TRIBHUVAN UNIVERSITY

INSTITUTE OF ENGINEERING

PULCHOWK CAMPUS

A REPORT ON DE-LLMIT:DECENTRALIZED LARGE-LANGUAGE MODEL INFERENCE AND TUNING

SUBMITTED BY:

AAVASH CHHETRI (PUL077BCT004)

KUSHAL PAUDEL (PUL077BCT039)

MUKTI SUBEDI (PUL077BCT048)

SUBMITTED TO:

DEPARTMENT OF ELECTRONICS & COMPUTER ENGINEERING

March 20, 2025

Acknowledgments

We would like to express our heartfelt gratitude to the **Department of Electronics and Computer Engineering** at Pulchowk Campus for providing us with the invaluable opportunity to undertake our major project. The unwavering support, guidance, and encouragement from the department have been instrumental in shaping the trajectory of our work.

Special appreciation goes to **Assistant Prof. Bibha Sthapit** for her inspiring words, encouragement, and guidance. Her belief in our potential has been a significant motivator, and we are truly thankful for her continuous support throughout this endeavor.

The feedback and suggestions from the department have greatly contributed to refining our approach and enhancing our perspective on the project. We are confident that with the continued support from the Department of Electronics and Computer Engineering and the guidance of Assistant Prof. Bibha Sthapit, our project will continue to grow and succeed.

Thank you to the department and Assistant Prof. Bibha Sthapit for their pivotal roles in the early stages of our project. We eagerly look forward to further collaboration and the discoveries that lie ahead.

Abstract

The traditional process of Large Language Model (LLM) inference and fine tuning includes use of large cluster of heavy usage GPUs, including the cluster of Nvidia A100s and H100s GPUs. This can be replaced using a large cluster of distributed edge computing resources which can be used to infer LLMs using low-powered and economic edge resources. The project aims to develop a decentralized inference network designed to perform AI model inference and fine tuning using consumer grade devices. De-LlMIT leverages peer-to-peer (P2P) and multi layered architecture for the inference and fine tuning of large language models. By distributing model inference tasks across participating nodes, De-LlMIT mitigates the need for centralized servers, thereby reducing latency and operational costs while enhancing scalability. Also, the system aims to incorporate WebGPU-based technologies for efficient DNN execution and adapts model partitioning strategies to optimize performance across diverse browser configurations. This project addresses critical challenges in decentralized AI inference, including security, scalability, and efficient resource utilization in browser-based environments.

Keywords: Decentralized AI, Federated Learning, Peer-to-Peer Networks, WebGPU, Browser-Based Inference, Volunteer Computing

Contents

List of Figures

List of Abbreviations

LLM	Large Language Model
P2P	Peer-to-Peer
GPU	Graphics Processing Unit
DHT	Distributed Hash Table
DNN	Deep Neural Network
NAT	Network Address Translation
STUN	Session Traversal Utilities for NAT
TURN	Traversal Using Relays around NAT
BIN	Browser Instance Node
TIN	Terminal Instance Node
MCN	Monitor Client Node
ONNX	Open Neural Network Exchange
RAM	Random Access Memory
vRAM	Video Random Access Memory
FFN	Feed-Forward Network
GPT	Generative Pre-trained Transformer
LLAMA	Large Language Model Meta AI
HPC	High-Performance Computing
HTTP	Hypertext Transfer Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
AI	Artificial Intelligence
API	Application Programming Interface
ML	Machine Learning
BLOOM	BigScience Large Open-science Open-access Multilingual Language Model
H100	NVIDIA Hopper-100 GPU
A100	NVIDIA Ampere-100 GPU

1. Introduction

The rapid advancements in artificial intelligence and large language models (LLMs) have revolutionized natural language processing, data analytics, and human-computer interaction. However, centralized LLM inference methods face challenges such as scalability, latency, privacy concerns, and high hardware requirements, making them inaccessible to many users and individual researchers. To address these issues, we introduce ”De-Llmit: Decentralized Large Language Model Inference and Tuning,” which leverages decentralized computing to perform LLM inference directly within consumer devices. By distributing model computations across a network of nodes, De-Llmit enhances scalability, and protects user privacy without the need for centralized infrastructure or complex installations. This approach democratizes access to LLMs, enabling efficient and private model inference on consumer-grade hardware, thus redefining the accessibility and usability of AI technologies.

1.1 Background

Large language models (LLMs) like GPT-4 and LLaMA are increasingly complex, with models such as GPT-4 having 175 billion parameters and requiring multiple gigabytes of storage even with reduced precision (16-bit). Managing such models demands high-end hardware and incurs significant costs[1]. Personal and home computers are typically inadequate for these models due to their substantial memory and computational requirements. A 175B model requires about 350GB of memory in fp16 settings just to load the whole model. A typical consumer has neither VRAM nor RAM required to store the model. Research by Li et al. [2] indicates that consumer-grade systems lack the necessary resources for effective use of large-scale models. Researchers at [3] ran BLOOM-176B which took 5.5 seconds per token using fast RAM offloading and 22 seconds per token using SSD offloading which is unusable for many purposes.

Decentralized AI approaches offer a solution by leveraging distributed computing power from end-user devices. Studies, such as Chen et al. [4], show that this method can alleviate centralized server burdens, reduce costs, and make AI technology more accessible.

1.2 Problem statements

Some of the problems with the traditional approach:

1. **Centralized Servers**: Traditional AI inference relies heavily on centralized servers, leading to single points of failure, high latency, and dependence on server availability

and performance.

2. **Memory Constraints**: LLMs like LLaMA-2 or GPT variants require hundreds of gigabytes of memory to store model parameters. Consumer-grade GPUs and CPUs often lack sufficient memory to load the entire model.

$$Required\ Condition : RAM + vRAM \geq Model\ Memory\ Requirement$$

3. **Computation Time**: Even if the model fits into memory, performing inference on large models requires significant computational power, leading to longer inference times and higher latency.

4. **Privacy Issues**: Centralized AI systems require sending user data to external servers, raising significant privacy and data security concerns.

5. **Installation Difficulties**: Many AI solutions require complex installation processes and hardware setups, making them inaccessible to users with limited technical expertise.

6. **Computational and Economic Overhead**: Performing AI inference on a single computer can lead to high computational overhead, while cloud-based solutions incur substantial economic costs for continuous usage and scaling.

1.3 Objectives

Our objective is to try to overcome the drawbacks of the traditional system of model training by building a peer-to-peer network solution comprising of the following main features:

1. An Model Parallel System: Develop a system that enables LLM inference and finetuning in an decentralized environment by sharding the LLM model and loading partial models in different devices to carry out inference and tuning by using an exhaustive set of clients that form a whole model.

2. A Decentralized Model inference and tuning: Develop a system that enables AI language model inference directly within web browsers or terminal instances, eliminating the need for centralized servers. This system leverages DHT technologies to ensure fault tolerance, reliability and scalability. It utilizes the computational power of client devices, ensuring efficient and seamless inference processes and tuning.

1.4 Scope

1. Distributed AI Inference: Develop AI models that run directly within terminal instances or web browsers, utilizing WebGPU and WebAssembly. Ensure low-latency, efficient inference for real-time applications.

2. Client Finetuning: Develop a system to enable finetuning by using decentralized computing powers of the client terminals in the decentralized network.

3. Decentralized Network: Create a network of computers consisting of three layered architecture that can handle inference, fine tuning in the client terminal layer, fault tolerance and communication between terminals to exchange metadata and logits.

4. Ease of Use and Accessibility: Enable easy participation for users with different technical backgrounds. Promote widespread adoption through user-friendly interfaces and minimal setup requirements.

2. Literature Review

2.1 Related Theory

2.1.1 Volunteer Computing

1. The paper *Hivemind: Decentralized Deep Learning for Large-Scale Models* [5] introduces a decentralized framework for training and serving large-scale models across a distributed network of devices. It addresses the limitations of centralized AI training by enabling multiple participants to collaboratively train models without sharing their raw data. Hivemind leverages peer-to-peer communication, consensus algorithms, and advanced synchronization techniques to coordinate training across nodes, making it possible to scale deep learning tasks beyond the capacity of single organizations or data centers

2. *Petals: Collaborative Inference of Large Models like BLOOM* [3] presents a framework for distributed inference of large language models through a collaborative peer-to-peer network. It allows users to run inference on models, such as BLOOM, by connecting to a network where each participant hosts parts of the model. This decentralized approach reduces the computational burden on individual users by sharing the workload across multiple nodes, making it feasible to run large models even on limited hardware. Petals aims to make advanced model inference accessible, scalable, and more cost-effective by leveraging community-driven participation.

3. *Folding@Home* [6] describes a distributed computing project that utilizes the idle computational power of millions of volunteer devices worldwide to simulate protein folding and study molecular dynamics. By distributing small computational tasks to participants' machines, the project tackles complex problems in biology and medicine, such as understanding protein misfolding in diseases like Alzheimer's and developing potential treatments. This collaborative model accelerates research by aggregating vast amounts of processing power, demonstrating the potential of decentralized computing to solve large-scale scientific challenges.

2.1.2 Distributed Machine Learning: Federated Learning and Model-Distributed ML

1. The article *The Future of Consumer Edge-AI Computing* by Stefanos Laskaridis et al. [7] explores the trajectory and potential of edge AI computing, particularly focusing on consumer applications. The paper discusses the challenges and opportunities associated with deploying AI models on edge devices, emphasizing the need for efficient resource management and model optimization. The authors provide a comprehensive overview of current technologies and future directions, highlighting the impact of edge AI on various consumer sectors.

2. The paper *Federated Learning: Challenges, Methods, and Future Directions* [8] provides a comprehensive overview of federated learning, a decentralized approach where multiple clients collaboratively train a global model without sharing their raw data. It addresses key challenges such as data heterogeneity, privacy preservation, communication efficiency, and security concerns. The paper reviews various methods to tackle these issues, including advanced aggregation techniques, differential privacy, and secure multi-party computation

3. *Model-Distributed Machine Learning* [9]explores a distributed approach to training and deploying machine learning models by partitioning models across multiple devices or nodes. Unlike traditional data-parallel methods, this approach distributes parts of the model itself, allowing each node to handle only a portion of the overall computation. This model distribution can optimize resource usage, reduce memory requirements, and enable training and inference on hardware with limited capacity. The paper discusses the architecture, challenges, and potential benefits of this approach, emphasizing its ability to scale large models efficiently and enhance collaboration in multi-device environments.

2.1.3 Distributed Hash Tables and Networking

1. *Kademlia: A Peer-to-Peer Information System Based on the XOR Metric* [10] introduces the Kademlia DHT, a fundamental framework for peer-to-peer (P2P) networks. The authors present an efficient XOR-based routing mechanism for locating data across a decentralized network. This routing approach is crucial for enabling the DHT in systems like *De-Llmit*, where peers need to exchange intermediate model outputs efficiently.

2. *P2P Networking and DHTs in Decentralized Systems* [11] examines the use of peer-to-peer (P2P) networks and Distributed Hash Tables (DHTs) as foundational technologies for decentralized systems. It explores how P2P networking enables direct communication between nodes without relying on centralized servers, enhancing scalability and fault tolerance. DHTs are highlighted as crucial for efficient data lookup and resource management in these networks, providing a structured way to distribute and retrieve information across nodes.

3. **WebSocket in Decentralized and Volunteer Computing:** Decentralized frameworks rely on efficient communication protocols to coordinate distributed tasks. For instance, Hivemind[5] leverages peer-to-peer communication for collaborative model training across devices. While the paper focuses on consensus algorithms, WebSocket's persistent connections and low overhead make it ideal for synchronizing model updates and gradients in real time, minimizing delays caused by repeated HTTP handshakes. Similarly, Folding@Home[6] distributes computational tasks across volunteer devices, where WebSocket's ability to maintain open channels ensures rapid task allocation and result aggregation, critical for scaling complex simulations like protein folding.

2.1.4 Large Language Models: Memory Requirements, Quantization, and Challenges

1. *Scaling Up Language Models: GPT-3* [12] introduces GPT-3, one of the largest and most powerful language models ever created, with 175 billion parameters. It demonstrates how scaling up model size significantly enhances the performance of language models across a wide range of tasks without task-specific training. The paper explores GPT-3's capabilities in natural language understanding and generation, showcasing its ability to perform complex tasks such as translation, question answering, and code generation with minimal prompts.

2. *Quantization and Efficient Inference in Large Models* [13] discusses techniques for reducing the computational and memory demands of large language models through quantization. Quantization involves converting model weights and activations from high-precision formats to lower precision, such as 8-bit integers, without significantly compromising accuracy. This process enables more efficient inference, making it feasible to deploy large models on resource-constrained devices like mobile phones and edge servers.

3. *Memory-Efficient Training of Transformers* [14]presents strategies to optimize the memory usage of transformer models during training. It focuses on techniques such as

mixed precision training and gradient checkpointing, which reduce the memory footprint while maintaining performance. By enabling the training of larger models or increasing batch sizes without requiring extensive hardware resources, these methods make it feasible to train state-of-the-art transformers efficiently.

2.1.5 Machine Learning in Browser Based Environment

1. The paper *WebInf: Accelerating WebGPU-based In-browser DNN Inference via Adaptive Model Partitioning* [15] proposes a novel approach to optimize deep neural network (DNN) inference within web browsers using WebGPU. It introduces adaptive model partitioning, which dynamically splits models between the client device and external servers based on available resources and network conditions. WebInf aims to enable efficient and scalable DNN inference directly in the browser, making advanced AI applications more accessible without relying on powerful local hardware.

2. *WebDNN: Fastest DNN Execution Framework on Web Browser* [16] by Mitsuhisa Sato et al. introduces WebDNN, a framework designed to execute deep neural network (DNN) models efficiently within web browsers. Leveraging technologies like WebAssembly and WebGPU, WebDNN achieves remarkable performance improvements, making it the fastest known framework for in-browser DNN execution at the time. This advancement opens up new possibilities for deploying machine learning models directly to users without the need for server-side computation.

2.2 Related work

1. On-Device Machine Learning with TensorFlow.js Smilkov et al. (2019) explored the capabilities of TensorFlow.js for running machine learning models within web browsers. Their work demonstrated that complex neural networks could be executed efficiently on client-side devices, paving the way for decentralized inference applications. The study highlighted the potential for interactive and real-time AI applications without relying on centralized servers [17].

2. Blockchain-Based Incentive Mechanisms for Decentralized Computing Zheng et al. (2018) proposed a blockchain-based framework to incentivize users to share their computational resources. This approach ensures transparency and security in tracking contributions and distributing rewards. The study provided insights into designing effective incentive schemes, which are essential for the success of decentralized inference systems [18].

3. Federated Learning: Challenges, Methods, and Future Directions Kairouz et al. (2021) reviewed the advancements in federated learning, discussing its potential to preserve data privacy and leverage distributed computational resources. Although focused on training, the principles of federated learning are relevant to decentralized inference, emphasizing the benefits of local data processing and decentralized architectures [8].

4. Efficient Decentralized Computing for AI Applications Li et al. (2021) investigated various decentralized computing frameworks for AI applications, analyzing their performance and scalability. The study highlighted the advantages of decentralized systems in handling large-scale AI workloads and provided a comprehensive evaluation of different approaches, including browser-based inference [19].

5. Distributed Inference and Fine-Tuning for Large Language Models
   A distributed inference framework for large models like GPT and DALL-E was proposed by Pudipeddi et al. (2023). This framework introduces a method for decentralized inference, allowing models to be split across devices, improving efficiency. This approach minimizes network latency and introduces fault-tolerant algorithms, ensuring reliable generation even with device or network failures [20].

3. Project Methodology

3.1 Block Diagram of the System

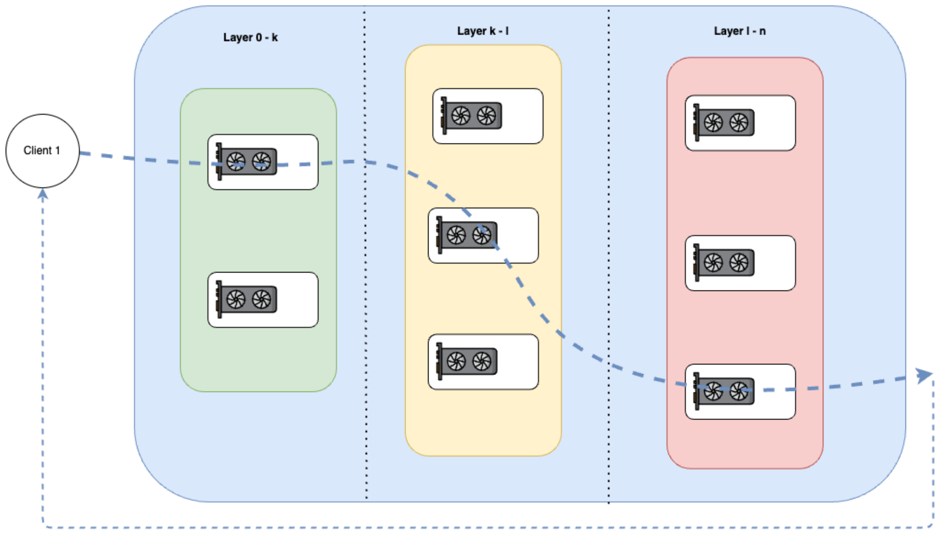


Figure 3.1: Overview of inference in De-LlMIT.
Client-1 wants to infer something from a large language model which the client can't afford (hardware limitations). It uses the hardware that hosts the layers of different layers of the model. A sequence is generated by the system which specifies the path of inference and the path is taken for proper inference.

De-LlMIT is designed to perform two main tasks: inference of prompt and fine-tuning of large models which can't be loaded in a single consumer grade device. In this section, we outline the design of De-LlMIT, both from inference point of view and fine tuning point of view.

3.1.1 Inference of Large-Language Models

Large-Language Models (LLMs) are typically made up of multiple layers of blocks such as Transformer blocks, Attention-Head blocks, and Feed-Forward Networks (FFNs). The large number of such blocks, interconnected in a structured manner, makes LLMs a powerful tool for natural language understanding, generation, and various other downstream tasks. However, this sheer scale poses a significant challenge when deploying these models on consumer-grade hardware due to the immense memory and computational requirements.

Model Sharding in De-LlMIT

The primary innovation in De-LlMIT is model sharding, which divides the LLM into smaller, manageable parts that can be independently processed across a distributed network of devices. Each shard contains a subset of the model's layers—such as a specific number of Transformer or Attention blocks—which can be loaded into the memory of a standard GPU or CPU. Following algorithm describes the model sharding process:

Algorithm 1 Sharding Model Weights for Distributed Loading

```
Require: Pretrained model prefix $P$, number of layers $L$, sharding range $[s, e]$
Ensure: Sharded model with subset of layers
 1: function ShardWeights($P, s, e$)
 2:     $M \leftarrow$ Empty Model Dictionary
 3:     if $s = 0$ then
 4:         $M$[embed_tokens] $\leftarrow$ InitializeEmbedding($P$)
 5:     end if
 6:     for $i \leftarrow s$ to $e - 1$ do
 7:         $W_i \leftarrow$ LoadLayerWeights($P, i$)
 8:         $M[\text{layer}_i] \leftarrow$ InitializeTransformerLayer($W_i$)
 9:     end for
10:     if $e = L$ then
11:         $M$[norm] $\leftarrow$ InitializeLayerNorm($P$)
12:         $M$[lm_head] $\leftarrow$ InitializeLMHead($P$)
13:     end if
14:     return $M$
15: end function
```

Distributing Model Weights

After shrading the model, each shard is stored as a separate repository on the HuggingFace Model Hub. The configuration file and the model weights are uploaded independently for each shard. This approach allows for efficient distributed training and inference by only loading the necessary parts of the model when needed. The dynamic loading of model is done with the following algorithm:

Algorithm 2 Load Model from HuggingFace Model Hub

```
Require: pretrained_model_prefix, start_layer, end_layer
 1: for $i \leftarrow$ start_layer to end_layer $- 1$ do
 2:     Set repo_name $\leftarrow$ pretrained_model_prefix concatenated with “-” and $i$
 3:     if $i ==$ start_layer then
 4:         Download config.json from repo_name
 5:         Parse config.json to obtain config_dict
 6:     end if
 7:     Download pytorch_model.bin from repo_name
 8:     Load the weights into state_dict from pytorch_model.bin
 9:     if $i ==$ start_layer then
10:         Initialize the model with config_dict and the specified layer range
11:     end if
12:     Update the model with state_dict for the current shard
13: end for
14: return the fully loaded model
```

The algorithm remaps a global state dictionary into a shard-specific dictionary for a large model by selectively including only the parameters that belong to the shard's designated layer range. It begins by initializing an empty dictionary for the shard. As it iterates over each key-value pair in the global state dictionary, it checks if the key corresponds to an embedding layer, a transformer layer, or a final module. The embedding layer is included only if the shard is the first one, while transformer layers are added only if their indices fall within the shard's start and end boundaries. Additionally, final modules such as normalization layers or the language modeling head are included only if the shard is the last segment of the model. This method ensures that each shard contains only the necessary parameters, facilitating efficient distributed training and inference by partitioning the overall model into manageable pieces.

Algorithm 3 Remap Global State Dictionary for Model Shard (Full Model)

```
Require: Global state dictionary $\theta_{\text{global}}$, shard start layer $s$, shard end layer $e$, total number of layers $L$
Ensure: Remapped state dictionary $\theta_{\text{shard}}$
 1: Initialize $\theta_{\text{shard}} \leftarrow \{\}$
 2: for each $(k, v)$ in $\theta_{\text{global}}$ do
 3:     if key $k$ corresponds to the embedding layer and $s = 0$ then
 4:         $\theta_{\text{shard}}[k] \leftarrow v$    ▷ Keep embed_tokens if this is the first shard.
 5:     else if key $k$ corresponds to a transformer layer then
 6:         $i \leftarrow$ ExtractLayerIndex($k$)
 7:         if $s \leq i < e$ then
 8:             $\theta_{\text{shard}}[k] \leftarrow v$    ▷ Include layers in the current shard range.
 9:         end if
10:     else if key $k$ corresponds to final modules (e.g., norm or lm_head) and $e = L$ then
11:         $\theta_{\text{shard}}[k] \leftarrow v$    ▷ Keep final normalization and language modeling head if last shard.
12:     else
13:         continue    ▷ Skip keys not applicable to this shard.
14:     end if
15: end for
16: return $\theta_{\text{shard}}$
```

Forward Pass with the Distributed Model

Each consumer-grade device is responsible for processing the forward pass through its assigned shard. After completing its computations, the device forwards the resulting intermediate tensor (i.e., the logits) to the next device in the sequence. In this way, the inference process moves from one shard to the next, mimicking the sequential layer computation typical of an LLM running on a single, larger device.

To ensure proper sequential execution, De-LlMIT orchestrates the flow of data across devices. When an input prompt is received, it is tokenized and fed into the first shard. The logits produced by this shard are transferred to the next shard in the sequence, where further computation occurs. This process repeats until the entire model has been traversed, and the final logits are generated as output. Following is the algorithm that describes the overall working:

Algorithm 4 Distributed Forward Pass with Byte-Based Tensor Transmission

```
Require: Received byte tensor $B_{in}$ from machine $M_{src}$, model $M$ with layer range $[s, e]$
Ensure: Processed byte tensor $B_{out}$ sent to machine $M_{dst}$
 1: function FORWARDPASS($B_{in}, M, s, e$)
 2:     $T \leftarrow$ ConvertBytesToTensor($B_{in}$)          ▷ Convert received bytes to tensor
 3:     if $s = 0$ then
 4:         $T \leftarrow$ EmbedTokens($M, T$)          ▷ Apply embedding layer if first shard
 5:     end if
 6:     for $i \leftarrow s$ to $e - 1$ do
 7:         $T \leftarrow$ ApplyLayer($M[\text{layer}_i], T$)    ▷ Forward pass through each transformer layer
 8:     end for
 9:     if $e =$ TotalLayers($M$) then
10:         $T \leftarrow$ ApplyLayerNorm($M, T$)
11:         $T \leftarrow$ ApplyLMHead($M, T$)        ▷ Apply final layer normalization and LM head
12:     end if
13:     $B_{out} \leftarrow$ ConvertTensorToBytes($T$)          ▷ Convert output tensor to bytes
14:     Send $B_{out}$ to machine $M_{dst}$          ▷ Transmit processed tensor to next machine
15:     return $B_{out}$
16: end function
```

The bottleneck here can be a central server trying to manage the layer shrads and the helping the logits to transfer from one client to another. De-LlMIT tackles this by employing a partially distributed network type of architecture. 3 different types of nodes are employed in the De-LlMIT viz. Monitor Client Nodes, Terminal Instance Nodes and Browser Instance Nodes.

Monitor Client Nodes These are the nodes which is primarily responsible only for maintaining the network. They are the nodes which will remain active at all times, and can be reached by any other nodes to join the network. These nodes maintains a Distributed Hash Table (DHT) which contains the list of other monitor client nodes, and the active peers, layers the peer holds. It also holds a Hash Table which holds the address of the peers connected to itself. The heavylifting of the transfer such as for the transfer of logits are primarily done using the WebSocket which is maintained across the client and the MCNs. DHT is responsible for managing the layers and client addresses with in different MCNs.

Terminal Instance Nodes These nodes are the terminal instance of the Medium-High Performance Computing Devices which can contribute to the inference and fine-tuning i.e.

loads the shred of model layers and performs the forward pass. They are client type of nodes responsible for computation and aren't accountable for maintaining the overall system processes.

Browser Instance Nodes These are special types of nodes, which is first used by De-LlMIT. This type of nodes are present in the browsers of consumer-grade devices. A compiled sharded model (ONNX format) is loaded in the browser which is then inferred within the browser using the available hardware accelerators. These types of client nodes can only be used for inference and not fine tuning or pretraining.

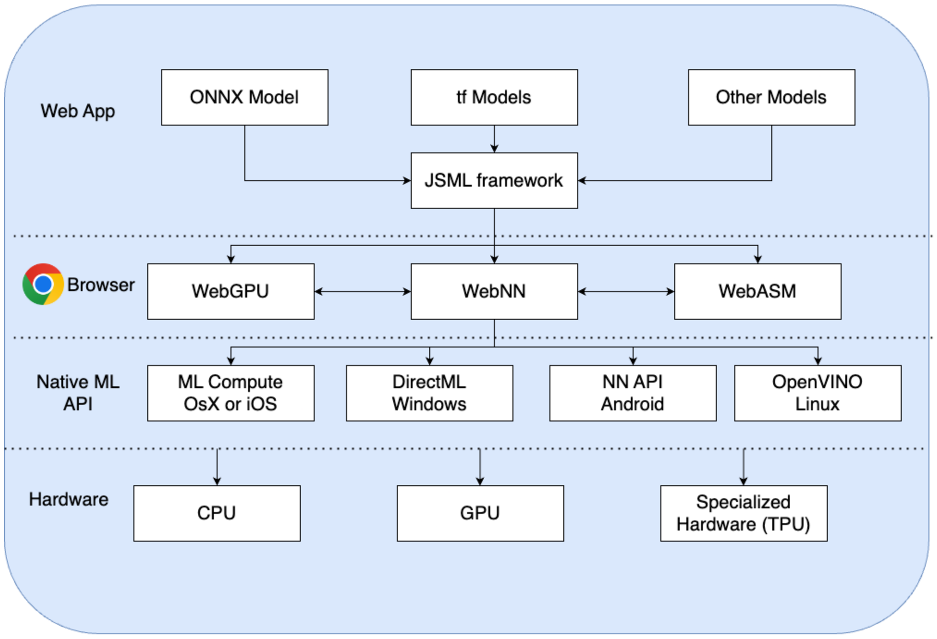


Figure 3.2: Working of Compiled Models in Browser Instance Nodes. [21]

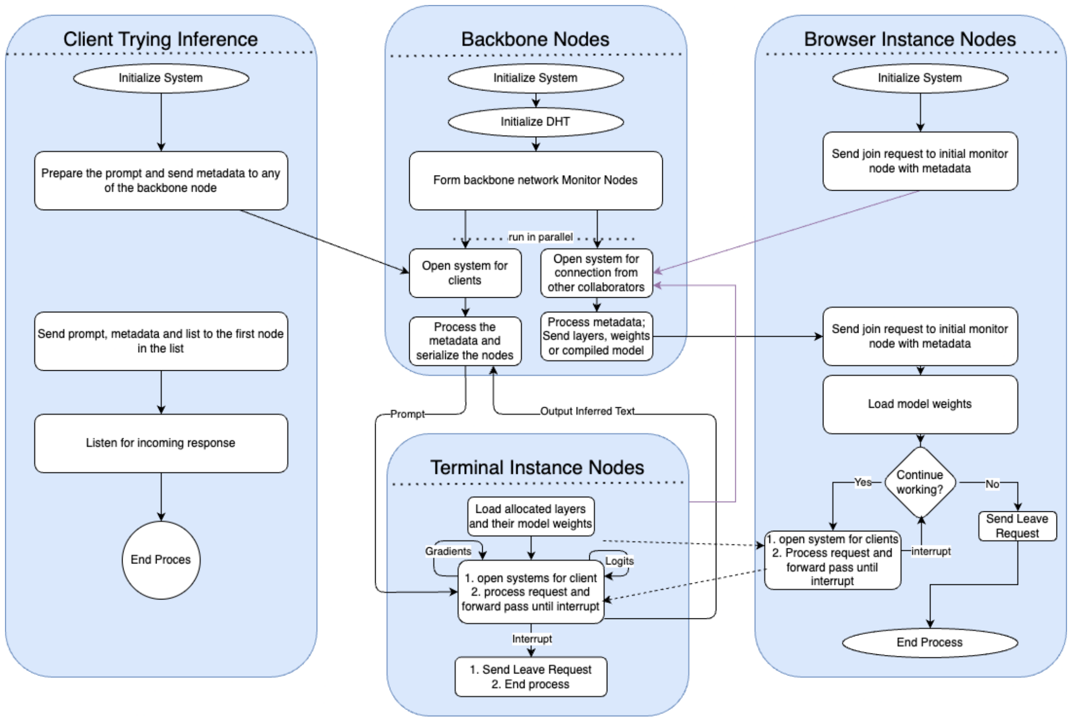
The *Monitor Client Nodes* are responsible for maintaining the DHT in addition to creating a serialized path of inference with in the network. This serialized path should ensure a proper inference of the model, of which layers are loaded in different consumer grade devices. The architecture incorporates fault-tolerance by employing multiple nodes to load the same layer, having consistent backbone MCN to sustain the network.

The majority of the connections within the system is handled using the WebSocket Protocol (RFC6455)[22]. This is done, since this set up required full duplex communication with minimal latency and needs to transfer large bandwidth of data i.e. logits.

Multi-Layer WebSocket Communication System

The system implements a multi-layer WebSocket communication framework that leverages asynchronous programming for real-time, dynamic message processing and performance tracking. On the server side, dedicated components manage client connections and worker assignments—each worker being responsible for a specific processing layer—with a central loop that iteratively refines messages by dispatching them to available workers. Clients establish connections through a binary handshake that encodes both their processing layer and unique identifier, then enter a processing loop to handle incoming messages, perform necessary data conversions, and log detailed timing metrics for performance analysis. This design ensures efficient load balancing, robust error handling, and smooth communication in line with WebSocket protocols as defined in RFC6455. Further is explained in Implementation Details.

3.2 Working Flow of the System



Client Trying Inference
Initialize System
Prepare the prompt and send metadata to any of the backbone node
Send prompt, metadata and list to the first node in the list
Listen for incoming response
End Proces
Backbone Nodes
Initialize System
Initialize DHT
Form backbone network Monitor Nodes
run in parallel
Open system for clients
Open system for connection from other collaborators
Process the metadata and serialize the nodes
Process metadata; Send layers, weights or compiled model
Prompt
Output Inferred Text
Terminal Instance Nodes
Load allocated layers and their model weights
Gradients
Logits
1. open systems for client 2. process request and forward pass until interrupt
Interrupt
1. Send Leave Request 2. End process
Browser Instance Nodes
Initialize System
Send join request to initial monitor node with metadata
Send join request to initial monitor node with metadata
Load model weights
Continue working?
Yes
No
Send Leave Request
1. open system for clients 2. Process request and forward pass until interrupt
interrupt
End Process

Figure 3.3: High-level workflow of the system.

The workflow presented illustrates the interaction between Client Nodes, Backbone Nodes, Terminal Instance Nodes and Browser Instance Nodes within the De-LlMIT system for distributed inference. The backbone nodes are initialized and the new joining nodes, browser instance or terminal instance is provided with their layer weights once they request to join.

Then, they are made available to the clients. The inference process begins with a client initializing a request, which is handled by backbone nodes. These nodes serialize the path of inference across the network. Browser Instance Nodes and Terminal Instance Nodes then perform inference using sharded models in different consumer devices, facilitating forward passes through smaller model segments. The system ensures efficient communication, fault tolerance, and seamless collaboration between distributed nodes.

3.2.1 Distributed Hash Table (DHT) Initialization

The DHT is employed to facilitate peer discovery. Each peer announces its presence and registers on the DHT, making its computational resources available for decentralized inference. The DHT stores the address of the BINs and TINs along with the layers they hold. The data exhange is done in real time using the WebSocket Protocol. This decentralized storage mechanism allows peers to communicate without relying on a central server, ensuring scalability and fault tolerance.

They are primarily initialized in the monitor client nodes. The Monitor Client Nodes are responsible for maintaining the DHT, and any new node, any of the 3, which wants to the join the network should join the network via one of the stable monitor client node.

3.2.2 Initialization and Joining the Network

The initialization of De-LlMIT involves the configuration of nodes and their integration into the network. Two key processes take place during this phase: *load shraded models* and *peer distribution*.

Each peer that joins the network announces its availability by registering with the Monitor Client Nodes, providing details about the shards they hold. This registration process ensures that the system can route inference requests to the appropriate peers based on their capabilities.

3.2.3 Client Request Initialization

The inference process begins when a client, e.g., *Client-1*, initiates a request. This request consists of a prompt that the client wishes to process using the distributed model.

Sending a Request for Path

- *Client-1* sends an inference request, containing the prompt and other necessary metadata, to one of the **Monitor Client Nodes**.
- The MCN handles the request by identifying the path through the network required for performing the inference based on the distributed layers.

Serialized Path for Inference

- The MCN refers to the Hash Table to identify the active peers that hold the necessary model shards.
- A *serialized path* is then generated, specifying the sequence of nodes through which the inference request will pass talking in regard factors like idleness.

This path ensures that the layers are processed in the correct order, allowing distributed execution across multiple nodes while maintaining the integrity of the model's computation.

Execution of Forward Pass and Inter-Node Communication

- *MCN* sends the input to the first node in the path, which is responsible for processing the first shard of the model.
- The first node processes the input through its assigned layers and computes the intermediate logits. It then forwards the intermediate result to the MCN and to next node in the sequence.
- This communication continues between nodes along the serialized path, with each node processing its shard and forwarding the logits until the entire model has been traversed.
- Finally, the output Inferred Text is passed pack to the client.

The communication between nodes is managed by the Monitor Client Node and is done using the WebSocket Protocol. DHT is responsible for managing the client and the layers they hold in the MCNs. This ensures seamless inter-node communication throughout the forward pass.

Returning the Inference Result

- The last node in the serialized path completes the computation and generates the final output.
- The final logits are returned to *MCN*, completing the inference request.
- Finally, the output is passed to the Original Client.

The distributed architecture of De-LlMIT enables the inference of large language models on consumer-grade hardware by dividing the workload across multiple peers. The final output is generated after passing through all the necessary layers of the model, distributed across the nodes in the network.

3.2.4 Fine-Tuning in De-LlMIT Architecture

Along with inference, the De-LlMIT architecture is also designed to support the fine-tuning of large models that cannot be fully loaded on consumer-grade hardware. The fine-tuning process is efficiently handled by distributing the model layers across multiple devices, while the final fully connected layer is fine-tuned on a client device. This section details the procedure for fine-tuning using this distributed system.

Initialization and Network Joining

To begin the fine-tuning process, the client node first initializes the system and joins the network. Similar to the inference process, the model is sharded into smaller, manageable layers that are distributed across the nodes within the network. Each shard is loaded onto a peer in the network in the Terminal Instance Node, depending on the computational capacity of the device.

Using the Network for Forward Pass

Once the network is initialized and the model shards are loaded across the peers, the client begins the forward pass through the network. The forward pass is similar to the inference process, where the input data is tokenized, passed to the first node in the sequence, and moves sequentially through the network. Each node computes its respective shard and passes the intermediate results (logits) to the next node. This continues until the forward pass reaches the final layer.

Fine-Tuning

After the forward pass through the distributed layers is complete, the client begins fine-tuning the fully connected layer. This is done by performing backpropagation on the final layer using the output logits received from the earlier layers. The TINs updates the weights of the layers based on the gradients calculated during backpropagation.

Backward Pass and Weight Updates

Once the fully connected layer is fine-tuned, the backward pass is initiated. During the backward pass, the gradients are calculated for the earlier layers, and the necessary weight updates are performed. These updates are sent back through the network, with each node updating its corresponding shard.

Algorithm:

Algorithm 5 StepTune: Single Fine-Tuning Step with Logit Sharing

```
Require: Input tensor logits; Target batch target_batch; Model model
Ensure: Gradient tensor for logits
 1: Move model, logits, and target_batch to the designated device.
 2: if model.start_layer $\neq 0$ then
 3:     Set logits.requires_grad $\leftarrow$ True
 4:     Retain gradients for logits.
 5: end if
 6: output $\leftarrow$ model(logits)
 7: Initialize optimizer $\leftarrow$ AdamW(model.parameters(), lr $= 5 \times 10^{-5}$, weight_decay $=$
    0.1).
 8: Call optimizer.zero_grad().
 9: if model.end_layer = model.num_hidden_layers then                ▷ Last layer case
10:     Compute loss:
            loss $\leftarrow$ cross_entropy(output.flatten(0,1), target_batch.flatten())
11:     Backpropagate: loss.backward().
12:     Update parameters: optimizer.step().
13:     return logits.grad.data.
14: else                                                       ▷ Intermediate layer case
15:     output_grad $\leftarrow$ StepTune($i + 1$, output.data, target_batch).
16:     Backpropagate using upper layer gradients: output.backward(gradient = output_grad).
17:     Update parameters: optimizer.step().
18:     if model.start_layer $= 0$ then
19:         return null.
20:     else
21:         return logits.grad.data.
22:     end if
23: end if
```

4. System Design

The architecture for the Delimit consists for 3 main layers:

1. **Browser Instances**
2. **Terminal Instances**
3. **Monitor Client**

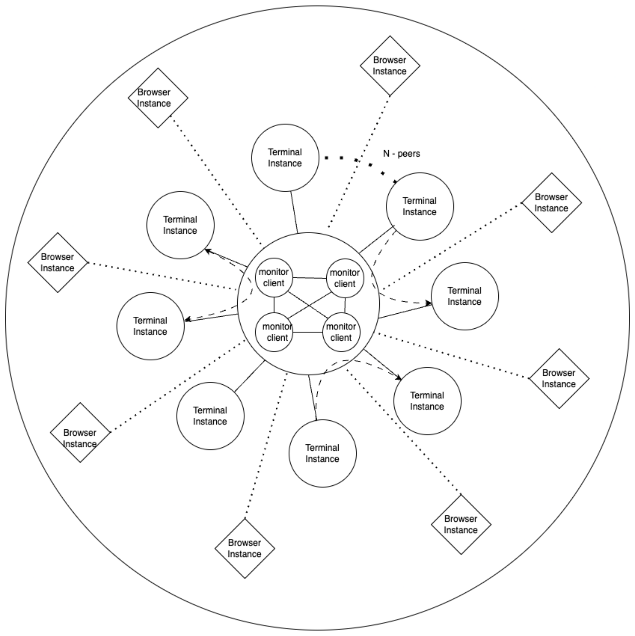


Figure 4.1: System Design

4.1 Layer Descriptions

4.1.1 Browser Instances

The outermost layer, denoted by the **Browser Instances**, consists of users interacting with the system through their web browsers. Each browser instance represents a client that communicates with inner layers or other browser instances to initialize or take part in inference. These browser instances can be seen as lightweight user endpoints that connect to the Delimit system via HTTP or WebSocket protocols. Their primary function is to provide an interface for end users to be able to use AI features without having to go through installation processes.

4.1.2 Terminal Instances

The **Terminal Instances** forms the middle layer of the system. Each terminal instance is responsible for executing specific computations related to model sharding and distributed tasks. These terminal instances are connected to each other in a peer-to-peer fashion if possible or by using the Monitor Clients as relay servers, creating a distributed network where each instance handles part of the workload. In the context of model sharding, each terminal instance might handle different layers of the GPT or llama model, with inputs being pipelined across instances.

The terminal instances can perform minor sequencing related task and also take part in finetuning of the model other than inference. They may be high performance servers in the universities or simple client terminal running on an PC.

4.1.3 Monitor Clients

At the center of the system are the **Monitor Clients**, which are responsible for orchestrating the entire process. These clients monitor the health and status of the terminal instances, ensuring that computations are progressing as expected and that instances are correctly synchronized. Additionally, monitor clients may also handle load balancing, distributing tasks to terminal instances to optimize performance.

Thse are the servers with their own IP addresses and are connected to each other in a purely P2P fashion. They also act as STUN server for NAT traversal between the terminals and/or browser instances behind NAT.

5. Implementation Details

5.1 Model Sharding

Delimit follows a model-parallel approach to distributed training. To achieve this, we need to divide a single large language model (LLM) into multiple partial models that can run independently on different machines, where the output from one partial model is pipelined into the next in sequence.

Model Sharding is essential for this process, where an LLM is divided into different contiguous layers. When inference or training is performed, a sequence of sharded models is taken, forming an ordered and exhaustive set of the entire model.

5.1.1 Model Sharding in Delimit

Model Sharding in De-llmit starts with the sharding of LLMs. It uses the base class from the `Hugging Face Transformers` library.

An LLM consists of an embedding layer at the beginning, which converts tokens into embedding vectors. It is followed by several transformer layers, and at the end, there is a normalization layer and a head layer. Each of these layers can be sharded and work independently to produce output once input from the preceding layer is provided.

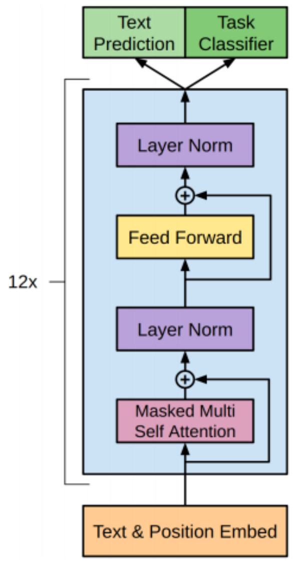


Figure 5.1: Architecture of a generic LLM

To shard the model, we create a `CustomLLMModel` using the base class. This class uses an initialization function that takes the start layer, end layer, and the total number of layers in the desired model as input. This initializes the custom model with a specific shard of layers from the start to the end layer. If the start layer is 0, it includes the embedding layers, and if the end layer equals the total number of layers, it includes the final normalization and head layers.

The next element of the custom class is the forward layer, which carries out one forward pass. If the model's starting layer is 0, it performs embedding at the start. Then, it processes the input through the sharded transformer layers. If the model's end layer equals the total layers, it performs normalization and head operations at the end.

Sharding the LLM into 3 equal parts produces 3 partial models as shown:

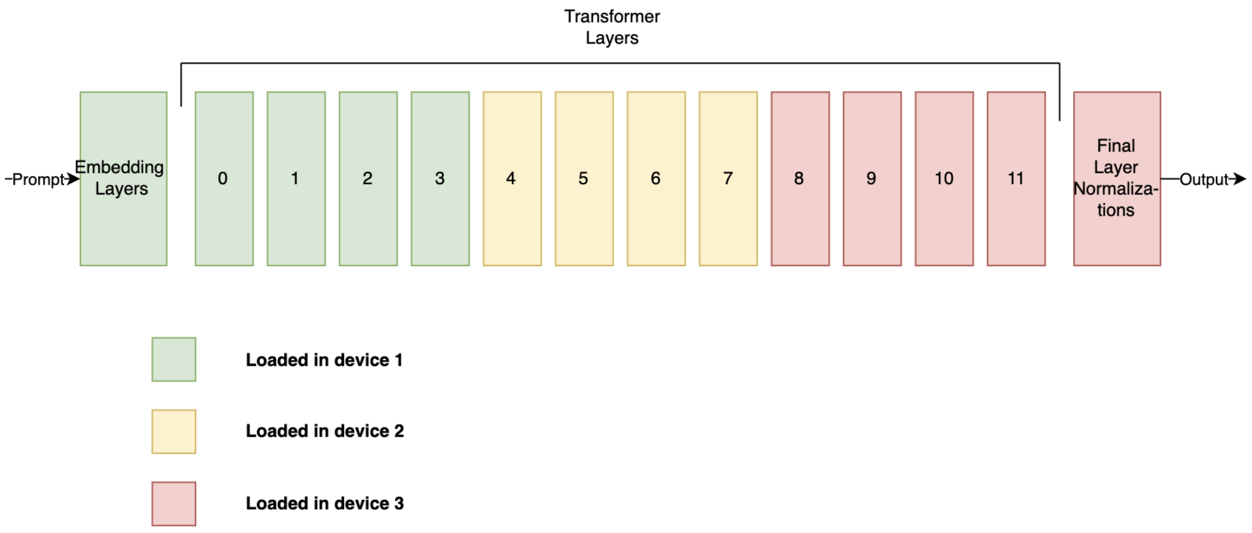


Figure 5.2: Vizualization of different layers in a Model

5.1.2 Splitting the Pretrained Weights

Since the models are split, the pretrained models used for inference should also be split to contain only the weights of the partial model. For this, we created a function `Split_state_dict()`. This function takes the start layer, end layer, and total layers as inputs and removes the transformer weights outside the range (start layer, end layer). The weights of the embedding layers are included if the start layer equals 0, and the head and normalization layer weights are included if the end layer equals the total number of layers.

The keys in the returned weights correspond to the original model's layer names, but when creating the partial model, their keys are adjusted to start from 0. A mapping function called `remap_state_dict()` was used to change state dict keys to match those of the model state dict keys by mapping the keys to 0 to the number of layers in the partial model.

The `remap_state_dict()` function adjusts the layer indices in a given state dictionary for a transformer model. It creates a new dictionary where, for each key-value pair, it checks if the key starts with "transformer.h". If it does, the function extracts the current layer number, subtracts a specified starting layer from it, and constructs a new key using the updated layer number along with the remaining parts of the original key. If the key does not match the transformer layer pattern, it is added unchanged to the new dictionary. Finally, the function returns this newly constructed state dictionary.

The `filtered_state_dict()` function extracts specific layers from a transformer model's state dictionary based on given start and end layer indices. It creates a new dictionary that includes keys starting with "transformer.h" if their layer numbers fall within the specified range. It also includes keys for the word and positional embeddings if the starting layer is zero, and for the final normalization and language model head if the end layer matches the total layers. Finally, it remaps the layer indices in the filtered dictionary and returns the updated version.

5.2 DHT and Logits Sharing

A distributed hash table (DHT) was implemented using the Hivemind library, a PyTorch library for decentralized deep learning across the Internet. Its intended usage is training one large model on hundreds of computers from different universities, companies, and volunteers.

A server with a public IP was set up using cloud credits as the initial peer to start the Hivemind DHT.

The `dht.get_visible_maddrs()` method retrieves the ID of the peer, which was then used by other peers to connect to the server. Data could then be passed between them using the store and get methods. This setup can be used to pass messages to peers for sequencing and synchronization but could not be used to share large files like tensor logits.

To share the logits, they were either sent directly using websocket established between monitor clients and terminal or browser clients, with the relevant peer being notified using the DHT. The Hivemind DHT also provides fault tolerance, allowing browser or terminal instances to join and leave as they please.

5.3 Llama Implementation

Our use case required fine tuned control over the Llama Architecture. We will need to shard the model into different layers and custom compile the model to run it in the browser settings. Thus we implemented the Llama architecture from scratch.

Here is an overview of the Llama Architecture:

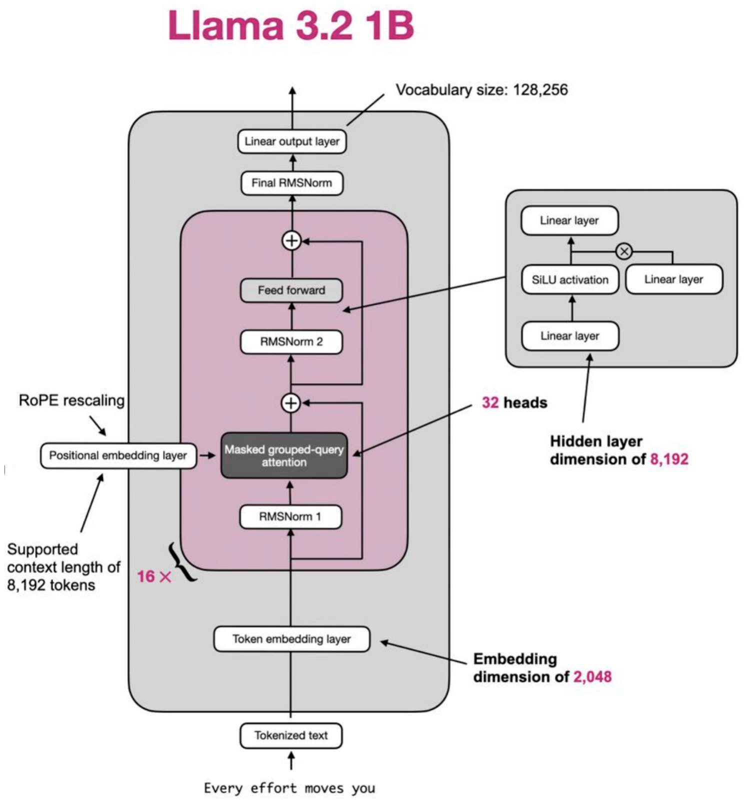


Figure 5.3: Llama3.2-1B Architecture

This implementation closely follows the original LLaMA model design while allowing for greater flexibility and customization.

The custom architecture includes the following key components:

1. **Feedforward Network**

   Feedforward Network in Llama consists of one up projection layer, one gate layer that applies SiLU function and one down projection at the end.

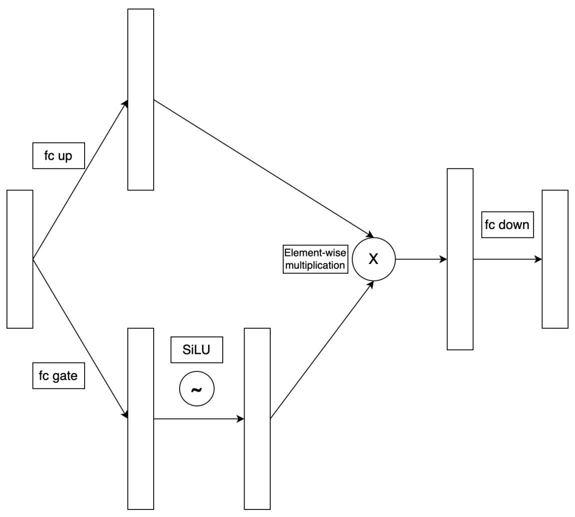


Figure 5.4: SwiGLU FFN

2. **MultiHead Attention** Llama uses MultiHead Attention to allow for parallel computation of context vectors.

   (a) **GroupedQuery Attention** Grouped-Query Attention (GQA) is an optimization of MultiHead Attention that reduces computational complexity while maintaining model quality. It groups queries into subsets and uses shared keys and values for each group.

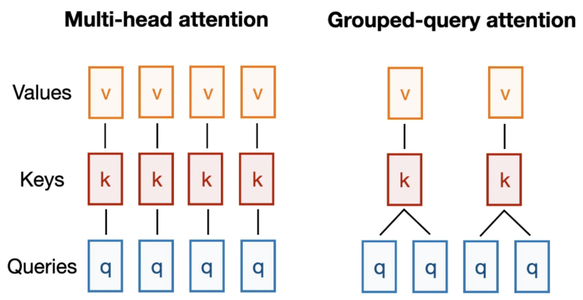


Figure 5.5: Multi-Head Grouped Query Attention

   (b) **Rotary Positional Embedding** Llama uses Rotary Positional Embedding (RoPE) which is a technique that encodes positional information by rotating the query and key vectors in the attention mechanism. It allows the model to capture relative positional information effectively.

5.4 Sharding Llama Model

After implementing the basic Llama Model, the Llama model was modified so the user can create a partial sharded model. To shard the model, the user would provide the start layer and end layer during the initialization of the model.

The new model will then have only the layers specified during initialization.

This is similar to what we did with the GPT2 model the only difference being that the base model is a custom Llama model.

5.4.1 Weights loading

After creating the model weights, the problem remained of how to load only the weights required since the models weights available in the internet are for the full model. So, we sharded the model weights manually into one layer each and uploaded the layerwise weights into the hugging face. Then we defined a custom .pretrainedfunction that will take the model prefix and download and load the required state dict layerwise.

5.5 Fine Tuning

After creating model and loading pretrained weights, we also implemented fine tuning. For fine tuning we looked at Parameter Efficent Fine Tuning(PEFT) techniques and more specifically Low Rank Adaptation(LoRA) fine tuning.

5.5.1 LoRA

A LoRALayer class was defined to create low-rank adaptations of linear layers. It defines two matrices A and B whose resulting dimension after multiplication matches that of the parameter of the linear layer that lora will work with. The dimension of A is set to (indim,rank) and that of B to (outdim,rank). Rank determines how heavy we want the fine tuning to be. The weights of A is initialized to random standard distribution and B to 0 so that AB=0 initially.

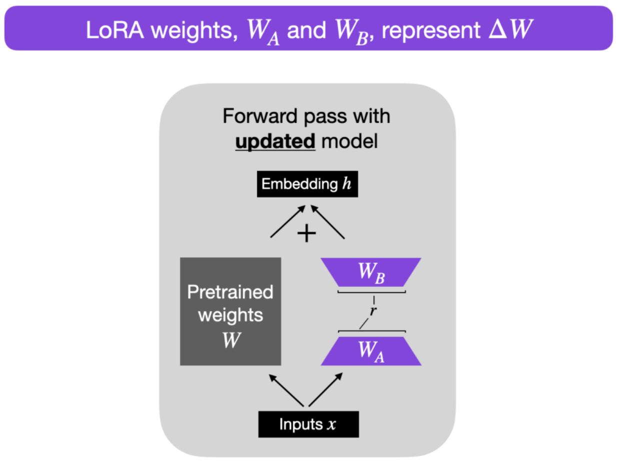

LoRA weights, W_A and W_B, represent ΔW
Forward pass with **updated** model
Embedding h
Pretrained weights W
W_B
r
W_A
Inputs x

Figure 5.6: LoRA

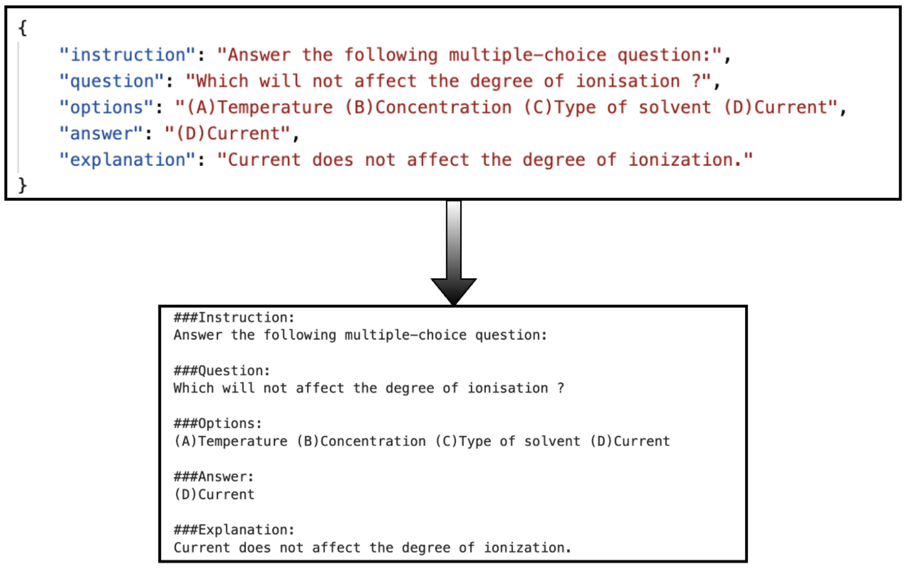
LinearWithLoRA wraps existing linear layers with LoRA functionality.

5.5.2 Setting up model for LoRA fine tuning

- Instantiate the custom llama model
- Load the pretrained weights
- Freeze all the parameters in the model
- Replace the desired linear layers with LinearWithLoRA

5.5.3 Preparing DataSets and Dataloaders

For the experimentation purposes, we used the web scraped entrance exam question, answers. This data was cleaned to form a dataset in json format containing keys: instruction, question, answer, and explanation.



```
{
    "instruction": "Answer the following multiple-choice question:",
    "question": "Which will not affect the degree of ionisation ?",
    "options": "(A)Temperature (B)Concentration (C)Type of solvent (D)Current",
    "answer": "(D)Current",
    "explanation": "Current does not affect the degree of ionization."
}
```

```
###Instruction:
Answer the following multiple-choice question:

###Question:
Which will not affect the degree of ionisation ?

###Options:
(A)Temperature (B)Concentration (C)Type of solvent (D)Current

###Answer:
(D)Current

###Explanation:
Current does not affect the degree of ionization.
```

Figure 5.7: Alpaca Style Prompt Formatting

We used alpaca style prompt formatting to transfer the dataset into appropriate form for finetuning.

An InstructionDataset class was created to handle the custom dataset format. A custom collate function pads and prepares batches for training. DataLoaders are created for train, validation, and test sets.

5.5.4 Distributed FineTuning

Finetuning has 4 main steps:

1. Forward Pass where graph of the computations is calculated
2. Loss calculation at the final layer using the output and target
3. Calculating the gradients
4. Changing the parameter's weights

The first two steps can be achieved easily using the inference implementations already achieved which involved passing the intermediate nodes to the next node.

But this is not sufficient for step 3 since training requires passing the logits to the next node during forward pass and gradients to the previous node during the backward pass.

After the last layer calculates the loss from the target and the output, it will then calculate the gradients of the weights all the way back to the input logits it received. This gradient will be then sent back to the previous node as the gradient for the output of the previous node.

We choose a recursive function approach for each training step. The initial node will call the step function produce its output and call the step function of the next node with its output. The last node will then return the gradients of the input it received as for the step function called by its previous node on it. The previous node will use this gradients to calculate the gradients of its relevant parameters and change their weights. This will go back to the initial node sequentially updating each node's weight in the process.

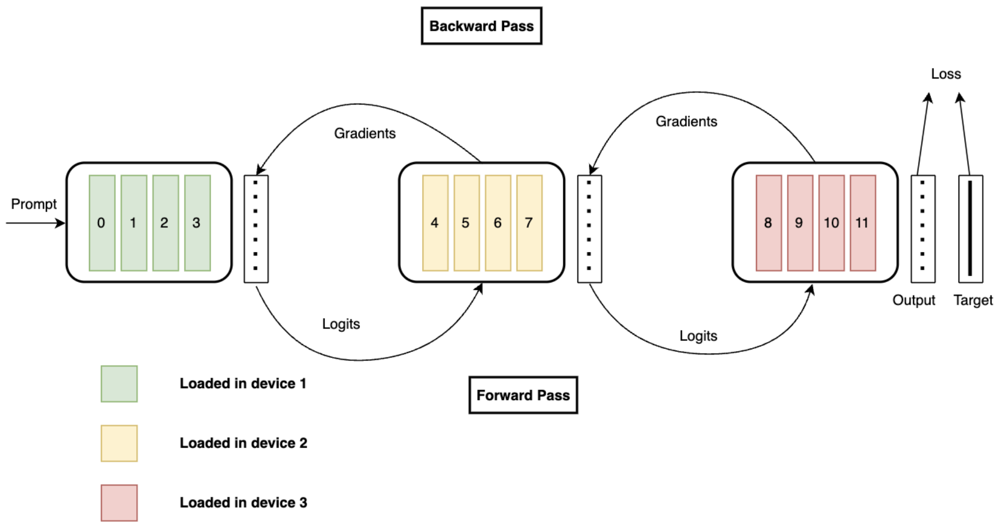


Figure 5.8: A single step in fine-tuning

For fine tuning of LLMs, cross entropy loss function was choosen.

$$\mathcal{L} = -\frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\log \hat{y}_{i,t}^{(y_{i,t})}$$

where $\hat{y}_{i,t}$ represents the probability with which the model predicts the next token is the target token for i^{th} batch and tth place in the sequence. N is the total batch size.

5.6 Multi-Layer WebSocket Communication System

We developed a robust multi-layer WebSocket communication system that supports dynamic message processing and real-time performance monitoring. Using asynchronous programming, our solution seamlessly integrates both server-side and client-side components. This architecture is designed to support multiple clients operating in distinct processing layers while efficiently managing and tracking the overall communication flow.

5.6.1 Server-Side Implementation

Worker and Connection Management

Worker Class: We defined a `Worker` class to encapsulate the properties of each client, including a unique identifier and a busy status flag. This abstraction helps us monitor whether a worker is currently engaged in processing a task.

Connection Manager: A dedicated `ConnectionManager` maintains a dictionary of active WebSocket connections. It provides methods to add new connections, remove disconnected clients, and robustly handle binary message exchanges with comprehensive error handling.

Worker Manager: To balance load distribution, our `WorkerManager` organizes workers into predefined processing layers. It supports adding workers to specific layers, removing inactive workers, and selecting the least populated layer when dispatching tasks.

Message Processing Logic

At the heart of our system lies a central message processing loop that iterates over a fixed number of cycles. In each iteration, the system:

1. Selects an available worker from each layer.
2. Constructs a payload based on the processing layer (e.g., initial prompt, accumulated data such as logits).
3. Sends the payload to the worker, waits for the response, and updates the message content for the next iteration.

This iterative process refines the input prompt across multiple layers until the final output is achieved.

WebSocket Connection Handling

Our WebSocket logic includes:

1. Accepting incoming connections and executing a binary handshake that encodes both the processing layer number and the client's unique ID.
2. Parsing handshake data to register the client with the `ConnectionManager` and `WorkerManager`.
3. Maintaining the connection with a keep-alive loop, ensuring resource cleanup and proper deregistration upon client disconnection.

5.6.2 Client-Side Implementation

Establishing the WebSocket Connection

Using `websockets` library, our client-side code establishes an asynchronous connection with the server. Upon connection, the client sends a binary-encoded handshake containing its designated processing layer number and unique client ID.

Processing Loop and Message Handling

After the handshake, the client enters a processing loop where it:

1. Receives messages from the server and logs the duration of data receipt.
2. Differentiates between binary and text messages, handling each appropriately.
3. Performs decoding and parsing, converting bytes to tensors and vice versa, while generating intermediate logits or model outputs.
4. Records detailed timestamps for each step (receiving, processing, decoding, and sending) to facilitate performance monitoring and debugging.

Performance Monitoring and Logging

Throughout the entire communication cycle, precise timestamps are logged for every message transfer. This detailed monitoring framework enables us to identify potential performance bottlenecks and optimize the communication flow across the system.

Communication Protocol and Error Handling

A standardized binary handshake initiates every connection, encoding both the processing layer and client ID to ensure mutual understanding between the server and client. This robust protocol, coupled with rigorous error handling, ensures smooth communication and swift recovery from potential errors.

6. Results

We compared the inference and training times using full model in a single device, with RAM offloading and with model parallelism(our framework).

6.1 Experimentations

6.1.1 Network Simulation

The llama 3.3 1B was used as the base model for inference and training purpose. The T4 16GB GPU was used for the inference and training purposes of llama 3.3 1B model: The llama 3.3 1B was splitted into 3 shards:

1. Model1: 0-5 layers
2. Model2: 5-10 layers
3. Model3: 10-16 layers

For the full model purpose, the whole llama model was loaded and inference and training were run on it.

For inference and training with RAM off loading, the model1 was loaded into gpu first, logits were calculated, and then it was offloaded to cpu, then model2 was loaded, logits calculated and offloaded and so on for each computation required from the sharded model.

For model parallelism, since we were having network instability issues, we used a simulated environment to calculate the inference and training times. For this, we loaded all three shards in the same GPU, then before passing logits or gradients to the next or previous shard, we calculated the size of the data we would need to send and added a delay function that will stop the execution for time required to send that size of data over some constant(1MBPS, 5MBPS, 10MBPS) data speed thus simulating real data transmission.

Graphs

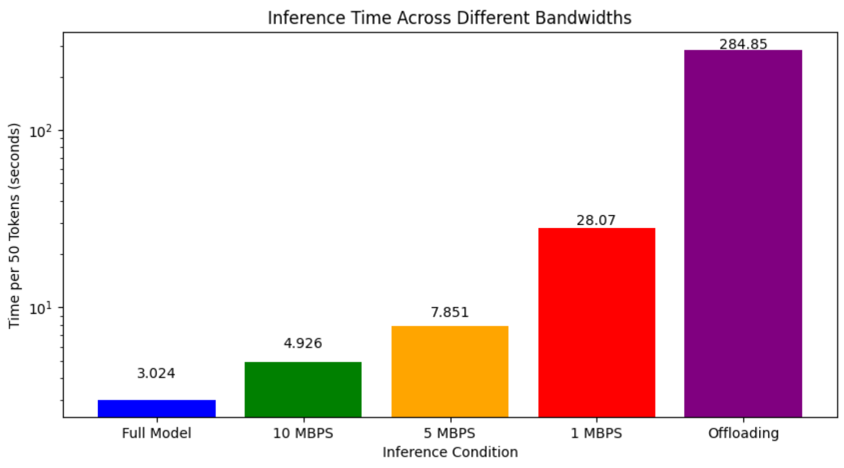


Figure 6.1: Average inference time per 50 token

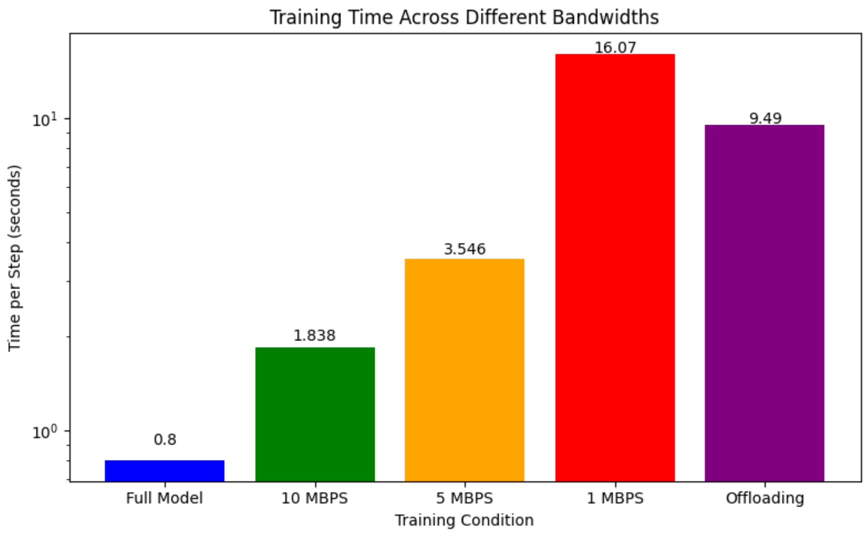


Figure 6.2: Average training time per step

6.1.2 Fine-Tuning

We finetuned 2 different llama models namely llama 3.3 1B and llama 3.3 3B using a custom-crafted dataset scraped from the internet, with more details in the Implementation Details Section. The models were divided into 2 and 3 shards respectively in the layers as:

- llama 1B:

    1. Model1: 0-8 layers

2. Model2: 8-16 layers

- llama 3B:

  1. Model1: 0-8 layers
  2. Model2: 8-16 layers
  3. Model3: 16-28 layers

Graphs

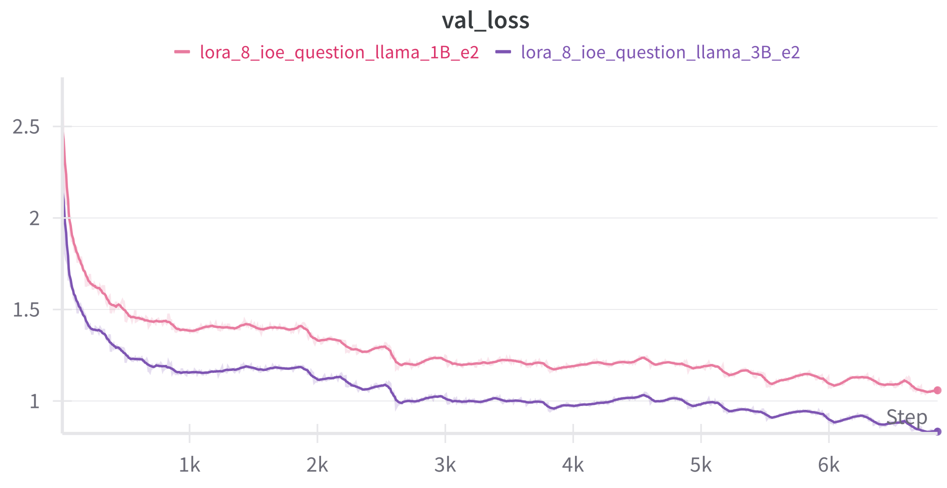


Figure 6.3: Validation Loss per step

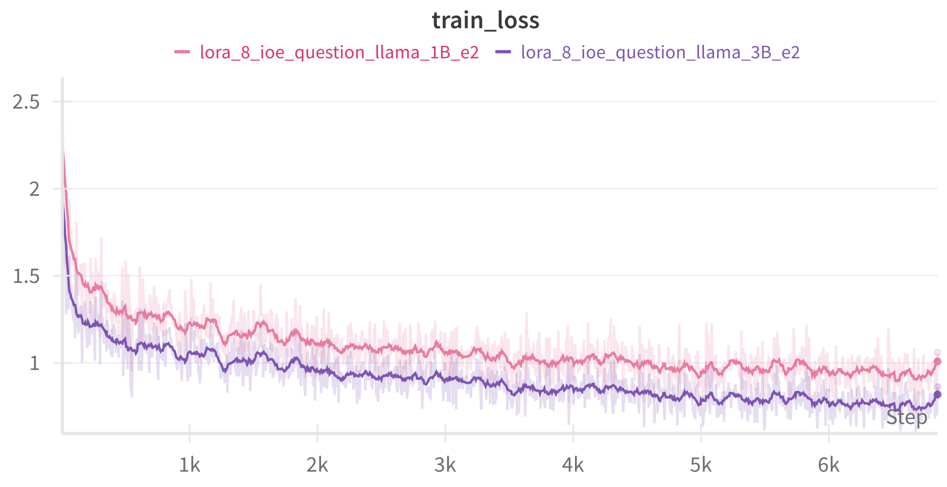


Figure 6.4: Training Loss per step

The loss converged as intended. This is since the gradient updates were successful and the model was fine-tuned on our experimental setup.

7. Future Work

This section outlines potential avenues for further research and engineering works based on the study's findings, limitations, and open questions.

7.1 Open Collaboration Incorporating NAT Traversal

To make the *De-LlMIT* system more decentralized and improve open collaboration and to make the system fully decentralized, future works can be done to incorporate P2P systems in De-LlMIT. But, the challenges posed by P2P is Network Address Translation (NAT) and unreliable peer communication. NAT often makes it difficult for peers behind different NATs to talk directly without some help. Here are a few ideas that can be explored:

- **NAT Traversal Techniques:** One possible approach is to look into NAT traversal methods like UDP and TCP hole punching. These techniques could help peers connect directly even if they're behind different NATs. Along with that, we could also consider using STUN (Session Traversal Utilities for NAT) and TURN (Traversal Using Relays around NAT). STUN helps peers discover each other's addresses, and TURN can act as a relay if direct communication fails. Combining these methods might allow peers to interact more smoothly without needing a centralized server.

- **Relay Server Utilization:** If direct connections aren't possible due to restrictive NATs or firewalls, relay servers could be a useful fallback. They can temporarily handle communication between peers, ensuring that data can still be transmitted even if direct connections aren't feasible.

- **Third-Party Storage Integration:** Another idea is to use third-party storage services, like AWS S3, to help with data exchange. By uploading and downloading data from a cloud storage service, peers can bypass some of the issues caused by NAT and firewalls. This way, even if direct communication is tricky, peers can still share and access necessary resources.

But with the advent of IPv6 based Network Infrastructure, this problem should be solved swiftly without the need of any of the above mentioned measures.

7.2 Browser-Based Inference

The primary goal is to shift from terminal-based inference to browser-based inference, making the system more accessible to users without requiring complex setups. The following steps will be taken:

- **ONNX Integration:** Leverage the ONNX Runtime Web to allow models to run directly in browsers. This will involve converting the existing PyTorch models to ONNX format and integrating them into the web environment. This will help create a large cluster of users readily available for inference, since will obviously increase the number of peers participating in the process.

- **Cross-Browser Compatibility:** Ensure that the browser-based inference system is compatible with all major web browsers (Chrome, Firefox, Safari) and can handle different device configurations.

7.3 Further Improvements and Work

To enhance the *De-Llmit* system, several improvements shall be sought to address privacy, security, and incentivization aspects.

Privacy and Security Enhancements: To ensure the confidentiality and integrity of user data throughout the decentralized inference process, various privacy and security techniques will be explored. One approach is to investigate homomorphic encryption, which allows computation on encrypted data without exposing sensitive inputs or outputs, thus preserving privacy.

Incentive Mechanisms: To encourage active participation and resource contribution in the decentralized inference process, an effective incentive mechanism can be developed. A token-based reward system can be designed where participants earn tokens based on their computational contributions, incentivizing them to provide more resources or computational power.

8. Conclusion

De-Llmit offers an innovative approach to large language model (LLM) inference by distributing computational tasks across multiple devices. This makes advanced AI more accessible to a wider audience. By leveraging decentralized technologies, De-Llmit not only overcomes hardware limitations but also invites more contributors to engage with LLM technologies. This project fosters collaboration and effectively utilizes the resources of a distributed network.

The project enhances accessibility and encourages collaboration within the decentralized network, where participants contribute their computational resources for model inference and fine-tuning. The distributed nature of the system ensures scalability, fault tolerance, and lower operational costs while maintaining high performance. Moreover, De-LlMIT integrates advanced technologies such as model sharding, WebGPU, and Distributed Hash Tables (DHT) to efficiently coordinate the flow of computation across various nodes, ensuring optimal utilization of network resources.

Overall, De-LlMIT fosters a more inclusive and collaborative AI ecosystem, inviting broader engagement in LLM technologies. By addressing critical challenges like resource distribution, network reliability, and accessibility, De-LlMIT paves the way for scalable, decentralized AI solutions that can reach a much wider audience without compromising on performance or security.

REFERENCES

[1] Allen Institute for AI, "The cost of training large ai models," Allen Institute for AI, Tech. Rep., 2023. [Online]. Available: `https://allenai.org/ai-economics-report`.

[2] X. Li, J. Xie, S. Zhang, *et al.*, "Scaling up machine learning models: Hardware and performance challenges," *Journal of Machine Learning Research*, vol. 23, no. 1, pp. 123–145, 2022. [Online]. Available: `http://jmlr.org/papers/volume23/li22a/li22a.pdf`.

[3] A. Borzunov, M. Ryabinin, and I. Titov, "Petals: Collaborative inference and fine-tuning of large models," *arXiv preprint arXiv:2209.01188*, 2022. [Online]. Available: `https://arxiv.org/abs/2209.01188`.

[4] M. Chen, A. Radford, and R. Child, "Decentralized ai: Opportunities and challenges," *arXiv preprint arXiv:2301.06543*, 2023. [Online]. Available: `https://arxiv.org/abs/2301.06543`.

[5] M. Ivanov, M. Ryabinin, A. Borzunov, *et al.*, "Hivemind: Decentralized deep learning for large-scale models," *arXiv preprint arXiv:2105.09312*, 2021. [Online]. Available: `https://arxiv.org/abs/2105.09312`.

[6] A. L. Beberg, D. L. Ensign, G. Jayachandran, S. Khaliq, and V. S. Pande, "Folding@home: Lessons from eight years of volunteer distributed computing," *arXiv preprint arXiv:2002.11729*, 2020. [Online]. Available: `https://arxiv.org/abs/2002.11729`.

[7] S. Laskaridis, S. I. Venieris, N. D. Lane, and D. Lucanin, "The future of consumer edge-ai computing," *arXiv preprint arXiv:2210.10514*, 2022. [Online]. Available: `https://arxiv.org/pdf/2210.10514`.

[8] P. Kairouz, H. B. McMahan, B. Avent, *et al.*, "Advances and open problems in federated learning," *Foundations and Trends in Machine Learning*, vol. 14, no. 1-2, pp. 1–210, 2021. [Online]. Available: `http://dx.doi.org/10.1561/2200000083`.

[9] T. Ben-Nun and T. Hoefler, "Model-distributed machine learning," *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–36, 2019.

[10] P. Maymounkov and D. Mazieres, "Kademlia: A peer-to-peer information system based on the xor metric," *Proceedings of the 2002 ACM SIGCOMM Workshop on Practice and Theory of Incentives in Networked Systems*, pp. 53–65, 2002.

[11] X. Wang, L. Liu, B. Yang, C. Li, and Y. Zhang, "P2p networking and dhts in decentralized systems," *IEEE Transactions on Network and Service Management*, vol. 17, no. 3, pp. 1234–1250, 2020.

[12] T. B. Brown, B. Mann, N. Ryder, *et al.*, "Scaling up language models: Gpt-3," *Proceedings of the 2020 Conference on Neural Information Processing Systems (NeurIPS 2020)*, pp. 1–53, 2020.

[13] Y. Shen *et al.*, "Quantization and efficient inference in large models," in *Proceedings of the 34th Conference on Neural Information Processing Systems (NeurIPS)*, NeurIPS, 2020, pp. 1–10. [Online]. Available: `https://arxiv.org/abs/2004.00280`.

[14] X. Li, X. Xu, and X. Liu, "Memory-efficient training of transformers," *Journal of Machine Learning Research*, vol. 21, no. 1, pp. 1–27, 2020.

[15] Y. Jiang, M. Li, Q. Xu, X. Zhao, J. Wang, and G. Lu, "Webinf: Accelerating webgpu-based in-browser dnn inference via adaptive model partitioning," *IEEE Transactions on Parallel and Distributed Systems*, 2023. [Online]. Available: `https://ieeexplore.ieee.org/document/10476167`.

[16] M. Sato, R. Sasaki, D. Saito, and D. Morikawa, "Webdnn: Fastest dnn execution framework on web browser," in *Proceedings of the 26th International Conference on World Wide Web Companion*, ACM, 2017, pp. 1157–1166. [Online]. Available: `https://doi.org/10.1145/3123266.3129394`.

[17] D. Smilkov, N. Thorat, C. Nicholson, E. Reif, F. Viégas, and M. Wattenberg, "On-device machine learning with tensorflow.js," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR) Workshops*, 2019, pp. 19–28. [Online]. Available: `https://ieeexplore.ieee.org/document/9009611`.

[18] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "Blockchain-based incentive mechanisms for decentralized computing," *IEEE Transactions on Services Computing*, vol. 13, no. 2, pp. 252–261, 2018. [Online]. Available: `https://ieeexplore.ieee.org/document/8436039`.

[19] T. Li, Y. Sun, J. Wang, *et al.*, "Efficient decentralized computing for ai applications," *Journal of Parallel and Distributed Computing*, vol. 148, pp. 1–13, 2021. [Online]. Available: `https://doi.org/10.1016/j.jpdc.2020.10.005`.

[20] N. Pudipeddi, A. Gupta, Y. Zhang, and C. Lee, "Distributed inference and fine-tuning for large language models," *Proceedings of the 2023 Conference on Neural Information Processing Systems (NeurIPS 2023)*, pp. 1–14, 2023.

[21] Microsoft, *Webnn: Bringing ai inference to the browser*, https://techcommunity.microsoft.com/t5/ai-ai-platform-blog/webnn-bringing-ai-inference-to-the-browser/ba-p/4175003, Accessed: Sep. 19, 2024, 2024.

[22] A. Melnikov and I. Fette, *The WebSocket Protocol*, RFC 6455, Dec. 2011. DOI: 10.17487/RFC6455. [Online]. Available: https://www.rfc-editor.org/info/rfc6455.