AI FOR BEGINNERS · SELF-STUDY EDITION

从零读懂人工智能

以微软 AI for Beginners 课程为线索
把知识讲成一本可以连续阅读的书

中文详解教材 · 全面扩展版

25 章主题详解 / 5 个强化专题 / 6 组实验 / 逐行代码 / 习题答案

2026 年 10 月 · 独立编写，非微软官方出版物

目录

目录里的章节名称可点击跳转；页码由排版程序自动生成。

阅读指南

写在前面：把课程真正读成一本书 5

第一部分 · 认识 AI 与规则系统

第01章｜人工智能究竟是什么 7

第02章｜让机器按照规则思考：符号 AI 11

第二部分 · 神经网络与 PyTorch（当前学习重点）

第03章｜感知机：神经网络的第一块积木 16

第04章｜多层感知机、矩阵和“形状” 23

第05章｜PyTorch：把神经网络写成可运行的程序 30

第三部分 · 计算机视觉

第06章｜计算机视觉：让计算机读懂像素 41

第07章｜卷积神经网络：在图片上寻找局部模式 45

第08章｜迁移学习：站在已有模型的肩膀上 49

第09章｜自编码器与 VAE：学习压缩信息 52

第10章｜GAN 与风格迁移：两种生成思路 55

第11章｜目标检测：物体是什么，又在哪里 58

第12章｜语义分割与 U-Net：给每个像素分类 61

第四部分 · 自然语言处理

第13章｜文本表示：计算机怎样“看见”一句话 65

第14章｜词向量：让“含义相近”有数学位置 69

第15章｜语言模型：预测下一个词 72
第16章｜循环神经网络：让信息沿着时间流动 76
第17章｜生成式 RNN：一个字接着一个字 80
第18章｜Transformer：不用循环，也能建模上下文 84
第19章｜命名实体识别：从句子中找出“谁、哪、何时” 88
第20章｜大语言模型与提示：认识今天常见的 AI 91

第五部分 · 其他 AI 技术与伦理

第21章｜遗传算法：用“选择、变异”寻找好方案 96
第22章｜强化学习：在尝试和反馈中改进行为 99
第23章｜多智能体系统：不止一个决策者 103
第24章｜负责任的 AI：技术之外还要考虑什么 106
第25章｜扩展阅读：多模态、CLIP 与 VQGAN 110

重点补课 · Python、数学与完整项目

学习专题 01｜Python 零基础补课：一行代码都不要跳 114
学习专题 02｜线性代数补课：从数字表到张量 120
学习专题 03｜概率、导数、梯度：深度学习数学不用先吓自己 124
学习专题 04｜不用框架，亲手实现一次梯度下降 128
学习专题 05｜做一个从数据到预测的完整 PyTorch 小项目 132

动手实验 · 六组完整练习

动手实验 01｜从一张计算表到可训练的感知机 138
动手实验 02｜PyTorch 多分类：从 logits 到准确率 142
动手实验 03｜不下载图片，也能理解 CNN 训练 146
动手实验 04｜中文短句分类：BoW 与线性层 150
动手实验 05｜注意力机制的矩阵形状逐项核对 154
动手实验 06｜诊断错误、复习安排与下一步学习路线 158

附录 · 语法、数学、术语、答案

附录 A｜Python 语法速查：看得懂再写得快 164
附录 B｜矩阵、张量与概率的最低必修课 166
附录 C｜随手查：神经网络术语表 168
附录 D｜全部章节思考题参考答案 170
版权与课程来源说明 173

写在前面：把课程真正读成一本书

微软的 **AI for Beginners** 采用 12 周、24 节核心课的学习路线，另外还提供扩展内容。原课程适合边读网页、边打开 Notebook 做实验。本书采用同样的主题顺序，却换成另一种读法：**先理解，再看公式，最后写代码**。这不是微软官方出版物，也不是原课的逐字翻译，而是一部围绕公开课程大纲重新编写的中文自学读本。

你的最近学习重点是神经网络、`nn.Linear`、`nn.ReLU`、`nn.Sequential` 和张量形状。按照这些练习的内容，你大致处在原课程第 3～5 课涉及的范围，但尚不能视为已经完成第 5 课。建议先读学习专题 01、02，再读第 3、4、5 章；学习专题 03、04、05 可随时补读。

本书的约定：每个关键术语写出英文；Python 例子尽可能逐行中文注释；公式只保留真正帮助理解的部分。遇到不懂的代码时，不必立刻运行，先弄清每一行的输入、输出与含义。每章有章节练习或检查题；部分答案紧跟题目，原课程思考题的参考答案集中在附录。另附五个从 Python 语法到端到端训练的学习专题。

> 每日阅读建议：一次读一章中的一个小节，合上书用自己的话复述概念，再做一道题。对于第 3～5 章，可以重复阅读，不用追求速度。

新版阅读说明：这是教材，不是三十页的提纲

这次重写坚持四个原则：**每一个重要概念至少有具体例子**；涉及形状时给出计算过程；展示 Python / PyTorch 代码时尽量逐行说明语法与用途；章节末尾安排能够核对答案的小练习。第 3～5 章与五个学习专题是重点，内容比其他章节更细，因为它们恰好对应初学者最容易卡住的基础。

本书既适合连续阅读，也适合按问题查阅。建议先阅读“学习专题 01 → 第 03 章 → 第 04 章 → 学习专题 02、03、04 → 第 05 章 → 学习专题 05”。当你能独立解释一次训练循环时，再系统阅读计算机视觉与自然语言处理。

版本与来源：这是一份基于微软公开课程主题重新编写的独立中文教材，并非微软官方出版或课程内容逐字翻译。官方课程还包含 Notebook、测验与实验，本书不代替动手运行。涉及框架 API 的示例以常见 PyTorch 使用习惯编写，具体环境版本仍应以官方文档为准。

第01章｜人工智能究竟是什么

1.1 先区分三个概念

人工智能（Artificial Intelligence，AI） 是一个目标：让机器完成通常需要人类智能的任务，例如识别图像、理解语言、制定计划、发现规律。它不只等于聊天机器人，也不一定必须使用神经网络。

机器学习（Machine Learning，ML） 是实现 AI 的一类方法。与其手工写出每条判断规则，不如让算法从数据中总结规律。比如，要识别垃圾邮件，你可以一条条规定“出现某词就判垃圾”，也可以提供已经标记好的邮件，让模型自己学习哪些特征比较有用。

深度学习（Deep Learning，DL） 则是机器学习中以多层神经网络为核心的方法。它很擅长处理图片、语音和文本，但往往需要更多数据、计算资源和调试。

因此，三个概念并不并列：深度学习属于机器学习，机器学习又属于人工智能的主要方法之一。还有一些 AI 方法不依赖机器学习，例如规则系统、搜索和逻辑推理。

1.2 一个具体例子

假设我们要做一个“猫狗图片识别器”。输入是一张图像，输出是“猫”或“狗”。传统规则很难穷举猫与狗的外形；机器学习则从大量带标签的图片中学习模式。

一个基本 AI 项目通常包括五步：确定任务、整理数据、选择模型、训练模型、用未见过的数据检验效果。第 3～5 章会逐个解释“模型”和“训练”到底是什么。

停下来想一想：一个写死了“如果温度超过 30℃，就打开风扇”的程序，算不算一定在使用机器学习？为什么？

1.3 把“会判断”拆成可以检查的五个环节

一个初学者常把“训练 AI”想象为：把一堆数据交给计算机，它就会自己获得智能。实际工程中，要先把问题拆清楚。以识别一封邮件是否为垃圾邮件为例，**输入（Input）**是邮件的标题与正文；**输出（Output）**是“垃圾邮件”或“正常邮件”；**标签（Label）**是训练时给出的正确答案；**模型（Model）**是把输入映射到输出的函数；**损失（Loss）**是预测和正确答案之间的差异。

这五个概念的关系是：我们先收集带标签的样本，再给模型输入，让模型输出预测，通过损失衡量预测质量，反复修改模型参数。如果只提供“正常邮件”而没有垃圾邮件，模型也许根本学不到区分两类的边界。数据不是越多越好，**是否代表将来真实使用的情况**更加关键。

举一个反例。你采集的垃圾邮件全是英文，正常邮件全是中文，模型可能只学会“英文等于垃圾”。训练准确率很高，但只要出现英文正常邮件，就会出错。这是**伪相关（Spurious Correlation）**：模型找到了容易利用的线索，却不是我们真正希望它理解的规则。

1.4 训练、验证、测试为什么一定要分开

想象你准备一次数学考试。**训练集（Training Set）** 是上课做过的题；**验证集（Validation Set）** 是你用来调整复习方法的模拟题；**测试集（Test Set）** 是最后考试前不能反复偷看的题。模型也一样：训练集更新权重，验证集帮助选择超参数，测试集只在最终评估时使用。

参数（Parameter） 通常指训练算法自动学出的权重和偏置；**超参数（Hyperparameter）** 是人为指定的训练设置，例如学习率、批次大小、隐藏层宽度和训练轮数。你可以在验证集上比较多个学习率，但如果反复在测试集上挑方案，测试集便不再是客观的“最后考试”。这类问题统称**数据泄漏（Data Leakage）**。

数据部分	主要作用	能否用来更新权重	一个常见错误
训练集	让模型学习	可以	混入重复样本而不知情
验证集	挑模型、调超参数	不直接更新	每轮盯着验证分数反复调到过拟合
测试集	最终报告能力	不可以	在测试集上挑出“最好的一次”

随机拆分并非永远正确。比如连续一天的传感器数据有强烈时间关系，就可能需要按时间先后拆分。又比如同一个人的多张医学图像，如果同时进入训练和测试，模型可能认出了这个人，而不是学会疾病特征。**拆分方式应当模拟真实部署时会遇到的新数据**。

1.5 预测正确率并不是唯一的评价标准

如果 1000 条交易只有 10 条欺诈，模型永远预测“正常”，仍然有 99% 的**准确率（Accuracy）**，但它完全没发现欺诈。此时还要看：**精确率（Precision）**，被模型说成“欺诈”的交易中有多少真的是；**召回率（Recall）**，所有真实欺诈中抓住了多少。不同任务在意的错误类型不同。

在药物安全监测中，漏报真正的严重事件和误报一次普通情况，代价不一样。在推荐系统中，评价可能用排序指标；在回归预测中，常用平均绝对误差（MAE）和均方误差（MSE）。不能仅凭一个百分数就断言模型很好。

1.6 初学者的第一个 AI 项目应该多简单

建议从能够在纸上解释的任务开始。比如利用“每天学习了几小时”和“做了几道习题”预测小测成绩。先画出输入与目标的关系，观察样本数量是否太少，再建立一个简单模型。不要为了“看起来先进”而直接上复杂网络。

判断项目是否做对了，可以用以下四个问题检查：①输入是什么，为什么能在预测时得到；②正确标签是谁提供的；③预测错了会怎样；④在从未见过的数据上表现如何。你能把四个问题说清楚，才真正开始做机器学习。

本章练习与思考

1. 你准备建立一个预测明天气温的模型。今天的最高温能做输入吗？明天的实际最高温能做输入吗？解释区别。
2. 一项欺诈检测任务中，正常交易占 99%。为什么单看准确率可能误导？
3. 训练过程中反复对照测试集做调参，为什么会让最终评价不可靠？

参考思路

第一题可以使用预测时已知的今天气温；不能提前使用尚未发生、预测时不可得的明天真实值。第二题因为“永远判正常”都能拿到 99% 准确率。第三题相当于考试前偷看试卷并按题目复习，测试集不再独立。

第02章｜让机器按照规则思考：符号 AI

2.1 不学习数据也能做 AI 吗？

可以。**符号人工智能（Symbolic AI）** 将知识表示成符号、事实和规则，再利用推理机制得出结论。比如：

- 事实一：小明是学生。
- 规则一：如果某人是学生，并且持有有效学生证，就可以享受学生票。

- 事实二：小明持有有效学生证。
- 结论：小明可以享受学生票。

这里并不需要训练神经网络。系统是在“已知事实 + 明确规则”的基础上进行推理。

2.2 专家系统与知识图谱

专家系统（Expert System） 把专业人员的经验转写成规则，常用于适合清晰规则的场景。它的优点是较容易解释判断路径；难点是维护规则昂贵，现实世界存在太多模糊和例外。

知识图谱（Knowledge Graph） 把实体以及关系组织为网络。例如“北京 → 是中国的首都 → 中国”。查询和推理可以沿着这些关系展开。它与神经网络可以结合，但并不是一回事。

要记住这两类方法的差异：符号 AI 更像把知识“写进机器”，机器学习更像让机器“从例子中学”。现代系统常常融合多种方法。

停下来想一想：如果你要做一个严格执行税法条款的检查系统，纯规则方法为什么可能比黑箱分类器更方便审查？

2.3 从一句“如果……那么……”到逻辑推理

规则系统的核心是**知识表示（Knowledge Representation）**。例如事实 `bird(Tweety)` 表示 Tweety 是一只鸟；规则 `bird(x) AND healthy(x) -> can_fly(x)` 表示“鸟并且健康，通常能飞”。这里的 `AND` 是逻辑与，箭头表示规则的推导方向。但要小心：

真实世界里企鹅也是鸟，却不会飞。“通常”与“总是”完全不同，随意把经验写成绝对规则会出错。

在**前向推理（Forward Chaining）**中，系统从已知事实出发，不断尝试应用规则得到新结论：已知“某人是学生”“有有效证件”，就推导“允许购买学生票”。在**后向推理（Backward Chaining）**中，系统从问题出发反问“要让某人可以买学生票，需要满足哪些条件”，再回头核实事实。

推理方向	起点	常见用途
前向	已知事实	根据不断更新的数据触发结论
后向	要证明的目标	回答一个具体查询、诊断条件是否满足

逻辑系统可能使用**封闭世界假设（Closed-World Assumption）**：未能证明为真的事，默认当作假。这对于完整数据库中的查询很实用，却不能套在所有现实问题上。数据库里没有记录“某人患病”，不等于能断言这个人没有患病。复杂知识需要区分“假”“真”“未知”。

2.4 用最少 Python 写一个规则引擎

先不要担心任何第三方框架，只看 Python 最基础的条件判断。下面是一个决定是否满足图书馆借书条件的例子。

```
# 变量表示现实中的事实；布尔值 True 表示真，False 表示假
has_card = True       # 是否持有有效借书证
has_debt = False      # 是否存在未处理的欠费

# and 表示两个条件同时成立，not 表示逻辑取反
can_borrow = has_card and (not has_debt)  # 证件有效并且没有欠费

# if 根据布尔结果选择分支，冒号表示下面开始一个缩进代码块
if can_borrow:                # 如果 can_borrow 为 True
    print('允许借书')          # 缩进的语句属于 if 分支
else:                         # 否则，执行另一个分支
    print('暂时不能借书')      # 在不满足条件时显示原因类别
```

你会发现，这个程序不需要训练：规则早就由人写好了。这是它的优点，也是限制。如果规则不断增加，可能遇到“规则 A 说可以、规则 B 说不可以”这样的冲突。此时需要定义规则优先级、例外机制与审计日志，否则系统不可维护。

2.5 本体和知识图谱不是一张普通的关系图

本体（Ontology） 定义一类知识的概念、属性与关系。例如“猫属于哺乳动物”“哺乳动物属于动物”，可以通过类别继承推导“猫属于动物”。**知识图谱（Knowledge Graph）** 则将具体实体与关系连成网络。图谱可能包含“某位作者写了某本书”“某本书属于机器学习主题”等边。

三者区别可以这样记：规则描述“满足何条件得何结论”；本体描述“世界中的概念如何组织”；知识图谱描述“具体实体之间有哪些可查询的关系”。今天的问答系统有时会把知识图谱与神经网络结合：让图谱提供可核查事实，让语言模型负责理解问题与组织回答。

课堂小实验：找到规则的漏洞

请设计一个“年龄满 18 且完成实名认证，可以开通某功能”的规则。然后问自己：年龄计算使用哪一天？如果认证过期怎么办？未成年人的监护人同意是否改变规则？这些问题都不在简单的 `and` 之中。真正的系统设计还需要时间、异常和业务约束。

本章练习与解答

1. “根据此前例子自动调整权重”和“照预写条件判断”谁属于典型机器学习？答：前者。
2. “数据库没有这条记录，因此现实中不存在这件事”有什么问题？答：把信息缺失误当成事实为假。
3. 为什么适合审计的场景喜欢规则系统？答：每一步结论可以追溯到具体事实与规则，但前提是规则本身完整且正确。

第03章｜感知机：神经网络的第一块积木

3.1 一个神经元做什么

感知机（Perceptron） 可以理解为“把多个数字加权相加，再作出决定”。设有两个输入特征 `x1`、`x2`，各自对应权重 `w1`、`w2`，再加上偏置 `b`：

```
z = x1*w1 + x2*w2 + b
```

其中，**特征（Feature）** 是输入信息；**权重（Weight）** 表示某个特征在当前模型中的作用大小；**偏置（Bias）** 让判断界线不必经过原点；`z` 是加权求和结果。

一个经典感知机随后使用阶跃函数：若 `z > 0`，输出 1；否则输出 0。你可以把它理解为一个二选一的判断开关。

3.2 做一次手算

假设 `x1=2`、`x2=3`、`w1=0.5`、`w2=-1`、`b=1`，那么：

```
z = 2*0.5 + 3*(-1) + 1 = -1
```

因为 `z` 不大于 0，所以输出 0。这一步没有神秘算法，只用到了乘法、加法和一次判断。

3.3 为什么还需要多层网络

单个感知机能处理一条线可以分开的简单分类问题，但遇到 XOR（异或）这种非线性关系时就无能为力。XOR 的规则是：两个输入不同才输出 1。无论怎样画一条直线，都无法把正例和反例完全分开。于是我们需要多个神经元与多个层共同工作。

3.4 不用框架，写出最小版本

```
# 定义两个输入特征：这里先使用普通 Python 数字
x1 = 2              # 第一个输入特征，数值为 2
x2 = 3              # 第二个输入特征，数值为 3
w1 = 0.5            # 第一个特征对应的权重
w2 = -1.0           # 第二个特征对应的权重
b = 1.0             # 偏置，让判断边界可以平移
z = x1*w1 + x2*w2 + b  # 加权求和，得到 -1.0
y = 1 if z > 0 else 0 # 条件表达式：大于 0 输出 1，否则输出 0
print(y)            # print 是显示结果的函数，这里打印 0
```

注意最后一行之前的 `if ... else ...`：它不是神经网络特有语法，而是 Python 的条件表达式。看懂普通语法，比急着记住大量神经网络函数更重要。

停下来想一想：如果 `w1` 变为 2，其他数字不变，上面的感知机输出会变成多少？

3.5 用几何图像理解“权重”和“偏置”

想象平面上每个样本只有两个特征 `x1` 与 `x2`。单个感知机通常先算 `z = w1*x1 + w2*x2 + b`，再根据 `z` 是否大于 0 决定类别。所有满足 `z=0`

的点组成一条直线，这就是**决策边界（Decision Boundary）**。调整 `w1`、`w2` 相当于改变线的方向；调整 `b` 则可以让线整体平移。

比如 `w1=1`、`w2=1`、`b=-3`，边界就是 `x1+x2=3`。点 `(2,2)` 的 `z=1`，落在输出 1 的一侧；点 `(1,1)` 的 `z=-1`，落在输出 0 的一侧。你不需要先掌握复杂几何，只需要知道：**训练的一个目标就是找到合适的分界位置**。

你还可以用一张表自己试算：

x1	x2	z=x1+x2-3	预测
0	0	-3	0
1	1	-1	0
1	3	1	1
2	2	1	1

有一点很容易混淆：经典感知机的输出是离散的 0 或 1，现代神经网络中也可能使用连续激活，如 Sigmoid 或 ReLU。它们都从线性组合开始，但后续函数与训练方法可以不同。

3.6 为什么 XOR 使单层感知机失败

异或（Exclusive OR，XOR）的逻辑是“只有两个输入不同，答案才是 1”。列出真值表：

x1	x2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

在二维坐标图中，两个正例 `(0,1)` 与 `(1,0)` 落在对角方向，两个负例 `(0,0)` 与 `(1,1)` 落在另一条对角方向。无法画出一条直线让两个正例在同一侧、两个负例在另一侧。单个线性决策边界就做不到。

但我们可以先让两个隐藏神经元分别处理“至少有一个输入为真”和“两个输入都为真”，再组合成 XOR。这个思想告诉我们：**隐藏层不是装饰，它能制造新的中间表示，让简单规则组合成更复杂的判断**。

3.7 感知机究竟怎样学习：亲手更新一次权重

当模型预测错误时，我们希望自动调整权重。经典感知机有一种简单更新规则：`w_new = w_old + lr*(target-prediction)*x`，偏置也用同样的误差信号更新。`lr` 叫**学习率（Learning Rate）**，决定每一步调整的大小。

假设样本是 `x=[1,2]`，真实标签 `target=1`；当前 `w=[0,0]`、`b=0`，阈值规则是“大于 0 输出 1”，因此预测 `prediction=0`。取学习率 `lr=0.1`，

误差为 `1-0=1`。新权重为 `[0.1, 0.2]`，新偏置为 `0.1`。再输入同一个样本，`z=0.1*1+0.2*2+0.1=0.6`，输出就会变成 1。

请注意：**这里的更新是经典感知机算法，不等同于后面用均方误差和反向传播训练的多层神经网络**。两者都在调整权重，却使用不同的目标与更新方法。

3.8 不用 PyTorch，写一个能够训练的感知机

下面的示例刻意只用 Python 的列表、循环和条件语句。用 OR（逻辑或）数据训练：只要两个输入中至少一个是 1，就输出 1。所有重要行都写了中文注释。

```
# 四个训练样本，列表里每个小列表代表一个样本的两个特征
samples = [[0, 0], [0, 1], [1, 0], [1, 1]]
# 标签分别表示 OR 规则下的正确答案
targets = [0, 1, 1, 1]

# 一共有两个输入特征，因此用两个权重，开始时都设为 0
weights = [0.0, 0.0]
bias = 0.0                    # 模型的偏置参数
lr = 0.2                      # learning rate，学习率为 0.2

# epoch 表示完整看一遍所有样本；这里训练 10 轮
for epoch in range(10):
    # zip 把样本和它对应的标签配成一对，便于一起遍历
    for x, target in zip(samples, targets):
        # 两个输入分别乘对应权重，再加偏置得到 z
        z = x[0] * weights[0] + x[1] * weights[1] + bias
        prediction = 1 if z > 0 else 0  # 阈值判断得到 0 或 1
        error = target - prediction     # 标签减预测，得到更新方向
        # range(2) 产生索引 0 和 1，分别修改两个权重
        for j in range(2):
            weights[j] += lr * error * x[j]  # += 意味着加到原值上
        bias += lr * error                # 偏置也根据误差更新

# 训练完成后，逐个测试四个输入组合
for x in samples:
    z = x[0] * weights[0] + x[1] * weights[1] + bias
    print(x, 1 if z > 0 else 0)          # 打印样本和模型预测
```

这一段代码里 `for` 就是循环，`zip` 的任务只是同步取两个列表的元素，`j` 是索引（Index）。一旦把这些普通 Python 语法看懂，机器学习中的“更新权重”就不那么神秘了。

一边读一边想

· 如果错误是 0，权重会怎样？答：这一样本不会让权重发生变化。

- 为什么更新式里乘了 `x[j]`？答：这一特征为 0 时，它对当前线性组合不直接贡献，更新量也为 0。
- 为什么不能保证这段代码解决 XOR？答：单个感知机的线性边界无法分开 XOR 数据，不是“训练轮数不够”这么简单。

3.9 把感知机与神经网络术语对上号

从计算角度，感知机完成三步：①接受数字形式的特征；②每个特征乘权重，最后求和加偏置；③根据激活规则决定输出。这里的**前向传播（Forward Pass）**指沿着输入到输出方向计算。经典感知机直接更新权重；现代多层网络通常还需要通过求导做**反向传播（Backpropagation）**。

当你看到 `nn.Linear(2,1)` 时，可以想成“把每个样本的两个特征，计算成一个输出数字”。`Linear` 默认包含两个输入权重和一个偏置。神经网络看起来有很多层，其基本计算仍可拆成这样的小步骤。

本章进阶习题与答案

1. 计算 `x=[2,1], w=[0.5,-1], b=0.25` 的 z。答案：`2*0.5 + 1*(-1) + 0.25 = 0.25`。
2. 将上述偏置换成 `-0.25`，使用 `z>0` 的阈值，输出变成什么？答案：`z=-0.25`，输出 0。
3. 单层感知机可以解决 OR 但不能解决 XOR，主要原因是？答案：前者可线性分开，后者不可线性分开。

第04章｜多层感知机、矩阵和“形状”

4.1 三种层各自负责什么

多层感知机（Multi-Layer Perceptron，MLP） 把神经元排成多个层。**输入层（Input Layer）** 接收原始特征；**隐藏层（Hidden Layer）** 把输入转换为中间特征；**输出层（Output Layer）** 给出最终结果。

举例：输入维度 4，隐藏层 5 个神经元，输出维度 2。可以把整个结构写成：

`4 个输入特征 → 5 个隐藏特征 → 2 个输出特征`

这里的 4、5、2 **不是样本数量**，而是每个样本在各层中的特征数量。初学者非常容易混淆两者。

4.2 为什么 (3,4) 乘 (4,2) 会得到 (3,2)

假设有 3 个样本，每个样本有 4 个特征：输入矩阵的形状（Shape）就是 `(3,4)`。现在要把每个样本转换为 2 个新特征，可以用一个 `(4,2)` 的参数矩阵：

`(3,4) × (4,2) = (3,2)`

理解方法：左边矩阵提供 3 行输入，每行是一个样本；右边矩阵提供 2 列，每列代表一种输出特征的计算规则。左边的每一行和右边的每一列配对，算出一个数字，因此会有 `3 × 2 = 6` 个输出数字。

中间两个 4 不是直接“约掉”，而是说明每次行列配对都要进行 4 次对应元素相乘，并把结果相加。只有两边的这个长度一样，常规矩阵乘法才有定义。

4.3 一行乘一列，亲手计算

`[1, 2, 3, 4] × [5, 6, 7, 8]^T`

结果是 `1*5 + 2*6 + 3*7 + 4*8 = 70`。这产生结果矩阵中的一个位置。记住：一个“行一列”配对，得到一个数；多少行乘多少列，决定最终有多少格。

4.4 激活函数为什么必要

把多个线性层连续相乘，如果中间没有**非线性激活函数（Activation Function）**，整体仍等价于一次线性变换。这样多层结构无法充分表达复杂关系。

常见的 **ReLU（Rectified Linear Unit）** 规则很简单：输入大于 0，就原样保留；输入不大于 0，就输出 0。写成 `ReLU(x) = max(0,x)`。例如输入 `[-2, 3, 0.5]`，输出就是 `[0, 3, 0.5]`。

4.5 神经网络为什么能变聪明

前向传播（Forward Propagation）根据当前权重计算预测；损失函数（Loss Function）衡量预测错了多少；反向传播（Backpropagation）通过链式法则计算每个参数对损失的影响；优化器（Optimizer）据此调整参数。这四步反复进行，就是最基本的训练循环。

你现在只要抓住：**前向算答案 → 损失衡量错误 → 反向算调整方向 → 更新权重**。具体求导在下一章结合 PyTorch 解释。

停下来想一想：如果输入形状是 `(8,10)`，经过 `10→6→2` 两个线性层，形状依次是什么？

4.6 矩阵乘法逐格算：不要只背形状口诀

设 `A` 为 3 行 4 列，`B` 为 4 行 2 列。为了让每个数字都能手算，我们选取简单整数：

```
A = [ [1, 2, 0, 1],
      [0, 1, 1, 2],
      [2, 0, 1, 1] ]          形状 (3,4)

B = [ [1, 0],
      [0, 2],
      [3, 1],
      [1, 1] ]              形状 (4,2)
```

计算结果第一行第一列：选 A 的第一行 `[1,2,0,1]` 和 B 的第一列 `[1,0,3,1]`，算 `1*1+2*0+0*3+1*1=2`。第一行第二列：选 A 的第一行和 B

的第二列 `[0,2,1,1]`，算 `1*0+2*2+0*1+1*1=5`。所以结果第一行是 `[2,5]`。

第二行：`C[1,0]=0*1+1*0+1*3+2*1=5`；`C[1,1]=0*0+1*2+1*1+2*1=5`。第三行：`C[2,0]=2*1+0*0+1*3+1*1=6`；`C[2,1]=2*0+0*2+1*1+1*1=2`。最后得到：

```
C = A @ B = [ [2, 5],
              [5, 5],
              [6, 2] ]      形状 (3,2)
```

这里 `@` 是 Python 和 PyTorch 里的**矩阵乘法运算符（Matrix Multiplication Operator）**。注意 A 有三行，每行对应结果的一行；B 有两列，每列对应结果的一列。中间的 4 决定每一个输出数字要相乘、相加多少次。两个 4 并没有进行普通代数意义上的“约分”。

4.7 神经网络里面的矩阵为什么长成这样

假设 `X.shape=(3,4)`，它是一个小批次（Batch）：3 个样本，每个样本 4 个特征。隐藏层要产生 5 个特征，可以理解为准备 **5 套不同的权重规则**，每套规则都观察输入的 4 个数。

从纯数学记法看，准备一张 `(4,5)` 的权重矩阵 `W`，运算 `X @ W` 就得到 `(3,5)`。再让每个元素通过 ReLU，形状仍是 `(3,5)`。第二层使用 `(5,2)` 的权重矩阵，得到最终 `(3,2)`。

步骤	运算	输入形状	输出形状	应当说得清什么
输入	原始数据 X	(3,4)	(3,4)	3 个样本，每个 4 个特征
隐藏层	Linear(4,5)	(3,4)	(3,5)	每个样本变成 5 个特征
非线性	ReLU	(3,5)	(3,5)	数值可变，形状不变
输出层	Linear(5,2)	(3,5)	(3,2)	每个样本得到 2 个输出

为什么 PyTorch 显示 `nn.Linear(4,5).weight.shape` 为 `(5,4)` 呢？因为它把每一个输出神经元的一组输入权重放成一行。调用时内部相当于 `X @ W.T + b`，转置后 `W.T.shape=(4,5)`，恰好符合我们刚才的数学图景。**存储方式和实际乘法方向要分别看，不能把 `weight.shape` 直接当作 X 右边乘的形状。**

4.8 什么时候能加，什么时候能乘

同形状矩阵之间可以逐元素相加，比如两个 `(3,4)` 相加仍是 `(3,4)`。但两个 `(3,4)` 做普通矩阵乘法 `A @ B` 却不合法：A 有 4 列，B 只有 3 行，不满足匹配要求。别把 `*` 和 `@` 混淆：在 NumPy、PyTorch 中，`*` 通常是**逐元素乘法（Element-wise Multiplication）**，`@` 才是矩阵乘法。

还有一种常见机制叫**广播（Broadcasting）**。例如 `(3,2)` 的结果加上长度为 2 的偏置向量 `(2,)`：系统会把这两个偏置数应用到三行中的每一行上。得到的形状仍然是 `(3,2)`。广播有明确规则，不是任何两个不同形状都能任意相加。

```
import torch                                # 导入 PyTorch
A = torch.tensor([[1., 2.], [3., 4.]])  # 创建 2 行 2 列的浮点张量
B = torch.tensor([[5., 6.], [7., 8.]])  # 创建另一个形状相同的张量
print(A * B)                                # 对应元素相乘，不是矩阵乘法
print(A @ B)                                # 行与列相乘后求和，得到矩阵积
b = torch.tensor([10., 20.])            # 长度为 2 的一维偏置张量
print(A + b)                                # 广播：两行都分别加 10 和 20
```

4.9 求导到底与“训练”有什么关系

设模型只有一个可训练权重 `w`，输入固定为 `x=2`，希望输出等于 `y=6`。模型预测 `prediction=2*w`，损失取平方误差 `L=(2*w-6)^2`。

若 `w=1`，预测值是 2，损失为 16；若 `w=2`，预测为 4，损失为 4；若 `w=3`，预测为 6，损失为 0。我们当然希望让 `w` 往 3 走，但在大型神经网络中有成千上万甚至更多权重，不可能靠试遍所有数字。

导数（Derivative） 给出一个局部问题的答案：当 w 微小改变时，损失大约怎样变化？对上式求导，得到 `dL/dw = 4*(2*w-6)`。在 `w=1` 时导数是 `-16`，说明增大 w 会使损失局部下降；在 `w=4` 时导数是 `8`，说明减小 w 会使损失下降。

梯度下降（Gradient Descent） 采用 `w_new = w_old - lr*gradient`。当梯度负，减去负数等于增加权重；当梯度正，减去正数等于减少权

重。这就是优化器更新的最简单直觉。学习率不能随意过大：走得太猛可能跨过最优位置，来回震荡。

4.10 反向传播：把导数从最后一层传回去

先看两步计算：`a = w*x`，`L=(a-y)^2`。第一步输入权重，第二步计算损失。我们想知道“修改 w 如何影响 L”，可以拆成“修改 w 如何影响 a”乘以“修改 a 如何影响 L”。这叫**链式法则（Chain Rule）**：

`dL/dw = (dL/da) * (da/dw)`。

在这里 `dL/da=2*(a-y)`，`da/dw=x`，所以 `dL/dw=2*(a-y)*x`。若 `x=2, w=1, y=6`，则 `a=2`，梯度为 `2*(2-6)*2=-16`，与上节一致。

对真正的多层网络，链式法则会跨越多个线性层和激活函数。之所以叫**反向传播（Backpropagation）**，是因为我们先完成从输入到预测的前向计算，再从最终损失沿计算依赖关系把梯度传回较早的参数。PyTorch 自动求导就是帮我们追踪并计算这些关系，但理解链式法则仍然重要。

4.11 小测试：形状与参数分别数清楚

给定 `nn.Linear(4,5)`，输入是 `(8,4)`。它输出 `(8,5)`。这个层的权重参数数目是 `4*5=20`，偏置参数数目是 `5`，合计 **25 个可训练参数**。它与批次中的样本数 8 无关：改成 100 个样本，模型参数数量仍是 25。

再给它接 `nn.Linear(5,2)`，第二层参数数目是 `5*2+2=12`。两层合计 37 个参数。中间 ReLU 不增加可训练参数。你能算出这三件事，就已经

具备读懂大量初学者神经网络结构的能力：**输入输出 shape、每层作用、参数规模**。

练习与详细解答

1. `(10,8) @ (8,3)` 得到什么形状？答案：`(10,3)`，因为 10 个输入行分别与 3 列配对。
2. `(5,2) @ (3,4)` 能算吗？答案：不能，左矩阵列数 2 与右矩阵行数 3 不相等。
3. `nn.Linear(6,4)` 有多少参数？答案：`6*4 + 4 = 28`。
4. 为什么 `ReLU` 不会把 `(7,5)` 变成 `(7,2)`？答案：它对各元素施加函数，不负责合成新特征数量；后者需使用改变维度的层。
5. 经过 `4→5→2` 网络，一个批次 12 个样本的形状怎样变化？答案：`(12,4) → (12,5) → (12,5) → (12,2)`。

第05章｜PyTorch：把神经网络写成可运行的程序

5.1 先认识两个对象

PyTorch 是一个深度学习框架，提供张量（Tensor）、自动求导（Autograd）和模型训练工具。张量可以先理解为带形状的多维数

组：一个数是零维张量，一排数是一维张量，一张数字表是二维张量。`shape` 表示每个维度分别有多长。

```
import torch                    # import 表示导入模块；torch 是 PyTorch 的主模块
x = torch.randn(3, 4)          # randn 生成形状 (3,4) 的标准正态随机数张量
print(x.shape)                 # shape 是张量的形状属性，会显示 torch.Size([3, 4])
```

标准正态分布（Standard Normal Distribution） 指均值为 0、标准差为 1 的正态分布。`randn` 随机生成的数据不要求每个数字都靠近 0，也不保证有限个样本的平均数恰好是 0。

5.2 从一层到多层

```
import torch                    # 导入 PyTorch 主模块，用于创建张量
from torch import nn           # from ... import ... 是从模块取出成员；nn 包含神经网络层

model = nn.Sequential(          # Sequential 表示按顺序把后面的层连接起来
    nn.Linear(4, 5),            # Linear 线性层：把每个样本的 4 个特征变成 5 个
    nn.ReLU(),                  # ReLU 激活层：把负数变为 0，正数保持不变
    nn.Linear(5, 2)             # 第二个 Linear：把 5 个特征变成 2 个
)

x = torch.randn(3, 4)          # 构造 3 个样本，每个样本有 4 个输入特征
output = model(x)              # 把 x 依次送过三层，得到模型输出
print(output.shape)            # 显示 torch.Size([3, 2])，样本数仍然是 3
```

逐步跟踪形状：输入 `(3,4)` → 第一线性层 `(3,5)` → ReLU 不改变形状 `(3,5)` → 第二线性层 `(3,2)`。一定要分清：**层改变的是最后一个特征维度；并没有把 3 个样本变成 2 个样本**。

5.3 一个容易让初学者困惑的地方

`nn.Linear(4, 2)` 表示输入特征数是 4、输出特征数是 2，但 PyTorch 保存的权重参数 `weight.shape` 实际是 `(2,4)`。这是因为内部计算使用 `x @ weight.T + bias`，其中 `.T` 是转置（Transpose）。所以 `(3,4) @ (4,2) = (3,2)`，没有矛盾。

5.4 为什么写出网络还不等于训练

刚刚的 `model` 虽然能产生输出，但是权重还没有经过任务数据学习。训练必须有真实目标 `y`、衡量错误的损失函数以及优化器。例如二分类和回归需要选择不同的损失函数；不能不看任务就随意套用。

```
import torch                                  # 导入张量和自动求导功能
from torch import nn                          # 从 PyTorch 导入神经网络模块

model = nn.Linear(1, 1)                       # 创建一个输入 1 维、输出 1 维的线性模型
x = torch.tensor([[1.0], [2.0], [3.0]])  # tensor 将数据变为张量；3 个样本，各 1 个特
征
y = torch.tensor([[2.0], [4.0], [6.0]])  # 目标标签，期望模型学到大致 y=2x
loss_fn = nn.MSELoss()                        # MSELoss 是均方误差损失函数，适合这个回归
例子
optimizer = torch.optim.SGD(                  # SGD 是随机梯度下降优化器，这里做参数更新
    model.parameters(), lr=0.01               # parameters() 取出可训练参数；lr 是学习率
)

for step in range(200):                       # for 用于循环；range(200) 产生 0 到 199
    prediction = model(x)                     # 前向传播：计算当前模型的预测值
    loss = loss_fn(prediction, y)             # 计算预测值与真实值的误差
    optimizer.zero_grad()                     # 清空上一次迭代留下的梯度，避免累积
    loss.backward()                           # 反向传播：自动计算参数对损失的梯度
    optimizer.step()                          # 使用梯度更新模型的权重和偏置

print(model(torch.tensor([[4.0]])))           # 用学过的模型对新输入 4.0 进行预测
```

这个例子的结果一般会接近 8，但随机初始权重和训练次数会影响实际数值。你不需要一次背诵所有代码，只要记住训练循环的四行核心步骤：计算预测、计算损失、反向传播、更新参数。

5.5 过拟合到底是什么

过拟合（Overfitting） 是指模型对训练数据记得非常好，但对新数据表现差。它像一个学生只背住模拟题答案，却无法解决稍有变化的题。常见应对方法包括：训练集与验证集分开、收集更可靠的数据、减少模型复杂度、权重正则化，以及在合适时使用 Dropout。

停下来想一想： 为什么 `nn.ReLU()` 不会让 `(3,5)` 变成 `(3,2)`？如果模型训练准确率很高、验证准确率低，你首先应该怀疑什么？

5.6 读懂 Python 代码前，先消除四个语法障碍

第一个常见障碍是 `import`。`import torch` 表示把名为 `torch` 的模块导入当前程序，你才能使用 `torch.tensor()`。`from torch import nn` 表示直接拿出 `torch` 中的 `nn` 成员，因此写 `nn.Linear(...)`；`import torch.nn as nn` 是另一种常见写法，`as` 表示取别名。导入不会自动“训练神经网络”，它只是告诉 Python 你要使用哪些工具。

第二个障碍是 `.` 点号。`model.parameters()` 中，`model` 是对象，`.` 表示访问对象的方法或属性。末尾的 `()` 表示调用方法，`parameters()` 会返回模型的参数集合。反过来，`x.shape` 通常是属性，不加括号。`x.reshape(3,2)` 是调用方法，它会返回一个新形状的张量视图或副本（取决于内存布局）。

第三个障碍是 `=` 与 `==`。`x=5` 是给变量赋值，`x==5` 是比较两个值是否相等。第四个障碍是缩进（Indentation）：在 `for` 或 `if` 后，缩进的语句属于这个代码块。神经网络的训练循环其实就是 Python 的普通循环，只是循环里调用了模型、损失和优化器。

5.7 张量的 shape、dtype、device 代表三个不同问题

`shape` 描述尺寸；`dtype` 描述元素的数据类型；`device` 描述数据在哪个计算设备上。刚开始只要分清三者，就能看懂绝大多数初级 PyTorch 报错。

```
import torch                                  # 导入 PyTorch
x = torch.tensor([[1.0, 2.0], [3.0, 4.0]]) # 创建一个二维浮点张量
print(x.shape)                               # 两行两列，torch.Size([2, 2])
print(x.dtype)                               # 通常是 torch.float32
print(x.device)                              # 默认在 cpu（除非改变默认设置）
print(x.ndim)                                # 维数为 2；ndim 不是元素总数
print(x.numel())                             # 元素总数为 4
print(x[0, 1])                               # 用索引读取第一行第二列，也就是 2.0
```

训练时常见问题是输入张量在 CPU，模型在 GPU；或者标签是浮点而某个分类损失要求长整型。遇到报错时，先打印形状、类型、设备和必要的张量值，远比盲目修改网络层更有效。

5.8 自动求导：手动看一次 backward 的结果

下面只用一个变量展示 PyTorch 的**自动微分（Automatic Differentiation，Autograd）**。由于 `requires_grad=True`，框架会追踪与 `w` 有关的运算，之后才能计算损失对 w 的梯度。

```
import torch                                # 导入张量与自动求导模块
w = torch.tensor(1.0, requires_grad=True) # 0 维张量，初值 1，允许求梯度
x = torch.tensor(2.0)                      # 固定输入 2，不需要更新
true_y = torch.tensor(6.0)                 # 期望的正确输出是 6
prediction = w * x                         # 模型预测：1*2 = 2
loss = (prediction - true_y) ** 2          # ** 2 表示平方：损失是 16
loss.backward()                            # 沿计算图求损失对 w 的导数
print(w.grad)                              # 梯度为 -16，表示增加 w 可降低损失
```

这一例子与第 4 章的手算对应。请区分：`loss` 是损失值，`loss.backward()` 执行梯度计算，`w.grad` 保存梯度；`backward()` **不会自动更新 w**。真正更新参数通常需要优化器的 `step()`。

5.9 从头走一遍完整训练循环

我们现在用 8 个简单的训练样本模拟回归任务，规律为 `y≈2*x+1`。这里把数据分成训练和验证部分，给出可以完整执行的最小版本。每一行既说明 Python 语法，也说明机器学习作用。

```
import torch                                  # 张量与自动求导的主模块
from torch import nn                          # 引入神经网络层与损失函数

torch.manual_seed(7)                          # 固定随机种子便于重复实验
x_train = torch.tensor([[0.], [1.], [2.], [3.], [4.], [5.]]) # 6 个训练输入
y_train = 2 * x_train + 1                     # 创建正确标签，满足 y=2x+1
x_val = torch.tensor([[6.], [7.]])            # 2 个验证输入，不用于更新权重
y_val = 2 * x_val + 1                         # 对应的验证标签

model = nn.Linear(1, 1)                       # 创建一个只有一层的线性回归模型
loss_fn = nn.MSELoss()                        # MSE：平均平方误差，适合回归
optimizer = torch.optim.SGD(                  # SGD：随机梯度下降优化器
    model.parameters(), lr=0.01               # 接收可训练参数，学习率设置为 0.01
)

for epoch in range(800):                      # epoch：完整遍历训练数据的一轮
    model.train()                             # 设置为训练模式；某些层行为会变化
    prediction = model(x_train)               # 前向传播，预测训练样本
    loss = loss_fn(prediction, y_train) # 与训练标签比较，计算损失
    optimizer.zero_grad()                     # 清空上一轮累积的梯度
    loss.backward()                           # 自动计算各参数的梯度
    optimizer.step()                          # 按梯度和学习率更新参数

model.eval()                                  # 切换为评估模式，关闭训练特定行为
with torch.no_grad():                         # 禁止构建梯度计算图，降低评估开销
    val_pred = model(x_val)                   # 只计算预测，不更新权重
    val_loss = loss_fn(val_pred, y_val) # 衡量未用于训练的数据的误差
print('validation loss:', val_loss.item()) # item() 将单元素张量转成 Python 数字
print('weight:', model.weight.item())   # 查看学到的权重，应接近 2
print('bias:', model.bias.item())       # 查看学到的偏置，应接近 1
```

这段代码里的 `model.train()` 与 `model.eval()` 尤其容易误读：`train()` 只是把模型切换到训练模式，并不是“执行完整训练”；`eval()` 只是切换推理模式，并不会帮你自动计算准确率。像 Dropout、BatchNorm 这样的层会根据模式改变行为。真正反向传播仍靠 `backward()` 和 `step()`。

5.10 训练循环中四行代码不能交换顺序

有必要把最常见的四行写成“动作清单”：`prediction=model(x)` 产生预测；`loss=loss_fn(prediction,y)` 衡量错误；`optimizer.zero_grad()` 清空旧梯度；`loss.backward()` 计算新梯度；最后 `optimizer.step()` 用梯度更新参数。

在较常见的写法中，你也可能看到 `zero_grad()` 被移到前向传播前，只要保证在当前一次 `backward()` 前清空应当清空的旧梯度即可。不要机械地记死某一行的位置；要理解梯度累积机制。**默认情况下，PyTorch 的 `.grad` 会累积，连续调用 backward 而不清零并不是“重新计算覆盖旧值”。** 只有你有意实现梯度累积（Gradient Accumulation）时才应该这么做。

假设模型只有一个 w，第一次得到梯度 3，第二次得到梯度 5。如果不清空，`.grad` 可能积累为 8。不是框架算错，而是它按设计在累积。初学者不理解这点，可能会出现训练步长突然变大、损失震荡的问题。

5.11 二分类和多分类，损失函数并不相同

回归（Regression） 预测一个连续数字，如价格或温度，常用 MSE。**二分类（Binary Classification）** 在两个类别中选择一个，常见做法是模型输出一个未归一化分数（Logit），再用 `nn.BCEWithLogitsLoss()`。**多分类（Multi-class Classification）** 从三个或更多互斥类别选一个，常用 `nn.CrossEntropyLoss()`。

`CrossEntropyLoss` 通常接收形状 `(batch_size, num_classes)` 的原始 logits，不要先自己手动加 `softmax` 再送给它。标签通常是形状 `(batch_size,)` 的整数类别索引，类型一般为 `torch.long`。例如 batch 有 3 个样本、每个样本 4 种类别，那么 logits 是 `(3,4)`，标签可以是 `[2,0,3]`，而不是每个标签写成 4 维独热向量。

```
import torch                                    # 导入 PyTorch
from torch import nn                           # 导入神经网络模块
logits = torch.tensor([[2., 0., -1.],       # 第一个样本有三个类别分数
                       [0., 1., 3.]])      # 第二个样本也有三个类别分数
labels = torch.tensor([0, 2], dtype=torch.long) # 正确类别索引为 0 和 2
loss_fn = nn.CrossEntropyLoss()             # 多类交叉熵，内部处理必要变换
loss = loss_fn(logits, labels)              # 计算损失，得到标量张量
print(loss.item())                          # 打印 Python 浮点损失值
```

5.12 过拟合、欠拟合、数据泄漏，如何区分

若训练损失很大、验证损失也很大，可能是**欠拟合（Underfitting）**：模型太简单、训练不够或者输入信息不足。若训练损失不断降低、验证损失却开始上升，常见怀疑是**过拟合（Overfitting）**：模型学到了训练样本特有的噪声。若验证分数异常完美，则还要检查**数据泄漏（Data Leakage）**，例如重复样本或直接把答案字段放进输入。

现象	优先检查	可尝试的办法
训练和验证都差	数据标签、模型容量、训练轮数	修数据、增加容量、调整优化设置

现象	优先检查	可尝试的办法
训练很好，验证差	数据量、噪声、模型复杂度	收集数据、正则化、早停
验证异常好，部署却差	重复样本、未来信息、分布变化	重新设计拆分与评估流程
损失为 NaN	学习率、数值范围、无效输入	降低学习率、检查数据和计算

5.13 一个排错顺序，比盲目修改模型更重要

当代码报出 shape 错误时，先写出“输入 → 层 → 输出”的形状表。比如模型第一层是 `nn.Linear(4,5)`，但实际输入是 `(3,6)`：最后一维 6 与 `in_features=4` 不匹配，因此报错。不要把 3 改成 4，除非你真的想改变样本数。正确修复通常要检查：数据实际有多少特征，或者模型定义是否写错。

遇到分类损失的类型错误时，打印 `labels.dtype`；遇到 CPU/GPU 混用时，打印 `x.device` 和 `next(model.parameters()).device`；遇到训练没效果时，先检查 `loss` 有没有变化、权重梯度是不是 `None`、`optimizer.step()` 有没有被执行。

第 5 章综合作业

使用 `nn.Sequential(nn.Linear(4,5), nn.ReLU(), nn.Linear(5,2))`，输入随机张量 `(3,4)`。先在纸上写出每一步形状，再运行 `print(model(x).shape)` 验证。接着打印两个线性层的权重与偏置形状，

解释为什么第一层是 `(5,4)` 而不是 `(4,5)`。最后回答：如果将输入改成 `(10,4)`，参数数量会变吗？答案：不会，只是批次中样本数变化。

阅读自检与答案

1. `loss.backward()` 会更新权重吗？不会，它只计算梯度。
2. `model.eval()` 会启动验证循环吗？不会，它只切换模型模式。
3. `nn.Linear(4,2)` 对 `(3,4)` 的输入输出是什么？`(3,2)`。
4. 为什么分类的 `CrossEntropyLoss` 输入一般是 logits？因为它内部以数值稳定的方式组合相关运算。
5. 为什么验证阶段常写 `with torch.no_grad()`？为了避免构建不必要的梯度计算图，节省计算与内存。

第06章｜计算机视觉：让计算机读懂像素

6.1 图片在计算机里是什么

计算机视觉（Computer Vision，CV） 研究让机器从图像、视频中提取信息的方法。对计算机来说，一张彩色图片通常是一组数字。如果使用 RGB 三个颜色通道，一张高 64、宽 64 的图可以表示为 `(64,64,3)`：最后的 3 分别对应红、绿、蓝。

图像任务不只有分类。**图像分类（Image Classification）** 问“图里大致是什么”；**目标检测（Object Detection）** 问“里面有哪些物体、分别在哪里”；**语义分割（Semantic Segmentation）** 问“每个像素属于什么类别”。后面三章会分别学习相关方法。

6.2 传统视觉与学习型视觉

在深度学习普及之前，人们使用边缘检测、颜色阈值、特征匹配等方法分析图像。**OpenCV** 就是常见的计算机视觉工具库。比如，灰度化把三个颜色通道转成一个通道；边缘检测则寻找亮度变化明显的位置。

```
import cv2                          # 导入 OpenCV 模块；安装包名通常是 opencv-python
image = cv2.imread('photo.jpg')     # 从文件读取图片；如果路径无效可能得到 None
gray = cv2.cvtColor(                # cvtColor 用于在不同颜色表示之间转换
    image, cv2.COLOR_BGR2GRAY       # BGR2GRAY 表示从 OpenCV 常用的 BGR 顺序转换为灰度图
)
print(gray.shape)                   # 二维灰度图通常具有（高度，宽度）的形状
```

这段代码假定当前文件夹里确实有一张 `photo.jpg`。如果读取失败，应先检查 `image is None`，而不是继续执行颜色转换。

停下来想一想：一张 `(100,200,3)` 的彩色图，大约有多少个通道数值？灰度化后它的形状通常变成什么？

6.3 一张图片到底有多少个数字

假设一张 RGB 彩色照片高 224 像素、宽 224 像素，每个像素有红绿蓝三个通道（Channel），那么在常见的 HWC 排列方式中，形状是 `(224,224,3)`，元素总数是 `224*224*3=150528`。若每个通道用一个 8 位无符号整数表示，就需要约 150528 字节保存这些原始通道值（不包含文件头，也不考虑压缩）。JPEG 文件往往更小，是因为使用压缩编码；不要把文件大小和张量元素数量混为一谈。

PyTorch 中的图像张量通常采用 **NCHW** 顺序，即 `(batch, channels, height, width)`。同样的 10 张彩色图，可能写作 `(10,3,224,224)`。一些图像库的默认格式是 `(height,width,channels)`。遇到形状错误时，优先确认当前库采用哪一种排列，不能看到“3”就随意猜它表示样本数。

6.4 像素归一化为什么常见

从图片文件读出的像素可能是 0～255 的整数，但训练时常把它们除以 255，变成 0～1 范围内的浮点数。这样做通常能帮助优化，但**归一化（Normalization）** 并不是让模型瞬间变聪明，也不是只能使用这一种方式。预训练模型可能要求使用特定的均值与标准差标准化：`(pixel-mean)/std`。迁移学习时务必遵守对应权重提供者的预处理规范。

一个小实验：一张 2×2 的灰度图可以写成 `[[0,255],[128,64]]`。除以 255 后，大致变成 `[[0,1],[0.502,0.251]]`，形状仍是 `(2,2)`。**数值范围改变，不意味着维度也改变。** 这和 ReLU 只改变数值、不改变形状的道理有相似之处。

6.5 人工设计特征与自动学习特征

经典图像处理可以使用阈值、边缘、颜色直方图等人工设计特征。如果你只想识别“是否出现大面积红色”，颜色阈值可能很高效；如果你要识别多种姿态、光照和背景下的猫，单靠简单阈值就很难。CNN 的价值之一是把多个局部计算层组合起来，从数据中学习适合任务的特征。

边缘检测（Edge Detection） 关注相邻像素是否出现明显变化；**模糊（Blur）** 则常将邻域像素进行平均以削弱噪声。两者都能使用小型卷积核实现，但核的数值及是否学习得到不同。别把所有“卷积运算”都等同于“卷积神经网络”：前者是计算操作，后者是包含可训练层的模型结构。

6.6 用一个安全的示例理解 OpenCV

```
import cv2                                      # 导入 OpenCV 的 Python 模块
image = cv2.imread('sample.jpg')                # 从磁盘读文件；路径不对会返回 None
if image is None:                               # is None 表示检查读取是否失败
    raise FileNotFoundError('找不到 sample.jpg')  # raise 主动抛出明确错误
height, width, channels = image.shape           # 将三个维度分别赋给变量
print(height, width, channels)                  # 展示高、宽和通道数
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY) # 将 BGR 彩色图变成灰度图
print(gray.shape)                               # 灰度图通常只有高、宽两个维度
```

代码执行前，工作目录必须真的存在 `sample.jpg`。OpenCV 常将彩色图读为 BGR 顺序，而不是常见的 RGB。若后续与使用 RGB 的工具混用，颜色可能异常。图像处理的第一步往往不是套模型，而是确认图片确实读到了、通道顺序是什么、像素值范围是多少。

章节实践

手算一张 32×32 的 RGB 图片一共有多少元素；再写出 5 张图作为 PyTorch 输入时的形状。答案：单张 `32*32*3=3072` 个通道值；批量形状通常是 `(5,3,32,32)`。

第07章｜卷积神经网络：在图片上寻找局部模式

7.1 为什么不能一直使用普通全连接层

假设图片为 `224×224×3`，展开后有 150,528 个特征。如果直接全连接到 1,000 个神经元，就需要大约 1.5 亿个权重（还不算偏置），参数量非常大。而且这样做还弱化了像素之间的空间邻近关系。

卷积神经网络（Convolutional Neural Network，CNN） 的核心想法是：让小型滤波器（Kernel，也称卷积核）在图像上滑动，检测边缘、纹理等局部模式。相同滤波器在不同位置重复使用参数，这叫**参数共享（Parameter Sharing）**。

7.2 一个小型卷积核

想象一张只有 5×5 的灰度图。我们用一个 3×3 的卷积核扫过它，每次只看一个 3×3 小区域，执行对应元素相乘求和，就得到输出中的一个值。

不填充（Padding=0）、步长（Stride=1）时，5×5 图像经过 3×3 卷积核，输出的空间大小为 3×3。这和“矩阵乘法中维度为什么变化”一样，都需要认真跟踪形状，只是计算规则不同。

7.3 用 PyTorch 看形状

```
import torch                          # 导入 PyTorch
from torch import nn                  # 导入神经网络层模块
x = torch.randn(2, 3, 32, 32)         # 2 张彩色图片，每张 3 通道、高 32、宽 32
conv = nn.Conv2d(                     # Conv2d 表示二维卷积层
    in_channels=3,                    # 输入通道数为 3（RGB）
    out_channels=8,                   # 输出 8 张特征图，相当于使用 8 组卷积核
    kernel_size=3,                    # 卷积核空间大小是 3×3
    padding=1                         # 四周填充 1，步长默认 1，因此保持空间大小
)
y = conv(x)                           # 把输入张量交给卷积层计算
print(y.shape)                        # 输出为 (2, 8, 32, 32)
```

这里的四个维度依次代表：批量大小（Batch Size）、通道数（Channels）、高度（Height）、宽度（Width），简写为 **NCHW**。不要把 `(2,8,32,32)` 当作“2 行 8 列的矩阵”，因为它是四维张量。

7.4 池化与特征层次

池化（Pooling） 常用来压缩空间尺寸。例如 2×2 最大池化会在每个小区域取最大值，典型情况下把 32×32 压缩为 16×16。网络前面的层可能学边缘，深层逐渐组合出形状和物体部件。实际模型还会使用归一化、残差连接等技巧，但先理解卷积与通道数即可。

停下来想一想：为什么经过卷积后，批量大小仍然是 2？上例中改变 `out_channels=16`，哪些维度会改变？

7.5 卷积为什么比全连接更适合处理图片

以 64×64 的 RGB 图片为例，展平后长度是 `64*64*3=12288`。如果直接用全连接层产生 100 个输出，仅权重就有 `12288*100=1,228,800` 个。若使用一个 `3×3` 的卷积层，输入 3 通道、输出 16 通道，则权重数量是 `16*3*3*3=432`，再加 16 个偏置，共 448 个。两者不是完全相同的模型，也不能仅凭参数少就判定更好，但这个例子说明局部连接和**权重共享（Weight Sharing）**可以显著降低参数量。

卷积核（Kernel）像一个小窗口，在图片上滑动。对每个位置，它取出窗口里的像素与卷积核权重逐个相乘并求和，再得到一个新的输出位置。许多位置使用**同一套**权重，所以模型会在全图中寻找类似的局部模式，例如边缘或纹理。

7.6 输出高宽怎样计算：Padding 和 Stride

对于一维空间尺寸，可以先记住公式：

`out = floor((input + 2*padding - kernel_size)/stride) + 1`（这里假设膨胀系数 dilation=1）。二维图片的高与宽分别计算。**步幅（Stride）**表示每次滑动几格；**填充（Padding）**表示在边缘添加几圈额外值，通常用 0。

例如输入宽度 8，卷积核宽 3，步幅 1，填充 0，输出宽度为 `(8-3)/1+1=6`。若填充 1，则变为 `(8+2-3)/1+1=8`。若输入为 8、核 3、步幅 2、填充 1，结果 `floor(7/2)+1=4`。你不必一口气背完公式，先画出核可以放在哪几个位置，理解公式是把位置数数出来的结果。

7.7 逐步跟踪一个 CNN

```
import torch                                  # 导入 PyTorch
from torch import nn                          # 导入网络层组件
model = nn.Sequential(                        # 顺序连接下面几层
    nn.Conv2d(3, 8, kernel_size=3, padding=1), # 3 输入通道，8 输出通道；空间大小保持
    nn.ReLU(),                                # 对每个元素应用非线性激活
    nn.MaxPool2d(kernel_size=2),              # 每 2x2 区域取最大值，常将高宽减半
    nn.Flatten(),                             # 保留 batch 维，把其他维度展成一维
    nn.Linear(8 * 16 * 16, 10)                # 8 个 16x16 特征图变成 10 类得分
)
x = torch.randn(4, 3, 32, 32)                # 4 张 RGB 的 32x32 随机图片
print(model(x).shape)                         # 输出 (4,10)，每张图 10 个类别得分
```

形状变化应写成：`(4,3,32,32)` → `(4,8,32,32)` → `(4,8,32,32)` → `(4,8,16,16)` → `(4,2048)` → `(4,10)`。这里 `Flatten()` 不是丢掉数据，而是把后面维度重新排成一条长向量，元素总数不变。

7.8 为什么池化并非万能

最大池化（Max Pooling） 在一个局部区域取最大值，能够减少空间大小，并让模型对某些细微位置变化不那么敏感。但它也可能抹掉精确位置信息。对于目标检测和分割，空间位置非常重要，因此架构设计不能一味反复池化。现代 CNN 也常采用有步幅的卷积、全局平均池化等其他方法。

阅读检查

1. `Conv2d(3,8,3,padding=1)` 对 32×32 图片输出多少通道？答：8。
2. 输入 `(4,3,32,32)` 中的 4 是卷积核数吗？答：不是，是批次样本数量。

3. 池化前 `(4,8,32,32)`，执行 2×2、步幅 2 的最大池化后通常变什么？答：`(4,8,16,16)`。

第08章｜迁移学习：站在已有模型的肩膀上

8.1 训练模型并不总要从零开始

迁移学习（Transfer Learning） 指把在某项任务上学到的知识迁移到新任务。想象一个已经见过大量常见物体图片的模型，它可能学会了识别边缘、纹理、颜色组合。现在你的任务是区分几种植物叶片，完全从零训练可能浪费数据和时间。

一个典型办法是使用**预训练模型（Pre-trained Model）**。它的前半部分当作通用特征提取器，最后替换为适用于新类别的分类头（Classification Head）。

8.2 冻结与微调

特征提取（Feature Extraction） 通常冻结原有参数，只训练新加的分类头；**微调（Fine-tuning）** 会让部分或全部预训练参数继续更新，从而更适应新任务。两者没有绝对优劣：新数据量少时，先冻结再逐步微调常是一种稳妥起点。

为了避免数据泄漏（Data Leakage），划分训练集与验证集应早于许多依赖数据统计的预处理。所有测试集结果都应该尽量模拟模型遇到未知数据时的表现。

8.3 数据增强、学习率、早停

数据增强（Data Augmentation） 可以把图片进行合理的翻转、裁剪、亮度调整，增加模型见到的变化；但不能破坏标签语义，例如识别数字“6”和“9”时随意旋转就可能改变类别。**学习率（Learning Rate）** 控制每次参数更新步幅，太大可能震荡，太小可能学习很慢。**早停（Early Stopping）** 在验证集效果不再改善时停止训练，以减少过拟合风险。

停下来想一想：一个模型在训练集上表现越来越好，但验证集越来越差，是继续增加训练轮次，还是考虑早停与正则化?

8.4 什么东西从预训练模型里转移过来

在 ImageNet 等大型数据上训练的图像网络，早期层可能学到边缘、颜色对比和纹理；更深的层学习复杂的视觉组合。当你的目标任务数据不多时，直接从头随机训练全部参数可能很困难。**迁移学习（Transfer Learning）** 让你从已有的参数起步，而不是假设所有任务都必须从零学习。

典型过程是：加载预训练权重 → 替换最后的分类头 → 冻结一部分特征提取层 → 只训练新分类头 → 如有必要，以较小学习率微调更多

层。假设旧模型最后输出 1000 类，而你要区分猫、狗两类，就必须让最终输出变成 2 类。别忘了输入预处理也需要与旧模型匹配。

8.5 冻结参数到底冻结了什么

在 PyTorch 中，设置 `parameter.requires_grad = False` 的含义是，不为这些参数计算并存储通常用于更新的梯度。它不会把这个参数从网络中删除，也不会让经过它的前向计算消失。优化器应当只管理计划训练的参数，或至少确保被冻结参数不会被更新。

```
import torch                                  # 导入 PyTorch
from torch import nn                          # 导入模型层
feature_extractor = nn.Linear(4, 8)          # 示例特征层：4 维变 8 维
head = nn.Linear(8, 2)                       # 新分类头：8 维变 2 类
for parameter in feature_extractor.parameters(): # 遍历特征层参数
    parameter.requires_grad = False          # 冻结这些参数的梯度计算
model = nn.Sequential(feature_extractor, nn.ReLU(), head) # 串联成完整模型
trainable = [p for p in model.parameters() if p.requires_grad] # 只挑可训练参数
optimizer = torch.optim.Adam(trainable, lr=0.001) # Adam 根据梯度更新可训练参数
print(model(torch.randn(3, 4)).shape)        # 3 个样本，各得到 2 个分数
```

这里特意没有下载网络权重，旨在让你看懂“冻结”这个语法。真实迁移学习还需要加载一个经过训练的特征提取器；随机初始化后立即冻结，并不能获得迁移学习的优势。

8.6 数据增强不是凭空创造新标签

数据增强（Data Augmentation） 通过合理变换已有样本，在训练过程中创造不同外观。例如猫的照片水平翻转后仍然是猫；但手写数字 6 翻转或旋转后不一定保持原来语义。增强方式必须由任务决定。验证

集和测试集通常只做确定性的基本预处理，不应随机增强到每次评估答案都变来变去。

对小型数据集，一项有效的实验往往比盲目更换模型架构更值得做：保持数据划分不变，比较“仅训练分类头”和“解冻最后一个阶段微调”的效果，同时记录每次训练使用的随机种子、学习率和预处理。这样你才能知道性能变化来自哪里。

小练习

1. 为什么预训练模型的最后一层可能需要替换？答：原任务类别数与新任务通常不同。
2. 冻结层就不用前向计算了吗？答：不是，仍要产生特征供后面使用。
3. 为什么不能对验证集做随机裁剪后随意比较实验？答：评估数据发生变化，会影响可比性。

第09章｜自编码器与 VAE：学习压缩信息

9.1 为什么要先压缩，再还原

自编码器（Autoencoder） 由编码器（Encoder）与解码器（Decoder）组成。编码器把输入映射为较小或受约束的潜在表示

（Latent Representation），解码器再尝试重建原始输入。训练时通常比较重建结果和原始输入的差别。

例如，把一张简单图像压缩成十几个数字，再从这些数字还原图片。模型必须学习什么信息最有助于重建。潜在表示可以用于降维、表征学习、异常检测等，但不是每个自编码器都会自动学到人类容易理解的语义。

9.2 VAE 为什么不只是普通自编码器

变分自编码器（Variational Autoencoder，VAE） 不直接给出一个固定的潜在向量，而是学习潜在变量的分布参数，再从中采样。训练目标通常包括**重建损失（Reconstruction Loss）**和一个与先验分布相关的 **KL 散度（Kullback–Leibler Divergence）** 项。

这样做有助于形成较连续、较规整的潜在空间。给定新的潜在变量样本，解码器就有机会产生此前没有出现过的图片。这里也体现了概率论的重要性：标准正态分布不仅是数学练习，还是许多生成模型的组成部分。

停下来想一想：普通自编码器侧重“把输入还原好”，而 VAE 为什么还关心潜在空间的概率分布？

9.3 先掌握自编码器的输入与输出

自编码器（Autoencoder，AE） 最常见的训练目标是让输出尽量还原输入。它有两段：编码器（Encoder）把输入压缩成**潜在表示**

（Latent Representation）；解码器（Decoder）根据潜在表示重建原始输入。一个简单结构可以写成 `4 → 2 → 4`。这不是说输出“预测另外四个标签”，而是说输出尝试恢复最初的四维数据。

假设四个输入数字存在冗余，例如总是满足 `x3≈x1+x2`、`x4≈x1-x2`。理论上两个维度包含的信息可能足以近似恢复四个维度。自编码器可以学习这样的压缩关系。但如果瓶颈太小，重要信息可能丢失；如果网络过强，可能学会一些不理想的记忆方式。

9.4 重建误差有什么用

在异常检测（Anomaly Detection）中，有时只用正常样本训练自编码器。当一个异常样本的结构明显不同，模型可能难以很好重建它，从而出现较大的重建误差。注意这里只是一个可尝试的思路，不是保证：过强的自编码器也可能把异常重建得很好，因此高重建误差不一定代表异常，低重建误差也不一定代表正常。

对于图片，自编码器会比较重建图片和原图的像素或感知特征差异；对于表格数据，可能比较各数值特征。但不同量纲的特征需要妥善处理，否则一个数值范围极大的列会主导损失。

9.5 VAE 为什么会谈到概率分布

变分自编码器（Variational Autoencoder，VAE） 与普通 AE 的重要差别，是编码器通常不只输出一个确定的潜在点，而是输出潜在分布的参数，例如高斯分布的均值与方差相关参数。模型从这个分布取样，再用解码器生成重建结果。训练目标一般包含两部分：重建损失，以及

让潜在分布接近选定先验的 **KL 散度（Kullback–Leibler Divergence）** 项。

“接近先验”使潜在空间更适合采样与生成。所谓**重参数化技巧（Reparameterization Trick）**，就是把采样写成 `z = mu + sigma*epsilon`，其中 `epsilon` 来自与模型参数独立的标准正态分布。这种表达使梯度可以沿着 `mu`、`sigma` 传播。

一道形状题

编码器把 `(batch,784)` 变成 `(batch,16)`；解码器再把 `(batch,16)` 变回 `(batch,784)`。如果 batch=32，各阶段形状是多少？答：`(32,784)` → `(32,16)` → `(32,784)`。这与 `784→16→784` 的结构一致。

第10章｜GAN 与风格迁移：两种生成思路

10.1 一个生成者，一个鉴别者

生成对抗网络（Generative Adversarial Network，GAN） 包含两个模块：生成器（Generator）试图制造看起来真实的数据；判别器（Discriminator）尝试区分真实样本和生成样本。两者在交替训练中互相推动。

你可以想象一位画家与一位鉴赏家：画家想画得更逼真，鉴赏家不断指出哪里不自然。但请注意，GAN 训练有时不稳定，可能出现模式崩溃（Mode Collapse）：生成器反复产出少量类似样本，看起来会生成，却缺乏多样性。

10.2 风格迁移不是“简单套滤镜”

神经风格迁移（Neural Style Transfer） 尝试保留内容图的主要结构，同时匹配风格图的纹理、笔触或统计特征。常见思想是用训练好的视觉网络提取不同层的特征，再寻找一张同时满足“内容接近”和“风格接近”的新图像。

风格迁移与 GAN 都属于生成相关技术，但它们的训练目标和生成机制并不相同。学习时最好先把问题分清：我是在学习数据分布，还是在优化一张图的内容与风格表示？

停下来想一想：如果 GAN 只能生成非常相似的十张人脸，却无法覆盖更多变化，它可能遇到了什么问题？

10.3 GAN 的对抗，到底在竞争什么

生成对抗网络（Generative Adversarial Network，GAN） 包含生成器（Generator）和判别器（Discriminator）。生成器从随机噪声产生候选图片，判别器试图区分真实图片与生成图片；两者在交替训练中相互影响。你可以把生成器想象为“努力模仿某类图像的画家”，判别器想象为“不断改进辨别能力的评审”。但数学上两者都只是可训练模型，不是人类意义上的理解或意图。

一个关键细节：判别器越强并不总是越好。如果生成器几乎得不到有用的训练信号，学习可能停滞。GAN 训练常常比普通分类训练更不稳定，可能出现**模式崩溃（Mode Collapse）**：生成器反复产生少数相似样本，看上去清晰却缺乏多样性。

10.4 生成样本“逼真”不代表模型学得全面

评价一个图像生成模型，要同时考虑质量、多样性和安全性。假设数据集里有红、蓝、绿三种玩具车，生成器只会做红色玩具车，它的单张图像也许很好看，却没有覆盖数据分布。相反，一个模型生成很多样式但图片模糊，同样存在问题。这就是质量与覆盖度要分别观察的原因。

现代图片生成方法还有扩散模型（Diffusion Models）等，与 GAN 的训练方式不同。掌握 GAN 的价值在于理解生成式学习的另一条路线，不必误以为今天所有图片生成系统都采用 GAN。

10.5 风格迁移不是 GAN 的同义词

神经风格迁移（Neural Style Transfer）的目标通常是保留一张内容图像的整体结构，同时模仿另一张图片的纹理、颜色或笔触风格。经典优化式风格迁移可使用预训练 CNN 的特征表示来定义内容损失和风格损失，并迭代调整输出图像的像素。它不要求像标准 GAN 那样有生成器与判别器对抗训练。

课堂思考：如果风格迁移把文字照片中的字形扭曲了，画面也许“很艺术”，但文字可读性下降。这提醒我们设计损失函数时要明确任务要求：你想保留什么，又允许改变什么？

本章检查

1. GAN 中随机噪声通常先喂给谁？答：生成器。
2. “反复生成同一种图”属于哪类常见问题？答：模式崩溃。
3. 风格迁移必须使用 GAN 吗？答：不必须。

第11章｜目标检测：物体是什么，又在哪里

11.1 分类不够用时怎么办

一张街景图里可能同时有行人、自行车、汽车。图像分类通常只给图整体一个或多个类别，却不一定指出位置。**目标检测（Object Detection）** 除了预测类别，还需要输出**边界框（Bounding Box）**，通常用坐标 `(x_min, y_min, x_max, y_max)` 表示一个矩形区域。

一个检测模型会生成多个候选框以及对应类别置信度。模型需要决定哪些框真正代表物体，哪些是重复或背景。

11.2 IoU 是什么

交并比（Intersection over Union，IoU） 衡量两个框有多大程度重合：用“交集面积 ÷ 并集面积”计算。完全一样时 IoU 为 1；没有

交集时为 0。它可用于衡量预测框准确程度，也常用于去掉相互重叠的候选框。

非极大值抑制（Non-Maximum Suppression，NMS） 是常见后处理步骤：当多个框表示同一对象时，依据分数和重叠程度进行筛选。不过不同检测模型的内部实现可能各不相同。

11.3 任务设计也决定模型质量

一套可靠的检测系统不仅需要准确率，还要考虑漏检成本、遮挡情况、推理速度和标注一致性。例如在医学或安全相关场景里，漏掉重要目标可能比多报几个目标更严重。这些取舍不能只由单个分数决定。

停下来想一想： 目标检测输出一个“狗”标签为什么还不够？你还需要哪一类信息？

11.4 一张图上有三只猫，分类为什么不够

图像分类可能只返回“猫”，但目标检测（Object Detection）需要知道猫分别在哪里，因此输出不仅包括类别，还包括**边界框（Bounding Box）**。一张图中可能有零个、一个或许多个物体，输出数量可变，比只输出单一类别更复杂。

常见的框表示有两种：`(x_min,y_min,x_max,y_max)` 表示左上与右下坐标；`(center_x,center_y,width,height)` 表示中心与宽高。两种格式可以互相转换，但坐标可能以像素为单位，也可能归一化到 0～1。初学者做检测项目最容易出的错误之一，就是把坐标格式与尺寸单位搞混。

11.5 手算一次 IoU

交并比（Intersection over Union，IoU） 用两个框重叠部分的面积除以两个框并集的面积。设框 A 范围是 `(0,0)` 到 `(4,4)`，面积 16；框 B 是 `(2,2)` 到 `(6,6)`，面积也是 16。交集是从 `(2,2)` 到 `(4,4)` 的 2×2 正方形，面积 4；并集面积 `16+16-4=28`，所以 `IoU=4/28≈0.143`。

IoU 取值通常介于 0 与 1。两个框完全重合时是 1；完全不重叠时是 0。IoU 可以用于匹配预测框与真实框，但“多少 IoU 算成功”由评估协议决定，不能脱离具体任务与阈值解释。

11.6 非极大值抑制在解决什么问题

非极大值抑制（Non-Maximum Suppression，NMS） 用来处理同一个物体被预测出多个非常相似框的情形。简化过程是：按置信分数排序，保留最高分的框，然后删除与其重叠程度超过阈值的其他候选框，继续处理剩余框。这样可以减少重复检测。

但 NMS 并不是“把图片中挨得近的物体都删掉”。如果两个不同物体的框高度重叠，简单 NMS 可能误删真实目标，因而阈值和算法变体要结合场景设计。对于密集人群、相互遮挡的场景尤其如此。

自测

1. 两个框完全相同，IoU 是多少？答：1。
2. 分类模型输出“狗”却不指出位置，它完成了检测吗？答：没有，缺少定位信息。

3. 为什么 NMS 常用于检测后处理？答：去除对同一目标的重复高重叠预测。

第12章｜语义分割与 U-Net：给每个像素分类

12.1 一张图不只一个答案

语义分割（Semantic Segmentation） 希望为图像中的每一个像素预测类别。例如在道路场景中，每个像素都可能是道路、行人、汽车、天空或建筑物。它的输出不再只是一个标签或一个矩形框，而是一张与原图空间尺寸对应的类别图（Mask）。

语义分割与**实例分割（Instance Segmentation）** 不同：语义分割可能把两个人都标为“人”，实例分割还需要区分“第一个人”和“第二个人”。

12.2 U-Net 的 U 形结构

U-Net 的一侧通过下采样（Downsampling）扩大感受野、提取高级特征；另一侧通过上采样（Upsampling）逐步恢复空间分辨率。两侧之间使用**跳跃连接（Skip Connection）**，让高分辨率的浅层信息帮助恢复细节。

这就像先把一张地图缩小，看清整个城市的结构，再慢慢放大，同时保留原来的街道边界信息。U-Net 最初在生物医学图像分割中很有代表性，也被用在其他图像分割任务中。

12.3 为什么准确率不能只看像素比例

假设医学影像里 99% 的像素都属于背景，一个“所有像素都是背景”的模型也能达到 99% 的像素准确率，但它没有找到真正关心的小区域。因此还会考虑 Dice 系数、IoU 等指标。好的指标必须与任务风险相符合。

停下来想一想：为什么 U-Net 需要从编码器把较浅层的图像信息传给解码器？

12.4 分割、检测和分类的输出有何不同

假设一张图里有一只狗和一把椅子：分类回答“有哪些类别”；检测给出狗和椅子的矩形位置；**语义分割（Semantic Segmentation）** 给每个像素赋一个类别，如背景、狗、椅子。模型输出因此可能具有 `(batch,classes,height,width)` 的形状。若图片大小为 128×128，共 3 类，batch=4，那么输出 logits 可以是 `(4,3,128,128)`。

还有**实例分割（Instance Segmentation）**：如果画面里有两只狗，它不仅知道哪些像素是狗，还要区分“第一只狗”和“第二只狗”。语义分割通常不区分同类的不同个体。任务目标不同，数据标注的难度也不同。

12.5 U-Net 中的“跳跃连接”到底救了什么

经典 U-Net 架构有一条向下的编码路径：逐步压缩空间分辨率、扩大感受野；还有一条向上的解码路径：逐步恢复空间大小。中间的**跳跃连接（Skip Connection）** 把编码路径中高分辨率的特征与解码阶段对应尺寸的特征结合，帮助找回边界细节。

设编码路径某处有 `(batch,32,64,64)` 的特征图，解码路径经过上采样后有 `(batch,32,64,64)`。若沿通道维拼接，可能得到 `(batch,64,64,64)`；随后用卷积融合信息。需要注意具体 U-Net 变体在通道数与尺寸对齐上可能有不同细节；**拼接（Concatenation）** 和逐元素相加不是一个操作。

12.6 为什么分割的总体准确率容易骗人

假设一张图 99% 是背景，只有 1% 是病灶。模型永远预测“背景”，像素准确率仍接近 99%，但病灶一个也没找到。因此分割常用 **Dice 系数（Dice Coefficient）** 和 IoU 等指标评估目标区域的重叠程度。

Dice 的一个常见定义是 `2*|A∩B|/(|A|+|B|)`。如果预测区域有 10 个像素，真实区域有 10 个，重叠 8 个，那么 Dice 是 `16/20=0.8`；IoU 是 `8/(10+10-8)=8/12≈0.667`。相同的预测与真值，Dice 与 IoU 不是一个数字，不能互换。

练习

1. 有 5 张 64×64 图片，要对每个像素预测 4 类，输出 logits 典型形状是什么？答：`(5,4,64,64)`。

2. 同一图片中两只狗，语义分割是否必须给它们不同编号？答：不必；实例分割才强调区分个体。
3. 为什么小目标分割不能只看像素准确率？答：背景像素占多数，错误的“全部背景”预测也可能得到高分。

第13章｜文本表示：计算机怎样“看见”一句话

13.1 文字首先要变成数字

自然语言处理（Natural Language Processing，NLP） 研究如何让计算机分析、生成人类语言。神经网络需要数字作为输入，所以必须把句子变成可计算的表示（Representation）。

最直观的方式是**词袋模型（Bag of Words，BoW）**。先列出词表（Vocabulary），再记录每个词在一句话中出现了几次。比如词表是“苹果、香蕉、好吃”，句子“苹果好吃，苹果好吃”就可以表示为 `[2,0,2]`。

这种做法简单，却丢失了词序。“狗追人”和“人追狗”可能有相同词袋表示，但含义显然不同。

13.2 TF-IDF 想修正什么问题

常见词不一定能区分文档。比如“的”“是”等词在许多文章里都出现。**TF-IDF（Term Frequency–Inverse Document Frequency）** 把“词在当前文档中有多重要”与“这个词在整个文档集合中有多常见”结合起来。

直觉上，某词在当前文章出现多次、却很少出现在其他文章，它可能很能说明文章主题。TF-IDF 是一种稀疏文本表示方法，在搜索、文档分类和基线模型中仍然实用。

13.3 一个小的 Python 示例

```
from collections import Counter              # 导入 Counter，用于自动统计元素出现次数
words = ['苹果', '好吃', '苹果']             # 列表 list，保存三个词，其中苹果出现两次
counts = Counter(words)                      # Counter 生成“词 → 次数”的映射
print(counts['苹果'])                        # 使用方括号按键取值，这里会显示 2
```

这里最值得理解的是 `from ... import ...` 与 `counts['苹果']`。前者从模块中取出一个工具，后者用字典键读取统计结果。这些是 Python 通用知识。

停下来想一想：为什么词袋模型无法有效区分“甲喜欢乙”和“乙喜欢甲”？

13.4 文本不能直接喂给神经网络

神经网络做的是数字运算，而人写的是文字。把句子输入模型之前，必须决定如何拆分文本、如何表示每个词，以及是否要保留位置。**分词（Tokenization）** 把文本切成 Token；Token 不一定是完整单词，也可能是子词或单个字符。中文没有天然的空格分隔词，需要特别留意切分策略。不同 tokenizer 可能将同一句话切成完全不同的片段。

最朴素的方法叫**词袋模型（Bag of Words，BoW）**：先建立词表，再数每句话中每个词出现多少次。它可以用于简单的文档分类，却通常不

保留词语顺序。例如“猫追狗”和“狗追猫”如果使用同样的词表，计数向量可能完全一样，尽管两个句子的意思不同。

13.5 手算一个词袋向量

假设只有三句已分好词的文本：`我 爱 数学`、`我 爱 代码`、`代码 很 有趣`。词表按固定顺序排成 `[我,爱,数学,代码,很,有趣]`。第一句变为 `[1,1,1,0,0,0]`，第二句为 `[1,1,0,1,0,0]`，第三句为 `[0,0,0,1,1,1]`。每一列对应一个词，每一行对应一份文档。

这有点像你熟悉的矩阵 shape：如果共有 3 个文档、6 个词，文本表示矩阵是 `(3,6)`。之后你可以将它输入 `nn.Linear(6,2)`，得到 `(3,2)`，表示每个文档有 2 个分类分数。核心矩阵规则完全相同，只是输入特征从“身高、体重”变成“词语计数”。

13.6 为什么 TF-IDF 往往比裸计数更合理

如果一个词在所有文档里都频繁出现，它未必有很强的区分能力。**TF-IDF（Term Frequency–Inverse Document Frequency）** 将“词在当前文档中的出现强度”和“词在整个语料库中的稀有程度”结合起来。在不同具体实现中，词频是否归一化、逆文档频率是否平滑，定义可能略有不同，因此不用死记一个唯一公式。

例如 100 篇文章里，“今天”出现 90 篇，而“反向传播”只出现 3 篇。在讨论机器学习的文章里，“反向传播”可能更能表明主题。TF-IDF 就是在尝试赋予这种相对罕见、但在本文中突出的词更大权重。它仍然无法真正理解上下文，也无法自动知道同义词的关系。

13.7 写一个不依赖第三方库的词袋表示

```
# 列表保存三条已经按空格分开的短句
sentences = ['我 爱 数学', '我 爱 代码', '代码 很 有趣']
# set 用来去重，再用 sorted 使词表顺序固定，方便复现
vocab = sorted(set(word for s in sentences for word in s.split()))
print(vocab)                                  # 查看组成词表的所有词
vectors = []                                  # 一个空列表，用来保存每句的向量
for sentence in sentences:                   # 遍历每一条短句
    words = sentence.split()                  # split 按空格拆成词列表
    counts = [words.count(w) for w in vocab]# 对词表中每个词计数
    vectors.append(counts)                   # 把这一句的计数添加进总表
print(vectors)                                # 显示三个句子的词袋向量
```

这段代码中的列表推导式 `[words.count(w) for w in vocab]` 可以从右往左读：对词表中每个 `w`，计算它在当前 `words` 里出现的次数，把所有次数按顺序放进新列表。暂时看不懂也没关系，可以把这一行换成普通 `for` 循环慢慢写。

自检

1. 3 份文档、词表大小 5000，BoW 输入通常是什么形状？答：`(3,5000)`。
2. TF-IDF 保留了词序吗？答：通常不保留。
3. 使用词袋时，“机器学习”和“学习机器”为什么可能难区分？答：出现的词集合和次数可能相同，缺乏顺序信息。

第14章｜词向量：让“含义相近”有数学位置

14.1 从独热编码走向稠密向量

独热编码（One-hot Encoding） 给词表里的每个词一个专属位置。词表有 10,000 个词时，每个词对应长度 10,000 的向量，其中只有一个位置为 1。它简单但维度很大，也不会自然体现“猫”和“狗”都属于动物这一关系。

词嵌入（Word Embedding） 则把词映射到长度较短的稠密向量，比如 100 维或 300 维。模型通过训练，让在相似上下文中出现的词具有某些相近表示。词嵌入既是数据压缩，也是在学习表示空间。

14.2 Word2Vec 的核心直觉

Word2Vec 的著名想法是：词的使用环境包含语义信息。你可以让模型根据中心词预测周围词，也可以根据周围词预测中心词。训练后保留下来的权重矩阵就成为词向量表。

GloVe（Global Vectors） 则利用词共同出现的统计结构来学习向量。两者路径不同，目标都不是单纯给词编号，而是学习有用的连续表示。

14.3 向量相似，不等于完全理解

两个词向量的夹角小，往往表示某种分布式相似性。但词汇的意义依赖语境：“银行”可以指金融机构，也可能在不同语言中出现其他用法。传统静态词向量通常一个词对应一个向量，处理一词多义的能力有限；后来的上下文化语言模型对此有所改善。

停下来想一想：为什么长度 100 的稠密向量不一定比长度 10,000 的独热向量损失更多“有用信息”？

14.4 什么叫“语义上接近”的向量

词向量（Word Embedding）把一个离散词映射为多个浮点数字。例如“猫”可能对应 `[0.2,0.7,-0.1]`，“狗”可能对应 `[0.3,0.6,-0.05]`。数字本身一般没有直接的人工解释，但它们的整体位置关系可能反映词在训练语料中的使用环境。

余弦相似度（Cosine Similarity）经常用来比较两个向量方向是否接近。其定义是“点积除以两向量的长度乘积”。设 `u=(1,0)`，`v=(0,1)`，它们的点积为 0，余弦相似度也为 0；若 `u=(1,0)`，`v=(2,0)`，余弦相似度为 1，因为方向一致。方向相似不代表语义完全等价，它只是模型给出的向量关系。

14.5 Word2Vec 的学习信号从哪里来

Word2Vec 有两种经典思路。CBOW（Continuous Bag of Words）利用周围词预测中心词；Skip-gram 用中心词预测周围词。两种训练任务迫使模型把经常出现在相似上下文中的词学到某种相近表示。比如

“我喝了一杯____”的上下文常与“茶”“咖啡”“水”等词共同出现，它们可能在向量空间里具有某种相似结构。

但“同一上下文中出现”不一定意味着常识上的同义。例如“热”和“冷”都可能用于“天气很____”，于是一些词向量会把反义词放得比较近。**相似上下文** 与 **相同意思** 不能画等号。

14.6 从 one-hot 到 embedding 的形状变化

假设词表大小是 10000，词向量维度是 64。使用 one-hot（独热向量）时，每个词是长度 10000 的大向量，只有一个位置为 1；使用 `nn.Embedding(10000,64)` 后，每个词编号可以查到长度 64 的可训练向量。若输入词编号张量形状是 `(batch=3,seq_len=5)`，输出就是 `(3,5,64)`。

```
import torch                                 # 导入 PyTorch
from torch import nn                         # 导入神经网络组件
embedding = nn.Embedding(10000, 64)          # 10000 个词，每个词映射到 64 维
ids = torch.tensor([[5, 2, 7], [3, 0, 4]]) # 2 条序列，每条有 3 个词编号
vectors = embedding(ids)                     # 将词编号映射成可训练向量
print(vectors.shape)                         # 输出 (2,3,64)
```

编号不是“越大表示词越重要”，仅仅是词表中的标识。`nn.Embedding` 的参数本质上可以理解为一个二维查找表，形状 `(10000,64)`，训练时其中的向量会调整。

14.7 词向量能学到社会偏见吗

可以。因为训练资料来自真实社会，语料中的刻板印象、偏见和历史不平衡也可能进入向量关系。这意味着“模型发现了某种关联”不等于“这种关联应该用于决策”。尤其在招聘、教育和司法等场景，要检查训练数据代表性、评估群体差异，并避免把相关性误作人的能力判断。

练习

1. 词编号矩阵 `(4,6)` 进入 `Embedding(5000,32)` 输出是什么？答：`(4,6,32)`。
2. 相似度高能证明两词同义吗？答：不能，可能只是出现在类似上下文。
3. 为什么 one-hot 不适合直接表示海量词表中的语义关系？答：稀疏、维度大，且任意不同词的 one-hot 彼此正交，无法天然表达接近程度。

第15章｜语言模型：预测下一个词

15.1 什么是语言模型

语言模型（Language Model） 为语言序列赋予概率。直观地说，给出“今天的天气很”，模型可能认为“好”“冷”“热”等词具有不同概率。训练目标之一是根据前面的上下文预测后续内容。

不同模型有不同的分词单位（Token），一个 token 不一定对应一个汉字、单词或完整词语。不要把 token 与自然语言里的“词”简单画等号。

15.2 为什么嵌入层很重要

假设词表大小为 5,000，嵌入维度是 32，那么一个**嵌入层（Embedding Layer）**可以看作形状 `(5000,32)` 的参数表。输入不是完整的 5,000 维独热向量，而是词对应的整数索引，模型根据索引取出长度 32 的向量。

```
import torch                          # 导入 PyTorch
from torch import nn                  # 导入神经网络模块
embedding = nn.Embedding(             # 创建可训练的词嵌入表
    num_embeddings=5,                 # 词表大小：索引可以为 0、1、2、3、4
    embedding_dim=3                   # 每个词被映射成长度为 3 的向量
)
ids = torch.tensor([0, 2, 4])       # 三个 token 的索引，使用整数张量
vectors = embedding(ids)              # 依索引查表，获得每个 token 的向量
print(vectors.shape)                  # 输出 (3,3)：3 个 token，每个向量有 3 个数字
```

`nn.Embedding` 与 `nn.Linear` 都有可训练参数，但计算机制不同：一个主要根据索引查表，另一个执行线性变换。不要因为两者都有 weight 就认为完全相同。

15.3 从文本输入到预测

一个简单语言模型通常依次完成：把文本切为 token、映射成整数索引、得到嵌入向量、利用序列模型融合上下文、输出每个候选 token 的分数。用 Softmax 可以将一组分数转为和为 1 的概率分布。

停下来想一想：为什么语言模型的输出类别数常常与词表大小有关?

15.4 语言模型到底在预测什么

语言模型（Language Model） 学习语言序列的概率关系。最容易理解的一种任务是根据已有前缀预测下一个 Token。例如在“今天天气很”之后，模型可能给“好”“冷”“热”不同概率。最终输出并非直接神奇地变成一句话，而是先产生一个关于词表的分数向量，再经适当变换得到概率分布，选择下一个 Token。

如果词表有 8000 个 Token、batch 里有 2 段文本、每段长度 5，那么模型的预测 logits 可以是 `(2,5,8000)`：每个位置都给词表中 8000 个候选 Token 一个分数。训练时常将相邻位置错开：输入是某段序列的前几项，目标是各位置真正的下一项。

15.5 为什么词向量与语言模型经常一起出现

模型不能直接处理字符串，需要先把 Token 变成整数编号，再用 Embedding 变成连续向量。接着序列模型考虑上下文，输出每个位置的隐藏表示，最后的线性层把隐藏维度变成词表大小的 logits。

示例：输入编号 `(2,5)` → Embedding 产生 `(2,5,32)` → 序列处理层保持 `(2,5,32)` → 输出层 `Linear(32,8000)` 得到 `(2,5,8000)`。这里最后一维 8000 不是“有 8000 个样本”，而是每个位置的候选词数量。

15.6 困惑度与交叉熵的直觉

语言模型常使用**交叉熵损失（Cross-Entropy Loss）**：如果真实下一个词得到很低的预测概率，损失就较大。**困惑度（Perplexity）** 在常见定义中可以理解为平均负对数似然取指数，因此当平均损失降低时，困惑度也降低。它反映模型预测语料 Token 的能力，但其数值会受到语料分布和分词方式影响，不能不加说明地横向比较不同 tokenizer 的模型。

请区分“语言模型能预测某个词在上下文中常见”与“模型知道一个事实是真的”。语言模型可以生成流畅、符合语法但事实错误的句子。可靠事实任务仍需要来源验证。

15.7 训练数据如何变成“前一个词预测后一个词”

下面用最简单的 Python 列表演示训练样本构造，暂时不使用神经网络。

```
# 假设一句话已变成整数 Token 编号
sequence = [10, 4, 8, 6, 2]          # 每个整数只是词表中的编号
inputs = sequence[:-1]               # 切片：从头取到倒数第二个元素
labels = sequence[1:]                # 切片：从第二个元素取到结尾
print(inputs)                        # 得到 [10,4,8,6]
print(labels)                        # 得到 [4,8,6,2]
```

这里 `inputs[0]=10` 对应 `labels[0]=4`，代表看见第一个 Token 后预测第二个。因果语言模型训练时通常还要使用注意力掩码，避免模型在预测当前位置时偷看未来内容。

小测

1. 词表 1000，模型输出 `(2,7,1000)`，最后的 1000 是什么？答：每个位置的词表候选分数。
2. 困惑度低就一定不会“编造事实”吗？答：不一定，两者衡量的是不同能力。
3. 为什么预测下一词的目标序列要错开一个位置？答：为每个输入前缀构造对应的真实下一 Token。

第16章｜循环神经网络：让信息沿着时间流动

16.1 普通前馈网络为什么不够

如果任务是识别一段文字或预测时间序列，每个位置往往依赖之前的内容。**循环神经网络（Recurrent Neural Network，RNN）** 在处理当前输入时，还使用上一时刻的隐藏状态（Hidden State），因此能沿着序列传递信息。

可以想象你正在逐字阅读一句话：当读到最后一个字时，大脑还保留着前面若干字的信息。RNN 就试图以向量的方式模拟这种连续状态。

16.2 序列输入的三个维度

在 PyTorch 中，如果 `batch_first=True`，常见 RNN 输入形状为：

`(batch_size, sequence_length, input_size)`

例如 `(4,10,8)` 表示 4 条序列，每条有 10 个时间步，每个时间步用 8 个数字表示。它与图像的 NCHW 格式完全不同，**看懂 shape 必须先看数据语义**。

```
import torch                          # 导入 PyTorch
from torch import nn                  # 导入神经网络层
rnn = nn.RNN(                         # 创建一个最基础的循环神经网络层
    input_size=8,                     # 每个时间步的输入向量长度是 8
    hidden_size=5,                    # 每个时间步的隐藏状态长度是 5
    batch_first=True                  # 输入按（批量，时间步，特征）排列
)
x = torch.randn(4, 10, 8)             # 4 条序列，每条 10 个时间步，每步 8 维
output, hidden = rnn(x)               # 得到所有时间步的输出和最后隐藏状态
print(output.shape)                   # (4,10,5)，每个时间步输出 5 维
print(hidden.shape)                   # (1,4,5)，一层、单向 RNN 的最终隐藏状态
```

16.3 长距离信息为什么困难

基础 RNN 在很长的序列上容易遇到**梯度消失（Vanishing Gradient）**或梯度爆炸（Exploding Gradient）问题，导致难以记住遥远信息。**LSTM（Long Short-Term Memory）**和 **GRU（Gated**

Recurrent Unit） 通过门控机制改进信息保留能力。它们曾广泛用于序列任务，现在也依然适用于一些场景。

停下来想一想：在上面的例子里，把 `hidden_size` 改为 7，`output.shape` 应该如何变化？

16.4 RNN 的隐藏状态在记住什么

循环神经网络（Recurrent Neural Network，RNN） 逐步处理序列。在时刻 `t`，它接收当前输入 `x_t` 和上一步隐藏状态 `h_(t-1)`，计算新的 `h_t`。隐藏状态相当于一个不断更新的摘要，帮助模型综合已经读过的信息。它不是人类记忆，也不保证保存全部历史。

例如读“因为昨天下雨，所以今天路面很____”。如果模型只看最后一个“很”，难以判断下一个词；保留前面“下雨”“路面”等线索，会更容易预测“湿”。RNN 通过逐步更新状态来携带上下文，但长句里的早期信息可能在重复变换中逐渐衰减。

16.5 RNN 的输入形状为什么是三维

文本批处理经常有三个关键维度：`batch` 表示一次多少条序列；`seq_len` 表示每条序列多少个时间步；`input_size` 表示每个时间步输入向量有多少特征。使用 `batch_first=True` 的 PyTorch RNN，输入通常是 `(batch,seq_len,input_size)`。

```
import torch                                  # 导入 PyTorch
from torch import nn                         # 导入神经网络模块
rnn = nn.RNN(input_size=8, hidden_size=5, batch_first=True) # 每步 8 维，隐藏 5 维
x = torch.randn(3, 4, 8)                    # 3 条序列，每条 4 步，每步 8 个特征
output, last_hidden = rnn(x)                # 前者是各步输出，后者是最终隐藏状态
print(output.shape)                          # (3,4,5)，每个时间步 5 个隐藏特征
print(last_hidden.shape)                     # (1,3,5)，1 层、3 条序列、5 个特征
```

这里有两个重点：`output` 保留所有时间步的结果；`last_hidden` 保留每一层在最后时间步的隐藏状态。上例只定义一层 RNN，因此 `last_hidden` 第一维为 1。若设置双向网络或多层网络，形状会相应变化。

16.6 什么是梯度消失与梯度爆炸

训练 RNN 时，需要把最终损失对较早时间步参数的影响传回去，称为**随时间反向传播（Backpropagation Through Time，BPTT）**。因为会反复乘许多导数，如果这些因子平均偏小，早期梯度可能趋近于 0，模型难以学到长距离依赖；如果平均偏大，梯度可能变得非常大，使训练不稳定。

LSTM（Long Short-Term Memory）与**GRU（Gated Recurrent Unit）**引入门控机制，更灵活地决定保留或更新哪些信息，以缓解部分长距离依赖问题。但它们不是永不遗忘的完美记忆器，仍需要认真训练和评估。

读懂形状

`nn.RNN(input_size=10, hidden_size=6, batch_first=True)` 输入 `(2,7,10)`，输出 `output` 是 `(2,7,6)`；单层单向默认的 `h_n` 是 `(1,2,6)`。将 `batch_first` 改为 False 时，`output` 的时间与批次维顺序会交换；`h_n` 的形状约定则不因这个参数改变。

第17章｜生成式 RNN：一个字接着一个字

17.1 生成不是一次输出整篇文章

早期字符级生成模型常用 RNN：先给一个起始字符，然后模型预测下一个字符的概率分布；从中选出或采样一个字符；把新字符送回模型；不断重复，形成序列。这里的关键是：生成出来的内容会影响后续预测。

17.2 温度（Temperature）控制什么

生成时可以通过**温度参数（Temperature）** 调整采样分布。温度较低时，概率大的选项更容易被选中，输出往往较保守；温度较高时，低概率选项也更可能出现，结果可能更有变化，也可能更混乱。

不过，温度不是“模型创造力”的精确度量。它只是影响概率分布采样的一种机制，内容质量仍取决于模型、训练数据、上下文与解码方式。

17.3 为什么生成结果可能跑偏

训练时模型常常看真实文本的历史上下文；生成时则必须依赖自己的历史输出。一旦某步生成了不理想的内容，后面可能继续受到影响。这种训练与使用场景的差别，是序列生成需要面对的问题之一。

停下来想一想：如果生成模型一直挑概率最大的下一个字符，它的输出会更稳定还是更随机？

17.4 为什么一个模型能连续写出很多字

生成式 RNN 的循环可以概括为：先有一个起始序列，让模型计算下一 Token 的概率，选出一个 Token，把它追加到序列尾部，再用更新后的上下文继续预测。每次只往前走一步，很多步连起来才形成一段文字。

如果每次永远选概率最大的候选，称为**贪心解码（Greedy Decoding）**。它可预测且简单，但可能导致文字重复。**采样（Sampling）**则从概率分布中随机抽取下一项，使生成有更多变化。温度（Temperature）通过缩放 logits 改变概率分布的尖锐程度：通常温度更低让高概率候选更集中，温度更高让分布更平缓，但过高可能产生混乱输出。

17.5 top-k 与停止条件有什么用

Top-k Sampling 先只保留概率最高的 k 个候选，再在其中采样；**Top-p / Nucleus Sampling** 则按概率从高到低累积到某个阈值，保

留动态数量的候选。它们都在试图平衡稳定性与多样性，并不是“参数越大越好”。

真正的生成程序还要考虑停止：遇到结束 Token 就停止，或者达到最大输出长度就停止。否则即便模型每步都能给出下一个 Token，也不代表它会自发结束。生产应用还需要内容安全、重复控制和运行成本限制。

17.6 Teacher Forcing 为什么只在训练时常见

在训练阶段，我们通常知道真实的下一个 Token。因此可以选择在每一步把真实 Token 送给模型，让模型更容易学习局部条件分布，这叫**教师强制（Teacher Forcing）**。生成阶段没有可供参考的“真实未来文本”，只能把模型自己的预测继续送回去。

这种训练与生成的差异会造成**暴露偏差（Exposure Bias）**：模型训练时习惯看到正确前缀，但实际生成一旦选错一个 Token，后面就需要在不完美的前缀上继续生成。理解它能帮助你解释为什么生成文本可能越写越偏题。

17.7 只用列表模拟自回归生成逻辑

```
# 这个例子只演示循环逻辑，并非已训练的语言模型
sequence = ['今天']                          # 初始文本序列
choices = {'今天': '天气', '天气': '不错', '不错': '。'} # 假想的预测表
for step in range(3):                      # 最多循环 3 次
    last = sequence[-1]                    # -1 索引表示列表最后一个元素
    next_token = choices.get(last, '。')    # dict.get 查不到时使用默认句号
    sequence.append(next_token)            # 将新 Token 加入已有序列
    if next_token == '。':                  # 如果预测到结束符
        break                              # break 提前结束循环
print(''.join(sequence))                   # join 将列表拼接为完整文本
```

这段程序完全靠人写好的词典做下一步选择，并不算训练了 RNN。它只是帮助你看懂生成器最外层的控制逻辑。真正模型会在每步输出大量候选的分数，而不是查固定字典。

小练习

1. 温度趋近 0 时，生成往往更随机还是更确定？答：通常更偏向高概率候选。
2. Teacher Forcing 为什么不能直接原样用于没有真实答案的生成阶段？答：此时没有真实下一 Token 可供输入。
3. 为什么要设置最大生成长度？答：避免无限持续生成并控制资源消耗。

第18章｜Transformer：不用循环，也能建模上下文

18.1 自注意力到底在“注意”什么

Transformer 的核心机制之一是**自注意力（Self-Attention）**。一句话里的每个位置，可以根据其他位置的信息来更新自己的表示。不同于基础 RNN 按时间顺序逐步传递信息，注意力层能够在同一次计算中考虑多个位置之间的关系。

例如句子“这只猫趴在椅子上，它正在睡觉”，理解“它”时，模型可以利用前面“猫”的信息。但注意力权重不应被直接当作模型完整的因果解释。

18.2 Query、Key 和 Value

自注意力常把输入向量投影成三组向量：**查询（Query，Q）**、**键（Key，K）**和**值（Value，V）**。可以把 Q 与 K 的相似程度看作“当前位置应该参考谁”的一部分依据，再用相关权重加权汇总 V。

对于单个注意力头，常见公式为：

```
Attention(Q,K,V) = softmax(Q K^T / sqrt(d_k)) V
```

公式看着吓人，其实拆开很简单：算匹配分数、做缩放、转成权重、加权取信息。`sqrt(d_k)` 用于控制分数尺度，让训练更稳定。

18.3 位置编码为什么重要

如果模型只看词之间的关系却不关心顺序，“小狗咬人”和“人咬小狗”就难以区分。**位置编码（Positional Encoding）**或其他位置表示机制，用于让模型知道 token 的顺序与相对位置。不同 Transformer 架构采取的实现方式可以不同。

18.4 BERT 与生成式模型的区别

BERT（Bidirectional Encoder Representations from Transformers）以编码器式表示学习著名，可结合左右上下文建立词语表示；它的典型预训练任务包括掩码语言建模（Masked Language Modeling）。许多现代文本生成模型则采用自回归方式预测后续 token。

需要记住：**Transformer 是一类架构，不等于每个 Transformer 都是聊天机器人，也不等于 BERT 一定按聊天模型方式生成文字**。

停下来想一想：为什么一个不关心词序的模型很难区分“我帮助你”和“你帮助我”？

18.5 自注意力究竟比循环多了什么

在 RNN 中，信息主要沿序列逐步传播；在**自注意力（Self-Attention）**中，一个位置可以直接根据当前层所有允许看到的位置计算加权组合。因此远距离词语之间不必只通过很多连续时间步传递信息。Transformer 的好处不仅是效果，还包括训练时更容易对整段序列进行并行矩阵计算。

注意力权重（Attention Weights） 可以粗略理解为：“在更新当前位置的表示时，对其他位置的数值信息分配多大比重”。但它并不直接等价于人类可解释的注意力或因果解释，不能仅凭某个权重高就说“模型作出判断的真正原因就是它”。

18.6 Q、K、V 与矩阵乘法的联系

Transformer 用当前隐藏表示经过可学习的线性变换得到 Query（查询向量）、Key（键向量）和 Value（值向量）。一个 Query 与各 Key 计算相似程度，再经过 Softmax 得到一组权重，最后用这些权重对 Value 进行加权求和。

常见缩放点积注意力写作：`Attention(Q,K,V)=softmax(Q @ K.T / sqrt(d_k)) @ V`。这里 `@` 就是你熟悉的矩阵乘法。若一条序列长 4，每个 Query/Key 向量维度为 8，单头计算中 `Q`、`K` 都可以看作 `(4,8)`；于是 `Q @ K.T` 是 `(4,4)`，它表示四个位置分别与另外四个位置的匹配分数。若 `V` 是 `(4,6)`，最后权重矩阵 `(4,4)` 乘 V 得到 `(4,6)`。

这说明中间维度 8 是匹配特征维度，输出中出现的 4×4 则是“位置与位置”的配对。换成批量计算时，还要加入 batch 与 head 维度，但基本矩阵关系不变。

18.7 为什么要除以 sqrt(d_k)

如果 Query 和 Key 的每个分量都有类似的尺度，维度较大时，点积的数值波动可能变大。未经缩放就送入 Softmax，有时会让概率分布过度尖锐，影响训练。除以 `sqrt(d_k)` 是一个常见的缩放设计，用来控制

点积分数的尺度。它不是普遍适用的“任何矩阵乘法都应该除以根号维度”的规则，只针对相关注意力形式。

Softmax 将一组实数分数变成非负且总和为 1 的权重。比如分数 `[0,0]` 会变成 `[0.5,0.5]`；分数 `[2,0]` 会让第一个位置得到更大权重。可以把它看作“分配关注比例”，但最终的 Value 加权和仍然是一个向量，不是一个单独类别。

18.8 掩码为什么不可缺少

对自回归语言模型，如果在预测第 3 个 Token 时能直接读到第 4 个真实 Token，就等于偷看答案。**因果掩码（Causal Mask）** 让当前位置只能访问自己及之前的位置。另有**填充掩码（Padding Mask）**，用于忽略为了凑齐相同序列长度而额外补上的位置。

注意这两种掩码用途不同：因果掩码控制“能不能看未来”，填充掩码控制“哪些位置只是占位符”。在 Transformer 中，正确设置掩码与正确设置 `shape` 同样重要。

18.9 BERT 与自回归生成模型的差异

BERT 典型预训练任务之一是遮蔽语言建模：把输入中的一部分 Token 遮住，利用双向上下文恢复它们。这种表示方式适合分类、实体识别等理解任务。自回归语言模型则主要按时间方向预测后续 Token，适合从左到右生成文本。两者都属于 Transformer 家族，却不是“只差模型大小”。

练习与解答

1. 序列长 7、单头的 Q 与 K 都是 `(7,16)`，`Q @ K.T` 的形状？答：`(7,7)`。
2. 如果注意力权重是 `(7,7)`，V 是 `(7,32)`，输出？答：`(7,32)`。
3. 生成模型为什么需要因果掩码？答：阻止训练时偷看真实未来 Token。

第19章｜命名实体识别：从句子中找出“谁、哪、何时”

19.1 把文字标成实体

命名实体识别（Named Entity Recognition，NER） 希望从文本中标出人名、机构、地名、日期等实体。例如句子：“小李于 2025 年加入北京的一家公司。” 我们可能把“小李”识别为人物，把“2025年”识别为时间，把“北京”识别为地点。

它与整句分类不同：分类任务输出一个整体标签，NER 通常需要给每个 token 或词片段分配标签。

19.2 BIO 标注方式

一个常见办法是用 **BIO 标签**：`B-` 表示实体开始（Beginning），`I-` 表示实体内部（Inside），`O` 表示不属于任何目标实体（Outside）。例如，两个 token 组成的机构名可以被标成 `B-ORG`、`I-ORG`。

实际训练要注意字符、单词与 tokenizer 切出的 token 未必一一对应。标注与 token 对齐错误，会直接伤害模型质量。NER 对语言、领域及标注标准也敏感。

19.3 从模型到可靠应用

如果要在合同、新闻或聊天记录中自动提取信息，需要同时设计后处理规则、人名歧义消解、未知实体处理与隐私保护。NER 的预测结果是统计判断，不应该未经审核就当作确定事实。

停下来想一想：NER 与“判断整篇新闻是体育类还是财经类”的任务，预测单位有什么不同?

19.4 命名实体识别和普通文本分类的区别

命名实体识别（Named Entity Recognition，NER）不只给整句话贴一个标签，而是识别文本中的具体片段及其类别。例如“李明于周二前往上海”，可以识别“李明”为 PERSON（人名）、“周二”为 DATE（日期或时间表达）、“上海”为 LOCATION（地点）。具体实体类别取决于任务数据标注规范；不同语料可能使用不同标签体系。

文本分类通常输出一个或几个文档级标签；NER 则需要定位实体从哪个字符或 Token 开始、到哪里结束。它与图像检测“物体是什么以及在哪里”有某种相似性，只不过这里的位置发生在文本序列中。

19.5 BIO 标注：每个字母到底在说什么

BIO 是一种序列标注规范：B 表示一个实体的开头（Begin），I 表示同一实体的内部（Inside），O 表示不属于当前任何实体（Outside）。例如序列“李 / 明 / 去 / 上 / 海”，可以标为 `B-PER / I-PER / O / B-LOC / I-LOC`。

如果模型把“李”标成 B-PER，却把“明”标成 O，预测实体的边界就错了。一个句子里实体类型都猜对了，但边界多切或少切一个字，按严格实体级评估仍可能算错。因此 NER 的评价通常使用实体级 Precision、Recall 和 F1，而不只是逐 Token 的准确率。

19.6 当一个实体被切成多个子词

现代 tokenizer 可能把一个词切成多个 subword，NER 的原始标签却是在词级或字符级定义的。训练时就需要明确标签如何与 Token 对齐：只给第一个子词打标签、还是给全部子词赋标签，不同实现可能使用不同约定。这不是小小的代码细节，而是可能直接影响训练质量与评估正确性的因素。

检查题

1. 句子“王老师住在北京”，如果“北京”被标为一个地点，BIO 可以怎么写？答：`北=B-LOC，京=I-LOC`。
2. NER 中为何边界重要？答：任务不只判断类别，还要准确指出原文片段。
3. 为什么分词方式变化可能影响 NER？答：实体标签与模型 Token 位置需要重新对齐。

第20章｜大语言模型与提示：认识今天常见的 AI

20.1 什么是大语言模型

大语言模型（Large Language Model，LLM） 通常在大量文本等数据上通过预训练学习语言中的统计模式与知识关联。许多现代 LLM 使用 Transformer 架构，并在此基础上通过指令微调、偏好优化等方式改善交互行为。

模型生成文本并不是先检索到一个“标准答案”再整段复制；在典型自回归生成中，它持续根据上下文计算下一个 token 的概率，再逐步生成后续内容。这也解释了为什么模型有时能给出流畅却错误的回答。

20.2 提示词不是魔法咒语

提示（Prompt） 可以包含任务、背景、约束、示例以及期望输出格式。例如让模型改写一封邮件，你最好说明读者是谁、语气怎样、必须保留什么事实，而不是只写“改好一点”。

少样本学习（Few-shot Learning） 在提示中给几个输入与期望输出的示例，帮助模型把握任务模式。它与重新训练模型参数不同：在典型提示使用中，参数没有因为这几个示例而被更新。

20.3 降低幻觉风险的办法

幻觉（Hallucination） 常指模型产生貌似可信但缺乏依据的内容。对日期、价格、法规、医学信息、数学结果等，你应优先核对可靠来源，并让模型明确表达不确定性。**检索增强生成（Retrieval-Augmented Generation，RAG）** 能把相关外部资料放入回答上下文，但检索结果也可能错误，仍需判断质量。

20.4 从 AI 学习者的角度看 LLM

学过矩阵、激活函数、损失、梯度和注意力之后，你已经拥有理解大模型技术文档所需的关键积木。真正理解一个模型，不是认识所有新名词，而是持续追问三个问题：输入是什么形状？中间怎样计算？训练目标是什么？

停下来想一想：在提示中给两个例子，为什么不等于通过梯度下降重新训练了大语言模型？

20.5 大语言模型与前面小语言模型的连续性

大语言模型（Large Language Model，LLM） 可以先视为在大量文本及其他可能的数据上训练的大规模神经网络。它并不是完全脱离前面章节的新魔法：Token 编号、Embedding、Transformer、矩阵乘法、损失函数、梯度下降这些概念依然在发挥作用。只是模型规模、训练数据、训练流程与推理策略更复杂。

用户输入一段文字后，系统会先分词，将 Token 转换成编号，计算当前位置的表示和输出分数，再决定如何生成下一个 Token。重复很多次，生成一段回答。聊天式模型可能还经过指令微调或偏好对齐训练，所以能更好地遵循对话形式。**会遵循指令** 与 **所有回答都正确** 是两回事。

20.6 上下文窗口不是永久记忆

模型一次推理可处理的 Token 数有上限，叫**上下文窗口（Context Window）**。它可包含用户问题、系统指令、之前的对话以及参考材料，但并不等于模型永久储存了这些内容。如果材料超出窗口，有些部分可能无法直接参与当前计算。把重要条件写得清楚、有结构，可以减少遗漏。

检索增强生成（Retrieval-Augmented Generation，RAG） 是在回答前查找相关外部资料，再把查到的内容提供给模型使用的一种系统设计。它可以改善对更新知识和特定文档的利用，但检索错了、来源过期或模型误读资料，仍会产生错误。RAG 并不自动保证事实正确。

20.7 什么样的提示更容易得到可检查的回答

一份清晰的提示（Prompt）通常说明四件事：任务目标、输入材料、限制条件、期望输出格式。例如你可以说：“请根据下面这段教材解释 `nn.Linear(4,5)`；假设读者只学过 Python 变量和列表；每个术语附英文；先用数字举例，再逐行解释代码；不要跳过 shape 的变化。”这比一句“讲一下这个代码”更容易评估是否满足要求。

但再好的提示也不能把未知事实变成已知事实。需要实时、精确或高风险的信息时，应使用可信来源核验。回答里出现了引用，不代表引文本身一定正确；还需要检查引用能否真正支持结论。

20.8 Few-shot 和模型微调不是一件事

少样本提示（Few-shot Prompting） 指在输入里给模型若干示例，帮助它理解当前任务。此时通常没有更新模型权重。**微调（Fine-tuning）** 则通过额外训练真正调整部分或全部可训练参数。把几条例子粘贴进提示词，不等同于训练神经网络。

假设你要把学习笔记分类为“数学”“代码”“阅读”，给出三条带类别的示例，然后让模型处理新笔记，这属于少样本提示。若收集大量已分类笔记，构造损失并运行优化器更新模型参数，才属于典型监督微调流程。

20.9 为什么评估要先于“上线”

对一个问答模型，至少需要关注事实正确率、是否遵循指令、是否泄漏隐私、是否过度自信、对不同输入表达的稳健性以及推理成本。假设它

90% 的时候能正确解释 PyTorch，另外 10% 把梯度符号讲反了，如果学生不去验证，很容易积累错误知识。因此学习时可以把模型当辅导工具，但涉及实际代码应运行检查，数学结论应手算验证。

本章练习

1. 在提示词里给三条例子，参数会自动改变吗？答：通常不会。
2. RAG 为何仍可能回答错误？答：检索内容可能不相关或不可靠，模型也可能错误使用资料。
3. 了解 LLM 为什么仍需要学矩阵乘法？答：线性层和注意力等核心计算都依赖矩阵及张量运算。

第21章｜遗传算法：用“选择、变异”寻找好方案

21.1 不用梯度，也能优化吗

可以。**遗传算法（Genetic Algorithm，GA）**是受生物进化过程启发的优化方法。把候选解看成“个体”，按照**适应度（Fitness）**评估表现，再经过**选择（Selection）**、**交叉（Crossover）**和**变异（Mutation）**产生下一代。

例如要为一次配送寻找较短路线，候选路线就是个体，路线长度可以成为评价依据。较好的候选解可能更有机会参与生成新解，但仍保留一定随机探索，避免过早陷入单一方案。

21.2 与梯度下降有什么不同

梯度下降（Gradient Descent）利用可微目标函数的梯度决定调整方向；遗传算法不要求能够计算梯度，但可能需要评估大量候选解。它们都在寻找更好的参数或方案，却适合不同的优化问题。

遗传算法不保证一定找到全局最优，也不是“模拟自然进化就一定比其他算法更聪明”。实际效果依赖编码方式、评价函数、种群规模与搜索预算。

停下来想一想：如果一个问题的目标函数根本无法求导，遗传算法可能有哪些优势？

21.3 用一个玩具问题理解“适应度”

假设我们要寻找一个整数 x，使函数 `score(x)=-(x-7)^2` 尽可能大。最优解显然是 x=7，因为此时得分为 0，其他位置得分是负数。但我们假装事先不知道答案，尝试用**遗传算法（Genetic Algorithm，GA）**搜索。

首先随机产生多个候选 x，称为**种群（Population）**。对每个候选计算分数，分数作为**适应度（Fitness）**的一种定义。接着优先选择适应度较高的候选，再通过**交叉（Crossover）**和**变异（Mutation）**产生新候选。如此重复多代（Generations），希望越来越接近优秀解。

这里的“遗传”只是算法类比，不表示模型真的具有生物学特性。适应度如何定义尤其重要：如果目标设错了，算法可能很努力地优化一个不符合实际需求的指标。

21.4 一次纸上选择、交叉与变异

假设种群有四个候选 `[2,5,9,12]`，适应度分别为 `-25,-4,-4,-25`，因此 5 和 9 相对好。你可以让好的候选更容易被选中，再通过某种规则产生新的候选，例如平均得到 7，再以小概率随机改变它。整数问题里这只是示意，真实遗传算法会按编码形式定义合适的交叉与变异操作。

与梯度下降相比，遗传算法不一定需要导数，可以在非光滑、离散搜索空间中工作。但它通常需要评估许多候选，计算成本也可能很高。不能说“进化算法一定比神经网络更智能”，也不能说“因为不求导就一定更快”。

21.5 写一个最小的进化搜索

```
import random                                    # 导入 Python 的随机数模块
random.seed(3)                                   # 固定随机种子便于重复实验
population = [random.randint(0, 14) for _ in range(8)] # 生成 8 个候选整数

def fitness(x):                                  # def 定义函数，x 是候选解
    return -((x - 7) ** 2)                       # 越接近 7，分数越高

for generation in range(12):                     # 一共演化 12 代
    population.sort(key=fitness, reverse=True) # 按得分从高到低排序
    parents = population[:4]                     # 保留表现较好的前 4 个
    children = parents.copy()                    # copy 复制列表，避免直接修改父代
    while len(children) < 8:                     # 未达到 8 个新候选就继续产生
        a, b = random.sample(parents, 2)         # 从父代不重复地抽取两个候选
        child = round((a + b) / 2)               # 简化交叉：取两个整数的均值
        child += random.choice([-1, 0, 1])       # 简化变异：随机扰动 -1、0 或 1
        children.append(max(0, min(14, child))) # 限制候选落在 0 到 14 之间
    population = children                        # 新一代取代旧一代

best = max(population, key=fitness)              # 找到最后一代适应度最大的候选
print(best, fitness(best))                       # 打印最优候选与它的分数
```

这只是用于理解流程的教学示例，父代选择和交叉方式都非常简化，不能拿它的性能去概括所有遗传算法。好处是你能清楚看见“候选解—评分—选择—产生后代”的循环。

练习

1. 如果一个搜索问题只能判断方案好坏、很难求导，进化搜索为什么可能合适？答：不依赖解析梯度。
2. 如果变异概率一直很高，会有什么问题？答：可能难以保留已学到的优秀结构，搜索接近随机。

3. 如果适应度只衡量速度，完全不考虑安全，会有什么风险？答：算法可能找到速度很快但不安全的方案。

第22章｜强化学习：在尝试和反馈中改进行为

22.1 与监督学习最根本的差异

强化学习（Reinforcement Learning，RL） 研究智能体（Agent）如何通过与环境（Environment）互动来改进决策。每一步通常涉及：观察状态（State）、选择动作（Action）、接收奖励（Reward）、进入下一个状态。

监督学习常有一对一的标准标签；强化学习更像玩游戏：你选择移动方向，可能过了几步才知道这个选择是好是坏。因此，**延迟奖励（Delayed Reward）** 与长期规划很重要。

22.2 探索与利用

如果你已经知道向左移动经常得分，是否应该永远向左？未必。也许向右的长期回报更高，只是你尚未探索过。**探索（Exploration）** 是尝试尚不确定的行为，**利用（Exploitation）** 是倾向选择已知较好的行为。好的策略需要权衡两者。

22.3 Q 值与深度强化学习

Q 值（Action-Value） 可以理解为：处于某状态并采取某动作，随后遵循某策略时预期得到的累积回报。传统表格方法记录有限状态与动作的价值；**深度 Q 网络（Deep Q-Network，DQN）** 则用神经网络近似 Q 值函数，以处理更复杂的输入。

然而，强化学习训练常受到样本效率、奖励设计、环境变化和安全探索的限制。部署前需要比“游戏里能得高分”更严格的评估。

停下来想一想：为什么某个动作的即时奖励为零，并不一定意味着它是坏动作?

22.4 监督学习与强化学习，反馈究竟哪里不同

监督学习（Supervised Learning）通常给每个样本一个正确标签，模型学习 `输入→正确输出`。**强化学习（Reinforcement Learning，RL）** 中，智能体（Agent）与环境（Environment）交互：观察状态（State），选择动作（Action），得到奖励（Reward）和新状态，再决定后续动作。

奖励不一定立刻体现动作的长期效果。下棋时，一步看似吃亏的走法可能让后面更容易获胜；导航时，绕远一小段可能避开危险路段。强化学习关注的往往是**累积回报（Return）**，不是单步即时奖励。

22.5 一张表理解 Q-learning

Q 值（Action Value，Q(s,a)） 可以理解为“从状态 s 采取动作 a，后续按某种策略行动时预期能够得到的累计奖励”。在简化 Q-learning 中，更新公式是：

```
Q_new(s,a)=Q_old(s,a) + alpha * [reward + gamma * max_a' Q(s_next,a') - Q_old(s,a)]
```

其中 `alpha` 是学习率，`gamma` 是折扣因子（Discount Factor），控制未来奖励的权重。假设旧 Q 值为 2、奖励为 1、下一个状态最大 Q 值为 5、`gamma=0.9`、`alpha=0.1`。目标是 `1+0.9*5=5.5`；误差为 `5.5-2=3.5`；新 Q 值为 `2+0.1*3.5=2.35`。

这与神经网络的权重更新有相似精神：用目标与当前估计之间的差异改进参数。但在表格 Q-learning 中被更新的是状态—动作表项；深度强化学习可能用神经网络近似 Q 函数。

22.6 探索与利用：为什么不能一直选当前最好的

利用（Exploitation） 是选择目前看来收益最高的动作；**探索（Exploration）** 是尝试不确定的动作，以便发现可能更好的策略。如果一直利用早期的偶然估计，模型可能永远不知道另一路线更好。常见的 ε-贪心（Epsilon-greedy）策略是：以概率 ε 随机探索，否则选择当前 Q 值最高的动作。

探索并非越多越好。在真实世界的医疗或自动驾驶场景，随机尝试危险动作不可接受。因此真实强化学习系统往往需要模拟器、离线数据、约束或安全保护，不能把游戏中的随机探索直接搬进高风险现场。

22.7 深度 Q 网络为什么仍然要学 PyTorch

当状态空间非常大，无法给每个状态、动作维护表格 Q 值时，可以训练神经网络，输入状态特征，输出各动作的 Q 值估计。这就是**深度 Q 网络（Deep Q-Network，DQN）**的基本思路之一。若状态特征维度为 10、可选动作数量为 4，模型可以把输入 `(batch,10)` 映射为 `(batch,4)`。

DQN 等方法还可能使用经验回放（Experience Replay）与目标网络（Target Network）缓解训练不稳定。我们现在不需要展开全部工程细节，只需要认出：网络仍然在做矩阵运算，优化器仍在更新参数，区别在于训练目标来自奖励和状态转移，而不是每一步都由人提供唯一正确动作。

练习与解答

1. 监督学习是否通常有明确标签？答：是；RL 的反馈通常是奖励及环境转移。
2. `Q` 值是立即奖励本身吗？答：通常不是，它包含对未来回报的估计。
3. 为什么 ε 不能盲目设为 1 并长期不变？答：那会始终高度随机，难以稳定利用已经学到的好动作。

第23章｜多智能体系统：不止一个决策者

23.1 许多智能体一起行动

多智能体系统（Multi-Agent Systems） 研究多个具有一定自主行为能力的参与者如何共享环境、交流、合作或竞争。例如自动驾驶车辆与行人、多个机器人组成的仓库系统，以及多个软件代理分工完成任务。

单个智能体要决定“我现在怎么做”；多个智能体还必须考虑“别人会怎么做”。因此，环境可能因其他智能体的行为持续变化。

23.2 合作与竞争

在合作任务里，多个智能体可能共享总体目标，例如多台机器人共同搬运物品；竞争任务里，每个参与者可能追求不同利益。系统可以使用明确协议、集中协调、分布式策略或学习型方法。

“多智能体”并不自动意味着“多个 LLM 在群聊”。传统机器人系统、博弈论模型和分布式控制同样属于这个领域。

23.3 可信协调很重要

当一个系统由多个代理组成，错误信息、任务冲突、循环调用、权限过大和资源浪费都可能被放大。因此要明确每个代理的角色、权限、通信边界以及终止条件。

停下来想一想：为什么两台各自表现很好的机器人，在同一个狭窄走廊里仍可能发生冲突?

23.4 多个智能体一起存在，会发生什么

多智能体系统（Multi-Agent System，MAS）研究多个可以观察、决策、行动的主体如何相互影响。它们可能合作、竞争，也可能同时存在合作与冲突。与单智能体相比，一个重要难点是：其他智能体也在改变策略，所以“环境”不再稳定。

举例：两个机器人共同搬运一张桌子，必须协调方向与速度；两个自动交易程序争夺相似订单，会相互影响市场状态；几个辅助代理共同写一份报告，需要明确谁搜资料、谁计算、谁审核，避免所有代理都重复做同一件事。

23.5 通信和协调如何设计

在合作场景中，一种简单方法是给角色明确职责：规划者负责分解任务，执行者完成指定步骤，审核者根据独立标准核对结果。但代理之间互相转述的信息可能被误解，错误也可能在多轮交互中放大。因此需要定义统一的数据格式、结束条件、冲突处理、日志记录与人为接管方式。

不应该假设“代理越多，结果一定越好”。多代理增加了通信成本、延迟和错误传播机会。对于只需一段简单计算的任务，让一个清晰函数直接完成往往比组织五个代理讨论更加可靠。

23.6 合作奖励与个体奖励可能冲突

假设两辆配送车都想尽快完成自己的路线，但城市总体目标是减少总拥堵。每辆车只优化自己的行驶时间，可能都挤向同一条短路，反而使所有车变慢。这是**个体目标与整体目标不一致**的例子。多智能体强化学习研究如何设计共享奖励、局部奖励和通信，使群体行动更合理。

在合作环境里，奖励设计要考虑“搭便车”：一个智能体几乎不做贡献却同样获得团队奖励。在竞争环境里，又要考虑彼此策略变化导致训练目标不稳定。多智能体系统难的并不只是把单智能体复制很多份。

小练习

1. 多个代理同时行动，为何一个代理的任务会变得更复杂？答：其他代理的策略会改变它面对的环境。
2. “所有代理一起做同样的工作”一定提高质量吗？答：不一定，可能增加重复与冲突。
3. 一个多代理系统怎样方便排错？答：保留职责、输入输出、通信记录和审核节点。

第24章｜负责任的 AI：技术之外还要考虑什么

24.1 模型准确不等于公平安全

负责任的 AI（Responsible AI） 要求开发和使用者同时考虑可靠性、安全、公平、隐私、透明度、可问责性和人类监督。一个模型总体准确率很高，却可能在某一类群体或少见场景上出错更多。

训练数据来自现实，也可能带有现实中的偏差（Bias）。如果不检查数据来源、标签质量以及不同人群和环境中的表现，模型可能复制或放大不合理差异。

24.2 透明度与可解释性

使用 AI 时，用户应该知道系统能做什么、不能做什么，以及它的输出存在什么不确定性。**可解释性（Explainability）** 研究如何理解模型预测背后的影响因素，但任何解释工具都有局限，不能把一张漂亮的热力图当作完全证明。

隐私保护（Privacy） 不只是把姓名隐藏，还涉及数据收集目的、访问权限、留存时间、敏感信息泄漏风险，以及用户是否真正同意相关处理。

24.3 给初学者的实用检查清单

在实际项目中至少追问：数据是否有权使用？标签是否可靠？哪些用户或场景更容易出错？模型给错答案会造成什么影响？是否允许人类复核？是否对不确定结果设置了合理的处理方式？

这些问题不是学完技术后才考虑的附加题，而应该贯穿项目从设计到部署的全过程。

停下来想一想：一个人脸识别系统整体准确率是 99%，为什么仍不足以证明它可以安全地用于所有场景？

24.4 为什么“准确率高”依然可能伤害某些人

假设一个模型总体准确率达到 95%，但在一个样本占比较小的人群中准确率只有 65%。总体平均指标可能掩盖群体差异。对一些高风险决策，还要分别观察不同人群的错误率、漏报率、误报率和数据代表性。**公平性（Fairness）** 没有对所有场景都唯一正确的数学定义，应结合用途、受影响人群与法律伦理要求判断。

数据可能包含历史偏见。例如历史招聘结果本身受不公平程序影响，那么用这些结果作标签训练的模型，可能把旧问题自动化。解决方法不仅是“删除敏感字段”：其他特征也可能与敏感属性高度相关。需要结合业务流程、数据收集方式和独立评估。

24.5 隐私与可解释性不是可有可无的附件

训练数据可能包含姓名、邮件、医疗记录等个人信息。**数据最小化（Data Minimization）** 要求只收集任务真正需要的信息，避免把所有可取得的数据都塞进模型。还要明确授权、用途、保留期限以及是否有权限向第三方工具传输。

可解释性（Explainability） 也应结合使用场景。比如贷款审核需要对当事人说明重要原因，可能不能只展示一个分数。但单纯画出“特征重要性条形图”并不等于提供了真实且充分的因果解释。模型、数据、决策流程需要配合才能实现有意义的说明。

24.6 把“负责任 AI”做成检查流程

假设你在开发一个 AI 学习助手。首先明确边界：它提供学习建议，不代表正式考试评分或医疗建议。接着检查输入：是否需要保存学生的真实姓名、住址、成绩？能否只保存匿名学习进度？再检查输出：模型是否会自信地给出错误代码？能否提示用户运行验证，并提供可复查的参考资料？最后设计反馈方式，让学生能报告错误并看到更正。

可用下面的风险表来组织初步检查：

风险	具体问题	防护思路
错误信息	错误公式却说得很肯定	数学推导和代码运行验证
隐私泄漏	把个人笔记公开传输	减少数据、权限控制、删除机制

风险	具体问题	防护思路
群体差异	某类学生总被低估	分组评估与人工复核
滥用	被用于欺骗或骚扰	访问控制、用途约束、审计
过度依赖	学生不再自主思考	要求给推理过程和可验证步骤

24.7 一个完整的错误报告应该写什么

若发现模型将 `nn.Linear(4,2)` 的权重形状解释成 `(4,2)`，要把错误记录成可以复现的报告：输入是什么、模型回答了什么、正确事实是什么、如何用 `print(layer.weight.shape)` 验证、错误是否影响其他解释，以及是否需要更新教程。这比一句“AI 又胡说八道”更能帮助改进系统。

负责任的 AI 不等于模型永远不会错，而是人们能够发现问题、追踪问题、限制损害并持续改进。

本章思考

1. 为什么删除人群属性不一定消除偏差？答：其他字段可能充当替代变量。
2. 为什么仅给一个准确率数字不足以说明公平？答：平均值可能掩盖重要子群体的差异。
3. 一个 AI 教学应用，最先应该保护什么？答：学生个人信息、答案可靠性以及对学习者造成的真实影响。

第25章｜扩展阅读：多模态、CLIP 与 VQGAN

25.1 当文字与图像走到一起

多模态学习（Multimodal Learning） 让模型处理多种形式的数据，比如图片和文字。任务可能是“看图回答问题”“用文字检索图片”或“根据文字生成图像”。难点在于不同模态表示方式不同，模型需要建立它们之间有用的关联。

25.2 CLIP 学习图文对齐

CLIP（Contrastive Language–Image Pre-training） 的代表性思路，是利用图像与文本配对数据进行对比学习，让匹配的图文表示更接近，不匹配的表示相对远离。这样可以用文字描述来比较图片，也可以用于开放词汇场景中的识别与检索。

25.3 VQGAN 的基本定位

VQGAN（Vector Quantized Generative Adversarial Network） 结合向量量化与对抗生成等思想，把图像转换为离散潜在代码，并通过解码器重建图像。它可以成为更复杂生成系统中的图像表示组件。不同生成架构的技术路线很多，不应把 VQGAN 当作现代图像生成的唯一方案。

读到这里，你已经沿着微软课程大纲走过了从符号 AI、神经网络、图像理解、语言模型到多智能体和 AI 伦理的整条路线。接下来真正提高水平的关键，是回到第 3～5 章，把前向、损失、梯度、参数更新以及 shape 变换真正算清楚。

停下来想一想：为什么把图片与文字映射到能相互比较的表示空间，会帮助实现“用一句文字找图片”？

25.4 多模态模型究竟把什么对齐

多模态（Multimodal） 是指模型处理不止一种信息类型，比如文字、图片、声音或视频。文字和图片的数据形式不同：文本先分词成为 Token，图片通常先切成 patch 或提取视觉特征。要判断一句话与一张图是否匹配，系统需要将它们转换成可以比较的表示。

CLIP 的典型思路是分别用图像编码器和文本编码器，把图片和描述变成同一向量空间中的向量。训练时，匹配的图文对希望相似度较高，错误配对希望相似度较低。这类训练常称为**对比学习（Contrastive Learning）**。

25.5 零样本分类为什么可以靠文本提示

假设你有一张图片，但没有针对“猫 / 狗 / 鸟”三类重新训练最后一个线性分类器。可以分别编码候选文本，如“一张猫的照片”“一张狗的照片”“一张鸟的照片”，然后计算图片向量与这些文本向量的相似程度。相似度更高的描述可作为候选预测。这就是 CLIP 风格的**零样本分类（Zero-shot Classification）**思路之一。

但这种相似度不直接等于经过良好校准的现实概率。例如某图对“猫”得分 0.28、对“狗”得分 0.25，不应直接理解为“有 28% 概率是猫”。分数尺度、候选描述方式及数据分布都会影响结果。零样本能力也存在类别偏差与领域适应性问题。

25.6 VQGAN 为什么有个 VQ

VQ 通常表示**向量量化（Vector Quantization）**。直觉上，编码器先把图像映射为连续特征，再从一个有限的向量字典（Codebook）中选择最接近的向量来表示局部信息，于是得到离散化的潜在编码。解码器据此重建图像；GAN 式对抗目标可帮助提升视觉逼真度。

这和标准 VAE 通过连续潜在分布采样的做法有所不同。你不必把每个生成模型都归纳成“输入噪声—输出图片”这一个模板：潜在变量是连续还是离散、训练目标包含什么、如何解码，都会改变模型的性质。

25.7 把本书的知识串成一张认知地图

现在可以回顾：神经网络先用矩阵乘法和激活函数把输入变成隐藏特征；图像模型让这些特征适合空间结构；文本模型让特征适合序列；Transformer 使用注意力在不同位置之间交换信息；多模态模型努力让不同媒体的表示进入可比较的空间。

虽然具体模型架构差别很大，但你反复遇到的基础只有几组：**数据与标签、向量与矩阵、损失与梯度、训练与评估、泛化与风险**。真正建立理解不是把模型名称记满，而是遇到新技术时能问出：输入是什么？输出

形状是什么？学什么参数？依据什么损失训练？用什么数据评估？在哪些条件下容易失败？

学完全书后的最终挑战

请找一段你感兴趣的 AI 模型代码，不求一口气运行成功，只做五件事：标出每个输入张量形状；找出可训练参数；说明前向计算的输出；指出损失与优化器在哪里；给出一种可能发生的数据泄漏或过拟合风险。你能独立完成这五步，就已经从“看不懂神经网络”进入“可以持续阅读新模型”的阶段。

学习专题 01｜Python 零基础补课：一行代码都不要跳

微软课程里的 Notebook 一般默认你有一点 Python 基础。但如果 `import`、`def`、`for` 和缩进还不熟，直接学神经网络容易把“没看懂语法”误当成“没理解数学”。这一专题可以放在第 3 章之前阅读，也可以遇到陌生代码时回来查。

S1.1 变量、类型、赋值到底是什么

Python 用变量名引用值。`age=20` 不是数学方程式，不是说“age 永远等于 20”，而是把整数 20 赋给变量 `age`。后面可以写 `age=21` 改变它。变量名不需要提前声明类型，但值有类型：整数 `int`、浮点数 `float`、字符串 `str`、布尔值 `bool` 等。

```
count = 3                # count 是变量名；3 是整数（int）
score = 0.75             # 带小数点的数字通常是浮点数（float）
name = 'PyTorch'         # 单引号括起来的是字符串（str）
is_ready = True          # 布尔值（bool），True 表示真
print(type(count))       # type() 用来查看一个对象的类型
count = count + 1        # 右边先算 3+1，再把 4 重新赋给 count
print(count)             # 输出 4
```

`=` 是赋值，`==` 是比较相等，`!=` 是不等于。`count += 1` 是 `count = count + 1` 的简写。`#` 开头的文字是**注释（Comment）**，Python 不执行注释内容。刚开始建议多写中文注释，帮助自己解释每行目的。

S1.2 列表与索引：从 0 开始数

Python **列表（List）** 用方括号表示，能够按顺序保存多个元素。`[10,20,30]` 有三个元素，但索引分别是 0、1、2。`values[0]` 取得第一个元素，不是第零个“有意义的数据”。负索引 `-1` 表示最后一个元素。

```
values = [10, 20, 30]        # 创建一个包含三个整数的列表
print(values[0])              # 读取第一个元素，结果是 10
print(values[-1])             # -1 表示倒数第一个元素，结果是 30
print(len(values))            # len() 统计列表长度，结果是 3
values.append(40)             # append() 把一个新值加到列表末尾
print(values[1:3])            # 切片 [起点:终点)；结果为 [20,30]
```

`[1:3]` 的右端 3 不包含在内，这是初学者最常犯的错误之一。你在语言模型章节看到的 `sequence[:-1]` 就是“取出除最后一个元素之外的所有元素”。后面还会看到 `shape[0]`：这意味着访问形状的第一个维度，一般是 batch 大小。

S1.3 for 循环、range 与 enumerate

```
for i in range(3):           # range(3) 依次产生 0、1、2
    print(i)                 # 缩进 4 个空格：表示属于 for 循环

names = ['小明', '小红']        # 两个字符串组成的列表
for index, name in enumerate(names): # enumerate 同时提供索引和值
    print(index, name)       # 第一次打印 0 小明，第二次打印 1 小红
```

`for i in range(3)` 可以理解为“把 0、1、2 依次放进变量 i，每放一个就执行一次缩进块”。`i` 只是一个变量名，换成 `step` 或 `epoch` 也可

以。在神经网络中，`for epoch in range(100)` 的意义是重复训练过程 100 轮，不是模型自动知道数学中的 100 个知识点。

S1.4 if、else 与缩进

```
score = 82                  # 将分数赋给变量 score
if score >= 60:             # >= 表示大于或等于
    print('合格')           # 条件为 True 才执行
else:                       # else 表示上面的条件没有成立
    print('需要继续练习')   # 在条件为 False 时执行
```

Python 用缩进确定代码块，而不是用大括号。多一个或少一个空格可能改变程序结构。如果代码报 `IndentationError`，先查看冒号后的缩进是否一致。神经网络代码里，`for epoch` 的循环体可能有七八行，漏缩进会让某一步只执行一次或者完全跑到循环外。

S1.5 def 函数与 return 的概念

函数（Function）用于给一段可重复的操作命名。函数定义时写 `def`，括号里是参数（Parameter），`return` 把结果返回给调用者。

```
def add_two_numbers(a, b):    # def 开始定义函数，a 与 b 是形参
    result = a + b            # 在函数内部把两数相加
    return result             # 返回计算结果给调用位置

answer = add_two_numbers(3, 5) # 用实参 3、5 调用函数，结果保存到 answer
print(answer)                 # 打印 8
```

注意 `print(result)` 与 `return result` 不同：前者只是把内容显示给读者，后者把值交还给调用方。很多代码“打印出来没问题，后面却拿不到结果”，原因就在于函数忘记使用 `return`。

S1.6 class、self、对象与方法

PyTorch 大量使用类（Class），例如 `nn.Linear` 是一个类，`nn.Linear(4,2)` 会创建一个层对象。类用于描述一类对象应有什么属性和方法，`self` 通常表示当前实例本身。

```
class Counter:                              # class 定义一个名叫 Counter 的类
    def __init__(self, start=0):            # __init__ 是创建对象时的初始化方法
        self.value = start                  # self.value 是当前对象保存的数值

    def increase(self):                     # 定义可供实例调用的方法
        self.value += 1                     # 在当前数值上加 1
        return self.value                   # 返回更新后的数值

counter = Counter(5)                        # 创建 Counter 实例，起始为 5
print(counter.increase())                   # 调用对象的方法，结果是 6
```

`self` 并不是 Python 保留关键字，但按习惯它用在实例方法的第一个参数位置。写 `counter.increase()` 时，Python 会自动把当前实例关联到这个位置，因此你不需要手动写成 `counter.increase(counter)`。知道这一点，以后看 `class MyModel(nn.Module)` 就容易了。

S1.7 import、from、as 的三种写法

```
import math                    # 将整个 math 模块引入，可写 math.sqrt
from math import sqrt          # 只把 math 模块里的 sqrt 名称导入当前作用域
import math as m               # as 把模块起别名 m，可写 m.sqrt
print(math.sqrt(9))            # 使用完整模块名，结果 3.0
print(sqrt(9))                 # 使用直接导入的函数名，结果 3.0
print(m.sqrt(9))               # 使用模块别名，结果 3.0
```

你看到 `from torch import nn` 时，翻译成“从 torch 这个模块中导入名为 nn 的工具集合”；看到 `import torch.nn as nn` 时，翻译成“把 torch.nn 导入并取短名 nn”。这两个常见写法通常让后续代码都可以写 `nn.Linear`。

S1.8 看神经网络代码时如何逐行注释

先弄清楚一行是“导入、创建对象、调用方法、读取属性、数学运算、循环、条件”中的哪一种。然后写出关键变量的**类型、形状和作用**。例如 `x=model(x)` 不能只写“模型运算”四个字；你应该写“将形状 `(batch,4)` 的输入依次经过层，得到形状 `(batch,2)` 的输出”。

常见括号也要分开理解：`[]` 可以创建列表、索引数据；`()` 可以调用函数、创建元组；`{}` 可以创建字典；`.` 访问属性或方法；`:` 可能用于定义代码块或切片。你一旦分得清这些符号，读框架代码的门槛就会明显降低。

Python 语法专项练习

1. `nums=[3,7,9]`，`nums[1]` 是多少？答案：7。

2. `range(5)` 会产生几个数字，最后一个是什么？答案：5 个，最后是 4。
3. `def f(x): return x*2`，调用 `f(4)` 得到什么？答案：8。
4. `from torch import nn` 的 `from` 与 `import` 分别表示什么？答案：从指定模块中导入指定名称。
5. `x.shape` 与 `x.reshape(2,3)` 为什么一个没有括号一个有？答案：前者通常读取属性，后者调用方法。

学习专题 02｜线性代数补课：从数字表到张量

这一专题刻意不从抽象的线性空间公理讲起，而是从你已经遇到的 `shape` 和 `nn.Linear` 出发，补齐理解神经网络所需的最小**线性代数（Linear Algebra）**知识。

S2.1 标量、向量、矩阵、张量的层级

标量（Scalar） 是一个数字，如 3.14；**向量（Vector）** 可以理解为一排数字，如 `[2,4,6]`；**矩阵（Matrix）** 是二维数字表；**张量（Tensor）** 是更一般的多维数值容器。张量的维度数用 `ndim` 表示，形状用 `shape` 表示。`(3,4)` 是二维张量，`(2,3,4)` 是三维张量。

注意“维数”的两种说法：我们口语里会说“长度为 5 的向量是五维向量”，指的是其**特征空间维度**；而 PyTorch 的 `ndim` 对这个向量通常返回 1，指的是其**张量轴数**。这两个维度概念不同，听到“几维”时要先弄清上下文。

S2.2 矩阵的 shape 怎样读

`(3,4)` 先读“3 行 4 列”。矩阵中的 `A[0,0]` 是左上角元素，`A[2,3]` 是第三行第四列的元素，因为 Python 索引从 0 开始。矩阵 A 的**转置（Transpose）** `A.T` 会交换行与列，形状从 `(3,4)` 变为 `(4,3)`。

例如二维向量 `x=[2,3]` 作为行向量是 `(1,2)`，写成列向量则是 `(2,1)`。数学推导常把向量看成列向量，深度学习框架处理 batch 时却常把每个样本放成一行。记号不同不是谁对谁错，而是要明确当前采用的布局。

S2.3 点积：矩阵乘法中最小的计算单位

向量点积（Dot Product）将对应元素相乘再加起来。如 `(1,2,3)` 与 `(4,5,6)` 的点积是 `1*4 + 2*5 + 3*6 = 32`。只有对应项数相同，才能这样逐项配对。点积得到一个数字，而不是一个新向量。

对于矩阵积 `C=A@B`，每个 `C[i,j]` 都是“A 第 i 行”和“B 第 j 列”的点积。这说明：A 的行数决定 C 有多少行，B 的列数决定 C 有多少列。形状口诀 `(m,n)@(n,p)=(m,p)` 背后就是这个事实。

S2.4 `*` 与 `@`：一定要在代码里区分

```
import torch                                  # 导入 PyTorch
A = torch.tensor([[1., 2.], [3., 4.]])       # 创建二维张量 A
B = torch.tensor([[5., 6.], [7., 8.]])       # 创建二维张量 B
print(A * B)                                  # 逐元素乘法：[[5,12],[21,32]]
print(A @ B)                                  # 矩阵乘法：[[19,22],[43,50]]
print(A.T)                                    # 转置：[[1,3],[2,4]]
```

你可以先独立手算第一行：`A@B` 左上角是 `1*5+2*7=19`，右上角是 `1*6+2*8=22`。如果得到逐元素乘法的结果，说明把 `@` 理解错了。计算机并不会根据“你想做神经网络”自动判断用哪一种乘法，必须明确运算符。

S2.5 广播与偏置：为什么能把一行加到所有行上

假设线性层的矩阵乘法结果是 `(3,2)`：

```
Y = [ [2,5],
      [4,1],
      [6,3] ]
```

偏置 `b=[10,-1]` 是长度为 2 的向量。当执行 `Y+b` 时，广播把同一偏置应用到每一行，结果为 `[[12,4],[14,0],[16,2]]`。为什么只需要两个偏置值？因为第二层需要输出 2 个特征，每个输出特征有一个对应偏置；不会因 batch 从 3 变成 300 而需要 300 套偏置。

S2.6 线性变换和激活函数的分工

用矩阵 `W` 对输入做 `X@W`，再加偏置，可以表达很多线性组合。可是如果网络只是连续执行多个没有非线性激活的线性层，它们可以合并成一次等效的仿射变换，不能因为层数多就获得任意复杂的非线性表达能力。**ReLU** 使部分输出变成 0，改变了数学函数的结构，让多层模型可以组合更灵活的决策边界。

S2.7 多一个 batch 维并不神秘

假设一条样本形状是 `(4,)`；把三条样本按行叠成一个 batch，就得到 `(3,4)`。`nn.Linear(4,2)` 可以处理单条 4 维输入，也可以处理由任意多条 4 维样本组成的 batch。改变 batch 大小通常不会修改线性层权重数量。

```
import torch                                # 导入 PyTorch
from torch import nn                        # 导入线性层
layer = nn.Linear(4, 2)                     # 定义从 4 特征到 2 特征的映射
single = torch.randn(4)                     # 单个样本，形状 (4,)
batch = torch.randn(3, 4)                   # 三个样本，形状 (3,4)
print(layer(single).shape)                  # 输出 (2,)
print(layer(batch).shape)                   # 输出 (3,2)
print(layer.weight.shape)                   # 权重为 (2,4)，与 batch 大小无关
```

形状专项题

1. `(7,6) @ (6,9)` 输出 `(7,9)`。
2. `(5,4) @ (3,2)` 不合法，因为 4 与 3 不等。
3. `Linear(10,8)` 的权重形状 `(8,10)`，偏置形状 `(8,)`。
4. `Linear(10,8)` 对 `(2,5,10)` 输入，输出 `(2,5,8)`，因为它作用于最后一个维度。
5. `ReLU` 对 `(4,9)` 输入输出形状保持 `(4,9)`。

学习专题 03｜概率、导数、梯度：深度学习数学不用先吓自己

S3.1 概率：1 表示“必然”，0 表示“不可能”

概率（Probability） 在 0 到 1 之间，表示事件发生的可能性。掷公平硬币出现正面的理论概率是 0.5，但连续掷四次得到的正面次数不一定恰好为 2。概率描述大量重复试验的规律，并不保证每次有限样本都严格达到理论比例。

机器学习里的“预测概率”还有一个重要概念叫**校准（Calibration）**。如果一个模型对许多样本都预测 80% 的正例概率，那么在这些样本中真实正例占比能否接近 80%，是可以统计检验的。模型输出 0.9 并不自动代表它校准良好，更不代表它永远有九成把握。

S3.2 平均值、方差、标准差描述了什么

均值（Mean） 是数据的平均位置；**方差（Variance）** 衡量数据围绕均值的平方偏离程度；**标准差（Standard Deviation）** 是方差的平方根，单位与原始数据一致。

例如样本 `[1,3,5]` 的均值是 3。它们与均值的差分别是 `[-2,0,2]`，平方后是 `[4,0,4]`。如果把这三个数字视作总体，方差为 `8/3≈2.67`；总体

标准差约为 1.63。如果按样本方差公式计算，分母是 `n-1`，值就不同。不要在不同定义之间悄悄混用。

S3.3 标准正态分布为什么常在神经网络里出现

标准正态分布（Standard Normal Distribution） 是均值 0、方差 1（因此标准差也为 1）的正态分布。可以记作 `Z~N(0,1)`。它的图像是一条左右对称的钟形曲线。大约 68.27% 的概率落在 `[-1,1]`，95.45% 落在 `[-2,2]`，99.73% 落在 `[-3,3]`。

调用 `torch.randn(3,4)` 时，生成一个形状 `(3,4)` 的随机张量，其中每个元素按标准正态分布的机制生成。但只有 12 个元素时，它们的平均数完全可能偏离 0，有些绝对值也可能超过 2。标准正态说的是**生成机制对应的理论分布**，不是对每一批少量样本的硬性约束。

S3.4 标准化公式中的每个符号

在常见的 **Z-score 标准化（Standardization）** 中，公式是 `z=(x-mu)/sigma`。`x` 表示原始数值，`mu` 是均值，`sigma` 是标准差。假设一次考试平均 70、标准差 10，你考 85，标准化分数为 `(85-70)/10=1.5`，表示高于均值 1.5 个标准差。

做机器学习预处理时，均值和标准差应从**训练集** 估计，再用相同参数转换验证集和测试集。不能把所有数据（特别是未来测试数据）一起拿来计算标准化参数，否则可能造成数据泄漏。

S3.5 函数、斜率与导数

函数（Function） 是输入到输出的规则，如 `f(x)=x^2`。在 `x=2` 时，输出是 4。导数（Derivative）反映输入微小变化会引起输出怎样的变化。对于 `f(x)=x^2`，导数 `f'(x)=2x`，因此在 x=2 的位置导数为 4。

你可以用小变化验证：x 从 2 变成 2.01，f 从 4 变为 4.0401，输出变化 0.0401，除以输入变化 0.01，得到 4.01，确实接近导数 4。这叫有限差分近似（Finite Difference）。数值近似不是精确导数，但可以帮助初学者建立直觉，还可用来检查自动求导程序。

S3.6 梯度下降的三步演算

我们要让 `L(w)=(w-3)^2` 尽可能小。导数为 `dL/dw=2*(w-3)`。从 `w=0` 开始，学习率 `lr=0.1`，第一步梯度为 -6，所以 `w_new=0-0.1*(-6)=0.6`。第二步在 `w=0.6`，梯度为 -4.8，新权重变成 1.08。第三步梯度为 -3.84，权重变成 1.464。你会发现它正逐步靠近 3。

如果学习率过大，比如 2，更新可能跨过最优位置并继续放大误差。梯度只告诉你**局部下降方向**，并不保证在任意复杂函数中直接找到全局最优。

S3.7 为什么神经网络离不开链式法则

考虑最简单的两步计算：`z=w*x+b`，`L=(z-y)^2`。损失 L 不直接把 w 写在表面，但它通过 z 依赖 w。**链式法则（Chain Rule）** 告诉我们 `dL/dw = dL/dz * dz/dw`。

因为 `dL/dz=2*(z-y)`，`dz/dw=x`，所以 `dL/dw=2*(z-y)*x`。如果 `x=3,w=2,b=1,y=10`，那么 `z=7`，损失为 9，梯度是 `2*(7-10)*3=-18`。负梯度说明稍微增加权重会让损失下降。这与自动求导输出的符号一致。

S3.8 数学核对：让 PyTorch 帮你复查手算

```
import torch                                # 导入 PyTorch
w = torch.tensor(2.0, requires_grad=True) # 需要对 w 求导，所以打开 requires_grad
x = torch.tensor(3.0)                      # 固定输入为 3
y = torch.tensor(10.0)                     # 正确输出为 10
z = w * x + 1.0                           # z=2*3+1=7
loss = (z - y) ** 2                        # 平方损失 L=(7-10)^2=9
loss.backward()                            # 自动计算对 w 的导数
print(loss.item())                         # 输出 9.0
print(w.grad.item())                       # 输出 -18.0，与手算一致
```

数学部分的最小掌握标准

如果你能解释“均值/标准差不是同一个概念”“`randn` 的随机数不必恰好均值为 0”“导数告诉局部变化方向”“梯度下降用减号更新参数”“链式法则把多步计算串起来”，就可以继续深入神经网络。不必为了怕微积分而停在课程门口。

学习专题 04｜不用框架，亲手实现一次梯度下降

这一专题把第 4、5 章的抽象说法变成能够完全看懂的 Python 程序。我们的任务很简单：根据一些 `(x,y)` 数据学到一条直线 `y=w*x+b`。它不是最复杂的神经网络，却足以完整呈现“参数、预测、损失、梯度、更新”五个要素。

S4.1 先固定任务与数据

设训练样本是 `(1,3)`、`(2,5)`、`(3,7)`。你可以看出它们符合 `y=2x+1`。但我们假设计算机不知道公式，只给它三个样本与一个有两个可训练参数的模型 `prediction=w*x+b`。起始时把 `w=0`、`b=0`。模型的第一次预测全部为 0，明显偏离真实值。

选择**均方误差（Mean Squared Error，MSE）**：把每个样本的预测值减去正确值，平方后取平均。平方让正误差和负误差都不会互相抵消，还会更重视较大的偏差。我们需要通过修改 w 和 b 让损失变小。

S4.2 手算第一步的损失

起始 `w=0,b=0`，三个预测是 `[0,0,0]`，正确值为 `[3,5,7]`，平方误差是 `[9,25,49]`，平均得到 `MSE=(9+25+49)/3=83/3≈27.667`。可见一开始损失很大。

对所有样本的平均 MSE，梯度可以写成：`dL/dw = (2/n)*sum((prediction_i-y_i)*x_i)`，`dL/db=(2/n)*sum(prediction_i-y_i)`。在起始位置，`dL/dw = (2/3)*(-3*1-5*2-7*3) = -68/3≈-22.667`；`dL/db=(2/3)*(-3-5-7)=-10`。

若取学习率 `lr=0.01`，更新后 `w_new=0-0.01*(-22.667)≈0.22667`，`b_new=0-0.01*(-10)=0.1`。一次调整就让 w 和 b 朝着正确方向移动。你可以再计算一次 MSE，看看是不是下降了。

S4.3 写出整个学习过程，每行都解释

```
# 特征 x 是模型的输入，这里用普通 Python 列表保存
xs = [1.0, 2.0, 3.0]                  # 三个训练输入样本
ys = [3.0, 5.0, 7.0]                  # 三个真实答案，满足 y=2*x+1
w = 0.0                               # 模型权重从 0 开始
b = 0.0                               # 模型偏置从 0 开始
lr = 0.01                             # 学习率：每步沿负梯度前进多远
n = len(xs)                           # 训练样本数，len() 结果为 3

for epoch in range(1000):             # 重复训练 1000 轮
    predictions = [w*x+b for x in xs] # 对每个 x 使用当前权重得到预测
    errors = [p-y for p,y in zip(predictions, ys)] # 预测减真值
    loss = sum(e*e for e in errors) / n # 平方误差求和后除以样本数
    grad_w = 2*sum(e*x for e,x in zip(errors, xs)) / n # w 的梯度
    grad_b = 2*sum(errors) / n        # b 的梯度
    w = w - lr*grad_w                 # 更新权重：减去学习率乘梯度
    b = b - lr*grad_b                 # 更新偏置：同样沿负梯度方向
    if epoch % 200 == 0:              # % 是取余，每到 200 的倍数打印一次
        print(epoch, round(loss, 5)) # round 保留指定小数位，便于观察

print('w:', w, 'b:', b)               # 最后查看学到的权重和偏置
print('predict x=4:', w*4+b)          # 对未在训练表出现的 x=4 做预测
```

这段程序不使用 PyTorch，但它已经是一个真正的小规模机器学习训练过程。模型由 w、b 决定；损失用 MSE；优化方法是梯度下降。神经网络里的 `loss.backward()` 正是把“手工推导很多参数的梯度”变成自动计算，`optimizer.step()` 把“手工一行一行改 w、b”变成统一操作。

S4.4 把手工梯度和 PyTorch 自动梯度比较

```
import torch                                    # 导入 PyTorch
w = torch.tensor(0.0, requires_grad=True)       # 模型权重，允许求导
b = torch.tensor(0.0, requires_grad=True)       # 模型偏置，允许求导
x = torch.tensor([1.0, 2.0, 3.0])               # 三个训练输入
y = torch.tensor([3.0, 5.0, 7.0])               # 三个正确标签
prediction = w*x+b                              # 前向预测，形状 (3,)
loss = ((prediction-y)**2).mean()               # 对平方误差求平均
loss.backward()                                 # 自动计算 L 对 w、b 的导数
print(w.grad.item(), b.grad.item())             # 应接近 -22.6667 与 -10
```

两段程序的数学定义完全一致，所以第一步梯度应该吻合。手动版本适合学原理，自动求导适合构建复杂模型。真正需要你理解的是“谁是参数，谁是固定数据，哪些运算构成损失，以及梯度如何影响更新”。

S4.5 一个问题：可以每个样本更新一次吗

可以。上面我们每轮先汇总三个样本的平均梯度，再更新一次，类似小型批量梯度下降。如果每处理一个样本就更新权重，便更接近**随机梯度下降（Stochastic Gradient Descent，SGD）**的单样本版本。实际训练常使用小批次（Mini-batch），如一次处理 32 个样本，在计算效率与梯度噪声之间取得平衡。

如果训练数据有 1000 个样本、batch_size=100，每轮一般有 10 次参数更新；训练 5 个 epoch，一共大约 50 次更新（假设每轮不丢弃尾部样本，恰好整除）。`epoch`、`batch`、`step` 不是同一个东西：epoch 是完整数据轮数，batch 是一次送入的数据块，step 常指一次更新。

本专题终点测试

1. 损失下降代表权重一定每一步都变大吗？答：不一定，梯度决定每个参数的变化方向。
2. 损失是 0 还需要继续大幅更新吗？答：对当前样本 MSE 为 0 时相关梯度也为 0，通常不应大幅更新。
3. 三个样本分成一个 batch 与三个 batch，更新次数相同吗？答：不相同，每个 epoch 前者通常一次，后者通常三次。

学习专题 05｜做一个从数据到预测的完整 PyTorch 小项目

这一次我们不只让网络打印 shape，而是把数据准备、模型创建、训练、验证、推理串成一个小项目。假设我们希望根据学习时间、做题数量等简单数字预测下一次测验的得分。真实教育预测要谨慎，这里使用**合成数据（Synthetic Data）**，不会使用任何人的真实学习记录。

S5.1 先确认问题类型与输入形状

输入包含两个特征：`hours`（学习小时数）和 `exercises`（完成练习数），输出是一个连续分数，因此这是一个**回归任务**。如果有 80 个训练样本，那么原始输入张量形状应为 `(80,2)`；标签张量形状可用 `(80,1)`。一个 `nn.Linear(2,1)` 已经足以学习简化的线性关系，没必要为了叫“AI 项目”就堆十层网络。

我们设定一个模拟规则 `score≈15 + 7*hours + 0.6*exercises + noise`，其中 noise 是随机噪声。请注意：这是**为教学制造的数据机制**，不是对现实学生成绩的有效因果结论。我们没有理由说真实学习一小时一定提高 7 分。

S5.2 生成合成数据与固定随机种子

```
import torch                                          # 导入张量库
from torch import nn                                  # 导入神经网络层

torch.manual_seed(11)                                 # 固定随机种子，便于复现实验
hours = torch.rand(100, 1) * 5                        # 100 个样本，学习时间 0 到 5 小时
exercises = torch.rand(100, 1) * 40                   # 每个样本的题数在 0 到 40 附近
X = torch.cat([hours, exercises], dim=1)              # 沿特征维拼接，X 形状 (100,2)
noise = torch.randn(100, 1) * 2                       # 添加随机噪声，每条一个数字
y = 15 + 7*hours + 0.6*exercises + noise              # 合成目标分数，形状 (100,1)
print(X.shape, y.shape)                               # 查看输入与标签的形状
```

`torch.rand` 默认从 0 到 1 的均匀分布（Uniform Distribution）中产生随机数，`torch.randn` 默认从标准正态分布产生随机数。它们只差一个字母，但数学分布不同。`torch.cat` 用于按指定维度拼接已有张量。这里 `dim=1` 指第二个轴，也就是样本的特征轴，因此由两列合成两列输入。

S5.3 分开训练集与验证集

```
X_train = X[:80]                                      # 前 80 个样本作为训练集
Y_train = y[:80]                                      # 训练集对应的正确标签
X_val = X[80:]                                        # 后 20 个样本作为验证集
Y_val = y[80:]                                        # 验证集对应的标签
print(X_train.shape, X_val.shape)                     # 分别是 (80,2) 与 (20,2)
```

这里因为数据本来是独立随机生成的，用固定切片演示划分不会带来时间顺序泄漏问题。真实数据则要检查样本是否具有时间、用户、设备等群组关系。尤其在预测下一次测验成绩时，同一个学生的重复记录可能需要按学生分组拆分，而不能随意打散。

S5.4 创建模型、损失和优化器

```
model = nn.Linear(2, 1)                          # 2 个输入特征映射为 1 个预测分数
loss_fn = nn.MSELoss()                           # 选择回归任务常用的均方误差
optimizer = torch.optim.Adam(                    # Adam 是自适应梯度优化器
    model.parameters(), lr=0.03                  # 传入模型参数，学习率设为 0.03
)
```

`model.parameters()` 取出 `weight` 与 `bias` 这些可训练参数。`lr` 是关键超参数，但没有对所有模型都适用的最佳数字。Adam 与 SGD 使用不同的更新机制，选择哪一个要看任务与训练表现。`nn.MSELoss()` 返回的损失是非负的，越小通常表示训练样本的预测与目标越接近。

S5.5 真正训练，并同时观察验证损失

```
for epoch in range(500):                         # 训练 500 轮，数据规模很小
    model.train()                                # 设置训练模式
    pred = model(X_train)                        # 对训练输入进行前向传播
    loss = loss_fn(pred, Y_train)                # 计算训练平均平方误差
    optimizer.zero_grad()                        # 清空之前留下的梯度
    loss.backward()                              # 反向传播计算参数梯度
    optimizer.step()                             # 根据梯度更新模型参数

    if epoch % 100 == 0:                         # 每训练 100 轮观察一次
        model.eval()                             # 切换到评估模式
        with torch.no_grad():                    # 验证时不计算参数梯度
            val_pred = model(X_val)              # 对验证集产生预测
            val_loss = loss_fn(val_pred, Y_val)  # 计算验证集损失
        print(epoch, loss.item(), val_loss.item()) # 输出轮数和两类损失
```

在这个示例里，真实规律本来就是线性的，所以线性模型通常可以较好地拟合。因为加入噪声，损失不应期待精确为 0。验证损失也不需要每

轮都单调下降；我们主要观察整体趋势，以及训练与验证是否出现明显分歧。

S5.6 对新样本做预测：不再训练

```
model.eval()                                    # 使用评估模式
new_student = torch.tensor([[2.0, 15.0]])       # 1 个样本：学习 2 小时、做 15 题
with torch.no_grad():                           # 推理不需要记录梯度
    estimated_score = model(new_student)        # 模型输出形状应是 (1,1)
print(estimated_score.item())                   # 打印模型估计的分数
```

这里 `new_student` 一定写成 `[[2.0,15.0]]` 的二维结构，形状是 `(1,2)`。如果写成 `[2.0,15.0]`，形状是 `(2,)`，`nn.Linear(2,1)` 也可能接受这种单条输入，但输出形状会不同。为后续批量部署，通常最好统一带 batch 维。

S5.7 再做一个更像神经网络的 XOR 实验

上面的线性回归不足以体现隐藏层的非线性能力。因此我们再构造 XOR 的四个输入：`[0,0]→0`、`[0,1]→1`、`[1,0]→1`、`[1,1]→0`。把它送入 `2→8→1` 的小网络，用 ReLU 引入非线性，并使用二分类损失。

```
import torch                                          # 导入张量库
from torch import nn                                  # 引入神经网络模块

torch.manual_seed(3)                                  # 固定初始随机种子
X = torch.tensor([[0.,0.], [0.,1.], [1.,0.], [1.,1.]]) # 四条 XOR 输入
Y = torch.tensor([[0.], [1.], [1.], [0.]])            # 四条标签，形状 (4,1)
model = nn.Sequential(                                # Sequential 把若干层顺序连接
    nn.Linear(2, 8),                                  # 2 个输入特征变成 8 个隐藏特征
    nn.ReLU(),                                        # 引入非线性，不能省略
    nn.Linear(8, 1)                                   # 8 个特征合成一个 logit
)
loss_fn = nn.BCEWithLogitsLoss()                      # 二分类损失，直接接收 logit
optimizer = torch.optim.Adam(model.parameters(), lr=0.05) # 创建优化器

for epoch in range(1500):                             # 多次更新，尝试拟合 XOR
    logits = model(X)                                 # 前向传播得到形状 (4,1) 的得分
    loss = loss_fn(logits, Y)                         # 对比真实标签算二分类损失
    optimizer.zero_grad()                             # 清零旧梯度
    loss.backward()                                   # 自动求导
    optimizer.step()                                  # 更新两个线性层的参数

model.eval()                                          # 切换为评估模式
with torch.no_grad():                                 # 推理阶段不需要梯度
    probs = torch.sigmoid(model(X))                   # 把 logit 变成 0～1 的概率样式数值
    predicted = (probs >= 0.5).int()                  # >=0.5 视为 1，其余为 0
print(predicted)                                      # 检查四个训练样本的预测
```

请注意，随机初值、优化器和数据规模会影响最终结果；这个例子是为了说明网络能够拟合 XOR，不应把四个样本的训练准确率当作真实世界泛化能力。现在你可以做一个对照实验：删掉 `nn.ReLU()`，让网络只剩两层线性变换，再观察是否能同时处理四个 XOR 条件。

S5.8 项目里应当记下哪些实验信息

至少记下随机种子、训练集/验证集拆分方式、模型结构、损失函数、优化器、学习率、训练轮数与评估指标。你可以把这些写在每天的学习记录网页里。很多训练结果难以复现，不是因为计算机“突然变笨”，而是缺少随机种子、数据版本或预处理设置。

项目总结自测

1. 为什么第一项任务选 MSE？答：要预测连续数值，是回归问题。
2. XOR 为什么要有非线性层？答：否则多个线性层仍无法形成非线性决策边界。
3. `.item()` 有什么限制？答：主要用于从只有一个元素的张量提取 Python 标量。
4. 验证集的标签可以用于反向传播更新权重吗？答：常规流程不应该。

动手实验 01｜从一张计算表到可训练的感知机

这组实验可以放在第 3 章之后。目标不是追求最高的模型准确率，而是让你能**独立说出每个变量的作用**。实验只用 Python 标准语法，不需要安装 NumPy 或 PyTorch。

L1.1 先训练眼睛，不急着敲代码

拿一张纸画五列：`输入 x`、`权重 w`、`偏置 b`、`线性结果 z`、`预测 y`。对于一个两输入的感知机，一次计算只有 `z=w1*x1+w2*x2+b` 和一次阈值判断。例如 `x=(1,2)`、`w=(0.3,-0.2)`、`b=0.1`，那么 `z=0.3-0.4+0.1=0`。若阈值是“严格大于 0 输出 1”，结果为 0。这里 `>=0` 与 `>0` 会产生不同答案，说明数学条件必须写清。

接着把 `x` 换为 `(2,1)`，其他参数不变，得到 `z=0.6-0.2+0.1=0.5`，输出 1。现在问自己：变的是模型参数，还是输入样本？答案是输入变了，模型权重仍固定。**推理（Inference）** 指用固定参数对新输入计算；**训练（Training）** 则包含通过数据改变参数。

L1.2 用函数减少重复代码

```
def predict(x1, x2, w1, w2, b):        # 定义函数，前两个是特征，后三个是模型参数
    z = x1 * w1 + x2 * w2 + b          # 加权求和得到线性结果
    if z > 0:                          # 按“严格大于 0”作为分类阈值
        return 1                       # 返回类别 1
    else:                              # 如果 z 小于或等于 0
        return 0                       # 返回类别 0

print(predict(1, 2, 0.3, -0.2, 0.1)) # 第一个样本，预测应为 0
print(predict(2, 1, 0.3, -0.2, 0.1)) # 第二个样本，预测应为 1
```

这个函数还没有学习功能。它只把固定参数对应的前向计算包装起来。因此以后你看到 `prediction=model(x)` 时，可把它看成调用一个更复杂的预测函数。区别在于 PyTorch 的模型还会保存一套可训练参数，并与自动求导系统合作。

L1.3 一轮训练应该怎样记录

假设我们要训练逻辑 OR。当前权重为 `(0,0)`、偏置 0、学习率 0.2。先给样本 `(0,1)`，真实标签 1，当前预测 0。误差 `target-prediction=1`。新权重第一项仍为 0，第二项变成 0.2，偏置变成 0.2。接着给样本 `(1,0)`，新线性结果为 `0*1 + 0.2*0 + 0.2=0.2`，预测已为 1，因此这一次误差为 0，不更新参数。顺序会影响经典在线感知机的中间过程，这是正常的。

现在开始练习：对 `(0,0)` 样本，模型预测为 1，因为偏置 0.2 大于 0；正确标签为 0，因此误差 -1，更新后偏置变成 0。这一步提醒我们：让

一个样本正确的更新，可能暂时使另一个样本出错，所以需要多轮遍历，直到满足条件或达到停止限制。

L1.4 实验记录表与复查方法

检查点	你要回答的事	正确判断依据
输入形状	一条样本包含几个特征	`x1` 与 `x2`，共 2 个
参数	哪些数会因训练改变	`w1`、`w2`、`b`
前向	预测之前算了什么	`z=w1*x1+w2*x2+b`
误差信号	为什么更新	`target-prediction` 非零
学习率	更新多少	更新量乘 `lr`
结束	为什么停	完成预设轮数或满足正确率标准

实验结束后，请把函数 `predict` 改写成接收 `x` 列表与 `weights` 列表的形式，并解释为什么函数参数从五个减少成三个：因为输入特征和权重被放进了列表，而不是数学上发生了变化。

L1.5 自己出一道题检验理解

让 `x=(2,3)`、`w=(1,-1)`、`b=0`，预测结果是什么？线性结果是 `2-3=-1`，输出 0。若目标标签是 1，学习率为 0.1，那么更新量为 `0.1*(1-0)*(2,3)=(0.2,0.3)`；新权重是 `(1.2,-0.7)`，偏置变成 0.1。重新

前向计算得 `2*1.2+3*(-0.7)+0.1=0.4`，输出 1。你已经完成一次小型学习算法的手工训练。

动手实验 02｜PyTorch 多分类：从 logits 到准确率

第 5 章已经讲了回归和 XOR。这次我们学习一个常见的新任务：**多分类（Multi-class Classification）**。我们人工生成三类二维数据：一类靠近左侧，一类靠近右侧，一类靠近上方。整个实验不需要联网下载数据，只需要安装 PyTorch。

L2.1 先设计正确的数据形状

每个样本只有两个坐标 `(x1,x2)`，因此输入最后一个维度为 2。我们给三类各创建 40 个训练样本，总共 120 个样本，得到输入 `X.shape=(120,2)`。每条样本的正确类别是整数 `0`、`1` 或 `2`，所以标签 `Y.shape=(120,)`。输出需要三个类分数，因此模型最后一层应是 `nn.Linear(hidden,3)`，整个预测 logits 的形状为 `(120,3)`。

注意：`(120,3)` 里每一行存的是一条样本针对三个类别的**原始分数（Logits）**，不是“120 条数据最终只剩三条数据”。行数 120 代表 batch 大小，最后一维 3 代表类别数量。`nn.CrossEntropyLoss` 就是处理这类原始分数与整数标签的常见损失函数。

L2.2 创建可重复的人工数据

```
import torch                                        # 导入 PyTorch
from torch import nn                                # 导入神经网络工具

torch.manual_seed(10)                               # 固定随机数种子，方便复现
n = 40                                              # 每一类产生 40 条训练数据
class0 = torch.randn(n, 2) * 0.35 + torch.tensor([-2.0, 0.0]) # 左侧簇
class1 = torch.randn(n, 2) * 0.35 + torch.tensor([2.0, 0.0])  # 右侧簇
class2 = torch.randn(n, 2) * 0.35 + torch.tensor([0.0, 2.5])  # 上方簇
X = torch.cat([class0, class1, class2], dim=0)  # 沿样本轴拼接，形状 (120,2)
Y = torch.cat([                                     # 把三组标签拼成一个向量
    torch.zeros(n, dtype=torch.long),               # 类别 0，40 个整数 0
    torch.ones(n, dtype=torch.long),                # 类别 1，40 个整数 1
    torch.full((n,), 2, dtype=torch.long)           # 类别 2，40 个整数 2
])
print(X.shape, Y.shape, Y.dtype)                    # 检查维度与整数类型
```

`torch.randn(n,2)` 产生标准正态随机点，乘以 0.35 缩小散布范围，最后加上一个二维向量，把每类点移动到不同中心。

`torch.full((n,),2,...)` 表示创建一个形状为 `(n,)`、所有值都是 2 的整数张量。这里 `dtype=torch.long` 十分关键，因为多分类交叉熵通常要求目标标签为整数类别索引。

L2.3 用模型把二维坐标转换成三个分数

```
model = nn.Sequential(                          # 按顺序组合各层
    nn.Linear(2, 8),                            # 每条样本从 2 特征变 8 特征
    nn.ReLU(),                                  # 非线性激活，形状不变
    nn.Linear(8, 3)                             # 最后输出三个类别的 logits
)
loss_fn = nn.CrossEntropyLoss()                 # 多分类损失，接收 logits 和整数标签
optimizer = torch.optim.Adam(                   # Adam 优化器会更新模型参数
    model.parameters(), lr=0.02                 # 使用模型参数，学习率 0.02
)
```

你先不要急着训练。请在纸上写出 `X.shape=(120,2)`，经过第一层 `(120,8)`，ReLU 后仍是 `(120,8)`，经过第二层 `(120,3)`。运行 `print(model(X).shape)` 验证。如果形状出错，先查模型最后一维是否匹配输入特征数，不要随机调学习率。

L2.4 完整训练，并学会从 logits 得到类别

```
for epoch in range(200):                        # 重复训练 200 轮
    model.train()                               # 设置为训练模式
    logits = model(X)                           # 前向传播，输出 (120,3)
    loss = loss_fn(logits, Y)                   # 计算三类交叉熵损失
    optimizer.zero_grad()                       # 清除旧梯度
    loss.backward()                             # 求当前损失对参数的梯度
    optimizer.step()                            # 按梯度更新模型参数

model.eval()                                    # 设置为评估模式
with torch.no_grad():                           # 不再创建梯度计算图
    logits = model(X)                           # 重新计算训练数据的类分数
    predicted = logits.argmax(dim=1)            # 每行选最高分的类别索引
    accuracy = (predicted == Y).float().mean()  # 逐项比较后算正确比例
print('train accuracy:', accuracy.item())       # 查看训练正确率
```

`argmax(dim=1)` 表示沿第二维，即类别维度，找到分数最大的位置。输出变成 `(120,)`，每个数是类别索引。这里的准确率仍然是在训练集上计算，只能说明模型对看过的例子拟合如何，**不能直接当成真实泛化性能**。

L2.5 为什么还应该有独立测试点

你可以再用新的随机数生成三组数据，并用相同中心位置但不同噪声作为**独立测试样本**，再计算准确率。更规范的课程实验会在训练前就留出测试数据，甚至使用验证集挑超参数，避免在看到最终测试结果后反复调模型。这个人工簇任务很容易，不能用它的高分证明模型适用于现实照片或自然语言。

常见错误有三类：把 `Y` 变成 `(120,1)` 后直接送入不匹配的损失；把 `Y` 设成浮点值但损失期望 `long`；在 `CrossEntropyLoss` 之前又对 logits 手动做 softmax。你可以故意制造这些错误，观察报错，再逐一修复。理解报错比只会复制运行更重要。

本实验问答

1. `nn.Linear(8,3)` 为什么输出 3 而不是 120？因为每条样本需要三个类分数，而 120 是样本数。
2. `argmax(dim=1)` 为什么改变形状？因为它把每行三个分数压缩成一个最大值索引。
3. 只报告训练准确率有什么问题？无法判断对新数据的能力。

动手实验 03｜不下载图片，也能理解 CNN 训练

这组实验使用程序生成的简单 8×8 灰度图片：一类图片中间有竖条，另一类图片中间有横条。图片很小，因此你可以直接想象像素图长什么样。我们不是要做实用视觉识别器，而是检验卷积层是否能从局部结构中学习区别。

L3.1 为什么输入有四个维度

我们创建 80 张 8×8 的灰度训练图像，所以张量形状是 `(80,1,8,8)`。四个数字依次是 `batch=80`、`channels=1`、`height=8`、`width=8`。标签只有两类，形状 `(80,)`。若换成彩色图，通道维通常从 1 变成 3，但图像的高、宽不一定变化。

卷积层 `nn.Conv2d(1,4,kernel_size=3,padding=1)` 表示输入一通道、输出四通道，每个输出通道可以学习不同的局部模式。对于高宽 8 的输入，核 3、填充 1、步幅默认 1 时，空间高宽仍是 8。于是卷积输出 `(80,4,8,8)`。

L3.2 生成水平条和竖直条

```
import torch                                        # 导入张量工具
from torch import nn                                # 导入神经网络模块

torch.manual_seed(5)                                # 固定随机种子
X = torch.randn(80, 1, 8, 8) * 0.08                 # 80 张含小噪声的灰度图
Y = torch.zeros(80, dtype=torch.long)               # 创建 80 个整数类别标签
for i in range(80):                                 # 依次修改每张图
    if i < 40:                                      # 前 40 张设置为竖条类别
        X[i, 0, :, 3:5] += 1.0                      # 所有行的第 3、4 列加亮度
        Y[i] = 0                                    # 竖条标签是 0
    else:                                           # 后 40 张为横条类别
        X[i, 0, 3:5, :] += 1.0                      # 第 3、4 行所有列增加亮度
        Y[i] = 1                                    # 横条标签是 1
print(X.shape, Y.shape)                             # 分别为 (80,1,8,8) 和 (80,)
```

`X[i,0,:,3:5]` 这类索引值得仔细读：`i` 选第 i 张图，`0` 选唯一的灰度通道，第三个位置的 `:` 选全部行，最后的 `3:5` 选第 3、4 列（右端 5 不包含）。这样把原来随机噪声很暗的图片中间两列加亮，形成竖条。横条写法只是交换高宽两个索引的位置。

L3.3 构造卷积模型，并逐层写出 shape

```
model = nn.Sequential(                              # 把各层顺序连接起来
    nn.Conv2d(1, 4, kernel_size=3, padding=1),      # (80,1,8,8)->(80,4,8,8)
    nn.ReLU(),                                      # 非线性激活，不改变维度
    nn.Flatten(),                                   # (80,4,8,8)->(80,256)
    nn.Linear(4 * 8 * 8, 2)                         # 把 256 个特征变成两类分数
)
print(model(X).shape)                               # 预期输出 (80,2)
```

为什么要写 `4*8*8`？因为在展平之前，每张图有 4 个特征通道，每个通道有 8×8 个位置，总共 `4*8*8=256` 个数。`Flatten()` 保留第一维 batch，把剩余三个维度合并。它不是重新学习特征，只是在调整数值的排列方式。

L3.4 训练并记录结果

```
loss_fn = nn.CrossEntropyLoss()                  # 两类互斥分类仍可用多类交叉熵
optimizer = torch.optim.Adam(                    # 创建 Adam 优化器
    model.parameters(), lr=0.01                  # 接收全部模型参数，学习率 0.01
)
for epoch in range(150):                         # 训练 150 轮
    logits = model(X)                            # 前向传播产生 (80,2) 类分数
    loss = loss_fn(logits, Y)                    # 计算预测与类别标签的差异
    optimizer.zero_grad()                        # 清空旧梯度
    loss.backward()                              # 反向传播计算梯度
    optimizer.step()                             # 更新模型参数
model.eval()                                     # 切换评估模式
with torch.no_grad():                            # 评估阶段不产生新的梯度图
    pred = model(X).argmax(dim=1)                # 选取两类中分数最高的类别
    correct = (pred == Y).float().mean()         # 与训练标签比较得到准确率
print(correct.item())                           # 查看模型对训练样本的拟合
```

这个任务通常不难拟合。但要注意，你在训练数据上评估并不能证明泛化。若要建立更合理的实验，请另外生成一组图，采用相同的规则但使用新的随机噪声，作为验证集。还可以把竖条和横条的位置左右移动，测试模型是否对位置变化鲁棒。

L3.5 什么算真正理解了 CNN 代码

至少能回答：输入张量的四个维度分别是什么；卷积后为何通道从 1 变成 4；为什么 `Flatten` 后是 256；输出为什么是 2；损失为什么接受 logits；`argmax` 为什么沿 `dim=1`。这些问题只要有一个解释不清，建议退回对应小节再读，而不是继续增加卷积层数。

扩展挑战

把 `nn.Conv2d(1,4,3,padding=1)` 改成 `nn.Conv2d(1,4,3,padding=0)`，此时空间大小从 8 变为 6，后面的 `nn.Linear` 输入特征应该改成多少？答案：`4*6*6=144`。这道题不用运行也应该能靠形状规则答出来。

动手实验 04｜中文短句分类：BoW 与线性层

这次的目标是把简短的中文学习反馈分成“积极”和“消极”两类。它是一个极小的教学数据集，不适合做真实情感分析，但足以把第 13 章的词袋向量与第 5 章的 PyTorch 分类结合起来。

L4.1 分词、词表与数值向量

为避免引入中文分词库，我们手工在词之间加空格。四条文本分别是“今天 学习 很 开心”“这个 练习 很 有趣”“今天 学习 很 累”“这个 练习 太 难”。标签分别是 `[1,1,0,0]`。把所有不同的词放进词表，就得到一个由词到列索引的对应关系。

请注意真实文本处理不能随意从**测试集**一起构造词表后再宣称完全没有使用测试信息；正式机器学习管线应该仅从训练数据拟合词表、TF-IDF、标准化等预处理参数，再应用到验证和测试数据。这里四条数据只为了练通完整流程，因此我们明确只看训练集拟合，并不声称已经验证泛化。

L4.2 把句子变成计数矩阵

```
import torch                                        # 导入 PyTorch
from torch import nn                                # 导入神经网络工具
texts = ['今天 学习 很 开心', '这个 练习 很 有趣', # 两条积极示例
         '今天 学习 很 累', '这个 练习 太 难'] # 两条消极示例
labels = torch.tensor([1, 1, 0, 0], dtype=torch.long) # 整数类别标签
vocab = sorted({w for text in texts for w in text.split()}) # 去重并排序
word_to_index = {word:i for i,word in enumerate(vocab)} # 字典把词映射到列索引
features = torch.zeros(len(texts), len(vocab))    # 创建文档数×词数的全 0 矩阵
for row, text in enumerate(texts):                # 依次处理每一条文本
    for word in text.split():                     # 遍历本句的各个词
        col = word_to_index[word]                 # 找到这个词对应的列号
        features[row, col] += 1.0                 # 该词出现一次，计数加一
print(features.shape)                             # (4,词表大小)
```

如果“很”在一句话里出现两次，它对应的值就是 2，而不是依然保持 1。这种表示是计数型 BoW；如果你只关心出现与否，也可以改成 0/1 二值表示。关键在于列顺序要统一，训练时与预测时都必须使用同一套词表映射。

L4.3 用线性层把词频变成分类分数

```
num_features = len(vocab)                         # 词表大小就是输入特征数
model = nn.Linear(num_features, 2)                # 两种类别，各得到一个 logit
loss_fn = nn.CrossEntropyLoss()                   # 处理两类互斥标签的交叉熵
optimizer = torch.optim.Adam(model.parameters(), lr=0.05) # 创建优化器
for epoch in range(300):                          # 对小训练集重复 300 轮
    logits = model(features)                      # 输入词袋矩阵，输出 (4,2)
    loss = loss_fn(logits, labels)                # 根据整数类别算损失
    optimizer.zero_grad()                         # 清空旧梯度
    loss.backward()                               # 求损失对权重偏置的梯度
    optimizer.step()                              # 执行一次参数更新
print(model(features).argmax(dim=1))              # 查看训练数据上的类别预测
```

由于训练集只有四条短句，模型很容易直接记住这些句子，不可用此成绩宣称它理解了情感。如果换成“这个 练习 很 难 但 学会 了 很 开心”，词袋缺乏词序，可能难以正确处理转折关系。这引出了后面 Embedding、RNN 与 Transformer 的价值：考虑更丰富的上下文结构。

L4.4 新句子怎样转成和训练时一样的维度

```
new_text = '今天 练习 很 开心'                    # 准备一条新的句子
new_vector = torch.zeros(1, len(vocab))          # 一条样本，词表宽度不变
for word in new_text.split():                     # 逐词处理输入
    if word in word_to_index:                     # 未在词表的词先忽略
        new_vector[0, word_to_index[word]] += 1.0 # 已知词的计数加 1
model.eval()                                      # 切换到评估模式
with torch.no_grad():                             # 推理不需要求梯度
    result = model(new_vector).argmax(dim=1)      # 选取最高类别分数的索引
print(result.item())                              # 输出分类索引
```

这里的 `new_vector` 必须是 `(1,vocab_size)`，而不是 `(1,新句子词数)`。这是文本模型最重要的形状原则之一：训练和推理需要遵守同一个特征定义。真实 NLP 系统对未知词还会使用 UNK（未知词）或子词机制等方法，而不是一概忽略。

实验检查

1. 词表增长时，线性层 `in_features` 会怎样？答：要与新的词表宽度匹配。
2. 为什么不能把测试集里的词混入训练词表再假装“完全没用测试数据”？答：词表也属于可能从数据中学习到的预处理状态。

3. 为什么小数据集的训练高分不能说明真实应用效果？答：模型可能只是记忆了四条样本，缺少独立验证。

动手实验 05｜注意力机制的矩阵形状逐项核对

Transformer 章节最容易让人迷路的不是英语缩写，而是连续出现的转置、矩阵乘法和 Softmax。这个实验不训练 Transformer，只做一个单头的**缩放点积注意力（Scaled Dot-Product Attention）**，一步一步打印每个中间结果的形状。

L5.1 先用三个词想象问题

假设序列只有三个位置：“猫 / 追 / 球”。每个位置被表示成一个向量。我们希望在更新“追”的表示时，不只看它自己，还能根据关系参考“猫”和“球”。自注意力会在这些位置之间计算关系分数，再用 Softmax 转换为权重，对 Value 向量进行加权求和。

这种计算不是固定地“动词只能看名词”，权重取决于模型学到的投影以及当前输入。如果没有训练，随机 Q、K、V 可能只产生毫无任务意义的注意力。实验的目的只是理解**形状和操作**，不把随机注意力权重当作真实语言学解释。

L5.2 准备三行向量

设序列长度为 3，每个 Query/Key 向量长度是 2，而 Value 的维度也是 2。我们可以写出 `Q.shape=(3,2)`、`K.shape=(3,2)`、`V.shape=(3,2)`。

```
import math                                          # 导入数学模块，用于开平方
import torch                                         # 导入 PyTorch
Q = torch.tensor([[1.,0.],[0.,1.],[1.,1.]])        # 三个 Query，各是二维向量
K = torch.tensor([[1.,0.],[0.,1.],[1.,1.]])        # 三个 Key，各是二维向量
V = torch.tensor([[1.,2.],[3.,4.],[5.,6.]])        # 三个 Value，各是二维向量
scores = Q @ K.T                                     # (3,2)@(2,3)，得到 (3,3)
scores = scores / math.sqrt(K.shape[-1])             # 根据 Key 维度 2 缩放分数
weights = torch.softmax(scores, dim=-1)              # 对每一行做 Softmax，权重和为 1
output = weights @ V                                 # (3,3)@(3,2)，输出 (3,2)
print('scores:', scores.shape)                       # 三个位置对三个位置的关系分数
print('weights:', weights.shape)                     # 每个 Query 对所有 Key 的权重
print('output:', output.shape)                       # 每个位置得到二维聚合结果
```

这里 `K.T` 把 `(3,2)` 转成 `(2,3)`，才能与 Q 做乘法。第一个乘法输出 `(3,3)`，可以理解为三个 Query 分别比较三个 Key。第二个乘法 `(3,3)@(3,2)` 仍然遵循行列配对规则，输出三个位置、每个位置两个特征。

L5.3 `dim=-1` 为什么代表对最后一维求 Softmax

PyTorch 中 `dim=-1` 表示最后一个轴。`weights.shape=(3,3)` 时，就是对每一行的三个 Key 分数单独做 Softmax，使每一行的权重和为 1。用代码核对：

```
row_sums = weights.sum(dim=-1)                       # 对每行最后一维求和
print(row_sums)                                      # 结果应当接近 [1.,1.,1.]
```

如果你错把 Softmax 写成 `dim=0`，就会沿第一维归一化，导致不符合“每个 Query 为自己的候选 Key 分配总权重 1”的意图。代码可能仍能运行，但语义已经错了。这是比直接报错更危险的错误。

L5.4 因果掩码：为什么未来位置不能被看见

对于自回归生成任务，第一个位置不应该看到第二、第三个真实 Token，否则会在训练时使用未来答案。可以在 Softmax 前，将禁止访问的位置打上极小分数，例如负无穷。这样 Softmax 后它们的权重会接近或等于 0。

```
length = Q.shape[0]                                  # 当前序列长度为 3
mask = torch.triu(torch.ones(length, length), diagonal=1).bool() # 上三角未来位置
masked_scores = scores.masked_fill(mask, float('-inf')) # 将未来位置分数设负无穷
causal_weights = torch.softmax(masked_scores, dim=-1) # 对允许位置做归一化
print(causal_weights)                                # 第一行只能看位置 0
```

`torch.triu` 是取上三角区域，`diagonal=1` 表示排除主对角线，只标记严格位于当前位置之后的未来位置。`masked_fill` 会在掩码为真的地方替换值。此处每行至少能看到自己，因此不会出现整行全为负无穷导致的无效 Softmax；真实模型要更加小心掩码与填充的组合。

L5.5 多头注意力为什么增加一个维度

单头注意力只有一套 Q/K/V 变换；**多头注意力（Multi-Head Attention）** 则使用多套投影，让不同 head 可以学习不同的关系模式。一个常见形状是 `Q=(batch,heads,seq_len,head_dim)`。若 batch=2、heads=4、seq_len=5、head_dim=8，那么 Q 为 `(2,4,5,8)`；K 的最后两维交换后，注意力分数为 `(2,4,5,5)`。这里最后的 5×5 仍表示位置与位置的关系，只是对每个 batch、每个 head 分别计算。

形状练习

1. `Q=(6,16)`、`K=(6,16)`，`Q@K.T` 的形状是 `(6,6)`。
2. 注意力权重 `(6,6)` 与 `V=(6,32)` 相乘，输出 `(6,32)`。
3. 为什么 Softmax 要沿候选 Key 的维度做？为了让每个 Query 的注意力权重构成自己的分配比例。
4. 为什么随机生成的 attention 权重不代表真正理解词语？因为相关投影尚未按语言任务学习。

动手实验 06｜诊断错误、复习安排与下一步学习路线

你已经看过许多看似不同的模型：感知机、MLP、CNN、RNN、Transformer。现在学习重点应从“认识新名词”转向“遇到代码时能定位问题”。这份实验把常见错误分成**形状（Shape）**、**类型（Dtype）**、**设备（Device）**、**损失函数（Loss）**和**训练流程（Training Loop）**五类。

L6.1 第一类错误：输入特征数与 Linear 不匹配

```
import torch                                  # 导入 PyTorch
from torch import nn                          # 导入神经网络层
layer = nn.Linear(4, 2)                       # 要求输入最后一维长度为 4
correct = torch.randn(3, 4)                   # 正确输入：3 个样本，每个 4 特征
print(layer(correct).shape)                   # 得到 (3,2)
# wrong = torch.randn(3, 5)                   # 若把最后一维变成 5，便不匹配
# print(layer(wrong))                         # 这行取消注释通常会触发矩阵乘法错误
```

最有效的排查方法是打印 `x.shape`、`layer.in_features`、`layer.weight.shape`。不要把错误归因于“PyTorch 好复杂”，只需判断最后一维是否与该层要求的输入特征数一致。

L6.2 第二类错误：把类别标签当成任意浮点数

```
logits = torch.tensor([[2., 1., 0.], [0., 3., 1.]]) # 两个样本，各有三类分数
labels = torch.tensor([0, 1], dtype=torch.long)    # 正确标签是整数类别索引
loss_fn = nn.CrossEntropyLoss()                    # 创建多类交叉熵损失函数
print(loss_fn(logits, labels).item())              # 计算标量损失
```

这里目标标签形状是 `(2,)`，而 logits 形状是 `(2,3)`。损失函数根据整数索引知道第一个样本正确类是 0、第二个是 1。若把标签写成 `[[0.], [1.]]` 或随意改成 float，可能与当前损失约定不符。查看官方 API 要求并打印 `labels.dtype` 与 `labels.shape`，是解决问题的标准步骤。

L6.3 第三类错误：混用 CPU 和 GPU

有 GPU 时，你可能希望模型和数据都在 GPU 上计算，但必须让**参与同一运算的张量处于兼容设备**。以下示例采用一个设备变量统一管理：

```
import torch                                          # 导入 PyTorch
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu') # 有 CUDA 则用 GPU
model = nn.Linear(4, 2).to(device)                    # 将模型参数移动到指定设备
x = torch.randn(3, 4).to(device)                      # 把输入也移动到同一个设备
print(model(x).shape)                                 # 设备匹配，输出 (3,2)
```

`.to(device)` 不会神奇地把所有 Python 对象都移动到 GPU，它主要用于支持该操作的张量、模型等对象。`device` 也不决定 shape：换设备并不会把 `(3,4)` 变成 `(4,3)`。

L6.4 第四类错误：损失不下降就一直加层

如果训练损失不动，先检查数据是否有真实规律，再检查标签与预测是否正确配对，最后检查 `loss.backward()`、`optimizer.step()` 是否真的被调用。学习率过大可能震荡，过小可能收敛很慢；ReLU 网络初始化不合适也可能出现一些神经元长期不激活的情形。先看训练数据能否被一个简单模型拟合，再决定是否增加复杂度。

你可以试一个**小批次过拟合测试（Overfit a Tiny Batch）**：从数据里取很少量正确标注的样本，让模型反复训练到很低的损失。如果一个足够表达该任务的小模型连几条样本都拟合不了，应优先怀疑训练管线实现错误。这个测试只检查“能否学习”，不是泛化性能保证。

L6.5 第五类错误：对验证集也做梯度更新

标准流程里，训练集用于更新参数；验证集用于观察模型对未参与当前训练优化的数据的表现。写验证循环时，通常先 `model.eval()` 再进入 `with torch.no_grad():`，只计算预测与指标，不执行优化器 `step()`。

当然，训练流程还有许多高级变体，但刚开始应坚持清楚的分工。否则你可能把验证数据变成第二份训练数据，然后错误地相信自己得到了独立评估。

L6.6 12 周学习路线：每天不求多，但要求能复述

下面把微软 AI for Beginners 的主题转换为一个**建议节奏**，不是官方必须遵循的作息表。你正在重点学习前五课，因此可以在第二至第四周适当多花时间，务必先讲清矩阵与梯度，再进入更复杂的架构。

周次	阅读范围	当周至少要会做的事
1	第 1～2 章	分清规则系统、机器学习、深度学习
2	第 3 章 + 实验 01	手算感知机一次预测与一次权重更新
3	第 4 章 + 专题 02/03	手算矩阵乘法，解释导数与链式法则
4	第 5 章 + 专题 01/04/05	逐行讲清 PyTorch 训练循环
5	第 6～8 章 + 实验 03	看懂图片的 NCHW 形状和 CNN 特征图
6	第 9～12 章	分清重建、生成、检测与分割任务
7	第 13～14 章 + 实验 04	亲手制作词袋与词向量形状表
8	第 15～17 章	分清语言模型、RNN 与逐词生成
9	第 18 章 + 实验 05	推导 Q、K、V 相乘后的形状
10	第 19～20 章	解释 NER、LLM 与提示工程的区别
11	第 21～23 章	比较进化搜索、强化学习和多智能体

周次	阅读范围	当周至少要会做的事
12	第 24～25 章 + 实验 06	评估风险，完成一个自己的小项目总结

L6.7 每日学习记录模板

每次读完一个小节，建议用五个句子记录：今天最重要的概念是什么；今天见到哪些中英文术语；哪段代码最难；我亲手算对或运行对了什么；明天要复习哪一个疑问。你也可以把这些字段写进自己的网页学习日志，并标注课程章节、花费时间和自信程度。

示例：**日期**：某日；**章节**：第 4 章；**概念**：`(3,4)@(4,2)=(3,2)`，因为 A 的三行与 B 的两列形成六次点积；**术语**：Dot Product（点积）、Batch（批次）、Feature（特征）；**代码**：`nn.Linear(4,2)`；**疑问**：为什么权重形状是 `(2,4)`；**复习计划**：手写转置后的矩阵乘法。

L6.8 最终毕业清单

请逐项自测，而不是只看阅读进度：

- 能解释深度学习、机器学习与符号 AI 之间的关系。
- 能从 3 行 4 列、4 行 2 列的矩阵中逐格算出结果。
- 能说出 `import`、`from`、`as`、`for`、`def`、`class`、`self` 的最基本含义。
- 能辨认 `nn.Sequential`、`nn.Linear`、`nn.ReLU` 与权重形状。
- 能解释损失、梯度、`backward()`、`optimizer.step()` 之间的区别。

- 能分开训练集、验证集与测试集，指出至少两种数据泄漏风险。
- 能沿着 CNN、RNN 或 Transformer 写一张形状变化表。
- 能区分训练分数与真实泛化能力，发现高准确率的误导。
- 能对一个 AI 项目写出输入、输出、评价指标与安全边界。

其中任何一项还不会，都可以把它当作“下一步学习任务”，而不是当成失败。一本教材的作用不是让你一天读完，而是让你遇到新知识时能随时回到最关键的基础概念。

附录 A｜Python 语法速查：看得懂再写得快

A.1 导入模块 import / from / as

```
import torch                      # 引入名为 torch 的模块，后续写 torch.tensor 等
from torch import nn              # 从 torch 里面直接引入 nn，后续写 nn.Linear
import torch.nn as neural         # as 表示给模块取别名，这里 neural 代表 torch.nn
```

`import` 是动词“导入”，不是某个神经网络功能；`from` 指定来源；`as` 是“起别名”。初学时每见到一次都要读懂，而不是把整行当作神秘咒语。

A.2 变量、列表、函数

```
n = 3                             # = 是赋值，意思是把整数 3 保存到变量 n 中
numbers = [1, 2, 3]               # 方括号创建列表 list，可以按位置存多个元素
first = numbers[0]                # 索引从 0 开始，因此第 0 项是 1

def square(x):                    # def 定义函数；x 是参数，表示传入的值
    return x * x                  # return 表示把 x 的平方作为函数结果返回

answer = square(n)                # 调用 square 函数，把 n 的值 3 传进去
print(answer)                     # 调用内置 print 函数，输出 9
```

`def`、`return`、圆括号与缩进，是学习 PyTorch 前最值得反复熟悉的基础语法。切记 Python 使用缩进确定代码块。

A.3 for 循环与 range

```
for i in range(3):                # range(3) 依次给出 0、1、2；i 是循环变量
    print(i)                      # 每循环一次打印当前 i；这一行因缩进属于循环体
```

这会输出三行：`0`、`1`、`2`。`range(3)` 不包含终点 3。训练循环中的 `for epoch in range(10)` 也是同样道理。

A.4 类与对象的最小认识

`nn.Linear(4,2)` 是从 `nn` 模块调用名为 `Linear` 的类来创建一个层对象。类（Class）像蓝图，对象（Object）是按蓝图创建出来的实例。`model(x)` 之所以能执行，是因为 PyTorch 的模块对象实现了可调用行为，并在内部调用其前向计算逻辑。刚开始不必自己写复杂的类，先把“类、对象、参数、调用”这几个词区分清楚。

附录 B｜矩阵、张量与概率的最低必修课

B.1 把形状读出来

`(3,4)`：一个二维张量，3 行 4 列。如果约定行是样本、列是特征，就是 3 个样本、每个样本 4 个特征。

`(2,3,32,32)`：一个四维张量。在 PyTorch 图像任务常用的 NCHW 约定下，它表示 2 张图片、3 个颜色通道、高 32、宽 32。**相同数字的形状不代表相同语义；要看每个维度的定义。**

常规二维矩阵乘法：`(m,n) × (n,p) = (m,p)`。中间两个 `n` 必须相等，它们是每个点积的配对长度。矩阵不能任意交换乘法顺序；`AB` 与 `BA` 即使都有定义，结果也通常不同。

B.2 标准正态分布

均值（Mean） 体现中心，**标准差（Standard Deviation）** 体现分散程度。标准正态分布写作 `N(0,1)`，意为均值 0、方差（Variance）1，因此标准差也是 1。

经验上的“68–95–99.7 法则”说的是正态分布样本落在均值左右 1、2、3 个标准差内的概率约为 68.27%、95.45%、99.73%。它不是“随机抽几个数就必定满足这些比例”的规则。

B.3 梯度下降为什么需要导数

如果你在山坡上想往低处走，地面的坡度会帮助你判断下降方向。**梯度（Gradient）** 在多维参数空间里起类似作用，表示损失函数对于各参数变化的敏感程度。最简单的更新形式是：

```
新参数 = 旧参数 - 学习率 × 梯度
```

学习率太大会越过低谷甚至发散，太小则可能进展缓慢。反向传播使用链式法则高效地计算大量参数的梯度；它不是让网络“反方向再预测一次”。

附录 C｜随手查：神经网络术语表

中文	English	最简单的理解
样本	Sample	一条输入记录
特征	Feature	描述样本的一个数或属性
张量	Tensor	具有形状的多维数值数组
形状	Shape	每个维度有多长
权重	Weight	模型要学习的乘法系数
偏置	Bias	模型要学习的加法项
线性层	Linear Layer	线性变换加偏置
激活函数	Activation Function	引入非线性变换
前向传播	Forward Propagation	从输入计算预测
损失函数	Loss Function	衡量预测偏差
梯度	Gradient	损失对参数变化的敏感程度
反向传播	Backpropagation	高效计算参数的梯度
优化器	Optimizer	根据梯度更新参数

中文	English	最简单的理解
学习率	Learning Rate	每次更新的步幅
训练集	Training Set	用来学习参数的数据
验证集	Validation Set	用来调整方案的数据
测试集	Test Set	用来做最终泛化评估的数据
过拟合	Overfitting	记住训练数据，泛化较差
嵌入	Embedding	把离散对象映射成稠密向量
注意力	Attention	依据关联程度聚合信息
生成模型	Generative Model	学习生成新数据的模型

附录 D｜全部章节思考题参考答案

第 1 章：不一定。写死阈值的程序是人为指定规则，并未自动从数据学习参数。它可以是更大智能系统中的一个规则模块。

第 2 章：因为规则路径明确，便于逐项审核与复现；但现实条款复杂时，规则维护仍然可能困难。

第 3 章：`z=2*2 + 3*(-1) + 1 = 2`，大于 0，因此输出 1。

第 4 章：形状依次是 `(8,10) → (8,6) → (8,2)`。8 是样本数量，经过逐样本的线性层不改变。

第 5 章：ReLU 只逐元素改变数值，不改变张量中元素的排列与数量；训练好、验证差首先考虑过拟合以及数据分布不一致等可能性。

第 6 章：`100×200×3=60,000` 个通道数值；灰度图通常成为 `(100,200)`。

第 7 章：同一卷积层对每张图片应用相同操作，批量数保持不变；改为 16 个输出通道后得到 `(2,16,32,32)`。

第 8 章：应考虑早停、正则化和数据检查，而非盲目增加训练轮次。

第 9 章：因为有规则的潜在空间有利于采样生成和分布层面的学习，而不仅仅是把见过的输入逐个记住。

第 10 章：可能发生模式崩溃（Mode Collapse）。

第 11 章：需要边界框坐标，才能确定每条检测结果在图中的位置。

第 12 章：浅层特征保留较细的空间信息，能帮助解码器还原边界与局部细节。

第 13 章：词袋统计词出现的次数而忽略顺序，两句话的词频可能完全一样。

第 14 章：有用信息取决于表示能否捕捉相关规律，而不只是向量长度。稠密向量可以通过训练压缩并组织语义特征。

第 15 章：模型在下一个 token 预测任务中，通常要对词表里的许多候选 token 给分数，因此输出维度往往与词表大小相关。

第 16 章：输出改为 `(4,10,7)`，最终隐藏状态改为 `(1,4,7)`，因为隐藏状态的特征数变了。

第 17 章：通常更稳定、更确定；但可能重复或缺少变化。采样通常能带来更多多样性。

第 18 章：词序决定语义角色；没有位置信息，仅知道哪些词出现过是不够的。

第 19 章：NER 的预测单位通常是 token 或文本片段，整句分类的预测单位则是完整句子或文档。

第 20 章：在典型推理使用中，提示示例被当作上下文输入，没有执行参数梯度更新。

第 21 章：它不依赖可微梯度，可在离散组合空间等情况下搜索，但通常需要较多评估。

第 22 章：有些动作当下得不到奖励，却能通向之后的大额奖励；强化学习关注长期累计回报。

第 23 章：各自的局部策略可能互相影响，缺乏协调便可能争抢资源、堵塞或碰撞。

第 24 章：总体指标掩盖不同群体、光照、设备与环境下的差异，还要考虑隐私、误识别后果和人工复核。

第 25 章：因为文字查询向量可以直接与图片向量比较，相似度高的图片能成为候选检索结果。

版权与课程来源说明

本书是参考 **Microsoft / AI for Beginners** 课程主题顺序编写的独立中文学习材料，不是微软授权出版或审定的官方教材，也不构成原始课程的完整替代。原课程包含讲义、可运行的 Jupyter Notebook、测验与实验，本书聚焦概念理解与初学者代码阅读。原项目以 MIT License 发布；进一步学习与核对细节请以原始课程为准。

项目主页：https://github.com/microsoft/AI-For-Beginners

课程目录：https://github.com/microsoft/AI-For-Beginners#content

编写日期：2026 年 10 月。本书适合先作为阅读版入门，随后回到原课程完成真实实验。