OPTIMIZATION AND SEGMENTATION OF LARGE LANGUAGE MODELS ON EDGE DEVICES

Project Report submitted by

CHETHANA R KINI
(4NM21AI018)

MANYA HEGDE
(4NM21AI038)

REYONA ELZA SABU
(4NM21AI059)

Under the Guidance of

Dr. Rashmi Adyapady R.
Asst. Professor, Dept. of AIML

In partial fulfillment of the requirements for the award of

the Degree of

Bachelor of Engineering in Artificial Intelligence and Machine Learning Engineering

from

Visvesvaraya Technological University, Belagavi

Department of Artificial Intelligence and Machine Learning Engineering
NMAM Institute of Technology, Nitte - 574110
(An Autonomous Institution affiliated to VTU, Belagavi)

NOVEMBER 2024

NITTE EDUCATION TRUST | N.M.A.M. INSTITUTE OF TECHNOLOGY
(An Autonomous Institution affiliated to Visvesvaraya Technological University, Belagavi)
Nitte – 574 110, Karnataka, India

DEPARTMENT OF ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING ENGINEERING

CERTIFICATE

Certified that the project work entitled

"Optimization of Large Language Models on Edge Devices"

is a bonafide work carried out by

Manya Hegde – 4NM21AI038

in partial fulfilment of the requirements for the award of

Bachelor of Engineering Degree in Artificial Intelligence and Machine Learning

prescribed by Visvesvaraya Technological University, Belagavi

during the year 2024-2025.

It is certified that all corrections/suggestions indicated for Internal Assessment have been incorporated in the report deposited in the departmental library.

The project report has been approved as it satisfies the academic requirements in respect of the project work prescribed for the Bachelor of Engineering Degree.

_____	_____	_____
Signature of the Guide	***Signature of the HOD***	***Signature of the Principal***

Semester End Viva Voce Examination

Name of the Examiners	**Signature with Date**
1. _____	_____
2. _____	_____

ACKNOWLEDGEMENT

The success and outcome of our project required the effort, assistance, and guidance of many people. We would like to acknowledge and thank all the parties involved in completing this project; without their support, we wouldn't have been able to make it successful.

First and foremost, we would like to thank our beloved Principal **Dr. Niranjan N. Chiplunkar**, and our college N. M. A. M. Institute of Technology for providing the resources and facilities required to execute the project.

Our sincere thanks to **Dr. Sharada U Shenoy**, Head of the Department, Department of Artificial Intelligence and Machine Learning, NMAMIT, Nitte, for her support and valuable input.

We would like to express our deepest gratitude to our project guide **Dr. Rashmi Adyapady R.**, Asst. Professor Gd-III, Department of Artificial Intelligence and Machine Learning, NMAMIT, Nitte, for her constant encouragement, guidance, and suggestions for improvement throughout the duration of the project.

Our heartfelt thanks to the **Department of Artificial Intelligence and Machine Learning** staff members and our friends for their honest opinions and unconditional support.

We would like to take this opportunity to express our gratitude to everyone directly or indirectly involved in making this project successful.

Chethana R Kini

Manya Hegde

Reyona Elza Sabu

ABSTRACT

Large language models (LLMs) have revolutionized the field of natural language processing by enabling tasks like text generation, summarization, and language translation. However, their large size and high computational requirements make it difficult to deploy them on edge devices, which often have limited memory and processing power. This project aims to optimize LLMs so they can run efficiently on edge devices, making advanced AI capabilities more accessible in real-time, low-power environments.

We explored various optimization techniques, including model pruning, segmentation, evolutionary algorithms, and federated learning (FL). Model pruning was used to remove redundant parameters, effectively reducing the model size without compromising performance. Segmentation allowed us to break down the model into smaller segments for parallel processing across multiple devices, thus improving speed. Additionally, evolutionary algorithms were applied to fine-tune the optimization process, iteratively searching for the best configurations that enhance model efficiency. Federated learning further enabled distributed training without centralizing data, thereby increasing data privacy and reducing network traffic.

Our results showed that these methods significantly reduced the memory requirements and processing time of LLMs, making it feasible to run them on resource-constrained edge devices. This project demonstrates the potential for deploying powerful language models in more compact environments, opening new possibilities for AI applications in areas like mobile devices, IoT, and embedded systems.

TABLE OF CONTENTS

Title Page.....i
Certificate.....ii
Acknowledgement.....iii
Abstract.....iv
Table of Contents.....v
List of Figures.....vi
CHAPTER 1.....1
1.1. OVERVIEW.....1
1.2. PROBLEM STATEMENT.....1
1.3. OBJECTIVE.....1
1.4. MOTIVATION.....2
CHAPTER 2.....3
2.1. EXISTING SYSTEMS.....3
2.2. OBSERVATIONS FROM THE SURVEY.....7
2.3. PROPOSED SYSTEM.....9
2.4. SYSTEM DESIGN.....10
CHAPTER 3.....12
3.1. FUNCTIONAL REQUIREMENTS.....12
3.1.1. HARDWARE REQUIREMENTS.....12
3.1.2. SOFTWARE REQUIREMENTS.....12
3.2. NON-FUNCTIONAL REQUIREMENTS.....12
CHAPTER 4.....14
4.1. PYTORCH.....14
4.2. TRANSFORMERS.....14
4.3. SOCKET PROGRAMMING.....15
4.4. JUPYTER NOTEBOOK.....15

CHAPTER 5……..16

5.1. OPTIMIZATION……...16

5.1.1. GENETIC ALGORITHM……..17

5.1.2. BAT ALGORITHM……...17

5.1.3. PARTICLE SWARM INTELLIGENCE……..18

5.1.4. HYBRID APPROACH……..18

6.2. SEGMENTATION……..……19

6.2.1. APPROACH 1……...19

6.2.2. APRROACH 2……...20

CHAPTER 6…….21

6.1. OPTIMIZATION RESULTS……....21

6.2. SEGMENTATION RESULTS……22

CHAPTER 7……..25

7.1. CONCLUSION AND FUTURE WORK……..25

REFERENCES……..26

LIST OF FIGURES

Figure 2.1. A general framework for integrated research of LLMs and EAs.....3

Figure 2.2. PSO-CS Algorithm.....4

Figure 2.3. Dynamic Weight PSO (DWPSO-SVM).....5

Figure 2.4. PipeDream's layer partitioning.....6

Figure 2.5. Network Partitioning, Naïve Parallelism and Pipeline Parallelism.....7

Figure 2.6. Architecture of Proposed Methodology.....9

Figure 2.7. System Design.....10

Figure 5.1. Sequence Flow.....16

Figure 5.2. Pipeline Parallel.....19

Figure 5.3. Pipeline Parallelism with Asynchronous Mini-Batch.....20

Figure 6.1. Client Side.....23

Figure 6.2. Server Side.....23

Figure 6.3. Model Part 1 layers in Client Side.....23

Figure 6.4. Model Part 2 layers in Server Side.....24

Figure 6.5. User Interface for Server Communication.....24

CHAPTER 1
INTRODUCTION

1.1. OVERVIEW

Large Language Models (LLMs) are powerful AI tools used in tasks like text generation, translation, and understanding natural language. These models have transformed how machines interact with human language. However, due to their large size and high computing needs, running LLMs on smaller devices like phones and embedded systems is challenging.

This research explores ways to make LLMs work efficiently on these limited devices. To achieve this, several optimization techniques were applied. Evolutionary algorithms, such as Particle Swarm Optimization (PSO), Genetic Algorithms (GA), and Bat Algorithm (BA), were used to fine-tune model settings for improved performance within smaller resource limits.

Additionally, model pruning—specifically L1 pruning—was used to remove less important parts of the model, making it smaller and faster without sacrificing accuracy. Techniques like pipeline parallelism were also tested, breaking down models into parts so they could be processed across multiple devices, reducing wait times and improving speed.

Finally, segmentation techniques were explored to divide the model into manageable sections that can run in parallel across multiple devices. This segmentation makes it possible to handle large models more efficiently and provides a way to balance processing loads, ensuring smoother, faster performance on resource-limited devices.

1.2. PROBLEM STATEMENT

Design and development of optimized LLM's using Evolutionary Algorithms on Edge Devices.

1.3. OBJECTIVE

The main objectives of this project are:

1. **Optimize Large Language Models (LLMs):** To reduce the suitable for execution on devices with limited resources.
2. **Segment and Distribute Models:** To develop techniques for dividing models into smaller components and distributing the processing tasks, enabling efficient execution across multiple devices.
3. **Efficiently Deploy LLMs on Edge Devices:** To facilitate the deployment of LLMs on edge devices such as smartphones, IoT devices, and embedded systems, ensuring advanced AI capabilities are delivered effectively in real-time.

1.4. MOTIVATION

The motivation for this project comes from the growing need for advanced AI capabilities in real-time applications, especially in settings with limited computing resources. Large language models (LLMs) have shown great potential in tasks like text generation, translation, and conversational AI, but their size and high resource demands make it difficult to use them on edge devices like smartphones, IoT devices, and embedded systems. These devices have limited memory, processing power, and energy, creating a gap between what LLMs can do and what is practical for edge computing.

The project is driven by the need to bring AI closer to the user, particularly in areas like healthcare, smart cities, autonomous systems, and mobile apps, where quick decision-making is important. Using cloud-based systems for LLMs can lead to delays, connectivity problems, and privacy risks. Optimizing LLMs for edge devices aims to improve performance and responsiveness while keeping data secure by reducing reliance on external servers. Additionally, the project seeks to make AI more accessible in underserved regions where cloud access may be limited or unreliable, helping democratize technology and reach a wider audience.

CHAPTER 2
LITERATURE SURVEY

2.1. EXISTING SYSTEMS

According to [1], the combination of Large Language Models (LLMs) and Evolutionary Algorithms (EAs) offers a new way to improve artificial intelligence systems. LLMs can help EAs find better solutions to complex problems by using their knowledge to guide the search process. This partnership is useful in tasks like code generation, where LLMs create code and EAs optimize it for better performance. However, current systems still struggle with solving very complicated optimization problems. Therefore, more research is needed to enhance this collaboration and broaden its use in different areas of AI.

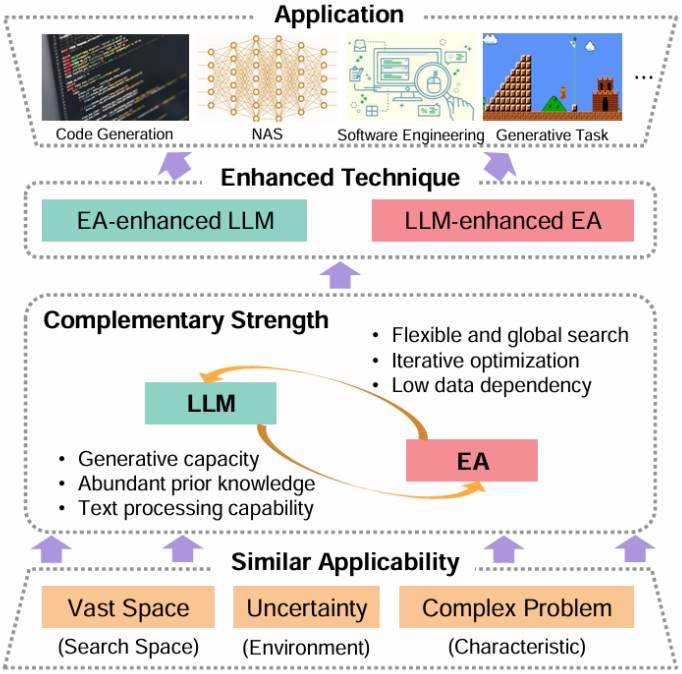

Figure 2.1. A general framework for integrated research of LLMs and EAs.

In Hybrid PSO-CS Algorithm [2], training feedforward neural networks (FNNs), algorithms like Particle Swarm Optimization (PSO) and Cuckoo Search (CS) are widely used. PSO offers fast convergence but can sometimes get stuck in local minima, while CS has strong global

optimization capabilities but slower convergence. To overcome these limitations, a hybrid PSOCS algorithm has been developed, combining the strengths of both PSO and CS. This hybrid algorithm strikes a balance between fast convergence and thorough exploration of the search space. In tests on benchmark problems, PSOCS-FNN demonstrated superior performance, achieving an accuracy rate of 0.948 on the Iris dataset, compared to 0.856 for PSO-FNN and 0.868 for CS-FNN. Additionally, PSOCS-FNN showed a more balanced trade-off between avoiding premature convergence and exploring the search space, making it a more reliable and accurate training method for FNNs.

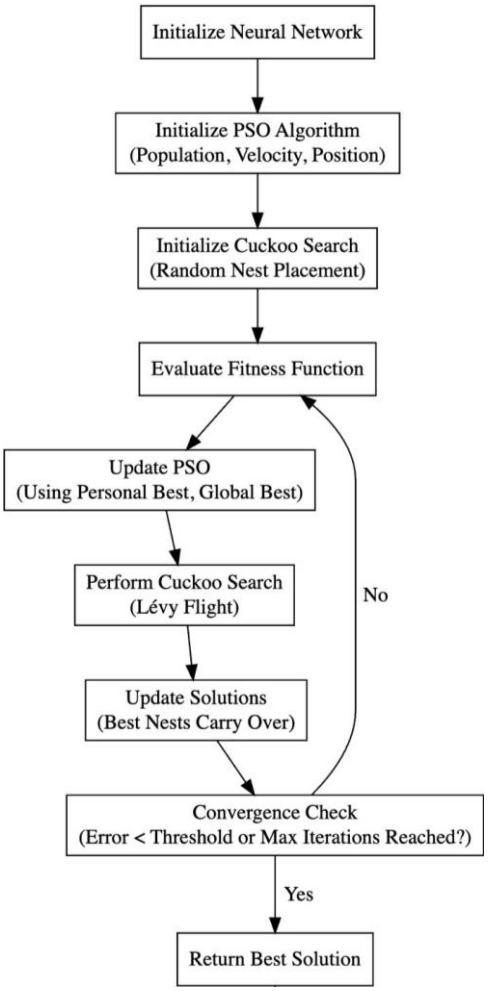

Figure 2.2. PSO-CS Algorithm

In existing systems for optimizing Support Vector Machines (SVMs), parameter tuning and feature selection are critical to improving

performance. A notable approach is the Feature Selection-score (FS-score) method, which enhances feature selection by promoting external sparsity and internal compactness within the class. This method is combined with an improved Particle Swarm Optimization (PSO) algorithm called Dynamic Weight PSO (DWPSO-SVM) in [3], which adjusts inertia weight nonlinearly during iterations to improve global search and prevent getting stuck in local minima. Experiments on UCI datasets, such as the Diabetes, Heart, and Sonar datasets, show that the DWPSO-SVM outperforms traditional PSO-SVM and other algorithms in terms of classification accuracy, TPR, and TNR. For example, DWPSO-SVM improves accuracy by 6.58% on the Vowel dataset and 4.8% on the Diabetes dataset. These results highlight that DWPSO-SVM effectively enhances the optimization of SVM classifiers, making it a superior choice for classification tasks.

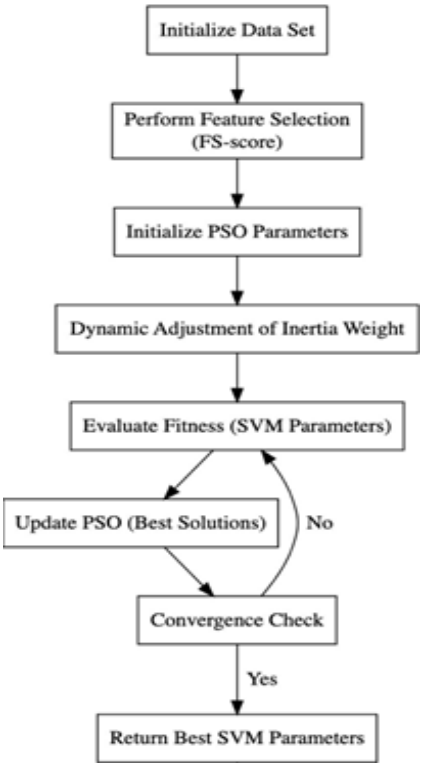

Figure 2.3. Dynamic Weight PSO (DWPSO-SVM)

PipeDream [4] is a distributed training system that optimizes the training of Deep Neural Networks (DNNs) across multiple GPUs through pipeline

parallelism, addressing the inefficiencies of data-parallel training for large models. This system minimizes communication-to-computation ratios by partitioning DNN layers across different machines and enabling asynchronous communication. PipeDream reduces communication by up to 95% compared to traditional data-parallel training methods, while simultaneously overlapping computation and communication to enhance performance. The system's ability to version model parameters for backward pass correctness and optimize forward and backward pass scheduling further accelerates the training process. Experiments with various DNNs on different GPU clusters have shown PipeDream to be up to five times faster than existing data-parallel approaches, thus providing a significant advantage in reducing the "time to target accuracy" for large DNNs.

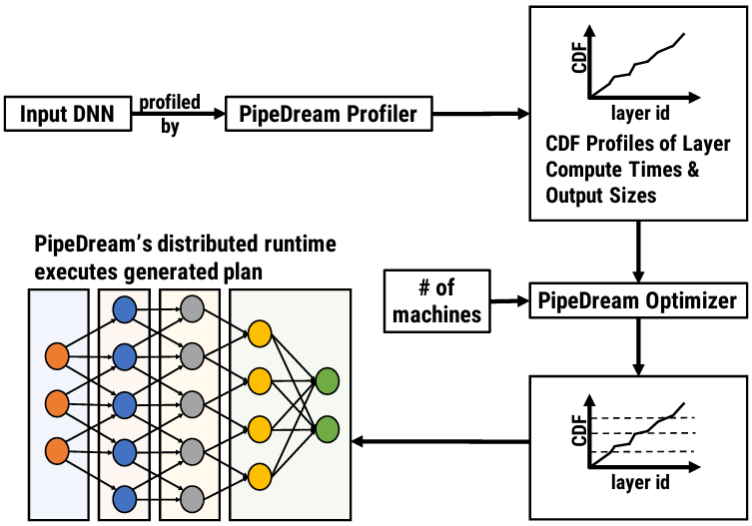

Figure 2.4. PipeDream's layer partitioning

GPipe [5] is a model parallelism framework developed by Google, specifically designed to facilitate the efficient training of large deep learning models across multiple devices, such as GPUs. By employing pipeline parallelism, GPipe distributes different layers of a neural network across various devices, allowing for simultaneous processing of multiple mini-batches of data. This approach not only mitigates memory constraints associated with large models but also enables gradient

accumulation, reducing communication overhead during training. GPipe seamlessly integrates with existing TensorFlow and PyTorch frameworks, providing automatic gradient computation and updates, which simplifies the implementation of distributed training. Its scalability allows researchers to experiment with models containing millions or billions of parameters, making it particularly valuable in high-demand fields like natural language processing and computer vision.

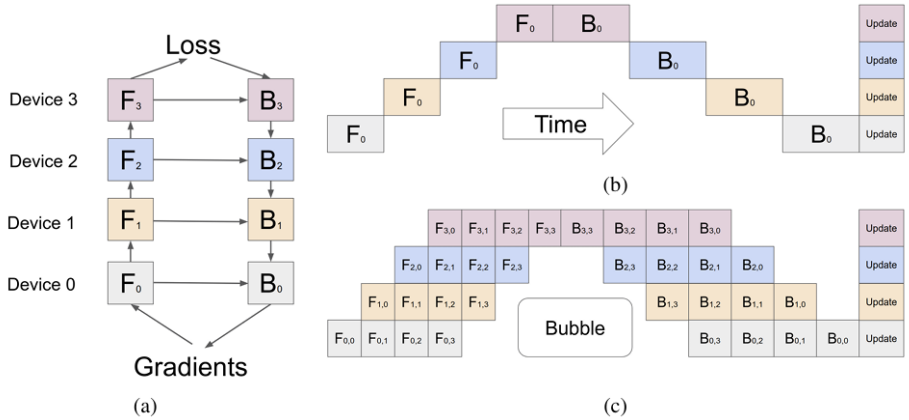

Figure 2.5. (a) Network Partitioning (b) Naive Parallelism (c) Pipeline Parallelism

2.2. OBSERVATIONS FROM THE SURVEY

The integration of Large Language Models (LLMs) with Evolutionary Algorithms (EAs) [1] demonstrates a promising approach to solving complex optimization problems. This synergy can enhance your project by leveraging LLMs to guide EAs in optimizing model performance and resource utilization. Focusing on this collaboration can lead to advancements in areas like model fine-tuning and feature optimization, thus improving overall system efficiency.

The hybrid PSOCS algorithm [2] for training feedforward neural networks (FNNs) illustrates the benefits of combining optimization techniques. By utilizing a mix of Particle Swarm Optimization (PSO) and Cuckoo Search (CS), you can achieve both fast convergence and thorough exploration of the search space. Implementing a similar hybrid approach in your

project could balance exploration and exploitation, leading to better performance in optimizing LLMs.

In the context of Support Vector Machines (SVMs), the Dynamic Weight PSO (DWPSO-SVM) [3] method emphasizes the importance of adaptive parameter tuning and feature selection. Applying these concepts to your LLM optimization efforts could enhance model accuracy and robustness. By integrating dynamic adjustments and feature selection strategies, you can improve the efficiency of your optimization process and achieve superior performance in complex tasks.

The PipeDream [4] system showcases how optimizing distributed training through pipeline parallelism can significantly reduce communication overhead while improving computation efficiency. Adopting similar strategies in your project may allow you to effectively distribute LLM training across multiple devices, thereby reducing time to target accuracy and enabling the training of larger models on resource-constrained environments.

The GPipe [5] framework demonstrates significant efficiency in scaling the training of large deep learning models by utilizing pipeline parallelism and gradient accumulation. These features allow GPipe to reduce training time and effectively manage memory constraints, enabling the handling of models with billions of parameters. The paper highlights GPipe's capability to enhance resource utilization, particularly in natural language processing and computer vision tasks, showcasing its potential to advance deep learning model complexity.

2.3. PROPOSED SYSTEM

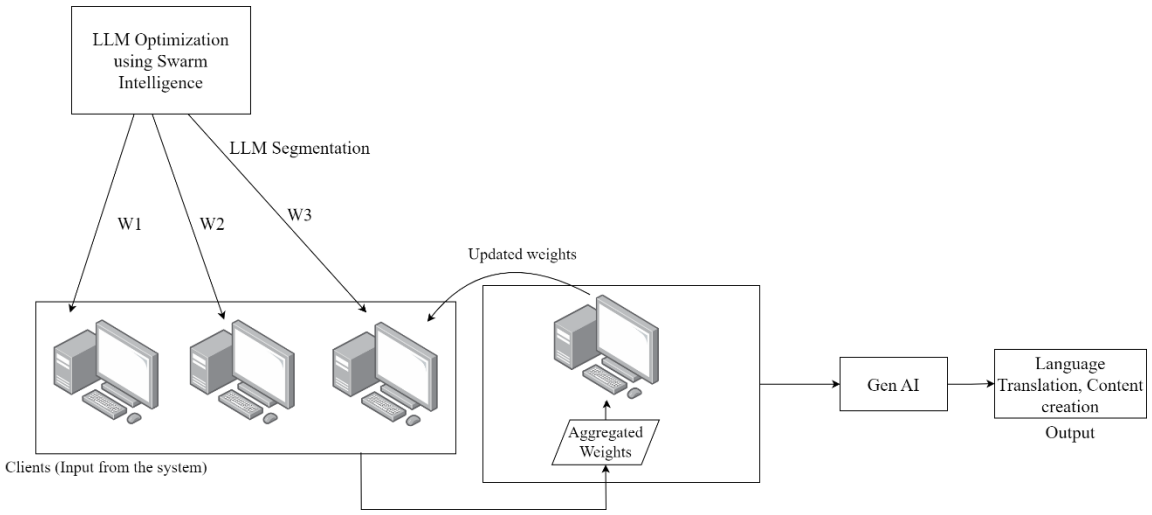

Figure 2.6. Architecture of Proposed Methodology

The first step involves optimizing LLMs with various evolutionary algorithms. These algorithms, such as Particle Swarm Optimization (PSO) and Genetic Algorithms (GA), fine-tune the model's parameters to reduce computational requirements, memory usage, and energy consumption. This optimization helps to make the models suitable for resource-limited environments.

After optimization, model pruning techniques are applied to further reduce the model size by removing less significant parameters. This step aims to simplify the model while maintaining accuracy, making it more lightweight and efficient for real-time inference on edge devices.

The pruned model is then segmented into smaller components, which are distributed across multiple edge devices. Each device processes a specific segment of the model, allowing parallel execution. This segmentation strategy enables effective utilization of distributed computing resources, reducing latency and improving throughput.

The results from each segment are aggregated to form a unified output. This step ensures that the segmented model performs as a cohesive unit,

delivering accurate and efficient results for tasks such as language translation, content generation, or summarization.

By combining optimization, pruning, segmentation, and aggregation, this methodology aims to make LLMs more feasible for deployment on edge devices, addressing limitations in computational power, memory, and energy resources.

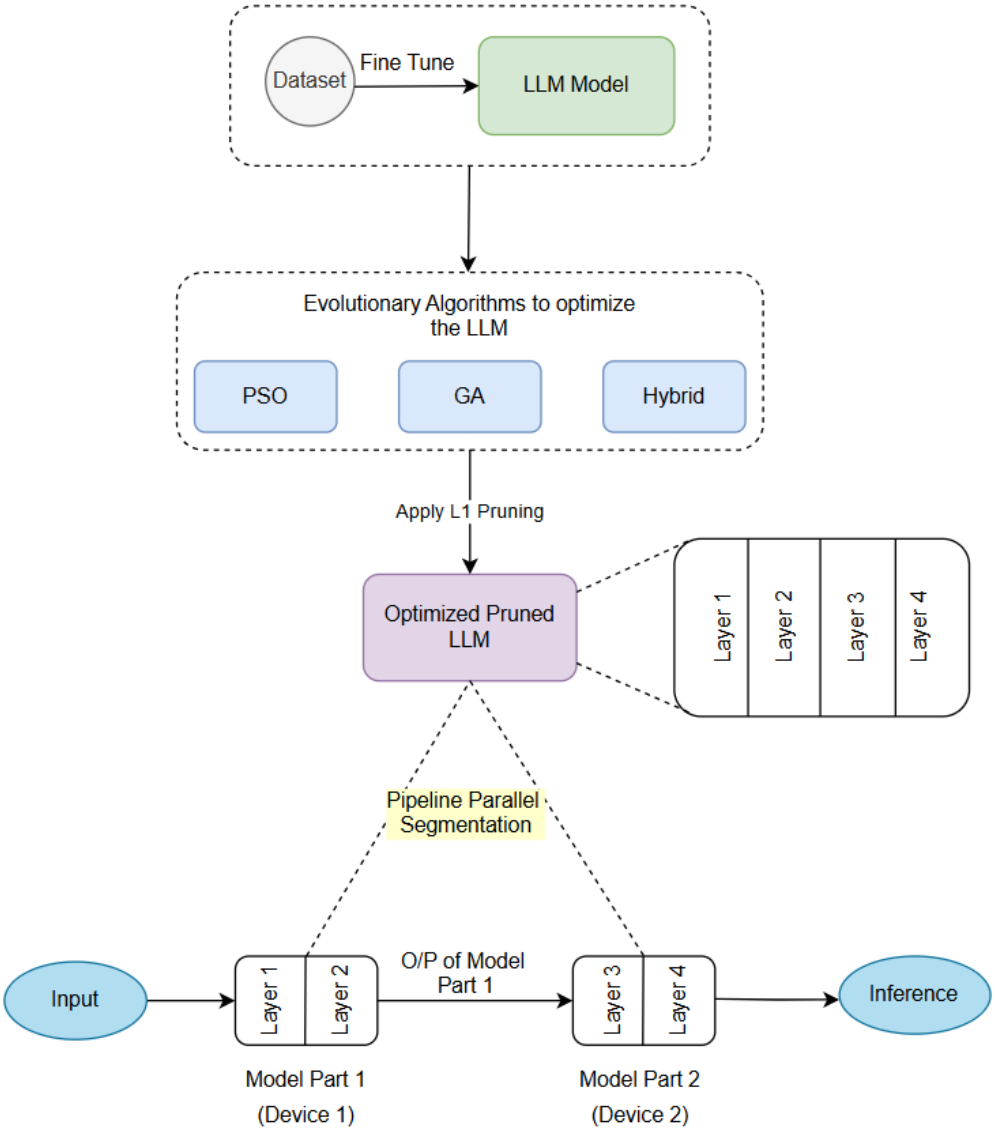

Figure 2.7. System Design

1. **Fine-Tuning:** A pre-trained LLM is adapted to specific tasks using a dataset, improving its performance and relevance for particular applications.

2. **Optimization:** To enhance efficiency, evolutionary algorithms such as Particle Swarm Optimization (PSO), Genetic Algorithm (GA), or a Hybrid approach are used. These techniques help refine the model to reduce resource usage and speed up computations.

3. **Pruning:** L1 pruning is applied to remove low-weight parameters, compressing the model and making it more efficient.

4. **Pipeline Parallel Segmentation:** The model is split across devices for parallel processing: Device 1 handles initial layers, while Device 2 completes the final layers, increasing processing speed.

5. **Inference:** During inference, each device processes its segment in sequence, achieving fast and efficient output by maximizing hardware capabilities.

CHAPTER 3
SYSTEM REQUIREMENTS

3.1. FUNCTIONAL REQUIREMENTS

3.1.1. HARDWARE REQUIREMENTS

- **System Type:** 64-bit Operating System
- **Processor Type:** x64-bit processor (CPU support is sufficient; GPU acceleration recommended for faster processing)
- **Operating System:** Windows 10 and above, or Ubuntu 20.04 and above, or macOS (latest version)
- **RAM:** Minimum 16 GB (32 GB recommended)
- **GPU (Optional):** NVIDIA GPU with CUDA support (RTX 3060 or higher recommended for acceleration)
- **Network Speed:** 100 Mbps and above for downloading datasets and dependencies

3.1.2. SOFTWARE REQUIREMENTS

- **Programming Language:** Python 3.8 and above
- **Development Environment:** Visual Studio Code, PyCharm, Google Colab, or Jupyter Notebook
- **Deep Learning Libraries:** PyTorch, TensorFlow, Transformers (Hugging Face)
- **Version Control:** Git, GitHub
- **Operating System Dependencies:** CUDA Toolkit (for GPU support)

3.2. NON-FUNCTIONAL REQUIREMENTS

- **Performance:**
    - Models should execute efficiently on edge devices without significant latency.

  - The system should be capable of processing real-time inputs, ensuring minimal delay.

- **Ease of Use:**
  - The code and documentation must be easy to follow for developers integrating the models into edge applications.
  - The setup process should be streamlined with clear instructions.
- **Scalability:**
  - The solution should be adaptable for different edge devices and configurations.
  - It should support scaling up to more powerful systems without major modifications.
- **Reliability:**
  - The models should deliver consistent performance across various edge devices.
  - Results should be reproducible given the same input data and configuration.
- **Maintainability:**
  - The codebase should be modular, allowing easy updates and modifications.
  - Dependencies should be regularly updated to ensure compatibility with the latest libraries.
- **Security:**
  - Sensitive data should not be transmitted to external servers, preserving user privacy.
  - Secure coding practices should be followed to avoid vulnerabilities in the code.

CHAPTER 4
SOFTWARE APPROACH

The software approach for this project involves the use of several key technologies and libraries to facilitate the optimization, segmentation, and deployment of large language models (LLMs) on edge devices. Each component plays a crucial role in achieving efficient on-device inference while addressing challenges such as limited computational resources and real-time processing requirements.

4.1. PYTORCH

PyTorch is a dynamic and flexible deep learning framework widely used for building and training machine learning models. In this project, PyTorch provides the essential tools for model optimization, pruning, and fine-tuning. Its dynamic computational graph allows for greater experimentation and rapid prototyping, making it ideal for implementing complex neural networks and custom training routines. PyTorch also supports distributed training, which is beneficial for segmenting models and running different parts across multiple devices.

4.2. TRANSFORMERS

Transformers, a library developed by Hugging Face, is utilized for working with state-of-the-art pre-trained models and fine-tuning tasks in natural language processing (NLP). It supports a variety of LLMs, including BERT, GPT, and T5, which can be adapted for specific applications. In this project, Transformers help manage tasks such as text generation, language translation, and summarization. The library simplifies model loading, tokenization, and the implementation of transformer-based architectures, enabling the integration of advanced language models on edge devices.

4.3. SOCKET PROGRAMMING

Socket programming is employed to enable communication between different devices or segments of the distributed model. It facilitates data transfer and synchronization between client devices hosting different parts of the model. By using sockets, the project ensures that intermediate outputs from one model segment can be passed to the next segment seamlessly. This approach is particularly useful in pipeline parallelism, where multiple devices collaborate to process data in a sequential manner.

4.4. JUPYTER NOTEBOOKS

Jupyter Notebooks are used for code development, experimentation, and visualization during the optimization and deployment phases. They offer an interactive environment that allows step-by-step debugging, real-time results analysis, and rapid iteration. The notebook format also aids in documenting the code, making it easier to share findings and collaborate.

CHAPTER 5
SYSTEM IMPLEMENTAION

5.1. OPTIMIZATION

The process begins with loading a large language model (LLM) and fine-tuning it using a specific dataset to improve its performance. After that, an optimization technique is chosen, and the necessary parameters are set up. The next step involves evaluating the model's performance, which guides further adjustments. Depending on the chosen approach, different operations are carried out to enhance the model. Additionally, pruning techniques are applied to simplify the model by removing less important components. This cycle repeats until predefined criteria for stopping are met, resulting in a more efficient and optimized LLM.

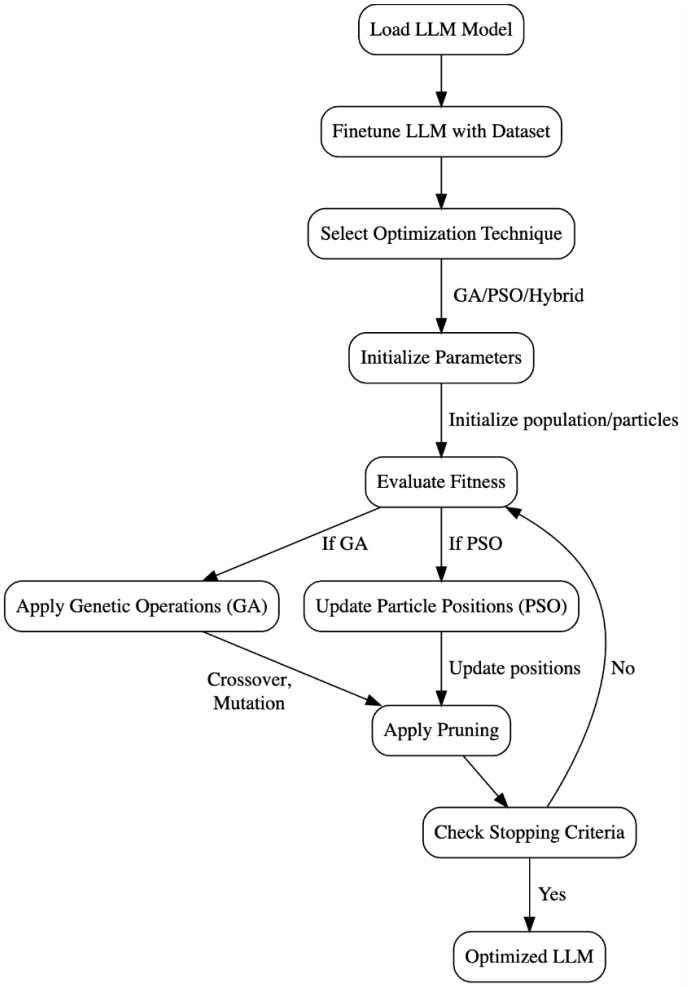

Figure 5.1. Sequence Flow

5.1.1. GENETIC ALGORITHM

The Genetic Algorithm (GA) was employed to optimize hyperparameters for Artificial Neural Networks (ANN), Convolutional Neural Networks (CNN), and the ALBERT LLM model using the IMDB dataset. The process started with initializing a population, where each individual represented a unique configuration of hyperparameters.

Each individual in the population was evaluated based on its performance using specific fitness functions. For this project, we employed Mean Squared Error (MSE) as well as a hybrid fitness function that combined MSE with Categorical Cross-Entropy (CCE).

After evaluating the individuals, the best-performing candidates were selected for reproduction. This selection process was designed to ensure that superior configurations had a higher probability of passing their traits to the next generation.

Crossover and mutation operations were then applied to the selected individuals to generate a new population. Crossover combined features from two parent solutions, while mutation introduced random changes to maintain diversity within the population.

This iterative process continued until a predefined stopping criterion was reached, allowing for the continuous refinement of model parameters and improvements in overall performance.

5.1.2. BAT ALGORITHM

The BAT Algorithm was employed to optimize the hyperparameters of an Artificial Neural Network (ANN) trained on the IMDB dataset. The implementation began by initializing a population of virtual bats, each representing a unique configuration of hyperparameters, including learning rate and number of hidden neurons.

Each bat's performance was evaluated using a fitness function, specifically the classification accuracy of the ANN on the IMDB dataset.

Based on their fitness scores, the bats adjusted their positions, simulating the echolocation behaviour of real bats.

5.1.3. PARTICLE SWARM INTELLIGENCE

Particle Swarm Optimization (PSO) is an optimization technique inspired by the social behaviour of birds, used to enhance the performance of machine learning models. By utilizing a group of candidate solutions, or particles, PSO explores the search space to find optimal hyperparameters, sharing information to converge on the best solutions.

PSO is applied to optimize the hyperparameters of Support Vector Machine (SVM), Convolutional Neural Network (CNN), Artificial Neural Network (ANN), and TinyBERT models, all fine-tuned on the IMDB dataset. For SVM, PSO searches for optimal kernel types and regularization parameters. In the case of CNN, it optimizes hyperparameters such as filter counts and dropout rates. For ANN, parameters like learning rate and hidden layer sizes are fine-tuned. Similarly, for TinyBERT, PSO optimizes parameters like learning rate and batch size, resulting in improved classification accuracy across all models.

5.1.4. HYBRID APPROACH

Hybrid optimization methods are also explored for Artificial Neural Network (ANN) models fine-tuned on the IMDB dataset by combining Particle Swarm Optimization (PSO) with Genetic Algorithms (GA) and Adam optimization. The PSO + GA hybrid facilitates enhanced exploration of the search space while ensuring effective convergence on optimal hyperparameters. In contrast, the PSO + Adam approach utilizes Adam's adaptive learning capabilities alongside PSO's swarm intelligence for dynamic adjustments during training. These hybrid methods lead to notable improvements in classification accuracy and overall performance of the ANN models.

5.2. SEGMENTATION

There are several segmentation approaches for optimizing large language models, but we chose pipeline parallelism because it's a newer method that efficiently divides the model into stages. This technique allows for sequential processing across different resources while maintaining continuous data flow between stages.

The pipeline parallel approach segments the LLM into multiple stages, where each stage processes a portion of the model's layers. By dividing the model in this way, the stages are executed sequentially, with each stage running on separate computational resources. The output from one stage serves as the input for the next, ensuring continuous data flow throughout the model. This segmentation allows for concurrent execution across devices or processing units, effectively overlapping computation and communication, optimizing resource utilization, and reducing overall latency during inference.

5.2.1. APPROACH 1

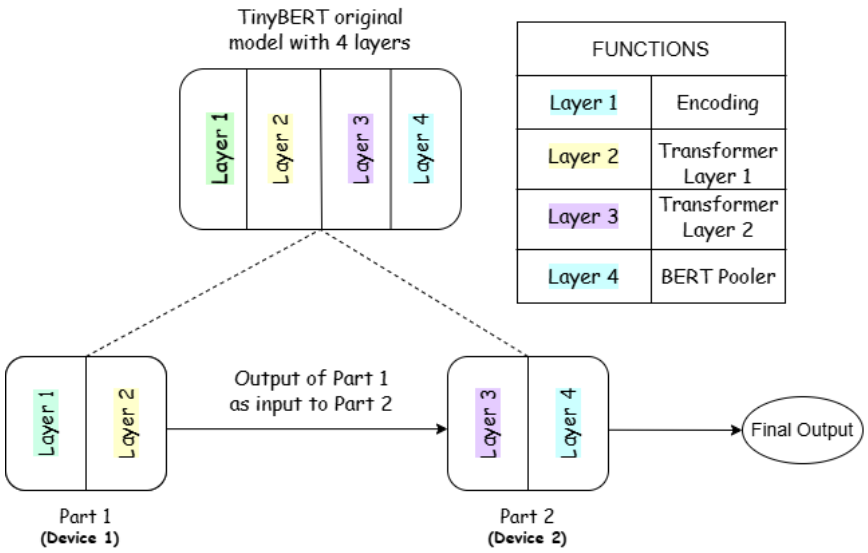

Figure 5.2. Pipeline Parallel

In the first approach, an optimized TinyBERT language model fine-tuned with the IMDB dataset is utilized. This model is enhanced through

Particle Swarm Optimization (PSO) and pruned using L1 pruning techniques. The optimized TinyBERT is divided into two parts: Model Part 1 is deployed on the client side, while Model Part 2 is implemented on the server. In this setup, the client processes incoming text reviews with Model Part 1 and sends the results to Model Part 2 on the server for final classification, determining whether each review is positive or negative.

5.2.2. APPROACH 2

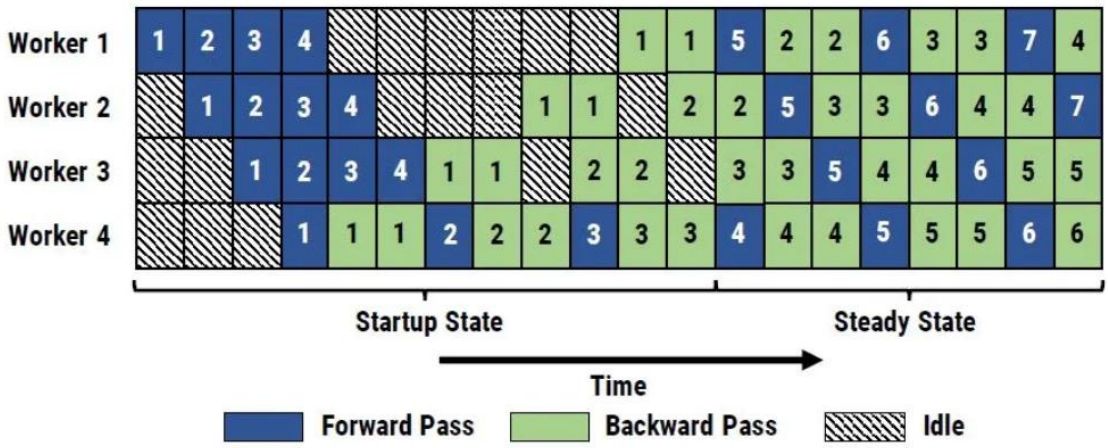

Figure 5.3. Pipeline Parallelism with Asynchronous Mini-Batch Processing

In the second approach, the TinyBERT model optimized with PSO and L1 pruning is employed, again split into two parts. This time, pipeline parallelism with microbatch overlapping is implemented during fine-tuning on the IMDB dataset. The dataset is divided into equal-sized microbatches, enabling efficient processing. Model Part 1 processes one microbatch and sends the output to Device 2 (Model Part 2) for further processing. While Device 2 handles the output from Device 1, Device 1 concurrently processes the next microbatch. This overlapping workflow maximizes resource utilization and improves overall processing speed. (Referred from PipeDream [4]; applied with only 2 devices in this approach).

CHAPTER 6
RESULTS

This section presents the results of optimizing various model architectures using different optimization techniques on the IMDB dataset. The models evaluated include Convolutional Neural Networks (CNN), Artificial Neural Networks (ANN), and TinyBERT. The performance of each model was measured in terms of test accuracy, time taken for optimization, and the number of generations or iterations required for convergence.

6.1. OPTIMIZATION RESULTS

1. Genetic Algorithm (GA)

The Genetic Algorithm was employed to optimize the model weights through a natural selection-inspired process. The results indicate varying levels of performance across the different architectures, as shown in Table 1.

Model	Accuracy
ANN	56%
CNN	51%

Table 1

2. Bat Algorithm (BA)

The Bat Algorithm optimized the models by simulating echolocation. As shown in Table 2, this method provided competitive accuracy and efficiency across the architectures tested.

Model	Accuracy
ANN	85%

Table 2

3. Particle Swarm Optimization (PSO)

The PSO approach utilized particle interactions to explore the weight space. The results indicate its effectiveness in optimizing all model types, as summarized in Table 3.

Model	Accuracy
ANN	87.49%
CNN	74%
SVM	88.7%
TinyBERT	85%

Table 3

4. Hybrid Approach

The results for the Hybrid approach combining different optimization techniques are summarized below in the Table 4.

Model	Optimization Technique used	Accuracy
ANN	PSO + Adam	85.34%
ANN	PSO + GA	85.42%

Table 4

6.2. SEGMENTATION RESULTS

Approach 1

In this approach, the client sends input prompts and processes them through several layers of the model. The output from the last layer on the client is then sent to the server, which contains the remaining layers. The server processes this output through the remaining layers to produce the final result. In this case, the output is a sentiment analysis classification, determining whether the review is positive or negative.

```
C:\Users\Exam\.conda\envs\project\Lib\site-packages\tqdm\auto.py:21: TqdmW
ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
Enter a sentence (or type 'done' to finish):  movie was bad
Enter a sentence (or type 'done' to finish):  it was amazing
Enter a sentence (or type 'done' to finish):  movie was good
Enter a sentence (or type 'done' to finish):  done
Asking to truncate to max_length but no maximum length is provided and the
Intermediate output sent to server.
```

Figure 6.1. Client Side

```
C:\Users\Exam\.conda\envs\Project\Lib\site-packages\tqdm\auto.p
ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
Server listening on 0.0.0.0:10300...
Connected to client at ('172.16.16.77', 50558)
Intermediate output received from client.
Intermediate output shape: torch.Size([3, 5, 128])
tensor([0, 1, 1])
Connection closed.
```

Figure 6.2. Server Side

```
Layer 0: BertEmbeddings(
  (word_embeddings): Embedding(30522, 128, padding_idx=0)
  (position_embeddings): Embedding(512, 128)
  (token_type_embeddings): Embedding(2, 128)
  (LayerNorm): LayerNorm((128,), eps=1e-12, elementwise_affine=True)
  (dropout): Dropout(p=0.1, inplace=False)
)
Layer 1: BertLayer(
  (attention): BertAttention(
    (self): BertSdpaSelfAttention(
      (query): Linear(in_features=128, out_features=128, bias=True)
      (key): Linear(in_features=128, out_features=128, bias=True)
      (value): Linear(in_features=128, out_features=128, bias=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
    (output): BertSelfOutput(
      (dense): Linear(in_features=128, out_features=128, bias=True)
      (LayerNorm): LayerNorm((128,), eps=1e-12, elementwise_affine=True)
      (dropout): Dropout(p=0.1, inplace=False)
    )
  )
  (intermediate): BertIntermediate(
    (dense): Linear(in_features=128, out_features=512, bias=True)
    (intermediate_act_fn): GELUActivation()
  )
  (output): BertOutput(
    (dense): Linear(in_features=512, out_features=128, bias=True)
    (LayerNorm): LayerNorm((128,), eps=1e-12, elementwise_affine=True)
    (dropout): Dropout(p=0.1, inplace=False)
  )
)
```

Figure 6.3. Model Part 1 layers in Client Side

```
ModelPart2(
  (layers): ModuleList(
    (0): BertLayer(
      (attention): BertAttention(
        (self): BertSdpaSelfAttention(
          (query): Linear(in_features=128, out_features=128, bias=True)
          (key): Linear(in_features=128, out_features=128, bias=True)
          (value): Linear(in_features=128, out_features=128, bias=True)
          (dropout): Dropout(p=0.1, inplace=False)
        )
        (output): BertSelfOutput(
          (dense): Linear(in_features=128, out_features=128, bias=True)
          (LayerNorm): LayerNorm((128,), eps=1e-12, elementwise_affine=True)
          (dropout): Dropout(p=0.1, inplace=False)
        )
      )
      (intermediate): BertIntermediate(
        (dense): Linear(in_features=128, out_features=512, bias=True)
        (intermediate_act_fn): GELUActivation()
      )
      (output): BertOutput(
        (dense): Linear(in_features=512, out_features=128, bias=True)
        (LayerNorm): LayerNorm((128,), eps=1e-12, elementwise_affine=True)
        (dropout): Dropout(p=0.1, inplace=False)
      )
    )
    (1): BertPooler(
      (dense): Linear(in_features=128, out_features=128, bias=True)
      (activation): Tanh()
    )
  )
  (classifier): Linear(in_features=128, out_features=2, bias=True)
)
```

Figure 6.4. Model Part 2 layers in Server Side

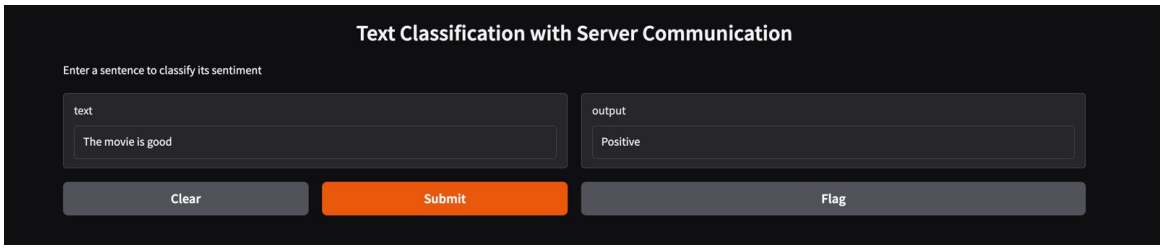

Figure 6.5. User Interface for Server Communication

Approach 2

In the second approach, pipeline parallelism is combined with micro-batching. This approach maintains an accuracy of 78.8%, consistent with the results from Approach 1. The implementation of micro-batching ensures better overlap between computation and communication, contributing to enhanced overall efficiency in the training process.

CHAPTER 7
CONCLUSION AND FUTURE WORK

7.1. CONCLUSION AND FUTURE WORK

Various optimization techniques were implemented and evaluated for improving the performance of a fine-tuned TinyBERT model on the IMDB dataset. Using Particle Swarm Optimization (PSO), better model accuracy and enhanced training efficiency were achieved, demonstrating the effectiveness of PSO in optimizing model parameters. Additionally, pipeline parallelism was explored to address hardware limitations, focusing on Approach 1, which involved partitioning the model across multiple GPUs.

The results show that Approach 1 of pipeline parallelism successfully reduced memory usage and training time, allowing for more efficient scaling of the model. Overall, the combination of PSO and pipeline parallelism offers a practical solution for optimizing large language models, especially in environments with limited computational resources.

Future work aims to explore more advanced model partitioning strategies for optimizing the TinyBERT model. Specifically, plans include splitting the model into segments based on task-specific requirements and computational power. While the current implementation utilized only two devices, there is significant potential to expand this approach to multiple devices, which would further enhance parallelism and efficiency.

Additionally, model partitioning based on the computational power of edge devices will be explored, using mathematical formulas to optimally distribute the layers. Another avenue for exploration includes task-specific partitioning, where model layers are allocated based on the specific demands of the task being performed. These strategies will help improve the scalability and adaptability of large language models across diverse computing environments.

REFERENCES

[1] Wu, X., Wu, S. H., Wu, J., Feng, L., & Tan, K. C. (2024). Evolutionary computation in the era of large language model: Survey and roadmap. arXiv preprint arXiv:2401.10034.

[2] Chen, J. F., Do, Q. H., & Hsieh, H. N. (2015). Training artificial neural networks by a hybrid PSO-CS algorithm. Algorithms, 8(2), 292-308.

[3] Wang, J., Wang, X., Li, X., & Yi, J. (2023). A hybrid particle swarm optimization algorithm with dynamic adjustment of inertia weight based on a new feature selection method to optimize SVM parameters. Entropy, 25(3), 531.

[4] Harlap, A., Narayanan, D., Phanishayee, A., Seshadri, V., Devanur, N., Ganger, G., & Gibbons, P. (2018). Pipedream: Fast and efficient pipeline parallel dnn training. *arXiv preprint arXiv:1806.03377*.

[5] Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, M. X., Chen, D., ... & Chen, Z. (2019). GPipe: Easy scaling with micro-batch pipeline parallelism. proceeding of Computer Science> Computer Vision and Pattern Recognition.

[6] Imteaj, A., Thakker, U., Wang, S., Li, J., & Amini, M. H. (2021). A survey on federated learning for resource-constrained IoT devices. *IEEE Internet of Things Journal*, *9*(1), 1-24.

[7] Fajemisin, A. O., Maragno, D., & den Hertog, D. (2024). Optimization with constraint learning: A framework and survey. *European Journal of Operational Research*, *314*(1), 1-14.

[8] Feng, S., Chen, Y., Zhai, Q., Huang, M., & Shu, F. (2021). Optimizing computation offloading strategy in mobile edge computing based on swarm intelligence algorithms. *EURASIP Journal on Advances in Signal Processing*, *2021*, 1-15.