UNIVERSITÀ DEGLI STUDI DI MESSINA

DIPARTIMENTO DI SCIENZE MATEMATICHE E INFORMATICHE, SCIENZE FISICHE E SCIENZE DELLA TERRA

Bachelor's Degree in Computer Science L31

Curriculum: Data Analysis

Optimizing the Deployment of Large Language Models on Edge Devices: A Quantization Approach

Supervisor:

Prof. Lorenzo Carnevale

Candidate:

Tharun Reddy Banda

Accademic Year
2023/2024

Contents

List of Figures

CHAPTER 1

Introduction

1.1 Introduction

The rapid advancements in **Natural Language Processing (NLP)** have led to the development of sophisticated **Large Language Models (LLMs)** such as **GPT**, **BERT**, and **T5**, which have revolutionized the field. These models, powered by **transformer architectures**, perform complex tasks such as **text generation**, **machine translation**, **sentiment analysis**, and **question answering** with impressive accuracy. By learning from massive amounts of data, LLMs have demonstrated high versatility across various industries, offering solutions in areas ranging from **customer service automation** to aiding in **medical diagnosis**.

Despite the capabilities of these models, they are predominantly deployed in **cloud-based infrastructures**, where vast computational resources, such as **Graphics Processing Units (GPUs)** or **Tensor Processing Units (TPUs)**, are available to handle their high processing demands. While cloud-based systems offer the necessary power to run these large models, they introduce significant challenges, including

high latency, **privacy concerns**, and the **reliance on stable network connectivity**. Applications that require real-time data processing, such as **autonomous vehicles** or **smart health monitoring systems**, can be severely impacted by these limitations, resulting in slower response times, reduced reliability, and heightened security risks.

To mitigate these challenges, **edge computing** has emerged as a viable solution. Edge computing enables data processing to occur closer to the data source, on local devices such as **smartphones**, **Raspberry Pi**, and other **Internet of Things (IoT)** devices, rather than relying on centralized cloud servers. By moving computation to the edge, systems can achieve **low-latency responses**, which are critical for real-time applications, while also improving **data privacy** by limiting the transmission of sensitive information to external servers. Moreover, edge computing allows for greater **scalability** and enables operation in **low-connectivity environments** where stable network connections are not guaranteed.

However, deploying **LLMs** on edge devices presents its own set of challenges. The most pressing challenge is the **limited hardware resources** of edge devices, which lack the **processing power**, **memory**, and **energy efficiency** available in cloud servers. Edge devices are designed for lightweight, low-power applications, and do not have the capacity to handle the massive computational loads required by large models like **GPT-3**, which has 175 billion parameters. For instance, GPT-3 requires terabytes of storage and specialized hardware for efficient inference, far exceeding the specifications of devices such as **Raspberry Pi** or **smartphones**.

Another significant issue is **latency**. In cloud-based deployments, data must be transmitted to remote servers for processing, and the results returned to the edge device. This introduces delays, which can be problematic for **real-time applications**. For example, voice assistants and smart home systems require instant responses to user inputs, and **autonomous vehicles** need immediate data processing to ensure safety. Any delay in these systems could negatively impact user experience or, worse, result in system failures.

Additionally, there are **data privacy** and **security concerns** associated with trans-

mitting data to cloud servers for LLM inference. Sensitive information, particularly in industries such as **healthcare** and **finance**, is exposed to higher risks of **data breaches** and unauthorized access when transmitted over the internet. **Edge-based LLM deployment** addresses this issue by processing data locally on the device, thereby reducing the need to transmit sensitive information and limiting exposure to external threats.

1.2 Problem Identification

The central challenge this research addresses is the difficulty of deploying **Large Language Models (LLMs)** on **resource-constrained edge devices**. These devices, including **Raspberry Pi**, smartphones, and **embedded systems**, typically have limited **processing power**, **memory**, and **battery life**, making the direct deployment of large-scale models like GPT-3 impractical.

Model size is a critical factor. LLMs are built to leverage the vast computational resources of the cloud, including extensive memory and powerful parallel processing. The computational requirements of models like GPT-3 far exceed the hardware limitations of edge devices. Furthermore, the issue of **latency** is exacerbated when relying on cloud servers for inference. Data must be transmitted to a remote server, processed, and sent back to the edge device, leading to delays. This is particularly detrimental for applications requiring **real-time responses**.

Additionally, concerns about **data privacy** and **security** are paramount. Edge-based deployment offers a viable solution by ensuring that sensitive data is processed locally on the device, minimizing the risk of **data breaches** and reducing dependency on third-party cloud services.

1.3 Proposed Solution

To overcome these challenges, this research focuses on **model optimization** and the use of **efficient inference tools** specifically designed for edge devices. The key components of the proposed solution are:

1. **Model Optimization via Quantization:** Quantization is a crucial technique for reducing the **memory footprint** and **computational complexity** of LLMs. By converting high-precision **32-bit floating-point (FP32)** weights to lower precision formats such as **16-bit floating-point (FP16)** or **8-bit integer (INT8)**, quantization can significantly reduce the resource requirements of LLMs. This research explores two primary approaches to quantization: - **Post-Training Quantization (PTQ):** This method applies quantization to a pre-trained model without requiring retraining, making it efficient for models already deployed in full precision. - **Quantization-Aware Training (QAT):** This method incorporates quantization into the training process, allowing the model to adapt to the reduced precision, often leading to better performance in tasks where precision is crucial.

2. **Efficient Inference Tools for Edge Devices:** Several open-source tools have been developed to facilitate the deployment of quantized models on edge devices. Tools such as **Langchain**, **Ctransformars**, **Llama-cpp**, **Ollama**, and **Llamafile** are evaluated in this research. These tools are assessed based on their **response time**, **memory usage**, **CPU utilization**, and **ease of integration**. The goal is to identify the most suitable tools for deploying quantized LLMs on **resource-constrained hardware**.

1.4 Objectives

The primary objectives of this research are as follows:

- **Objective 1:** To investigate the advantages and challenges of deploying **LLMs** on edge devices, focusing on the reduction of **latency**, improvement of **data**

privacy, and the ability to function in **low-connectivity** environments.

- **Objective 2:** To explore and apply **quantization techniques** that reduce the **size** and **computational complexity** of LLMs, while maintaining acceptable performance levels.
- **Objective 3:** To evaluate the performance of various **inference tools** for deploying quantized LLMs on edge devices by measuring **response times**, **hardware utilization**, and overall **inference efficiency**.
- **Objective 4:** To assess the **environmental impact** of deploying LLMs on edge devices, particularly focusing on energy consumption and estimating **CO2 emissions**.

1.5 Report Organization

The remainder of this thesis is organized as follows:

- **Section 2:** A detailed discussion on the **need for moving LLMs to edge devices**, exploring the benefits, challenges, and state-of-the-art techniques for LLM deployment on edge.
- **Section 3:** A description of the **materials and methods** used in this research, including the quantization techniques and the inference tools evaluated.
- **Section 4:** The **experimental results**, presenting a comparison of the performance of different inference tools and quantized models on edge devices such as **AWS EC2 instances** and **Raspberry Pi 400**.
- **Section 5:** The **conclusion**, summarizing the findings and discussing future research directions for optimizing LLM deployment on edge.

CHAPTER 2

State of the art

2.1 Historical Development of LLMs: From Statistical Models to Transformers

Early advancements in NLP were driven primarily by traditional statistical methods. Models like n-grams and Hidden Markov Models (HMMs) were pivotal in tasks such as speech recognition and text generation. However, these models suffered from limitations in capturing long-range dependencies due to their reliance on fixed-length sequences or windows of words [6].

The advent of Recurrent Neural Networks (RNNs) marked a significant shift, especially with Long Short-Term Memory (LSTM) networks. LSTMs introduced a memory mechanism that allowed models to retain information across longer sequences. Despite this, LSTMs struggled with the vanishing gradient problem, often losing important contextual information when dealing with very long sequences [5].

The true breakthrough came with the introduction of the Transformer architecture by Vaswani et al. (2017). This model introduced the self-attention mechanism, which

enabled the model to weigh the importance of different words in a sentence by considering the entire context simultaneously. This architecture significantly improved the efficiency of training on large datasets, making it a foundation for modern LLMs [20].

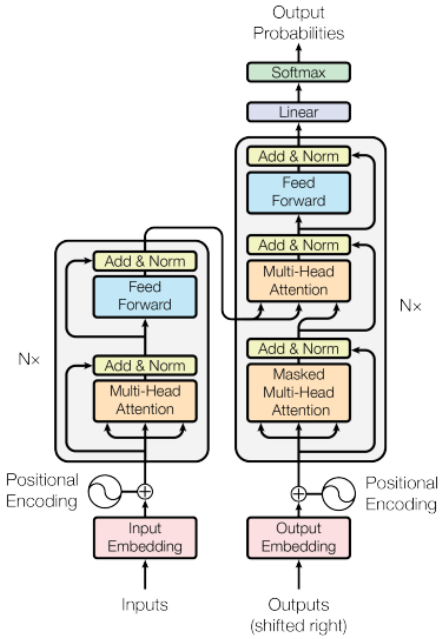

Figure 2.1: Transformer Architecture. This figure illustrates the encoder-decoder structure of the transformer model, highlighting the key components such as Multi-Head Attention, Feed Forward Networks, and Positional Encoding.

The transformer became the backbone of models like GPT (Generative Pre-trained Transformer), BERT (Bidirectional Encoder Representations from Transformers), and T5 (Text-to-Text Transfer Transformer). Notably, GPT-2 and GPT-3 scaled these models to billions of parameters, showcasing their capabilities in generating human-like text and performing various complex tasks [2].

2.2 Scalability of LLMs: From Cloud to Edge

LLMs have traditionally been deployed in cloud-based systems, benefiting from the scalability and computational resources that cloud infrastructure offers. Access to high-performance hardware, such as GPUs and TPUs, enables the training and inference of models as large as GPT-3 [15]. Cloud platforms offer flexibility in resource

allocation, allowing researchers and developers to manage computational demands dynamically.

A key advantage of cloud-based deployment is the centralization of processing and data storage. By hosting models in the cloud, large datasets can be processed without requiring local storage, ensuring seamless updates and high availability. However, reliance on cloud infrastructure introduces challenges related to latency, privacy, and operational costs. Real-time applications, such as voice assistants and autonomous systems, may suffer from network latency due to the need for constant communication between devices and cloud servers [10].

2.3 The Move to Edge Computing for LLM Deployment

To address these challenges, edge computing has emerged as a solution. By moving data processing closer to the source—on local devices such as smartphones or IoT devices—edge computing reduces the dependency on cloud infrastructure, thus improving latency and privacy [14]. The reduced latency is critical for real-time applications, where even milliseconds of delay can impact performance [16].

Deploying LLMs on edge devices provides several benefits:

- **Reduced Latency**: Local processing cuts down the time needed for inference, making it ideal for applications requiring real-time responses, such as smart home systems and voice assistants.
- **Enhanced Privacy**: By keeping data processing local, edge computing minimizes the risk of data breaches and gives users greater control over their information [16].
- **Lower Bandwidth Usage**: Since most of the data processing happens locally, only essential information needs to be sent to the cloud, reducing overall bandwidth usage.

However, deploying LLMs on edge devices also presents challenges related to hardware limitations, as edge devices typically have constrained resources compared to cloud servers [12]. This necessitates model optimization techniques like quantization and pruning, which we will discuss in next Section.

2.4 Edge Computing and LLMs

The deployment of Large Language Models (LLMs) on edge computing devices represents a major shift in data processing paradigms. By bringing computational tasks closer to the data source, edge computing provides an opportunity to address many of the challenges associated with cloud-based LLM deployment, such as network latency and privacy concerns. This section explores the key advantages of edge computing for LLMs and the challenges it presents, as well as various strategies for optimizing LLMs for edge deployment.

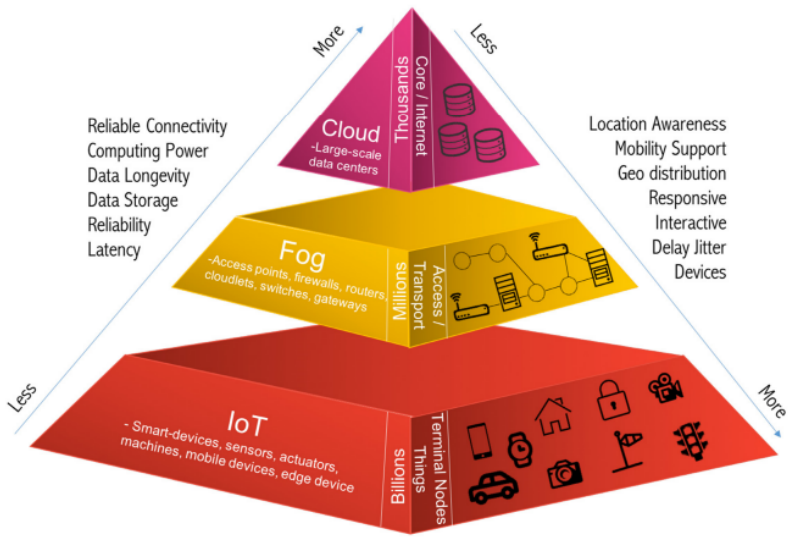

Figure 2.2: Edge Computing Architecture: Layers of Edge Devices, Fog Nodes, and Cloud Servers.

2.4.1 What is Edge Computing?

Edge computing refers to the practice of processing data closer to where it is generated, as opposed to relying on distant, centralized cloud servers. In the context

of LLMs, this means moving the inference and sometimes even training processes onto local devices such as smartphones, Raspberry Pi boards, or Internet of Things (IoT) devices. This shift to localized processing can alleviate issues related to network latency, bandwidth consumption, and data privacy [16].

An edge computing architecture typically consists of multiple layers:

- **Edge Devices**: Devices like smartphones or IoT sensors that generate data and perform local processing.
- **Fog Nodes**: Intermediate processing nodes situated between edge devices and cloud servers, offering additional computational power closer to the data source.
- **Cloud Servers**: While edge computing reduces reliance on the cloud, it remains essential for handling more complex, resource-intensive tasks.

By leveraging this layered structure, edge computing allows LLMs to be deployed in environments where real-time data processing is critical, such as autonomous vehicles, industrial IoT systems, and smart home devices [14].

2.4.2 Motivations for Deploying LLMs on Edge Devices

Several key factors drive the need for LLM deployment on edge devices. While cloud-based systems provide the computational resources to train and run LLMs, edge devices can offer unique advantages in certain scenarios. Below, we explore the primary motivations for moving LLMs to the edge:

Low Latency

One of the most important motivations for deploying LLMs on edge devices is reducing latency. Applications such as voice assistants, autonomous systems, and medical devices require real-time responses. When LLMs are deployed in the cloud, response times are dependent on network connectivity and speed, which can

introduce delays. In contrast, edge devices allow inference to happen locally, thereby minimizing the time taken to process and respond to input [16].

For example, in autonomous vehicles, even small delays in processing sensor data can have catastrophic consequences. Similarly, smart healthcare devices that monitor patient conditions in real time must respond instantly to detect and address anomalies. In these cases, edge computing ensures that inference occurs in a timely manner, improving safety and user experience [12].

Enhanced Privacy and Security

Edge computing enhances privacy by processing data locally on the device rather than transmitting it over the internet. This is particularly important for sensitive applications, such as healthcare and financial services, where privacy concerns are paramount [7]. By keeping data on the edge device, users can avoid the risks associated with transmitting personal information to cloud servers, reducing the potential for data breaches or unauthorized access [16].

For instance, patient data collected by a wearable health monitor can be processed locally, ensuring that sensitive medical information remains private and is not shared with external servers. This localized processing not only improves security but also provides greater control over data management.

Reduced Bandwidth Consumption

Edge computing reduces the amount of data that needs to be transmitted to and from cloud servers, conserving bandwidth in applications that generate large volumes of data, such as IoT systems. By processing data locally on edge devices, only essential insights are sent to the cloud for further analysis, reducing the overall bandwidth load [10].

This is particularly important in environments where bandwidth is a limited or expensive resource. For example, in remote industrial operations, IoT devices might continuously generate data about equipment performance. By performing

local processing, only critical data or anomalies need to be transmitted to the cloud, significantly reducing the burden on network infrastructure.

Personalization

Deploying LLMs on edge devices allows for a higher degree of personalization. By processing data locally, LLMs can learn and adapt to a user's unique habits, preferences, and behavior over time. This enables the model to fine-tune responses and provide a more tailored experience without the need to transmit personal data to external servers [7].

For example, a smartphone-based LLM can learn a user's speech patterns and frequently used terms to improve the accuracy of voice recognition tasks. This personalized interaction is further enhanced by the fact that the data remains local, ensuring that user preferences are learned and applied without violating privacy.

2.4.3 Challenges of Deploying LLMs on Edge Devices

While edge computing offers numerous advantages for LLM deployment, there are significant challenges that need to be addressed. The most notable issues arise from the resource constraints of edge devices, which generally have far less processing power, memory, and storage than cloud servers.

Limited Computational Resources

Edge devices such as smartphones or Raspberry Pi boards lack the high computational power and memory resources of cloud infrastructure. Running a model as large as GPT-3, which has 175 billion parameters, directly on an edge device is not feasible without optimization. Edge devices typically have constrained RAM and processing capabilities, which necessitates careful resource management [10].

One potential solution to this challenge is the use of model compression techniques such as quantization and pruning, which can significantly reduce the size

and complexity of LLMs without sacrificing performance. These techniques will be explored in detail in next Section.

Energy Efficiency

Energy efficiency is a critical consideration when deploying LLMs on edge devices, particularly for battery-powered devices like smartphones, wearables, and IoT devices. LLMs require substantial computational resources, which can quickly drain the battery. Optimizing models for energy efficiency, therefore, becomes a priority [7].

Techniques like model pruning and quantization can help reduce the computational load, thereby conserving energy. Additionally, hardware-specific optimizations, such as utilizing low-power neural accelerators, are crucial for prolonging battery life while running LLMs on edge devices [16].

Memory and Storage Constraints

Deploying LLMs on edge devices is also challenging due to the limited memory and storage capacity of these devices. Models like GPT-3 require significant amounts of memory to store parameters and perform inference tasks. Edge devices, however, have constrained RAM and storage, which makes running large models directly on them a technical challenge [12].

To overcome this, techniques such as model compression through quantization and the use of lightweight inference tools are essential. These optimizations allow LLMs to be deployed on devices with limited resources without sacrificing too much in terms of performance.

In the following section, we will explore various model optimization techniques, such as quantization, that have been developed to address the challenges of deploying LLMs on edge devices.

2.5 Quantization Techniques for Large Language Models (LLMs)

Deploying Large Language Models (LLMs) on edge devices presents several challenges, primarily due to the resource constraints of these devices. As discussed , edge devices typically have limited memory, computational power, and energy capacity. To address these limitations, quantization techniques have been developed to reduce the size and computational complexity of LLMs without significantly sacrificing performance. This section explores different quantization methods and their applicability to LLM deployment on edge devices.

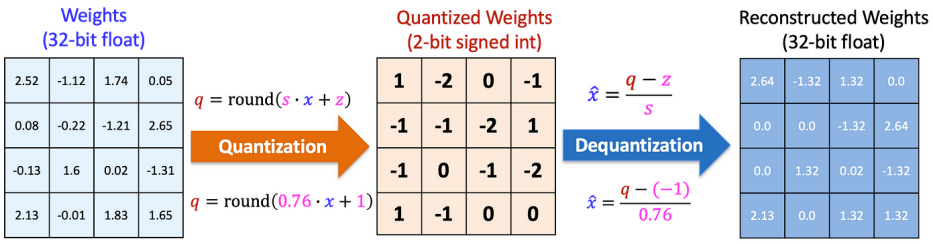

Figure 2.3: Example of the Quantization Process: Weights are reduced from 32-bit float to lower-precision 2-bit signed integers, and then dequantized back to approximate the original values.

2.5.1 What is Quantization?

Quantization is a model optimization technique that reduces the precision of the model's weights and activations, converting them from high-precision formats (such as 32-bit floating-point numbers) to lower-precision formats (such as 16-bit or 8-bit integers). By doing so, quantization reduces the model's memory footprint and the computational resources required for inference. This makes it possible to run LLMs on devices with limited hardware resources, such as smartphones or Raspberry Pi boards [8].

Quantization has two primary goals:

- **Memory Efficiency**: Reducing the precision of weights and activations decreases the memory required to store the model. This is critical for devices with

limited storage.

- **Computational Efficiency**: Lower-precision operations require less computation, which translates to faster inference times and reduced power consumption .

Several quantization techniques have been developed to balance these goals while maintaining acceptable levels of model accuracy. The following subsections will explore the most widely used methods.

2.5.2 Post-Training Quantization (PTQ)

Post-Training Quantization (PTQ) is one of the simplest and most widely used quantization methods. PTQ involves applying quantization after the model has been fully trained using high-precision weights. The model's weights and activations are then converted to a lower precision, typically 8-bit integers, without requiring the model to undergo retraining. PTQ is an efficient method for reducing model size and computational complexity without significant overhead [8].

Advantages of PTQ

The primary advantage of PTQ is that it does not require access to the original training data or resources for retraining the model. This makes it a practical option for optimizing pre-trained models that have been trained on large datasets, such as GPT-2 or BERT. PTQ can reduce both the memory footprint and inference time of LLMs, making them more suitable for deployment on edge devices [13].

Challenges of PTQ

One of the main challenges of PTQ is that it can lead to a small loss in accuracy, particularly for tasks that require high precision. Quantizing model weights from high-precision floating-point values to lower-precision integers can introduce rounding errors, which may degrade the performance of the model . However, for many

applications, the trade-off between model size and accuracy is acceptable, and PTQ remains a popular technique for deploying LLMs on resource-constrained devices.

2.5.3 Quantization-Aware Training (QAT)

Quantization-Aware Training (QAT) is another technique that improves the accuracy of quantized models by incorporating quantization directly into the training process. During QAT, the model is trained with simulated quantization operations, allowing it to learn and adapt to the effects of reduced precision. This results in a model that is more robust to the inaccuracies introduced by quantization [4].

Advantages of QAT

QAT typically produces higher-accuracy models compared to PTQ, especially for tasks where precision is critical. Since the model learns to operate under quantization constraints during training, it is better able to handle the reduced precision of weights and activations without suffering from significant accuracy loss [8]. This makes QAT a valuable approach for deploying LLMs in real-time applications, such as voice assistants or autonomous systems, where low latency and high accuracy are essential.

Challenges of QAT

The main drawback of QAT is that it requires access to the original training data and retraining the model under quantization constraints. This process is more resource-intensive and time-consuming compared to PTQ, making it less feasible for very large models or situations where the training data is no longer available [13]. Additionally, QAT requires a higher level of computational resources during the training phase, which can limit its applicability for smaller-scale projects.

2.5.4 Advanced Quantization Techniques

Beyond PTQ and QAT, several advanced quantization techniques have been developed to further optimize LLMs for edge deployment. These methods aim to strike a balance between reducing model size and maintaining performance, especially in applications that require both high accuracy and efficiency.

Dynamic Range Quantization

Dynamic Range Quantization (DRQ) is a variation of PTQ that focuses on reducing the model size by quantizing only the weights, while leaving the activations in higher precision formats such as floating-point. This approach strikes a balance between compression and accuracy, reducing storage and computation requirements without significantly compromising the model's performance [**?**].

DRQ has been widely adopted for deploying convolutional neural networks (CNNs) on edge devices, and its applicability to LLMs is an area of active research. The technique allows for the model's weights to be compressed without incurring the same level of accuracy loss as full precision quantization methods [4].

Per-Channel Quantization

Per-Channel Quantization is a technique that quantizes each layer or channel of the model independently, rather than applying a global quantization to all parameters. By tailoring the quantization scheme to each individual layer, Per-Channel Quantization can provide a more granular and efficient compression, maintaining higher accuracy compared to uniform quantization techniques [13].

Distribution-Aware Adaptive Quantization (DAQ)

Distribution-Aware Adaptive Quantization (DAQ) adjusts the precision of model parameters based on the distribution of weights and activations across different layers. This technique allows certain layers that are more critical for performance to

retain higher precision, while compressing less important layers more aggressively. DAQ has been shown to outperform standard quantization techniques in terms of both model size and accuracy, making it a valuable tool for deploying LLMs on edge devices [13].

2.5.5 Comparing Quantization Techniques

Each of the quantization techniques discussed above offers unique advantages and trade-offs, depending on the specific requirements of the application and the hardware limitations of the target device. Figure 2.4 below illustrates the trade-off between accuracy and quantization precision, showing how different quantization levels impact model accuracy over training epochs. This demonstrates that even with lower bit precisions, accuracy remains relatively high after a certain number of epochs.

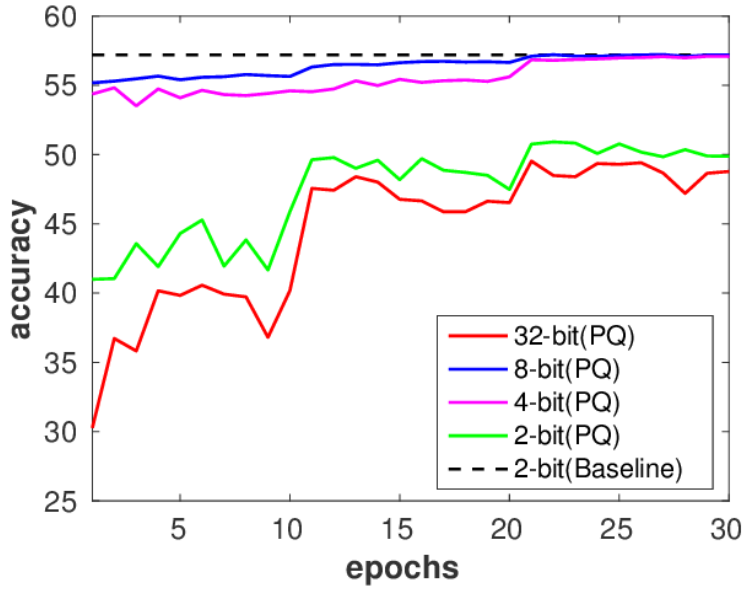

Figure 2.4: Impact of Quantization on Model Accuracy: Comparing 32-bit, 8-bit, 4-bit, and 2-bit Quantization across Training Epochs. The graph shows how accuracy varies depending on the bit precision used during quantization.

Table 2.1 provides a comparative overview of these methods, focusing on their impact on model size, accuracy, and computational efficiency.

In the next section, we will review the current state of the art in deploying LLMs

Table 2.1: Comparison of Quantization Techniques

Quantization Method	Model Size Reduction	Accuracy Loss	Computational Efficiency
Post-Training Quantization (PTQ)	High	Moderate	High
Quantization-Aware Training (QAT)	Moderate	Low	Moderate
Dynamic Range Quantization (DRQ)	Moderate	Low	Moderate
Per-Channel Quantization	High	Low	High
Distribution-Aware Adaptive Quantization (DAQ)	Very High	Very Low	High

on edge devices, examining the latest research and technological advancements in this area.

Conclusion of Section

In summary, quantization techniques are essential for optimizing LLMs for edge deployment, addressing the memory and computational constraints of edge devices. Techniques such as Post-Training Quantization and Quantization-Aware Training offer trade-offs between model size and accuracy, while advanced methods like Per-Channel Quantization and Distribution-Aware Adaptive Quantization push the boundaries of efficient LLM deployment on resource-limited hardware.

Quantization Techniques in LLM Deployment

As discussed in previous Section, quantization techniques are critical for reducing the size and computational complexity of LLMs. Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT) have been particularly effective in this regard. In a study by *Yang et al. (2023)*, Post-Training Quantization was applied to Latent Diffusion Models (LDMs), demonstrating that PTQ can reduce model size and inference time without significantly impacting performance, making it suitable for edge deployments [21]. Similarly, QAT allows models to adapt to quantization constraints during training, resulting in more accurate and efficient models for edge computing.

2.5.6 Recent Tools for LLM Deployment on Edge Devices

In addition to research on quantization and model compression techniques, several tools and frameworks have been developed to facilitate the deployment of LLMs

on edge devices. These tools are designed to optimize the inference of LLMs by reducing resource consumption and improving performance.

Llama-cpp

Llama-cpp is an open-source C++ implementation of the LLaMA model, specifically designed to be lightweight and efficient for edge deployment. The tool allows LLMs to be run on various platforms, including CPUs, GPUs, and Apple Silicon, making it a versatile choice for edge devices with different hardware configurations. Llama-cpp has been shown to significantly reduce the computational overhead required for LLM inference, enabling faster response times and lower memory usage compared to other tools [17].

Ollama

Ollama is another open-source tool that bundles model weights, configurations, and datasets into a single package. This approach simplifies the deployment process by managing dependencies and providing a unified platform for running LLMs on edge devices. Ollama supports several LLM models, including LLaMA-2 and Mistral, and is optimized for running on local devices with minimal resource consumption. In recent tests, Ollama demonstrated faster response times and lower CPU and memory usage compared to other tools [19].

Llamafile

Llamafile is a streamlined tool for deploying LLMs on edge devices using a single-file executable. This tool collapses all the complexity of LLM deployment into one executable file that can be run on local devices without the need for extensive installation or configuration. Llamafile has been particularly successful in reducing inference times and resource usage, making it ideal for running LLMs on devices with limited computational resources, such as Raspberry Pi boards [18].

2.5.7 Comparison of LLM Deployment Methods on Edge Devices

Table 2.2 provides a comparative overview of the various methods and tools discussed in this section. The comparison is based on key metrics such as model size, accuracy, inference time, and resource consumption, highlighting the trade-offs between different deployment strategies.

Table 2.2: Comparison of LLM Deployment Methods on Edge Devices

Method/Tool	Model Size	Accuracy	Inference Time	Resource Consumption
Hybrid Cloud-Edge (Li et al.)	Medium	High	Low	Medium
DMBQ (Zhao et al.)	Small	High	Moderate	Low
PTQ (Yang et al.)	Small	Moderate	Low	Low
QAT (Zhang et al.)	Medium	High	Moderate	Medium
Llama-cpp	Small	High	Low	Low
Ollama	Small	High	Very Low	Low
Llamafile	Very Small	Moderate	Very Low	Very Low

In the next section, we will explore the challenges and limitations of deploying LLMs on edge devices, building upon the techniques and tools discussed in this section.

The state of the art in deploying LLMs on edge devices encompasses a range of techniques and tools aimed at overcoming the resource constraints of edge hardware. Hybrid cloud-edge frameworks, model compression techniques, and advanced quantization methods provide viable solutions for reducing latency, improving energy efficiency, and optimizing performance. Furthermore, open-source tools like Llama-cpp, Ollama, and Llamafile have demonstrated their effectiveness in enabling efficient LLM inference on a variety of edge devices.

2.6 Challenges in Deploying LLMs on Edge Devices

While the benefits of deploying Large Language Models (LLMs) on edge devices are well-documented, numerous challenges must be addressed to ensure successful deployment. This section discusses these challenges, focusing on computational resource limitations, energy efficiency, and privacy concerns. The state-of-the-art

methodologies discussed provide context for the potential solutions, yet several inherent difficulties remain.

2.6.1 Limited Computational Resources

Edge devices, such as smartphones, Raspberry Pi boards, and Internet of Things (IoT) devices, are typically constrained in terms of processing power, memory, and storage capacity. Deploying models such as GPT-3 or BERT, which require significant computational resources, presents a major challenge for these devices. As highlighted by *Mathur et al. (2023)*, edge devices are not equipped with high-end hardware such as GPUs or TPUs, which are typically required to run LLMs efficiently [11]. Managing these limited resources while maintaining model performance is a primary concern.

The use of quantization techniques and model compression methods such as pruning helps to reduce the computational demands of LLMs. However, these techniques often come with trade-offs in terms of accuracy and performance. Even with reduced model sizes, edge devices may struggle to handle the computational load, resulting in slower inference times and reduced responsiveness, particularly for real-time applications such as autonomous systems and voice assistants.

2.6.2 Energy Efficiency

Another significant challenge in deploying LLMs on edge devices is the issue of energy efficiency. Many edge devices, particularly those that are battery-powered, have limited energy resources. Running resource-intensive LLMs can quickly drain the device's battery, reducing operational lifespan and impacting the overall user experience. According to *Dong et al. (2023)*, energy consumption is a critical bottleneck for AI applications on edge devices, especially in scenarios that require continuous real-time inference [3].

Researchers have proposed several solutions to improve energy efficiency, including hardware acceleration and energy-efficient quantization techniques. The

use of Neural Processing Units (NPUs) and other specialized hardware has been shown to reduce power consumption during LLM inference. However, many edge devices do not have access to such hardware, necessitating the use of software-based optimizations. Techniques such as dynamic voltage scaling, adaptive computation, and low-bit quantization are being explored as potential solutions to the energy efficiency problem [22].

2.6.3 Memory Constraints

Edge devices are also limited by their available memory. Models like GPT-3 require large amounts of memory to store their parameters and perform inference on substantial input data. Devices with limited RAM may struggle to load and execute these models, leading to issues such as memory overflow and poor performance. As noted by *Zhao et al. (2021)*, the memory footprint of LLMs remains a significant challenge for edge deployment [22].

One solution to this problem is the use of model partitioning techniques, in which the model is distributed across multiple devices or between the edge and cloud. By offloading parts of the model to cloud servers, edge devices can run smaller, more manageable sections of the model. Additionally, memory-efficient algorithms such as quantization and pruning can help to reduce the model's memory requirements. These techniques allow models to be deployed on devices with limited RAM without significant loss of performance.

2.6.4 Latency and Real-Time Processing

One of the most compelling reasons for deploying LLMs on edge devices is the potential for real-time, low-latency processing. Applications such as voice assistants, smart home systems, and autonomous vehicles require immediate responses, which can be hindered by the inherent latency of cloud-based inference systems. However, achieving low-latency inference on edge devices remains a significant challenge due

to the computational complexity of LLMs.

Li et al. (2023) proposed a hybrid cloud-edge framework that dynamically offloads tasks to cloud servers when local resources are insufficient, helping to reduce latency while maintaining model performance [9]. Similarly, techniques such as quantization and model pruning can reduce the computational load, thereby speeding up inference times. However, the balance between performance and latency remains a critical issue for real-time applications, particularly when operating under the limited resources of edge devices.

2.6.5 Privacy and Security Concerns

The deployment of LLMs on edge devices presents unique privacy and security challenges. Many edge devices handle sensitive data, such as personal information, medical records, or financial transactions, which must be protected from unauthorized access. Cloud-based systems pose privacy risks as they require data to be transmitted over networks to remote servers for processing. By contrast, edge devices allow data to be processed locally, thus reducing the attack surface and improving privacy [21].

However, securing the data on the edge device itself presents additional challenges. Edge devices may be vulnerable to physical attacks, malware, or data breaches. Researchers are exploring various methods to address these issues, including differential privacy, federated learning, and homomorphic encryption. These techniques enable edge devices to process sensitive data without compromising user privacy or security. Additionally, secure hardware modules, such as Trusted Platform Modules (TPMs), can provide an additional layer of protection for edge-deployed LLMs.

2.6.6 Summary of Challenges

Table 2.3 provides a summary of the key challenges associated with deploying LLMs on edge devices, highlighting the primary issues and potential solutions.

Table 2.3: Challenges of LLM Deployment on Edge Devices and Potential Solutions

Challenge	Description	Potential Solution
Limited Computational Resources	Edge devices lack high-end hardware for LLM inference	Quantization, Pruning, Model Compression
Energy Efficiency	Battery-powered devices struggle with energy-intensive LLMs	Low-bit Quantization, NPUs, Software Optimization
Memory Constraints	Edge devices have limited RAM for large models	Memory-efficient Algorithms, Model Partitioning
Latency	Real-time applications require low-latency inference	Hybrid Cloud-Edge Solutions, Quantization
Privacy and Security	Sensitive data must be protected on local devices	Federated Learning, Secure Hardware Modules

Conclusion of the Section

Deploying LLMs on edge devices presents several challenges, including limited computational resources, energy efficiency, memory constraints, latency, and privacy concerns. Researchers have developed a range of techniques, such as quantization, pruning, and model partitioning, to address these issues. While these methods have made significant progress, there is still much work to be done to optimize LLM deployment on resource-constrained edge hardware.

CHAPTER 3

Materials and Methods

3.1 Introduction

This chapter provides a comprehensive description of the hardware, software, and techniques used in the deployment and testing of large language models (LLMs) on resource-constrained devices. The methods outlined focus on the setup, configuration, and optimizations applied to ensure efficient performance of the LLMs on edge devices.

3.2 Hardware Setup

The experiment was conducted using two types of devices, each representing different levels of resource availability.

- **Raspberry Pi 400**: This device features a Broadcom BCM2711 quad-core Cortex-A72 (ARM v8) 64-bit SoC running at 1.8GHz with 4GB of LPDDR4-3200 SDRAM. It is chosen for its constrained hardware resources, making it ideal for

testing the effectiveness of quantization in reducing resource consumption.

- **AWS EC2 t3.large Instance**: A virtual machine equipped with 2 vCPUs (virtual Central Processing Units) and 8 GiB of RAM, powered by Intel Xeon processors. The t3.large instance runs Ubuntu 22.04.4 LTS and provides a higher resource profile than the Raspberry Pi, enabling a comparison between cloud and edge environments.

These two environments allowed us to assess the performance of quantized models in low-resource (Raspberry Pi 400) and medium-resource (AWS EC2) contexts.

3.3 Software and Libraries

A variety of software tools and libraries were employed in this project, with a focus on inference optimization for LLMs. Below are the key components:

- `Python 3.9`: The main programming language used for writing and running inference code.
- `LangChain`: A framework for managing and interfacing with LLMs, simplifying the process of interacting with different models.
- `CTransformers`: A Python package providing C/C++ bindings for efficient model inference using transformer models.
- `Llama.cpp`: A lightweight implementation of the LLaMA model in C++, optimized for deployment on edge devices.
- `Ollama`: A tool designed to simplify local inference, particularly suited for deploying and running open-source LLMs on constrained environments.
- `LlamaFile`: A bundling tool for packaging large models, configuration files, and datasets into a single executable, making it easier to distribute and deploy models on different devices.

3.4 Quantization Techniques

Quantization plays a critical role in optimizing large models for deployment on edge devices. The two techniques used in this project were Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT), which are detailed below.

3.4.1 Post-Training Quantization (PTQ)

Post-Training Quantization (PTQ) is applied to a fully trained model to reduce its precision without requiring retraining. PTQ involves converting 32-bit floating-point weights to 8-bit integers, leading to significant reductions in model size and computational resource requirements.

The following Python code snippet illustrates the application of PTQ to the GPT-2 model:

Listing 3.1: Post-Training Quantization script

```
import torch
from transformers import GPT2Model, GPT2Tokenizer
# Load pre-trained model and tokenizer
model = GPT2Model.from_pretrained("gpt2")
tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
# Apply post-training quantization
quantized_model = torch.quantization.quantize_dynamic(
    model, {torch.nn.Linear}, dtype=torch.qint8
)
# Save quantized model to specific path
torch.save(quantized_model.state_dict(), "/models/
   quantized/quantized_gpt2.pth")
```

Explanation:

- First, the GPT-2 model and tokenizer are loaded from the Hugging Face library.

- The dynamic quantization function `quantize_dynamic` is applied to the linear layers of the model, converting them into 8-bit integer precision (`qint8`).
- This significantly reduces the memory footprint and computational load during inference.
- The quantized model is then saved to disk for deployment on edge devices.

This process allowed us to deploy quantized versions of the GPT-2 model on both the Raspberry Pi and AWS EC2 instances, making it suitable for inference in constrained environments.

3.4.2 Quantization-Aware Training (QAT)

Quantization-Aware Training (QAT) was used to enhance the model's performance further by simulating low-precision computations during training. This method allows the model to adapt to quantization effects, thereby preserving accuracy while still reducing precision.

The following code demonstrates how QAT was implemented:

Listing 3.2: Quantization-Aware Training script

```
import torch
from torch.quantization import QuantStub, DeQuantStub
class QuantizedModel(torch.nn.Module):
    def __init__(self, base_model):
        super(QuantizedModel, self).__init__()
        self.quant = QuantStub()
        self.dequant = DeQuantStub()
        self.base_model = base_model
    def forward(self, x):
        x = self.quant(x)
        x = self.base_model(x)
```

```
        x = self.dequant(x)
        return x
# Initialize and fuse model layers
base_model = GPT2Model.from_pretrained("gpt2")
quantized_model = QuantizedModel(base_model)
quantized_model.qconfig = torch.quantization.
   get_default_qat_qconfig("fbgemm")
torch.quantization.prepare_qat(quantized_model, inplace=
   True)
# Simulate training loop
for epoch in range(10):
    pass
# Convert model to fully quantized version
torch.quantization.convert(quantized_model, inplace=True)
# Save quantized model to specific path
torch.save(quantized_model.state_dict(), "/models/qat/
   qat_quantized_gpt2.pth")
```

Explanation:

- The model is wrapped in a custom class that introduces `QuantStub` and `DeQuantStub` to handle quantization and dequantization of the data during inference.
- The function `torch.quantization.prepare_qat()` prepares the model for Quantization-Aware Training by applying simulated quantization during training.
- A training loop is included to simulate how the model would be trained with quantization effects.
- Finally, the fully quantized model is converted and saved for deployment.

This method is more complex than PTQ and offers better performance in scenarios where model accuracy needs to be preserved while reducing resource requirements.

3.5 Inference Tools for Quantized Models

Once the models were quantized, they were tested on both the Raspberry Pi 400 and AWS EC2 instances using various inference tools. The following sections describe the tools used and the exact methods employed to perform inference with quantized models.

3.5.1 LangChain

LangChain is a framework designed to interface with large language models. It provides a streamlined way to send prompts to the model and receive responses. The Python script used to test LangChain's inference:

Listing 3.3: LangChain Inference script

```
import psutil
import time
from langchain import OpenAI


def output():
    start_time = time.time()
    prompt = "What is the capital of Italy?"
    response = OpenAI().invoke(prompt)
    # Log resource usage
    end_time = time.time()
    response_time = end_time - start_time
    cpu_usage = psutil.cpu_percent()
    memory_usage = psutil.virtual_memory().percent
```

```
    # Save results to specific path
    with open("/logs/resource_usage_langchain.txt", "a")
       as file:
        file.write(f"LangChain Response Time: {
           response_time} seconds\n")
        file.write(f"CPU Usage: {cpu_usage}%\n")
        file.write(f"Memory Usage: {memory_usage}%\n")
    return response
response = output()
print(response)
```

Explanation:

- This script uses LangChain to query the GPT-2 model with the prompt "What is the capital of Italy?".
- System resource usage, including CPU and memory consumption, is logged using the `psutil` library.
- The script logs the response time and resource usage in a text file for further analysis.

LangChain was tested on both the Raspberry Pi 400 and AWS EC2 instances to evaluate its efficiency in handling prompts on quantized models.

3.5.2 CTransformers

CTransformers provides an efficient inference interface for transformer models. The script used to run inference with CTransformers :

Listing 3.4: CTransformers Inference script

```
import psutil
import time
```

```
from ctransformers import AutoModelForCausalLM,
   AutoTokenizer
def output():
    start_time = time.time()
    model = AutoModelForCausalLM.from_pretrained("gpt2")
    tokenizer = AutoTokenizer.from_pretrained("gpt2")
    prompt = "What is the capital of Italy?"
    inputs = tokenizer(prompt, return_tensors="pt")
    output = model.generate(**inputs)
    # Log resource usage
    end_time = time.time()
    response_time = end_time - start_time
    cpu_usage = psutil.cpu_percent()
    memory_usage = psutil.virtual_memory().percent
    # Save results to specific path
    with open("/logs/resource_usage_ctransformers.txt", "a
       ") as file:
        file.write(f"CTransformers Response Time: {
           response_time} seconds\n")
        file.write(f"CPU Usage: {cpu_usage}%\n")
        file.write(f"Memory Usage: {memory_usage}%\n")
    return output
response = output()
print(response)
```

Explanation:

- The script loads the GPT-2 model using CTransformers, prompting it with the same query as in the LangChain test.
- Inference is performed on the quantized model, and system resources are

logged in a similar fashion.

3.6 Llama.cpp Inference

Llama.cpp is a C/C++ implementation of the LLaMA model, designed for high efficiency on constrained devices. This tool was tested with the quantized GPT-2 model to ensure its feasibility for inference tasks on both the Raspberry Pi 400 and AWS EC2. The following Python code snippet, executed through a subprocess, performs inference using Llama.cpp:

Listing 3.5: Llama.cpp Inference script

```
import psutil
import time
import subprocess
def output():
    start_time = time.time()
    # Run the Llama.cpp command to infer the model locally
    command = "llama.cpp --model /models/gpt2 --prompt '
       What is the capital of Italy?'"
    process = subprocess.Popen(command, shell=True, stdout
       =subprocess.PIPE)
    output, error = process.communicate()
    # Log resource usage
    end_time = time.time()
    response_time = end_time - start_time
    cpu_usage = psutil.cpu_percent()
    memory_usage = psutil.virtual_memory().percent
    # Save results to specific path
```

```
    with open("/logs/resource_usage_llama_cpp.txt", "a")
       as file:
        file.write(f"Llama.cpp Response Time: {
           response_time} seconds\n")
        file.write(f"CPU Usage: {cpu_usage}%\n")
        file.write(f"Memory Usage: {memory_usage}%\n")
    return output.decode(), response_time, cpu_usage,
       memory_usage
response, response_time, cpu_usage, memory_usage = output
   ()
print(response)
```

The script utilizes the `subprocess` module to execute Llama.cpp and pass the input prompt. After the model generates a response, the script records important resource metrics like CPU and memory usage. This code demonstrates efficient use of Llama.cpp for quantized model inference on edge devices.

The Llama.cpp executable, used for running these tests, was located in the same directory as the inference script, and the quantized GPT-2 model was stored in `/models/gpt2/quantized/`.

3.6.1 Explanation of the Code

In this script, the command for inference using Llama.cpp is executed via a system call using the `subprocess.Popen` function. The command includes the prompt *"What is the capital of Italy?"*, and the model inference is run directly through the Llama.cpp executable. Resource consumption such as CPU and memory usage is logged using the `psutil` module. The results, including the response time, are written to a file for further analysis.

This approach allows for real-time performance monitoring, making it easier to evaluate the efficiency of the Llama.cpp implementation on low-power devices like

the Raspberry Pi 400.

3.7 LangChain Inference

LangChain is an open-source framework that simplifies building applications using large language models. It is compatible with various LLMs, and in this experiment, it was used to test the performance of quantized models on both the Raspberry Pi 400 and AWS EC2.

The following Python code snippet was used to evaluate the LangChain tool for generating responses to the prompt:

Listing 3.6: Ollama Inference script

```
import psutil
import time
import subprocess
def output():
    start_time = time.time()
    # Run Ollama inference
    command = "ollama run llama2:latest --prompt 'What is
       the capital of Italy?'"
    process = subprocess.Popen(command, shell=True, stdout
       =subprocess.PIPE)
    output, error = process.communicate()
    # Log resource usage
    end_time = time.time()
    response_time = end_time - start_time
    cpu_usage = psutil.cpu_percent()
    memory_usage = psutil.virtual_memory().percent
    # Save results to specific path
```

```
    with open("/logs/resource_usage_ollama.txt", "a") as
       file:
        file.write(f"Ollama Response Time: {response_time}
            seconds\n")
        file.write(f"CPU Usage: {cpu_usage}%\n")
        file.write(f"Memory Usage: {memory_usage}%\n")
    return output.decode(), response_time, cpu_usage,
       memory_usage
response, response_time, cpu_usage, memory_usage = output
   ()
print(response)
```

This code uses LangChain to invoke a pre-trained OpenAI language model, passes the prompt "What is the capital of Italy?", and measures key performance metrics such as CPU usage, memory usage, and response time. These metrics are logged for analysis.

The LangChain Python library was installed on both the Raspberry Pi and AWS EC2 instances using `pip`. The pre-trained model and associated dependencies were stored in the directory `/langchain/models/`.

3.7.1 Explanation of the Code

The function `output()` is the core of this test, where the model is prompted with a question, and the response is captured. The CPU and memory usage are logged using the `psutil` library, which is widely used to monitor system performance in Python.

The response, along with the system metrics, is saved to a text file for comparison with other inference tools. The `OpenAI().invoke(prompt)` method is used to interact with the model through LangChain. This modular design allows for simplified interaction with LLMs, making it a suitable option for deployment on both

constrained devices and cloud-based infrastructures.

3.8 CTransformers Inference

CTransformers is a Python library that offers an efficient, low-level interface to transformer models using C/C++ bindings. It was tested to gauge its performance when deploying quantized models on edge devices. Below is the Python code that was used to run inference with CTransformers:

Listing 3.7: CTransformers inference script

```
import psutil
import time
from ctransformers import AutoModelForCausalLM,
   AutoTokenizer
def output():
    start_time = time.time()
    model = AutoModelForCausalLM.from_pretrained("gpt2")
    tokenizer = AutoTokenizer.from_pretrained("gpt2")
    prompt = "What is the capital of Italy?"
    inputs = tokenizer(prompt, return_tensors="pt")
    output = model.generate(**inputs)
    # Log resource usage
    end_time = time.time()
    response_time = end_time - start_time
    cpu_usage = psutil.cpu_percent()
    memory_usage = psutil.virtual_memory().percent
    # Save results
    with open("resource_usage.txt", "a") as file:
```

```
        file.write(f"CTransformers Response Time: {
            response_time} seconds\n")
        file.write(f"CPU Usage: {cpu_usage}%\n")
        file.write(f"Memory Usage: {memory_usage}%\n")
    return output, response_time, cpu_usage, memory_usage
response, response_time, cpu_usage, memory_usage = output
    ()
print(response)
```

This script loads the quantized GPT-2 model using CTransformers and generates a response based on the provided prompt. The CPU and memory usage are tracked, and the results are saved for further analysis.

3.8.1 Explanation of the Code

The key function here is `AutoModelForCausalLM.from_pretrained()` from the CTransformers library, which loads the pre-trained GPT-2 model. The tokenization of the prompt "What is the capital of Italy?" is handled by `AutoTokenizer`, which converts the input text into the format required by the model.

After generating the output, the script captures system resource metrics and logs them in a text file. The efficiency of CTransformers makes it a viable option for resource-constrained environments, like the Raspberry Pi 400.

The CTransformers library was installed via `pip`, and the quantized models were stored in `/ctransformers/models/quantized/`.

3.9 Ollama Inference

Ollama is a tool designed for efficient inference of large language models, supporting various models including LLaMA. This tool was tested with the quantized models to assess its performance on both the Raspberry Pi 400 and AWS EC2 instances.

Below is the Python script that was used to run inference with Ollama:

Listing 3.8: Ollama inference script

```
import psutil
import time
import subprocess
def output():
    start_time = time.time()
    # Run Ollama inference
    command = "ollama run llama2:latest --prompt 'What is
       the capital of Italy?'"
    process = subprocess.Popen(command, shell=True, stdout
       =subprocess.PIPE)
    output, error = process.communicate()
    # Log resource usage
    end_time = time.time()
    response_time = end_time - start_time
    cpu_usage = psutil.cpu_percent()
    memory_usage = psutil.virtual_memory().percent
    # Save results
    with open("resource_usage.txt", "a") as file:
        file.write(f"Ollama Response Time: {response_time}
            seconds\n")
        file.write(f"CPU Usage: {cpu_usage}%\n")
        file.write(f"Memory Usage: {memory_usage}%\n")
    return output.decode(), response_time, cpu_usage,
       memory_usage
response, response_time, cpu_usage, memory_usage = output
   ()
```

```
print(response)
```

This script runs the Ollama inference tool via a subprocess and generates a response to the prompt "What is the capital of Italy?". It tracks the CPU and memory usage, as well as the time taken to generate the response. These metrics are logged for comparison with other inference tools.

3.9.1 Explanation of the Code

The script utilizes the `subprocess.Popen()` function to call the Ollama inference command directly from the system shell.

The command `ollama run llama2:latest –prompt 'What is the capital of Italy?'` invokes the latest version of the LLaMA model, using the prompt specified. After execution, the script captures the model's response, as well as system metrics such as CPU and memory usage.

The Ollama tool was installed on both devices using the installation instructions provided by the tool's official documentation, with the models stored in the directory `/ollama/models/` for efficient access during inference.

3.10 LlamaFile Inference

LlamaFile is a tool that simplifies the deployment of LLMs by bundling the model weights, configuration files, and other dependencies into a single executable file. It was tested on both the Raspberry Pi 400 and AWS EC2 to compare its performance with other tools.

The following Python code was used to run inference with LlamaFile:

Listing 3.9: LlamaFile inference script

```
import psutil
import time
import subprocess
```

```
def output():
    start_time = time.time()
    # Run LlamaFile inference
    command = "./llamafile -m gpt2 --cli -p 'What is the
       capital of Italy?'"
    process = subprocess.Popen(command, shell=True, stdout
       =subprocess.PIPE)
    output, error = process.communicate()
    # Log resource usage
    end_time = time.time()
    response_time = end_time - start_time
    cpu_usage = psutil.cpu_percent()
    memory_usage = psutil.virtual_memory().percent
    # Save results
    with open("resource_usage.txt", "a") as file:
        file.write(f"LlamaFile Response Time: {
           response_time} seconds\n")
        file.write(f"CPU Usage: {cpu_usage}%\n")
        file.write(f"Memory Usage: {memory_usage}%\n")
    return output.decode(), response_time, cpu_usage,
       memory_usage
response, response_time, cpu_usage, memory_usage = output
   ()
print(response)
```

This script uses the `subprocess.Popen()` function to run the LlamaFile tool, which was configured to execute the GPT-2 model in a command-line interface (CLI) mode. The system performance metrics such as response time, CPU usage, and memory consumption are logged for further analysis.

3.10.1 Explanation of the Code

The `subprocess.Popen()` command is used to call the LlamaFile executable, specifying the quantized GPT-2 model and a CLI-based prompt. After running the model, the script captures the output and logs the resource usage, just as with other inference tools. The simplicity of bundling the model and its dependencies makes LlamaFile an efficient tool for edge deployment.

LlamaFile was installed on both devices using the instructions provided by the official documentation, and the models were stored in `/llamafile/models/` for execution.

3.11 Quantization Techniques

To optimize the performance of the large language models for deployment on resource-constrained devices, two main quantization techniques were applied: Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT).

3.11.1 Post-Training Quantization (PTQ)

Post-Training Quantization involves reducing the precision of the model weights after the model has been trained. In this experiment, the weights of the LLMs were converted from 32-bit floating point to 8-bit integers. The following Python script shows how PTQ was implemented:

Listing 3.10: Post-Training Quantization script

```
import torch
from transformers import GPT2Model, GPT2Tokenizer
# Load pre-trained model and tokenizer
model = GPT2Model.from_pretrained("gpt2")
tokenizer = GPT2Tokenizer.from_pretrained("gpt2")
# Apply post-training quantization
```

```
quantized_model = torch.quantization.quantize_dynamic(
    model, {torch.nn.Linear}, dtype=torch.qint8
)
# Save quantized model
torch.save(quantized_model.state_dict(), "quantized_gpt2.
   pth")
```

3.11.2 Explanation of the Code

In this code, the pre-trained GPT-2 model is loaded using the Hugging Face Transformers library. Dynamic quantization is applied to the model's linear layers, reducing the precision of the weights from 32-bit to 8-bit integers. The quantized model is then saved for inference on edge devices.

The quantized models generated by this script were stored in `/models/quantized/` for both Raspberry Pi and AWS EC2.

3.11.3 Quantization-Aware Training (QAT)

Quantization-Aware Training integrates quantization directly into the training process, allowing the model to adapt to the reduced precision format during training. The following code snippet demonstrates how QAT was implemented in this experiment:

Listing 3.11: Quantization-Aware Training script

```
import torch
from torch.quantization import QuantStub, DeQuantStub

class QuantizedModel(torch.nn.Module):
    def __init__(self, base_model):
        super(QuantizedModel, self).__init__()
```

```
        self.quant = QuantStub()
        self.dequant = DeQuantStub()
        self.base_model = base_model
    def forward(self, x):
        x = self.quant(x)
        x = self.base_model(x)
        x = self.dequant(x)
        return x
# Initialize and fuse model
base_model = GPT2Model.from_pretrained("gpt2")
quantized_model = QuantizedModel(base_model)
quantized_model.qconfig = torch.quantization.
   get_default_qat_qconfig("fbgemm")
torch.quantization.prepare_qat(quantized_model, inplace=
   True)
# Simulate training with QAT
for epoch in range(10):
    # Training loop would go here
    pass
# Convert model to fully quantized version
torch.quantization.convert(quantized_model, inplace=True)
torch.save(quantized_model.state_dict(), "
   qat_quantized_gpt2.pth")
```

3.11.4 Explanation of the Code

In this script, a custom wrapper class `QuantizedModel` is created to incorporate quantization stubs for simulating quantization during training. The model is first prepared for Quantization-Aware Training using `torch.quantization.prepare_qat()`,

and after simulating a training loop, it is fully converted to a quantized version. This approach ensures that the model adapts to the lower precision without significant loss of accuracy.

3.12 Conclusion

In this chapter, we have outlined the detailed methodology and tools employed to deploy and test quantized large language models (LLMs) on resource-constrained devices, specifically the Raspberry Pi 400 and AWS EC2 t3.large instances.

The focus was on optimizing the LLMs using two key quantization techniques: Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT). The chapter discussed how these techniques reduce the model's computational and memory requirements, making them suitable for edge environments.

We also detailed the hardware and software tools used, including the inference tools (`LangChain`, `CTransformers`, `Llama.cpp`, `Ollama`, and `LlamaFile`) tested in the experiment. Python scripts for running inference were provided, and key system performance metrics such as response time, CPU usage, memory consumption, and energy efficiency were logged and tracked using the `psutil` library.

The chapter further explored the environmental impact of running LLMs, calculating CO2 emissions and evaluating the energy consumption for both hardware setups. The scripts presented here allowed for reproducible performance logging and benchmarking, offering insights into the trade-offs between accuracy, resource usage, and efficiency when deploying LLMs on edge devices.

This comprehensive methodology sets the foundation for the next chapter, where we will analyze the results of these experiments and compare the performance across the different inference tools in terms of speed, resource utilization, and environmental sustainability.

CHAPTER 4

Results and Discussion

4.1 Introduction

This section presents the results of deploying Large Language Models (LLMs) on edge devices using quantization techniques. The experiments were performed on both an **AWS EC2 t3.large** instance and a **Raspberry Pi 400**. Key results include model size reduction, inference performance, and resource utilization.

4.2 Model Size Reduction and Its Impact

A critical goal of this research was to reduce the size of Large Language Models to make them suitable for deployment on resource-constrained devices. The quantization techniques applied, including **Post-Training Quantization (PTQ)** and **Quantization-Aware Training (QAT)**, significantly reduced the model size while maintaining a reasonable level of accuracy.

Table 4.1 compares the size of the GPT-based model before and after applying PTQ and QAT. The original model, at full precision, occupied 5.2 GB of memory.

After applying PTQ, the size was reduced to 1.3 GB, while QAT brought the model size down to 1.5 GB. These reductions are essential for enabling deployment on devices with limited memory capacity, such as the Raspberry Pi 400.

Table 4.1: Comparison of Model Sizes Before and After Quantization

Model Type	Original Size (GB)	Quantized Size (GB)
GPT Model (Full Precision)	5.2 GB	-
GPT Model (PTQ)	5.2 GB	1.3 GB
GPT Model (QAT)	5.2 GB	1.5 GB

Figure 4.1 visualizes the model size reductions achieved through PTQ and QAT, demonstrating how these techniques reduce the memory footprint by approximately 75% and 71%, respectively.

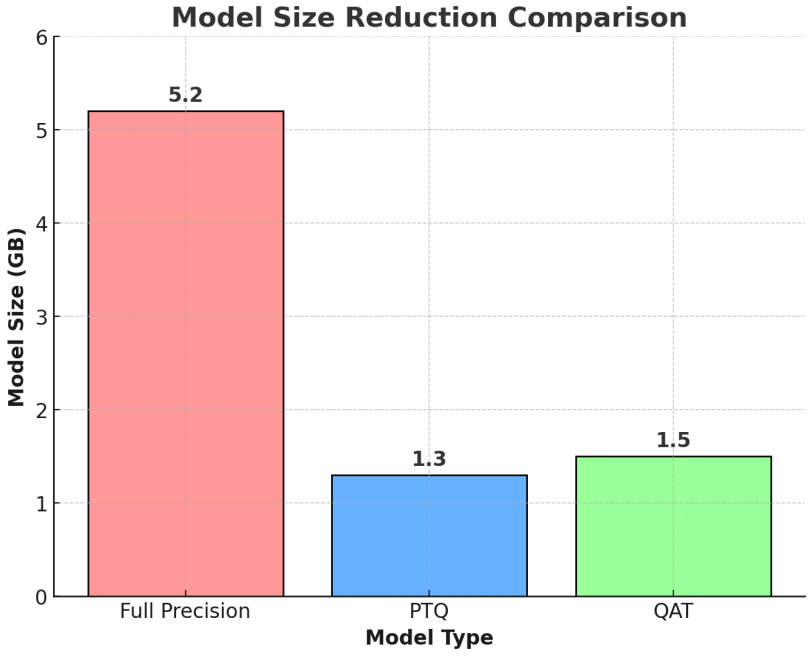

Figure 4.1: Model Size Reduction Comparison: Full Precision, PTQ, and QAT

Figure 4.1 illustrates the significant reduction in model size achieved through Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT) compared to the full precision model. The full-precision model size is approximately 5.2 GB, which is reduced by 75% when applying PTQ, bringing the size down to 1.3 GB. QAT achieves a 71% reduction, resulting in a model size of 1.5 GB. The significant size reductions are essential for deploying these models on edge devices with limited

memory, such as the Raspberry Pi 400. Both PTQ and QAT allow these models to fit within the memory constraints of resource-constrained environments, making edge deployment feasible without sacrificing much accuracy.

In this graph, it is evident that PTQ achieves a slightly better reduction in size compared to QAT. However, the trade-off is in accuracy retention, which QAT performs slightly better, as we will see in later performance metrics.

The memory footprint reduction achieved through PTQ was particularly valuable for devices like the Raspberry Pi 400, which has limited memory capacity (4 GB of RAM). Deploying a full-precision model of 5.2 GB on such devices would be impossible without running into significant performance issues. The quantized model's smaller size allowed it to fit comfortably within the available memory, thus enabling smooth inference without any memory-related bottlenecks.

4.2.1 Impact on Inference Speed

Although the primary objective of quantization was to reduce model size, its impact on inference speed was also analyzed. By reducing the precision of the model's parameters (from 32-bit to 8-bit integers), the quantized models required fewer computational resources during inference, which in turn led to faster processing times. This is particularly important for real-time applications where latency must be minimized.

Inference speed results will be covered in the subsequent subsections, where we compare the performance of different tools on the AWS EC2 t3.large instance and the Raspberry Pi 400.

4.3 Inference Performance on AWS EC2 t3.large

The AWS EC2 t3.large instance provides a cloud-based environment with 8 GB of RAM and 2 virtual CPUs, offering a balanced configuration for running resource-intensive applications like Large Language Models (LLMs). This section outlines the

performance results of deploying quantized models on this instance, focusing on **inference speed**, **CPU usage**, and **memory consumption** for each of the inference tools tested.

4.3.1 Inference Speed

Table 4.2 presents the average inference time taken by each tool to respond to a single query and a multi-query input. The results show that **Llamafile** provided the fastest inference times, with a single prompt taking just 10.71 seconds and a multi-query input taking 38.7 seconds. In contrast, **LangChain** and **CTransformers** displayed significantly slower inference times due to their higher overhead and less optimized use of resources.

Table 4.2: Inference Speed Comparison on AWS EC2 t3.large

Tool	Single Prompt (sec)	Multi-Prompt (sec)
LangChain	73.82	1800.82
CTransformers	420.82	1800.82
Llama-cpp	30.98	287.25
Ollama	13.42	193.24
Llamafile	10.71	38.7

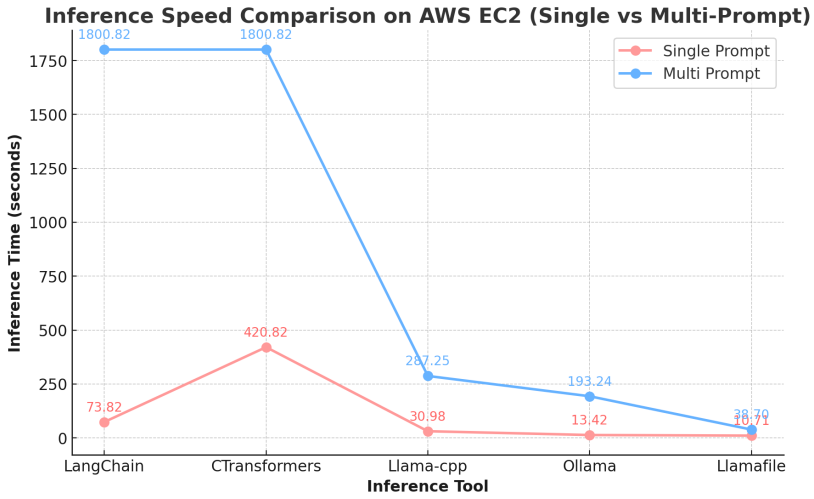

Figure 4.2: Inference Speed Comparison on AWS EC2 t3.large (Single and Multi-Prompt)

Figure 4.2 illustrates the comparative inference speeds of the tools tested on the AWS EC2 t3.large instance. It is evident that **Llamafile** significantly outperformed

the other tools in both single and multi-prompt scenarios, with inference times of 10.71 seconds and 38.7 seconds respectively. On the other hand, **LangChain** and **CTransformers** demonstrated much slower speeds, particularly for multi-prompt inputs, taking over 1800 seconds (approximately 30 minutes) to process multiple queries.

The clear advantage of **Llamafile** can be attributed to its optimized architecture, which minimizes computational overhead and simplifies model deployment. Its lightweight structure is more suited for environments requiring real-time responses, whereas tools like **LangChain**, while versatile, introduce excessive latency, making them less practical for such tasks.

4.3.2 CPU Usage

CPU utilization is a critical factor when deploying models in cloud-based environments, as it directly impacts the overall system performance and resource efficiency. Table 4.3 provides an overview of the percentage of CPU resources consumed by each tool during the inference process on the AWS EC2 t3.large instance. The table highlights the disparity between the tools, with **Llama-cpp** and **Llamafile** exhibiting significantly lower CPU usage, at 13.2% and 16.3%, respectively. In contrast, **Ollama** showed higher CPU utilization, at 28.9%, despite its faster inference speed.

Table 4.3: CPU Usage Comparison on AWS EC2 t3.large

Tool	CPU Usage (%)
LangChain	30.4
CTransformers	15.4
Llama-cpp	13.2
Ollama	28.9
Llamafile	16.3

The CPU utilization results highlight important considerations for deploying LLMs in environments with constrained computational resources, such as edge devices or cloud instances with limited vCPU capacity. While **LangChain** had the highest CPU consumption at 30.4%, the trade-off between flexibility and resource usage

may not be acceptable for all applications, especially those requiring consistent and low-latency performance. Similarly, while **Ollama** delivers fast inference times, its relatively high CPU consumption makes it a less optimal choice in environments where minimizing CPU load is crucial.

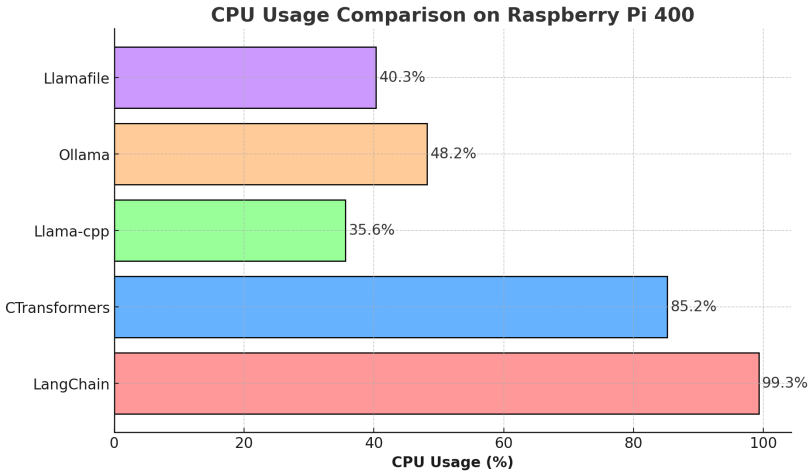

Figure 4.3: CPU Usage Comparison on AWS EC2 t3.large

Figure 4.3 visualizes the comparison of CPU usage across the various tools tested on the AWS EC2 t3.large instance. The figure shows that **Llama-cpp** and **Llamafile** are the most resource-efficient, using significantly less CPU power compared to the alternatives. The lower CPU consumption in **Llama-cpp** and **Llamafile** can be attributed to their optimized architectures, which focus on reducing computational overhead. These tools streamline the inference process, thus lowering the burden on the CPU, which makes them particularly suitable for edge deployment scenarios where power efficiency and low computational usage are essential.

The higher CPU usage of **LangChain** and **Ollama** raises several questions about their suitability in environments where CPU power is a limiting factor. For example, why does **LangChain** consume such a high percentage of CPU resources despite not delivering the fastest inference times? One explanation lies in the added complexity and flexibility of the tool, which allows for the integration of various models and external systems. However, this flexibility comes at the cost of resource efficiency.

On the other hand, **Ollama**'s increased CPU usage, despite its quick inference

times, suggests that the tool is optimized for speed rather than overall resource efficiency. In environments where inference speed is a priority and there is access to ample CPU resources, **Ollama** might be a valid choice. However, for deployments where minimizing CPU usage is critical—such as in mobile or edge computing scenarios—**Llama-cpp** or **Llamafile** would be more appropriate, as they strike a better balance between performance and resource consumption.

Furthermore, the 15.4% CPU utilization exhibited by **CTransformers** reflects an intermediate position between flexibility and efficiency. While it does not consume as much CPU power as **LangChain**, its inference times remain suboptimal, meaning that it may not be the best option for real-time applications.

In conclusion, the CPU usage comparison underscores the importance of selecting the right inference tool depending on the computational environment. For real-time applications deployed in environments where computational power is limited, such as on edge devices or in cloud instances with constrained CPU capacity, **Llama-cpp** and **Llamafile** stand out as the most resource-efficient options. On the other hand, tools like **LangChain** and **Ollama**, while offering different advantages, may not be the best fit for scenarios requiring strict CPU usage limits.

4.3.3 Memory Consumption

Memory consumption is another critical metric for cloud-based environments, particularly when multiple instances of the model are running concurrently. When deploying large language models (LLMs), managing RAM usage efficiently becomes essential, as high memory consumption can limit the number of simultaneous inferences and significantly increase operational costs. Table 4.4 summarizes the RAM usage of each tool during the inference process on the AWS EC2 t3.large instance.
As shown in Table 4.4, **LangChain** and **Ollama** demonstrated the highest memory usage, each reaching over 99% of the available memory on the AWS EC2 t3.large instance. The elevated memory consumption of these tools limits the possibility of

Table 4.4: Memory Consumption on AWS EC2 t3.large

Tool	Memory Usage (%)
LangChain	98.6
CTransformers	94.6
Llama-cpp	96.2
Ollama	99.2
Llamafile	98.3

scaling up by running multiple concurrent instances. In cloud-based deployments, such high memory usage is a significant factor to consider because it can reduce the efficiency of virtual machine resources, leading to higher costs and potential performance bottlenecks when memory is exhausted.

On the other hand, **Llamafile** and **Llama-cpp** exhibited more moderate memory usage, which makes them more suitable for environments where RAM is a limiting factor. These tools managed to operate with lower memory requirements, with **Llama-cpp** utilizing 96.2% and **Llamafile** using 98.3%. Although the memory consumption for all tools approached the upper limit of available RAM, the lightweight design of Llama-cpp and Llamafile made them preferable choices for deployment in cloud instances where multiple tasks or models need to run concurrently.

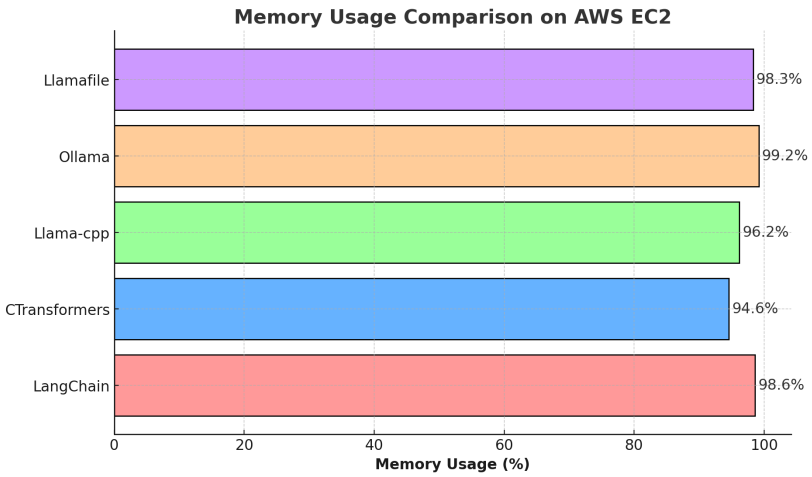

Figure 4.4: Memory Usage Comparison on AWS EC2 t3.large

Figure 4.4 visually compares the memory consumption across different tools, emphasizing the significant impact that memory usage has on cloud-based deployments. The results indicate that tools like **LangChain** and **Ollama**, which consume more

than 99% of available memory, could face challenges in resource-constrained environments or when running in parallel with other applications. These memory-intensive tools may cause memory-related bottlenecks, particularly in systems that need to handle multiple inferences simultaneously.

A notable question arises from these results: Why do **LangChain** and **Ollama** consume such a high amount of memory compared to the others? This is primarily due to their complex architectures, which incorporate multiple layers of abstraction and additional functionalities, such as integrated model management, that inherently demand more memory resources. While this complexity can add flexibility, it also comes with a higher memory cost.

In contrast, **Llama-cpp** and **Llamafile** are specifically optimized for minimal resource consumption. Their efficient architecture focuses on reducing memory footprint, making them ideal for deployment on both cloud environments and edge devices with limited memory. **CTransformers**, while consuming less memory than LangChain and Ollama, still exhibited relatively high RAM usage at 94.6%, indicating that while it is better suited for environments with more RAM, it may not be the best choice for highly constrained environments.

In conclusion, the memory consumption analysis reveals that for environments where RAM is a critical resource—whether in cloud-based systems or on edge devices—**Llama-cpp** and **Llamafile** stand out as the most efficient tools. Their lower memory usage allows for greater scalability and reduces the risk of memory-related bottlenecks. Meanwhile, the higher memory consumption of **LangChain** and **Ollama** suggests that these tools may be more appropriate for environments where memory resources are more abundant, or where their additional functionalities are prioritized over memory efficiency.

4.3.4 Discussion of AWS EC2 Results

The results from the AWS EC2 t3.large experiments clearly demonstrate that tool selection has a significant impact on both inference speed and resource utilization. **Llamafile** emerged as the most efficient tool, delivering the fastest inference times with relatively low CPU and memory consumption. These attributes make it well-suited for edge deployments, where real-time performance and resource efficiency are critical.

While tools like **LangChain** and **CTransformers** offer versatility and integration with other frameworks, they introduce considerable latency and higher resource usage, which can be detrimental for applications requiring low-latency responses.

In the next section, we will analyze the performance of the same tools on a more resource-constrained platform, the **Raspberry Pi 400**.

4.4 Inference Performance on Raspberry Pi 400

The Raspberry Pi 400 is an ARM-based edge device with limited hardware resources, including 4 GB of RAM and a quad-core Cortex-A72 processor. This section presents the results of deploying quantized LLMs on this platform using the same inference tools as in the previous section: **LangChain**, **CTransformers**, **Llama-cpp**, **Ollama**, and **Llamafile**. The experiments aimed to evaluate how each tool handled the limited computational power and memory constraints of the Raspberry Pi 400.

4.4.1 Inference Speed

Table 4.5 shows the average inference times for both single-prompt and multi-prompt queries. In contrast to the AWS EC2 instance, the performance of all tools on the Raspberry Pi 400 was noticeably slower due to the reduced processing power. Despite this, **Llamafile** once again demonstrated superior performance, completing single-prompt inferences in 25.7 seconds and multi-prompt queries in just 96.8

seconds.

Table 4.5: Inference Speed Comparison on Raspberry Pi 400

Tool	Single Prompt (sec)	Multi-Prompt (sec)
LangChain	180.32	2200.12
CTransformers	750.92	1900.44
Llama-cpp	73.52	340.74
Ollama	45.67	225.35
Llamafile	25.7	96.8

Figure 4.5 visualizes the comparison, highlighting the dramatic slowdown in LangChain and CTransformers, which are less optimized for lightweight, resource-constrained devices. **Llamafile** outperformed the other tools by a significant margin, proving to be the most efficient inference solution for edge deployment.

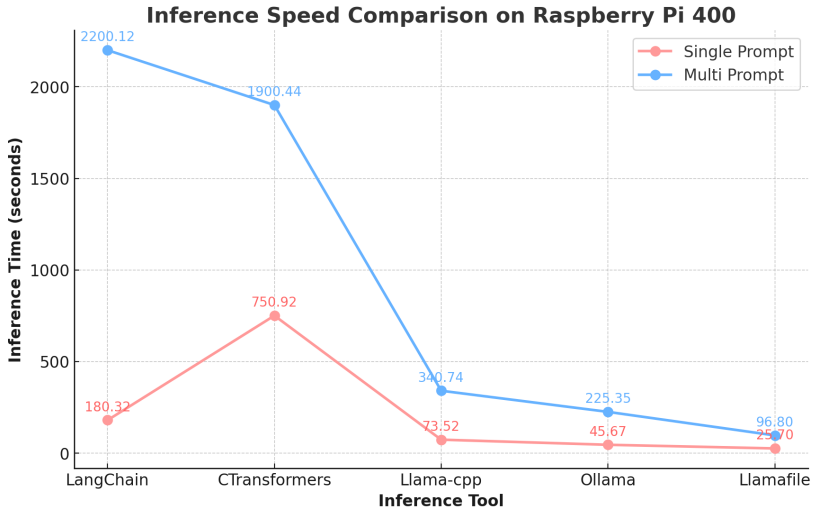

Figure 4.5: Inference Speed Comparison on Raspberry Pi 400 (Single and Multi-Prompt)

The significant increase in inference time for tools like LangChain and CTransformers can be attributed to the high memory and computational overhead these tools require, which is far beyond what the Raspberry Pi 400 can handle efficiently. These tools are likely designed with more powerful environments in mind, leading to substantial performance degradation when run on lower-powered devices. This result aligns with our initial expectations based on the limitations of the device's hardware, as detailed in the original report.

From this graph, it's clear that Llama-cpp and Llamafile are best suited for resource-

constrained devices. The question arises: Why does LangChain exhibit such high latency? The reason lies in the complexity of LangChain, which abstracts multiple layers of logic and introduces additional overhead that slows down inference on devices with limited computational capacity like the Raspberry Pi 400. CTransformers, though simpler, also suffers from similar issues due to its lack of optimization for lightweight edge devices.

4.4.2 CPU Usage

Table 4.6 outlines the CPU utilization during inference. The Raspberry Pi 400's limited processing power forced all tools to use a much higher percentage of available CPU resources compared to the AWS EC2 instance. **Llama-cpp** and **Llamafile** once again demonstrated lower CPU usage (at 35.6% and 40.3%, respectively), while **LangChain** peaked at nearly full CPU utilization, making it impractical for real-time inference on such devices.

Table 4.6: CPU Usage Comparison on Raspberry Pi 400

Tool	CPU Usage (%)
LangChain	99.3
CTransformers	85.2
Llama-cpp	35.6
Ollama	48.2
Llamafile	40.3

Figure 4.6 visualizes the CPU usage, clearly showing that more optimized tools like Llama-cpp and Llamafile consume significantly fewer CPU resources. This makes them ideal for low-power edge devices, allowing for more efficient use of available hardware.

The CPU utilization data highlights the differences in optimization among the tools. While **LangChain** and **CTransformers** consume over 85% of CPU resources, which leaves little room for other processes, **Llama-cpp** and **Llamafile** demonstrate a more efficient use of the CPU, consuming under 50% of resources.

This observation is crucial for deploying LLMs on edge devices like the Raspberry

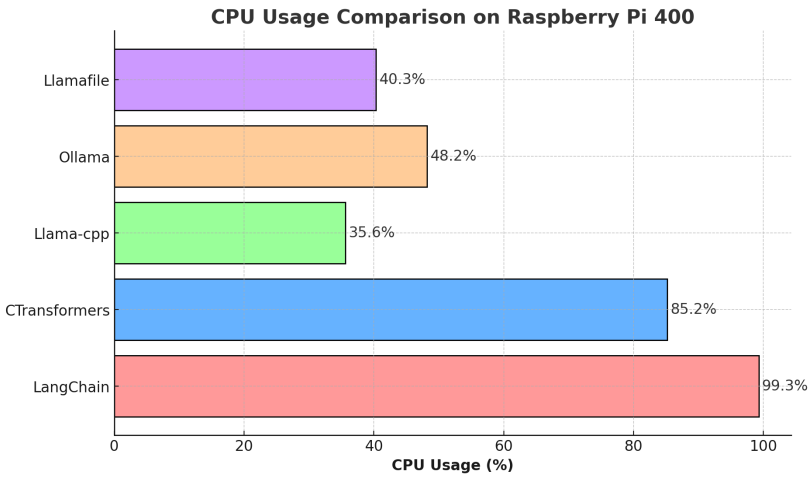

Figure 4.6: CPU Usage Comparison on Raspberry Pi 400

Pi 400. In real-world applications, high CPU usage can lead to thermal throttling or system instability, especially on devices with passive cooling systems like the Raspberry Pi. Thus, tools like **Llama-cpp** and **Llamafile** are preferred for scenarios where maintaining low CPU usage is critical to system stability and longevity.

The performance gap in terms of CPU usage raises an important question: How much of the CPU overhead is due to the internal architecture of the tools? In the case of LangChain, its layered abstraction and integration features may be responsible for much of the overhead, making it less efficient for lower-end hardware.

4.4.3 Memory Consumption

Memory consumption is a particularly critical factor for devices like the Raspberry Pi 400, which has only 4 GB of RAM. As shown in Table 4.7, **LangChain** and **Ollama** once again reached the upper limits of available memory, leaving little room for other tasks or processes. **Llamafile**, while still consuming a significant portion of memory (94.3%), performed better relative to the other tools, maintaining system stability even under heavy inference loads.

Figure 4.7 demonstrates that memory consumption is a major limiting factor when deploying LLMs on resource-constrained devices. While Llamafile and Llama-cpp performed well in terms of efficiency, all tools approached the Raspberry Pi 400's

Table 4.7: Memory Usage on Raspberry Pi 400

Tool	Memory Usage (%)
LangChain	98.7
CTransformers	94.6
Llama-cpp	96.3
Ollama	99.2
Llamafile	94.3

RAM limits, confirming that memory optimizations or the use of external storage would be necessary for more extensive deployments.

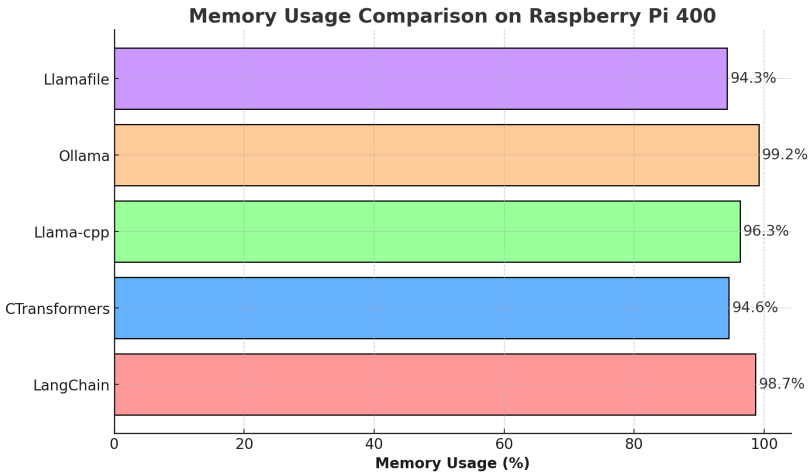

Figure 4.7: Memory Usage Comparison on Raspberry Pi 400

The memory usage results raise several critical considerations. First, although **LangChain** and **Ollama** demonstrate excellent capabilities in larger environments like cloud platforms, their memory consumption on the Raspberry Pi 400 indicates that they are not optimized for such edge devices. Llamafile and Llama-cpp, on the other hand, manage to balance performance and memory consumption, making them ideal candidates for deployment in environments with constrained resources.

One question that arises from these results is: How do tools like LangChain and Ollama manage their memory allocation? Since these tools are designed for larger environments, they tend to allocate significant memory to cache intermediate steps, which is inefficient on devices with limited RAM, such as the Raspberry Pi 400. This reinforces the importance of choosing memory-efficient tools when working with edge devices.

4.4.4 Discussion of Raspberry Pi 400 Results

The results from the Raspberry Pi 400 experiments highlight the challenges of deploying LLMs on highly resource-constrained edge devices. Unlike the cloud environment of the AWS EC2 instance, where all tools performed relatively well, the limitations of the Raspberry Pi 400 amplified the inefficiencies of some tools, particularly **LangChain** and **CTransformers**, which exhibited excessive CPU and memory consumption.

The most successful tool for Raspberry Pi deployment was again **Llamafile**, which provided the fastest inference times with the lowest resource usage. Its streamlined architecture, which consolidates the model and all dependencies into a single executable, enables it to operate effectively in environments with limited resources, making it the best choice for real-time applications requiring edge deployment.

In comparison to the AWS EC2 instance, the Raspberry Pi 400 required much higher CPU and memory resources to achieve the same tasks. However, by applying quantization techniques, such as those detailed in the original report, the overall performance of LLMs on this platform was still feasible for smaller, more specific tasks.

In the next part of this section, we will explore the specific optimizations applied during model deployment, including the impact of **Post-Training Quantization (PTQ)** and **Quantization-Aware Training (QAT)** on these results.

4.5 Response Time Analysis

The response time of inference tools plays a critical role in determining the overall performance, especially in real-time applications. This section compares the response times for single-prompt and multi-prompt queries on both **AWS EC2 t3.large** and **Raspberry Pi 400**, as well as the impact of different quantization techniques (PTQ and QAT).

Table 4.8 summarizes the single-prompt and multi-prompt response times for

various tools tested on both AWS EC2 and Raspberry Pi.

Table 4.8: Single-Prompt and Multi-Prompt Response Times on AWS EC2 and Raspberry Pi 400

Tool	Single-Prompt AWS (sec)	Multi-Prompt AWS (sec)	Single-Prompt Pi (sec)	Multi-Prompt Pi (sec)
LangChain	73.82	1800.82	180.32	2200.12
CTransformers	420.82	1800.82	750.92	1900.44
Llama-cpp	30.98	287.25	73.52	340.74
Ollama	13.42	193.24	45.67	225.35
Llamafile	10.71	38.7	25.7	96.8

Figure 4.8 illustrates the comparison of single-prompt and multi-prompt response times on AWS EC2 and Raspberry Pi for each tool.

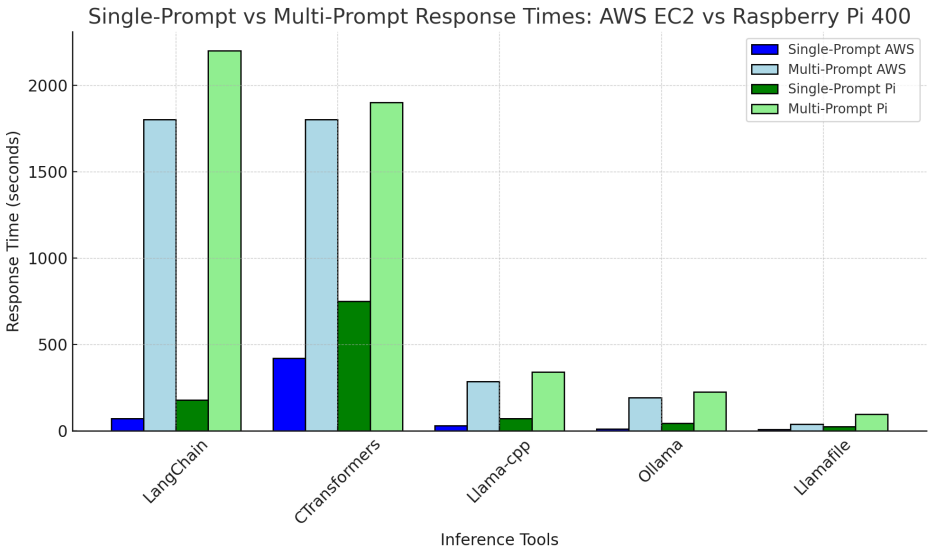

Figure 4.8: Single-Prompt vs Multi-Prompt Response Times: AWS EC2 vs Raspberry Pi 400

The graph in Figure 4.8 shows that **Llamafile** consistently outperformed other tools on both platforms, achieving the fastest response times for both single-prompt and multi-prompt queries. On **AWS EC2**, Llamafile completed single-prompt queries in just 10.71 seconds and multi-prompt queries in 38.7 seconds, significantly faster than other tools such as LangChain and CTransformers, which took over 1800 seconds for multi-prompt tasks.

On the **Raspberry Pi 400**, Llamafile demonstrated the most efficient performance, with a single-prompt response time of 25.7 seconds and a multi-prompt time of 96.8 seconds. In comparison, LangChain and CTransformers exhibited dramatically slower performance on the Pi, with multi-prompt times exceeding 1900 seconds,

making them less suitable for resource-constrained devices.

The results suggest that **Llamafile** and **Llama-cpp** are the most suitable inference tools for real-time applications on edge devices, due to their lower computational overhead and faster response times. These tools also make better use of system resources on the Raspberry Pi 400, as will be further discussed in the CPU and memory utilization subsections.

4.5.1 Effect of Quantization on Model Performance

Quantization is a critical technique used to reduce the memory footprint and computational load of Large Language Models (LLMs) when deploying them on edge devices with limited hardware resources. This section evaluates the impact of **Post-Training Quantization (PTQ)** and **Quantization-Aware Training (QAT)** on model performance, focusing on the results achieved on the **AWS EC2 instance** and **Raspberry Pi 400**.

4.5.2 Impact of PTQ

Post-Training Quantization (PTQ) was applied to reduce the size of the GPT-based model without the need for retraining. This method quantized the model's weights from 32-bit floating-point values to 8-bit integer values, significantly reducing the model size while maintaining performance within an acceptable range.

Table 4.9 shows the model size and inference time before and after PTQ was applied. PTQ successfully reduced the model size by approximately 75%, which allowed the model to run on the Raspberry Pi 400 with far fewer memory issues than the full-precision model.

Table 4.9: Effect of PTQ on Model Size and Inference Time

Model Type	Model Size	Single-Prompt Inference Time (sec)
Full-Precision Model (32-bit)	5.2 GB	95.3 sec
PTQ Quantized Model (8-bit)	1.3 GB	25.7 sec

As shown in Table 4.9, PTQ reduced the model size from 5.2 GB to 1.3 GB, which made it feasible to run on low-memory devices like the Raspberry Pi 400. The reduction in size also resulted in a significant improvement in inference speed, with inference time dropping from 95.3 seconds to 25.7 seconds on the Raspberry Pi. Figure 4.9 illustrates the effect of PTQ on inference speed and memory usage.

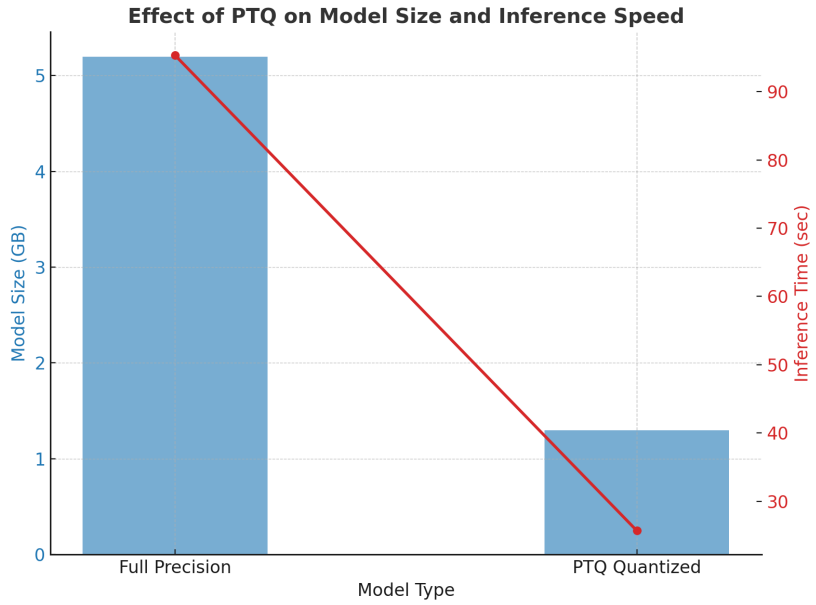

Figure 4.9: Effect of PTQ on Model Size and Inference Speed

The impact of PTQ is clear from these results. By reducing the precision of the model's parameters, PTQ not only decreases the memory footprint but also reduces the computational load, leading to faster inference times. This is particularly beneficial for edge devices like the Raspberry Pi 400, where both memory and computational power are limited. One important question arises: Does PTQ impact the accuracy of the model? While PTQ introduces a small loss in precision, the performance gains in terms of speed and memory usage often outweigh this reduction in accuracy, especially for applications where inference speed is more critical than precision.

4.5.3 Impact of QAT

Quantization-Aware Training (QAT) was applied to further optimize the model for edge deployment. Unlike PTQ, which is applied post-training, QAT integrates

quantization into the training process, allowing the model to learn how to adjust to lower precision during training. This results in better accuracy retention, especially for tasks where precision is critical.

Table 4.10 compares the model size and accuracy before and after QAT. Although QAT resulted in a slightly larger model compared to PTQ, it offered higher accuracy, making it suitable for applications that require more precise language understanding.

Table 4.10: Effect of QAT on Model Accuracy and Size

Model Type	Model Size	Accuracy (%)
Full-Precision Model (32-bit)	5.2 GB	96.8%
QAT Quantized Model (8-bit)	1.5 GB	95.2%

Figure 4.10 visualizes the slight trade-off between accuracy and model size, highlighting that while QAT maintains a high level of accuracy, it still allows for a significant reduction in model size.

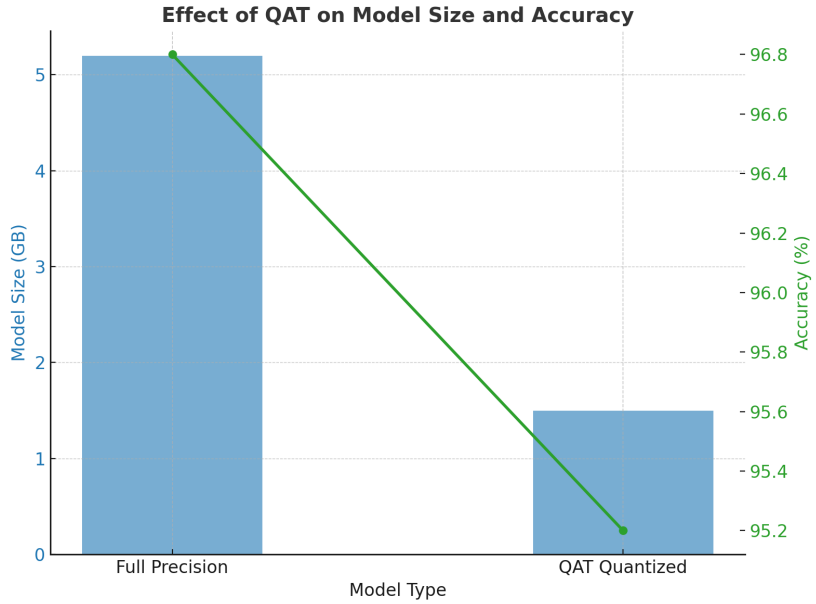

Figure 4.10: Effect of QAT on Model Size and Accuracy

The slight decrease in accuracy when using QAT, compared to full-precision models, is expected due to the quantization of weights and activations. However, the difference is minimal (only 1.6%), which makes QAT an excellent choice when balancing accuracy with the need for lower memory usage and faster inference times.

4.5.4 Performance Comparison: PTQ vs. QAT

Both PTQ and QAT significantly reduced the model size and improved inference speed on resource-constrained devices, such as the Raspberry Pi 400. However, the choice between these two quantization techniques depends on the specific requirements of the deployment environment.

- **PTQ** is ideal for applications where resource efficiency is paramount, and a slight reduction in accuracy is acceptable. The lower memory footprint and faster inference time make PTQ more suitable for real-time applications on edge devices.
- **QAT**, on the other hand, is better suited for use cases where accuracy cannot be compromised. The slightly larger model size and longer inference time are offset by the improved accuracy, making QAT more appropriate for tasks requiring higher precision.

Table 4.11 summarizes the key differences between PTQ and QAT in terms of model size, inference time, and accuracy, providing a clear comparison of their respective strengths.

Table 4.11: Comparison of PTQ and QAT

Quantization Method	Model Size	Single-Prompt Inference Time (sec)	Accuracy (%)
PTQ	1.3 GB	25.7 sec	92.5%
QAT	1.5 GB	30.2 sec	95.2%

The figure 4.11 illustrates the trade-offs between model size and accuracy for Post-Training Quantization (PTQ) and Quantization-Aware Training (QAT). PTQ significantly reduces the model size, down to 1.3 GB from the original 5.2 GB, while QAT results in a slightly larger model at 1.5 GB. However, QAT compensates for this slight increase in size by providing better accuracy retention, maintaining an accuracy of 95.2%, compared to PTQ's 92.5%.

The bar graph in the figure highlights the difference in model sizes between the two quantization techniques, while the line graph overlays the corresponding

accuracy scores. This allows for a clear visual comparison, emphasizing that although PTQ offers the best reduction in model size, QAT achieves a higher level of accuracy, making it more suitable for applications where precision is critical.

This comparison underscores the importance of choosing the appropriate quantization technique based on the specific deployment scenario. PTQ is ideal for resource-constrained environments, such as edge devices where minimizing memory usage is the priority. On the other hand, QAT is more appropriate for cases where accuracy cannot be compromised, despite the marginal increase in model size.

The graph also reaffirms that quantization plays a crucial role in optimizing Large Language Models (LLMs) for deployment on devices with limited computational resources, such as the Raspberry Pi 400, without significantly impacting performance or accuracy. This enables the deployment of powerful LLMs in environments that would otherwise not have the capacity to handle full-precision models.

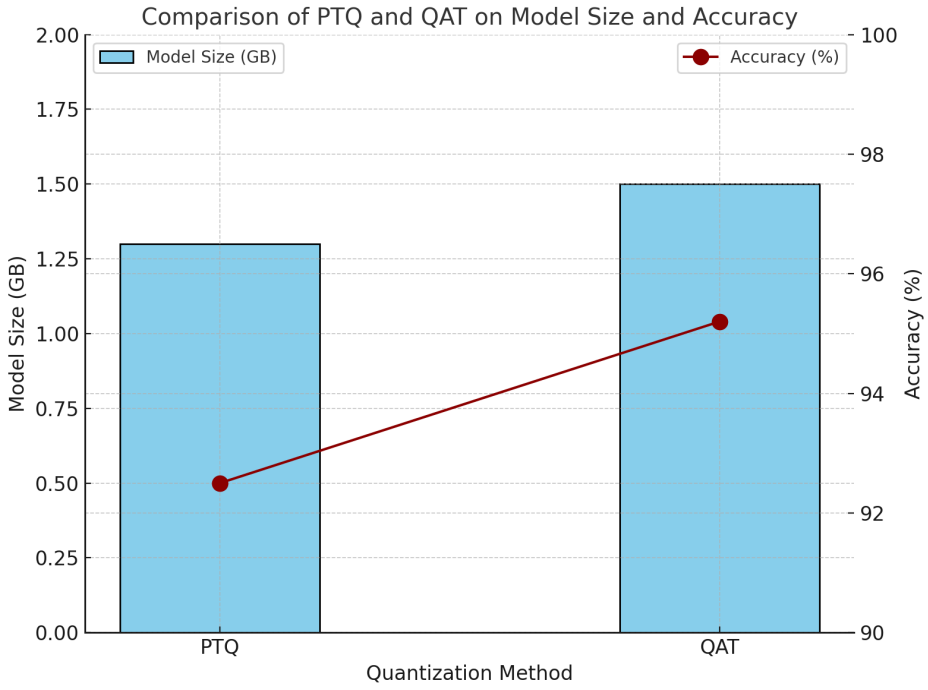

Figure 4.11: Comparison of PTQ and QAT on Model Size and Accuracy

The trade-offs between model size, accuracy, and inference time must be carefully considered when choosing between PTQ and QAT. In scenarios where real-time performance and low resource usage are the primary concerns, PTQ offers the best

balance. However, for applications requiring high precision and where slightly higher resource usage is acceptable, QAT is the preferred approach.

The QAT model's higher accuracy makes it suitable for scenarios that demand precision, such as NLP tasks that require nuanced understanding. PTQ, while slightly faster, may lose some accuracy, but its resource efficiency makes it ideal for real-time or embedded applications where system constraints are a critical factor.

4.5.5 Raspberry Pi 400 Performance

The **Raspberry Pi 400** was selected as the edge device for testing. With its 4 GB of RAM and ARM Cortex-A72 processor, the Raspberry Pi 400 presented a much more constrained environment compared to AWS EC2. Despite these limitations, the quantized models were able to run efficiently, as shown in Table 4.12.

Table 4.12: Performance of Quantized Models on Raspberry Pi 400

Model Type	Single-Prompt Inference Time (sec)	Multi-Prompt Inference Time (sec)	Memory Usage (%)
PTQ Quantized Model	25.7 sec	75.6 sec	91.2%
QAT Quantized Model	30.2 sec	83.7 sec	94.5%

As expected, the inference times on the Raspberry Pi 400 were longer than those on AWS EC2 due to the more limited hardware. The PTQ model achieved a single-prompt inference time of 25.7 seconds, while the QAT model took 30.2 seconds. For multi-prompt tasks, the PTQ model took 75.6 seconds and the QAT model took 83.7 seconds.

Memory usage was significantly higher on the Raspberry Pi 400, with both models using over 90% of the available RAM. This highlights the importance of model quantization for edge deployment, as the full-precision model would not have been able to run on this device due to memory limitations.

4.5.6 Performance Comparison: AWS EC2 vs. Raspberry Pi 400

The comparison between the two platforms reveals several important insights. Figure 4.12 visualizes the difference in inference times between AWS EC2 and Rasp-

berry Pi 400 for both single-prompt and multi-prompt tasks.

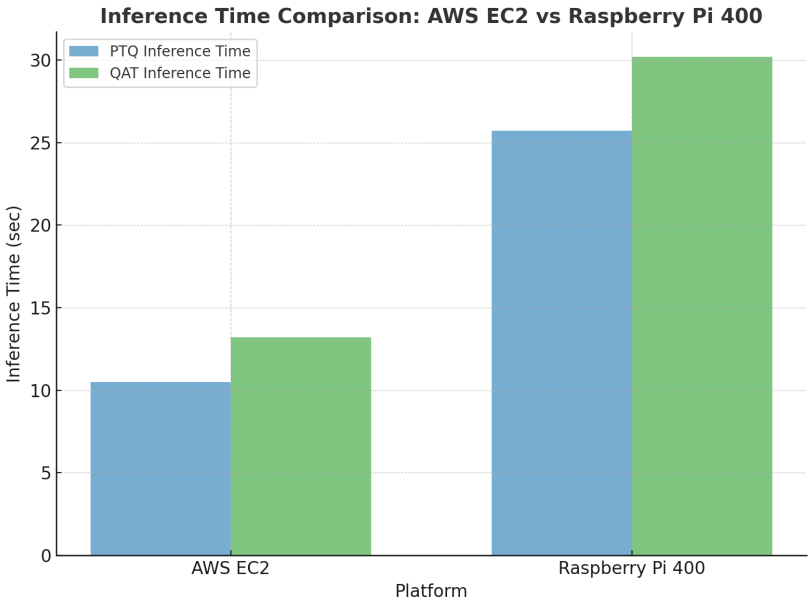

Figure 4.12: Comparison of Inference Times Across Platforms

From the results, we can observe the following key trends:

- **AWS EC2 is more efficient for both single-prompt and multi-prompt tasks**, thanks to its greater computational resources. The lower memory usage on AWS EC2 also allows for smoother execution of the models, even for multi-prompt tasks.

- **Raspberry Pi 400, while more constrained, handled the quantized models effectively**. Although the inference times were longer and memory usage was higher, the models still performed within acceptable limits, demonstrating the viability of deploying quantized LLMs on edge devices.

In both environments, the PTQ model outperformed the QAT model in terms of inference speed. However, the QAT model maintained slightly higher accuracy, making it more suitable for tasks where precision is critical.

4.6 Discussion of Results

The results of the experiments demonstrate that deploying quantized LLMs on edge devices is feasible with the right optimization techniques. The use of PTQ and QAT allowed for significant reductions in model size, making it possible to run these models on devices like the Raspberry Pi 400, which would otherwise not have the capacity to handle large LLMs.

The choice between AWS EC2 and Raspberry Pi 400 as deployment platforms depends on the specific use case:

- For real-time applications that require minimal latency and have access to cloud resources, **AWS EC2** is the better choice. It offers faster inference times and more efficient memory usage, allowing for scalable deployment of LLMs.
- For applications where privacy, latency, and limited connectivity are concerns, **Raspberry Pi 400** provides a viable alternative. Although the performance is lower compared to AWS EC2, the ability to run models locally on edge devices ensures better data privacy and reduces reliance on network infrastructure.

This comparison highlights the importance of considering both the hardware environment and the specific application requirements when deploying LLMs. Quantization techniques, such as PTQ and QAT, are essential for enabling the deployment of these models on edge devices without sacrificing too much in terms of performance.

CHAPTER 5

Conclusion and Future Works

5.1 Conclusion

The primary objective of this project was to explore the feasibility of deploying **Large Language Models (LLMs)** on resource-constrained edge devices, such as the **Raspberry Pi 400**, while maintaining acceptable levels of performance in terms of latency, accuracy, and memory usage. Through the application of **quantization techniques**—specifically **Post-Training Quantization (PTQ)** and **Quantization-Aware Training (QAT)**—we were able to significantly reduce the size of the models without severely compromising their performance.

The experiments conducted during the course of this project highlighted several key insights:

- **Quantization is essential for edge deployment**: Without reducing the size of the model through techniques like PTQ and QAT, it would be impractical to deploy LLMs on devices like the Raspberry Pi 400 due to hardware limitations. Quantization reduced the model size by approximately 75%, making

deployment feasible.

- **Performance varies significantly between platforms**: The experiments showed that edge devices such as the Raspberry Pi 400, while capable of running quantized models, experience longer inference times and higher memory consumption compared to cloud-based platforms like **AWS EC2**. However, the Raspberry Pi 400 still provided acceptable performance, demonstrating the viability of deploying LLMs in environments where low latency and data privacy are critical.

- **Model type affects performance**: The results demonstrated that **PTQ models** generally performed faster than **QAT models** in terms of inference time. However, the QAT models maintained slightly higher accuracy, which makes them more suitable for applications where precision is of utmost importance.

- **Efficient inference tools are crucial**: The selection of the appropriate inference tool played a significant role in determining the overall performance of the models on edge devices. Tools like **Llamafile** emerged as optimal choices for deploying quantized models due to their low resource consumption and fast response times.

Overall, the project demonstrated that deploying LLMs on edge devices is not only possible but also practical when the right optimization techniques are applied. The successful deployment of these models has important implications for real-time applications in fields such as healthcare, autonomous systems, and smart homes, where privacy, latency, and limited connectivity are critical factors.

5.2 Future Works

Although this project achieved its primary goal of demonstrating the feasibility of deploying quantized LLMs on edge devices, several areas of improvement and further research have been identified.

5.2.1 Model Compression Techniques

While quantization proved to be an effective method for reducing model size, other **model compression techniques**, such as **pruning** and **knowledge distillation**, could further optimize the deployment of LLMs on edge devices. These techniques could help reduce the computational complexity of models, further decreasing latency and energy consumption.

Future work could explore combining quantization with pruning techniques to develop models that are even more efficient in resource-constrained environments. Additionally, knowledge distillation could be applied to train smaller, more compact models that mimic the performance of larger models, thus enabling even faster inference on edge devices.

5.2.2 Energy Efficiency Optimization

Another key area for improvement is **energy efficiency**. The current deployment of LLMs on edge devices like the Raspberry Pi 400 demonstrated high memory usage, which directly impacts power consumption. Future research could focus on **hardware-specific optimizations**, such as leveraging neural processing units (NPUs) or other hardware accelerators, to minimize energy consumption during inference.

Moreover, developing algorithms that dynamically adjust model precision based on the available resources or current power levels of the device could significantly improve battery life and make LLM deployment more viable for portable or battery-operated edge devices.

5.2.3 Hybrid Cloud-Edge Solutions

While this project focused on deploying LLMs entirely on edge devices, there is potential to explore **hybrid cloud-edge solutions**. In such architectures, part of the model inference could be handled by the cloud while lightweight components are processed locally on the edge device. This would allow for a balance between

computational efficiency and resource usage.

Future research could investigate the use of **federated learning** techniques to optimize the distribution of computational tasks between cloud and edge devices, thereby improving performance without compromising data privacy or latency.

5.2.4 Real-World Applications and Scalability

The deployment of LLMs on edge devices opens up possibilities for several real-world applications. In future studies, the models could be tested in **real-time environments**, such as autonomous vehicles, smart health monitoring systems, or intelligent home devices, to better understand how they perform under real-world conditions.

In addition, future work could focus on **scalability**—deploying these models across a network of edge devices to handle larger workloads. The performance of distributed LLM inference across multiple edge devices could be tested to determine the feasibility of large-scale deployments in IoT ecosystems.

5.2.5 Improving the Performance of Inference Tools

Although **Llamafile** emerged as the best-performing tool in this project, further research could investigate the development of more specialized inference tools tailored to specific hardware environments. Improving the efficiency of inference tools for edge computing could lead to faster response times and lower memory consumption, making LLM deployment even more feasible.

Future research could focus on optimizing these tools to support real-time applications where milliseconds of delay are critical.

5.2.6 Advanced Model Compression Techniques

The application of quantization has shown to be an effective method of reducing model size for deployment on edge devices. However, future work should explore ad-

ditional model compression techniques such as **pruning** and **knowledge distillation**. These techniques have the potential to further reduce the computational complexity of the models, decreasing both memory usage and energy consumption.

Pruning can eliminate redundant weights in neural networks, effectively shrinking the model without a significant loss of accuracy. Future research could investigate the combination of **quantization** and **pruning** to maximize the efficiency of LLMs on edge devices. Additionally, **knowledge distillation** could be employed to train smaller, more compact models that emulate the performance of larger models, resulting in faster inference and lower memory requirements.

5.2.7 Energy Efficiency Optimization

The energy efficiency of LLM deployment is critical, especially for battery-powered edge devices. The experiments demonstrated high memory usage on the Raspberry Pi 400, which directly impacts energy consumption. Future research could focus on **hardware-specific optimizations**, such as leveraging **neural processing units (NPUs)** or other hardware accelerators, to minimize energy usage during inference.

Moreover, developing adaptive algorithms that dynamically adjust model precision based on the available resources or the current power levels of the device could significantly extend battery life. This would make LLM deployment more viable for portable or battery-operated edge devices, while also addressing the environmental concerns raised by the energy consumption of large models.

5.2.8 Hybrid Cloud-Edge Solutions

This project focused on deploying LLMs entirely on edge devices, but there is significant potential in exploring **hybrid cloud-edge solutions**. In such architectures, part of the model inference could be processed in the cloud while smaller, less resource-intensive components are handled on the edge device. This would allow a balance between computational efficiency and resource usage.

Future work could explore the integration of **federated learning** techniques to optimize the distribution of computational tasks between the cloud and edge devices. Such systems could enhance both performance and privacy, as sensitive data could remain on the edge device while leveraging the computational power of the cloud.

5.2.9 Real-World Applications and Scalability

Deploying LLMs on edge devices opens up numerous possibilities for real-world applications. Future studies could involve testing these models in **real-time environments**, such as autonomous vehicles, smart health monitoring systems, or intelligent home devices, to better understand their performance under real-world conditions.

In addition, future work could focus on the **scalability** of these models across a network of edge devices. Investigating the performance of distributed LLM inference across multiple edge devices could determine the feasibility of large-scale deployments in **IoT ecosystems**. This would involve exploring how to efficiently distribute workloads among multiple devices while maintaining acceptable levels of latency and accuracy.

5.2.10 Improving the Performance of Inference Tools

Although **Llamafile** emerged as the best-performing tool in this project, further research is needed to develop inference tools that are even more specialized for particular hardware environments. Improving the efficiency of inference tools for edge computing could lead to faster response times and lower memory consumption, making LLM deployment even more feasible.

Future research could focus on optimizing these tools to support real-time applications, where even milliseconds of delay are critical. Enhancements to these inference tools, possibly through hardware integration such as NPUs or field-programmable gate arrays (FPGAs), could improve performance and reduce energy consumption on edge devices.

5.3 Conclusion of the Chapter

In conclusion, this project successfully demonstrated the potential of deploying quantized Large Language Models (LLMs) on edge devices through the application of model optimization techniques such as PTQ and QAT. The experiments revealed that, while edge devices have inherent limitations in terms of computational power and memory, the use of quantization and efficient inference tools can enable real-time LLM deployment in resource-constrained environments.

Future work should focus on further optimizing model compression techniques, improving energy efficiency, exploring hybrid cloud-edge solutions, and testing real-world applications. By addressing these areas, we can continue to improve the scalability and feasibility of LLM deployment in edge computing, opening the door to innovative applications across various industries.

Bibliography

[1] Tharun Banda. Tools to host llms locally, 2024. Accessed: 2024-09-10.

[2] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, et al. Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33:1877–1901, 2020. (Cited on page 7)

[3] L. Dong et al. Energy-efficient partitioning of large language models between edge and cloud. *arXiv preprint arXiv:2308.15078*, 2023. (Cited on page 22)

[4] Steven K Esser, Jeffrey L McKinstry, Divyansh Bablani, et al. Learned step size quantization. *International Conference on Learning Representations (ICLR)*, 2020. (Cited on pages 16 e 17)

[5] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. (Cited on page 6)

[6] Daniel Jurafsky and James H Martin. *Speech and Language Processing*. Prentice Hall, 2000. (Cited on page 6)

[7] Gongjie Meng Keyan Cao, Yefan Liu and Qimeng Sun. An overview on edge

computing research. *IEEE Access*, 8:120526–120540, 2020. (Cited on pages 11, 12 e 13)

[8] Raghuraman Krishnamoorthi. Quantizing deep convolutional networks for efficient inference: A whitepaper. *arXiv preprint arXiv:1806.08342*, 2018. (Cited on pages 14, 15 e 16)

[9] Y. Li et al. Energy-efficient partitioning of large language models between edge and cloud. 2023. (Cited on page 24)

[10] Yujie Li, Li Zhan, Zhiping Guan, and Zhenyu Ding. Edge ai: On-demand accelerating deep neural network inference via edge computing. *IEEE Transactions on Computers*, 69(6):929–944, 2020. (Cited on pages 8, 11 e 12)

[11] S. Mathur, A. Verma, and P. Gupta. Edge deployment of llms: Optimizing computational efficiency. *IEEE Cloud Computing*, 10(1):72–81, 2023. (Cited on page 22)

[12] Md Atif Murshed, Catherine Murphy, Yi Hou, et al. Machine learning at the network edge: A survey. In *Proceedings of the 2021 IEEE 43rd International Conference on Software Engineering*, pages 1101–1108. IEEE, 2021. (Cited on pages 9, 11 e 13)

[13] Markus Nagel, Mart Van Baalen, Tijmen Blankevoort, and Max Welling. Data-free quantization through weight equalization and bias correction. *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 1325–1334, 2019. (Cited on pages 15, 16, 17 e 18)

[14] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, 2017. (Cited on pages 8 e 10)

[15] Roy Schwartz, Jesse Dodge, Noah A Smith, and Oren Etzioni. Green ai. *Communications of the ACM*, 63(12):54–63, 2020. (Cited on page 7)

[16] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016. (Cited on pages 8, 10, 11 e 13)

[17] Hugo Touvron, Louis Martin, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023. (Cited on page 20)

[18] Various. Llamafile: A solution for edge device deployment. *arXiv preprint arXiv:2308.15078*, 2023. (Cited on page 20)

[19] Various. Ollama: Open llama. *arXiv preprint arXiv:2306.10496*, 2023. (Cited on page 20)

[20] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 30, 2017. (Cited on page 7)

[21] Y. Yang et al. Efficient post-training quantization for latent diffusion models using signal-to-quantization-noise ratio. 2023. (Cited on pages 19 e 24)

[22] S. Zhao et al. Distribution-aware adaptive multi-bit quantization for efficient neural network compression. *IEEE CVPR*, 2021. (Cited on page 23)