Programming Assignment 1 - LLM Fine Tuning

Assignment Overview

Goal: Fine-Tuning LLaMA on 1 GPU with Memory Optimizations

Dataset
Climate documents dataset of IPCC reports and climate change and AI publications from the last 5 years.

- PDF file format
- On the cloud burst compute file system: /scratch/BDML25SP/

Students can use random 90% files from the dataset for fine-tuning. Note that you need to extract the text from the data and create txt files as part of preprocessing.

Pretrained Model

- LLaMA 3B model
- On the cloud burst compute file system: /scratch/BDML25SP/

Key Focus
We will be focusing on memory optimizations, for example

- Low Rank Adaptation (LoRA)
- Mixed Precision Fine-Tuning, Quantization
- Gradient Accumulation and Checkpointing

The goal of the assignment is to increase the batch size of the training process as much as possible.

Deliverables

1. A report documenting:
    a. Memory optimization strategies used.
    b. Training performance (maximum batch size) and evaluation results.
    c. Step by step guide on how to run the training code.

2. Code access on HPC

Evaluation

Compute the perplexity metric on the remaining 10% of the dataset. The assignment will be evaluated primarily on the basis of how memory and time efficient the fine-tuning code is, and the final perplexity score will not hold much weight.

Data Processing

Digital PDFs have **embedded text**, making them directly readable by computers. Text can be extracted easily using tools like **PyPDF2** (docs), **pdfplumber** (docs), or **PyMuPDF** (docs) in Python.

Example code

```
from PyPDF2 import PdfReader

# Read the PDF file
pdf_path = '/path/to/data/file.pdf'
reader = PdfReader(pdf_path)

# Extract text from the first page
first_page = reader.pages[0].extract_text()
first_page[:1000]  # Display the first 1000 characters as a preview
```

To split the dataset, you can create a list of filenames in the folder and split the list in 9:1 for train and test sets.

Low Rank Adaptation (LoRA)

LoRA (Low-Rank Adaptation) is a **parameter-efficient fine-tuning (PEFT) technique** designed to fine-tune large language models (LLMs) like LLaMA, GPT, or BERT with **significantly fewer trainable parameters and lower memory requirements**. It adds **trainable low-rank matrices** to **pre-trained model weights** instead of updating the entire model, making it faster and more memory-efficient.

You can fine-tune LLaMA-3B with LoRA using Hugging Face's **`peft`** library.

Example code

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

# Load pre-trained LLaMA model
model_name = "path/to/model"
model = AutoModelForCausalLM.from_pretrained(model_name)
tokenizer = AutoTokenizer.from_pretrained(model_name)

# Define LoRA config
lora_config = LoraConfig(
    task_type=TaskType.CAUSAL_LM,  # Language modeling
    r=8,  # Low-rank matrix dimension
    lora_alpha=32,  # Scaling factor
    lora_dropout=0.1,  # Dropout rate
)

# Apply LoRA
lora_model = get_peft_model(model, lora_config)
lora_model.print_trainable_parameters()
```

Precision Optimization

Using **FP16 (Half-Precision)** significantly reduces memory usage while maintaining training stability.
Example code

```python
from transformers import TrainingArguments, Trainer

training_args = TrainingArguments(
    output_dir="./llama-finetune",
    ...
    fp16=True,  # Enables FP16 (Half-Precision)
    bf16=False,  # Use BF16 (Better for A100 GPUs)
)


trainer = Trainer(
    model=model,  # Fine-tuned LLaMA model
    args=training_args,  # Training arguments
    train_dataset=train_dataset,  # Training data
    ...
    tokenizer=tokenizer,  # Tokenizer

)

trainer.train()  # Start training
```

QLoRA allows efficient fine-tuning of LLMs **without full model updates** by using **4-bit quantization** + **LoRA (Low-Rank Adaptation)**.

Dependencies:

```
pip install bitsandbytes transformers accelerate peft
```

Example code

```python
from transformers import AutoModelForCausalLM, AutoTokenizer, BitsAndBytesConfig
from peft import LoraConfig, get_peft_model

# Load LLaMA-3B with 4-bit quantization
model_name = "path/to/model"
quantization_config = BitsAndBytesConfig(
    load_in_4bit=True,  # Enables 4-bit quantization
    bnb_4bit_compute_dtype=torch.float16,  # Compute in FP16
    bnb_4bit_use_double_quant=True,  # Further memory optimization
)

# Load model with quantization
model = AutoModelForCausalLM.from_pretrained(model_name, quantization_config=quantization_config)
tokenizer = AutoTokenizer.from_pretrained(model_name)

# Apply LoRA on quantized model
lora_config = LoraConfig(
    r=8,  # Low-rank dimension
    lora_alpha=32,
    target_modules=["q_proj", "v_proj"],  # Apply LoRA to attention layers
    lora_dropout=0.1,
    bias="none"
)
lora_model = get_peft_model(model, lora_config)
lora_model.print_trainable_parameters()
```

Gradient Accumulation and Checkpointing

Gradient Accumulation – Simulate larger batch sizes without exceeding memory.
Gradient Checkpointing – Save memory by recomputing activations instead of storing them.

Example code

```python
training_args = TrainingArguments(
    ...
    per_device_train_batch_size=2,  # Small batch size per GPU
    gradient_accumulation_steps=8,  # Simulate batch size of 16
    ...
)
```

Example code

```python
from transformers import AutoModelForCausalLM

model = AutoModelForCausalLM.from_pretrained("model/path")
model.gradient_checkpointing_enable()  # Enable memory optimization

```

Evaluation

On the test set, you can evaluate the fine-tuned model by using the perplexity metric. **Perplexity** (**PPL**) is a measure of how well a language model predicts a given dataset on the next token prediction task. It is commonly used to evaluate fine-tuned language models**.**

Mathematically, perplexity is the exponentiated average negative log-likelihood of the model predictions:

$$PPL = e^{\left(-\frac{1}{N}\sum_{i=1}^{N}\log P(w_i)\right)}$$

Where, P(w_i) is the probability assigned by the model to the ith word, and N is the number of words in the dataset.

Lower perplexity means better predictions (more confident and accurate). Higher perplexity indicates poor model performance (more uncertain predictions).

You can evaluate perplexity on your fine-tuned LLaMA model using the Hugging Face `transformers` library**.**

Example code

```python
import torch
import math
from transformers import AutoModelForCausalLM, AutoTokenizer

# Load fine-tuned LLaMA model (Replace with your model path)
model_name = "your-finetuned-llama"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name)

# Sample evaluation text (use held-out dataset)
evaluation_text = """
Climate change is caused by an increase in greenhouse gases such as CO2.
"""
tokens = tokenizer(evaluation_text, return_tensors="pt", truncation=True, padding=True)

# Compute loss
with torch.no_grad():
    outputs = model(**tokens, labels=tokens["input_ids"])
    loss = outputs.loss.item()

# Compute perplexity
perplexity = math.exp(loss)
print(f"Perplexity: {perplexity:.2f}")
```

Interpreting perplexity score:

Perplexity Score	Interpretation
≤ 10	Very good model (accurate predictions)
10 - 50	Decent model (still useful, but can be improved)
> 100	Poor performance (struggles with predictions)
> 1000	Model is guessing almost randomly