一．训练 LLM 的三个步骤

①Pretraining 预训练 ②Supervised Fine-Tuning 监督微调 ③Preference Tuning 偏好微调

创建高质量 LLM 的常见步骤有三个：

1. Language modeling：第一步是在一个或多个海量文本数据集上对其进行预训练（图 12-1）。在训练过程中，它尝试预测下一个 token，以准确学习文本中的语言和语义表示。正如我们之前在第 3 章和第 11 章中看到的，这称为语言建模，是一种自监督方法。这会产生一个基础模型 base model，通常也称为预训练模型 pretrained model 或基础模型 foundation model。基础模型是训练过程的关键，但用户很难使用。这就是为什么需要对模型进一步处理。

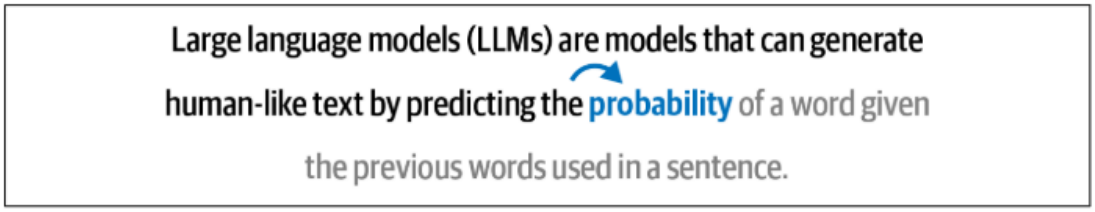

Figure 12-1. During language modeling, the LLM aims to predict the next token based on an input. This is a process without labels.

2. Fine-Tuning 1(supervised fine-tuning，SFT)监督微调

   这一步的目的是使大模型能够遵循用户的指令(instruction)。当人类要求模型撰写文章时，他们希望模型生成文章，而不是列出其他指令（这是基础模型可能会做的）。通过监督微调，我们可以微调基础模型以遵循指令。在此微调过程中，基础模型的参数会更新，使其更符合我们的目标任务。与预训练模型一样，它是通过预测下一个 token 进行训练的，但它不仅仅只预测下一个令牌，还要根据用户输入进行预测（图 12-2）。SFT 还可用于其他任务，比如分类。

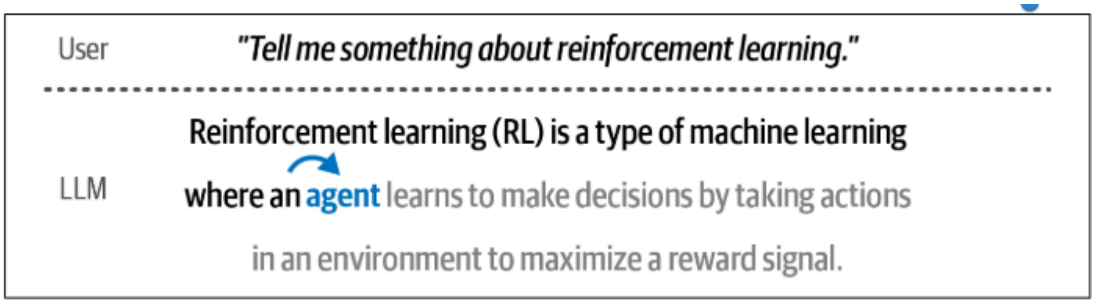

Figure 12-2. During supervised fine-tuning, the LLM aims to predict the next token based on an input that has additional labels. In a sense, the label is the user's input.

3. Fine-Tuning 2(preference tuning)偏好微调

偏好微调进一步提高了模型的质量，使其更符合人工智能安全或人类偏好的预期行为。偏好调整顾名思义，它使模型的输出与我们的偏好保持一致，这些偏好由我们提供的数据定义。与 SFT 一样，它可以改进原始模型，但具有在训练过程中提炼输出偏好的额外好处。这三个步骤如图 12-3 所示，并演示了从未经训练的架构开始到以偏好微调结束的过程。

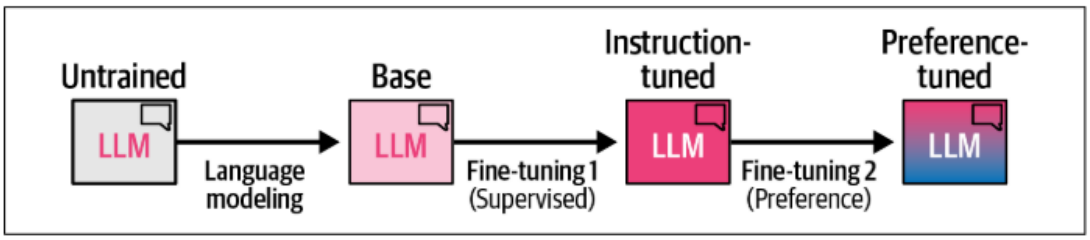

Figure 12-3. The three steps of creating a high-quality LLM.

在本章中，我们使用已经在海量数据集上训练完毕的基础模型，探索如何使用这两种微调方法对其进行微调。对于每种方法，我们都会从理论基础开始，然后再在实践中使用它们。

二．监督微调 Supervised Fine-Tuning(SFT)

当模型还未被训练为能够遵循指令，模型通常会尝试完成问题而不是回答问题。

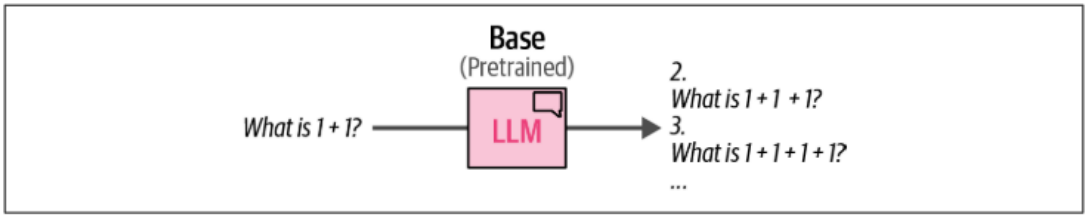

Figure 12-5. A base LLM will not follow instructions but instead attempts to predict each next word. It may even create new questions.

1. Full Fine-Tuning 全参数微调

全参数微调是最常见的微调方法。与预训练 LLM 类似，此过程设计更新模型的全部参数使其与目标任务保持一致。主要区别在于微调使用有标签的数据集，预训练过程是在没有任何标签的大型数据集上完成的。

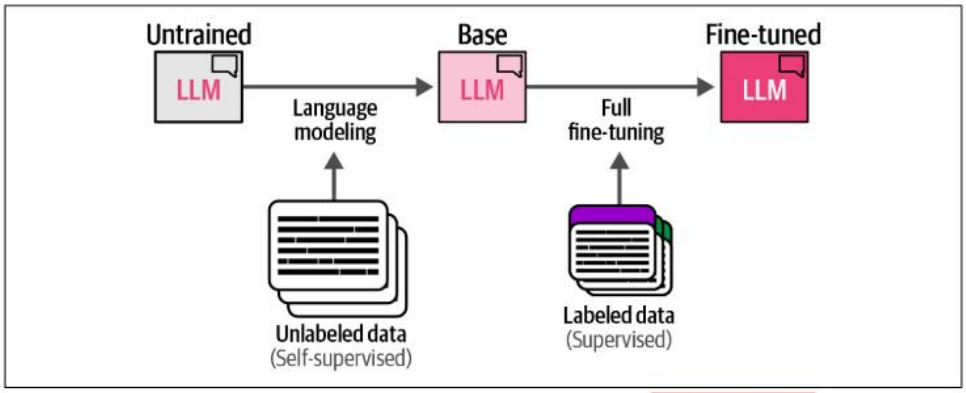

Figure 12-6. Compared to language modeling (pretraining), full fine-tuning uses a smaller but labeled dataset.

为了使我们的 LLM 遵循指令，我们需要 question-response data(问题-回答数据集)。该数据如图 12-7 所示，是用户提出的问题及其相应答案。

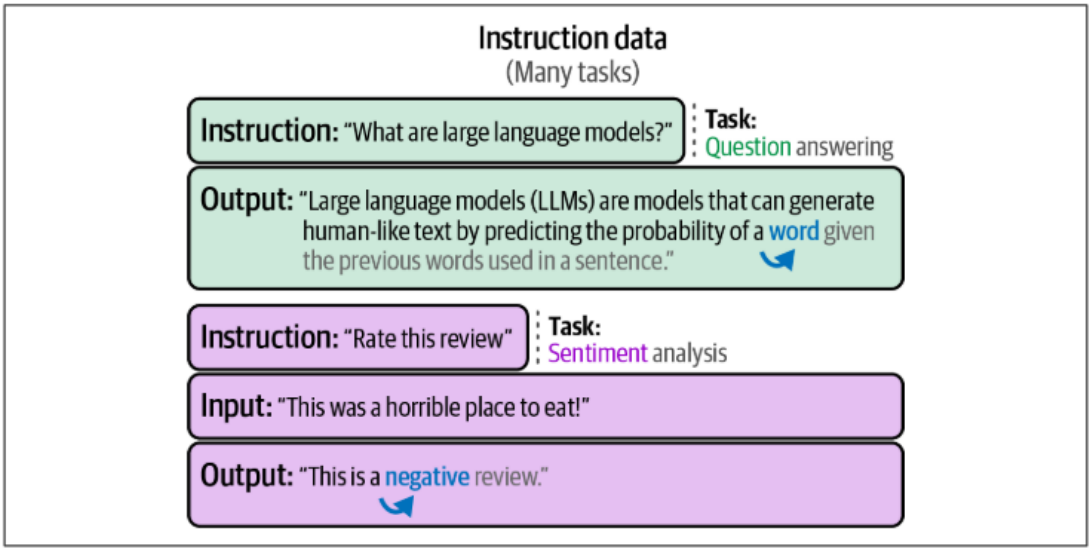

Figure 12-7. Instruction data with instructions by a user and corresponding answers. The instructions can contain many different tasks.

在全参数微调过程中，模型将输入（指令）纳入参考并应用于输出 next-token prediction。因此，它不会生成新问题而是遵循指令。

2. Parameter-Efficient Fine-Tuning (PEFT) 参数高效微调

更新模型的所有参数具有提高其性能的巨大潜力，但也有缺点。它的训练成本高昂，训练时间慢，并且需要大量内存空间。为了解决这些问题，人们开始关注参数高效微调（PEFT），这些替代方案专注于以更高的计算效率微调预训练模型。

① Adapters 适配器

适配器是许多基于 PEFT 的技术的核心组件。该方法在 Transformer 中提供了一组附加的模块化组件，可以对这些组件进行微调，以提高模型在特定任务中的性

能，而不必微调所有模型的权重。这节省了大量的时间和计算成本。

适配器技术首次发表于论文《NLP 参数高效迁移学习》，该研究证明针对特定任务仅微调 BERT 模型 3.6%的参数，即可达到与全参数微调相当的性能。在 GLUE 基准测试中，适配器方法的性能与完整微调仅存在 0.4%的微小差距。如图 12-8 所示，该论文提出的架构在 Transformer 单个模块中，将适配器模块放置在注意力层和前馈神经网络之后。

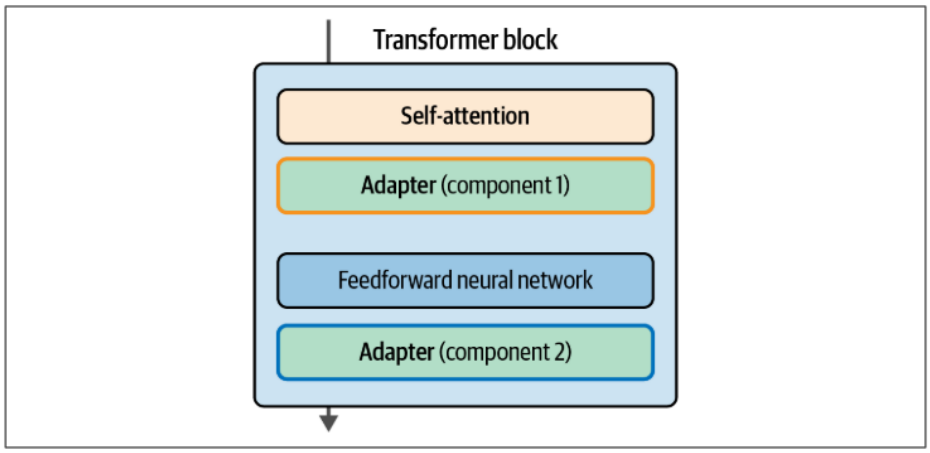

Figure 12-8. Adapters add a small number of weights in certain places in the network that can be fine-tuned efficiently while leaving the majority of model weights frozen.

然而，仅更改一个编码块是不够的，因此这些组件是模型中每个编码块的一部分，如图 12-9 所示。

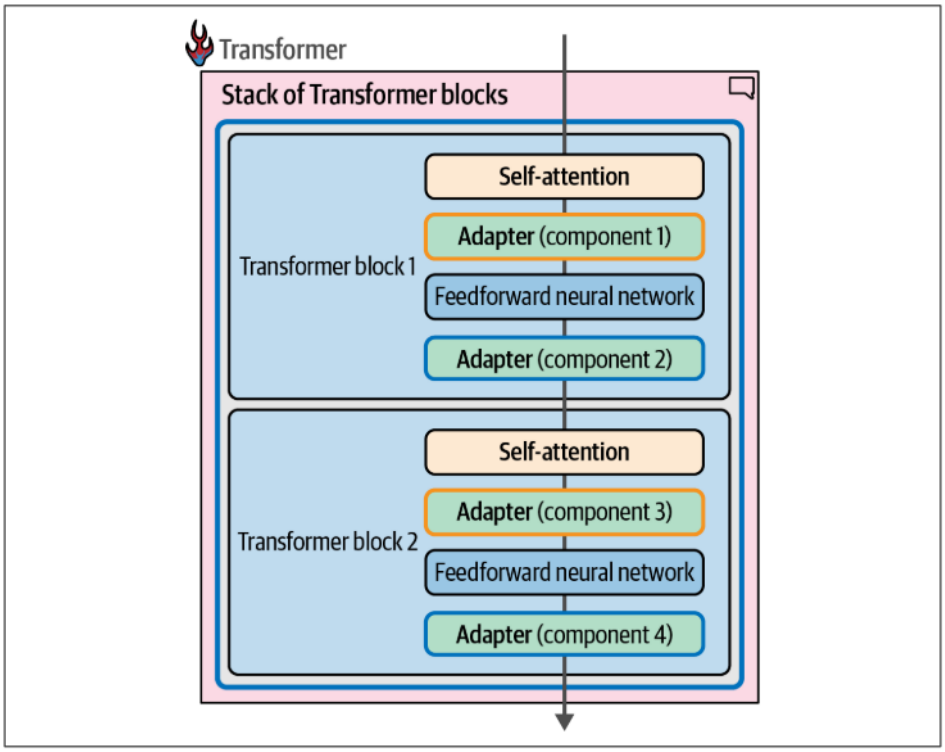

Figure 12-9. Adapter components span the various Transformer blocks in the model.

如图 12-10 所示，将每个编码块中的 adapter 组件提取组成单独的 adapter。通过这种方式观察模型中所有适配器组件，我们可以清晰地看到单个适配器的结构，这些组件横跨模型的全部区块。针对不同的任务可以训练不同的适配器，例如 adapter 1 可专门处理医疗文本分类任务，而 adapter 2 可专注于命名实体识别（NER）任务。用户可通过 https://oreil.ly/XraXg 下载专用适配器。

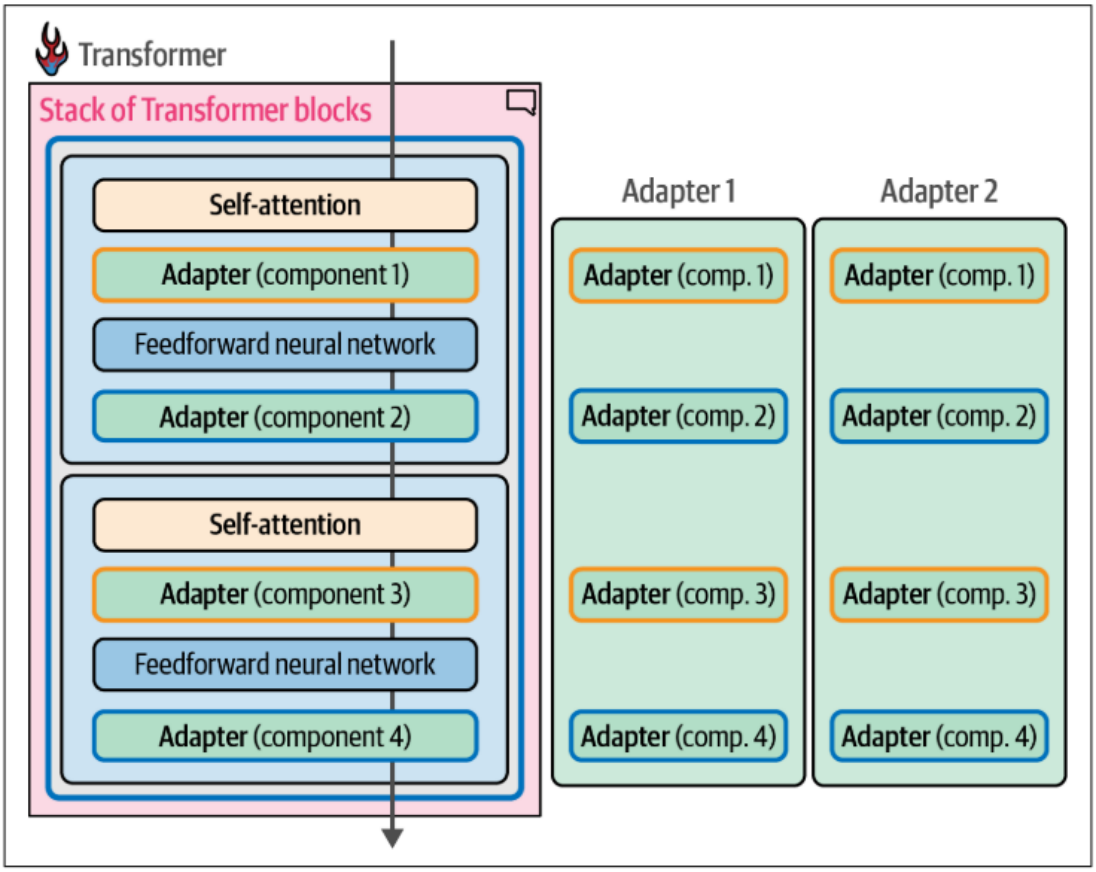

Figure 12-10. Adapters that specialize in specific tasks can be swapped into the same architecture (if they share the same original model architecture and weights).

论文《AdapterHub: A framework for adapting transformers》提出了一个名为 **"Adapter Hub"** 的框架。这个框架就像一个中心仓库或应用商店，研究人员和开发者可以在这里分享、存储和获取为不同任务训练好的适配器。然而，早期的这些适配器大部分主要是为 BERT 这类**自然语言理解（NLU）**模型架构设计和使用的。

后来，适配器的概念得到了进一步的发展和应用范围的扩展。特别是在文本生成领域，新的研究（例如论文《LLaMA-Adapter: Efficient fine-tuning of language models with zero-init attention》）成功地将适配器技术应用到了像 LLaMA 这样的

文本生成类 Transformer 模型上。这项工作还引入了一种名为“零初始化注意力”的新技术，使得微调更加高效。

BERT 和 LLaMA 的区别：

BERT 是一个**理解者**。它擅长分析语言，比如做分类、问答、情感分析。LLaMA 是一个**生成者**。它擅长创造语言，比如写文章、对话、编程、翻译。BERT 的训练数据规模是 16GB，而 LLaMA 的训练数据规模是万亿（Token）级别的，比 BERT 大了几个数量级。

BERT 输出的一句话的嵌入向量，不能直接生成自然语言，可以在 BERT 基础上加上分类层来做各种 NLU 任务。它只使用 Transformer 模型中的编码器部分。其核心创新是**双向注意力机制**。在训练时，它会随机掩盖（Mask）输入句子中的一些词，然后让模型根据所有上下文的词来预测被掩盖的词是什么。这使得 BERT 能深刻理解每个词在上下文中的含义。

LLaMA 输出的是序列化的文本，给它 prompt 就能生成文本。它只使用 Transformer 模型中的**解码器**部分（类似于 GPT 系列）。它采用**因果注意力机制**，也称为**自回归**，只能看到当前词及之前的词，防止“偷看”未来。它被训练来预测序列中的下一个词。在生成时，它从左到右逐词产生输出，每个新词的生成都依赖于之前已经生成的所有词。这非常像我们人类写字或说话的过程。

② Low-Rank Adaptation (LoRA)

作为适配器的替代方案，引入了低秩适配（LoRA），这是一种广泛使用且有效的 PEFT 技术。LoRA 是一种（与适配器一样）只需要更新一小部分参数的技术。它会创建基础模型的一个较小子集以进行微调，而不是向模型添加额外的层。

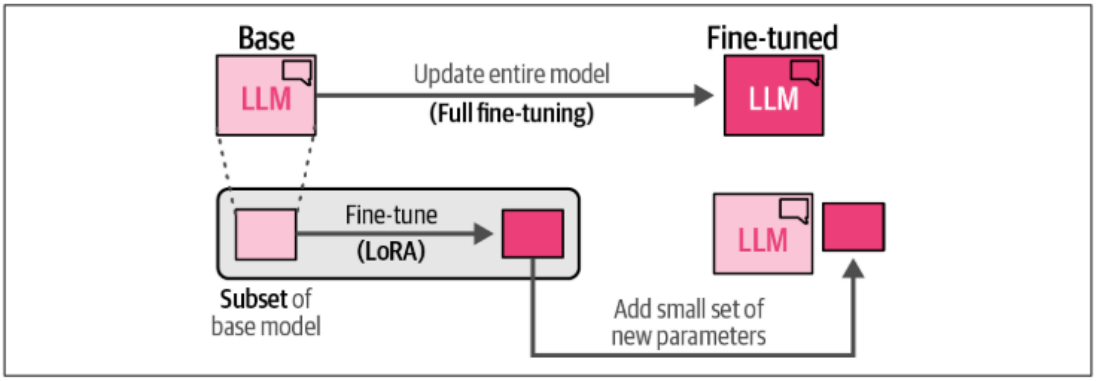

Figure 12-11. LoRA requires only fine-tuning a small set of parameters that can be kept separately from the base LLM.

与适配器（Adapters）技术一样，这里所选的参数子集也能实现快速微调，因为我们只需要更新基础模型的一小部分参数，而不是全部。我们创建这个参数子集的方法是：**用更小的矩阵来近似模拟（或分解）**原始大语言模型中那些大型的权重矩阵（例如 Attention 或 FFN 层中的矩阵）。这里描述的就是 LoRA 技术，它基于一个数学假设：模型在适应新任务时，权重矩阵的更新（ΔW）其实是一个**低秩（Low-Rank）** 矩阵。这意味着这个复杂的更新过程可以用两个更小矩阵（B 和 A）的乘积（ΔW＝B*A）来高效近似。在微调过程中，我们**冻结（freeze）** 原始的大型权重矩阵（W），不让它们更新。然后，我们只训练新引入的这两个小矩阵（B 和 A）。在推理时，将两个小矩阵的乘积（BA）加到原始权重上（$W + \Delta W = W + BA$），就得到了适应新任务后的模型效果。由于 B 和 A 非常小，需要训练的参数数量就变得极少，因此微调速度非常快，占用的内存也少。例如，LLM 矩阵是 10×10，我们可以用两个小矩阵 10×2 和 2×10 来近似它的更新过程。这样，需要训练的参数就从原始的 100 个减少到了 40 个。

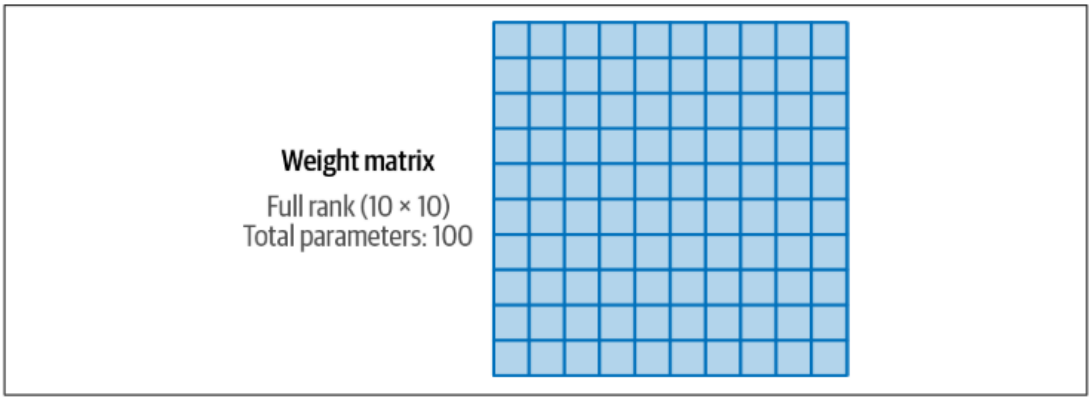

Figure 12-12. A major bottleneck of LLMs is their massive weight matrices. Only one of these may have 150 million parameters and each Transformer block would have its version of these.

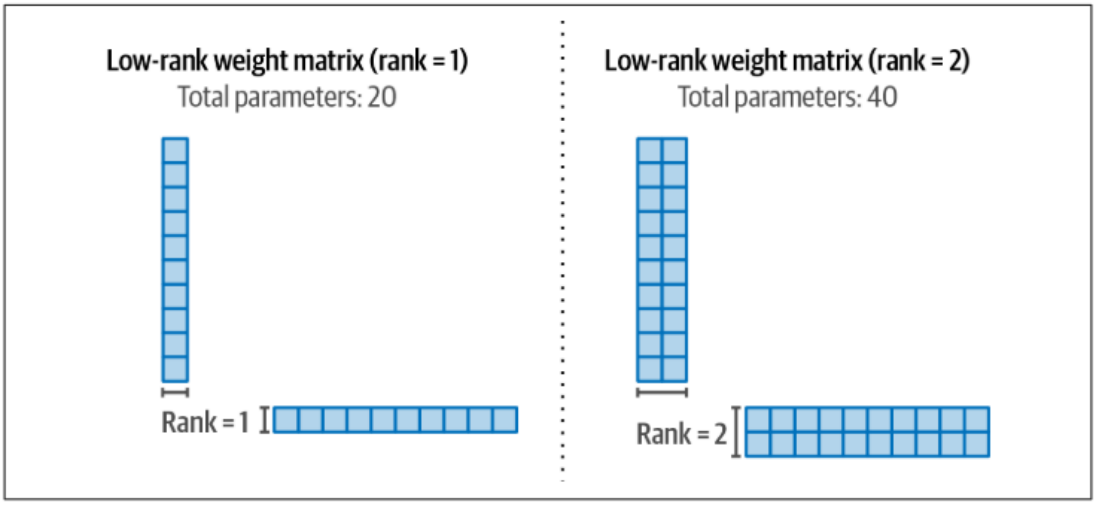

Figure 12-13. Decomposing a large weight matrix into two smaller matrices leads to a compressed, low-rank version of the matrix that can be fine-tuned more efficiently.

然后，将两个小矩阵与完整（冻结）的权重相结合。

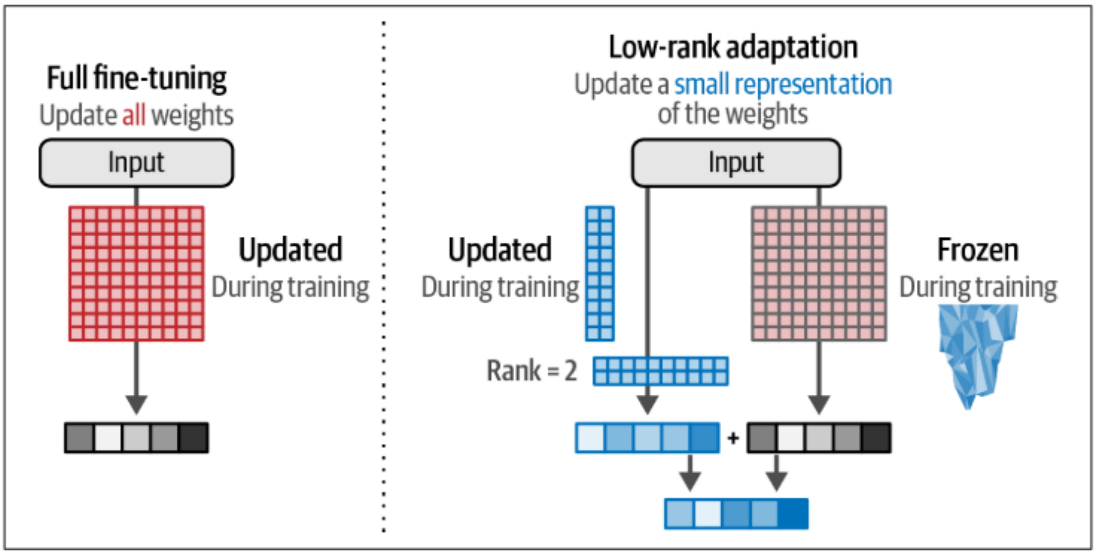

Figure 12-14. Compared to full fine-tuning, LoRA aims to update a small representation of the original weights during training.

但你可能会怀疑这样做性能会下降。没错，你的怀疑是对的。但这种权衡在什么场景下是合理的呢？

《Intrinsic dimensionality explains the effectiveness of language model fine-tuning》等论文研究表明，语言模型"具有非常低的内在维度"。这意味着虽然语言模型有成千上万亿个参数（非常高维），但驱动它学习新任务所**真正需要调整的“核心维度”其实非常低**。我们能够找到极小的秩（rank）来近似模拟 LLM 的庞大的矩阵。例如，一个拥有 1750 亿参数的 GPT-3 模型，其每个 Transformer 模块（共 96 个）内部都包含一个 12,288×12,288 的权重矩阵——单是这一个矩阵就涉及 1.5 亿个

参数。如果我们能成功将该矩阵适配到秩为 8 的表示，则仅需使用两个 12,288×8 的矩阵，最终每个模块只需 19.7 万个参数，参数减少了 99.87%。正如先前引用的 LoRA 论文所述，这种方法能在速度、存储和计算资源方面实现显著节约。这种低维表示具有高度灵活性，您可以自主选择基础模型中需要微调的部分。例如，我们可以仅针对每个 Transformer 层中的查询（Query）和值（Value）权重矩阵进行微调。

③ Compressing the model for (more) efficient training 压缩模型以进行（更）高效的训练：Quantized LoRA (QLoRA)

LoRA 技术虽然需要训练的参数量少了，但**模型本身在训练时仍然需要被加载到显存中**。对于非常大的模型（如 65B 参数），即使只是加载成 FP16（2 字节/参数），也需要 130GB+ 的显存，这远远超过了大多数研究者和公司的硬件能力。QLoRA 就是为了解决 LoRA 的上述局限而诞生的。它的核心创新是**将量化技术引入到 LoRA 中**，从而进一步压榨显存。

LLM 的权重是具有给定精度的数值，可以用 Float 64、Float 32 和 Float 16 这样的位数来表示。如图 12-15 所示，如果我们降低表示一个值的位数，我们得到的结果就不那么准确了。但是，如果我们降低位数，也就降低了该模型的内存要求。QLoRA 将预训练模型权重压缩到 4 位（NF4 格式）后再加载到 GPU 显存中，相比 FP16 (16 位)，显存占用减少了 4 倍，一个参数 65B 的模型原本需要 130GB+ 显存，现在只需要约 33GB，使得单张消费级显卡（如 24GB/48GB）微调超大模型成为可能。

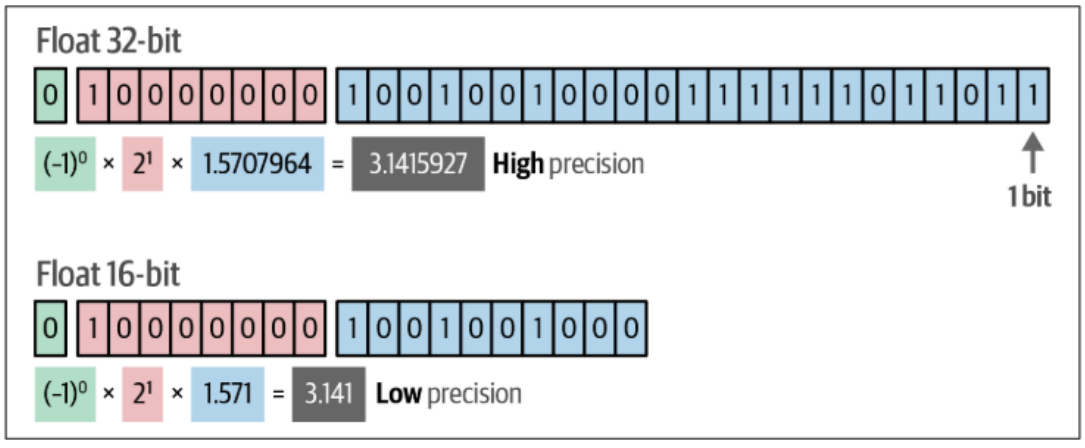

Figure 12-15. Attempting to represent pi with float 32-bit and float 16-bit representations. Notice the lowered accuracy when we halve the number of bits.

在 QLoRA 中，前向和反向传播使用的是 4 位权重，但计算梯度时会**临时将权**

重反量化为计算数据类型（通常是 FP16/BF16） 以保持精度。计算完梯度后，这些高精度的权重值会被丢弃，只更新**低秩适配器（LoRA Adapters）** 的权重。这意味着，整个训练过程中，模型的基础权重始终保持在 4 位状态。论文表明，使用 QLoRA 微调出来的模型性能，与使用 FP16 的 LoRA 微调性能相当，甚至可以达到全参数微调（16 位）的性能水平。如果你显存非常紧张，或者想要微调一个特别大的模型，QLoRA 是毫无疑问的首选。它极大地降低了高效微调的门槛。

通过量化 quantization，将模型权重从 FP32 或 FP16 降为 NF4，减少了显存的使用，但是，当将较高精度的值直接映射到较低精度的值时，多个较高精度的值最终可能会由相同的较低精度值表示。

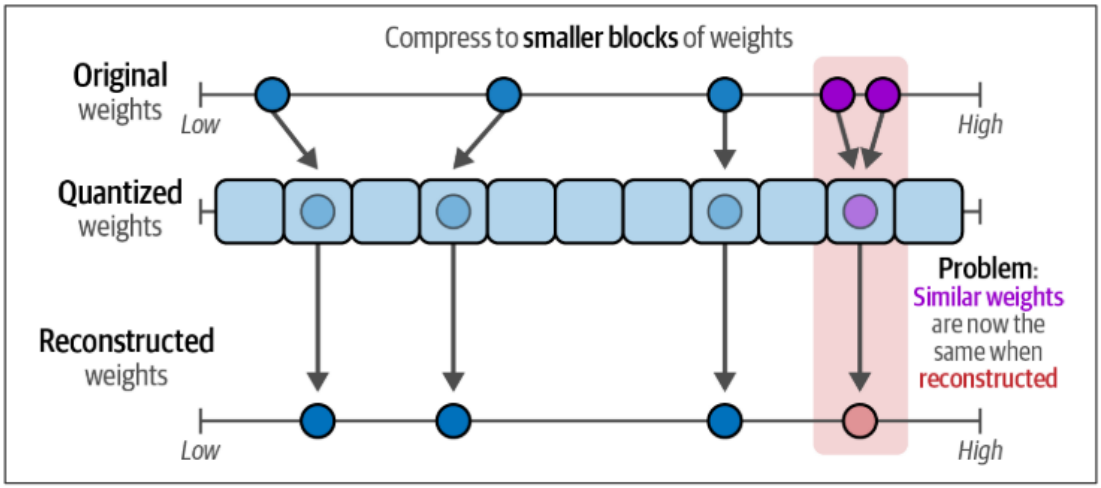

Figure 12-16. Quantizing weights that are close to one another results in the same reconstructed weights thereby removing any differentiating factor.

QLoRA 的作者找到了一种从高位降为低位同时尽可能减少误差的方法。这个转换过程的核心是一个预先定义好的量化常量（Quantization Constant）。论文证明，模型权重服从正态分布，基于这个前提，需要预先计算好一组最优的四位值，这组值有 16 个（2^4=16 四位有 16 种状态），每个值都是一个 FP16 数字。QLoRA 论文中给出的 NF4 量化常量如下：

[-1.0, -0.6961928009986877, -0.5250730514526367, -0.39491748809814453, -0.28444138169288635, -0.18477343022823334, -0.09105003625154495, 0.0, 0.07958029955625534, 0.16093020141124725, 0.24611230194568634, 0.33791524171829224, 0.44070982933044434, 0.5626170039176941,

0.7229568362236023, 1.0]。这 16 个值有自己的索引（index），用 0,1,2，…，15,16 来表示。

这 16 个值的特点就是在 0 附近更密集，在两侧更稀疏，完美匹配了权重的分布，从而最小化量化误差。在加载模型时，将原始的 FP16 权重矩阵转换为 NF4 格式存储，以节省显存。首先对权重矩阵的每一个权重进行归一化，找到每个矩阵的最大值 absmax，将矩阵中的全部权重除以这个最大值，将整个矩阵缩放到[-1,1]的范围。对于归一化后的每个值，在那 16 个预定义的 NF4 常量中，查找与之**最接近**的那个常量值。例如，一个归一化后的值是 0.2。我们在常量表中查找，发现 0.16093 更接近。不再存储原始的 FP16 值 0.2，也不存储 FP16 常量 0.16093，而是存储这个常量在表中的索引（Index）。0.16093020141124725 在表中的索引是 9（从 0 开始计数），那么我们只需要用 4 位（半个字节）来存储这个索引值 9。最终，一个原始的 FP16 权重矩阵被转换成了两个东西：一个由 4 位索引组成的 NF4 数据块（体积变为原来的 1/4）和一个缩放因子（scale）absmax，用于后续的反量化。

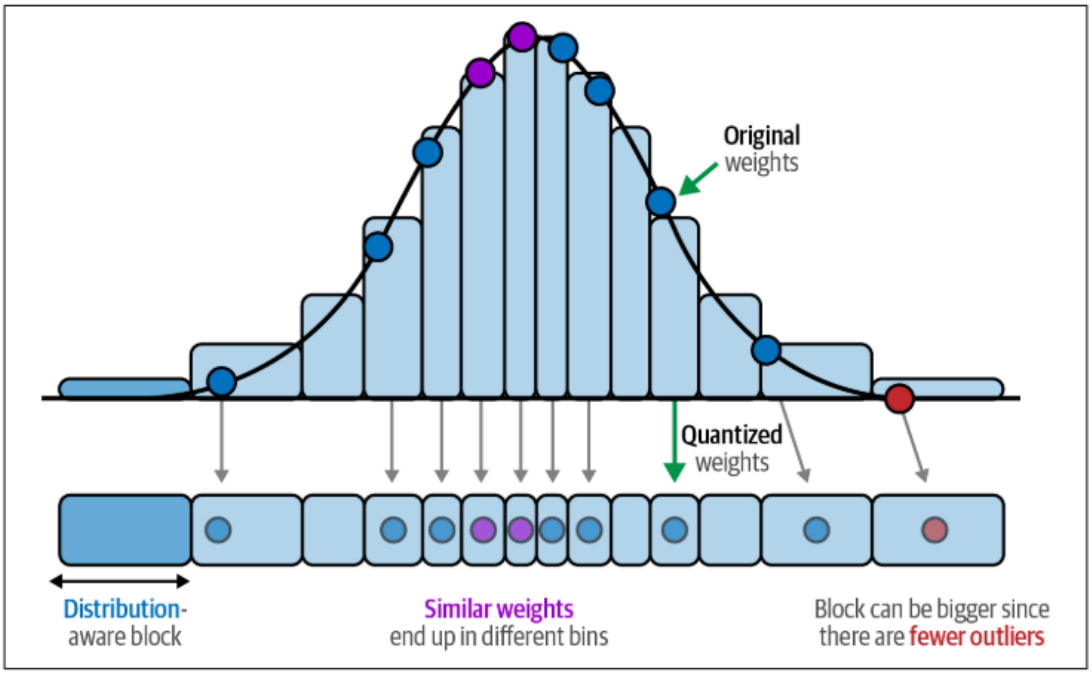

Figure 12-18. Using distribution-aware blocks we can prevent values close to one another from being represented with the same quantized value.

反量化的过程发生在计算时（前向传播或反向传播）。当需要用到这些权重进行计算时，必须将它们临时恢复成一个高精度的格式（FP16）。

反量化的步骤：根据 4 位索引，找回它对应的原始 FP16 值的**最佳近似值**，例如

9。使用这个索引 9，去预定义的 NF4 常量表中查找对应的 FP16 常量值，即 0.16093020141124725。将这个常量值，乘以之前存储的**缩放因子（scale）**absmax，将其恢复到原始的数值范围。最终，这个反量化得到的 FP16 值被送入 GPU 的计算核心（CUDA Core）进行矩阵乘法等运算。计算一完成，这些临时恢复的 FP16 权重值就会被丢弃，显存中保存的仍然是高效的 NF4 格式。

由于每个矩阵的缩放因子 absmax 不同，所以每个矩阵都可以缩放成与原来误差很小的张量。你可能仍会觉得反量化后的权重与原始的权重有较大出入，例如，原始值为 0.2，被舍入到 NF4 常量 0.1609，误差就是 0.0391。但实验证明模型的性能不会有大的影响，用 QLoRA 微调出的模型性能可以媲美全精度（FP16）微调。既然有误差，为什么模型还能正常工作？这是因为 QLoRA 的量化策略非常“聪明”，从以下**多个层面**将误差的负面影响降到了最低。

1. 量化误差不是随机噪声。它是有结构的、确定性的，并且其分布与神经网络权重的分布相匹配。NF4 的 16 个常量值是专门为**正态分布**的数据设计的。这意味着在数值密度最高的区域（靠近 0 的地方），量化等级最密集，因此大部分权重值的量化误差非常小。只有那些罕见的大权重值误差会大一些，但因为它们数量少，对整体输出的影响也相对较小。
2. 神经网络本身就不是一个精确的计算系统，它对权重中的微小扰动具有很强的容错能力。它通过大量神经元和层级的组合来工作，个别权重值的误差会在求和、激活函数等操作中被“平滑”掉。
3. 误差只存在于前向传播和反向传播的计算过程中。计算时，权重被反量化为 FP16，产生误差。计算梯度，更新 **LoRA 适配器**的参数（这些参数始终是 FP16 高精度的）。**模型的权重本身始终以冻结的、无损的 NF4 格式存储。** 误差不会累积，因为每次计算都是从“无损存储”的 NF4 状态重新反量化，而不是在上一次有误差的结果基础上进行。
4. QLoRA 并不是直接使用量化后的模型进行推理，而是用它来进行训练。在训练过程中，LoRA 适配器会学习去补偿量化带来的误差。最终的性能取决于**适配器参数+量化基础模型**这个组合的效果，而不是量化基础模型本身的效果。

还存在更精巧的优化方法，例如双重量化技术和分页优化器，您可以通过前文讨

论过的 QLoRA 论文《QLoRA: Efficient Finetuning of Quantized LLMs》深入了解。若想获得完整且高度可视化的量化技术指南，请参阅相关博客文章量化视觉指南 - 作者：Maarten Grootendorst。

补充：数据类型：

FP64：1 符号位+11 指数位+52 小数位（8 字节）

FP32：1 符号位+8 指数位+23 小数位（4 字节）

FP16：1 符号位+5 指数位+10 小数位（2 字节）

BF16(Google Brain 开发)：1 符号位+8 指数位+7 小数位（2 字节）

TF32(NVIDIA 开发，Tensor-Float 32)：1 符号位+8 指数位+10 小数位（19bit）

FP8：E4M3：1 符号位+4 指数位+3 小数位（1 字节）

E5M2：1 符号位+5 指数位+2 小数位（1 字节）

INT8：1 符号位+7 指数位（1 字节）

FP16 和 NF4 之间的转换是 QLoRA 技术的核心，这个过程分为两个方向：量化(Quantization)和反量化(Dequantization)

三．用 QLoRA 进行指令微调

在本节中，我们将微调一个完全开源且较小版本的 Llama 模型，TinyLlama，使用 QLoRA 使模型遵循用户指令。此模型为基础模型或预训练模型，并已使用语言建模进行训练。

1. 模板化指令数据

要使 LLM 遵循指令，我们需要准备遵循聊天模板的指令数据。如图 12-19 所示，该聊天模板是 LLM 生成的内容与用户生成的内容之间的差异。

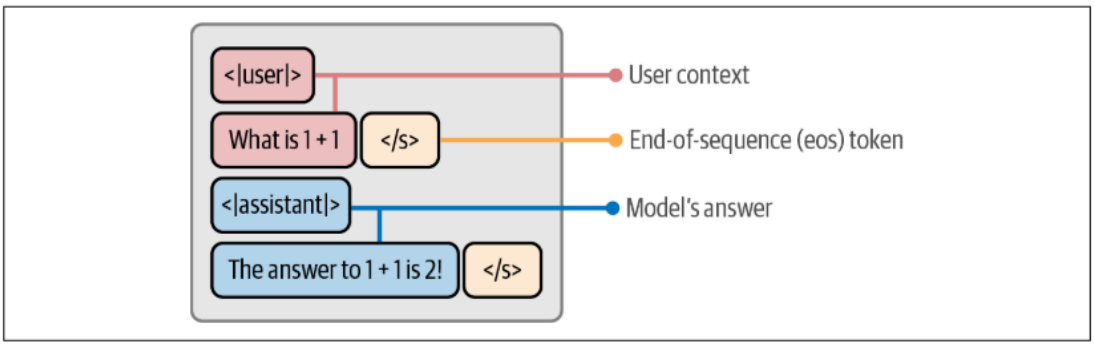

Figure 12-19. The chat template that we use throughout this chapter.

我们选择此聊天模板，是因为 Tinyllama 的聊天版本使用了相同的格式。我们正

在使用的数据是 UltraChat 数据集的一小部分。此数据集是原始 UltraChat 数据集（包含近 20 万组用户与大语言模型之间的对话）的过滤版本。

Supervised Fine-Tuning (SFT)

Data Preprocessing

```
from transformers import AutoTokenizer
from datasets import load_dataset


# Load a tokenizer to use its chat template
# 从 Hugging Face 模型库下载并加载名为 TinyLlama/TinyLlama-1.1B-Chat-v1.0 的聊天模型所对应的分词器
# 这里加载它的主要目的不是为了分词，而是为了使用它内置的聊天模板（Chat Template）
template_tokenizer = AutoTokenizer.from_pretrained("TinyLlama/TinyLlama-1.1B-Chat-v1.0")

# 创建format_prompt（）函数，确保对话遵循模版
def format_prompt(example):
        """Format the prompt to using the <|user|> template TinyLlama is using"""

        # Format answers
        chat = example["messages"]
        # 将原始的 messages 列表格式化成模型训练或推理时所期望的特定字符串格式
        prompt = template_tokenizer.apply_chat_template(chat, tokenize=False)

        return {"text": prompt}

# 下载数据集
dataset = (
        load_dataset("HuggingFaceH4/ultrachat_200k", split="test_sft")
            .shuffle(seed=42) # 随机种子，将数据集随机打乱顺序
            .select(range(3_000)) # 从打乱后的数据集中仅选择前 3000 条样本。
        # 这是一个常用的技巧，用于快速创建一个小型的、用于测试或演示的样本子集。
)
# 使用 TinyLlama 使用的模板加载并格式化数据
# .map()将dataset中的每一个样本取出传入给format_prompt()进行处理，返回"text":prompt
dataset = dataset.map(format_prompt)
```

```
/usr/local/lib/python3.12/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning:
The secret `HF_TOKEN` does not exist in your Colab secrets.
To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tok
You will be able to reuse this secret in all of your notebooks.
Please note that authentication is recommended but still optional to access public models or datasets.
```

使用“text”键，我们可以打印格式化后的提示：

```
# Example of formatted prompt
print(dataset["text"][2576])
```

```
<|user|>
Given the text: Knock, knock. Who’s there? Hike.
Can you continue the joke based on the given text material "Knock, knock. Who’s there? Hike"?</s>
<|assistant|>
Sure! Knock, knock. Who’s there? Hike. Hike who? Hike up your pants, it’s cold outside!</s>
<|user|>
Can you tell me another knock-knock joke based on the same text material "Knock, knock. Who’s there? Hike"?</s>
<|assistant|>
Of course! Knock, knock. Who’s there? Hike. Hike who? Hike your way over here and let’s go for a walk!</s>
```

2. 模型量化

加载了数据集后，我们可以开始加载模型，在这一步对模型进行量化。我们使用 bitsandbytes package 来压缩预处理的模型为 4 位表示。在 BitsandBytesConfig 中，

您可以定义量化方案。

这种量化方法使我们能够在减小原始模型大小的同时，保留原始权重的大部分精度。现在加载模型仅需约 1GB 的显存，而未经过量化处理则需要约 4GB 的显存。需要注意的是，在微调过程中需要更多的显存，训练过程需要保存优化器状态、计算梯度、更新权重等。这些操作会产生大量的中间变量，需要额外的显存开销。因此，虽然模型本身只占 1GB，但整个微调过程可能需要数倍于此的显存。**加载模型所需的显存 ≠ 训练模型所需的显存**。

3. LoRA 配置

接下来，我们需要使用 peft 库来定义我们的 LoRA 配置，该库代表微调过程的超参数：

r（秩）

这是压缩矩阵的秩（可回顾图 12-13 的内容）。增加该值会同时增大压缩矩阵的尺寸，从而降低压缩率并增强模型表征能力。该值通常取值范围在 4 到 64 之间。

lora_alpha（LoRA 缩放系数）

控制添加到原始权重上的变化量。本质上，它平衡了原始模型知识与新任务知识之间的关系。根据经验法则，通常建议取值为 r 的两倍。

target_modules（目标模块）

控制需要适配的神经网络层。LoRA 过程可以选择忽略特定层（例如某些投影层）。这种选择可以加速训练过程但可能会降低模型性能，反之亦然（即选择更多层可能提升性能但会增加训练时间）。

prepare_model_for_kbit_training()说明：

核心作用：为进行 k-bit 训练的模型做一些必要的准备工作。

当您使用 bitsandbytes 等库将模型以 4-bit 或 8-bit 形式加载后（例如通过 load_in_4bit=True），模型处于一种**推理就绪状态**，但还不是**训练就绪状态**。这个函数就是用来解决这个问题的，它主要做以下几件事：

启用梯度检查点（Gradient Checkpointing）：这是一种“用时间换空间”的内存优化技术。它会在前向传播过程中丢弃一些中间激活值，在反向传播需要时再重新计算它们。这可以显著降低训练时的显存占用，但会稍微增加训练时间。

调整输入嵌入层（Input Embedding Layer）：确保嵌入层在执行前向和反向传播时使用正确的数据类型（如 float32），以避免因低精度计算可能带来的数值不稳

定问题。

调整输出层（Output Layer）：同样地，确保模型最后的语言模型头（LM Head）与嵌入层在数据类型上对齐，以保证计算的稳定性和收敛性。

简而言之，这个函数的作用是：让一个已经被量化的模型变得**稳定且内存高效**，从而能够进行训练。如果你跳过这一步，直接训练量化模型，很可能会遇到内存溢出（OOM）错误或训练不收敛的问题

get_peft_model()说明：

核心作用：将原始模型和应用了 LoRA 配置的适配器（Adapter）封装成一个 PEFT 模型。

这个函数是 PEFT 库的核心。它接收一个基础模型（在我们的例子中，是已经经过 prepare_model_for_kbit_training 准备好的量化模型）和一个配置对象（即我们定义的 peft_config），然后执行以下操作：

冻结原始模型参数：它将基础模型的所有参数设置为 requires_grad = False，这意味着在训练过程中，这些庞大的原始参数**不会被更新**。

注入可训练适配器：根据 LoraConfig 中的设置（如 target_modules），它在指定的模块（如 q_proj, v_proj 等）旁添加**小的、可训练的 LoRA 适配器层**。只有这些新增的适配器参数是**可训练**的。

返回一个 PEFTModel 对象：这个封装后的模型在训练时，只会计算和更新那部分占比很小的适配器参数，而基础模型保持不变。在推理时，PeftModel 可以无缝地结合基础模型和适配器的计算结果。

简而言之，这个函数的作用是：创建一个**参数高效**的微调模型，它只训练新增的少量参数，从而极大节省显存和加速训练。

最后，我们可以在 Sebastian Raschka 撰写的使用 LoRA 优化 LLM 的实用技巧（低秩适应） (sebastianraschka.com)中找到有关 LoRA 微调的额外技巧。

如果要执行全参数微调，可以在加载模型时删除 quantization_config 参数并跳过 peft_config。

Models - Quantization

```python
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer, BitsAndBytesConfig

# 指定要加载的模型名称。这是一个拥有 11 亿参数的 TinyLlama 模型，是在3T token上训练了1431k步
model_name = "TinyLlama/TinyLlama-1.1B-intermediate-step-1431k-3T"

# 4-bit quantization configuration - Q in QLoRA
bnb_config = BitsAndBytesConfig(
        load_in_4bit=True,     # 以4位精度而非通常的16位或32位加载模型，大幅减少内存使用
        bnb_4bit_quant_type="nf4",     # 使用 NF4 (Normal Float 4) 量化类型
        bnb_4bit_compute_dtype="float16",     # 在计算时使用 float16 精度，平衡计算速度和数值稳定性
        bnb_4bit_use_double_quant=True,     # 使用双重量化，对量化参数本身也进行量化，进一步减少内存使用
)

# Load the model to train on the GPU
model = AutoModelForCausalLM.from_pretrained(
        model_name,
        device_map="auto",  # 自动将模型的不同部分分配到可用的 GPU 设备上

        # 应用上面定义的量化配置
        # 如果进行常规的监督微调（SFT）而不是 QLoRA，可以省略这个参数
        quantization_config=bnb_config,
)
model.config.use_cache = False  # 禁用键值缓存，这在训练时是必要的，因为缓存会干扰梯度计算
model.config.pretraining_tp = 1  # 设置张量并行度为 1（不使用张量并行），这对于微调是常见的设置

# Load LLaMA tokenizer
tokenizer = AutoTokenizer.from_pretrained(model_name, trust_remote_code=False)
tokenizer.pad_token = "<PAD>" # 设置填充标记为 <PAD>
tokenizer.padding_side = "left" # 在左侧进行填充，这对于自回归模型（如LLaMA）很重要，
                                # 因为它确保在生成文本时不会受到填充标记的影响
```

Configuration

LoRA Configuration

```python
from peft import LoraConfig, prepare_model_for_kbit_training, get_peft_model

# Prepare LoRA Configuration
peft_config = LoraConfig(
        lora_alpha=32,     # 控制添加到原始权重上的变化量。本质上，它平衡了原始模型知识与新任务知识之间的关系。
        lora_dropout=0.1,     # Dropout for LoRA Layers
        r=64,     # 压缩矩阵的秩
        bias="none",  # 不训练偏置参数
        task_type="CAUSAL_LM",  # 指定任务类型为因果语言建模
        target_modules=     # 控制需要适配的神经网络层
          ['k_proj', 'gate_proj', 'v_proj', 'up_proj', 'q_proj', 'o_proj', 'down_proj']
)

# prepare model for training
model = prepare_model_for_kbit_training(model) # 准备模型进行k-bit训练
model = get_peft_model(model, peft_config) # 将基础模型转换为PEFT模型
```

4. 训练配置

最后，我们需要像第 11 章一样配置我们的训练参数：

Training Configuration

```
[ ] from transformers import TrainingArguments

    output_dir = "./results"

    # Training arguments
    training_arguments = TrainingArguments(
            output_dir=output_dir,  #  指定输出目录，用于保存训练结果和检查点。
                                    #  这里设置为当前目录下的"results"文件夹。
            per_device_train_batch_size=2,  #  每个设备（如GPU）上的训练批次大小。
                                            #  这里设置为2，即每个GPU每次训练处理2个样本。
            gradient_accumulation_steps=4,
            optim="paged_adamw_32bit",  #  使用的优化器类型，有助于稳定训练和减少内存使用。
            learning_rate=2e-4,  #  学习率。2e-4（即0.0002）是训练中常用的学习率大小。
            lr_scheduler_type="cosine",  #  学习率调度器的类型
            num_train_epochs=1,
            logging_steps=10,    #  每隔多少步记录一次日志/训练信息
            fp16=True,  #  是否使用半精度浮点数（16位）进行训练
            gradient_checkpointing=True  #  是否使用梯度检查点技术。设置为True可以在显存中节省大量内存，
                        #  但会以稍微增加计算时间为代价。梯度检查点通过在前向传播时不保存全部中间变量，
                        #  而是在反向传播时重新计算部分中间结果来实现内存节省。
    )
```

优化这些参数是一项艰巨的任务，并且没有设定的指南。它需要不断实验才能找出最适合特定数据集、模型大小和目标任务的最佳数值。

尽管这一节的主要内容是讲解指令微调，但 QLoRA 同样可以用来做一件更具体的事——对指令模型进行微调。

5. 训练

Training!

```
from trl import SFTTrainer

# Set supervised fine-tuning parameters
trainer = SFTTrainer(
        model=model,
        train_dataset=dataset,
        dataset_text_field="text",
        tokenizer=tokenizer,
        args=training_arguments,
        max_seq_length=512,

        # Leave this out for regular SFT
        peft_config=peft_config,
)
```

```
/usr/local/lib/python3.12/dist-packages/huggingface_
```

```
# Train model
trainer.train()

# Save QLoRA weights
trainer.model.save_pretrained("TinyLlama-1.1B-qlora")
# 最后训练得到的是适配器（Adapter）权重，而不是完整的模型，后面需要合并权重
```

```
wandb: WARNING The `run_name` is currently set to the same value as `TrainingA
/usr/local/lib/python3.12/dist-packages/notebook/notebookapp.py:191: SyntaxWar
  | |_| | '_ \/ _` / _` |  _/ -_)
```

6. 合并权重

训练了 QLoRA 权重后，我们仍然需要将它们与原始权重结合起来。我们将模型重新加载 16 位，而不是量化的 4 位，以合并权重。

Merge Adapter

```
from peft import AutoPeftModelForCausalLM

# AutoPeftModelForCausalLM.from_pretrained()是PEFT库提供的专用方法，
# 用于加载使用PEFT技术（如LoRA、QLoRA）微调过的因果语言模型
model = AutoPeftModelForCausalLM.from_pretrained(
        "TinyLlama-1.1B-qlora", # 模型保存路径
        low_cpu_mem_usage=True,
        device_map="auto",
)

# Merge LoRA and base model
merged_model = model.merge_and_unload()
# merge将LoRA适配器的权重与原始基础模型的权重合并
# unload移除LoRA适配器结构，只保留合并后的完整模型
```

将适配器与基本模型合并后，我们可以结合之前的提示模板对模型发出指令：

<|user|>

Tell me something about Large Language Models.</s>

<|assistant|>

Large Language Models (LLMs) are a type of artificial intelligence (AI) that can generate human-like language. They are trained on large amounts of data, including text, audio, and video, and are capable of generating complex and nuanced language.LLMs are used in a variety of applications, including natural language processing (NLP), machine translation, and chatbots. They can be used to generate text, speech, or images, and can be trained to understand different languages and dialects. One of the most significant applications of LLMs is in the field of natural language generation (NLG). LLMs can be used to generate text in a variety of languages, including English, French, and German. They can also be used to generate speech, such as in a chatbot or voice assistant. LLMs have the potential to revolutionize the way we communicate and interact with each other. They can help us create more engaging and personalized content, and they can also help us to communicate more effectively with machines.

四．评估生成模型

评估生成式模型是一项重大挑战。生成式模型应用于众多不同的使用场景，这使得依赖单一评估指标进行判断变得困难。与专业化模型不同，生成式模型解决数学问题的能力并不能保证其在编程问题上的成功。与此同时，对这些模型的评估

至关重要，特别是在需要保持一致性的生产环境中。鉴于其概率性本质，生成式模型不一定会产生一致的输出，因此需要建立稳健的评估体系。

在本节中，我们将探讨几种常见的评估方法，但需要强调的是，目前尚缺乏黄金标准。没有任何一个指标能完美适用于所有使用场景。

1. Word-Level Metrics 词级指标

在生成式模型的比较中，常见的评估指标类别是基于词级别的评估。这类经典技术通过在词元（集）层面上对比参考数据集与生成文本来进行评估。常用的词级别指标包括困惑度（perplexity）、ROUGE、BLEU 以及 BERTScore。

需要特别说明的是困惑度指标，它用于衡量语言模型预测文本的能力。给定输入文本后，模型会预测下一个词元的出现概率。通过困惑度指标，我们认为如果模型能为下一个词元赋予高概率，则其性能更优。换句话说，当面对一篇书写规范的文档时，模型不应表现出"困惑"。

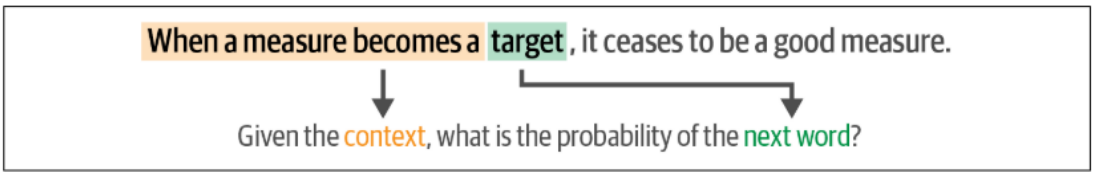

Figure 12-20. Next-word prediction is a central feature of many LLMs.

尽管这些指标是有用的，但它们并不是完美的衡量标准，它们不考虑生成文本的一致性、流利性、创造性和正确性。

2. Benchmarks 基准测试

评估生成式模型在语言生成与理解任务性能的常用方法，是采用知名公共基准测试，如 MMLU、GLUE、TruthfulQA、GSM8k 和 HellaSwag 等。这些基准测试不仅能衡量基础语言理解能力，还能评估复杂分析性问题解答能力（如数学题求解）。除自然语言任务外，部分模型专注于编程等其他专业领域。这类模型通常采用不同的评估基准，例如由高难度编程题目构成的 HumanEval 测试集，要求模型完成代码编写任务。表 12-1 汇总了生成式模型常用公共基准测试的概览信息。

Table 12-1. Common public benchmarks for generative models

Benchmark	Description	Resources
MMLU	The Massive Multitask Language Understanding (MMLU) benchmark tests the model on 57 different tasks, including classification, question answering, and sentiment analysis.	*https://oreil.ly/nrG_g*
GLUE	The General Language Understanding Evaluation (GLUE) benchmark consists of language understanding tasks covering a wide degree of difficulty.	*https://oreil.ly/LV_fb*
TruthfulQA	TruthfulQA measures the truthfulness of a model's generated text.	*https://oreil.ly/i2Brj*
GSM8k	The GSM8k dataset contains grade-school math word problems. It is linguistically diverse and created by human problem writers.	*https://oreil.ly/oOBXY*
HellaSwag	HellaSwag is a challenge dataset for evaluating common-sense inference. It consists of multiple-choice questions that the model needs to answer. It can select one of four answer choices for each question.	*https://oreil.ly/aDvBP*
HumanEval	The HumanEval benchmark is used for evaluating generated code based on 164 programming problems.	*https://oreil.ly/dIJIX*

基准测试是评估模型在多样化任务上表现的基础手段。然而，公共基准测试存在若干局限性：首先，模型可能针对这些基准测试过拟合，从而生成最优应答却影响实际泛化能力；其次，现有基准测试范围仍显宽泛，难以覆盖特定垂直场景的需求；最后，部分基准测试需要高性能 GPU 并耗费长时间（超过数小时）进行计算，这给迭代优化带来了显著困难。

3. Leaderboards 排行榜

基准测试有很多中，但是为你的模型选择最适合的测试很难。每当有新模型发布时，你通常会看到它在一系列基准测试上的评估结果，以此来全面展示其综合性能。因此，业界开发了包含多种基准测试的排行榜。以目前常见的开放 LLM 排行榜（Open LLM Leaderboard）为例，截至本文撰写时，该榜单共包含六项基准测试，涵盖 HellaSwag、MMLU、TruthfulQA 及 GSM8k 等评估项目。荣登排行榜首的模型，只要未对测试数据过拟合，通常被视为"最佳"模型。然而，由于这些排行榜通常采用公开可用的基准测试，仍然存在模型对排行榜指标过拟合的风险。

4. Automated Evaluation

评估生成式输出的重要组成部分在于文本质量。例如，即使两个模型对同一问题给出了相同的正确答案，它们推导答案的过程也可能截然不同。这不仅关乎最终答案的正确性，更涉及答案的构建过程。同样地，虽然两份摘要内容相似，但其中一份可能比另一份更为简洁——这往往是优秀摘要的关键特质。

为了在最终答案正确性之外评估生成文本的质量，业界引入了"LLM 评判员"（LLM-as-a-judge）的方法。其核心思想是使用另一个独立的 LLM 来评估待测 LLM 的输出质量。该方法有个有趣的变体——双模型对比评估：让两个不同的 LLM 分别生成问题答案，再由第三个 LLM 担任裁判来判定孰优孰劣。

这种方法实现了对开放式问题的自动化评估。其最大优势在于：随着 LLM 技术的持续进步，其评判输出质量的能力也会同步提升。换言之，这种评估方法能够伴随整个领域共同演进。

5. Human Evaluation

尽管基准测试很重要，但评估的黄金标准仍被公认为人类评估。即使某个 LLM 在通用基准测试中表现出色，但其在特定领域任务中的表现仍可能不尽如人意。更重要的是，基准测试无法完全捕捉人类偏好，之前讨论的所有方法都只是人类评估的替代方案。

基于人类评估的典型范例是 Chatbot Arena。当您访问这个排行榜时，系统会展示两个匿名 LLM 供您交互。您提出的任何问题或指令都会同时发送给两个模型，并接收它们的输出结果，然后由您决定哪个输出更符合偏好。这种机制让社区用户能在不知道模型身份的情况下进行投票选择，只有在完成投票后才会揭晓各输出对应的模型。

该方法已累计收集超过 80 万次人类投票，并据此生成排行榜。这些投票数据通过计算胜率来评估 LLMs 的相对能力水平——例如，当低排名模型击败高排名模型时，其排名将发生显著变化。

这种方法通过众包投票帮助我们理解 LLM 的质量，但它终究是广泛用户群体的综合意见，未必符合您的具体使用场景。因此，并不存在完美的 LLM 评估方法。所有提及的方法论和基准测试都提供了重要但有限的评估视角。我们的建议是根据目标应用场景进行评估。例如对于编程任务，选用 HumanEval 就比 GSM8k 更具合理性。

但最重要的是，最终应由您来决定 LLM 是否满足您的使用需求。你可以亲自尝试这些模型，甚至可以设计自己的测试问题。

五．偏好调优/对齐/基于人类反馈的强化学习

尽管我们的模型现在能够遵循指令，但我们可以通过最终的训练阶段进一步改进其行为，使其在不同场景下的表现符合预期。例如，当被问及“什么是 LLM？”时，相较于仅回答“它是一种大语言模型”而不作进一步解释，我们可能更倾向于获得一个详细描述 LLM 内部机制的答案。那么，如何将人类对某一答案的偏好与 LLM 的输出精确对齐呢？

我们可以让一名人员（偏好评估员）对该模型生成结果的质量进行评估。假设他们为其分配了特定分数，例如 4 分（见图 12-22）。

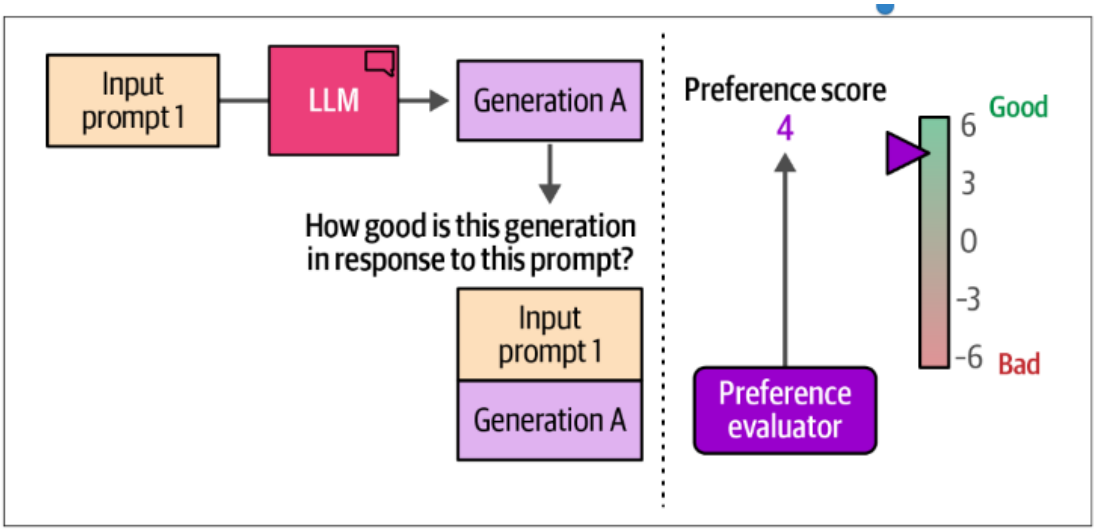

Figure 12-22. Use a preference evaluator (human or otherwise) to evaluate the quality of the generation.

然后，会基于这个特定分数更新模型。图 12-23 展示了基于该分数更新模型的偏好调优步骤：

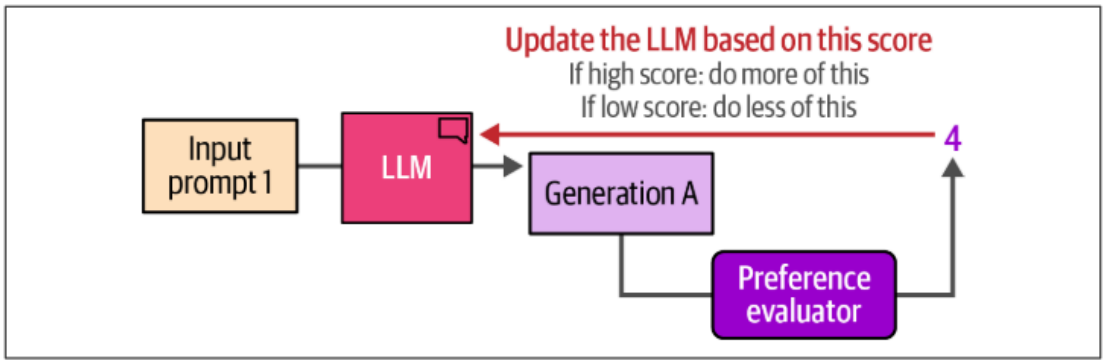

Figure 12-23. Preference tuning methods update the LLM based on the evaluation score.

若得分较高，则更新模型以鼓励其生成更多此类输出；若得分较低，则更新模型以抑制此类生成。与往常一样，我们需要大量训练样本。那么能否实现偏好评估的自动化？通过训练一个称为 reward model 奖励模型的独立模型即可实现。

六．基于奖励模型的偏好评估自动化

为实现偏好评估的自动化，我们需要在偏好调优步骤之前增加一个训练奖励模型的环节，如图 12-24 所示。在偏好微调开始之前训练一个 reward model。

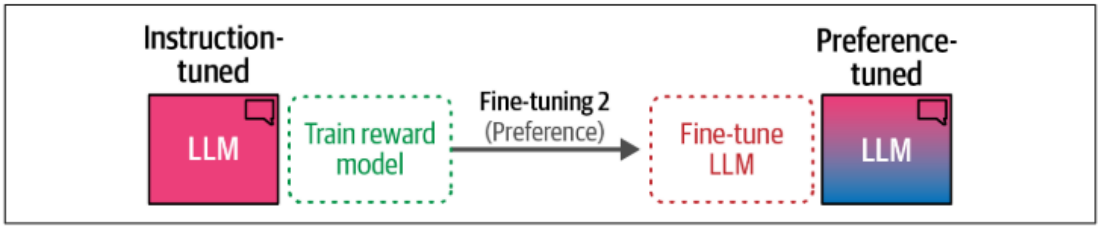

Figure 12-24. We train a reward model before fine-tuning the LLM.

图 12-25 表明，为了创建奖励模型，我们复制指令调优模型的副本并稍作修改，使其不再生成文本，而是输出单个评分。通过将其语言建模头替换为质量分类头转变为奖励模型。

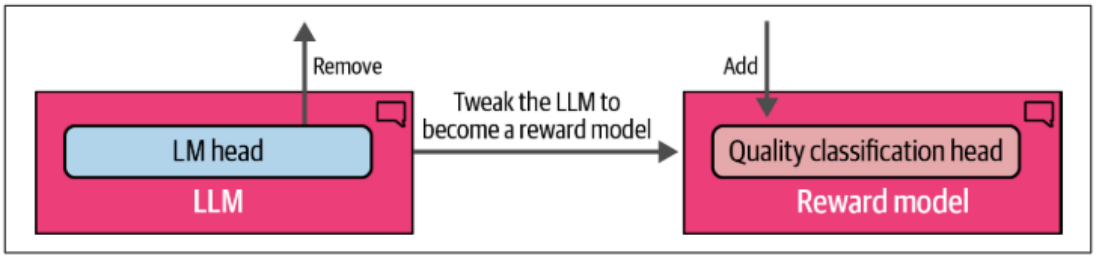

Figure 12-25. The LLM becomes a reward model by replacing its language modeling head with a quality classification head.

1. 奖励模型的输入和输出

我们预期该奖励模型的工作方式是：输入一个提示（prompt）和模型的生成内容 generation，奖励模型将输出一个数值，用于表示该生成内容相对于提示的偏好或质量。图 12-26 展示了该奖励模型生成这一数值的过程。

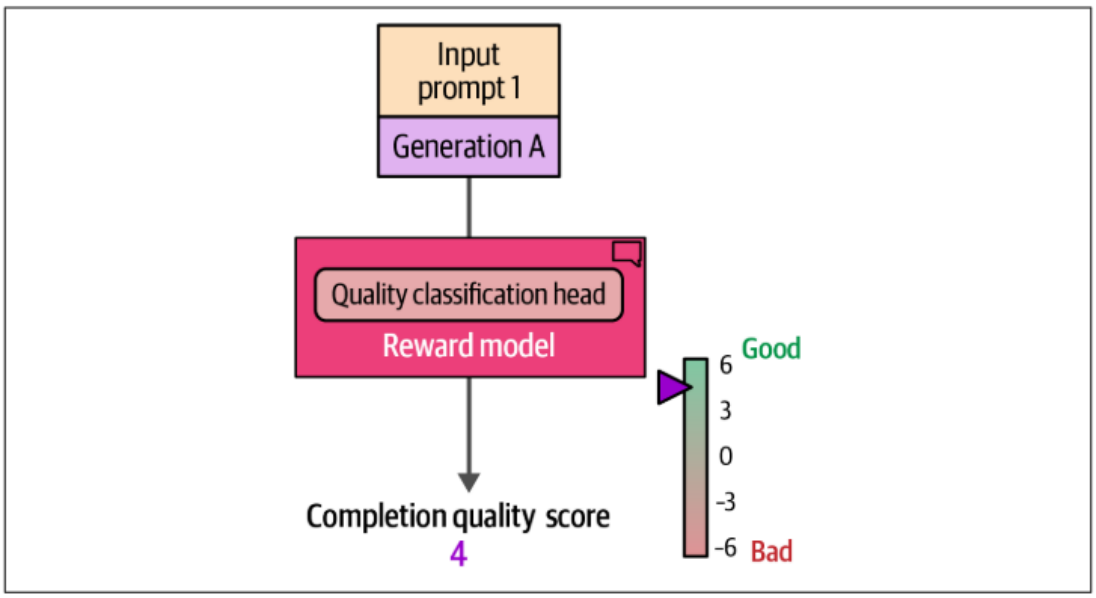

Figure 12-26. Use a reward model trained on human preference to generate the completion quality score.

2. 训练奖励模型

我们无法直接使用奖励模型。该模型需要先经过训练才能正确评估生成内容。因此，我们需要获取一个偏好数据集供模型学习。

① 奖励模型的训练数据集

偏好数据集的一种常见形式是：每个训练样本包含一个 prompt、一个被采纳的生成结果和一个被拒绝的生成结果。（PS：这不代表"好"与"坏"；有时两个生成结果都较好，但其中一个更优）。图 12-27 展示了一个包含两个训练样本的偏好训练集示例。

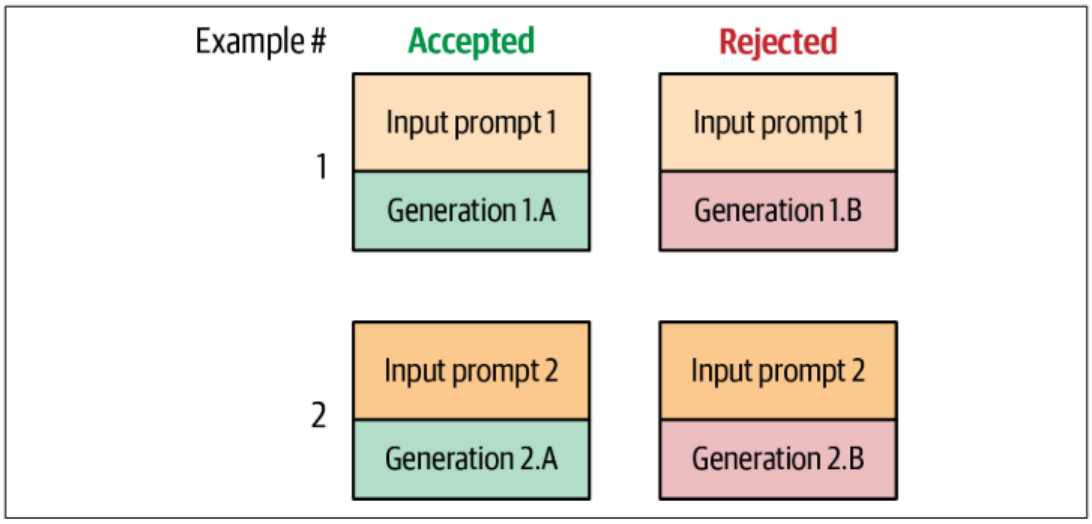

Figure 12-27. Preference tuning datasets are often made up of prompts with accepted and rejected generations.

生成偏好数据的一种方法是向 LLM 提供 prompt，要求其生成两种不同的输出。然后让人类选择他们更倾向的版本，即打标签。

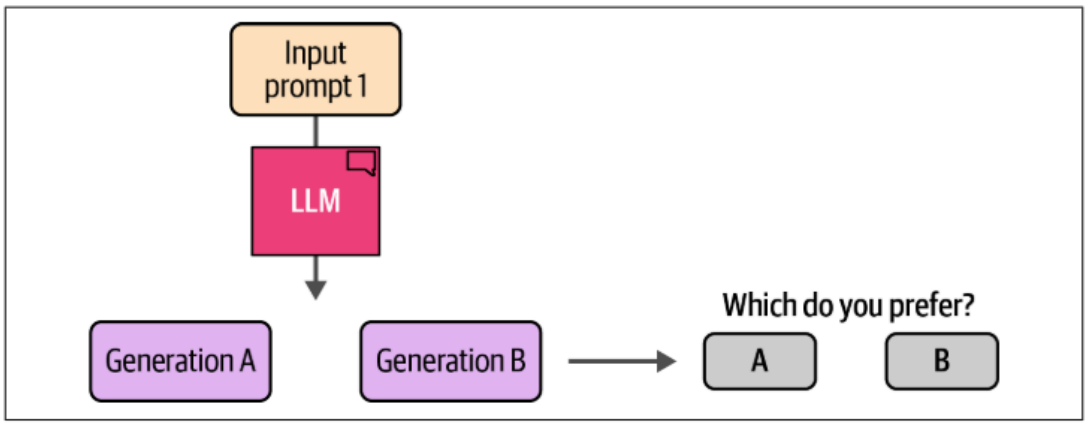

Figure 12-28. Output two generations and ask a human labeler which one they prefer.

② 奖励模型训练步骤

现在我们有了数据集，可以继续训练奖励模型。一个简单的训练步骤是使用奖励模型进行以下操作：1. 对 accepted 的生成内容进行打分；2. 对 rejected 的生成内容进行打分。确保 accepted 比 rejected 具有更高的评分。

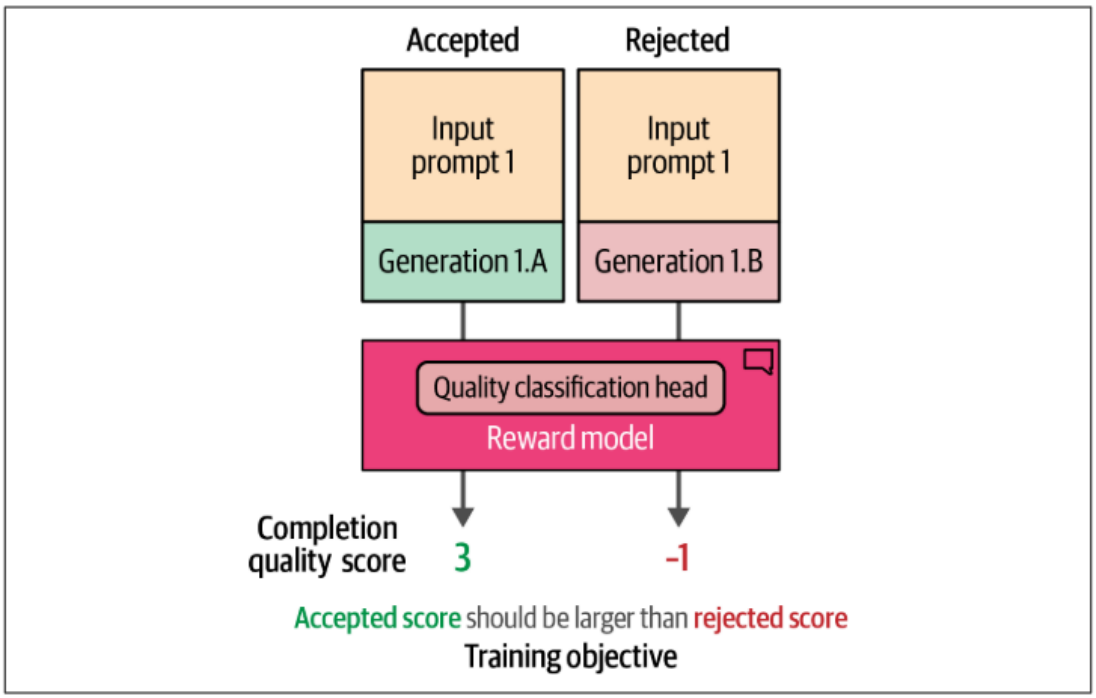

Figure 12-29. The reward model aims to evaluate the quality scores of generations in response to a prompt.

将各环节整合后可得偏好微调的流程：1. 收集偏好数据 2. 训练奖励模型 3. 利用奖励模型对 LLM 进行微调（此时奖励模型作为偏好评估器运作）

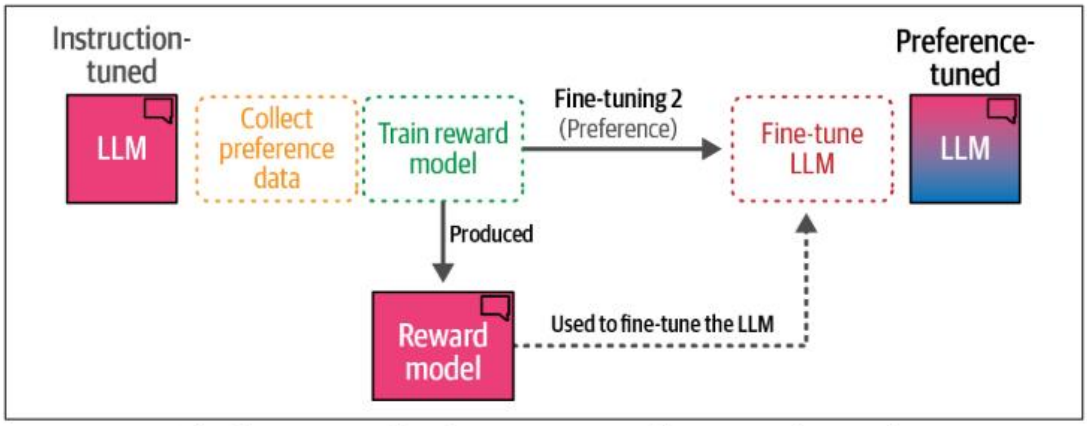

Figure 12-30. The three stages of preference tuning: collecting preference data, training a reward model, and finally fine-tuning the LLM.

奖励模型是一种极具潜力的创新方法，其应用范围可进一步扩展与深化。以 Llama2 为例，该系统同时训练了两个奖励模型：一个用于评估回答的“有用性”，另一个则用于评估回答的“安全性”（图 12-31）。

有用性奖励模型（Helpfulness RM）： 判断回答是否有帮助、是否准确、是否切题、信息量是否充足。例如，对一个技术问题给出详细正确的解答会得高分。

安全性奖励模型（Safety RM）： 判断回答是否有害、是否有偏见、是否包含不安全或不道德的内容。例如，生成暴力、仇恨言论或违法内容会得低分（或负分）。

在训练时，Llama 2 会同时考虑这两个分数，目标是生成**既有用又安全**的回答。这避免了 AI 为了一味地讨好用户（获得“有用”高分）而可能输出有害但看似“有用”的内容（比如详细描述如何实施犯罪行为）。

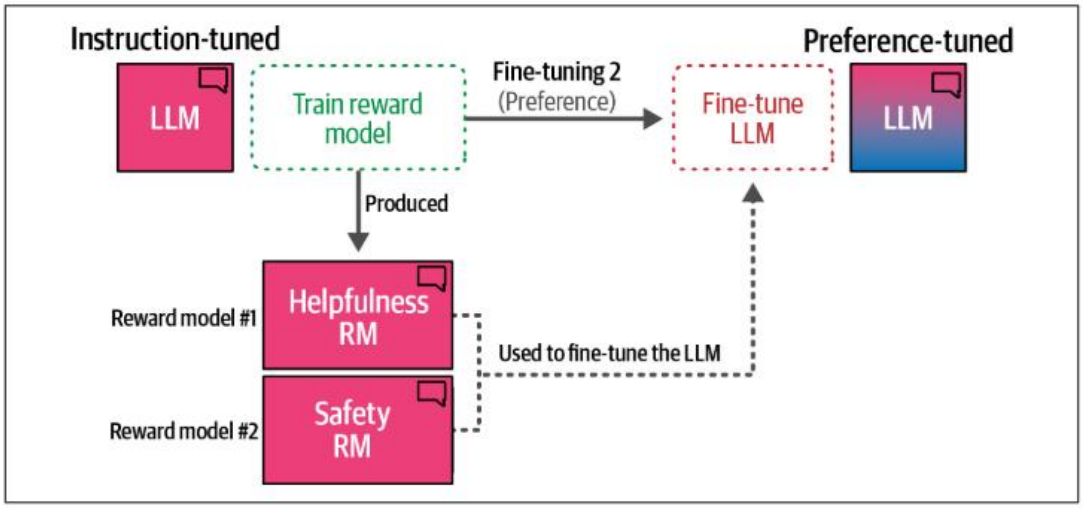

Figure 12-31. We can use multiple reward models to perform the scoring.

利用训练好的奖励模型对 LLM 进行微调的常用方法是近端策略优化（Proximal

Policy Optimization, **PPO**)。PPO 是一种流行的强化学习技术，通过确保 LLM 的输出与预期奖励不会偏离过大，从而优化经过指令微调的 LLM。

PPO 是什么：模型为了骗取奖励模型的高分可能会生成一堆无意义但能的高分的乱码，或者完全丧失语言能力，忘记以前训练学会的语言知识等，这被称为性能崩溃（collapse），PPO 为了防止这种现象的出现，会约束 constrain 每次更新的幅度，确保新模型和旧模型的参数不会相差太多。

3. 训练无奖励模型

PPO 的劣势在于其方法复杂度较高，需要同时训练奖励模型和大语言模型（LLM）至少两个模型，可能导致不必要的计算成本。直接偏好优化（Direct Preference Optimization, **DPO**）作为 PPO 的替代方案，摒弃了基于强化学习的过程。该方法不再依赖奖励模型评估生成质量，而是让 LLM 自身判断好坏。如图 12-32 所示，我们使用指令微调后的 LLM 作为参考模型，它是当前 LLM 的一个冻结的、不可训练的副本，代表了“原始认知”；可训练模型是我们要微调的 LLM，是参考模型的一个可训练副本。通过对比可训练模型与参考模型在生成"accepted generation"和"rejected generation"的概率偏移进行评估。

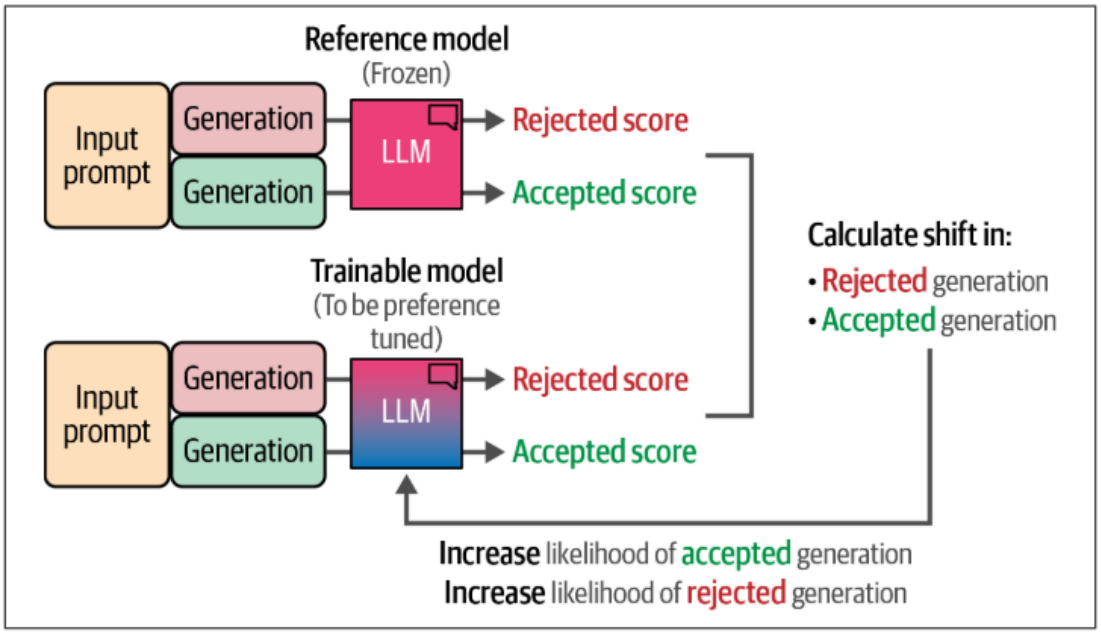

Figure 12-32. Use the LLM itself as the reward model by comparing the output of a frozen model with the trainable model.

概率的偏移体现了认知偏移，工作原理：数据集与奖励模型一样，每条数据包含一个 prompt，一个 accepted generation（人类偏好的回答），一个 rejected

generation（人类不喜欢的回答），对于一个指令，DPO 会随机选择一个 generation（无论是被接受的还是被拒绝的）同时询问两个模型：“你有多大可能性会生成这个回答？”每个模型通过计算生成每个 Token 的**对数概率（Log Probabilities）**来得到这个可能性。如果可训练模型生成 accepted generation 的可能性远高于参考模型，同时生成 rejected generation 的可能性远低于参考模型，那么说明它的优化方向是正确的。

图中 Calculate token probability (Per model)这一步，两个模型都在各自计算生成 ″got″、″no″、″clue″、″!″ 的概率。然后将分数聚合（aggregation），即回答的所有“token”的概率组合起来，得到整个句子的“分数”。偏移 = Score_trainable - Score_reference，DPO 的损失函数就是基于这个偏移来设计的。对于 rejected generation，我们希望偏移是负值，并且要最大化这个负值；对于 accepted generation，我们希望偏移是正值，并且最大化这个正值。

通过这种直接的概率比较和优化，可训练模型的参数被调整为：更自信地生成好的回答，更不愿意生成坏的回答。整个过程完全不需要训练一个单独的奖励模型，也不需要复杂的 PPO 算法，只需要标准的梯度下降即可完成，因此更加稳定和高效。

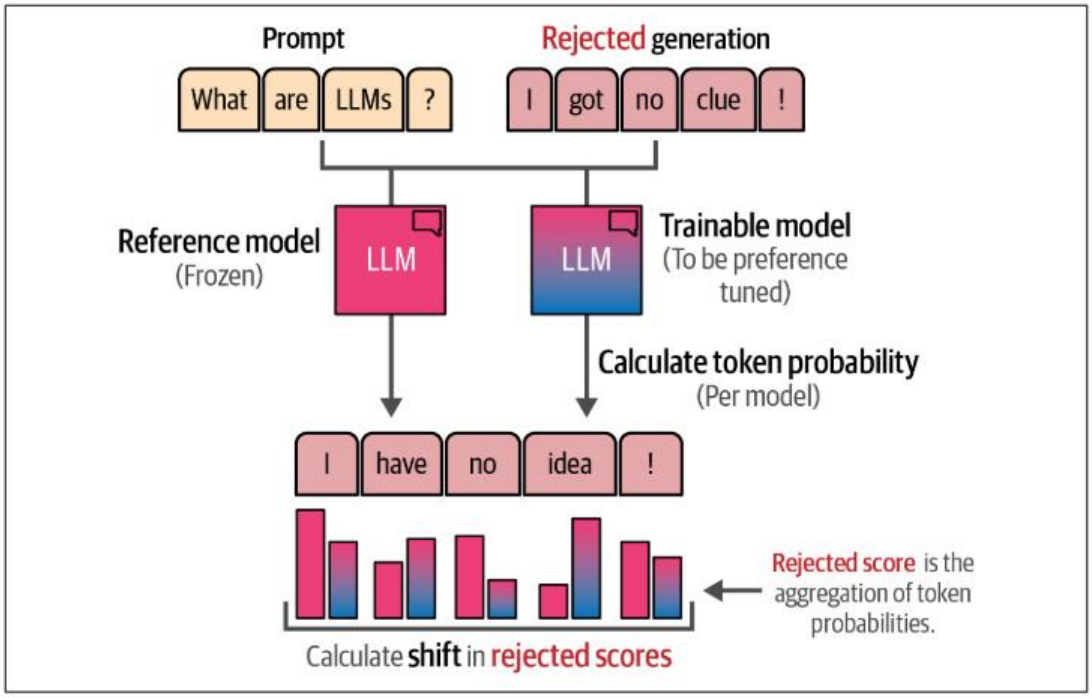

Figure 12-33. Scores are calculated by taking the probabilities of generation on a token level. The shift in probabilities between the reference model and the trainable model is optimized. The accepted generation follows the same procedure.

Figure 12-33 的流程：**Prompt + Bad Response → 输入给两个模型 → 分别计算每个词的概率 → 分别聚合出总分 → 比较两个总分的差异（Shift）→ 利用这个差异值来指导模型优化**

与 PPO（近端策略优化）相比，作者发现 **DPO**（直接偏好优化）在训练过程中更稳定且更精确。鉴于其稳定性，我们将采用 DPO 作为主要模型，用于对先前经过指令微调的模型进行偏好调优。

七．基于 DPO 的偏好调优

通过过滤步骤将数据规模从最初的 13,000 个样本进一步缩减至约 6,000 个样本。

Preference Tuning (PPO/DPO)

Data Preprocessing

```
from datasets import load_dataset

def format_prompt(example):
    """Format the prompt to using the <|user|> template TinyLLama is using"""

    # Format answers
    system = "<|system|>\n" + example['system'] + "</s>\n" # /n是换行
    prompt = "<|user|>\n" + example['input'] + "</s>\n<|assistant|>\n"
    chosen = example['chosen'] + "</s>\n" # chosen就是accepted generation
    rejected = example['rejected'] + "</s>\n"

    return {
        "prompt": system + prompt,
        "chosen": chosen,
        "rejected": rejected,
    }

# Apply formatting to the dataset and select relatively short answers
dpo_dataset = load_dataset("argilla/distilabel-intel-orca-dpo-pairs", split="train")
dpo_dataset = dpo_dataset.filter(
    lambda r:
        r["status"] != "tie" and # 排除平局样本。DPO需要明确的偏好，不能是两者差不多的。
        r["chosen_score"] >= 8 and # 只选择质量很高的胜出回答（评分≥8分，通常是10分制）。
                                   # 确保学习的偏好是明确的优质回答。
        not r["in_gsm8k_train"] # 排除属于GSM8K训练集的数据。这是为了防止数据泄露，
                                # 确保模型在数学推理基准测试上的评估结果是公平的。
)
# 移除所有原始列，只保留format_prompt函数返回的新列。
dpo_dataset = dpo_dataset.map(format_prompt, remove_columns=dpo_dataset.column_names)
dpo_dataset
```

数据集格式：一个 prompt+一个高分回答和一个低分回答

```
Dataset({
    features: ['chosen', 'rejected', 'prompt'],
    num_rows: 5922
})
```

Models - Quantization

```python
from peft import AutoPeftModelForCausalLM
from transformers import BitsAndBytesConfig, AutoTokenizer

# 4-bit quantization configuration - Q in QLoRA
bnb_config = BitsAndBytesConfig(
        load_in_4bit=True,    # Use 4-bit precision model loading
        bnb_4bit_quant_type="nf4",    # Quantization type
        bnb_4bit_compute_dtype="float16",    # Compute dtype
        bnb_4bit_use_double_quant=True,    # Apply nested quantization
)

# Merge LoRA and base model
model = AutoPeftModelForCausalLM.from_pretrained(
        "TinyLlama-1.1B-qlora",  # 上一节中训练得到的适配器在这个文件中
        low_cpu_mem_usage=True,
        device_map="auto",
        quantization_config=bnb_config,
)
merged_model = model.merge_and_unload()

# Load LLaMA tokenizer
# 加载原始模型的tokenizer
model_name = "TinyLlama/TinyLlama-1.1B-intermediate-step-1431k-3T"
tokenizer = AutoTokenizer.from_pretrained(model_name, trust_remote_code=False)
tokenizer.pad_token = "<PAD>" # 设置填充token，确保批处理时长度一致
tokenizer.padding_side = "left" # 左侧填充（适用于因果语言模型）
```

设置 LoRA 配置

Configuration

```python
from peft import LoraConfig, prepare_model_for_kbit_training, get_peft_model

# Prepare LoRA Configuration
# 见上面SFT中Configuration的代码解释
peft_config = LoraConfig(
        lora_alpha=32,    # LoRA Scaling
        lora_dropout=0.1,    # Dropout for LoRA Layers
        r=64,    # Rank
        bias="none",
        task_type="CAUSAL_LM",
        target_modules=    # Layers to target
          ['k_proj', 'gate_proj', 'v_proj', 'up_proj', 'q_proj', 'o_proj', 'down_proj']
) # q_proj, k_proj, v_proj：注意力机制的查询、键、值投影
  # o_proj：注意力输出投影 gate_proj, up_proj, down_proj：FFN层的前向投影

# prepare model for training
model = prepare_model_for_kbit_training(model)
model = get_peft_model(model, peft_config) # 将基础模型转换为PEFT模型，冻结原始模型的所有参数，仅添加和训练LoRA适配器层
```

DPO 训练的超参数：

```
[ ] from trl import DPOConfig

    output_dir = "./results"

    # Training arguments
    training_arguments = DPOConfig( # 上面STF也有training_arguments，
                                    # 区别在于超参数调用的函数名是TrainingArguments
                                  # trainer的函数名是SFTTrainer
            output_dir=output_dir,
            per_device_train_batch_size=2,
            gradient_accumulation_steps=4,
            optim="paged_adamw_32bit",
            learning_rate=1e-5,
            lr_scheduler_type="cosine",
            max_steps=200,
            logging_steps=10,
            fp16=True,
            gradient_checkpointing=True,
            warmup_ratio=0.1
    )
```

创建 DPO 的 Trainer，开始训练。

```
[ ] from trl import DPOTrainer

    # Create DPO trainer
    dpo_trainer = DPOTrainer( # DPO的trainer函数名是DPOTrainer
            model,
            args=training_arguments,
            train_dataset=dpo_dataset,
            tokenizer=tokenizer,
            peft_config=peft_config,
            beta=0.1,
            max_prompt_length=512,
            max_length=512,
    )

    # Fine-tune model with DPO
    dpo_trainer.train()

    # Save adapter
    dpo_trainer.model.save_pretrained("TinyLlama-1.1B-dpo-qlora")
```

参数	DPOConfig	SFT的TrainingArguments	说明
学习率	通常更小 (1e-5)	可能更大 (1e-4)	DPO需要精细调优
批次大小	通常很小	可以较大	DPO内存消耗大
优化器	常用paged_adamw	标准adamw	防止内存溢出
训练步数	较少 (100-500)	可能较多	DPO收敛快

DPO 训练完成后，先合并基础模型和 SFT 的 LoRA 适配器，再在其基础上合并 DPO 的 LoRA 适配器。这是一种层级式的模型集成方法。

```python
from peft import PeftModel

# Merge LoRA and base model
model = AutoPeftModelForCausalLM.from_pretrained(
        "TinyLlama-1.1B-qlora", # SFT阶段训练的LoRA适配器
        low_cpu_mem_usage=True,
        device_map="auto",
)
sft_model = model.merge_and_unload() # 合并得到SFT基础模型

# Merge DPO LoRA and SFT model
dpo_model = PeftModel.from_pretrained(
        sft_model, # 使用SFT模型作为新基础
        "TinyLlama-1.1B-dpo-qlora", # DPO阶段训练的LoRA适配器
        device_map="auto",
)
dpo_model = dpo_model.merge_and_unload() # 最终合并
```

```python
from transformers import pipeline

# Use our predefined prompt template
prompt = """<|user|>
Tell me something about Large Language Models.</s>
<|assistant|>
"""

# Run our instruction-tuned model
pipe = pipeline(task="text-generation", model=dpo_model, tokenizer=tokenizer)
print(pipe(prompt)[0]["generated_text"])
```

```
<|user|>
Tell me something about Large Language Models.</s>
<|assistant|>
Large Language Models (LLMs) are a type of artificial intelligence (AI) that can ge

LLMs are used in a variety of applications, including natural language processing (
```

<|user|>

Tell me something about Large Language Models.</s>

<|assistant|>

Large Language Models (LLMs) are a type of artificial intelligence (AI) that can generate human-like language. They are trained on large amounts of data, including text, audio, and video, and are capable of generating complex and nuanced language.LLMs are used in a variety of applications, including natural language processing (NLP), machine translation, and chatbots. They can be used to generate text, speech, or images, and can be trained to understand different languages and dialects.One of the most significant applications of LLMs is in the field of natural language generation (NLG). LLMs can be used to generate text in a variety of languages, including English, French, and German. They can also be used to generate speech, such as in a chatbot or voice assistant.LLMs have the potential to revolutionize the way we communicate and interact with each other. They can help us create more engaging and personalized content, and they can also help us to communicate more effectively with machines.