

第 0 周 · Day 5: 字符级语言模型实战 + Week 0 整合

时长 8 小时: §1 (形式化、压缩视角) 2h · §2-§4 (数据、模型、训练) 2h · §5-§6 (解码、评估) 1.5h · §A-§C 1h · §D (Week 0 整合与出关测试) 1.5h

配套代码 `day5_checks.py` (纯 numpy): 最大似然 $\Leftrightarrow$ 最小 KL、语言建模即压缩 (Huffman + 三个模型的 BPC)、困惑度换算、temperature、nucleus 的 off-by-one、权重绑定

§B 参考实现 (`code/` 目录): `d5_charlm.py`。先自己写, 再对照; 一键自检 `python code/run_selftests.py`, 说明见 `code/README.md`。

今天是 **Week 0** 的收官。前四天的每样东西都会在今天这一个脚本里用上。而且这份代码第 1 周会被改造成 **mini-GPT**: 把 LSTM 换成你手写的 Transformer decoder, 其余全部保留。今天写的不是练习, 是下周项目的地基。

§0 今天的主线

0.1 一个问题

怎么让模型学会"写字"?

§1 形式化	自回归分解: 把"生成一段文本"拆成"反复预测下一个字符" 训练目标为什么是交叉熵: 最大似然 $\Leftrightarrow$ 最小 KL $\Leftrightarrow$ 压缩
§2 数据	错位一格, 以及为什么 T 个位置能一次算完
§3 模型	Embedding $\rightarrow$ 序列模块 $\rightarrow$ 输出投影 (+ 权重绑定)
§4 训练	AdamW、warmup、裁剪 $\leftarrow$ Day 1/2 全部用上
§5 解码	贪心 / temperature / top-k / top-p $\leftarrow$ Lecture 3 的预习
§6 评估	困惑度与 BPC
§D 整合	一张全景图 + Week 0 出关测试

0.2 前四天的回收清单

Day 1 的 $\hat{y} - y$ 、交叉熵的由来、显存账; Day 2 的 AdamW、裁剪、LayerNorm、初始化; Day 3 的嵌入与分词; Day 4 的 LSTM、teacher forcing、API 坑。今天一个不落。§D.1 有一张完整的对照表。

§1 语言模型的形式化

1.1 自回归分解: 为什么是"从左到右预测下一个"

动机。 语言模型要建模整段文本的概率 $P(x_1, \dots, x_T)$ 。直接建模一个 $|V|^T$ 维的联合分布显然不可能。

推导。 概率的链式法则 (恒等式, 不是近似):

$$P(x_1, \dots, x_T) = \prod_{t=1}^T P(x_t | x_{<t})$$

"建模一个天文数字维度的联合分布"变成了"反复做一次 $|V|$ 类分类", 一个我们完全知道怎么做的问题 (Day 1 §5.2 的 softmax + 交叉熵)。

为什么从左到右? 链式法则对任何顺序都成立。选从左到右有两个理由: **和人类书写的顺序一致**, 生成时天然可用; **因果性只需要一个简单的掩码就能实现** (Day 4 §A.4), 任意顺序要复杂得多。

(BERT 走另一条路: 随机遮住一些位置让模型双向猜。它在**理解**任务上更强, 但不能直接生成, 因为没有有一个自然的生成顺序。Lecture 2 讲这个取舍。)

意义。 这个分解就是所有 GPT 系模型的全部训练目标。从今天这个几十万参数的字符级模型, 到几千亿参数的 LLM, 目标函数一模一样, 区别只在模型容量和数据规模。

1.2 训练目标从哪来: 最大似然 $\Leftrightarrow$ 最小 KL

动机。 Day 1 §2.2 从分类的角度推过"最大似然 $\Rightarrow$ 交叉熵"。在语言模型上再走一遍, 因为这里它有一个更清楚的含义。

推导。 语料是从某个真实分布 p_{data} 里采出来的。模型 q_θ 的平均对数似然 (取负) 是

$$\mathcal{L}(\theta) = -\mathbb{E}_{x \sim p_{\text{data}}} [\log q_\theta(x)] = \underbrace{H(p_{\text{data}})}_{\text{与 } \theta \text{ 无关}} + \text{KL}(p_{\text{data}} \parallel q_\theta)$$

(拆法和 Day 1 §2.2 完全一样。) 所以训练语言模型就是在**最小化模型分布与数据分布之间的 KL 散度**。KL 非负, 等于 0 当且仅当两个分布相同。

`day5_checks.py` 用梯度下降拟合一个 5 类分布 (梯度就是 Day 1 的 $q - p$), 300 步后 KL 降到 10^{-10} , q 和 p_{data} 完全重合。

意义与代价。 两个直接后果:

1. **模型能做的最好的事, 就是完全复制数据分布。** 训练损失有一个下限 $H(p_{\text{data}})$, 就是语言本身的熵, 永远降不到 0。
2. **KL 的方向是 $\text{KL}(p_{\text{data}} \parallel q)$** , 它严厉惩罚"数据里有、模型给低概率"的情形 ($p > 0, q \rightarrow 0$ 时这一项趋于无穷), 对"模型给了概率、数据里却没有"的情形则宽容得多。所以最大似然训练出的模型倾向于覆盖所有数据模式, 宁可把概率摊得宽一点。`day5_checks.py` 用单个高斯去拟合一个双峰分布 (峰在 ± 3): 最大似然给出 $\mu = 0, \sigma = 3.05$, 一个宽高斯盖住两个峰, **把大量概率放在两峰之间数据根本没有的地方**; 反方向的 $\text{KL}(q \parallel p)$ 则只锁定其中一个峰 ($\mu = -3, \sigma = 0.5$)。这是它生成时会冒出不合理内容的一个原因, 也是 §5 需要 top-k / top-p 把长尾切掉的原因之一。

1.3 困惑度: 定义、推导与直觉

训练损失是平均负对数似然 (nats/token):

$$\mathcal{L} = -\frac{1}{N} \sum_{t=1}^N \log P(x_t \mid x_{<t})$$

困惑度是它的指数:

$$\text{PPL} = \exp(\mathcal{L})$$

为什么要取指数？为了得到一个有直觉的单位。一个什么都没学、只会均匀猜的模型： $P = 1/V$ ，于是 $\mathcal{L} = \log V$ ， $\text{PPL} = V$ 。

$\text{PPL} =$ ”模型平均每一步在多少个候选之间犹豫”

这叫**有效分支数**。PPL 接近 V 说明模型什么都没学到； $\text{PPL} = 4$ 说明它把候选缩小到了 4 个左右。

字符级常用 **BPC** (bits per character)，换成以 2 为底：

$$\text{BPC} = \log_2 \text{PPL} = \mathcal{L} / \ln 2$$

PPL	nats/token	BPC
27.0	3.296	4.755
10.0	2.303	3.322
4.0	1.386	2.000
2.3	0.833	1.202

目标线要看语料和模型规模。大语料 (enwik8、text8) 上的大模型能做到 **BPC 1.0–1.3**；但今天用的 Tiny Shakespeare 只有约 1MB，小模型很快就过拟合，**验证集**上的合理水平是 **loss ≈ 1.5 nats**，即 **BPC ≈ 2.2** 。作为参照，nanoGPT 的字符级 Shakespeare 示例 (6 层 Transformer，约 10M 参数) 报告的验证 loss 约 1.47。训练集 loss 可以降到 1.0 左右，那是在背训练集。**今天的目标线是验证集 BPC < 2.3** 。

陷阱：PPL 只能在**同一个分词方案**下比较。字符级 PPL 和 BPE 的 PPL 完全不可比，后者每个 token 携带的信息多得多，PPL 自然更高。看到跨模型的 PPL 对比，第一个问题永远是“分词器一样吗”。§1.4 会给出解决办法。

1.4 语言建模就是压缩

动机。为什么评估语言模型用“对数似然”而不是别的指标？准确率、编辑距离都可以用，为什么偏偏是 $-\log P$ ？这一节给出最深的那个理由。

直觉。想象你要把一段文本通过电报发出去，按比特收费。聪明的做法是：**常见的符号用短码，罕见的用长码**（摩尔斯电码里 E 是一个点，Q 是四下）。那么每个符号该用多长的码？

推导（香农信源编码定理）。对一个概率为 p 的符号，最优的码长是

$$\ell = -\log_2 p \text{ 比特}$$

直觉上：概率 $1/2$ 的符号用 1 比特， $1/4$ 用 2 比特， $1/8$ 用 3 比特。整个分布的平均最优码长就是熵 $H = -\sum p \log_2 p$ ，任何无损编码都做不到比它更短。实际的 Huffman 编码能做到 $H \leq L < H + 1$ ；算术编码可以把整段文本编码到几乎恰好 $-\log_2 P(\text{整段})$ 比特。

把这个和语言模型放在一起：如果你用一个语言模型 q 来给每个字符分配概率，再用算术编码压缩，每个字符平均花的比特数是

$$-\frac{1}{N} \sum_t \log_2 q(x_t | x_{<t}) = \text{BPC}$$

这正是模型的交叉熵损失（换成以 2 为底）。所以：

语言模型的损失 = 用它压缩数据时每个符号的平均码长

验证。 `day5_checks.py` 在一段约 2000 字符的英文上做了两件事。

(a) 用训练半段的字符频率建 Huffman 码：平均码长 4.173 比特，熵 4.127 比特，满足 $H \leq L < H + 1$ 。

(b) 用三个越来越“懂”的模型，在没见过的测试半段上算 BPC：

模型	它知道什么	BPC	PPL
均匀	什么都不知道	4.858	29.0
unigram	每个字母多常见	4.113	17.3
bigram	前一个字母后面常跟什么	3.568	11.9

每多知道一点结构，每个字符就少花一点比特。

意义。

1. “训练 loss 下降”就是“压缩率提高”，也就是模型找到了数据里更多的规律。“学习即压缩”这句话在这里是字面意义上成立的。读 Lecture 4 的缩放律时记住这个图景：更低的 loss 总是意味着模型把数据描述得更短了。
2. 它解决了 §1.3 的跨分词器比较问题：不同分词器的 PPL 不可比，但把总比特数除以原始文本的字节数（bits per byte）就可比了，因为大家压缩的是同一份原始数据。很多 LLM 报告里用的就是这个指标。
3. 这也回答了开头的问题：对数似然之所以是“对的”评估指标，因为它有一个不依赖任何任务的物理含义，描述这份数据需要多少信息。

1.5 为什么用字符级起步

字符级在生产中不用（Day 3 §7.2：序列太长，注意力是平方复杂度）。但作为教学起点它无可替代：

- 词表极小（英文约 65 个字符），单卡几分钟就能训出效果
- 没有分词器这一层，数据管线透明到底，不会困惑“到底是模型的问题还是分词的问题”
- 能看见学习的层次：训练中模型会依次学会字符频率 → 拼出真词 → 空格和标点的位置 → 引号括号配对 → 长程结构。这正是 §1.4 表格里从 unigram 到 bigram 再往上走的那条路

训完一定要真的读一读生成的文本。在不同 loss 水平下看模型写出什么，比看任何曲线都更能建立直觉。

§2 数据管线

2.1 字符级词表

```

text  = open("input.txt", encoding="utf-8").read()
chars = sorted(set(text))          # sorted 保证可复现
V      = len(chars)
stoi  = {c: i for i, c in enumerate(chars)}
itos  = {i: c for c, i in stoi.items()}
encode = lambda s: [stoi[c] for c in s]
decode = lambda l: "".join(itos[i] for i in l)

```

`sorted` 不是可有可无：`set` 的迭代顺序在不同运行间可能不同，不排序的话 checkpoint 换次运行就对不上了 (§A.6)。

2.2 错位一格：样本怎么构造

语言模型训练最核心的技巧，简单到容易被忽略：

```
原文   : "hello world"
输入 x: "hello worl"    ← 去掉最后一个
目标 y: "ello world"    ← 去掉第一个
```

位置 t 的输入是 $x[t]$ ，要预测的是 $y[t] = x[t+1]$ 。一次错位，就把“预测下一个字符”变成了标准的逐位置分类任务。

2.3 为什么能一次算 T 个位置的损失

动机。 朴素的想法是：预测位置 1 跑一次前向，预测位置 2 再跑一次..... T 个位置跑 T 次。

直觉。 模型是**因果的**：位置 t 的输出只依赖 $x_{\leq t}$ （RNN 天然如此，Transformer 靠因果掩码保证）。所以一次前向就能同时得到全部 T 个位置的预测，互不泄漏：

$$\text{logits} \in \mathbb{R}^{B \times T \times V}, \quad \mathcal{L} = \frac{1}{BT} \sum_{b,t} \text{CE}(\text{logits}_{b,t}, y_{b,t})$$

这就是 Day 4 §4.3 的 **teacher forcing**，也是训练能高效的根本原因： T 个监督信号一次拿到。

代价也在 Day 4 §4.3 讲过：exposure bias。今天生成文本时你会亲眼看到，模型写错一个字符，后面可能整个跑偏。

2.4 分块采样

```
BLOCK, BATCH = 128, 64
def get_batch(data):
    ix = torch.randint(len(data) - BLOCK, (BATCH,))
    x = torch.stack([data[i : i+BLOCK] for i in ix])
    y = torch.stack([data[i + 1 : i+BLOCK+1] for i in ix])
    return x.to(device), y.to(device)
```

为什么随机起点？ 顺序切分会让样本边界固定，模型可能记住块边界；随机起点相当于数据增强，也让每个位置都有机会出现在上下文的不同位置。

上界的边界。 起点 i 合法的条件是 $y = \text{data}[i+1 : i+BLOCK+1]$ 取满 `BLOCK` 个，即 $i \leq \text{len}(\text{data}) - \text{BLOCK} - 1$ 。`torch.randint(high)` 是左闭右开，所以上界写 `len(data) - BLOCK` 恰好覆盖全部合法起点；网上常见的 `len(data) - BLOCK - 1` 也安全，只是少用了最后一个窗口。真正的坑在 Python 自带的 `random.randint(a, b)`，它是闭区间，照搬同一个上界就会多取一个起点，那个样本的 y 少一个字符，`torch.stack` 直接报形状错误。而且它只在极少数 batch 里出现，很难复现。（`code/d5_charlm.py --selftest` 第 2 项。）

§3 模型

3.1 三段式骨架

几乎所有语言模型都是这个骨架，从今天的 LSTM 到 GPT-4:

token ids (B, T)		
↓ Embedding	查表: 离散 id → 稠密向量	[Day 3 §1.2]
↓ (B, T, d)		
↓ 序列模块	让每个位置看到上文	[Day 4 → 第 1 周换成
Transformer]		
↓ (B, T, d)		
↓ LayerNorm	稳住尺度	[Day 2 §4.3]
↓ Linear → logits	投影回词表	[Day 1 §5.3]
↓ (B, T, V)		
↓ CrossEntropy	梯度就是 $\hat{y} - y$	[Day 1 §5.2]

第 1 周要做的事，就是把中间那一层从 **nn.LSTM** 换成你手写的 Transformer decoder，其余一行不动。

3.2 代码

```
class CharLM(nn.Module):
    def __init__(self, V, d=384, layers=2, dropout=0.2, tie=True):
        super().__init__()
        self.emb = nn.Embedding(V, d)
        self.core = nn.LSTM(d, d, layers, batch_first=True,
# Day 4 §A.1 坑 1
                                dropout=dropout if layers > 1 else 0.0)
# Day 4 §A.1 坑 3
        self.ln = nn.LayerNorm(d)
        self.head = nn.Linear(d, V)
        if tie:
            self.head.weight = self.emb.weight # 权重绑定, §3.3
        self.apply(self._init)
        for name, p in self.core.named_parameters(): # 遗忘门 bias = 1 (Day
4 §3.4、§A.1)
            if name.startswith("bias_ih"):
                n = p.size(0) // 4
                with torch.no_grad(): p[n:2*n].fill_(1.0)

    def _init(self, m): # Day 2 §5
        if isinstance(m, nn.Linear):
            nn.init.normal_(m.weight, std=0.02)
            if m.bias is not None: nn.init.zeros_(m.bias)
        elif isinstance(m, nn.Embedding):
            nn.init.normal_(m.weight, std=0.02)

    def forward(self, idx, hidden=None):
        x = self.emb(idx) # (B,T,d)
```

```

out, hidden = self.core(x, hidden)          # (B,T,d)
return self.head(self.ln(out)), hidden      # (B,T,V)

```

`hidden` 显式传进传出，生成时靠它做增量解码 (§5.1)，这也是第 1 周 KV Cache 的思想雏形。遗忘门只改 `bias_ih` 一份 (Day 4 §A.1: 两份 `bias` 是相加的)。

3.3 权重绑定

动机。 语言模型有两个 $V \times d$ 的大矩阵：输入 embedding 和输出投影。小模型上它们能占总参数的 30–50%。

直觉 (这才是关键)。 输入 embedding 回答“这个词是什么意思”；输出投影回答“什么样的隐状态该产出这个词”。这两件事本来是一体两面：如果“猫”的含义向量是 v ，那么隐状态接近 v 时就该输出“猫”。语义一致，为什么要用两套参数各学一遍？

用 Day 1 §5.3 的式子看得更清楚：输出层 logit 是 $z_i = w_i^\top h$ ，第 i 行 w_i 就是“词 i 的输出向量”；embedding 表的第 i 行是“词 i 的输入向量”。绑定就是规定这两个向量是同一个。

Press & Wolf (2017)、Inan et al. (2017) 独立提出了这个做法：不仅省参数，**困惑度还更好**。

机制。 由 Day 1 §7.1 (多路径梯度相加)，绑定后这个矩阵有**两条梯度路径**：输入端每次出现这个词时的梯度，和输出端每一步 softmax 的梯度（输出端对**所有**词都有梯度，因为 $\hat{y} - y$ 在每个位置都非零）。低频词作为输入很少出现，但作为输出端的“负类”每一步都在被更新，它的表示能从输出端“蹭”到大量梯度，缓解了数据稀疏。

收益 (day5_checks.py) :

V	d	绑定前	绑定后	省下
65	384	0.05M	0.02M	0.02M
50257	768	77.19M	38.60M	38.60M
152064	4096	1245.71M	622.85M	622.85M

约束与坑：

- 要求 `hidden_size == embedding_dim`，不等时要加投影或放弃绑定。
- `self.head.weight = self.emb.weight` 是让两者指向同一个 **Parameter** 对象，不是复制。
- `nn.Linear(d, V)` 的权重形状是 (V, d) ，`nn.Embedding(V, d)` 也是 (V, d) 。**形状恰好一致**，是 PyTorch 的布局约定 (Day 1 §6.1) 刚好对上。
- **大模型未必绑定**：参数量到几十亿后，embedding 占比变小，省下的参数不再关键，不少大模型会把两者分开以获得更多表达能力。

§4 训练

4.1 训练循环

```

model = CharLM(V).to(device)
opt = torch.optim.AdamW(model.parameters(), lr=3e-3, weight_decay=0.1,
betas=(0.9, 0.95))
sched = torch.optim.lr_scheduler.OneCycleLR(opt, max_lr=3e-3,
total_steps=STEPS, pct_start=0.05,
                                cycle_momentum=False) # 否则
 $\beta_1$  会被改写, 见下

for step in range(STEPS):
    x, y = get_batch(train_data)
    logits, _ = model(x) #
    (B,T,V)
    loss = F.cross_entropy(logits.reshape(-1, V), y.reshape(-1)) #
    §A.1
    opt.zero_grad(set_to_none=True)
    loss.backward()
    gnrm = nn.utils.clip_grad_norm_(model.parameters(), 1.0)
    opt.step(); sched.step()

```

每个超参的理由（不是抄来的配方）：

- $lr=3e-3$ ：小模型可以比 LLM 激进得多（LLM 通常 10^{-4} 量级）。参数少、地形没那么病态（Day 2 §1.3）。
- $betas=(0.9, 0.95)$ ： β_2 用 0.95 让二阶矩更"近视"，对梯度尖峰反应更快（Day 2 §6.3）。
- $weight_decay=0.1$ ：必须用 AdamW（Day 2 §2.7）。更讲究的做法是用 Day 2 §A.1 的参数分组把 bias 和 LayerNorm 排除掉。
- $pct_start=0.05$ ：前 5% 步数 warmup（Day 2 §6.1）。OneCycleLR 要求 STEPS 与实际步数精确一致（Day 2 §A.4）。
- OneCycleLR 的一个隐藏行为：它默认 $cycle_momentum=True$ ，会把 Adam 的 β_1 在 0.85–0.95 之间反向调度，覆盖你传给优化器的 $betas[0]$ ，而且不打任何提示。想保留自己设的 β_1 ，就显式写 $cycle_momentum=False$ （code/d5_charlm.py 就是这样写的，selftest 第 7 项验证了这个覆盖）。
- $clip=1.0$ ：把 $gnrm$ log 出来，梯度范数突然飙升是训练要炸的先兆（Day 4 §A.6）。

4.2 截断 BPTT（可选路线）

上面每次随机起点、hidden 从零开始，所以不需要截断 BPTT。如果想让模型跨块保持记忆（顺序遍历语料、hidden 一路传下去），就必须按 Day 4 §A.3 切断梯度：

```

hidden = None
for x, y in sequential_batches:
    logits, hidden = model(x, hidden)
    ...
    loss.backward()
    hidden = tuple(h.detach() for h in hidden)

```

4.3 该看到什么

阶段	loss (nats)	BPC	生成的文本
初始化	$\approx \ln V \approx 4.2$	≈ 6.0	纯随机字符
早期	~ 2.5	~ 3.6	字符频率对了 (§1.4 表里的 unigram 水平)
中期	~ 1.9	~ 2.7	拼得出真词，语法混乱
训好（验证集）	~ 1.5	~ 2.2	词和标点基本成形，会模仿剧本格式（人名、冒号、换行）

这张表是验证集上的大致水平（Tiny Shakespeare、2 层 LSTM），阈值是经验值，不是本仓库实测。训练集 loss 会继续降到 1.0 附近，和验证集拉开的差距就是过拟合，这时该加 dropout 或早停，而不是继续训。

如果初始 loss 明显不等于 $\ln V$ ，初始化有问题。未训练的模型应该对所有字符输出近似均匀的分布（所以 $\mathcal{L} = \ln V$ ，§1.3）。这是一个零成本的检查，每次搭新模型都该做。std=0.02 的初始化让 logits 都接近 0，softmax 接近均匀，正好满足这一点。

§5 解码策略（Lecture 3 的预习）

训练完，模型给你的的是一个概率分布。怎么从里面把字符一个个取出来，是一个独立的、影响巨大的问题。

5.1 自回归生成循环

```
@torch.no_grad()
def generate(model, prompt, max_new=500, temperature=1.0, top_k=None,
top_p=None):
    model.eval()
    idx = torch.tensor([encode(prompt)], device=device)
    logits, hidden = model(idx)
    out = encode(prompt)
    for _ in range(max_new):
        nxt = sample_next(logits[:, -1, :], temperature, top_k, top_p)
        out.append(nxt.item())
        logits, hidden = model(nxt, hidden)
    return decode(out)
```

增量解码

注意最后一行：每步只把新生成的那一个 token 喂进去，靠 hidden 携带全部历史。每步代价 $O(1)$ 。

Transformer 没有这个天然优势：它每步要对整个前缀重新算注意力。KV Cache 就是为了把这个代价追回来而发明的（Lecture 2）。今天这一行，是你理解 KV Cache 的锚点。

5.2 Temperature

$$P_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)}$$

直觉。 T 是“把分布拉平还是拉尖”的旋钮。由 Day 1 §2.4 的平移不变性，只有 logits 的差值有意义；除以 T 就是把所有差值放大 ($T < 1$) 或缩小 ($T > 1$)。Day 4 §5.3 的低温极限在这里是同一件事： $T \rightarrow 0$ 退化为 argmax。

`day5_checks.py` (6 个候选, 熵上界 $\ln 6 = 1.79$) :

T	最大概率	熵	行为
0.1	1.000	0.0005	接近贪心
0.5	0.860	0.491	保守
1.0	0.605	1.148	原始分布
3.0	0.303	1.697	接近均匀

边界: $T = 0$ 会除零。要么 `max(T, 1e-8)`, 要么 $T = 0$ 时直接走 `argmax`。

5.3 Top-k、Top-p, 以及那个要命的 off-by-one

动机。 即使 $T = 1$, 分布的长尾里也有成千上万个概率极低但非零的候选。§1.2 说过最大似然天然倾向于把概率摊宽。每一步都有几个百分点的概率采到一个荒谬的字符, 一旦采到, 后面就全跑偏 (exposure bias)。

top-k: 只保留概率最高的 k 个, 重新归一化。**缺点:** k 是固定的, 但分布的陡峭程度在变。有时前 2 个就占了 99% ($k = 50$ 太宽), 有时前 50 个才占 80% ($k = 50$ 又太严)。

top-p (nucleus) : 保留累积概率刚好达到 p 的最小集合。 k 自适应, 正是 top-k 缺的。

```
def sample_next(logits, temperature=1.0, top_k=None, top_p=None):
    logits = logits / max(temperature, 1e-8)
    if top_k is not None:
        v, _ = torch.topk(logits, min(top_k, logits.size(-1)))
        logits = logits.masked_fill(logits < v[:, [-1]], float("-inf"))
    if top_p is not None:
        sorted_logits, sorted_idx = torch.sort(logits, descending=True)
        cum = torch.cumsum(F.softmax(sorted_logits, dim=-1), dim=-1)
        remove = cum > top_p
        remove[..., 1:] = remove[..., :-1].clone() # ★ 右移一格
        remove[..., 0] = False # ★ 至少保住第一个
        logits = logits.masked_fill(remove.scatter(1, sorted_idx, remove),
float("-inf"))
    probs = F.softmax(logits, dim=-1)
    return torch.multinomial(probs, num_samples=1)
```

那两行带 ★ 的, 是整段代码最容易漏、后果最严重的地方。

`cum > top_p` 标记的是"累积和已经超过 p "的位置, 但正是那个让累积和跨过 p 的 token 该被保留 (否则保留下来的总概率不到 p)。所以要整体右移一格。

考虑一个高置信度的步骤: 最大概率 0.9998, `top_p=0.8`。不右移的话, `cum[0]=0.9998>0.8` 会把第 0 个也标记删除, 所有候选全被删光, `multinomial` 直接报错或采出垃圾。`day5_checks.py` 复现了这个边界:

```
高置信分布 [0.9998, 0.0001, 0.0001, 0.0], top_p = 0.8:
不右移 → [0. 0. 0. 0.] 总和 0.0 ← 全删光, 无法采样
```

右移 → [1. 0. 0. 0.] 总和 1.0

HuggingFace 源码里 `sorted_indices_to_remove[..., 1:] = sorted_indices_to_remove[..., :-1].clone()` 干的就是这件事。自己实现漏掉它，模型越自信越容易崩，而且只在特定输入上出现，极难复现。

回收 Day 4 §A.4 的伏笔：这里用 `-inf` 是安全的，因为右移保证了每行至少保留一个候选。那条规则的本质从来不是“别用 `-inf`”，而是“每行至少保住一个未被掩的元素”。

5.4 为什么 beam search 在开放生成上反而更差

Beam search 每步保留总对数概率最高的 B 条候选序列。它在翻译和摘要上是标准做法，明显好于贪心。

但在开放式生成（写故事、聊天）上，它的输出出了名的枯燥和重复。为什么？

因为“概率最高的序列”不等于“好的序列”。Holtzman et al. (2019) 观察到：人类文本的 **surprisal**（每个词的 $-\log P$ ）是起伏的，有可预测的词，也有出人意料的词，这种起伏正是“有信息量”的表现。用 §1.4 的话说，人类写的东西在模型看来不是最可压缩的。beam search 专找低 surprisal 的路径，产出异常平滑、异常可预测的文本，而且极易陷入重复循环（重复的内容 surprisal 最低）。

为什么翻译没这个问题？翻译是封闭式的：给定源句，好译文的分布是尖锐的，“概率最高”和“质量最好”高度重合。开放式生成则有成千上万条同样好的续写，分布是平坦的，这时最大化概率就是在做一件错误的事。

封闭式任务用搜索，开放式任务用采样。这条规则一路管到 LLM，Lecture 3 会再讲一遍。

5.5 重复惩罚

即使采样，模型仍可能陷入循环。常见补丁：**repetition penalty**（对出现过的 token 的 logit 除以一个大于 1 的系数）、**no_repeat_ngram_size**（禁止重复 n-gram）、**frequency / presence penalty**（按出现次数或是否出现过扣分）。

这些都是补丁而非解法。根本原因是训练目标（最大似然，§1.2）和生成目标（有趣、不重复）之间的错位。Lecture 5 的 RLHF 和 Lecture 6 的 RL，本质上是在用另一个目标函数修补这个错位。今天看到的这个小毛病，是后面两讲的起点。

§6 评估

```
@torch.no_grad()
def evaluate(model, data, iters=50):
    model.eval()
    losses = []
    for _ in range(iters):
        x, y = get_batch(data)
        logits, _ = model(x)
        losses.append(F.cross_entropy(logits.reshape(-1, V),
y.reshape(-1)).item())
    model.train() # ← 切回来
```

```
m = sum(losses) / len(losses)
return m, math.exp(m), m / math.log(2)           # loss, PPL, BPC
```

三个坑：

1. **reduction** 要一致。用 mean 时每个 batch 的 loss 已是 per-token 平均，直接平均各 batch 是对的。前提是每个 batch 的 token 数相同。有 padding 时必须用 sum 累加再除以总 token 数，否则短样本被高估（Day 1 §6.4）。
2. **padding** 位置必须排除：F.cross_entropy(..., ignore_index=PAD_ID)。字符级连续采样没有 padding，第 1 周做变长数据时一定会遇到。
3. **model.eval()** 之后要 **model.train()** 切回来（Day 1 §A.3 坑 2）。漏掉的话后续训练 dropout 就关了，很隐蔽。

§A 语言模型与解码：API 用法与边界

A.1 F.cross_entropy 的形状与类型

```
loss = F.cross_entropy(logits.reshape(-1, V), y.reshape(-1))
```

- 它要 (N, C) 和 (N,)，不接受 (B, T, V)（除非转成 (B, V, T)，那是为图像分割设计的布局）。语言模型一律压平成 (B*T, V)。
- 目标必须是 long，int32 会报错。
- 吃 logits 不吃概率（Day 1 §5.2.5）。
- ignore_index 默认 -100，设成 PAD id 就能自动跳过 padding。
- label_smoothing=0.1 可用。回忆 Day 1 §5.2.4：软标签仍满足 $\sum y = 1$ ，梯度仍是干净的 $\hat{y} - y$ 。

A.2 reshape 与 view

```
logits.view(-1, V)      # 要求内存连续，否则 RuntimeError
logits.reshape(-1, V)   # 不连续时自动复制，总能成功
```

transpose / permute 之后的张量不连续，view 会报 "view size is not compatible with input tensor's size and stride"。写模型时一律用 reshape 更省心，除非你想用 view 的报错来捕获意外的内存布局。

A.3 torch.multinomial 的边界

- 输入必须非负，不能有 NaN 或 inf。
- 所有元素为 0 时报错，这正是 §5.3 那个 off-by-one 的表现。
- num_samples 超过非零元素个数时要 replacement=True。
- 逐 token 生成时它是热点之一。

A.4 torch.topk 与 torch.sort

- `torch.topk(x, k)` 返回 `(values, indices)`, `k` 不能超过该维长度, 所以写 `min(top_k, logits.size(-1))`。
- `torch.sort(x, descending=True)` 返回 `(sorted, indices)`。要把排序后的掩码还原到原顺序, 用 `scatter` (§5.3 代码里那样)。直接拿 `indices` 去索引方向是反的, 这是个高频错误。

A.5 生成时的其他注意点

- `@torch.no_grad()` 或 `torch.inference_mode()` 必加, 否则整个生成过程建图, 显存随长度线性增长 (Day 1 §A.8)。
- `model.eval()` 必加, 否则 dropout 还在生效, 同一个 prompt 每次结果不同且更差。
- `.item()` 强制 GPU 同步 (Day 1 §A.13), 逐 token 循环里调用会明显拖慢。大量生成时把结果攒在 GPU 张量里, 最后一次性搬回 CPU。

A.6 保存与加载

```
torch.save({"model": model.state_dict(), "stoi": stoi, "itos": itos,
"config": {...}}, "ckpt.pt")
```

一定要把词表一起存。只存 `state_dict` 的话, 换次运行 `sorted(set(text))` 万一变了 (或换了语料), 加载后输出就是乱码, 而且不报任何错。

新版 PyTorch 的 `torch.load` 默认 `weights_only=True`, 加载含 Python 对象 (比如上面的 `stoi` 字典) 的 checkpoint 需要显式 `weights_only=False`, 并且只对可信来源这么做 (它会执行 pickle 里的代码)。

§B 实操

1. 跑通 `day5_checks.py`, 七组结论都看懂。第 1 组的 KL 方向和第 2 组的 "压缩" 实验最值得停下来想。
2. 从零写出字符级 LM (3h), 训到验证集 `BPC < 2.3` (§1.3 末尾解释了为什么不是 1.x)。语料可以用 Tiny Shakespeare (约 1MB)。每 500 步生成 200 个字符, 对照 §4.3 的表读一读, 看模型在不同阶段学会了什么。
3. 验证初始 $\text{loss} \approx \ln V$, 再故意把初始化 std 改成 1.0, 看初始 loss 变成多少、为什么。
4. 解码实验: 实现 temperature / top-k / top-p 三种采样, 包含 **nucleus** 的右移, 并用一个极端高置信分布验证不会崩。对同一个 prompt 扫 $T \in \{0.3, 0.7, 1.0, 1.5\}$ 各生成 200 字符, 写下你看到的差异。
5. 压缩视角的小实验: 用训好的模型在验证集上算出总比特数, 除以验证集原始字节数, 得到 bits per byte。和 `gzip` 压缩同一份文本的结果比一比 (`len(gzip.compress(text.encode())) * 8 / len(text.encode())`)。想一想这个比较公平吗: 模型的参数本身也是 "解码器" 的一部分。

参考实现: `code/d5_charlm.py`, 一个文件覆盖第 2–5 题: `--train`、`--init-check` (第 3 题)、`--sample / --sweep-temp` (第 4 题)、`--compress` (第 5 题)。`--core gru` 可以换掉中间那层, 第 1 周你会在这里接上自己的 Transformer。

全部参考实现和一键自检见 `code/README.md`。

§C 自测

动机与直觉层

1. 自回归分解为什么是恒等式而不是近似？为什么选从左到右？
2. 为什么说训练语言模型就是在最小化 KL？训练损失的下限是什么？
3. 最大似然对应哪个方向的 KL？这个方向为什么让模型"覆盖所有模式"？这和 top-p 有什么关系？
4. 为什么说语言建模就是压缩？"训练 loss 下降"在压缩的语言里是什么意思？
5. 困惑度为什么取指数？PPL=27 的含义是什么？跨分词器怎么比较？
6. 错位一格为什么能让 T 个位置一次算完？依赖模型的什么性质？
7. 权重绑定的直觉是什么？除了省参数，它为什么能帮到低频词？
8. 为什么 beam search 在翻译上好、在开放生成上差？用"分布尖锐 vs 平坦"和"surprisal"解释。

推导与实现层

9. 写出 PPL、BPC 和 nats/token 的换算关系。
10. 香农最优码长是什么？Huffman 码的平均码长满足什么不等式？
11. 未训练模型的初始 loss 应该是多少？为什么 std=0.02 的初始化能满足这一点？
12. 写出 nucleus 采样的完整实现。为什么要右移？不右移会在什么输入上崩、怎么崩？
13. `-inf` 掩码什么时候安全、什么时候危险？本质规则是什么？
14. RNN 的增量解码为什么是每步 $O(1)$ ？Transformer 为什么不是？KV Cache 在追什么？

API 层

15. `F.cross_entropy` 对形状和类型有什么要求？`ignore_index` 干什么用？
16. `view` 和 `reshape` 有什么区别？什么操作之后 `view` 会失败？
17. `torch.multinomial` 在哪些输入上会出错？
18. checkpoint 里为什么必须存词表？不存会报错吗？

§D Week 0 整合

D.1 全景图：今天这个脚本里的每一行来自哪里

今天的代码	来自	一句话
<code>F.cross_entropy(logits, y)</code>	Day 1 §2、§5.2	最大似然推出交叉熵；梯度是 $\hat{y} - y$ ；吃 logits
<code>loss.backward()</code>	Day 1 §1.2	雅可比连乘从左边开始乘
训练比生成吃显存	Day 1 §1.5、§9	反向要留住前向的中间激活
<code>zero_grad(set_to_none=True)</code>	Day 1 §7.3、§A.6	<code>.grad</code> 是累加语义
权重绑定的梯度	Day 1 §7.1	两条路径，梯度相加
<code>AdamW(betas=(0.9, 0.95))</code>	Day 2 §2.6–§2.7	带平滑的符号下降 + 解耦衰减
<code>OneCycleLR(pct_start=0.05)</code>	Day 2 §6.1	warmup 治 Adam 早期的高方差
<code>clip_grad_norm_(1.0)</code>	Day 1 §10.3	全局范数裁剪，保方向缩尺度
<code>nn.LayerNorm</code>	Day 2 §4.3	特征维归一化；雅可比是投影
<code>init.normal_(std=0.02)</code>	Day 2 §5	初始 loss 应 $\approx \ln V$

今天的代码	来自	一句话
<code>nn.Embedding</code>	Day 3 §1.2	分布式表示，解决 one-hot 的三宗罪
字符级 vs 子词	Day 3 §7	序列长度与语义粒度的取舍
<code>nn.LSTM</code> 、遗忘门 $\text{bias} = 1$	Day 4 §3.4、§A.1	每步雅可比 $\approx I + E_t$
<code>hidden</code> 增量解码	Day 4 §5.7	RNN 每步 $O(1)$ ，Transformer 靠 KV Cache 追
teacher forcing	Day 4 §4.3	训练/推理不一致 $\rightarrow$ exposure bias
<code>-inf</code> 掩码的安全条件	Day 4 §A.4	每行至少保住一个元素
loss 就是码长	本文 §1.4	学习即压缩

能把这张表讲一遍，Week 0 就通关了。

D.2 贯穿五天的四条主线

这五天不是五个独立主题，有四条线一直在走：

连乘。 Day 1 §7.2 指出梯度是矩阵连乘 $\rightarrow$ Day 2 §4 归一化控制连乘增益 $\rightarrow$ Day 2 §5 初始化设定连乘起点 $\rightarrow$ Day 4 §2.3 RNN 的连乘沿时间展开 $\rightarrow$ Day 4 §3.4 LSTM 用 $I + E_t$ 绕开连乘。**残差连接、LSTM 的 cell state、LayerNorm**，是同一个问题的三种解法。

尺子与尺度。 Day 2 §1.5 说每个优化器都在选一把尺子 $\rightarrow$ Day 2 §4.3 LayerNorm 的雅可比带着 $1/\sigma \rightarrow$ Day 2 §4.5 Pre-LN 的梯度按 $1/\sqrt{L}$ 缩放 $\rightarrow$ Day 4 §5.5 注意力分数要除以 $\sqrt{d}$ 。深度学习里的很多"魔法常数"，都是在把某个会随规模增长的尺度拉回 $O(1)$ 。

softmax 与归一化常数。 Day 1 §2.4 softmax 是 argmax 的软化 $\rightarrow$ Day 1 §5.2 它和交叉熵的消去 $\rightarrow$ Day 3 §3.3 负采样把分母消掉 $\rightarrow$ Day 4 §5.3 注意力用它做可微查表、§5.5 它的雅可比决定了 $\sqrt{d} \rightarrow$ Day 5 §5 解码时重新塑造它。同一个 softmax，在五个地方扮演五种角色。

训练与推理的不一致。 Day 1 §A.3 的 `model.eval()` $\rightarrow$ Day 4 §4.3 teacher forcing 与 exposure bias $\rightarrow$ Day 5 §1.2 最大似然的模式覆盖 $\rightarrow$ §5.4 beam search 失效 $\rightarrow$ §5.5 重复问题 $\rightarrow$ 最终指向 Lecture 5 的 RLHF 和 Lecture 6 的 RL。

D.3 Week 0 出关测试（2h，闭卷）

理论（60 分钟，纸笔，不许查任何东西）

1. 用"雅可比连乘的括号顺序"解释为什么训练用反向模式 AD，说出它的代价和三种偿还方式。
2. 从最大似然推出交叉熵；说明分类为什么不用 MSE（给出梯度表达式）。
3. 在白纸上推出两层 MLP 的全部梯度，索引级，包括 softmax 雅可比与交叉熵的消去。
4. 逐项算出 7B 模型混合精度 AdamW 全量微调的显存，说明 LoRA 省了哪三项、没省哪一项及原因。
5. 用"尺子"视角统一 SGD、牛顿法、Adam、符号下降；推出动量把 κ 降为 $\sqrt{\kappa}$ 的关键步骤。
6. 索引级推出 LayerNorm 的反向，说明它的雅可比为什么是投影；说明 Pre-LN 与 Post-LN 的梯度随深度如何变化。
7. 推出全 softmax 的梯度，讲清负采样替换了其中哪一项；完整推出 Levy-Goldberg 结论。

8. 论证静态词向量的天花板，说明它为什么导向注意力。
9. 推出 RNN 的 BPTT 连乘项；说明 LSTM 完整的单步雅可比，以及它真正缓解消失的机制。
10. 讲清 Q/K/V 在"软查表"里的角色和 key/value 分离的理由；完整推出 $\sqrt{d}$ 缩放。
11. 说明语言建模为什么等价于压缩，以及最大似然为什么会"覆盖所有模式"。

实战（60 分钟）

12. 从零写出今天的字符级 LM 并训到验证集 $BPC < 2.3$ 。
13. 实现 temperature / top-k / top-p，包含 nucleus 的右移，用极端分布验证不会崩。
14. 验证"未训练模型的初始 $\text{loss} \approx \ln V$ "。
15. 手写一个带掩码的缩放点积注意力，`assert` 每行权重和为 1。

通过线：理论 11 题至少答对 9 题（能讲清推导过程和动机，不是背结论），实战 4 项全做到。达不到的项，回对应的 Day 重读那一节。**Week 0 是地基**，这里省的力气第 1 周会加倍还回来。

D.4 进入第 1 周之前

把今天的 **CharLM** 保存好。第 1 周第一件事，就是把 `self.core` 从 `nn.LSTM` 换成你手写的 **Transformer decoder**，其余代码一行不改。到时候你会有一个干净的对照实验：同样的数据、训练循环、解码，只换序列模块，看 loss 曲线和生成质量的差别。

这个对照是整个课程里最有说服力的一次演示。你会亲眼看到 Transformer 到底强在哪，而且每一处"强"，你都能用这五天学的东西解释。

出关标准

- ☐ 能推出自回归分解，解释为什么从左到右
- ☐ 能推出"最大似然 = 最小 KL"，说出损失下限和 KL 方向的后果
- ☐ 能讲清"语言建模就是压缩"，并用它解释为什么评估用对数似然
- ☐ 能做 PPL / BPC / nats 换算，知道跨分词器该怎么比
- ☐ 能说清错位一格 + 因果性如何让 T 个位置一次算完
- ☐ 能讲出权重绑定的直觉、机制和约束
- ☐ 从零写出字符级 LM 并训到验证集 $BPC < 2.3$
- ☐ 实现三种采样策略，nucleus 的右移写对并验证过边界
- ☐ 能解释 beam search 在开放生成上失效的原因
- ☐ §A 的 API 坑能说出现象和正确写法
- ☐ 通过 §D.3 的 Week 0 出关测试

疑问清单

-
-
-