模型训练与评估实验手册

一、实验概述

本实验聚焦于使用特定数据集对预训练语言模型进行微调，并对微调后的模型开展推理和评估工作。整个实验涵盖数据预处理、模型训练、模型推理和模型评估四大核心步骤，每个步骤都由对应的 Python 脚本实现。

二、实验环境准备

2.1 安装必要的库

在启动实验前，需安装以下 Python 库：

- torch：深度学习计算的基础库，提供了张量运算、自动求导等功能，为模型训练和推理提供支持。
- transformers：Hugging Face 推出的自然语言处理库，集成了丰富的预训练模型以及模型训练、推理所需的工具和方法。
- tqdm：用于展示进度条，能直观呈现训练和处理的进度，方便用户实时了解任务进展。
- ipywidgets（可选）：若在 Jupyter Notebook 环境中运行实验，安装该库可显示交互式进度条，增强用户体验。

可使用如下命令进行安装：

```
pip install torch transformers tqdm ipywidgets
```

若使用 Jupyter Notebook，还需启用 ipywidgets：

```
jupyter nbextension enable --py widgetsnbextension
```

若使用 JupyterLab，则运行以下命令：

```
jupyter labextension install @jupyter-widgets/jupyterlab-manager
```

2.2 准备预训练模型

从 Hugging Face 下载所需的预训练模型，如 deepseek-r1-1.5b。访问 Hugging Face 官方网站（https://huggingface.co/ ），在搜索栏输入模型名称，找到对应的模型页面。在模型页面中，根据页面提示，使用 git lfs 等工具下载模型文件，并将其存储到本地指定路径，例如 D:\app\Ollama\deepseek-r1-1.5b。

2.3 准备数据集

将数据集整理为每行一个样本的格式，保存为 preprocessed_data.txt 文件。确保文件编码为 UTF - 8，避免因编码问题导致数据读取错误。

三、预训练语言模型的选择和配置说明

3.1 模型选择依据

- **任务适配性**：根据实验任务的类型选择合适的模型。例如，本次实验主要涉及文本生成和 SQL 相关的自然语言处理任务，deepseek-r1-1.5b 这类通用语言模型在经过微调后，能够较好地适应此类任务，它在大规模语料上进行预训练，具备强大的语言理解和生成能力。
- **模型规模**：模型规模在一定程度上决定了模型的性能和计算资源需求。deepseek-r1-1.5b 包含 15 亿参数，在处理复杂任务时表现出色，但同时也需要更多的计算资源和训练时间。如果实验环境资源有限，可以考虑选择规模较小的模型，如 distilbert-base-uncased，它具有较小的模型体积和较快的推理速度，适用于对资源要求较高的场景。

3.2 模型配置调整

- **加载配置文件**：预训练模型的配置信息存储在 config.json 文件中，在加载模型时，transformers 库会自动读取该文件。文件中包含模型的架构信息，如层数、隐藏层维度、注意力头数等。以 deepseek-r1-1.5b 为例，可通过以下方式查看部分配置信息：

```
import json
config_path = r"D:\app\Ollama\deepseek-r1-1.5b\config.json"
with open(config_path, 'r', encoding='utf-8') as f:
    config = json.load(f)
    print("模型层数:", config.get('num_hidden_layers'))
    print("隐藏层维度:", config.get('hidden_size'))
```

- **调整超参数**：在微调模型时，需要根据实验需求调整超参数。在 train_model.py 脚本中，通过 TrainingArguments 进行超参数设置。例如：

```
training_args = TrainingArguments(
    output_dir='./results',
    num_train_epochs=3,
    per_device_train_batch_size=2,
    save_steps=10_000,
    save_total_limit=2,
    prediction_loss_only=True,
    gradient_accumulation_steps=2,
    logging_steps=100,
    warmup_steps=500,
    weight_decay=0.01,
    fp16=False,
    use_cpu=True
)
```

- num_train_epochs：表示训练的轮数，增加轮数可能会提高模型性能，但也会增加训练时间，需根据数据集大小和模型收敛情况进行调整。
- per_device_train_batch_size：是每个设备上的训练批次大小，较大的批次大小可以利用更多的计算资源，但可能会导致内存不足，需根据硬件条件进行选择。
- learning_rate：学习率控制模型参数更新的步长，过大的学习率可能导致模型训练不稳定，过小的学习率则会使训练速度过慢，通常需要通过试验进行调优。

四、脚本功能及使用说明

4.1 preprocess_data.py

功能

该脚本用于对原始数据集进行预处理，将其转化为模型可接受的格式。

代码示例

```
# preprocess_data.py
```

```
import os
def preprocess_data(input_file, output_file):
    # 这里可以添加具体的预处理逻辑，例如去除特殊字符、分词等
    with open(input_file, 'r', encoding='utf-8') as f_in, open(output_file, 'w', encoding='utf-8') as f_out:
        for line in f_in:
            # 简单示例：去除首尾空格
            processed_line = line.strip()
            f_out.write(processed_line + '\n')
if __name__ == "__main__":
    input_file = 'raw_data.txt'
    output_file = 'preprocessed_data.txt'
    preprocess_data(input_file, output_file)
```

解释

- `preprocess_data` 函数：接收输入文件路径和输出文件路径作为参数，对输入文件的每一行数据进行预处理操作，并将处理后的结果写入输出文件。
- 在 `__main__` 部分，指定输入文件和输出文件的路径，调用 `preprocess_data` 函数执行数据预处理。

4.2 train_model.py

功能

该脚本用于加载预训练模型，对其进行微调，并保存微调后的模型。

代码示例

```
# train_model.py
import torch
import os
import json
import time
from torch.utils.data import Dataset
```

```
from transformers import AutoTokenizer, AutoModelForCausalLM, TrainingArguments, Trainer

# 自定义数据集类

class TextDataset(Dataset):
    def __init__(self, file_path, tokenizer, max_length):
        start_time = time.time()
        with open(file_path, 'r', encoding='utf-8') as f:
            self.texts = f.readlines()
        end_time = time.time()
        print(f"数据加载耗时: {end_time - start_time} 秒")
        self.tokenizer = tokenizer
        self.max_length = max_length
    def __len__(self):
        return len(self.texts)
    def __getitem__(self, idx):
        text = self.texts[idx]
        encoding = self.tokenizer.encode_plus(
            text,
            add_special_tokens=True,
            max_length=self.max_length,
            padding='max_length',
            truncation=True,
            return_tensors='pt'
        )
        input_ids = encoding['input_ids'].flatten()
        attention_mask = encoding['attention_mask'].flatten()
        return {
            'input_ids': input_ids,
            'attention_mask': attention_mask,
            'labels': input_ids.clone()
        }
if __name__ == "__main__":
```

```
    # 本地模型路径
    local_model_path = r"D:\app\Ollama\deepseek-r1-1.5b"
    config_path = f"{local_model_path}/config.json"
    tokenizer_path = f"{local_model_path}/tokenizer.json"
    try:
        # 检查配置文件是否存在
        if not os.path.exists(config_path):
            raise FileNotFoundError(f"未找到 {config_path} 文件，请检查模型路径是否正确。
")
        # 检查配置文件内容
        with open(config_path, 'r', encoding='utf-8') as f:
            config = json.load(f)
            if'model_type' not in config:
                raise ValueError(f"{config_path} 文件中缺少'model_type' 字段，请检查文件内
容。")
        # 检查分词器文件是否存在
        if not os.path.exists(tokenizer_path):
            raise FileNotFoundError(f"未找到 {tokenizer_path} 文件，请检查模型路径是否正
确。")
        # 加载预训练的 tokenizer 和模型
        start_time = time.time()
        tokenizer = AutoTokenizer.from_pretrained(local_model_path)
        model = AutoModelForCausalLM.from_pretrained(
            local_model_path,
            torch_dtype=torch.float16,
            trust_remote_code=True  # 如果模型代码包含自定义部分，可能需要添加此参数
        )
        end_time = time.time()
        print(f"模型加载耗时: {end_time - start_time} 秒")
        # 训练参数设置
        training_args = TrainingArguments(
```

```
            output_dir='./results',
            num_train_epochs=3,
            per_device_train_batch_size=2,
            save_steps=10_000,
            save_total_limit=2,
            prediction_loss_only=True,
            gradient_accumulation_steps=2,
            logging_steps=100,  # 每 100 步记录一次日志
            warmup_steps=500,  # 热身步数
            weight_decay=0.01,  # 权重衰减
            fp16=False,  # CPU 不支持混合精度训练，设置为 False
            use_cpu=True  # 强制使用 CPU
        )
        # 创建数据集
        dataset = TextDataset('preprocessed_data.txt', tokenizer, max_length=512)
        # 创建 Trainer 对象
        trainer = Trainer(
            model=model,
            args=training_args,
            train_dataset=dataset,
        )
        print("开始训练前的准备工作...")
        start_time = time.time()
        trainer.train()
        end_time = time.time()
        print(f"训练总耗时: {end_time - start_time} 秒")
        # 保存训练好的模型
        trainer.save_model('./fine_tuned_model')
    except (FileNotFoundError, ValueError, OSError) as e:
        print(f"加载模型时出现错误: {e}，请检查本地模型路径 {local_model_path} 是否正确。")
```

解释

- TextDataset 类：自定义的数据集类，负责加载和处理数据集，将文本数据转换为模型输入所需的格式。
- 在__main__部分：
  - 检查模型配置文件和分词器文件是否存在，确保模型加载路径正确。
  - 加载预训练的分词器和模型。
  - 设置训练参数，包括训练轮数、批次大小、保存步数等，这些参数直接影响模型的训练效果和效率。
  - 创建数据集和 Trainer 对象，Trainer 对象封装了模型训练的核心逻辑。
  - 启动训练过程，并保存微调后的模型。

4.3 model_inference.py

功能

该脚本用于加载微调后的模型，并进行文本生成推理。

代码示例

```
# model_inference.py
from transformers import AutoTokenizer, AutoModelForCausalLM
# 加载训练好的模型和分词器
model_path = './fine_tuned_model'
tokenizer = AutoTokenizer.from_pretrained(model_path)
model = AutoModelForCausalLM.from_pretrained(model_path)
# 输入文本
input_text = "这是一个测试输入："
# 对输入文本进行编码
input_ids = tokenizer.encode(input_text, return_tensors='pt')
# 生成文本
output = model.generate(input_ids, max_length=100, num_beams=5,
no_repeat_ngram_size=2)
```

```
# 解码生成的文本
generated_text = tokenizer.decode(output[0], skip_special_tokens=True)
print("生成的文本：", generated_text)
```

解释

- 加载微调后的模型和分词器，确保模型和分词器路径正确。
- 输入待生成的文本，并使用分词器将其编码为模型可接受的格式。
- 调用 model.generate 方法进行文本生成，通过设置 max_length、num_beams、no_repeat_ngram_size 等参数控制生成文本的长度、搜索策略和重复度。
- 使用分词器解码生成的文本，并输出结果。

4.4 evaluate_model.py

功能

该脚本用于对微调后的模型进行评估，计算困惑度等指标。

代码示例

```
# evaluate_model.py
import torch
from transformers import AutoTokenizer, AutoModelForCausalLM
from datasets import load_dataset
# 加载训练好的模型和分词器
model_path = './fine_tuned_model'
tokenizer = AutoTokenizer.from_pretrained(model_path)
model = AutoModelForCausalLM.from_pretrained(model_path)
# 加载评估数据集
dataset = load_dataset('text', data_files='preprocessed_data.txt')
# 计算困惑度
total_loss = 0
total_tokens = 0
for example in dataset['train']:
```

```
    input_text = example['text']
    input_ids = tokenizer.encode(input_text, return_tensors='pt')
    with torch.no_grad():
        outputs = model(input_ids, labels=input_ids)
        loss = outputs.loss
        total_loss += loss.item() * input_ids.size(1)
        total_tokens += input_ids.size(1)
perplexity = torch.exp(torch.tensor(total_loss / total_tokens))
print(f"困惑度: {perplexity.item()}")
```

解释

- 加载微调后的模型和分词器。
- 使用 `load_dataset` 函数加载评估数据集，确保数据集路径正确。
- 遍历评估数据集中的每个样本，将样本输入模型计算损失值。
- 根据累计的损失值和样本中的总词数计算困惑度，并输出结果。困惑度是评估语言模型性能的重要指标之一，越低表示模型对数据的拟合效果越好。

五、实验步骤

1. **数据预处理**：运行 `preprocess_data.py` 脚本，对原始数据集进行预处理，生成 `preprocessed_data.txt` 文件。
2. **模型训练**：运行 `train_model.py` 脚本，加载预训练模型，按照设定的超参数对其进行微调，并将微调后的模型保存到 `fine_tuned_model` 目录。
3. **模型推理**：运行 `model_inference.py` 脚本，加载微调后的模型，输入待生成的文本，获取模型生成的文本结果。
4. **模型评估**：运行 `evaluate_model.py` 脚本，对微调后的模型进行评估，计算困惑度等指标，评估模型性能。

六、注意事项

5. 确保数据集格式正确，每行代表一个样本，且文件编码为 UTF - 8。
6. 若使用 GPU 进行训练，需安装相应的 CUDA 驱动和 CUDA Toolkit，并保证 PyTorch 版本与 CUDA 版本兼容。

7. 在训练过程中，可能会出现警告信息，如 Sliding Window Attention 警告，可根据实际情况更新库版本或调整模型配置进行处理。
8. 调整模型超参数时，需充分考虑计算资源和模型性能的平衡，避免因超参数设置不当导致模型训练失败或性能不佳。