微调

微调（Fine-tuning）是大模型从“通用能力”迁移到“特定任务”的核心技术，其本质是基于迁移学习思想，预训练是利用海量的数据，巨大的算力来获得通用的知识（参数），而微调是在预训练得到的模型基础上使用少数高质量，特定方向领域的数据略微调整参数（不调整参数本身，而是调整变化量，最后加载一起），从而获得专精模型。

预训练 (Pre-training)

预训练是从零开始训练一个全新模型的过程，这是一个极其耗费资源和时间的过程，通常只有大型机构或公司才能完成。

数据收集与准备：收集海量的、无标注的文本数据（如网页、书籍、代码等），通常达到 TB 级别。然后进行数据清洗、去重、格式化等。

选择模型架构：确定模型的骨架，例如基于 Transformer 的 GPT（仅解码器，自回归）或 BERT（仅编码器，自编码）等架构。

定义训练目标（损失函数）：

自回归语言建模（如 GPT）：目标是预测下一个 token（P(x_t | x_<t)）。给定前文，让模型预测下一个词是什么。

自编码语言建模（如 BERT）：目标是掩码语言建模（MLM）。随机掩盖输入中的一些 token，让模型预测被掩盖的词。

大规模分布式训练：将准备好的海量数据输入模型，在成千上万的 GPU/TPU 上进行数周甚至数月的训练。这个过程会消耗巨大的算力和电力。

得到基础模型（Base Model）：训练完成后，得到一个具备了通用语言知识（语法、事实、推理能力等）的“基座模型”，如 LLaMA、GPT-4、ChatGLM 等。

微调的功能作用

预训练模型是通用模型，但对特定任务效果不一定最好，微调的核心逻辑是“知识迁移 + 任务适配”，使得微调后的模型能够专精于特定方向领域。

知识复用：预训练模型已通过海量数据学习了通用语言规律、世界常识等基础能力（如语义理解、逻辑推理），这些知识对下游任务具有普遍价值（例如“情感分析”和“文本分类”都依赖语义理解能力）。

定向优化：针对特定任务（如“医疗问答”“法律文书分类”），用少量标注数据“修正”预训练模型的参数，让模型聚焦于任务专属规律（如医疗术语的特殊含义、法律文本的格式规范）。

参数更新：通过反向传播算法，基于任务数据的监督信号（如“文本 - 标签”对）调整模型参数（部分或全部），使模型在任务上的预测误差最小化，最终形成适配任务的“专用模型”。

微调的基本组件

微调的核心组件围绕“任务适配”设计，缺一不可：

任务特定数据集

作用：提供任务专属的监督信号，是模型学习任务规则的“教材”。

特点：需人工标注（如“文本 - 标签”“问题 - 答案”），规模远小于预训练数据（通常数千至数万样本），但需与任务高度相关（如“金融情感分析”需聚焦股市评论数据）。

预训练模型 Checkpoint

作用：作为微调的“初始参数”，提供通用知识基础，避免从零训练。

形式：预训练结束后保存的模型权重文件（如 PyTorch 的.pth、TensorFlow 的.ckpt），包含模型所有层的参数（如 Transformer 的注意力层、全连接层权重）。

微调策略

作用：控制参数更新的范围和方式，平衡效果与效率。

核心：决定“哪些参数更新”（全量参数 / 部分参数）和“如何更新”（学习率、优化器选择等），例如“冻结底层参数，仅更新顶层”可减少计算量。

任务专属评估指标

作用：衡量微调效果，指导模型优化。

示例：分类任务用“准确率（Accuracy）”“F1 值”；问答任务用“精确匹配率（Exact Match）”；翻译任务用“BLEU 值”。

轻量训练环境

作用：支撑微调计算，无需预训练级别的大规模集群。

配置：单卡 GPU（如 RTX 3090、A100）或少量显卡，搭配深度学习框架（PyTorch、TensorFlow）和分布式工具（如 Accelerate）。

常用微调方法及原理

根据参数更新范围和效率，微调方法可分为两类：

1. 全量微调（Full Fine-tuning）

原理：加载预训练模型后，用任务数据更新模型的**所有参数**（从底层到顶层），让模型在保留通用知识的同时，全方位适配任务。

优点：理论上性能最优（参数调整最充分），适合数据量充足（如 10 万 + 样本）的场景。

缺点：计算成本高（需更新数十亿甚至千亿参数），易因小数据导致过拟合，且保存的模型体积大（与预训练模型相当）。

适用场景：数据充足（如通用文本分类）、算力充足（多卡 GPU）的任务。

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

针对全量微调的缺陷，PEFT 方法仅更新**少量参数**（通常占总参数的 0.1%-1%），在保证性能的同时大幅降低成本。主流方法包括：

LoRA（Low-Rank Adaptation，低秩适应）

原理：冻结预训练模型所有参数，在 Transformer 的注意力层中插入**低秩矩阵**（通过分解高维矩阵为两个低维矩阵乘积，减少参数），仅更新这些低秩矩阵参数。

优点：参数更新量极小（如 70 亿参数模型仅需更新百万级参数），训练速度快，保存的模型体积小（仅需存储低秩矩阵）。

适用场景：小数据任务（如专业领域问答）、算力有限的场景（单卡训练）。

Prefix Tuning（前缀调优）

原理：冻结预训练模型参数，仅在输入序列前添加一段可学习的 “前缀向量”（Prefix），模型通过调整前缀向量适配任务（前缀向量相当于 “任务提示”，引导模型生成任务相关输出）。

优点：参数更新量小（仅需学习前缀向量），尤其适合生成式任务（如文本摘要、对话）。

缺点：对分类等判别式任务效果略逊于 LoRA。

Adapter（适配器）

原理：在 Transformer 的每一层（或部分层）中插入小型 “适配器模块”（如两个全连接层 + 激活函数），冻结预训练参数，仅更新适配器模块的参数。

优点：模块化设计，可灵活插入不同层，适合多任务场景（不同任务共享预训练参数，仅替换适配器）。

缺点：参数更新量略大于 LoRA，训练速度稍慢。

LoRA

LoRA 通过 “低秩矩阵分解” 这一核心技巧，在冻结预训练模型参数的同时，仅用少量参数更新实现高效微调。其优势在于：**低成本（参数少、算力低）、高性能（接近全量微调）、易扩展（多任务切换）**，因此成为大模型（尤其是 7B、13B 等中小规模模型）落地的首选微调方案，广泛应用于情感分析、专业问答、文本生成等场景。

LoRA 的核心原理

LoRA（Low-Rank Adaptation，低秩适应）是当前大模型参数高效微调中最流行的方法之一，由微软团队在 2021 年提出。其核心思想是通过 “低秩矩阵分解” 减少微调时的参数更新量，在几乎不损失性能的前提下，大幅降低计算成本和存储开销。

LoRA 的设计基于两个关键观察：

1. 预训练模型已经学习了丰富的通用知识，微调的本质是在这些知识基础上进行 “微小调整” 以适配特定任务，而非彻底改变模型的参数分布。
2. 模型微调时的参数更新量通常具有 “低秩特性”—— 即更新矩阵可以用两个低维度矩阵的乘积近似表示，无需保存完整的高维更新矩阵。

1.低秩矩阵分解：核心数学基础

假设预训练模型某一层的权重矩阵为 $W \in R^{T \times d}$（T 为输入维度，d 为输出维度，通常是数万级，如 7B 模型的隐藏层维度为 4096）。全量微调时，我们需要更新整个矩阵 W，得到新权重 $W+\Delta W$（ΔW 为更新量）。

LoRA 的关键改进是：用两个低秩矩阵的乘积近似表示更新量 ΔW，即：$\Delta W=BA$
其中：

- $A \in R^{T \times r}$ 是随机初始化的低秩矩阵（输入维度→低秩维度），

- $B \in R^{r \times d}$ 是初始化为 0 的低秩矩阵（低秩维度→输出维度），
- r 是低秩维度（通常取 8、16、32，远小于 d 和 T，如 r=16≪4096）。

2. 模型结构：仅在关键层插入低秩矩阵

LoRA 并非对模型所有层进行修改，而是**仅在 Transformer 的注意力层（Query 和 Value 投影层）插入低秩矩阵**（实践证明这两层对任务适配最关键）。具体操作如下：

- 冻结预训练模型的所有原始参数（不更新 W），仅训练 A 和 B。
- 前向传播时，输入先经过原始权重 W 得到基础输出，再加上低秩矩阵的输出 BA×输入，最终结果作为该层的输出：输出=W×输入+BA×输入

3. 推理阶段：合并低秩矩阵以加速预测

训练完成后，LoRA 仅需保存低秩矩阵 A 和 B（参数总量约为 r×(T+d)，远小于原始权重）。推理时，可将 BA 与原始权重 W 合并为 W+BA，使模型结构与原始预训练模型完全一致，无需额外计算开销：W 合并 =W+BA

LoRA 的核心功能

LoRA 的设计直接解决了全量微调的痛点，核心功能体现在三个方面：

1. **大幅减少参数更新量，降低计算成本**

全量微调一个 7B 参数的模型（如 LLaMA-7B）需要更新 70 亿参数，而 LoRA 仅需更新低秩矩阵 A 和 B。以 r=16 为例，每一层的参数更新量约为 16×(4096+4096)=131072，整个模型（约 30 层注意力层）的总更新量仅约 400 万参数，仅为全量微调的 0.0057%。

这意味着：

- 训练时的显存占用从数十 GB 降至数 GB（单张 RTX 3090 即可训练 7B 模型）；
- 训练时间从数天缩短至数小时；
- 保存的模型文件从数十 GB（全量参数）降至数 MB（仅低秩矩阵）。

2. 避免小数据过拟合，保留预训练知识

全量微调时，若任务数据量小（如数千样本），模型容易过度拟合任务数据，破坏预训练阶段学习的通用知识（如语言规律）。

LoRA 通过冻结原始参数，仅用低秩矩阵进行“微调”，相当于在预训练知识的基础上做“局部修正”，既能适配任务，又能保留通用能力。例如，用小数据微调医疗问答模型时，LoRA 可让模型学会识别“药名”“症状”等专业术语，同时不丢失基本的语言理解能力。

3. 支持多任务灵活切换，降低存储开销

由于 LoRA 仅需保存低秩矩阵（数 MB 级），一个预训练模型可搭配多个任务的 LoRA 参数，实现 “一基多专”。例如：

- 基于 LLaMA-7B 模型，保存 “情感分析”“垃圾邮件分类”“金融问答” 三个任务的 LoRA 参数（共约 15MB）；
- 推理时只需加载基础模型 + 对应任务的 LoRA 参数，无需为每个任务保存完整模型（节省数百 GB 存储）。

实例：用 LoRA 微调 LLaMA-7B 模型做 “电影评论情感分析”

以具体任务为例，直观理解 LoRA 的使用流程和效果：

任务背景

目标：将通用的 LLaMA-7B 模型微调为 “电影评论情感分析模型”，输入电影评论（如 “剧情拖沓，浪费时间”），输出情感标签（0 = 负面，1 = 正面）。

步骤 1：数据准备

- 数据集：收集 1 万条电影评论，每条包含 “文本 + 标签”（如 “这部电影特效惊艳，推荐！”→1），划分为训练集（8000 条）和验证集（2000 条）。
- 预处理：用 LLaMA 的 Tokenizer 将文本转为 token ID（如 “特效”→3856），标签转为数字（0/1）。

步骤 2：配置 LoRA 参数

通过 Hugging Face 的 peft 库配置 LoRA（核心参数）：

```
from peft import LoraConfig, get_peft_model

lora_config = LoraConfig(
    r=16,  # 低秩维度，常用 8-32
    lora_alpha=32,  # 缩放因子（控制更新量的权重）
    target_modules=["q_proj", "v_proj"],  # 仅对注意力层的 Q 和 V 投影层插入 LoRA
    lora_dropout=0.05,  # dropout 防止过拟合
    bias="none",  # 不更新偏置参数
    task_type="SEQ_CLASSIFICATION"  # 任务类型：序列分类)
```

步骤 3：加载模型并应用 LoRA

```
from transformers import AutoModelForSequenceClassification, AutoTokenizer
# 加载预训练模型（LLaMA-7B）和 Tokenizer
model = AutoModelForSequenceClassification.from_pretrained(
    "chavinlo/alpaca-native",  # LLaMA-7B 的开源变体
    num_labels=2  # 情感标签：0（负面）、1（正面）)
tokenizer = AutoTokenizer.from_pretrained("chavinlo/alpaca-native")
# 冻结原始参数，仅添加 LoRA 模块
model = get_peft_model(model, lora_config)
```

model.print_trainable_parameters() # 输出可训练参数：约 400 万（仅占总参数的 0.005%）

步骤 4：训练模型

用训练集训练，仅更新 LoRA 的低秩矩阵 A 和 B：

```
from transformers import TrainingArguments, Trainer
training_args = TrainingArguments(
    output_dir="./lora_senti",
    per_device_train_batch_size=4,  # 单卡 batch size（显存有限时设小）
    learning_rate=2e-4,  # LoRA 学习率通常比全量微调高（因参数少）
    num_train_epochs=5,  # 小数据训练轮次少，避免过拟合
    evaluation_strategy="epoch"  # 每轮评估验证集)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset)

trainer.train()  # 训练约 2 小时（单张 RTX 3090）
```

步骤 5：推理与效果

训练完成后，将 LoRA 参数与基础模型合并，进行预测：

```
# 合并 LoRA 参数到基础模型
merged_model = model.merge_and_unload()
# 预测示例
text = "剧情拖沓，演员演技尴尬，不推荐"
inputs = tokenizer(text, return_tensors="pt")
outputs = merged_model(**inputs)
pred = outputs.logits.argmax().item()  # 输出 0（负面），符合预期
```

效果对比

- 性能：LoRA 微调的模型在测试集上准确率达 92%，与全量微调（93%）几乎持平。
- 成本：训练时间从全量微调的 3 天缩短至 2 小时，显存占用从 48GB 降至 12GB，模型存储从 13GB（全量参数）降至 8MB（仅 LoRA 参数）。

问题探究

不同参数层对任务的影响

在 LoRA 等参数高效微调方法中，注意力层（尤其是 query 和 value 矩阵）是核心微调对象，但模型的其他参数层（如嵌入层、前馈网络层、输出层等）对下游任务也有影响，只是其作用方式、调整优先级与注意力层存在显著差异。

微调最核心的参数是注意力层的 query 和 value 矩阵，这是因为不同任务关注点以及需要提取的信息侧重点是不同的，嵌入层则相对更加普遍，且参数量巨大，一般不轻易调整，除非预训练数据集和特定任务数据集文本差异极大；ffn 层更倾向于对注意力层提取的特征进一步的深化挖掘，是锦上添花而不是最核心的层。输出层则是必须微调，直接影响任务的适配。

具体分析如下：

一、嵌入层（Embeddings）：通用语义的“地基”

嵌入层是模型的输入层，负责将离散的 token（如单词）转换为连续的向量表示，包含：

- **词嵌入（word embeddings）**：编码单词的通用语义（如“猫”和“狗”的向量相似性）；
- **位置嵌入（position embeddings）**：编码 token 在序列中的位置信息；
- **段落嵌入（segment embeddings，如 BERT）**：编码句子边界信息。

影响与调整策略：

- **核心作用**：提供底层语义表示，是模型理解输入的“基础语言”。预训练模型的嵌入层在海量文本上学习了通用语义规律（如同义词、语义关联），对几乎所有下游任务都有价值。
- **为何通常冻结**：
    - 嵌入层参数量大（如 BERT-base 的词嵌入矩阵约占总参数的 10%），微调会显著增加参数规模，违背“参数高效”原则；
    - 下游任务数据集通常较小，微调嵌入层易导致过拟合（例如将“好”的嵌入过度适配特定情感标签，失去通用语义）。
- **特殊情况**：若下游任务的文本分布与预训练数据差异极大（如专业领域术语密集），可小范围微调嵌入层（如仅调整领域特有词的嵌入），但需配合正则化策略。

二、前馈网络层（FFN）：特征的“深化器”

Transformer 的每个编码器层（encoder.layer）在注意力层之后，都包含一个前馈网络（FFN），其结构通常为：

Linear(d_model, 4*d_model) → Gelu → Linear(4*d_model, d_model)

核心作用：

- 对注意力层输出的特征进行**非线性变换和维度映射**，强化局部特征（如短语级语义、语法结构）；
- 弥补注意力机制的不足（注意力更关注全局依赖，FFN 更擅长局部特征提炼）。

影响与调整策略：

- **作用次于注意力层**：FFN 的功能更偏向“特征深化”而非“任务适配”。它的输出会被传递到下一层，但其对任务特异性的影响（如情感分析中的情感倾向）弱于注意力层（注意力直接决定“关注哪些词”）。
- **微调性价比低**：

  FFN 参数量大（每层 FFN 的参数约是注意力层的 2 倍），若对所有 FFN 层微调，会导致新增参数过多（远超 LoRA 的 0.1%-1%）；

  实验表明（如 LoRA 原论文），仅微调注意力层已能覆盖大部分任务需求，额外微调 FFN 对效果提升有限，但会增加计算成本。

- **特殊场景**：在长文本理解（如文档摘要）等对局部特征敏感的任务中，可选择性微调顶层 FFN（接近输出的几层），但需控制范围。

三、输出层（Task-specific Head）：任务的“转换器”

输出层是模型的最后一层，负责将 Transformer 的特征映射到下游任务的输出空间（如分类任务的类别概率、生成任务的 token 分布）。
核心作用：

- 直接决定模型输出与任务目标的匹配度（如情感分析中输出“正面 / 负面”概率）。

影响与调整策略：

- **必须微调**：预训练模型的输出层（如 BERT 的 cls 头）是为预训练任务（如掩码语言模型）设计的，与下游任务不兼容，必须重新初始化或微调。
- **参数规模小**：输出层参数量极少（如二分类任务仅需一个 Linear(d_model, 2)层，约 2k 参数），微调不会增加太多负担。
- **与 LoRA 的配合**：输出层通常独立于 LoRA 的低秩矩阵调整，直接作为可训练参数参与优化（因为其任务特异性最强，必须完全适配下游目标）。

四、总结：不同层的“优先级”与设计逻辑

LoRA 等方法选择聚焦注意力层（而非其他层），本质是**按“任务影响度”和“参数效率”排序**的结果：

层类型	对任务的影响度	参数量	微调优先级	典型策略
注意力层（Q/V）	极高（决定关注重点）	中	最高	必微调（LoRA 核心）
输出层	高（直接输出结果）	极小	高	必微调（独立优化）
前馈网络层	中（深化特征）	大	低	通常冻结，特殊任务微调
嵌入层	低（通用语义）	中	极低	几乎不微调

这种设计的核心逻辑是：**用最少的参数调整，覆盖对下游任务最关键的模块**。注意力层是任务适配的“杠杆点”（微调少量参数即可显著改变模型行为），而其他层的作用更偏向“通用能力支撑”，冻结它们既能保留预训练知识，又能最大化参数效率。

当然，实际应用中可根据任务特性灵活调整（如领域差异大时微调部分嵌入层，长文本任务微调顶层 FFN），但需始终平衡“效果提升”与“参数成本”。

如何冻结特定参数

首先明确一下为什么要冻结参数？

参数储存着模型预训练后得到的通用知识和语言规律，底层参数（如词嵌入，早期注意力层）通常编码了基础语义特征（如词性，句法结构等），冻结这些参数可避免被下游任务的小数据集“污染”。冻结参数后可大大降低训练成本，缓解因小数据集导致的过拟合问题。概括来说，冻结参数是为了保留预训练知识，降低训练成本，缓解过拟合。

那么如何冻结参数呢？

冻结参数的本质是**阻止优化器更新这些参数**，在实现上通过设置参数的 requires_grad 属性为 False 完成。具体冻结哪些参数需根据任务特性决定：

- **全量冻结预训练参数**：仅训练新增的适配层（如 LoRA 的低秩矩阵），这是 LoRA 的标准做法。
- **部分冻结**：冻结底层参数（如前 N 层），训练高层参数 + 适配层。适用于下游任务与预训练目标差异较大的场景（如从文本生成到情感分析）。

参数冻结逻辑

通过 model.base_model.parameters()获取预训练模型主体参数（排除分类头），设置 requires_grad=False 使其不可训练。

```
## 方案 1：冻结所有预训练参数（LoRA 标准做法）

for param in model.base_model.parameters():  #获取的参数是字典，键为参数名，值为对应参数张量

    param.requires_grad = False  # 冻结 BERT 主体参数
```

部分冻结时，通过参数名称（如 layer.0.、layer.1.）区分不同层，选择性冻结底层。

```
## 方案 2：部分冻结（例如：冻结前 4 层，训练后 8 层+LoRA）

# for name, param in model.base_model.named_parameters():
```

```
#         if "layer." in name:
#             layer_num = int(name.split("layer.")[1].split(".")[0])
#             if layer_num < 4:   # 冻结前 4 层
#                 param.requires_grad = False
#         else:
#             param.requires_grad = False   # 冻结嵌入层等非层参数
```

LoRA 与冻结的协同

get_peft_model 会在 target_modules（如注意力层的 query/value 矩阵）上注入低秩矩阵（W = W0 + BA，其中 W0 是冻结的预训练参数，B/A 是可训练的低秩矩阵）。此时仅 B/A 参数可训练，实现参数高效微调。

```
# 4. 配置 LoRA 并注入适配器
lora_config = LoraConfig(
    r=8,  # 低秩矩阵的秩，控制参数数量
    lora_alpha=32,
    target_modules=["query", "value"],  # BERT 中注意力层的 query 和 value 矩阵
    lora_dropout=0.05,
    bias="none",  # 不训练偏置项
    task_type="SEQ_CLASSIFICATION",
)
model = get_peft_model(model, lora_config)  # 注入 LoRA 适配器，仅适配器参数可训练
```

验证冻结效果

model.print_trainable_parameters()可查看可训练参数占比（通常 LoRA 仅占 0.1%-1%），确认冻结是否生效。

lora 为什么仅需微调注意力层的 query 和 value 矩阵？

LoRA 选择仅微调注意力层的 query 和 value 矩阵，本质是因为：

1. 二者分别决定注意力的 "查询目标" 和 "提取内容"，是下游任务适配的核心；
2. 预训练模型的 key 矩阵已具备较强通用性，无需额外调整；
3. 这种选择能在参数效率和任务效果之间取得最优平衡。

在 LoRA（Low-Rank Adaptation）中，通常选择仅微调注意力层的 query 和 value 矩阵（而非 key 或其他模块），这一设计并非随机，而是基于注意力机制的核心作用、下游任务需求以及参数效率的综合考量。具体原因可从以下三方面理解：

1. 注意力机制中 query 和 value 的核心作用

Transformer 模型的注意力机制是其性能的核心，而 query、key、value 三个矩阵的功能存在本质差异：

- key 矩阵：将输入特征转换为 "键向量"（Key），主要用于提供上下文信息的 "标识"，其作用更偏向于**通用特征的编码**（如文本中的语义标识）。预训练模型在海量数据上已学习到较为通用的 key 表示，对下游任务的适配性较强，无需频繁调整。
- query 矩阵：将输入特征转换为 "查询向量"（Query），用于 "主动查询" 上下文信息（即 "我需要关注什么"）。其核心作用是**针对下游任务动态调整关注的内容**（如情感分析中关注情感词，命名实体识别中关注实体词），对任务特异性需求更敏感。
- value 矩阵：将输入特征转换为 "值向量"（Value），用于提供被关注位置的具体信息（即 "关注的内容是什么"）。其作用是**提取与任务相关的特征细节**，同样需要根据下游任务进行针对性调整。

简言之，query 决定 "关注哪里"，value 决定 "提取什么"，二者直接影响注意力分布和最终输出的任务相关性，是下游任务适配的关键；而 key 更多承担通用标识功能，预训练的通用特征已足够。

2. 下游任务对 query 和 value 的敏感性更高

LoRA 的目标是通过少量参数适配下游任务（如分类、问答等），而下游任务往往对模型的 "注意力分配能力" 要求极高：

- 例如在情感分析中，模型需要学会关注 "好 / 坏 / 精彩" 等情感词；在问答任务中，需要关注与问题相关的上下文片段。这些能力的核心是 query 矩阵对任务目标的 "理解"（即知道该查什么）和 value 矩阵对关键信息的 "提取"（即知道该取什么）。
- 相比之下，key 矩阵的主要功能是将文本转换为通用的 "键"，供 query 匹配，其通用特征在预训练阶段已充分学习，下游任务对其调整的需求较低。

因此，仅微调 query 和 value 即可高效提升模型对下游任务的适配性，无需改动 key 矩阵。

3. 参数效率与效果的平衡

LoRA 的核心优势是 "参数高效"（仅训练少量参数），而选择 query 和 value 作为目标模块，是在 "参数量" 和 "效果" 之间的最优权衡：

- 若同时微调 query、key、value，参数量会增加 50%（以 BERT 为例，每层注意力有 3 个矩阵，选 2 个可减少 1/3 参数），但效果提升有限（因 key 的贡献较低）。

- 若仅微调 query 或仅微调 value，虽参数更少，但无法同时覆盖 "关注哪里" 和 "提取什么" 两个核心环节，效果会下降。

实践证明（LoRA 原论文及后续研究），仅微调 query 和 value 矩阵，既能将新增参数量控制在极低水平（通常占模型总参数的 0.1%-1%），又能达到与全量微调接近的效果，是性价比最高的选择。

这一设计也体现了参数高效微调的核心思想：**精准定位对下游任务最关键的模块，用最少的参数实现最优适配**。

低秩为什么能够表征所有信息？

总结：低秩矩阵的“有效性”源于“任务特性 + 模型冗余”的结合
低秩矩阵并非“能表征所有信息”，而是**恰好能表征下游任务所需的“增量信息”**—— 因为：

1. 预训练模型已提供了大部分通用信息；
2. 下游任务的适配仅需少量关键维度的调整；
3. 低秩结构通过聚焦这些关键维度，用极少参数实现了高效适配。

这正是 LoRA 的精妙之处：不追求“完整表达所有可能的变化”，而是利用任务与模型的特性，用最低成本捕捉“最关键的调整”。

LoRA（Low-Rank Adaptation）中低秩矩阵能够有效表征下游任务所需信息的核心原因，并非“低秩矩阵本身能表达所有信息”，而是**预训练模型已具备强大的通用能力，下游任务的适配仅需在原有知识基础上做“细微调整”，而这些调整恰好可以通过低秩结构高效捕捉**。具体可从以下三个角度理解：

1. 低秩矩阵的本质：捕捉“关键变化方向”

低秩矩阵的核心特性是“参数少但能聚焦核心模式”。

假设预训练模型的某层权重矩阵为 W0 （高秩，如维度为 d×d），LoRA 通过低秩分解将参数更新表示为：ΔW=BA

其中 A∈Rd×r、B∈Rr×d，r≪d（通常 r=8,16）。

这种分解的本质是：**将高维空间中的参数更新压缩到 r 个“关键方向”上**。下游任务的适配不需要改变模型所有维度的权重，而只需在少数对任务敏感的维度上调整（例如情感分析中对“情感词”敏感的维度）。低秩矩阵 BA 正是通过这 r 个方向，精准捕捉这些关键调整，而忽略无关的冗余变化。

2. 预训练模型的“冗余性”：提供充足的“基础信息”

大型预训练模型（如 BERT、GPT）具有极强的“过度参数化”特性 —— 模型参数量远超拟合训练数据所需的最小值。这种冗余性使其：

- 已学习到海量通用知识（语法、语义、世界常识等），覆盖了下游任务所需的大部分基础信息；
- 存在大量“等价参数组合”，即不同的参数配置可实现相似的功能（例如用不同的权重矩阵组合表达相同的语义关联）。

因此，下游任务的适配不需要“从零开始学习”，而只需在预训练模型的基础上“修修补补”。这些“修补”本质上是对预训练知识的“微调”而非“重构”，而这种微调的幅度和方向往往是有限的（低秩的）。

3. 下游任务的“有限特异性”：调整需求可被低秩结构覆盖

下游任务（如分类、问答）与预训练任务（如掩码语言模型）虽有差异，但差异通常是“局部的”：

- 预训练任务已掌握通用的语言理解能力，下游任务只需调整模型对“任务特有信号”的关注度（例如情感分析关注情感词，命名实体识别关注实体词）；
- 这些“特有信号”对应的特征维度往往是有限的（远小于模型总维度 d），因此对模型参数的调整也集中在少数维度上，恰好符合低秩结构的表达能力。

实验也证实了这一点：LoRA 原论文中，即使 r=8（远小于 BERT 的 d=768），在多数任务上仍能达到与全量微调接近的效果，说明下游任务的调整需求确实可被低秩矩阵覆盖。

如何将保存的低秩参数与原参数融合得到微调后的模型？

在 LoRA 微调中，训练过程中保存的是低秩矩阵参数（即 LoRA 适配器的 A 和 B 矩阵），而非完整的模型参数。若要得到最终可用的微调模型，需要将这些低秩参数与预训练模型的原始参数融合（即计算 W=W0 +BA）。以下是具体的实现方法（基于 Hugging Face 的 peft 库）：

核心原理

LoRA 的参数存储形式是“原始预训练参数 W0 ”+“低秩矩阵 A 和 B”。融合时，需将低秩矩阵的乘积（BA）叠加到原始参数上，得到最终的微调参数 W。融合后，模型可脱离 LoRA 框架独立使用，与普通模型完全兼容。

实现步骤（代码示例）

假设已用 LoRA 训练并保存了模型（通常保存的是 peft_model 的适配器参数），融合步骤如下：

1. 加载预训练模型和 LoRA 适配器

首先需要加载原始预训练模型和训练好的 LoRA 低秩参数：

```
from transformers import AutoModelForSequenceClassification, AutoTokenizer
from peft import PeftModel
# 1. 加载原始预训练模型（与 LoRA 训练时使用的模型一致）
```

```
base_model_name = "bert-base-uncased"
base_model = AutoModelForSequenceClassification.from_pretrained(
    base_model_name,
    num_labels=2)
tokenizer = AutoTokenizer.from_pretrained(base_model_name)
# 2. 加载训练好的 LoRA 适配器（低秩参数）
lora_model_path = "./lora_results"  # LoRA 训练时的 output_dir
model = PeftModel.from_pretrained(base_model, lora_model_path)
```

2. 融合低秩参数与原始参数

使用 peft 库提供的 merge_and_unload()方法，自动完成参数融合并移除 LoRA 适配器：

```
# 3. 融合参数（核心步骤）
merged_model = model.merge_and_unload()
```

merge_and_unload()的作用：

- 遍历所有注入了 LoRA 适配器的模块（如注意力层的 query 和 value 矩阵）；
- 计算 W=W0 +BA，将低秩矩阵的更新合并到原始参数中；
- 移除模型中的 LoRA 适配器结构，返回一个纯 PyTorch 模型（与普通预训练模型结构一致）。

3. 保存融合后的模型

融合后的模型可像普通模型一样保存和加载：

```
# 4. 保存融合后的完整模型
merged_model.save_pretrained("./final_finetuned_model")
tokenizer.save_pretrained("./final_finetuned_model")
```

4. 验证融合效果

可通过对比融合前后的模型输出，确认融合是否正确：

```
# 测试文本
text = "This movie is amazing! I love it."
inputs = tokenizer(text, return_tensors="pt")
# 融合前的 LoRA 模型输出 with torch.no_grad():
    outputs_lora = model(**inputs)
    logits_lora = outputs_lora.logits
# 融合后的模型输出（应与 LoRA 模型完全一致）
with torch.no_grad():
    outputs_merged = merged_model(** inputs)
    logits_merged = outputs_merged.logits
print("LoRA 模型输出:", logits_lora)
print("融合后模型输出:", logits_merged)  # 两者应完全相同
```

关键细节

1. **融合时机**：通常在训练完成后、部署前进行融合，因为融合后的模型推理速度更快（无需实时计算 W0 +BA）。
2. **参数占用**：融合后的模型参数量与原始预训练模型相同（因为只是修改了参数值，未改变结构），而 LoRA 适配器本身仅占用少量磁盘空间（通常几 MB）。
3. **可逆性**：融合是单向操作（融合后无法分离出 LoRA 适配器），因此建议同时保留原始预训练模型和 LoRA 适配器，以便后续调整或重新融合。

总结

通过 peft 库的 merge_and_unload()方法，可一键完成 LoRA 低秩参数与原始参数的融合，得到一个可独立使用的微调模型。这一步骤是 LoRA 训练流程的最后环节，确保模型能以高效方式部署到生产环境。

显存是什么？和参数的关系是什么？

显存（Video Memory，简称 VRAM）是显卡（GPU）上的专用内存，主要用于临时存储 GPU 处理的数据，包括模型参数、输入数据、中间计算结果等。它是 GPU 高效运行的核心资源，直接影响模型训练和推理的速度与可行性。

显存与参数的关系

模型参数是显存的主要占用者之一，二者的关系可以从以下几个角度理解：

1. 参数直接占用显存

模型的每一个参数（如权重矩阵、偏置项）在加载到 GPU 时，都会占用一定的显存空间。具体来说：

- 一个 32 位浮点数（float32）参数占用 4 字节显存；
- 一个 16 位浮点数（float16）参数占用 2 字节显存；
- 更低精度（如 int8、float8）的参数占用空间更少。

例如，一个 10 亿参数的模型（如 GPT-2）：

- 用 float32 存储时，显存占用约 4GB（10 亿 × 4 字节）；
- 用 float16 存储时，显存占用约 2GB，节省一半空间。

2. 显存不仅包含参数，还有其他开销

除了模型参数，显存还需要存储：

- **输入数据**：如批量的文本、图像等（批量越大，占用越多）；
- **中间计算结果**：模型各层的输出张量（如 Transformer 的注意力矩阵、前馈网络输出等）；

- **优化器状态**：训练时优化器（如 Adam）需要存储的动量、二阶矩等信息（通常是参数的 2-3 倍大小）。

因此，实际显存占用远大于模型参数本身的大小。例如，训练一个 10 亿参数的模型，加上中间结果和优化器状态，可能需要 20GB 以上的显存。

3. 参数效率影响显存需求

参数越少的模型（或参数高效微调方法，如 LoRA），对显存的需求越低：

- 全量微调时，模型所有参数都需要加载到显存，且优化器需存储所有参数的状态；
- LoRA 仅训练少量低秩矩阵参数（通常是原模型的 0.1%-1%），显存占用可大幅降低（主要节省优化器状态的显存）。

4. 显存不足会导致训练 / 推理失败

如果模型参数 + 中间数据的总大小超过显存容量，会触发“显存溢出（OOM, Out Of Memory）”错误，导致程序中断。因此，显存大小是限制可训练模型规模的关键因素（例如，显存不足时无法训练超大模型）。

总结

- **显存**是 GPU 的专用内存，用于存储模型运行时的所有数据（参数、输入、中间结果等）。
- **参数**是显存的重要组成部分，其数量和精度直接影响显存占用。
- 显存需求 = 参数占用 + 输入数据 + 中间结果 + 优化器状态，因此实际占用远大于参数本身。
- 减少参数数量（如用 LoRA）或降低精度（如 float16），是降低显存需求的主要手段。

model.base_model.parameters()

在 PyTorch 中，model.base_model.parameters()返回的是一个**参数迭代器（generator）**，其中每个元素是一个独立的 torch.Tensor 对象（即模型中的单个参数矩阵或向量，如权重矩阵、偏置项等）。

它的结构特点可以总结为：

1. 数据结构本质

- **不是字典**，也不是按层分组的列表，而是**扁平的参数迭代器**，直接遍历所有可训练参数（以 torch.Tensor 为单位）。
- 迭代顺序与模型的层结构一致（从输入层到输出层依次返回各层的参数）。

2. 参数的层级对应关系

以 BERT 模型为例，model.base_model 是完整的 Transformer 结构，包含：

- embeddings（词嵌入层）
- encoder（编码器层，内含 12 层 layer）
- 每个 layer 包含 attention（注意力层）、intermediate（中间层）、output（输出层）等子模块

parameters()迭代器会按以下顺序返回参数：

1. 先返回 embeddings 层的所有参数（词嵌入矩阵、位置嵌入矩阵等）
2. 再依次返回 encoder.layer.0 到 encoder.layer.11 每层的参数：
    - 每层的注意力层（query/key/value 权重矩阵、偏置项等）
    - 每层的中间层（dense 权重矩阵、偏置项等）
    - 每层的输出层（dense 权重矩阵、偏置项等）

3. 如何查看参数的层级信息？

如果需要知道每个参数对应的层结构，可以用 named_parameters()（返回(参数名称, 参数张量)的迭代器），例如：

```
for name, param in model.base_model.named_parameters():
    print(name)  # 打印参数对应的层级名称
    # 示例输出：
    # encoder.layer.0.attention.self.query.weight
    # encoder.layer.0.attention.self.query.bias
    # ...（后续参数）
```

通过参数名称可以清晰看到其所属的层（如 layer.0 表示第 1 层）和具体模块（如 attention.self.query 表示注意力层的 query 矩阵）。

总结

- model.base_model.parameters()返回**单个参数张量（Tensor）的迭代器**，按模型层结构顺序排列。
- 若需查看参数对应的层级信息，使用 named_parameters()获取(名称, 张量)对，名称中包含完整的层级路径。

这种设计的目的是方便遍历所有参数进行统一操作（如冻结、初始化等），而无需关心具体的层级结构。