



TRIBHUVAN UNIVERSITY  
INSTITUTE OF ENGINEERING  
PULCHOWK CAMPUS

A  
MID-TERM 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

September 20, 2024

# 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-LIMIT 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-LIMIT 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

<b>Acknowledgements</b>	<b>ii</b>
<b>Abstract</b>	<b>iii</b>
<b>Contents</b>	<b>v</b>
<b>List of Figures</b>	<b>vi</b>
<b>List of Abbreviations</b>	<b>vii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Background . . . . .	1
1.2 Problem statements . . . . .	1
1.3 Objectives . . . . .	2
1.4 Scope . . . . .	2
<b>2 Literature Review</b>	<b>4</b>
2.1 Related Theory . . . . .	4
2.1.1 Volunteer Computing . . . . .	4
2.1.2 Distributed Machine Learning: Federated Learning and Model-Distributed ML . . . . .	5
2.1.3 Distributed Hash Tables and Networking . . . . .	6
2.1.4 Large Language Models: Memory Requirements, Quantization, and Challenges . . . . .	6
2.2 Related work . . . . .	7
<b>3 Project Methodology</b>	<b>9</b>
3.1 Block Diagram of the System . . . . .	9
3.1.1 Inference of Large-Language Models . . . . .	9
3.2 Working Flow of the System . . . . .	12
3.2.1 Distributed Hash Table (DHT) Initialization . . . . .	12
3.2.2 Initialization and Joining the Network . . . . .	13
3.2.3 Client Request Initialization . . . . .	13

3.2.4	Response for Path . . . . .	14
3.2.5	Forward Pass through Model Shards . . . . .	14
3.2.6	Final Output to the Client . . . . .	15
3.2.7	Fine-Tuning in De-LIMIT Architecture . . . . .	15
<b>4</b>	<b>System Design</b>	<b>17</b>
4.1	Layer Descriptions . . . . .	17
4.1.1	Browser Instances . . . . .	17
4.1.2	Terminal Instances . . . . .	18
4.1.3	Monitor Clients . . . . .	18
<b>5</b>	<b>Work Completed</b>	<b>19</b>
5.1	Model Sharding . . . . .	19
5.1.1	Model Sharding in Delimit . . . . .	19
5.1.2	Splitting the Pretrained Weights . . . . .	22
5.2	DHT and Logits Sharing . . . . .	25
<b>6</b>	<b>Future Work</b>	<b>26</b>
6.1	Open Collaboration Incorporating NAT Traversal . . . . .	26
6.2	Browser-Based Inference . . . . .	26
6.3	Further Improvements and Work . . . . .	27
<b>7</b>	<b>Conclusion</b>	<b>28</b>
	<b>REFERENCES</b>	<b>31</b>
	<b>APPENDIX A</b>	<b>32</b>

# List of Figures

3.1	Overview of inference in De-LIMIT. . . . .	9
3.2	Working of Compiled Models in Browser Instance Nodes. . . . .	11
3.3	High-level workflow of the system. . . . .	12
4.1	System Design . . . . .	17
5.1	Architecture of a generic LLM . . . . .	19
5.2	Sharding of a LLM Model . . . . .	20
5.3	Forward Pass . . . . .	21
5.4	Remapping State Dict Keys . . . . .	23
5.5	Filtered State Dict for a Shard . . . . .	24
7.1	Output from peer loading layers 0 to 6 . . . . .	32
7.2	Output from peer loading layers 6 to 12 . . . . .	33

# List of Abbreviations

<b>LLM</b>	Large Language Model
<b>P2P</b>	Peer-to-Peer
<b>GPU</b>	Graphics Processing Unit
<b>DHT</b>	Distributed Hash Table
<b>DNN</b>	Deep Neural Network
<b>NAT</b>	Network Address Translation
<b>STUN</b>	Session Traversal Utilities for NAT
<b>TURN</b>	Traversal Using Relays around NAT
<b>BIN</b>	Browser Instance Node
<b>TIN</b>	Terminal Instance Node
<b>MCM</b>	Monitor Client Node
<b>ONNX</b>	Open Neural Network Exchange
<b>RAM</b>	Random Access Memory
<b>vRAM</b>	Video Random Access Memory
<b>FFN</b>	Feed-Forward Network
<b>GPT</b>	Generative Pre-trained Transformer
<b>HPC</b>	High-Performance Computing
<b>HTTP</b>	Hypertext Transfer Protocol
<b>TCP</b>	Transmission Control Protocol
<b>UDP</b>	User Datagram Protocol
<b>AI</b>	Artificial Intelligence
<b>API</b>	Application Programming Interface
<b>ML</b>	Machine Learning
<b>BLOOM</b>	BigScience Large Open-science Open-access Multilingual Language Model
<b>H100</b>	NVIDIA Hopper GPU
<b>A100</b>	NVIDIA Ampere 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 Browser-Based Language Model Inference," 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. Research by Li et al. [2] indicates that consumer-grade systems lack the necessary resources for effective use of large-scale models.

Decentralized AI approaches offer a solution by leveraging distributed computing power from end-user devices. Studies, such as Chen et al. [3], 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.

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

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 in a 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. **An In-Browser Decentralized Language Model Inference:** Develop a system that enables AI language model inference directly within web browsers, eliminating the need for centralized servers. This system leverages WebGPU and WebAssembly technologies to utilize the computational power of client devices, ensuring efficient and seamless inference processes.

## 1.4 Scope

1. **In-Browser AI Inference:** Develop AI models that run directly within 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* [4] 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* [5] 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 paper *WebInf: Accelerating WebGPU-based In-browser DNN Inference via Adaptive Model Partitioning* [7] 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* [8] by Mitsuhsa 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.
3. The article *The Future of Consumer Edge-AI Computing* by Stefanos Laskaridis et al. [9] 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.
4. The paper *Federated Learning: Challenges, Methods, and Future Directions* [10] 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.
5. *Model-Distributed Machine Learning* [11] 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* [12] 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-Limit*, where peers need to exchange intermediate model outputs efficiently.
2. *P2P Networking and DHTs in Decentralized Systems* [13] 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.

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

1. *Scaling Up Language Models: GPT-3* [14] 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* [15] 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* [16] 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.

## 2.2 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 [10].
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

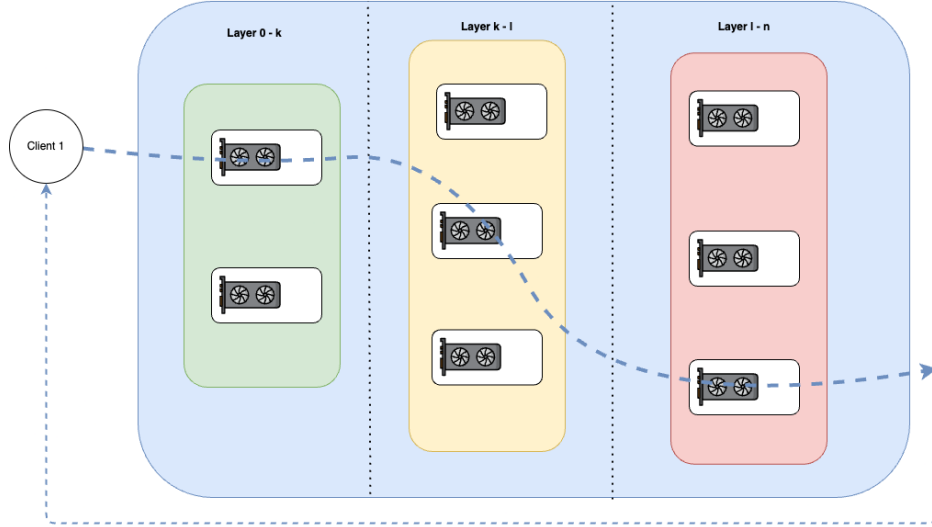


Figure 3.1: Overview of inference in De-LIMIT.

*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-LIMIT 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-LIMIT, 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-LIMIT

The primary innovation in De-LIMIT 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.

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-LIMIT 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.

The bottleneck here can be the communication between the node. De-LIMIT tackles this by employing a peer-to-peer network type of architecture. 3 different types of nodes are employed in the De-LIMIT 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 active peers, layers the peer holds, and the address of the peers.

**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. These nodes can also be used as interface of the browser Instance Nodes.

**Browser Instance Nodes** These are special types of nodes, which is first used by De-LIMIT. This type of nodes are present in the browsers of consumer-grade devices. A compiled shredded model (ONNX format) is loaded in the browser which is then infered within the browser using the available hardware accelerators.

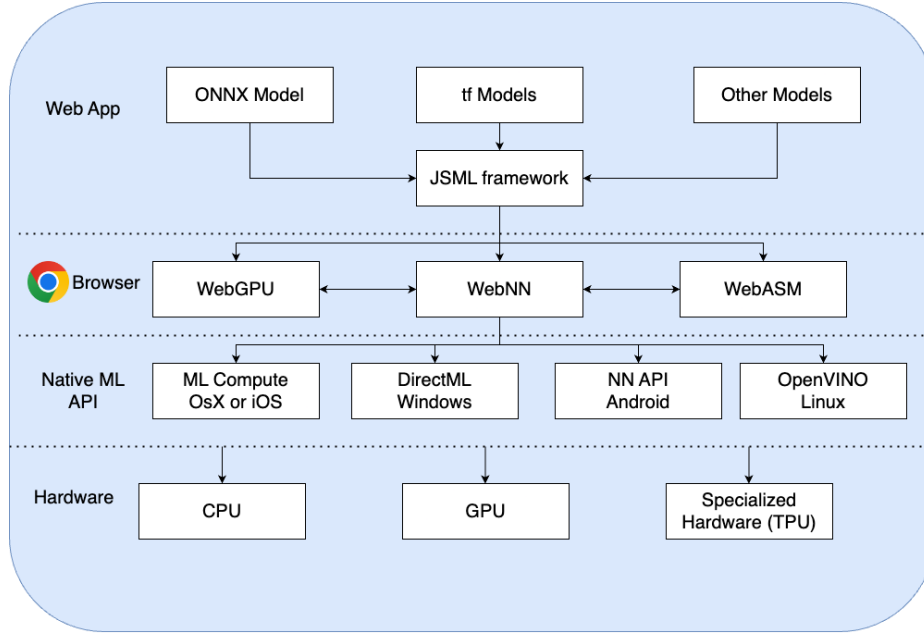


Figure 3.2: Working of Compiled Models in Browser Instance Nodes.

[21]

The *Monitor Client Nodes* and *Terminal Instance Nodes* are responsible for maintaining the DHT in addition to creating a serialized path of inference within the P2P 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 Monitor Client Nodes to sustain the network.

The main problem the system then faces will be the number of Browser Instance Nodes connecting and disconnecting nodes. This should be addressed by the serialization protocol which we aim to employ in the Backbone nodes viz. Monitor Client Nodes and Terminal Instance Nodes.

## 3.2 Working Flow of the System

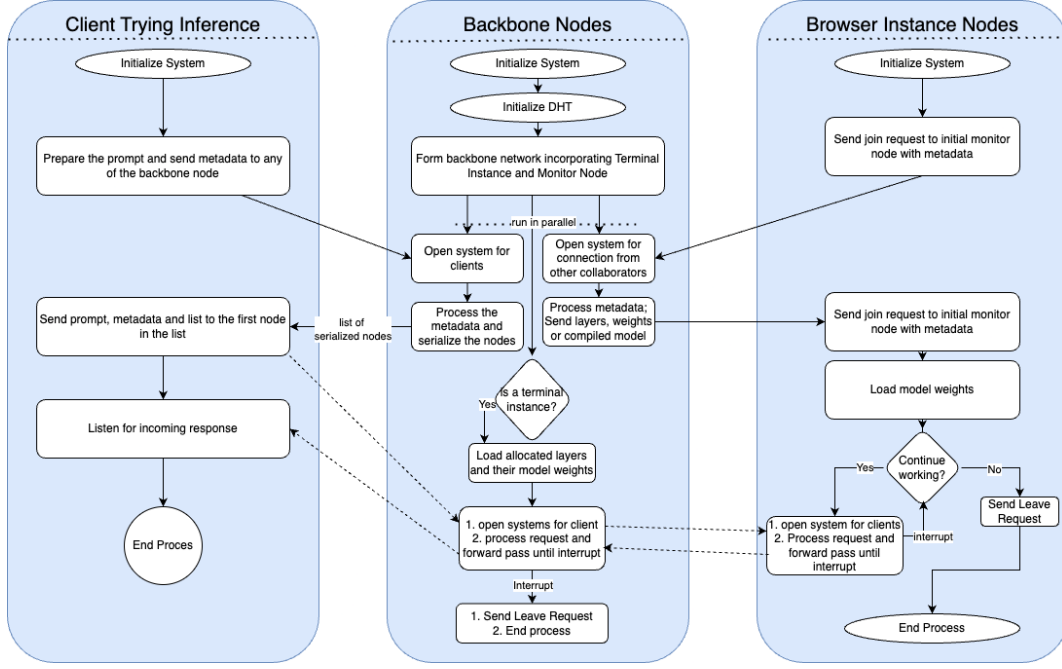


Figure 3.3: High-level workflow of the system.

The workflow presented illustrates the interaction between Client Nodes, Backbone Nodes, and Browser Instance Nodes within the De-LIMIT system for distributed inference. The backbone nodes are initialized with their layers and weights, if is a terminal instance, 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 responsible for managing the Distributed Hash Table (DHT) and sharded model layers. 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 and data exchange between peers. Each peer announces its presence and registers on the DHT, making its computational resources available for decentralized inference. The DHT stores the intermediate outputs generated by the model during the inference process, which other peers can retrieve to continue the computation. 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 backbone nodes comprising of terminal instance nodes and monitor client nodes. The backbone nodes are responsible for maintaining the DHT, and any new node, any of the 3, which wants to 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 peer-to-peer (P2P) network. Two key processes take place during this phase: *model sharding* and *peer distribution*.

#### Model Sharding and Distribution

- **Model Sharding:** The Large Language Model (LLM) is divided into smaller components known as *shards*. Each shard corresponds to a subset of model layers, such as Transformer blocks, Attention-Head blocks, and Feed-Forward Networks (FFNs). These shards are small enough to be processed on consumer-grade hardware.
- **Distribution:** The shards are distributed across three types of nodes:
  - **Monitor Client Nodes (MCM):** These nodes maintain a Distributed Hash Table (DHT) containing the list of active peers, the layers they hold, and their network addresses.
  - **Terminal Instance Nodes (TIN):** Representing medium to high-performance devices, these nodes are responsible for running specific shards of the model and handling computations like forward passes.
  - **Browser Instance Nodes (BIN):** Shards are loaded in browsers using formats like ONNX, where they are executed using hardware accelerators available in consumer-grade devices. WebNN is used to run the model efficiently within the browser environment.

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 tokenized prompt and other necessary metadata, to one of the backbone nodes, either a **Monitor Client Node** or a **Terminal Instance Node**.
- The backbone node handles the request by identifying the path through the network required for performing the inference based on the distributed layers.

### 3.2.4 Response for Path

Once the backbone node receives the request, it generates a serialized path that outlines how the model shards will be processed sequentially across the nodes.

#### Serialized Path for Inference

- The backbone node refers to the Distributed Hash Table (DHT) 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.
- The path is returned to *Client-1*, detailing which peers will handle each shard of the model.

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.

### 3.2.5 Forward Pass through Model Shards

With the serialized path received, the client can begin the inference process by sending the tokenized prompt and associated metadata to the first node in the sequence.

#### Execution of Forward Pass and Inter-Node Communication

- *Client-1* sends the tokenized 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 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.

The communication between nodes is managed by the P2P network, where each node contacts the next one using the information provided by the DHT. This ensures seamless inter-node communication throughout the forward pass.

### 3.2.6 Final Output to the Client

Once the forward pass is complete, the final node generates the output logits for the entire inference process.

#### 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 *Client-1*, completing the inference request.

The distributed architecture of De-LIMIT 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 P2P network.

### 3.2.7 Fine-Tuning in De-LIMIT Architecture

Along with inference, the De-LIMIT 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 distributed 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 (either a Terminal Instance Node or a Browser Instance Node), depending on the computational capacity of the device.

## **Loading the Fully Connected Layer**

For fine-tuning, the client device is responsible for loading the fully connected layer of the model. This is the final layer, typically used for classification or other downstream tasks, and it is where the fine-tuning occurs.

## **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—the fully connected layer, which resides on the client device.

## **Fine-Tuning the Fully Connected Layer**

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 client updates the weights of the fully connected layer based on the gradients calculated during backpropagation. Since the fully connected layer is relatively small, it can be fine-tuned efficiently on the client device without the need for distributing this task to other nodes.

## **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.

## **Completion of Fine-Tuning**

After the backward pass completes, the fine-tuning process concludes. The client continues to use the updated fully connected layer for subsequent tasks, while the distributed layers across the network are also updated and can be utilized for further fine-tuning or inference. This architecture allows efficient fine-tuning of large models on consumer-grade devices by leveraging distributed computing across the network.

# 4. System Design

The architecture for the Delimit consists for 3 main layers:

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

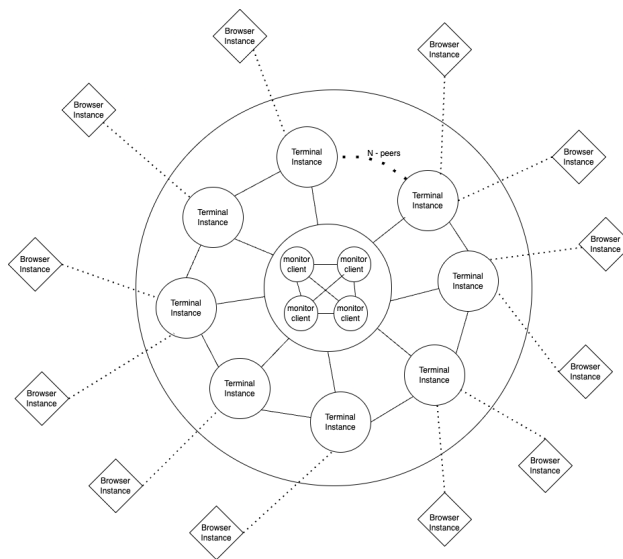


Figure 4.1: System Design

## 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 lower 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 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.

These 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. Work Completed

## 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.

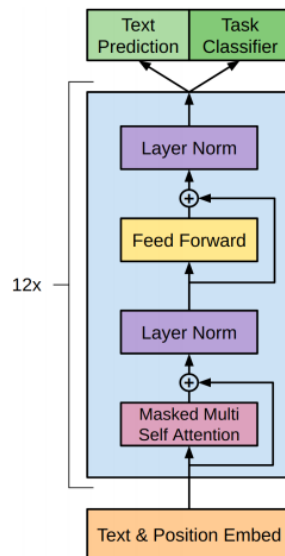


Figure 5.1: Architecture of a generic LLM

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.

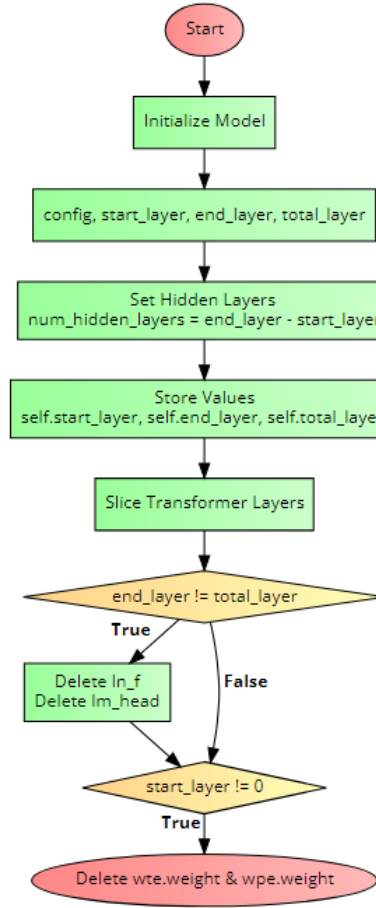


Figure 5.2: Sharding of a LLM Model

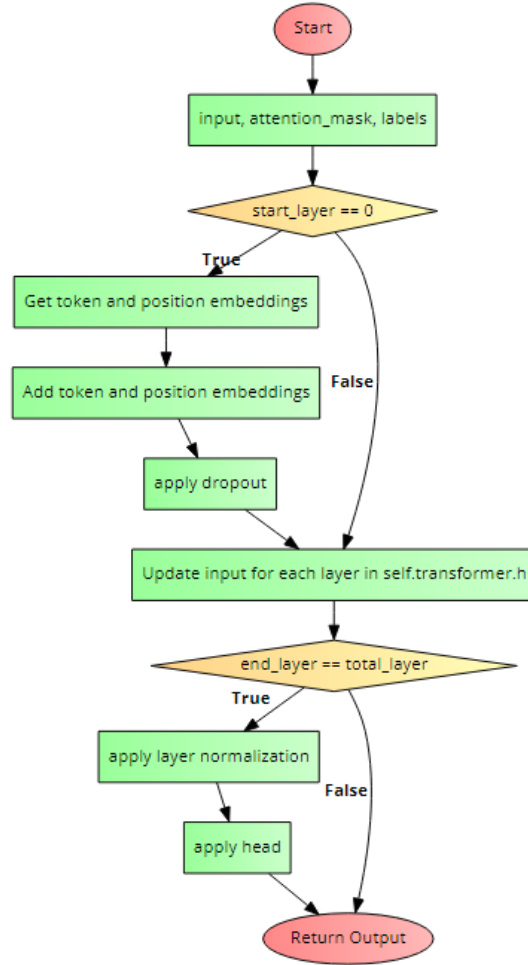


Figure 5.3: Forward Pass

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:

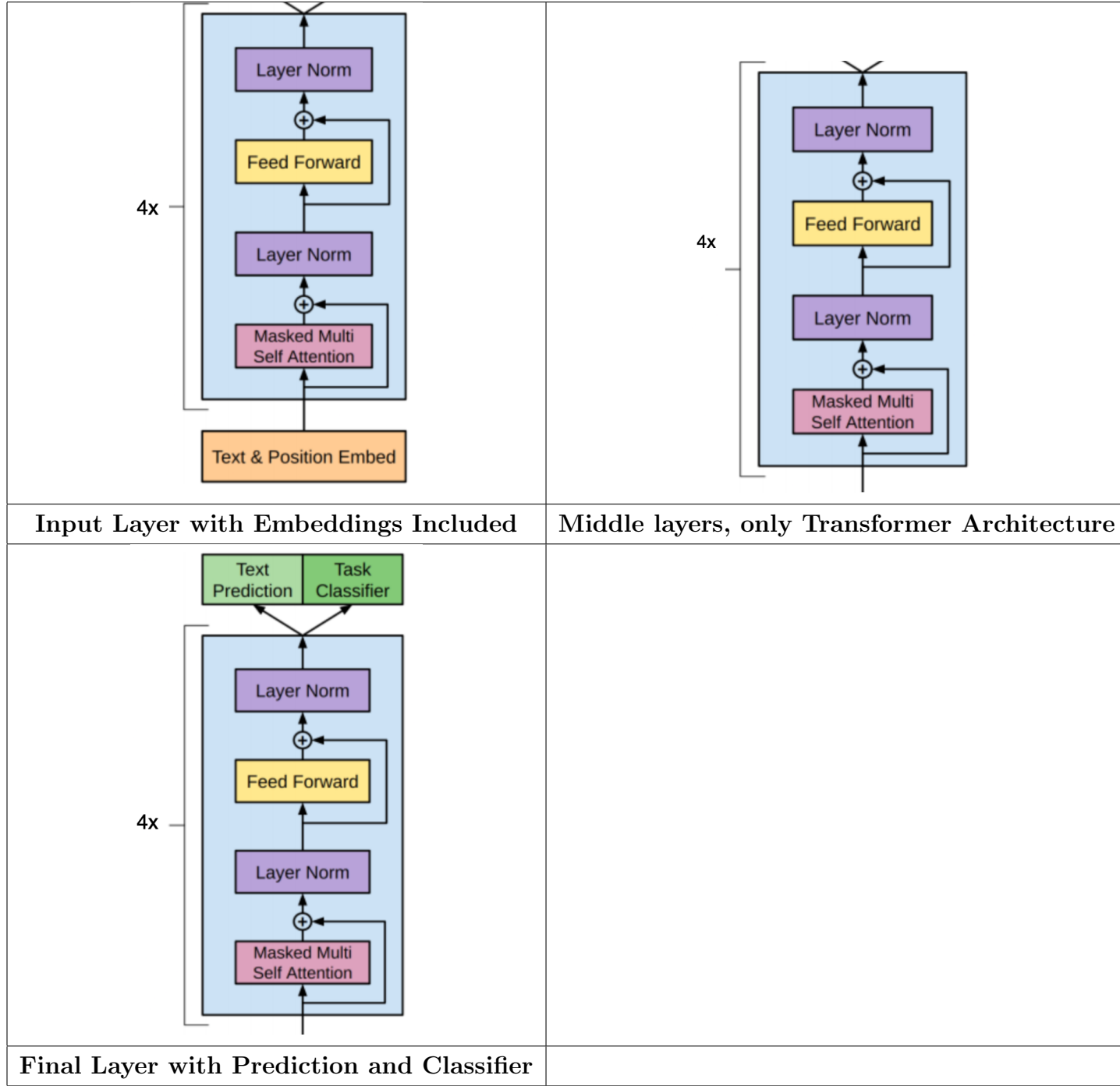


Table 5.1: Visualization of Different Layers in the Model

### 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.

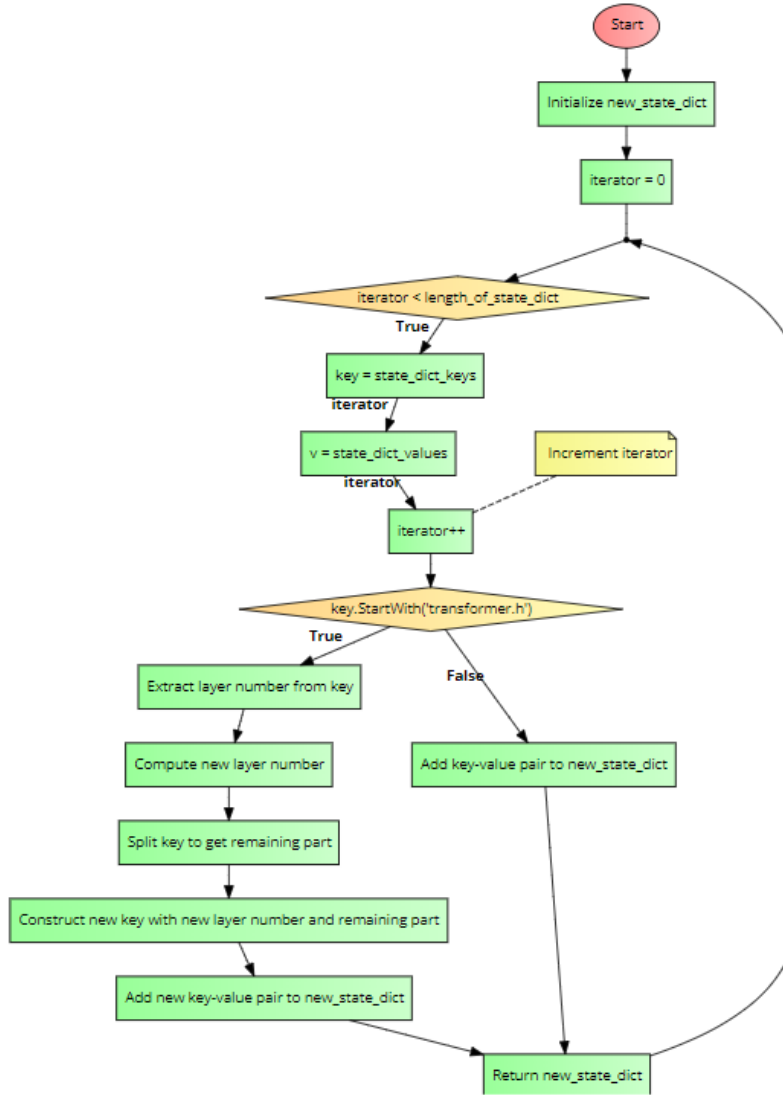


Figure 5.4: Remapping State Dict Keys

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.

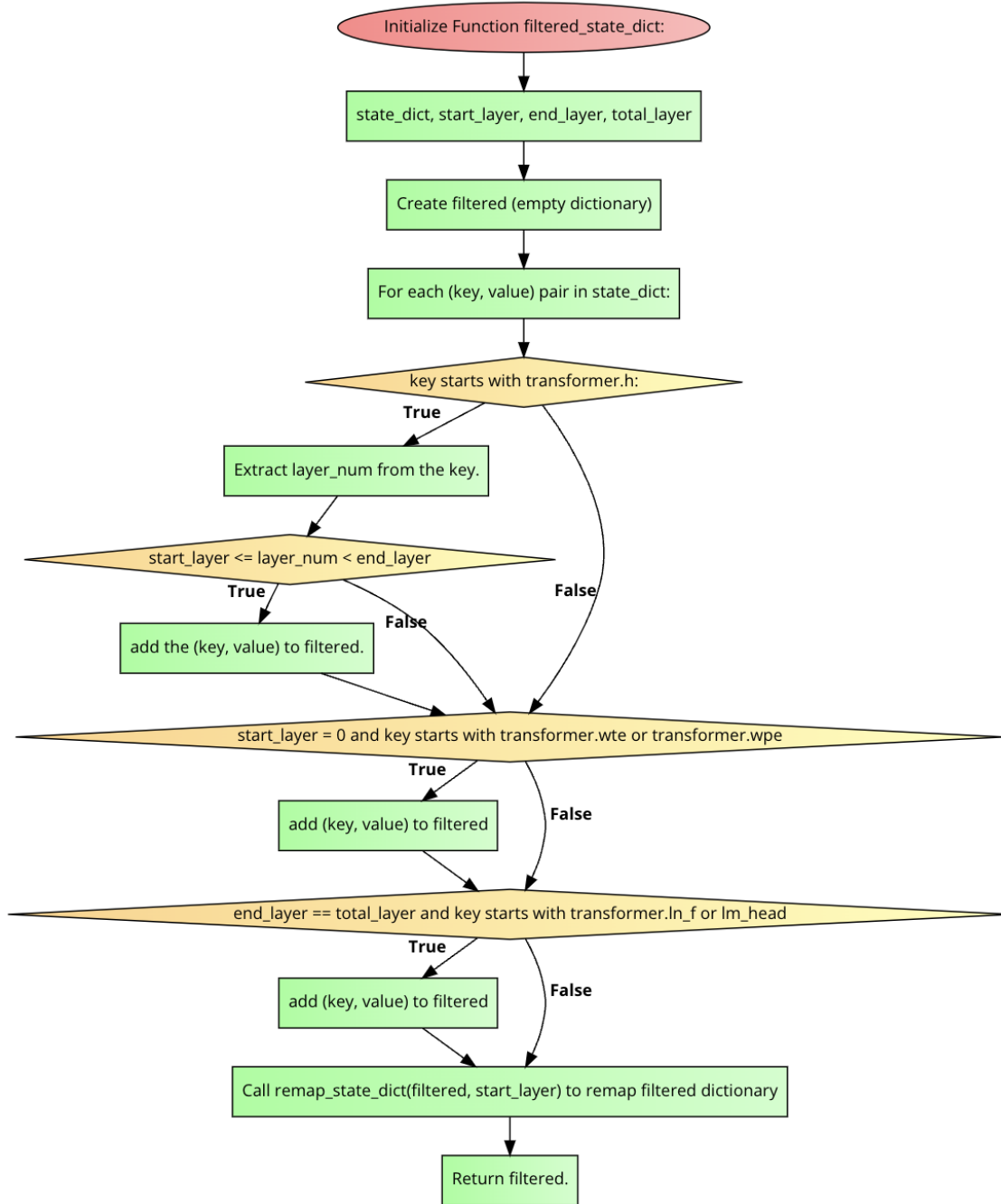


Figure 5.5: Filtered State Dict for a Shard

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 the available P2P communication and TCP connection or stored on a server, 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.



## 6. Future Work

This section outlines the tasks that will be completed after the mid-term defense in the *De-Limit* project.

### 6.1 Open Collaboration Incorporating NAT Traversal

To make the *De-Limit* system more decentralized and improve open collaboration, we need to tackle the challenges posed by Network Address Translation (NAT). NAT often makes it difficult for peers behind different NATs to talk directly without some help. Here are a few ideas we can explore:

- **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.

### 6.2 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:

- **Web Interface Development:** Design and implement a user-friendly web interface where users can easily upload inputs and view results of the inference without requiring any terminal interaction.
- **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.
- **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.

## 6.3 Further Improvements and Work

To enhance the *De-Limit* 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.

## 7. 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-LIMIT 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-LIMIT 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-LIMIT 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] 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>.
- [4] 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>.
- [5] 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>.
- [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] 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>.
- [8] 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>.
- [9] 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>.

- [10] 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/10.1561/22000000083>.
- [11] T. Ben-Nun and T. Hoefler, “Model-distributed machine learning,” *ACM Computing Surveys (CSUR)*, vol. 52, no. 4, pp. 1–36, 2019.
- [12] 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.
- [13] X. Wang, L. Liu, B. Yang, C. Li, and Y. Zhang, “P2p networking and dhds in decentralized systems,” *IEEE Transactions on Network and Service Management*, vol. 17, no. 3, pp. 1234–1250, 2020.
- [14] 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.
- [15] 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>.
- [16] 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.
- [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.

# APPENDIX A

## Output and Inference

In this experiment, we used De-LIMIT to create a sharded model distributed across two peers. Specifically, we distributed the layers of a GPT-2 Small model (12 layers total) between the peers, with the first peer holding layers 0 to 6 and the second peer holding layers 6 to 12.

### Setup :

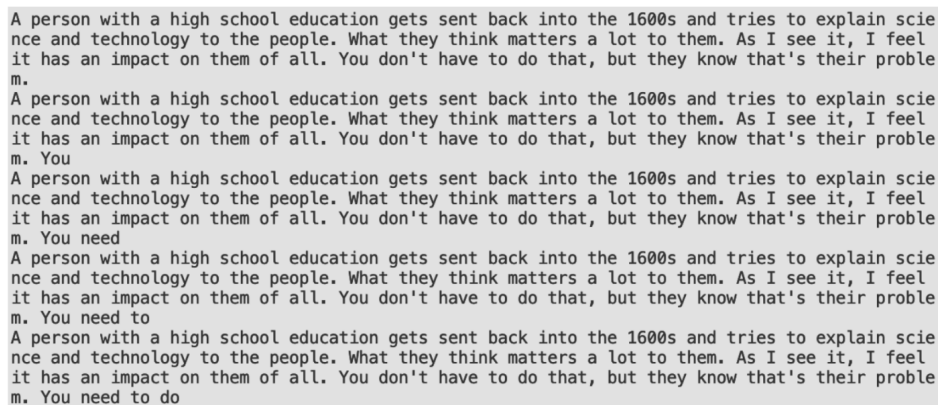
Peer 1: Layers 0 to 6 Peer 2: Layers 6 to 12 Model: GPT-2 Small (124M parameters)

We provided a prompt and executed the model in a distributed fashion, where each peer contributed its part of the forward pass. The inference output was successfully generated, although with some limitations in quality and accuracy due to the small size of the GPT-2 model.

**Results:** The output reflected the GPT-2 Small model's inherent limitations, such as less coherent text generation and fewer meaningful patterns in longer sequences. This is expected, as GPT-2 Small is less accurate in generating high-quality outputs compared to larger models.

### Screenshots :

#### Peer-1:



A person with a high school education gets sent back into the 1600s and tries to explain science and technology to the people. What they think matters a lot to them. As I see it, I feel it has an impact on them of all. You don't have to do that, but they know that's their problem. You

A person with a high school education gets sent back into the 1600s and tries to explain science and technology to the people. What they think matters a lot to them. As I see it, I feel it has an impact on them of all. You don't have to do that, but they know that's their problem. You

A person with a high school education gets sent back into the 1600s and tries to explain science and technology to the people. What they think matters a lot to them. As I see it, I feel it has an impact on them of all. You don't have to do that, but they know that's their problem. You need

A person with a high school education gets sent back into the 1600s and tries to explain science and technology to the people. What they think matters a lot to them. As I see it, I feel it has an impact on them of all. You don't have to do that, but they know that's their problem. You need to

A person with a high school education gets sent back into the 1600s and tries to explain science and technology to the people. What they think matters a lot to them. As I see it, I feel it has an impact on them of all. You don't have to do that, but they know that's their problem. You need to do

Figure 7.1: Output from peer loading layers 0 to 6

#### Peer-2:

```

tensor([[[ -68.1045, -66.5724, -69.1511, ..., -76.4108, -74.8194,
           -67.5567],
          [-113.8865, -111.1190, -117.0282, ..., -120.5204, -117.8485,
           -112.3448],
          [-80.2682, -77.4696, -81.6062, ..., -81.6042, -82.8660,
           -79.2532],
          ...,
          [-126.8072, -124.7483, -131.8362, ..., -126.6839, -129.7671,
           -125.6535],
          [-119.6819, -119.8686, -129.2626, ..., -127.7316, -128.0069,
           -122.2026],
          [-139.2656, -137.9473, -140.4463, ..., -148.1424, -151.7383,
           -132.3405]]], grad_fn=<UnsafeViewBackward0>)

What
tensor([[[ -95.9951, -93.4422, -96.7608, ..., -105.8759, -102.0297,
           -96.3085],
          [-112.8814, -110.4468, -117.2183, ..., -118.7617, -116.3579,
           -112.7281],
          [-103.3433, -98.2888, -105.0266, ..., -104.8888, -107.5100,
           -103.2637],
          ...,
          [-94.1869, -93.8140, -102.4643, ..., -102.9069, -102.5614,
           -96.0070],
          [-139.1981, -137.5769, -140.1206, ..., -150.4780, -153.3722,
           -132.0227],
          [-130.5598, -132.1706, -135.8128, ..., -137.7500, -140.8928,
           -133.0457]]], grad_fn=<UnsafeViewBackward0>)

they
tensor([[[ -73.3074, -71.7930, -73.7626, ..., -80.6290, -78.8228,
           -72.4482],
          [-128.6103, -125.4200, -132.6918, ..., -136.0874, -133.4266,
           -127.9448],
          [-90.6275, -83.8941, -91.2571, ..., -89.4361, -92.2675,
           -87.9269],
          ...,
          [-165.6028, -163.3201, -166.7379, ..., -174.3250, -177.9392,
           -158.8521],
          [-115.2437, -117.1101, -119.6985, ..., -121.8125, -125.1616,
           -117.4640],
          [-154.7905, -154.4700, -159.1801, ..., -163.7935, -160.1620,
           -155.6284]]], grad_fn=<UnsafeViewBackward0>)

think
tensor([[[ -55.3763, -53.8667, -56.7583, ..., -64.6458, -62.6961,
           -54.5359],
          [-109.1046, -105.4873, -112.7066, ..., -115.6011, -111.5086,
           -108.4732],
          [-92.5596, -87.3656, -94.0556, ..., -89.9617, -93.5438,
           -91.5607],
          ...,
          [-130.7665, -132.4689, -136.3932, ..., -135.3525, -138.9864,
           -133.3967],
          [-146.8143, -146.3755, -152.5399, ..., -155.8020, -151.1479,
           -148.4802],
          [-79.7167, -82.5336, -88.7897, ..., -90.8106, -92.8131,
           -84.2084]]], grad_fn=<UnsafeViewBackward0>)

matters
tensor([[[ -75.3877, -74.0462, -77.0320, ..., -83.5529, -82.5827,
           -74.9760],
          [-99.8501, -98.0455, -104.2693, ..., -107.3671, -103.6364,
           -100.0065],
          [-85.0656, -81.4196, -86.6094, ..., -84.0650, -87.1409,
           -84.0043]

```

Figure 7.2: Output from peer loading layers 6 to 12

**Prompt** A person with a high school education gets sent back into the 1600s and tries to explain science and technology to the people.

**Inference Results Result 1:** A person with a high school education gets sent back into the 1600s and tries to explain science and technology to the people. What they think matters a lot to them. As I see it, I feel it has an impact on them of all. You don't have to do that, but they know that's their problem. You need to do



**Result 2:** A person with a high school education gets sent back into the 1600s and tries to explain science and technology to the people.

"I have good teachers and bad teachers, and if something is wrong, I don't feel any better about it's wrong path," the person says, as if they know he's still a kid.

When that person asks to see a picture, the student does not get a response.

The person who is "wrong" gets a second response.

"I don't know who you are, but I