充分”，是“结构上做不到”。**
>
> - 均值：信息被销毁
> - 无位置：对称性未被打破
> - 零向量：输入在分布外
> - 换 tokenizer：训练/推理不匹配
> - 截断：信息根本没进来
> - 误读注意力：解释的路径选错了
>
> **前五条改数据或改结构就能解决；第六条改的是“你怎么看它”。**

8.39 从二维到真实模型：三处不同

本章用两个 token、两个点、四个类。真实模型用 77 个 token、4096 个潜位置。
哪些能搬?

8.39.1 第一处：token 的数量

本章 2 个，真实模型 77 个。
这改变的是“绑定”的难度： 两个 token 只有两种排列，模型很容易学； 77 个 token 的排列组合是天文数字，**模型只能学会“常见的那些”**。
这解释了为什么真实模型在“三个以上对象的关系”上经常出错—— 不是机制不对，是训练分布覆盖不到。

8.39.2 第二处：位置的含义

本章的两个位置是“左点”和“右点”，是固定的两槽。
真实图像的 4096 个位置是二维网格，而且经过多层卷积/注意力之后，每个位置感受野不同、语义不同。
所以真实模型里“位置”不是一个编号，是一个二维坐标 + 层级。位置编码通常是二维的（或者用 RoPE 的二维版本）。

8.39.3 第三处：文本编码器本身

本章的“编码器”只是一张查找表——没有自注意力、没有上下文。
真实编码器有 12–24 层 Transformer， 所以“红色”的向量会受“深红色”“红色背景”影响。

!这三处不同，对应三种失败

差异	对应的失败
token 多	对象多于两个时绑定出错
位置是二维网格	空间关系（“在……左边”）难精确
编码器有上下文	换 tokenizer / 换模型后行为漂移

本章能验证的是机制（绑定需要逐位置的条件），不能验证的是规模效应。

8.40 五个常见误解

8.40.1 误解一：“提示词写得越长越好”

错，而且有两个上限。

硬上限：CLIP 的 77 个 token。**超出的部分直接消失**，而且丢掉的往往是句尾——**人写提示词时习惯把重要的放最后。**

软上限：编码器的注意力会被稀释。77 个 token 里如果 60 个是废话，**真正有用的词在交叉注意力里的份额会变小。**

实践含义：**短提示词 + 具体名词**，通常好过长提示词 + 形容词堆砌。

8.40.2 误解二：“换一个更强的文本编码器就能提升效果”

不一定，因为 UNet 是在旧编码器的输出分布上训练的。

换编码器之后：

- 输出维度可能变（768 → 4096）
- 语义空间完全不同（CLIP 的“红”和 T5 的“红”不是同一个方向）
- 空条件的编码也变了

所以换编码器 = 换了一门语言 = 必须重训 UNet。SD3 换 T5 是重训过的，不是“插上就用”。

8.40.3 误解三：“同义词可以随便替换”

在嵌入空间里，同义词不是同一个点。

“红色”和“赤色”的向量不同，**模型对它们的行为可能不同**—— 尤其当训练数据里“赤色”很少时。

这也解释了一个现象：某些“魔法提示词”（比如特定的艺术家名字）效果特别好，**因为它们在训练数据里有密集的对应。**换成同一个人名的另一种写法，效果可能差很多。

8.40.4 误解四：“提示词权重（`word:1.3`）是模型理解语法”

不是。大多数实现里，权重语法在**编码器之前**就被处理掉了：

1. 解析 `word:1.3`
2. 单独编码 `word`
3. 把它的向量乘 1.3
4. 拼回序列

模型看到的不是“1.3 这个数”，是一个被放大过的向量。

所以权重的效果依赖实现：不同管线对权重的归一化方式不同，**同一个 1.3 在不同模型上强度不同。**

8.40.5 误解五：“注意力图能告诉我模型在看哪个词”

本章的实测：不能。

权重 0.44–0.58，与绑定属性相关 −0.079，用它猜颜色 47.0%。

能告诉你的是：“这个位置和这个词的 key 比较相似”。**不能告诉你的是**：“这个位置的信息来自这个词”。

区别在 V 和后面的 MLP——信息可以从别处流进来。

五个误解的共同点

它们都把“文本侧”和“图像侧”当成可以独立优化的两部分。

- 一：以为编码器能装下任意长
- 二：以为编码器可以单独升级
- 三：以为词义是稳定的
- 四：以为语法被理解了
- 五：以为内部权重等于因果

实际上，文本和图像是通过“训练时一起见到的分布”绑定的。任何一侧单独改变，都会打破这个绑定。

8.41 本章没有回答的问题

每一章都应该留下几个“知道怎么查、但这一章没查”的问题。下面每个问题都配了一个可执行的实验设计——它们是可以直接开工的题目。

8.41.1 问题一：为什么注意力图不可解释?

已知：权重几乎均匀（0.44–0.58），与绑定属性的相关 −0.079，用它猜颜色 47.0%，而输出准确率 1.000。

未知：是“权重本身没信息”，还是“权重有信息但被下游掩盖”?

实验设计：

1. **硬注意力**：把 softmax 换成 argmax（或 straight-through），重训
2. **权重冻结**：训练后固定注意力权重，只训后面的 MLP
3. **权重干预**：人为把左点的权重设成 0.9/0.1，看输出是否改变

判据：

- 如果硬注意力**仍然能绑定**，说明绑定不依赖权重的精细分布
- 如果冻结权重后**性能不变**，说明权重根本不是瓶颈
- 如果人为干预**能改变输出**，说明权重是因果链的一环，只是自然训练时不承担这个角色

我的预期（可证伪）：硬注意力会掉点，但不会归零；冻结权重会掉很多——因为权重仍然携带“哪个点是哪个点”的信息，只是这些信息**不是以人类可读的方式编码的**。

8.41.2 问题二：三个以上的对象会怎样?

已知：两个对象、两种排列，交叉注意力 1.000，均值池化 0.550/0.449。

未知：对象数从 2 增加到 3、4、5 时，准确率怎么衰减?

实验设计：

1. 槽位数 $k = 2, 3, 4$，每个槽 c 种颜色
2. 类别数 c^k：$4, 8, 16$
3. 每种配置训 3 个种子，每类 300 个样本
4. 画“对象数 vs 双槽/全槽准确率”

判据：如果准确率随 k **指数衰减**，说明模型在学“每个排列”而不是“绑定规则”；如果是**线性衰减**，说明它学到了可组合的规则。

这个区分很重要：前者意味着“再加一个对象就要加数据”，后者意味着“规则可以外推”。

8.41.3 问题三：位置嵌入是否足够?

已知：去掉 token 的位置嵌入，准确率从 1.000 掉到 0.602/0.397，但对调实验显示输出**逐位相同**——那 0.602 是“同一批样本被从两边数”的假象。
未知：不同的位置表示，哪一种最适合绑定?
实验设计：四种编码，同一套训练：

编码	形式
无	$k_i = e_{w_i}$
可学习绝对	$k_i = e_{w_i} + p_i$
正弦绝对	$k_i = e_{w_i} + \mathrm{Sin}(i)$
相对	在打分里加 $b(i-j)$

判据：不光看准确率，还要看**对调差**。准确率会被“互补假象”骗，对调差不会。

8.41.4 问题四：空字符串的编码有多稳定?

已知：空条件必须是“训练时见过的那个向量”，零向量会崩。
未知：不同编码器对 `""` 的输出，彼此有多远？这个距离会不会改变最优 w?
实验设计：

1. 取 CLIP-L、CLIP-G、T5-XXL 三个编码器
2. 编码 `""`，得到三个向量
3. 算两两余弦相似度
4. 在同一个 UNet 上（如果可能）扫 w

判据：如果余弦相似度低于 0.5，说明“空条件”在三者之间**不可迁移**—— **换编码器必须重新定标 w，不能沿用旧值**。

8.41.5 问题五：文本编码器该不该微调?

已知：大多数模型冻结编码器，理由是省显存、省配对数据、稳定。
未知：微调带来的收益，值不值这个代价?
实验设计：

1. 固定 UNet，只微调文本编码器的最后 n 层（$n = 0, 2, 4, 8$）
2. 用同一批图文对，同样的训练步数
3. 测三个指标：绑定准确率、CLIP score、生成多样性

判据：如果 $n = 2$ 就能带来明显提升，说明“冻结全部”是过度保守；如果必须 $n = 8$ 才有收益，而多样性明显下降，说明冻结是对的。

8.41.6 问题六：能不能“学”出一个绑定头?

这是本章最想做但没做的一个实验。
想法：在交叉注意力之后加一个小的“绑定模块”，显式地把 token 和位置配对，再喂给 UNet。
它相当于把“绑定”从隐式学习**变成**显式结构**。
风险：真实提示词没有“哪个词属于哪个对象”的标注，**这个模块没有监督信号**。
一个可能的弱监督：用现成的目标检测器，在生成图上检测物体，反推“哪个 token 应该对应哪个框”。
如果这个思路成立，它会是一个比“加大模型”更便宜的改进方向。

> **i** 这六个问题的共同点
>
> **它们都不是“再多训一会儿”能回答的。**
> **每一个都需要改结构、换数据、或者设计新的测量。这就是“研究问题”和“工程问题”的区别**—— 工程问题是“怎么让它跑起来”，研究问题是“为什么它这样跑”。

8.42 每一节的边界

结论	前提	什么时候失效
均值池化会丢绑定	条件里有顺序/对应关系	只有一个对象时不丢（RR/BB 是 1.000）
交叉注意力能绑定	K 里有位置信息	去掉位置嵌入后对调差 0.000（准确率 0.602 是互补假象）
注意力图不可解释	这个 2 层 MLP 的模型	更深的网络、更大的数据可能不同
空串是“被训练过的条件”	训练时做过文本丢弃	没做过丢弃 → 空串没被训练过
CFG 收益为零	$w = 1$ 已饱和	更难的任务上仍有收益
77 token 上限	CLIP 的位置编码	T5/Qwen 没有这个限制

最需要警惕的是第三行。
本章的“注意力图不可解释”是在一个 10 万参数的玩具模型上测的。**它不能推广成“所有注意力图都没用”**——第 4 章在语言模型上就找到了 可解释的头。
能推广的是方法论：别假设注意力图能解释，先做一次消融验证。

8.43 术语对照

中文	English	代码里
分词器	tokenizer	`tokenizer`
词嵌入	token embedding	`embed_tokens`
位置编码	positional encoding	`position_ids`、`pos_embed`
交叉注意力	cross-attention	`attn2`、`encoder_hidden_states`
自注意力	self-attention	`attn1`
潜空间	latent space	`latents`、`latent_image`
缩放因子	scaling factor	`vae.config.scaling_factor`
空字符串	empty prompt	`""`、`uncond_prompt`
绑定	binding / attribute binding	—

⚠ 一个容易混的字段名

`encoder_hidden_states` 在 `diffusers` 的 UNet 里指**文本**，不是图像编码器的隐藏状态。
看到它出现在 `attn2`（交叉注意力）的输入里，就是文本那一路。

8.44 文本条件怎么评估

不能只看“像不像”，也不能只看 CLIP score。

8.44.1 三种评估，对应三个问题

问题	指标	它能发现什么
图和文字配不配?	CLIP score	整体语义偏离
属性有没有绑对?	受控提示词的槽位准确率	绑定失败
改一个词，输出变不变?	消融差分	文本有没有被用起来

本章的“双槽准确率”就是第二种——它要求两个位置都摆对， 而不是“整体像不像”。

8.44.2 CLIP score 的盲区

CLIP score 是“整图 vs 整句”的相似度。
它看不见:

- **顺序**:“猫在狗左边”和“狗在猫左边”的 CLIP score 可能一样高
- **数量**:“两只猫”和“三只猫”
- **否定**:“没有猫”

所以一个模型可以 CLIP score 很高，绑定全错。

> ⚠ 和第 7 章“准确率会饱和”是同一类问题
>
> **CLIP score 是一个标量，它把“哪里对、哪里错”平均掉了。**
> **当一个指标只给一个数，它必然丢掉结构信息。**
> **所以**: 至少加一个“结构化的”指标——槽位准确率、数量准确率、 或者简单的消融差分。

8.44.3 真实模型怎么测：把提示词拆成原子事实

本章的“双槽准确率”在玩具上是够的。真实模型怎么测?
核心思路是一样的: **把一句话拆成若干条可以独立判定的“原子事实”。**
例: `a red cube and a blue sphere`

原子事实	怎么判
有一个方块	检测器找到 cube
有一个球	检测器找到 sphere
方块是红的	在方块框里测主色
球是蓝的	在球框里测主色

四条全对，才算这张图“完全符合”。
这就是 2023 年之后几个组合性 benchmark 的基本做法:

- **T2I-CompBench**: 用 VQA 模型回答“这个物体是什么颜色”之类的问题
- **GenEval**: 用目标检测器定位物体，再判颜色/数量/位置
- **两者共同的假设**: **一个足够好的检测器 + 一个足够好的 VQA，可以代替人**

> ⚠ 这个假设会带来三种偏差
>
> ① **检测器漏检 = 判成“错”**。 小物体、遮挡、非典型姿态，检测器经常找不到。 **模型可能画对了，但被判错。**
> ② **VQA 模型有它自己的先验**。 问“这个球是蓝色的吗”，VQA 可能因为“球通常是某种颜色”而答是。 **这是把偏差从一个模型转移到了另一个模型。**
> ③ **原子事实之间不独立**。 “有一个方块”和“方块是红的”其实共享同一个检测框。 **分开判会把一个错误数两次。**
> **所以 benchmark 的分数只能横向比较，不能当成“准确率”。**

一个更稳的做法：报告每个原子事实的通过率，而不是总分。
总分会把“颜色全对、数量全错”和“颜色错一半、数量对一半”压成同一个数。而这两种失败对应完全不同的修法。

8.44.4 用 VLM 当裁判：能做，但要校准

近两年很多工作直接用多模态大模型打分：把图和提示词一起给 VLM，问“符合吗，1–5 分”。
优点：不需要预先定义原子事实，能处理复杂提示词。
缺点：

- **分数漂移**：同一个模型，换个问法，分数变
- **偏宽松**：VLM 倾向于给“大致符合”高分
- **不可复现**：模型版本一变，历史分数不可比

校准的办法：

1. 用一批人工标注的样本，测 VLM 和人的一致率
2. **报告一致率**，不只报告分数
3. 固定 VLM 版本，写进实验记录

如果一致率低于 80%，这个裁判不能用。

8.44.5 本章的测量为什么不需要这些

因为本章的数据是四个点，判定是“符号对不对”。
这是一个“零歧义”的任务：x 坐标的符号就是颜色，没有检测器、没有 VQA、没有主观判断。
这正是选择玩具模型的价值：把“测量”这件事从问题里去掉，只剩下“机制”。真实模型上，测量本身就是一个研究方向。

8.44.6 一个五分钟的消融

不需要标注数据，也不需要训练分类器：

1. 固定种子，用提示词 A 出 4 张
2. 固定种子，把 A 里一个词换掉（比如“红”→“蓝”）出 4 张
3. 比较两组的差异

如果几乎一样，说明那个词没被用起来。
这比看注意力图可靠——它是行为证据，不依赖内部解读。

8.45 本章数字总表

所有数都可以用 `notebooks/run_ch8_*.py` 复现。这一节按“来源”归类，方便核对。

8.45.1 来自 `run_ch8_case.py`

量	值	说明
均值池化参数	67,332	3 层 MLP
交叉注意力参数	100,740	同样 3 层
训练损失（均值 / 交叉）	0.0476 / 0.0295	6000 步，同数据同优化器
RB 准确率（均值 / 交叉）	0.550 / 1.000	5 个种子 × 300 样本
BR 准确率（均值 / 交叉）	0.449 / 1.000	同上
注意力相关	−0.079	与左色
用注意力猜色	47.0%	二元分类

8.45.2 来自 `run_ch8_ablation.py`

量	值
均值池化对调差	0.000
交叉注意力对调差	0.972

8.45.3 来自 `run_ch8_ablate2.py`

量	值
`no_pos` RB / BR	0.602 / 0.397（和 0.999）
`no_pos` 对调差	**0.000**
`no_xy` RB / BR	1.000 / 1.000
`no_xy` 对调差	0.931
学习曲线（mean，500→6000 步）	0.400 / 0.443 / 0.497 / 0.497 / 0.500
学习曲线（cross）	1.000（每一步）
注意力熵（full / no_xy / no_pos）	0.984 / 0.998 / 0.685 bit

8.45.4 来自 `run_ch8_capacity.py`

层数	参数	损失
3	67,332	0.0476
4	83,844	0.0469
5	100,356	0.0480
6	116,868	0.0482

8.45.5 来自 `bpe_demo.py` / 留出集测量

合并次数	留出集汉字/token	77 token 装得下
300	0.625	48 字
1000	0.967	74 字
2000	1.210	93 字

> **i** 一张总表的价值
>
> **它让你能一眼看出“哪些数互相矛盾”。**
> **比如**：`no_pos` 的 0.602 和它的对调差 0.000 放在一起，**矛盾立刻暴露**——准确率说“部分成功”，干预说“完全失败”。
> **如果这两个数分散在两节里，读者很可能只记住 0.602。**
> **所以“数字总表”不是附录，是**校验工具：任何一对数放在一起读，都应该讲同一个故事。**

8.46 一张速查卡

贴显示器旁边的那种。

8.46.1 三步

步骤	做什么	关键
分词	字符串 → token	词表大小决定“一句话多长”
查表	token → 向量	和语言模型一样
编码	向量带上下文	自注意力 + FFN

8.46.2 交叉注意力

量	来自
Q	图像位置（+ 时间步）
K	文本 token（+ 位置嵌入）
V	文本 token（+ 位置嵌入）

8.46.3 八个数

量	值
均值池化 RB/BR	0.558 / 0.442（和 0.999）
交叉注意力 RB/BR	1.000 / 1.000（和 2.000）
对调差：完整 / 均值 / 无位置 / 无坐标	0.972 / 0.000 / 0.000 / 0.931
注意力相关（与左色）	−0.079
用注意力猜色	47.0%
注意力熵：完整 / 无坐标 / 无位置	0.984 / 0.998 / 0.685 bit
损失：均值（同参数）vs 交叉	0.0480 vs 0.0295
CLIP 上下文	77 token

8.46.4 五个坑

1. **均值池化** → 顺序信息被销毁
2. **没有位置嵌入** → 输出逐位不变；准确率 0.602 是互补假象
3. **零向量当空条件** → 输入在分布外
4. **换 tokenizer 不复验** → 空条件变了
5. **用注意力图判断** → 尖锐的权重也可能零信息（熵 0.685 却不绑定）

如果只能记一行：先看镜像类准确率之和——等于 1 就是没有顺序，等于 2 才是真的绑上了。

8.47 这一章的方法论

为什么用“两个点、两个词、四个类”这么小的设定?
因为要回答的问题很具体：**顺序信息能不能传到输出。**
这个问题的答案有三种证据，强度依次递增；最后再加一条边界。

8.47.1 第一层：代数

均值是置换不变的。 这一点不需要实验——$\frac{1}{2}(a+b)=\frac{1}{2}(b+a)$ 是恒等式。
所以“均值池化会丢顺序”是一个定理，不是一个发现。
但定理只告诉你“会丢”，不告诉你“丢了之后表现成什么样”。

8.47.2 第二层：统计

训练两个模型，测四类的准确率。
本章的结果：均值池化在 RB/BR 上 0.550/0.449， 交叉注意力 1.000/1.000。
这一层告诉你“丢多少”，但不告诉你“模型内部怎么走的”。

8.47.3 第三层：行为

对调两个词，看输出变不变。

- 均值池化：条件向量相同 → 输出**逐位一致**
- 交叉注意力：输出**翻转**

这一层最强，因为它不依赖任何内部量的解读。

> ! 为什么第三层比“看注意力”强
>
> **注意力权重是内部量，它的含义依赖你怎么解释。** 本章的权重看起来均匀，但绑定确实发生了——**解释失败了，行为没失败。**
> **对调词是外部干预**：你改变了输入，观察输出。**因果方向清楚，不需要解释中间发生了什么。**
> **这也是第 4 章那个教训的延续：要证明“某部分在起作用”，改它、看输出，而不是看它的数值。**

8.47.4 第四层：规模——这一层本章给不了

前三层都能在玩具上完成。第四层不能。
“这个机制在 10 亿参数的模型上还成立吗?” 本章的每一个结论，都只在 10 万参数、4 维数据上验证过。
规模可能改变什么?
① 数据量可以让“学不会”变成“学得会”。 本章的 `no_xy` 在 2 个对象上成功了。**4 个对象时，下游 MLP 要学的映射从 2! 种变成 4! 种—— 可能需要的数据量差一个数量级。**
② 更深的网络可以补偿更弱的条件。 本章的下游是 3 层 MLP。**如果换成 30 层，均值池化的条件向量仍是同一个—— 它不会因为网络更深而获得信息。 但“全局条件 + 下游路由”这条路，会随着深度变得更可行。**
③ 真实数据的分布比四个点复杂得多。 本章的“左/右”是坐标的符号，**真实图像的“左/右”要先被卷积层理解成空间概念。 这一步的难度，玩具模型完全没有体现。**

> ⚠ 所以本章的结论要这样读
>
> **能推广的是“必要条件”，不是“数值”。**
>
> - **能推广**：条件必须携带顺序；池化会销毁它；注意力权重不是因果
> - **不能推广**：0.550 这个数、0.972 这个差、熵 0.984 这个值
>
> **看到一篇论文说“我们的方法在 XXX 上提升了 3 个点”，先问：它改变的是“信息”，还是“优化”? 本章的容量对照（参数相同、损失差 63%）就是区分这两者的办法。**

8.47.5 三个设计准则

如果你要做一个类似的“最小实验”，这三条值得抄。
准则一：让“错误做法”在结构上必然失败，而不是碰巧失败。

均值池化不是“训练不好”，是“数学上不可能” —— $\frac{1}{2}(a+b)=\frac{1}{2}(b+a)$ 是恒等式。 **这样你不需要解释“为什么它没学会”，只需要解释“它为什么不可能学会”。**

反过来，如果你的错误做法只是“效果差一点”，你就无法排除“再多训一会儿就好了”。

准则二：把“容量”和“信息”分开。

本章做了两次：训练损失（0.0476 vs 0.0295）和容量对照（参数加到 11.7 万，损失不变）。

没有这两步，“均值池化更差”可以被解释成“它参数少”。有了这两步，只剩一个解释：条件里的信息少了。

准则三：至少有一个“不依赖准确率”的指标。

本章有三个：对调差、镜像类准确率之和、注意力熵。

为什么需要它们？ 因为 0.602 那个假象 —— **准确率会在“互补计数”时给出一个看起来像“部分成功”的数。 换一个指标，假象立刻消失。**

8.47.6 这套方法可以搬到哪

任何“条件怎么进模型”的问题，都可以用同样的三层：

1. **代数**：这个操作对什么不变？（均值对顺序不变、注意力对位置等变）
2. **统计**：在受控数据上测准确率
3. **行为**：改条件、看输出

再加一条贯穿始终的原则：每次看到一个“变好了”或“变差了”的数，先问“有没有另一种解释”。

本章出现过两次“另一种解释”：均值池化的损失差可能是容量（用容量对照排除），无位置嵌入的 0.602 可能是部分绑定（用对调实验排除）。

两次都是“看起来合理”的解释，两次都被实验推翻了。

这就是这一章真正想教的东西：不是交叉注意力怎么算，是“怎么知道自己算对了”。

8.48 动手练习

六个练习，从“复现一个数”到“自己设计一种条件”。前三个只需要本章的代码，后三个要改结构。

8.48.1 练习一：复现对调实验

跑 `run_ch8_ablation.py`，确认你能得到：

编码器	对调后的最大逐元素差
均值池化	0.000
交叉注意力	0.972

然后只改一个变量：把采样种子从 7 换成 8，重跑。

预测：两个数会变吗？

提示：想清楚“逐位一致”是**结构性**的还是**运气好**。如果是结构性的，换任何种子都应该是 0.000。**如果你测出非零，先检查是不是浮点求和顺序造成的**（10^{-7} 量级不算）。

8.48.2 练习二：一个看起来对、其实错的修法

有人说：“均值池化丢顺序，那我改成**求和 + 位置嵌入**不就行了？ 每个 token 加上它的位置，再求和。”

任务：

1. **先不跑代码**，写出 RB 和 BR 的条件向量，看它们是否相同。
2. 再改成“不池化，直接返回两个位置的向量”，重跑。

提示：求和是交换的；位置嵌入是按**下标**加的，不是按**词**加的。

> **i** 这道题的答案
>
> 设 RB 的两个词是 $w_0 =$ 红, $w_1 =$ 蓝，BR 是 $w_0 =$ 蓝, $w_1 =$ 红。
> **求和 + 位置**：
>
> $$c_{\text{RB}} = (e_{\text{红}} + p_0) + (e_{\text{蓝}} + p_1) \tag{8.55}$$
>
> $$c_{\text{BR}} = (e_{\text{蓝}} + p_0) + (e_{\text{红}} + p_1) \tag{8.56}$$
>
> **两式完全相等**。加法交换，p_0 和 p_1 只是被重新分组。
> **所以这个“修法”一点用都没有。问题不在“有没有位置信息”，在“池化这个操作把位置抹掉了”**。
> **正确的修法是取消池化**：保留 $(e_{w_0} + p_0, e_{w_1} + p_1)$ 这个**序列**，让下游自己决定怎么用。

这道题的价值：它区分了“**有位置信息**”和“**用了位置信息**”。两者不是一回事。

8.48.3 练习三：加第三个槽

本章是“左、右”两个槽。改成三个槽（左、中、右），四个颜色。
任务：

1. 改 `make_two_point_data`，从 4 维改成 6 维，类别从 4 个变成 8 个
2. 两种编码器各训一次
3. 报告每个类的槽位准确率

预测：

- 均值池化在“三种颜色各不同”的类上，准确率应该是多少？
- 交叉注意力会掉点吗？为什么？

提示：三个槽、每种颜色可重复，排列数是 $3! = 6$ 对于三个不同色。**均值池化对这 6 种排列给出同一个向量**。
第二个问题的答案取决于数据量：交叉注意力的参数量没变多少，但**要学的对应关系从 4 种变成了 8 种。样本效率会下降**。

8.48.4 练习四：把 NULL token 换成零向量

本章的空条件是学出来的 NULL token（编号 2）。
任务：把它换成全零向量，重跑 CFG 扫描（$w = 1, 3, 5, 10$）。
预测：参考第 7 章的实测——零向量当空条件，$w = 3$ 时越界 100%、多样性归零。
然后解释：为什么“全零”比“学出来的 NULL”更糟？**提示**：零向量在嵌入空间里是一个特定的点，它离哪个词近、离哪个词远，是**偶然的**。

8.48.5 练习五：用掩码代替学习

本章的绑定是“学”出来的：模型自己学会左点看第一个词。
任务：不给它学——**直接加硬掩码**。

```
# 左点只能看 token 0，右点只能看 token 1
mask = torch.tensor([[[1., 0.], [0., 1.]]])   # (1, 2 点, 2 词)
att = att.masked_fill(mask == 0, float("-inf"))
```

在 softmax 之前加这一行，重训，测四类准确率。
预测：应该 1.000——因为掩码直接告诉它“左点听第一个词”。
然后想一个问题：既然掩码这么有效，**为什么真实模型不用它？**
提示：掩码假设“第 0 个词对应左边”。**但真实提示词里，“左边”这个词可能出现在任何位置。掩码编码的是位置的对应关系，不是语义的对应关系**。

8.48.6 练习六：写一个绑定诊断器

把本章的诊断动作做成一个类：

```
class BindingDiagnoser:
    def swap_test(self, model, sch, tok_a, tok_b): ...
    def classify(self, samples): ...
    def report(self): ...
```

要求：

- `swap_test` 返回**最大逐元素差**，不是平均差
- `classify` 要能处理“两个槽都摆对”和“只有一个槽对”
- `report` 要给出三档判断：**绑定成功 / 部分绑定 / 完全没绑定**，并且**每一档都要有数字依据**

判据建议：

判断	条件
完全没绑定	对调差 $< 10^{-6}$，且两类准确率之和 ≈ 1，各 ≈ 0.5
部分绑定	对调差明显 > 0，但准确率在 0.6–0.95
绑定成功	对调差 > 0.5，准确率 ≥ 0.99

做完这些练习之后

六个练习大约四小时。

- **练习一、二**：把“置换不变”从一个词变成你能手推的东西
- **练习三**：看到规模效应——同样的机制，更多的组合，更难的优化
- **练习四**：把第 7 章的结论迁移到文本条件
- **练习五**：理解“掩码”和“学习”的分工
- **练习六**：一套能直接用在真实模型上的诊断工具

如果只做一个，做练习二。它用一个五行的代数推导，拆掉了“加位置嵌入就能救池化”这个很常见的错觉。

8.49 把这一章讲给别人听

同一份内容，三种时长。

8.49.1 五分钟版

三句话：

1. **文本编码器把一句话变成一串向量，不是一个向量**——77 个 token 就是 77 份条件。
2. **交叉注意力让每个图像位置各自去问一遍**；均值池化让所有位置共用同一个答案，所以 RB 和 BR 在它眼里是同一个条件（0.550 / 0.449）。
3. **注意力权重不能解释输出**：权重几乎是均匀的（相关 −0.079），但对调两个词，交叉注意力的输出翻转 0.972，均值池化的输出差**恰好 0.000**。

如果对方只问一个问题，多半是“那提示词怎么写才管用”。**答**：写模型训练数据里常见的句式，把最重要的词放前面，**别指望用文字精确控制位置——那需要另一条通道。**

8.49.2 三十分钟版

在五分钟版之上加两块：
① **手算的那个例子。**两个 token、两个点，左点的注意力从 0.893 翻到 0.781。**这一组数能让人看见“绑定的全部机制就是下标和内容被分开”。**

② **空条件的那个坑。** 零向量当空条件，$w = 3$ 就崩。 **它说明“文本条件”不是一个可以随手拼出来的张量，是一个必须被训练过的分布。**

8.49.3 两小时版

完整讲，但顺序调整过：

1. 先讲**失败**（均值池化在 RB/BR 上 50/50）
2. 再讲**机制**（手算交叉注意力）
3. 然后**实测**（1.000 vs 0.550，以及对调实验的 0.000 / 0.972）
4. 最后**边界**（注意力图不可信、空间控制要另开通道）

先失败后机制，比“先公式后例子”更容易让人记住—— **因为“同一个词袋、两种顺序”本身就是一个让人想不通的问题。**

讲的时候最容易被追问的三个问题

① **“那我把提示词写详细一点，不就能控制顺序了吗?”** 不能。**详细只是增加 token，不改变“token 之间没有修饰关系”这个事实。**“红方块和蓝球”写成 50 个词，交叉注意力里红和蓝仍然在竞争。 **要解决它，得换条件结构（区域掩码、结构化引导），不是换措辞。**

② **“注意力图看起来很正常，为什么输出还是错的?”因为权重只是链条上的一环。** V 里带着位置，query 里带着坐标， 后面还有 MLP。**权重均匀不代表信息没传过去。 想判断，就做干预实验，别看热图。**

③ **“换个更强的文本编码器有用吗?” 有，但必须重训 UNet。**编码器换了，条件分布就换了。 **“插上就用”只在两个编码器输出分布几乎相同时成立**，而那种情况很少。

8.50 如果你只有一小时

这一章约四万字。时间不够时，按这个顺序读。

时间	读什么	得到什么
0–12 分	案例 + 手算交叉注意力	绑定问题的全部机制
12–25 分	实测表 + 对调实验	0.550 与 1.000，0.000 与 0.972
25–40 分	注意力图 + 空字符串	两个最容易踩的坑
40–50 分	潜空间 + 端到端八步	真实模型长什么样
50–60 分	速查卡 + 排错十条	带走一页纸

可以跳过的：CLIP 的训练目标、条件注入的五代谱系、caption 工程、延伸阅读。
不要跳过的：对调实验与空条件。**前者教你“怎么证明文本真的起作用”，后者教你“为什么 $w > 0$ 会崩”。**

8.51 一句话

条件不是“一句话”，是“带顺序的一份信息，加上一条取用的路径”。
把它拆开看：

- **带顺序**：均值池化把两句话压成同一个点，信息在那一步就没了
- **取用路径**：交叉注意力让每个位置自己来取；`no_xy` 让下游 MLP 用坐标来取
- **两者缺一不可**：只有顺序没有路径，或者只有路径没有顺序，都绑不上

三句话对应三个坑：

- 不懂“带顺序” → 用池化，绑定失败还以为是训练不够
- 不懂“取用路径” → 换模型时不知道该检查哪一层
- 不懂“两者缺一不可” → 看到准确率 0.602 就以为“部分成功”

所以这一章的每一个实验，本质上都在测这两件事：顺序在不在，路径通不通。

8.52 几个术语的由来

术语不是随便起的。知道它从哪来，能少记一半。

“条件”（conditioning）来自概率论。 $p(x \mid c)$ 读作“给定 c 时 x 的分布”。**它原本没有“控制”的意思**——只是一个已知量。**工程上把它变成“控制”，是因为我们开始优化这个已知量。**

“交叉注意力”（cross-attention）原本叫“编码器-解码器注意力”。2017 年的 Transformer 用它让解码器看编码器的输出。**后来大家发现“两个不同序列之间的注意力”到处都是，**就改叫 cross-attention。**本章的图像-文本只是其中一种。**

“提示词”（prompt）来自自然语言处理。最早是“给模型一个开头，让它续写”。**扩散模型借用了这个词，但含义变了：它不再是“续写的起点”，是“生成的条件”。同一个词，两种机制——这是很多混淆的来源。**

“引导”（guidance）来自分类器引导。2015 年的工作用一个单独训练的分类器给扩散过程指方向。**后来发现不需要分类器，条件模型自己就能提供方向**（CFG）。**“无分类器引导”这个名字里还留着历史的痕迹。**

“潜空间”（latent space）来自表示学习。它指“压缩后的、没被解码的表示”。**注意它不是一个特定的空间**：VAE 有 VAE 的潜空间，CLIP 有 CLIP 的嵌入空间。**本章说的“潜空间”特指 VAE 的那个。**

“绑定”（binding）来自认知科学。它研究的是“大脑怎么把颜色和形状组合成一个物体”。**视觉语言模型借用了这个词，因为失败的样子很像：颜色和形状都在，但配错了对象。**

“token”没有公认的中文翻译。“词元”“标记”“令牌”都有人用，**本书用“token”**——因为它不完全等于“词”，也不完全等于“字”。**一个 token 可能是一个字、半个词、或者三个字节。**

> ℹ 为什么值得知道这些
>
> **因为术语会误导设计。**
>
> **如果你以为“条件”就是“控制”，就会奇怪为什么加大 w 不听话；如果你以为“prompt”就是“开头”，就会奇怪为什么它不能续写；如果你以为“注意力”就是“关注”，就会拿热图当解释。**
>
> **每一个误解，都对应一个术语的历史包袱。**

8.53 写在这一章的最后

这一章讲了四万字，如果只留下一件事，我希望是这个：

接口的形状，比模型的大小更重要。

本章的两个模型，参数只差 1.5 倍。一个在 RB/BR 上是 0.550，一个是 1.000。差别不在容量，在“条件怎么进来”。

这件事在真实模型上同样成立。同一个 UNet，换一种条件注入方式，效果可以差一个档次；同一个编码器，换一种提示词写法，效果也可以差很多。而我们花在“换更大的模型”上的时间，往往多于花在“改接口”上的时间。

第二个想留下的是方法论。

这一章有两次“差点被骗”：

第一次，均值池化的损失更高（0.0476 vs 0.0295），看起来像“模型太小”。**容量对照把它排除了。**

第二次，无位置嵌入的准确率是 0.602 / 0.397，看起来像“部分绑定”。**对调实验把它排除了。**

两次都是“一个看起来很合理的解释”，两次都被一个更简单的实验推翻。

而这两个实验都不复杂：一个是“把参数加到一样多”，一个是“把两个词对调”。**难的不是做实验，是想出“该做哪个实验”。**

第三个想留下的是一个边界。

本章的所有结论，都建立在一个 10 万参数的玩具模型上。它能告诉你“必要条件”，不能告诉你“数值”。

真实模型的绑定失败，可能来自完全不同的原因：训练数据的 caption 不够细、UNet 的容量不够、采样步数太少。**本章的机制只是其中一条。**

所以读完之后，不要急着把“均值池化 0.550”套到 Stable Diffusion 上。要问的是：这个模型的“池化”在哪里？它的“位置嵌入”是什么？

找到了，本章的方法就能用；找不到，说明你还没看懂它的条件路径。

最后一句，给要做研究的人：

这一章留下了六个问题，每一个都可以做成一篇论文。它们都不需要更大的算力，只需要更好的实验设计。而“更好的实验设计”，正是这本书想教的东西。

8.54 本章回顾

把这一章压缩成六句话。

一、文字进入图像要过三步：分词、查表、编码。前两步只是“换一种表示”，**第三步（编码器）才让词与词互相影响。**

二、交叉注意力是“图像问、文本答”。Q 来自图像位置，K, V 来自文本 token。**每个位置得到自己的条件向量，而不是全图共用一个。**

三、绑定的必要条件是“条件里带着顺序”，不是“每个位置各一份”。均值池化把顺序平均掉了，RB/BR 是 0.550/0.449——**加起来 0.999，即零信息。**交叉注意力是 1.000/1.000——**加起来 2.000，才是真的绑上了。**

四、注意力权重不能解释绑定，而且“越尖锐越有用”也是错的。完整模型的权重几乎均匀（熵 0.984）；去掉坐标后完全均匀（熵 0.998），**绑定依然完美**；去掉位置嵌入后权重很尖锐（熵 0.685），**却完全不绑定。**

五、空字符串不是零向量。它是 `[BOS][EOS]` 之类特殊 token 的编码结果，是一个被训练过的表示。**自己拼零向量当空条件，第 7 章的实测是越界 100%。**

六、文本条件下的 CFG 公式不变，但收益可能为零。本章的 w 扫描从 1 到 10，准确率一直是 1.000（$w = 10$ 时 0.999）—— **因为任务在 $w = 1$ 时已经解决了。引导没有可提升的空间，只有成本。**

如果把这一章压成一张表：

问题	答案	本章的数
条件怎么进模型	每个位置一个向量（或一份全局顺序表示）	—
为什么不能用均值	顺序信息被平均掉	RB/BR 0.550/0.449，和 0.999
交叉注意力强在哪	条件带顺序 + 每位置可取	四类全 1.000
去掉位置嵌入会怎样	输出逐位不变	对调差 0.000
注意力图能信吗	不能	熵 0.998 仍绑定；熵 0.685 不绑定
空条件是什么	编码器的非零输出	零向量 → 越界 100%
CFG 还有用吗	取决于 $w = 1$ 是否已饱和	$w = 1..10$ 均 $\geqslant 0.999$

最后留一个问题

本章把 query 里的坐标抹掉，两个点问出**完全一样**的问题，注意力权重**完全均匀**（0.525/0.525）。

结果：绑定依然 1.000。

那“每个位置各一份条件”这件事，到底是不是必需的？

本章的答案：不是必需的。**必需的是“条件里带着顺序”。**只要 K 里的位置嵌入让 RB 和 BR 产生不同的全局向量，下游 MLP 就能借助输入 x 里的坐标把颜色分配对。

那交叉注意力买到了什么？

买到的是“组合性”：全局条件要背下 $k!$ 种排列，逐位置条件只要学 k 组对应。**对象一多，前者需要指数级的数据。**

这个问题还可以再推一步：如果把对象数加到 4 个、6 个，“全局条件 + 下游路由”会在哪里崩？**本章只做到 2 个。这是一个明确留下的边界。**

下一章是附录：数学补充、术语表、实验索引、notebook 环境与作业、延伸阅读。**但如果只带走一件事：先看镜像类的准确率之和——等于 1，说明顺序根本没进去。**

附录

A. 数学补充

这份附录是正文的“推导仓库”。

正文里每一处“完整展开见附录 A.2”的承诺，都在这里兑现。读法有两种：遇到卡住的地方来查一节；或者从头读一遍，当作全书的数学复习。

每一条都标了“用在哪”，你可以只读自己需要的那几条。

A.1 符号与约定

先把记号钉死。全书只用这一套。

记号	含义	例子
x	向量（默认列向量）	$x_t \in \mathbb{R}^d$
x_i	向量的第 i 个分量	x_1 是第一个数
$A^\top$	矩阵转置	$(QK^\top)_{ij} = q_i \cdot k_j$
$\|x\|$	欧氏范数 $\sqrt{x^\top x}$	$\|x\|^2 = \sum_i x_i^2$
$\nabla_x f$	对 x 的梯度（和 x 同形状）	$\nabla_x \|x\|^2 = 2x$
$\mathbb{E}[X]$	期望	$\mathbb{E}[\varepsilon] = 0$
$\mathrm{Var}(X)$	方差	$\mathrm{Var}(\varepsilon) = I$
$\mathcal{N}(\mu, \Sigma)$	高斯分布	$\mathcal{N}(0, I)$ 是标准正态
I	单位矩阵	$Ix = x$
$\odot$	逐元素乘	$a \odot b$
$\propto$	正比于（差一个常数因子）	$p(x) \propto e^{-x^2/2}$
$\log$	自然对数	$\log e = 1$

三个约定值得单独说。

① **下标 t 永远指扩散步，不指时间连续变量。** 连续版本用 τ，只在 A.2 和 A.4 出现。

② **噪声用 ε，不用 ξ 或 z。** z 留给潜变量，ξ 留给“另一个噪声”（比如重参数化里的辅助变量）。

③ **所有向量默认是列向量，但代码里是行向量。** 数学上 $QK^\top$ 的 Q 是 (N, d)、K 是 (M, d)，**写代码时不用转置**——这是 numpy 传统，不是数学错误。

> **i** 为什么记号值得花一节
>
> **因为这本书里同一个字母在不同章节有不同含义。**
>
> - w：第 7 章是引导强度，第 8 章是 token 的权重（`word:1.3`）
> - c：第 7 章是类别下标，第 8 章是一整句话
> - t：扩散步；第 8 章的 t 也指扩散步，但第 1 章的 t 是序列位置
>
> **每次换章，先看一眼这一节的表。**

A.2 四个完整推导

这一节是全书推导密度最高的地方。 四个推导分别对应正文里四个“只给结论”的地方。

A.2.1 配方法：后验 $q(x_{t-1} \mid x_t, x_0)$ 是怎么来的

正文第 5 章直接给了结果：

$$q(x_{t-1} \mid x_t, x_0) = \mathcal{N}\big(\tilde{\mu}_t(x_t, x_0), \tilde{\beta}_t I\big) \tag{1.1}$$

$$\tilde{\mu}_t = \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t\, x_0 + \sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})\, x_t}{1-\bar{\alpha}_t}, \qquad \tilde{\beta}_t = \frac{\beta_t(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t} \tag{1.2}$$

这里把每一步都写出来。
起点是两个高斯相乘：

$$q(x_{t-1} \mid x_t, x_0) \propto \underbrace{\exp\left[-\frac{\|x_{t-1} - \sqrt{\bar{\alpha}_{t-1}}x_0\|^2}{2(1-\bar{\alpha}_{t-1})}\right]}_{q(x_{t-1}|x_0)} \cdot \underbrace{\exp\left[-\frac{\|x_t - \sqrt{\alpha_t}x_{t-1}\|^2}{2\beta_t}\right]}_{q(x_t|x_{t-1})} \tag{1.3}$$

第一步：把指数里的平方展开，只保留和 x_{t-1} 有关的项。
第一项的指数：

$$-\frac{\|x_{t-1}\|^2 - 2\sqrt{\bar{\alpha}_{t-1}}\, x_0^\top x_{t-1} + \bar{\alpha}_{t-1}\|x_0\|^2}{2(1-\bar{\alpha}_{t-1})} \tag{1.4}$$

第二项的指数：

$$-\frac{\|x_t\|^2 - 2\sqrt{\alpha_t}\, x_t^\top x_{t-1} + \alpha_t\|x_{t-1}\|^2}{2\beta_t} \tag{1.5}$$

和 x_{t-1} 无关的项（$\|x_t\|^2$、$\bar{\alpha}_{t-1}\|x_0\|^2$）在归一化时会被消掉，先扔掉。
第二步：收集 $\|x_{t-1}\|^2$ 的系数。

$$\frac{1}{2(1-\bar{\alpha}_{t-1})} + \frac{\alpha_t}{2\beta_t} = \frac{\beta_t + \alpha_t(1-\bar{\alpha}_{t-1})}{2\beta_t(1-\bar{\alpha}_{t-1})} \tag{1.6}$$

用 $\alpha_t = 1 - \beta_t$ 化简分子：

$$\beta_t + \alpha_t - \alpha_t\bar{\alpha}_{t-1} = 1 - \alpha_t\bar{\alpha}_{t-1} = 1 - \bar{\alpha}_t \tag{1.7}$$

所以二次项系数是

$$\frac{1-\bar{\alpha}_t}{2\beta_t(1-\bar{\alpha}_{t-1})} \tag{1.8}$$

第三步：认方差。
高斯密度里二次项的系数是 $1/(2\sigma^2)$，所以

$$\tilde{\beta}_t = \frac{\beta_t(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t} \tag{1.9}$$

第四步：收集 x_{t-1} 的一次项。

$$\frac{\sqrt{\bar{\alpha}_{t-1}}\, x_0^\top x_{t-1}}{1-\bar{\alpha}_{t-1}} + \frac{\sqrt{\alpha_t}\, x_t^\top x_{t-1}}{\beta_t} \tag{1.10}$$

第五步：从一次项读出均值。
对 $\mathcal{N}(\mu, \sigma^2)$，一次项是 $\mu x/\sigma^2$。所以 $\mu = \sigma^2 \times$ (一次项系数)：

$$\tilde{\mu}_t = \frac{\beta_t(1-\bar{\alpha}_{t-1})}{1-\bar{\alpha}_t}\left[\frac{\sqrt{\bar{\alpha}_{t-1}}x_0}{1-\bar{\alpha}_{t-1}} + \frac{\sqrt{\alpha_t}x_t}{\beta_t}\right] \tag{1.11}$$

约掉两个因子：

$$\tilde{\mu}_t = \frac{\sqrt{\bar{\alpha}_{t-1}}\beta_t\, x_0 + \sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})\, x_t}{1-\bar{\alpha}_t} \tag{1.12}$$

这就是正文那个式子。

> ! 这个推导里唯一“技巧”的地方
>
> **只有一步：认出二次项系数等于 $1/(2\sigma^2)$。**
> **其余全是展开、合并、约分。**
> **所以如果你在别处看到“配方法很神奇”，那是在骗你—— 它只是“把高斯乘起来，按二次型重新读参数”。**
> **同样的套路可以用在任何“两个高斯相乘”的场合**，包括第 8 章那个“先验乘似然得到后验”的贝叶斯视角。

A.2.2 ELBO 逐项展开

正文第 5 章给了 ELBO 的分解。这里补齐每一步。
起点是负对数似然：

$$-\log p_\theta(x_0) = -\log \int p_\theta(x_{0:T})\, dx_{1:T} \tag{1.13}$$

引入一个任意分布 $q(x_{1:T} \mid x_0)$，乘一除一：

$$-\log p_\theta(x_0) = -\log \int q(x_{1:T} \mid x_0) \frac{p_\theta(x_{0:T})}{q(x_{1:T} \mid x_0)}\, dx_{1:T} \tag{1.14}$$

用 Jensen 不等式（$-\log$ 是凸函数，所以 $\mathbb{E}[-\log X] \geq -\log \mathbb{E}[X]$）：

$$-\log p_\theta(x_0) \leq -\mathbb{E}_q\left[\log \frac{p_\theta(x_{0:T})}{q(x_{1:T} \mid x_0)}\right] \tag{1.15}$$

右边就是 ELBO 的相反数。把它写成 L：

$$L = \mathbb{E}_q\left[\log \frac{q(x_{1:T} \mid x_0)}{p_\theta(x_{0:T})}\right] \tag{1.16}$$

现在拆分子分母。
分母（用马尔可夫性）：

$$p_\theta(x_{0:T}) = p_\theta(x_T) \prod_{t=1}^{T} p_\theta(x_{t-1} \mid x_t) \tag{1.17}$$

分子：

$$q(x_{1:T} \mid x_0) = \prod_{t=1}^{T} q(x_t \mid x_{t-1}) \tag{1.18}$$

把连乘代进去，并把 $t=1$ 那一项单独拿出来：

$$L = \mathbb{E}_q\left[\log \frac{q(x_1 \mid x_0)}{p_\theta(x_0 \mid x_1)} + \sum_{t=2}^{T} \log \frac{q(x_t \mid x_{t-1})}{p_\theta(x_{t-1} \mid x_t)} - \log p_\theta(x_T)\right] \tag{1.19}$$

关键一步：把 $q(x_t \mid x_{t-1})$ 换成“以 x_0 为条件的后验”。
用贝叶斯公式：

$$q(x_t \mid x_{t-1}) = q(x_t \mid x_{t-1}, x_0) = \frac{q(x_{t-1} \mid x_t, x_0)\, q(x_t \mid x_0)}{q(x_{t-1} \mid x_0)} \tag{1.20}$$

这一步之所以成立，是因为给定 x_0 之后，x_t 只依赖 x_{t-1} —— 反向也成立（扩散过程是马尔可夫的）。

代进求和项：

$$\log \frac{q(x_t \mid x_{t-1})}{p_\theta(x_{t-1} \mid x_t)} = \log \frac{q(x_{t-1} \mid x_t, x_0)}{p_\theta(x_{t-1} \mid x_t)} + \log \frac{q(x_t \mid x_0)}{q(x_{t-1} \mid x_0)} \tag{1.21}$$

第二项是个望远镜求和：

$$\sum_{t=2}^{T} \log \frac{q(x_t \mid x_0)}{q(x_{t-1} \mid x_0)} = \log \frac{q(x_T \mid x_0)}{q(x_1 \mid x_0)} \tag{1.22}$$

把它和第一项、最后一项合起来：

$$L = \underbrace{\mathbb{E}_q\left[\log \frac{q(x_T \mid x_0)}{p_\theta(x_T)}\right]}_{L_T} + \underbrace{\mathbb{E}_q\left[\log \frac{q(x_{t-1} \mid x_t, x_0)}{p_\theta(x_{t-1} \mid x_t)}\right]}_{L_{t-1}\ (t=2\ldots T)} + \underbrace{-\mathbb{E}_q[\log p_\theta(x_0 \mid x_1)]}_{L_0} \tag{1.23}$$

这就是正文那张三项分解。

> **i** 三项各是什么
>
> - $\boldsymbol{L_T}$：先验和噪声分布差多远。**训练中不优化**——q 没有参数，$p_\theta(x_T) = \mathcal{N}(0, I)$ 固定。
> - $\boldsymbol{L_{t-1}}$：每一步的去噪误差。**这是训练的主项。**
> - $\boldsymbol{L_0}$：最后一步还原 x_0 的负对数似然（越小越好）。
>
> **L_{t-1} 里的两个分布都是高斯**（q 是 A.2.1 推的后验，p_θ 是模型），**所以它可以写成闭式**——下一小节。

A.2.3 高斯 KL 与“为什么最终变成 MSE”

两个同方差高斯的 KL：

$$\mathrm{KL}\left(\mathcal{N}(\mu_1, \sigma^2) \,\|\, \mathcal{N}(\mu_2, \sigma^2)\right) = \frac{\|\mu_1 - \mu_2\|^2}{2\sigma^2} \tag{1.24}$$

推导：两个密度相除，指数部分合并，

$$\log \frac{p_1}{p_2} = -\frac{\|x - \mu_1\|^2}{2\sigma^2} + \frac{\|x - \mu_2\|^2}{2\sigma^2} \tag{1.25}$$

对 $x \sim p_1$ 取期望。 用 $\mathbb{E}\|x - \mu_1\|^2 = d\sigma^2$：

$$\mathbb{E}[\log p_1 - \log p_2] = -\frac{d}{2} + \frac{\mathbb{E}\|x - \mu_2\|^2}{2\sigma^2} = -\frac{d}{2} + \frac{d\sigma^2 + \|\mu_1 - \mu_2\|^2}{2\sigma^2} = \frac{\|\mu_1 - \mu_2\|^2}{2\sigma^2} \tag{1.26}$$

方差相同，归一化项抵消，只剩均值差。

代进 L_{t-1}：

$$L_{t-1} = \mathbb{E}_q\left[\frac{\|\tilde{\mu}_t(x_t, x_0) - \mu_\theta(x_t, t)\|^2}{2\tilde{\beta}_t}\right] + \text{常数} \tag{1.27}$$

所以“最小化 KL”就是“最小化均值的平方误差”。

再把两个均值都写成 ε 的形式（正文式 5-x）：

$$\tilde{\mu}_t = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\tilde{\varepsilon}_t\right), \qquad \mu_\theta = \frac{1}{\sqrt{\alpha_t}}\left(x_t - \frac{\beta_t}{\sqrt{1-\bar{\alpha}_t}}\varepsilon_\theta\right) \tag{1.28}$$

相减，x_t 和 $1/\sqrt{\alpha_t}$ 都消掉：

$$\tilde{\mu}_t - \mu_\theta = \frac{\beta_t}{\sqrt{\alpha_t}\sqrt{1-\bar{\alpha}_t}}(\varepsilon_\theta - \tilde{\varepsilon}_t) \tag{1.29}$$

代回 L_{t-1}，前面的系数是

$$\frac{1}{2\tilde{\beta}_t} \cdot \frac{\beta_t^2}{\alpha_t(1-\bar{\alpha}_t)} \tag{1.30}$$

用 $\tilde{\beta}_t = \beta_t(1-\bar{\alpha}_{t-1})/(1-\bar{\alpha}_t)$ 化简。 先把 $1/(2\tilde{\beta}_t)$ 展开：

$$\frac{1}{2\tilde{\beta}_t} = \frac{1-\bar{\alpha}_t}{2\beta_t(1-\bar{\alpha}_{t-1})} \tag{1.31}$$

代回去，$(1-\bar{\alpha}_t)$ 上下约掉：

$$\frac{1-\bar{\alpha}_t}{2\beta_t(1-\bar{\alpha}_{t-1})} \cdot \frac{\beta_t^2}{\alpha_t(1-\bar{\alpha}_t)} = \frac{\beta_t}{2\alpha_t(1-\bar{\alpha}_{t-1})} \tag{1.32}$$

这是一个只和 t 有关的正系数。

!到这一步，DDPM 的训练目标就“推”出来了

完整的 L_{t-1} 是：

$$L_{t-1} = \frac{\beta_t}{2\alpha_t(1-\bar{\alpha}_{t-1})}\mathbb{E}\left[\|\varepsilon - \varepsilon_\theta(x_t,t)\|^2\right] \tag{1.33}$$

DDPM 的做法是：把这个系数扔掉，直接用 $\|\varepsilon - \varepsilon_\theta\|^2$。
这不是“近似”，是“换了一个目标”—— 新目标仍然是 ELBO 的一个加权版本（权重是上面那个系数），**只是权重不再等于理论值。**
为什么可以这么换？ 看那个系数的形状：

$$c_t = \frac{\beta_t}{2\alpha_t(1-\bar{\alpha}_{t-1})} \tag{1.34}$$

t 小（低噪声）时 $1-\bar{\alpha}_{t-1} \approx \beta_1$，系数很大；$t$ 大（高噪声）时 $1-\bar{\alpha}_{t-1} \approx 1$，系数小一个多数量级。
所以理论目标把注意力集中在低噪声步。
扔掉权重 = 把各步拉平 = 相对提高高噪声步的重要性。
DDPM 的实验发现这样训练反而更稳、生成更好。 一个常见的解释是：低噪声步的像素级误差和感知质量关系不大， 而高噪声步决定了整体结构。
这是本书反复出现的主题：理论给你一个目标，工程给你一个更好的目标。

A.2.4 score matching 与 ε-预测的等价性

这是正文承诺过的“完整证明”。
先定义两个东西。
score：对数密度的梯度

$$s(x,t) = \nabla_x \log p_t(x) \tag{1.35}$$

去噪分数匹配（denoising score matching）的目标：

$$J_{\mathrm{DSM}}(\theta) = \mathbb{E}_{t,x_0,x_t}\left[\left\| s_\theta(x_t,t) - \nabla_{x_t}\log q(x_t \mid x_0)\right\|^2\right] \tag{1.36}$$

注意它不需要知道 $p_t(x)$ 本身——只需要知道“给定 x_0 时 x_t 的分布”， 而那个分布是我们自己定义的前向过程，有闭式。
第一步：算 $\nabla_{x_t}\log q(x_t \mid x_0)$。

$$q(x_t \mid x_0) = \mathcal{N}\left(\sqrt{\bar{\alpha}_t}x_0, (1-\bar{\alpha}_t)I\right) \tag{1.37}$$

高斯密度的对数:

$$\log q = -\frac{\|x_t - \sqrt{\bar{\alpha}_t}x_0\|^2}{2(1-\bar{\alpha}_t)} + \text{常数} \tag{1.38}$$

求梯度:

$$\nabla_{x_t} \log q = -\frac{x_t - \sqrt{\bar{\alpha}_t}x_0}{1-\bar{\alpha}_t} \tag{1.39}$$

第二步：用前向过程的定义替换分子。

$$x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\,\varepsilon \quad \Rightarrow \quad x_t - \sqrt{\bar{\alpha}_t}x_0 = \sqrt{1-\bar{\alpha}_t}\,\varepsilon \tag{1.40}$$

代进去:

$$\nabla_{x_t} \log q = -\frac{\sqrt{1-\bar{\alpha}_t}\,\varepsilon}{1-\bar{\alpha}_t} = -\frac{\varepsilon}{\sqrt{1-\bar{\alpha}_t}} \tag{1.41}$$

第三步：假设模型用 ε-参数化。

$$s_\theta(x_t, t) = -\frac{\varepsilon_\theta(x_t, t)}{\sqrt{1-\bar{\alpha}_t}} \tag{1.42}$$

代进 J_{DSM}:

$$J_{\text{DSM}} = \mathbb{E}\left[\left\| -\frac{\varepsilon_\theta}{\sqrt{1-\bar{\alpha}_t}} + \frac{\varepsilon}{\sqrt{1-\bar{\alpha}_t}} \right\|^2\right] = \mathbb{E}\left[\frac{\|\varepsilon - \varepsilon_\theta\|^2}{1-\bar{\alpha}_t}\right] \tag{1.43}$$

这就是结论:

$$J_{\text{DSM}}(\theta) = \mathbb{E}_{t, x_0, \varepsilon}\left[\frac{1}{1-\bar{\alpha}_t}\|\varepsilon - \varepsilon_\theta(x_t, t)\|^2\right] \tag{1.44}$$

! “等价”到底等价在哪

它和 DDPM 的简单损失差一个权重 $1/(1-\bar{\alpha}_t)$。
这个权重在 t 大时很大（$\bar{\alpha}_t \to 0$），在 t 小时接近 1。
所以：

- **如果你相信“模型足够表达”，两个目标的全局最优是同一个函数**—— 都是 $s^*(x,t) = \nabla \log p_t(x)$，只是**在不同 t 上强调的程度不同。**
- **如果模型容量有限，权重会改变它优先拟合哪些噪声水平。**

DDPM 扔掉这个权重，等于“主动降低高噪声步的重要性”。这个选择在实践上更好，但没有理论上的“必须”。
这就是“等价”的准确含义：不是“完全一样”，是“同一个最优解，不同的加权路径”。

还有一件事值得说清楚：为什么 score matching 有用?
因为 $s(x,t) = \nabla \log p_t(x)$ 不需要归一化常数。
如果 $p_t(x) = \frac{1}{Z}\tilde{p}_t(x)$，那么

$$\nabla \log p_t = \nabla \log \tilde{p}_t - \underbrace{\nabla \log Z}_{=0} \tag{1.45}$$

Z 是常数，梯度为零。
所以模型可以直接学“未归一化的密度”的梯度，而不必算那个积不出来的 Z。

这是扩散模型能绕开“配分函数”的根本原因。

A.3 高斯分布的五条性质

这一节是工具，不是推导。每条都标了用在哪。

A.3.1 性质一：线性变换还是高斯

若 $x \sim \mathcal{N}(\mu, \Sigma)$，则

$$Ax + b \sim \mathcal{N}(A\mu + b,\, A\Sigma A^{\top}) \tag{1.46}$$

用在哪：第 5 章的“加噪”就是 $x_t = \sqrt{\bar{\alpha}_t}x_0 + \sqrt{1-\bar{\alpha}_t}\varepsilon$，**一个线性变换**。

A.3.2 性质二：独立高斯的和

若 $x \sim \mathcal{N}(\mu_1, \Sigma_1)$、$y \sim \mathcal{N}(\mu_2, \Sigma_2)$ 独立，则

$$x + y \sim \mathcal{N}(\mu_1 + \mu_2,\, \Sigma_1 + \Sigma_2) \tag{1.47}$$

注意方差是相加，不是相减。
用在哪：$q(x_t \mid x_0)$ 的闭式，就是把 t 个独立噪声加起来。

A.3.3 性质三：重参数化

若 $\varepsilon \sim \mathcal{N}(0, I)$，则

$$x = \mu + \sigma\varepsilon \sim \mathcal{N}(\mu, \sigma^2 I) \tag{1.48}$$

反过来，任何高斯样本都可以写成“确定性变换 + 标准噪声”。
用在哪：整个扩散模型的采样。**也是“为什么可以对随机过程求梯度”的答案—— 随机性被隔离在 ε 里，μ, σ 是可导的。**

A.3.4 性质四：条件高斯

若

$$\begin{pmatrix} x \\ y \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} \mu_x \\ \mu_y \end{pmatrix}, \begin{pmatrix} \Sigma_{xx} & \Sigma_{xy} \\ \Sigma_{yx} & \Sigma_{yy} \end{pmatrix} \right) \tag{1.49}$$

则

$$x \mid y \sim \mathcal{N}\big(\mu_x + \Sigma_{xy}\Sigma_{yy}^{-1}(y - \mu_y),\, \Sigma_{xx} - \Sigma_{xy}\Sigma_{yy}^{-1}\Sigma_{yx}\big) \tag{1.50}$$

用在哪：A.2.1 的配方法就是它的“标量版”。**第 8 章的贝叶斯视角**（条件分数 = 无条件分数 + 匹配项）也是它的梯度版。

A.3.5 性质五：高斯熵

$$H(\mathcal{N}(\mu, \Sigma)) = \frac{1}{2}\log\big((2\pi e)^d \det \Sigma\big) \tag{1.51}$$

用在哪：第 8 章把“损失差”换算成“信息差”那一步：

$$\Delta H = \frac{1}{2}\log\frac{\mathrm{MSE}_1}{\mathrm{MSE}_2} \tag{1.52}$$

注意这个换算只在“条件高斯”的假设下成立，正文已经标注了这个前提。

A.4 分数、Tweedie 与 x_0 估计

第 5 章给过 x_0 的估计式。这里说明它和 score 的关系。

Tweedie 公式（高斯情形）：

$$\mathbb{E}[x_0 \mid x_t] = \frac{1}{\sqrt{\bar{\alpha}_t}}\big(x_t + (1-\bar{\alpha}_t)\nabla_{x_t}\log p_t(x_t)\big) \tag{1.53}$$

把它和 $\varepsilon = -\sqrt{1-\bar{\alpha}_t}\,\nabla \log p_t$ 代进去：

$$\mathbb{E}[x_0 \mid x_t] = \frac{x_t - \sqrt{1-\bar{\alpha}_t}\,\varepsilon}{\sqrt{\bar{\alpha}_t}} \tag{1.54}$$

这正是正文那个 $\hat{x}_0$。

所以三个东西是同一件事的三种写法：

写法	形式	优点
x_0 预测	$\hat{x}_0$	直观，能直接 clamp
ε 预测	$\hat{\varepsilon}$	数值范围稳定（第 5 章实测差 8500 倍）
score 预测	$\nabla \log p_t$	理论工具多（SDE、ODE）

转换关系：

$$\hat{x}_0 = \frac{x_t - \sqrt{1-\bar{\alpha}_t}\hat{\varepsilon}}{\sqrt{\bar{\alpha}_t}}, \qquad \hat{\varepsilon} = \frac{x_t - \sqrt{\bar{\alpha}_t}\hat{x}_0}{\sqrt{1-\bar{\alpha}_t}}, \qquad s_\theta = -\frac{\hat{\varepsilon}}{\sqrt{1-\bar{\alpha}_t}} \tag{1.55}$$

记住任意一行，另外两行都能推出来。

A.5 softmax 与交叉熵的梯度

第 1 章说“softmax 把 logits 变成概率”，第 8 章说“梯度会流回文本”。这里把两个都用到的梯度补全。

A.5.1 softmax 的雅可比

设 $p = \mathrm{softmax}(z)$，即

$$p_i = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{1.56}$$

求导：

$$\frac{\partial p_i}{\partial z_j} = p_i(\delta_{ij} - p_j) \tag{1.57}$$

写成矩阵：

$$\frac{\partial p}{\partial z} = \mathrm{diag}(p) - pp^\top \tag{1.58}$$

用在哪：第 8 章“梯度怎么流回文本”那一节，**把 $\partial L/\partial s_j$ 写成 $a_j(g_j - \sum_k a_k g_k)$，用的就是这一条。**

A.5.2 交叉熵的梯度

设损失 $L = -\sum_i y_i \log p_i$，y 是 one-hot。

$$\frac{\partial L}{\partial z_j} = p_j - y_j \tag{1.59}$$

推导：链式法则加上面那条雅可比，

$$\frac{\partial L}{\partial z_j} = -\sum_i \frac{y_i}{p_i} \cdot p_i(\delta_{ij} - p_j) = -\sum_i y_i(\delta_{ij} - p_j) = -y_j + p_j \sum_i y_i = p_j - y_j \tag{1.60}$$

这个式子漂亮得不像话：梯度就是“预测减真值”。
用在哪：第 3 章的训练。**它也是“为什么 logits 平移不变”的答案——给所有 z_i 加同一个常数，p 不变，梯度也不变。**

A.5.3 数值稳定

实现里必须先减最大值：

$$p_i = \frac{e^{z_i - m}}{\sum_j e^{z_j - m}}, \qquad m = \max_j z_j \tag{1.61}$$

数学上完全等价，数值上避免 e^{1000} 溢出。
用在哪：第 1 章失败三。**在那个规模下看不出差别，但真实模型上会 NaN。**

A.6 方差传播与 $\sqrt{d_k}$

第 1 章说“点积的方差随 d 增长，所以要除以 $\sqrt{d_k}$”。这里把方差算出来。
设 q_i, k_i 独立、均值 0、方差 σ^2。则

$$\mathrm{Var}(q \cdot k) = \mathrm{Var}\left(\sum_{i=1}^{d} q_i k_i\right) = \sum_{i=1}^{d} \mathrm{Var}(q_i k_i) \tag{1.62}$$

独立同分布，所以是 d 倍。而

$$\mathrm{Var}(q_i k_i) = \mathbb{E}[q_i^2]\mathbb{E}[k_i^2] - (\mathbb{E}[q_i]\mathbb{E}[k_i])^2 = \sigma^4 \tag{1.63}$$

所以

$$\mathrm{Var}(q \cdot k) = d\sigma^4, \qquad \text{标准差} = \sigma^2\sqrt{d} \tag{1.64}$$

除以 $\sqrt{d}$ 之后，标准差变成 σ^2——和 d 无关。

> **i** 一个容易混的点
>
> **很多人记成“除以 $\sqrt{d_k}$ 让方差变成 1”。**
> **不对。**除以 $\sqrt{d}$ 让**标准差**从 $\sigma^2\sqrt{d}$ 变成 σ^2。**只有 $\sigma = 1$ 时才是 1。**
> **第 1 章的实测（$\sigma \approx 0.02$）里，缩放后的标准差约 4×10^{-4}——所以那个模型上 softmax 根本不饱和，缩放看起来“没用”。这就是“小模型上看不出差别”的数学原因。**

A.7 信息论三条

A.7.1 熵

离散：

$$H(X) = -\sum_x p(x) \log p(x) \tag{1.65}$$

连续（微分熵）：

$$H(X) = -\int p(x)\log p(x)\,dx \tag{1.66}$$

注意微分熵可以为负（方差小于 $1/(2\pi e)$ 时）。

A.7.2 互信息

$$I(X;Y) = H(X) - H(X \mid Y) = H(Y) - H(Y \mid X) \tag{1.67}$$

它衡量“知道 Y 之后，X 的不确定性减少了多少”。

A.7.3 数据处理不等式

若 $X \to Y \to Z$ 构成马尔可夫链（Z 只通过 Y 依赖 X），则

$$I(X;Z) \le I(X;Y) \tag{1.68}$$

直觉：加工不会增加信息。 你只能丢信息，不能凭空造。
用在哪：第 8 章“均值池化丢的是信息，不是精度”。 **均值池化是 Y 的一个确定性函数，所以 $I(x_0; c_{\text{mean}}) \le I(x_0; c_{\text{token}})$。**
这个不等式是“结构上不可能”，不是“训练不够”。

A.7.4 一个具体的换算

第 8 章用

$$\Delta H = \frac{1}{2}\log\frac{\text{MSE}_1}{\text{MSE}_2} \tag{1.69}$$

把两个模型的损失差换成“比特”。
这个式子来自高斯熵（A.3 性质五）：**同方差假设下，熵只和方差的对数有关。**
代入 0.0480 和 0.0295：

$$\Delta H = \frac{1}{2}\log\frac{0.0480}{0.0295} = 0.243\ \text{nat} = 0.351\ \text{bit} \tag{1.70}$$

它的含义是“条件多提供了 0.35 比特”，不是“模型准确率高了 35%”。

A.8 五个常用不等式

不等式	形式	用在哪
Jensen	$f(\mathbb{E}X) \le \mathbb{E}f(X)$（$f$ 凸）	ELBO（A.2.2）
Cauchy–Schwarz	$\lvert\langle x,y\rangle\rvert \le \lVert x\rVert\lVert y\rVert$	余弦相似度、CLIP
log-sum	$\sum a_i \log\frac{a_i}{b_i} \ge (\sum a_i)\log\frac{\sum a_i}{\sum b_i}$	KL 非负
AM–GM	$\frac{a+b}{2} \ge \sqrt{ab}$	方差界
union bound	$P(\cup_i A_i) \le \sum_i P(A_i)$	多重比较的保守估计

最常用的是 Jensen。它一句话就能证明 KL 非负：

$$\text{KL}(p \,\|\, q) = \mathbb{E}_p\left[-\log\frac{q}{p}\right] \ge -\log\mathbb{E}_p\left[\frac{q}{p}\right] = -\log 1 = 0 \tag{1.71}$$

注意不等号方向：$-\log$ 是凸函数，所以 $\mathbb{E}[-\log X] \ge -\log\mathbb{E}[X]$。

A.9 矩阵求导三条

这三个式子在第 5、7、8 章都会出现。

$$\frac{\partial (a^\top x)}{\partial x} = a \tag{1.72}$$

$$\frac{\partial (x^\top A x)}{\partial x} = (A + A^\top)x \tag{1.73}$$

$$\frac{\partial \log\det A}{\partial A} = A^{-\top} \tag{1.74}$$

第一条：线性函数的梯度就是系数。
第二条：二次型。**若 A 对称，就是 $2Ax$**。
第三条：用在“高斯归一化常数对协方差求导”时。
再加一条链式法则的矩阵版：

$$\frac{\partial L}{\partial x} = \left(\frac{\partial y}{\partial x}\right)^\top \frac{\partial L}{\partial y} \tag{1.75}$$

注意转置。形状不对的时候，先检查这一条。

A.10 DDIM 的代数

第 6 章说“DDIM 是确定性采样”。这里把它的更新式推出来。

A.10.1 从“保持边缘分布”开始

我们想要一个非马尔可夫的前向过程，但要求它的边缘分布 $q(x_t \mid x_0)$ 和 DDPM 完全一样。
DDPM 的边缘是

$$q(x_t \mid x_0) = \mathcal{N}\big(\sqrt{\bar\alpha_t}x_0, (1-\bar\alpha_t)I\big) \tag{1.76}$$

现在定义

$$q_\sigma(x_{t-1} \mid x_t, x_0) = \mathcal{N}\left(\sqrt{\bar\alpha_{t-1}}x_0 + \sqrt{1-\bar\alpha_{t-1}-\sigma_t^2}\cdot\frac{x_t - \sqrt{\bar\alpha_t}x_0}{\sqrt{1-\bar\alpha_t}}, \sigma_t^2 I\right) \tag{1.77}$$

其中 σ_t 是一个可以选的参数。
验证它给出的边缘：
均值：第二项的期望是 0（因为 x_t 的均值是 $\sqrt{\bar\alpha_t}x_0$），**所以均值是 $\sqrt{\bar\alpha_{t-1}}x_0$**。
方差：

$$\frac{1-\bar\alpha_{t-1}-\sigma_t^2}{1-\bar\alpha_t}\,\mathrm{Var}(x_t \mid x_0) + \sigma_t^2 = (1-\bar\alpha_{t-1}-\sigma_t^2) + \sigma_t^2 = 1-\bar\alpha_{t-1} \tag{1.78}$$

正好是 DDPM 的边缘。

! σ_t 是一个“自由度”

它控制“每一步注入多少新噪声”。

- $\sigma_t^2 = \tilde{\beta}_t \to$ 回到 DDPM（马尔可夫、随机）
- $\sigma_t = 0 \to$ 完全确定性
- 中间值 $\to$ 一族插值

DDIM 就是“选 $\sigma_t = 0$”的那一个特例。
注意：边缘分布没变，变的只是“路径”。所以 DDIM 和 DDPM 在 $T \to \infty$ 时给出同一个分布，只是走法不同。

A.10.2 采样更新式

从 $q_\sigma(x_{t-1} \mid x_t, x_0)$ 采一个样本：

$$x_{t-1} = \sqrt{\bar{\alpha}_{t-1}}\hat{x}_0 + \sqrt{1-\bar{\alpha}_{t-1}-\sigma_t^2}\cdot\frac{x_t - \sqrt{\bar{\alpha}_t}\hat{x}_0}{\sqrt{1-\bar{\alpha}_t}} + \sigma_t\varepsilon \tag{1.79}$$

其中 $\hat{x}_0$ 是模型给出的估计：

$$\hat{x}_0 = \frac{x_t - \sqrt{1-\bar{\alpha}_t}\,\varepsilon_\theta(x_t, t)}{\sqrt{\bar{\alpha}_t}} \tag{1.80}$$

把 $\hat{x}_0$ 代进去，整理成“x_t 的线性组合 + 噪声”：

$$x_{t-1} = \sqrt{\bar{\alpha}_{t-1}}\,\hat{x}_0 + \sqrt{1-\bar{\alpha}_{t-1}-\sigma_t^2}\,\varepsilon_\theta + \sigma_t\varepsilon \tag{1.81}$$

这一步用了

$$\frac{x_t - \sqrt{\bar{\alpha}_t}\hat{x}_0}{\sqrt{1-\bar{\alpha}_t}} = \varepsilon_\theta \tag{1.82}$$

这是 $\hat{x}_0$ 定义的直接变形。

A.10.3 两个端点

$\sigma_t = 0$：

$$x_{t-1} = \sqrt{\bar{\alpha}_{t-1}}\,\hat{x}_0 + \sqrt{1-\bar{\alpha}_{t-1}}\,\varepsilon_\theta \tag{1.83}$$

没有随机项，给定 x_t 就唯一确定 x_{t-1}。这是 DDIM。
$\sigma_t^2 = \tilde{\beta}_t$：
代回去，可以验证它等于 DDPM 的后验均值加 $\sqrt{\tilde{\beta}_t}\varepsilon$。这一步的代数有点长，但方向是清楚的：“选多大的 σ” = “在确定性和随机性之间插值”。

i 为什么“确定性”能跳步

因为确定性采样对应一个 ODE。
ODE 的数值解可以用高阶方法，也可以跨大步。随机采样对应 SDE，SDE 的离散化误差更难控制。
所以 DDIM 能 20 步走完，DDPM 不行—— 不是 DDIM “更聪明”，是它解的问题更容易。
第 6 章的那些采样器（DPM-Solver、Heun），都是在这个 ODE 上做文章。

A.11 PCA 与有效维度

第 7 章用“PC1 占比”和“有效维度”来说明分布塌缩。这里把定义和算法写清楚。

A.11.1 协方差与特征值

给定 n 个样本 $x_i \in \mathbb{R}^d$，先中心化：

$$\tilde{x}_i = x_i - \bar{x}, \qquad \bar{x} = \frac{1}{n}\sum_i x_i \tag{1.84}$$

协方差矩阵：

$$\Sigma = \frac{1}{n}\sum_i \tilde{x}_i \tilde{x}_i^\top \tag{1.85}$$

特征分解：

$$\Sigma = U\Lambda U^\top, \qquad \Lambda = \mathrm{diag}(\lambda_1, ..., \lambda_d), \lambda_1 \geq \cdots \geq \lambda_d \geq 0 \tag{1.86}$$

第 k 个主成分方向是 U 的第 k 列，方差是 λ_k。

A.11.2 两个指标

PC1 占比：

$$\mathrm{share}_1 = \frac{\lambda_1}{\sum_i \lambda_i} \tag{1.87}$$

它衡量“最大的那个方向占了多少”。
有效维度：

$$d_{\mathrm{eff}} = \frac{\left(\sum_i \lambda_i\right)^2}{\sum_i \lambda_i^2} \tag{1.88}$$

它衡量“真正用了几个维度”。

> ! 为什么有效维度长这样
>
> **先看两个极端。**
> **各向同性：** $\lambda_1 = \cdots = \lambda_d = \lambda$。
>
> $$d_{\mathrm{eff}} = \frac{(d\lambda)^2}{d\lambda^2} = d \tag{1.89}$$
>
> **正好是 d。**
> **一个方向独大：** $\lambda_1 = \lambda$，其余为 0。
>
> $$d_{\mathrm{eff}} = \frac{\lambda^2}{\lambda^2} = 1 \tag{1.90}$$
>
> **正好是 1。**
> **所以 d_{eff} 是一个“介于 1 和 d 之间”的数，越接近 1，分布越像一条线。**
> **它比“PC1 占比”更稳健：PC1 占比只看一个方向，d_{eff} 看整个谱。**

A.11.3 一个手算的例子

四维，特征值 $\lambda = (0.90, 0.05, 0.03, 0.02)$。

$$\text{share}_1 = \frac{0.90}{1.00} = 0.90 \tag{1.91}$$

$$d_{\text{eff}} = \frac{1.00^2}{0.81 + 0.0025 + 0.0009 + 0.0004} = \frac{1.00}{0.8138} = 1.229 \tag{1.92}$$

PC1 占比 90%，有效维度 1.23。
两个数说的不是一件事：“90% 的方差在一个方向上”不等于“只有一个维度”—— 剩下 10% 还摊在三个方向上。
第 7 章报的是两个数一起看，就是这个原因。

A.11.4 实现的三个坑

① **一定要中心化。** 不中心化，第一主成分会指向均值方向。
② **用 `torch.linalg.eigh`，不要用 `eig`。** 协方差矩阵对称，`eigh` 更快更稳，而且返回实数。
③ **特征值可能略小于 0**（浮点误差）。**算有效维度前先 `clamp_min(0)`。**

A.12 用在哪：一张索引

小节	内容	正文用到的地方
A.2.1	配方法	第 5 章后验、第 7 章条件模型
A.2.2	ELBO 分解	第 5 章训练目标
A.2.3	高斯 KL → MSE	第 5 章、第 6 章
A.2.4	score matching	第 6 章、第 5 章 §5.10
A.3	高斯五性质	全书
A.4	Tweedie	第 5 章 $\hat{x}_0$
A.5	softmax 梯度	第 1 章、第 8 章梯度
A.6	方差传播	第 1 章 $\sqrt{d_k}$
A.7	信息论	第 8 章损失→比特
A.8	不等式	ELBO、KL
A.9	矩阵求导	第 5、7、8 章
A.10	DDIM 的代数	第 6 章
A.11	PCA 与有效维度	第 7 章

怎么用这份附录

不要从头读到尾。
遇到下面三种情况再来查：

1. **正文说“可以证明”**——来这里找证明
2. **公式里有个符号不认识**——回 A.1
3. **想不通“为什么要这一项”**——看对应小节的“用在哪”

如果你在准备考试：A.2 的四个推导必须能独立写出来。 **其余可以查。**

B. 术语中英对照

这份表有三个用途：读论文时查、写代码时对字段名、讲课时统一说法。
每一行给三样东西：中文、English、一句话解释。**最后一列是首次出现的章**——想不起来上下文时，回去看那一章。

B.1 注意力与 Transformer

中文	English	一句话解释	章
注意力	attention	用 query 和 key 的相似度给 value 加权求和	1
自注意力	self-attention	Q, K, V 都来自同一串 token	1
交叉注意力	cross-attention	Q 来自图像、K, V 来自文本	1, 8
联合注意力	joint attention	两串 token 拼成一条，做一次自注意力	8
多头注意力	multi-head attention	把 d 维切成 h 份，各自做注意力再拼回	1
缩放点积注意力	scaled dot-product attention	点积除以 $\sqrt{d_k}$ 再 softmax	1
查询	query	“我要找什么”	1
键	key	“我有什么可以被匹配”	1
值	value	“匹配上之后我提供什么”	1
注意力权重	attention weights	softmax 之后的那个概率分布	1
注意力图	attention map	把权重画成热图（**不等于解释**）	1, 4, 8
因果掩码	causal mask	每个位置只能看左边	1
填充掩码	padding mask	忽略补齐出来的位置	1
位置编码	positional encoding	把“第几个”注入表示	1
正弦编码	sinusoidal encoding	用不同频率的 sin/cos 表示位置	1
可学习位置嵌入	learned positional embedding	每个位置一个可训练向量	1

中文	English	一句话解释	章
旋转位置编码	RoPE (rotary position embedding)	用旋转矩阵把相对位置编进 q, k	1
相对位置	relative position	只关心两个位置差多少	1
残差连接	residual connection	$h \leftarrow h + F(h)$	2
层归一化	layer normalization	对每个 token 的特征维归一化	2
RMSNorm	root-mean-square normalization	只除以均方根，不减均值	2
前馈网络	FFN (feed-forward network)	每个位置独立的 两层 MLP	2
激活函数	activation function	非线性，常见 GELU / SwiGLU	2
词嵌入	token embedding	词表里每个 token 一个向量	1
输出投影	output projection	把隐藏状态映射回词表大小	1
权重共享	weight tying	输出投影复用词嵌入矩阵	1
上下文窗口	context window	一次能看多少个 token	1
KV 缓存	KV cache	推理时缓存已算过的 K, V	1
分组查询注意力	GQA	多个 query 头共享一组 K, V	1
多查询注意力	MQA	所有头共享一组 K, V	1
闪存注意力	FlashAttention	分块计算，不把 $N \times N$ 矩阵写进显存	1
头数	number of heads	注意力切成几份	1
头维度	head dimension	每个头多少维	1

⚠ “缩放因子”有两个

注意力里的 scaling factor 是 $1/\sqrt{d_k}$。
VAE 里的 scaling factor 是 0.18215。
两者没有关系，中文都叫“缩放因子”。看到这个词，先看上下文有没有 VAE。

B.2 训练与优化

中文	English	一句话解释	章
预训练	pretraining	在大数据上先训一个通用模型	3
微调	fine-tuning	在特定数据上继续训	3
学习率	learning rate	每步走多远	3
热身	warmup	前若干步把学习率从 0 升上去	3
余弦调度	cosine schedule	学习率按余弦曲线衰减	3
梯度裁剪	gradient clipping	梯度范数超过阈值就按比例缩	3
权重衰减	weight decay	每步把权重往 0 拉一点	3
批量	batch	一次前向用多少样本	3
轮次	epoch	把训练集过一遍	3
交叉熵	cross-entropy	语言模型的训练损失	3
困惑度	perplexity	$e^{\text{交叉熵}}$，可解释成“几选一”	3
过拟合	overfitting	训练集好、验证集差	3
正则化	regularization	任何抑制过拟合的手段	3
丢弃	dropout	训练时随机把激活置零	2
标签平滑	label smoothing	把 one-hot 的 1 改成 0.9	3
混合精度	mixed precision	前向用 bf16，部分运算用 fp32	3
梯度累积	gradient accumulation	累几个小批的梯度再更新一次	3
检查点	checkpoint	存下来的模型权重	3
消融	ablation	去掉一个部件，看性能变化	全书
超参数	hyperparameter	人定的、不通过梯度学的参数	3
验证集	validation set	调超参数用的、不参与训练的数据	3
数据泄漏	data leakage	验证集的信息漏进了训练	3

B.3 扩散模型

中文	English	一句话解释	章
扩散模型	diffusion model	加噪再学会去噪的生成模型	5
前向过程	forward process	逐步加高斯噪声，$q(x_t \mid x_{t-1})$	5
反向过程	reverse process	逐步去噪，$p_\theta(x_{t-1} \mid x_t)$	5
噪声调度	noise schedule	β_t 随 t 怎么变	5
后验	posterior	已知 x_0 和 x_t 时 x_{t-1} 的分布	5

中文	English	一句话解释	章
重参数化	reparameterization	把随机采样写成“确定性变换 + 标准噪声”	5
变分下界	ELBO	对数似然的下界，训练目标从它来	5
去噪器	denoiser	预测噪声（或 x_0、或 score）的网络	5
ε-预测	epsilon prediction	网络输出噪声	5
x_0-预测	x0 prediction	网络输出干净样本	5
v-预测	v prediction	网络输出噪声与样本的混合	5
分数	score	$\nabla_x \log p_t(x)$	5
分数匹配	score matching	直接学 score 的目标	5, A
朗之万动力学	Langevin dynamics	用 score 做梯度上升采样	5
随机微分方程	SDE	连续时间的随机过程	5
概率流 ODE	probability flow ODE	和 SDE 边缘分布相同的确定性方程	5
采样器	sampler	从噪声走到样本的算法	6
DDPM	denoising diffusion probabilistic model	1000 步、每步加噪声的原始采样	5, 6
DDIM	denoising diffusion implicit model	确定性、可跳步的采样	6
步数	steps	采样走多少步	6
函数求值次数	NFE (number of function evaluations)	网络被调用几次，**比较成本的正确单位**	6
离散化误差	discretization error	用有限步近似连续过程带来的误差	6
截断	truncation	把 $\hat{x}_0$ 限制在合理范围	5, 6
分类器引导	classifier guidance	用外部分类器的梯度指方向	7
无分类器引导	classifier-free guidance	用有条件/无条件两个分支的差指方向	7
引导强度	guidance scale (w)	那个差放大多少倍	7
潜空间	latent space	VAE 压缩后的表示空间	8
变分自编码器	VAE	编码到潜空间再解码的自编码器	8

中文	English	一句话解释	章
缩放因子	scaling factor	潜变量乘 0.18215 那个常数	8
潜变量	latent	潜空间里的张量	8
模式坍塌	mode collapse	生成样本挤到一个模式上	7
多样性	diversity	同样条件下样本的分散程度	7
越界	out-of-bounds	样本跑出训练数据的范围	7

B.4 条件生成

中文	English	一句话解释	章
条件	conditioning	生成时额外给的信息 c	7, 8
无条件	unconditional	不给条件的生成	7
空条件	null condition	CFG 里“无条件”那一路的输入	7, 8
条件注入	conditioning injection	条件以什么方式进网络	7, 8
文本编码器	text encoder	把文字变成向量的模型	8
分词器	tokenizer	把字符串切成 token	8
词表	vocabulary	token 的全集	8
子词	subword	比词小、比字大的单位	8
字节对编码	BPE	反复合并最高频相邻对的算法	8
池化	pooling	把一串向量压成一个	8
均值池化	mean pooling	取平均（**丢顺序**）	8
绑定	binding	把属性正确分配给对象	8
属性绑定	attribute binding	“红色的方块”里的红属于方块	8
组合性	compositionality	用有限部件组合出无限句子/图像	8
提示词	prompt	给模型的文字条件	8
负面提示词	negative prompt	希望远离的描述	7, 8
提示词权重	prompt weighting	`word:1.3` 这种强调语法	8
文本丢弃	text dropout	训练时随机把整段文本换成空串	8
标签丢弃	label dropout	类别条件版的同一件事	7
区域提示词	regional prompting	把画面分区，各区用不同提示词	8
控制网	ControlNet	用额外结构图控制生成的旁路网络	8
零卷积	zero convolution	初始化为 0 的卷积，保证“从不动开始”	8
适配器	adapter	冻结主干、只训一小块的结构	8

中文	English	一句话解释	章
联合注意力	joint attention	文本和图像拼成一条做注意力	8
自适应层归一化	adaLN	用条件算出 LN 的缩放和平移	7, 8
时间步嵌入	timestep embedding	把 t 编成向量的正弦编码	5

B.5 评估与工程

中文	English	一句话解释	章
CLIP 分数	CLIP score	图文相似度，**看不见顺序和否定**	8
FID	Fréchet inception distance	生成分布和真实分布的距离	8
人类评估	human evaluation	让人打分（最贵、最可信）	8
视觉问答	VQA	让模型回答关于图像的问题	8
基准	benchmark	一组固定任务和指标	8
槽位准确率	slot accuracy	每个属性/位置分别判对错	8
消融差分	ablation delta	改一个词，输出差多少	8
可复现性	reproducibility	同样的输入能否得到同样的输出	全书
种子	seed	伪随机数发生器的初始状态	全书
推理	inference	训练完之后跑模型	3
推理成本	inference cost	出一次结果要多少计算	6
显存	VRAM	显卡上的内存	6
吞吐	throughput	单位时间能出多少	6
延迟	latency	出一次结果要等多久	6
量化	quantization	用更低位宽存权重	3
蒸馏	distillation	用大模型教小模型	3
LoRA	low-rank adaptation	给权重加一个低秩补丁	8
文本反转	textual inversion	只学一个新词向量	8
DreamBooth	DreamBooth	用少量图微调整个模型	8

B.6 十组最容易混的词

术语表只能解决“叫什么”，解决不了“区别在哪”。这一节专治十组最容易混的。

B.6.1 注意力权重 vs 注意力图

权重是数（$N \times M$ 的矩阵）。**图是把权重画出来的热图。**

“权重不可解释”说的是数本身不携带因果信息；“图不可信”说的是画法会进一步误导（选了哪层、哪个头、有没有基线）。

第 8 章的实测：权重熵 0.998（几乎均匀）仍然绑定成功。**所以问题不在“图画得好不好”，在“数本身不是因果”。**

B.6.2 条件 vs 引导

条件是“给什么”（文本、类别、参考图）。**引导是“把条件的效应放大多少”**（w）。
换条件是改输入；调 w 是改采样过程。前者能带来新信息，后者只能放大已有信息。

B.6.3 潜空间 vs 嵌入空间

潜空间是 VAE 的 $64 \times 64 \times 4$，**有空间结构**，每个格子对应原图一块。
嵌入空间是 CLIP 的 768 维向量空间，**没有空间结构**，只表示语义。
两者都叫“空间”，但一个是图像的下采样，一个是语义的坐标。

B.6.4 两个缩放因子

注意力的 $1/\sqrt{d_k}$：防止点积方差随维度爆炸。
VAE 的 0.18215：把潜变量标准差拉到 1。
一个在模型内部，一个在模型边界。没有关系。

B.6.5 步数 vs NFE

步数是“循环跑了几次”。NFE 是“网络被调用了几次”。
CFG 让每一步调用两次，所以 NFE = 2 × 步数。比较采样器时，必须统一到 NFE，否则不公平。

B.6.6 采样器 vs 调度器

采样器决定“怎么从 x_t 走到 x_{t-1}”（DDPM / DDIM / Euler）。
调度器决定“β_t 长什么样”（linear / cosine）。
diffusers 里 `scheduler` 这个词把两者混在一起了——它的 `scheduler.step()` 同时管噪声表和更新规则。读文档时要看清它指的是哪一个。

B.6.7 分词 vs 编码

分词：字符串 → token 编号（tokenizer）。
编码：token 编号 → 向量（text encoder）。
“换个分词器”和“换个编码器”是两件事，但经常一起换。

B.6.8 参数 vs 超参数

参数是梯度学出来的（权重、偏置）。**超参数**是人定的（学习率、层数、w）。
但 w 有点特殊：它是推理时的超参数，不参与训练。**所以“调 w”不需要重训。**

B.6.9 绑定 vs 组合性

绑定是“红色属于方块”这种**局部对应**。
组合性是“用已知部件组合出新整体”的**全局能力**。
绑定失败是组合性失败的一种，但不是全部——**数量、否定、关系都算组合性。**

B.6.10 训练损失 vs 验证损失

训练损失是模型在见过的数据上的误差。**验证损失**是它在没见过的数据上的误差。
两者差距大 = 过拟合。两者都高 = 欠拟合。
第 8 章的容量对照看的是训练损失——因为要排除的是“容量不够”，**不是“泛化不好”。**

B.7 中文译名的三个取舍

术语表给的是“标准译名”。但有些译名本身就会误导，值得说清楚。

B.7.1 token 为什么不译

“词元”“标记”“令牌”都有人用，本书保留 token。

因为它的边界是“分词器的输出”，不是任何语言学单位。

一个 token 可能是一个汉字、半个英文单词、三个字节、或者一个特殊符号。译成“词”会让你以为它有语言学意义——它没有。

同理，“tokenizer”也不译。

B.7.2 attention 译“注意力”，但它不是“注意”

“注意力”暗示了一种主动的选择——“模型在看哪里”。

数学上它只是“用相似度做加权平均”。没有意图、没有焦点、没有“看”。

第 4 章和第 8 章反复讲“注意力图不是解释”，根源就在这个译名上。如果它一开始叫“加权查询”，很多误解不会发生。

但译名已经约定俗成，改不了。能做的是：每次看到“注意力”，在心里替换成“相似度加权”。

B.7.3 diffusion 译“扩散”，来自物理，但不一样

热力学里的扩散是“粒子从高浓度走向低浓度”。

扩散模型里的“扩散”是“数据被噪声淹没”——方向相反：不是从有序到无序的物理过程，是“人为定义的加噪过程”。

所以“扩散模型”这个名字其实在说“我们借用了扩散的数学形式”，不是说“模型在模拟扩散”。

更准确的名字是“加噪-去噪模型”，但没人这么叫。

B.7.4 latent 译“潜”，是“潜在”不是“潜伏”

“潜空间”对应 latent space，意思是“没有直接观测到的表示空间”。

不是“隐藏起来的空间”，是“解码之前的中间表示”。

中文的“潜”容易让人想到“潜伏”“隐藏”，而它的本意是“潜在的、待解码的”。

> **i** 一个通用的判断
>
> 当一个译名让你产生“模型在做某件像人一样的事”的联想时，多半是译名在骗你。
>
> · “注意力” → 模型在选择
> · “记忆” → 模型在存储
> · “理解” → 模型在懂
>
> 三个都是隐喻，不是机制。看公式，不要看名字。

> **怎么用这张表**
>
> **写论文时**：第一次出现的术语给中英对照，之后统一用中文。
>
> **读代码时**：先查字段名（`encoder_hidden_states`、`scaling_factor`），这些名字比论文里的叫法更容易混。
>
> **讲课时**：把 B.6 的十组讲一遍，比讲十个定义有用。

C. 交互实验索引

全书 15 个交互实验，都在 `experiments/` 下，双击就能打开。
不联网、不需要服务器、不需要装任何东西——每个页面是一个自包含的 HTML，数据已经嵌在页面里。

C.1 怎么用

三种打开方式：

1. **直接双击** `experiments/exp-1-1.html`
2. **从正文点链接**（HTML 版每一节都有入口）
3. **在 PDF 里看静态快照**（`assets/exp-static/`，每个实验一张）

每个实验页都有三个部分：

- **控件**（滑块、按钮）——改参数
- **画布**——实时重算
- **结论框**——告诉你“该看什么”

页面里的数字全是现算的，不是预先录好的动画。**所以你可以把参数拖到书里没写过的位置，看它怎么变。**

> ℹ 为什么 PDF 里是静态图
>
> **PDF 装不下交互。**
> **每个实验在 PDF 里配一张快照，选的是“最能说明问题的那一帧”**，不是初始帧。比如实验 6-2 的快照是 $b = 0.10$、输出已经塌缩到 54% 的那一帧。
> **想看完整的，回到 HTML 版。**

C.2 总表

编号	章	一句话	主要控件	数据来源
1-1	1	缩放因子与 softmax 饱和	d_k、位置数、种子	现算
1-2	1	真实模型学到的注意力	层、头、是否缩放	预计算（42 万参数模型）
1-3	1	改 Q, K, V，看注意力怎么变	九个数字、五个预设	现算（3 token）
3-1	3	温度、top-k、top-p	步数、τ、k、p、顺序	真实模型分布
3-2	3	先截断还是先调温	τ、p、步数	真实模型分布
3-3	3	KV 缓存省下了什么	生成长度、提示词长度	现算
4-1	4	逐层逐头看注意力图	层、头	真实权重 + 随机对照

编号	章	一句话	主要控件	数据来源
4-2	4	消融实验室	点头屏蔽	预计算消融
5-1	5	前向过程	t	现算（700 点）
5-2	5	反向过程：加不加噪声	t、并排开关	预计算轨迹
5-3	5	噪声调度	调度、T、β 端点	现算
6-1	6	采样器对比	采样器、步数	现算（1D 高斯）
6-2	6	偏差诊断	偏差 b、步数、η	现算
7-1	7	CFG 引导强度	类别、w	预计算（每 w 600 点）
8-1	8	绑定实验室	编码器、类别	预计算（每类 300 点）

15 个实验里，7 个是“现算”、8 个是“预计算”。

区别在数据量：现算的页面只需要几十个参数；预计算的需要真实模型的采样结果（几十万个浮点数），**只能离线算好再嵌进去**。

C.3 逐个说明

C.3.1 实验 1-1　缩放因子与 softmax 饱和

拖 d_k 从 4 到 512。

看三件事同时发生：logits 的分布变宽、softmax 变尖、梯度（最大概率处的斜率）塌向 0。

结论：**$\sqrt{d_k}$ 不是“让数值好看”，是“让 softmax 留在还能学的位置”。d_k 小时看不出差别**——这就是正文案例里“小模型上没事”的原因。

C.3.2 实验 1-2　真实模型学到的注意力

一个真实的 2 层 4 头模型（425,984 参数），在真实唐诗上训练后的注意力。

可以切层、切头，也可以切到“不缩放”的那个模型做对照。

看什么：正常的模型有清晰的局部模式（盯前一位、盯句首）；不缩放的模型注意力摊平，几乎没有结构。

数据来源：教师机离线算好，页面只做回放。

C.3.3 实验 1-3　改 Q, K, V，看注意力怎么变

3 个 token、2 维，九个数字全部可改。

五个预设：正文那个例子、只让自己可见、所有位置一样、制造一个尖锐分布、随机。

结论：注意力前向传播没有黑箱——改一个数，打分、softmax、输出立刻跟着变。**你可以亲手制造“注意力退化成平均”的那个 bug。**

C.3.4 实验 3-1　采样实验室

四个滑块：生成到第几步、温度 τ、top-k、top-p。

还有一个关键的开关：先截断后调温 / 先调温后截断。

结论：同样一组 (τ, p)，只换顺序，候选集能差几十倍。这不是理论上的差别，是你会不会采到那个词。

C.3.5 实验 3-2　顺序陷阱

3-1 的“顺序”那一个维度，单独放大。

固定 τ 和 p，只换先后，看候选集怎么变。

结论：`top_p` 和 `temperature` 不是可交换的操作。 很多实现出错就出在这个顺序上。

C.3.6 实验 3-3 KV 缓存省下了什么

每一格代表“某一步里某个位置被重新算了一次”。
拖生成长度 N 和提示词长度 P，看格子怎么被涂满。
结论：没有缓存时，第 i 步要重算前 i 个位置，总量是 $O(N^2)$；有缓存时是 $O(N)$。提示词越长，省得越多。

C.3.7 实验 4-1 逐层逐头看注意力图

左边是训练好的模型，右边是随机初始化的同结构模型。
切层（0/1）、切头（H0–H3）。
结论：随机模型也会画出“看起来有结构”的图。没有对照，你分不清“学到了”和“初始化碰巧”。

C.3.8 实验 4-2 消融实验室

点一个头把它屏蔽掉，看验证集 loss 和生成文本怎么变。
结论：有些头一屏蔽 loss 就涨，有些头屏蔽了几乎没变化。“这个头很重要”必须用消融证明，不能看注意力图猜。

C.3.9 实验 5-1 前向过程实验室

700 个点，同一个 x_0、同一串噪声，拖 t 从 0 到 1000。
结论：双月分布逐渐溶成标准正态。 注意 $t = 300$ 时形状已经看不出来，但 $\bar{\alpha}_t$ 还有 0.40——“看起来是噪声”和“数学上是噪声”不是一回事。

C.3.10 实验 5-2 反向过程实验室

同一个模型、同一批初始噪声，只改“反向每一步加不加 $\sigma_t z$”。
并排回放两条轨迹。
结论：加噪声的轨迹每一步都在抖，不加的是一条平滑曲线。两条轨迹的终点分布相同，路径完全不同—— 这就是 DDPM 和 DDIM 的关系。

C.3.11 实验 5-3 噪声调度实验室

改调度类型（线性/余弦）、步数 T、β 的起点和终点。
看 $\bar{\alpha}_t$ 和 log-SNR 的形状。
红色区域是“信号已被破坏殆尽”的区间。
结论：线性调度在高噪声端浪费了很多步； 余弦调度把步子更多分给“信号还在”的区间。

C.3.12 实验 6-1 采样器对比实验室

同一个解析问题（1D 高斯 + 完美去噪器），比较 DDPM、DDIM、Heun。
拖步数，看精度随 NFE 怎么变。
结论：同样的 NFE 下，二阶的 Heun 明显好于一阶的 DDIM， DDIM 好于 DDPM。 “步数少”不等于“便宜”——要统一到 NFE。

C.3.13 实验 6-2 偏差诊断实验室

给完美去噪器注入可控偏差 b，看 DDIM 会不会发散。
结论和直觉相反：均匀偏差不会让 DDIM 发散，只会让输出塌缩到目标的 $1 - b$ 倍。 真正让 DDIM 崩的，是“误差随 t 变化”或“模型把误差放大”。
这一条推翻了正文初稿的一个假设——实验做出来才发现。

C.3.14 实验 7-1　CFG 引导强度实验室

选一个类（外月/内月），拖 w 从 0 到 40。
看两件事同时发生：类内越来越纯，样本越来越少、越来越散。
结论：准确率在 $w = 3$ 就饱和，之后它不再告诉你任何事；标准差和越界率继续恶化。“准确率”是一个会骗人的指标。

C.3.15 实验 8-1　绑定实验室

选编码器（均值池化/交叉注意力）、选类别（RR/RB/BR/BB）。
每个点是一个生成样本，画在“左点 y × 右点 y”平面上。
结论：均值池化在 RB 和 BR 上给出镜像对称的两团，各占一半—— 因为它拿到的条件向量完全相同。 交叉注意力只落在目标象限。

> ! 第 3 步是实验 8-1 的全部意义
>
> **同一个词袋、两种顺序、同一个分布。**
> **这不是“生成质量差”，是“条件里没有信息”。看到镜像对称的两团，就是绑定失败的指纹。**

C.4 三条推荐路线

C.4.1 六十分钟：只看“会骗人的指标”

时间	实验	得到什么
0–15	7-1	准确率饱和，标准差爆炸
15–30	8-1	0.55 不是“一半对”，是零信息
30–45	6-2	均匀偏差不会发散，只会塌缩
45–60	4-1	随机模型也有“结构”

四个实验的共同点：都推翻了“看一个数就够”的直觉。

C.4.2 半天：按章走一遍

上午：1-1 → 1-3 → 3-1 → 3-2 → 3-3（上篇的机制）
下午：5-1 → 5-2 → 5-3 → 6-1 → 7-1（下篇的机制）
每个 20–30 分钟，够你把参数拖到书里没写过的位置。

C.4.3 一学期：把实验当作业的起点

每个实验页都可以改。 把“只读”变成“改写”，是这份材料真正的用法—— 附录 D 的作业就是按这个思路设计的。

C.5 自己改一个实验

每个实验页的结构都一样：

```
<h1>实验 X-Y  标题</h1>
<p class="sub">一句话说明</p>
<div class="row">
  <div class="ctl">  ← 控件
    <label>参数</label>
    <input type="range" ...>
  </div>
  <canvas id="cv"></canvas>  ← 画布
```

```
</div>
<div class="task">动手：...</div>
<script>
const D = {...};        ← 数据
function draw() {...}   ← 重绘
</script>
```

改三处就能做出一个新实验：

1. **换数据**：改 `const D = {...}`
2. **换算公式**：改 `draw()` 里的计算
3. **换结论**：改 `.task` 里那段话

> ⚠ 改完必须重新截图
>
> **PDF 里的快照不会自动更新。**
> **改完 HTML，要跑：**
>
> ```
> python3 qa/snap_experiments.py exp-8-1
> ```
>
> **它用无头浏览器截图，按 `qa/snap_experiments.py` 里 `SPECS` 的尺寸和 URL 参数。如果新实验的“最能说明问题的那一帧”不是默认帧，在 `SPECS` 里给它加一个 URL 参数。**

还有一个隐藏要求：页面加载完成后要设 `document.body.dataset.qaReady = "1"`。这是给自动检查用的——没有它，截图脚本不知道什么时候算“画完了”。

D. notebook 环境与作业

这本书的所有数字都能在你自己的机器上跑出来——包括一台没有显卡的笔记本。
这一章说清楚三件事：环境怎么装、每个脚本跑多久、作业怎么做。

D.1 环境：一台 CPU 笔记本就够

全书只依赖两个库：PyTorch 和 NumPy。
不需要 transformers、不需要 diffusers、不需要下载任何预训练权重。所有模型都是几十万参数的玩具模型，从零训练。

D.1.1 安装

```
# 用 conda 或 micromamba 建一个环境
micromamba create -n tdm python=3.10 -c conda-forge
micromamba activate tdm

# 安装 PyTorch（CPU 版即可）
pip install torch numpy
```

验证：

```
python -c "import torch; print(torch.__version__, torch.cuda.is_available())"
```

输出 `2.x.x False` 也没关系——`False` 表示没有 GPU，**全书照样能跑**。

D.1.2 为什么 CPU 够用

因为模型小。下面是实测数字（不是估算）：

任务	规模	CPU 用时
第 1–4 章的字符级 GPT	42 万参数、1500 步	**约 1.1 分钟**（150 步实测外推）
第 5 章的扩散模型	3000 步训练	**6.7 秒**
第 5 章的采样	1000 个点 × 1000 步	**0.9 秒**
第 8 章的文本条件模型	6000 步训练	**25.6 秒**
第 8 章的采样	400 个样本 × 1000 步	**1.4 秒**

这些数字是在一台普通服务器 CPU 上测的。 你的笔记本可能慢 2–3 倍，但仍然在“泡杯茶”的量级。

> ℹ 书里的 GPU 数字和你的不一样，正常
>
> **正文标注的耗时是 RTX 3090 上的。**
> **GPU 快在“大矩阵、大 batch”；这些玩具模型的矩阵太小，GPU 的并行优势发挥不出来，有时甚至比 CPU 还慢**（kernel 启动开销占比高）。
> **所以：如果你的 CPU 跑出来比书里的 GPU 数字还快，不要惊讶。**

D.1.3 把 device 改成 CPU

`tiny_gpt.py` 的默认 `device` 是 `"cpu"`，开箱即用。
`tiny_diffusion.py` 的默认值是 `"cuda:1"`（作者机器的第二块卡）。在你的机器上，把它改成 `"cpu"`：

```
from tiny_diffusion import Config
cfg = Config(device="cpu")
```

或者用环境变量批量改（不推荐，容易忘）：

```
# 只影响当前 shell
export TDM_DEVICE=cpu
```

最稳的做法：每个脚本开头显式写 `Config(device="cpu")`。

D.2 目录结构

```
ai-mechanisms-book/
├── chapters/            ← 正文（.qmd）
├── appendix/            ← 附录 A–E
├── notebooks/           ← 全部可运行代码
│   ├── tiny_gpt.py          第 1–4 章的模型
│   ├── tiny_diffusion.py    第 5–8 章的扩散骨架
│   ├── conditional_diffusion.py  第 7 章的条件模型
│   ├── text_conditioning.py      第 8 章的文本条件
│   ├── run_chN_case.py      每章的主实验
│   └── ...
├── data/                ← 语料和实验结果（JSON）
├── experiments/         ← 15 个交互实验（HTML）
├── assets/              ← 插图、样式、静态快照
└── qa/                  ← 生成图、截图、lint 的脚本
```

三个原则：

1. **`notebooks/` 里只有代码，没有数据**——数据在 `data/`
2. **每个脚本都能单独运行**——不依赖 notebook 的执行顺序
3. **结果写成 JSON**——正文的数字和 `data/` 里的文件一一对应

D.3 每章跑什么

章	主脚本	依赖	产出	CPU 量级
1	`run_ch1_case.py`	`tiny_gpt.py`	`data/ch1_case.json`	分钟级
2	`run_ch2_case.py`	`tiny_gpt.py`	`data/ch2_case.json`	分钟级
3	`run_ch3_case.py`	`tiny_gpt.py`	`data/ch3_case.json`	分钟级
4	`run_ch4_case.py`、`run_ch4_extra.py`	`tiny_gpt.py`	`data/ch4_*.json`	几分钟
5	`run_ch5_case.py`、`run_ch5_schedule.py`	`tiny_diffusion.py`	`data/ch5_*.json`	秒级
6	`run_ch6_case.py`、`run_ch6_bias_grid.py`、`run_ch6_grid.py`	`tiny_diffusion.py`	`data/ch6_*.json`	分钟级

章	主脚本	依赖	产出	CPU 量级
7	`run_ch7_case.py`、`run_ch7_extra.py`、`run_ch7_more.py`	`conditional_diffusion.py`	`data/ch7_*.json`	几分钟
8	`run_ch8_case.py`、`run_ch8_ablation.py`、`run_ch8_ablate2.py`、`run_ch8_capacity.py`	`text_conditioning.py`	`data/ch8_*.json`	约十分钟

“量级”不是精确值——你的机器可能差两三倍。精确的单项数字在 D.1 那张表里，那些是实测的。
为什么第 8 章反而久？ 因为它要训五个模型、每个都采样几千个样本。**批小、步多，GPU 的并行优势发挥不出来，CPU 和 GPU 的差距会缩小。**

先跑最短的

如果你想确认环境没问题，先跑：

```
cd notebooks
python run_ch5_case.py
```

它只要一分钟，而且会打印一组你能和正文对上的数。跑通了，再跑别的。

D.4 复现书里的数字

每一步都按“跑脚本 → 看 JSON → 对正文”的顺序。

D.4.1 第 1 步：确认输出位置

所有脚本都把结果写到 `data/`，不覆盖原文件。如果目录里已经有同名 JSON，脚本会覆盖它——想保留原结果，先备份。

D.4.2 第 2 步：对照三个数

每个脚本都会打印几个关键数。以第 8 章为例：

```
mean RB: left_blue 0.455 right_blue 0.55
cross RB: left_blue 0.0 right_blue 1.0
cross: 对调后最大逐元素差 = 9.722e-01
```

和正文对不上时，按这个顺序查：

1. **种子**：脚本里有没有 `torch.manual_seed`？
2. **设备**：CPU 和 GPU 的浮点结果可能差 10^{-7}，**不影响结论**
3. **版本**：PyTorch 2.x 的不同小版本，初始化可能不同
4. **步数**：`steps` 参数有没有被改过？

D.4.3 第 3 步：允许的误差

下面这些差异是正常的，不算复现失败：

量	允许的差异	原因
训练损失	±0.005	初始化、浮点顺序
准确率	±0.03	采样噪声
对调差	量级相同即可	具体值依赖模型
注意力熵	±0.05 bit	同上

真正不能接受的是“定性相反”： 如果均值池化在你的机器上绑定了，那一定是哪里错了。

D.5 作业

五组，对应五个部分。每组 3–4 题，建议 2–3 小时。

D.5.1 作业一 注意力与 Block（第 1–2 章）

1.1 验证 $\sqrt{d_k}$

用 NumPy 生成 $q, k \sim \mathcal{N}(0,1)$，取 $d = 4, 64, 512$， 各算 10000 次点积，报告标准差。

要求：画出“标准差 vs d”的图，并在同一张图上画 $\sqrt{d}$ 的曲线。 **问**：实测斜率是多少？和理论差多少？

1.2 把 bug 加回去

把 `Config(scaled=False)` 打开，训 1500 步。

报告：训练损失、验证损失、注意力熵（每层每头平均）。 **问**：和正常版本差多少？为什么正文说“小模型上看不出差别”？

1.3 改头数

把 `n_head` 从 4 改成 8，`d_model` 保持 128。

先手算：参数量变不变？每个头的维度变不变？ **再实测**：训练 1500 步，比较 loss。

1.4 RoPE 外推

用 `rope_extrapolation.py`，在长度 64 上训练、在 128 上推理。

报告：两种位置编码（学习式 vs RoPE）在外推长度上的 loss。**问**：哪一个崩得更厉害？为什么？

D.5.2 作业二 训练、采样与可解释性（第 3–4 章）

2.1 顺序陷阱

实现“先截断后调温”和“先调温后截断”两种采样。

报告：固定 $(\tau, p) = (0.7, 0.9)$，两种顺序的候选集大小。 **问**：差多少倍？哪个更保守？

2.2 数一数 KV 缓存

不跑代码，先手算：生成长度 $N = 100$、提示词 $P = 10$，无缓存和有缓存各要做多少次 $QK^\top$ 的行计算？

再用 exp-3-3 验证。

2.3 消融热图

用 `run_ch4_extra.py` 的消融数据，画一张“层 × 头”的 loss delta 热图。

报告：最重要的三个头、最不重要的三个头。 **问**：最重要的头，它的注意力图有什么共同模式？

2.4 随机对照

把训练好的模型和随机初始化的模型并排， 比较它们的注意力熵。

问：随机模型的熵更高还是更低？这说明什么？

D.5.3 作业三 扩散的数学与采样（第 5–6 章）

3.1 手推后验

不看附录，自己推一遍 $q(x_{t-1} \mid x_t, x_0)$ 的均值和方差。

再用代码数值验证：取 $x_0 = 1$、$t = 500$，采样 10000 次 x_t，统计 x_{t-1} 的均值和方差，和公式比。

3.2 实现 DDIM

从 DDPM 的采样循环改出 DDIM（$\eta = 0$）。

报告：20、50、100 步下的样本均值和协方差。 **问**：多少步之后，结果和 1000 步的 DDPM 无法区分？

3.3 偏差实验

复现实验 6-2：给完美去噪器注入偏差 b。

报告：$b = 0.01, 0.05, 0.10, 0.20$ 时输出的均值和标准差。 **问**：为什么均匀偏差只导致塌缩，不导致发散？

3.4 三种参数化

实现 ε-预测、x_0-预测、v-预测。

报告：三者在一组固定的 t 上的损失量级。 **问**：哪一个的动态范围最小？和正文的 8500 倍对得上吗？

D.5.4 作业四　CFG（第 7 章）

4.1　三段曲线
跑 `run_ch7_case.py` 的 w 扫描。
画三张图：准确率、样本标准差、越界率。 **标出**：欠引导 / 正常 / 过引导三段的分界。
4.2　负向引导
用 $w = -1$ 复现负引导。
报告：准确率、多样性、越界率。 **问**：为什么负引导会让准确率掉到 0.05 左右?
4.3　PCA 有效维度
采 1000 个 $w = 40$ 的样本，自己算协方差矩阵的特征值。
报告：PC1 占比、有效维度 $(\sum \lambda)^2 / \sum \lambda^2$。 **问**：去掉最大的 1% 样本后，结论变不变?
4.4　不丢弃标签
训练时不丢弃标签（`drop=0`），再扫 w。
报告：$w = 5$ 时的越界率。 **问**：为什么没有空条件，CFG 就崩了?

D.5.5 作业五　文本条件（第 8 章）

5.1　三个数
跑 `run_ch8_ablation.py`。
报告：RB/BR 准确率、两者之和、对调差。 **问**：为什么“和”比单个准确率更能说明问题?
5.2　去掉位置嵌入
把 `variant="no_pos"` 打开，重训。
先预测：准确率会是多少？对调差会是多少？ **再实测**，解释 0.602 这个数是怎么来的。
5.3　自己训一个 BPE
用 `bpe_demo.py`，换一份你自己的语料（比如你的课程笔记）。
报告：合并 1000 次后，77 个 token 能装多少汉字。 **问**：和书里的 74 个差多少？为什么?
5.4　空条件
分别用“学出来的 NULL token”和“全零向量”当空条件，扫 $w = 1, 3, 5$。
报告：越界率、多样性。 **问**：为什么零向量会崩，而 NULL token 不会?

> **i** 评分标准
>
> **每题按四档：**
>
> | 档 | 标准 |
> |---|---|
> | A | 数字对、解释对、能指出边界 |
> | B | 数字对、解释基本对 |
> | C | 数字对，但解释错或缺失 |
> | D | 数字不对，或没跑 |
>
> **“解释对”比“数字对”更难，也更值钱。 如果数字对不上但你能说清楚为什么，给 B。**

D.6 大作业（三选一）

D.6.1 题目 A：复现一篇论文的核心图

从下面三篇里选一篇，复现它的一张核心图：

- DDIM（采样步数与质量）
- CFG（引导强度与多样性）
- MMDiT（联合注意力 vs 交叉注意力的成本）

要求：

1. 用本书的玩具模型，不要求复现原论文的规模
2. 报告“哪些结论复现了、哪些没有”

3. **必须有一张图是你自己算的**，不能截图

D.6.2 题目 B：设计一个新的条件接口

给扩散模型加一种新条件：点击、骨架、布局、音频包络——任选。
要求：

1. 按第 8 章“六个问题”回答设计决策
2. 在 2D 玩具数据上验证：**这个条件真的被用上了吗?**
3. 必须做**对调实验**（换条件、看输出）

评分重点：**你有没有证明“条件生效”**，而不是“图画得好看”。

D.6.3 题目 C：写一个条件诊断工具包

输入：一个训练好的条件模型 + 一对镜像条件。 **输出**：三档判断（绑定成功 / 部分 / 没绑定）+ 数字依据。
要求：

1. 至少三个指标（准确率、和、对调差、熵，任选）
2. 能在第 7 章和第 8 章的模型上跑通
3. 附一份“误判分析”：什么情况下它会判错

D.7 作业提示

只给方向，不给答案。 照着做之前，先自己想十分钟。

D.7.1 作业一

1.1 标准差应该正比于 $\sqrt{d}$。**先画散点再拟合斜率**，不要只报一个数——**看图能发现“大 d 时点积的分布是不是还近似高斯”**。
1.2 重点不是 loss 差多少，是**注意力熵差多少。 如果熵几乎一样，说明这个规模下 softmax 没饱和**——回去看第 1 章的方差计算。
1.3 参数量的答案：**不变**。因为 d_{model} 不变，多头只是把同一个矩阵切成更多份。 **如果你算出来变了，检查是不是把每个头的投影矩阵当成了独立的**。
1.4 RoPE 的外推会“渐变”，学习式位置编码会“突崩”。 **报告崩溃发生在第几个位置**，比报告平均 loss 有信息量。

D.7.2 作业二

2.1 先截断后调温：候选集是“先砍掉尾巴，再重新分配概率”。先调温后截断：温度改变了分布形状，再砍。 **两种顺序的候选集大小比，应该和实验 3-2 对得上**。
2.2 无缓存：第 i 步要算 $P+i$ 个位置，总量是 $\sum_{i=1}^{N}(P+i)$。有缓存：每步只算 1 个新位置，加上注意力的 $P+i$ 次点积。 **把“矩阵乘法”和“点积”分开数**，别混在一起。
2.3 热图的颜色用 Δ loss，不要用原始 loss。 **原始 loss 的层间差异会把头的差异盖住**。
2.4 随机模型的熵通常**更高**（更接近均匀）。 **如果它更低，先检查初始化尺度**——这本身就是一个值得报告的发现。

D.7.3 作业三

3.1 数值验证的关键是**固定 x_0 和 t，只对 ε 采样。 别把 x_0 也采样了**，否则你验证的是边缘分布，不是后验。
3.2 DDIM 的更新式在附录 A.10。 **先实现 $\eta=0$，再实现 $\eta=1$ 验证它能退回 DDPM**。
3.3 “均匀偏差”指 $\hat{\varepsilon}=\varepsilon+b$，$b$ 和 t、x 都无关。 **先预测输出均值会偏移多少，再实测**。
3.4 动态范围用“分位数”比用“最大最小值”稳。 **报告 1%–99% 分位数的比值**。

D.7.4 作业四

4.1 三段的分界不是固定的，它随模型质量移动。**报告“你的模型”的三段分界，而不是抄书里的 2 和 20。**
4.2 负引导的准确率掉到 0.05，不是“变差了”，是“被推向了反面”。**看混淆矩阵**，不要只看准确率。
4.3 有效维度对极端值敏感。**先算一遍全样本，再算一遍去掉最大 1%，两个数都报。**
4.4 `drop=0` 时，空条件从未被训练。**CFG 的 $w > 1$ 会放大一个“模型没见过的输入”的输出——这正是第 7 章“零向量”那个坑的另一个版本。**

D.7.5 作业五

5.1 “和”接近 1 说明两个类互补。**如果你的和是 1.2，先检查是不是采样种子没对齐。**
5.2 预测的时候写下两个数：准确率和对调差。**对调差是你的“真相判据”，准确率不是。**
5.3 语料越小，BPE 越容易“记住”而不是“学会”。**用留出集测，不要用训练集。**
5.4 零向量崩的原因是“分布外”，不是“零”。**验证方法：把零向量换成一个小范数的随机向量，看是否也崩。**

D.8 大作业评分表

三个题目共用一张表，五个维度，每维 0–4 分，总分 20。

维度	0 分	2 分	4 分
问题定义	没写清楚要回答什么	有目标但不可检验	一句话能说清，且能设计实验
实验设计	只有演示，没有对照	有对照但缺一个关键变量	对照完整，能排除替代解释
数据与复现	数字来源不明	有脚本但跑不通	一条命令复现，含随机种子
解释	只报数不解释	解释与数据基本一致	能指出边界和反例
表达	图看不懂	图能看懂但缺标注	图自解释，结论一句话

> ⚠ 最容易丢分的地方
>
> **不是“结果不好”，是“没有对照”。**
> **一个跑了三天的实验，如果没有 baseline，在评分表上是 0 分的“实验设计”。**
> **反过来，一个两小时的玩具实验，如果有完整的对照和一句诚实的边界，可以拿满分。**

D.9 常见问题

Q：`torch.cuda.is_available()` 是 False，能跑吗?
能。全书不需要 GPU。把 `Config(device="cpu")` 显式写上即可。
Q：报错 `Expected a 'cpu' device type for generator`
这是生成器和张量不在同一个设备上。`torch.Generator()` 默认在 CPU；如果你的张量在 GPU 上，要么把生成器也放到 GPU，要么把数据放回 CPU 再取 batch。
Q：matplotlib 画图时中文变成方块
设置字体：

```
import matplotlib
matplotlib.rcParams["font.sans-serif"] = ["Noto Sans CJK SC", "SimHei"]
matplotlib.rcParams["axes.unicode_minus"] = False
```

Q：第一次运行很慢，后面很快

正常。第一次要导入 PyTorch、初始化线程池。**第二次开始才是真实速度**。

Q：结果和书里差一点

先看 D.4 的“允许的误差”。**差 10^{-3} 是浮点，差 0.1 是 bug**。

Q：内存不够

把 `batch_size` 减半。**这些模型的 batch 从 256 降到 64，结论不变**。

Q：能不能用 Jupyter Notebook?

能，但**书里的脚本是 `.py`，不是 `.ipynb`。用 `%run run_ch5_case.py` 或者在 IDE 里直接跑。这样做的原因**：`.py` 能 diff、能 lint、能被 CI 跑；`.ipynb` 的 JSON 结构不适合版本控制。

一句话

这套代码的设计目标是：一个没有显卡的学生，一个下午能跑完全书。如果你花了超过半天还没跑通第一个脚本，那是环境问题，不是你的问题。

E. 延伸阅读与开源项目

这份附录不是“参考文献列表”，是“读什么的顺序”。
每一篇都标了三件事：它能给你什么、可以跳过什么、读之前需要哪一章。

E.1 怎么读一篇论文

三遍读法（Keshav 的经典方法，改成了这本书的语境）：
第一遍（10 分钟）：只读标题、摘要、图、结论。
问自己三个问题：

- 它解决什么问题？
- 它改了什么？（数据、结构、目标函数、采样）
- 它的图里，哪一张是“核心证据”？

第二遍（1 小时）：读方法，跳过证明。
重点看：条件的形状（第 8 章那四个问题）、训练的配置（丢弃率、batch）、以及“它和 baseline 的唯一差别”。
第三遍（半天）：复现一个数。
不复现，你永远不知道它的“提升”来自哪里。这本书的每一章都是按第三遍的标准写的。

> ⚠ 读文生图论文的四个筛子
>
> ① **条件从哪来？**（文本/图像/布局） ② **怎么进模型？**（拼接/AdaGN/交叉/联合/旁路） ③ **在哪一层？**（几层、什么分辨率） ④ **怎么训练？**（丢弃率、caption 来源、batch）
> **四个问题答不上来的论文，可复现性要打问号。**

E.2 按章列原始论文

E.2.1 第 1 章　注意力

Vaswani et al. (2017). Attention Is All You Need. 读什么：§3.2 的缩放点积、§3.3 的多头、图 1。**跳过**：训练细节、机器翻译的实验。**注意**：论文里对“为什么除以 $\sqrt{d_k}$”只有一句脚注——**本书第 1 章把它算了出来。**

Bahdanau, Cho, Bengio (2014). Neural Machine Translation by Jointly Learning to Align and Translate. 读什么：注意力最早的样子——**它是从“翻译对齐”里长出来的。跳过**：RNN 的实现细节。**读它能得到什么**：理解“注意力”最初不是“自己看自己”。

Su et al. (2021). RoFormer: Enhanced Transformer with Rotary Position Embedding. 读什么：§3 的旋转构造。**跳过**：实验部分。**配第 1 章 §1.9 读。**

Dao et al. (2022). FlashAttention: Fast and Memory-Efficient Exact Attention with IO-Awareness. 读什么：为什么“不把矩阵写出来”能快 2–4 倍。**跳过**：CUDA 实现。**读它能得到什么**：知道“$O(N^2)$”是计算复杂度，不是显存复杂度。

Ainslie et al. (2023). GQA: Training Generalized Multi-Query Transformer Models. Shazeer (2019). Fast Transformer Decoding: One Write-Head is All You Need. 读什么：MQA/GQA 怎么省 KV 缓存。**配第 1 章 §1.8 读。**

E.2.2 第 2 章　Block

Ba, Kiros, Hinton (2016). Layer Normalization. 读什么：为什么是“每个 token 自己归一化”而不是 batch 维。**跳过**：RNN 的实验。

Xiong et al. (2020). On Layer Normalization in the Transformer Architecture. 读什么：Pre-LN 和 Post-LN 的梯度分析。**读它能得到什么**：理解“为什么现在的模型几乎都用 Pre-LN”。

Zhang & Sennrich (2019). Root Mean Square Layer Normalization. Shazeer (2020). GLU Variants Improve Transformer. 读什么：两个“去掉一点东西反而更好”的例子。**配第 2 章的 Block 变体读**。

E.2.3 第 3 章　训练与推理

Radford et al. (2019). Language Models are Unsupervised Multitask Learners (GPT-2). 读什么：§2 的输入表示、§3 的生成。**跳过**：zero-shot 的实验列表。

Brown et al. (2020). Language Models are Few-Shot Learners (GPT-3). 读什么：§2 的模型规模、§3 的 in-context learning。**注意**：它和本书的关系是“同一个结构，放大 1000 倍”。

Kaplan et al. (2020). Scaling Laws for Neural Language Models. Hoffmann et al. (2022). Training Compute-Optimal Large Language Models (Chinchilla). 读什么：两篇一起读，看“参数和数据该怎么分配”的结论怎么变。**跳过**：拟合细节。

Holtzman et al. (2019). The Curious Case of Neural Text Degeneration. 读什么：top-p 的动机——**为什么“取最大”会退化**。**配第 3 章的采样实验室读**。

E.2.4 第 4 章　可解释性

Jain & Wallace (2019). Attention is not Explanation. Wiegreffe & Pinter (2019). Attention is not not Explanation. 读什么：两篇对着读。**它们的争论比结论更有价值**。**读它能得到什么**：知道“注意力图≠解释”不是一个新发现，是一个老争论。

Abnar & Zuidema (2020). Quantifying Attention Flow in Transformers. 读什么：attention rollout 的公式（第 8 章引用过）。**注意**：它解决的是“多层怎么合并”，不解决“权重是不是因果”。

Vig (2019). A Multiscale Visualization of Attention in the Transformer Model (BertViz). 读什么：不用读，去用。**它是一个工具，不是一篇理论**。

E.2.5 第 5 章　扩散的数学

Sohl-Dickstein et al. (2015). Deep Unsupervised Learning using Nonequilibrium Thermodynamics. 读什么：扩散模型的最初想法。**跳过**：物理类比。**读它能得到什么**：知道“加噪去噪”不是 2020 年才有的。

Ho, Jain, Abbeel (2020). Denoising Diffusion Probabilistic Models. 读什么：§3 的 ELBO 分解、§4 的简化目标。**必读**：这是整本书下篇的源头。**配附录 A.2 读**。

Song et al. (2021). Score-Based Generative Modeling through Stochastic Differential Equations. 读什么：§3 的 SDE、§4 的 probability flow ODE。**跳过**：§5 的采样器细节（第 6 章再读）。**读它能得到什么**：理解“扩散=解微分方程”。

Nichol & Dhariwal (2021). Improved Denoising Diffusion Probabilistic Models. 读什么：余弦调度、学习的方差。**配第 5 章 §5.8 读**。

Salimans & Ho (2022). Progressive Distillation for Fast Sampling of Diffusion Models. 读什么：v-预测的定义（附录里）。**注意**：正文第 5 章 §5.7 关于“动态范围”的论证，原始出处在这里。

E.2.6 第 6 章　采样器

Song, Meng, Ermon (2020). Denoising Diffusion Implicit Models (DDIM). 读什么：§3 的非马尔可夫前向、§4 的确定性采样。**必读**：第 6 章的主角。

Lu et al. (2022). DPM-Solver: A Fast ODE Solver for Diffusion Probabilistic Model Sampling. 读什么：为什么“指数积分器”比 Euler 好。**跳过**：收敛性证明。

Liu et al. (2022). Pseudo Numerical Methods for Diffusion Models on Manifolds (PNDM). Karras et al. (2022). Elucidating the Design Space of Diffusion-Based Generative Models (EDM). 读什么：EDM 的 §3——**它把“噪声调度、采样器、训练目标”拆成三个正交的轴**。**读它能得到什么**：一张比本书更完整的“设计空间”地图。

Dhariwal & Nichol (2021). Diffusion Models Beat GANs on Image Synthesis. 读什么：分类器引导的推导。**配第 7 章读**。

Chen (2023). On the Importance of Noise Scheduling for Diffusion Models. 读什么：为什么“调度”比“结构”更重要。**配第 5 章 §5.8 读**。

E.2.7 第 7 章 CFG

Ho & Salimans (2022). Classifier-Free Diffusion Guidance. 读什么：整篇（很短）。**这就是第 7 章的源头。 注意**：它没有解释“为什么 $w > 1$ 会外推”——**那是第 7 章补的**。

Saharia et al. (2022). Photorealistic Text-to-Image Diffusion Models with Deep Language Understanding (Imagen). 读什么：§3 的 CFG rescale、§4 的 T5 vs CLIP 对比。**读它能得到什么**：知道“图文对齐”不等于“适合做条件”。

Gandikota et al. (2023). Erasing Concepts from Diffusion Models. 读什么：负向引导的机制与副作用。**配第 7 章的负引导实验读**。

E.2.8 第 8 章 文本条件

Radford et al. (2021). Learning Transferable Visual Models From Natural Language Supervision (CLIP). 读什么：§2 的对比学习、§3 的 prompt engineering。**必读**：所有 CLIP 系文本编码器的源头。

Rombach et al. (2022). High-Resolution Image Synthesis with Latent Diffusion Models. 读什么：§3 的潜空间、§3.3 的交叉注意力。**必读**：Stable Diffusion 的论文。

Podell et al. (2023). SDXL: Improving Latent Diffusion Models for High-Resolution Image Synthesis. 读什么：§2 的两个文本编码器、§3 的 refiner。**配第 8 章“走一遍 SDXL”读**。

Esser et al. (2024). Scaling Rectified Flow Transformers for High-Resolution Image Synthesis (SD3/MMDiT). 读什么：§3 的 MMDiT、§4 的联合注意力。**必读**：第 8 章“条件注入谱系”的 ④。

Zhang, Rao, Agrawala (2023). Adding Conditional Control to Text-to-Image Diffusion Models (ControlNet). 读什么：§3 的零卷积。**配第 8 章“旁路注入”读**。

Ye et al. (2023). IP-Adapter: Text Compatible Image Prompt Adapter for Text-to-Image Diffusion Models. 读什么：§3 的解耦交叉注意力。

Li et al. (2023). GLIGEN: Open-Set Grounded Text-to-Image Generation. 读什么：§3 的 grounding token。**配第 8 章“空间条件”读**。

Chefer et al. (2023). Attend-and-Excite: Attention-Based Semantic Guidance for Text-to-Image Diffusion Models. Feng et al. (2023). Structured Diffusion Guidance. 读什么：两种“改注意力分配”的思路。**配第 8 章“属性泄漏”读**。

Gal et al. (2022). An Image is Worth One Word: Personalizing Text-to-Image Generation using Textual Inversion. Ruiz et al. (2023). DreamBooth: Fine Tuning Text-to-Image Diffusion Models for Subject-Driven Generation. 读什么：两种“教模型一个新词”的极端做法。**配第 8 章“教模型一个新词”读**。

Yuksekgonul et al. (2023). When and Why Vision-Language Models Behave Like Bags-of-Words. 读什么：第 8 章“CLIP 为什么不懂顺序”的实证来源。**必读**：如果你关心组合性。

Hertz et al. (2022). Prompt-to-Prompt Image Editing with Cross Attention Control. 读什么：用注意力图做编辑的可能与边界。

E.3 教材与综述

书/综述	适合谁	读法
Goodfellow et al., Deep Learning (2016)	想补基础	第 6–8 章（前馈、正则、优化）
Prince, Understanding Deep Learning (2023)	想要一本现代的	全书，尤其第 12、17 章

书/综述	适合谁	读法
Murphy, Probabilistic Machine Learning	想要概率视角	第 2、3、11 章
Yang et al., Diffusion Models: A Comprehensive Survey (2022)	想纵览扩散	读第 2–4 节，其余当索引

不要从头读教材。带着一个具体问题去读一章，比通读三章有用。

E.4 课程与博客

Karpathy, Neural Networks: Zero to Hero. 看什么：从零写 GPT 的那几集。**配第 1–3 章看。**他讲“为什么要缩放”的角度和本书不同，可以对照。

Harvard NLP, The Annotated Transformer. 看什么：逐行实现。**配第 1 章看。**注意它是 2018 年的版本，没有 RoPE、没有 Pre-LN。

Jay Alammar, The Illustrated Transformer. 看什么：图。**它的图比大多数论文清楚。跳过**：文字部分（比较浅）。

Lilian Weng, Lil' Log。看什么：What are Diffusion Models?（2021）、Controllable Neural Text Generation（2021）。**她的博客是“综述 + 公式”的最佳平衡。**

3Blue1Brown, Transformers 系列与 Diffusion 系列。看什么：直觉。**不要指望它给你推导。**

HuggingFace, Diffusion Models Course. 看什么：notebook。**和本书的写法互补：他们用 diffusers，本书从零写。**

E.5 开源项目

按“读代码的难度”排序，从易到难。

项目	是什么	读什么
`karpathy/nanoGPT`	300 行的 GPT	`model.py` 全文
`karpathy/minGPT`	更小的 GPT	`model.py`
`lucidrains/denoising-diffusion-pytorch`	极简 DDPM	`denoising_diffusion_pytorch.py` 的 `p_sample`
`crowsonkb/k-diffusion`	采样器大全	`k_diffusion/sampling.py`
`huggingface/diffusers`	工业级管线	`schedulers/` 和 `pipelines/`
`facebookresearch/DiT`	DiT 官方实现	`models.py` 的 `DiTBlock`
`Stability-AI/generative-models`	SD3 官方	`mmdit.py`
`black-forest-labs/flux`	FLUX 推理	`model.py` 的双流/单流块
`comfyanonymous/ComfyUI`	节点式工作流	`comfy/` 的采样节点
`AUTOMATIC1111/stable-diffusion-webui`	最流行的 UI	`modules/processing.py` 的 CFG 循环

> **i** 读代码的三个建议
>
> ① **先跑通，再读。一个跑不起来的仓库，读它等于读天书。**
> ② **从“最小的那个文件”开始。**`nanoGPT/model.py` 只有 300 行，**比 diffusers 的一个 pipeline 还短。**
> ③ **带着一个问题读。**“CFG 在哪里做的?”——找到那一行，比通读整个仓库有用。

E.6 数据集与基准

名字	是什么	注意
LAION-5B	58.5 亿图文对，网页抓取	**2023 年因内容问题被撤下**
CC12M	1200 万图文对	比 LAION 干净，但小
DataComp	数据筛选的基准	比的是“同样的算力，谁的数据好”
COCO	检测/描述的标准集	小，但标注质量高
T2I-CompBench	组合性基准	用 VQA 判属性/关系/数量
GenEval	组合性基准	用检测器判物体/颜色/位置
DrawBench	提示词集合	没有自动指标，靠人评
PartiPrompts	提示词集合	同上

用这些基准时，先看它们的判据是什么（第 8 章“真实模型怎么测”一节）。**一个用 VQA 打分的基准，会继承 VQA 的偏差。**

E.7 一份十二周阅读计划

每周 5–8 小时。

周	读什么	配哪一章
1	Attention Is All You Need	1
2	RoFormer + FlashAttention	1
3	Layer Normalization + Pre-LN 论文	2
4	GPT-2 + Chinchilla	3
5	Attention is not Explanation（两篇对照）	4
6	DDPM	5
7	Score-Based SDE + 附录 A	5
8	DDIM + DPM-Solver	6
9	EDM	6
10	CFG + Imagen	7
11	LDM + SDXL	8
12	SD3/MMDiT + ControlNet	8

每一周的最后一步：用本书的玩具模型复现那篇论文的一个数。复现不了，说明你没读懂——回去读第二遍。

E.8 一页论文笔记模板

读完第三遍，用这一页把论文压缩成你能复述的东西。

```
标题：
一句话问题：
它改了什么：        □数据 □结构 □目标 □采样 □训练

核心图：            第 __ 页 / 图 __
```

```
这张图证明了：
它没证明：

关键假设：
  1.
  2.

我能复现的那个数：
  数值：                正文第 __ 页
  我的结果：
  差异及原因：

它和本书哪一章最近：
  用到的机制：
  不同的地方：

一个可做的后续实验：
```

填不满的格子，就是你还没读懂的地方。

> 为什么是“一页”

因为超过一页，你会开始抄论文，而不是理解论文。
一页逼你做三件事：选出核心图、写出关键假设、找到一个可复现的数。
这三件事，正好对应“读懂了”的三个标准。

> 如果只有一周

读三篇：DDPM、DDIM、CFG。
它们是这本书下篇的三根柱子：DDPM 给了目标，DDIM 给了速度，CFG 给了控制。其余的都是在这三根柱子上加东西。