

第5章 微调与对齐

让大模型听懂人话、遵循指令、符合人类价值观的关键技术

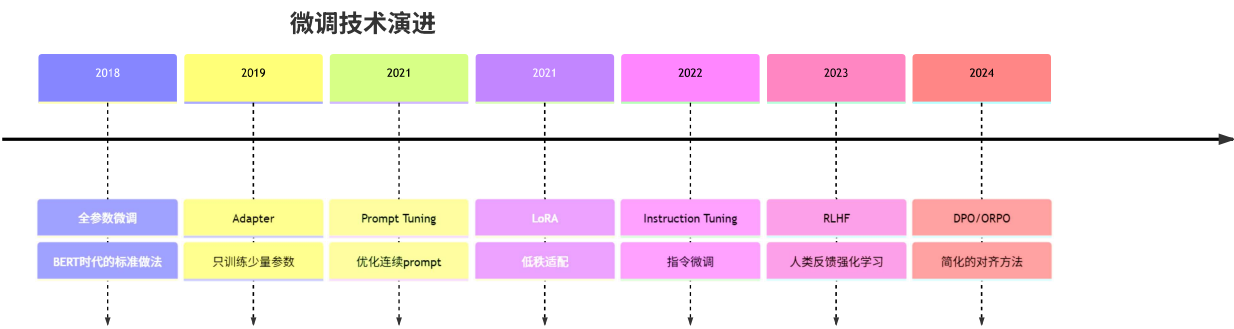
5.1 微调概述

5.1.1 什么是微调？

定义： 在预训练模型基础上，使用特定任务的数据进行进一步训练。



5.1.2 微调的发展



5.2 Full Fine-Tuning（全参数微调）

5.2.1 标准微调流程

```
from transformers import AutoModelForCausalLM, AutoTokenizer, Trainer

# 1. 加载预训练模型
model = AutoModelForCausalLM.from_pretrained("gpt2")
tokenizer = AutoTokenizer.from_pretrained("gpt2")

# 2. 准备数据
train_dataset = load_dataset("your_dataset")

# 3. 训练配置
training_args = TrainingArguments(
    output_dir="./results",
    num_train_epochs=3,
```

```

per_device_train_batch_size=4,
learning_rate=5e-5, # 通常比预训练小1-2个数量级
warmup_steps=500,
logging_steps=100,
save_steps=1000,
)

# 4. 训练
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
)

trainer.train()

```

5.2.2 微调的超参数选择

超参数	预训练	微调	说明
学习率	1e-4 ~ 6e-4	2e-5 ~ 5e-5	微调时要小1-2个数量级
Batch Size	大 (256-4096)	小 (4-32)	数据量小, batch也小
Epochs	1 (大数据)	3-10	数据量小, 需要多轮
Warmup	5-10%	10-20%	比例更大
权重衰减	0.1	0.01-0.1	根据数据量调整

5.2.3 灾难性遗忘 (Catastrophic Forgetting)

问题：微调时，模型可能遗忘预训练学到的知识。



解决方案：

```

class RegularizedLoss(nn.Module):
    """
    加入正则化项，防止参数偏离预训练权重太远
    """
    def __init__(self, pretrained_model, alpha=0.01):
        super().__init__()
        self.pretrained_params = {

```

```

        name: param.clone().detach()
    for name, param in pretrained_model.named_parameters():
    }
    self.alpha = alpha

def forward(self, model, task_loss):
    # 任务损失
    total_loss = task_loss

    # L2正则化: 惩罚参数偏离预训练权重
    reg_loss = 0
    for name, param in model.named_parameters():
        if name in self.pretrained_params:
            reg_loss += torch.sum((param - self.pretrained_params[name]) ** 2)

    total_loss += self.alpha * reg_loss

    return total_loss

```

5.3 Parameter-Efficient Fine-Tuning (PEFT)

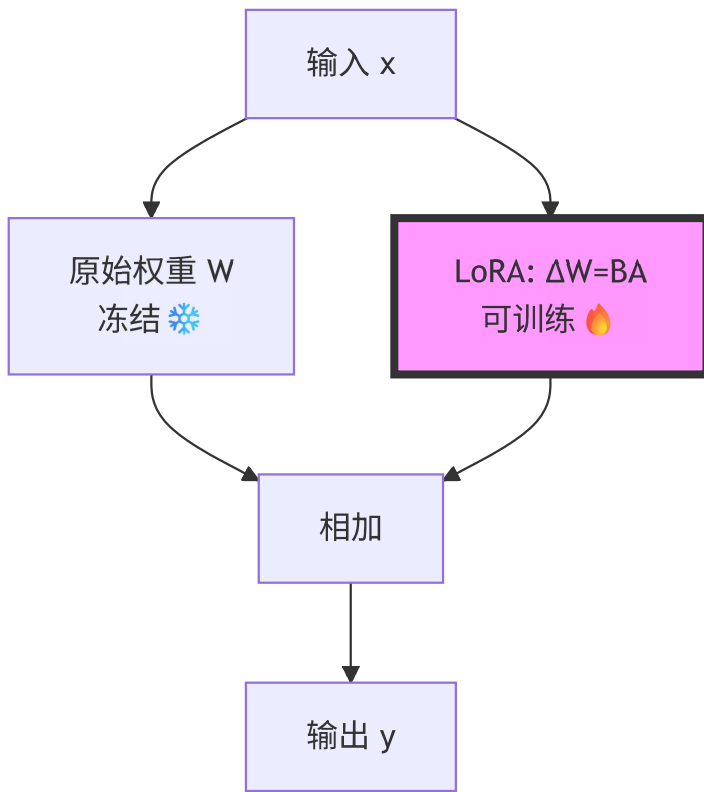
全参数微调的问题:

- ❌ 需要为每个任务存储完整模型
- ❌ GPU显存需求大
- ❌ 训练慢

PEFT目标: 只训练少量参数, 达到接近全参数微调的效果。

5.3.1 LoRA (Low-Rank Adaptation)

核心思想: 冻结原始权重, 添加低秩矩阵进行训练。



数学表达：

$$h = W_0x + \Delta Wx = W_0x + BAx$$

其中：

- $W_0 \in \mathbb{R}^{d \times k}$: 原始权重 (冻结)
- $B \in \mathbb{R}^{d \times r}$, $A \in \mathbb{R}^{r \times k}$: LoRA矩阵 (可训练)
- $r \ll \min(d, k)$: 秩 (通常4-64)

参数量对比：

原始参数: $d \times k = 4096 \times 4096 = 16,777,216$

LoRA参数 (r=8): $d \times r + r \times k = 4096 \times 8 + 8 \times 4096 = 65,536$

减少: 256倍!

代码实现：

```

class LoRALayer(nn.Module):
    def __init__(self, in_features, out_features, rank=8, alpha=16):
        """
        LoRA层

        Args:
            in_features: 输入维度
            out_features: 输出维度
            rank: LoRA的秩
  
```

```

        alpha: 缩放因子
    """
    super().__init__()

    self.rank = rank
    self.alpha = alpha
    self.scaling = alpha / rank

    # LoRA矩阵
    self.lora_A = nn.Parameter(torch.randn(in_features, rank))
    self.lora_B = nn.Parameter(torch.zeros(rank, out_features))

    # 初始化
    nn.init.kaiming_uniform_(self.lora_A, a=math.sqrt(5))

    def forward(self, x):
        #  $\Delta W = B @ A$ 
        return (x @ self.lora_A @ self.lora_B) * self.scaling

class LinearWithLoRA(nn.Module):
    def __init__(self, linear_layer, rank=8, alpha=16):
        super().__init__()

        # 冻结原始层
        self.linear = linear_layer
        for param in self.linear.parameters():
            param.requires_grad = False

        # 添加LoRA
        self.lora = LoRALayer(
            linear_layer.in_features,
            linear_layer.out_features,
            rank, alpha
        )

    def forward(self, x):
        # 原始输出 + LoRA输出
        return self.linear(x) + self.lora(x)

def apply_lora_to_model(model, rank=8, alpha=16, target_modules=None):
    """
    为模型添加LoRA

    Args:

```

```

        target_modules: 要应用LoRA的模块名列表，如 ["q_proj", "v_proj"]
    """
    if target_modules is None:
        target_modules = ["q_proj", "k_proj", "v_proj", "o_proj"]

    for name, module in model.named_modules():
        if any(target in name for target in target_modules):
            if isinstance(module, nn.Linear):
                # 替换为带LoRA的层
                parent_name = name.rsplit('.', 1)[0]
                child_name = name.rsplit('.', 1)[1]
                parent = model.get_submodule(parent_name)

                setattr(parent, child_name, LinearWithLoRA(module, rank, alpha))

    return model

```

使用LoRA微调:

```

from peft import LoraConfig, get_peft_model

# 1. 定义LoRA配置
lora_config = LoraConfig(
    r=8, # rank
    lora_alpha=16,
    target_modules=["q_proj", "v_proj"], # 对哪些层应用LoRA
    lora_dropout=0.1,
    bias="none",
    task_type="CAUSAL_LM"
)

# 2. 将LoRA应用到模型
model = AutoModelForCausalLM.from_pretrained("meta-llama/llama-2-7b-hf")
model = get_peft_model(model, lora_config)

# 3. 查看可训练参数
model.print_trainable_parameters()
# 输出: trainable params: 4,194,304 || all params: 6,742,609,920 || trainable%: 0.0

# 4. 训练（只训练LoRA参数）
trainer = Trainer(model=model, ...)
trainer.train()

```

5. 保存（只保存LoRA权重）

```
model.save_pretrained("./lora_weights") # 只有几MB!
```

LoRA的优势:

- ☒ 参数量少（通常<1%）
- ☒ 训练快，显存占用小
- ☒ 可以为不同任务保存不同LoRA，方便切换
- ☒ 推理时可合并到原始权重，无额外开销

5.3.2 QLoRA (Quantized LoRA)

创新： 结合量化和LoRA，用更少显存微调更大模型。

关键技术:

1. **4-bit NormalFloat**: 新的量化数据类型
2. **双量化**: 连量化常数也量化
3. **Paged Optimizers**: 利用统一内存

```
from transformers import BitsAndBytesConfig

# QLoRA配置
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4", # NormalFloat 4bit
    bnb_4bit_compute_dtype=torch.bfloat16,
    bnb_4bit_use_double_quant=True, # 双量化
)

# 加载量化模型
model = AutoModelForCausalLM.from_pretrained(
    "meta-llama/Llama-2-70b-hf",
    quantization_config=bnb_config,
    device_map="auto"
)

# 应用LoRA
model = get_peft_model(model, lora_config)
```

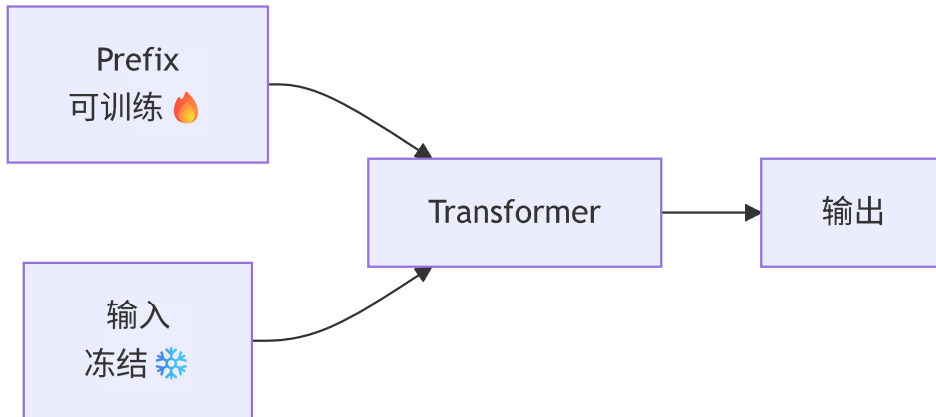
显存对比:

模型	Full FT	LoRA	QLoRA
LLaMA-7B	~28GB	~12GB	~6GB
LLaMA-13B	~52GB	~20GB	~10GB

模型	Full FT	LoRA	QLoRA
LLaMA-65B	~260GB	~120GB	~48GB

5.3.3 Prefix Tuning

思想： 在输入前添加可训练的"prefix"向量。



```

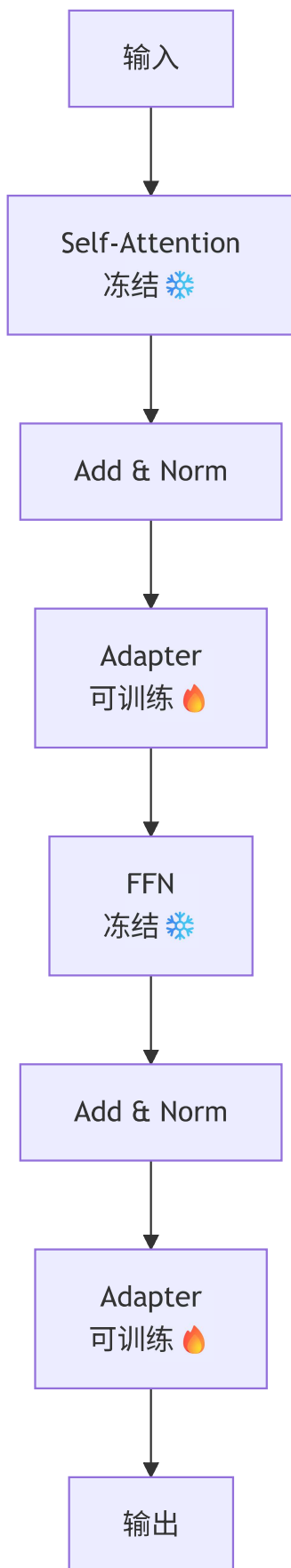
class PrefixTuning(nn.Module):
    def __init__(self, num_layers, num_heads, d_model, prefix_len=20):
        super().__init__()

        # 为每一层创建prefix
        self.prefix_len = prefix_len
        self.prefix = nn.Parameter(
            torch.randn(num_layers, 2, num_heads, prefix_len, d_model // num_heads)
        )
        # shape: (layers, 2[key+value], heads, prefix_len, d_k)

    def forward(self, layer_idx):
        """
        返回第layer_idx层的prefix key和value
        """
        return self.prefix[layer_idx, 0], self.prefix[layer_idx, 1]
  
```

5.3.4 Adapter

思想： 在Transformer层中插入小的adapter模块。



```
class Adapter(nn.Module):  
    def __init__(self, d_model, bottleneck_dim=64):  
        super().__init__()  
  
        self.down_proj = nn.Linear(d_model, bottleneck_dim)
```

```

self.activation = nn.ReLU()
self.up_proj = nn.Linear(bottleneck_dim, d_model)

# 残差连接

def forward(self, x):
    residual = x

    x = self.down_proj(x)
    x = self.activation(x)
    x = self.up_proj(x)

    return x + residual # 残差连接

```

5.3.5 PEFT方法对比

方法	可训练参数数	训练速度	推理开销	效果	适用场景
Full FT	100%	慢	无	最好	数据充足、资源充足
LoRA	<1%	快	无（可合并）	接近Full FT	推荐首选
QLoRA	<1%	快	无	略低于LoRA	显存受限
Prefix Tuning	~0.1%	很快	有（增加序列长度）	中等	极度资源受限
Adapter	~3%	快	有（额外计算）	良好	需要切换多任务

5.4 Instruction Tuning（指令微调）

5.4.1 什么是Instruction Tuning?

目标： 让模型学会遵循各种格式的指令。

传统微调：

输入： "The movie was great."
输出： "Positive"

Instruction Tuning:

输入： "Classify the sentiment: The movie was great."
输出： "Positive"

输入： "判断情感：这部电影真棒。"
输出： "正面"

输入: "Is this positive or negative? The movie was great."

输出: "Positive"

5.4.2 指令数据格式

标准格式:

```
{
  "instruction": "将以下英文翻译成中文",
  "input": "Hello, how are you?",
  "output": "你好, 你好吗? "
}

{
  "instruction": "回答以下问题",
  "input": "地球上最高的山是什么? ",
  "output": "珠穆朗玛峰, 海拔8848.86米。"
}

{
  "instruction": "根据给定的主题写一首诗",
  "input": "主题: 春天",
  "output": "春风拂面花盛开, \n燕子归来绕青苔。 \n..."
}
```

提示模板 (Prompt Template) :

```
def format_instruction_data(example):
    """
    格式化指令数据
    """
    if example['input']:
        prompt = f"""Below is an instruction that describes a task, paired with an

### Instruction:
{example['instruction']}

### Input:
{example['input']}

### Response:
{example['output']}"""
    else:
        prompt = f"""Below is an instruction that describes a task. Write a response
```

```
### Instruction:
{example['instruction']}

### Response:
{example['output']}"""

return prompt
```

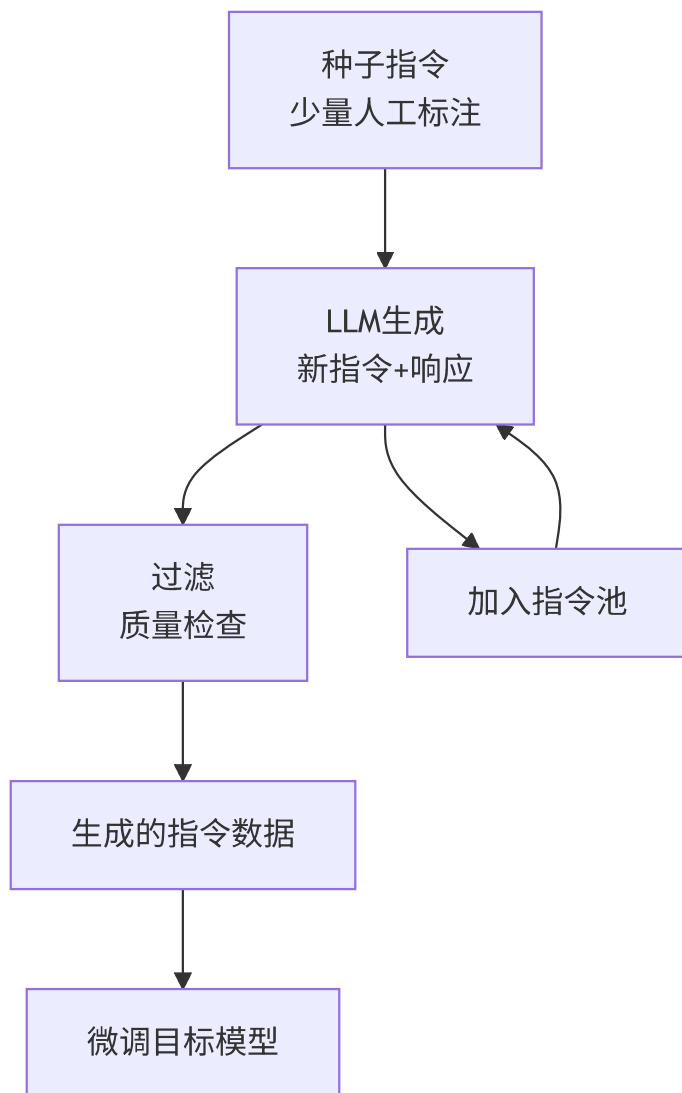
5.4.3 著名的指令数据集

数据集	规模	语言	特点
FLAN	1800+ tasks	英文	Google, 涵盖62个任务类型
P3	170+ datasets	英文	基于PromptSource
Alpaca	52K	英文	斯坦福, Self-Instruct生成
Belle	2M	中文	链家开源, 中文指令
COIG	多个子集	中文	中文开源指令集合
ShareGPT	~90K	多语言	用户与ChatGPT的真实对话

5.4.4 Self-Instruct

问题： 人工标注指令数据成本高。

Self-Instruct方法： 使用强大的LLM自动生成指令数据。



代码示例:

```
def generate_instructions(seed_instructions, model, num_to_generate=52000):  
    """  
    Self-Instruct生成指令数据  
    """  
    generated_data = []  
    instruction_pool = seed_instructions.copy()  
  
    for i in range(num_to_generate):  
        # 1. 随机采样示例  
        examples = random.sample(instruction_pool, k=3)  
  
        # 2. 构造prompt让模型生成新指令  
        prompt = f"""Come up with a series of tasks:  
  
Examples:  
{format_examples(examples)}  
  
Generate a new task:
```

Task: ""

3. 生成新指令

new_instruction = model.generate(prompt)

4. 生成该指令的输入输出

prompt_io = f""Instruction: {new_instruction}

Input: ""

new_input = model.generate(prompt_io)

prompt_output = f""Instruction: {new_instruction}

Input: {new_input}

Output: ""

new_output = model.generate(prompt_output)

5. 质量检查

if is_high_quality(new_instruction, new_input, new_output):

```
    data = {
        "instruction": new_instruction,
        "input": new_input,
        "output": new_output
    }
```

generated_data.append(data)

instruction_pool.append(new_instruction)

return generated_data

def is_high_quality(instruction, input_text, output):

"""

质量检查

"""

检查1: 长度合理

if len(output) < 10 or len(output) > 2000:

return False

检查2: 不是重复的指令

...

检查3: 语法正确性

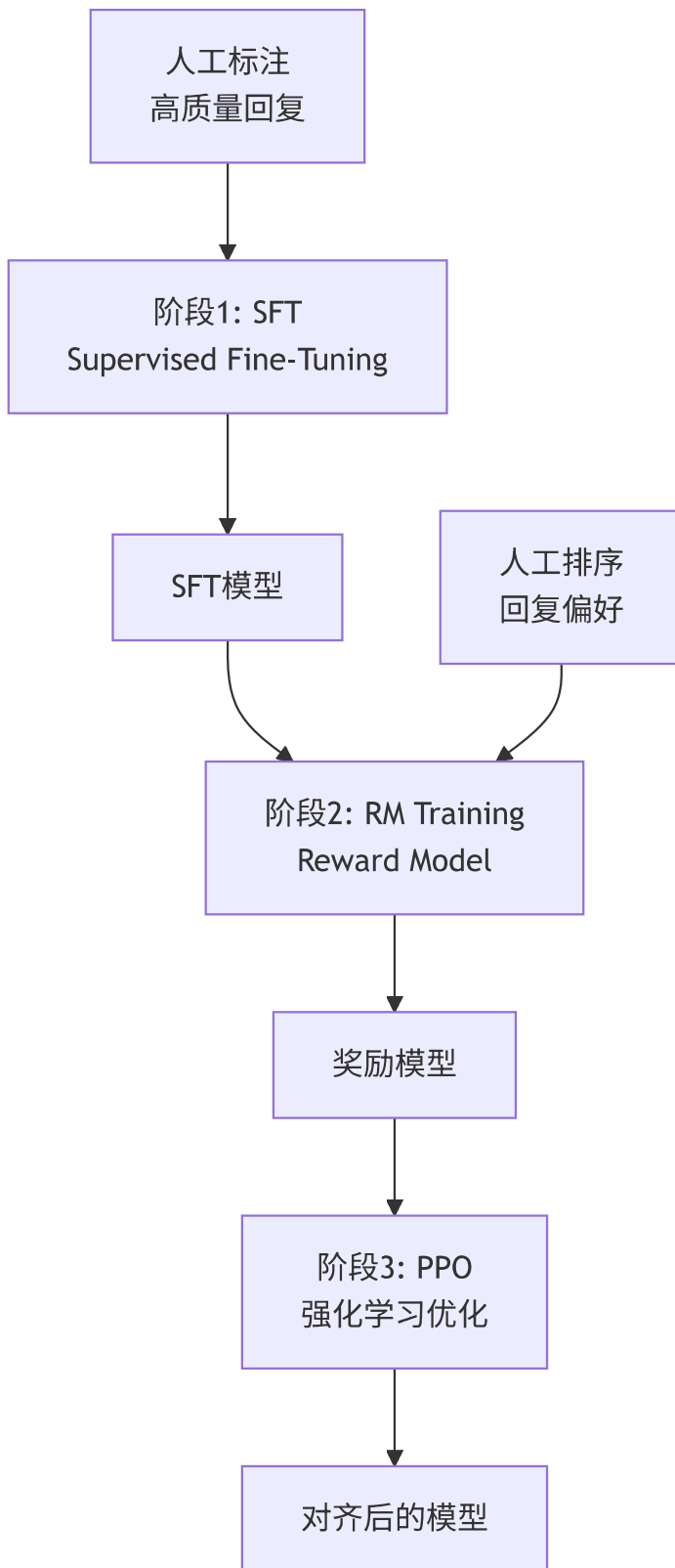
...

return True

5.5 RLHF (Reinforcement Learning from Human Feedback)

RLHF是ChatGPT成功的关键，让模型对齐人类价值观。

5.5.1 RLHF三阶段流程



5.5.2 阶段1: Supervised Fine-Tuning

数据收集:

```
{  
  "prompt": "如何学习编程？",
```

```
"response": "学习编程可以从以下几个步骤开始：\n1. 选择一门编程语言..."
}
```

训练：标准的监督学习，最大化 $P(response|prompt)$ 。

```
def train_sft(model, dataset):
    """
    SFT训练
    """
    for batch in dataset:
        prompts = batch['prompts']
        responses = batch['responses']

        # 拼接成完整输入
        inputs = [p + r for p, r in zip(prompts, responses)]

        # 标准语言模型训练
        loss = model(inputs, labels=inputs)
        loss.backward()
        optimizer.step()
```

InstructGPT的SFT数据：

- 约13,000条高质量对话
- 由40名标注员编写

5.5.3 阶段2: Reward Model Training

目标：训练一个模型预测人类偏好。

数据收集：

给定prompt，模型生成4-9个回复，人工排序：

Prompt: "解释量子计算"

回复A: "量子计算是一种..." ★★★★★

回复B: "量子..." ★★★★★

回复C: "不知道" ★

回复D: "量子计算利用..." ★★★★★★

排序：D > A > B > C

Reward Model架构：


```

class RewardModel(nn.Module):
    def __init__(self, base_model):
        super().__init__()

        # 使用SFT模型作为基础
        self.base_model = base_model

        # 移除Language modeling head, 添加reward head
        self.reward_head = nn.Linear(base_model.config.hidden_size, 1)

    def forward(self, input_ids, attention_mask):
        # 获取最后一个token的表示
        outputs = self.base_model(input_ids, attention_mask=attention_mask)
        last_hidden = outputs.last_hidden_state[:, -1, :]

        # 预测reward
        reward = self.reward_head(last_hidden)

        return reward

def train_reward_model(model, comparison_dataset):
    """
    训练奖励模型
    """
    for batch in comparison_dataset:
        prompts = batch['prompts']
        chosen = batch['chosen'] # 更好的回复
        rejected = batch['rejected'] # 更差的回复

        # 计算两个回复的reward
        reward_chosen = model(prompts + chosen)
        reward_rejected = model(prompts + rejected)

        # 损失: chosen的reward应该更高
        # 使用对比损失
        loss = -torch.log(torch.sigmoid(reward_chosen - reward_rejected))
        loss = loss.mean()

        loss.backward()
        optimizer.step()

```

InstructGPT的RM数据:

- 约33,000条比较数据
- 每个prompt有多个回复的排序

5.5.4 阶段3: PPO Optimization

目标： 使用强化学习，让模型生成高reward的回复。

目标函数：

$$\mathcal{L}^{RLHF}(\theta) = \mathbb{E}_{x \sim D, y \sim \pi_{\theta}}[r_{\phi}(x, y)] - \beta \mathbb{D}_{KL}[\pi_{\theta}(y|x) || \pi_{ref}(y|x)]$$

其中：

- $r_{\phi}(x, y)$: Reward Model的奖励
- \mathbb{D}_{KL} : KL散度，防止偏离SFT模型太远
- β : KL惩罚系数

PPO训练流程：

```
def train_ppo(policy_model, reward_model, ref_model, prompts,
              beta=0.1, clip_epsilon=0.2):
    """
    PPO训练循环

    Args:
        policy_model: 正在训练的策略模型
        reward_model: 冻结的奖励模型
        ref_model: 冻结的参考模型（SFT模型）
        beta: KL惩罚系数
    """
    for batch in prompts:
        # 1. 生成回复
        with torch.no_grad():
            responses, log_probs_old = policy_model.generate(
                batch,
                return_log_probs=True
            )

        # 2. 计算奖励
        with torch.no_grad():
            rewards = reward_model(batch, responses)

        # KL惩罚
        log_probs_ref = ref_model.get_log_probs(batch, responses)
        log_probs_policy = policy_model.get_log_probs(batch, responses)
        kl_penalty = beta * (log_probs_policy - log_probs_ref)

        # 总奖励
        rewards = rewards - kl_penalty

    # 3. PPO更新（多个epochs）
```

```

for ppo_epoch in range(4):
    # 重新计算log probs
    log_probs_new = policy_model.get_log_probs(batch, responses)

    # Importance sampling ratio
    ratio = torch.exp(log_probs_new - log_probs_old)

    # PPO clipped objective
    advantages = rewards # 简化版，实际需要计算advantage

    surr1 = ratio * advantages
    surr2 = torch.clamp(ratio, 1.0 - clip_epsilon, 1.0 + clip_epsilon) * ac

    policy_loss = -torch.min(surr1, surr2).mean()

    # 反向传播
    policy_loss.backward()
    optimizer.step()
    optimizer.zero_grad()

```

为什么需要KL惩罚？

没有KL惩罚：

模型："这是最好的!" (reward: 10, 但语无伦次)

有KL惩罚：

模型："这是一个很好的选择，因为..." (reward: 8, 但合理)

KL惩罚确保模型不会为了追求高reward而生成不自然的文本。

5.6 DPO (Direct Preference Optimization)

问题： RLHF太复杂（需要训练RM、PPO不稳定）

DPO创新： 直接优化偏好，无需单独的RM和RL。

5.6.1 DPO原理

核心insight： 将RM和PPO合并成一个损失函数。

DPO损失函数：

$$\mathcal{L}_{DPO}(\theta) = -\mathbb{E}_{(x, y_w, y_l) \sim D} \left[\log \sigma \left(\beta \log \frac{\pi_{\theta}(y_w|x)}{\pi_{ref}(y_w|x)} - \beta \log \frac{\pi_{\theta}(y_l|x)}{\pi_{ref}(y_l|x)} \right) \right]$$

其中：

- y_w ：preferred response (更好的回复)

- y_l : dispreferred response (更差的回复)

直观理解:

目标:

- 增加preferred response的概率
- 降低dispreferred response的概率
- 不偏离reference model太远

5.6.2 DPO实现

```
def dpo_loss(policy_model, ref_model, batch, beta=0.1):
    """
    DPO损失函数

    Args:
        batch: {
            'prompts': prompts,
            'chosen': preferred responses,
            'rejected': dispreferred responses
        }
    """
    prompts = batch['prompts']
    chosen = batch['chosen']
    rejected = batch['rejected']

    # 1. 计算policy model的log probs
    log_probs_chosen = policy_model.get_log_probs(prompts, chosen)
    log_probs_rejected = policy_model.get_log_probs(prompts, rejected)

    # 2. 计算reference model的log probs
    with torch.no_grad():
        log_probs_chosen_ref = ref_model.get_log_probs(prompts, chosen)
        log_probs_rejected_ref = ref_model.get_log_probs(prompts, rejected)

    # 3. 计算Log ratios
    log_ratio_chosen = log_probs_chosen - log_probs_chosen_ref
    log_ratio_rejected = log_probs_rejected - log_probs_rejected_ref

    # 4. DPO损失
    logits = beta * (log_ratio_chosen - log_ratio_rejected)
    loss = -F.logsigmoid(logits).mean()

    return loss
```

```
def train_dpo(model, ref_model, dataset, beta=0.1):
    """
    DPO训练
    """
    for batch in dataset:
        loss = dpo_loss(model, ref_model, batch, beta)

        loss.backward()
        optimizer.step()
        optimizer.zero_grad()
```

5.6.3 RLHF vs DPO

维度	RLHF	DPO
复杂度	高（三阶段）	低（单阶段）
训练稳定性	PPO不稳定	更稳定
计算资源	大（需要4个模型）	小（2个模型）
效果	经过验证（ChatGPT）	接近RLHF
实现难度	难	简单
适用场景	有充足资源	资源受限、快速迭代

模型数量对比：

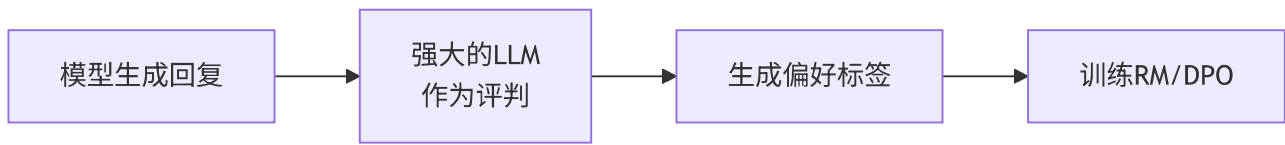
- RLHF需要：
- 1. Policy Model（训练）
 - 2. Reference Model（冻结）
 - 3. Reward Model（冻结）
 - 4. Value Model（训练，可选）

- DPO只需要：
- 1. Policy Model（训练）
 - 2. Reference Model（冻结）

5.7 其他对齐方法

5.7.1 RLAIIF (RL from AI Feedback)

用AI代替人类提供反馈。



优势:

- ☒ 成本低 (无需人工标注)
- ☒ 可扩展
- ☒ 可能继承AI的偏见

5.7.2 ORPO (Odds Ratio Preference Optimization)

进一步简化, 将SFT和偏好优化合并。

```
def orpo_loss(model, batch):  
    """  
    ORPO: SFT + Preference Optimization in one step  
    """  
    # SFT损失 (在chosen上)  
    sft_loss = -model.get_log_probs(batch['prompts'], batch['chosen']).mean()  
  
    # Preference损失  
    log_odds_chosen = model.get_log_odds(batch['prompts'], batch['chosen'])  
    log_odds_rejected = model.get_log_odds(batch['prompts'], batch['rejected'])  
  
    preference_loss = -F.logsigmoid(log_odds_chosen - log_odds_rejected).mean()  
  
    # 总损失  
    return sft_loss + preference_loss
```

5.7.3 Constitutional AI

Anthropic提出的方法, 使用"宪法" (规则集) 指导对齐。

步骤:

1. 模型生成可能有害的回复
2. 模型自我批评 (根据宪法)
3. 模型生成改进的回复
4. 使用改进的数据训练

```
constitutional_principles = [  
    "Please choose the response that is most helpful, honest, and harmless.",  
    "Please choose the response that is least racist or sexist.",  
    "Please choose the response that sounds most similar to what a peaceful, ethical  
]
```

```
def constitutional_ai_critique(model, response, principle):
    prompt = f"""Response: {response}

    Critique the response according to this principle: {principle}

    Critique: """

    critique = model.generate(prompt)

    revision_prompt = f"""Response: {response}

    Critique: {critique}

    Revision that addresses the critique: """

    improved_response = model.generate(revision_prompt)

    return improved_response
```

5.8 面试高频问题

Q1: LoRA为什么有效?

答案要点:

- 本质原因：** 微调的权重更新是低秩的
 - 研究发现， ΔW 的固有秩(intrinsic rank)很低
 - 可以用低秩矩阵近似

2. 数学解释：

$$W_{new} = W_0 + \Delta W \approx W_0 + BA$$

其中 $B \in \mathbb{R}^{d \times r}$, $A \in \mathbb{R}^{r \times k}$, $r \ll \min(d, k)$

3. 经验证据：

- rank=8时已能达到95%+的全参数微调效果
- rank=16几乎等价于全参数微调

Q2: RLHF的每个阶段为什么必要?

阶段	作用	如果跳过会怎样
SFT	学习任务格式、基础能力	RM和PPO无法收敛（起点太差）
RM	学习人类偏好	无法量化"好坏"（PPO无奖励信号）




阶段	作用	如果跳过会怎样
PPO	根据偏好优化生成	无法生成符合偏好的回复

完整流程的意义：




Base Model (知识丰富但不听话)
↓ SFT
SFT Model (懂格式但质量不稳定)
↓ RM Training
RM (能判断好坏)
↓ PPO
Aligned Model (听话、有用、无害)

Q3: DPO相比RLHF的优势和劣势？

优势：

-  简单：单阶段训练
-  稳定：无PPO的不稳定性
-  高效：减少50%计算资源

劣势：

-  灵活性：难以融入复杂的奖励设计
-  在线学习：无法像PPO那样在线采样
-  经验较少：RLHF在工业界验证更充分

选择建议：

- 资源充足、追求极致性能 → RLHF
- 快速迭代、资源受限 → **DPO**

Q4: 如何评估对齐的效果？

自动评估：

- Reward Model score
- GPT-4打分
- 各种benchmark (MMLU, TruthfulQA等)

人工评估：

维度：

1. Helpfulness (有用性)：回答是否解决问题
2. Harmlessness (无害性)：是否包含有害内容
3. Honesty (诚实性)：是否准确、不编造事实

A/B测试:

呈现两个模型的回复，人工选择更好的：

Model A: ...

Model B: ...

选择: [] A [] B [] 相当

Q5: 如何防止过度对齐 (Over-alignment) ?

问题: 过度对齐可能让模型过于谨慎，拒绝合理请求。

Over-aligned模型:

用户: "讲个笑话"

模型: "作为AI，我不应该..." ❌

用户: "帮我写个病毒程序"

模型: "这是不道德的，我不能..." ✅ (应该拒绝)

解决方案:

1. **平衡数据**: 包含积极和拒绝的示例
2. **多样化评估**: 避免单一维度优化
3. **Red Teaming**: 主动测试边界情况
4. **控制KL惩罚**: 不要偏离base model太远

5.9 本章小结

本章详细讲解了微调与对齐技术:

✅ **PEFT方法**: LoRA是最实用的选择, QLoRA适合显存受限 ✅ **Instruction Tuning**: 让模型学会遵循指令 ✅ **RLHF**: ChatGPT的成功秘诀, 但复杂度高 ✅ **DPO**: 简化的对齐方法, 效果接近RLHF ✅ **对齐评估**: HHH原则 (Helpful, Harmless, Honest)

实践建议:

1. 先尝试LoRA微调 (性价比最高)
2. 有偏好数据时, 优先考虑DPO
3. 资源充足且追求极致性能时, 使用RLHF
4. 持续评估对齐效果, 避免过度对齐

下一章预告: 第6章将讲解高效训练与推理技术, 包括量化、剪枝、Flash Attention等。