‘Fine-tuning Large Language Model (LLM) using Azure for IC Automation’

By Shilpa Bombale

Under the supervision of

Professor Dr. Lihong Zhang

MASc Computer Engineering

Acknowledgement

I would like to express my deepest gratitude to my supervisor, **Dr. Lihong Zhang**, for his continuous support, guidance, and insightful feedback throughout this project. This work would not have been possible without his expertise and encouragement.

I extend my thanks to **Morteza Golzan**, for his valuable contributions and collaboration. His insights and discussions have greatly enriched this project.

I am also grateful to **Hamed Nasiri** for providing valuable insights and feedback, which helped shape this project.

INDEX

Abstract

Large Language Models (LLMs) have garnered significant attention due to their exceptional performance in various natural language tasks since the release of ChatGPT in November 2022. The ability of LLMs to understand and generate language is achieved by training billions of parameters on vast amounts of text data, aligning with predicted scaling laws [1], [2]. Despite being a recent research area, LLMs are evolving rapidly. This gives an overview of techniques for building and augmenting LLMs, fine-tuning with Azure for IC automation, and evaluating the performance of LLMs using representative benchmarks. Additionally, open challenges and future research directions are discussed [3].

I. Introduction

LLMs are large-scale, pre-trained, statistical language models based on neural networks. LLMs have demonstrated remarkable capabilities in natural language processing and beyond, leading to a surge in research contributions. These works cover diverse topics such as architectural innovations, training strategies, context length improvements, fine-tuning, multi-modal LLMs, robotics, datasets, benchmarking, and efficiency [4]. Fine-tuning these pre-trained models on specific datasets can significantly enhance their performance on domain-specific tasks. This process involves training the model on a smaller, domain- specific dataset, allowing it to learn the nuances and intricacies of the target domain. Azure Machine Learning (Azure ML) provides a robust platform for training and deploying machine learning models at scale. With its extensive range of tools and services, Azure ML facilitates the entire machine learning lifecycle, from data preparation to model training and deployment.

In this paper, we explore the process of fine-tuning a large language model on a specific dataset using Azure ML. We utilize the LLaMA 3 70B model, a state-of-the-art LLM, and demonstrate the steps involved in fine-tuning this model to achieve optimal performance for a specific task, Netlist generation for a two stage operational amplifier.

II. Literature Review

Language modeling has been a pivotal area of research since the 1950s, starting with Shannon’s application of information theory to human language. Shannon's early work focused on how well simple n-gram models could predict or compress natural language text. Since then, statistical language modeling has become fundamental to various natural language understanding and generation tasks, including speech recognition, machine translation, and information retrieval.

The recent surge in transformer-based large language models (LLMs), pretrained on vast Web-scale text corpora, has significantly advanced the capabilities of language models. Examples like OpenAI’s ChatGPT and GPT-4 showcase not only natural language processing but also general task-solving abilities, which power systems like Microsoft’s Co-Pilot. These advancements position LLMs as essential building blocks for developing general-purpose AI agents or artificial general intelligence (AGI).

A. Development of LLMs

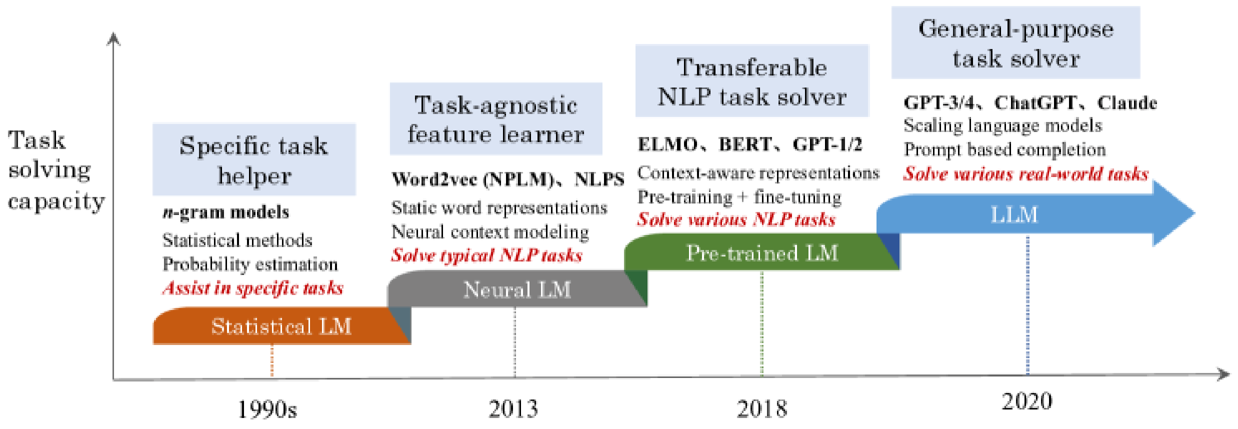


Fig. 1: An evolution process of the four generations of language models (LM) from the perspective of task solving capacity. Note that the time period for each stage may not be very accurate, and we set the time mainly according to the publish date of the most representative studies at each stage. [4]

Statistical language models (SLMs) treat text as a sequence of words and estimate the probability of text as the product of individual word probabilities. The dominant form of SLMs are n-gram models, which compute the probability of a word based on its preceding words. However, these models struggle with data sparsity, which requires smoothing techniques to assign probabilities to unseen n-grams. Despite their widespread use in NLP systems, SLMs cannot fully capture the complexity of natural language.

Early neural language models (NLMs) addressed data sparsity by mapping words to low-dimensional continuous vectors (embeddings) and predicting the next word based on the context provided by preceding word embeddings using neural networks. These models, however, were task-specific, trained on specific data sets, and did not generalize well across different tasks.

Pre-trained language models (PLMs), unlike early NLMs, are task-agnostic and follow a pre-training and fine-tuning paradigm. Models such as those based on recurrent neural networks or transformers are pre-trained on extensive unlabeled text corpora for general tasks and then fine-tuned for specific tasks with smaller labeled data sets. This approach has led to significant improvements in NLP tasks.

LLMs, characterized by their vast number of parameters (ranging from tens to hundreds of billions) and training on massive text data, represent a leap forward from PLMs. Models such as PaLM, LLaMA, and GPT-4 exhibit stronger language understanding and generation capabilities and possess emergent abilities like in-context learning, instruction following, and multi-step reasoning. These abilities enable LLMs to be used as AI agents, interacting with users and environments effectively, and continually improving through feedback mechanisms like reinforcement learning with human feedback (RLHF).

B. How LLMs are built

In this section, we first review the popular architectures used for LLMs and then discuss data and modeling techniques ranging from data preparation, and tokenization, to pre-training, instruction tuning, and alignment. Once the model architecture is chosen, the major steps involved in training an LLM include: data preparation (collection, cleaning, deduplication, etc.), tokenization, model pre-training (in a self- supervised learning fashion), instruction tuning, and alignment. We will explain each of them in a separate subsection below. These steps are also illustrated in Figure 2.

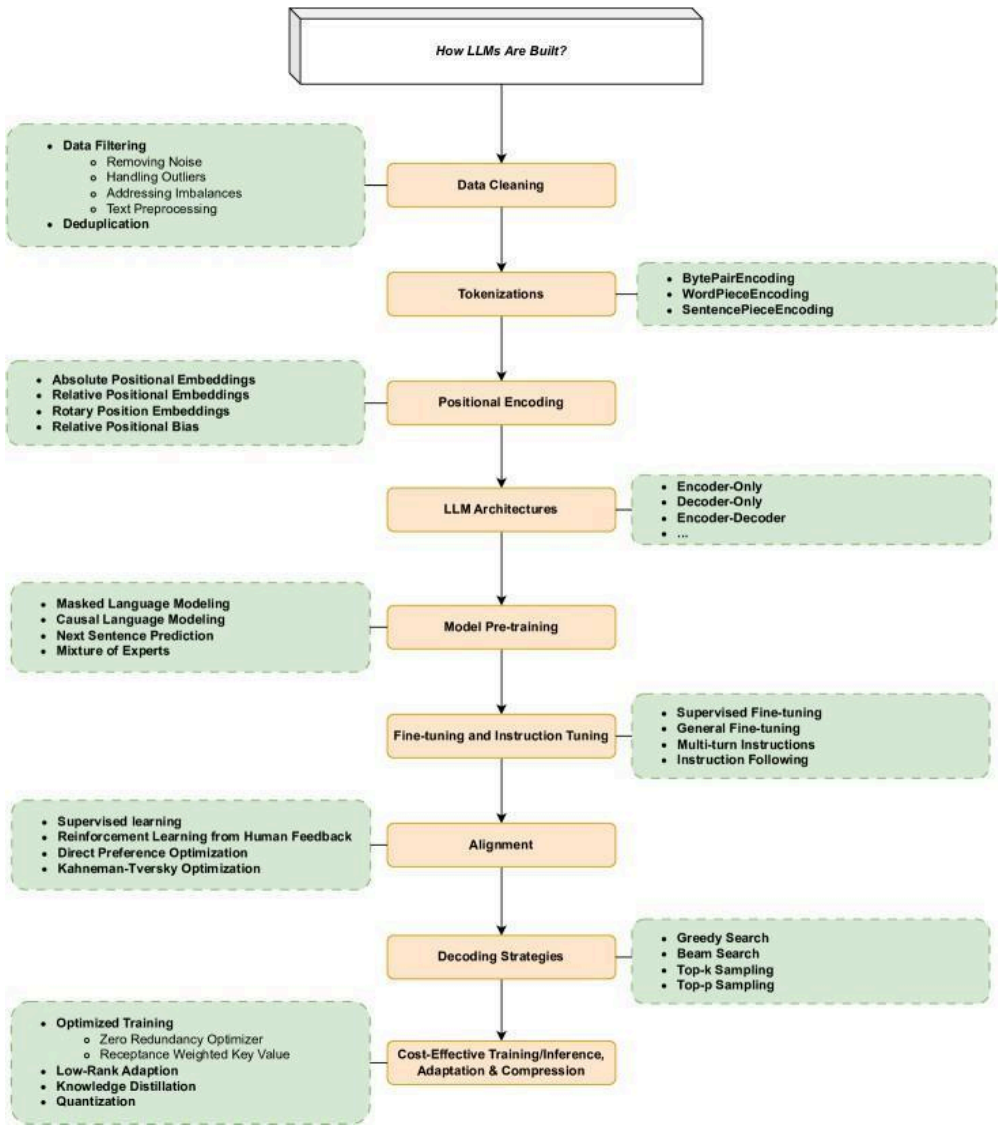


Fig. 2: This figure shows different components of LLMs.

Data Preparation

Data preparation is a critical step in building LLMs. It involves collecting large volumes of text data from various sources, cleaning it to remove noise and irrelevant information, and deduplicating it to ensure the quality of the dataset. This step ensures that the model is trained on diverse and representative data, improving its generalization capabilities.

Tokenization

Tokenization is the process of breaking down text into smaller units, such as words or subwords, that the model can process. Techniques like Byte Pair Encoding (BPE) and WordPiece are commonly used to achieve efficient tokenization. Proper tokenization ensures that the model can handle various linguistic phenomena, such as compound words and out-of-vocabulary terms, effectively.

Model Pre-Training

Model pre-training involves training the LLM on a large corpus of text in a self-supervised manner. This means that the model learns to predict missing or masked tokens in the text, enabling it to understand the language structure and context. Pre-training provides the model with a strong foundational understanding of language, which can be fine-tuned for specific tasks.

Instruction Tuning

Instruction tuning involves fine-tuning the pre-trained model on specific tasks using task-specific instructions. This step adapts the general language understanding of the model to the particular requirements of the target tasks. Techniques like supervised fine-tuning and reinforcement learning can be used to optimize the model's performance on these tasks.

Alignment

Alignment ensures that the model's outputs are aligned with human expectations and values. This step involves techniques like human-in-the-loop training and feedback loops, where human feedback is used to guide the model's behavior. Alignment is crucial for building safe and trustworthy AI systems that can interact effectively with users.

C. Dominant LLM Architectures

The most widely used LLM architectures are encoder-only, decoder-only, and encoder-decoder. Most of them are based on the Transformer architecture (as the building block). Therefore, we also review the Transformer architecture here.

1. Encoder-Only Architectures

Encoder-only models, such as BERT (Bidirectional Encoder Representations from Transformers), focus on understanding and generating representations of input text. These models are particularly effective for tasks that require a deep understanding of context, such as text classification and named entity recognition. The encoder reads the entire input text and creates a context-aware representation of each token. [6]

2. Decoder-Only Architectures

Decoder-only models, such as GPT (Generative Pre-trained Transformer), are designed for text generation tasks. They generate text one token at a time, predicting the next token based on the previous tokens. This unidirectional approach allows the model to excel in tasks like language modeling, text completion, and creative writing.[7]

3. Encoder-Decoder Architectures

Encoder-decoder models, such as T5 (Text-To-Text Transfer Transformer), use an encoder to understand the input text and a decoder to generate the output text. This architecture is highly versatile and can be used for various tasks, including translation, summarization, and question-answering. The encoder processes the input text into context-aware representations, which the decoder then uses to generate the desired output.[8]

4. Transformer Architecture

The Transformer architecture, introduced by Vaswani et al. (2017), serves as the foundation for most modern LLMs. It uses self-attention mechanisms to weigh the importance of different words in a sentence, allowing it to capture long-range dependencies and relationships in text. This architecture enables efficient parallelization during training and inference, making it suitable for large-scale language models.[9]

Transformer-based models have revolutionized natural language processing tasks by providing a robust architecture for understanding and generating text. The typical components of these models include:

Input Embeddings: The input text is tokenized into smaller units, such as words or sub-words. Each token is embedded into a continuous vector representation, capturing the semantic and syntactic information of the input.

Positional Encoding: To provide information about the token positions (as transformers do not inherently encode order), positional encodings are added to the input embeddings. This enables the model to consider the sequential order of tokens during processing.

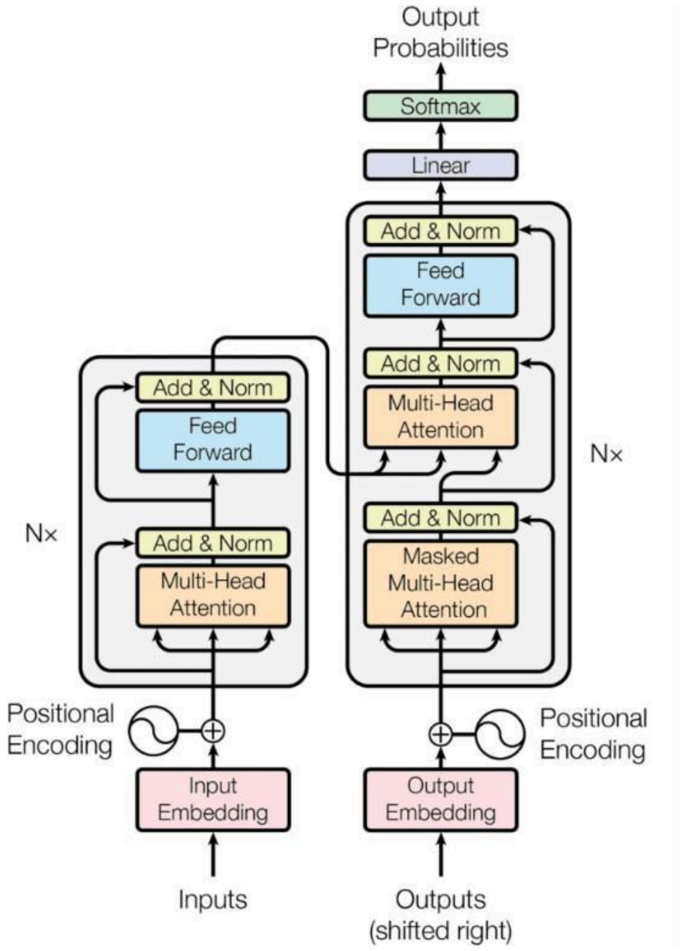


Fig. 3: Transformer based LLM Architecture.

Encoder: The encoder analyzes the input text and creates several hidden states that capture the context and meaning of the text data. The core of the transformer architecture consists of multiple encoder layers. Each encoder layer has two fundamental sub-components:

- **Self-Attention Mechanism:** This mechanism allows the model to weigh the importance of different tokens in the input sequence by computing attention scores. It enables the model to consider dependencies and relationships between tokens in a context-aware manner.
- **Feed-Forward Neural Network:** After self-attention, a feed-forward neural network is applied independently to each token. This network, consisting of fully connected layers with non-linear activation functions, captures complex interactions between tokens.

Decoder Layers: In some transformer-based models, a decoder component is included alongside the encoder. Decoder layers enable autoregressive generation, where the model generates sequential outputs by attending to previously generated tokens.

Multi-Head Attention: Transformers often use multi-head attention, performing self-attention simultaneously with different learned attention weights. This allows the model to capture various types of relationships and attend to different parts of the input sequence concurrently.

Layer Normalization: Applied after each sub-component or layer, layer normalization stabilizes the learning process and improves the model's generalization across different inputs.

Output Layers: The output layers vary depending on the specific task. For example, in language modeling, a linear projection followed by a SoftMax activation generates the probability distribution over the next token.

Enhancements and Variations

The actual architecture of transformer-based models can be modified and enhanced based on specific research and model development goals. Various models, such as GPT, BERT, and T5, integrate additional components or modifications to fulfill different tasks and objectives. For instance: GPT (Generative Pre-trained Transformer): Utilizes a decoder-only architecture focused on text generation tasks. BERT (Bidirectional Encoder Representations from Transformers): Uses an encoder-only architecture designed for understanding text and context in a bidirectional manner. T5 (Text-to-Text Transfer Transformer): Employs an encoder-decoder architecture for converting various text-based tasks into a unified framework.

The flexibility and effectiveness of transformer architectures have led to their widespread adoption and continuous evolution in the field of natural language processing.

III. LLM Fine-Tuning

Large Language Model (LLM) Fine-Tuning is the process of taking pre-trained models and further training them on smaller, specialized datasets to refine their capabilities and improve performance in specific tasks or domains. This process bridges the gap between generic pre-trained models and the unique requirements of particular applications, ensuring that the model aligns closely with human expectations. For instance, OpenAI's GPT-3, a state-of-the-art model designed for a broad range of natural language processing (NLP) tasks, might need fine-tuning to handle domain-specific language, such as medical terminology for healthcare applications.

Fine-tuning enhances the model's ability to perform specific tasks by adapting it to the nuances and specialized knowledge required for those tasks. While pre-trained models have broad general knowledge, fine-tuning helps them excel in particular areas by adjusting their parameters based on a task-specific dataset. This process helps tailor the model to meet the unique demands of the target application, improving accuracy and relevance.

A. When to Use Fine-Tuning

Fine-tuning is often considered when:

In-Context Learning: This method improves the prompt by including examples within it, guiding the model on what it needs to accomplish. While effective for some tasks, in-context learning may not always work well, especially for smaller LLMs. Moreover, examples included in the prompt can consume valuable context window space, limiting the amount of additional information that can be provided.

Zero-Shot, One-Shot, and Few-Shot Inference: These methods involve using the model without additional training examples (zero-shot), with a single example (one-shot), or with a few examples (few-shot) included in the prompt. If these approaches do not yield satisfactory results, fine-tuning might be employed to provide more targeted performance improvements.

Supervised Fine-Tuning (SFT)

Supervised Fine-Tuning involves updating a pre-trained model using labeled data to perform a specific task. This contrasts with unsupervised learning, where models are trained without explicit labels. Supervised fine-tuning is typically performed using labeled prompt-response pairs, which helps the model better complete specific tasks by learning from these examples.

B. How Fine-Tuning Is Performed

Data Preparation: Fine-tuning begins with preparing a dataset of labeled examples, which may involve converting open-source data into instructional prompts. The dataset is divided into training, validation, and test splits.

Training Process: During fine-tuning, the model processes prompts from the training dataset and generates responses. The model's predictions are compared with the actual labels to calculate error. Optimization algorithms, such as gradient descent, are used to adjust the model's weights based on these errors. Over multiple training iterations (epochs), the model's weights are updated to minimize the error and enhance performance on the specific task. Example: For a science educational chatbot, fine-tuning a model to provide detailed explanations might involve adjusting it to respond with comprehensive information on scientific topics. For instance, after fine-tuning, a model might provide a detailed explanation of why the sky is blue rather than a brief response.

Methods for Fine-Tuning LLMs

Instruction Fine-Tuning: This involves training the model using examples that demonstrate how it should respond to specific instructions. This method allows the model to perform tasks like summarization or translation more effectively by learning from prompt-response pairs tailored to those tasks.

Full Fine-Tuning: This method updates all the model's weights, resulting in a new version of the model with improved performance on specific tasks. However, it requires significant computational resources and memory.

Parameter-Efficient Fine-Tuning (PEFT):

PEFT updates only a small subset of parameters, making the fine-tuning process more efficient. Techniques like LoRA (Low-Rank Adaptation) freeze the majority of the model's parameters while adapting only a small number of them, reducing memory and computational requirements.

To set the stage for LoRA, consider this: if we only update ΔW without modifying the original pre-trained weights W (which remain frozen or have req_grads = False), we can save a lot of GPU memory. Moreover, if these updated weights are just a small fraction of the original W, the savings are even greater. Pre-trained models have a very low intrinsic dimension, or low rank, which means they can be represented with matrices of much lower dimensions. The rank of a matrix represents the number of linearly independent rows or columns it has.

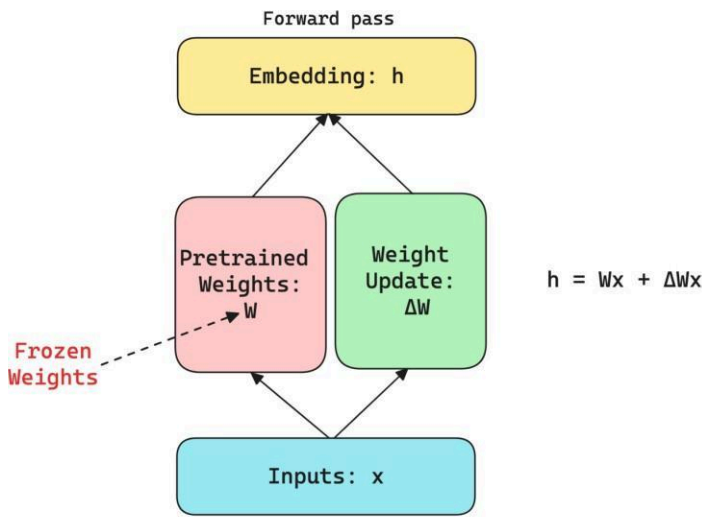


Fig. 4: Representation of Fine-Tuning process.

During adaptation or fine-tuning, ΔW (of dimension A×B) also has a low rank and can be decomposed into two matrices, WA (of dimension A×r) and WB (of dimension r×B), where r is the rank of matrix ΔW.

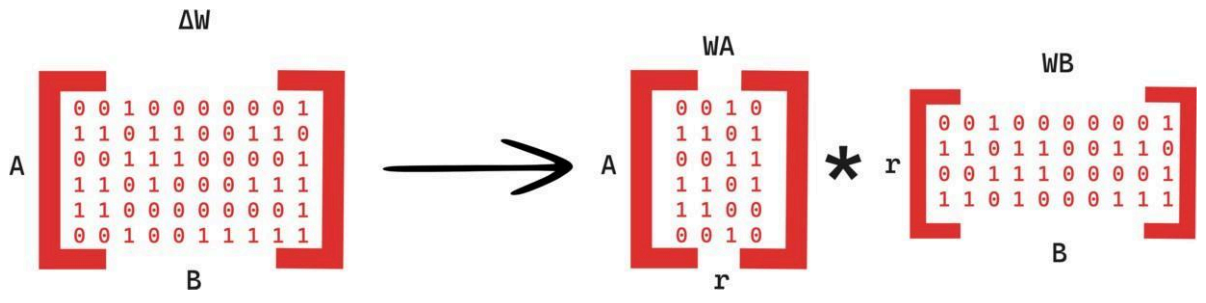


Fig. 5: Low Rank Decomposition.

For example, if ΔW has a dimension of 100×100 and a rank of just 5, it can be decomposed into two matrices with dimensions 100×5 and 5×100. This effectively reduces the number of trainable parameters from 10,000 to just 1,000. In this setup, r becomes a hyper-parameter that needs to be tuned during training. The key point is that WA and WB can effectively represent ΔW while having a significantly lower rank and therefore fewer parameters to fine-tune.

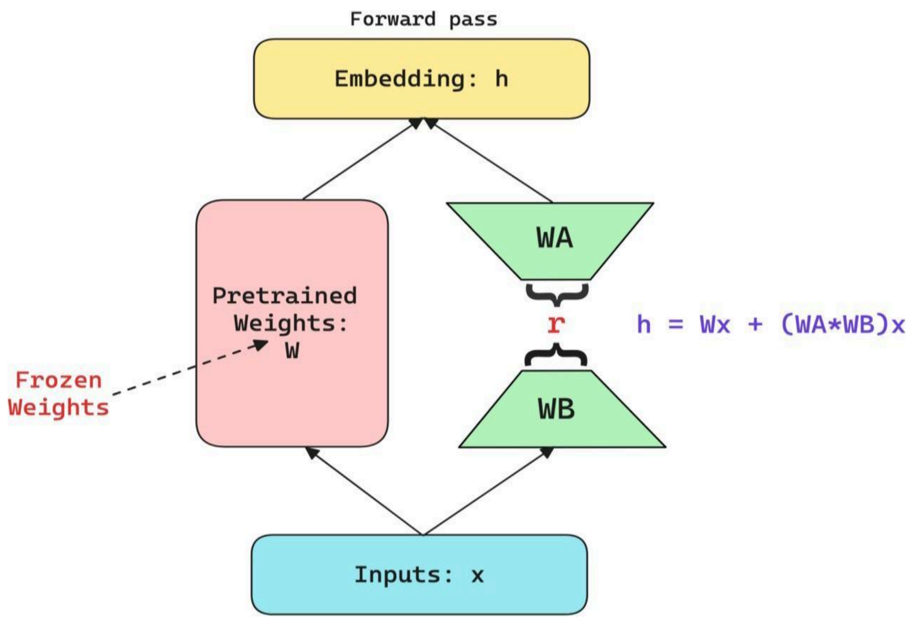


Fig. 6: Diagram of LoRA

Transfer Learning: This method involves adapting a pre-trained model to a specific task using domain-specific data. It offers higher learning rates and accuracy by leveraging the general knowledge acquired during the initial training.

Task-Specific Fine-Tuning: This approach involves training the model on a dataset tailored to a particular task or domain, leading to improved performance for that specific task.

Multi-Task Learning: The model is trained on a dataset containing examples for multiple tasks. This approach helps avoid catastrophic forgetting, where performance on one task deteriorates as the model learns another task.

Sequential Fine-Tuning: This method involves fine-tuning the model on a sequence of related tasks. For example, a model might be first adapted to general language and then fine-tuned for medical language before focusing on a more specialized domain like pediatric cardiology.

Fine-tuning allows LLMs to adapt from general-purpose models to specialized tools, enhancing their performance for specific applications while managing computational and storage resources efficiently.

C. Leveraging Hugging Face for Fine-tuning

Hugging Face has become a cornerstone in the natural language processing (NLP) community, providing a rich set of tools and models that facilitate the fine-tuning process for various language models. Their platform offers access to numerous pre-trained language models, such as BERT, GPT-3, and RoBERTa, which can be adapted for specific tasks with minimal computational resources and data.

Tools and Features

Hugging Face's ecosystem includes the widely used Transformers library, which supports thousands of pre- trained models designed for tasks like text classification, question answering, and text generation. The library simplifies the process of leveraging state-of-the-art models by providing easy-to-use APIs for model loading, fine-tuning, and inference.

Fine-Tuning with Hugging Face

The process of fine-tuning models using Hugging Face's tools is straightforward and efficient. Developers can use the Trainer API, which abstracts the complexities of the training loop and allows customization to meet specific needs. This API facilitates seamless integration with various datasets and optimizes the training process, making it easier to achieve high performance on domain-specific tasks.

Incorporating Hugging Face into the workflow for fine-tuning LLMs enhances productivity and ensures access to cutting-edge NLP capabilities. By leveraging these tools, developers can focus on optimizing their models for specific applications without the need to build and maintain complex training pipelines from scratch.

IV. Retrieval-Augmented Generation (RAG)

One of the main limitations of pre-trained LLMs is their lack of up-to-date knowledge or access to private or use-case-specific information. This is where Retrieval-Augmented Generation (RAG) comes into the picture . RAG, illustrated in Figure 7, involves extracting a query from the input prompt and using that query to retrieve relevant information from an external knowledge source. The relevant information is then added to the original prompt and fed to the LLM to generate the final response. A RAG system includes three important components: Retrieval, Generation, and Augmentation.[3]

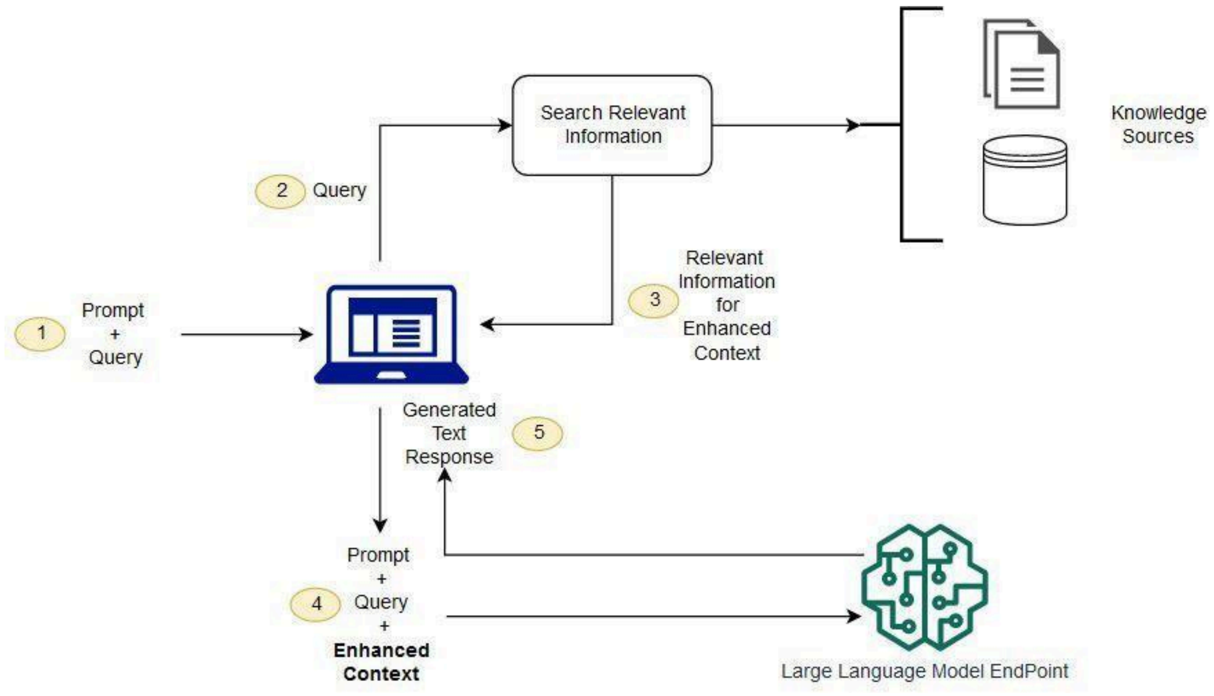

Search Relevant Information
Knowledge Sources
2 Query
1 Prompt + Query
3 Relevant Information for Enhanced Context
Generated Text Response 5
4 Prompt + Query + Enhanced Context
Large Language Model EndPoint

Fig. 7: An example of synthesizing RAG with LLMs for question answering application.

In the retrieval step, the system extracts a query from the input prompt. This query is used to search for relevant information from an external knowledge source, such as a search engine, database, or knowledge graph. The retrieval process ensures that the model has access to the most recent and pertinent information, overcoming the static nature of pre-trained LLMs.

After the relevant information is retrieved, it is combined with the original input prompt. The combined prompt is then fed into the LLM. The generation component leverages the LLM's capabilities to process the enriched prompt and produce a response. This response is more informed and contextually relevant because it includes up-to-date or specialized information obtained during the retrieval phase.

The augmentation step involves seamlessly integrating the retrieved information into the original prompt in a manner that enhances the LLM's ability to generate accurate and relevant responses. This step is crucial as it ensures that the additional information is presented in a way that the LLM can effectively utilize, thus improving the overall quality and coherence of the generated response.

By incorporating RAG, LLMs can overcome their inherent limitations of static knowledge and provide responses that are more accurate, relevant, and tailored to specific use cases. This approach enhances the usability of LLMs in various applications, particularly those requiring current information or specialized knowledge.

A. Fine-Tuning Vs. RAG

LLMs, pre-trained on extensive datasets, possess significant generative and analytical capabilities but are limited by their static knowledge base. RAG addresses this limitation by retrieving relevant information from external sources to provide context for the LLM, enhancing its responses with up-to-date and specific information.

RAG offers several advantages over fine-tuning. It preserves the general capabilities of pre-trained LLMs, which fine-tuning might erode. Additionally, RAG requires less domain-specific training data as it relies on external knowledge sources, making it more adaptable and customizable without the need for retraining. This flexibility allows RAG systems to quickly incorporate new information and adapt to changing contexts, whereas fine-tuning can lead to forgetting previously learned capabilities and is dependent on the quality and quantity of the training data.

The choice between RAG and fine-tuning depends on the model size and the specific task. For large models like GPT-4, RAG is generally preferred because it retains the broad capabilities of the model while integrating external knowledge. Medium-sized models can benefit from either approach depending on the task's nature; RAG is useful for domain-specific generation, while fine-tuning is better for tasks requiring memorization. For small models, fine-tuning is typically more effective, directly embedding the necessary knowledge with minimal risk of forgetting.

Practical considerations for choosing between RAG and fine-tuning include access to LLMs, training infrastructure, data availability, and the need for dynamic versus static knowledge. RAG can introduce latency due to the retrieval process, while fine-tuned models are faster at inference since they are self-contained. Knowledge flexibility is another factor; RAG supports frequent updates without retraining, whereas fine-tuning creates static models that need periodic updates.

The article also discusses advanced perspectives, including hybrid approaches that combine the strengths of RAG and fine-tuning. Dynamic versus static models, hyper-personalization, catastrophic forgetting mitigation, and on-device deployment are all areas where RAG and fine-tuning offer different advantages. Ultimately, blending these techniques can maximize performance, balancing conversational competence and domain expertise.

The key recommendations suggest using RAG for applications that need to maintain broad conversational abilities and resorting to fine-tuning for tasks that require embedding specific knowledge. Document and text analysis tasks often benefit more from fine-tuning, except in rapidly evolving areas where RAG might be advantageous. Carefully selecting knowledge sources for RAG and considering ensemble approaches that combine specialized fine-tuned models with general RAG models can optimize performance.

In conclusion, the choice between RAG and fine-tuning is complex and context dependent. Understanding the specific needs of the application, the available infrastructure, and the nature of the task can guide the decision. Blending RAG and fine-tuning approaches often yields the best results, allowing companies to leverage the strengths of both methods to create AI systems that are both knowledgeable and flexible. [10]

V. IC Automation

The objective is to utilize a Large Language Model (LLM) to assist in designing and optimizing a two-stage amplifier using TSMC 65nm CMOS technology, as shown in Fig. 8. This involves training the LLM on a dataset of approximately 8800 data points, each containing various configurations of transistor sizing, component values, and biasing voltages along with corresponding performance metrics such as gain, bandwidth, power consumption, phase margin, and gain margin. This process aims to streamline and automate the design and fine-tuning of the amplifier circuit, ensuring efficient and accurate performance optimization.

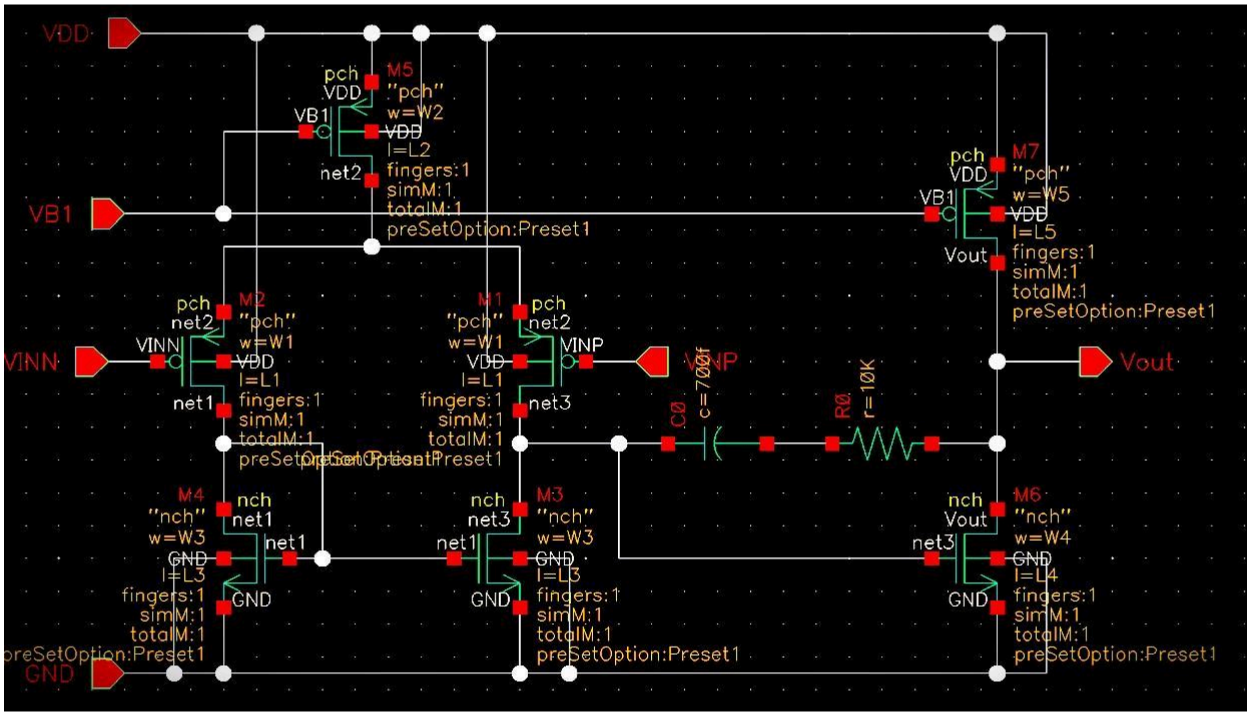


Fig. 8: Schematic of two stage operational amplifier circuit.

We achieve our goal by feeding following specifications to the model:

Determine the resistor and capacitor values, the sizing of MOSFET transistors, and the biasing voltages (i.e., R0, C0, L1, W1, L2, W2, L3, W3, L4, W4, L5, W5, VIN (for both VINP and VINN), VB1) for the provided two-stage amplifier using TSMC 65nm CMOS technology to achieve the specified amplifier performance metrics. The amplifier consists of 5 MOSFET transistors in the first stage and 2 MOSFET transistors in the second stage, along with a capacitor and a resistor as a feedback connection (Miller) at the second stage NMOS transistor (M6).

First Stage: Input Transistors: M1, M2 (PMOS): Differential input transistors with the same sizing (L1, W1). Differential input signal applied.

Current Mirror & Load: M3, M4 (NMOS): Current mirror and load transistors with the same sizing (L3, W3).

Biasing Transistor: M5 (PMOS): Current source transistor with sizing (L2, W2) and gate voltage (VB1).

Second Stage: Biasing Transistor: M7 (PMOS): Biasing transistor with sizing (L5, W5) and gate voltage (VB1).

Amplifying Transistor: M6 (NMOS): Connected with a resistor (R0) in series with a capacitor (C0) as feedback. Sizing (W4, L4) controls the gain of the second stage.

Netlist: MOSFET Transistors:

```
M1 pch net3 VINP net2 VDD L1 W1

M2  pch  net1  VINN  net2  VDD  L1

W1 M5 pch net2 VB1 VDD VDD L2

W2

M3 nch net3 net1 GND GND L3 W3

M4 nch net1 net1 GND GND L3 W3

M6 nch VOUT net3 GND GND L4 W4

M7 pch VOUT VB1 VDD VDD L5

W5
```

Capacitor: C0 net3 net4 C0_value

Resistor: R0 net4 VOUT

R0_value Netlist Column

Explanation:

Component Name (M for MOSFET, C for Capacitor, R for Resistor)

For transistors:Type (pch for PMOS, nch for NMOS), Drain Net Name, Gate Net Name, Source Net Name, Bulk Net Name, Length (L) of the Transistor, Width (W) of the Transistor.

For resistor and capacitor: One terminal net name, Other terminal net name, Capacitor or Resistor Value

Biasing Voltages: VINP, VINN, VB1.

VI. Fine-tuning LLMs using Azure services.

For the fine-tuning of the base model, we'll utilize Azure AI Studio due to its comprehensive suite of tools and features. Azure AI Studio provides a user-friendly interface that simplifies the model training process, offering advanced capabilities such as automated machine learning and efficient data preparation. The integration with Azure's scalable cloud resources ensures access to powerful hardware, including GPUs and TPUs, which accelerates training. Additionally, Azure AI Studio's robust security measures and cost management tools contribute to a secure and cost-effective training environment. Overall, Azure AI Studio's capabilities facilitate a streamlined and effective fine-tuning process, optimizing model performance and deployment.

Fine-Tuning steps:

Prerequisites:

Here's a summary of the prerequisites and permissions required for fine-tuning with Azure:

To run a fine-tuning job on Azure, you need an Azure subscription with a valid payment method, as free or trial subscriptions are not eligible. If you don't have one, you must create a paid Azure account. Create a resource group that holds related resources for an Azure solution.

In case of Azure OpenAI, the best region to choose would be North Central US or Sweden Central.

Additionally, an Azure AI hub resource is necessary, and for Meta Llama 3.1 models, the pay-as-you-go fine-tune offering is only available with AI hubs in the West US 3 region.

You also need an Azure AI project set up in Azure AI Studio. Access to Azure AI Studio is controlled through Azure role-based access controls (RBAC). Your user account must have the owner or contributor role for the Azure subscription or a custom role with specific permissions.

We are all set to start the finetuning process now.

1. **Choose base model:**

The first step in creating a custom model is to choose a base model. The Base model pane lets you choose a base model to use for your custom model. Your choice influences both the performance and the cost of your model.

In case of Azure ML Studio, you need to create an AML compute cluster. Under the Create compute cluster pane, specify the Location (e.g., West Europe), set the Virtual Machine Tier to Dedicated, choose GPU for the Virtual Machine Type, and select either the NVIDIA ND40 or ND96 VM sizes, which are currently supported for fine-tuning. If these VM sizes are not available, consider selecting a different location or requesting additional quota. Assign a name to your compute cluster and set the minimum number of nodes (typically 0) and the maximum (usually 1 for testing purposes). Click Next to start the creation process, which may take a few minutes to complete.

2. **Training data:**

The training data and validation data sets consist of input and output examples for how you would like the model to perform. Different model types require a different format of training data.

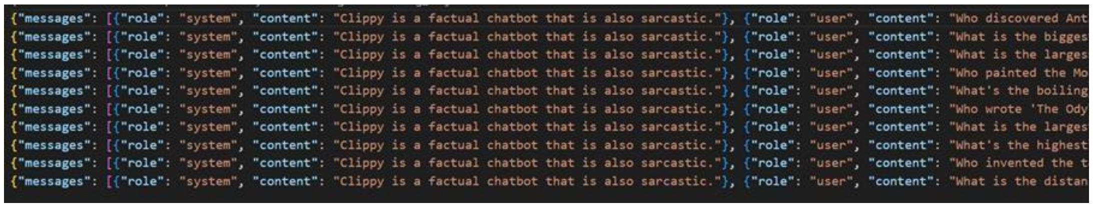
For OpenAI completion models (Babbage-002/Davinci-002):

The training and validation data must be formatted as JSON Lines (JSONL), where each line represents a single prompt-completion pair. The OpenAI command-line interface (CLI) provides a data preparation tool that validates, suggests improvements, and reformats your data into a JSONL file suitable for fine-tuning.

```
{"prompt": "<prompt text>", "completion":" <ideal generated text>"}
{"prompt": "<prompt text>", "completion":" <ideal generated text>"}
{"prompt": "<prompt text>", "completion":" <ideal generated text>"}
```

For OpenAI chat completion models like GPT-3.5:

The training and validation data you use must be formatted as a JSON Lines (JSONL) document. For gpt-35-turbo-0613 the fine-tuning dataset must be formatted in the conversational format that is used by the chat completion API.



```
{"messages": [{"role": "system", "content": "Clippy is a factual chatbot that is also sarcastic."}, {"role": "user", "content": "Who discovered Ant
{"messages": [{"role": "system", "content": "Clippy is a factual chatbot that is also sarcastic."}, {"role": "user", "content": "What is the bigges
{"messages": [{"role": "system", "content": "Clippy is a factual chatbot that is also sarcastic."}, {"role": "user", "content": "What is the larges
{"messages": [{"role": "system", "content": "Clippy is a factual chatbot that is also sarcastic."}, {"role": "user", "content": "Who painted the Mo
{"messages": [{"role": "system", "content": "Clippy is a factual chatbot that is also sarcastic."}, {"role": "user", "content": "What's the boiling
{"messages": [{"role": "system", "content": "Clippy is a factual chatbot that is also sarcastic."}, {"role": "user", "content": "Who wrote 'The Ody
{"messages": [{"role": "system", "content": "Clippy is a factual chatbot that is also sarcastic."}, {"role": "user", "content": "What is the larges
{"messages": [{"role": "system", "content": "Clippy is a factual chatbot that is also sarcastic."}, {"role": "user", "content": "What's the highest
{"messages": [{"role": "system", "content": "Clippy is a factual chatbot that is also sarcastic."}, {"role": "user", "content": "Who invented the t
{"messages": [{"role": "system", "content": "Clippy is a factual chatbot that is also sarcastic."}, {"role": "user", "content": "What is the distan
```

Fig. 9: Example Training dataset formatted to use by Chat completions API

The files must be encoded in UTF-8 with a byte-order mark (BOM) and should not exceed 512 MB in size.

In case we are fine-tuning models on Azure Machine Learning studios, we'll have models from Meta, Hugging Face and a few curated by Azure ML. Depending on type of task the model is finetuned for, following is required the dataset format:

Text classification	Two columns: Sentence (string) and Label (integer/string)
Token classification	Two columns: Token (string) and Tag (string)
Question answering	Fivecolumns: Question (string), Context (string), Answers (string), Answers_start (int), and Answers_text (string)
Summarization	Two columns: Document (string) and Summary (string)
Translation	Two columns: Source_language (string) and Target_language (string)
Text classification	Two columns: Sentence (string) and Label (integer/string)

Table 1: Dataset format based on type of task the model is fine-tuned for.

Now, that the dataset is ready, optionally choose your validation data. Else, the training data will be automatically split into training and validate data while fine-tuning run.

3. Optionally**, configure advanced options** for your fine-tuning job**:**

The Create custom model wizard shows the parameters for training your fine-tuned model on the Advanced options pane. The following parameters are available:

Name	**Type**	**Description**
batch_size	integer	The batch size to use for training. The batch size is the number of training examples used to train a single forward and backward pass. In general, we've found that larger batch sizes tend to work better for larger datasets. The default value as well as the maximum value for this property are specific to a base model. A larger batch size means that model parameters are updated less frequently, but with lower variance.

Name	**Type**	**Description**
learning_rate_multiplier	number	The learning rate multiplier to use for training. The fine-tuning learning rate is the original learning rate used for pre-training multiplied by this value. Larger learning rates tend to perform better with larger batch sizes. We recommend experimenting with values in the range 0.02 to 0.2 to see what produces the best results. A smaller learning rate may be useful to avoid overfitting.
n_epochs	integer	The number of epochs to train the model for. An epoch refers to one full cycle through the training dataset.
seed	integer	The seed controls the reproducibility of the job. Passing in the same seed and job parameters should produce the same results, but may differ in rare cases. If a seed isn't specified, one will be generated for you

Table 2: Fine-tuning parameters

Review your choices and submit your training job for your new custom model.

4. **Check the status of your custom fine-tuned model.**

After you start a fine-tuning job, it can take some time to complete (from minutes to hours).

5. **Deploy your custom model for use.**

When the fine-tuning job succeeds, you can deploy the custom model from the Models pane to make it available for use with completion calls.

In case you are using, Azure ML studio, before deploying the model, you need to register the model first. After that you need to specify the Virtual machine (preferably choose Nvidia NC & ND VM series), Instance count, Endpoint name and Deployment name. Deployment might take a moment.

6. **Use your custom model.**

After your custom model deploys, you can use it like any other deployed model. You can use the Playgrounds in Azure OpenAI Studio to experiment with your new deployment. You can also consume the API using a popular programming language such as Python.

7. Optionally, **analyze your custom model** for performance and fit**.**

You can directly test the deployed model via the handy test playground, check the performance metrics and graphs.

8. **Clean up your deployment resources:**

When you're done with your custom model, you can delete the deployed endpoint, model, and the compute cluster so that no further cost is incurred. You can also delete the training (and validation and test) files you uploaded to the service, if needed.

VII. Results

We finetuned Llama 2 7b model on our IC automation dataset using Azure AI studio which took 1 h12 mins to train, using the batch size multiplier =1, epochs =1, learning rate =0.0003

Following are the fine-tuned model's performance metrics:

Eval_loss	0.1296790
Eval_runtime	91.3513
Eval_samples_per_second	19.135
Eval_steps_per_second	2.397
Loss	0.1207
Total_flos	209,039,742,839,424
Train_loss	0.1206603
Train_runtime	1178.728
Train_samples_per_second	5.932
Train_steps_per_second	0.741

Table 3: Performance metrics of fine-tuned model

Training and Evaluation Metrics:

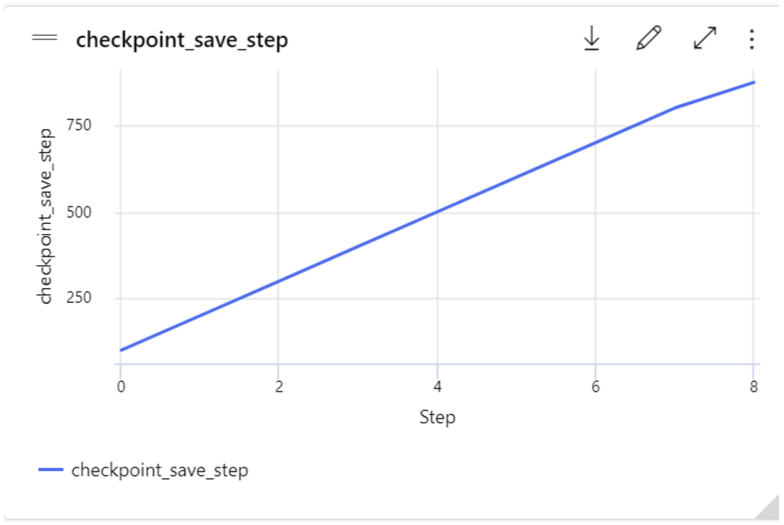
The low loss values suggest that the model has learned well from the training data and can generalize to some extent.

Runtime and Throughput:

The throughput is adequate, indicating efficient processing during both training and evaluation.

Computational Workload:

The model has performed significant computational work, which is expected for a large-scale fine-tuning task.



Graph 1: A line chart of the checkpoint_save_step over time.

The graph illustrates the consistency of checkpoint saves during the fine-tuning process, ensuring that model states are regularly saved for potential recovery and analysis.

We also gave a prompt to our fine-tunned model for netlist generation:

Determine the resistor and capacitor values, the sizing of MOSFET transistors, and the biasing voltages (i.e., R0, C0, L1, W1, L2, W2, L3, W3, L4, W4, L5, W5, VIN (for both VINP and VINN), VB1) for the provided two-stage amplifier using TSMC 65nm CMOS technology to achieve the specified amplifier performance metrics. The amplifier consists of 5 MOSFET transistors in the first stage and 2 MOSFET transistors in the second stage, along with a capacitor and a resistor as a feedback connection (Miller) at the second stage NMOS transistor (M6).

First Stage: Input Transistors: M1, M2 (PMOS): Differential input transistors with the same sizing (L1, W1). Differential input signal applied.

Current Mirror & Load: M3, M4 (NMOS): Current mirror and load transistors with the same sizing (L3, W3).

Biasing Transistor: M5 (PMOS): Current source transistor with sizing (L2, W2) and gate voltage (VB1).

Second Stage: Biasing Transistor: M7 (PMOS): Biasing transistor with sizing (L5, W5) and gate voltage (VB1).

Amplifying Transistor: M6 (NMOS): Connected with a resistor (R0) in series with a capacitor (C0) as feedback. Sizing (W4, L4) controls the gain of the second stage.

Netlist: MOSFET Transistors:

```
M1 pch net3 VINP net2 VDD L1 W1

M2 pch net1 VINN net2 VDD L1 W1
```

M5 pch net2 VB1 VDD VDD L2 W2

M3 nch net3 net1 GND GND L3 W3

M4 nch net1 net1 GND GND L3 W3

M6 nch VOUT net3 GND GND L4

W4 M7 pch VOUT VB1 VDD VDD L5

W5

Capacitor:

C0 net3 net4 C0_value

Resistor:

R0 net4 VOUT R0_value

Netlist Column Explanation:

Component Name (M for MOSFET, C for Capacitor, R for Resistor)

For transistors:Type (pch for PMOS, nch for NMOS), Drain Net Name, Gate Net Name, Source Net Name, Bulk Net Name, Length (L) of the Transistor, Width (W) of the Transistor

For resistor and capacitor: One terminal net name, Other terminal net name, Capacitor or Resistor Value

Biasing Voltages: VINP, VINN, VB1.

Performance Metrics: Power=2.13852e-05W, Gain=38.70441dB, 3dB Bandwidth (BW_3dB)=153147.4Hz, Unity Gain Bandwidth (UGB)=10472090.0Hz, Phase Margin (PM)=52.6323degree, Gain Margin (GM)=13.0831dB

Following is the comparison of the expected output and the model's output.

Expected Output	**Fine-tuned model's output**
To achieve the given performance metrics with the provided amplifier netlist, you can use the following values in the netlist: M1 pch net3 VINP net2 VDD L1=622.0nm W1=15.0um M2 pch net1 VINN net2 VDD L1=622.0nm W1=15.0um M5 pch net2 VB1 VDD VDD L2=117.0nm W2=24.0um M3 nch net3 net1 GND GND L3=410.0nm W3=22.0um M4 nch net1 net1 GND GND L3=410.0nm W3=22.0um M6 nch VOUT net3 GND GND L4=491.0nm W4=64.0um M7 pch VOUT VB1 VDD VDD L5=220.0nm W5=24.0um C0 net3 net4 C0_value=1.0pF R0 net4 VOUT R0_value=3.0KOhm And with biasing voltages of VINP=VINN=VIN= 605.0mV, VB1=932.0mV.	To achieve the given performance metrics with the provided amplifier netlist, you can use the following values in the netlist: M1 pch net3 VINP net2 VDD L1=691.0nm W1=84.0um M2 pch net1 VINN net2 VDD L1=691.0nm W1=84.0um M5 pch net2 VB1 VDD VDD L2=333.0nm W2=73.0um M3 nch net3 net1 GND GND L3=543.0nm W3=14.0um M4 nch net1 net1 GND GND L3=543.0nm W3=14.0um M6 nch VOUT net3 GND GND L4=819.0nm W4=37.0um M7 pch VOUT VB1 VDD VDD L5=903.0nm W5=48.0um C0 net3 net4 C0_value=1.0pF R0 net4 VOUT R0_value=2.0KOhm And with biasing voltages of VINP=VINN=VIN= 584.0mV, VB1=827.0mV.

VIII. Conclusion

The fine-tuned Llama-2-7b model shows promising performance metrics but fails to produce the expected outputs accurately, indicating the need for further refinement. Several factors could contribute to this issue, including insufficient or non-representative training data, overfitting, suboptimal hyperparameters, inappropriate model complexity, or incorrect data preprocessing.

The fine-tuning process, completed in 1 hour and 12 minutes, demonstrated the model's potential but also highlighted areas for improvement. To address these issues, it is recommended to increase the training data, either by gathering more data or augmenting the existing dataset. Hyperparameter tuning should be performed to find the optimal settings for learning rate, batch size, and number of epochs. Regularization techniques, such as dropout, weight decay, or early stopping, can help prevent overfitting. Additionally, experimenting with different model architectures might better capture the patterns in the data. Ensuring proper data preprocessing and normalization is also crucial.

By optimizing the training process, adjusting hyperparameters, and ensuring high-quality data, the model's performance can be improved to better match the expected outcomes. Continuous iteration and validation are essential to achieving a model that meets the desired performance standards. While the current fine- tuning process of the Llama-2-7b model has demonstrated its potential, further efforts are necessary to refine its predictions and achieve the desired accuracy and reliability in real-world applications.

IX. Future Scope

The completion of this project opens several avenues for future work and improvements. Below are some potential areas for future exploration and development:

1. **Enhanced Model Performance**:

Investigate advanced fine-tuning techniques to further improve model accuracy and efficiency. Explore the use of more sophisticated optimization algorithms and hyperparameter tuning methods.

2. **Scalability and Deployment**:

Develop strategies for scaling the model to handle larger datasets and more complex tasks. Explore cloud-based deployment options to make the model accessible to a wider audience.

3. **Integration with Other Technologies**:

Investigate the integration of the model with other AI and machine learning technologies to enhance its capabilities. Explore potential applications in different domains such as healthcare, finance, and education.

4. **User Interface and Experience**:

Develop a more user-friendly interface to facilitate easier interaction with the model. Gather user feedback to continuously improve the user experience and address any issues.

5. **Ethical Considerations and Bias Mitigation**:

Conduct in-depth studies on the ethical implications of using such models, focusing on fairness, transparency, and accountability. Implement techniques to detect and mitigate biases in the model to ensure it is fair and unbiased.

By exploring these future directions, the project can continue to evolve and make a significant impact in the field of artificial intelligence and machine learning.

X. References:

1. J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford, J. Wu, and D. Amodei, “Scaling laws for neural language models,” arXiv preprint arXiv:2001.08361, 2020.J. Kaplan, S. McCandlish, T. Henighan, T. B. Brown, B. Chess, R. Child, S. Gray, A. Radford,

   J. Wu, and D. Amodei, “Scaling laws for neural language models,” arXiv preprint arXiv:2001.08361, 2020.
2. J. Hoffmann, S. Borgeaud, A. Mensch, E. Buchatskaya, T. Cai, E. Rutherford, D. d. L. Casas, L. A. Hendricks, J. Welbl, A. Clark et al., “Training compute-optimal large language models,” arXiv preprint arXiv:2203.15556, 2022.
3. Shervin Minaee, Tomas Mikolov, Narjes Nikzad, Meysam Chenaghlu, Richard Socher, Xavier Amatriain, Jianfeng Gao, “Large Language Models: A Survey” arXiv:2402.06196
4. Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie and Ji-Rong Wen, “A Survey of Large Language Models” arXiv:2303.18223
5. Humza Naveed1, Asad Ullah Khan, Shi Qiu,∗, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, Ajmal Mian, "A Comprehensive Overview of Large Language Models" arXiv:2307.06435
6. Devlin, J., Chang, M. W., Lee, K., & Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. arXiv preprint arXiv:1810.04805.
7. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language models are unsupervised multitask learners. OpenAI Blog, 1(8), 9.
8. Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., ... & Liu, P. J. (2020). Exploring the limits of transfer learning with a unified text-to-text transformer. Journal of Machine Learning Research, 21(140), 1-67.
9. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N. & Polosukhin, I. (2017). Attention is all you need. Advances in Neural Information Processing Systems, 30.
10. Ghosh, B. (2024, February 25). When to Apply RAG vs Fine-Tuning. Medium.Ghosh, B. (2024, February 25). When to Apply RAG vs Fine-Tuning. Medium.
11. https://learn.microsoft.com/en-us/azure/ai-studio/how-to/fine-tune-model-llama?tabs=llama-three%2Cchatcompletion