

大语言模型先进技术与评估方法

LoRA · QLoRA · vLLM · Flash Attention · BLEU · ROUGE

参数高效微调、推理加速与评估方法全解析

AI/ML 系列教程

大语言模型先进技术与评估方法

2026 年 1 月 6 日

目录

1	引言	3
I	第一部分：大语言模型先进技术	3
2	参数高效微调技术	3
2.1	LoRA (Low-Rank Adaptation)	4
2.2	QLoRA (Quantized LoRA)	9
2.3	PEFT 框架	11
2.4	LoRA 变体方法	13
2.5	参数效率对比	13
3	监督微调技术	13
3.1	SFT (Supervised Fine-Tuning)	13
3.2	指令微调 (Instruction Tuning)	16
3.3	对话微调 (Chat Fine-Tuning)	17
4	推理加速技术	18
4.1	vLLM (Very Large Language Model)	18
4.2	Flash Attention	19
4.3	量化推理	20
4.4	模型并行与张量并行	21
5	模型部署技术	21

5.1	模型压缩	21
5.2	服务化部署	21
5.3	边缘部署	22
6	推理优化技术	22
6.1	KV Cache	22
6.2	连续批处理 (Continuous Batching)	22
6.3	动态批处理	23
7	总结	23
II	第二部分：评估方法与评测基准	24
8	评估指标与方法	24
8.1	困惑度 (Perplexity)	24
8.2	BLEU 分数	28
8.3	ROUGE 分数	34
8.4	METEOR 分数	38
8.5	BERTScore	38
9	评测基准与数据集	39
9.1	GLUE 和 SuperGLUE	39
9.2	MMLU	39
9.3	HumanEval 和 MBPP	39
9.4	MT-Bench 和 AlpacaEval	39
10	QA Pair (问答对)	39
10.1	LoRA 相关问答	39
10.2	评估指标相关问答	40
11	综合练习	41
11.1	概念理解题	41
11.2	计算题	41
11.3	代码实现题	42

11.4 案例分析题	42
------------------	----

1 引言

大语言模型 (Large Language Model, LLM) 已成为人工智能领域最活跃的研究方向之一。从 GPT 系列到 LLaMA, 从 BERT 到 T5, 大语言模型在自然语言处理、代码生成、多模态理解等任务上取得了突破性进展。然而, 大模型的训练、微调、部署和评估仍面临诸多挑战。

本文档涵盖的核心内容:

- 参数高效微调技术: LoRA、QLoRA、PEFT 等, 大幅降低微调成本
- 监督微调技术: SFT、指令微调、对话微调等, 提升模型能力
- 推理加速技术: vLLM、Flash Attention、量化推理等, 提高推理效率
- 模型部署技术: 模型压缩、服务化部署、边缘部署等
- 推理优化技术: KV Cache、连续批处理、动态批处理等
- 评估指标与方法: 困惑度、BLEU、ROUGE、METEOR、BERTScore 等
- 评测基准与数据集: MMLU、HumanEval、MT-Bench 等

文档结构:

1. 第一部分: 先进技术 (参数高效微调、监督微调、推理加速、部署技术等)
2. 第二部分: 评估方法与评测基准、QA Pair、综合练习

Part I

第一部分: 大语言模型先进技术

2 参数高效微调技术

传统的全参数微调需要更新模型的所有参数, 对于大模型来说成本极高。参数高效微调 (Parameter-Efficient Fine-Tuning, PEFT) 技术通过只更新少量参数就能达到接近全参数微调的效果, 大幅降低了微调成本。

2.1 LoRA (Low-Rank Adaptation)

概念解释: LoRA 是 Microsoft 在 2021 年提出的参数高效微调方法。其核心思想是：对于预训练模型的权重矩阵 \mathbf{W} ，不直接更新它，而是学习一个低秩分解的增量 $\Delta\mathbf{W}$ ，使得 $\mathbf{W} + \Delta\mathbf{W}$ 能够适应新任务。

数学公式:

对于原始权重矩阵 $\mathbf{W} \in \mathbb{R}^{d \times k}$ ，LoRA 将其更新分解为：

$$\mathbf{W}' = \mathbf{W} + \Delta\mathbf{W} = \mathbf{W} + \mathbf{BA} \quad (1)$$

其中：

- $\mathbf{A} \in \mathbb{R}^{r \times k}$ ：低秩矩阵，随机初始化
- $\mathbf{B} \in \mathbb{R}^{d \times r}$ ：低秩矩阵，初始化为零
- $r \ll \min(d, k)$ ：秩 (rank)，通常 $r \in \{4, 8, 16, 32, 64\}$

在前向传播时：

$$\mathbf{h} = \mathbf{W}'\mathbf{x} = (\mathbf{W} + \mathbf{BA})\mathbf{x} = \mathbf{W}\mathbf{x} + \mathbf{B}(\mathbf{A}\mathbf{x}) \quad (2)$$

参数量对比:

- 全参数微调： $d \times k$ 个参数
- LoRA： $r \times (d + k)$ 个参数
- 参数减少比例： $\frac{r(d+k)}{dk} = r \left(\frac{1}{k} + \frac{1}{d} \right)$

当 $r = 8$ ， $d = 4096$ ， $k = 4096$ 时，参数量从 16,777,216 减少到 65,536，减少了约 256 倍。

算法原理:

代码实现:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class LoRALayer(nn.Module):
    """LoRA 层实现"""
```

Algorithm 1 LoRA 微调算法**Require:** 预训练模型权重 \mathbf{W} , 训练数据 \mathcal{D} , 秩 r , 学习率 η **Ensure:** LoRA 权重 \mathbf{A} 和 \mathbf{B}

```

1: 初始化  $\mathbf{A}$  为随机小值,  $\mathbf{B}$  为零矩阵
2: 冻结原始权重  $\mathbf{W}$ 
3: repeat
4:   for 每个批次  $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}$  do
5:     前向传播:  $\mathbf{h} = \mathbf{W}\mathbf{x} + \mathbf{B}(\mathbf{A}\mathbf{x})$ 
6:     计算损失:  $\mathcal{L} = \text{loss}(\mathbf{h}, \mathbf{y})$ 
7:     反向传播, 只更新  $\mathbf{A}$  和  $\mathbf{B}$ 
8:      $\mathbf{A} \leftarrow \mathbf{A} - \eta \nabla_{\mathbf{A}} \mathcal{L}$ 
9:      $\mathbf{B} \leftarrow \mathbf{B} - \eta \nabla_{\mathbf{B}} \mathcal{L}$ 
10:   end for
11: until 收敛

```

```

def __init__(self, in_features, out_features, rank=8, alpha=16,
             dropout=0.0):
    """
    参数:
        in_features: 输入特征维度
        out_features: 输出特征维度
        rank: LoRA 的秩
        alpha: 缩放因子, 通常等于 rank
        dropout: Dropout 概率
    """
    super().__init__()
    self.rank = rank
    self.alpha = alpha
    self.scaling = alpha / rank

    # LoRA 矩阵 A 和 B
    self.lora_A = nn.Parameter(torch.randn(rank, in_features) * 0.02)
    self.lora_B = nn.Parameter(torch.zeros(out_features, rank))

    # Dropout
    self.dropout = nn.Dropout(dropout) if dropout > 0 else nn.
Identity()

    # 原始权重 (冻结)

```

```
self.weight = None # 将在外部设置

def forward(self, x, weight):
    """
    前向传播

    参数:
        x: 输入张量 (batch_size, ..., in_features)
        weight: 原始权重矩阵 (out_features, in_features)
    """
    # 原始输出
    original_output = F.linear(x, weight)

    # LoRA 输出:  $B @ (A @ x)$ 
    x_dropout = self.dropout(x)
    lora_output = F.linear(
        F.linear(x_dropout, self.lora_A),
        self.lora_B
    )

    # 缩放并相加
    return original_output + self.scaling * lora_output

class LoRALinear(nn.Module):
    """包装的 LoRA 线性层"""

    def __init__(self, linear_layer, rank=8, alpha=16, dropout=0.0):
        super().__init__()
        self.original = linear_layer
        self.lora = LoRALayer(
            linear_layer.in_features,
            linear_layer.out_features,
            rank,
            alpha,
            dropout
        )
        # 冻结原始权重
        for param in self.original.parameters():
            param.requires_grad = False
```

```
def forward(self, x):
    return self.lora(x, self.original.weight)

# 使用示例
# 假设我们有一个预训练的线性层
pretrained_linear = nn.Linear(768, 3072)

# 包装为 LoRA 层
lora_linear = LoRALinear(pretrained_linear, rank=8, alpha=16)

# 前向传播
x = torch.randn(32, 768) # batch_size=32
output = lora_linear(x)
print(f"输入形状: {x.shape}")
print(f"输出形状: {output.shape}")

# 检查可训练参数
total_params = sum(p.numel() for p in lora_linear.parameters() if p.requires_grad)
print(f"可训练参数数量: {total_params}") # 8 * (768 + 3072) = 30720
print(f"原始参数数量: {pretrained_linear.weight.numel()}") # 768 * 3072 = 2359296
print(f"参数减少比例: {pretrained_linear.weight.numel() / total_params:.2f}%")
```

Listing 1: LoRA 从零实现

使用 PEFT 库实现:

```
from transformers import AutoModelForCausalLM, AutoTokenizer
from peft import LoraConfig, get_peft_model, TaskType

# 加载预训练模型
model_name = "meta-llama/Llama-2-7b-hf" # 示例模型
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    torch_dtype=torch.float16,
    device_map="auto"
)
tokenizer = AutoTokenizer.from_pretrained(model_name)
```

```
# 配置 LoRA
lora_config = LoraConfig(
    task_type=TaskType.CAUSAL_LM,
    r=8, # rank
    lora_alpha=16, # alpha
    lora_dropout=0.1,
    target_modules=["q_proj", "v_proj", "k_proj", "o_proj"], # 目标模块
    bias="none"
)

# 应用 LoRA
model = get_peft_model(model, lora_config)

# 打印可训练参数
model.print_trainable_parameters()
# trainable params: 4,194,304 || all params: 6,738,415,616 || trainable
# \%: 0.06

# 训练（只更新 LoRA 参数）
# ... 训练代码 ...
```

Listing 2: 使用 Hugging Face PEFT 库

适用场景：

- 资源受限环境下的模型微调
- 需要快速适应多个任务的场景
- 模型服务化部署前的快速迭代

优势与局限：

优势：

- 参数量大幅减少（通常减少 100-1000 倍）
- 训练速度快，内存占用低
- 可以保存多个 LoRA 适配器，快速切换任务
- 效果接近全参数微调

局限：

- 在某些复杂任务上可能不如全参数微调
- 需要选择合适的 rank 和目标模块
- 多个 LoRA 适配器组合时可能产生冲突

2.2 QLoRA (Quantized LoRA)

概念解释：QLoRA 是 LoRA 的量化版本，通过 4-bit 量化进一步降低内存占用，使得在消费级 GPU 上微调大模型成为可能。

量化原理：

QLoRA 使用 4-bit NormalFloat (NF4) 量化，将 32-bit 浮点数映射到 4-bit 整数：

$$Q(x) = \text{round} \left(\frac{x - \text{offset}}{\text{scale}} \right) \times \text{scale} + \text{offset} \quad (3)$$

其中：

- offset：量化偏移量
- scale：量化缩放因子
- $Q(x)$ ：量化后的值

与 LoRA 的区别：

- LoRA：原始权重保持 FP16/BF16，只量化激活值
- QLoRA：原始权重量化为 4-bit，激活值量化为 8-bit，LoRA 权重保持 16-bit
- 内存节省：QLoRA 可以节省约 75% 的内存

数学表达式：

对于量化权重 \mathbf{W}_Q 和 LoRA 增量：

$$\mathbf{W}' = \mathbf{W}_Q + \frac{\alpha}{r} \mathbf{B} \mathbf{A} \quad (4)$$

其中 \mathbf{W}_Q 是量化后的权重，在推理时动态反量化。

代码实现：

```
from transformers import AutoModelForCausalLM, AutoTokenizer,
    BitsAndBytesConfig
from peft import LoraConfig, get_peft_model,
    prepare_model_for_kbit_training
import torch

# 4-bit 量化配置
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,
    bnb_4bit_quant_type="nf4",
    bnb_4bit_compute_dtype=torch.float16,
    bnb_4bit_use_double_quant=True, # 嵌套量化
)

# 加载模型（自动量化）
model_name = "meta-llama/Llama-2-7b-hf"
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    quantization_config=bnb_config,
    device_map="auto"
)
tokenizer = AutoTokenizer.from_pretrained(model_name)

# 准备模型进行 k-bit 训练
model = prepare_model_for_kbit_training(model)

# LoRA 配置
lora_config = LoraConfig(
    r=8,
    lora_alpha=16,
    target_modules=["q_proj", "v_proj"],
    lora_dropout=0.1,
    bias="none",
    task_type="CAUSAL_LM"
)

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

```
# 打印内存使用
model.print_trainable_parameters()
# trainable params: 4,194,304 || all params: 6,738,415,616 || trainable
  \%: 0.06

# 内存占用对比:
# FP16 LoRA: ~14GB
# QLoRA: ~6GB (节省约 57\%)
```

Listing 3: 使用 bitsandbytes 和 PEFT 实现 QLoRA

内存优化效果:

对于 7B 参数的模型:

- 全参数微调 (FP16): 约 28GB
- LoRA (FP16): 约 14GB
- QLoRA (4-bit): 约 6GB

优势与局限:

优势:

- 内存占用极低, 可在消费级 GPU 上运行
- 训练速度与 LoRA 相当
- 效果损失很小 (通常 < 1%)

局限:

- 量化可能引入轻微的性能损失
- 需要支持 4-bit 量化的硬件
- 某些操作可能不支持量化

2.3 PEFT 框架

概念解释: PEFT (Parameter-Efficient Fine-Tuning) 是 Hugging Face 提供的统一框架, 支持多种参数高效微调方法。

支持的方法:

- LoRA
- Prefix Tuning
- P-Tuning v2
- Prompt Tuning
- AdaLoRA
- 自定义方法

统一接口:

```
from peft import (
    LoraConfig,
    PrefixTuningConfig,
    PromptTuningConfig,
    get_peft_model
)

# LoRA 配置
lora_config = LoraConfig(
    r=8,
    lora_alpha=16,
    target_modules=["q_proj", "v_proj"]
)

# Prefix Tuning 配置
prefix_config = PrefixTuningConfig(
    task_type="CAUSAL_LM",
    num_virtual_tokens=20
)

# Prompt Tuning 配置
prompt_config = PromptTuningConfig(
    task_type="CAUSAL_LM",
    num_virtual_tokens=20
)

# 统一接口应用
model = get_peft_model(model, lora_config) # 或 prefix_config,
    prompt_config
```

Listing 4: PEFT 框架使用示例

2.4 LoRA 变体方法

AdaLoRA: 自适应 LoRA，动态调整 rank。

数学公式:

$$\mathbf{W}' = \mathbf{W} + \sum_{i=1}^r s_i \mathbf{b}_i \mathbf{a}_i^T \quad (5)$$

其中 s_i 是重要性分数，用于剪枝不重要的 LoRA 模块。

DoRA (Weight-Decomposed Low-Rank Adaptation): 将权重分解为幅度和方向。

数学公式:

$$\mathbf{W}' = \frac{m}{\|\mathbf{W} + \Delta\mathbf{W}\|_c} (\mathbf{W} + \Delta\mathbf{W}) \quad (6)$$

其中 m 是可学习的幅度参数， $\|\cdot\|_c$ 是列范数。

2.5 参数效率对比

表 1: 不同 PEFT 方法的参数效率对比

方法	参数量比例	内存占用	训练速度	效果
全参数微调	100%	高	慢	最佳
LoRA	0.1-1%	中	快	接近最佳
QLoRA	0.1-1%	低	快	接近最佳
Prefix Tuning	0.1-0.5%	低	快	良好
P-Tuning v2	0.1-1%	中	中	良好
AdaLoRA	0.1-1%	中	中	接近最佳

3 监督微调技术

3.1 SFT (Supervised Fine-Tuning)

概念解释: SFT 是在有标签数据上对预训练模型进行微调，使模型适应特定任务。

数据格式:

```
# JSON 格式
{
    "instruction": "将以下文本翻译成英文",
    "input": "你好，世界",
    "output": "Hello, world"
}

# 对话格式
{
    "messages": [
        {"role": "user", "content": "什么是机器学习?"},
        {"role": "assistant", "content": "机器学习是..."}
    ]
}
```

Listing 5: SFT 数据格式示例

训练流程:

```
from transformers import (
    AutoModelForCausalLM,
    AutoTokenizer,
    TrainingArguments,
    Trainer,
    DataCollatorForLanguageModeling
)
from datasets import load_dataset
import torch

# 加载模型和分词器
model_name = "meta-llama/Llama-2-7b-hf"
model = AutoModelForCausalLM.from_pretrained(model_name)
tokenizer = AutoTokenizer.from_pretrained(model_name)
tokenizer.pad_token = tokenizer.eos_token

# 准备数据
def format_prompt(example):
    prompt = f"### Instruction:\n{example['instruction']}\n\n### Input:\n{example['input']}\n\n### Response:\n{example['output']}"
```

```
        return {"text": prompt}

dataset = load_dataset("json", data_files="train.json")
dataset = dataset.map(format_prompt)

def tokenize_function(examples):
    return tokenizer(
        examples["text"],
        truncation=True,
        max_length=512,
        padding="max_length"
    )

tokenized_dataset = dataset.map(tokenize_function, batched=True)

# 训练参数
training_args = TrainingArguments(
    output_dir="./results",
    num_train_epochs=3,
    per_device_train_batch_size=4,
    gradient_accumulation_steps=4,
    learning_rate=2e-5,
    fp16=True,
    logging_steps=100,
    save_steps=500,
)

# 数据整理器
data_collator = DataCollatorForLanguageModeling(
    tokenizer=tokenizer,
    mlm=False # 因果语言建模
)

# 训练器
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=tokenized_dataset["train"],
    data_collator=data_collator,
)
```

```
# 开始训练
trainer.train()
```

Listing 6: SFT 训练示例

3.2 指令微调 (Instruction Tuning)

概念解释： 指令微调通过大量指令-输出对训练模型，使模型能够理解和遵循指令。

数据构建：

```
# 指令数据格式
instruction_data = [
    {
        "instruction": "解释以下概念",
        "input": "量子计算",
        "output": "量子计算是利用量子力学原理..."
    },
    {
        "instruction": "总结以下文本",
        "input": "长文本...",
        "output": "总结内容..."
    }
]

# 使用 Alpaca 格式
def format_alpaca(example):
    return {
        "text": f"""Below is an instruction that describes a task. Write
a response that appropriately completes the request.

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

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

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

```
}
```

Listing 7: 指令数据构建示例

效果评估：

- 指令遵循准确率
- 输出质量评分
- 任务完成率

3.3 对话微调 (Chat Fine-Tuning)

概念解释：对话微调专门针对多轮对话场景，训练模型进行自然对话。

多轮对话数据处理：

```
def format_chat(messages):  
    """格式化多轮对话"""  
    formatted = ""  
    for msg in messages:  
        role = msg["role"]  
        content = msg["content"]  
        if role == "user":  
            formatted += f"User: {content}\n"  
        elif role == "assistant":  
            formatted += f"Assistant: {content}\n"  
    return formatted  
  
# 对话数据示例  
chat_data = {  
    "messages": [  
        {"role": "user", "content": "你好"},  
        {"role": "assistant", "content": "你好！有什么可以帮助你吗？"},  
        {"role": "user", "content": "介绍一下Python"},  
        {"role": "assistant", "content": "Python是一种高级编程语言..."}  
    ]  
}
```

Listing 8: 对话数据处理示例

4 推理加速技术

4.1 vLLM (Very Large Language Model)

概念解释：vLLM 是专门为大模型推理优化的服务框架，通过 PagedAttention 等技术大幅提升吞吐量。

PagedAttention 原理：

传统 Attention 的 KV Cache 是连续的，导致内存碎片化。PagedAttention 将 KV Cache 分页管理：

$$\text{Attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right) \mathbf{V} \quad (7)$$

PagedAttention 将 \mathbf{K} 和 \mathbf{V} 存储在非连续的页面中，按需分配。

KV Cache 优化：

```
from vllm import LLM, SamplingParams

# 初始化模型
llm = LLM(
    model="meta-llama/Llama-2-7b-hf",
    tensor_parallel_size=1,
    gpu_memory_utilization=0.9
)

# 采样参数
sampling_params = SamplingParams(
    temperature=0.7,
    top_p=0.9,
    max_tokens=512
)

# 批量推理
prompts = [
    "什么是机器学习？",
    "解释一下深度学习",
    "Python 的特点是什么？"
]
```

```

outputs = llm.generate(prompts, sampling_params)

for output in outputs:
    print(f"Prompt: {output.prompt}")
    print(f"Generated: {output.outputs[0].text}\n")

```

Listing 9: vLLM 使用示例

性能优势:

- 吞吐量提升 2-4 倍
- 内存利用率提高 50-80%
- 支持连续批处理

4.2 Flash Attention

概念解释: Flash Attention 通过分块计算和在线 softmax 优化注意力机制，减少内存占用。

数学原理:

标准 Attention:

$$\mathbf{O} = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right) \mathbf{V} \quad (8)$$

Flash Attention 分块计算:

$$\mathbf{O}_i = \sum_{j=1}^N \frac{\exp(\mathbf{s}_{ij} - m_i)}{\sum_{k=1}^N \exp(\mathbf{s}_{ik} - m_i)} \mathbf{V}_j \quad (9)$$

$$m_i = \max_j \mathbf{s}_{ij}, \quad \mathbf{s}_{ij} = \frac{\mathbf{Q}_i \mathbf{K}_j^T}{\sqrt{d_k}} \quad (10)$$

内存优化:

- 标准 Attention: $O(N^2)$ 内存
- Flash Attention: $O(N)$ 内存

```

import torch
from flash_attn import flash_attn_func

```

```

# 输入
q = torch.randn(32, 128, 8, 64, dtype=torch.float16, device="cuda")
k = torch.randn(32, 128, 8, 64, dtype=torch.float16, device="cuda")
v = torch.randn(32, 128, 8, 64, dtype=torch.float16, device="cuda")

# Flash Attention
output = flash_attn_func(q, k, v, dropout_p=0.0, softmax_scale=1.0/sqrt
    (64))

print(f"输出形状: {output.shape}") # (32, 128, 8, 64)

```

Listing 10: Flash Attention 使用示例

4.3 量化推理

概念解释：量化推理通过降低模型精度减少内存占用和加速推理。

INT8 量化：

$$Q(x) = \text{round} \left(\frac{x}{\text{scale}} \right) \times \text{scale} \quad (11)$$

GPTQ 量化：

GPTQ (GPT Quantization) 是一种后训练量化方法：

$$\arg \min_{\hat{\mathbf{W}}} ||\mathbf{W}\mathbf{X} - \hat{\mathbf{W}}\mathbf{X}||_2^2 \quad (12)$$

```

from transformers import AutoModelForCausalLM, AutoTokenizer
from auto_gptq import AutoGPTQForCausalLM

# 加载模型
model_name = "meta-llama/Llama-2-7b-hf"

# GPTQ 量化
model = AutoGPTQForCausalLM.from_quantized(
    model_name,
    use_safetensors=True,
    device="cuda:0"
)

```

```
tokenizer = AutoTokenizer.from_pretrained(model_name)

# 推理
inputs = tokenizer("Hello, how are you?", return_tensors="pt").to("cuda")
outputs = model.generate(**inputs, max_length=50)
print(tokenizer.decode(outputs[0]))
```

Listing 11: GPTQ 量化示例

4.4 模型并行与张量并行

概念解释：当模型太大无法放入单卡时，需要将模型分布到多个 GPU。

张量并行：将矩阵乘法分块：

$$\mathbf{Y} = \mathbf{XW} = \mathbf{X}[\mathbf{W}_1 | \mathbf{W}_2] = [\mathbf{XW}_1 | \mathbf{XW}_2] \quad (13)$$

流水线并行：将模型按层分割到不同 GPU。

5 模型部署技术

5.1 模型压缩

知识蒸馏：用大模型（教师）指导小模型（学生）学习。

数学公式：

$$\mathcal{L} = \alpha \mathcal{L}_{CE}(\mathbf{y}, \mathbf{p}_s) + (1 - \alpha) \mathcal{L}_{KL}(\mathbf{p}_t, \mathbf{p}_s) \quad (14)$$

其中 \mathbf{p}_t 是教师模型的输出， \mathbf{p}_s 是学生模型的输出。

5.2 服务化部署

API 服务：使用 FastAPI、Flask 等框架部署模型服务。

```
from fastapi import FastAPI
from transformers import pipeline
import torch

app = FastAPI()
```

```
# 加载模型
model = pipeline(
    "text-generation",
    model="meta-llama/Llama-2-7b-hf",
    device=0 if torch.cuda.is_available() else -1
)

@app.post("/generate")
async def generate_text(prompt: str, max_length: int = 100):
    result = model(prompt, max_length=max_length)
    return {"generated_text": result[0]["generated_text"]}

# 运行: uvicorn app:app --host 0.0.0.0 --port 8000
```

Listing 12: 模型服务化部署示例

5.3 边缘部署

模型量化：INT8/INT4 量化减少模型大小。

模型剪枝：移除不重要的权重。

专用硬件：使用 NPU、TPU 等专用芯片加速。

6 推理优化技术

6.1 KV Cache

概念解释：KV Cache 缓存之前计算的 Key 和 Value，避免重复计算。

优化效果：

- 减少计算量：从 $O(n^2)$ 到 $O(n)$
- 提升速度：2-10 倍加速

6.2 连续批处理 (Continuous Batching)

概念解释：动态管理批次，新请求可以立即加入，完成的请求可以立即释放。

优势：

- 提高 GPU 利用率
- 降低延迟
- 支持动态负载

6.3 动态批处理

概念解释： 根据请求长度动态调整批次大小。

策略：

- 短请求优先
- 长度相似请求分组
- 最大批次大小限制

7 总结

第一部分介绍了大语言模型的先进技术：

参数高效微调：

- LoRA：低秩适应，大幅减少参数量
- QLoRA：量化 + LoRA，进一步降低内存
- PEFT：统一框架，支持多种方法

监督微调：

- SFT：有监督微调
- 指令微调：提升指令遵循能力
- 对话微调：优化多轮对话

推理加速：

- vLLM：PagedAttention 优化

- Flash Attention: 内存高效注意力
- 量化推理: INT8/INT4 量化

部署与优化:

- 模型压缩: 知识蒸馏、剪枝
- 服务化部署: API 服务
- 推理优化: KV Cache、连续批处理

Part II

第二部分：评估方法与评测基准

8 评估指标与方法

评估指标是衡量大语言模型性能的重要工具。不同的任务需要不同的评估指标，本节详细介绍各种评估指标的定义、计算方法和实现。

8.1 困惑度 (Perplexity)

概念解释: 困惑度 (Perplexity, PPL) 是语言模型评估中最常用的指标，衡量模型对测试数据的预测不确定性。困惑度越低，模型性能越好。

数学定义:

对于测试序列 $\mathbf{w} = w_1, w_2, \dots, w_N$ ，困惑度定义为:

$$\text{PPL}(\mathbf{w}) = P(w_1, w_2, \dots, w_N)^{-\frac{1}{N}} = \sqrt[N]{\frac{1}{P(w_1, w_2, \dots, w_N)}} \quad (15)$$

使用链式法则展开:

$$\text{PPL}(\mathbf{w}) = \sqrt[N]{\prod_{i=1}^N \frac{1}{P(w_i | w_1, \dots, w_{i-1})}} \quad (16)$$

使用对数形式 (数值稳定):

$$\text{PPL}(\mathbf{w}) = \exp \left(-\frac{1}{N} \sum_{i=1}^N \log P(w_i | w_1, \dots, w_{i-1}) \right) \quad (17)$$

符号说明：

- N : 序列长度（词数）
- w_i : 第 i 个词
- $P(w_i|w_1, \dots, w_{i-1})$: 给定前文的条件概率
- \log : 自然对数

从零实现：

```
import torch
import torch.nn.functional as F
import math

def calculate_perplexity(model, tokenizer, text, device="cuda"):
    """
    计算文本的困惑度

    参数:
        model: 语言模型
        tokenizer: 分词器
        text: 输入文本
        device: 设备
    """
    # 分词
    tokens = tokenizer.encode(text, return_tensors="pt").to(device)
    input_ids = tokens[:, :-1] # 输入（除了最后一个token）
    target_ids = tokens[:, 1:] # 目标（除了第一个token）

    model.eval()
    with torch.no_grad():
        # 前向传播
        outputs = model(input_ids)
        logits = outputs.logits # (batch_size, seq_len, vocab_size)

        # 计算每个位置的对数概率
        log_probs = F.log_softmax(logits, dim=-1)

        # 获取目标token的对数概率
        # log_probs: (batch_size, seq_len, vocab_size)
```

```

    # target_ids: (batch_size, seq_len)
    # 需要选择每个位置对应target的对数概率
    selected_log_probs = log_probs.gather(
        dim=2,
        index=target_ids.unsqueeze(2)
    ).squeeze(2)  # (batch_size, seq_len)

    # 计算平均负对数似然
    nll = -selected_log_probs.mean().item()

    # 困惑度
    perplexity = math.exp(nll)

    return perplexity, nll

# 使用示例
from transformers import AutoModelForCausalLM, AutoTokenizer

model_name = "gpt2"  # 示例模型
model = AutoModelForCausalLM.from_pretrained(model_name)
tokenizer = AutoTokenizer.from_pretrained(model_name)

text = "The quick brown fox jumps over the lazy dog."
ppl, nll = calculate_perplexity(model, tokenizer, text)
print(f"文本: {text}")
print(f"负对数似然: {nll:.4f}")
print(f"困惑度: {ppl:.4f}")

```

Listing 13: 困惑度从零实现

使用库实现:

```

from evaluate import load

perplexity = load("perplexity", module_type="metric")

# 计算困惑度
results = perplexity.compute(
    model_id="gpt2",
    add_start_token=False,

```

```

    predictions=["The quick brown fox jumps over the lazy dog."]
)

print(f"困惑度: {results['mean_perplexity']:.4f}")

```

Listing 14: 使用 evaluate 库计算困惑度

计算案例:

案例 1: 给定序列“the cat sat”，假设模型预测的概率为：

- $P(\text{cat}|\text{the}) = 0.3$
- $P(\text{sat}|\text{the cat}) = 0.5$

逐步计算:

$$\text{PPL} = \exp \left(-\frac{1}{2} [\log(0.3) + \log(0.5)] \right) \quad (18)$$

$$= \exp \left(-\frac{1}{2} [-1.204 + (-0.693)] \right) \quad (19)$$

$$= \exp \left(-\frac{1}{2} \times (-1.897) \right) \quad (20)$$

$$= \exp(0.9485) \quad (21)$$

$$= 2.582 \quad (22)$$

代码验证:

```

import math

# 给定概率
probs = [0.3, 0.5]

# 计算困惑度
log_probs = [math.log(p) for p in probs]
nll = -sum(log_probs) / len(probs)
ppl = math.exp(nll)

print(f"概率: {probs}")
print(f"对数概率: {log_probs}")
print(f"平均负对数似然: {nll:.4f}")
print(f"困惑度: {ppl:.4f}") # 2.582

```

Listing 15: 案例 1 代码验证

案例 2: 更长的序列

假设序列长度为 10，平均对数概率为 -2.0:

$$\text{PPL} = \exp\left(-\frac{1}{10} \times (-2.0) \times 10\right) \quad (23)$$

$$= \exp(2.0) \quad (24)$$

$$= 7.389 \quad (25)$$

8.2 BLEU 分数

概念解释: BLEU (Bilingual Evaluation Understudy) 是机器翻译和文本生成任务中最常用的评估指标，通过比较 n-gram 匹配来衡量生成文本与参考文本的相似度。

数学定义:

n-gram 精确度:

$$P_n = \frac{\sum_{\text{n-gram} \in \text{candidate}} \text{Count}_{\text{clip}}(\text{n-gram})}{\sum_{\text{n-gram} \in \text{candidate}} \text{Count}(\text{n-gram})} \quad (26)$$

其中 $\text{Count}_{\text{clip}}$ 是截断计数，不超过参考文本中该 n-gram 的最大出现次数。

BLEU 分数:

$$\text{BLEU} = \text{BP} \times \exp\left(\sum_{n=1}^N w_n \log P_n\right) \quad (27)$$

其中:

- BP: 简短惩罚 (Brevity Penalty)
- w_n : n-gram 权重, 通常 $w_n = 1/N$
- N : 最大 n-gram 阶数, 通常 $N = 4$

简短惩罚:

$$\text{BP} = \begin{cases} 1 & \text{if } c > r \\ e^{1-r/c} & \text{if } c \leq r \end{cases} \quad (28)$$

其中 c 是候选文本长度, r 是参考文本长度。

从零实现:

```
from collections import Counter
import math

def get_ngrams(tokens, n):
    """获取 n-gram"""
    return [tuple(tokens[i:i+n]) for i in range(len(tokens)-n+1)]

def calculate_bleu(reference, candidate, max_n=4):
    """
    计算 BLEU 分数

    参数:
        reference: 参考文本 (列表, 每个元素是一个参考)
        candidate: 候选文本 (token 列表)
        max_n: 最大 n-gram 阶数
    """
    # 如果 reference 是字符串, 转换为列表
    if isinstance(reference[0], str):
        reference = [ref.split() for ref in reference]
    if isinstance(candidate, str):
        candidate = candidate.split()

    # 计算简短惩罚
    c = len(candidate)
    r = min(len(ref) for ref in reference) # 最接近的参考长度
    bp = 1.0 if c > r else math.exp(1 - r / c)

    # 计算各阶 n-gram 精确度
    precisions = []

    for n in range(1, max_n + 1):
        # 候选文本的 n-gram
        candidate_ngrams = get_ngrams(candidate, n)
        candidate_counts = Counter(candidate_ngrams)

        # 参考文本的 n-gram (所有参考)
        reference_counts_list = []
        for ref in reference:
            ref_ngrams = get_ngrams(ref, n)
```

```

        reference_counts_list.append(Counter(ref_ngrams))

    # 计算截断计数
    clipped_count = 0
    total_count = sum(candidate_counts.values())

    for ngram, count in candidate_counts.items():
        # 在所有参考中找到该 n-gram 的最大计数
        max_ref_count = max(
            ref_counts.get(ngram, 0) for ref_counts in
reference_counts_list
        )
        clipped_count += min(count, max_ref_count)

    # n-gram 精确度
    precision = clipped_count / total_count if total_count > 0 else 0
    precisions.append(precision)

    # 计算几何平均
    if any(p == 0 for p in precisions):
        return 0.0

    log_precision_sum = sum(math.log(p) for p in precisions)
    bleu = bp * math.exp(log_precision_sum / len(precisions))

    return bleu, precisions, bp

# 测试案例
reference = ["the cat is on the mat"]
candidate = "the cat the cat on the mat"

bleu_score, precisions, bp = calculate_bleu(reference, candidate)
print(f"参考: {reference}")
print(f"候选: {candidate}")
print(f"1-gram 精确度: {precisions[0]:.4f}")
print(f"2-gram 精确度: {precisions[1]:.4f}")
print(f"3-gram 精确度: {precisions[2]:.4f}")
print(f"4-gram 精确度: {precisions[3]:.4f}")
print(f"简短惩罚: {bp:.4f}")
print(f"BLEU 分数: {bleu_score:.4f}")

```

Listing 16: BLEU 分数从零实现

使用库实现：

```
from sacrebleu import BLEU

bleu = BLEU()

# 单个参考
reference = ["the cat is on the mat"]
candidate = "the cat the cat on the mat"
score = bleu.sentence_score(candidate, reference)
print(f"BLEU 分数: {score.score:.4f}")

# 多个参考
references = [
    ["the cat is on the mat"],
    ["there is a cat on the mat"]
]
score = bleu.sentence_score(candidate, references)
print(f"BLEU 分数 (多参考) : {score.score:.4f}")
```

Listing 17: 使用 sacrebleu 库

计算案例：

案例 1：

- 参考: "the cat is on the mat"
- 候选: "the cat the cat on the mat"

逐步计算：

1-gram:

- 候选: the(2), cat(2), the(2), cat(2), on(1), the(2), mat(1)
- 参考: the(2), cat(1), is(1), on(1), the(2), mat(1)
- 截断计数: the(2), cat(1), on(1), mat(1) = 5
- 总数: 7

- $P_1 = 5/7 = 0.7143$

2-gram:

- 候选: (the cat)(2), (cat the)(1), (the cat)(2), (cat on)(1), (on the)(1), (the mat)(1)
- 参考: (the cat)(1), (cat is)(1), (is on)(1), (on the)(1), (the mat)(1)
- 截断计数: (the cat)(1), (cat on)(0), (on the)(1), (the mat)(1) = 3
- 总数: 6
- $P_2 = 3/6 = 0.5000$

简短惩罚:

- $c = 7, r = 6$
- $BP = e^{1-6/7} = e^{0.1429} = 1.1537$

BLEU 分数:

$$BLEU = BP \times \exp\left(\frac{1}{4}[\log P_1 + \log P_2 + \log P_3 + \log P_4]\right) \quad (29)$$

$$= 1.1537 \times \exp\left(\frac{1}{4}[\log(0.7143) + \log(0.5) + \log(0) + \log(0)]\right) \quad (30)$$

$$= 0 \quad (\text{因为 } P_3 = P_4 = 0) \quad (31)$$

代码验证:

```
reference = ["the cat is on the mat"]
candidate = "the cat the cat on the mat"

bleu_score, precisions, bp = calculate_bleu(reference, candidate)
print(f"1-gram 精确度: {precisions[0]:.4f}") # 0.7143
print(f"2-gram 精确度: {precisions[1]:.4f}") # 0.5000
print(f"简短惩罚: {bp:.4f}") # 1.1537
print(f"BLEU 分数: {bleu_score:.4f}") # 0.0000 (因为3-gram和4-gram为0)
```

Listing 18: 案例 1 代码验证

案例 2: 更好的匹配

参考: "the cat sat on the mat"

候选: "the cat sat on the mat"

这是完美匹配, BLEU 分数应该接近 1.0。

逐步计算:

1-gram:

- 候选: the(2), cat(1), sat(1), on(1), the(2), mat(1)
- 参考: the(2), cat(1), sat(1), on(1), the(2), mat(1)
- 截断计数: 所有匹配 = 6
- 总数: 6
- $P_1 = 6/6 = 1.0$

2-gram:

- 候选: (the cat)(1), (cat sat)(1), (sat on)(1), (on the)(1), (the mat)(1)
- 参考: (the cat)(1), (cat sat)(1), (sat on)(1), (on the)(1), (the mat)(1)
- 截断计数: 所有匹配 = 5
- 总数: 5
- $P_2 = 5/5 = 1.0$

类似地, $P_3 = 1.0$, $P_4 = 1.0$ 。

简短惩罚:

- $c = 6, r = 6$
- $BP = 1.0$ (因为 $c = r$)

BLEU 分数:

$$\text{BLEU} = 1.0 \times \exp \left(\frac{1}{4} [\log(1.0) + \log(1.0) + \log(1.0) + \log(1.0)] \right) \quad (32)$$

$$= 1.0 \times \exp(0) \quad (33)$$

$$= 1.0 \quad (34)$$

代码验证:

```
reference = ["the cat sat on the mat"]
candidate = "the cat sat on the mat"

bleu_score, precisions, bp = calculate_bleu(reference, candidate)
print(f"所有 n-gram 精确度: {precisions}") # [1.0, 1.0, 1.0, 1.0]
print(f"简短惩罚: {bp:.4f}") # 1.0000
print(f"BLEU 分数: {bleu_score:.4f}") # 1.0000
```

Listing 19: 案例 2 代码验证

案例 3: 部分匹配

参考: "the cat is sitting on the mat"

候选: "a cat sits on mat"

逐步计算:

1-gram:

- 候选: a(1), cat(1), sits(1), on(1), mat(1)
- 参考: the(2), cat(1), is(1), sitting(1), on(1), the(2), mat(1)
- 匹配: cat(1), on(1), mat(1) = 3
- 总数: 5
- $P_1 = 3/5 = 0.6$

简短惩罚:

- $c = 5, r = 7$
- $BP = e^{1-7/5} = e^{-0.4} = 0.6703$

由于 2-gram、3-gram、4-gram 匹配较少, 最终 BLEU 分数较低。

8.3 ROUGE 分数

概念解释: ROUGE (Recall-Oriented Understudy for Gisting Evaluation) 主要用于文本摘要评估, 关注召回率。

ROUGE-N:

$$\text{ROUGE-N} = \frac{\sum_{S \in \text{References}} \sum_{n\text{-gram} \in S} \text{Count}_{\text{match}}(n\text{-gram})}{\sum_{S \in \text{References}} \sum_{n\text{-gram} \in S} \text{Count}(n\text{-gram})} \quad (35)$$

ROUGE-L (最长公共子序列):

$$\text{ROUGE-L} = \frac{(1 + \beta^2) R_{lcs} P_{lcs}}{R_{lcs} + \beta^2 P_{lcs}} \quad (36)$$

其中:

$$R_{lcs} = \frac{LCS(X, Y)}{m} \quad (\text{召回率}) \quad (37)$$

$$P_{lcs} = \frac{LCS(X, Y)}{n} \quad (\text{精确率}) \quad (38)$$

$LCS(X, Y)$ 是最长公共子序列长度, m 是参考长度, n 是候选长度。

从零实现:

```
from collections import Counter

def lcs_length(x, y):
    """计算最长公共子序列长度"""
    m, n = len(x), len(y)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    for i in range(1, m + 1):
        for j in range(1, n + 1):
            if x[i-1] == y[j-1]:
                dp[i][j] = dp[i-1][j-1] + 1
            else:
                dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    return dp[m][n]

def rouge_n(reference, candidate, n=1):
    """计算 ROUGE-N"""
    ref_ngrams = Counter(get_ngrams(reference.split(), n))
    cand_ngrams = Counter(get_ngrams(candidate.split(), n))

    matches = sum(min(ref_ngrams[ngram], cand_ngrams[ngram])
                  for ngram in ref_ngrams)
    total = sum(ref_ngrams.values())
```

```

    return matches / total if total > 0 else 0.0

def rouge_l(reference, candidate, beta=1.2):
    """计算 ROUGE-L"""
    ref_tokens = reference.split()
    cand_tokens = candidate.split()

    lcs_len = lcs_length(ref_tokens, cand_tokens)

    if len(ref_tokens) == 0 or len(cand_tokens) == 0:
        return 0.0

    recall = lcs_len / len(ref_tokens)
    precision = lcs_len / len(cand_tokens)

    if recall + precision == 0:
        return 0.0

    f_score = (1 + beta**2) * recall * precision / (recall + beta**2 *
precision)
    return f_score, recall, precision

# 测试
reference = "the cat is on the mat"
candidate = "the cat sat on the mat"

rouge_1 = rouge_n(reference, candidate, n=1)
rouge_2 = rouge_n(reference, candidate, n=2)
rouge_l_score, recall, precision = rouge_l(reference, candidate)

print(f"ROUGE-1: {rouge_1:.4f}")
print(f"ROUGE-2: {rouge_2:.4f}")
print(f"ROUGE-L: {rouge_l_score:.4f} (Recall: {recall:.4f}, Precision: {
precision:.4f})")

```

Listing 20: ROUGE 分数从零实现

计算案例:

案例 1:

- 参考: "the cat is on the mat"
- 候选: "the cat sat on the mat"

ROUGE-1 计算:

- 参考 1-gram: the(2), cat(1), is(1), on(1), mat(1), 共 6 个
- 候选 1-gram: the(2), cat(1), sat(1), on(1), mat(1), 共 6 个
- 匹配: the(2), cat(1), on(1), mat(1) = 5
- ROUGE-1 = $5/6 = 0.8333$

ROUGE-L 计算:

- 参考序列: [the, cat, is, on, the, mat]
- 候选序列: [the, cat, sat, on, the, mat]
- LCS: [the, cat, on, the, mat], 长度为 5
- Recall = $5/6 = 0.8333$
- Precision = $5/6 = 0.8333$
- ROUGE-L = $\frac{2 \times 0.8333 \times 0.8333}{0.8333 + 0.8333} = 0.8333$

代码验证:

```
reference = "the cat is on the mat"
candidate = "the cat sat on the mat"

rouge_1 = rouge_n(reference, candidate, n=1)
rouge_2 = rouge_n(reference, candidate, n=2)
rouge_l_score, recall, precision = rouge_l(reference, candidate)

print(f"ROUGE-1: {rouge_1:.4f}") # 0.8333
print(f"ROUGE-2: {rouge_2:.4f}")
print(f"ROUGE-L: {rouge_l_score:.4f}") # 0.8333
print(f"Recall: {recall:.4f}, Precision: {precision:.4f}")
```

Listing 21: ROUGE 案例 1 代码验证

案例 2:

- 参考: "the cat is on the mat"
- 候选: "cat mat"

ROUGE-1 计算:

- 参考 1-gram: 6 个
- 候选 1-gram: cat(1), mat(1), 共 2 个
- 匹配: cat(1), mat(1) = 2
- ROUGE-1 = $2/6 = 0.3333$

ROUGE-L 计算:

- LCS: [cat, mat], 长度为 2
- Recall = $2/6 = 0.3333$
- Precision = $2/2 = 1.0$
- ROUGE-L = $\frac{2 \times 0.3333 \times 1.0}{0.3333 + 1.0} = 0.5000$

8.4 METEOR 分数

概念解释: METEOR 考虑同义词匹配, 比 BLEU 更灵活。

数学公式:

$$\text{METEOR} = (1 - \text{Penalty}) \times F_{\text{mean}} \quad (39)$$

其中:

$$F_{\text{mean}} = \frac{P \times R}{\alpha P + (1 - \alpha) R} \quad (40)$$

$$\text{Penalty} = 0.5 \times \left(\frac{\text{chunks}}{\text{unigrams_matched}} \right)^3 \quad (41)$$

8.5 BERTScore

概念解释: BERTScore 使用 BERT 嵌入计算语义相似度。

数学公式:

$$\text{Precision} = \frac{1}{|\hat{\mathbf{x}}|} \sum_{\hat{\mathbf{x}}_i \in \hat{\mathbf{x}}} \max_{\mathbf{x}_j \in \mathbf{x}} \hat{\mathbf{x}}_i^T \mathbf{x}_j \quad (42)$$

$$\text{Recall} = \frac{1}{|\mathbf{x}|} \sum_{\mathbf{x}_j \in \mathbf{x}} \max_{\hat{\mathbf{x}}_i \in \hat{\mathbf{x}}} \hat{\mathbf{x}}_i^T \mathbf{x}_j \quad (43)$$

$$\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \quad (44)$$

9 评测基准与数据集

9.1 GLUE 和 SuperGLUE

GLUE: General Language Understanding Evaluation, 包含 9 个自然语言理解任务。

SuperGLUE: GLUE 的升级版, 包含更具挑战性的任务。

9.2 MMLU

概念解释: Massive Multitask Language Understanding, 包含 57 个任务, 涵盖数学、物理、历史等多个领域。

9.3 HumanEval 和 MBPP

HumanEval: 164 个 Python 编程问题, 评估代码生成能力。

MBPP: 974 个 Python 编程问题。

9.4 MT-Bench 和 AlpacaEval

MT-Bench: 多轮对话评估基准。

AlpacaEval: 指令遵循能力评估。

10 QA Pair (问答对)

10.1 LoRA 相关问答

Q1: 什么是 LoRA? 它的核心思想是什么?

A: LoRA (Low-Rank Adaptation) 是一种参数高效微调方法。核心思想是：对于预训练权重矩阵 \mathbf{W} ，不直接更新它，而是学习一个低秩分解的增量 $\Delta\mathbf{W} = \mathbf{B}\mathbf{A}$ ，其中 $\mathbf{A} \in \mathbb{R}^{r \times k}$ ， $\mathbf{B} \in \mathbb{R}^{d \times r}$ ， $r \ll \min(d, k)$ 。这样只需要训练 $r(d+k)$ 个参数，而不是 dk 个参数。

Q2: LoRA 和全参数微调的区别是什么？

A:

- **参数量：** LoRA 只更新 0.1-1% 的参数，全参数微调更新 100% 的参数
- **内存占用：** LoRA 内存占用大幅降低
- **训练速度：** LoRA 训练更快
- **效果：** LoRA 效果通常接近全参数微调 (95-99%)
- **灵活性：** LoRA 可以保存多个适配器，快速切换任务

Q3: 如何选择 LoRA 的 rank？

A: rank 的选择需要权衡：

- **较小的 rank (4-8)：** 参数量少，训练快，但可能表达能力不足
- **中等 rank (16-32)：** 平衡性能和效率，适用于大多数任务
- **较大的 rank (64-128)：** 表达能力更强，但参数量增加
- **建议从 $r = 8$ 开始，根据效果调整**

10.2 评估指标相关问答

Q4: 如何计算 BLEU 分数？请给出详细步骤。

A: BLEU 分数计算步骤：

1. 计算各阶 n-gram (1-4) 的精确度 P_n
2. 计算简短惩罚 BP
3. 计算几何平均： $\exp(\frac{1}{4} \sum_{n=1}^4 \log P_n)$
4. 最终 BLEU = BP \times 几何平均

Q5: 困惑度和 BLEU 的区别是什么？

A:

- **困惑度**：评估语言模型的预测不确定性，不需要参考文本，值越小越好
- **BLEU**：评估生成文本与参考文本的相似度，需要参考文本，值越大越好 (0-1)
- **应用场景**：困惑度用于语言模型评估，BLEU 用于翻译和文本生成任务

11 综合练习

11.1 概念理解题

1. 解释 LoRA 的数学原理，说明为什么低秩分解能够有效。
2. 比较 QLoRA 和 LoRA 的区别，说明量化的作用。
3. 解释 Flash Attention 如何减少内存占用。
4. 说明 BLEU 分数的简短惩罚的作用。
5. 解释 ROUGE-L 和 ROUGE-N 的区别。

11.2 计算题

1. 困惑度计算：

- 给定序列长度为 100，平均对数概率为 -2.5，计算困惑度
- 手算并编写代码验证

2. BLEU 计算：

- 参考: "the cat sat on the mat"
- 候选: "a cat sat on mat"
- 计算 1-gram 到 4-gram 的精确度、简短惩罚和 BLEU 分数
- 手算并编写代码验证

3. ROUGE 计算：

- 参考: "the cat is on the mat"
- 候选: "cat mat"
- 计算 ROUGE-1、ROUGE-2 和 ROUGE-L
- 手算并编写代码验证

11.3 代码实现题

1. 实现一个完整的 LoRA 训练脚本，包括数据加载、模型配置、训练循环和评估。
2. 实现 BLEU 分数计算函数，支持多个参考文本。
3. 实现 ROUGE 分数计算函数，包括 ROUGE-N 和 ROUGE-L。
4. 实现困惑度计算函数，支持批量计算。

11.4 案例分析题

1. 给定一个文本生成任务，设计完整的评估方案，包括选择合适的评估指标、实现评估代码、分析结果。
2. 分析 LoRA 在不同任务上的效果，比较不同 rank 设置的影响。
3. 设计一个模型部署方案，包括模型压缩、服务化部署和性能优化。

通过完成以上练习，读者可以深入理解大语言模型的先进技术和评估方法，掌握从模型微调到评估部署的完整流程。