Towards Efficient Fine-Tuning of LLMs in Edge-to-Cloud Environments

Sertan Pekel*, Müge Sayıt*,†
* International Computer Institute, Ege University, İzmir, Türkiye
† Computer Science and Electronic Engineering, University of Essex, Colchester, UK
E-mails: sertan.pekel@dhmi.gov.tr, muge.sayit@essex.ac.uk, muge.sayit@ege.edu.tr

***Abstract*—The rapid evolution of Large Language Models (LLMs) has revolutionized Artificial Intelligence (AI) applications across various domains. In particular, fine-tuning LLMs—essential for adapting models to domain-specific tasks—requires specialized strategies for distributed environments. In this paper, we present a novel framework for LLM fine-tuning in an edge-to-cloud continuum. By leveraging real-time network metrics and edge compute resources, an AI orchestrator dynamically manages workload distribution by selectively eliminating underperforming nodes to optimize fine-tuning efficiency. Experimental results demonstrate that our approach significantly reduces completion time while preserving model convergence, offering an energy-efficient solution for edge-enabled LLM fine-tuning.**

***Index Terms*—SDN, edge computing, LLMs**

I. Introduction

Recent advancements in Artificial Intelligence (AI), particularly the rapid emergence of Large Language Models (LLMs) such as GPT, LLaMA, and Google Gemini, have significantly accelerated research and development across a wide range of domains. The success of these models is largely attributed to the transformer architecture and the availability of massive-scale training datasets. Running LLMs on distributed systems has become essential due to their enormous data requirements and hundreds of millions—or even billions—of parameters. However, conventional job scheduling strategies developed for general deep learning models may not be well-suited for LLMs, which exhibit unique computational and memory characteristics. As a result, novel workload partitioning and scheduling techniques are required to efficiently support LLM tasks in distributed environments [1].

As interest in and adoption of LLMs continue to expand, the operational challenges associated with their deployment are becoming increasingly critical. In particular, the high computational demands of LLMs—driven by their reliance on AI accelerators—pose significant obstacles to delivering cost-effective services with consistent Quality of Service (QoS) [2]. LLM processes create extensive workload on datacenters, costly, and cause users to suffer from long service latency [3]. Consequently, there is an urgent need to develop efficient serving architectures that minimize resource consumption while enhancing user experience [4]. Edge computing is emerging as a promising solution in this direction. Recent technological advancements are also contributing to this trend. For instance, some smartphone models now feature Qualcomm's Snapdragon 8 Gen 3 chipset [5], which includes Neural Processing Units (NPUs) designed to efficiently execute AI models. Another recent development is Google's Gemini Nano, an AI model specifically designed to run on mobile hardware.

Fine-tuning of LLMs involves adapting pre-trained models to specific downstream tasks or domains. Unlike pre-training, which is computationally intensive and requires large-scale data and resources, fine-tuning is relatively lightweight, as it typically involves updating only a small subset of parameters while keeping the majority of the model frozen. Therefore, implementing fine-tuning on a set of edge devices can be a suitable alternative to solve the long service latency problem.

In this paper, we propose a framework for LLM fine-tuning in an edge-to-cloud paradigm. The framework employs an AI orchestrator, implemented as a northbound application of a Software Defined Networking (SDN) controller, to coordinate the fine-tuning process by leveraging edge resources. Fine-tuning operations can be distributed through model or data parallelism, each offering distinct advantages. To date, only a limited number of studies have investigated LLM-related activities at the network edge, highlighting the novelty of our approach. The contributions of our study are as follows:

- We introduce a novel framework that leverages emerging network paradigms—such as SDN and edge-to-cloud architectures—to enhance AI workloads, demonstrating how networking can actively support and optimize large-scale AI operations.
- Unlike existing work that primarily investigates model partitioning for LLM fine-tuning at the edge, this study is, to the best of our knowledge, the first to explore data-parallel fine-tuning of LLMs across edge devices.
- We develop a lightweight scheduling algorithm that dynamically selects edge nodes by accounting for heterogeneous computational capabilities and network characteristics, thereby improving both efficiency and robustness.
- Demonstrate through experiments that selectively utilizing edge resources—rather than employing all available devices—can achieve superior performance, underscoring the importance of deliberate edge resource allocation.

The remainder of the paper is organized as follows. Section II provides background on LLM workflows and reviews related work on LLM deployment in distributed architectures. Section III presents the proposed system in detail. Experimental results are discussed in Section IV, and conclusions are drawn in Section V.

978-3-903176-75-1 ©2025 IFIP

II. Background and Related Work

A. Background on LLM

LLMs are a subclass of generative AI that uses transformer-based architectures to process natural language and generate contextually relevant outputs. Since the introduction of the transformer model in [6], it has become the foundation for state-of-the-art LLMs such as GPT, BERT, and Claude; thanks to its attention mechanisms and layered structure.

LLMs typically contain hundreds of millions to billions of parameters, requiring significant computational resources for training, fine-tuning, and inference—often beyond the capabilities of standard computing environments, necessitating data center infrastructure.

An LLM's lifecycle includes pre-training on large general-purpose corpora, fine-tuning on task-specific data, which is optional, and inference—where the model generates outputs based on learned representations.

1) Pre-training: The primary objective of training an LLM is to predict the next token in a sequence. LLMs generate one token at a time, where a token may represent a word, sub-word, or character, depending on the tokenizer used. Each token is first mapped to a high-dimensional vector (embedding), which is then passed through multiple transformer layers equipped with self-attention mechanisms. These mechanisms allow the model to weigh the relevance of each token in the input relative to others, enabling contextual understanding.

After passing through the transformer layers, the model's output is compared against the actual token in the training data [6]. The difference between the predicted and actual tokens is quantified using a loss function, which measures the model's error. This loss is then used to compute gradients, which guide the update of model parameters using optimization techniques such as Stochastic Gradient Descent (SGD).

This process—forward pass, loss computation, backpropagation, and weight update—repeats billions of times across vast datasets until the model converges, i.e., reaches a point where further training yields minimal improvement.

2) Fine-tuning: Fine-tuning is a key phase in LLM development, where a pre-trained model is adapted to a specific task or domain. Unlike pre-training—which is resource-intensive and requires massive datasets—fine-tuning typically uses smaller, domain-specific data [7]; enabling models to gain specialized knowledge while preserving general linguistic capabilities [8].

The model's architecture remains unchanged, but its weights are updated using new training examples, which may include domain-specific terms, structured formats, or user content. This allows the model to perform better in niche contexts without retraining from scratch. Since pre-trained models already capture rich language representations, fine-tuning usually converges faster and demands fewer resources.

The process is influenced by hyperparameters such as learning rate, batch size, and number of epochs. A high learning rate may lead to unstable updates, while a low rate slows convergence. Smaller batch sizes are common in fine-tuning due to memory constraints and the need for more precise updates. Multiple epochs are often required to adapt effectively to task-specific data.

These hyperparameters are critical to balancing efficiency, accuracy, and resource usage when fine-tuning LLMs, particularly in resource-constrained or specialized deployment environments.

3) Inference: The inference phase involves deploying the trained or fine-tuned model to generate outputs in response to user inputs or external queries. Inference is latency-sensitive and often executed in real-time environments, which makes efficient resource utilization and low-latency design critical. Depending on deployment requirements, inference may occur on high-performance servers, edge devices, or mobile platforms [9].

B. LLM Training, Fine Tuning, and Inference over Distributed Infrastructures

Training and fine-tuning tasks for LLMs are typically parallelized using data parallelism, model parallelism, or hybrid approaches. In data parallelism, the dataset is divided into smaller subsets, each processed independently by different computing nodes. In contrast, model parallelism partitions the model itself, with distinct segments handled by separate devices. Since the training, fine-tuning, and inference stages of LLMs exhibit different computational and communication characteristics, the distributed execution of these phases requires customized strategies.

Given the traffic patterns and scalability demands of LLM training, recent studies have focused mainly on data center-oriented solutions. For example, a dual-plane Top-of-Rack (ToR) switch architecture is proposed in [10] to reduce path optimization complexity and enable efficient routing in large-scale training environments.

The challenges of deploying personalized LLMs in resource-constrained environments are explored in [11], where various model compression techniques are evaluated based on their impact on edge performance. In [12], the authors proposed a Deep Reinforcement Learning (DRL) based algorithm to optimize the offloading decisions of inference at the edge.

Several studies have addressed fine-tuning LLMs at the edge. For instance, NetGPT [13] proposes a model-splitting framework to enable AI-native functionalities in cellular networks. Similarly, EdgeShard [2] dynamically allocates LLM shards across edge devices to minimize inference latency and maximize throughput using a dynamic programming algorithm. Energy efficiency in Federated Learning (FL) systems, especially in light of emerging regulations such as the EU AI Act, was explored in [14], where edge devices are connected to the fine-tuning process in the first epoch.

The Edge-LLM framework [15] introduces a server–edge collaboration strategy for inference and fine-tuning. Their design includes a quantization method that accounts for memory overhead and inference time, along with a mechanism for caching feature maps from Graphics Processing Unit (GPU) to Central Processing Unit (CPU) to enhance efficiency.

Unlike previous work, our approach focuses on data parallelism for the fine-tuning process at the edge. One key advantage of data parallelism is that it avoids the need for continuous server–edge communication, which is often required in model-parallelism-based methods. In the following section, we present the details of our proposed framework.

III. Proposed Model

LLM training and fine-tuning are complex processes, and new task management or communication orchestration techniques are needed on edge-to-cloud environments, even for data centers. In a heterogeneous environment such as edge-to-cloud continuum, most of the edge devices have limited capacity, battery and computing power—compared to the cloud devices. Therefore, their resources should be efficiently managed.

A. Edge-to-cloud Platform for LLM Fine-Tuning

In this work, we focus on federated fine-tuning of LLMs across a distributed edge devices. Fine-tuning tasks may serve a variety of purposes, such as analyzing real-time traffic imagery for vehicle classification or for detecting anomalies in sensor-data streams from Internet of Things (IoT) devices at manufacturing plants.

For our proposed method, we assume the network is managed by an SDN controller and the fine-tuning process is coordinated by an AI orchestrator—which works as a northbound application of the controller. An illustration of the overall framework is shown in Fig. 1. As seen from the figure, at the lowest layer, there are edge devices with different characteristics. The SDN controller at the middle layer provides network statistics to the AI orchestrator.

We assume that a third-party organization initiates a fine-tuning request based on a specific application domain and provides the AI orchestrator the specific database that will be used in fine-tuning the model along with a list of accessible edge devices and their characteristics, such as computing power. Such a scenario may arise, for example, when the organization disseminates an application through which edge devices can enroll, thereby enabling the third party to systematically collect information regarding the participating devices. In the proposed framework, the SDN controller periodically measures end-to-end bandwidth, delay, and packet loss between the AI orchestrator and the edge nodes. These network metrics are retrieved by the AI orchestrator via the northbound API of the SDN controller.

B. Fine-Tuning Workflow

Once the fine-tuning process begins, the AI orchestrator distributes the learning model to the edge nodes, along with the dataset partitions which is provided by the fine-tuning process initiator. Upon receiving its partition, each node integrates it with its previously-stored local data for fine-tuning. We assume that this local data acquisition takes place prior to the fine-tuning process.

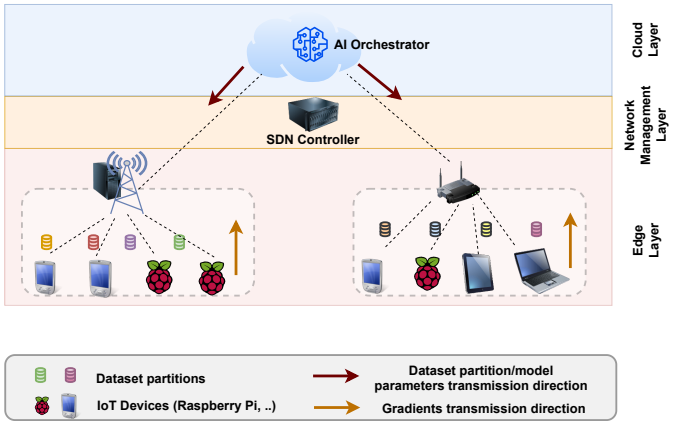


Fig. 1: The illustration of the proposed architecture

Let the network consist of $|N|$ edge devices, where N denotes the set of devices. We define P as the data partitioning factor, with $1 \leq P \leq |N|$. In our architecture, we adopt a partitioning strategy analogous to full model sharding, where each edge device holds approximately $1/|N|$ of the dataset. We assume that the provided dataset is uniform and our algorithm shards the dataset regardless from its characteristics. Although this study considers equal dataset partitioning across devices, this is not a strict requirement of the proposed framework, which can seamlessly accommodate alternative dataset distribution strategies.

After the dataset and model have been distributed, the AI orchestrator initiates the epochs. The overall procedure for distributed fine-tuning is outlined in Algorithm 1. In line 2, the initial $epoch\_loss$ is set to the loss measured upon the completion of pre-training. The AI orchestrator sends the model and dataset partitions to all active edge nodes, where then each node performs a forward pass over the data and returns gradient sets. During this process, each node also records its training time and sends this value along with the gradients to the orchestrator. The orchestrator waits until all gradients are received (line 6) and computes their sum (line 7), which is later used in the SGD calculation.

Between lines 8 and 10, the AI orchestrator evaluates each node scoring their performance, which is described at III-C. In order to select the set of edge nodes, the AI orchestrator must take into account several characteristics of the nodes such as their computing power and training time, as well as their connection parameters such as bandwidth, delay and packet loss rate. The best set of nodes are the ones having the highest capacity and bandwidth while having minimum training time, delay and loss rate.

After sorting nodes based on their performance scores (line 9), the number of nodes to eliminate is determined using Eq. 1. In the formula, T denotes the number of epochs, with the constraint $|N| \geq T$.

$$\left\lfloor \frac{N-1}{T-1} \right\rfloor \tag{1}$$

The list of active nodes is updated (line 11). The orchestrator then aggregates gradients SGD, updates model parameters according to calculated_sgd value and computes the loss reduction (lines 12-15). The update magnitude formula which is calculated at line 12, is the implementation of federated average formula which is exactly equivalent to that in the centralized SGD calculation.

Finally, in lines 16-17, the orchestrator sends the updated model parameters to all nodes, including those eliminated from gradient computation, to maintain model consistency across the network, therefore, each node can run inference operations.

C. Performance Scoring

We define performance scoring as a multi-objective combination of compute and network metrics and define a utopia point representing the optimal solution. The orchestrator assigns performance score to each node in the system according to its distance to the utopia point. Let C_i represent the compute power score of node i, B_i represent end-to-end bandwidth between node i and AI orchestrator, D_i is the network delay between node i and the AI orchestrator, L_i denote the packet loss rate on the path between the node i and the orchestrator, and finally, $T_i^{(t)}$ is the training duration of node i at epoch t.

For each $i \in N$, each value of node i is normalized with min-max normalization. Normalized variables are weighted according to their importance with the inner product (Eq. 2), where $\mathbf{w}$ and $\overrightarrow{\mathrm{N_i}}$ represents the non-negative weights ($\sum_k w_k = 1$) and the set of normalized variables at epoch t, respectively.

$$s_i^{(t)} = \left\langle \mathbf{w}, \overrightarrow{\mathrm{N_i^{(t)}}} \right\rangle \tag{2}$$

Let $\lambda \in [0,1]^5$ denote the utopia point. The performance score of each node i at epoch t is calculated by using Eq. 3.

$$\lambda_s = \langle \mathbf{w}, \lambda \rangle \, , \mathit{PerfScore}_i^{(t)} = 1 - ||s_i^{(t)} - \lambda_s|| \tag{3}$$

Nodes are ranked by $\mathrm{PerfScore}_i^{(t)}$ and the lowest-scoring nodes are eliminated. The number of nodes—which are eliminated is determined by using Eq. 1.

D. Scalability and Applicability of the Proposed System

The proposed framework integrates multiple components, including SDN technology and the utilization of IoT devices. For real-world, large-scale deployment scenarios, two primary challenges arise: *(i)* collecting real-time data related to network and device characteristics, and *(ii)* managing the energy consumption of edge devices. To obtain network characteristics, we periodically measure key parameters via the controller's southbound API and apply smoothing techniques to stabilize the values. Our results demonstrate that this method is sufficiently effective to support real-time video streaming, a latency—and quality-sensitive Internet application that depends heavily on accurate and timely network measurements [16]. In scenarios where edge devices are distributed over wide geographic areas and exist in large numbers, data collection can be efficiently managed through the use of multiple controllers and/or hierarchically organized controllers.

Algorithm 1: Node-Elimination Algorithm

Input: LEARNING_RATE, PRE_TRAINING_LOSS, EPOCH_COUNT

1 Initialize **edge_list**, **active_nodes**, **epoch_loss**, **eliminated_nodes**, **gradient_sum**, **calculated_sgd**;
2 epoch_loss ← PRE_TRAINING_LOSS;
3 **foreach** *node* ∈ *edge_list* **do**
4 Send dataset partitions and the model to $node$;
5 **for** $epoch \leftarrow 0$ **to** *EPOCH_COUNT* **do**
6 Wait until all gradients are received;
7 $gradient\_sum \leftarrow \sum received\_gradient\_size$;
8 Measure performance scores based on Eq. 3;
9 Sort edge_performances ascending;
10 Eliminate n number of weakest nodes using Eq. 1
11 active_nodes ← active_nodes - eliminated_nodes;
12 update_magnitude ← LEARNING_RATE * (1/$|active\_nodes|$) * gradient_sum;
13 calculated_sgd ← calculated_sgd - update_magnitude;
14 param_size ← Updated parameter size depending on calculated_sgd;
15 epoch_loss ← Model predicted values - Ground truth values;
16 **foreach** *node* ∈ *edge_list* **do**
17 Send param_size to $node$;
18 FINAL_LOSS ← epoch_loss;

1) Scalability Analysis: Let $|N|$ denote the total number of nodes, and let $|N|_t$ represent the number of active nodes at epoch t. In each epoch, the number of active nodes decreases due to the elimination process. The algorithm executed by the AI orchestrator consists of two main components: (i) exchanging messages between the orchestrator and the edge nodes, and (ii) determining the set of nodes to be eliminated in each epoch.

During epoch t, one message is sent to each active device and one reply is received, resulting in a message complexity of $O(|N|_t)$. The performance scoring step requires a linear computation, $O(|N|_t)$, followed by the ordering of scores, $O(|N|_t \log |N|_t)$. Therefore, the time complexity per epoch is $O(|N|_t \log |N|_t)$.

Over the full training process with T epochs, the total time complexity is

$$O(\sum_{t=1}^{T} |N|_t log|N|_t) \tag{4}$$

Since $|N|_t$ is non-increasing with t due to the elimination strategy, the total complexity is strictly lower than $O(T|N| \log |N|)$, which represents the worst-case bound when no nodes are eliminated.

In the proposed framework, the elimination function only excludes a device from gradient aggregation for consecutive epochs; the device retains its local data and continues to receive model updates, enabling it to perform inference and to rejoin training in subsequent epochs. This strategy also enhances energy efficiency at the edge, as non-participating nodes conserve resources by avoiding computationally intensive fine-tuning operations.

IV. Experimental Results

A. Experimental Setup

To evaluate the performance of the proposed approach, we conducted experiments in a Mininet environment, where the network is managed by an SDN controller. We used Ryu as the SDN controller. The experimental topology is an enhanced version of the one shown in Fig. 1, where edge devices are connected to switches representing gateway routers of their respective domains, and an additional layer of switches connects the gateway routers to the server hosting the AI orchestrator. To introduce heterogeneity, we assign computational powers to edge devices drawn from a distribution with an average of 30 and a standard deviation of 15.

To define the hyperparameters for LLM fine-tuning, we scaled down the data volume used in data parallelism (All-Reduce method) following the approach in [10], and used it to determine the training dataset size. The reference model in our study is GPT-3 (175B parameters). It is important to note that, in this study, we did not perform actual large-scale LLM fine-tuning. Instead, we artificially modeled realistic LLM traffic patterns, following the methodology proposed in [17]. This modeling approach has been shown to capture the essential communication and computation characteristics of real LLM activities, while enabling scalable experimentation without prohibitive computational costs. By adopting this methodology, our framework produces scientifically valid and realistic results in terms of communication overhead, scalability, and system performance.

We employed the All-Reduce method due to its alignment with our gradient computation pattern, in which each node contributes local gradients to a reduction operation (e.g., sum, min, max), and the aggregated result is broadcast to all nodes. Likewise, we adopted the reported data volume for tensor parallelism (involving both All-Reduce and All-Gather) to define the model size, since all nodes maintain a complete copy of the model after each operation.

Hyperparameter settings (e.g., batch size, learning rate, and number of epochs) were adopted from prior studies [11], [13], [14] to ensure realistic and comparable evaluation conditions. Specifically, we set the learning rate to 5×10^{-5}, the number of epochs to 10, the model size to 500 MB, and the dataset size to 1000 MB. For evaluation purposes, we did not use a task-specific dataset; instead, we generated a synthetic dataset composed of random characters.

To simulate diverse network conditions, we configured three experimental scenarios representing normal, limited, and high-capacity (prolific) environments. In these scenarios, end-to-end bandwidth, delay, and loss values were assigned following an exponential distribution, with average values summarized in Table I. In the table, the Avg. Bandwidth value represents the average available bandwidth of the links, while Avg. Delay value represents the average delay value of the end-to-end path between the AI orchestrator and the edge devices. Note that, the bandwidth and delay values show the values where there is no traffic. With the fine-tuning operation traffic, these values dynamically change during the experiments. Therefore the results presented here represent the test with dynamic network environment.

TABLE I: Network Configurations

	Avg. Bandwidth	**Avg. Delay**	**Loss**
Setup 1	16 Mbps	58 ms	1%
Setup 2	7.5 Mbps	78.5 ms	1%
Setup 3	21 Mpbs	19 ms	1%

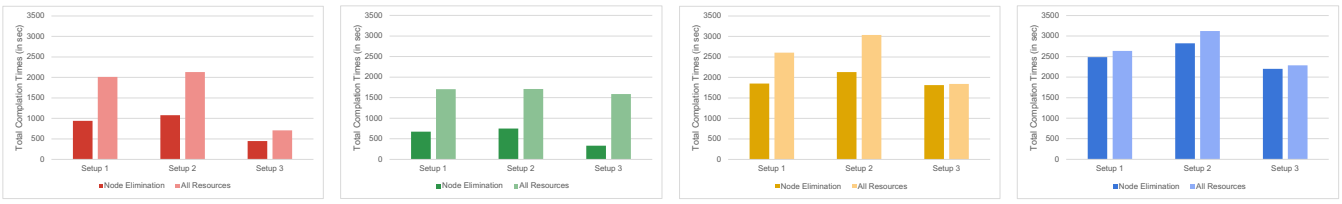
B. Performance Evaluations

In order to provide comparable results, we also implemented an alternative approach where node elimination is not applied. This baseline shares the same characteristics as our proposed framework, except for the node elimination strategy. In the evaluation graphs presented in this section, our framework is labeled *Node Elimination (NE)*, while the baseline is labeled *All Resources (AR)*.

In Fig. 2, we present the total completion times for experiments conducted with 12 to 40 edge nodes. Completion time is measured from the moment the AI orchestrator sends the dataset and model to the nodes until the final loss value is computed. As observed in the results, particularly in scenarios with limited resources and bandwidth, the proposed *NE* method completes training up to twice as fast as the *AR* approach. These findings highlight the significant negative impact that lower-performing edge devices can have on training time. Interestingly, utilizing only a carefully selected subset of nodes—rather than all available resources—can yield substantial performance gains.

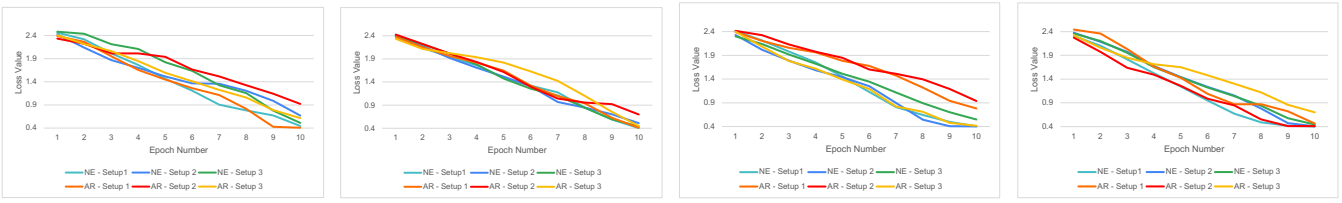
However, minimizing completion time does not necessarily imply successful training. To assess learning quality, we also measured the cross-entropy loss at each epoch for all scenarios. As shown in Fig. 3, the loss values obtained with the *NE* method are consistently comparable to—or slightly better than—those from the *AR* approach. Both methods reduce the initial loss from approximately 2.4 to a final convergence near 0.4, demonstrating that our elimination strategy preserves model accuracy while significantly improving computational efficiency.

Our study presents practical insights for real-world deployment where SDN assisted node elimination is used in fine-tuning. Overall, the results provide strong empirical evidence supporting performance-based optimization in resource-constrained, distributed environments.



(a) 12 edge devices (b) 20 edge devices (c) 32 edge devices (d) 40 edge devices

Fig. 2: Total completion times



(a) 12 edge devices (b) 20 edge devices (c) 32 edge devices (d) 40 edge devices

Fig. 3: Loss per epoch

V. Conclusion

In this study, we propose a framework for efficient and scalable fine-tuning of LLMs across heterogeneous edge devices within an edge-to-cloud continuum. The fine-tuning process is managed by a cloud-based AI orchestrator, deployed as a northbound application of an SDN controller. By leveraging network state information collected via the controller, the orchestrator applies a performance-aware elimination strategy that incrementally excludes underperforming edge nodes based on a composite performance score, which is calculated by using the metrics collected by the controller. This score incorporates compute capacity, bandwidth, delay, and packet loss.

Experimental results show that the proposed method significantly reduces the overall training time—by up to 79.2% while preserving convergence behavior, outperforming the baseline approach that does not include node-elimination.

This work offers early insights into the potential of data-parallel LLM fine-tuning at the edge. As future work, we aim to investigate joint optimization of routing and distributed fine-tuning using SDN capabilities with hybrid parallelism strategies.

References

[1] J. Duan *et al.*, "Efficient training of large language models on distributed infrastructures: A survey," 2024. [Online]. Available: https://arxiv.org/abs/2407.20018

[2] M. Zhang, X. Shen, J. Cao, Z. Cui, and S. Jiang, "Edgeshard: Efficient llm inference via collaborative edge computing," *IEEE Internet of Things Journal*, vol. 12, no. 10, pp. 13 119–13 131, 2025.

[3] G. Qu, Q. Chen, W. Wei, Z. Lin, X. Chen, and K. Huang, "Mobile edge intelligence for large language models: A contemporary survey," *IEEE Communications Surveys Tutorials*, pp. 1–1, 2025.

[4] Y. Wang *et al.*, "Burstgpt: A real-world workload dataset to optimize llm serving systems," 2025. [Online]. Available: https://arxiv.org/abs/2401.17644

[5] "Qualcomm snapdragon 8 gen 3 mobile platform," acc.: 2025-06-07. [Online]. Available: https://www.qualcomm.com/products/mobile/snapdragon/smartphones/snapdragon-8-series-mobile-platforms/snapdragon-8-gen-3-mobile-platform

[6] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," ser. NIPS'17, 2017, p. 6000–6010.

[7] Q. Hu, Z. Ye, Z. Wang, G. Wang, M. Zhang, Q. Chen, P. Sun, D. Lin, X. Wang, Y. Luo, Y. Wen, and T. Zhang, "Characterization of large language model development in the datacenter," 2024. [Online]. Available: https://arxiv.org/abs/2403.07648

[8] Y. Hong, X. Yin, X. Wang, Y. Tu, Y. Guo, S. Duan, W. Wang, L. Fang, D. Wang, and H. Zhu, "Keep the general, inject the specific: Structured dialogue fine-tuning for knowledge injection without catastrophic forgetting," 2025. [Online]. Available: https://arxiv.org/abs/2505.00029

[9] S. Park, S. Jeon, C. Lee, S. Jeon, B.-S. Kim, and J. Lee, "A survey on inference engines for large language models: Perspectives on optimization and efficiency," 2025. [Online]. Available: https://arxiv.org/abs/2505.01658

[10] K. Qian *et al.*, "Alibaba hpn: A data center network for large language model training," ser. ACM SIGCOMM '24, 2024, p. 691–706.

[11] R. Qin *et al.*, "Empirical guidelines for deploying llms onto resource-constrained edge devices," 2024. [Online]. Available: https://arxiv.org/abs/2406.03777

[12] J. Fang, Y. He, F. R. Yu, J. Li, and V. C. Leung, "Large language models (llms) inference offloading and resource allocation in cloud-edge networks: An active inference approach," in *VTC2023*, 2023, pp. 1–5.

[13] Y. Chen, R. Li, Z. Zhao, C. Peng, J. Wu, E. Hossain, and H. Zhang, "Netgpt: An ai-native network architecture for provisioning beyond personalized generative services," *IEEE Network*, vol. 38, no. 6, 2024.

[14] H. Woisetschläger, A. Erben, S. Wang, R. Mayer, and H.-A. Jacobsen, "Federated fine-tuning of llms on the very edge: The good, the bad, the ugly," ser. DEEM '24, 2024, p. 39–50.

[15] F. Cai, D. Yuan, Z. Yang, and L. Cui, "Edge-llm: A collaborative framework for large language model serving in edge computing," in *ICWS 2024*, 2024, pp. 799–809.

[16] S. Bikas and M. Sayıt, "Improving qoe with genetic algorithm-based path selection for mptcp," *IEEE Transactions on Network and Service Management*, vol. 21, no. 4, pp. 3874–3888, 2024.

[17] J. Santos, P. Maniotis, C. Wang, A. Tantawi, O. Tardieu, T. Wauters, and F. De Turck, "Evaluating the network effects of orchestration strategies for ai workloads in modern data centers," in *2025 IEEE 11th International Conference on Network Softwarization (NetSoft)*, 2025, pp. 285–293.