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

[scale=0.9]

[circle, draw, minimum size=0.6cm, fill=blue!20, line width=1pt] (x1) at (0, -1.8) ; [circle, draw, minimum size=0.6cm, fill=blue!20, line width=1pt] (x2) at (0, -0.6) ; [circle, draw, minimum size=0.6cm, fill=blue!20, line width=1pt] (x3) at (0, 0.6) ; [circle, draw, minimum size=0.6cm, fill=blue!20, line width=1pt] (x4) at (0, 1.8) ;

[circle, draw, minimum size=0.5cm, fill=green!20, line width=1pt] (h11) at (3, -4.2) ; [circle, draw, minimum size=0.5cm, fill=green!20, line width=1pt] (h12) at (3, -3.0) ; [circle, draw, minimum size=0.5cm, fill=green!20, line width=1pt] (h13) at (3, -1.8) ; [circle, draw, minimum size=0.5cm, fill=green!20, line width=1pt] (h14) at (3, -0.6) ; [circle, draw, minimum size=0.5cm, fill=green!20, line width=1pt] (h15) at (3, 0.6) ; [circle, draw, minimum size=0.5cm, fill=green!20, line width=1pt] (h16) at (3, 1.8) ; [circle, draw, minimum size=0.5cm, fill=green!20, line width=1pt] (h17) at (3, 3.0) ; [circle, draw, minimum size=0.5cm, fill=green!20, line width=1pt] (h18) at (3, 4.2) ;

[circle, draw, minimum size=0.5cm, fill=orange!20, line width=1pt] (h21) at (6, -1.8) ; [circle,

draw, minimum size=0.5cm, fill=orange!20, line width=1pt] (h22) at (6, -0.6) ; [circle, draw, minimum size=0.5cm, fill=orange!20, line width=1pt] (h23) at (6, 0.6) ; [circle, draw, minimum size=0.5cm, fill=orange!20, line width=1pt] (h24) at (6, 1.8) ;

[circle, draw, minimum size=0.6cm, fill=red!20, line width=1pt] (y) at (9, 0) ;

ıin 1,...,4 ȷin 1,...,8 [->, gray!50, line width=0.3pt] (xı) – (h1ȷ);

ıin 1,...,8 ȷin 1,...,4 [->, gray!50, line width=0.3pt] (h1ı) – (h2ȷ);

ıin 1,...,4 [->, gray!50, line width=0.3pt] (h2ı) – (y);

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

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:

$$\mathbf{W}' = \mathbf{W} + \Delta\mathbf{W} = \mathbf{W} + \mathbf{BA} \tag{1}$$

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:

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

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 $\mathbf{B}$

初始化 $\mathbf{A}$ 为随机小值，$\mathbf{B}$ 为零矩阵
冻结原始权重 $\mathbf{W}$
repeat
  for 每个批次 $(\mathbf{x}, \mathbf{y}) \in \mathcal{D}$ **do**
    前向传播：$\mathbf{h} = \mathbf{W}\mathbf{x} + \mathbf{B}(\mathbf{A}\mathbf{x})$
    计算损失：$\mathcal{L} = \text{loss}(\mathbf{h}, \mathbf{y})$
    反向传播，只更新 $\mathbf{A}$ 和 $\mathbf{B}$
    $\mathbf{A} \leftarrow \mathbf{A} - \eta \nabla_{\mathbf{A}} \mathcal{L}$
    $\mathbf{B} \leftarrow \mathbf{B} - \eta \nabla_{\mathbf{B}} \mathcal{L}$
  end for
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:.2
    f}x")
```

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：

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

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：

$$\mathbf{W}' = \mathbf{W}_Q + \frac{\alpha}{r}\mathbf{BA} \tag{4}$$

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：

```python
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：

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

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：

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

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:

```python
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！有什么可以帮助你的吗？"},
        {"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：

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

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：

```python
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

5 SGLang、vLLM、Transformer、PyTorch 对比

5.1 框架定位与用途

PyTorch:

- **定位**：深度学习框架，提供底层张量计算和自动微分
- **用途**：模型训练、研究、原型开发
- **特点**：灵活、动态图、丰富的生态系统

Transformer:

- **定位**：模型架构，定义网络结构
- **用途**：构建编码器-解码器、自注意力等结构
- **特点**：标准化的注意力机制实现

vLLM:

- **定位**：推理服务框架，优化推理性能
- **用途**：生产环境部署、高吞吐量推理服务
- **特点**：PagedAttention、连续批处理、高吞吐量

SGLang:

- **定位**：结构化生成框架，优化复杂提示推理
- **用途**：函数调用、JSON 生成、批量推理
- **特点**：RadixAttention、结构化生成、高性能

5.2 性能对比

特性	PyTorch	Transformer	vLLM	SGLang
训练速度	中等	中等	不支持训练	不支持训练
推理吞吐量	低	低	高	很高
内存效率	中等	中等	高	很高
结构化生成	需手动实现	需手动实现	需手动实现	原生支持
前缀共享	需手动实现	需手动实现	部分支持	自动优化
易用性	高	中等	高	高

表 2: 框架特性对比

5.3 优劣对比

PyTorch:

优势:

- 灵活性强，支持动态图
- 丰富的预训练模型和工具
- 活跃的社区和文档
- 适合研究和实验

劣势:

- 推理性能一般
- 内存占用较大
- 生产部署需要额外优化

Transformer：

优势：

- 标准化实现，易于理解
- 广泛使用，兼容性好
- 支持多种注意力机制

劣势：

- 推理性能未优化
- 内存效率一般
- 需要手动优化才能用于生产

vLLM：

优势：

- 高吞吐量推理（2-10 倍提升）
- PagedAttention 优化内存
- 连续批处理，提高 GPU 利用率
- 易于部署和使用

劣势：

- 不支持训练
- 对结构化生成支持有限
- 复杂提示场景性能一般

SGLang：

优势：

- 极高的推理吞吐量（在某些场景下比 vLLM 快 2-5 倍）
- RadixAttention 自动前缀共享

- 原生支持结构化生成
- 复杂提示场景性能优秀

劣势:

- 不支持训练
- 相对较新，生态不如 vLLM 成熟
- 主要针对特定场景优化

5.4 使用场景建议

选择 PyTorch:

- 需要训练模型
- 研究和实验阶段
- 需要高度定制化

选择 Transformer:

- 需要标准化的注意力实现
- 构建自定义模型架构
- 学习和理解 Transformer 原理

选择 vLLM:

- 生产环境部署
- 需要高吞吐量推理
- 通用文本生成任务

选择 SGLang:

- 函数调用和 JSON 生成
- 批量处理相似请求
- 复杂提示场景
- 需要极致推理性能

6 训练和推理代码示例

6.1 PyTorch 训练和推理

训练示例：

```
import torch
import torch.nn as nn
from transformers import GPT2LMHeadModel, GPT2Tokenizer, Trainer,
    TrainingArguments

# 1. 加载模型和分词器
model = GPT2LMHeadModel.from_pretrained('gpt2')
tokenizer = GPT2Tokenizer.from_pretrained('gpt2')
tokenizer.pad_token = tokenizer.eos_token

# 2. 准备数据
def prepare_dataset(texts):
    encodings = tokenizer(texts, truncation=True, padding=True,
                          max_length=512, return_tensors='pt')
    return encodings

train_texts = ["Your training data here..."]
train_dataset = prepare_dataset(train_texts)

# 3. 配置训练参数
training_args = TrainingArguments(
    output_dir='./results',
    num_train_epochs=3,
    per_device_train_batch_size=4,
    gradient_accumulation_steps=2,
    learning_rate=5e-5,
    logging_dir='./logs',
)

# 4. 创建 Trainer
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
```

```
)

# 5. 开始训练
trainer.train()

# 6. 保存模型
model.save_pretrained('./saved_model')
tokenizer.save_pretrained('./saved_model')
```

Listing 10: PyTorch 训练示例

推理示例:

```
import torch
from transformers import GPT2LMHeadModel, GPT2Tokenizer

# 1. 加载模型
model = GPT2LMHeadModel.from_pretrained('./saved_model')
tokenizer = GPT2Tokenizer.from_pretrained('./saved_model')
model.eval()

# 2. 准备输入
prompt = "The future of AI is"
inputs = tokenizer(prompt, return_tensors='pt')

# 3. 生成
with torch.no_grad():
    outputs = model.generate(
        inputs.input_ids,
        max_length=100,
        num_return_sequences=1,
        temperature=0.7,
        do_sample=True,
    )

# 4. 解码输出
generated_text = tokenizer.decode(outputs[0], skip_special_tokens=True)
print(generated_text)
```

Listing 11: PyTorch 推理示例

代码解释：

- **训练**：使用 Trainer API，配置训练参数，自动处理批次、优化器、学习率调度
- **推理**：使用 generate() 方法，支持多种生成策略（采样、贪婪等）
- **优势**：灵活、易于调试、支持自定义训练循环
- **劣势**：推理性能一般，需要手动优化批处理

6.2 Transformer 架构训练和推理

自定义 Transformer 训练：

```
import torch
import torch.nn as nn
from transformers import Transformer, TransformerConfig

# 1. 定义 Transformer 配置
config = TransformerConfig(
    vocab_size=50257,
    d_model=768,
    nhead=12,
    num_encoder_layers=12,
    num_decoder_layers=12,
    dim_feedforward=3072,
)

# 2. 创建模型
model = Transformer(config)

# 3. 定义损失函数和优化器
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=1e-4)

# 4. 训练循环
model.train()
for epoch in range(num_epochs):
    for batch in dataloader:
        src, tgt = batch
```

```
        # 前向传播
        output = model(src, tgt)
        loss = criterion(output.view(-1, vocab_size), tgt.view(-1))

        # 反向传播
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
```

Listing 12: Transformer 架构训练示例

Transformer 推理:

```
import torch

# 1. 设置为评估模式
model.eval()

# 2. 编码输入
src = tokenizer.encode("Hello, world!")
src = torch.tensor([src])

# 3. 生成（自回归）
tgt = torch.zeros((1, 1), dtype=torch.long)
for i in range(max_length):
    output = model(src, tgt)
    next_token = output[:, -1, :].argmax(dim=-1)
    tgt = torch.cat([tgt, next_token.unsqueeze(1)], dim=1)

    if next_token == eos_token_id:
        break

# 4. 解码
generated = tokenizer.decode(tgt[0].tolist())
```

Listing 13: Transformer 架构推理示例

代码解释:

- **训练**：手动实现训练循环，完全控制训练过程
- **推理**：自回归生成，逐步预测下一个 token

- **优势**：理解底层机制，可以自定义注意力等组件
- **劣势**：需要手动实现很多功能，代码复杂

6.3 vLLM 推理服务

服务部署：

```
from vllm import LLM, SamplingParams

# 1. 加载模型
llm = LLM(
    model="meta-llama/Llama-2-7b-hf",
    tensor_parallel_size=1,  # 单 GPU
    gpu_memory_utilization=0.9,  # GPU 内存使用率
)

# 2. 配置采样参数
sampling_params = SamplingParams(
    temperature=0.7,
    top_p=0.9,
    max_tokens=100,
)

# 3. 批量推理
prompts = [
    "The future of AI is",
    "Machine learning is",
    "Deep learning enables",
]

outputs = llm.generate(prompts, sampling_params)

# 4. 处理输出
for output in outputs:
    prompt = output.prompt
    generated_text = output.outputs[0].text
    print(f"Prompt: {prompt}")
    print(f"Generated: {generated_text}\n")
```

Listing 14: vLLM 推理服务示例

API 服务:

```python
from vllm.engine.arg_utils import AsyncEngineArgs
from vllm.engine.async_llm_engine import AsyncLLMEngine
from vllm.sampling_params import SamplingParams
import asyncio

# 1. 初始化异步引擎
engine_args = AsyncEngineArgs(
    model="meta-llama/Llama-2-7b-hf",
    tensor_parallel_size=1,
)
engine = AsyncLLMEngine.from_engine_args(engine_args)

# 2. 异步生成函数
async def generate_async(prompt: str):
    sampling_params = SamplingParams(temperature=0.7, max_tokens=100)
    request_id = "0"

    async for request_output in engine.generate(prompt, sampling_params,
   request_id):
        if request_output.finished:
            return request_output.outputs[0].text

# 3. 使用示例
async def main():
    result = await generate_async("The future of AI is")
    print(result)

asyncio.run(main())
```

Listing 15: vLLM API 服务示例

代码解释:

- **批量推理**：自动批处理，PagedAttention 优化内存
- **API 服务**：异步引擎，支持高并发

- **优势**：高吞吐量、内存高效、易于部署
- **劣势**：不支持训练，需要预训练模型

6.4 SGLang 推理服务

基础推理：

```
import sglang as sgl

# 1. 加载模型
runtime = sgl.Runtime(model_path="meta-llama/Llama-2-7b-hf")

# 2. 简单生成
prompt = "The future of AI is"
response = runtime.generate(prompt, max_new_tokens=100, temperature=0.7)
print(response.text)
```

Listing 16: SGLang 基础推理示例

结构化生成（JSON）：

```
import sglang as sgl
import json

# 1. 定义 JSON Schema
schema = {
    "type": "object",
    "properties": {
        "name": {"type": "string"},
        "age": {"type": "integer"},
        "city": {"type": "string"}
    }
}

# 2. 创建运行时
runtime = sgl.Runtime(model_path="meta-llama/Llama-2-7b-hf")

# 3. 结构化生成
prompt = "Generate a person's information:"
response = runtime.generate(
```

```
    prompt,
    schema=schema,
    max_new_tokens=200,
)

# 4. 解析 JSON
result = json.loads(response.text)
print(result)
```

Listing 17: SGLang JSON 生成示例

函数调用:

```
import sglang as sgl

# 1. 定义函数
functions = [
    {
        "name": "get_weather",
        "description": "Get the weather for a location",
        "parameters": {
            "type": "object",
            "properties": {
                "location": {"type": "string"},
                "unit": {"type": "string", "enum": ["celsius", "
   fahrenheit"]}
            }
        }
    }
]

# 2. 创建运行时
runtime = sgl.Runtime(model_path="meta-llama/Llama-2-7b-hf")

# 3. 函数调用生成
prompt = "What's the weather in Beijing?"
response = runtime.generate(
    prompt,
    functions=functions,
    function_call="auto",
```

```
    max_new_tokens=200,
)

# 4. 提取函数调用
function_call = response.function_call
print(f"Function: {function_call['name']}")
print(f"Arguments: {function_call['arguments']}")
```

Listing 18: SGLang 函数调用示例

批量推理（前缀共享）:

```
import sglang as sgl

# 1. 创建运行时
runtime = sgl.Runtime(model_path="meta-llama/Llama-2-7b-hf")

# 2. 共享前缀的批量请求
# 这些请求共享相同的系统提示
system_prompt = "You are a helpful assistant."
user_queries = [
    "What is AI?",
    "Explain machine learning",
    "What is deep learning?",
]

# 3. 批量生成（自动前缀共享）
responses = runtime.generate_batch(
    [system_prompt + "\n\n" + query for query in user_queries],
    max_new_tokens=100,
    temperature=0.7,
)

# 4. 处理响应
for i, response in enumerate(responses):
    print(f"Query: {user_queries[i]}")
    print(f"Response: {response.text}\n")
```

Listing 19: SGLang 批量推理示例

代码解释:

- **基础推理**：简洁的 API，类似 vLLM
- **结构化生成**：原生支持 JSON Schema，自动验证格式
- **函数调用**：原生支持函数调用，自动解析参数
- **批量推理**：RadixAttention 自动共享公共前缀，大幅提升性能
- **优势**：结构化生成能力强，复杂提示性能优秀
- **劣势**：相对较新，文档和示例较少

6.5 性能对比代码示例

吞吐量测试：

```
import time
import torch
from vllm import LLM, SamplingParams
import sglang as sgl

# 测试数据
prompts = ["The future of AI is"] * 100

# 1. PyTorch 测试
def test_pytorch():
    model = GPT2LMHeadModel.from_pretrained('gpt2')
    model.eval()

    start = time.time()
    for prompt in prompts:
        inputs = tokenizer(prompt, return_tensors='pt')
        with torch.no_grad():
            outputs = model.generate(inputs.input_ids, max_length=50)
    pytorch_time = time.time() - start

    return pytorch_time

# 2. vLLM 测试
def test_vllm():
    llm = LLM(model="meta-llama/Llama-2-7b-hf")
    sampling_params = SamplingParams(max_tokens=50)
```

```
    start = time.time()
    llm.generate(prompts, sampling_params)
    vllm_time = time.time() - start

    return vllm_time

# 3. SGLang 测试
def test_sglang():
    runtime = sgl.Runtime(model_path="meta-llama/Llama-2-7b-hf")

    start = time.time()
    runtime.generate_batch(prompts, max_new_tokens=50)
    sglang_time = time.time() - start

    return sglang_time

# 运行测试
pytorch_time = test_pytorch()
vllm_time = test_vllm()
sglang_time = test_sglang()

print(f"PyTorch: {pytorch_time:.2f}s")
print(f"vLLM: {vllm_time:.2f}s ({pytorch_time/vllm_time:.1f}x faster)")
print(f"SGLang: {sglang_time:.2f}s ({pytorch_time/sglang_time:.1f}x
   faster)")
```

Listing 20: 性能对比测试示例

6.6 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：

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

Flash Attention 分块计算：

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

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

内存优化：

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

```python
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 21: Flash Attention 使用示例

6.7 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：

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

GPTQ 量化：

GPTQ（GPT Quantization）是一种后训练量化方法：

$$\arg\min_{\hat{\mathbf{W}}} ||\mathbf{WX} - \hat{\mathbf{W}}\mathbf{X}||_2^2 \tag{12}$$

```python
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 22: GPTQ 量化示例

6.8 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：

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

流水线并行：将模型按层分割到不同 GPU。

7 模型部署技术

7.1 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：

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

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

7.2 服务化部署

API 服务：使用 FastAPI、Flask 等框架部署模型服务。

```python
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 23: 模型服务化部署示例

7.3 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。

8 推理优化技术

8.1 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

8.2 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

8.3 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

9 检索增强生成（RAG）技术详解

检索增强生成（Retrieval-Augmented Generation, RAG）是大语言模型应用中的核心技术，通过结合检索和生成，显著提升模型在知识密集型任务上的表现。

9.1 RAG 核心原理

概念解释：RAG 将信息检索与文本生成相结合，在生成答案前先从外部知识库检索相关信息，然后将检索到的信息作为上下文输入给生成模型。

数学表示：

RAG 的生成过程可以表示为：

$$P(y|x) = \sum_{z \in \text{Top-K}(x)} P(z|x) \cdot P(y|x,z) \tag{15}$$

其中：

- x 是用户查询
- z 是从知识库检索到的文档
- Top-K(x) 表示检索到的 Top-K 相关文档
- $P(z|x)$ 是检索模型给出的文档相关性分数
- $P(y|x,z)$ 是生成模型基于查询和检索文档生成答案的概率

RAG 的优势：

- **知识更新**：无需重新训练模型即可更新知识
- **可解释性**：可以追溯到生成答案的来源文档
- **减少幻觉**：基于检索到的真实文档生成，降低编造信息的风险
- **领域适应**：可以快速适应特定领域的知识库

9.2 RAG 系统架构

两阶段检索架构：

1. **召回层（Recall Layer）**：

- 使用快速检索算法（如 BM25、KNN）从大规模知识库中召回候选文档
- 目标：高召回率，不遗漏相关文档
- 常用方法：BM25（关键词匹配）、向量检索（语义相似度）

2. **排序层（Ranking Layer）**：

- 使用精细排序模型（如 Cross-Encoder）对召回文档进行重排序
- 目标：高精确度，确保最相关的文档排在前面
- 常用方法：Cross-Encoder、LambdaMART

混合检索策略：

结合关键词检索和语义检索：

$$\text{Score}(q, d) = \alpha \cdot \text{BM25}(q, d) + (1 - \alpha) \cdot \text{Sim}(E(q), E(d)) \tag{16}$$

其中：

- $\text{BM25}(q, d)$ 是 BM25 关键词匹配分数
- $\text{Sim}(E(q), E(d))$ 是查询和文档嵌入向量的相似度
- α 是混合权重，通常设置为 0.3-0.7

9.3 查询重写（Query Rewrite）

概念解释：用户查询往往简短且不完整，查询重写模型将简短查询扩展为完整的语义查询，提高检索准确性。

技术实现：

使用 T5 等序列到序列模型进行查询重写：

$$q' = \text{T5}(q, \text{context}) \tag{17}$$

应用场景：

- **查询扩展**：将” 退改政策” 扩展为” 酒店退改政策、退改时间限制、退改费用”
- **同义词替换**：将” 取消” 替换为” 退订”、” 退款” 等
- **意图理解**：理解用户真实意图，如” 改签” 实际需要查询” 变更政策”

效果提升：查询重写可以提升检索相关性 30-40%，显著改善 RAG 系统的整体表现。

9.4 重排序（Rerank）技术

Cross-Encoder 重排序：

Cross-Encoder 将查询和文档拼接后输入模型，进行深度交互建模：

$$\text{Score}(q, d) = \text{CrossEncoder}([\text{CLS}]q[\text{SEP}]d) \tag{18}$$

优势：

- **深度交互**：查询和文档在模型内部充分交互，理解更准确
- **高精确度**：相比 Bi-Encoder，精确度提升 20-40%
- **解决误召回**：有效解决” 相似政策误召回” 问题

LambdaMART 融合排序：

LambdaMART 是学习排序（Learning to Rank）算法，用于优化多个排序信号的融合权重：

$$\text{FinalScore} = \sum_{i=1}^{n} w_i \cdot \text{Score}_i(q, d) \tag{19}$$

其中 Score_i 可以是：

- BM25 分数
- 向量相似度分数
- Cross-Encoder 分数
- 其他特征（文档长度、时间戳等）

10 对比学习与负样本挖掘

10.1 对比学习（Contrastive Learning）

概念解释：对比学习通过拉近相似样本、推远不相似样本来学习表示，是训练高质量嵌入模型的核心技术。

数学原理：

对比学习的损失函数（InfoNCE）：

$$\mathcal{L}_{\text{contrastive}} = -\log \frac{\exp(\text{sim}(q, d^+)/\tau)}{\exp(\text{sim}(q, d^+)/\tau) + \sum_{d^-} \exp(\text{sim}(q, d^-)/\tau)} \tag{20}$$

其中：

- q 是查询向量
- d^+ 是正样本（相关文档）
- d^- 是负样本（不相关文档）
- τ 是温度参数，控制分布的尖锐程度
- $\text{sim}(\cdot, \cdot)$ 是相似度函数（如余弦相似度）

在嵌入模型微调中的应用：

对于 RoBERTa 等预训练模型，使用对比学习进行领域适应：

1. **正样本构建**：查询和其对应的相关文档
2. **负样本构建**：查询和随机采样的不相关文档
3. **训练目标**：最大化正样本相似度，最小化负样本相似度

效果：对比学习可以提升业务意图识别准确率 20-30%，提升长尾查询匹配率 30-40%。

10.2 困难负样本挖掘（Hard Negative Mining）

概念解释：困难负样本是与查询相似但不相关的样本，挖掘这些样本可以显著提升模型的学习效果。

困难负样本识别：

$$d_{\text{hard}} = \arg\max_{d \in \mathcal{D}_{\text{neg}}} \text{sim}(E(q), E(d)) \tag{21}$$

即选择与查询最相似但不相关的文档作为困难负样本。

挖掘策略：

1. **在线挖掘**：在训练过程中动态选择困难负样本
2. **离线挖掘**：使用当前模型在数据集中挖掘困难负样本，然后用于下一轮训练

3. **混合策略**：结合随机负样本和困难负样本

效果提升：困难负样本挖掘可以提升召回率 20-30%，使模型更好地区分相似但不相关的文档。

11 知识蒸馏（Knowledge Distillation）

11.1 知识蒸馏原理

概念解释：知识蒸馏是一种模型压缩技术，通过让小型学生模型学习大型教师模型的知识，在保持性能的同时大幅降低模型大小和推理成本。

数学表示：

知识蒸馏的损失函数：

$$\mathcal{L}_{\text{KD}} = \alpha \cdot \mathcal{L}_{\text{CE}}(y, \text{softmax}(z_s)) + (1-\alpha) \cdot \mathcal{L}_{\text{KL}}(\text{softmax}(z_t/T), \text{softmax}(z_s/T)) \tag{22}$$

其中：

- z_t 是教师模型的 logits
- z_s 是学生模型的 logits
- T 是温度参数（通常 $T = 3 - 5$），温度越高，分布越平滑
- α 是平衡系数（通常 $\alpha = 0.3 - 0.5$）
- $\mathcal{L}_{\text{CE}}$ 是交叉熵损失
- $\mathcal{L}_{\text{KL}}$ 是 KL 散度损失

温度参数的作用：

温度参数 T 用于软化概率分布：

$$p_i = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)} \tag{23}$$

温度越高，分布越平滑，学生模型可以学习到教师模型的” 软标签” 中包含的丰富信息。

11.2 在 RAG 系统中的应用

轻量级模型部署：

- **高频查询处理**：将 50% 的高频简单查询切换到本地轻量级模型
- **成本降低**：减少 API 调用成本 80-85%
- **延迟降低**：本地推理延迟更低，用户体验更好

蒸馏策略：

1. **任务特定蒸馏**：针对特定任务（如意图分类）训练小型模型
2. **多任务蒸馏**：将多个任务的知识蒸馏到单一小型模型
3. **渐进式蒸馏**：逐步减小模型大小，保持性能

12 大语言模型中的强化学习

12.1 大模型 RL 概述

严谨解释：

大语言模型中的强化学习（RL for LLM）是一种通过与环境交互来优化语言模型生成策略的方法。在数学上，这可以形式化为一个部分可观测马尔可夫决策过程（POMDP）：

$$\mathcal{M} = (\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma) \tag{24}$$

其中：

- $\mathcal{S}$ 是状态空间，表示当前对话历史和上下文
- $\mathcal{A}$ 是动作空间，表示所有可能的 token 序列
- $\mathcal{P} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ 是状态转移概率，由语言模型决定
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ 是奖励函数，评估生成质量
- $\gamma \in [0, 1]$ 是折扣因子

目标是最优策略 π^*，最大化期望累积奖励：

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\tau\sim\pi}\left[\sum_{t=0}^{T}\gamma^t r_t\right] \tag{25}$$

通俗解释：

想象你在训练一个 AI 助手，它需要学会说” 好话”（符合人类偏好）。传统方法是给它看很多例子让它模仿，但强化学习就像给它一个” 评分系统”：

- AI 生成一段话
- 人类（或奖励模型）给它打分
- AI 根据分数调整策略，下次生成更好的话
- 反复这个过程，AI 越来越会说” 好话”

这就像训练宠物：做对了给奖励，做错了不给奖励，久而久之它就知道什么行为是好的。

12.2 大模型 RL 的应用场景

1. 人类反馈强化学习（RLHF）

严谨解释：RLHF 通过人类标注的偏好数据训练奖励模型，然后使用强化学习优化生成策略，使模型输出符合人类价值观和偏好。

通俗解释：让 AI 学会” 投其所好”。通过人类打分，AI 学会生成人类喜欢的内容，比如更安全、更有帮助、更符合事实的回答。

应用场景：

- ChatGPT、Claude 等对话模型的训练
- 内容安全对齐，减少有害内容生成
- 风格对齐，使输出符合特定风格

2. 检索策略优化

严谨解释：在 RAG 系统中，使用强化学习优化检索策略，使检索到的文档能够生成更高质量的回复。

通俗解释：教 AI” 找资料” 的技巧。通过强化学习，AI 学会从知识库中检索最相关的文档，从而生成更好的答案。

应用场景：

- 智能客服系统
- 知识问答系统
- 文档摘要生成

3. 代码生成优化

严谨解释：使用强化学习优化代码生成策略，使生成的代码更符合编程规范、更易读、更高效。

通俗解释：教 AI 写" 好代码"。通过强化学习，AI 学会生成不仅功能正确，而且风格优雅、效率高的代码。

应用场景：

- GitHub Copilot 等代码助手
- 代码重构工具
- 代码审查助手

12.3 大模型 RL 与传统 RL 的对比

特性	传统 RL	大模型 RL
状态空间	离散/连续（如游戏状态）	高维离散（token 序列）
动作空间	有限动作集合	巨大动作空间（词汇表大小）
奖励信号	环境直接给出	需要人类或奖励模型
样本效率	通常较低	极低（需要大量交互）
探索策略	ϵ-贪婪、UCB 等	采样策略、温度参数
训练方式	在线学习	离线 + 在线混合
主要挑战	探索-利用平衡	奖励设计、稳定性

表 3: 传统 RL 与大模型 RL 对比

关键区别详解：

1. 动作空间规模

严谨解释：

- **传统 RL**：动作空间通常是有限的，如 Atari 游戏有 4-18 个动作
- **大模型 RL**：动作空间是词汇表大小，通常 $|\mathcal{V}| = 30,000 - 100,000$

通俗解释:

- **传统 RL**：就像在一个有 10 个按钮的控制面板上操作
- **大模型 RL**：就像在一个有 5 万个按钮的控制面板上操作，每个按钮代表一个词

2. 奖励信号

严谨解释:

- **传统 RL**：环境直接提供奖励，如游戏得分、到达目标
- **大模型 RL**：需要人类标注或训练奖励模型 $r_\phi(x, y)$ 来评估生成质量

通俗解释:

- **传统 RL**：游戏自动告诉你得分，很明确
- **大模型 RL**：需要人工判断” 这段话好不好”，主观性强

3. 训练稳定性

严谨解释:

- **传统 RL**：策略更新通常较稳定，因为动作空间小
- **大模型 RL**：策略更新容易导致模型” 崩溃”（生成无意义文本），需要 KL 散度约束

通俗解释:

- **传统 RL**：调整策略后，机器人可能走偏一点，但还能走
- **大模型 RL**：调整策略后，AI 可能突然” 胡言乱语”，需要小心约束

4. 样本效率

严谨解释:

- **传统 RL**：可能需要 $10^6 - 10^8$ 个样本
- **大模型 RL**：可能需要 $10^9 - 10^{12}$ 个样本，但通过预训练可以大幅减少

通俗解释:

- **传统 RL**：需要大量试错，但相对可控
- **大模型 RL**：需要海量数据，但预训练模型提供了很好的起点

13 PPO、GRPO、DPO 详解

13.1 PPO（Proximal Policy Optimization）

严谨解释：

PPO 是一种策略梯度算法，通过限制策略更新的幅度来保证训练稳定性。PPO 的目标函数为：

$$\mathcal{L}_{\text{PPO}}(\theta) = \mathbb{E}_{(s_t,a_t)\sim\pi_{\theta_{\text{old}}}}\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t\right)\right] \tag{26}$$

其中：

- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ 是重要性采样比率
- $\hat{A}_t$ 是优势函数估计（通常使用 GAE）
- ϵ 是裁剪参数（通常 $\epsilon = 0.1 - 0.2$）
- $\text{clip}(x, a, b) = \max(a, \min(x, b))$ 是裁剪函数

通俗解释：

PPO 就像给 AI 一个” 安全绳”：

- 传统策略梯度：AI 可能” 大步前进”，容易” 摔倒”（策略崩溃）
- PPO：限制 AI 每次只能” 小步前进”，即使走错也不会太远
- 裁剪机制：如果 AI 想” 走太远”，就把它” 拉回来”

这就像学骑自行车时，有人在后面扶着，防止你摔倒。

在 RLHF 中的应用：

1. **训练奖励模型**：使用人类偏好数据训练 $r_\phi(x, y)$
2. **优化生成策略**：使用 PPO 优化 π_θ，最大化奖励
3. **KL 散度约束**：防止策略偏离参考模型太远

PPO 的优势：

- **稳定性**：通过裁剪保证训练稳定

- **效率**：可以多次使用同一批数据
- **简单**：实现相对简单，超参数少

PPO 的劣势：

- **需要奖励模型**：必须先训练奖励模型
- **样本效率**：相比 DPO，样本效率较低
- **超参数敏感**：ϵ 的选择影响性能

13.2 GRPO（Group Relative Policy Optimization）

严谨解释：

GRPO 是一种无需奖励模型的策略优化方法，通过组内相对比较来优化策略。GRPO 的损失函数为：

$$\mathcal{L}_{\text{GRPO}}(\theta) = -\mathbb{E}_{(x,\{y_i\}_{i=1}^n)\sim\mathcal{D}}\left[\log \frac{\exp(\beta \log \pi_\theta(y_{\text{best}}|x))}{\sum_{i=1}^n \exp(\beta \log \pi_\theta(y_i|x))}\right] \tag{27}$$

其中：

- $\{y_i\}_{i=1}^n$ 是一组候选回复
- y_{best} 是组内最优回复（由人类标注）
- β 是温度参数

通俗解释：

GRPO 就像” 选美比赛”：

- 给 AI 看一组候选回复（比如 5 个）
- 人类选出最好的一个
- AI 学习” 为什么这个最好”，调整策略生成类似的回复

这比” 打分” 更简单：不需要精确的分数，只需要知道” 哪个更好”。

GRPO 的优势：

- **无需奖励模型**：直接使用人类偏好

- **相对简单**：实现比 DPO 更简单
- **组内比较**：可以同时比较多个候选

GRPO 的劣势：

- **需要多个候选**：每个样本需要生成多个候选回复
- **计算成本**：生成多个候选增加计算量
- **性能**：通常不如 DPO

13.3 DPO（Direct Preference Optimization）

严谨解释：

DPO 是一种无需奖励模型的偏好对齐方法，通过直接优化策略来符合人类偏好。DPO 的核心思想是将奖励模型隐式地嵌入到策略优化中。

DPO 的损失函数：

$$\mathcal{L}_{\text{DPO}}(\theta) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}\left[\log\sigma\left(\beta\log\frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta\log\frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)}\right)\right] \tag{28}$$

其中：

- (x, y_w, y_l) 是偏好数据，y_w 是偏好回复，y_l 是非偏好回复
- π_θ 是待优化的策略
- π_{ref} 是参考策略（通常是 SFT 后的模型）
- β 是温度参数，控制优化强度
- σ 是 sigmoid 函数

为什么 DPO 无需奖励模型？数学解释：

关键洞察：DPO 通过数学变换，将奖励模型的优化问题转化为策略优化问题。

步骤 1：奖励模型的最优形式

在 RLHF 中，给定奖励模型 $r_\phi(x, y)$，最优策略为：

$$\pi^*(y|x) = \frac{1}{Z(x)}\pi_{\text{ref}}(y|x)\exp\left(\frac{1}{\beta}r_\phi(x,y)\right) \tag{29}$$

其中 $Z(x) = \sum_y \pi_{\text{ref}}(y|x) \exp(\frac{1}{\beta} r_\phi(x, y))$ 是归一化常数。

步骤 2：奖励函数的隐式表示

从最优策略可以反推出奖励函数：

$$r_\phi(x, y) = \beta \log \frac{\pi^*(y|x)}{\pi_{\text{ref}}(y|x)} + \beta \log Z(x) \tag{30}$$

注意：$Z(x)$ 只依赖于 x，在比较两个回复 y_w 和 y_l 时会抵消。

步骤 3：偏好概率建模

使用 Bradley-Terry 模型建模人类偏好：

$$P(y_w \succ y_l|x) = \frac{\exp(r_\phi(x, y_w))}{\exp(r_\phi(x, y_w)) + \exp(r_\phi(x, y_l))} = \sigma(r_\phi(x, y_w) - r_\phi(x, y_l)) \tag{31}$$

步骤 4：替换奖励函数

将步骤 2 的奖励函数代入步骤 3：

$$P(y_w \succ y_l|x) = \sigma\left(\beta \log \frac{\pi^*(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta \log \frac{\pi^*(y_l|x)}{\pi_{\text{ref}}(y_l|x)}\right) \tag{32}$$

$$= \sigma\left(\beta \log \frac{\pi^*(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta \log \frac{\pi^*(y_l|x)}{\pi_{\text{ref}}(y_l|x)}\right) \tag{33}$$

步骤 5：直接优化策略

最大化对数似然：

$$\max_\theta \mathbb{E}_{(x, y_w, y_l)} \left[\log \sigma\left(\beta \log \frac{\pi_\theta(y_w|x)}{\pi_{\text{ref}}(y_w|x)} - \beta \log \frac{\pi_\theta(y_l|x)}{\pi_{\text{ref}}(y_l|x)}\right)\right] \tag{34}$$

这就是 DPO 的损失函数（取负号后最小化）。

关键点：

- **奖励模型被隐式表示**：通过策略比率 $\frac{\pi_\theta(y|x)}{\pi_{\text{ref}}(y|x)}$ 隐式表示奖励
- **归一化常数抵消**：$Z(x)$ 在比较时抵消，不需要显式计算
- **直接优化策略**：不需要先训练奖励模型，直接优化策略即可

通俗解释：

传统 RLHF 需要两步：

1. 训练奖励模型：教 AI" 什么是好，什么是坏"（打分系统）
2. 优化策略：让 AI 生成高分内容

DPO 只需要一步：

1. 直接优化策略：让 AI 生成" 人类更喜欢" 的内容

为什么可以这样?

想象你在学做菜：

- **传统方法**：先学" 评分标准"（什么菜好吃），再学" 怎么做高分菜"
- **DPO 方法**：直接学" 怎么做人类更喜欢的菜"，不需要明确的评分标准

DPO 的" 魔法" 在于：通过比较两个回复，AI 自动学会了" 什么是好"，不需要明确的分数。

DPO 的优势：

- **无需奖励模型**：减少训练步骤和计算成本
- **训练稳定**：相比 RLHF，训练更稳定
- **计算高效**：只需要一次前向传播
- **效果优秀**：在风格对齐任务上效果显著

DPO 的劣势：

- **需要参考模型**：需要 SFT 后的参考模型
- **偏好数据质量**：对偏好数据质量要求高
- **难以处理复杂奖励**：对于需要多维度评估的任务，不如显式奖励模型灵活

13.4 PPO、GRPO、DPO 对比

14 强化学习在 RAG 中的应用

14.1 REINFORCE 算法优化检索策略

概念解释：REINFORCE 是一种策略梯度算法，用于优化 RAG 系统中的检索策略，使生成的回复更符合业务标准。

特性	PPO	GRPO	DPO
需要奖励模型	是	否	否
训练步骤	2 步（奖励模型 + 策略优化）	1 步	1 步
样本效率	中等	较低	较高
训练稳定性	中等	较高	高
计算成本	高	中等	低
适用场景	复杂奖励、多维度评估	组内比较	风格对齐、偏好学习

表 4: PPO、GRPO、DPO 对比

数学原理:

REINFORCE 的策略梯度:

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau\sim\pi_\theta}\left[\sum_{t=0}^{T}\nabla_\theta \log \pi_\theta(a_t|s_t)\cdot R(\tau)\right] \tag{35}$$

其中:

- $\pi_\theta(a_t|s_t)$ 是在状态 s_t 下选择动作 a_t 的策略
- $R(\tau)$ 是轨迹 τ 的累积奖励
- θ 是策略参数

在 RAG 中的应用:

1. **状态**：当前查询和已检索的文档
2. **动作**：选择检索哪些文档
3. **奖励**：生成回复的质量（如 ROUGE-L 分数、人工评估分数）
4. **目标**：最大化生成回复的质量

效果：REINFORCE 优化可以提升 ROUGE-L 分数 25-30%，使回复更符合客服黄金标准。

15 噪声嵌入训练（NEFTune）

15.1 NEFTune 原理

概念解释：NEFTune（Noise Embeddings for Fine-Tuning）通过在嵌入层添加噪声来提升模型在小样本上的泛化能力，解决过拟合导致的机械重复问题。

数学表示：

NEFTune 在嵌入层添加噪声：

$$\tilde{\mathbf{e}}_i = \mathbf{e}_i + \mathbf{n}_i, \quad \mathbf{n}_i \sim \mathcal{N}(0, \sigma^2 \mathbf{I}) \tag{36}$$

其中：

- $\mathbf{e}_i$ 是原始嵌入向量
- $\tilde{\mathbf{e}}_i$ 是添加噪声后的嵌入向量
- σ 是噪声强度（通常 $\sigma = 0.1 - 0.2$）

作用机制：

- **正则化效果**：噪声起到正则化作用，防止过拟合
- **提升泛化**：增强模型在未见数据上的表现
- **减少重复**：有效解决小样本微调中的机械重复问题

应用效果：

在 LoRA 微调中，NEFTune 可以：

- 提升生成多样性 15-25%
- 减少重复生成 30-40%
- 提升下游任务性能 5-10%

16 检索算法详解

16.1 BM25 算法

概念解释：BM25（Best Matching 25）是信息检索中最经典的关键词匹配算法，基于词频和逆文档频率计算文档相关性。

数学公式：

BM25 分数计算：

$$\text{BM25}(q,d) = \sum_{t \in q} \text{IDF}(t) \cdot \frac{f(t,d) \cdot (k_1 + 1)}{f(t,d) + k_1 \cdot (1 - b + b \cdot \frac{|d|}{\text{avgdl}})} \tag{37}$$

其中：

- $f(t,d)$ 是词项 t 在文档 d 中的词频
- $|d|$ 是文档长度
- avgdl 是平均文档长度
- $\text{IDF}(t) = \log \frac{N - n(t) + 0.5}{n(t) + 0.5}$，$N$ 是总文档数，$n(t)$ 是包含词项 t 的文档数
- k_1 和 b 是超参数（通常 $k_1 = 1.2, b = 0.75$）

优势：

- **快速**：计算效率高，适合大规模检索
- **有效**：在关键词匹配任务上表现优秀
- **可解释**：分数计算过程透明

16.2 KNN 向量检索

概念解释：KNN（K-Nearest Neighbors）向量检索通过计算查询向量和文档向量的相似度，找到最相似的 K 个文档。

相似度计算：

常用余弦相似度：

$$\text{sim}(q,d) = \frac{\mathbf{q} \cdot \mathbf{d}}{||\mathbf{q}|| \cdot ||\mathbf{d}||} = \cos(\theta) \tag{38}$$

加速方法：

- **HNSW**：分层导航小世界图，近似最近邻搜索
- **IVF**：倒排文件索引，快速检索
- **PQ**：乘积量化，压缩向量

16.3 HNSW 算法

概念解释：HNSW（Hierarchical Navigable Small World）是一种高效的近似最近邻搜索算法，通过构建多层图结构实现快速检索。

算法特点：

- **多层结构**：构建多个层次的图，上层节点少，下层节点多
- **快速搜索**：从上层开始搜索，逐步向下层细化
- **高精度**：在保持高检索精度的同时，大幅提升检索速度

复杂度：

- **搜索复杂度**：$O(\log N)$，其中 N 是文档数量
- **空间复杂度**：$O(N \cdot M)$，其中 M 是平均连接数

16.4 SimHash 和 MinHash

SimHash：

SimHash 用于快速计算文本相似度，生成固定长度的哈希值：

$$\text{SimHash}(d) = \text{sign}(\sum_{t \in d} w(t) \cdot \mathbf{h}(t)) \tag{39}$$

其中 $\mathbf{h}(t)$ 是词项 t 的随机哈希向量。

应用：

- **去重**：快速识别相似文档
- **聚类**：基于哈希值进行快速聚类

MinHash：

MinHash 用于估计集合的 Jaccard 相似度：

$$\text{Jaccard}(A, B) \approx \frac{1}{k} \sum_{i=1}^{k} \mathbf{1}[\min(A_i) = \min(B_i)] \tag{40}$$

17 聚类算法在 NLP 中的应用

17.1 K-Means 聚类

概念解释：K-Means 是一种经典的聚类算法，将数据点分为 K 个簇，使得簇内距离最小、簇间距离最大。

算法流程：

1. 随机初始化 K 个聚类中心
2. 将每个数据点分配到最近的聚类中心
3. 更新聚类中心为簇内点的均值
4. 重复步骤 2-3 直到收敛

目标函数：

$$J = \sum_{i=1}^{K} \sum_{\mathbf{x} \in C_i} ||\mathbf{x} - \boldsymbol{\mu}_i||^2 \tag{41}$$

其中 C_i 是第 i 个簇，$\boldsymbol{\mu}_i$ 是簇中心。

在 NLP 中的应用：

- **文档聚类**：将相似文档分组
- **主题发现**：发现文档中的主题
- **数据预处理**：在 RAG 系统中对文档进行预处理

17.2 DBSCAN 聚类

概念解释：DBSCAN（Density-Based Spatial Clustering of Applications with Noise）是一种基于密度的聚类算法，可以发现任意形状的簇并识别噪声点。

核心概念：

- **核心点**：邻域内至少有 minPts 个点的点
- **边界点**：在核心点的邻域内但不是核心点的点

- **噪声点**：既不是核心点也不是边界点的点

优势：

- **无需预设簇数**：自动发现簇的数量
- **发现任意形状**：可以发现非球形的簇
- **识别噪声**：自动识别异常点

18 主成分分析（PCA）

18.1 PCA 原理

概念解释：PCA（Principal Component Analysis）是一种降维技术，通过找到数据的主要变化方向，将高维数据投影到低维空间。

数学原理：

PCA 的目标是找到投影方向 $\mathbf{w}$，使得投影后的方差最大：

$$\max_{\mathbf{w}} \mathbf{w}^T \mathbf{C} \mathbf{w}, \quad \text{s.t.} \quad ||\mathbf{w}|| = 1 \tag{42}$$

其中 $\mathbf{C}$ 是协方差矩阵。

求解方法：

通过特征值分解：

$$\mathbf{C} = \mathbf{U} \mathbf{\Lambda} \mathbf{U}^T \tag{43}$$

主成分就是协方差矩阵的特征向量，按特征值大小排序。

在 NLP 中的应用：

- **特征降维**：降低词向量维度
- **可视化**：将高维数据可视化到 2D/3D
- **噪声去除**：去除数据中的噪声成分

19 流行大语言模型介绍

19.1 Qwen 系列

Qwen2.5-1.5B：

- **发布机构**：阿里巴巴通义千问团队
- **参数量**：1.5B（15 亿参数）
- **特点**：
  - 轻量级模型，适合本地部署
  - 支持多语言（中文、英文等）
  - 开源可商用
  - 在对话、代码生成等任务上表现优秀
- **应用场景**：
  - 个人数字分身
  - 边缘设备部署
  - 低成本微调实验

Qwen2.5 系列其他模型：

- **Qwen2.5-7B**：中等规模，平衡性能和效率
- **Qwen2.5-14B**：大规模模型，性能更强
- **Qwen2.5-72B**：超大规模模型，接近 GPT-4 性能

19.2 LLaMA 系列

LLaMA 3：

- **发布机构**：Meta（Facebook）
- **参数量**：8B, 70B, 405B
- **特点**：

  - 开源可商用
  - 训练数据质量高
  - 指令遵循能力强
  - 代码能力优秀

LLaMA 2:

- **特点**:
  - 首个开源可商用的大语言模型
  - 支持对话和代码生成
  - 安全性对齐

19.3 GPT 系列

GPT-4:

- **发布机构**: OpenAI
- **特点**:
  - 多模态能力（文本、图像）
  - 强大的推理能力
  - 代码生成能力优秀
  - 闭源，通过 API 使用

GPT-3.5:

- **特点**:
  - 175B 参数
  - 强大的少样本学习能力
  - 广泛的应用生态

19.4 Claude 系列

Claude 3：

- **发布机构**：Anthropic
- **模型**：Opus, Sonnet, Haiku
- **特点**：
  - 安全性强
  - 长上下文支持（200K tokens）
  - 推理能力优秀

19.5 GLM 系列

GLM-4：

- **发布机构**：智谱 AI
- **特点**：
  - 中文能力优秀
  - 多模态支持
  - 开源版本可用

19.6 Mistral 系列

Mistral 7B：

- **发布机构**：Mistral AI
- **特点**：
  - 7B 参数，性能优秀
  - 开源可商用
  - 推理效率高

20 预训练模型架构详解

20.1 RoBERTa

概念解释：RoBERTa（Robustly Optimized BERT Pretraining Approach）是 BERT 的优化版本，通过改进训练策略显著提升了性能。

改进点：

- **动态掩码**：每次训练时动态生成掩码，而不是静态掩码
- **移除 NSP 任务**：去除了下一句预测任务
- **更大批次**：使用更大的批次大小（8K）
- **更长训练**：训练更多步数
- **更多数据**：使用更多训练数据

应用场景：

- **文本分类**：情感分析、意图识别
- **文本匹配**：相似度计算、检索
- **命名实体识别**：NER 任务
- **嵌入模型**：作为嵌入模型的基础

微调策略：

使用对比学习和困难负样本挖掘进行领域适应：

- 提升业务意图识别准确率 20%
- 提升长尾查询匹配率 30%
- 提升召回率 25%

20.2 T5

概念解释：T5（Text-To-Text Transfer Transformer）将所有 NLP 任务统一为文本到文本的生成任务。

架构特点：

- **编码器-解码器架构**：使用 Transformer 的编码器-解码器结构
- **统一框架**：所有任务都转换为” 输入文本 → 输出文本”
- **任务前缀**：通过任务前缀区分不同任务（如”translate:”, ”summarize:”）

应用场景：

- **文本生成**：摘要、翻译、改写
- **查询重写**：将简短查询扩展为完整查询
- **文本转换**：格式转换、风格转换

微调效果：

在查询重写任务上：

- 提升重写查询与原始意图的相关性 35%
- 有效提升召回准确率

20.3 Cross-Encoder

概念解释：Cross-Encoder 将查询和文档拼接后输入模型，进行深度交互建模，用于精细排序。

架构：

$$\text{Score}(q, d) = \text{Linear}(\text{CLS}(\text{Transformer}([\text{CLS}]q[\text{SEP}]d))) \tag{44}$$

优势：

- **深度交互**：查询和文档在模型内部充分交互
- **高精确度**：比 Bi-Encoder 精确度提升 20-40%

- **解决误召回**：有效解决” 相似政策误召回” 问题

应用：

在 RAG 系统的排序层：

- Top K 数据精确度提升 40%
- P@1 精确度提升 37%

21 框架与工具

21.1 LangChain

概念解释：LangChain 是一个用于构建基于大语言模型应用的框架，提供了 RAG、Agent 等高级功能的实现。

核心组件：

- **LLMs**：大语言模型接口
- **Chains**：链式调用，组合多个组件
- **Agents**：智能代理，可以调用工具
- **Memory**：记忆管理
- **Vector Stores**：向量数据库集成

RAG 实现：

```
from langchain.llms import OpenAI
from langchain.vectorstores import FAISS
from langchain.chains import RetrievalQA

# 创建向量存储
vectorstore = FAISS.from_documents(documents, embeddings)

# 创建 RAG 链
qa_chain = RetrievalQA.from_chain_type(
    llm=OpenAI(),
    chain_type="stuff",
```

```
    retriever=vectorstore.as_retriever()
)

# 查询
result = qa_chain.run("What is RAG?")
```

Listing 24: LangChain RAG 示例

应用场景：

- **RAG 应用**：快速构建检索增强生成系统
- **智能代理**：构建可以调用工具的 AI 代理
- **对话系统**：构建多轮对话系统

21.2 DeepSpeed

概念解释：DeepSpeed 是微软开发的深度学习优化库，提供分布式训练、内存优化、模型压缩等功能。

核心功能：

- **ZeRO**：零冗余优化器，大幅降低内存占用
- **梯度压缩**：压缩梯度，减少通信开销
- **混合精度训练**：FP16/BF16 训练，加速训练
- **模型并行**：支持模型并行和数据并行

ZeRO 优化：

- **ZeRO-1**：优化器状态分片
- **ZeRO-2**：优化器状态 + 梯度分片
- **ZeRO-3**：优化器状态 + 梯度 + 参数分片

效果：

- 内存占用降低 4-8 倍
- 支持训练更大规模的模型
- 训练速度提升 2-4 倍

22 上下文学习（In-Context Learning, ICL）

22.1 ICL 原理

概念解释：In-Context Learning 是大语言模型的核心能力之一，模型通过观察少量示例（few-shot examples）就能理解任务并执行，无需参数更新。

数学表示：

给定任务示例 $(x_1, y_1), (x_2, y_2), \ldots, (x_k, y_k)$ 和查询 x_{k+1}，模型预测：

$$P(y_{k+1}|x_1, y_1, \ldots, x_k, y_k, x_{k+1}) = \prod_{i=1}^{|y_{k+1}|} P(y_{k+1}^{(i)}|x_1, y_1, \ldots, x_k, y_k, x_{k+1}, y_{k+1}^{(<i)}) \tag{45}$$

ICL 的特点：

- **无需训练**：不需要梯度更新，直接推理
- **快速适应**：通过示例快速理解任务
- **灵活性强**：可以处理各种任务

22.2 ICL 的应用场景

角色认知增强：

在个人数字分身等应用中，使用 ICL 增强模型的角色认知：

```
prompt = """
你是一个数字分身，具有以下特点：
- 语言风格：简洁、直接
- 常用表达："好的"、"没问题"、"明白了"
- 回复习惯：先确认理解，再给出建议

示例对话：
用户：今天天气怎么样？
分身：好的，今天天气晴朗，温度25度，适合外出。

用户：帮我安排一下明天的行程
分身：明白了，我来帮你安排明天的行程...
"""
```

Listing 25: ICL 角色认知示例

动态适应:

通过添加最近的对话记录作为 Context，实现动态适应:

$$\text{Context} = [\text{历史对话}_1, \text{历史对话}_2, \ldots, \text{最近对话}_k] \tag{46}$$

这样模型可以:

- 理解对话上下文
- 保持对话连贯性
- 适应最新的对话风格

23 长上下文管理

23.1 基于轮次衰减的上下文管理

概念解释：在长对话中，需要管理上下文长度，保留重要信息，压缩或删除不重要信息。

重要性评分公式:

$$S_i = e^{-\lambda \Delta t_i} \tag{47}$$

其中:

- S_i 是第 i 轮对话的重要性分数
- Δt_i 是当前轮次与第 i 轮的距离（轮次数）
- λ 是衰减系数（通常 $\lambda = 0.2$）

上下文策略:

1. **保留高重要性对话**：保留最近 K 轮高重要性分数的对话
2. **语义摘要**：对低重要性分数的旧对话进行语义摘要
3. **混合输入**：构造” 摘要 + 最近上下文” 的混合输入

效果:

- 有效解决长对话中的遗忘问题
- 提升回复连贯性 25-30%
- 减少上下文长度 40-60%

24 Agent、Function Calling、Tools 与 MCP

24.1 Agent（智能代理）

严谨解释:

Agent 是一个能够感知环境、做出决策并执行动作的自主系统。在大语言模型应用中，Agent 通常指能够使用工具、与环境交互、完成复杂任务的智能系统。

数学表示:

Agent 可以形式化为一个元组:

$$\text{Agent} = (\mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \pi) \tag{48}$$

其中:

- $\mathcal{S}$ 是状态空间（当前上下文、对话历史等）
- $\mathcal{A}$ 是动作空间（生成文本、调用工具等）
- $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ 是状态转移函数
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ 是奖励函数
- $\pi : \mathcal{S} \to \mathcal{A}$ 是策略函数（由 LLM 实现）

通俗解释:

Agent 就像一个"AI 助手"，它不仅能回答问题，还能:

- **思考**：分析当前情况，决定下一步做什么
- **行动**：调用工具（如搜索、计算、查询数据库）
- **观察**：获取工具返回的结果

- **调整**：根据结果调整策略，继续执行任务

就像人类助手一样：你让他” 查一下明天的天气”，他会先思考” 需要调用天气 API”，然后执行，最后告诉你结果。

Agent 的核心组件：

1. **LLM 核心**：负责推理和决策
2. **工具集（Tools）**：Agent 可以调用的外部功能
3. **记忆（Memory）**：存储对话历史和中间结果
4. **规划器（Planner）**：制定执行计划
5. **执行器（Executor）**：执行工具调用

Agent 的类型：

- **ReAct Agent**：结合推理（Reasoning）和行动（Acting）
- **Plan-and-Execute Agent**：先制定计划，再执行
- **AutoGPT Agent**：自主规划多步骤任务
- **BabyAGI Agent**：基于目标的任务管理

24.2 Function Calling（函数调用）

严谨解释：

Function Calling 是大语言模型的一种能力，允许模型在生成文本时，识别需要调用外部函数的场景，并生成符合函数签名的结构化调用请求。

数学表示：

给定输入 x 和函数集合 $\mathcal{F} = \{f_1, f_2, \ldots, f_n\}$，Function Calling 的过程为：

$$\text{FunctionCall}(x, \mathcal{F}) = \arg\max_{f \in \mathcal{F}} P(f|x) \cdot \text{GenerateArgs}(f, x) \tag{49}$$

其中：

- $P(f|x)$ 是模型判断是否需要调用函数 f 的概率

- GenerateArgs(f, x) 是生成函数参数的过程

通俗解释:

Function Calling 就像给 AI 一个” 工具箱”:

- AI 看到一个问题，比如” 今天北京天气怎么样？ ”
- AI 意识到需要调用” 天气查询” 函数
- AI 生成函数调用：`get_weather(location=" 北京")`
- 系统执行函数，返回结果
- AI 基于结果生成最终回复

这就像你告诉助手” 查天气”，助手知道要去调用天气 API，而不是自己” 编造” 天气信息。

Function Calling 的工作流程:

1. **函数定义**：定义可用的函数及其参数
2. **模型推理**：模型分析输入，决定是否需要调用函数
3. **参数生成**：如果需要，生成函数调用的参数
4. **函数执行**：系统执行函数调用
5. **结果整合**：将函数结果整合到模型回复中

Function Calling 的优势:

- **实时信息**：可以获取最新数据（如天气、股价）
- **精确计算**：可以调用计算工具，避免模型计算错误
- **外部系统集成**：可以连接数据库、API、文件系统等
- **可扩展性**：可以轻松添加新功能

24.3 Tools（工具）

严谨解释：

Tools 是 Agent 可以调用的外部功能接口，通常定义为函数或 API，用于扩展模型的能力边界。

工具定义：

一个工具可以形式化为：

$$\text{Tool} = (\text{name}, \text{description}, \text{parameters}, \text{execute}) \tag{50}$$

其中：

- name 是工具名称
- description 是工具描述（用于模型理解）
- parameters 是参数模式（JSON Schema）
- execute 是执行函数

通俗解释：

Tools 就像给 AI 的” 超能力”：

- **搜索工具**：让 AI 可以搜索互联网
- **计算工具**：让 AI 可以做精确计算
- **文件工具**：让 AI 可以读写文件
- **数据库工具**：让 AI 可以查询数据库
- **API 工具**：让 AI 可以调用各种 API

就像给机器人装上各种” 手臂” 和” 传感器”，让它能做更多事情。

常见工具类型：

- **信息检索**：搜索引擎、向量数据库
- **计算工具**：计算器、代码执行器
- **文件操作**：文件读写、目录遍历

- **网络工具**：HTTP 请求、API 调用
- **数据库工具**：SQL 查询、NoSQL 操作
- **系统工具**：系统命令、进程管理

24.4 MCP（Model Context Protocol）

严谨解释：

MCP（Model Context Protocol）是一个标准化的协议，用于定义模型如何与外部工具和上下文进行交互。它提供了一套统一的接口规范，使得不同的模型和工具可以无缝集成。

协议结构：

MCP 定义了三层协议：

1. **Context Layer**：定义上下文如何传递和管理
2. **Tool Layer**：定义工具如何注册和调用
3. **Protocol Layer**：定义通信协议和消息格式

通俗解释：

MCP 就像” 通用插头标准”：

- 不同的 AI 模型（如 GPT-4、Claude）就像不同的” 电器”
- 不同的工具（如搜索、计算）就像不同的” 插座”
- MCP 定义了” 插头标准”，让任何” 电器” 都能插到任何” 插座” 上

这就像 USB 标准：有了 USB 标准，任何 USB 设备都能连接到任何 USB 接口。

MCP 的核心特性：

- **标准化**：统一的接口规范
- **可扩展**：易于添加新工具
- **可组合**：工具可以组合使用
- **类型安全**：强类型的参数定义

MCP 的优势：

- **互操作性**：不同模型和工具可以无缝协作
- **可维护性**：统一的协议便于维护和更新
- **可扩展性**：易于添加新功能
- **标准化**：减少集成成本

24.5 Agent、Function Calling、Tools、MCP 对比

特性	Agent	Function Calling	Tools	MCP
定位	智能系统	模型能力	功能接口	协议标准
范围	完整系统	调用机制	单个功能	通信协议
自主性	高（自主决策）	中（模型决定）	低（被动调用）	无（协议层）
复杂度	高	中	低	中
主要用途	复杂任务执行	外部功能调用	能力扩展	标准化集成

表 5: Agent、Function Calling、Tools、MCP 对比

关系说明：

- **Agent** 是完整的智能系统，可以使用 **Tools** 来扩展能力
- **Function Calling** 是 Agent 使用 Tools 的机制
- **MCP** 定义了 Agent、Tools 之间的通信协议
- **Tools** 是 Agent 可以调用的具体功能

通俗类比：

- **Agent** = 智能机器人（完整的系统）
- **Tools** = 机器人的各种工具（锤子、螺丝刀等）
- **Function Calling** = 机器人使用工具的方式（” 拿起锤子”、” 敲打”）
- **MCP** = 工具接口标准（确保工具能装到机器人上）

24.6 实际应用案例

24.6.1 案例 1：天气查询 Agent

场景：用户询问” 明天北京的天气如何？ ”

实现步骤：

1. **工具定义**（MCP 格式）：

```
{
    "name": "get_weather",
    "description": "查询指定城市的天气信息",
    "parameters": {
        "type": "object",
        "properties": {
            "location": {
                "type": "string",
                "description": "城市名称"
            },
            "date": {
                "type": "string",
                "description": "日期，格式：YYYY-MM-DD"
            }
        },
        "required": ["location"]
    }
}
```

Listing 26: 天气查询工具定义

2. **Agent 推理**：

```
# 用户输入
user_input = "明天北京的天气如何？"

# Agent 分析
# 1. 识别意图：需要查询天气
# 2. 提取信息：location="北京", date="明天"
# 3. 决定调用：get_weather 函数
```

Listing 27: Agent 推理过程

3. **Function Calling**:

```
# 模型生成函数调用
function_call = {
    "name": "get_weather",
    "arguments": {
        "location": "北京",
        "date": "2025-01-07"  # 明天的日期
    }
}

# 执行函数
result = execute_function(function_call)
# 返回: {"temperature": 5, "condition": "晴", "wind": "3级"}
```

Listing 28: Function Calling 执行

4. **生成回复**:

```
# Agent 基于结果生成回复
response = f"""
根据查询结果，明天（2025-01-07）北京的天气情况：
- 温度：5°C
- 天气：晴天
- 风力：3级

建议穿着轻便外套，适合外出活动。
"""
```

Listing 29: Agent 生成最终回复

24.6.2 案例 2：数据分析 Agent

场景：用户要求” 分析销售数据，找出销量最好的产品”

实现步骤:

1. **工具定义**:

```
tools = [
    {
        "name": "read_csv",
        "description": "读取CSV文件",
        "parameters": {
            "type": "object",
            "properties": {
                "file_path": {"type": "string"}
            }
        }
    },
    {
        "name": "query_data",
        "description": "查询数据",
        "parameters": {
            "type": "object",
            "properties": {
                "sql": {"type": "string"}
            }
        }
    },
    {
        "name": "analyze_data",
        "description": "分析数据",
        "parameters": {
            "type": "object",
            "properties": {
                "data": {"type": "array"},
                "operation": {"type": "string"}
            }
        }
    }
]
```

Listing 30: 数据分析工具定义

2. **Agent 执行流程**:

```
# 步骤1: 读取数据
result1 = agent.call_tool("read_csv", {"file_path": "sales.csv"})

# 步骤2: 查询销量数据
result2 = agent.call_tool("query_data", {
    "sql": "SELECT product, SUM(quantity) as total FROM sales GROUP
   BY product ORDER BY total DESC"
})

# 步骤3: 分析结果
result3 = agent.call_tool("analyze_data", {
    "data": result2,
    "operation": "find_max",
    "field": "total"
})

# 步骤4: 生成报告
report = agent.generate_response(
    f"根据数据分析, 销量最好的产品是: {result3['product']}, "
    f"总销量为: {result3['total']}件"
)
```

Listing 31: 数据分析 Agent 执行

24.6.3 案例 3: 代码生成与执行 Agent

场景: 用户要求” 写一个函数计算斐波那契数列, 并测试前 10 个数”

实现步骤:

```
# 工具定义
tools = [
    {
        "name": "generate_code",
        "description": "生成Python代码",
        "parameters": {
            "type": "object",
            "properties": {
```

```
                "task": {"type": "string"},
                "language": {"type": "string", "default": "python"}
            }
        }
    },
    {
        "name": "execute_code",
        "description": "执行Python代码",
        "parameters": {
            "type": "object",
            "properties": {
                "code": {"type": "string"}
            }
        }
    }
]

# Agent 执行
# 1. 生成代码
code = agent.call_tool("generate_code", {
    "task": "写一个函数计算斐波那契数列"
})
# 返回:
# def fibonacci(n):
#     if n <= 1:
#         return n
#     return fibonacci(n-1) + fibonacci(n-2)

# 2. 执行测试
test_code = f"""
{code}
for i in range(10):
    print(fibonacci(i))
"""
result = agent.call_tool("execute_code", {"code": test_code})

# 3. 生成回复
response = f"""
已生成斐波那契函数并测试前10个数:
{result}
```

```
"""
```

Listing 32: 代码生成与执行 Agent

24.6.4 案例 4：多工具组合 Agent

场景：用户要求” 搜索最新的 AI 论文，总结要点，并保存到文件”

实现步骤：

```
# 工具链
tools = [
    {"name": "search_web", "description": "搜索网页"},
    {"name": "summarize_text", "description": "总结文本"},
    {"name": "write_file", "description": "写入文件"}
]

# Agent 执行流程
# 步骤1: 搜索论文
papers = agent.call_tool("search_web", {
    "query": "latest AI papers 2025",
    "num_results": 5
})

# 步骤2: 总结每篇论文
summaries = []
for paper in papers:
    summary = agent.call_tool("summarize_text", {
        "text": paper["content"],
        "max_length": 200
    })
    summaries.append({
        "title": paper["title"],
        "summary": summary
    })

# 步骤3: 保存到文件
agent.call_tool("write_file", {
    "file_path": "ai_papers_summary.txt",
    "content": "\n\n".join([f"{s['title']}\n{s['summary']}" for s in
```

```
    summaries])
})

# 步骤4: 生成回复
response = f"""
已完成任务:
1. 搜索了5篇最新的AI论文
2. 总结了每篇论文的要点
3. 保存到文件: ai_papers_summary.txt
"""
```

Listing 33: 多工具组合 Agent

24.7 使用 LangChain 实现 Agent

ReAct Agent 示例:

```
from langchain.agents import initialize_agent, Tool
from langchain.llms import OpenAI
from langchain.chains import LLMChain

# 1. 定义工具
def search_tool(query: str) -> str:
    """搜索工具"""
    # 实际实现搜索逻辑
    return f"搜索结果: {query}"

def calculator_tool(expression: str) -> str:
    """计算器工具"""
    try:
        result = eval(expression)
        return str(result)
    except:
        return "计算错误"

tools = [
    Tool(
        name="Search",
        func=search_tool,
```

```
        description="用于搜索最新信息"
    ),
    Tool(
        name="Calculator",
        func=calculator_tool,
        description="用于数学计算"
    )
]

# 2. 初始化 Agent
llm = OpenAI(temperature=0)
agent = initialize_agent(
    tools,
    llm,
    agent="react-chat",
    verbose=True
)

# 3. 使用 Agent
response = agent.run("搜索'Python编程', 然后计算 123 * 456")
print(response)
```

Listing 34: LangChain ReAct Agent

自定义 Agent 示例:

```
from langchain.agents import AgentExecutor, create_react_agent
from langchain.prompts import PromptTemplate
from langchain.tools import Tool

# 1. 定义提示模板
prompt = PromptTemplate.from_template("""
你是一个有用的AI助手，可以使用以下工具:
{tools}

使用以下格式:
Question: 输入的问题
Thought: 你应该思考要做什么
Action: 要采取的行动，应该是[{tool_names}]中的一个
Action Input: 行动的输入
```

```
Observation: 行动的结果
... (这个思考/行动/行动输入/观察可以重复N次)
Thought: 我现在知道最终答案了
Final Answer: 原始输入问题的最终答案

Question: {input}
Thought: {agent_scratchpad}
""")

# 2. 创建 Agent
agent = create_react_agent(llm, tools, prompt)

# 3. 执行 Agent
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)
result = agent_executor.invoke({"input": "计算 100 的平方根"})
```

Listing 35: 自定义 Agent 实现

24.8 最佳实践

1. 工具设计原则:

- **单一职责**: 每个工具只做一件事
- **清晰描述**: 工具描述要清晰，帮助模型理解何时使用
- **参数验证**: 严格验证参数，避免错误调用
- **错误处理**: 优雅处理错误，返回有用信息

2. Agent 设计原则:

- **明确目标**: Agent 应该有明确的任务目标
- **有限工具**: 不要给 Agent 太多工具，避免混乱
- **错误恢复**: 设计错误恢复机制
- **可观测性**: 记录 Agent 的决策过程

3. Function Calling 最佳实践:

- **函数命名**：使用清晰、描述性的函数名
- **参数设计**：参数应该明确、类型安全
- **返回值**：返回结构化数据，便于模型理解
- **文档完善**：提供详细的函数文档

4. MCP 使用建议：

- **遵循标准**：严格遵循 MCP 协议规范
- **类型安全**：使用强类型的参数定义
- **版本管理**：管理协议版本，保持兼容性
- **错误处理**：定义标准的错误码和错误信息

25 更多大模型算法与技术

说明：本节介绍其他重要的大模型算法和技术。关于强化学习相关内容（RLHF、PPO、DPO、GRPO），请参考前面的”大语言模型中的强化学习”章节。

25.1 Orca 和 Orca 2

概念解释：Orca 是微软提出的通过模仿学习提升小模型能力的框架。

核心思想：

- 使用大模型（如 GPT-4）生成高质量数据
- 小模型学习大模型的推理过程
- 通过逐步推理（step-by-step reasoning）提升能力

训练数据：

$$\mathcal{D} = \{(x, y_{\text{teacher}}, \text{reasoning}_{\text{teacher}})\} \tag{51}$$

其中 $\text{reasoning}_{\text{teacher}}$ 是大模型的推理过程。

25.2 Chain-of-Thought (CoT)

概念解释：Chain-of-Thought 通过引导模型进行逐步推理，提升复杂推理任务的表现。

示例：

```
prompt = """
Q: 一个商店有15个苹果，卖出了8个，又进货了12个，现在有多少个?

A: 让我们一步步思考:
1. 初始有15个苹果
2. 卖出了8个，剩余: 15 - 8 = 7个
3. 又进货了12个，现在有: 7 + 12 = 19个
所以答案是19个。
"""
```

Listing 36: CoT 示例

效果：

- 在数学推理任务上提升 20-30%
- 在逻辑推理任务上提升 15-25%

25.3 Tree of Thoughts (ToT)

概念解释：ToT 扩展了 CoT，通过树状结构探索多个推理路径。

算法流程：

1. **生成**：为当前状态生成多个可能的推理步骤
2. **评估**：评估每个步骤的质量
3. **搜索**：使用广度优先或深度优先搜索最佳路径

优势：

- 探索多个解决方案
- 回溯能力，可以修正错误
- 在复杂推理任务上表现更好

25.4 ReAct（Reasoning + Acting）

概念解释：ReAct 结合推理和行动，使模型可以调用外部工具。

框架：

$$\text{Action} = \text{Reasoning} + \text{Acting} + \text{Observation} \tag{52}$$

示例：

```
# 模型可以：
# 1. 思考：需要查询天气信息
# 2. 行动：调用天气API
# 3. 观察：获取天气数据
# 4. 思考：基于天气数据给出建议
# 5. 回答：根据天气建议穿衣
```

Listing 37: ReAct 示例

25.5 Mixture of Experts (MoE)

概念解释：MoE 通过稀疏激活多个专家模型，在保持参数量的同时提升模型容量。

架构：

$$y = \sum_{i=1}^{n} g_i(x) \cdot E_i(x) \tag{53}$$

其中：

- E_i 是第 i 个专家模型
- $g_i(x)$ 是门控函数，决定激活哪些专家
- 通常只激活 k 个专家（如 $k = 2$）

优势：

- 参数总量大，但激活参数少
- 训练和推理效率高
- 模型容量大

代表模型：

- **Switch Transformer**：Google 的 MoE 模型
- **GLaM**：Google 的通用语言模型
- **Mixtral**：Mistral AI 的 MoE 模型

25.6 Retrieval-Augmented Generation (RAG) 进阶

Self-RAG：

Self-RAG 让模型自主决定何时检索、如何检索：

1. **检索决策**：判断是否需要检索
2. **检索执行**：如果需要，执行检索
3. **生成**：基于检索结果生成回复
4. **自我评估**：评估生成质量

Corrective RAG：

Corrective RAG 通过错误检测和纠正提升 RAG 质量：

1. 生成初始回复
2. 检测错误或不确定部分
3. 针对错误部分重新检索
4. 生成纠正后的回复

25.7 多模态大模型

CLIP：

- **架构**：图像编码器 + 文本编码器
- **训练**：对比学习，拉近匹配的图像-文本对
- **应用**：图像检索、图像生成（DALL-E）

GPT-4V：

- **能力**：理解图像和文本
- **应用**：图像问答、图像描述、视觉推理

LLaVA：

- **架构**：视觉编码器（CLIP）+ 语言模型（LLaMA）
- **训练**：视觉指令微调
- **应用**：视觉问答、图像理解

25.8 代码生成模型

Codex / GitHub Copilot：

- **基础模型**：GPT-3
- **训练数据**：GitHub 代码
- **能力**：代码生成、代码补全、代码解释

CodeLlama：

- **基础模型**：LLaMA 2
- **特点**：专门针对代码训练
- **能力**：代码生成、代码补全、代码调试

StarCoder：

- **参数量**：15B
- **训练数据**：800+ 编程语言
- **特点**：开源、可商用

25.9 评估基准补充

HellaSwag：

- **任务**：常识推理
- **格式**：选择题，选择最合理的句子续写
- **评估**：准确率

TruthfulQA：

- **任务**：评估模型的真实性
- **关注点**：避免生成错误信息
- **评估**：真实率（Truthfulness）

GSM8K：

- **任务**：小学数学问题
- **评估**：准确率
- **特点**：需要多步推理

HumanEval：

- **任务**：Python 代码生成
- **评估**：通过率（Pass@k）
- **特点**：164 个编程问题

26 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

27 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。

27.1 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 $\mathbf{w} = w_1, w_2, \ldots, w_N$，困惑度定义为：

$$\text{PPL}(\mathbf{w}) = P(w_1, w_2, \ldots, w_N)^{-\frac{1}{N}} = \sqrt[N]{\frac{1}{P(w_1, w_2, \ldots, w_N)}} \tag{54}$$

使用链式法则展开：

$$\text{PPL}(\mathbf{w}) = \sqrt[N]{\prod_{i=1}^{N} \frac{1}{P(w_i | w_1, \ldots, w_{i-1})}} \tag{55}$$

使用对数形式（数值稳定）：

$$\text{PPL}(\mathbf{w}) = \exp\left(-\frac{1}{N} \sum_{i=1}^{N} \log P(w_i | w_1, \ldots, w_{i-1})\right) \tag{56}$$

符号说明：

- N：序列长度（词数）
- w_i：第 i 个词
- $P(w_i | w_1, \ldots, w_{i-1})$：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 38: 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 39: 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：

$$\text{PPL} = \exp\left(-\frac{1}{2}[\log(0.3) + \log(0.5)]\right) \tag{57}$$

$$= \exp\left(-\frac{1}{2}[-1.204 + (-0.693)]\right) \tag{58}$$

$$= \exp\left(-\frac{1}{2} \times (-1.897)\right) \tag{59}$$

$$= \exp(0.9485) \tag{60}$$

$$= 2.582 \tag{61}$$

代码验证：

```python
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 40: 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：

$$\text{PPL} = \exp\left(-\frac{1}{10} \times (-2.0) \times 10\right) \tag{62}$$

$$= \exp(2.0) \tag{63}$$

$$= 7.389 \tag{64}$$

27.2 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：

$$P_n = \frac{\sum_{\text{n-gram} \in \text{candidate}} \text{Count}_{\text{clip}}(\text{n-gram})}{\sum_{\text{n-gram} \in \text{candidate}} \text{Count}(\text{n-gram})} \tag{65}$$

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：

$$\text{BLEU} = \text{BP} \times \exp\left(\sum_{n=1}^{N} w_n \log P_n\right) \tag{66}$$

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：

$$\text{BP} = \begin{cases} 1 & \text{if } c > r \\ e^{1-r/c} & \text{if } c \leq r \end{cases} \tag{67}$$

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 41: 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 42: 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：

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

BLEU 分数：

$$\text{BLEU} = \text{BP} \times \exp\left(\frac{1}{4}[\log P_1 + \log P_2 + \log P_3 + \log P_4]\right) \tag{68}$$

$$= 1.1537 \times \exp\left(\frac{1}{4}[\log(0.7143) + \log(0.5) + \log(0) + \log(0)]\right) \tag{69}$$

$$= 0 \quad （因为P_3 = P_4 = 0） \tag{70}$$

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 43: 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：

$$\text{BLEU} = 1.0 \times \exp\left(\frac{1}{4}[\log(1.0) + \log(1.0) + \log(1.0) + \log(1.0)]\right) \tag{71}$$

$$= 1.0 \times \exp(0) \tag{72}$$

$$= 1.0 \tag{73}$$

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 44: 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：

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

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

27.3 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：

$$\text{ROUGE-N} = \frac{\sum_{S \in \text{References}} \sum_{\text{n-gram} \in S} \text{Count}_{\text{match}}(\text{n-gram})}{\sum_{S \in \text{References}} \sum_{\text{n-gram} \in S} \text{Count}(\text{n-gram})} \tag{74}$$

ROUGE-L（最长公共子序列）：

$$\text{ROUGE-L} = \frac{(1+\beta^2) R_{lcs} P_{lcs}}{R_{lcs} + \beta^2 P_{lcs}} \tag{75}$$

其中：

$$R_{lcs} = \frac{LCS(X,Y)}{m} \quad \text{（召回率）} \tag{76}$$

$$P_{lcs} = \frac{LCS(X,Y)}{n} \quad \text{（精确率）} \tag{77}$$

$LCS(X,Y)$ 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 45: 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 = $\frac{2\times0.8333\times0.8333}{0.8333+0.8333}$ = 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 46: 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 = $\frac{2\times0.3333\times1.0}{0.3333+1.0} = 0.5000$

27.4 METEOR 分数

概念解释: METEOR 考虑同义词匹配，比 BLEU 更灵活。

数学公式:

$$\text{METEOR} = (1 - \text{Penalty}) \times \text{F}_{\text{mean}} \tag{78}$$

其中:

$$\text{F}_{\text{mean}} = \frac{P \times R}{\alpha P + (1-\alpha)R} \tag{79}$$

$$\text{Penalty} = 0.5 \times \left(\frac{\text{chunks}}{\text{unigrams\_matched}}\right)^3 \tag{80}$$

27.5 BERTScore

概念解释: BERTScore 使用 BERT 嵌入计算语义相似度。

数学公式:

$$\text{Precision} = \frac{1}{|\hat{\mathbf{x}}|}\sum_{\hat{\mathbf{x}}_i \in \hat{\mathbf{x}}} \max_{\mathbf{x}_j \in \mathbf{x}} \hat{\mathbf{x}}_i^T \mathbf{x}_j \tag{81}$$

$$\text{Recall} = \frac{1}{|\mathbf{x}|}\sum_{\mathbf{x}_j \in \mathbf{x}} \max_{\hat{\mathbf{x}}_i \in \hat{\mathbf{x}}} \hat{\mathbf{x}}_i^T \mathbf{x}_j \tag{82}$$

$$\text{F1} = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{83}$$

28 评测基准与数据集

28.1 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。

28.2 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。

28.3 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。

28.4 MT-Bench 和 AlpacaEval

MT-Bench：多轮对话评估基准。

AlpacaEval：指令遵循能力评估。

29 QA Pair（问答对）

29.1 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 $\Delta\mathbf{W} = \mathbf{BA}$，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

29.2 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 × 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

30 综合练习

30.1 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。

30.2 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

30.3 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。

30.4 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。