大模型原理与应用

第 6 课：大模型工程实践 | 微调、部署与调用

把模型真正调起来、跑起来、用起来

这一课看什么

- 一个开源大模型怎样跑通最小微调闭环
- 为什么量化、LoRA、SFT、GRPO 会同时出现
- 哪些细节最容易把训练和部署搞坏

本课主线

1. 从基座模型、量化和 adapter 开始
2. 从数据格式和 SFT 走到训练流程
3. 从 post-training 走到部署与评测

先建立这节课的工程视角

前几课讲的是原理，这一课讲的是：

> 给你一个开源大模型，怎样在有限显存和有限时间里把它调成你想要的样子？

工程上最常见的路径不是“从头训练一个模型”，而是：

1. 选基座模型
2. 量化加载
3. 加 LoRA / adapter
4. 处理数据格式
5. 训练
6. 保存 adapter
7. 推理 / 评测 / 继续后训练

为什么“调模型”远比想象中麻烦

- 模型很大，显存紧张
- 全参数微调代价高
- 数据格式要求严格
- tokenizer、pad token、chat template 经常踩坑
- 训练完还要考虑保存、部署和调用

所以“会写几行 Trainer 代码”并不等于会做 LLM 工程。

先看一个最小可运行配方

1. 安装 `transformers`、`peft`、`trl`、`datasets`、`bitsandbytes`
2. 加载一个 quantized base model
3. 配置 LoRA
4. 准备数据集
5. 用 `SFTTrainer` 做监督微调
6. 用新 adapter 做生成测试

这几步就是今天很多开源微调 notebook 的标准模板。

第一步：先把基座模型选对

- `microsoft/Phi-3-mini-4k-instruct`
- `Qwen/Qwen2-0.5B-Instruct`
- `Qwen/Qwen2.5-3B-Instruct`

这也说明一个工程原则：

- 不要一上来就追最大模型
- 先选能在当前硬件上稳定跑通的模型

在教学和实验里，3B 左右往往是很实用的起点。

第二步：先把模型装进显存

材料反复强调 4-bit 量化，因为它几乎是消费级 GPU 上的默认做法。

典型配置：

- `load_in_4bit=True`
- `bnb_4bit_quant_type="nf4"`
- `bnb_4bit_use_double_quant=True`
- `bnb_4bit_compute_dtype=torch.float16` 或 `torch.float32`

意义：

- 大幅减少显存占用
- 让原本装不下的模型可以被训练和推理

为什么量化后还不能直接训练

- 量化后的 `Linear4bit` 层节省了内存
- 但这些层本身不适合继续直接更新

所以工程上通常不是“训练量化层本体”，而是：

- 保持基座参数冻结
- 在其上附加可训练的小模块

这就把我们带到了 LoRA。

第三步：再决定怎么低成本微调

LoRA 的工程价值在这里特别明显：

- 不改全部参数
- 只训练少量低秩增量
- 和量化非常好配合

这也是为什么 today's default stack 常常是：

- 4-bit quantization + LoRA + SFTTrainer

即所谓 QLoRA 风格工作流。

LoRA 在代码里怎么配置

典型参数包括：

- `r=16`
- `lora_alpha=32`
- `lora_dropout=0.1`
- `bias="none"`
- `task_type=TaskType.CAUSAL_LM`
- `target_modules=["q_proj", "v_proj"]`

直觉上：

- `r` 决定适配容量
- `target_modules` 决定改哪些层
- `dropout` 是正则

第四步：把数据准备成模型真能吃的样子

工程里一个经常被低估的问题是：

- 数据本身不只是“有文本”
- 而是必须变成模型能正确消费的格式

你通常要处理：

- prompt / response 结构
- system / user / assistant 角色
- 标签位置
- EOS / PAD token
- chat template

数据格式比你想象得更重要

如果数据格式不一致，模型可能学到的是：

- 错误边界
- 错误角色
- 错误输出格式
- 错误停止条件

所以工程上常见做法是：

- 先把原始样本映射成统一 conversation schema
- 再交给 tokenizer 和 trainer

这也是 TRL 在很多例子里重点封装的内容。

第五步：跑通监督微调闭环

`SFTTrainer` 核心实战工具。

它的价值在于：

- 简化 supervised fine-tuning 流程
- 帮你处理 tokenization、batching、label 构造等常见细节

典型配置包括：

- `SFTConfig`
- `SFTTrainer`
- 数据集
- tokenizer
- model

一个典型 SFT 工程流程

把几份材料合起来，可以写成：

1. `load_dataset()`
2. 映射成 prompt / completion 或 chat messages
3. `AutoTokenizer.from_pretrained()`
4. 处理 `pad_token`
5. `AutoModelForCausalLM.from_pretrained()`
6. 接入 `get_peft_model()`
7. 用 `SFTTrainer` 训练
8. 保存 adapter
9. 用生成接口做 sanity check

这就是第一个“能落地”的工程闭环。

为什么训练后一定要马上测

工程习惯：

- 训练完不要只看 loss
- 要直接生成样例看看行为

原因很简单：

- loss 下降不代表输出格式对
- 模型可能过拟合模板
- 可能学会了错误的停词
- 可能根本没学会目标风格

所以 inference sanity check 是必不可少的一步。

保存的通常不是整个模型

在 LoRA 方案下，常见保存物不是整套 3B / 7B / 13B 模型，而是：

- adapter 权重

这样做的好处是：

- 小得多
- 易于版本管理
- 同一个基座模型可以挂很多任务 adapter

这也是 PEFT 真正适合工程协作的原因。

但工程实践不止 SFT

- reasoning post-training
- GRPO

这很关键，因为现实里很多团队的 pipeline 不是“SFT 结束就完”，而是：

- SFT 先让模型学会基本格式与任务
- 再做 post-training 强化某类能力

为什么要做 reasoning post-training

- 数学题、复杂推理题不是普通聊天回答
- 需要更长链条、更稳定的中间推理

所以训练目标不只是“答对一句话”，而是：

- 会按结构思考
- 会输出推理过程
- 会给出最终答案

这就是 reasoning model 后训练的动机。

GRPO 是什么

GRPO = Group Relative Policy Optimization

材料里把它定位为：

- 一种 RL post-training 技术
- 与 DeepSeek-R1 系列 reasoning 训练有关
- 在一些场景下可以替代更传统的 PPO pipeline 的某些复杂部分

工程上你不一定先关心它的理论细节，而要先理解：

- 它是“让模型通过奖励信号学会更好的推理行为”

GRPO 和普通 SFT 的区别

SFT 在做的是：

- 给定标准答案，最大化监督似然

GRPO 在做的是：

- 让模型生成候选
- 用 reward 判断好坏
- 再更新策略

所以它更像：

- “学会追求更优行为”

而不是：

- “逐 token 模仿参考答案”

GRPO 实战里的数据长什么样

- `AI-MO/NuminaMath-TIR`
- `openai/gsm8k`

共同特点是：

- 有问题
- 有答案
- 常常还带详细 reasoning

这些数据非常适合训练：

- step-by-step 解题
- reasoning + final answer 的结构化输出

为什么要先把数据改成对话格式

GRPO 例子里有一个很关键的工程步骤：

- 先把样本映射成带 system prompt 的 conversation

例如要求模型输出：

- `<think> ... </think>`
- `<answer> ... </answer>`

或者：

- `<start_working_out> ... <end_working_out>`
- `<SOLUTION> ... </SOLUTION>`

这说明：

- 结构化输出本身也可以通过 prompt 和 reward 联合塑形

System Prompt 在工程上的作用

这些 GRPO 材料把 system prompt 当成关键组件，而不是附属品。

它的作用包括：

- 规定角色
- 规定输出结构
- 规定 reasoning 区域和 answer 区域
- 降低训练目标的模糊性

对工程实践来说，这意味着：

- prompt design 也是训练设计的一部分

Multi-Reward Training 为什么重要

它用四类奖励同时约束模型：

- 格式合规
- 近似匹配
- 最终答案正确
- 数字抽取

这个思路非常工程化，因为真实系统目标通常不是单一指标。

为什么单一奖励不够

如果只奖励“答案对不对”，可能会出现：

- 格式乱
- 推理缺失
- 输出不稳定

如果只奖励“格式规范”，模型可能：

- 很会套模板，但答不对

所以多奖励的价值在于：

- 同时优化正确性、可解析性和行为一致性

GRPO 工程栈长什么样

典型栈：

- `transformers`
- `datasets`
- `trl`
- `bitsandbytes`
- `peft`
- `trackio`

其中：

- `transformers` 负责模型与 tokenizer
- `datasets` 负责数据集
- `peft` 负责 LoRA
- `bitsandbytes` 负责量化
- `trl` 负责 SFT / GRPO 等训练器
- `trackio` 负责监控

训练前要先看硬件

- CUDA 是否可用
- GPU 数量
- 显存大小

原因：

- 训练方案受硬件上限强约束
- batch size、max seq length、模型大小、precision 都要据此调整

第六课的工程默认范式

1. 选一个可承载的 instruct base model
2. 用 4-bit 量化加载
3. 用 LoRA 做参数高效适配
4. 规范化数据和 chat format
5. 用 `SFTTrainer` 做第一阶段监督微调
6. 如有需要，再用 `GRPOTrainer` 做 reasoning post-training
7. 保存 adapter，并做推理验证

什么时候该用 SFT，什么时候该用 GRPO

可以这样粗分：

适合 SFT

- 风格对齐
- 指令跟随
- 标准任务映射
- 有高质量参考答案

适合 GRPO / RL post-training

- 推理过程重要
- 标准答案之外还有格式和行为要求
- 可以设计 reward
- 希望模型学会更优策略而不只是模仿

工程里最容易踩的坑

最常见的问题其实不是“不会写代码”，而是：

- 模型太大，显存炸掉
- tokenizer / pad token 配错
- 数据列名和 trainer 不匹配
- prompt 模板混乱
- 输出格式不稳定
- 训练指标下降但生成效果差
- reward 设计过于单一

真正的工程能力，就是系统地绕开这些坑。

这节课你该带走什么

1. 第一原则是先跑通最小闭环
2. 量化 + LoRA 是最常见微调组合
3. 数据格式和 prompt 模板同样关键
4. `SFTTrainer` 解决监督微调落地
5. `GRPOTrainer` 代表更进一步的 post-training
6. 目标不是降 loss，而是把行为调稳定

为什么这节课决定你能不能真正落地

“调一个模型”，不应该只是概念，而应该马上联想到：

- 用什么基座
- 多大显存
- 要不要量化
- 用哪种 adapter
- 数据长什么样
- 是 SFT 还是 RL post-training
- 最后怎么调用和验证

这才是大模型工程实践真正的起点。

第 6 课收束

大模型微调不是一个命令，而是一整套工程流水线