

第 0 周 · Day 1：张量与自动微分

时长 7-8 小时，建议拆成一天半：§1-§5（理论主干）约 4h，§6-§10（推广与工程后果）约 2.5h，§A-§C（API、实操、自测）约 1.5h。

配套代码（都只需要 numpy，装 PyTorch 之前就能跑）

- `day1_checks.py`：AD 的括号顺序与代价、交叉熵 = 熵 + KL、MSE 为什么在分类上失效、softmax 平移不变、路径求和
- `day1_numeric_check.py`：单样本两层网络的手推梯度 vs 数值微分
- `day1_batch_check.py`：batch 版公式 vs 数值微分，外加 mean/sum 对照
- `day1_backprop_verify.py`：手推 vs PyTorch autograd 对拍（这个需要 torch）

§B 参考实现（code/ 目录）：`d1_mlp_numpy.py`、`d1_train_mlp.py`。先自己写，再对照；一键自检 `python code/run_selftests.py`，说明见 `code/README.md`。

§0 今天的主线

0.1 一个问题

整天只回答一个问题：

给定一个标量损失 L 和 n 个参数（ n 可以是 7×10^9 ），怎么高效地求出全部 n 个偏导数？

0.2 论证链

节	回答什么	为什么排在这里
§1	为什么必须"反向"	先证明反向传播是 被迫 的选择，而不是一种风格。它的代价（必须存中间量）是后面所有显存优化的起点
§2	损失为什么是交叉熵	要求导先得有个损失。交叉熵不是随手选的，统计和优化两条理由都指向它
§3	求导的语言	推错的头号原因不是不会链式法则，而是搞不清 $\partial L / \partial W$ 是什么形状
§4	向量链式法则与 VJP	autograd 的本质。理解了它，框架就不再是黑盒
§5	完整手推	用一套 可复用的五步法 把两层网络推到底，索引级不跳步
§6	推广到 batch	真实代码永远是批量的；顺带得到"参数共享 $\Rightarrow$ 梯度求和"
§7	多路径	同一个变量被用了两次怎么办？这条规则解释了残差连接
§8	autograd 的实现	图里存了什么、为什么必须逆拓扑序
§9	换算成显存	7B 模型全量微调 112 GB、LoRA 14 GB，每个数都从前面的公式里来
§10	数值问题	log-sum-exp、fp16 的动态范围、为什么 bf16 赢了

节	回答什么	为什么排在这里
§A- §C	API、实操、自测	把上面的每一条落到代码上

0.3 怎么读这份文档

每个核心概念都按同样的四拍来写：

动机 —— 碰到了什么墙，不解决会怎样 **直觉** —— 如果是你，怎么会想到这样做 **推导** —— 索引级，不跳步 **意义与代价** —— 换来了什么，付出了什么，后面哪里还会用到

读的时候，**每读完"动机"就停下来，先自己想想"直觉"那一拍会是什么**，再往下看。想对了说明你真懂了，想错了，那个落差本身就是要学的东西。

0.4 和正课的关系

今天的每条结论在正课里都会被默认你已经会了。几个最直接的对应：

- **Lecture 4** 的显存账、梯度检查点、LoRA、混合精度 → 本文 §1.5、§9、§10.2
- **Lecture 1** 的残差连接 → 本文 §7.2
- **Lecture 1** 的注意力反向 → §5.8 的矩阵乘练习是它的核心零件
- 所有讲次的训练目标 → §2 的交叉熵

§1 为什么必须是"反向"

1.1 三个候选方案

我们有 $L : \mathbb{R}^n \rightarrow \mathbb{R}$ ，由一长串基本运算复合而成，要的是梯度 $\nabla L \in \mathbb{R}^n$ 。

方案一：数值微分。

$$\frac{\partial L}{\partial \theta_i} \approx \frac{L(\theta + \epsilon e_i) - L(\theta - \epsilon e_i)}{2\epsilon}$$

要得到全部 n 个分量，需要 $2n$ 次前向。 $n = 7 \times 10^9$ 时，求一次梯度要跑 140 亿次前向，**直接出局**。它还有一个绕不开的两难： ϵ 太大，截断误差大（差商不等于导数）； ϵ 太小，舍入误差大（两个相近的浮点数相减，有效位大量丢失）。中心差分的最优 ϵ 约在 10^{-6} ，精度上限约 10^{-10} 。

数值微分虽然不能用来训练，但它是**校验推导的黄金标准**：它直接来自导数的定义，不依赖任何框架。本周所有校验脚本用的都是它。

方案二：符号微分。 像 Mathematica 那样对解析表达式求导再化简。问题是**表达式膨胀**：每求一次导，表达式可能翻倍，几十层之后大到无法存储，而且丢掉了公共子表达式的复用。**出局**。

方案三：自动微分（AD）。 不求解析式，也不做差商，而是沿着计算图，把每个基本运算的局部导数数值地组装起来。结果精确到机器精度，不是近似。

问题是：怎么组装？下一节会看到，这个问题的答案其实是一道小学生都会的结合律题。

1.2 链式法则就是雅可比连乘：括号怎么加决定了代价

动机。 先把"前向""反向"这些词放一边，只看链式法则写出来到底是什么。

网络是一串复合： $x \rightarrow h_1 \rightarrow h_2 \rightarrow \cdots \rightarrow h_{L-1} \rightarrow L$ ，其中 $h_k = f_k(h_{k-1})$ 。记第 k 步的雅可比 $J_k = \partial h_k / \partial h_{k-1}$ 。链式法则说：

$$\frac{\partial L}{\partial x} = J_L J_{L-1} \cdots J_2 J_1$$

形状是： $J_1 \in \mathbb{R}^{h \times n}$ （从 n 维输入到 h 维隐层），中间的 $J_k \in \mathbb{R}^{h \times h}$ ，最后 $J_L \in \mathbb{R}^{1 \times h}$ （到标量损失）。

直觉。 矩阵乘法满足**结合律**：同一个乘积，括号可以随便加，**结果一样，代价不一样**。这正是算法课里"矩阵链乘法"那道动态规划例题。所以问题变成：这一串乘积，从哪边开始乘最便宜？

推导。 比较两种极端顺序。

从右往左： $J_L(\cdots(J_3(J_2 J_1)))$ 。第一步 $J_2 J_1$ 是 $(h \times h)(h \times n)$ ，花 $h^2 n$ 次乘加，结果仍是 $h \times n$ 。之后每一步都携带一个 $h \times n$ 的矩阵往前走。总代价

$$C_{\text{右}} \approx (L-2)h^2 n + hn$$

从左往右： $((J_L J_{L-1})J_{L-2}) \cdots J_1$ 。第一步 $(1 \times h)(h \times h)$ 只花 h^2 ，结果是 $1 \times h$ 的行向量。之后每一步都只携带一个行向量。总代价

$$C_{\text{左}} \approx (L-2)h^2 + hn$$

两者之比在 $h^2 n$ 项占主导时约为 n 。`day1_checks.py` 实测 ($n = 2000, h = 50, L = 4$)：

顺序	乘加次数	两种顺序的梯度差异
从右往左	10,100,000	—
从左往右	105,000	3.5×10^{-17} （结合律保证结果一致）

差 96 倍，而这还只是 $n = 2000$ 的玩具。 $n = 7 \times 10^9$ 时就是十亿倍量级。

意义与代价。 从左往右乘的时候，一路携带的那个行向量是什么？它是 $\partial L / \partial h_k$ ，损失对第 k 层的梯度。这正是反向传播里一层层往回传的"上游梯度"。所以：

反向传播 = 给雅可比连乘选了代价最低的括号顺序。

它不是一项独立的发明，而是三件事的直接推论：链式法则、结合律、**输出端只有 1 维**。

还有一个细节：从左往右的代价里剩下的 hn 那一项，是"把每个参数碰一遍"的代价。你要算 n 个偏导，至少得碰 n 个数，**这是任何方法都省不掉的下限**。反向模式已经贴着这个下限了。

1.3 前向模式与反向模式：同一件事的两种说法

AD 文献把上一节的两种顺序叫做两种"模式"。现在你知道它们本质上只是括号的区别了，再看它们的正式描述就很清楚。

前向模式沿计算方向走，每个节点上携带的是

$$\dot{h} \equiv \frac{\partial h}{\partial x_j} \quad (\text{对某一个固定的输入分量 } x_j)$$

从种子 $\dot{x} = e_j$ 出发一路推到 $\partial L / \partial x_j$ 。一次传播只能得到"对一个输入"的导数，要全部 n 个就要 n 次。这就是"从右往左乘"：每次用一个种子向量 e_j 去右乘。

反向模式逆计算方向走，每个节点上携带的是

$$\bar{h} \equiv \frac{\partial L}{\partial h} \quad (\text{对某一个固定的输出 } L)$$

从种子 $\bar{L} = 1$ 出发一路推回 $\partial L / \partial x$ 。一次传播得到"一个输出对所有输入"的导数。这就是"从左往右乘"。

	前向模式	反向模式
对应的括号顺序	从右往左	从左往右
种子在	输入端，要 n 个	输出端，要 m 个
得到全部偏导的代价	n 次传播	m 次传播
额外内存	$O(1)$ ，不用存中间量	必须存中间量 (§1.5)
适用	窄进宽出 ($n \ll m$)	宽进窄出 ($n \gg m$)

决策规则只有一句话：哪一头窄，就从哪一头出发。 训练时 $n \approx 10^9$ 、 $m = 1$ ，只能从输出端出发。

(前向模式并非无用。算雅可比-向量积 JVP、做高阶导、算 Hessian-向量积时它很好用，PyTorch 里有 `torch.func.jvp`。只是训练用不上。)

1.4 廉价梯度原理

一个值得记住的事实：**反向模式求完整梯度的计算量，只是一次前向的常数倍，与参数量 n 无关。**

实践中的经验数字是：反向约为前向的 **2 倍**，所以一个训练 step (前向 + 反向) 约等于 **3 次前向**。这个 3 倍后面会反复用到：估算训练时间要用它；理解梯度检查点为什么让训练慢 33% 也要用它 (多做一次前向， $3 \rightarrow 4$ ，恰好 +33%，§9.4)。

这件事其实相当反直觉：**求 70 亿个偏导数，和算一次损失，只差常数倍。** 没有这条原理，就没有深度学习。

1.5 代价：反向模式必须存中间量

回到 §1.3 那张表的第四行。为什么反向模式要额外内存？

因为反向走到第 k 层时，要算那一层的局部雅可比 J_k ，而 J_k 通常**依赖于前向时那一层的输入**。比如线性层 $z = Wa$ ，要算 $\partial L / \partial W$ 就必须知道 a ；ReLU 要算局部导数就必须知道输入的符号。这些量是前向时产生的，**得一直躺在显存里，等反向来取。**

前向模式没有这个问题：它和计算同方向走，算到哪一层局部导数就用到哪一层，用完即弃。

这笔"债"是后面一整条技术线的起点：

- 训练比推理吃显存得多：推理不需要留激活，前向完就能扔
- 激活显存随 batch size 和序列长度线性增长，长上下文训练所以才贵
- **梯度检查点**：只留少数几个存档点，其余扔掉，反向时重新前向算一遍，用时间换空间 (§9.4)
- **FlashAttention**：不把 $N \times N$ 的注意力矩阵写进显存，反向时重算。同一个思路的极致版本 (Lecture 2)

§2 损失从哪来：为什么是交叉熵

要求导，先得有一个损失。分类任务几乎永远用交叉熵，但为什么？这一节给两条独立的理由，一条来自统计，一条来自优化。两条都指向同一个答案，这不是巧合。

2.1 动机：分类器输出的是"概率"，怎么给概率打分？

分类器对输入 x 输出一个分布 $q_\theta(\cdot | x)$ ，真实标签是 y 。我们需要一个标量来衡量"这个分布预测得好不好"。候选有三个：

- **0-1 损失**（猜对得 0，猜错得 1）：这才是我们真正关心的，但它是阶梯函数，**梯度处处为 0**，没法用梯度下降优化。
- **均方误差**： $\sum_k (q_k - y_k)^2$ ，看起来很自然。
- **交叉熵**： $-\log q_y$ 。

下面分别从统计和优化两个角度论证，交叉熵是对的，而 MSE 是错的。

2.2 统计理由：最大似然直接推出交叉熵

直觉。 最朴素的原则是：一个好的模型，应该给真实发生的数据分配尽量高的概率。

推导。 设训练数据 $\{(x_i, y_i)\}_{i=1}^N$ 独立同分布。模型给整个数据集的概率（似然）是

$$\prod_{i=1}^N q_\theta(y_i | x_i)$$

要最大化它。连乘不好处理（而且 N 个小于 1 的数相乘会下溢），取对数变成连加；最大化变成最小化，取负号；再除以 N 使它和数据量无关：

$$\mathcal{L}(\theta) = -\frac{1}{N} \sum_{i=1}^N \log q_\theta(y_i | x_i)$$

这就是**负对数似然（NLL）**。把 y_i 写成 one-hot 向量， $-\log q_{y_i} = -\sum_k y_{i,k} \log q_k$ ，正好就是交叉熵。

再往前推一步：把经验平均换成数据分布下的期望，就得到了交叉熵的一般形式

$$\text{CE}(p, q) = -\sum_k p_k \log q_k$$

它可以拆成两项（一行推导）：

$$-\sum_k p_k \log q_k = \underbrace{-\sum_k p_k \log p_k}_{H(p)} + \underbrace{\sum_k p_k \log \frac{p_k}{q_k}}_{\text{KL}(p \| q)}$$

`day1_checks.py` 验证了这个分解（误差 10^{-16} ）。

意义。 数据的熵 $H(p)$ 与模型参数无关，所以

$$\text{最小化交叉熵} \iff \text{最小化 } \text{KL}(p_{\text{data}} \| q_\theta) \iff \text{最大化似然}$$

三件事是同一件事。交叉熵不是拍脑袋选的，它是"让模型分布逼近数据分布"这个目标的直接翻译。Day 5 会把同一个推导用到语言模型上，并看到它还等价于"压缩"。

2.3 优化理由：MSE 在"自信地错了"时会停止学习

动机。 就算不管统计，MSE 用在分类上也有一个致命的优化问题。用二分类看得最清楚。

推导。 模型输出 $s = \sigma(z)$ ，真实标签 $y = 1$ 。记住 $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ 。

交叉熵 $L = -\log s$ ：

$$\frac{\partial L}{\partial z} = -\frac{1}{s} \cdot s(1 - s) = -(1 - s) = s - 1$$

均方误差 $L = \frac{1}{2}(s - 1)^2$ ：

$$\frac{\partial L}{\partial z} = (s - 1) \cdot s(1 - s)$$

MSE 多了一个因子 $s(1 - s)$ 。 当模型**自信地错了** ($z \rightarrow -\infty, s \rightarrow 0$)，这个因子趋于 0，**梯度消失**。而交叉熵的梯度趋于 -1 ，满力度纠错。

`day1_checks.py` 实测（每个值都与数值微分对拍过）：

logit z	$\sigma(z)$	$\partial \text{CE} / \partial z$	$\partial \text{MSE} / \partial z$	CE / MSE
-8.0	0.00034	-0.99966	-0.000335	2983x
-4.0	0.01799	-0.98201	-0.017345	56.6x
-1.0	0.26894	-0.73106	-0.143735	5.1x
0.0	0.50000	-0.50000	-0.125000	4.0x

$z = -8$ 时模型以 99.97% 的信心押错，**MSE 给的纠错信号只有交叉熵的 1/3000**。这恰好反了：越离谱的错误本该学得越快，MSE 却让它学得最慢。

实测：比推导更微妙。 推导说的是**单个样本**：自信地错了时，MSE 给它的纠错信号几乎为零。放到整个训练里会发生什么？`code/d1_mlp_numpy.py` 用本节和 §5 手推的公式从零训练两层网络（digits 数据集，3 个种子平均，epoch 30 测试准确率）：

组合	正常初始化	输出层 ×30 (60% 样本自信地错了)
CE + SGD, lr=0.1	0.959	0.971
MSE + SGD, lr=0.1	0.519	0.892
MSE + SGD, lr=10	0.970	0.100 (发散)
MSE + SGD, lr=1	0.948	0.973
CE + Adam	0.969	—
MSE + Adam	0.969	—

读这张表能得到三件事，每一件都比"MSE 不好"这句结论更有用：

1. 同样学习率下 MSE 慢，主要是有效步长不对。初始化时 MSE 的整体梯度比 CE 小约 48 倍：
`F.mse_loss` 对类别取平均贡献 10 倍，softmax 雅可比 $\text{diag}(p) - pp^\top$ ($p \approx 0.1$ 时元素也在 0.1 量级) 再贡献几倍。把 SGD 的学习率放大 100 倍，MSE 就追上了。
2. Adam 把这件事掩盖了。Adam 的步长与梯度尺度无关 (Day 2 §2.6)，自动做了这个缩放，所以两者几乎一样。同一个坑，换个优化器就看不见了。
3. 饱和的真正代价是"没有一个学习率能一直合适"。CE 用同一个 $\text{lr}=0.1$ 在两种初始化下都好；MSE 的最佳学习率却跟着状态走：正常时 $\text{lr}=10$ 最好，饱和时 $\text{lr}=10$ 直接发散，要降到 1。因为 MSE 的梯度尺度取决于有多少样本处在饱和区，这个比例在训练中一直在变。

推导告诉你机制，实验告诉你机制在实际中有多大影响。两者都要看。

意义。交叉熵里的 $\log$ 正好抵消了 sigmoid 在饱和区的平坦。这是一个更一般的规律：输出激活和损失函数要配对。sigmoid 配二元交叉熵，softmax 配交叉熵，恒等输出配 MSE (回归)。配错了，饱和区就会杀死梯度。§5.2 会在多分类的情形下看到同样的抵消，而且形式更漂亮。

2.4 为什么是 softmax

动机。网络最后一层吐出的是任意实数向量 z (logits)，要变成一个概率分布。要求：非负、和为 1、单调 (z_i 越大 q_i 越大)、可微。

直觉。用 $\exp$ 保证非负且单调，再除以总和保证归一：

$$q_i = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

它是 $\arg \max$ 的软化： $\arg \max$ 把全部概率给最大的那一个，不可微；softmax 按指数比例分配，可微。把 logits 放大，它就越来越接近 $\arg \max$ (Day 5 的 temperature 就是这个旋钮)。这个"用 softmax 把离散选择软化成可微加权"的套路，Day 4 会在注意力里再见到一次。(统计物理里这个分布叫玻尔兹曼分布。)

推导：平移不变性。对任意常数 c ,

$$\frac{e^{z_i+c}}{\sum_j e^{z_j+c}} = \frac{e^c e^{z_i}}{e^c \sum_j e^{z_j}} = \frac{e^{z_i}}{\sum_j e^{z_j}}$$

意义。两个直接后果：

1. logits 的绝对值没有意义，只有差值有意义。
2. 可以放心地先减去 $\max_j z_j$ 再算指数，结果不变，却能防止 e^{1000} 上溢。这就是 §10.1 的 log-sum-exp 技巧的数学依据。

§3 求导的语言：标量对向量和矩阵求导

动机。推导出错的头号原因往往不是链式法则不熟，而是记号的形状没搞清。这一节把约定钉死。

3.1 梯度约定：梯度与被求导对象同形状

设 L 是标量， $W \in \mathbb{R}^{c \times h}$ 。定义

$$\left(\frac{\partial L}{\partial W} \right)_{ij} \equiv \frac{\partial L}{\partial W_{ij}}$$

也就是说 $\partial L / \partial W$ 是 $c \times h$ 的矩阵，和 W 同形状。同理， L 对 $x \in \mathbb{R}^n$ 的梯度是 $\mathbb{R}^n$ 的列向量。

这不是定理，是约定（文献里叫 denominator layout 或梯度约定）。选它有两个好处：

1. 参数更新 $\theta \leftarrow \theta - \eta g$ 要求 g 与 θ 同形状，用这个约定就不用到处转置。
2. 它白送一个验算手段 (§3.3)。

3.2 另一套约定：雅可比

对 $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$ ，雅可比定义为

$$J \in \mathbb{R}^{m \times n}, \quad J_{ij} = \frac{\partial f_i}{\partial x_j}$$

行标是输出，列标是输入。注意 $m = 1$ 时雅可比是行向量 $\mathbb{R}^{1 \times n}$ ，而 §3.1 的梯度是列向量，两者互为转置。§1.2 用的就是雅可比约定，所以那里出现的是“行向量从左往右乘”。

混用这两套约定，是手推反向传播出错的第一大来源。本文档统一：写成 $\partial L / \partial (\cdot)$ 的量都与分母同形状；雅可比只在明确写 J 的地方出现，采用“行输出、列输入”。

3.3 形状凑法：合法且高效的验算工具

推到某一步，知道结果里会出现 W 和 g ，但拿不准是 Wg 、 $W^\top g$ 还是外积？用形状凑。

目标形状是确定的（等于被求导对象的形状），可选组合通常只有一种能对上。例：已知 $g \in \mathbb{R}^c$ 、 $a \in \mathbb{R}^h$ ，要求 $\partial L / \partial W \in \mathbb{R}^{c \times h}$ ，能用这两个向量拼出 $c \times h$ 的只有外积 $ga^\top$ 。

这不是耍赖，Goodfellow 的《Deep Learning》和 CS231n 都正式教这个方法。但要清楚它的边界：形状对了不代表数学对了。它用来缩小候选、零成本复核；真正的正确性要靠 §5 的索引级推导，加上数值微分对拍。

§4 链式法则的向量形式与 VJP

4.1 从标量到向量

标量情形： $\frac{dL}{dx} = \frac{dL}{dy} \cdot \frac{dy}{dx}$ 。

现在设 $y = f(x)$ ， $x \in \mathbb{R}^n$ ， $y \in \mathbb{R}^m$ ， L 是标量。 x_j 可以通过 y 的每一个分量影响 L ，所以要把所有路径加起来：

$$\frac{\partial L}{\partial x_j} = \sum_{i=1}^m \frac{\partial L}{\partial y_i} \cdot \frac{\partial y_i}{\partial x_j}$$

这个求和是多元链式法则的全部内容，也是今天所有推导的唯一工具。写成矩阵形式：

$$\boxed{\frac{\partial L}{\partial x} = J^\top \frac{\partial L}{\partial y}} \quad J_{ij} = \frac{\partial y_i}{\partial x_j}$$

形状： $J^\top \in \mathbb{R}^{n \times m}$ 乘 $\mathbb{R}^m$ 得 $\mathbb{R}^n$ ，与 x 同形状。✓

（和 §1.2 对照：那里写的是行向量 $\bar{y}^\top J$ ，这里写的是列向量 $J^\top \bar{y}$ ，是同一个东西的转置。）

4.2 关键洞察：我们从不构造雅可比

动机。 上式出现了 J ，但 J 可能大得离谱。

直觉。 我们需要的从来不是 J 本身，而是 " $J^\top$ 乘以某个向量" 的结果。能不能只实现这个乘法，而不把 J 造出来？

这个操作叫 **VJP (vector-Jacobian product, 向量-雅可比积)**：给定 g ，算出 $J^\top g$ 。看三个例子就明白它为什么重要。

例一：线性层 $y = Wx$, $W \in \mathbb{R}^{1000 \times 1000}$ 。雅可比就是 W ， 10^6 个数。但 $J^\top g = W^\top g$ 只是一次矩阵-向量乘， W 本来就在显存里，**不需要额外存储**。

例二：逐元素激活 $y = f(x)$, $x \in \mathbb{R}^{1000}$ 。雅可比是 1000×1000 的对角阵， 10^6 个位置只有 1000 个非零。显式构造是浪费； $J^\top g = f'(x) \odot g$ 只要 $O(n)$ 。

例三：Transformer 的一个注意力层。 输入输出展平后维度动辄 10^6 ，雅可比是 10^{12} 个数，**物理上不可能存在**。但 VJP 完全可算。

意义。 PyTorch 的 autograd 就是一台 **VJP 引擎**。每个算子注册的不是 "我的雅可比是什么"，而是一个函数 `backward(g) -> J⊤g`。这也是为什么自定义算子要实现的是 `backward` 而不是 `jacobian`。§A.5 会看到，连 `backward()` 的 API 设计都直接体现了这一点。

4.3 对参数也一样

层 $y = f(x; \theta)$ 有两个 VJP：一个回传给输入（让梯度继续往前流），一个落到参数上（用来更新）：

$$\frac{\partial L}{\partial x} = J_x^\top g, \quad \frac{\partial L}{\partial \theta} = J_\theta^\top g$$

反向传播的每一层就做这两件事，仅此而已。

§5 完整手推

5.0 五步推导法（本节真正要教的东西）

动机。 手推的目的不是背下这一个网络的梯度，而是**遇到任何没见过的层都能推出来**。第 1 周你要推注意力的反向，第 2 周要看懂 LoRA 为什么有效，靠的都是同一套方法。

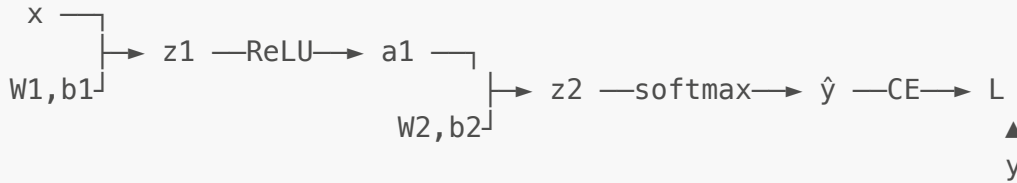
面对任何层，按这五步走：

Step 1 写出前向的**标量索引形式**，把所有矩阵乘法展开成显式求和。大多数人偷懒跳过这一步，然后推错。**Step 2** 明确目标：要求 $\partial L / \partial(\text{输入})$ 和 $\partial L / \partial(\text{参数})$ ，它们的**形状已知** (§3.1)。**Step 3** 对目标的**每一个标量分量**套用 §4.1 的链式法则，把求和写出来： $\frac{\partial L}{\partial x_j} = \sum_i g_i \frac{\partial y_i}{\partial x_j}$ 。**Step 4** 算局部导数 $\partial y_i / \partial x_j$ 。通常会冒出 **Kronecker delta** δ_{ij} ，它把求和塌缩掉，这是整个推导里最爽的一步。**Step 5** 把剩下的标量表达式**识别成矩阵运算**（转置、外积、逐元素乘），再用形状复核。

5.1 网络设定与计算图

$$\begin{aligned}
z_1 &= W_1 x + b_1 & W_1 &\in \mathbb{R}^{h \times d}, b_1 \in \mathbb{R}^h \\
a_1 &= \text{ReLU}(z_1) \\
z_2 &= W_2 a_1 + b_2 & W_2 &\in \mathbb{R}^{c \times h}, b_2 \in \mathbb{R}^c \\
\hat{y} &= \text{softmax}(z_2) \\
L &= - \sum_k y_k \log \hat{y}_k
\end{aligned}$$

这个网络小到能手算，但结构和 Transformer 的每个前馈块同构（线性 + 逐元素激活 + 线性）。



反向要依次得到 $\bar{z}_2 \rightarrow (\bar{W}_2, \bar{b}_2, \bar{a}_1) \rightarrow \bar{z}_1 \rightarrow (\bar{W}_1, \bar{b}_1)$ 。记号 $\bar{u} \equiv \partial L / \partial u$ 叫 **adjoint**，是 AD 文献的标准写法。

顺序必须是计算图的**逆拓扑序**：算 $\bar{a}_1$ 之前必须先有 $\bar{z}_2$ 。这是依赖关系的硬约束，§8.3 会看到 autograd 怎么保证它。

5.2 Step 1: softmax + 交叉熵

这是全场唯一需要动脑的一步。走**两条路**：先走长路（分别求两段再组合），亲眼看见消去；再给捷径。

5.2.1 长路之一：softmax 的雅可比

$\hat{y}_i = e^{z_i} / S$, $S = \sum_j e^{z_j}$ 。注意 S 依赖所有的 z_k ，很多人在这里漏掉 S 的导数。

零件： $\partial S / \partial z_k = e^{z_k}$ 。

情形 $i = k$ （商法则）：

$$\frac{\partial \hat{y}_i}{\partial z_i} = \frac{e^{z_i} S - e^{z_i} e^{z_i}}{S^2} = \hat{y}_i - \hat{y}_i^2 = \hat{y}_i(1 - \hat{y}_i)$$

情形 $i \neq k$ （分子与 z_k 无关，只有 S 贡献）：

$$\frac{\partial \hat{y}_i}{\partial z_k} = \frac{0 - e^{z_i} e^{z_k}}{S^2} = -\hat{y}_i \hat{y}_k$$

合并：

$$J_{ik}^{\text{sm}} = \hat{y}_i(\delta_{ik} - \hat{y}_k), \quad J^{\text{sm}} = \text{diag}(\hat{y}) - \hat{y}\hat{y}^\top$$

这是个 $c \times c$ 的满矩阵，而且对称。 $-\hat{y}\hat{y}^\top$ 那一项是“各输出互相竞争”的数学体现：推高一个 logit，必然压低其他的。

5.2.2 长路之二：交叉熵对 $\hat{y}$ 的梯度

$$\frac{\partial L}{\partial \hat{y}_i} = -\frac{y_i}{\hat{y}_i}$$

有个 $1/\hat{y}_i$, $\hat{y}_i \rightarrow 0$ 时会爆炸。记住这个隐患。

5.2.3 组合：看消去发生

用 VJP, $\bar{z} = (J^{\text{sm}})^{\top} \partial L / \partial \hat{y}$, J 对称所以转置可省：

$$\begin{aligned}\bar{z}_k &= \sum_i J_{ik} \cdot \left(-\frac{y_i}{\hat{y}_i}\right) = \sum_i \hat{y}_i (\delta_{ik} - \hat{y}_k) \left(-\frac{y_i}{\hat{y}_i}\right) \\ &= -\sum_i (\delta_{ik} - \hat{y}_k) y_i \quad \leftarrow \hat{y}_i \text{ 被约掉} \\ &= -y_k + \hat{y}_k \sum_i y_i = \hat{y}_k - y_k \quad \leftarrow \sum_i y_i = 1\end{aligned}$$

$$\frac{\partial L}{\partial z_2} = \hat{y} - y$$

盯着第二行看一秒。softmax 雅可比里的 $\hat{y}_i$ 和交叉熵梯度里的 $1/\hat{y}_i$ 正好抵消：一个满矩阵加一个会爆炸的量，组合之后只剩"预测减真实"。

这就是 §2.3 那个"输出激活与损失配对"规律在多分类上的样子：二分类里是 $\sigma(z) - y$ ，多分类里是 $\hat{y} - y$ ，线性回归配 MSE 也是 $\hat{y} - y$ 。在广义线性模型里这叫**典范链接函数**，这个干净的形式是必然的，不是巧合。

5.2.4 捷径与为什么还要走长路

捷径：

$$\log \hat{y}_k = z_k - \log \sum_j e^{z_j} \Rightarrow \frac{\partial \log \hat{y}_k}{\partial z_i} = \delta_{ki} - \hat{y}_i \Rightarrow \frac{\partial L}{\partial z_i} = -\sum_k y_k (\delta_{ki} - \hat{y}_i) = \hat{y}_i - y_i$$

三行搞定。那为什么还要走长路？三个实在的理由：

1. 只会捷径的人遇到变体就卡住。label smoothing、加权交叉熵、知识蒸馏的软标签，这些情形下 $\sum_i y_i = 1$ 是否还成立、结果变成什么，只有走过长路的人能当场判断。练习：若 $\sum_i y_i = s$ ，结论变成什么？（答案 $s\hat{y} - y$, [day1_checks.py](#) 第 6 组已验证，先自己推再去看）
2. softmax 的满雅可比后面会实打实用到。注意力里的 softmax 没有交叉熵来跟它消去，反向时那个 $\text{diag}(\hat{y}) - \hat{y}\hat{y}^{\top}$ 必须真的处理。Day 4 §5.5 还会用它解释为什么注意力要除以 $\sqrt{d}$ 。
3. 看见消去本身就是理解。

5.2.5 工程后果

F.cross_entropy 直接吃 logits，不要你先 softmax。它在内部走捷径，既省掉满雅可比，又绕开 $1/\hat{y}_i$ 和 $\log 0$ 。

自己写 `-(y * torch.log(torch.softmax(z, -1))).sum()` 数学上等价，数值上明显更差：某个 $\hat{y}_i$ 下溢到 0 时直接得到 `-inf` 和 `nan`。这是新手代码里很常见的坑，面试也爱问。

往后连一根线：语言模型里 z_2 是词表维度的 logits ($c \approx 5 \times 10^4$)， y 是下一个 token 的 one-hot。 $\hat{y} - y$ 就是"模型以为的下一个词的分布"减去"真实的下一个词"。所有 LLM 预训练的最后一层都是这个式子。

5.3 Step 2: 线性层对参数（索引级）

记 $g \equiv \bar{z}_2 = \hat{y} - y \in \mathbb{R}^c$ 。按五步法。

Step 1（索引形式）： $z_{2,k} = \sum_j (W_2)_{kj} a_{1,j} + b_{2,k}$

Step 2（目标与形状）： $\partial L / \partial W_2 \in \mathbb{R}^{c \times h}$

Step 3（链式法则，显式求和）：

$$\frac{\partial L}{\partial (W_2)_{il}} = \sum_k g_k \frac{\partial z_{2,k}}{\partial (W_2)_{il}}$$

Step 4（局部导数，delta 塌缩）： $(W_2)_{il}$ 只在 $k = i, j = l$ 那一项出现，所以 $\frac{\partial z_{2,k}}{\partial (W_2)_{il}} = \delta_{ki} a_{1,l}$ ，代回：

$$\frac{\partial L}{\partial (W_2)_{il}} = \sum_k g_k \delta_{ki} a_{1,l} = g_i a_{1,l}$$

Step 5（识别）： (i, l) 元素是 $g_i a_{1,l}$ ，正是外积：

$$\boxed{\frac{\partial L}{\partial W_2} = g a_1^\top} \quad (c \times h) \checkmark \quad \frac{\partial L}{\partial b_2} = g \quad (c) \checkmark$$

直觉。 $(W_2)_{il}$ 连接 a_1 的第 l 维和 z_2 的第 i 维。它该改多少 = （第 i 个输出错了多少） $\times$ （第 l 个输入当时有多活跃）。前向没参与的连接（ $a_{1,l} = 0$ ）梯度为 0，不背锅。

记住这句话：**梯度 = 上游的错 $\times$ 下游的活跃度**。整个深度学习的参数更新都是它的展开。

5.4 Step 3: 线性层对输入

目标 $\partial L / \partial a_1 \in \mathbb{R}^h$ 。局部导数 $\frac{\partial z_{2,k}}{\partial a_{1,j}} = (W_2)_{kj}$ ，于是

$$\frac{\partial L}{\partial a_{1,j}} = \sum_k g_k (W_2)_{kj} = (W_2^\top g)_j \quad \Rightarrow \quad \boxed{\frac{\partial L}{\partial a_1} = W_2^\top g}$$

注意这里没有 delta，求和没有塌缩。 对比 §5.3：参数只影响一个输出分量（delta 塌缩），输入影响所有输出分量（求和保留，要把 c 份责任按权重汇总回来）。这个对比就是线性层反向的本质。

通用规律：**前向用 W ，反向用 $W^\top$** 。Transformer 里每一个 `nn.Linear` 都是如此。

5.5 Step 4: 穿过 ReLU

逐元素函数的雅可比是对角阵，求和只剩一项：

$$\boxed{\frac{\partial L}{\partial z_1} = \frac{\partial L}{\partial a_1} \odot 1[z_1 > 0]}$$

直觉。 前向被压成 0 的神经元，反向也拿不到梯度。如果某个神经元对**所有**样本都输出 0，它永远不会被更新，**永久死亡**。这就是 dead ReLU，也是 GELU（Transformer 的标配，处处导数非零）存在的理由之一。

（ReLU 在 0 处不可导，实现里约定取 0。恰好落在 0 是零测事件，实践上无所谓。）

5.6 Step 5: 第一层参数

与 §5.3、§5.4 同构，记 $g_1 \equiv \bar{z}_1$ ：

$$\frac{\partial L}{\partial W_1} = g_1 x^\top (h \times d), \quad \frac{\partial L}{\partial b_1} = g_1 (h)$$

推完了。现在合上文档，在白纸上从 L 默写到 $\partial L / \partial W_1$ 。

5.7 汇总表

量	公式	形状	依赖的前向中间量
$\bar{z}_2$	$\hat{y} - y$	c	z_2
$\bar{W}_2$	$\bar{z}_2 a_1^\top$	$c \times h$	a_1
$\bar{b}_2$	$\bar{z}_2$	c	—
$\bar{a}_1$	$W_2^\top \bar{z}_2$	h	—
$\bar{z}_1$	$\bar{a}_1 \odot 1[z_1 > 0]$	h	z_1
$\bar{W}_1$	$\bar{z}_1 x^\top$	$h \times d$	x
$\bar{b}_1$	$\bar{z}_1$	h	—

最右列就是显存账单：反向必须留住 x, z_1, a_1, z_2 。这是 §1.5 那笔"债"的具体清单，§9 会把它换算成 GB。

`day1_numeric_check.py` 用中心差分逐项验证了这张表，误差 10^{-10} 量级。

5.8 两个模式拼出一切

整个推导只用了两条规则：

层类型	对输入的 VJP	对参数的 VJP
线性 $y = Wx + b$	$W^\top g$	$\partial W = gx^\top, \partial b = g$
逐元素 $y = f(x)$	$g \odot f'(x)$	无参数

这两条能拼出 MLP、CNN（卷积是特殊的线性层）、RNN，以及 Transformer 的大部分。剩下要单独推的只有 softmax（§5.2 推过了）、LayerNorm（Day 2 §4.3）、以及注意力里"两个乘数都是变量"的矩阵乘（下面练习 3）。

练习（用五步法，现在就做）：

- 逐元素乘 $y = x \odot w$ (w 可训练)。求 $\bar{x}, \bar{w}$ 。（提示：结果非常对称）
- 二次型 $L = x^\top A x$ 。求 $\partial L / \partial x$ 。（提示： x 出现了两次，要用 §7 的多路径规则。 A 不对称时答案是 $(A + A^\top)x$ ）
- 矩阵乘 $C = AB$ ， A, B 都是变量。已知 $\bar{C}$ ，求 $\bar{A}, \bar{B}$ 。答案是 $\bar{A} = \bar{C} B^\top$ 、 $\bar{B} = A^\top \bar{C}$ ，请自己推出来并用形状验证。这是注意力反向的核心零件，第 1 周直接用。

§6 从单样本到 batch

动机。 前面全是单样本。真实代码永远是批量的，这里有几个实打实的坑，还藏着一条贯穿全课的规律。

6.1 PyTorch 的形状约定

`nn.Linear(in, out)` 的权重形状是 $W \in \mathbb{R}^{\text{out} \times \text{in}}$ ，前向是

$$Z = XW^\top + b, \quad X \in \mathbb{R}^{B \times \text{in}}, Z \in \mathbb{R}^{B \times \text{out}}$$

样本按行排。那个转置常让人困惑，好处是 W 的行对应输出神经元，和单样本的 $z = Wx$ 保持一致。

6.2 batch 版的梯度

设 $G = \partial L / \partial Z \in \mathbb{R}^{B \times \text{out}}$ ，输入 $A \in \mathbb{R}^{B \times \text{in}}$ 。索引形式 $Z_{bi} = \sum_j A_{bj} W_{ij} + b_i$ ，于是

$$\frac{\partial L}{\partial W_{ij}} = \sum_b \sum_k G_{bk} \frac{\partial Z_{bk}}{\partial W_{ij}} = \sum_b G_{bi} A_{bj} = (G^\top A)_{ij}$$

$$\boxed{\frac{\partial L}{\partial W} = G^\top A, \quad \frac{\partial L}{\partial b} = \sum_b G_{b,:}, \quad \frac{\partial L}{\partial A} = GW}$$

batch 维在参数梯度里被求和掉，在输入梯度里被保留。 这是本节最重要的一句话。`day1_batch_check.py` 验证了全部公式。

6.3 为什么是求和：参数共享 $\Rightarrow$ 梯度求和

直觉。 看 $\sum_b$ 从哪来： W 被 batch 里所有样本共享，每个样本都拿同一个 W 做了一次前向。 B 个样本就是 B 条影响 L 的路径，§4.1 的链式法则要求把各路径贡献加起来。而 A 的第 b 行只被第 b 个样本用到，只有一条路径，所以 batch 维保留。

这条规律的射程远超 batch：

- **RNN 的 BPTT：** 同一个 W_{hh} 在 T 个时间步重复使用 $\rightarrow$ 梯度是 T 份贡献之和。Day 4 会看到这个求和正是 RNN 梯度消失的数学来源。
- **权重绑定：** embedding 和输出投影共用一个矩阵，梯度来自两处，相加（Day 5 §3.3）。
- **卷积：** 同一个卷积核在所有位置共享 $\rightarrow$ 梯度在空间维上求和。
- **梯度累积：** 多个 micro-batch 不清零，本质是人工实现这个求和（§7.3）。

凡是参数被共享的地方，梯度就在共享的那个维度上求和。 记住这一条，遇到任何新架构都不会推错。

6.4 mean 还是 sum：一个影响学习率的细节

损失通常写成 $L = \frac{1}{B} \sum_b L_b$ (`reduction='mean'`，默认)。这个 $1/B$ 会原封不动出现在所有梯度里。

改用 `reduction='sum'`，所有梯度放大 B 倍，等效于学习率乘了 B 。`day1_batch_check.py` 末尾的对照实验显示范数正好差 B 倍。常见现象“换了 batch size 训练就发散”，经常就是代码里混用了 mean 和 sum。

实用法则： 默认用 mean。改 batch size 时，学习率的经验缩放是 Adam 约按 $\sqrt{B}$ 、SGD 约按 B （Day 2 §1.6 解释为什么）。

§7 多路径：梯度为什么相加

7.1 规则

动机。 计算图不总是一条链。一个变量可能被好几个地方用到：残差连接里 x 既走恒等路径又走 $f(x)$ ，二次型里 x 出现两次。这时梯度怎么算？

推导。 最简单的例子 $L = f(x) + g(x)$ ，显然 $\frac{dL}{dx} = f'(x) + g'(x)$ 。推广：若 x 被 k 个下游节点 $u_1, \dots, u_k$ 消费，

$$\frac{\partial L}{\partial x} = \sum_{m=1}^k \left(\frac{\partial u_m}{\partial x} \right)^\top \frac{\partial L}{\partial u_m}$$

它其实就是 §4.1 那个求和：把 $(u_1, \dots, u_k)$ 拼成一个大向量 y ，求和自然出现。`day1_checks.py` 第 5 组对一个两路径函数做了验证。

在 `autograd` 里，这就是“一个张量的 `.grad` 从多个分支累加”。

7.2 这解释了残差连接

动机。 梯度从第 n 层回到第 1 层要经历

$$\frac{\partial L}{\partial x} = W_1^\top D_1 W_2^\top D_2 \cdots W_n^\top D_n g$$

(D_i 是激活的对角导数矩阵。) 这是一串**矩阵连乘**。平均增益小于 1 就指数衰减，大于 1 就指数爆炸。深网络训不动，根源就在这里。

直觉。 能不能给梯度开一条**不经过连乘**的路？如果 x 能直接“跳”到后面去，那条路上的梯度就不会被压缩。

推导。 残差块 $y = x + f(x)$ ， x 有两条路径：

$$\frac{\partial L}{\partial x} = \underbrace{\frac{\partial L}{\partial y}}_{\text{恒等路径}} + \underbrace{f'(x)^\top \frac{\partial L}{\partial y}}_{\text{残差路径}} = (I + f'(x)^\top) \frac{\partial L}{\partial y}$$

那个 I 就是恒等路径的贡献。不管 f 把梯度压得多厉害，总有一份**原封不动**的梯度直接传到 x 。堆 n 层之后，展开式里有一项是 n 个 I 相乘，也就是一条**梯度高速公路**。

意义。 这一条解释了一整串设计：

- **sigmoid 为什么被淘汰**： $\sigma' \leq 0.25$ ，每穿过一层至少衰减 4 倍，10 层后约 10^{-6} (`day1_numeric_check.py` 末尾会打印这个数)
- **残差连接**：Transformer 能堆到上百层的根本原因 (Lecture 1)
- **LSTM 的 cell state 是加法更新**：同样的高速公路，只是方向是时间 (Day 4 §3.2)
- **梯度裁剪**：对付爆炸那一侧 (§10.3)
- **LayerNorm**：控制每层输出的尺度，间接控制连乘增益 (Day 2 §4)

7.3 这也解释了 `.grad` 的累加语义

PyTorch 的 `backward()` 把梯度 `+=` 到 `.grad` 上，不是覆盖。新手常被坑（忘了清零，梯度越滚越大）。

但现在你知道：**累加不是设计失误，而是 §7.1 那条数学规则的直接实现**。框架无法预知一个张量会被多少条路径消费，唯一正确的语义就是累加，由用户决定何时清零。

它还白送一个功能，**梯度累积**：显存放不下 batch 64，就跑 4 个 batch 16，各自 `backward()` 但不清零，最后 `step()` 一次。由 §6.3，这在数学上**精确等价**于 batch 64（用 mean 时每个 micro-batch 的 loss 要除以 4）。Lecture 4 的标准手段。

§8 autograd 的实现机制

8.1 动态图

PyTorch 用**动态计算图**（define-by-run）：图在每次前向执行时**实时构建**，`backward()` 之后默认释放。

代价是每次都要重建图；好处是**控制流可以是任意 Python**：`if`、`for`、递归都行，图跟着实际执行路径走。这是 PyTorch 打败早期静态图框架的主要原因。

8.2 图里存了什么

每个非叶子张量带一个 `grad_fn`，指向创建它的运算节点。每个节点有：

- 一个 `backward` 方法，实现的就是 §4.2 的 **VJP**
- `next_functions`，指向输入各自的 `grad_fn`，构成图的边
- **前向时保存的中间量**（`saved_tensors`），也就是 §5.7 汇总表最右列那些

亲手看一眼：

```
import torch
x = torch.randn(3, requires_grad=True)
y = (x * 2).sum()
print(y.grad_fn)                                # <SumBackward0>
print(y.grad_fn.next_functions)                  # ((<MulBackward0>,
0),)
print(y.grad_fn.next_functions[0][0].next_functions)  #
((<AccumulateGrad>, 0), (None, 0))
```

叶子节点叫 `AccumulateGrad`，名字直接告诉你 §7.3 的累加语义。

8.3 为什么必须逆拓扑序

`backward()` 从标量出发，按**逆拓扑序**遍历图。为什么不能随便走？

因为一个节点的梯度只有在它**所有**下游消费者都回传完之后才是完整的（§7.1 要把各路径加起来）。如果提前把残差块里 x 的梯度往前传，就漏掉了另一条分支的贡献。所以 autograd 维护一个依赖计数，一个节点的所有出边都回传完毕，才轮到它。

§9 把推导换算成显存：真实数字

这一节是今天所有推导的变现。

9.1 训练一个 7B 模型要多少显存

标准混合精度 + AdamW:

项目	来源	字节/参数
bf16 权重	前向要用	2
bf16 梯度	§3.1 梯度与参数同形状	2
fp32 主权重副本	bf16 精度不够，小更新加上会被舍入掉 (§10.2)，必须在高精度副本上累加	4
fp32 Adam 一阶矩 m	与参数同形状 (Day 2 §2.6)	4
fp32 Adam 二阶矩 v	与参数同形状	4
合计		16

$$7 \times 10^9 \times 16 \text{ B} = 112 \text{ GB}$$

再加上激活值 (§5.7 最右列那些，随 batch × 序列长度增长，7B 模型典型几十 GB)。

所以单张 **80 GB 的 A100 放不下 7B 全量微调**，必须多卡切分 (ZeRO / FSDP) 或换方法。Lecture 4 "为什么需要 ZeRO" 的答案就在这张表里。

9.2 LoRA 省了什么，没省什么

LoRA 冻结基座，只训练低秩适配器 $\Delta W = BA$ 。基座参数**不需要梯度**，也不需要优化器状态：

项目	显存
基座 bf16 权重 (冻结)	$7\text{B} \times 2 = 14 \text{ GB}$
基座梯度	0
基座优化器状态	0
LoRA 参数 (约 0.3%, ~20M) × 16 B	~0.3 GB
小计	~14.3 GB

从 112 GB 降到 14 GB，**8 倍**，单张 24 GB 消费卡能跑。

但 **LoRA 不省激活值显存**。很多人搞错这一点，而它直接从今天的推导看得出来：梯度必须穿过整个冻结网络才能到达 LoRA 层 (§5.4 的 $W^\top g$ 一层层往回传)，所以前向的中间激活仍要全部保留。LoRA 省的是"参数相关"的三项，不是"数据相关"的那一项。

顺带回答"冻结基座为什么还能学到东西"：冻结 W 只是不更新它，不妨碍梯度流经它。 $\bar{a}_{l-1} = W^\top \bar{z}_l$ 这个回传照常发生。"冻结"影响的是要不要把 $\bar{W}$ 存下来并拿去更新，和"梯度能不能通过"是两件事。这是 **Lecture 4** 的核心理解点，也是面试高频题。

9.3 QLoRA

基座量化到 4-bit (NF4) : $7B \times 0.5 = 3.5$ GB, 加 LoRA 和激活, 16 GB 的卡就能微调 7B。"消费级显卡微调大模型"的全部原理就是这个。

9.4 梯度检查点

由 §1.5, 反向必须存中间激活。梯度检查点: L 层网络只保留 $\sqrt{L}$ 个存档点, 其余扔掉, 反向走到某段时重新前向算一遍。

- 激活显存: $O(L) \rightarrow O(\sqrt{L})$
- 时间代价: 多一次前向。由 §1.4 的 3 倍关系, $3 \rightarrow 4$, **+33%**

纯粹的时间换空间, §1.5 那笔债的标准还法。

9.5 小抄

方案	7B 参数相关显存	能跑的卡
全量微调 (混合精度 AdamW)	~112 GB	多卡
LoRA	~14 GB	单张 24 GB
QLoRA (4-bit)	~4 GB	单张 16 GB
纯推理 (bf16)	~14 GB + KV cache	单张 24 GB

推理为什么只要权重? 不需要梯度、不需要优化器状态、**激活算完就能扔**: §1.5 的另一面。

§10 数值问题

10.1 log-sum-exp 为什么要减最大值

$\log \sum_j e^{z_j}$ 直接算会溢出, $z_j = 1000$ 时 e^{z_j} 就是 **inf**。由 §2.4 的平移不变性, 对任意 c :

$$\log \sum_j e^{z_j} = c + \log \sum_j e^{z_j - c}$$

是恒等式, 不是近似。取 $c = \max_j z_j$, 最大项变成 $e^0 = 1$, 绝不上溢; 小项下溢成 0 也无所谓, 它们本来贡献就微乎其微。

10.2 fp16 的动态范围与 loss scaling

格式	指数位	尾数位	最小正规数	最大值
fp32	8	23	$\sim 1.2 \times 10^{-38}$	$\sim 3.4 \times 10^{38}$
fp16	5	10	$\sim 6.1 \times 10^{-5}$	65504
bf16	8	7	$\sim 1.2 \times 10^{-38}$	$\sim 3.4 \times 10^{38}$

fp16 的问题是范围太窄。训练中后期梯度常在 $10^{-6} \sim 10^{-8}$, **直接下溢成 0**, 参数不再更新。

解法 **loss scaling**: backward 前把 loss 乘一个大常数 S 。**梯度对 loss 是线性的**, 所以所有梯度同步放大 S 倍, 从下溢区被抬进可表示范围; step 前再除回去。数学上完全等价, 只是把数值挪了个位置。

bf16 用尾数位换指数位：范围和 fp32 一样宽，代价是精度只有 7 位尾数。对深度学习来说**范围远比精度重要**，所以 bf16 赢了，现代训练几乎全用它，也就不再需要 loss scaling（Day 2 §A.5 会看到这省掉了多少代码）。

精度低也有代价，就是 §9.1 表里那个 fp32 主权重副本：bf16 只有约 3 位十进制有效数字， $1.0 + 0.0001$ 在 bf16 里就等于 1.0，**小的更新会被舍入吃掉**，所以更新必须累加在 fp32 副本上。

10.3 梯度裁剪

对付 §7.2 连乘的爆炸侧。按**全局范数**裁剪：

$$g \leftarrow g \cdot \min \left(1, \frac{\tau}{\|g\|_2} \right)$$

是**所有参数拼成一个大向量**算范数，不是逐参数。这样能**保持梯度方向不变**，只缩尺度。Transformer 训练里 $\tau = 1.0$ 是事实标准。

§A PyTorch 与 autograd：API 用法与边界

这一节全是不看文档就会踩、而且大多不报错的坑。每一条都对应前面的某个原理，第 1 周手写 Transformer 时它们会原样再来一遍。

A.1 nn.Module 的自动注册

```
class MLP(nn.Module):
    def __init__(self, d=64, h=128, c=10):
        super().__init__() # 必须，否则注册机制没初始化
        self.fc1 = nn.Linear(d, h)
        self.fc2 = nn.Linear(h, c)
    def forward(self, x): # 只定义前向，反向由 autograd 给出
        return self.fc2(torch.relu(self.fc1(x)))
```

赋值给 `self` 的 `nn.Module` / `nn.Parameter` 会被**自动注册**，`model.parameters()`、`.to(device)`、`.state_dict()` 才找得到它们。

子模块塞进普通 Python list 会注册失败：那些参数不会被优化器看到，表现为“模型部分不学习”，**而且不报错**，极难排查。要用 `nn.ModuleList` 或 `nn.Sequential`。第 1 周堆 Transformer block 时一定会碰到。

A.2 训练循环的七步

```
for xb, yb in train_dl:
    xb, yb = xb.to(device), yb.to(device) # 1. 数据搬到设备
    optimizer.zero_grad() # 2. 清梯度 (§7.3 的累加语义)
    logits = model(xb) # 3. 前向：建图、存激活
    ($1.5)
    loss = criterion(logits, yb) # 4. 损失：吃 logits
    ($5.2.5)
    loss.backward() # 5. 反向：跑 §5.7 那张表
```

```
clip_grad_norm_(model.parameters(), 1.0) # 6. 裁剪 (§10.3)
optimizer.step()                        # 7. 更新
```

整个深度学习就是这七行的循环，而现在每一行你都能说出背后的数学。

A.3 五个高频坑

1. 忘了 **zero_grad()**：由 §7.3，梯度累加，loss 乱跳或发散。
2. 验证时忘了 **model.eval()**：Dropout 还在随机丢、BatchNorm 还在更新统计量，指标虚假。验证完记得 **model.train()** 切回来。
3. 验证时没加 **torch.no_grad()**：建了一整张用不到的图，存下全部激活，显存暴涨甚至 OOM。
4. **total += loss** 而不是 **total += loss.item()**：这个引用把整张计算图留住，显存随 step 线性泄漏。现象是"跑着跑着 OOM"，极其隐蔽。
5. 模型和数据不在同一 **device**：直接报错，算是友好的一种。

A.4 requires_grad 的边界

```
x = torch.randn(3, requires_grad=True) # 只有浮点/复数张量能求导
i = torch.tensor([1, 2], requires_grad=True) # x RuntimeError: 整型不能求导
```

- 只有浮点和复数张量能设 **requires_grad**。整型（包括 token id）不行。这也是为什么梯度流到 **nn.Embedding** 时，流的是权重表的梯度，不是 id 的梯度。
- **x.requires_grad_(True)** 是原地版本，只能作用在叶子张量上，对中间结果调用会报 *"you can only change requires_grad flags of leaf variables"*。
- 冻结子模块：for p in m.parameters(): p.requires_grad_(False)。LoRA 冻结基座就是这么做的 (§9.2)。它只阻止"把梯度累积到这些参数上"，不阻止"梯度流经它们"。

A.5 backward() 的三个参数

```
loss.backward() # loss 必须是标量
out.backward(gradient=v) # 非标量时必须给上游梯度 v
loss.backward(retain_graph=True) # 保留计算图，允许再次 backward
loss.backward(create_graph=True) # 为梯度本身建图，可求二阶导
```

- 默认只接受标量，对非标量调用会报 *"grad can be implicitly created only for scalar outputs"*。此时要传 **gradient=v**，而这个 **v** 正是 §4.2 里 VJP 的那个向量。API 的设计本身就在告诉你"autograd 是一台 VJP 引擎"。
- 图默认用完即弃，第二次 **backward()** 报 *"Trying to backward through the graph a second time"*。绝大多数时候出现这个错误是代码有 bug（比如循环间忘了 **detach** 隐状态，Day 4 §A.3），不要用 **retain_graph=True** 去掩盖它，那只会把问题变成更难查的显存泄漏。
- **create_graph=True** 用于二阶导（元学习、梯度惩罚），显存和时间都会大幅上升。

A.6 zero_grad(set_to_none=True)：PyTorch 2.x 默认值变了

```
opt.zero_grad()          # 2.x 默认等价于 set_to_none=True
opt.zero_grad(set_to_none=False) # 旧行为：填 0
```

	set_to_none=True (新默认)	set_to_none=False (旧)
p.grad	None	全零张量
显存	释放	一直占着
优化器	跳过 grad is None 的参数	对零梯度也执行更新

第三行是真正的语义差异：带动量的优化器在“零梯度”时仍会用历史动量更新参数，在 `grad is None` 时则完全跳过。冻结部分参数或做梯度累积时，两者结果不同。

老代码里的 `p.grad.zero_()`、`p.grad.norm()` 在新默认下会 `AttributeError: 'NoneType' object has no attribute ...`，加 `if p.grad is not None:` 即可。

A.7 detach() 与 .data：后者永远不要用

```
y = x.detach()      # ✓ 切断图，保留版本计数器的保护
y = x.data           # ✗ 绕过 autograd 的所有安全检查
```

两者都返回不带梯度的张量，但 `.data` 绕过了版本计数器。如果你通过 `.data` 原地修改了一个反向还要用的张量，autograd 不会报错，而是安静地算出错误的梯度。`detach()` 则会在出问题时报 *“one of the variables needed for gradient computation has been modified by an inplace operation”*。

`.data` 是 PyTorch 0.3 时代的遗物，现代代码里出现它基本都是 bug 或抄来的旧代码。

A.8 no_grad 与 inference_mode

```
with torch.no_grad():      # 不建图
    ...
with torch.inference_mode(): # 更激进：连版本计数器都不维护，更快更省
    ...
```

`inference_mode` 有一个硬限制：在它里面创建的张量被打上标记，之后不能参与任何 autograd 计算，否则报 *“Inference tensors cannot be saved for backward”*。

判断规则：纯推理服务用 `inference_mode`；验证循环用 `no_grad`。验证产生的张量有时会被拿去做别的（记录、拼接、再算点什么），用 `inference_mode` 容易在几十行之外炸出莫名其妙的错误。

`detach()` 和 `no_grad()` 的区别：前者作用于一个张量（在图上剪一刀，前面的图还在），后者作用于一段代码（里面的运算根本不建节点，既省时间又省显存）。

A.9 原地操作与 autograd

```
x = torch.randn(3, requires_grad=True)
y = x.exp()
y.add_(1)          # x exp 的反向需要它自己的输出 y，而 y 被改了
```

报错：

```
RuntimeError: one of the variables needed for gradient computation has been modified by an
inplace operation: [torch.FloatTensor [3]], which is output 0 of ExpBackward0, is at version 1;
expected version 0 instead.
```

怎么读：某个算子的反向要用到某个张量（这里 $\frac{d}{dx}e^x = e^x$ 用的就是输出 y ），而那个张量被原地改过了。解法是把那处原地操作改成非原地（ $y = y + 1$ ）。

哪些原地操作安全，取决于该算子的反向是否保存了那个张量（§8.2 的 `saved_tensors`）。

`ReLU(inplace=True)` 安全（反向只需要符号，可从输出恢复）。**不确定时就别用 `inplace`**，省下的那点显存不值得。

A.10 `torch.autograd.grad()`：要梯度但不想累积

```
g = torch.autograd.grad(loss, params, create_graph=True)  # 返回元组，不写
.grad
```

与 `backward()` 的区别：**返回梯度，而不是累加到 `.grad`**。适用于梯度惩罚、元学习、或者“我只想看看梯度是多少”。不会污染 `.grad`，也就不需要额外清零。

A.11 用 `gradcheck` 验证自定义算子

第 1 周如果你写了 `torch.autograd.Function`，用它验证：

```
from torch.autograd import gradcheck
x = torch.randn(4, 5, dtype=torch.double, requires_grad=True)  # ★ 必须
double
assert gradcheck(MyOp.apply, (x,), eps=1e-6, atol=1e-4)
```

必须用 `float64`。`float32` 的精度不够让数值微分和解析梯度对上，会得到大量假失败。原理和本周的校验脚本完全一样（§1.1 的数值微分），只是 PyTorch 帮你封装了。

A.12 张量创建的几个坑

```
t2 = torch.tensor(t1)          # ⚠ UserWarning: 建议用
clone().detach()              # ✓ 明确表达意图
t2 = t1.clone().detach()      # 与 numpy 共享内存，零拷贝
t3 = torch.as_tensor(np_array) # 同上，只接受 numpy
t4 = torch.from_numpy(np_array)
```

- 对已有张量调 `torch.tensor()` 会警告：它复制数据又切断梯度，意图不明确。
- `torch.tensor([1, 2])` 是 `int64`，`torch.tensor([1., 2.])` 是 `float32`。**token id 必须是 long，特征必须是 float**，弄错会在 `nn.Embedding` 或矩阵乘那里报类型错误。
- `from_numpy / as_tensor` 共享内存：改了 `numpy` 数组张量也变。要独立副本得 `.clone()`。

A.13 `.item()` 与 GPU 同步

`.item()`、`.cpu()`、`.numpy()`、`print(tensor)` 都会强制 CPU 等 GPU 算完（CUDA 是异步的）。

- 性能：热循环里每步 `.item()` 会打断流水线。日志每 N 步取一次就好。
- 计时：不同步就计时会得到荒谬的结果。做性能测量先 `torch.cuda.synchronize()`。

而累计 loss 时（A.3 坑 4）`.item()` 又是必须的。两个需求方向相反，所以正确做法是：累计用 `.item()`，但不要每步都打印。

A.14 device 写法

```
device = ("cuda" if torch.cuda.is_available()
          else "mps" if torch.backends.mps.is_available()
          else "cpu")
```

Mac 上 `mps` 能跑小实验，真训练上 Colab / Kaggle 的 `cuda`。从一开始就写与设备无关的代码，切环境零成本。

§B 实操

1. 跑通四个校验脚本：`day1_checks.py`、`day1_numeric_check.py`、`day1_batch_check.py`，装好 `torch` 后再跑 `day1_backprop_verify.py`。先在纸上推完 §5 再跑，否则脚本只是一段没用的输出。
2. §5.8 三道练习，用五步法推完，再对照 `day1_checks.py` 第 7 组。
3. 亲眼看 MSE 的失效：先跑 `code/d1_mlp_numpy.py --compare` 和 `--saturate`（纯 `numpy`，用的就是 §5 手推的公式），读懂 §2.3 末尾那张表；再用 PyTorch 版 `code/d1_train_mlp.py --compare-loss` 复现一遍。
4. 制造 A.3 的坑 1 和坑 4：注释掉 `zero_grad()` 看 loss 曲线；把 `.item()` 去掉，用 `torch.cuda.memory_allocated()` 看显存随 step 增长。参考：`code/d1_train_mlp.py --bug no_zero_grad / leak_graph / no_eval`，它同时会打印计算图节点数随 step 的增长，没有 GPU 也能看到。

全部参考实现和一键自检见 `code/README.md`。

§C 自测

先合上文档。答不出的回对应小节。

动机与直觉层

1. 用“矩阵连乘的括号顺序”解释为什么训练用反向模式。什么情况下前向模式更好？

2. 反向模式多付出的代价是什么？现代训练用哪些技术偿还？（至少两个）
3. 交叉熵从哪来？写出从最大似然到交叉熵的推导，并说明它和 KL 的关系。
4. 分类为什么不用 MSE？给出梯度表达式，并说明"自信地错了"时会发生什么。
5. 为什么 softmax 是 argmax 的"软化"？平移不变性有哪两个后果？
6. 什么是 VJP？为什么 autograd 实现 VJP 而不是算雅可比？

推导层

7. 完整推出 softmax 的雅可比，展示它和交叉熵组合时 $\hat{y}_i$ 如何消去。
8. 若标签是软标签且 $\sum_i y_i = s$, $\partial L / \partial z$ 是什么？ $s = 1$ 时呢？
9. 用五步法推出 $C = AB$ 的反向，用形状验证。
10. batch 情形下，为什么参数梯度对 batch 维求和、输入梯度保留 batch 维？
11. 用多路径规则推出残差块的梯度，解释那个 I 的意义。

工程层

12. 7B 模型混合精度 AdamW 全量微调，逐项说出 16 字节/参数怎么来。
13. LoRA 没有降低哪部分显存？为什么？"冻结"和"不回传"有什么区别？
14. 梯度检查点为什么是 +33% 而不是 +100%？
15. bf16 为什么打败 fp16？为什么还需要 fp32 主权重副本？

API 层

16. `backward()` 什么时候必须传 `gradient`？那个参数在数学上是什么？
17. `zero_grad(set_to_none=True)` 和填 0 在带动量的优化器上有什么不同？
18. `.data` 为什么危险？`detach()` 好在哪？
19. `inference_mode` 和 `no_grad` 各该用在哪？

编程层

20. 不查文档，从零写一个完整脚本，在 `sklearn.datasets.load_digits` 上训两层 MLP，测试准确率 > 95%，每 5 个 epoch 打印 train loss 和 test acc。写完之前不要看附录。

附录：参考实现（先自己写完再看）

```
import torch, torch.nn as nn
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split
from torch.utils.data import TensorDataset, DataLoader

device = ("cuda" if torch.cuda.is_available()
          else "mps" if torch.backends.mps.is_available() else "cpu")
torch.manual_seed(0)

X, y = load_digits(return_X_y=True)
X = (X / 16.0).astype("float32") # 8x8 灰度，归一化
Xtr, Xte, ytr, yte = train_test_split(X, y, test_size=0.2,
                                      random_state=0, stratify=y)
```

```

def make_loader(X, y, bs, shuffle):
    ds = TensorDataset(torch.tensor(X), torch.tensor(y))    # y 是 int64
    (A.12)
    return DataLoader(ds, batch_size=bs, shuffle=shuffle)

train_dl = make_loader(Xtr, ytr, 64, True)
test_dl = make_loader(Xte, yte, 256, False)

class MLP(nn.Module):
    def __init__(self, d=64, h=128, c=10):
        super().__init__()
        self.net = nn.Sequential(                                # Sequential, 不用 list
            nn.Linear(d, h), nn.ReLU(), nn.Dropout(0.1), nn.Linear(h, c)
        )
    def forward(self, x):
        return self.net(x)

model = MLP().to(device)
opt = torch.optim.AdamW(model.parameters(), lr=1e-3, weight_decay=1e-2)
criterion = nn.CrossEntropyLoss()                            # 吃 logits, 不要先
softmax (§5.2.5)

@torch.no_grad()                                             # 坑 3
def evaluate():
    model.eval()                                             # 坑 2
    correct = total = 0
    for xb, yb in test_dl:
        xb, yb = xb.to(device), yb.to(device)
        correct += (model(xb).argmax(1) == yb).sum().item()
        total += yb.numel()
    model.train()
    return correct / total

for epoch in range(30):
    running = 0.0
    for xb, yb in train_dl:
        xb, yb = xb.to(device), yb.to(device)                # 1
        opt.zero_grad()                                       # 2 ← 坑 1
        loss = criterion(model(xb), yb)                       # 3 + 4
        loss.backward()                                       # 5
        nn.utils.clip_grad_norm_(model.parameters(), 1.0)    # 6
        opt.step()                                            # 7
        running += loss.item() * yb.size(0)                  # ← 坑
    4: .item()
    if (epoch + 1) % 5 == 0:
        print(f"epoch {epoch+1:2d}  loss {running/len(ytr):.4f}  acc
{evaluate():.4f}")

```

30 个 epoch 约 97–98%。明显更差就按 A.3 逐个排查。

出关标准

- ☐ 能用"雅可比连乘的括号顺序"解释反向模式，并说出它的代价
- ☐ 能从最大似然推出交叉熵，写出 $CE = H + KL$
- ☐ 能推出 MSE 在分类上的梯度，说明"自信地错了"时为什么学不动
- ☐ 能解释 VJP，以及为什么 autograd 不构造雅可比
- ☐ 能在白纸上用五步法从 L 推到 $\partial L / \partial W_1$ ，索引级不跳步
- ☐ 能推出 softmax 雅可比并展示与交叉熵的消去
- ☐ 能推出 batch 版公式，并解释"参数共享 $\Rightarrow$ 梯度求和"
- ☐ 能用多路径规则说明残差连接的梯度高速公路
- ☐ 能逐项算出 7B 全量微调的 112 GB，说清 LoRA 省了哪三项、没省哪一项
- ☐ 能解释 bf16 为什么打败 fp16，以及为什么还要 fp32 主副本
- ☐ §A 的 API 边界能说出现象和正确写法
- ☐ 不查文档写出完整训练脚本并跑到 95%+

全部打勾再进 Day 2。打不了勾的，把具体卡点发给我。

疑问清单

(看不懂的写在这里，不要停下来死磕，晚上集中处理)

-
-
-