Efficient Pre-processing and Fine-Tuning of Large Language Model

MSBD5003 Project Proposal

Project Type: Exploratory Project.

RONG Shuo
21126613
srongaa@connect.tust.hk

LI Zhenghao
21114012
zh.li@connect.tust.hk

SUN Zixuan
21091674
zsunbl@connect.ust.hk

WU Yongjin
20564741
ywudf@connect.ust.hk

1 Introduction

The rapid evolution of large language models (**LLMs**) has revolutionized natural language processing, with open-source breakthroughs like DeepSeek providing access to state-of-the-art technologies. These foundation LLMs are typically trained on vast amounts of publicly available internet data, effectively compressing a broad spectrum of open knowledge. However, their performance often suffers when applied to nonpublic or domain-specific data.

A common solution to address such issues is Supervised Fine-Tuning (**SFT**), which adapts the foundation model to specific tasks or datasets by further training it on labeled examples. Compared to full-scale pre-training, SFT requires substantially less time and computational resources, while still enabling the model to achieve strong performance on targeted applications.

However,considering the tremendous size of LLMs nowadays, the SFT of such models still presents significant computational and data-driven challenges, necessitating the integration of big data technologies and distributed systems. This project aims to systematically explore the end-to-end process of fine-tuning **BERT** with a large-scale dataset by leveraging **Apache Spark** for distributed data pre-processing and utilizing several parallel deep learning frameworks for fine-tuning.

Specifically, we propose a Spark-centric pipeline for distributed data preparation to address the challenges of processing massive training corpora. For example, we will implement scalable algorithms using Spark RDDs and DataFrames to eliminate duplicate documents and low-quality text while ensuring language consistency. Additionally, we will develop a distributed BERT tokenization workflow utilizing Huffman-based partitioning to balance computational loads across nodes.

In addition to pre-processing, we will explore four different parallelism approaches for fine-tuning, including data- parallelism, optimizor-, pipeline- and tensor-parallelism. Relevant open-sourced framework such as **PyToch Distributed**, **DeepSpeed** and **Megatron** will be adopted.

Through these methodologies, we aim to enhance the efficiency and scalability of fine-tuning language models, paving the way for future advancements in the field.

2 Dataset

Amazon Product Reviews dataset will be used in this project to fine-tune the large language model BERT for sentiment analysis.

The dataset consists of user-generated reviews, including the review text and corresponding ratings. In our preliminary approach, we will classify reviews with ratings of 3 to 5 as positive and those with 1 to 2 as negative. This dataset contains approximately 35 million reviews spanning up to March 2013, providing a robust foundation for training our model. Each review includes product and user information, a numerical rating, and a plaintext review, making it well-suited for our sentiment analysis task.

3 Workflow

The implementation of the project will follow these steps:

- **Dataset and Model Acquisition**: Download the Amazon Product Reviews dataset and the pretrained BERT model to the SuperPod.
- **Data Pre-processing**: Utilize PySpark for dataset preprocessing, which will include file reading, filtering, deduplication, and augmentation, leveraging multiple CPU cores.
- **Supervised Fine-Tuning**: Employ PyTorch Distributed, DeepSpeed, and Megatron to fine-tune the pretrained model, leveraging multiple GPU cores.
- **Evaluation and Analysis**: Verify that the classification accuracy across various training configurations remains consistent. Additionally, document the speedup and Model FLOPS Utilization (MFU) to assess the efficiency of the distributed training process.

Further technical details regarding Data Pre-processing and SFT will be provided in Part 4.

4 Technical Details

4.1 Large-scale Data Pre-processing

We plan to use PySpark for preprocessing massive amounts of text data. PySpark is the Python interface for Apache Spark, which allows for distributed computing, making it ideal for handling large-scale datasets. Since we will later use PyTorch for deep learning model training, choosing PySpark not only enables efficient data processing but also facilitates seamless integration with PyTorch, making it easier to transfer the processed data to PyTorch for subsequent handling.

Specifically, we will use PySpark's DataFrame API to perform the following data processing operations:

1. **File Reading**
   For different data formats, we will use `spark.read.csv` or `spark.read.text` to read the text data.
2. **Data Filtering**
   We will use the `df.filter` method to filter and select the data. For example, we will delete records that do not meet specified criteria, ensuring the quality of the data.
3. **Deduplication**
   We will use `df.dropDuplicates` to remove duplicate records and ensure the diversity of the data.
4. **Data Augmentation**
   To enhance the model's generalization ability, we will use `pyspark.sql.functions.udf` to create custom data augmentation functions and apply these operations to the DataFrame columns using `df.withColumn`. We plan to use Easy Data Augmentation techniques, such as random synonym replacement, word order swapping, synonym insertion, and word deletion. These methods will effectively expand the dataset and increase the model's robustness.
5. **Tokenization**
   For tokenizing the text, we plan to use a greedy algorithm or the tokenizer from the `transformers` library.

4.2 Large-scale Fine-tuning

Parallel computing technique is an ideal solution to accelerate the fine-tuning process of LLMs. Four different parallelism techniques are selected in our project.

4.2.1 Data Parallelism

Data Parallelism is a technique to assign non-overlapping subsets of the dataset to different GPU devices. In this approach, each device holds a complete copy of the model,

Table 1: Summary of Parallelism Techniques.

Technique	Framework
Data Parallelism	PyTorch DDP
Zero Redundancy Optimizer (ZeRO)	PyTorch FSDP
Pipeline Parallelism	DeepSpeed
Tensor Parallelism	Megatron

and each device processes a different subset of the input data. The gradients computed from each subset are then aggregated to update the model parameters.

PyTorch DistributedDataParallel (DDP) library is a widely used implementation of data parallelism. It enables efficient training of deep learning models across multiple GPUs and nodes. Here is the basic workflow with DDP:

1. Initialize the process group for distributed training.

2. Wrap the model with **torch.nn.parallel.DistributedDataParallel**. Then, the model is replicated on all the devices.

3. Split the dataset using **torch.utils.data.DistributedSampler**, which ensures each device gets a non-overlapping input batch.

4. Perform the training loop, where each process computes gradients on its subset of data.

5. DDP handles the synchronization of gradients automatically. Each replica calculates gradients and synchronizes with the others using the AllReduce functions. Make sure the parameters are consistent across all copies.

However, the process of data parallelism has limitations. Data parallelism cannot be used for infinite scale-out, which illustrates that the convergence rate may be affected when the global batch size is too large. Another problem is memory issues. In DDP, each device needs to maintain a complete copy of the model (parameters, gradients, and optimizer status). This is impossible for current LLMs.

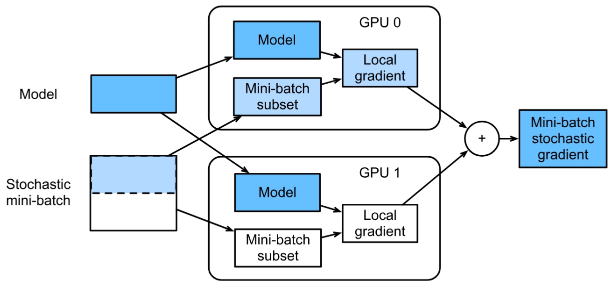

Figure 1: Illustration of Data Parallelism

4.2.2 Zero Redundancy Optimizer (ZeRO)

In traditional Distributed Data Parallelism (DDP), each process keeps a complete copy of the optimizer state, which will cause memory redundancy when training large models. Zero Redundancy Optimizer (ZeRO) provides a way to share model parameters, gradients, and optimizer states across multiple GPUs, minimizing memory usage and maximizing training efficiency.

PyTorch Fully Sharded Data Parallel (FSDP) is an implementation of ZeRO to slice up the memory footprints so that only a portion of the memory is used by each process. The core of it is distributing the optimizer states, gradients, and parameters efficiently across multiple devices. On the other hand, by sharding optimizer states and gradients, FSDP minimizes the amount of data that needs to be communicated between GPUs, which can be a bottleneck in distributed training.

Figure below illustrates the memory usage with and without ZeRO.

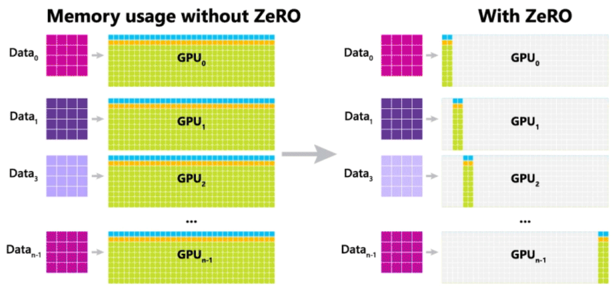

Figure 2: Illustration of ZeRO

4.2.3 Pipeline Parallelism

For those large models that don't fit on a single GPU, pipeline parallelism partitions the model into multiple stages. The model is split into different segments (or stages), and each segment is assigned to a different device. This allows for the simultaneous processing of multiple mini batches, effectively increasing the utilization of the devices.

In pipeline parallelism, different layers of the model are assigned to different GPUs. Each GPU processes its assigned layers and passes the output to the next GPU in the pipeline. The figure illustrates the training process of pipeline parallelism in different devices. After each stage completes forward propagation, the activation memory is passed to the next stage; for backward propagation, the gradient is passed backwards through the pipeline.

However, only one GPU is active at a time according to the naïve version of the pipeline parallel, which makes the MFU very low. To optimize the approach, each batch is split into micro-batches so that computation can be executed as a pipeline paradigm. The process is shown below.

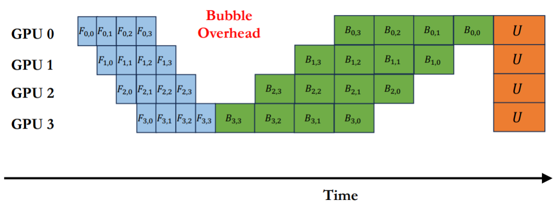

Figure 3: Illustration of Pipeline Parallelism

4.2.4 Tensor Parallelism

Tensor Parallelism is a technique used to distribute the computation of large tensors across multiple GPUs. Megatron is a framework developed by NVIDIA for training large transformer-based models efficiently using tensor parallelism. Megatron allows users to split the model's layers and tensors across multiple GPUs, enabling the training of models with billions of parameters. This means that each GPU only holds a part of the model's parameters and performs computations on its assigned tensors.

For example, the weight matrix A from the MLP module in transformers is split vertically into two parts by columns in the figure. And matrix B is split horizontally into two parts by rows. The result of multiplication Z1 and Z2 each has half dimension of Z, which has identical dimension to input X. In this way, communication during the computations in different GPUs is simplified. This is crucial for maintaining performance as the number of GPUs increases.

Megatron also can be combined with data parallelism, allowing for an effective hybrid approach that leverages both methods for improved training speed and efficiency.

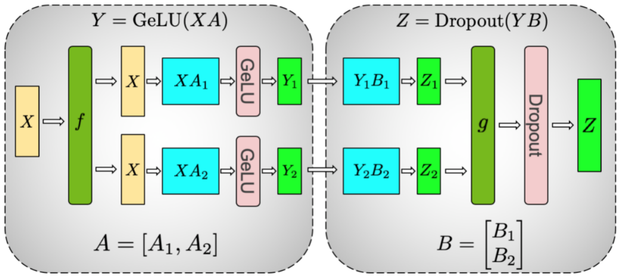

Figure 4: Illustration of Tensor Parallelism