大模型原理与应用

1：Chapter 0 TL;DR

从零到一跑通一条最小 LLM 微调链路

这一节到底在干什么

- 不是先系统讲完所有微调理论
- 而是先跑通一条最小闭环
- 让你看到：
  - 模型怎样装进显存
  - 怎样让它变成“可训练但只训练很少参数”
  - 数据怎样整理成 chat / instruction 形式
  - 训练完怎样验证、保存和复用

一句话：

- 这一章是微调地图，不是微调百科全书

对应材料

- 对应 notebook：`11/2-prog/notebook/Chapter0.ipynb`
- 对应参考文章：
  - `11/2-prog/local/huggingface.co/fine-tuning-your-first-large-language-model-llm-with-pytorch-and-hugging-face.md`
- 示例模型：
  - `microsoft/Phi-3-mini-4k-instruct`
- 示例任务：
  - 把英文句子翻成 Yoda 风格表达

先记住整条主线

1. 用量化把基础模型装进单卡显存
2. 用 LoRA 把可训练参数压到极少
3. 把数据整理成模型真正会学的聊天格式
4. 用 `SFTTrainer` 把训练闭环跑起来
5. 用生成测试确认模型真的学到目标风格
6. 保存 adapter 和 tokenizer，保证后续可复用

这条链路里每一步分别负责什么

- quantization：
  - 解决“能不能装进去”
- LoRA：
  - 解决“能不能低成本训练”
- dataset formatting：
  - 解决“模型到底在学什么”
- tokenizer / chat template：
  - 解决“文本怎么变成模型熟悉的 token 结构”
- SFTTrainer：
  - 解决“如何把前面的准备真正跑起来”
- generation / save：
  - 解决“训练结果是否成立、能否复用”

第一步：先加载量化后的基础模型

- 文章里用的是 4-bit quantization
- 核心配置来自 `BitsAndBytesConfig`
- 目标不是提升效果
- 目标是先把模型塞进有限 GPU 显存

代码核心长这样：

```
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_use_double_quant=True,
    bnb_4bit_compute_dtype=torch.float32
)
```

这些量化参数要怎么理解

- `load_in_4bit=True`
  - 把关键线性层用 4-bit 形式加载
- `bnb_4bit_quant_type="nf4"`
  - 使用 NF4 这种常见的 4-bit 表示方式
- `bnb_4bit_use_double_quant=True`
  - 进一步压缩量化常数，继续省显存
- `bnb_4bit_compute_dtype=torch.float32`
  - 计算时使用 FP32

这里最值得注意的是：

- 量化后的模型能很好做 inference
- 但不能直接像普通全精度模型那样继续训练全部权重

为什么文章先打印 `get_memory_footprint()`

- 因为“看上去已经量化了”不等于“真的够省内存”
- 打印 footprint 是最直观的 sanity check
- 文章里的量化后模型仍然占了大约 `2.2 GB`

所以这一节要建立的直觉是：

- 3.8B 模型即使量化了，也不是“几乎不要显存”
- 只是从“普通单卡很难装”变成“有机会装进去”

还要看模型结构，为什么

- 因为你后面要给 LoRA 指定 `target_modules`
- 而 `target_modules` 不是凭感觉写
- 应该从实际模型结构里看：
  - 哪些层已经变成 `Linear4bit`
  - 它们的模块名到底叫什么

在这个例子里，关键层是：

- `o_proj`
- `qkv_proj`
- `gate_up_proj`
- `down_proj`

这一阶段最容易搞错什么

- 误以为“量化后就能直接 full fine-tuning”
- 误以为“显存够装下就一定能训练”
- 误以为“只要 `from_pretrained()` 成功就说明后面不会 OOM”

更准确的理解是：

- 量化只解决了第一关：装载
- 训练时还会叠加：
  - optimizer state
  - activations
  - gradients
  - batch / sequence length 带来的额外压力

第二步：给量化模型接上 LoRA

- 量化层本身不方便继续更新
- 所以常见做法不是去改量化权重
- 而是在这些层外挂小型 adapter
- 真正训练的是这些新增的小矩阵

这就是 LoRA 的工程价值：

- 大部分基座冻结
- 只训练少量新增参数
- 把“几乎训不动”的问题改造成“可以在单卡上尝试”

LoRA 配置里要看哪些参数

```
config = LoraConfig(
    r=8,
    lora_alpha=16,
    bias="none",
    lora_dropout=0.05,
    task_type="CAUSAL_LM",
    target_modules=['o_proj', 'qkv_proj', 'gate_up_proj', 'down_proj'
)
```

- `r`
  - adapter rank，越小参数越少
- `lora_alpha`
  - 缩放系数，文中给的是 `2*r`
- `lora_dropout`
  - LoRA 路径上的 dropout
- `task_type="CAUSAL_LM"`
  - 明确这是自回归语言模型任务
- `target_modules`
  - 指定把 LoRA 接到哪些层上

为什么 target_modules 特别重要

- 对一些成熟、常见模型，peft 可能会自动识别目标层
- 但对较新的模型，这一步常常不稳定
- 文中专门提醒：
  - 如果你看到
  - `ValueError: Please specify target_modules in peft_config`
  - 就应该回到模型结构里手动查层名

这其实是在提醒你：

- 不要把模型结构当黑盒
- 你至少要知道 LoRA 接到了哪里

`prepare_model_for_kbit_training()` 在做什么

- 它不是“可有可无的装饰步骤”
- 它是量化模型训练前的重要准备
- 主要目的是提升训练数值稳定性

但它也带来一个很现实的副作用：

- 非量化层会被转成更高精度
- 所以模型整体 footprint 会变大
- 文中从约 `2206 MB` 增加到约 `2651 MB`

这说明：

- LoRA 省的是“可训练参数”
- 不代表训练前准备完全不涨内存

为什么要打印 trainable parameters

- 因为这能直接证明 LoRA 是否真的生效
- 文中的结果大概是：
  - trainable `12.58M`
  - total `3833.66M`
  - 占比 `0.33%`

这组数字背后的工程意义是：

- 你不是在训练整个 3.8B 模型
- 你只是在训练一小撮新参数

第三步：整理数据，不只是“把文本读进来”

- 文中数据集是 `dvgodoy/yoda_sentences`
- 总共 `720` 条
- 包含三列：
  - `sentence`
  - `translation`
  - `translation_extra`

这里最重要的不是语料大小

- 而是监督信号的组织方式

模型不是直接学“文件里的字符串”

- 它学的是你喂给它的 token 序列结构

为什么作者选 `translation_extra`

- 因为它比基础翻译更像“目标风格”
- 不只是语序变换
- 还加入了 `Yes, hrrrm.` 这种 Yoda 式口头禅

这提醒我们：

- 微调数据不只是“答对”
- 还决定模型最后会学到怎样的输出风格
- 你想让模型学到什么，就必须在监督样本里明确给出来

这里最大的坑：TRL 新版本对数据格式的变化

- 旧思路里，很多人会把数据写成 `prompt/completion`
- 但文章明确提醒：
  - 新版 `trl` 对 instruction 格式支持不再稳定
  - chat template 可能不会按预期应用

所以作者做了一个关键调整：

- 把数据转成 conversational format

也就是：

```
{"messages": [
  {"role": "user", "content": "..."},
  {"role": "assistant", "content": "..."}
]}
```

为什么一定要转成 conversational format

- 因为后面的 tokenizer 有 chat template
- chat template 期望的输入就是一组带 `role` 的 messages
- 如果格式不对：
  - special tokens 可能不对
  - user / assistant 边界可能不对
  - generation prompt 可能不对
  - 最终训练目标就会偏掉

所以这里不是“换个数据结构而已”

- 而是在保证训练时文本结构和模型原生对话格式一致

`format_dataset()` 真正在干什么

- 把旧的：
  - `prompt`
  - `completion`
- 转成新的：
  - `messages = [user, assistant]`

这一层转换虽然简单，但很关键：

- 它决定 trainer 看到的到底是普通文本
- 还是“角色明确、边界清楚”的对话样本

更通用的经验是：

- 任何 instruction tuning
- 最好都先问一句：
  - 我的数据结构，和目标模型的 chat template 兼容吗

第四步：tokenizer 不是附属品，而是训练协议的一部分

- `AutoTokenizer.from_pretrained(repo_id)` 不是顺手一写
- 它必须和基础模型对应
- 因为 tokenizer 决定：
  - 文本如何切 token
  - 特殊 token 是什么
  - chat template 是什么
  - 生成时该如何拼 prompt

如果 tokenizer 和模型不匹配：

- 训练能跑也可能学歪
- 推理格式更容易直接错掉

chat template 到底决定了什么

- system / user / assistant 各自放在哪里
- 每段消息的起止 token 是什么
- 一轮对话结束时用什么结束
- 推理时怎样追加 generation prompt

文中的 Phi-3 模板会生成类似：

```
<|user|>
...<|end|>
<|assistant|>
...<|end|>
<|endoftext|>
```

你要有这个意识：

- 训练文本不是“自然语言原文”
- 而是“自然语言 + chat protocol”

文章里一个非常关键的坑：pad_token 不能直接沿用 eos_token

- 文中提醒：
  - 在新版 SFTTrainer 默认 collator 下
  - 如果 PAD 和 EOS 是同一个 token
  - label mask 可能把 EOS 也一起遮掉
- 后果是：
  - 模型学不会正确停下来
  - 推理时更容易一直生成不停

所以作者做了：

```
tokenizer.pad_token = tokenizer.unk_token
tokenizer.pad_token_id = tokenizer.unk_token_id
```

这一步为什么特别值得记

- 很多教程只教“怎么训起来”
- 但不解释“为什么模型停不下来”
- 这里其实就是一个经典的：
  - 数据 collator
  - label mask
  - special token
  - generation behavior 之间的连锁问题

所以你以后如果看到模型：

- 会答
- 但不爱停

先别急着怪采样参数

- 也要检查 tokenizer / pad / eos / labels 的关系

第五步：进入 SFTTrainer

- SFTTrainer 是这条链路的执行器
- 但它不是魔法黑盒
- 它只是把你前面准备好的：
  - model
  - tokenizer
  - dataset
  - training config 真正接起来

所以真正重要的是：

- 你在喂给 SFTTrainer 之前
- 前面每一层语义都已经对了

SFTConfig 要按四组参数去理解

1. 显存相关
    - gradient_checkpointing
    - gradient_accumulation_steps
    - per_device_train_batch_size
    - auto_find_batch_size
2. 数据相关
    - max_length
    - packing
    - packing_strategy
3. 训练相关
    - num_train_epochs
    - learning_rate
    - optim
4. 环境与日志
    - output_dir

这里最容易 OOM 的不是 learning rate，而是 sequence length

- 文中明确提醒：
  - 对单卡微调来说
  - 比起先纠结学习率
  - 更应该先盯住 `max_length`
- 因为序列长度一变长：
  - attention 开销会上去
  - activation 内存会上去
  - batch 能装下的样本数会明显下降

这个例子里句子很短

- 所以 `max_length=64` 就够了

这是非常重要的经验：

- 不要为了“保险”把序列长度设很大
- 过大的 `max_length` 是最常见的浪费显存方式之一

`packing=True` 是什么意思

- packing 的意思是：
  - 把多条短样本拼接进同一个 sequence
  - 尽量减少 padding 浪费
- 对这种短句数据特别有价值
- 因为单条样本太短，不 packing 会很浪费 token budget

文中还专门加了：

- `packing_strategy='wrapped'`

原因是：

- `trl` 新版本里 packing 行为变了
- 这个参数是为了更接近旧版本预期

版本变化是这一章里必须特别警惕的事

文章里至少明确点了三类版本相关变化：

- `trl>=0.20`
  - `max_seq_length` 改名为 `max_length`
- `trl>=0.20`
  - packing 行为改变
- 新版默认 `bf16=True`
  - 但并不会总是先检查硬件是否真支持

所以作者写了：

```
bf16=torch.cuda.is_bf16_supported(including_emulation=False)
```

这其实是在教你一个态度：

- 不要把教程里的默认参数直接当成永远成立的事实

gradient_checkpointing 为什么这么常见

- 因为它非常省显存
- 代价是会增加一些计算开销
- 对单卡微调来说通常很值

文中还加了：

```
gradient_checkpointing_kwargs={'use_reentrant': False}
```

这不是“多余参数”

- 而是为了避免新版本 PyTorch 下的一些异常

这类参数的共同特点是：

- 它们不是算法核心
- 但经常决定你能不能顺利把代码跑完

`auto_find_batch_size=True` 是什么思路

- 不是让程序自动帮你找到最优配置
- 而是当当前 batch size OOM 时
- 自动往下缩

这对教学示例很实用

- 因为不同同学显卡差异很大

但你要知道：

- 它是兜底，不是调参策略
- 最终还是应该理解：
  - 长度
  - batch
  - accumulation
  - precision 之间的平衡关系

SFTTrainer 这里还有一个真实坑

- 文中指出：
  - 当前版本 `trl` 有一个已知问题
  - 如果 LoRA 已经先应用到 model 上
  - trainer 可能把 adapter 也一起 freeze 掉

所以更稳妥的写法是：

```
trainer = SFTTrainer(
    model=model.base_model.model,
    peft_config=config,
    processing_class=tokenizer,
    args=sft_config,
    train_dataset=dataset,
)
```

重点有两个：

- 把底层原始模型传进去
- 把 `peft_config` 交给 trainer 来应用

为什么这里不是直接传 `model`

- 因为文章不仅要训练成功
- 还要保证后面 `save_model()` 行为正确
- 如果对象层级传错：
  - 训练可能表面能跑
  - 但保存 adapter 时可能出现不符合预期的结果

这是很典型的工程问题：

- “能跑”和“后续可保存、可复现、可发布”
- 不是同一层要求

为什么作者还去看 dataloader 里的 batch

- 因为这是检查 trainer 预处理是否符合预期的最好方式之一
- 它能回答这些问题：
  - token 是不是已经拼好了
  - labels 是不是自动生成了
  - input 和 label 的关系是什么

文章里看到的是：

- `labels` 和 `input_ids` 一样

这说明：

- 这是 causal LM 的自回归监督
- label shift 会在内部自动处理

这一点要特别讲清楚：SFT 里为什么 `labels == input_ids`

- 对自回归语言模型来说
- 训练目标通常就是“预测下一个 token”
- 所以输入和监督目标来自同一串 token
- 真正的“错位”发生在 loss 计算内部

因此：

- 你看到 `labels` 和 `input_ids` 一样
- 不代表训练有问题
- 这是正常现象

但要进一步追问：

- 哪些 token 被 mask 了
- 哪些 token 会参与 loss

这又回到了前面的：

训练时到底该看什么

- loss 是否持续下降
- 有没有一开始就 nan
- 有没有频繁 OOM / 重启
- 训练耗时是否符合你的机器能力

文中给的 loss 从大约 `2.99` 降到 `0.25`

- 这说明模型至少在训练集目标上明显收敛了

但你要记住：

- loss 下降不等于任务真的学会
- 训练后还必须做 generation test

第六步：生成验证不是装饰，而是闭环的一半

- 微调结束后，不能只看 loss
- 还要真的喂一句新输入进去
- 看输出是否符合目标风格

这里的关键不是复杂 benchmark

- 而是先做最小 inference sanity check

文中的 prompt 生成函数做了两件重要事情：

1. 把输入句子包装成 `messages`
2. 用 `apply_chat_template(..., add_generation_prompt=True)` 拼成模型熟悉的 prompt

`add_generation_prompt=True` 为什么重要

- 因为模型需要一个明确的“现在轮到 assistant 说话了”的触发信号
- 在这个模板里，它会在最后追加：

```
<|assistant|>
```

没有这一步，模型看到的只是历史对话

- 不一定知道你现在要它继续生成回答

这也是为什么：

- 训练格式
- 推理格式
- 必须共享同一种 chat protocol

生成函数里有哪些细节值得学

- `add_special_tokens=False`
  - 因为 chat template 已经把特殊 token 加进去了
- `model.eval()`
  - 推理前切到 eval 模式
- `torch.autocast(...)`
  - 如果模型是半精度 / bf16，生成时自动处理精度上下文
- `eos_token_id=tokenizer.eos_token_id`
  - 明确何时停止生成
- `max_new_tokens`
  - 限制新增长度，防止输出无限拖长

这些都不是“可有可无的样板”

- 它们直接影响生成行为是否稳定

如何判断这个最小微调是成功的

不是只看“模型输出了东西”

至少要看三件事：

1. 输出格式对
    - 说明 prompt / template / tokenizer 没坏
2. 输出内容开始呈现目标风格
    - 说明监督信号被学到了
3. 输出能自然结束
    - 说明 eos / pad / label masking 没搞坏

文中的示例输出：

- 已经明显变成 Yoda 风格
- 而且能在合适位置结束

第七步：保存 adapter，不是保存整个基础模型

- LoRA 微调最有价值的一点
- 就是最后保存的不是完整大模型副本
- 而是一份很小的 adapter

文中说明：

- adapter 本体大约只有 `50 MB`

这意味着：

- 你能便宜地训练
- 也能便宜地分发
- 别人只要有同一个 base model，就能重新挂载你的 adapter

保存后应该包含哪些文件

- `adapter_config.json`
- `adapter_model.safetensors`
- `training_args.bin`
- tokenizer 相关文件：
  - `tokenizer.json`
  - `tokenizer.model`
  - `tokenizer_config.json`
  - `added_tokens.json`
  - `special_tokens_map.json`
- README

这里一个常见误区是：

- 只保存 adapter 权重
- 却忘了 tokenizer 和 special tokens 也是这次训练协议的一部分

为什么还要提 `push_to_hub()`

- 因为完整工程闭环不只包含“本机训完”
- 还包括：
  - 保存
  - 分享
  - 复用
  - 在别的环境重新加载

`output_dir` 在这里不只是日志目录

- 它还会影响你 push 到 Hub 后的模型命名

所以这类路径参数最好一开始就取清楚

这一章最容易踩的坑，集中总结

- `target_modules` 不匹配真实模型结构
- `trl` 新版本导致 instruction 格式不再按预期工作
- `pad_token` 和 `eos_token` 共用，导致模型不容易停
- `max_length` 设太大，训练频繁 OOM
- 先把 LoRA 应到模型上，再错误地传给 `SFTTrainer`
- 只看 loss，不做 generation test
- 只保存 adapter，不保存 tokenizer / special tokens

如果你要把这个例子迁移到自己的任务

你至少要重做四件事：

1. 换 base model
    - 同时重新检查量化层名字和 `target_modules`
2. 换 dataset
    - 明确 prompt / response 到底怎么组织
3. 换 chat template 对齐方式
    - 不同模型协议不同
4. 重估 `max_length`
    - 不要照抄 64

最不应该照抄的，恰恰是这些“看起来最像配置项”的地方

这部分你真正该带走什么

- 微调不是一个 API 调用，而是一条工程链路
- 量化负责装载，LoRA 负责可训练，chat template 负责可学习
- `SFTTrainer` 只是执行器，不会替你自动纠正前面每一层语义错误
- 真正决定成败的，往往是：
  - 数据格式
  - special tokens
  - 版本差异
  - sequence length
  - 保存与复用边界

1 收束

- 对应原书：Chapter 0
- 对应 notebook：`11/2-prog/notebook/Chapter0.ipynb`
- 对应参考文章：
  - `11/2-prog/local/huggingface.co/fine-tuning-your-first-large-language-model-llm-with-pytorch-and-hugging-face.md`

如果只记一句话：

- 最小微调闭环不是“加载模型然后训练一下”，而是：
  - 量化装载
  - LoRA 适配
  - 数据格式对齐
  - tokenizer / template 对齐
  - SFT 执行
  - 生成验证
  - adapter 保存