# 课程大作业二基础题:GPU学习智能体（agent）

✓补充资料（一）

# LoRA的核心原理

- ✓SFT（全参数监督微调）：传统全量微调需要更新模型的所有参数（例如 Llama-2-7B 有 70 亿参数），显存占用极高（往往需要几十 GB 甚至上百 GB），且训练效率低。
- ✓LoRA（Low-Rank Adaptation）是一种参数高效微调技术，核心是通过冻结预训练模型的大部分参数，仅训练少量低秩矩阵参数来适配下游任务，从而在大幅降低计算和显存成本的同时，保持甚至超越全量微调的效果。

# LoRA的核心原理

- ✓冻结预训练权重：保持预训练模型的原始参数不变，避免破坏模型已学习的通用知识。
- ✓低秩矩阵分解：对于模型中需要更新的权重（如注意力层的Wq/Wk/Wv/Wo），不直接更新原始权重，而是通过两个低秩矩阵（W_A和W_B）的乘积来近似权重的增量（$\Delta W = W\_A \times W\_B$）。
  - ✓假设原始权重矩阵维度为$d \times d$，LoRA 将其分解为维度为$d \times r$的W_A和维度为$r \times d$的W_B，其中$r$是“秩”（通常取 8 - 32，远小于d）。
  - ✓参数数量从$d^2$减少到$r \times d + r \times d = 2rd$，例如$d$=4096、$r$=16时，参数仅为原来的$(2 \times 16 \times 4096)/(4096^2) \approx 0.8\%$。
- ✓推理时合并：训练完成后，将ΔW与原始权重W合并（$W\_new = W + \Delta W$），不影响推理效率。

# 数据准备（两者通用）

```
# 示例数据：（输入，输出）格式，适配对话场景
data = [
    {"input": "什么是人工智能？", "output": "人工智能是模拟人类智能的技术..."},
    {"input": "机器学习和深度学习的区别？", "output": "机器学习是基础，深度学习是其分支..."},
]
# 转换为 Unsloth 要求的格式（对话模板）
from unsloth import to_packed_dataset
dataset = to_packed_dataset(data, input_column="input", output_column="output")
```

# LoRA 微调（Unsloth 实现）

✓核心：冻结原模型参数，仅训练低秩矩阵

✓关键：配置lora_parameters

```
from unsloth import FastLanguageModel, training_args
# 1. 加载模型（4-bit 量化，Unsloth 核心优化）
model, tokenizer =
FastLanguageModel.from_pretrained(
    model_name="unsloth/llama-2-7b-chat",
    load_in_4bit=True,  # 启用 4-bit 量化，降低显存
    max_seq_length=2048,
)
```

# LoRA 微调（Unsloth 实现）

```
# 2. 配置 LoRA（核心：仅训练低秩矩阵）
model = FastLanguageModel.get_peft_model(
    model,
    r=16,  # 低秩矩阵的秩（核心参数，r 越大能力越强但参数越多，通常 8-32）
    lora_alpha=32,  # 缩放因子，一般为 r 的 2 倍
    lora_dropout=0.05,
    target_modules=["q_proj", "v_proj"],  # 仅对注意力层的 Q/V 矩阵注入 LoRA
    use_gradient_checkpointing="unsloth",  # Unsloth 特有的梯度 checkpoint 优化
    random_state=42,
    use_rslora=False,  # 禁用 R-LoRA（简化版）
    loftq_config=None,
)
# 查看可训练参数：仅约 100 万（7B 模型的 0.01%）
print(f"可训练参数: {model.print_trainable_parameters()}")  # 输出: Trainable params: 1,100,800
```

# LoRA 微调（Unsloth 实现）

```
# 3. 配置训练参数（低资源即可）
training_args = training_args(
    per_device_train_batch_size=2,
    gradient_accumulation_steps=4,
    learning_rate=2e-4,  # LoRA 学习率通常比 SFT 高（参数少，需更大步长）
    num_train_epochs=3,
    fp16=not FastLanguageModel.is_apple_silicon(),  # 自动适配设备
    logging_steps=10,
    output_dir="./lora_results",
)
```

# LoRA 微调（Unsloth 实现）

```
# 4. 开始训练（Unsloth 加速训练循环）
from trl import SFTTrainer
trainer = SFTTrainer(
    model=model,
    train_dataset=dataset,
    peft_config=model.peft_config,  # 传入
LoRA 配置
    dataset_text_field="text",
    max_seq_length=2048,
    tokenizer=tokenizer,
    args=training_args,
)
trainer.train()
```

# LoRA 微调（Unsloth 实现）

```
# 5. 保存模型（仅保存 LoRA 适配器，体积 < 10MB）
model.save_pretrained("llama-2-7b-lora")
```

***助教同学将为大家提供一份基础题的模板代码！***

# SFT（全参数微调，Unsloth 实现）

✓核心：解冻所有参数，更新原模型权重，无需配置 LoRA 参数，但需要更多资源

```
from unsloth import FastLanguageModel, training_args
# 1. 加载模型（4-bit 量化，Unsloth 优化后可单卡运行）
model, tokenizer =
FastLanguageModel.from_pretrained(
    model_name="unsloth/llama-2-7b-chat",
    load_in_4bit=True,  # 同样支持 4-bit，但全参数更新仍比 LoRA 耗显存
    max_seq_length=2048,
)
```

# SFT（全参数微调，Unsloth 实现）

```
# 2. SFT 无需 LoRA 配置：解冻所有参数（Unsloth 自动处理）
# 查看可训练参数：约 70 亿（全部参数）
print(f"可训练参数: {model.print_trainable_parameters()}")  # 输出：Trainable params: 7,000,000,000

# 3. 配置训练参数（资源需求更高）
training_args = training_args(
    per_device_train_batch_size=1,  # 批次更小，避免显存溢出
    gradient_accumulation_steps=8,  # 累积梯度弥补小批次
    learning_rate=2e-5,  # SFT 学习率通常更低（避免破坏原模型知识）
    num_train_epochs=3,
    fp16=not FastLanguageModel.is_apple_silicon(),
    logging_steps=10,
    output_dir="./sft_results",
)
```

# SFT（全参数微调，Unsloth 实现）

```
# 4. 开始训练（Unsloth 优化训练循环，比原生 HuggingFace 快 2-3 倍）
from trl import SFTTrainer
trainer = SFTTrainer(
    model=model,
    train_dataset=dataset,
    dataset_text_field="text",
    max_seq_length=2048,
    tokenizer=tokenizer,
    args=training_args,
    peft_config=None,  # SFT 无需 LoRA 配置
)
trainer.train()

# 5. 保存模型（保存完整模型权重，体积 ~13GB）
model.save_pretrained("llama-2-7b-sft")
```

# 检索增强生成（Retrieval-Augmented Generation，RAG）

- ✓RAG（Retrieval-Augmented Generation，检索增强生成）：一种结合信息检索和生成式AI的技术，核心目标是让语言模型（LLM）在回答问题时，不仅依赖自身预训练的“内置知识”，还能实时检索外部文档、数据库等“外挂知识”，从而生成更准确、更可靠、更具时效性的内容。

# 检索增强生成（Retrieval-Augmented Generation，RAG）

- ✓RAG 的工作流程（核心四步）：
  - ✓文档预处理（离线准备）：将原始知识（如 PDF、TXT、网页等）转化为模型可检索的格式
  - ✓向量存储（离线存储）：将嵌入后的向量存入向量数据库（如 Chroma、FAISS、Milvus），方便后续快速检索
  - ✓检索（在线触发）：当用户输入问题时，实时从向量库中找到相关知识
  - ✓生成（在线生成）：将问题和检索到的上下文一起输入语言模型，让模型基于这些信息生成回答

# 检索增强生成（Retrieval-Augmented Generation，RAG）

- ✓环境准备，安装依赖：
  - ✓安装 Unsloth（高效 LLM 微调库）、LangChain（RAG 流程管理）、Chroma（轻量向量数据库）等核心工具

```
# 安装依赖（首次运行需执行）
!pip install "unsloth[colab-new] @ git+https://github.com/unsloth/unsloth.git"
!pip install langchain chromadb sentence-transformers python-dotenv
```

# 检索增强生成（Retrieval-Augmented Generation，RAG）

- ✓导入核心库：
  - ✓加载实现 RAG 所需的模块，分工明确：Unsloth 负责 LLM，LangChain 负责流程，Chroma 负责向量存储

```
# 1. LLM相关：Unsloth加载高效模型
from unsloth import FastLanguageModel
from unsloth.chat_templates import get_chat_template

# 2. RAG流程相关：LangChain处理文档、检索
from langchain.document_loaders import TextLoader  # 加载文本文档
from langchain.text_splitter import RecursiveCharacterTextSplitter  # 分割文档
from langchain.vectorstores import Chroma  # 向量数据库
from langchain.embeddings import HuggingFaceEmbeddings  # 文本转向量（嵌入模型）
from langchain.chains import RetrievalQA  # RAG链（检索+生成）
```

# 检索增强生成（Retrieval-Augmented Generation，RAG）

- ✓RAG处理文档（加载→分割→转向量）：
  - ✓RAG 的核心是“先检索相关文档”，需先将原始文档拆成小块并存储为向量（方便快速匹配）

```
# 3.1 加载文档（这里以TXT为例，可替换为PDF/Word，需对应Loader如PyPDFLoader）
loader = TextLoader("your_document.txt")  # 替换为你的文档路径
documents = loader.load()
```

# 检索增强生成（Retrieval-Augmented Generation，RAG）

```
# 3.2 分割文档（关键！LLM有上下文长度限制，需拆成小块）
text_splitter = RecursiveCharacterTextSplitter(
    chunk_size=1000,  # 每个小块1000个字符（根据LLM上下文调整，如Llama-2 7B支持4096）
    chunk_overlap=200,  # 小块间重叠200字符（避免拆分切断语义，如“苹果”拆成“苹”和“果”）
    length_function=len  # 按字符数计算长度
)
split_docs =
text_splitter.split_documents(documents)  # 分割后的数据
```

# 检索增强生成（Retrieval-Augmented Generation，RAG）

```
# 3.3 初始化嵌入模型（将文本转为数值向量，用于计算相似度）
embedding_model = HuggingFaceEmbeddings(
    model_name="all-MiniLM-L6-v2"  # 轻量高效模型，适合检索（速度快、显存占用低）
)

# 3.4 构建向量数据库（将分割后的文档块转向量，存入Chroma）
vector_db = Chroma.from_documents(
    documents=split_docs,  # 分割后的文档块
    embedding=embedding_model,  # 用上面的模型转向量
    persist_directory="./chroma_db"  # 向量数据保存路径（下次可直接加载，不用重复构建）
)
vector_db.persist()  # 保存向量库（避免重启后丢失）
```

# 检索增强生成（Retrieval-Augmented Generation，RAG）

## ✓用 Unsloth 加载LLM

```
# 4.1 加载Unsloth优化的LLM（以Llama-2 7B为例，支持其他模型如Mistral、Phi-2）
model, tokenizer = FastLanguageModel.from_pretrained(
    model_name="unsloth/llama-2-7b-chat",  # Unsloth预优化的模型（chat版本适合对话生成）
    max_seq_length=4096,  # LLM最大上下文长度（需≥文档块大小+问题长度）
    dtype=None,  # 自动匹配数据类型（GPU用float16，CPU用float32）
    load_in_4bit=True,  # 关键！开启4-bit量化，显存占用从13GB→4GB
)

# 4.2 适配LLM为LangChain格式（Unsloth模型需转换后才能被LangChain调用）
model_for_langchain = FastLanguageModel.get_langchain_model(model, tokenizer)

# 4.3 设置聊天模板（让LLM理解“问题+检索到的文档”格式，避免输出混乱）
tokenizer = get_chat_template(
    tokenizer,
    chat_template="llama-2",  # 匹配Llama-2的对话格式
    mapping={"human": "USER", "ai": "ASSISTANT"},  # 定义角色名称
)
```

# 检索增强生成（Retrieval-Augmented Generation，RAG）

- ✓构建 RAG 链并运行：
  - ✓将“检索器”与“LLM”串联，实现“先检索相关文档→再用 LLM 基于文档回答”

```
# 5.1 构建检索器（从向量库中快速找与问题最相关的文档块）
retriever = vector_db.as_retriever(
    search_kwargs={"k": 3}  # 每次检索返回Top3最相关的文档块（k值越大，信息越全但速度越慢）
)

# 5.2 构建RAG链（检索器+LLM的组合）
rag_chain = RetrievalQA.from_chain_type(
    llm=model_for_langchain,  # Unsloth加载的高效LLM
    chain_type="stuff",  # 简单高效的链类型：将检索到的文档块“塞”进LLM的prompt中
    retriever=retriever,  # 上面构建的检索器
    return_source_documents=True  # 回答时返回用到的原始文档块（方便验证来源）
)
```

# 检索增强生成（Retrieval-Augmented Generation，RAG）

```
# 5.3 运行RAG（输入问题，得到基于文档的回答）
question = "请解释文档中提到的‘人工智能的核心技术’？"  # 你的问题
result = rag_chain({"query": question})

# 5.4 输出结果
print("RAG回答：\n", result["result"])
print("\n用到的参考文档：")
for doc in result["source_documents"]:
    print(f"- 内容：{doc.page_content[:200]}...")  # 显示前200字符（避免过长）
    print(f"  来源：{doc.metadata['source']}")  # 显示文档路径
```

# 人类偏好对齐（PPO为例）

- ✓PPO（Proximal Policy Optimization，近端策略优化）：强化学习中常用的算法，尤其在大模型RLHF（基于人类反馈的强化学习）流程中，用于将“监督微调（SFT）模型”优化为“符合人类偏好的策略模型”。
- ✓PPO 的核心：通过“奖励信号”调整模型输出
  - ✓让模型（策略网络，Policy）生成回答
  - ✓用奖励模型（Reward Model）或人工反馈给回答打分（奖励值）
  - ✓基于奖励值更新策略网络，让模型更倾向于生成高分回答（同时限制更新幅度，避免模型剧烈波动）

# 人类偏好对齐（PPO为例）

✓第一步、安装依赖

```
# 安装 TRL（提供 PPO 实现）和强化学习相关库
pip install trl==0.7.4 peft==0.7.1 datasets==2.14.6 accelerate==0.24.1
```

# 人类偏好对齐（PPO为例）

## ✓第二步、PPO训练

```
# ---------------------------------------------------
# 步骤1：导入核心库
# ---------------------------------------------------
from unsloth import FastLanguageModel
from trl import PPOTrainer, PPOConfig,
AutoModelForCausalLMWithValueHead
from trl.core import respond_to_batch
from datasets import Dataset
import torch
```

# 人类偏好对齐（PPO为例）

```
# ----------------------------------------
# 步骤2：配置参数（根据显存调整）
# ----------------------------------------
class Config:
    MODEL_NAME = "unsloth/llama-2-7b-chat"  # 基础模型（Unsloth优化版）
    MAX_SEQ_LENGTH = 2048  # 最大序列长度
    PPO_BATCH_SIZE = 2  # PPO批次大小（显存小则调小）
    LEARNING_RATE = 2e-5  # PPO学习率（通常比SFT低）
    NUM_EPOCHS = 3  # 训练轮次
    REWARD_SCALE = 1.0  # 奖励缩放因子（控制奖励影响程度）
    LOAD_IN_4BIT = True  # 启用4-bit量化（关键：降低显存）
```

# 人类偏好对齐（PPO为例）

```
# ----------------------------------------------------
# 步骤3：加载基础模型（用Unsloth优化）
# ----------------------------------------------------
# 加载模型和分词器（4-bit量化）
model, tokenizer = FastLanguageModel.from_pretrained(
    model_name=Config.MODEL_NAME,
    load_in_4bit=Config.LOAD_IN_4BIT,
    max_seq_length=Config.MAX_SEQ_LENGTH,
)
# 为模型添加价值头（Value Head）：PPO需要策略网络（生成回答）和价值网络（预测奖励）
# Unsloth模型需转换为TRL兼容的带价值头模型
model = AutoModelForCausalLMWithValueHead.from_pretrained(
    model,
    device_map="auto",  # 自动分配设备（GPU优先）
)
# 配置分词器（设置pad_token，避免警告）
tokenizer.pad_token = tokenizer.eos_token
```

# 人类偏好对齐（PPO为例）

```
# ------------------------------------------------
# 步骤4：准备训练数据（对话样本）
# ------------------------------------------------
# 示例数据：(查询，初始回答)，模拟SFT后的模型输出
data = [
    {
        "query": "推荐一本机器学习入门书籍",
        "response": "《机器学习实战》适合入门，案例丰富，适合动手实践。"
    },
    {
        "query": "什么是过拟合？",
        "response": "过拟合是模型在训练数据上表现好，但在新数据上表现差的现象。"
    },
]
# 转换为PPO所需的数据集格式（需包含"query"和"response"字段）
dataset = Dataset.from_list(data)
# 格式化数据：将query和response拼接为对话格式（适配Llama-2模板）
def format_prompt(sample):
    return f"USER: {sample['query']}\nASSISTANT: {sample['response']}"
dataset = dataset.map(lambda x: {"text": format_prompt(x)})
```

# 人类偏好对齐（PPO为例）

```
# ------------------------------------------------
# 步骤5：定义奖励函数（核心：告诉模型“什么是好回答”）
# 这里用简单规则模拟奖励模型（实际可用训练好的Reward Model）
# ------------------------------------------------
def reward_function(responses):
    """
    输入：模型生成的回答列表
    输出：每个回答的奖励值（越高表示越符合偏好）
    这里的规则：长度越长+包含关键词（如“推荐”“解释”）则奖励越高
    """
    rewards = []
    for response in responses:
        # 基础奖励（长度）
        reward = len(response) / 100  # 长度越长，奖励越高（归一化）
        # 关键词奖励（包含目标词加分）
        keywords = ["推荐", "解释", "步骤", "原因"]
        for kw in keywords:
            if kw in response:
                reward += 0.5
        # 限制奖励范围（避免极端值）
        rewards.append(torch.tensor(reward * Config.REWARD_SCALE))
    return rewards
```

# 人类偏好对齐（PPO为例）

```
# ---------------------------------------------------
# 步骤6：配置PPO训练参数
# ---------------------------------------------------
ppo_config = PPOConfig(
    batch_size=Config.PPO_BATCH_SIZE,
    learning_rate=Config.LEARNING_RATE,
    num_train_epochs=Config.NUM_EPOCHS,
    logging_steps=1,  # 每步打印日志
    optimize_cuda_cache=True,  # 优化CUDA缓存（节省显存）
    gradient_accumulation_steps=4,  # 梯度累积（小批次模拟大批次）
    clip_ratio=0.2,  # PPO核心参数：限制策略更新幅度（避免过大波
动）
    vf_coef=0.1,  # 价值损失系数（平衡策略损失和价值损失）
    ent_coef=0.01,  # 熵系数（鼓励探索，避免模型输出过于单一）
)
```

# 人类偏好对齐（PPO为例）

```
# ------------------------------------------------
--------
# 步骤7：初始化PPO Trainer
# ------------------------------------------------
--------
ppo_trainer = PPOTrainer(
    model=model,
    config=ppo_config,
    dataset=dataset,
    tokenizer=tokenizer,
)
```

# 人类偏好对齐（PPO为例）

```
# ------------------------------------------------------
# 步骤8：运行PPO训练
# ------------------------------------------------------
# 训练循环
for epoch in range(Config.NUM_EPOCHS):
    print(f"\n===== 第 {epoch+1} 轮训练 =====")
    # 迭代数据集（按PPO批次大小处理）
    for batch in ppo_trainer.dataloader:
        # 1. 从批次中提取查询（query）和原始回答（response）
        queries = [f"USER: {q}\nASSISTANT:" for q in batch["query"]]
        responses = batch["response"]

        # 2. 用当前模型生成回答（策略网络输出）
        # 注意：需用tokenizer编码查询，作为模型输入
        inputs = tokenizer(queries, return_tensors="pt", padding=True, truncation=True).to("cuda")
        generated_responses = respond_to_batch(model, inputs, tokenizer, max_new_tokens=100)

        # 3. 计算奖励（用自定义奖励函数）
        rewards = reward_function(generated_responses)

        # 4...
```

# 人类偏好对齐（PPO为例）

```
# ------------------------------------------------
# 步骤8：运行PPO训练
# ------------------------------------------------
# 训练循环
for epoch in range(Config.NUM_EPOCHS):
    print(f"\n===== 第 {epoch+1} 轮训练 =====")
    # 迭代数据集（按PPO批次大小处理）
    for batch in ppo_trainer.dataloader:
        # 1...
        # 2...
        # 3...
        # 4. 准备PPO训练的输入数据（查询、回答、奖励）
        # 将文本转换为token ID（PPO内部需要token级别的输入）
        query_tensors = [tokenizer(q, return_tensors="pt", truncation=True).input_ids[0] for q in
queries]
        response_tensors = [tokenizer(r, return_tensors="pt", truncation=True).input_ids[0] for r in
generated_responses]

        # 5. 执行PPO更新（核心步骤）
        stats = ppo_trainer.step(query_tensors, response_tensors, rewards)

        # 6. 打印训练 stats（监控损失、奖励等指标）
        ppo_trainer.log_stats(stats, batch, rewards)
```

# 人类偏好对齐（PPO为例）

```
# ------------------------------------------------------------
# 步骤9：保存训练后的模型
# ------------------------------------------------------------
model.save_pretrained("llama-2-7b-ppo-unsloth")
tokenizer.save_pretrained("llama-2-7b-ppo-unsloth")
print("PPO训练完成，模型已保存！")
```

# 高效推理（vLLM为例）

- ✓vLLM ： 加载微调后的模型进行高效推理（支持动态批处理、PagedAttention 等技术，提升吞吐量）。
- ✓第一步、环境准备（安装依赖）

```
# 安装VLLM（推理核心，需匹配CUDA版本）
# 注意：VLLM对CUDA版本要求较高（推荐11.8+），根据实际环境选择
pip install vllm==0.4.0.post1
```

# 高效推理（vLLM为例）

- ✓第二步、Unsloth 微调模型（以 LoRA 为例）
- ✓第三步、合并 LoRA 权重到基础模型（供 vLLM 加载）
  - ✓vLLM 需加载完整模型权重（而非单独的 LoRA 适配器），因此需将 Unsloth 微调的 LoRA 权重合并到基础模型中。

# 高效推理（vLLM为例）

```
# 1. 重新加载基础模型（关闭4-bit，用于合并权重）
model, tokenizer = FastLanguageModel.from_pretrained(
    model_name="unsloth/llama-2-7b-chat",
    load_in_4bit=False,  # 合并权重需用FP16/FP32
    max_seq_length=2048,
)

# 2. 加载之前保存的LoRA权重
from peft import PeftModel
model = PeftModel.from_pretrained(model, "llama-2-7b-lora")

# 3. 合并LoRA权重到基础模型（关键步骤）
model = model.merge_and_unload()  # 合并后模型为完整权重

# 4. 保存合并后的模型（供VLLM加载）
merged_model_path = "./llama-2-7b-merged"
model.save_pretrained(merged_model_path)
tokenizer.save_pretrained(merged_model_path)
print(f"合并后的模型已保存至：{merged_model_path}")
```

# 高效推理（vLLM为例）

## ✓第四步、vLLM 加载合并模型进行高效推理

### ✓vLLM 通过 PagedAttention 技术优化显存使用，支持高并发请求，适合生产环境部署。

```
# 1. 导入vLLM推理库
from vllm import LLM, SamplingParams

# 2. 配置采样参数（控制生成效果）
sampling_params = SamplingParams(
    temperature=0.7,  # 随机性（0=确定性，1=高随机）
    top_p=0.9,  # 核采样
    max_tokens=512,  # 最大生成长度
)

# 3. 用vLLM加载合并后的模型
# 注意：model_path需指向合并后的模型目录
llm = LLM(
    model=merged_model_path,
    tensor_parallel_size=1,  # 按GPU数量设置（如2张卡设为2）
    gpu_memory_utilization=0.9,  # 允许使用的GPU显存比例
    max_num_batched_tokens=4096,  # 动态批处理最大token数（提升吞吐量）
    trust_remote_code=True,
)
```

# 高效推理（vLLM为例）

```
# 4. 推理示例（支持批量请求）
prompts = [
    "USER: 什么是过拟合？ ASSISTANT:",
    "USER: 如何解决过拟合？ ASSISTANT:",
]

# 5. 生成结果
outputs = llm.generate(prompts, sampling_params)

# 6. 打印结果
for output in outputs:
    prompt = output.prompt
    generated_text = output.outputs[0].text
    print(f"输入: {prompt}\n输出: {generated_text}\n---
")
```