UNIVERSITÉ DU LUXEMBOURG

PhD-FSTM-2025-034
Faculty of Science, Technology and Medicine

DISSERTATION

Defence held on 24 March 2025 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG EN INFORMATIQUE

by

Karthick Panner Selvam
Born on 9th March 1995 in Chennai, Tamil Nadu (India)

Performance Prediction Models for Deep Learning: A Graph Neural Network and Large Language Model Approach

Dissertation Defense Committee

Abstract

In this thesis, we developed an advanced performance prediction model to estimate critical metrics of Deep Learning (DL) models, such as latency, memory consumption, and energy usage. These models are designed to support neural architecture search and efficient cloud deployment.

DL has transformed domains such as computer vision, natural language processing, climate modeling, and scientific computing. However, the increasing complexity of DL models introduces significant computational demands that require efficient hardware utilization and resource allocation. Accurate prediction of performance metrics is essential for optimizing hardware-specific compilers, enabling cost-effective cloud deployments, and minimizing environmental impacts.

To address these challenges, we present a comprehensive framework for performance prediction in this thesis. The work begins with a systematic benchmarking study of DL models, highlighting computational bottlenecks and establishing the necessity of performance prediction as a foundation for further development.

We introduce a Graph Neural Network (GNN) based performance prediction model capable of analyzing DL models from various software frameworks, including PyTorch and TensorFlow. This model predicts performance metrics and recommends NVIDIA multi-GPU instance profiles for efficient deployment. Building on this, we propose a semi-supervised performance prediction approach that leverages unlabeled data to accelerate training convergence. Using a graph autoencoder for unsupervised learning, we generate high-quality embeddings that enhance supervised training, leading to faster and more accurate predictions.

For Large Language Models (LLMs), which present unique challenges due to their extensive nodes and edges, we proposed a tree-based performance prediction model. This method significantly improves inference speed compared to traditional GNN-based techniques, making it particularly suitable for complex LLM architectures.

Finally, we explore multimodal learning by combining LLM with GNN to create a hybrid performance prediction model. This model quickly adapts to new hardware environments with sparse training samples, leveraging a novel three-stage training strategy to effectively integrate GNN and LLM for quick adaptation.

Acknowledgements

Completing a PhD is a challenging endeavor that would not have been possible without the immense support I received from many people.

First and foremost, I owe my deepest gratitude to my supervisor, Dr. Mats Brorsson, for his unwavering trust, insightful guidance, and consistent encouragement throughout this journey. His belief in my work and his valuable advice helped shape this thesis into what it is today.

I'm grateful to my thesis committee members, Prof. Radu State and Prof. Pascal Bouvry, for their thoughtful feedback, which significantly enhanced the quality of my research. I sincerely thank the Luxembourg National Research Fund (FNR) for their financial support.

I also want to express my appreciation to my collaborators from Google Research, Dr. Mangpo Phothilimthana, Dr. Sami Abu-El-Haija, and Dr. Bryan Perozzi, for their constructive feedback, technical insights, and inspiring discussions that continuously pushed me to improve my work.

I'm incredibly fortunate to have been part of the EuroHPC MAELSTROM team. Special thanks to Dr. Peter Duben from ECMWF for his support and collaboration, which greatly enriched my research and broadened my perspective.

I'm deeply grateful to my parents and friends for standing by me, offering their constant support and words of encouragement during this challenging journey.

Most of all, I want to thank my wife, Mahalakshmi, and my daughter, Haarikaa. Mahalakshmi, your love, patience, and constant support kept me grounded through the toughest times. And Haarikaa, your laughter and endless energy made even the hardest days feel easier. This achievement is as much yours as it is mine.

Karthick Panner Selvam
Luxembourg. March 2025

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Deep learning (DL), a subset of machine learning, utilizes multi-layered neural networks to process complex data. It has revolutionized fields such as computer vision [1], natural language processing [2, 3], autonomous systems [4] and climate modeling [5, 6] by uncovering intricate data patterns without relying on human-defined features.

Despite this transformative impact, deploying DL models across different computational environments remains a significant challenge. These challenges arise from the increasing complexity of modern DL architectures, the variety of hardware platforms, and the specific requirements of different workloads. Selecting suitable hardware configurations for a given model is a complex and time-intensive process, often requiring extensive trial and error.

AI accelerators like Graphics Processing Units (GPUs)[1], Tensor Processing Units (TPUs)[2], and Field Programmable Gate Arrays (FPGAs)[3] have been instrumental in meeting the computational demands of DL. GPUs, with their high degree of parallelism, are widely utilized for both training and inference tasks, while TPUs are optimized for large-scale matrix operations, making them suitable for high-performance DL deployments. FPGAs, on the other hand, offer customization capabilities that allow tailored optimizations for specific tasks. However, this diversity in hardware platforms introduces a pressing need for adaptable and efficient deployment strategies. Performance prediction becomes critical in this context to maximize resource utilization and minimize costs.

For example, hardware inefficiencies can lead to prolonged training times, increased latency, and high energy consumption. Without accurate performance predictions, developers and researchers may over-provision resources, leading to unnecessary expenses or under-provisioning resources, resulting in degraded system performance. Furthermore, as DL models continue to grow in complexity, including examples such as Transformers [7], Diffusion Models [8], and Large Language models (LLMs) like Generative Pre-trained Transformers (GPT) [9], their computational requirements increase substantially, resulting in high environmental costs due to inefficiencies.

[1]https://www.nvidia.com/en-us/data-center/h200/

[2]https://cloud.google.com/tpu

[3]https://www.intel.com/content/www/us/en/learn/fpga-for-ai.html

1.2 Research Challenges & Questions

Performance prediction of DL models on modern hardware architectures is a relatively new and rapidly evolving research area. Initially, researchers focused on predicting inference latency using simple machine learning techniques such as Multi-Layer Perceptron (MLP) regressors, which took high-level features like batch size, number of layers, and floating point operations (FLOPS) as input parameters [10–12]. While these approaches provided valuable insights, they failed to capture the intricate dependencies and dynamic interactions within DL models. As DL models grew more complex, it became evident that these initial methods could not adequately address the variability and heterogeneity of modern DL workloads.

The researchers then turned to kernel additive methods, a technique that first estimates the performance of individual operations, such as convolutions and dense layers, and then aggregates these values to estimate the overall performance of the model [13–18]. While effective for capturing layer-wise computation, this approach does not account for inter-layer dependencies and network topology, limiting accuracy for complex architectures. The relationships between different layers and their dependencies are critical for accurately predicting performance, especially for large and complex models.

Recent advancements have shown that graph-based methods, particularly Graph Neural Networks (GNNs), provide a more effective way to capture the hierarchical structure and computational dependencies within DL models. GNNs represent DL architectures as graphs, where nodes correspond to layers and edges represent data flow between them [19–21]. This approach allows for a holistic understanding of the model's structure and enables more accurate predictions of latency, memory usage, and energy consumption. By leveraging GNNs, researchers can move beyond simple feature-based predictions and incorporate the full complexity of modern DL architectures.

Despite these advancements, several key challenges remain. The following sections outline the core issues in performance prediction and discuss the specific gaps that this thesis aims to address.

1.2.1 Current Limitations in Performance Prediction

1.2.1.1 Deep Learning Framework Diversity

One significant challenge in performance prediction is the need to handle models from various deep learning frameworks, including TensorFlow[4], PyTorch[5], and ONNX[6]. Each framework has unique characteristics, such as dynamic versus static computation graphs, which impact performance. For instance, PyTorch's dynamic graphs allow greater flexibility but make static analysis difficult. On the other hand, TensorFlow uses static

[4]https://www.tensorflow.org/
[5]https://pytorch.org/
[6]https://onnx.ai/

graphs, simplifying performance prediction but reducing model flexibility. ONNX serves as an interoperability standard, but converting models between frameworks can introduce inaccuracies.

Handling this diversity requires robust tools to abstract framework-specific details and provide a unified representation of DL models. Ensuring compatibility across frameworks is essential for building performance prediction models that are agnostic to the underlying software stack.

1.2.1.2 Efficient NVIDIA MIG Profile Recommendation

NVIDIA's Multi-Instance GPU (MIG)[7] technology introduces an innovative way to partition a single GPU into multiple isolated instances, allowing for more efficient resource utilization. However, selecting the optimal MIG profile for a given DL model remains a complex task. The challenge lies in balancing resource allocation to ensure that each instance has sufficient memory and computing power while avoiding under-utilization of the GPU.

Existing performance prediction models do not adequately address MIG profile recommendations, as they typically focus on traditional metrics like latency and memory usage without considering partitioning strategies. This thesis aims to fill this gap by developing a framework that predicts the best MIG profile based on the specific characteristics of the input DL model.

1.2.1.3 Sparse Labeled Data

Obtaining labeled datasets for performance prediction is a costly and time-consuming process. Unlike traditional supervised learning tasks, where labeled data is abundant, performance prediction involves profiling models on various hardware platforms to collect metrics such as latency, memory usage, and energy consumption. This process requires significant resources, especially when dealing with emerging hardware architectures.

To address the issue of sparse labeled data, researchers have explored transfer learning and domain adaptation techniques. By leveraging existing datasets and adapting models to new hardware platforms with minimal data, it is possible to improve prediction accuracy without exhaustive profiling. However, this remains an ongoing challenge, particularly for new and specialized hardware accelerators.

1.2.1.4 Efficient Handling of Large Models

LLMs introduce unique challenges for performance prediction due to their hierarchical structures, high parameter counts, and extensive nodes and edges. These models require accurate resource consumption, throughput, memory usage, and energy consumption predictions to optimize their deployment on various hardware platforms.

While traditional GNN-based performance prediction methods are effective for computer vision models, they often struggle with LLMs. This difficulty arises because LLMs

[7] https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html

feature highly repetitive architectures and hierarchical dependencies that conventional GNNs find hard to capture.

To address these challenges we used tree-based model as an alternative, offering improved scalability and accuracy for LLM performance prediction tasks. The tree-based approach simplifies model representations, focusing on critical architectural and hardware features such as layer counts, attention heads, and FLOPs. This structured input enables faster and more accurate predictions for LLMs, addressing scalability concerns without compromising prediction quality.

1.2.1.5 Adaptability to New Hardware

The rapid evolution of hardware accelerators like GPUs, TPUs, and specialized AI chips demands performance prediction models that can quickly adapt to emerging hardware configurations. Traditional prediction models struggle with new hardware environments because they require extensive retraining.

This thesis proposes a multi modal learning framework that integrates GNNs and LLMs to enhance adaptability. The proposed solution introduces a three-stage pre-training strategy, consisting of GNN-based encoding, graph-to-text adaptation, and fine-tuning. This structured approach significantly reduces the need for extensive retraining by allowing models to generalize across different hardware environments with minimal labeled data.

The framework leverages GNNs to extract structural features from DL models and utilizes LLMs to handle diverse hardware descriptions and configurations. This integration enables the performance prediction model to adapt quickly to new hardware, offering accurate predictions even with sparse data samples.

1.2.2 Summary of Research Questions

The challenges outlined above give rise to several key research questions that this thesis seeks to address:

Research Question 1 (RQ1)

How can performance prediction models handle the diversity of DL frameworks for DL performance prediction?

Research Question 2 (RQ2)

What strategies can be employed to mitigate the impact of sparse labeled data and improve overall performance prediction accuracy?

Research Question 3 (RQ3)

How can performance prediction models scale to accommodate large DL architectures, such as LLMs?

Research Question 4 (RQ4)

What methodologies can enhance the adaptability of performance prediction models to new and emerging hardware environments?

1.3 Contributions

This thesis makes significant contributions to the field of performance prediction for DL models by addressing critical challenges in deploying and optimizing DL models across diverse hardware environments. The contributions are structured around four key areas: enhancing compatibility across frameworks, addressing sparse labeled data issues, improving performance prediction for large models, and ensuring adaptability to emerging hardware. These contributions are presented through a series of novel methodologies and frameworks developed in the five papers that form the core of this cumulative thesis.

1.3.1 Benchmarking and Profiling for Performance Prediction

The work begins with a systematic benchmarking study of DL models, highlighting computational bottlenecks and establishing the necessity of performance prediction as a foundation for further development. This study focuses on the MAELSTROM Temperature Downscaling (MTD) application, a DL model used for statistical temperature downscaling in climate science.

Through detailed roofline analysis and multi-level profiling across operators, training, distributed training, and inference, this contribution provides a comprehensive understanding of the model's computational requirements. The results reveal critical performance bottlenecks, including GPU utilization and memory access patterns, across different hardware configurations such as NVIDIA V100 and A100 GPUs.

The study emphasizes the importance of optimizing both hardware usage and model architecture to improve efficiency and reduce cloud computing costs. By identifying these bottlenecks, the research lays the groundwork for developing performance prediction models that can predict resource consumption and suggest optimal configurations for deploying deep learning models in high-performance computing (HPC) environments.

Operator-level profiling, training analysis, and distributed training benchmarks form the backbone for developing performance prediction models. These insights enhance understanding of how DL architectures interact with hardware platforms. This initial contribution demonstrates the practical implications of performance analysis in real-world scenarios and highlights the necessity for adaptable, accurate prediction tools capable of addressing the evolving landscape of DL workloads across diverse hardware environments.

1.3.2 Framework-Agnostic Performance Prediction Using Generalized Graph Structures

One of the core contributions of this thesis is the development of a Deep Learning Inference Performance Prediction Model (DIPPM), a framework-agnostic solution capable of handling diverse DL frameworks such as TensorFlow, PyTorch, and ONNX. Existing performance prediction models often focus on a single framework, limiting their generalizability. DIPPM introduces a unified approach that abstracts framework-specific details and represents DL models as generalized graph structures using Relay IR. This representation enables DIPPM to parse DL models from multiple frameworks and accurately predict performance metrics such as latency, memory usage, and energy consumption.

DIPPM converts the input DL models into graph representations, extracting both node and static features to create a comprehensive performance prediction dataset. The framework leverages GNNs to model the complex dependencies between layers in a DL architecture, significantly improving prediction accuracy. By adopting this framework-agnostic approach, DIPPM simplifies the integration of performance prediction into existing workflows, making it easier for practitioners to optimize their DL models for deployment.

1.3.3 Enhancing Prediction Accuracy Through Semi-Supervised Learning

A major challenge in performance prediction is the scarcity of labeled datasets that capture detailed performance metrics such as training time, memory usage, and energy consumption. Profiling DL models across various hardware platforms to gather labeled data is a time-consuming and resource-intensive process. To address this issue, this thesis introduces TraPPM (Training characteristics Performance Predictive Model), which leverages a semi-supervised learning approach to improve prediction accuracy.

TraPPM integrates unsupervised graph learning with supervised regression. In the first phase, an unsupervised Graph Autoencoder (GAE) [22] is used to generate embeddings from a large set of unlabeled DL models. These embeddings capture the structural information of the input DL models. In the second phase, these embeddings are combined with static features and used to train a supervised GNN-based regressor that predicts training characteristics such as step time and memory usage. The hybrid semi-supervised approach employed by TraPPM significantly enhances prediction accuracy. Empirical evaluation shows that TraPPM achieves a Mean Absolute Percentage Error of 4.92% for memory usage prediction and 9.51% for step time prediction, outperforming traditional supervised models such as MLP, GBoost, and other GNN-based methods. This contribution demonstrates the efficacy of harnessing unlabeled data to improve the performance of predictive models, reducing the dependency on extensive labeled datasets.

TraPPM has a substantial dataset, consisting of 8,079 labeled graphs and 25,053 unlabeled graphs from various DL model families. This dataset immensely contributes to the research community and helps future studies in performance prediction and optimization.

1.3.4 Performance Prediction for Large Models

Deploying LLMs in cloud environments poses significant challenges due to their high computational demands and complex architectures. Traditional GNN-based performance prediction methods, while effective for computer vision models, often struggle with LLMs due to their hierarchical and repetitive structures.

This thesis introduces an alternative approach using tree-based models to predict key performance metrics for LLMs, including throughput, memory usage, and energy consumption. Our framework demonstrates that tree-based models can achieve superior accuracy compared to GNNs, particularly for LLMs, while also significantly reducing prediction time. In empirical evaluations, tree-based model outperforms GNNs with improvements of approximately 68.81% in throughput prediction accuracy, 80.85% in memory usage prediction, and 88.21% in energy consumption prediction. Additionally, the tree-based model delivers a remarkable speed enhancement of approximately 267x over GNN-based approaches.

1.3.5 Adaptability to New Hardware Environments Using GNN-LLM Integration

A critical challenge in performance prediction is the ability to quickly adapt to new hardware environments with minimal retraining. The rapid evolution of GPUs, TPUs, and FPGAs makes maintaining prediction accuracy across diverse hardware configurations difficult, particularly when labeled data is limited.

This thesis addresses this issue by integrating GNNs with LLMs to improve adaptability and accuracy. GNNs capture the structural intricacies of DL models, while LLMs provide generalization capabilities across various hardware environments. A structured pre-training strategy, involving GNN pre-training with Graph Masked Autoencoder (GraphMAE) [23], graph-to-text adaptation, and performance prediction fine-tuning, allows the framework to adapt quickly to new hardware with sparse data samples.

Experimental results show an 8.8 percentage-point improvement in accuracy over GNN baselines and a 30–70 percentage-point increase when adapting to new hardware environments. This approach significantly reduces the need for extensive retraining, making it a practical solution for real-world scenarios where hardware configurations evolve rapidly and labeled data availability is limited.

1.3.6 Summary of Contributions

In summary, this thesis makes the following key contributions to the field of performance prediction for deep learning models:

- Establishment of a systematic benchmarking methodology for DL performance profiling, laying the foundation for developing the performance prediction model.
- Development of a framework-agnostic performance prediction model that ensures compatibility across TensorFlow, PyTorch, and ONNX.
- Implementation of a semi-supervised learning framework to enhance prediction accuracy using both labeled and unlabeled datasets.
- Introduction of a tree-based performance prediction framework for LLMs, achieving high accuracy and efficiency compared to GNNs.
- Proposal of a GNN-LLM framework with a structured pre-training strategy to improve adaptability and accuracy across emerging hardware environments.

These contributions collectively advance the state-of-the-art in performance prediction for DL models, providing practical solutions for optimizing DL workloads across diverse hardware environments.

1.4 Overview

This section provides an overview of the thesis.

Chapter 2 provides the technical foundation for this thesis, covering key concepts essential for DL performance prediction. It begins with an in-depth discussion of AI accelerators, focusing on NVIDIA GPU architectures. The chapter then introduces computation graphs as a fundamental representation of DL models and explains how GNNs capture model structure for performance estimation. Finally, it examines pre-trained LLMs and Parameter-Efficient Fine-tuning (PEFT) techniques, demonstrating their role in adapting performance prediction models to new hardware environments with minimal retraining.

Chapter 3 provides an in-depth analysis of computational bottlenecks in a DL based temperature downscaling application used in climate science. It introduces a structured approach for benchmarking DL models across various hardware setups. By applying roofline analysis and multi-level profiling, this chapter identifies key performance issues, such as GPU utilization and memory patterns on NVIDIA V100 and A100 GPUs, laying a foundation for predictive modeling. This chapter is based on this paper [24].

Chapter 4 introduces the DIPPM, a framework-agnostic solution for predicting DL performance across TensorFlow, PyTorch, and ONNX models. The model uses GNNs to capture architectural dependencies, enabling accurate predictions of latency, memory usage, and energy consumption. Additionally, DIPPM recommends optimal MIG profiles for efficient GPU resource allocation. This chapter addresses *RQ1* by providing a generalized approach to handle framework diversity. This chapter is based on this paper [25].

Chapter 5 addresses the challenge of sparsely labeled data by introducing a semi-supervised learning approach. TraPPM combines an unsupervised graph autoencoder with a supervised GNN regressor to improve prediction accuracy. The hybrid model leverages both labeled and unlabeled datasets to enhance predictions of training time and memory usage. This chapter addresses *RQ2* by demonstrating how semi-supervised learning can overcome the limitations of less labeled data in performance prediction. This chapter is based on this paper [26].

Chapter 6 presents a tree-based approach for predicting the performance of LLMs. Introduces a gradient boost model as an alternative to GNN, demonstrating high accuracy in predicting throughput, memory usage, and energy consumption for LLMs, along with faster prediction times. This chapter addresses *RQ3* by proposing an efficient, scalable solution for performance prediction in LLM architecture. This chapter is based on this paper [27].

Chapter 7 proposes a multi-modal learning framework that integrates GNNs and LLMs to improve performance prediction adaptability to new hardware environments. The chapter outlines a structured three-stage pre-training strategy, including GNN pre-training, graph-to-text adaptation, and performance prediction fine-tuning, enabling quick adaptation to new hardware with minimal retraining. This chapter answers *RQ4* by demonstrating how performance prediction models can efficiently adapt to emerging hardware platforms with sparse data samples. This chapter is based on this paper [28].

Chapter 8 concludes by summarizing the key contributions.

Chapter 9 discusses the limitations of this thesis and suggesting future research directions.

Chapter 10 list of papers.

2 Background

In this chapter, we present the essential technical background for our work on DL performance prediction. We begin by examining the role of specialized AI hardware accelerators, such as NVIDIA GPU architectures and MIG profiles, in efficiently managing DL workloads. Next, we describe how DL models are represented as computation graphs and explain how GNNs capture their complex structures. Finally, we review pre-trained LLMs and parameter-efficient fine-tuning techniques that enable rapid adaptation to specific tasks.

Contents

2.1 AI Accelerators

DL models require substantial computational resources due to the high dimensionality of input data and the complexity of model architectures. Traditional Central Processing Units (CPUs) struggle to handle such workloads efficiently, necessitating specialized AI accelerators such as GPUs, TPUs, GraphCore[1] and FPGAs. These accelerators are designed to handle matrix operations, optimize memory access patterns, and provide massive parallelism, making them ideal for DL workloads. In this thesis, we mainly developed performance models for NVIDIA GPUs.

2.1.1 Graphics Processing Units (GPUs)

GPUs leverage massive parallelism by executing thousands of threads simultaneously. Unlike CPUs, which are optimized for sequential execution, GPUs use the Single Instruction Multiple Thread (SIMT) architecture, enabling parallel computation across multiple data points. This is particularly advantageous in DL, where operations such as matrix multiplications and convolutions dominate computational workloads.

For a fully connected layer, let M be the weight matrix and X the input matrix. The matrix multiplication required for forward propagation is:

$$Y = WX + b$$

where b is the bias term.

GPUs optimize this computation by leveraging both SIMT cores and specialized hardware accelerators such as Tensor Cores[2] (in NVIDIA GPUs) and Matrix Cores[3] (in AMD GPUs). While Tensor Cores and Matrix Cores provide dedicated support for mixed-precision matrix multiplications, the SIMT cores also play a crucial role in accelerating these operations, particularly for workloads that do not fully utilize tensor processing units or require greater flexibility in precision and execution models. The CUDA[4] programming model allows for fine-grained control over parallel execution through thread blocks and warps, optimizing workload distribution.

Furthermore, GPUs support advanced memory hierarchies, including global, shared, and register memory, which help optimize memory access patterns. Techniques like tiling and memory coalescing further enhance performance by reducing memory bandwidth bottlenecks.

2.1.1.1 Multi-Instance GPU (MIG) Profiles

The MIG profile technology is available from NVIDIA A100 and H100 GPUs and with this technology it is possible to partition a single GPU into multiple isolated instances, each with dedicated compute cores, memory, and cache. This hardware-level

[1]https://www.graphcore.ai/

[2]https://www.nvidia.com/en-eu/data-center/tensorcore/

[3]https://rocm.blogs.amd.com/software-tools-optimization/matrix-cores/README.html

[4]https://developer.nvidia.com/cuda-toolkit

segmentation allows each instance to run its workload independently, with resource allocation tailored to specific computational requirements. MIG profiles are configured via NVIDIA's management libraries, which enable administrators to define profiles based on the number of Streaming Multiprocessors (SMs), the available high-bandwidth memory, and the cache assigned to each instance. However, Tensor Cores are not partitioned within MIG instances, they remain accessible but are not specifically allocated to a given instance.

The selection of Tensor Cores over CUDA cores occurs at the software level, within DL frameworks such as PyTorch. If a model is configured to use mixed precision, Tensor Cores are automatically leveraged for matrix multiplications. However, when running in a MIG instance, the proportion of available Tensor Cores is not explicitly controlled per instance, which can lead to variability in performance depending on how many instances share the underlying hardware.

Furthermore, the scalable architecture of MIG allows a GPU to be divided into a variable number of instances, where the instance count refers to how many partitions are created from a single GPU. This scalability is especially beneficial in multi-tenant cloud environments, where workloads vary and precise resource matching is essential for maintaining performance efficiency.

In Chapter 4, we will discuss how to predict the optimal MIG profile for a given DL model based on its computational requirements.

2.2 Deep Learning Models for performance prediction

2.2.1 Performance Prediction as a Regression Task

Performance prediction in DL is cast as a regression problem, where the goal is to estimate quantitative metrics, such as inference latency, memory consumption, and energy usage, from features extracted from both the model architecture and the hardware configuration. Let y denote the performance metric and $\mathbf{x} \in \mathbb{R}^d$ the corresponding feature vector. The regression task seeks a function $f : \mathbb{R}^d \to \mathbb{R}$ such that

$$y = f(\mathbf{x}) + \epsilon,$$

with ϵ representing noise and unmodeled variability. Typically, f is parameterized by θ, and training minimizes a loss function, often the Mean Squared Error (MSE):

$$\mathcal{L}(\theta) = \frac{1}{N} \sum_{i=1}^{N} \left(y_i - f(\mathbf{x}_i; \theta)\right)^2, \tag{2.1}$$

where N is the number of training samples.

Initial approaches used MLP-based regressors that learned the mapping from aggregated features—such as batch size, number of layers, and total FLOPS—to performance metrics [10–12]. Although effective in demonstrating the feasibility of performance prediction, these methods were limited by their inability to capture the complex dependencies inherent in DL models.

2.2.2 Computation Graphs and Graph Neural Networks

As DL architectures became more complex, graph-based representations were adopted to capture their hierarchical structure and interdependencies. A DL model can be represented as a computation graph $G = (V, E)$, where V is the set of computational operations (e.g., matrix multiplications, convolutions, activations) and E denotes the directed edges that define data flow. For a node $v_i \in V$, the output is computed as:

$$O_i = f(W_i X_i + b_i),$$

where X_i is the input, W_i is the weight matrix, b_i is the bias term, and $f(\cdot)$ is an activation function.

GNNs encode the structural information of these computation graphs. In a typical GNN, each node v has an embedding $\mathbf{h}_v^{(l)}$ that is updated iteratively using a message passing mechanism. At layer l, the update is given by:

$$\mathbf{h}_v^{(l)} = \sigma\left(\mathbf{W}^{(l)} \cdot \mathbf{m}_v^{(l)} + \mathbf{b}^{(l)}\right),$$

where σ is a nonlinear activation function, $\mathbf{W}^{(l)}$ and $\mathbf{b}^{(l)}$ are learnable parameters, and

$$\mathbf{m}_v^{(l)} = \text{AGG}\left(\{\mathbf{h}_u^{(l-1)} : u \in \mathcal{N}(v)\}\right)$$

is the aggregated message from neighboring nodes. The initial node embeddings are set as:

$$\mathbf{h}_v^{(0)} = \mathbf{x}_v.$$

After L layers, a readout function aggregates the node embeddings into a graph-level representation:

$$\mathbf{h}_G = \text{READOUT}\left(\{\mathbf{h}_v^{(L)} : v \in V\}\right).$$

For performance prediction, this embedding is passed through a regression head:

$$y = f_{\text{reg}}(\mathbf{h}_G; \theta) + \epsilon.$$

The model is trained by minimizing the MSE over the training samples.

This GNN-based framework effectively captures the intricate structural details of DL models by integrating them into a unified embedding space, leading to more accurate and robust performance predictions. The methods described here form the foundation for the approaches presented in Chapters 4 through 7.

2.2.3 Performance Prediction Dataset

Accurate performance prediction for DL workloads requires benchmarking datasets capturing execution behavior across diverse hardware architectures. These datasets enable predictive models to estimate inference latency, memory consumption, and energy efficiency, but their collection is complex due to DL workload variability, hardware heterogeneity, and profiling costs.

The dataset comprises DL models represented as computation graphs G. Each data point consists of $\mathcal{X}$ (node feature matrix), $\mathcal{A}$ (adjacency matrix), $\mathcal{Y}$ (performance metrics), and $\mathcal{F}_s$ (static model features). The $\mathcal{X}$ captures properties such as operation type (e.g., convolution, matrix multiplication, activation function), input/output tensor shapes, and computational cost in FLOPs. The $\mathcal{A}$ represents the directed edges defining data dependencies between operations in the computation graph. The $\mathcal{Y}$ include inference latency (ms), memory consumption (MB), and energy usage (J) measured through multiple inference runs. The $\mathcal{F}_s$ encapsulate architecture-level characteristics such as total parameter count, depth, number of attention heads (for transformer-based models) and batch size.

Profiling is performed on NVIDIA GPUs under varying batch sizes, precision settings (FP32, FP16, INT8), and memory constraints. Performance metrics are recorded using the NVIDIA Management Library, CUDA toolkit, and profiling tools such as Nsight and PyTorch Profiler. Multiple inference runs are averaged for robust estimation.

Dataset construction is challenging due to execution variability, even within the same hardware family, such as NVIDIA A100 vs. H100. Limited access to emerging hardware constrains large-scale benchmarking, while profiling large models like LLMs is computationally expensive. Sparse labeled data necessitate semi-supervised learning to integrate labeled and unlabeled data effectively.

2.3 Training Regimes for Performance Prediction

We now describe the training regimes used in our performance prediction framework, encompassing supervised, self-supervised, and semi-supervised learning approaches.

2.3.1 Supervised Learning

In the supervised regime, we use a labeled dataset

$$\mathcal{D}_s = \{(G_i, y_i)\}_{i=1}^{N_s},$$

where each G_i is a computation graph representing a DL model and $y_i \in \mathbb{R}$ is the corresponding performance metric. Our objective is to learn a regression function f_{reg} that maps features extracted from G_i to y_i:

$$y_i \approx f_{\text{reg}}\Big(\text{GNN}(G_i; \theta_{\text{gnn}}); \theta_{\text{reg}}\Big).$$

The graph-level embedding $\mathbf{h}_{G_i}$ is obtained using a READOUT function on the final node embeddings:

$$\mathbf{h}_{G_i} = \text{READOUT}\left(\{\mathbf{h}_v^{(L)} : v \in G_i\}\right).$$

Training minimizes the MSE:

$$\mathcal{L}_{\text{sup}}(\theta_{\text{gnn}}, \theta_{\text{reg}}) = \frac{1}{N_s} \sum_{i=1}^{N_s} \left(y_i - f_{\text{reg}}(\mathbf{h}_{G_i}; \theta_{\text{reg}})\right)^2.$$

2.3.2 Self-Supervised Learning

Self-supervised learning leverages unlabeled data to learn meaningful representations.

$$\mathcal{D}_u = \{G_j\}_{j=1}^{N_u}$$

Let $\mathcal{D}_u$ be an unlabeled dataset. A Graph Autoencoder (GAE) [22] is used to learn latent representations by reconstructing the input graphs. The encoder, parameterized by ϕ, maps a graph G to a latent embedding $\mathbf{z}$:

$$\mathbf{z} = \text{Encoder}(G; \phi),$$

and the decoder, parameterized by ψ, reconstructs the graph:

$$\hat{G} = \text{Decoder}(\mathbf{z}; \psi).$$

The reconstruction loss is defined as:

$$\mathcal{L}_{\text{recon}}(\phi, \psi) = \frac{1}{N_u} \sum_{j=1}^{N_u} \mathcal{L}_{\text{graph}}(G_j, \hat{G}_j),$$

where $\mathcal{L}_{\text{graph}}(\cdot, \cdot)$ is an appropriate graph reconstruction loss.

2.3.3 Semi-Supervised Learning

Our semi-supervised framework combines the strengths of supervised and self-supervised learning. First, a GAE is pre-trained on the unlabeled dataset $\mathcal{D}_u$ to learn robust structural embeddings, and the encoder parameters ϕ are subsequently frozen. For a graph G in the labeled dataset $\mathcal{D}_s$, we obtain two embeddings: one from the frozen GAE encoder,

$$\mathbf{z} = \text{Encoder}(G; \phi),$$

and one from the GNN trained in the supervised phase,

$$\mathbf{h}_G = \text{GNN}(G; \theta_{\text{gnn}}).$$

These embeddings are concatenated to form the final representation:

$$\mathbf{h}_{\text{final}} = \mathbf{z} \| \mathbf{h}_G.$$

The regression function is then applied to this combined embedding:

$$y \approx f_{\text{reg}}(\mathbf{h}_{\text{final}}; \theta_{\text{reg}}).$$

The supervised loss for this semi-supervised regime is:

$$\mathcal{L}_{\text{semi}}(\theta_{\text{gnn}}, \theta_{\text{reg}}) = \frac{1}{N_s} \sum_{i=1}^{N_s} \Big(y_i - f_{\text{reg}}(\mathbf{z}_i \| \mathbf{h}_{G_i}; \theta_{\text{reg}}) \Big)^2 .$$

This approach leverages the robust structural features learned by the GAE along with the task-specific features from the GNN, leading to improved performance prediction.

2.4 Pre-trained Large Language Models

LLMs are typically pre-trained on vast corpora using self-supervised objectives, enabling them to capture rich linguistic and semantic patterns. In this section, we mathematically describe the key components of pre-trained LLMs, including tokenization, embedding generation, and the transformer architecture, and explain how parameter-efficient fine-tuning methods, such as Low-Rank Adaptation (LoRA) [29] and soft prompting [30], modify these models for downstream tasks.

2.4.1 Tokenization and Embedding Generation

Let X be an input text string. A tokenizer $\mathcal{T}$ converts X into a sequence of tokens:

$$\mathbf{t} = \mathcal{T}(X) = [t_1, t_2, \ldots, t_n],$$

where each t_i is an element from a fixed vocabulary $\mathcal{V}$. These tokens are then mapped into continuous vector representations via an embedding matrix $\mathbf{E} \in \mathbb{R}^{|\mathcal{V}| \times d_{\text{emb}}}$, where d_{emb} is the embedding dimension. The embedding for token t_i is given by:

$$\mathbf{e}_i = \mathbf{E}[t_i] \in \mathbb{R}^{d_{\text{emb}}}.$$

Thus, the entire input text is represented as an embedding sequence:

$$\mathbf{X}_{\text{emb}} = [\mathbf{e}_1, \mathbf{e}_2, \ldots, \mathbf{e}_n] \in \mathbb{R}^{n \times d_{\text{emb}}}.$$

2.4.2 Pre-trained Transformer Model

The pre-trained LLM (e.g., GPT, BERT, Llama, etc.) is typically built upon a Transformer architecture. Let the pre-trained model be denoted as a function $\mathcal{M}$ parameterized by θ. For a given input embedding sequence $\mathbf{X}_{\text{emb}}$, the Transformer produces an output sequence:

$$\mathbf{Y} = \mathcal{M}(\mathbf{X}_{\text{emb}}; \theta),$$

where $\mathbf{Y} \in \mathbb{R}^{n \times d_{\text{model}}}$ and d_{model} is the model's hidden dimension. The output $\mathbf{Y}$ is typically further processed (e.g., via a linear layer and softmax) to generate probabilities over the vocabulary for next-token prediction:

$$\hat{\mathbf{y}}_i = \text{softmax}\left(\mathbf{W}_o \mathbf{y}_i + \mathbf{b}_o\right), \quad \forall i = 1, \ldots, n,$$

where $\mathbf{W}_o \in \mathbb{R}^{|\mathcal{V}| \times d_{\text{model}}}$ and $\mathbf{b}_o \in \mathbb{R}^{|\mathcal{V}|}$.

2.4.3 Parameter-Efficient Fine-tuning (PEFT)

Traditional fine-tuning updates all parameters θ of the pre-trained LLM, which can be computationally expensive. Parameter-efficient fine-tuning methods modify only a small subset of parameters, allowing the pre-trained model to retain much of its original knowledge while adapting to new tasks.

2.4.3.0.1 Low-Rank Adaptation (LoRA): In LoRA, instead of updating the full weight matrix $\mathbf{W}$ in a Transformer layer, we introduce a low-rank decomposition:

$$\Delta\mathbf{W} = \mathbf{B}\mathbf{A},$$

where $\mathbf{B} \in \mathbb{R}^{d_{\text{model}} \times r}$ and $\mathbf{A} \in \mathbb{R}^{r \times d_{\text{model}}}$ with $r \ll d_{\text{model}}$. The weight matrix is then updated as:

$$\mathbf{W}' = \mathbf{W} + \Delta\mathbf{W} = \mathbf{W} + \mathbf{B}\mathbf{A}.$$

Only $\mathbf{B}$ and $\mathbf{A}$ are optimized during fine-tuning, while the original weight $\mathbf{W}$ remains frozen. The gradients with respect to $\mathbf{B}$ and $\mathbf{A}$ are computed as:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{B}} = \frac{\partial \mathcal{L}}{\partial \Delta\mathbf{W}} \cdot \mathbf{A}^T, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{A}} = \mathbf{B}^T \cdot \frac{\partial \mathcal{L}}{\partial \Delta\mathbf{W}},$$

where $\mathcal{L}$ is the loss function for the downstream task.

2.4.3.0.2 Soft Prompting: Soft prompting involves prepending learnable prompt embeddings $\mathbf{P} \in \mathbb{R}^{m \times d_{\text{emb}}}$ to the input embeddings. Given an input sequence $\mathbf{X}_{\text{emb}}$, the augmented input becomes:

$$\mathbf{X}_{\text{aug}} = [\mathbf{P}; \mathbf{X}_{\text{emb}}] \in \mathbb{R}^{(m+n) \times d_{\text{emb}}}.$$

During fine-tuning, only the prompt embeddings $\mathbf{P}$ (and possibly the LoRA parameters) are updated. The output is then:

$$\mathbf{Y} = \mathcal{M}(\mathbf{X}_{\text{aug}}; \theta),$$

where the pre-trained parameters θ are largely preserved.

2.4.4 Fine-Tuning: Input, Output, and Optimization

For LLM fine-tuning, the input is a tokenized textual prompt representing the task context. Let $\mathbf{Q} \in \mathbb{R}^{n \times d_{\text{emb}}}$ denote the sequence of embeddings for the input prompt, where each token is mapped into a d_{emb}-dimensional vector. The fine-tuned LLM processes $\mathbf{Q}$ and produces an output sequence, from which a scalar value $\hat{y}$ is derived as the predicted performance metric (e.g., inference latency). Mathematically, this is expressed as:

$$\hat{y} = f_{\text{LLM}}(\mathbf{Q}; \theta') \in \mathbb{R},$$

where f_{LLM} represents the function computed by the LLM and θ' denotes the set of parameters updated during fine-tuning (e.g., prompt embeddings and parameters adjusted via techniques such as LoRA or soft prompting), while the bulk of the pre-trained parameters remain fixed.

The training objective is to minimize the Mean Squared Error (MSE) between the predicted performance $\hat{y}$ and the true performance y:

$$\mathcal{L}(\theta') = \frac{1}{N} \sum_{i=1}^{N} \left(y_i - f_{\text{LLM}}(\mathbf{Q}_i; \theta')\right)^2 .$$

Here, N is the number of training samples. Fine-tuning is typically performed using gradient descent or its variants, ensuring that only a small subset of parameters is updated to efficiently adapt the pre-trained LLM to the specific performance prediction task.

2.5 Summary

This background section lays the technical foundation for our work by first outlining the critical role of specialized hardware, particularly GPUs, in supporting the intensive computations of deep learning (DL) models. Next, we framed performance prediction as a regression task, where key metrics (e.g. latency and memory usage) are estimated from features derived from both the DL model architecture and its hardware environment.

To capture the inherent complexity of DL models, we represented them as computation graphs and demonstrated how GNNs can encode these graphs into unified embeddings. These embeddings effectively capture the hierarchical and interdependent nature of DL architectures, forming the basis for accurate performance prediction. We also discussed various training regimes, supervised, self-supervised, and semi-supervised, that leverage both labeled and unlabeled data to enhance model robustness.

Finally, we provided an overview of pre-trained LLMs and explained how parameter-efficient fine-tuning techniques such as LoRA and soft prompting allow these models to be adapted to specific tasks with minimal updates. We believe that this background section will provide a solid foundation for understanding the remainder of the thesis.

3 Performance Analysis and Benchmarking of a Temperature Downscaling Deep Learning Model

We are presenting here a detailed analysis and performance characterization of a Statistical temperature downscaling application used in the MAELSTROM EuroHPC project. This application uses a deep learning methodology to convert low-resolution atmospheric temperature states into high-resolution. We have performed in-depth profiling and roofline analysis at different levels (Operators, Training, Distributed Training, Inference) of the downscaling model on different hardware architectures (Nvidia V100 & A100 GPUs). Finally, we compare the training and inference cost of the downscaling model with various cloud providers. Our results identify the model bottlenecks which can be used to enhance the model architecture and determine hardware configuration for efficiently utilizing the HPC. Furthermore, we provide a comprehensive methodology for in-depth profiling and benchmarking of the deep learning models.

This chapter builds upon the research presented in the following publication:

Peer reviewed

Karthick Panner Selvam, and Mats Brorsson. "Performance Analysis and Benchmarking of a Temperature Downscaling Deep Learning Model," in 2023 31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), Naples, Italy, 2023, pp. 1-8, doi: 10.1109/PDP59025.2023.00010.

Contents

3.1 Introduction

The use of AI and machine learning involves a considerable amount of computing resources in training the models and, to a lesser degree, in production when the models are used. It is essential to understand the computational needs of a model in training and inference so as to i) give it enough resources and ii) not give it more than needed so that resources are wasted. In this paper, we present a thorough performance analysis and characterization of the MAELSTROM Temperature Downscaling (MTD) application which is a post-processing methodology to convert low-resolution atmospheric grid space into high-resolution grid space using a Convolution Neural Network (CNN). The statistical downscaling technique used in MTD is highly inspired by CNN-based image super-resolution because it takes grid-based input, learns spatial-temporal patterns from trained data, and converts the low-resolution atmospheric states to a high-resolution [31, 32]. The MTD application uses U-Net [33] architecture to enhance the spatial resolution of atmospheric T2M (2 meters above surface air temperature) [34–36].

The approach we take to analyze and characterize the Downscaling application is to combine modular-level deep learning benchmarking [37] with roofline analysis[37, 38], where we study i) the operators that make up the model, ii) the inference network, and iii) the training of the network. We specifically concentrate on the convolutional operators which make up the U-net model architecture. These are studied with the help of roofline graphs to understand how close the application performance is to the architecture's empirical limits. The platforms we have chosen to study the application are Nvidia A100 and V100 GPUs. The contributions of our work include the following:

- An analysis of the main computational components of the MTD application and their computational need,
- Roofline characterizations of the convolutional operators of the MTD application on A100 and V100 GPUs,
- Benchmarking of inference and training performance on A100 and V100 and provided methodology to effectively to utilise A100 GPU for MTD model inference
- A cost-analysis on some of the major cloud providers,
- A methodology to capture performance data for the above-mentioned performance measurements.

Our benchmark and profiling technique shows how well the model utilizes the hardware by varying batch size, kernel, strides, and filters at the convolution operator level by using roofline analysis. It shows tensor core utilization across multiple GPUs and energy utilization for training on V100 and A100 GPUs. Furthermore, we provided a methodology to efficiently use A100 GPU for inference. Using our performance metrics results, developers can enhance the MTD model architecture and adjust the hardware configuration to utilize the underlying hardware effectively. For example, increasing the input and output grid space, fixing the batch size, selecting the optimal number of GPUs for training, and finding cost-effective cloud instances for inference and training.

3.2 Background

3.2.1 The MAELSTROM Temperature Downscaling Application

The MTD model uses a U-Net architecture to convert low-resolution grid space atmospheric 2m temperature states into a high resolution [33]. For training the MTD model, we used data provided by the European Centre for Medium-Range Weather Forecasts (ECMWF) in form of the HRES[1] dataset using the Bi-linear interpolation technique. The target T2M (2 meters above surface air temperature) is the direct output of the ECMWF HRES dataset; that is why input T2M looks like smooth temperature fields, whereas target T2M is sharpened because the original data is captured in complex terrain, as shown in Figure 3.1.

The MTD model architecture is illustrated in Figure 3.2. The most compute-intensive parts are the 2D convolution layers (Conv2D). The training data contains 20496 time steps over our target territory with a 96×128 grid space in the zonal and meridian directions. The MTD model accepts input of $96 \times 128 \times 3$ grid space, data variables are (Low-resolution T2M, elevation and High-resolution elevation) and generates two outputs, each with the shape of $96 \times 128 \times 1$ (High-resolution T2M and elevation). The current dataset contains only the central part of Europe. More coarsened HRES data across Europe on different periods will be added to improve the precision, so the MTD models will require retraining with new data to increase the accuracy of enhancing the spatial resolution of T2M.

3.2.2 Hardware aspects

We have chosen to base most of our analysis on measurements on the Nvidia V100 and A100 GPUs. These are computational engines built specifically for AI and machine learning applications. Table 3.1 summarizes the main characteristics of these platforms. Both A100 and V100 use a Tensor Cores (TC) to accelerate the computing of matrix-matrix

[1]https://www.ecmwf.int/en/forecasts/datasets/set-i

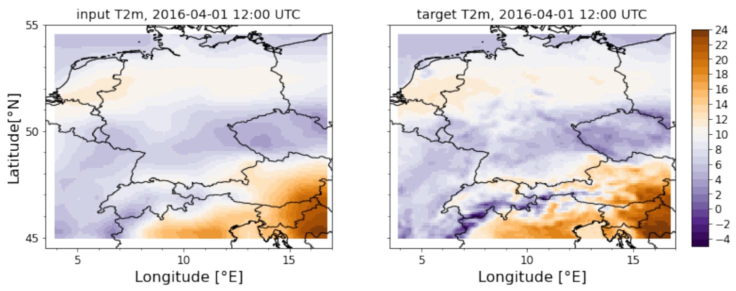

input T2m, 2016-04-01 12:00 UTC
target T2m, 2016-04-01 12:00 UTC
Latitude[°N]
55
50
45
5
10
15
Longitude [°E]
24
22
20
18
16
14
12
10
8
6
4
2
0
−2
−4

Figure 3.1: MAELSTROM Temperature Downscaling Model Input and Target Output of T2m (2 meters above surface air temperature) Grid Space [96 x 128].

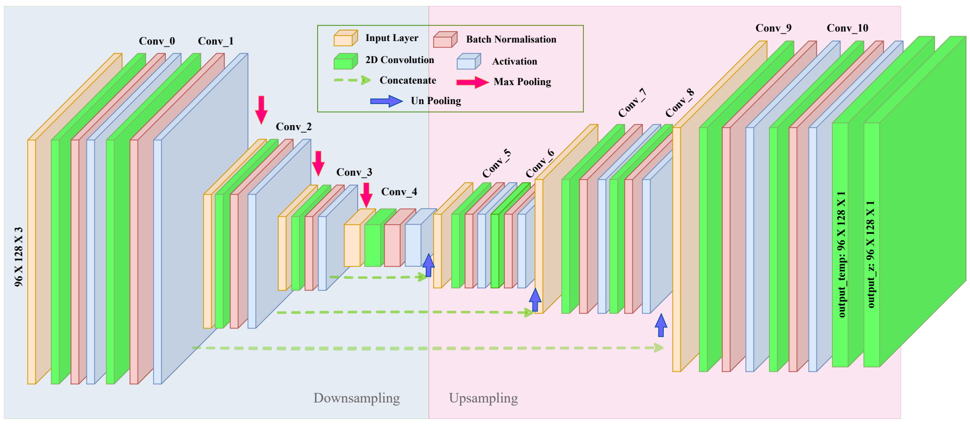


Figure 3.2: MAELSTROM Temperature Downscaling Model Architecture
.

Table 3.1: Nvidia V100 and A100 main specifications.

	V100	A100
FP64	7.8 TFlop/s	9.7 TFlop/s
FP32	15.7 TFlop/s	19.5 TFlop/s
Tensor float 32	125 TFlop/s	312 TFlop/s
GPU Memory	16 GB	40 GB

multiplication, and it is widely used in deep learning model training. A100 TC supports single precision (FP32), half-precision (FP16), BFLOAT16 and INT8. Whereas V100 only supports half precision tensor core.

3.2.3 Benchmarking Deep Learning Models

Various deep learning benchmark suites are available like MLPerf [39], Dawnbench [40], and Deepbench [41][42]. Most suites support high-level profiling aspects, like wallclock performance metric, accuracy, training, and validation loss. Ben-Nun et al. [37]. proposed a modular level approach to benchmark a deep learning model on various levels, including distributed training, but it lacks to identify the bottlenecks of operators and cloud costs. However, low-level profiling is vital to identify the bottlenecks of model architecture efficiency. Researchers used roofline analysis to identify the bottlenecks of deep learning applications, but it is limited to the operator-level investigation[43, 44]. Yang et al.[38] proposed hierarchical roofline analysis to identify bottlenecks in convolution operator and model training; however, this methodology lacks to extend it to distributed training.

We propose a comprehensive and in-depth profiling methodology for benchmarking and performance analysis of deep learning models to fill the gap in the previous research work. We combine the modular level approach of deep learning benchmark and roofline technique [37, 38] with cost analysis and dissect the model into five levels: Operator, Training, Distributed training, Inference, and Cloud cost comparison. We performed deep profiling at each level and identified the performance bottlenecks of the model for future enhancement. Furthermore, we provided a methodology to effectively use A100 GPU for inference.

3.3 Methodology

We separated the experiments into multiple levels: Operator, Training, Distributed training, Inference, and cloud compute cost comparison. For the operator level, we conduct the experiment on the MTD model's convolution operators. We perform roofline analysis on NVIDIA V100 and A100 GPUs for Conv2D operators by the varying batch size, strides, kernel, and filters to identify the bottlenecks of these operators. Using the knowledge gained from the operator level, we train the MTD model with two precision modes, single precision and mixed precision, on single V100 and A100 GPU, and we measure Tensor core, GPU utilization, GPU power, memory usage, average epoch time, training and validation loss. Then, we use HPC clusters for distributed training to find out whether increasing GPUs reduces the training time for the MTD model. For each node, we increase GPU gradually from 1 to 4 for training. We used TensorFlow mirrored strategy API for distributed training because it synchronous training across multiple GPUs on a single node.

At the inference level, we calculate the one full forward propagation time of the MTD model trained with single and mixed precision. Then we used the DLProf[2]tool to extract the TC utilization, GPU utilization, and wallclock time for inference. We used roofline analysis to find the computational bottleneck of the kernel. Then we compare inference with V100 and A100 GPUs, and we suggest a methodology to use NVIDIA Multi-instance GPU (MIG[3]) technology for efficient inference on A100 GPU. Finally, we compare training and inference cost with various cloud vendors.

3.3.1 Systems used

3.3.1.1 Hardware

We used two HPC systems for our research: JUWELS[4] Booster and JUWELS cluster. Juwels Booster contains 936 nodes, and each node contains 4 x A100 Nvidia GPUs with 40 GB HBM connected via NVLink3 with AMD EPYC 7402 processor and 512GB memory. Juwels cluster contains 56 computing nodes, each containing 4× NVIDIA V100 GPU and 16 GB HBM with 2× Intel Xeon Gold 6148 processor. We used single-node JUWELS Booster and JUWELS Cluster extensively across all levels.

[2]https://docs.nvidia.com/deeplearning/frameworks/dlprof-user-guide/
[3]https://docs.nvidia.com/datacenter/tesla/mig-user-guide/index.html
[4]https://www.fz-juelich.de/en/ias/jsc/systems/supercomputers/juwels

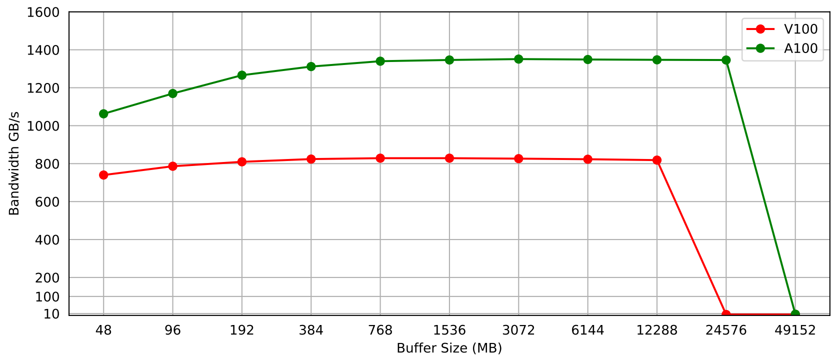


Figure 3.3: High Bandwidth Memory (HBM) Peak Throughput of V100 and A100 GPUs

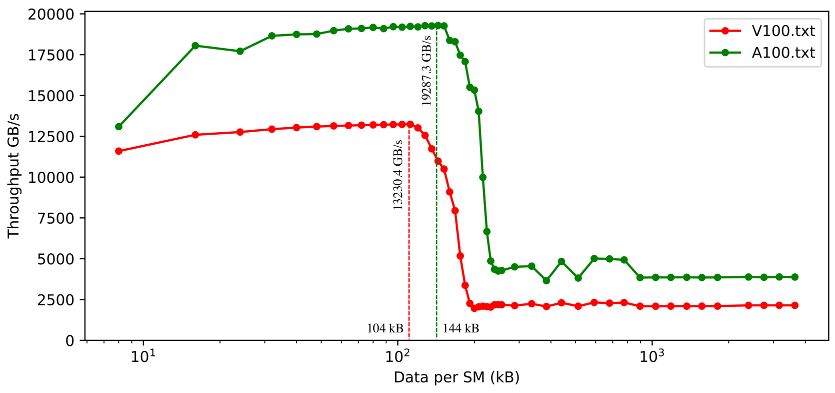


Figure 3.4: L1 and L2 cache Peak Throughput of V100 and A100 GPUs.

3.3.1.2 Software framework

Throughout the experiments, we used TensorFlow version 2, Python 3.9.6, CUDA 11.6, and cuDNN v8.3.1. We used the following precision modes on all levels: single precision and half-precision for operator levels and mixed precision (It enables TC on GPU to utilize FP32 and FP16 for accelerated computing) for training level.

3.3.2 Roofline characterization

The roofline model is a visually-intuitive way to understand the kernel performance and identify the bottlenecks of kernel execution on a given machine [38]. We perform roofline analysis on V100 and A100 GPUs of Conv2D operators because they are the most compute-intensive operations in the MTD application. The roofline model is an intuitive model based on bounds of data transfer and compute capacity and can be expressed as in equation 3.1. In order to plot the roofline model for MTD Conv2D operators on GPUs, we need two components:

- *Machine Characterization*: In terms of peak GFLOP/s for (single precision, half-precision, and TC) and Peak bandwidth (GB/s) of L1 cache, L2 cache, and High Bandwidth Memory (HBM).
- *Application Characterization*: Conv2D kernel Arithmetic Intensity (AI) expressed as floating point operations by data movement (FLOPs per Bytes) and Conv2D kernel throughput GFLOP/s of (single precision, half-precision, and TC).

$$\mathrm{GFLOP/s} \leq \min \begin{cases} \text{Peak GFLOP/s} \\ \text{Peak GB/s } \times \text{AI} \end{cases} \tag{3.1}$$

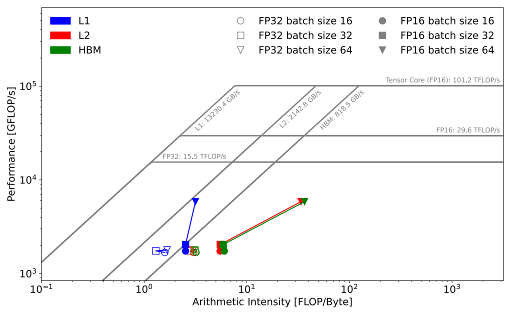
3.3.2.1 Machine Characterization

We used the Empirical Roofline Tool [38] (ERT) kit and Cuda-Bench[45] tool to collect machine characterization metrics empirically. We collected single and half-precision values for GPUs using ERT. V100 peak single precision is 15.5 TFLOP/s, and half-precision is 29.6 TFLOP/s. A100 peak single precision is 18.5 TFLOP/s, and half-precision is 58.8 TFLOP/s. We used Cutlass GEMMs (General Matrix Multiplications) to calculate Peak TC TFLOP/s. GEMMs are defined in Equation 3.2.

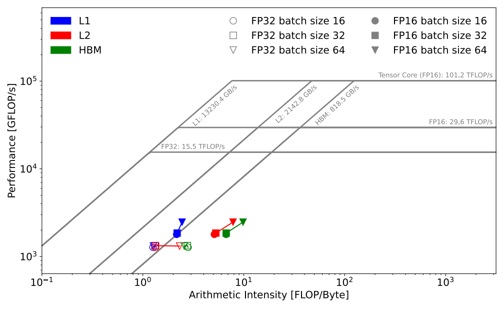
$$D = \alpha \mathrm{AB} + \beta C \tag{3.2}$$

where A and B are input matrices, and C is already existing matrix and overwritten by output matrix D. α and β are scalar constants. For matrix A is M x K matrix, B is M x N and C and D are N x K matrix. We used $M = N = K = 16384$ matrix to get the peak TC results for V100 is 101.2 TFLOP/s, and A100 is 292.2 TFLOP/s. To calculate the Peak HBM throughput, we used a stream-kernel technique from Cuda-Bench by increasing the buffer size from 48 MB to 49152 MB, which guarantees to capture the device memory within. The V100 we used has a 16GB memory limit, whereas the A100 has a 40 GB memory limit. As a result, V100 can reach a peak HBM bandwidth of 818.5 GB/s on a buffer size of 12288 MB, and A100 reaches a peak HBM bandwidth of 1346 GB/s on 24576 MB, as shown in Figure 3.3.

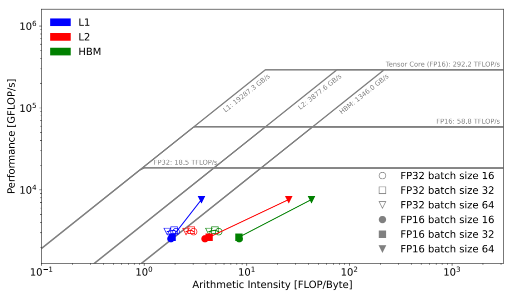
To read the L1 and L2 cache bandwidth, we launch one thread block per Stream Multiprocessor (SM). Every thread block reads the same buffer continuously, and we vary the buffer size to measure peak L1 and L2 bandwidth, as shown in Figure 3.4. As a result, V100 L1 bandwidth is 13230.4 GB/s, and L2 bandwidth is 2142.8 GB/s. And A100 L1 bandwidth is 19287.3 GB/s and L2 is 3877.6 GB/s. We use these values to plot the roofline model of L1, L2, HBM, Single and half-precision and Tensor Core ceilings as shown in Figure 3.5



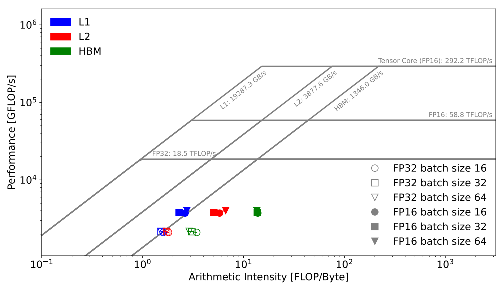
(a) V100 Forward Pass



(b) V100 Backward Pass



(c) A100 Forward Pass



(d) A100 Backward Pass

Figure 3.5: Batch Size: Roofline Analysis of V100 and A100 GPUs for Conv2D (128x96x112 tensor, 56 filters, 3x3 kernel, strides 3) with batch sizes 16, 32, 64.

3.3.2.2 Application Characterization

We use the Nsight[5] Compute CLI tool to collect the kernel performance and bandwidth metrics for roofline analysis. We use PyCUDA[6] marker API to specify the specific region of the code to extract the metrics. We used the following command to extract kernel metrics.

ncu -profile-from-start off –metrics ***[metrics]*** *python app.py*

We used Equation (3.3) to calculate the kernel execution time K_t from the ncu output. We calculate single precision and half-precision and TC FLOPs using Equation (3.4), (3.5), and (3.6), respectively. We calculate L1 and L2 cache from this metrics $\{l1tex, lts\}\_\_t\_bytes.sum$ and device memory from this metrics $dram\_\_bytes.sum$

$$K_t = \frac{elapsed\_avg}{avg\_per\_second} \tag{3.3}$$

[5]https://docs.nvidia.com/nsight-compute/NsightComputeCli/index.html
[6]https://documen.tician.de/pycuda/

$$FP32\ FLOPs = \_fadd\_ + (2 * \_ffma\_) + \_fmul\_ \tag{3.4}$$

$$FP16\ FLOPs = \_hadd\_ + (2 * \_hfma\_) + \_hmul\_ \tag{3.5}$$

$$TC\ FLOPs = \_pipe\_tensor.sum * 512 \tag{3.6}$$

3.4 Experiments and Results

3.4.1 Operator Level

We use roofline analysis to identify the computational bottlenecks and data movement of different level cache (L1 and L2) and HBM of convolution operators. We use V100 and A100 GPUs for roofline analysis. First, we collected the GPU kernel metrics using the nsight-compute-cli tool and use PyCUDA marker API to specify the code region to extract metrics. Then, we pre-process the metrics to get the Arithmetic intensity of L1, L2, HBM, and performance GFLOP/s as mentioned in Section 3.3.2.2 and plot these results in the roofline chart as shown in Figure 3.5.

In the roofline chart, the horizontal ceiling represents peak GFLOP/s of Tensor Core, single precision, and half precision. The color blue, red, and the green represents kernel associated with L1, L2 cache, and HBM. If the kernel value is near the ceilings of (TC, FP32, and FP16) of the roofline chart, it shows the kernel has higher performance. If the kernel values of blue, red, and green colors near the respective diagonal ceilings (L1, L2, and HBM) show good data locality. If any color overlaps, it shows poor data locality.

For this experiment, we used the MTD model's highest FLOP operation, which is Conv2D: (tensor shape: 128x96x112, batch size:32, filters: 56, kernel: 3x3, strides:3) as base configuration, and we vary the batch size, kernel size, strides, and filters of the above-mentioned Conv2d operator with single and half-precision modes. The roofline analyses for the mentioned Conv2D operators are performed in both forward and backward passes. However, we can use and change only batch size configuration for further experiments of MTD's model. Still, we perform roofline analysis for varying strides, kernels, and filters to find out the bottlenecks. Due to the page limit, we only included the roofline chart for batch size configuration and only backward passes of varying kernel size. For the remaining configurations, we summarised them in their appropriate sections.

Batch Size: For this experiment, we used the above-mentioned Conv2D operator for roofline analysis by varying batch sizes from 16, 32, and 64. As shown in Figure 3.5, the forward pass of the V100 GPU shows all batch sizes of single precision and half-precision L2 and HBM overlap, which means poor L2 cache data locality, but there is a gap between L1 and L2 that indicates there is less L1 cache misses. Half precision mode outperformed single precision for both (V100 and A100) forward pass and backward pass. In half-precision, batch sizes 16 and 32 perform less than batch size 64. V100 forward pass results are the same as A100 forward pass, but A100 gives better data locality of L1, L2, and HBM for single and half-precision. In A100 backward pass increasing batch size from 16 to 64, performance is not linearly increasing.

Strides: For this experiment, we use the base Conv2D operator and vary the strides from 2, 3, 4, and 5. The backward pass of V100, for single and half-precision stride size 2, gives more performance than others. There is more L1 cache misses on single precision. In half-precision stride sizes, 2 and 3 give good L1 cache data locality. Half-precision of A100 GPU, there is a good gap between L1, L2, and HBM. Therefore, it gives good data locality than V100. Strides 4 and 5 perform more than strides 2 and 3 in half-precision of A100 GPU.

Kernel: Now we vary the kernel size from 3x3, 5x5, and 7x7 in the base Conv2d operator. As shown in Figure 3.6, V100 backward pass single and half-precision give good data locality, and performance increases linearly with increasing the kernel size but A100 backward pass kernel 5x5 gives better performance than 7x7 in half-precision, but single precision performance increase linearly like V100 and it gives poor L1 cache data locality on single precision.

Filters: We use the base Conv2D operator for this experiment and vary the filter sizes to 56, 112, and 224. The backward pass A100 GPU half-precision performance increases linearly with the filters, whereas in single precision, filter size 112 gives more performance than filter size 224. Furthermore, in V100 backward pass, both single and half-precision performance increases linearly with filter size. As a result, both A100 and V100 backward pass give good data locality in half-precision, and some L1 caches miss on single precision. On the other hand, V100 forward pass has poor data locality and linear performance on single precision. Both A100 and V100 forward pass half-precision of filters 112 and 224 give the same level of performance but comparatively very high than filter 56.

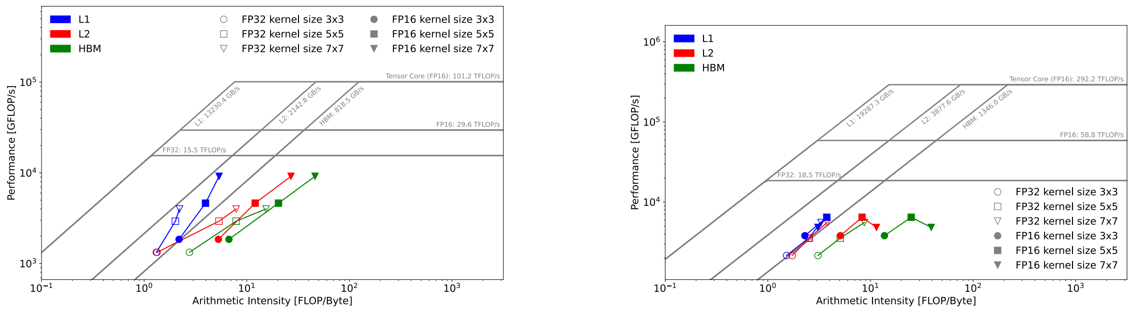
Summary :

- Increasing the batch size from 32 to 64 gives a significant performance boost.
- In the backward pass of A100, half-precision strides 4 and 5 give more performance than 2 and 3. whereas V100 backward pass (half and single precision) stride 2 provides higher performance than with 4 and 5.
- The V100 backward pass kernel size ($3 \times 3, 5 \times 5, 7 \times 7$) performance increases linearly in both half and single precision. Whereas half-precision of A100 backward pass of kernel size 5×5 gives more performance than 7×7.
- In A100, half-precision backward pass performance increases linearly with the number of filters (56, 112, 224). In single precision, 112 filters give more performance than 224 filters.

Comparatively, A100 offers better data locality of L1, L2, and HBM than V100, and more L1 cache miss on single precision than half precision. We can use batch size 64 and mixed precision rather than 32 batch size and single precision for training. Bottleneck findings of strides, kernel, and filters help the developer modify the MTD's model architecture to effectively utilize the hardware.

GPU	**V100**				**A100**			
Batch Size	**32**		**64**		**32**		**64**	
Precision	**Single**	**Mixed**	**Single**	**Mixed**	**Single**	**Mixed**	**Single**	**Mixed**
Data loading time (s)	8.87	17.29	13.42	12.02	5.08	5.14	5.05	5.58
Total run-time (s)	2847	1892	2756	1779	1530	1319	1450	1212
Total training time (s)	2805	1844	2706	1734	1507	1296	1427	1188
Avg. epoch time (s)	40.07	26.34	38.66	24.78	21.53	18.52	20.39	16.98
Training loss	0.0532	0.0535	0.0554	0.0551	0.0532	0.0536	0.0552	0.0556
Validation loss	0.0585	0.0586	0.0599	0.0591	0.0584	0.0601	0.0588	0.0594
Avg. GPU Usage %	90.53	81.86	91.38	83.55	22.38	19.27	22.46	22.02
Avg. Mem. Usage Rate %	63.24	44.41	66.13	47.52	14.06	7.25	14.94	10.72
Avg. GPU Temp °C	59.92	57.5	58.49	57.74	50.96	48.45	50.9	47.85
Avg. GPU Power W	259.13	244.14	252.35	238.49	116.01	98.73	116.99	107.51

Table 3.2: Single GPU Training with Mixed Precision and Single-Precision Mode on A100 and V100 GPUs.



(a) V100 Backward Pass

(b) A100 Backward Pass

Figure 3.6: Roofline Analysis of V100 and A100 GPUs Backward pass of MTD Convolutional Operator by varying kernel size.

3.4.2 Training Level

In single GPU training, we trained the MTD model with two precision modes, single precision, and mixed precision. As we already know, half-precision gives more performance than single precision for Conv2D operators from the operator level. The mixed precision contains both single and half-precision but is automatically selected by the deep learning framework. Our assumption is mixed precision will perform more than single precision. We will verify it from training-level results. MTD's model contains a total of 3525650 trainable parameters and 3360 non-trainable parameters with 7.5GiB training data and 0.95 GiB of validation data. We trained in V100 and A100 GPUs with batch sizes (32, 64) and 70 epochs. As results shown in Table 3.2, single precision of V100 using 32 batch sizes consumes a higher total training time. Whereas A100 mixed precision of 64 batch size consumes less training time than others. The training and validation loss of batch sizes 32 and 64 is almost the same for single precision and mixed precision on V100 and A100 GPUs. Mixed precision gives good training time per epoch than single precision mode. We exclude the first epoch time for the average training time per epoch calculation because the first epoch always takes higher time in single precision and mixed precision on both V100 and A100. While increasing the batch size from 32 to 64, there is only slight performance improvement for V100 and A100 in both modes.

The mixed precision of V100 gives more performance than single precision, whereas A100 gives only a slight marginal improvement. Because tensor cores are activated only half-precision in V100 but single and half-precision in A100. Mixed precision consumes less average GPU usage and average memory usage than single precision mode. V100 consumes more additional GPU power than A100 GPU. While using mixed precision in V100 and A100, GPU temperature and power consumption were also reduced.

Summary **:** Mixed precision training consumes less GPU power and memory than single precision training on both GPUs. V100 mixed precision of batch size 64 utilizes Tensor

Core at 89.5%, whereas A100 utilizes only 31.7% of TC.

3.4.3 Distributed Training Level

We conduct this experiment to find out that increasing the GPU for MTD model training will affect the training time. We used two nodes for this experiment JUWELS Booster and JUWELS Cluster. Due to the inadequate training data volume on the MTD model, we used a maximum of 4 GPUs for distributed training, and we increased GPU gradually from 1 to 4. We used TensorFlow mirrored strategy API for distributed training as discussed in section 3.3.1. We used only ten epochs and dynamic batch sizes (Batch size gradually increased with the number of GPUs). We already discussed in section 3.4.2 that mixed precision gives more performance than single precision, so we used mixed precision with Adam [46] optimizer for training. The loading data time is lesser in a single GPU than in multiple GPUs because data needs to transfer to multiple GPUs. These variations are higher in V100 than in A100. The average epoch time for training in V100 decreases significantly from using 1 GPU to 2 GPUs but later, using 3 and 4 GPUs does not give enough performance boost. Whereas in A100, using 1 GPU gives an average epoch time of 22.9 seconds, and using 2 GPUs gives 19.3, there is no substantial improvement in using multiple GPUs. Final training loss and validation loss are almost the same for using 1 and 2 GPUs for both V100 and A100, but with further increasing GPUs, there is a drop in training and validation losses because we increase batch size gradually with GPUs, batch size implicit connection with gradient estimator, so increasing batch size above 64 causes low variance.
Summary : Using four GPUs on V100 and A100 is not sufficiently utilizing all the GPUs in the node for training. V100 utilizes 81.8% of TC, whereas A100 only utilizes 26.9% of TC. Average GPU usage and memory usage decreases gradually with an increasing number of GPUs in V100. On average, A100 consumes less GPU energy than V100. Overall the MTD model does not sufficiently utilize the multiple GPUs for distributed training. We suggest using a single GPU for training the MTD model is sufficient.

3.4.4 Inference Level

The inference level calculates the time to complete the full forward propagation of the MTD model. Furthermore, we investigate the roofline analysis on the most time-consuming kernel during the inference. The MTD model network is explained in the section 3.2. We compare the results with V100 and A100 GPUs. We used single and mixed precision MTD models for the inference benchmark. Both the model trained with batch size 64. As shown in Table 3.3, the model is trained with mixed precision utilizing tensor core very well on V100 than the model trained with single precision. Whereas in A100, both single and mixed precision utilizes the tensor core because, as we already discussed in section 3.2.2 A100 will utilize the tensor core on both single and half-precision, but V100 only uses tensor core on half precision. The model's accuracy with the single and mixed precision modes is almost the same. In A100 mixed precision model took less time

Table 3.3: Inference Performance Metrics on A100 and V100 GPUs

Device	V100		A100	
Precision	Single	Mixed	Single	Mixed
Wallclock time (s)	0.09	0.051	0.10	0.037
TC utilization %	None	58.7	16.1	6.4
GPU utilization %	8.1	3.2	2.6	1.1

for inference than other variants.

Inference on the mixed precision model performs more than a single precision-trained model on both GPUs. So we will use mixed precision trained model for inference roofline analysis. If we perform roofline analysis for all the kernels during the inference will not be very effective in studying the system because there are too many kernel launches during inference. So first, we used the NSight System CLI tool to extract the kernel statistics, we identified **conv2d_grouped_direct_kernel** and **implicit_convolve_sgemm_kernel** took considerable GPU computing time than others. We also performed roofline analysis on the most time-consuming kernels. Both A100 and V100 conv2d_grouped kernel utilize only 1% of the device's FP32 Peak performance. V100 implicit_convolve_sgemm utilizes 10% device's FP32 Peak performance, and in A100, it utilizes 13% device's FP32 Peak performance. As a result, the MTD model inference in A100 is not giving a bigger performance boost than V100 GPU.

***Summary* :** For inference, A100 mixed-precision trained models provide 2.7 times more performance than A100 single precision and 2.4 times more than V100 single precision. After roofline analysis of the MTD model inference, both V100 and A100 GPUs do not effectively utilize the device's peak performance. The A100 GPU does not provide a significant performance increase compared to the V100 GPU.

***Suggestion* :** We can utilize the Nividia MIG technology to partition GPU into multiple instances and deploy the model into any one of the instances for inference. This technology is applicable only starting from Ampere architecture. For A100, there is seven MIG profile available. But we categorize into 4 profiles

- **7_5GB**: split GPU into 7 instances. Each one has 5 GB of memory
- **3_10GB**: split GPU into 3 instances. Each one has 10 GB of memory
- **2_20GB**: split GPU into 2 instances. Each one has 20 GB of memory
- **1_40GB**: split GPU into 1 instance. Each one has 40 GB of memory

If the model inference is utilizing the device less than 25%, we use **7_5GB** profile. If usage is 25% to 50%, we can use **3_10GB** profile. Likewise, we can use other profiles for inference. Our MTD model utlise only 1.1% of GPU for inference in A100, so we use **7_5GB** profile and split the A100 GPU into 7 instances by using the following command ***sudo nvidia-smi mig -cgi 19,7g.5gb -C*** and deploy the MTD model into any of the

Table 3.4: GPU Compute Cost Comparison with AWS, GCP, and AZURE

Cloud	Instance Type	Device	GPUs	Per Hour USD	Total USD
AWS	p4d.24xlarge	A100	8	32.77	120.92
	p3.2xlarge	V100	1	3.06	19.52
	p3.8xlarge		4	12.24	53.12
GCP	NVIDIA A100	A100	1	2.93	13.03
			4	11.74	43.32
	NVIDIA V100	V100	1	2.48	15.82
			4	9.92	43.05
AZURE	NC6s v3	V100	1	3.06	19.52
	NC24rs v3		4	13.46	58.41
	NC24ads v4	A100	1	3.67	16.33
	NC96ads v4		4	14.69	54.20

instances or on many instances and use a load balancer[7] for the inference server to effectively utilize the underlying A100 GPU.

3.4.5 Cloud Compute Cost

Training: We compare the public cloud computing cost of V100 and A100 (1 GPU and 4 GPU) on three cloud vendors AWS, GCP, and Azure. As we already gathered training time from training and distributed training level, V100 single GPU and four GPUs' average epoch times are 32.8s and 22.3s. In A100, a single GPU and four GPUs are 22.88s and 18.94s, respectively. At least 70 epochs are needed to get good accuracy of the MTD model. To get a fair comparison, we were going to calculate the computing cost for ten times training the model. So V100 single and four GPU total training time 6.38 hrs and 4.34 hrs. The total training time in A100 single and four GPU is 4.45 hrs and 3.69 hrs, respectively. As shown in Table 3.4, GCP gives the least computing cost for one A100 GPU than V100. AWS costs higher GPU costs than other vendors. A single A100 GPU from GCP is good for MTD model training.

Inference: As we discussed in section 3.4.4, both V100 and A100 GPUs give almost the same performance for the MTD model inference on test data. We checked MTD model inference for CPU to determine whether it will give low cost than GPU. We used AWS c5.4xlarge instance, which has Intel(R) Xeon(R) Platinum 8275CL CPU @ 3.00GHz CPU total of 16 cores for the MTD model trained with mixed-precision mode takes an average time of 5.15 seconds approx for inference which is slower than A100 inference time. GPUs outperform CPUs in terms of performance and cost for MTD model inference. For example, per hour cost for a c5.4xlarge instance is \$0.68 and GCP single A100 GPU per hour cost is \$2.93. If we perform 10^4 times inference c5.4xlarge instance cost is \$9.724

[7]https://www.envoyproxy.io/docs/envoy

whereas cost of GCP A100 is $0.301. We suggest using A100 for inference to leverage Nividia MIG technology to effectively utilize the hardware.

3.5 Conclusions

We provided a comprehensive methodology for profiling the MAELSTROM Temperature Downscaling (MTD) application starting from the operator level and identifying the bottleneck of the convolutional operator using roofline analysis and using the parameters to train the model and analyze the training and distribute training level. We profiled the MTD model inference to identify the device utilization, and we suggested a methodology to use Multi-Instance GPU for inference. Finally, we compared computing costs for training and inference from cloud vendors. Our results will help to choose the hardware configuration but also helps to enhance the MTD model architecture because we detailed operator bottlenecks for varying batch sizes, kernels, strides, and filters using roofline analysis. To the author's knowledge, our methodologies and results will help other researchers and developers enhance and profile their deep learning models.

4 DIPPM: a Deep Learning Inference Performance Predictive Model using Graph Neural Networks

Deep Learning (DL) has developed to become a corner-stone in many everyday applications that we are now relying on. However, making sure that the DL model uses the underlying hardware efficiently takes a lot of effort. Knowledge about inference characteristics can help to find the right match so that enough resources are given to the model, but not too much. We have developed a DL Inference Performance Predictive Model (DIPPM) that predicts the inference latency, energy, and memory usage of a given input DL model on the NVIDIA A100 GPU. We also devised an algorithm to suggest the appropriate A100 Multi-Instance GPU profile from the output of DIPPM. We developed a methodology to convert DL models expressed in multiple frameworks to a generalized graph structure that is used in DIPPM. It means DIPPM can parse input DL models from various frameworks. Our DIPPM can be used not only helps to find suitable hardware configurations but also helps to perform rapid design-space exploration for the inference performance of a model. We constructed a graph multi-regression dataset consisting of 10,508 different DL models to train and evaluate the performance of DIPPM, and reached a resulting Mean Absolute Percentage Error (MAPE) as low as 1.9%.

This chapter builds upon the research presented in the following publication:

Peer reviewed

Karthick Panner Selvam, and Mats Brorsson. " DIPPM: A Deep Learning Inference Performance Predictive Model Using Graph Neural Networks," Euro-Par 2023: Parallel Processing. Euro-Par 2023. Lecture Notes in Computer Science, vol 14100. Springer, Cham. DOI: 10.1007/978-3-031-39698-4_1.

Contents

4.1 Introduction

Many important tasks a now relying on Deep learning models, for instance in computer vision and natural language processing domains [47, 48]. In recent years, researchers have focused on improving the efficiency of deep learning models to reduce the computation cost, energy consumption and increase the throughput of them without losing their accuracy. At the same time, hardware manufacturers like NVIDIA increase their computing power. For example, the NVIDIA A100[1] GPU half-precision Tensor Core can perform matrix operations at 312 TFLOPS. But all deep learning models will not fully utilize the GPU because the workload and number of matrix operations will vary according to the problem domain. For this reason, NVIDIA created the Multi-Instance GPU (MIG[2]) technology starting from the Ampere architecture; they split the single physical GPU into multi-isolated GPU instances, so multiple applications can simultaneously run on different partitions of the same GPU, which then can be used more efficiently.

However, determining the DL model's efficiency on a GPU is not straightforward. If we could predict parameters such as inference latency, energy consumption, and memory usage, we would not need to measure them on deployed models which is a tedious and costly process. The predicted parameters could then also support efficient Neural Architecture Search (NAS) [49], efficient DL model design during development, and avoid job scheduling failures in data centers. According to Gao et al. [50], most failed deep learning jobs in data centers are due to out-of-memory errors.

In order to meet this need, we have developed a novel *Deep Learning Inference Performance Predictive Model* (DIPPM) to support DL model developers in matching their models to the underlying hardware for inference. As shown in Figure 4.1, DIPPM takes a deep learning model expressed in any of the frameworks: PyTorch, PaddlePaddle, Tensorflow, or ONNX, and will predict the latency (ms), energy (J), memory requirement (MB), and MIG profile for inference on an Nvidia A100 GPU without running on it. At the moment, the model is restricted to inference and the Nvidia A100 architecture, but we aim to relax these restrictions in future work. As far as we are aware, this is the first predictive model that can take input from any of the mentioned frameworks and to predict all of the metrics above.

Our contributions include the following:

- We have developed, trained and evaluated a performance predictive model which predicts inference latency, energy, memory, and MIG profile for A100 GPU with high accuracy.
- We have developed a methodology to convert deep learning models from various deep learning frameworks into generalized graph structures for graph learning tasks in our performance predictive model.

[1]`https://www.nvidia.com/en-us/data-center/a100/`

[2]`https://docs.nvidia.com/datacenter/tesla/mig-user-guide/`

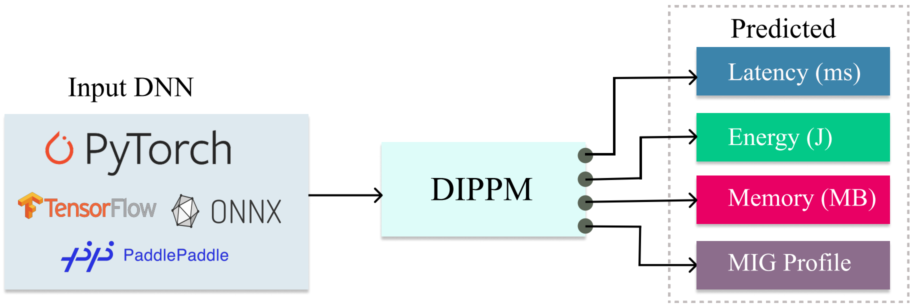


Figure 4.1: DIPPM can predict the Latency, Energy, Memory requirement, and MIG Profile for inference on an NVIDIA A100 GPU without actually running on it.

- We have devised an algorithm to suggest the MIG profile from predicted Memory for the given input DL model.
- We have created an open-sourced performance predictive model dataset containing 10,508 graphs for graph-level multi-regression problems.

Next, we discuss our work in relation to previous work in this area before presenting our methodology, experiments, and results.

4.2 Related Work

Performance prediction of deep learning models on modern architecture is a rather new research field being attended to only since a couple of years back. Bouhali et al. [10] and Lu et al. [11] have carried out similar studies where a classical Multi-Layer Perceptron (MLP) is used to predict the inference latency of a given input DL model. Their approach was to collect high-level DL model features such as batch size, number of layers, and the total number of floating point operations (FLOPS) needed. They then fed these features into an MLP regressor as input to predict the latency of the given model. Bai et al. [12] used the same MLP method but predicted both the latency and memory. However, the classical MLP approach did not work very well due to the inability to capture a detailed view of the given input DL model.

To solve the above problems, some researchers came up with a kernel additive method; they predict each kernel operation, such as convolution, dense, and LSTM, individually and sum up all kernel values to predict the overall performance of the DL model [13–18]. Yu et al. [51] used the wave-scaling technique to predict the inference latency of the DL model on GPU, but this technique requires access to a GPU in order to make the prediction.

Kaufman et al. and Dudziak et al. [19, 20] used graph learning instead of MLP to predict each kernel value. Still, they used the kernel additive method for inference latency prediction. However, this kernel additive method did not capture the overall network topology of the model, and instead it will affect the accuracy of the prediction. To solve the above problem, Liu et al. [21] used a Graph level task to generalize the entire DL model into node embeddings and predicted the inference latency of the given DL model. However, they did not predict other parameters, such as memory usage and energy consumption. Gao et al. [52] used the same graph-level task to predict the single iteration time and memory consumption for deep learning training but not for inference.

Li et al. [53] tried to predict the MIG profiles on A100 GPU for the DL models. However, their methodology is not straightforward; they used CUDA Multi-Process Service (MPS) values to predict the MIG, So the model must run at least on the target hardware once to predict the MIG Profile.

Most of the previous research work concentrated on parsing the input DL model from only one of the following frameworks (PyTorch, TensorFlow, PaddlePaddle, ONNX). As far as we are aware, none of the previous performance prediction models predicted Memory usage, Latency, Energy, and MIG profile simultaneously.

Our novel Deep Learning Inference Performance Predictive Model (DIPPM) fills a gap in previous work; a detailed comparison is shown in Table 4.1. DIPPM takes a deep learning model as input from various deep learning frameworks such as PyTorch, PaddlePaddle, TensorFlow, or ONNX and converts it to generalize graph with node features. We used a graph neural network and MIG predictor to predict the inference latency (ms), energy (J), memory (MB), and MIG profile for A100 GPU without actually running on it.

4.3 Methodology

The architecture of DIPPM consists of five main components: Deep Learning Model into Relay IR, Node Feature Generator, Static Feature Generator, Performance Model Graph Network Structure (PMGNS), and MIG Predictor, as shown in Figure. 4.2. We will explain each component individually in this section.

4.3.1 Deep Learning Model into Relay IR

The Relay Parser takes as input a DL model expressed in one of several supported DL frameworks, converts it to an Intermediate Representation (IR), and passes this IR into the Node Feature Generator and the Static Feature Generator components.

Most of the previously proposed performance models are able to parse the given input DL model from a single DL framework, not from several, as we already discussed in Section 4.2. To enable the use of multiple frameworks, we used a relay, which is a high-level IR for DL models [54]. It has been used to compile DL models for inference in the TVM[3] framework.

We are inspired by the approach of converting DL models from different frameworks into a high-level intermediate representation (IR), so we incorporated their techniques into our architecture. However, we couldn't directly employ relay IR in DIPPM. To

[3]`https://tvm.apache.org/`

Table 4.1: Related Work comparison

Related Works	**A100**	**MIG**	**GNN**[a]	**Multi-SF**[b]	**Latency**	**Power**	**Memory**
Ours (**DIPPM**)	✓	✓	✓	✓	✓	✓	✓
Bai et al. [12]	-	-	-	-	✓	-	✓
Bouhali et al. [10]	-	-	-	-	✓	-	-
Dudziak et al. [20]	-	-	✓	-	✓	-	-
Gao et al. [52]	-	-	✓	-	✓	-	✓
Justus et al. [13]	-	-	-	-	✓	-	-
Kaufman et al. [19]	-	-	✓	-	✓	-	-
Li et al. [53]	✓	✓	-	-	-	-	-
Liu et al. [21]	-	-	✓	-	✓	-	-
Lu et al. [11]	-	-	-	-	✓	✓	✓
Qi et al. [14]	-	-	-	-	✓	-	-
Sponner et al. [15]	✓	-	-	-	✓	✓	✓
Wang et al. [16]	-	-	-	-	✓	-	-
Yang et al. [17]	-	-	-	-	✓	-	-
Yu et. al. [51]	✓	-	-	-	✓	-	-
Zhang et al. [18]	-	-	-	-	✓	-	-

[a]Using Graph Neural Network for performance prediction
[b]Able to parse DL model expressed in Multiple DL Software Framework

overcome this, we developed a method explained in Section 4.3.2. It involves parsing the Relay IR and transforming it into a graph representation with node features.

It allows parsing given input DL models from various frameworks, including PyTorch, TensorFlow, ONNX, and PaddlePaddle. However, for the purposes of this study, we have focused on the implementation and evaluation of the framework specifically within the PyTorch environment. We pass this DL IR to the subsequent components in our DIPPM architecture.

4.3.2 Node Feature Generator

The Node Feature Generator (NFG) converts the DL IR into an Adjacency Matrix ($\mathcal{A}$) and a Node feature matrix ($\mathcal{X}$) and passes this data to the PMGNS component.

The NFG takes the IR from the relay parser component. The IR is itself a computational data flow graph containing more information than is needed for our performance prediction. Therefore we filter and pre-process the graph by post-order graph traversal to collect necessary node information. The nodes in the IR contain useful features such as operator name, attributes, and output shape of the operator, which after this first filtering step are converted into a suitable data format for our performance prediction. In the subsequent step, we loop through the nodes and, for each operator node, generate node

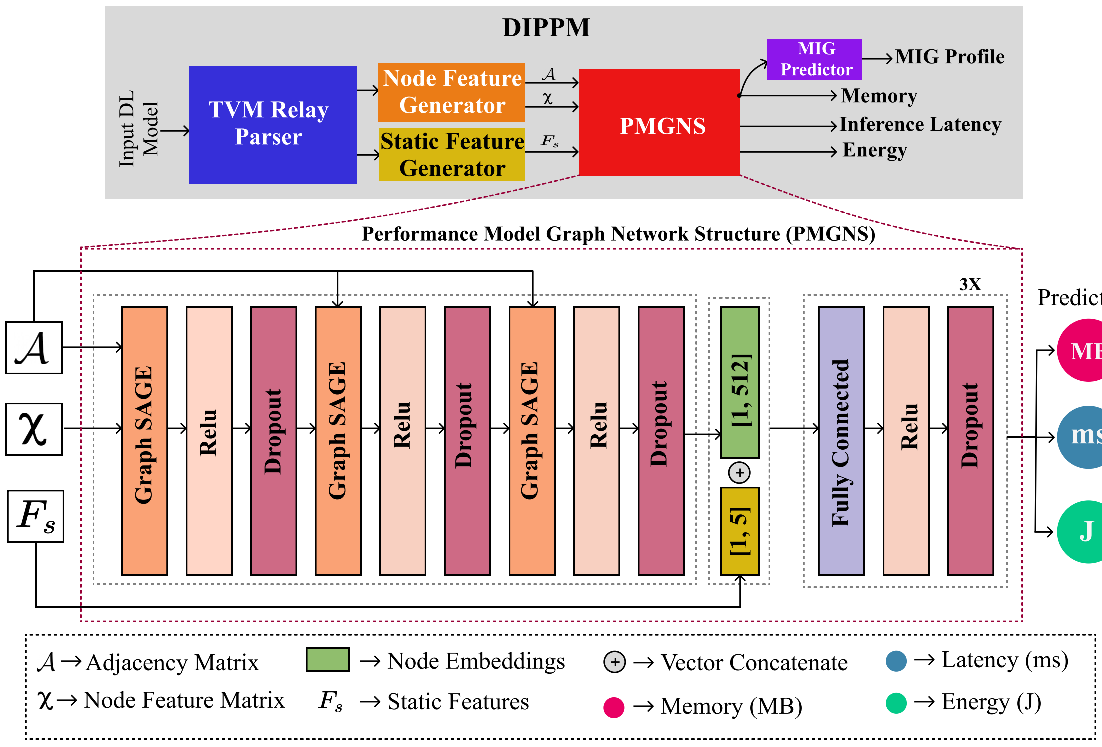


Figure 4.2: Overview of DIPPM Architecture

features Fnode with a fixed length of 32 as discussed on line 7 in Algorithm 1.

The central part of the NFG is to generate an **Adjacency Matrix** ($\mathcal{A}$) and a **Node feature matrix** ($\mathcal{X}$) as expressed in Algorithm 1. $\mathcal{X}$ has the shape of $[N_{op}, N_{features}]$, where N_{op} is the number of operator nodes in the IR and $N_{features}$ is the number of features. In order to create node features $\mathcal{F}_n$ for each *node*, first, we need to encode the node operator name into a one hot encoding as can be seen on line 4 in Algorithm 1. Then extract the node attributes $\mathcal{F}_{attr}$ and output shape $\mathcal{F}_{shape}$ into vectors. Finally, perform vector concatenation to generate $\mathcal{F}_n$ for a node. We repeat this operation for each node and create the $\mathcal{G}$. From the $\mathcal{G}$, we extract $\mathcal{A}$, $\mathcal{X}$ that are passed to the main part of our model, the Performance Model Graph Network Structure.

4.3.3 Static Feature Generator

The Static Feature Generator (SFG) takes the IR from the relay parser component and generates static features $\mathcal{F}_s$ for a given DL model and passes them into the graph network structure.

For this experiment, we limited ourselves to five static features. First, we calculate the $\mathcal{F}_{mac}$ total multiply-accumulate (MACs) of the given DL model. We used the TVM relay analysis API to calculate total MACs, but it is limited to calculating MACs for the following

Algorithm 1: Algorithm to convert DL model IR into a graph with node features

CreateGraph takes input IR and filters it by post-order traversal. Collect node features for each node and generate a new graph $\mathcal{G}$ with node features, finally extract node feature matrix $\mathcal{X}$ and adjacency matrix $\mathcal{A}$ from $\mathcal{G}$.

```
1:  function CreatGraph($IR$)                              ▷ IR from Relay Parser Component
2:      $\mathcal{N} \leftarrow filter\_and\_preprocess(IR)$
3:      $\mathcal{G} \leftarrow \emptyset$          ▷ Create empty directed graph foreach $node \in \mathcal{N}$ do
_           where $node$ is node in node_list $\mathcal{N}$ if $node.op \in$ [operators] then
_           Check node is an operator
4:      $\mathcal{F}_{oh} \leftarrow one\_hot\_encoder(node.op)$
5:      $\mathcal{F}_{attr} \leftarrow ExtractAttributes(node)$
6:      $\mathcal{F}_{shape} \leftarrow ExtractOutshape(node)$
7:      $\mathcal{F}_{node} \leftarrow \mathcal{F}_{oh} \oplus \mathcal{F}_{attr} \oplus \mathcal{F}_{shape}$
8:      $\mathcal{G}.add\_node(node.id, \mathcal{F}_{node})$               ▷ Nodes are added in sequence
9:
10:
11:     $\mathcal{A} \leftarrow GetAdjacencyMatrix(\mathcal{G})$
12:     $\mathcal{X} \leftarrow GetNodeFeatureMatrix(\mathcal{G})$
13:     return $\mathcal{A}, \mathcal{X}$
14: end function
```

operators (in TVM notation): Conv2D, Conv2D transpose, dense, and batch matmul. Then we calculate the total number of convolutions F_{Tconv}, Dense F_{Tdense}, and Relu F_{Trelu} operators from the IR. We included batch size F_{batch} as one of the static features because it gives the ability to predict values for various batch sizes of a given model. Finally, we concatenate all the features into a vector $\mathcal{F}_s$ as expressed in equation 4.1. The feature set $\mathcal{F}_s$ is subsequently passed to the following graph network structure.

$$\mathcal{F}_s \leftarrow \mathcal{F}_{mac} \oplus \mathcal{F}_{batch} \oplus \mathcal{F}_{Tconv} \oplus \mathcal{F}_{Tdense} \oplus \mathcal{F}_{Trelu} \tag{4.1}$$

4.3.4 Performance Model Graph Network Structure

The PMGNS takes the node feature matrix ($\mathcal{X}$), the adjacency matrix ($\mathcal{A}$) from the Node Feature Generator component, and the feature set ($\mathcal{F}_s$) from the Static feature generator and predicts the given input DL model's memory, latency, and energy, as shown in Figure. 4.2.

The PMGNS must be trained before prediction, as explained in section 4.4. The core idea of the PMGNS is to generate the node embedding z from $\mathcal{X}$ and $\mathcal{A}$ and then to perform vector concatenation of z with $\mathcal{F}_s$. Finally, we pass the concatenated vector into a Fully Connected layer for prediction, as shown in Figure. 4.2. In order to generate

z, we used the graphSAGE algorithm suggested by Hamilton et al. [55], because of its inductive node embedding, which means it can generate embedding for unseen nodes without pretraining. GraphSAGE is a graph neural network framework that learns node embeddings in large-scale graphs. It performs inductive learning, generalizing to unseen nodes by aggregating information from nodes and neighbors. It generates fixed-size embeddings, capturing features and local graph structure. With a neighborhood aggregation scheme, it creates node embeddings sensitive to their local neighborhood, even for new, unobserved nodes.

We already discussed that we generate node features of each node in the Section 4.3.2. The graphSAGE algorithm will convert node features into a node embedding z which is more amenable for model training. The PMGNS contains three sequential graphSAGE blocks and three sequential Fully connected (FC) blocks as shown in Figure. 4.2. At the end of the final graphSAGE block, we get the generalized node embedding of given $\mathcal{X}$ and $\mathcal{A}$, which we concatenate with $\mathcal{F}_s$. Then we pass the concatenated vector into FC to predict the memory (MB), latency (ms), and energy (J).

4.3.5 MIG Predictor

The MIG predictor takes the memory prediction from PMGNS and predicts the appropriate MIG profile for a given DL model, as shown in Figure. 4.2.

As mentioned in the introduction, the Multi-instance GPU (MIG) technology allows to split an A100 GPU into multiple instances so that multiple applications can use the GPU simultaneously. The different instances differ in their compute capability and, most importantly, in the maximum memory limit that is allowed to be used. The four MIG profiles of the A100 GPU that we consider here are: 1g.5gb, 2g.10gb, 3g.20gb, and 7g.40gb, where the number in front of "gb" denotes the maximum amount of memory in GB that the application can use on that instance. For example, the maximum memory limit of 1g.5gb is 5GB, and 7g.40gb is 40GB. For a given input DL model, PMGNS predicts memory for 7g.40gb MIG profile, which is the full GPU. We found that this prediction can be used as a pessimistic value to guide the choice of MIG profile. Figure. 4.3 shows manual memory consumption measurements of the same DL model inference on different profiles. The results show no significant difference in the memory allocation of DL in the different MIG profiles even though the consumption slightly increases with the capacity of the MIG profile. The memory consumption is always the highest when running on the 7g.40gb MIG profile.

As mentioned, PMGNS predicts memory for 7g.40gb, so we claim that predicted memory will be an upper bound. Then we perform a rule-based prediction to predict the MIG profile for the given input DL model, as shown in equation 4.2. Where α is predicted memory from PMGNS.

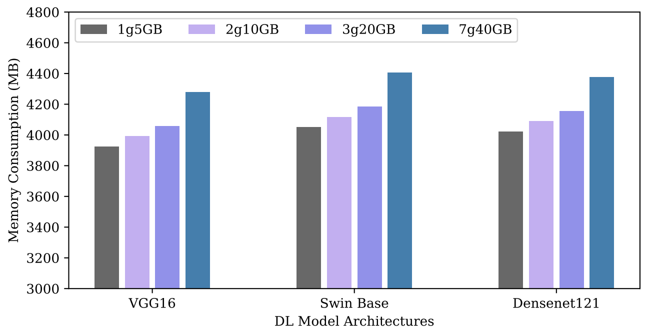


Figure 4.3: MIG Profile comparison of three different DL models memory consumption on A100 GPU. We used batch size 16 for VGG16 and Densenet121 model and batch size 8 for Swin base model.

$$\text{MIG}(\alpha) = \begin{cases} \text{1g.5gb}, & \text{if } 0gb < \alpha < \text{5gb} \\ \text{2g.10gb}, & \text{if 5gb} < \alpha < \text{10gb} \\ \text{3g.20gb}, & \text{if 10gb} < \alpha < \text{20gb} \\ \text{7g.40gb}, & \text{if 20gb} < \alpha < \text{40gb} \\ \text{None}, & \text{otherwise} \end{cases} \tag{4.2}$$

4.4 Experiments & Results

4.4.1 The DIPPM Dataset

We constructed a graph-level multi-regression dataset containing 10,508 DL models from different model families to train and evaluate our DIPPM. The dataset distribution is shown in Table 5.1. To the best of our knowledge, the previous predictive performance model dataset doesn't capture memory consumption, inference latency, and energy consumption parameters for wide-range DL models on A100 GPU so we created our own dataset for performance prediction of DL models.

Our dataset consists of DL models represented in graph structure, as generated by the Relay parser described in Section 4.3.1. Each data point consists of four variables: $\mathcal{X}$, $\mathcal{A}$, $\mathcal{Y}$, and $\mathcal{F}_s$, where $\mathcal{X}$ and $\mathcal{A}$ are the Node feature matrix and Adjacency Matrix, respectively, as discussed in Section 4.3.2, and $\mathcal{F}_s$ is the static features of the DL model as discussed in Section 4.3.3. We used the Nvidia Management Library[4] and the CUDA toolkit[5] to measure the energy, memory, and inference latency of each given model in the dataset. For each model, we ran the inference five times to warm up the architecture and then the inference 30 times, and then took the arithmetic mean of those 30 values

[4]`https://developer.nvidia.com/nvidia-management-library-nvml`
[5]`https://developer.nvidia.com/cuda-toolkit`

Table 4.2: DIPPM Graph dataset distribution

Model Family	**# of Graphs**	**Percentage (%)**
Efficientnet	1729	16.45
Mnasnet	1001	9.53
Mobilenet	1591	15.14
Resnet	1152	10.96
Vgg	1536	14.62
Swin	547	5.21
Vit	520	4.95
Densenet	768	7.31
Visformer	768	7.31
Poolformer	896	8.53
Total	10 508	100%

to derive the $\mathcal{Y}$, where $\mathcal{Y}$ consists of inference latency (ms), memory usage (MB), and energy (J) for a given DL on A100 GPU. We used a full A100 40GB GPU, or it is equivalent to using 7g.40gb MIG profile to collect all the metrics.

4.4.2 Enviroment setup

We used an HPC cluster at the Jülich research centre in Germany called JUWELS Booster for our experiments[6]. It is equipped with 936 nodes, each with AMD EPYC 7402 processors, 2 sockets per node, 24 cores per socket, 512 GB DDR4-3200 RAM and 4 NVIDIA A100 Tensor Core GPUs with 40 GB HBM.

The main software packages used in the experiments are: Python 3.10, CUDA 11.7 torch 1.13.1, torch-geometric 2.2.0, torch-scatter 2.1.0, and torch-sparse 0.6.16.

4.4.3 Evaluation

The Performance Model Graph Network Structure is the main component in DIPPM, and we used the PyTorch geometric library to create our model, as shown in Figure. 4.2. We split our constructed dataset into three parts randomly: training set 70%, validation set 15%, and a test set 15%.

In order to validate that graphSAGE performs better than other GNN algorithms and plain MLP, we compared graphSAGE with the following other algorithms:, GAT [56], GCN [57], GIN [58], and finally, plain MLP without GNN. Table 4.3 summarizes the settings used. The learning rate was determined using a learning rate finder as suggested by Smith [59]. The Huber loss function achieved a higher accuracy than mean square error, which is why we chose that one. For the initial experiment, we trained for 10 epochs and used Mean Average Percentage Error (MAPE) as an accuracy metric to

[6]`https://apps.fz-juelich.de/jsc/hps/juwels/booster-overview.html`

Table 4.3: Settings in GNN comparison.

Setting	**Value**
Dataset partition	Train (70%) / Validation (15%) / Test (15%)
Nr hidden layers	512
Dropout probability	0.05
Optimizer	Adam
Learning rate	$2.754 \cdot 10^{-5}$
Loss function	Huber

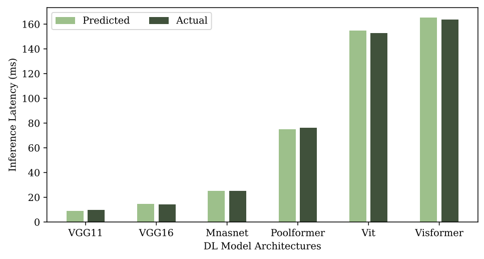
Table 4.4: Comparison with different GNN algorithms and MLP with graphSAGE, we trained all the models for 10 epochs and used Mean Average Percentage Error for validation. The results indicate that DIPPM with graphSAGE performs significantly better than other variants.

Model	**Training**	**Validation**	**Test**
GAT	0.497	0.379	0.367
GCN	0.212	0.178	0.175
GIN	0.488	0.394	0.382
MLP	0.371	0.387	0.366
(Ours) GraphSAGE	**0.182**	**0.159**	**0.160**

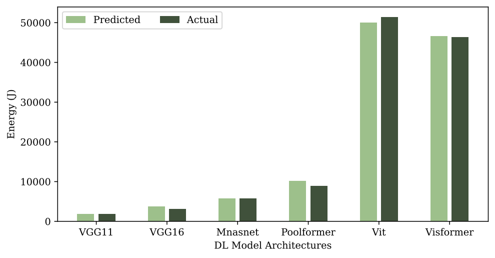
validate DIPPM. A MAPE value close to zero indicates good performance on regression prediction. Table 4.4 shows that graphSAGE gives a lower MAPE value in all of the training, validation, and test datasets. Without using a GNN, MLP gives 0.366 of MAPE. With graphSAGE, MAPE is 0.160 on the test dataset which is a significant improvement on a multi-regression problem. We conclude that graphSAGE outperforms other GNN algorithms, and MLP because of its inductive learning, as discussed in section 4.3.4. After this encouraging result we increased the number of epochs for training our DIPPM with graphSAGE to increase the prediction accuracy. After 500 epochs, we attained MAPE of 0.041 on training and 0.023 on the validation dataset. In the end, we attained 1.9% MAPE on the test dataset. Some of the DIPPM predictions on the test dataset are shown in Figure. 4.4.

4.4.4 Prediction of MIG Profiles

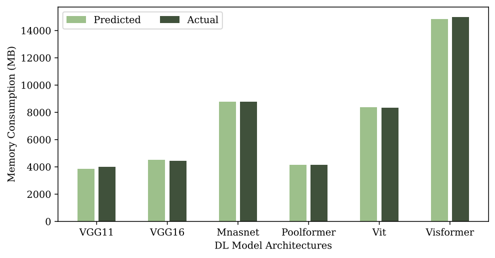
In order to verify the MIG profile prediction for a given DL model, we compared the actual MIG profile value with the predicted MIG profile from the DIPPM, as shown in Table 4.5. To calculate the actual suitable MIG profile, we divide actual memory consumption by the maximum memory limit of the MIG profiles. The higher the value is, the more appropriate profile for the given DL model. For example, the predicted memory



(a) Inference latency (ms).



(b) Energy (J).



(c) Memory consumption (MB).

Figure 4.4: Comparison of actual value with DIPPM predicted values on the test dataset. Results show that DIPPM predictions are close to the actual predictions.

consumption for densenet121 at batch size 8 is 2865 MB. The actual memory consumption for the 7g.40gb MIG profile is 3272 MB. The actual memory consumption of 1g.5GB is 2918 MB, the percentage is 58%. Which is higher than other MIG profiles. Results show that DIPPM correctly predicted the MIG profile 1g.5gb for densenet121. It is interesting to note that the densent121 models are from our test dataset and the swin base patch4 model is not in our DIPPM dataset but a similar swin base model family was used to train DIPPM. The convnext models are completely unseen to our DIPPM, but it's still predicting the MIG profile correctly.

4.4.5 DIPPM Usability aspects

DIPPM takes basic parameters like frameworks, model path, batch, and input size, and finally, device type. As of now, we only considered A100 GPU; we are working to extend DIPPM to various hardware platforms. With a simple python API call, DIPPM predicts memory, latency, energy, and MIG profile for the given model, as can be seen in Figure. 4.5.

```
import dippm
import torchvision

model = (torchvision.models.vgg16(pretrained=True)).eval()

predicted = dippm.predict(model, batch=8, input="3,244,244", device="A100")
print("Memory {0} MB, Energy {1} J, Latency {2} ms, MIG{3}".format(*predicted))
```

Figure 4.5: An example code demonstrating the utilization of DIPPM for performance prediction of a VGG16 deep learning model with a batch size of 8.

4.5 Conclusion

We have developed a novel Deep Learning (DL) Inference Performance Predictive Model (DIPPM) to predict the inference latency, energy, and memory consumption of a given input DL model on an A100 GPU without running on it. Furthermore, We devised an algorithm to select the appropriate MIG profile from the memory consumption predicted by DIPPM. The model includes a methodology to convert the DL model represented in various frameworks to a generalized graph structure for performance prediction. To the best of our knowledge, DIPPM can help to develop an efficient DL model to utilize the underlying GPU effectively. Furthermore, we constructed and open-sourced[7] a multi-regression graph dataset containing 10,508 DL models for performance prediction. It can even be used to evaluate other graph-based multi-regression GNN algorithms. Finally, we achieved 1.9% MAPE on our dataset.

[7]https://github.com/karthickai/dippm

Table 4.5: DIPPM MIG profile prediction for seen and unseen DL model architectures. (densenet*: seen, swin*: partially seen, convnext*: unseen).

Model	**Batchsize**	**Predicted**		**Actual**				
		MIG	**Mem**	**Mem**	**1g.5gb**	**2g.10gb**	**3g.20gb**	**7g.40gb**
densenet121	8	1g.5gb	2865	3272	**58%**	30%	15%	8%
densenet121	32	2g.10gb	5952	6294		**60%**	30%	16%
swin_base_patch4	2	1g.5gb	2873	2944	**52%**	27%	14%	7%
swin_base_patch4	16	2g.10gb	6736	6156		**59%**	30%	15%
convnext_base	4	1g.5gb	4771	1652	**61%**	31%	16%	8%
convnext_base	128	7g.40gb	26439	30996				**77%**

5 Can Semi-Supervised Learning Improve Prediction of Deep Learning Model Resource Consumption?

As computational demands for deep learning models escalate, accurately predicting training characteristics like training time and memory usage has become crucial. These predictions are essential for optimal hardware resource allocation. Traditional performance prediction methods primarily rely on supervised learning paradigms. Our novel approach, TraPPM (Training characteristics Performance Predictive Model), combines the strengths of unsupervised and supervised learning to enhance prediction accuracy. We use an unsupervised Graph Neural Network (GNN) to extract complex graph representations from unlabeled deep learning architectures. These representations are then integrated with a sophisticated, supervised GNN-based performance regressor. Our hybrid model excels in predicting training characteristics with greater precision. Through empirical evaluation using the Mean Absolute Percentage Error (MAPE) metric, TraPPM demonstrates notable efficacy. The model achieves a MAPE of 9.51% for predicting training step duration and 4.92% for memory usage estimation. These results affirm TraPPM's enhanced predictive accuracy, significantly surpassing traditional supervised prediction methods.

This chapter builds upon the research presented in the following publication:

Peer reviewed

Karthick Panner Selvam, and Mats Brorsson. “Can Semi-Supervised Learning Improve Prediction of Deep Learning Model Resource Consumption?” International Journal of Advanced Computer Science and Applications(IJACSA), 15(6), 2024. DOI: 10.14569/IJACSA.2024.0150610.

This research was also presented at the Machine Learning for Systems Workshop co-located with the 37th NeurIPS Conference, 2023, in New Orleans, LA, USA.

Contents

5.1 Introduction

Deep learning (DL) has significantly advanced various fields by analyzing complex patterns in extensive datasets. The escalating complexity of DL models, driven by advances in computational resources and data availability, necessitates increased memory and computational power for training. This heightened demand complicates the training process and increases costs. Accurately predicting both memory consumption and step time for DL models is challenging due to a variety of hidden factors, including the choice of convolutional algorithms, garbage collection mechanisms, memory pre-allocation strategies, and the specifics of implementations like cuDNN[60]. These factors complicate the task of making precise predictions, highlighting the need for sophisticated approaches to accurately estimate these critical training characteristics. Effective prediction of memory and step time is not only essential for preventing out-of-memory errors but also plays a crucial role in optimizing resource allocation and enhancing the effectiveness of neural architectural search (NAS), ultimately enhancing the efficiency of the model development process

In past studies, researchers primarily employed supervised Multi-Layer Perceptron (MLP) and GNNs to predict the training and inference attributes of DL models [13, 19, 20, 25, 51, 52, 60–63]. These methods, while effective, are confined to the limits of supervised learning and do not fully exploit the potential of unlabeled data, which can significantly enhance prediction performance.

In response to this gap, we introduce TraPPM, a novel approach that leverages semi-supervised learning. First, we utilize unsupervised GNN to learn graph representations from an unlabeled dataset. GNNs are adept at capturing patterns and relationships within graph-structured data. Next, we combine the learned graph representations with static features of a DL model. With this integrated vector, we train the supervised GNN-based performance regressor using a labeled dataset, allowing it to accurately estimate the training step time and memory usage of a given DL model. Utilizing an embedding generated from unsupervised learning in conjunction with supervised training boosts performance prediction accuracy compared to relying solely on supervised training. Our key contributions include the following:

- We have implemented TraPPM, a novel methodology that leverages the unsupervised GNN for learning embeddings from unlabelled datasets. And combine the embedding with DL static features to train the GNN-based regressor model using a labeled dataset to predict the training characteristics without running it on target hardware.
- We rigorously assessed the performance of TraPPM against state-of-the-art baselines, including supervised GNN, MLP, and GBoost, TraPPM exhibits superior performance, achieving a remarkable 910 MB RMSE and 4.92% MAPE for memory and 23 ms RMSE and 9.51% MAPE for step-time prediction. This superior performance underscores the efficacy of harnessing unlabeled data for performance

prediction.

- Furthermore, our comprehensive dataset, encompassing 8,079 labeled graphs and 25,053 unlabelled graphs from various DL model families, presents a substantial contribution to the community, paving the way for future research in performance prediction and optimization.

5.2 Background

5.2.0.1 Computational Graphs

Deep learning models are usually represented as directed computational graphs, where each node represents mathematical operations, such as matrix multiplication, and edges represent the data flow between these nodes. For example, a simple Convolutional Neural Network (CNN) model. The image data is fed into the network via the input node, and it just passes data to the next node. The Conv nodes perform convolution operations on the input image data. The Pooling node is responsible for reducing the spatial dimensions of the input data to reduce computational requirements. The Fully Connected (FC) node, where each neuron is interconnected with all neurons from the previous layer, applies the activation function to a weighted sum of their inputs. Finally, the output node takes the data from the FC node and provides prediction results.

5.2.0.2 Graph Neural Networks

GNNs constitute a specialized class of deep learning models that operate on graph-structured data, denoted as $\mathcal{G} = (V, E)$, where V represents the set of nodes and E represents the set of edges in the graph. Each node $v_i \in V$ is associated with a feature vector, which encodes information about that node. The fundamental principle underlying GNNs is the iterative process known as message passing, which facilitates the generation of embeddings for nodes or entire graphs.

In the message passing process, each node v_i updates its embedding by aggregating information from its neighboring nodes. This aggregation is achieved through functions such as summation, averaging, or more intricate operations like neural networks or attention mechanisms. Let $\mathbf{h}_i^{(l)}$ denote the embedding of node v_i after l message passing iterations, where l represents the layer in the GNN. Initially, $\mathbf{h}_i^{(0)}$ corresponds to the node's original feature vector.

The update equation for node v_i at layer l in a GNN can be expressed as follows:

$$\mathbf{h}_i^{(l)} = \mathrm{TRANSFORM}\left(\mathbf{h}_i^{(l-1)}, \left\{\mathbf{h}_j^{(l-1)} : v_j \in \mathcal{N}(v_i)\right\}\right)$$

Here, $\mathbf{h}_j^{(l-1)}$ represents the embeddings of the neighboring nodes of v_i at the $(l-1)$-th layer, and $\mathcal{N}(v_i)$ denotes the set of neighbors of node v_i. The `TRANSFORM` function combines the embeddings of the node's neighbors with its own embedding from the previous layer. Through multiple layers of message passing, each node gathers information from an increasingly wider neighborhood in the graph. Thus, the final embedding $\mathbf{h}_i^{(l)}$ for node v_i after l layers encapsulates information from both its immediate and more distant neighbors within the graph. GNNs have demonstrated remarkable success in various graph-related tasks, including node classification, link prediction, and graph-level classification. Prominent GNN variants such as GraphSAGE [55], Graph Attention Networks (GAT) [64], and Graph Convolutional Networks (GCN) [65] have gained widespread adoption in the research community and have yielded state-of-the-art results in these tasks.

5.2.0.3 Graph Auto Encoders

GAEs [22], play a critical role in unsupervised learning with graphs. They are particularly useful when we have a lot of unlabeled data. A GAE comprises two essential parts: an encoder and a decoder. The encoder's role is to transform the input graph into lower-dimensional representations known as *embeddings* of nodes. This is often accomplished with a Graph Convolution Network (GCN), converting the input adjacency matrix A and feature matrix X into an embedding matrix Z. Where the adjacency matrix represents the connectivity between nodes in a graph. This can be written as $Z = encoder(X, A)$. The decoder takes the node embeddings produced by the encoder, the matrix Z, and tries to rebuild the original adjacency matrix. A common way of achieving this is using the node embeddings' inner product as the decoder function. The motivation here is that the inner product, as a similarity measure, can capture the likelihood of a link between two nodes. $A' = decoder(Z)$. The effectiveness of this transformation is evaluated using a loss function. This function measures the reconstruction error - the difference between the original adjacency matrix A and the reconstructed one A'. This discrepancy is usually calculated using a method like Binary Cross Entropy (BCE). The model is trained to minimize this loss, thus improving the GAE's precision. Having an established foundational understanding of DL as a computational graph and GAE, we can now delve into TraPPM's methodology. TraPPM leverages unsupervised GNN, particularly with a GAE, to learn the graph representation of unlabelled datasets. We utilize the computation graph as input to the TraPPM, with nodes representing operators and node features corresponding to operator attributes. The edges signify the connections between operators. We will explore it further in Section 5.4.

5.3 Related Work

The study of performance prediction of deep learning models is relatively recent, having only started to receive focus just a few years ago. Qi et al. [66] use a straightforward approach to estimate the training time of DL models, layer by layer, using an analytical model. They calculated each step duration and summed it to calculate the overall estimation. The model presumes no concurrent operations, which may only be accurate for some hardware types. Gao et al. [60] also used an analytical model to predict the memory consumption for the training DL model. Bouhali et al. [10] used an MLP-based regressor to predict the execution time of a DL model. They used input features such as trainable parameter count, memory size, and input size to predict the execution time. Nevertheless, the traditional MLP method could have been more effective due to its limited understanding of the DL layers.

Justus et al. [13] used the layer-by-layer technique proposed by Qi et al. [66] to improve the performance prediction accuracy. But use an MLP-based regression model instead of an analytic model. Gianti et al. [62] used a layer-by-layer technique as Justus et al. [13]. Instead of layer parameters, they used complex parameters such as FLOPS to predict the execution time and power of an individual layer of the DL model. Other researchers [11, 15, 67, 68] also used the same layerwise approach to predict the execution time, memory allocation, and power consumption of the DL model. Yu et al. [51] employed a wave-scaling method for estimating the training step time of the deep learning model on a GPU. They also used the layerwise approach. However, this wave scaling technique necessitates the availability of a GPU to facilitate the prediction.

On the other hand, researchers used a graph neural network instead of MLP in a layerwise approach to predict the performance of the DL model [19, 20]. The layerwise approach did not capture the DL model network topology, and therefore prediction accuracy is sub-optimal [63]. To solve the above problem, Gao et al. [52] and other researchers, [25, 61, 63], used a graph learning to understand the model network topology by generating embeddings. Furthermore, they combine embeddings with overall DL features to predict the training and inference characteristics. The majority of prior studies utilized supervised techniques for DL model performance prediction, neglecting the vast pool of unlabelled DL model data. Our innovative approach, TraPPM, bridges this gap using a semi-supervised learning paradigm, enhancing prediction accuracy by harnessing unlabelled data.

In the first step, we employ an unsupervised graph neural network using unlabelled DL models. This network generates embeddings for input DL models, facilitating an in-depth understanding of the input DL model's network topology. In the subsequent step, we combine the embeddings with the static features extracted from a DL model. This fused data is utilized for training a GNN-based regressor using labeled data to predict the training characteristics. Our approach provides a more comprehensive and effective performance prediction mechanism than the previous works.

5.4 Methodology

Our methodology consists of two phases. **Phase 1:** Unsupervised Learning, unlabelled DL graphs are trained using a GAE to generate embeddings, as explained in Section 5.4.2. However, we cannot directly feed the DL model in Open Neural Network Exchange (ONNX)[1] format into the input of GAE for training. Instead, we need to convert it

[1]https://github.com/onnx/onnx

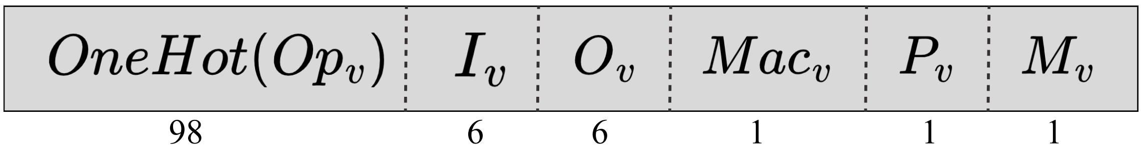


Figure 5.1: The graph's nodes are augmented with node features, each consisting of 113 elements. To accommodate 3D convolution, padding was appended at the end of both the input and output shapes.

to PyTorch Geometric (PyG) [69] data format before training, as explained in Section 5.4.1. **Phase 2:** Supervised Learning, the trained encoder from the GAE is utilized to generate embeddings for the labeled DL model. These embeddings and static features, along with the labeled DL models to, train GNN-based regressors to predict the training characteristics, as described in Section 5.4.3.

5.4.1 Graph Transformation

Given a DL model M with operations $O = \{o1, o2, ..., on\}$, we transform M (in ONNX format) into a graph G compatible with PyG. In G, nodes represent M's operations stored in the node feature matrix X, while A captures directed relationships. Specifically, $G = (X, A)$ where $X = O$ and $A[i][j] = 1$ if a directed edge exists from o_i to o_j, else $A[i][j] = 0$. If M is labeled, we incorporate a target vector Y into PyG data. For each node v in the DL model graph, we define an attribute vector A_v as: $[\mathbf{E}_O(v), I_v, \mathcal{O}_v, \text{Mac}_v, P_v, M_v]$. Here, $\mathbf{E}_O(v)$ is a one-hot encoded vector of length $|O|$, where $|O| = 98$, surpassing the previous work supported only 32 operators [63]. The vectors I_v and $\mathcal{O}_v$, each of length 6, encapsulate the input and output shape, respectively, with an extension to consider 3D convolution. The attributes Mac_v, P_v, and M_v symbolize the MAC, parameters, and memory of node v, respectively. Thus, our node feature vector n has a dimensionality of 113, offering a more exhaustive representation as shown in Figure 5.1. To the best of our knowledge, this is the first work to incorporate 2D and 3D convolutions, alongside transformer-based architectures, into node features. This advancement distinguishes our approach from prior studies that were limited to 2D convolutions.

5.4.2 Phase 1: Unsupervised Learning

In order to leverage the potential of unlabelled data, we train the GAE model in unsupervised manner. The fundamentals of GAE are explained in Section 5.2.0.3. However, it is not possible to directly use the DL model in ONNX format as input to the GAE. Therefore, we first transform the ONNX format to G as described in Section 5.4.1. The overview of our GAE is illustrated in Figure 5.2.

The GAE's encoder is composed of four GraphSAGE convolution layers, which process node features of dimension $[\#nodes, 113]$. These layers aggregate neighborhood features, followed by batch normalization and ReLU activation to introduce non-linearity and

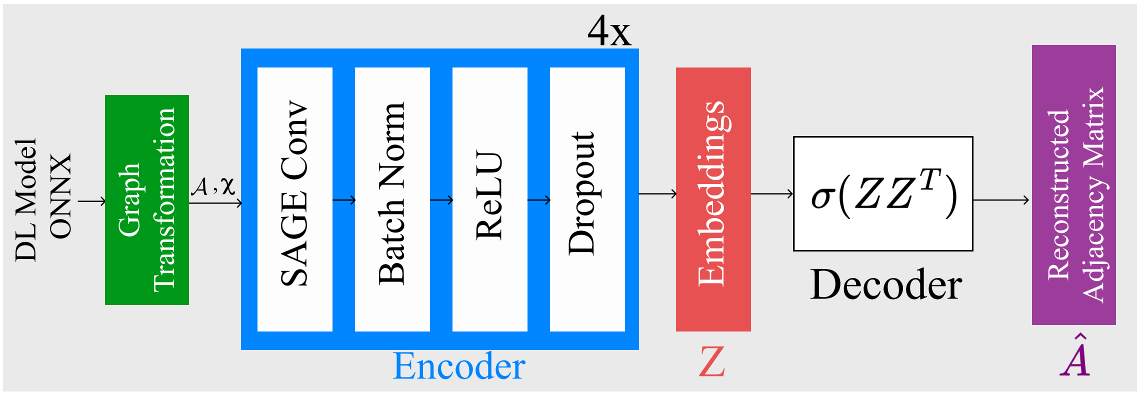


Figure 5.2: Unsupervised Learning - Training Graph Auto Encoder to minimize reconstruction loss of unlabelled DL model graphs.

enhance training stability. A dropout layer with a rate of 0.5 prevents overfitting. The encoder outputs embeddings Z in a latent space of dimension $[\#nodes, 512]$, as depicted in Figure 5.2. The decoder reconstructs the adjacency matrix $\hat{A}$ using the embeddings Z through the operation $\sigma(ZZ^T)$, where σ is the sigmoid function. The BCE loss for the GAE is defined as:

$$L_{\text{BCE}} = -\log(\hat{A}(z, i_{\text{pos}}, j_{\text{pos}}) + \epsilon) - \log(1 - \hat{A}(z, i_{\text{neg}}, j_{\text{neg}}) + \epsilon)$$

In this equation, $\hat{A}$ denotes the predicted adjacency matrix. The terms $i_{\text{pos}}, j_{\text{pos}}$ signify the indices of positive edges, while $i_{\text{neg}}, j_{\text{neg}}$ correspond to negative edges, obtained through negative sampling. To ensure stability during the computation of logarithms, we used a small constant $\epsilon = 1 \times 10^{-15}$. The essence of this loss metric lies in its ability to guide the GAE towards accurately reflecting the original graph structure. The model optimizes this loss, aiming to accurately reconstruct the graph's adjacency matrix. Upon minimizing this loss, the weights of the GAE's encoder are frozen, setting the stage for Phase 2's supervised training.

5.4.3 Phase 2: Supervised Learning

The primary objective of TraPPM is to predict training characteristics such as memory usage (MB) and training step time (ms). To achieve this, we employ a GNN-based regressor for prediction. The overview of supervised learning is shown in Figure 5.3. The input G includes both actual values, represented by [mb, W], and static characteristics. The static features encompass the batch size B, the total number of nodes N_t, the total number of edges E_t, total MAC operations (MAC_t), total parameters (P_t), and total memory (M_t). The values N_t and E_t are directly extracted from G, while the values MAC_t, P_t, and M_t are obtained using the ONNX tool. Consequently, the static feature vector F_s has a length of 6. Supervised learning consists of three components.

GNN Component: It consists of two layers of the SAGEConv layer, and each SAGEConv layer is succeeded by a ReLU activation function and a dropout mechanism with a rate of 0.05.

Feature Aggregation Component: The node features produced by the SAGEConv layer were aggregated using the sum reduce function, resulting in a [1, 512] dimension. Similarly, embeddings generated from the GAE were reduced to [1, 512] using another sum aggregator function. These two embeddings were then combined with a static feature F_s, forming a vector of dimensions [1, 1030], which was subsequently fed into the MLP component.

MLP Component: The concatenated feature vector is passed through two Fully connected (FC) layers. Both FC layers are succeeded by ReLU activations and dropout layers with a rate of 0.05. The processed features are passed through a final layer that produces a single output value.

In the forward pass, the model processes the input G, performs graph convolutions, aggregates node features, integrates it with static features and aggregated embeddings generated from GAE, and passes it through the FC layers to produce the final prediction. We employed the Mean Squared Error (MSE) as our loss function and utilized the Adam optimizer for the training phase. In the backward pass, the model updates the parameters θ, in both the GNN and MLP components. To individually predict memory usage (MB) and step time (ms), we have trained two distinct models: M1 for memory and M2 for step time and frozen their weights. A given input G is simultaneously processed by all two models (M1 and M2). Alongside G, each model also receives the static feature vector F_s and the aggregated embeddings generated by the GAE as we discussed earlier. The combined input helps these models produce accurate predictions on the training characteristics of a given DL model.

5.4.4 Evaluation Metrics

To assess the performance of our TraPPM model compared to the baseline models, we employ two widely used evaluation metrics: MAPE and Root Mean Square Error (RMSE). We chose MAPE because it measures the average percentage difference between the predicted and actual values. It allows us to assess the relative accuracy of the predictions as shown in equation 5.1. On the other hand, RMSE is used to measure the overall magnitude of prediction errors on the same scale as the predicted variable, providing a standardized and interpretable metric for assessing the performance of prediction models as shown in equation 5.2. By utilizing both MAPE and RMSE in our experiment, we thoroughly evaluate TraPPM's performance compared to the baseline models.

$$\text{MAPE} = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right| \times 100 \tag{5.1}$$

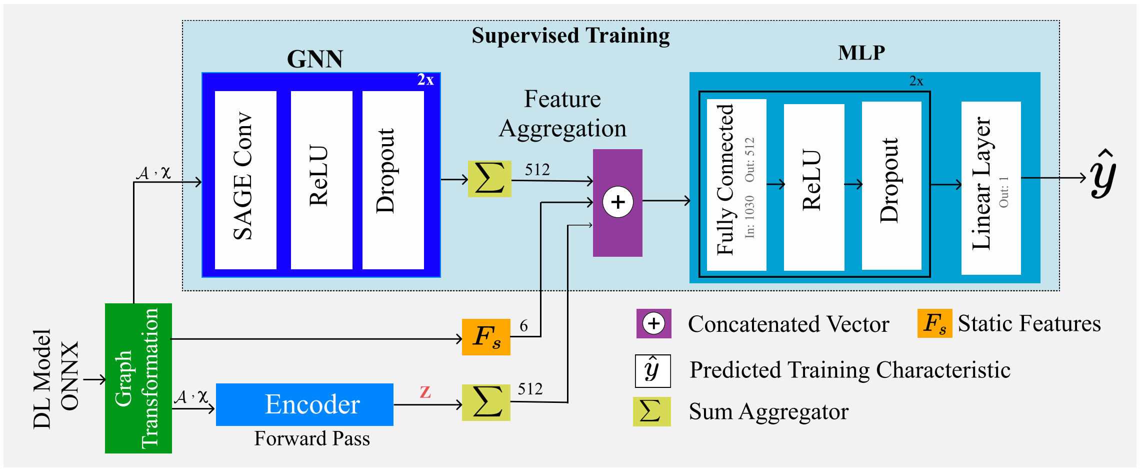


Figure 5.3: Supervised Learning - Training a GNN regressor using MSE loss to minimize the actual y vs. predicted $\hat{y}$. We train three different models separately for predicting step time (ms), memory usage (MB), and power consumption (W).

$$\text{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}, \tag{5.2}$$

5.5 Experiments and Results

5.5.1 Enviroment setup

We used two hardware configurations for data collection: the first comprised an AMD EPYC 7402 processor with two sockets (24 cores per socket), 512 GB DDR4-3200 RAM, and a NVIDIA A100 GPU with 40 GB HBM; while the second utilized 2× Intel Xeon Gold 6148 CPUs (2× 20 cores at 2.4 GHz) and a NVIDIA V100 GPU with 16 GB HBM. However, we exclusively used the hardware equipped with the NVIDIA A100 GPU for the experiments. The experimental environment for developing TraPPM involved the utilization of several essential Python libraries. The important libraries used were PyTorch 2.0.0, torch-geometric 2.3.0, torch-cluster 1.6.1, ONNX 1.13.1, and torch-sparse 0.6.17. These libraries played a crucial role in implementing and training the TraPPM model. The experiments for training TraPPM and generating the dataset were conducted on the abovementioned system using CUDA 11.7.

5.5.2 Datasets

For our TraPPM experiment, we employed a dual-method approach, harnessing both unsupervised and supervised datasets. As we already discussed, unsupervised datasets are used for training GAE, and the supervised dataset is used to train the GNN-based regressor.

5.5.2.1 Unsupervised Dataset

For the TraPPM experiment, we harnessed the Timm library [70] to generate a diverse unsupervised dataset comprising various CNN and transformer-based architectures. These models were exported in ONNX format and subsequently converted to PyG data, a process detailed in Section 5.4.1. The dataset includes 25,053 unlabeled DL models, spanning eleven distinct model families as outlined in Table 5.1. This extensive collection of models underpins our unsupervised learning approach, as elaborated in Section 5.4.2.

The dataset features a range of model variants within each family. For instance, the ConvNext family [71] encompasses variants like Base, Large, Small, and Tiny. The DenseNet family [72] includes DenseNet121, 161, 169, and 201. Other represented families are EfficientNet [73], MnasNet [74], MobileNet [75], PoolFormer [76], ResNet [77], Swin [48], VGG [78], along with additional models such as Visformer [79] and ViT [80]. This breadth ensures a comprehensive representation of current DL model architectures, facilitating robust unsupervised learning.

5.5.2.2 Supervised Dataset

Our supervised dataset is subset of unsupervised dataset. The data collection was conducted using two GPUs: the NVIDIA A100 and the NVIDIA V100, as described in Section 7.7.1. Specifically, using the A100 GPU, we collected a total of 7536 labeled DL models. Conversely, with the V100 GPU, we gathered 543 labeled DL models. For baseline model comparisons, we primarily utilized the labeled DL models from the A100 GPU. Meanwhile, the dataset collected from the V100 GPU was exclusively reserved for evaluating TraPPM's transfer learning capabilities. We again utilized the Timm library to generate DL models. However, instead of saving them to the ONNX format, we trained each model for 55 iterations, with the initial five iterations serving as a warm-up phase. We calculated the CUDA time during each iteration, representing the time taken to process a single iteration or step time in the training process. Our focus was primarily on step time, as it remains consistent during the training of the DL model, except for the initial few iterations that may exhibit variations due to warm-up effects. Therefore, we excluded the first five iterations when calculating the metrics. Additionally, we collected memory usage and power consumption data using the NVML[2] Python library. For each of the eleven different model families, we repeated this process, averaging the step time (ms), memory usage (MB), and power consumption (W). The results, along with the corresponding ONNX model files, were saved. During converting these models to PyG data format, we appended the measured values into the graph data Y.

[2]https://pypi.org/project/pynvml/

Table 5.1: TraPPM dataset distribution across eleven DL model families. Each number represents a different variant of the same graph structure within a family. The unsupervised set includes 25,053 models; the supervised sets contain 7,536 (A100) and 543 (V100) labeled models.

Family	**Unsupervised**	**Supervised**	
		A100	**V100**
DenseNet	838	466	27
EfficientNet	1370	566	44
MnasNet	7208	795	64
MobileNet	2449	1613	123
PoolFormer	601	377	36
ResNet	1805	821	56
Swin	787	421	36
VGG	6171	937	61
VisFormer	237	235	17
ConvNext	1530	439	27
ViT	2057	866	52
Total	**25053**	**7536**	**543**

5.5.3 Training - Graph Auto Encoder

The first phase of the TraPPM experiment involves training the GAE, a key component of our TraPPM. Initially, we considered the Masked Graph Autoencoder technique, as presented in Hou's study [23]. This method masks random node features and attempts to reconstruct them, facilitating graph representation learning. However, our node features, largely sparse due to one-hot encoding as explained in Section 5.4.1, did not align well with this strategy. As a result, we turned to the classical GAE, which proved to be a better fit for our needs. The GAE model was developed using the PyG Library, the detailed model architecture explained in Section 5.4.2. For training, we employed the BCE loss function and utilized the Adam optimizer with a lr=5×10^{-4}, betas=(0.9, 0.999), eps=1×10^{-8}. To train the GAE, we utilized an unsupervised dataset, as described in Section 5.5.2.1, for a total of 400 epochs. Finally, we have achieved a BCE loss of 0.9291. The entire training process for the GAE took approximately 25.6 hours on a single A100 GPU. We employed t-SNE [81] to visualize the sum aggregated embeddings generated by the GAE, as shown in Figure 5.4. The widespread distribution of the ResNet models is due to its numerous variants, distinguishing it from other models.

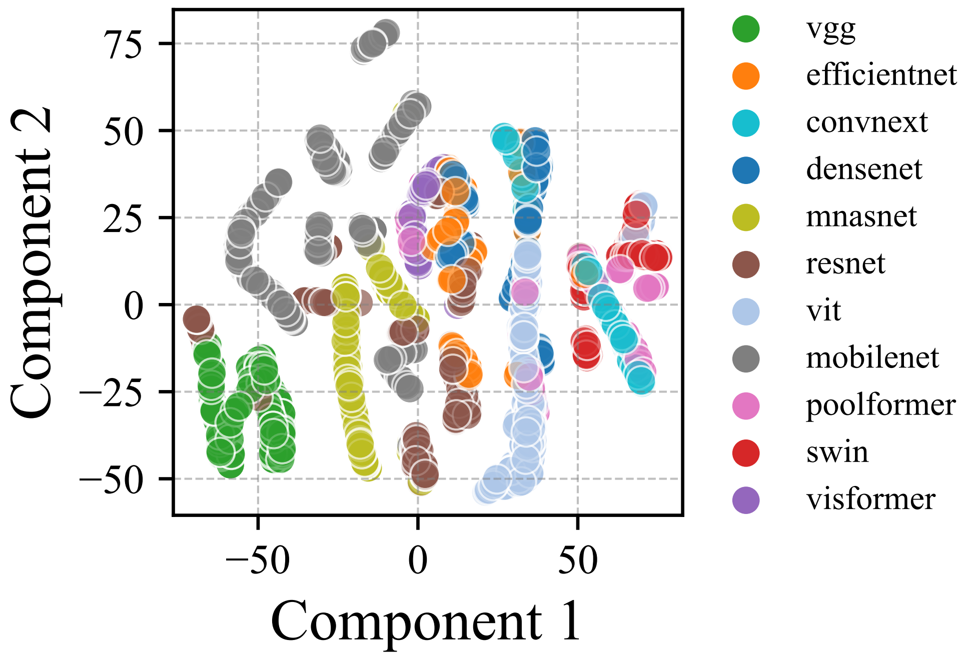


Figure 5.4: t-SNE visualization of sum aggregated embeddings generated by the GAE

5.5.4 Training - TraPPM

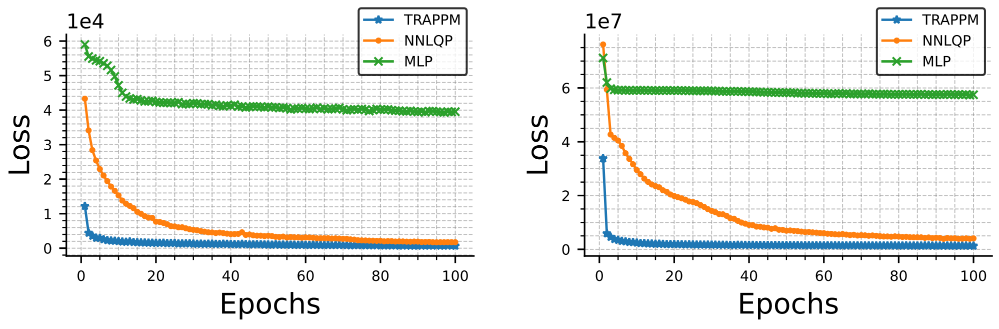
The core part of the experiment involves training the TraPPM model, with the detailed architecture explained in Section 5.4.3. We used the PyTorch library to create the TraPPM model. We used a labeled dataset collected from A100 GPU to train the model, as mentioned in Section 5.5.2.2. We partitioned our dataset according to a 70:30 ratio for each model family. Specifically, 70% of the data was used for training and 30% for testing. This split aligns with the standards established in previous research [63]. However, instead of a conventional split, we adopted a Monte Carlo validation approach. To ensure robustness and reliability in our results, we employed five distinct seeds: 1337, 1338, 1339, 1340, and 1341. By utilizing these seeds, we generated five different dataset splits and subsequently averaged the results to derive a more comprehensive performance evaluation. During the training process, we utilized the Adam optimizer with a lr=1×10^{-3}, betas=(0.9, 0.999), eps=1×10^{-8}. The training was performed over 100 epochs to fair comparison with baseline models. Training the TraPPM model for a single fold takes about 1 hour and 17 minutes for 100 epochs.

5.5.5 Baseline Models

In our evaluation, we compared TraPPM with three baseline models: Gboost, MLP, and the supervised GNN. Gboost served as a strong foundation for further development,

Table 5.2: Comparative Performance Analysis of Memory Usage (MB) Prediction: Averaged results over five distinct splits. Results highlight TraPPM's enhanced accuracy compared with baseline models. The lower the values, the higher the accuracy.

Family	**TraPPM**		**NNLQP**		**MLP**		**GBoost**	
	MAPE	**RMSE**	**MAPE**	**RMSE**	**MAPE**	**RMSE**	**MAPE**	**RMSE**
convnext	**4.95%**	**1005.71**	7.01%	1490.67	54.74%	8906.85	14.62%	3230.67
densenet	**3.52%**	**730.47**	8.29%	1675.17	66.44%	8317.23	14.64%	3113.40
efficientnet	**3.55%**	**537.84**	7.67%	1687.05	51.70%	11952.91	16.70%	3458.01
mnasnet	**4.53%**	**585.65**	6.75%	1804.18	94.42%	4908.33	14.72%	2756.27
mobilenet	**5.28%**	**633.74**	6.74%	1587.84	108.22%	5051.35	24.65%	2879.35
poolformer	**4.41%**	**1441.70**	6.96%	2027.24	76.10%	8951.67	15.04%	3332.57
resnet	**4.65%**	**658.40**	8.09%	1229.99	124.26%	7393.93	16.78%	2479.84
swin	**5.09%**	**774.91**	10.37%	1853.26	53.68%	8294.61	15.12%	2909.59
vgg	**10.48%**	2341.17	10.76%	**2271.89**	42.29%	7145.38	15.87%	3911.28
visformer	**3.92%**	**318.97**	9.49%	722.83	191.19%	8671.54	13.97%	1170.94
vit	**3.78%**	**985.22**	9.07%	2219.88	72.05%	8908.69	14.99%	3444.81



(a) Step Time (ms) (b) Memory Usage (MB)

Figure 5.5: Epoch vs Loss plot comparing the convergence rates of TraPPM, NNLQP, and MLP. TraPPM showcases rapid convergence due to its ability to leverage unsupervised learning from unlabeled data, as trained over 100 epochs.

while MLP was chosen for its wide usage in performance prediction [13]. Finally, we included the supervised GNN model introduced by Lu et al. [63], referred to as NNLQP. This model served as a reference for evaluating the performance of TraPPM in relation to a well-established supervised GNN approach.

5.5.5.1 Gradient Boosting

To develop the GBoost model, we conducted training using the XGBoost [82] python library. The training process involved utilizing a supervised dataset that solely consisted of DL static features as input. To optimize its hyperparameters, we conducted a grid search. The hyperparameters explored during the grid search were estimators with values [500, 1000, 2000], lr with values [1×10^{-3}, 1×10^{-4}], max depth with values [10, 30, 50], subsample with values [0.5, 0.75, 1], and colsample bytree with values [0.5, 0.75, 1]. After performing the grid search, we identified the best hyperparameters as follows: colsample bytree: 1, lr: 1×10^{-3}, max depth: 50, estimators: 2000, subsample: 1.

5.5.5.2 MLP

We created a baseline MLP model that is similar to the TraPPM MLP component, with the only difference being that it accepts only static features as input during training. We trained the baseline MLP model using 100 epochs, utilizing the MSE loss function and the Adam optimizer with lr=1×10^{-3}, which is the same setting as the TraPPM supervised training.

5.5.5.3 NNLQP

It is important to note that a key distinction between the NNLQP model and the TraPPM model is that the NNLQP model is unable to utilize unsupervised datasets. It can only operate with supervised datasets. To ensure a fair comparison, we kept the

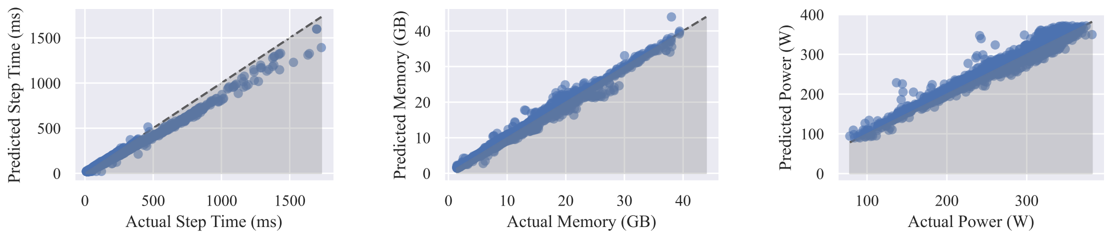


Figure 5.6: Comparison of actual values with predictions from TraPPM on the test set. The model was trained for 100 epochs using a supervised dataset, with a split seed of 1337.

model architecture unchanged, only adapting the node features to accommodate the TraPPM dataset as discussed in Section 5.4.1. We trained the model for 100 epochs using the Adam optimizer with lr=1×10^{-3}, following the same settings as the TraPPM model. The NNLQP model takes the graph representation G as input, generates embeddings, concatenates them with static features, and employs an MLP to predict performance.

5.5.6 Baseline - Comparison

We assessed the performance of the TraPPM model by comparing it with baseline models. Both the TraPPM model and the baselines were trained using a supervised dataset of A100 GPU, with a specific focus on predicting step time (ms) and memory usage (MB). We trained the TraPPM, NNLQP, and MLP models for 100 epochs, repeating the process five times using different seeds as outlined in Section 5.5.4. When we assessed the models for their capability to predict memory usage and step time, the epoch-versus-loss plot as shown in Figure 5.5, revealed that the TraPPM model converges more rapidly compared to both NNLQP and MLP. This faster convergence can be attributed to TraPPM's ability to leverage unsupervised learning from unlabeled data.

5.5.7 Model Evaluation and Comparison

The performance evaluation of the TraPPM model against baseline models was conducted using a test dataset, with MAPE and RMSE as the key metrics, as detailed in Section 7.7.2. These metrics were computed for each model family individually to provide a comprehensive performance assessment. Lower MAPE and RMSE values indicate closer alignment of predictions with actual values.

Table 5.2 details the predictive accuracy for memory consumption, while Table 5.3 focuses on training step latency. The TraPPM model notably outperforms the baselines in both aspects. In memory consumption prediction, TraPPM achieves a significant relative improvement in MAPE of 40.6% over the NNLQP model, demonstrating its robustness. Similarly, for training step time predictions, TraPPM exhibits superior accuracy, with a relative MAPE improvement of 34.2% compared to NNLQP. Additionally, the aggregate

Table 5.3: Comparative Performance Analysis of Step time (ms) Prediction: Averaged results over five distinct splits. Results highlight TraPPM's enhanced accuracy compared with baseline models.

Family	**TraPPM**		**NNLQP**		**MLP**		**GBoost**	
	MAPE	**RMSE**	**MAPE**	**RMSE**	**MAPE**	**RMSE**	**MAPE**	**RMSE**
convnext	**8.09%**	**46.00**	9.50%	61.06	64.92%	354.59	16.15%	102.72
densenet	**6.69%**	**15.46**	18.41%	36.50	104.45%	155.09	14.08%	33.61
efficientnet	**6.81%**	**13.46**	9.45%	22.51	49.18%	118.56	15.36%	34.94
mnasnet	**7.98%**	**12.72**	18.59%	40.05	106.86%	80.87	14.49%	41.18
mobilenet	**9.20%**	**8.95**	14.66%	21.69	116.48%	52.97	25.79%	31.44
poolformer	**13.02%**	27.05	13.23%	**26.79**	166.75%	119.56	14.59%	32.51
resnet	**11.26%**	**16.20**	25.45%	36.02	192.95%	122.75	24.13%	46.33
swin	9.01%	35.18	**8.89%**	**33.66**	60.08%	263.86	15.68%	72.44
vgg	**10.74%**	**22.51**	13.20%	30.29	69.91%	83.70	15.89%	43.59
visformer	14.79%	17.88	18.61%	15.29	437.33%	287.67	**13.99%**	**14.85**
vit	**7.06%**	**40.13**	9.15%	83.41	105.84%	432.32	16.60%	146.39

Table 5.4: Average Performance Comparison of TraPPM with Baseline Models.

	Memory Usage (MB)		**Step Time (ms)**	
Model	**MAPE $\downarrow$**	**RMSE $\downarrow$**	**MAPE $\downarrow$**	**RMSE $\downarrow$**
TraPPM	**4.92%**	**910.34**	**9.51%**	**23.23**
NNLQP	8.29%	1688.18	14.47%	37.02
MLP	85.01%	8045.68	134.07%	188.36
GBoost	16.10%	2971.52	16.98%	54.54

performance of the TraPPM model, encompassing all tested model families, is summarized in Table 5.4. This table provides a holistic view of the TraPPM model's performance across various prediction tasks, reinforcing its overall efficacy in comparison to the baseline models.

5.5.8 Theoretical Insights into TraPPM's Semi-Supervised Learning Approach

The TraPPM model leverages a semi-supervised framework, integrating unsupervised learning for generating embeddings, which significantly enhances its predictive capabilities for step time and memory usage. This methodology stands in contrast to the purely supervised models like NNLQP, which rely exclusively on labeled data. Theoretically, the effectiveness of TraPPM is attributed to its ability to access a richer representation space, capturing latent structural features within the data through these unsupervised embeddings, features that remain elusive in a solely supervised paradigm.

From a mathematical perspective, the TraPPM model can be seen as operating within an expanded function space, F', compared to the more limited function space, F, accessible by conventional supervised learning. This expanded space F', achieved through the integration of unsupervised embeddings, encapsulates the original space F but extends further to incorporate additional dimensions reflecting data variance and underlying structure. The empirical benefits of this expansion are evidenced by the improved MAPE and RMSE metrics detailed in Tables 5.2 and 5.3, with aggregate performance enhancements further demonstrated in Table 5.4.

To provide empirical validation of these theoretical and mathematical concepts, we include actual versus predicted plots in Figure 5.6. These plots vividly illustrate the alignment between TraPPM's predictions and actual outcomes, thereby substantiating the model's proficiency in accurately forecasting training characteristics. They visually reinforce the theoretical and mathematical merits of the semi-supervised learning approach employed by TraPPM, highlighting its superiority in a variety of prediction tasks across multiple model families.

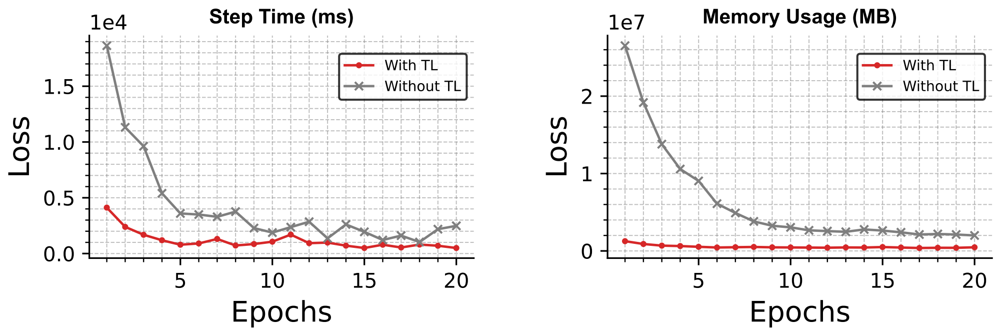


Figure 5.7: Epoch vs. Loss plot demonstrating TraPPM's enhanced convergence through transfer learning.

5.5.9 Ablation Study: Impact of Weight Initialization

In this ablation study, we examine the influence of weight initialization on the TraPPM model's performance, focusing on two distinct GAE configurations: one using pre-trained weights and another with randomly initialized weights. Both configurations are integrated with a GNN for regression tasks, as detailed in Section 5.4.3.

Empirical results indicate a marked difference in performance based on the initialization approach. The model with pre-trained weights demonstrates a notable decrease in MSE loss for training step time, starting from 1.26×10^4 and reaching 6.82×10^2 by the 100th epoch, signifying effective and efficient learning. In contrast, the randomly initialized model begins with a substantially higher initial MSE loss of approximately 5.43×10^{12}, which only marginally improves to 1.05×10^5 by the 2nd epoch and then stagnates, showing no further significant decrease in subsequent epochs. This pattern is consistently observed for both training step time and memory consumption prediction tasks.

Theoretically, this disparity can be attributed to the different starting points in the parameter space optimization landscape. Pre-trained weights provide a beneficial starting position, facilitating a more focused and stable gradient descent path ($\nabla_\theta L$). Conversely, random initialization tends to place the model in a less favorable starting point, often characterized by steeper initial gradients and a higher likelihood of getting trapped in local minima.

These observations underscore the critical role of initial weight settings in the performance of GAEs, especially in the context of the TraPPM model. The study highlights the substantial advantage of employing pre-trained weights for complex structured data tasks, as they significantly enhance the model's ability to learn efficiently and effectively.

5.6 Discussion

5.6.1 TraPPM's Adaptability to Predicting Diverse Metrics

The TraPPM model demonstrates its versatility in metric prediction, such as power consumption, by leveraging the GAE's embeddings Z. These embeddings, derived from DL models, are crucial for extending prediction capabilities beyond standard metrics like memory usage and step time.

The GAE transforms high-dimensional inputs G into a comprehensive latent space $Z = f_{\text{GAE}}(G)$, forming the foundation for a GNN-based regression model as explained in the Section 5.4.2. For power consumption prediction, this GNN model, trained on the supervised dataset for 100 epochs (as outlined in Section 5.5.4), aims to minimize the MSE between the predicted $\hat{y}_{\text{power}}$ and actual power consumption values y_{power}:

$$L_{\text{power}} = \frac{1}{n} \sum_{i=1}^{n} (y_{i,\text{power}} - \hat{y}_{i,\text{power}})^2$$

This method highlights TraPPM's adaptability in using the same set of GAE embeddings for diverse predictions. The effectiveness of this approach is validated by TraPPM's performance in power consumption prediction, achieving a MAPE of 5.01% and an RMSE of 17 W, thereby demonstrating the robustness and versatility of GAE embeddings in various predictive scenarios. Figure 5.6, clearly depicts TraPPM's predictive accuracy, illustrating the close alignment between predicted and actual power consumption values.

5.6.2 Transfer Learning Capability of TraPPM

Transfer learning, crucial in DL when labeled data is scarce, was employed in TraPPM to address the limited labeled data for the V100 GPU (Section 5.5.2.2). By initializing the V100 GPU training with weights W_{A100} from the A100 GPU-trained model, depicted in Figure 5.7, we aimed to expedite convergence compared to starting from scratch.

In TraPPM, transfer learning theoretically embodies domain adaptation, transitioning the function $f_{\text{source}}(X; W_{A100})$ to $f_{\text{target}}(X; W)$. This strategy circumvents the initial generic feature learning phase, directly fine-tuning the model to the target dataset's specificities.

The effectiveness of this approach in TraPPM led to substantial relative improvements in prediction accuracy: approximately 55.03% in RMSE for Step Time and 48.76% for Memory usage. Table 5.5 details these enhancements, underscoring the robustness of transfer learning in optimizing TraPPM's predictive performance for different hardware contexts, especially where labeled data is limited.

5.6.3 Ease of Use with TraPPM

We have developed a TraPPM as a Python library for predicting the step time, memory usage, and power consumption of DL models in the ONNX format. Users can effortlessly

Table 5.5: Comparison of Metrics With/Without TL.

Label	**Metric**	**With TL**	**Without TL**
Step Time	MAPE (%)	**19.13**	28.24
	RMSE (ms)	**20.05**	44.59
Memory	MAPE (%)	**11.22**	28.49
	RMSE (MB)	**603.03**	1176.90

```python
import trappm

config = { 'model': 'resnet101.onnx', 'batch_size': 32,
           'device': 'GPU:A100-SXM4-40GB:1' }
out = trappm.predict(config)
print("ms: {0}, MB: {1}, W: {2}".format(*out))
```

Figure 5.8: A sample Python code using TraPPM to predict.

leverage TraPPM's performance prediction capabilities with just a few lines of code, as shown in Figure 5.8.

5.6.4 Optimizing Cloud Costs and Resources with TraPPM

TraPPM is instrumental not only in Neural Architectural Search but also in datacenter job scheduling and cloud cost estimation. Its predictive capability enables efficient resource planning in datacenters and accurate estimation of cloud computing expenses. For example, using TraPPM to predict the training duration of an EfficientNet_b6 model, with an predicted step time of 350 ms over 2×10^5 iterations, yields a training time of around 19.44 hours. On a cloud platform with an A100 GPU costing 2.934 USD per hour, the total cost is approximately 57.04 USD. This application of TraPPM for cost prediction showcases its utility in optimizing computational resources and budgeting for cloud-based DL tasks.

5.7 Conclusions

We present TraPPM, a novel framework that combines unsupervised GAE with a supervised GNN regressor to precisely predict DL model training characteristics without necessitating execution on target hardware, a significant departure from traditional approaches reliant solely on labeled datasets. TraPPM demonstrates exceptional predictive accuracy, achieving MAPEs of 4.92% for memory usage, 9.51% for step time, and 5.01% for power consumption, along with robust RMSE values. The release of our comprehensive dataset comprising 25,053 unlabelled DL graphs and 8,079 labeled DL graphs further enriches the field, providing a valuable resource for future research. TraPPM's innovative use of unlabeled data in a semi-supervised learning context marks a significant advancement in the DL performance prediction community.

6 Can Tree-Based Model Improve Performance Prediction for LLMs?

The deployment of Large Language Models (LLMs) in cloud environments underscores the imperative for optimal hardware configurations to enhance efficiency and reduce environmental impact. Prior research on performance prediction models predominantly focuses on computer vision, leaving a void for models adept at the unique demands of LLMs. This study bridges that gap, evaluating the potential of Tree-based models, particularly XGBoost, against traditional Graph Neural Networks (GNNs) in predicting the performance of LLMs. Our analysis shows that XGBoost achieves significant improvements in predicting throughput, memory usage, and energy consumption showcasing relative enhancements of MAPE approximately 68.81%, 80.85%, and 88.21%, respectively, compared to the GNN baseline, with a remarkable speed enhancement of approximately 26761.39% over GNN. These findings underscore XGBoost's effectiveness in accurately forecasting LLM performance metrics, offering a promising avenue for hardware configuration optimization in LLM deployment.

This chapter builds upon the research presented in the following published paper:

Peer reviewed

Karthick Panner Selvam, and Mats Brorsson. "Can Tree-Based Model Improve Performance Prediction for LLMs?." ARC-LG workshop at 51st International Symposium on Computer Architecture (2024). hdl.handle.net/10993/62222.

Contents

6.1 Introduction

The rapid evolution of Large Language Models (LLMs) has been a cornerstone of recent advancements in natural language processing, enabling unprecedented capabilities in text generation, comprehension, and translation. Models such as BERT, GPT, and their variants have demonstrated the profound impact of deep learning in understanding and generating human language. However, the deployment of these sophisticated models poses significant challenges, chiefly due to their substantial computational demands and the environmental impact of their operation. As LLMs become increasingly central to a wide range of applications, the imperative to optimize their deployment on various hardware configurations becomes evident, not only to enhance computational efficiency but also to mitigate costs and reduce carbon emissions.

LLM performance across different hardware platforms is influenced by a myriad of factors, including throughput (T), peak memory usage (M_{peak}), energy consumption per sample (E) as shown in Figure 6.1. These metrics are critical for selecting the optimal hardware configuration that balances performance with energy efficiency and environmental sustainability. Yet, accurately predicting these metrics presents a complex challenge, compounded by the intricate architecture of LLMs and the dynamic nature of hardware utilization. Considering the importance of hardware selection in optimizing performance and reducing environmental impact, it is essential to thoroughly analyze these metrics when deploying LLMs.

Historically, the prediction of deep learning model performance has relied on Graph Neural Networks (GNNs), which, while effective in general contexts, face challenges when applied to LLMs. LLMs are characterized by their extensive node and edge counts, as demonstrated in Table 6.1, and a repetitive layer structure that complicates traditional performance prediction methods. To illustrate, Figure 6.2 displays the frequency of operations within the BERT model, revealing a repetitive stacking of encoder layers. Each layer comprises distinct multi-head attention and feed-forward networks. This repetitive structure is not just a superficial characteristic but a crucial factor influencing performance across diverse hardware configurations.

Acknowledging these hurdles, our study advocates for a departure from graph-based representations to tabular formats when predicting LLM performance. We posit that this transition could potentially improve both accuracy and prediction speed owing to the hierarchical and repetitive nature inherent in LLM architectures. To investigate this hypothesis, we curated a unique dataset comprising nine distinct LLMs by varying batch and sequence lengths and captured metrics across three NVIDIA GPUs: H100, A100, and V100. This dataset is augmented with comprehensive model and hardware features, enabling an exhaustive assessment of XGBoost against conventional GNN methodologies.

Furthermore, we have developed an intuitive user interface that simplifies the performance prediction process. Users can input the Hugging Face model ID, batch size, sequence length, and the number of forward passes, alongside their hardware choice,

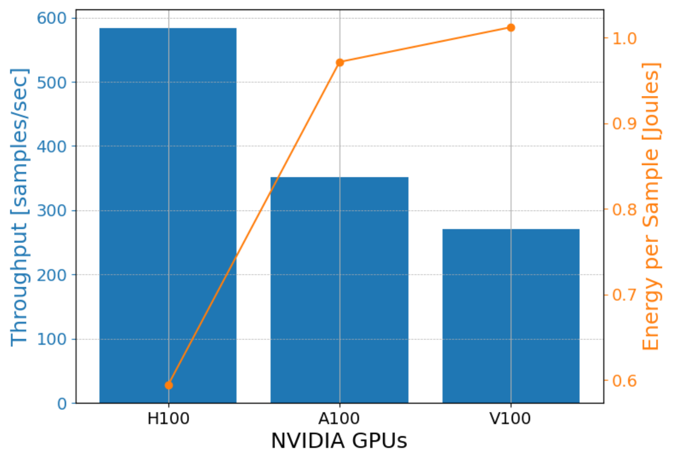


Figure 6.1: Performance analysis of running DeBERTa V3 Large model with batch size 512 and sequence length 64 on Nvidia H100, A100, and V100 GPUs, comparing throughput (samples/sec) and energy per sample (J).

to receive predictions on throughput, peak memory, energy consumption, and carbon emissions within a few seconds. This tool is designed to empower users to make informed decisions about hardware configurations without the need for extensive computational resources.

Contributions:

- Intuitive Tree-based Model: Introducing XGBoost as a solution, we effectively address the complex hierarchical structures of LLMs, demonstrating superior performance in prediction tasks.
- Comprehensive Feature Selection Strategy: By meticulously selecting features that encapsulate the architecture of LLMs and the specifications of hardware, we facilitate nuanced predictions of model performance across various platforms.
- Creation of a Unique Dataset: This first-of-its-kind dataset, detailing LLM performance on NVIDIA's H100, A100, and V100 GPUs, serves as a vital resource for benchmarking and further research into hardware optimization for AI deployment.
- Through these contributions, our research not only tackles the practical challenges of LLM deployment but also fosters discussions on sustainable AI practices, paving the way for the development of more efficient and environmentally friendly AI systems.

6.2 Related Work

The field of performance prediction for deep learning (DL) models, especially in the context of hardware optimization, has witnessed growing interest in recent years. Early

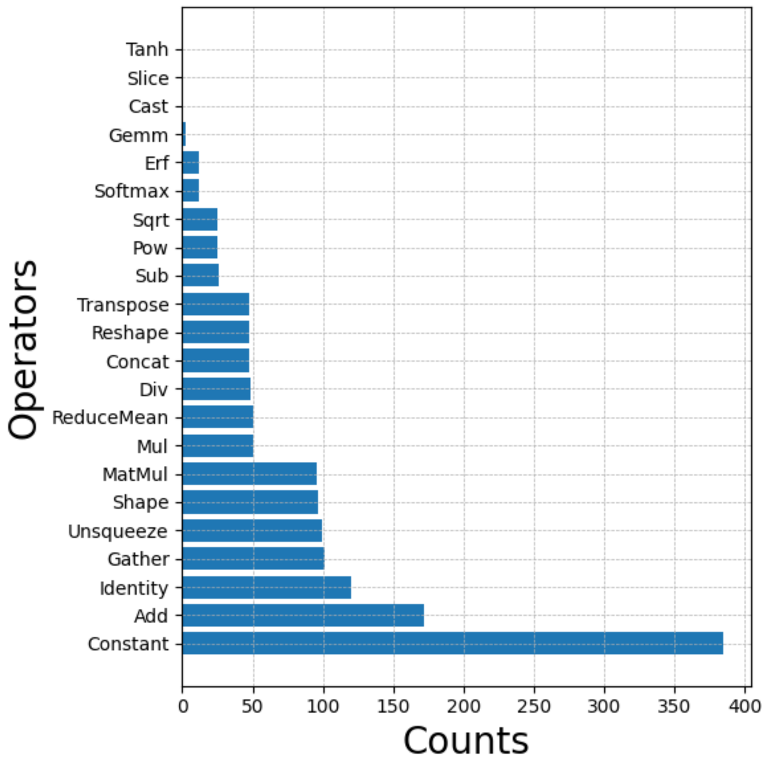


Figure 6.2: Frequencies of different operations (ops) in the BERT base model. The figure illustrates the repetitive nature of certain operations within the model's architecture, highlighting the prevalence of operations like 'Constant', 'Identity', and 'Add'.

Table 6.1: Total Nodes and Edges of LLMs

Model	**Nodes**	**Edges**
bert-large-uncased [83]	2896	6621
xlm-roberta-base [84]	1495	3417
roberta-large [85]	2923	6681
microsoft-deberta-v3-small [86]	2450	5379
roberta-base [85]	1495	3417
bert-base-uncased [83]	1468	3357
distilbert-base-uncased [87]	685	1579
microsoft-deberta-v3-large [86]	9398	20643
microsoft-deberta-v3-base [86]	4766	10467

Table 6.2: Comparison with Recent Work on Performance Prediction. Our model is the first to predict the throughput (T), carbon emissions (C), and peak memory usage (M_{peak}) for LLMs.

Previous Work	**LLM**	C	T	(M_{peak})
[63] NNLQP (2022)	x	x	✓	x
[25] DIPPM (2023)	x	✓	✓	✓
[88] NarformerV2 (2023)	x	x	✓	x
[89] TraPPM (2023)	x	✓	✓	✓
[90] LLM Carbon (2024)	✓	✓	x	x
[91] CDMPP (2024)	(limited)	x	✓	x
Ours	✓	✓	✓	✓

work by Qi et al. [66] proposed an analytical model to estimate DL model training times, but its accuracy was limited due to the neglect of concurrent operations. Subsequent studies like that of Gao et al. [60] extended these methods to predict memory consumption, utilizing analytical models to estimate resource utilization during training.

To improve prediction accuracy, researchers have explored machine learning approaches. Bouhali et al. [10] used an MLP-based regressor with features like trainable parameter counts, but it was constrained by a shallow understanding of DL layers' dynamics. Others, such as Justus et al. [13] and Gianti et al. [62], adopted a layer-by-layer technique, incorporating parameters like FLOPs to predict execution times and power consumption, showcasing progress towards more nuanced models.

In contrast, some researchers opted for a GNN over an MLP in a layerwise approach for DL model performance prediction [19, 20]. However, this layerwise strategy failed to capture the network structure of DL models [63]. To address this limitation, Gao et al. [52]

and other researchers [25, 61, 63, 88, 89] utilized graph learning techniques to generate embeddings that encapsulate the model network topology. These embeddings were then combined with overall DL features to predict training and inference characteristics accurately.

Previous work in performance prediction has predominantly focused on computer vision tasks rather than LLMs. While GNNs have shown effectiveness in computer vision tasks, their performance can be hindered when applied to LLMs due to the larger number of nodes and edges in the graphs representing LLM architectures. GNNs typically involve message passing and node aggregation, which can lead to significant computational overhead and information loss when processing such large and complex graphs. Notably, extensive studies or datasets are scarce for LLMs, with most existing models tailored for computer vision tasks as shown in Table 6.2. For instance, the CDMPP [91] works have created performance models for computer vision tasks, with only slight utilization of the BERT-base model [83]. This scarcity underscores the need for dedicated performance models specifically tailored for LLMs, prompting our novel approach leveraging Tree-based models, particularly XGBoost, to accurately predict LLM performance across diverse hardware configurations.

In our research, we establish a thorough dataset encompassing performance metrics for nine LLMs, systematically varying batch size and sequence length across three NVIDIA GPU architectures, as illustrated in Table 6.1, and Table 6.4, respectively. Our work represents a groundbreaking initiative in the field, marking the inaugural development of a predictive performance model tailored specifically for LLM. This pioneering endeavor constitutes the first comprehensive study in LLM performance prediction.

6.3 Methodology

This section outlines our methodology for predicting key performance metrics of LLMs, focusing on T, M_{peak}, E, and carbon emissions. We begin by discussing the impact of parameters on LLM performance, followed by our approach to feature selection based on thorough analysis and data collection. Finally, we describe the evaluation metrics used to assess the performance of our predictive models.

T: Measured in samples per second, throughput is a critical metric for evaluating the processing speed and efficiency of LLMs. High throughput indicates a model's ability to quickly process data, enhancing user experience and operational productivity.

$$T = \frac{\text{Number of Samples Processed}}{\text{Time Taken (seconds)}} \tag{6.1}$$

$M_{\textbf{peak}}$: Represented in megabytes (MB), peak memory usage determines the maximum amount of memory required during model execution. This metric is vital for assessing whether a model can be deployed on specific hardware configurations without out-of-memory error.

E: Expressed in joules per sample, this metric provides insights into the energy efficiency of running LLMs, directly correlating with operational costs and environmental footprint.

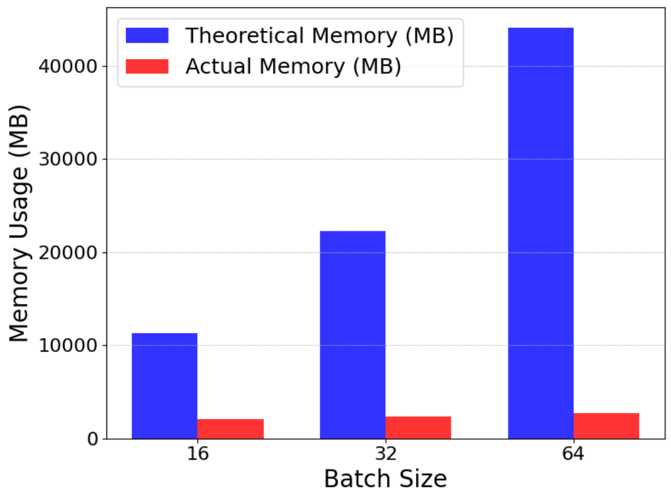


Figure 6.3: Comparison of Theoretical and Actual Memory Usage for BERT-base-uncased with batch size 256 on the H100 device. Theoretical memory calculations were obtained using the ONNX tool.

6.3.1 Empirical Analysis of Memory Usage

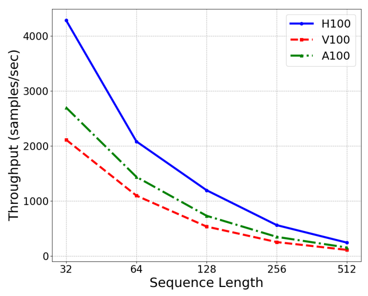
We address the discrepancies between theoretical and actual memory requirements through empirical analysis, as shown in Figure 6.3. Theoretical estimates often overlook runtime overheads and dynamic memory allocation strategies employed by deep learning frameworks, leading to inaccurate hardware selection. This necessitates an empirical approach to performance prediction.

6.3.2 Carbon Emission Calculation

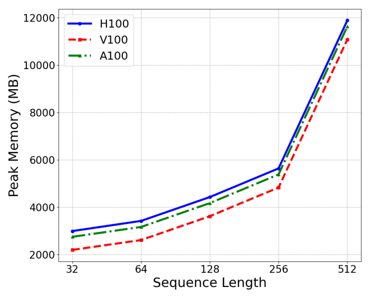
Carbon emissions (C) result from the energy consumed per sample (E) multiplied by the carbon intensity of the electricity source (CI), with the total carbon emissions given by $C = E \times CI$. This calculation is essential for evaluating the environmental impact of deploying LLMs and is influenced by factors such as the number of samples processed.

6.3.3 Influence of Batch Size and Sequence Length

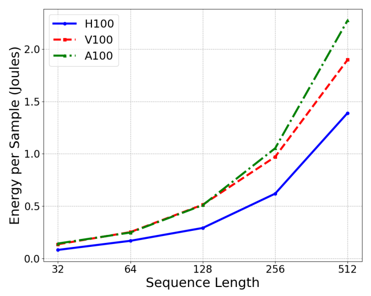
Our investigation, depicted in Figures 6.4 and 6.5, illuminates the impact of sequence length and batch size on LLM performance metrics (T, M_{peak}, E) across diverse Nvidia hardware configurations (H100, A100, and V100). Notably, we observe a substantial decrease in T as sequence length increases, reflecting the computational complexity



(a) Throughput vs. Sequence Length

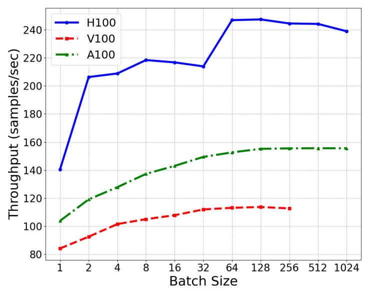


(b) Peak Memory vs. Sequence Length

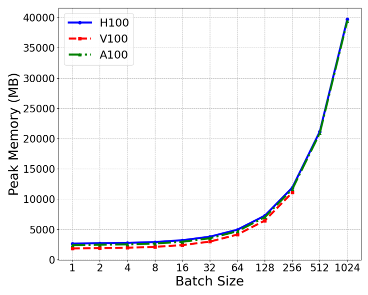


(c) Energy per sample vs. Sequence Length

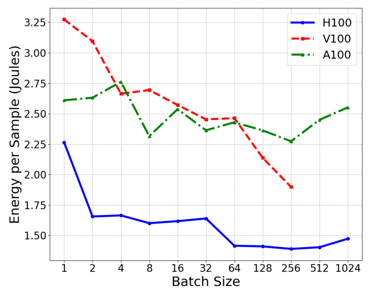
Figure 6.4: Comparative analysis of throughput, peak memory, and energy per sample for varying sequence lengths of Roberta XLM model with a fixed batch size of 256.



(a) Throughput vs. Batch Size



(b) Peak Memory vs. Batch Size



(c) Energy per sample vs. Batch Size

Figure 6.5: Comparative analysis of throughput, peak memory, and energy per sample for varying batch sizes of Roberta XLM model with a fixed sequence length of 512.

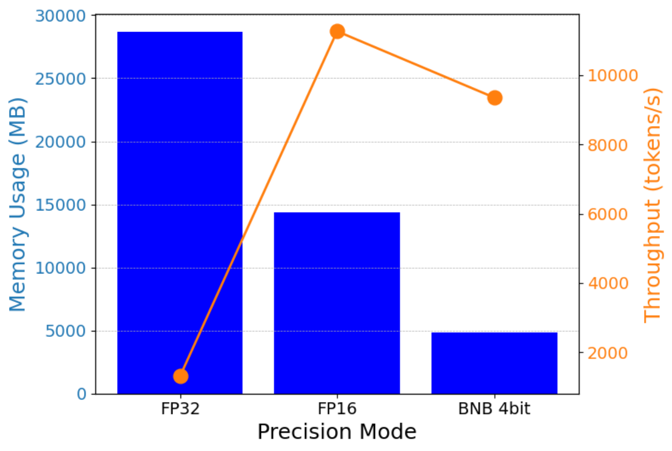


Figure 6.6: Comparison of Quantization Strategies (FP32, FP16, and BNB 4bit) for the LLaMA 7B model on the A100 device. The figure illustrates the impact of quantization on peak memory usage and throughput.

inherent in processing longer sequences. However, this trend is offset by an increase in T with larger batch sizes, showcasing the intricate relationship between batch processing and computational efficiency.

Furthermore, our analysis unveils the influence of sequence length and batch size on M_{peak}, E. As sequence length increases, M_{peak} escalates due to the greater memory demands of processing longer input sequences, a trend consistent across all hardware configurations. Conversely, the impact of batch size on M_{peak} is akin to that of varying sequence length because increasing batching allocates more High Bandwidth Memory (HBM), facilitating efficient memory utilization. However, increasing batch size results in a reduction in energy consumption per sample, as the hardware can effectively handle larger batch sizes, optimizing energy efficiency during processing.

6.3.4 Quantization Effects on LLM Performance

Quantization strategies, such as FP32 (Single Precision Floating Point), FP16 (Half Precision Floating Point), and BNB 4bit (BitsAndBytes[1]), significantly impact the performance metrics of LLMs. Our analysis of the LLaMA 7B model [92] on the A100 device reveals distinct outcomes for each quantization strategy. Transitioning from FP32 to FP16 quantization results in a substantial reduction in M_{peak}, as shown in Figure 6.6. This reduction indicates that FP16 representation requires less memory for model storage and computation. Consequently, the throughput increases significantly from FP32 to FP16. Adopting BNB 4bit quantization further reduces memory usage to 4848.55 MB while

[1]https://github.com/TimDettmers/bitsandbytes

maintaining a high throughput. The E decreases from 2.6×10^{-5} J (FP32) to 1.6×10^{-5} J (FP16) and then increases slightly to 1.8×10^{-5} J for BNB 4bit quantization, showcasing the trade-off between M_{peak} usage and E.

6.3.5 Feature Selection for Performance Prediction

To predict LLM performance metrics (T), (M_{peak}), and (E), we've compiled a comprehensive set of model and hardware features in the Table 6.3.

We measured the computational capabilities and memory bandwidth of Nvidia GPUs using two methods. We ran a CUDA[2] kernel for intensive floating-point calculations for (PF_{32}). Another CUDA kernel measured the data transfer rate for memory bandwidth, yielding peak bandwidth (PB_HBM) in GB/s. The other hardware features are directly extracted from CUDA APIs.

Model features were extracted by converting the LLMs to the ONNX format, facilitating the use of ONNX tool[3] for calculating (TM_{mem}), (TM), and (TP). Additionally, we traversed the ONNX graph to calculate the (TN) and (TE), providing a comprehensive understanding of the model's computational graph. The other model metrics like (HS), (NHL), (NAH), (VS), and (TVS) were directly extracted from the Hugging Face library[4] using the model ID.

6.4 Dataset Collection

Our dataset collection methodology enables a comprehensive analysis of LLMs across diverse computational paradigms, focusing on tree-based models and GNN baselines. To gather data, we conducted forward passes across various LLMs and hardware configurations to empirically measure T, M_{peak}, and E. This process involved initializing models and tokenizers, configuring CUDA settings, and systematically iterating over batch sizes B and sequence lengths S. Utilizing PyTorch for model operations and specialized trackers for latency, memory, and energy metrics, we ensured accurate measurements with a cooldown phase between iterations. We employed 100 warmup iterations W and 300 total iterations T for data collection.

The collected metrics were then compiled into a dataset $\mathcal{D}$, divided into tabular and graph formats for distinct analytical approaches. Algorithm 2, outlines the process for collecting LLM performance metrics, taking inputs such as batch sizes B, sequence lengths S, LLM models P, hardware configuration H, warmup steps W, and total steps T, and producing the dataset $\mathcal{D}$ containing the measured metrics.

[2]https://developer.nvidia.com/cuda-toolkit
[3]https://github.com/ThanatosShinji/onnx-tool
[4]https://huggingface.co/

Table 6.3: Model and Hardware Features

Feature (Abbreviation)	**Description**
	Model Features (MF)
BS	Batch Size
SL	Sequence Length
QT	Quantization Type
TN	Total Nodes
TE	Total Edges
TM	Total MACs
TP	Total Parameters
TM_{mem}	Total Memory
MPE	Max Position Embeddings
NHL	Num Hidden Layers
NAH	Num Attention Heads
HS	Hidden Size
VS	Vocab Size
TVS	Type Vocab Size
	Hardware Features (HF)
PF_{32}	Peak FP32 GFLOPS
PB_{HBM}	Peak Bandwidth HBM
CM	Compute Major
Cm	Compute Minor
MTPB	Max Threads Per Block
MTPM	Max Threads Per Multiprocessor
RPB	Regs Per Block
RPM	Regs Per Multiprocessor
WS	Warp Size
SMPB	Shared Mem Per Block
SMMP	Shared Mem Per Multiprocessor
SMS	Num Streaming Multiprocessors
SMPBO	Shared Mem Per Block Opt-In
CC	CPU Clock
RAM	System RAM
LC1	L1 Cache
LC2	L2 Cache
LC3	L3 Cache
NC	Num Cores
CA	CPU Architecture

Table 6.4: Hardware Specifications of Data Collection Devices

Specification	Juwels	Booster	Jueraca
CPU	2× Intel Xeon Gold 6148	2× AMD EPYC Rome 7402	2× Intel Xeon Platinum 8452Y
Cores	2× 20 cores, 2.4 GHz	2× 24 cores, 2.8 GHz	2× 36 cores, SMT-2
Memory	192 GB DDR4, 2666MHz	512 GB DDR4, 3200 MHz	512 GiB DDR5-4800 RAM
GPU	NVIDIA V100, 16 GB HBM	NVIDIA A100, 40 GB HBM2e	NVIDIA H100, 80 GB HBM
Network	2× InfiniBand EDR	4× InfiniBand HDR	1× BlueField-2 ConnectX-6

Algorithm 2: LLM Performance Metrics Collection

```
Input: $B = \{b_1, b_2, \ldots, b_n\}$, $S = \{s_1, s_2, \ldots, s_m\}$, $M = \{m_1, m_2, \ldots, m_k\}$, $H$,
       $W, T$
Output: Dataset $\mathcal{D}$
for $p \in P$ do
    Initialize model $m$, tokenizer $\tau$;
    for $b \in B$ do
        for $s \in S$ do
            if config $(m, b, s)$ not in $\mathcal{D}$ then
                CooldownGPU;
                Initialize trackers;
                Warm up $M$ for $W$ iterations using $\tau$;
                Record metrics for $T$ iterations;
                Calculate $T$, $M_{\text{peak}}$, $E_{\text{sample}}$, and $C$;
                $\mathcal{D} \leftarrow \mathcal{D} \cup \{(m, b, s, T, M_{\text{peak}}, E, C)\}$;

CooldownGPU() Clear CUDA cache and collect garbage;
Sleep for cooldown period;
```

6.4.1 Tabular and Graph Dataset Construction

For tree-based models, we organize performance metrics and extracted features into tabular format, saved as CSV files. This dataset directly aids algorithms like XGBoost, facilitating efficient performance prediction of LLMs across hardware setups.

To exploit relational information for GNN baseline comparison, we convert LLMs (in ONNX[5] format) into graphs compatible with PyTorch Geometric (PyG[6]). Each DL model, denoted as M with operations O, transforms into a graph $G = (X, A)$, where X and A represent node feature and adjacency matrices. Nodes in G correspond to operations in M, capturing computational characteristics and information flow. This transformation encodes operation types, input/output shapes, MACs, parameters, and memory requirements into node attribute vectors, enhancing graph representation fidelity. We utilized the same methodologies as previously published work for constructing graphs [89].

6.4.2 Evaluation Metrics

We assess model performance using three key metrics: Root Mean Square Error (RMSE), Mean Absolute Percentage Error (MAPE), and Kendall's Tau (τ). RMSE measures

[5]https://onnx.ai/
[6]https://www.pyg.org/

Table 6.5: Comparison of GNN and XGBoost Models Across Different Metrics

Target	GNN			XGBoost		
	MAPE (%) ↓	RMSE ↓	τ ↑	MAPE (%) ↓	RMSE ↓	τ ↑
Throughput	17.60	481.82	0.852	5.49	109.73	0.96
Memory	9.45	2657.45	0.874	1.81	1209.22	0.98
Energy	53.18	1.94	0.41	6.27	0.20	0.96

the average deviation between predicted and actual values, while MAPE quantifies the percentage error. Kendall's Tau evaluates the correlation between predicted and actual rankings.

6.5 Experiments and Results

6.5.1 Enviroment setup

We utilized three distinct hardware configurations for data collection, outlined in Table 6.4. Our experiments were specifically conducted on systems featuring NVIDIA A100 GPU. Additionally, we employed PyTorch 2.2.1, torch-geometric 2.5.0, ONNX 1.13.1, CUDA 12.1, and XGBoost 2.0.3 for the experimental setup.

6.5.2 GNN Baseline Model

Our GNN baseline model architecture takes an input graph $G = (X, A)$, where X captures node features and A represents the adjacency matrix. Two GraphSAGE [93] (GS) layers transform the node features: $X' = \text{ReLU}(\text{Dropout}(GS(X, A; \theta_1)), p = 0.05)$ and $X'' = \text{ReLU}(\text{Dropout}(GS(X', A; \theta_2)), p = 0.05)$. These layers extract meaningful representations from the input features. Static features S contains static $= [TM, TP, TM_{\text{mem}}, BS, SL, HF]$ undergo linear transformation to align their dimensionality with the output of the GS layers: $S' = \text{ReLU}(\text{Dropout}(\text{FC}_{S \to H_{\text{gnn}}}(S)), p = 0.05)$. The processed static features S' are concatenated with X'' to create a unified feature representation. The concatenated features are fed into fully connected layers, resulting in the final prediction y: $y = \text{FC}_{H_{\text{fc}} \to 1}(F')$, where $H_{\text{fc}} = 512$.

The model minimizes L_{MSE} using Adam (lr=0.001) over mini-batches of size 16. The choice of the GNN architecture is motivated by its widespread use in previous literature [63], although previous implementations often lacked hardware-related properties in their static features. We incorporated these hardware-related properties into the static features to ensure a fair comparison with the XGBoost model. This integration enables the Baseline model to effectively capture complex relationships among input graphs, hardware configurations, and model features, facilitating a more robust comparison with the XGBoost model.

6.5.3 Our Model: XGBoost

Our approach to predicting LLM performance metrics utilizes the XGBoost algorithm, renowned for its efficiency with structured data. XGBoost optimizes an objective function Obj, composed of a loss function L and a regularization term Ω, defined as:

$$\text{Obj} = \sum_i L(y_i, \hat{y}_i) + \sum_k \Omega(f_k) \tag{6.2}$$

where y_i represents true metric values, $\hat{y}_i$ denotes predictions, and f_k are decision trees. The loss function L measures prediction discrepancy, while Ω penalizes model complexity to prevent overfitting.

Hyperparameter tuning involves a grid search over parameters such as the number of estimators: [100, 500, 1000, 2000], learning rate: [1×10^{-1}, 1×10^{-2}, 1×10^{-3}], maximum tree depth: [3, 5, 8, 10, 30], and subsample ratios: [0.5, 0.75, 1]. The optimal set of hyperparameters includes a learning rate of 1×10^{-1}, a maximum depth of 8, 2000 estimators, and a subsample ratio of 1.

6.6 Experimental Evaluation

In our experimental evaluation, we aimed to assess the predictive capabilities of GNN and XGBoost models regarding T, M_{peak}, and E for LLMs. This assessment was conducted within the framework of stratified 5-fold cross-validation to ensure thorough and unbiased evaluation.

Each model, denoted by $\mathcal{M} \in \{\text{GNN}, \text{XGBoost}\}$, was trained on the training set and evaluated on the test set across 5 folds. The predictions for T, M_{peak}, and E for each fold are then averaged to quantify the overall performance of the model.

6.6.1 Baseline Comparison

The comparison between GNN and XGBoost models reveals stark differences in predictive accuracy, as shown in Table 6.5. XGBoost's superior performance may stem from its adeptness at handling structured data and leveraging ensemble learning techniques. In contrast, GNNs may struggle to capture complex relationships in LLM data due to inherent limitations in modeling graph structures effectively.

The observed disparities may also be attributed to the algorithmic complexity of each model. XGBoost optimizes for predictive accuracy through iterative improvement, while GNNs rely on message-passing mechanisms, potentially hindering their ability to capture nuanced patterns in LLM data.

6.6.2 Ablation Study

We conduct an ablation study to assess the impact of feature inclusion and hardware specifics on the XGBoost model as shown in Table 6.6. The study examines different feature sets: MAC operations TM, combined MACs with parameters and memory $TM +$

Table 6.6: Ablation Study on the Effect of Model (MF) and Hardware Features (HF) on Prediction Metrics (Throughput (T), Peak Memory Usage (M_{peak}), and Energy Consumption (E))

Metrics	Features	Without HF			With HF		
		MAPE (%) $\downarrow$	RMSE $\downarrow$	$\tau \uparrow$	MAPE(%) $\downarrow$	RMSE $\downarrow$	$\tau \uparrow$
T	TM	39.36	682.13	0.782	17.89	406.54	0.889
	$TM + TP + TM_{\text{mem}}$	43.25	695.95	0.764	6.16	116.17	0.959
	MF	43.19	695.79	0.764	5.49	109.73	0.962
M_{peak}	TM	24.21	1742.64	0.607	11.68	1785.61	0.804
	$TM + TP + TM_{\text{mem}}$	25.57	1660.02	0.608	1.94	1484.39	0.980
	MF	25.44	1468.69	0.608	1.81	1209.22	0.980
E	TM	31.08	0.77	0.833	14.82	0.42	0.912
	$TM + TP + TM_{\text{mem}}$	32.21	0.87	0.822	6.28	0.21	0.959
	MF	32.11	0.87	0.822	6.27	0.20	0.959

$TP + TM_{\text{mem}}$, and all available model features MF, across three key metrics: T, M_{peak}, and E.

Initially, TM features provide baseline accuracy, with notable enhancements observed when incorporating model parameters and memory usage $TM + TP + TM_{\text{mem}}$. Integrating hardware details, including FLOPS and bandwidth, significantly boosts predictive performance. Utilizing the full model feature set alongside hardware features yields the most substantial improvements, emphasizing the pivotal role of hardware in computational performance.

6.6.3 Efficiency Analysis of Predictive Models

Our evaluation rigorously compares the computational efficiency of the Baseline GNN and XGBoost models, focusing on throughput as a crucial performance metric. The experiments were conducted on Booster CPUs specified in Table 6.4. As depicted in Figure 6.7, XGBoost exhibits markedly superior efficiency in predicting performance for LLMs. This efficiency advantage is attributed to XGBoost's optimized algorithms, specifically tailored for handling structured data and large datasets. In contrast, GNNs, encounter challenges due to the intricate graph structures inherent in LLMs.

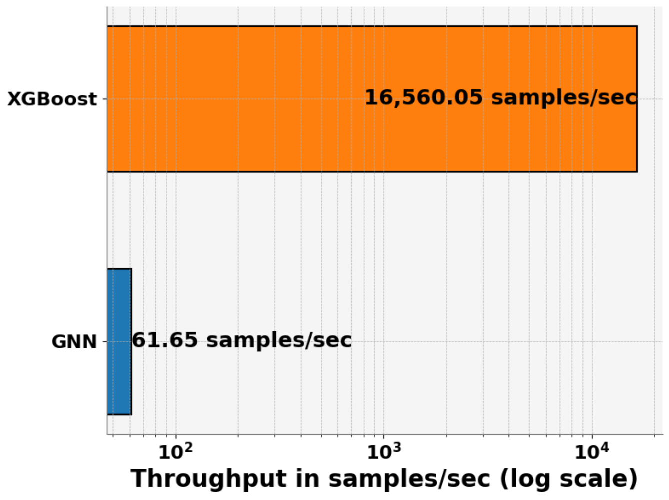


Figure 6.7: Comparison of model throughput on CPU between GNN and XGBoost models, illustrating the significant efficiency advantage of the XGBoost model in processing samples per second.

6.6.4 Usability

We developed a user-friendly interface developed with Gradio[7], streamlining the prediction of performance metrics (T), M_{peak}), and carbon emissions (C) for LLMs. This

[7]https://www.gradio.app/

interface allows users to input a model ID from the Hugging Face repository, batch size, sequence length, and device selection (H100, A100, V100), or directly input hardware features. Behind this interface lies a backend that transforms user inputs into a feature set as mentioned in Table 6.3, then features are passed to pre-trained XGBoost models to predict performance metrics accurately. This seamless integration of a user-friendly interface with powerful predictive capabilities signifies a significant advancement in optimizing LLM deployments, fostering more efficient and environmentally conscious AI applications.

6.7 Conclusion

Our study demonstrates the superiority of XGBoost over Baseline GNN in predicting Large Language Model (LLM) performance across diverse hardware configurations. XGBoost achieves substantial improvements in throughput, memory usage, and energy consumption prediction accuracy, with relative enhancements of MAPE approximately 68.81%, 80.85%, and 88.21% compared to the GNN baseline, respectively. Remarkably, XGBoost achieves a remarkable 26761.39% relative increase in throughput speed over GNNs.

These findings highlight XGBoost's efficiency in rapid prediction processing, providing a strong basis for optimizing hardware selection strategies. By enhancing computational efficiency and advocating for sustainable LLM deployment, our research makes a significant contribution to the AI community, facilitating informed decision-making and promoting environmental consciousness in LLM deployment.

7 Can LLMs Enhance Performance Prediction for Deep Learning Models?

Accurate performance prediction of Deep Learning (DL) models is essential for efficient resource allocation and optimizations in various stages of the DL system stack. While existing approaches can achieve high prediction accuracy, they lack ability to quickly adapt to new hardware environments or emerging workloads. This paper leverages both Graph Neural Networks (GNNs) and Large Language Models (LLMs) to enhance the accuracy and adaptability of DL performance prediction. Our intuition is that GNNs are adept at capturing the structural information of DL models, naturally represented as graphs, while LLMs provide generalization and the ability to quickly adapt to various tasks thanks to extensive pre-training data. We empirically demonstrate that using GNN-derived graph embeddings as inputs to an LLM outperforms traditional representations, including high-level text summary and lossless semi-structured text (e.g., JSON), for this task. Furthermore, we propose a structured pre-training strategy to enable model adaptation to new hardware environments, significantly reducing the need for extensive retraining. Our experiments validate the effectiveness of this approach, showing an 8.8 percentage-point improvement in accuracy over a state-of-the-art GNN baseline. Notably, when adapted to new hardware with few samples, our method achieves a remarkable 30–70 percentage-point increase in accuracy compared to the GNN baseline.

This chapter builds upon the research presented in the following publication:

Peer reviewed

Karthick Panner Selvam, Phitchaya Mangpo Phothilimthana, Sami Abu-El-Haija, Bryan Perozzi and Mats Brorsson. "Can LLMs Enhance Performance Prediction for Deep Learning Models?" in the 2024 WANT Workshop at the International Conference on Machine Learning (ICML) . https://openreview.net/pdf?id=bpS4vaOg7q. *This research was also presented at the 2024 KDD Ph.D. Consortium and featured in a Google Research expo talk at ICML 2024..*

Contents

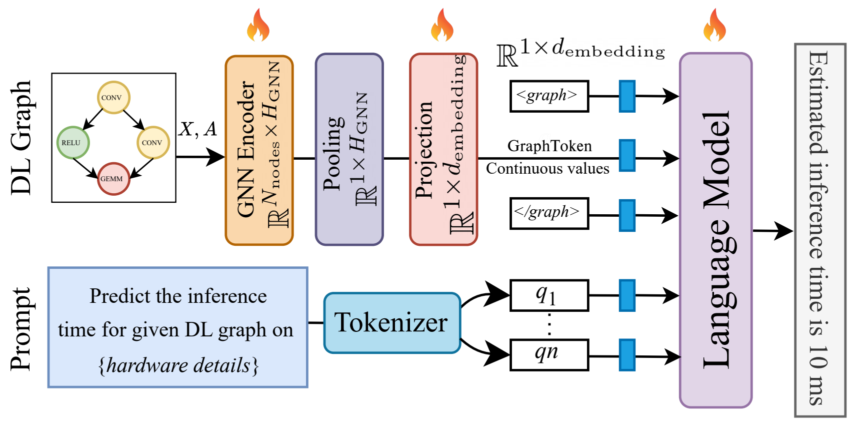

DL Graph
CONV
RELU
CONV
GEMM
X, A
GNN Encoder
$\mathbb{R}^{N_{\text{nodes}} \times H_{\text{GNN}}}$
Pooling
$\mathbb{R}^{1 \times H_{\text{GNN}}}$
Projection
$\mathbb{R}^{1 \times d_{\text{embedding}}}$
$\mathbb{R}^{1 \times d_{\text{embedding}}}$
<graph>
GraphToken
Continuous values
</graph>
Prompt
Predict the inference time for given DL graph on {hardware details}
Tokenizer
q_1
qn
Language Model
Estimated inference time is 10 ms

Figure 7.1: Our approach integrates GNNs and LLMs for DL model performance prediction. The methodology utilizes soft prompting to fine-tune pre-trained GNN weights and projection layer weights, while updating pre-trained LLMs with the LoRA technique. During fine-tuning, gradients flow from the LLM to the GNN, allowing the system to process graphs and prompts effectively and generate accurate performance metrics predictions.

7.1 Introduction

Performance prediction for Deep Learning (DL) models is essential for all sorts of optimization methods in the DL system stack: from Neural Architecture Search, to model partitioning and sharding, to low-level compiler optimizations. Performance prediction involves estimating various operational metrics — such as inference time, memory usage, and power consumption — that are crucial for efficient hardware utilization and scheduling. Since DL models are computation graphs, researchers have employed Graph Neural Networks (GNNs) to extract information from the DL model for various optimization decisions given hardware components [25, 63, 89, 94].

Unfortunately, aforementioned GNN-based approaches require comprehensive retraining to accommodate new hardware environments or DL architectures, often requiring large labeled datasets. These requirements can hinder a rapid adaptation and optimization, limiting the flexibility of these models when new architectures or configurations emerge. Fortunately, the recent successes of Large Language Models (LLMs) in various domains have underscored their capability to understand and generate complex systems [95–99]. This includes not only natural language but also structured data such as code, configuration settings, and textual descriptions of hardware configurations and DL architectures. Given their extensive pre-training on diverse datasets, LLMs can generalize effectively when fine-tuned on specific tasks. Their generalization capability makes them good candidates for enhancing DL performance prediction.

However, employing LLMs in the performance prediction domain poses challenges, primarily due to the need for representing DL models in a format that LLMs can efficiently process. Prior works have considered using high-level descriptions to represent programs and graphs as text inputs for compiler optimizations and performance predictions [100, 101]. Nonetheless, these representations often fail to maintain the full structural intricacies of DL models, losing crucial connectivity and hierarchical information. An alternative representation is to use structured text format (e.g. JSON, XML, Protobuf, etc.), which maintains detailed information of node features and their connections. However, DL models can contain tens-of-thousands of nodes [94], which can hinder the processing efficiency and scalability when used with LLMs.

Recent research has explored the use of GNNs as encoders to convert graph data into embeddings as inputs to LLMs, thereby effectively bridging the gap between graph data and the textual input preferred by LLMs. However, these studies primarily focus on graph-based question answering, rather than directly on performance prediction [102, 103].

In line with [102] findings, We hypothesize that graph embeddings, derived from GNNs, represent DL models more effectively for performance prediction than conventional text representations because the graph embeddings could better capture structures and connectivity. Based on this hypothesis, we propose the GNN-LLM model for DL performance prediction, as illustrated in Figure 7.1. Our experiment has confirmed that using graph embeddings significantly outperforms using a semi-structured text format (JSON) and a high-level text format in both accuracy and computational efficiency. Specifically, our approach surpasses a JSON format by approximately 6% in accuracy and is 21 times faster in terms of training time. Likewise, our approach surpasses high-level text by 134% in accuracy and is 2 times faster in terms of training time, demonstrating a substantial improvement over text-based representation.

To enhance the adaptability and accuracy of the model, we further develop a structured pre-training strategy that obviates the need for extensive retraining from scratch. The approach begins by training a GNN using a mask autoencoding technique on unlabeled DL models, inspired by [23] research. In this initial phase, the GNN learns to capture DL graph structures and node information. Subsequently, we refine the integration between the DL graph data and the LLM by fine-tuning the projection layer and the LLM through a graph-to-text task. This graph-to-text translation will enable the LLM to comprehend DL graph structures and improve the model's ability to adapt to new hardware with minimal training samples for downstream performance prediction tasks. Finally, all components are fine-tuned for the final performance prediction task.

In the evaluation, our method achieves a 8.8 percentage-point increase in accuracy over the state-of-the-art GNN baseline on the NNLQP multi-platform dataset, and a remarkable 30–70 percentage-point increase in accuracy when adapted to new hardware with few samples. The results confirms our method's efficacy in enhancing both the accuracy and adaptability of performance predictions across varied computational

Table 7.1: Comparison of GNN baselines against our proposed architecture for performance prediction of new DL architectures (ViT) with sparse training samples.

Models	**MAPE ↓**	**Acc (10%) ↑**
GNN-Baseline	26.86	0
GNN-NF	20.32	1
Ours-Llama3-8B	3.36	95.17
Ours-Mistral7B	**1.69**	**99.05**

environments.

1. We empirically evaluate different DL model representations for LLMs on performance prediction tasks, showing that a graph embedding-based input is most effective.

2. We introduce a method integrating GNNs and LLMs for the DL performance prediction domain, combining GNNs' structural insights and LLMs' generalization capabilities.

3. We propose a structured pre-training strategy to enhance model performance in a new hardware environment with limited training samples.

4. We contribute a specialized graph-to-text dataset designed to further research into the integration of GNN and LLMs. This dataset is particularly valuable for benchmarking and advancing the application of GNN-LLM combinations in graph learning tasks.

5. Our research offers a promising direction for improving DL performance prediction accuracy and adaptability across diverse hardware environments.

7.2 Related Work

The field of performance prediction for DL models has witnessed growing interest in recent years. Early work by [66] proposed an analytical model to estimate DL model training time. Subsequent studies like that of [60] extended these methods to predict memory consumption, utilizing analytical models to estimate resource utilization during training. To improve prediction accuracy, researchers have explored machine learning approaches. [10] used an MLP-based regressor with features like trainable parameter counts, but it was constrained by a shallow understanding of DL layers' dynamics. Others, such as [13], [62], [18], [104] and adopted a layer-by-layer technique, they predicted performance for each layer instead of the whole model, incorporating parameters like FLOPs and layer features to predict execution times and power consumption.

However, this layerwise strategy failed to capture the network structure of DL models [63] To address this limitation, many methods [19, 20, 25, 61, 63, 88, 89, 94, 105] utilized graph learning techniques to generate embeddings that encapsulate the DL model network topology, as well as the features of the computation graph. These embeddings are trained to predict performance characteristics. [88] improved the work of [63] by introducing graph attention based transformer block to predict the latency of the given DL model. However, their architecture doesn't support multi-platform performance prediction.

Despite these advancements, prior approaches lack online adaptability. Current methods require retraining for new DL architectures or hardware configurations. On the other hand, our proposed approach aims to overcome this challenge by integrating GNNs with LLMs to create a predictive system that is more adaptable and flexible in real-world scenarios.

7.3 Background

7.3.1 DL Models as Computational Graphs & Graph Neural Networks (GNNs)

DL models can be represented as directed acyclic computational graphs, where nodes correspond to mathematical operations and edges represent data flow between these operations. The input features of every node include the *op code* (*e.g.*, einsum, relu, *etc*), the output data type (*e.g.*, float32, uint8, *etc*) and the shape of the output tensor – See [94] for comprehensive list of node-wise features.

GNNs are designed to operate on graph-structured data. Let graph of n nodes be represented with a node feature matrix $\mathbf{X} \in \mathbb{R}^{N \times \cdot}$ and an adjacency matrix $\mathbf{A} \in \{0,1\}^{N \times N}$. GNNs use an iterative message passing process to generate embeddings for nodes. During message passing, each node updates its embedding by aggregating information from its neighbors. GNN layer can be written as:

$$\mathbf{H}^{(l)} = \text{TRANSFORM}\left(\mathbf{H}_i^{(l-1)}, \mathbf{A}\right) \tag{7.1}$$

where $\mathbf{H}^{(l)}$ is the node embedding matrix at the l-th layer and $\mathbf{H}^{(l)} = \mathbf{X}$. Through multiple message-passing layers, each node aggregates information from a wider neighborhood, capturing both immediate and distant neighbor information. GNNs have excelled in tasks such as node classification, link prediction, and graph-level classification. There are many possible choices for `TRANSFORM` function [55, 64, 65, 106]. In our work, we use a variant of the GIN model [106]:

$$\mathbf{H}_{\text{GIN}}^{(l)} = \text{MLP}\left(\left(\mathbf{A} + \mathbf{I}\epsilon\right)\mathbf{H}^{(l-1)}\right), \tag{7.2}$$

where MLP stands for multi-layer perceptron, $\mathbf{I}$ is $n \times n$ identity matrix, and ϵ is small constant.

7.3.2 Large Language Models

7.3.2.1 Pre-trained Large Langauge Models

Pre-trained LLMs are advanced neural networks for natural language processing tasks. They leverage the Transformer architecture [107], which uses self-attention mechanisms to manage long-range dependencies in text. LLMs are pre-trained on extensive corpora to predict subsequent tokens, enabling them to capture intricate linguistic patterns. This pre-training is followed by finetuning task-specific datasets to adapt to various applications like text classification and translation.

7.3.2.2 Parameter-Efficient Fine-Tuning

With the rapid increase in the size of state-of-the-art LLMs, traditional fine-tuning has become resource intensive. Parameter-Efficient Fine-Tuning (PEFT) aims to adapt models to new tasks by updating only a small subset of parameters [108].

Low-Rank Adaptation (LoRA): LoRA introduces low-rank matrices into model layers, represented as $\Delta W = BA$, where B and A are trainable low-rank matrices. This approach reduces the computational burden by updating fewer parameters while keeping the main model's parameters frozen, thus preserving the pre-trained knowledge [29].

Soft Prompting: Soft prompts are learnable vectors integrated into the model's input to guide its behavior toward specific tasks. This method updates only a small number of parameters, making it computationally efficient and preserving the broad knowledge of the model [30].

7.4 Methodology

Figure 7.1 displays our proposed model architecture. Our approach takes a DL model graph and a textual prompt as inputs. The DL graph is initially processed by a GNN encoder and then projected as an embedding to an LLM, along with the token embeddings of the textual prompt.

7.4.1 DL Representation

We consider the following methods to represent DL models for processing by LLMs.

7.4.1.0.1 Graph Representation. This method first encodes a DL model in the Open Neural Network Exchange (ONNX) format, represented as a graph with node feature matrix $\mathbf{X}$ formulated as:

$$\mathbf{X}_v = \mathbf{X}_v^{(\text{op})} \oplus \mathbf{X}_v^{(\text{attr})} \oplus \mathbf{X}_v^{(\text{shape})} \quad \forall v \leq N \tag{7.3}$$

where $\mathbf{X}_v^{(\text{op})}$ is the one-hot encoded vector indicating the type of the node operation. $\mathbf{X}_v^{(\text{attr})}$ includes the node's attribute vector, containing parameters such as kernel size and stride, and $\mathbf{X}_v^{(\text{shape})}$ encodes the output shape. The operation $\oplus$ represents a vector concatenation. This method is adapted from the framework established in [63]. Subsequently, we feed the node feature matrix $\mathbf{X}$ and the adjacency matrix $\mathbf{A}$ into the GNN. The GNN then produces a graph embedding for input into the LLM, along with prompt's token embeddings, to predict the model's performance.

7.4.1.0.2 High-level Text Representation. We use a predefined template that captures essential computational and structural properties of a DL model. This includes overall model statistics — such as FLOPs, parameter count, and batch size — offering insights into the model's complexity and capacity. We also include layer-specific statistics, detailing each layer's FLOPs and parameter counts. These elements together offer a holistic view of a DL model's architecture and its computational behavior. To predict its performance, we simply tokenize and apply a conventional word encoder on the textual prompt for LLM processing.

7.4.1.0.3 Semi-Structured Text Representation. We adopt a semi-structured JSON format to comprehensively encapsulate a DL architecture. This format itemizes each node's characteristics, including the operator type, input and output shapes, computation complexities, and node attributes. Additionally, it capture node connectivity. For LLM processing, we tokenize and apply a conventional word encoder on the semi-structured description.

7.4.2 Graph Encoding

Our GNN encoder is based on the Graph Isomorphism Network (GIN) [106], defined as:

$$\begin{gathered}\mathbf{H}^{(l)}_{\text{ours}} = (\mathbf{A} + \mathbf{I}\epsilon)\,\text{MLP}(\mathbf{H}^{(l-1)}) \\ \text{with MLP}(\mathbf{Z}) = \text{ReLU}(\text{BN}(\mathbf{Z}\mathbf{W}_1 + \mathbf{b}_1))\mathbf{W}_2 + \mathbf{b}_2\end{gathered} \tag{7.4}$$

We inspired this architecture by [23]. This setup ensures each node feature undergoes transformation, normalization, and activation, promoting the learning of non-linear dependencies. After processing through L layers, we aggregate node features to form a graph-level representation:

$$\mathbf{g} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{H}_i^{(L)}. \tag{7.5}$$

Next, the projection layer transforms the GNN output $\mathbf{g}$ into an embedding vector of size $d_{\text{embedding}}$ for the LLM processing.:

$$\text{GraphToken} = \text{MLP}_{\text{proj}}(\mathbf{g}), \tag{7.6}$$

where MLP_{proj} encapsulates a series of linear transformations and non-linear activations. It ensures the alignment of dimensionalities and contextual relevance. Note that the output dimension size of the projection is larger than the input dimension size: $|\text{GraphToken}| > |\mathbf{g}|$.

The LLM input is then constructed by integrating graph embeddings GraphToken with token embeddings $\mathbf{Q}$. A textual prompt describing the task like "`Predict the inference time of DL model`" is tokenized as $\mathbf{q} = [q_1, q_2, \ldots, q_n]$. The tokens are then converted into word embeddings: $\mathbf{Q} = \mathbf{E}[\mathbf{q}]$, where $\mathbf{E}$ represents the embedding matrix. The complete LLM input is the concatenation of the projected graph embedding and the token embeddings:

$$\text{Input}_{\text{LLM}} = [\text{<graph>}, \text{GraphToken}, \text{</graph>}, \mathbf{Q}]. \tag{7.7}$$

In this sequence, <graph> and </graph> are text tokens directly generated by the tokenizer, marking the beginning and end of the graph embedding. A single graph embedding vector GraphToken efficiently encapsulates the entire graph's structure, compactly representing complex information in a form that complements textual embeddings in LLMs.

7.4.3 Training Strategy: 3-stage training

We hypothesize that directly fine-tuning both LLMs and GNNs for performance prediction tasks, starting from scratch, may not yield optimal adaptability for new tasks. The challenge lies in the initial lack of domain-specific knowledge, which is crucial for the model to effectively process and predict the DL performance metrics. To address this, we propose a novel structured pre-training methodology, designed to enhance the model's intrinsic understanding of DL graph structures before fine-tuning for performance prediction. The pre-training strategy comprises the three stages as shown in Figure 7.2.

7.4.3.0.1 Stage 1. GNN Pre-training. We employ the Graph Maked Auto Encoder technique (GraphMAE) for GNN pre-training [23]. We use GIN as both encoder and decoder. Given a DL graph with $\mathbf{X}$ and $\mathbf{A}$, we mask a portion of $\mathbf{X}$ using a learnable mask vector to produce $\widetilde{\mathbf{X}}$. The GIN encoder processes $(\widetilde{\mathbf{X}}, A)$ to generate latent embeddings $\mathbf{Z}$, effectively capturing the obscured structural details. The GIN decoder reconstructs the node features from $\mathbf{Z}$ to $\widehat{\mathbf{X}}$, aimed at closely approximating the original $\mathbf{X}$. Reconstruction accuracy is quantified using Scaled Cosine Error (SCE), which evaluates alignment in both direction and magnitude of the feature vectors. Using GIN for both encoding and decoding optimizes the preservation and reconstruction of local graph structures, essential for understanding DL graphs. The SCE, by assessing both vector orientation and length, enhances model sensitivity to structural and feature variations, preparing it for robust performance on subsequent tasks.

7.4.3.0.2 Stage 2. Graph-Text Adaptation. For this stage we update only projection layer and LLM weights. The projection layer $\mathbf{W}_p$ adapts the graph embeddings GraphToken for integration with the LLM. During training, we update the projection layer weights using soft prompting techniques.

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_p} = \frac{\partial \mathcal{L}}{\partial \text{Output}} \cdot \frac{\partial \text{Output}}{\partial \text{GraphToken}} \cdot \text{GraphToken}^T$$

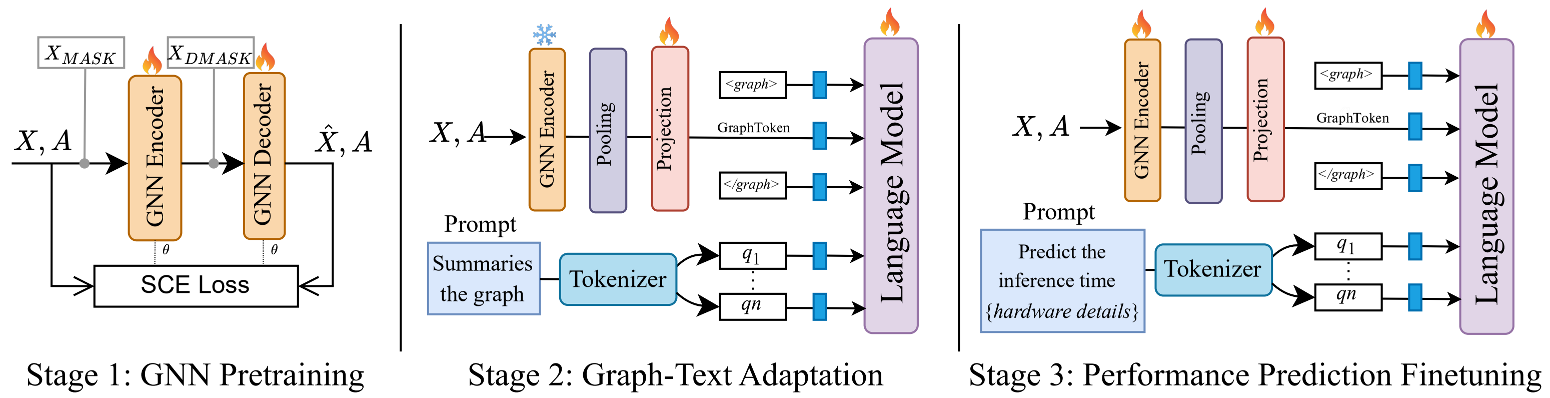


Figure 7.2: The three stages of our approach: (1) **GNN Pre-training**: Using Scaled Cosine Error (SCE) loss with masked node features (X_{MASK}) approach to pre-train the GNN. (2) **Graph-Text Adaptation**: Fine-tuning the pre-trained GNN encoder (frozen) and updating LLM weights and projection weights using soft prompting and LoRA techniques. (3) **Performance Prediction Fine-tuning**: Updating all GNN projection and LLM parameters through soft prompting and LoRA techniques to predict performance metrics for deep learning graphs on various hardware.

Here, $\frac{\partial \mathcal{L}}{\partial \text{Output}}$ represents the gradient of the loss with respect to the LLM's output, and $\frac{\partial \text{Output}}{\partial \text{GraphToken}}$ captures how changes in GraphToken affect the output. We used cross-entropy loss for the next word prediction. We utilize the LoRA technique to efficiently update the LLM weights. The updates for the low-rank matrices $\mathbf{B}$ and $\mathbf{C}$ are given by:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{B}} = \frac{\partial \mathcal{L}}{\partial \Delta \mathbf{W}} \cdot \mathbf{C}^T, \quad \frac{\partial \mathcal{L}}{\partial \mathbf{C}} = \mathbf{B}^T \cdot \frac{\partial \mathcal{L}}{\partial \Delta \mathbf{W}}$$

where $\Delta \mathbf{W} = \mathbf{BC}$ represents the low-rank update to the LLM weights. The GNN encoder weights remain frozen during this stage to preserve the integrity of the initial graph embeddings learned during pre-training. This selective updating strategy helps maintain foundational graph understanding and ensures consistent model performance across various adaptation scenarios.

7.4.3.0.3 Stage 3. Performance Prediction Fine-Tuning. In this final stage, we load the pre-trained GIN encoder weights from Stage 1 and the projection $\mathbf{W}_p$ and LoRA weights from Stage 2. We fine-tune the entire GNN to LLM model for performance prediction.

Note that naively feeding the GNN embedding outputs as multiple concrete text tokens to the LLM does not work because the gradient does not flow from the LLM to the GNN. This is why we adopt the proposed approach.

7.4.4 Training Datasets.

We utilize three distinct datasets for the different stages of our model's training process.

For GNN pre-training, we use a dataset containing 20,000 unlabeled DL graphs, [63], including of ten DL model families (ResNets, EfficientNets, MobileNetV2s, MobileNetV3s, MnasNets, SqueezeNets, VGGs, Alexnets, NasBench201s and GoogleNets). These ONNX models are transformed into node feature matrices and adjacency matrices, then converted these into the PyTorch Geometric data format (PyG), as detailed in Section 7.4.1, for GNN pre-training (Section 7.4.3). This extensive set of graphs allows our GNN to capture a wide range of node and edge features, providing robust initial embeddings.

For graph-to-text adaptation, we introduce a novel dataset based on [63] dataset. We structured the dataset into $\{(G, Q, A)\}$ format: G represents the DL model's graph structure in PyG format, Q is a textual prompt (*Summarise the graph*), and A is the summary of DL architecture, which is the response from the LLM (as referenced in the sample prompt in Section 7.7.3). The summary provides comprehensive details, including the total number of nodes, edges, model complexity, and statistics for each layer. The dataset comprises 20,000 prompts.

For performance prediction fine-tuning, we use the NNLQP Multi-platform dataset, which includes ten DL model families across nine computational architectures (cpu-openppl-fp32, hi3559A-nnie11-int8, gpu-T4-trt7.1-fp32, gpu-T4-trt7.1-int8, gpu-P4-trt7.1-

Table 7.2: Performance comparison of different representations of DL models for performance prediction tasks. Our proposed method (DL graph as embedding) demonstrates superior accuracy and efficiency, outperforming both JSON and high-level text representations.

Method	MAPE ↓	Acc(10%) ↑	TTT(hr) ↓	Max Token Length ↓
Text	41.42	22.50	0.46	512
JSON	13.55	49.80	5.02	2048
Ours-Llama3-8B	**12.61**	**52.83**	**0.23**	512

fp32, gpu-P4-trt7.1-int8, hi3519A-nnie12-int8, atlas300-acl-fp16, mul270-neuware-int8). This dataset contains DL graphs, platform IDs, and inference latency metrics. It consists of 7,396 graphs for training and 3,201 for testing. During the forward pass, queries Q are provided to the LLM, combined with G, to predict inference times. The predicted inference latency A is directly derived from the LLM's output.

7.5 Experiments

This section presents a series of experiments designed to validate the efficacy of our integrated model for DL performance prediction. We utilized our performance prediction datasets described in Section 7.4.4 to challenge our model under different conditions and compared it with the GNN baseline to underscore its advantages and unique capabilities. The computing details are explained in Section 7.7.1. To assess the accuracy of our performance prediction models, we used Mean Absolute Percentage Error (MAPE) and Accuracy within a delta threshold (ACC(δ)) metrics as detailed in Section 7.7.2.

7.5.1 Experiment: DL Representation

This experiment explores the efficacy of different representations of DL models for performance prediction tasks using LLMs. The DL representations are mentioned in Section 7.4.1. We investigated three primary formats: our proposed method (DL graph as embedding), semi-structured format (JSON), and high-level text. Each format presents unique challenges in how effectively it can be processed by LLMs.

Setting: For this experiment, we utilized the Llama3-8B[1] pre-trained model as the base LLM. The Adam optimizer was used with a learning rate of 1×10^{-5}, and LoRA with rank 8 was employed for efficient parameter updating. The GIN encoder used a learning rate of 1×10^{-3}. Each model was trained over 10 epochs, repeated 3 times to ensure stability and convergence of results.

In experiments with the performance prediction dataset, the entire ONNX models was converted to JSON format and tokenized using the Llama3-8B model tokenizer to

[1]https://llama.meta.com/llama3/

Table 7.3: Impact of LoRA rank (α) on our model performance and trainable parameters (θ) in millions using Llama3-8B as base LLM. The rank 8 achieves the best balance of accuracy and efficiency.

α	MAPE (%) $\downarrow$	Acc (%) $\uparrow$	θ (M)
8	**12.61**	52.83	**24**
16	12.74	51.50	26
32	12.63	**54.50**	33
64	13.43	49.50	47
128	13.86	44.50	74

assess context length. The JSON format reached a maximum context length of 18,000 tokens. Therefore, we selected the AlexNet family in the dataset due to its shorter context length compared to other families. A 90:10 train-test split was used for this experiment, consistent with previous work [63].

Result: Our proposed approach outperforms both JSON and high-level text representations significantly in terms of MAPE and ACC(10%), as shown in Table 7.2. Our method also demonstrated substantial efficiencies in training time, with the Total Training Time (TTT) notably lower than that required for JSON, which had the highest tokenization length and training duration. These results highlight the critical impact of DL model representation on the performance prediction capabilities of LLMs. High-level text, while simple, fails to capture the necessary connectivity information, leading to poor prediction accuracy. The semi-structured JSON format offers some improvement by providing hierarchical data, but its verbosity and resulting long token sequences increase computational costs. Our proposed method, which embeds the DL graph structure into a compact representation, strikes an optimal balance by preserving essential connectivity information within a manageable token length. This approach not only enhances prediction accuracy but also ensures computational efficiency. The graph embeddings naturally align with the inherent structure of DL models, enabling the LLM to process and predict performance metrics more effectively.

In our architecture, we update the LLM through the LoRA component. Therefore, we conducted an additional experiment on varying the rank size in LoRA updates. As detailed in Table 7.3, a rank of 8 offers the optimal trade-off between model performance and computational efficiency. While higher rank (32) can improve accuracy, they do so at the cost of a substantial increase in θ of 33M. The rank 8 configuration achieves the lowest MAPE while maintaining accuracy, all with the θ of 24M. This configuration proves to be the most efficient choice for our proposed architecture.

Table 7.4: Performance comparison of the GNN baselines against our models with different base LLMs (Llama3-8B and Mistral-7B) using a multi-platform performance prediction dataset. Both our models utilize GNN pre-training and graph-text adaptation. The test results demonstrate that our approach outperforms the all baselines, highlighting the effectiveness of the integrated method.

Models	**MAPE** $\downarrow$	**Acc (10%)** $\uparrow$
GNN-Baseline	12.96	51.72
GNN-NF	12.75	53.93
GNN-DistilRoberta	16.02	40.97
GNN-MiniLM	18.38	34.90
GNN-mpnet	16.12	40.37
Ours-Llama3-8B	<u>12.50</u>	<u>57.10</u>
Ours-Mistral7B	**11.89**	**60.49**

7.5.2 Experiment: Comparison with State-of-the-Art GNN

To rigorously evaluate our proposed architecture, we conducted a comparative analysis against the established GNN baseline [63] model across the multi-platform performance prediction dataset, which contains ten different DL model familes and nine different hardware platforms, as mentioned in Section 7.4.4. This comparison is crucial to validate the enhancements offered by our approach, particularly in terms of accuracy. In this experiment, we used two variants of our model: one with Llama3-8B and one with Mistral-7B [109] as the base LLM, both utilizing GNN pre-training and graph-text adaptation.

Settings: The baseline GNN model was utilized with no architectural modifications as described in its original implementation. For both of our models, we used the Adam optimizer with a learning rate of 0.0001 for the LLM and 0.001 for the GNN. We trained all models for 10 epochs conducted three times.

Results: According to the results shown in Table 7.4, both variants of our model with the pre-training strategy outperformed the GNN baseline. Notably, our model with the Mistral-7B base LLM outperforms the baseline by approximately 8.26% reduction in MAPE and 16.96% (8.8 percentage-point) increase in Acc (10%). To further improve the baseline, we incorporated additional node features such as FLOPS, MACs, and the number of parameters to enhance the representation of the DL models in the graph. This enhanced GNN baseline (referred to as GNN-NF) was designed to provide more comprehensive information for each node, ensuring that the model had a richer set of features to inform its performance predictions. However, our proposed architecture still outperforms the enhanced GNN model, confirming that the performance improvements are indeed attributable to the LLM integration, rather than merely to a more comprehensive feature representation. These results highlight the critical impact of the effective model

Table 7.5: Performance comparison of LLM models with and without graph-text adaptation combined with GNNs having either random or pre-trained weights. Results indicate that graph-text adaptation significantly improves LLM performance.

Models	**MAPE** ↓	**Acc (10%)** ↑
LLM + GNN	14.71	49.27
LLM + GNN_{PRE}	20.02	36.58
LLM_{PRE} + GNN	13.57	55.12
LLM_{PRE} + GNN_{PRE}	**12.50**	**57.10**

representation and the pre-training strategy on the performance prediction capabilities of LLMs.

Justification for the Proposed Architecture: Initially, we experimented with simpler models, which yielded suboptimal results, as summarized in Table 7.4. These models involved modifying the NNLQP GNN architecture by adding text-based descriptions of static features and hardware configurations, represented using embeddings generated by three variants of the pre-trained sentence-BERT language model (MiniLM-L6-v2, mpnet-base-v2, distilroberta-v1) [110]. These embeddings were then concatenated with GNN embeddings to predict performance. However, these models consistently underperformed when compared to the GNN baseline, highlighting the need for a more advanced architecture. In contrast, our proposed method, which integrates large pre-trained LLMs like Mistral-7B, demonstrated superior performance across all metrics. Notably, despite utilizing large LLMs, our model maintains a compact size of 24M trainable parameters through the efficient use of LoRA for weight updates.

Additionally, the results show that the choice of LLM significantly affects performance prediction accuracy. For instance, our model with the Mistral-7B base LLM consistently outperforms across various platforms, as shown in Table 7.8, demonstrating the importance of model selection in achieving higher accuracy.

7.5.3 Experiment: Effect of Pre-training Strategy

This study assesses the impacts of the GNN pre-training and the graph-to-text adaptation. The hypothesis driving this experiment is that pre-training can provide foundational knowledge that aids in subsequent performance prediction tasks. In this study, we leverage the performance prediction fine-tuning dataset as mentioned in Section 7.4.4. We used the Llama3-8B as the base LLM, optimizing with a learning rate of 0.0001 for the LLM and 0.001 for the GNN. The training was conducted over 10 epochs for 3 times.

Result: The configuration with graph-text adaptation (LLM_{PRE}) and pre-trained GNN initialization (GNN_{PRE}) significantly outperforms other setups as shown in Table 7.5. This validates our hypothesis that initial knowledge acquisition through auxiliary tasks can substantially enhance the model's ability to predict performance metrics accurately.

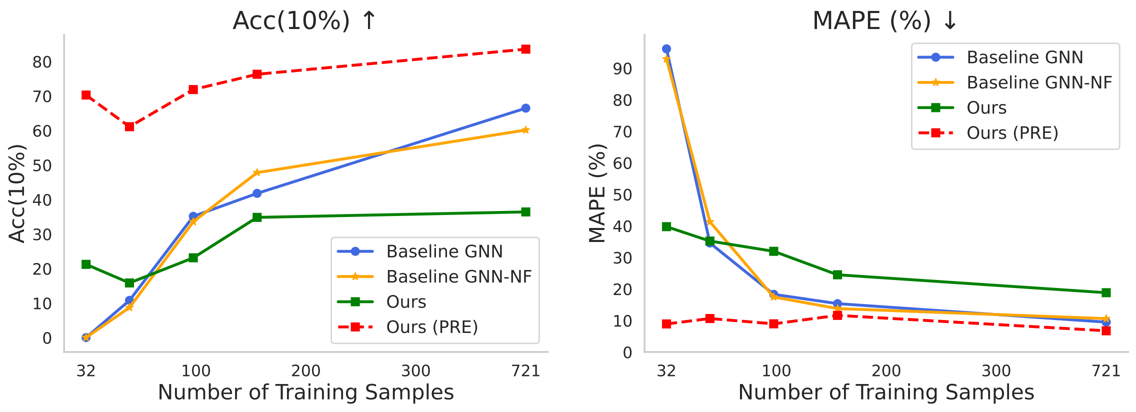
Interestingly, GNN pre-trained alone performs worse than randomly initialized GNN. We believe that randomly initialized GNN weights prevent overfitting to pre-existing biases, encouraging the LLM to learn more generalized and robust features during training.

7.5.4 Experiment: Adaptation

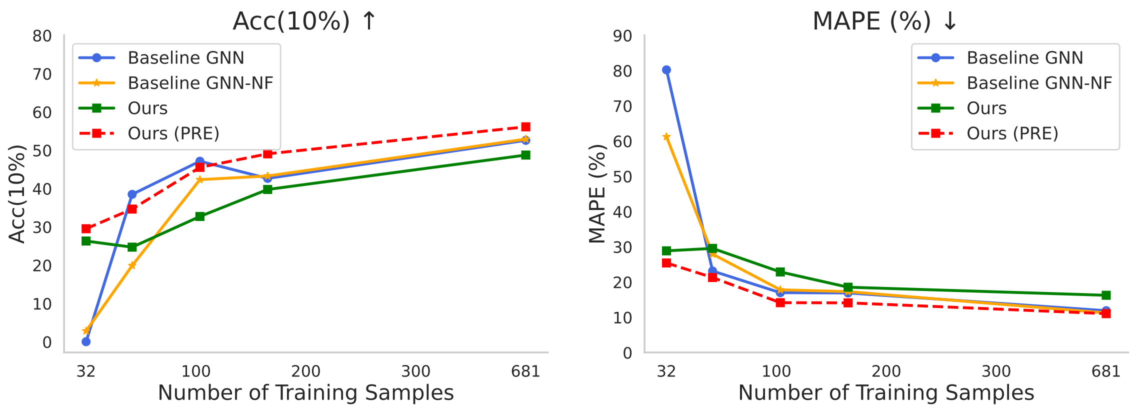
New Hardware Configurations This experiment assesses the real-world adaptability of our model to new hardware environments, particularly under conditions of limited training data. Our comparative analysis involved four models: a GNN baseline, enhanced GNN baseline with additional node features (GNN-NF) as detailed in Section 7.5.2 and two variants of our model, one with and one without both GNN pre-training and graph-text adaptation. Both variants of our model employ Llama3-8B as the base LLM, consistent with the settings described in Section 7.5.2. Each model was trained across eight distinct hardware platforms for ten epochs, after which the learned weights were transferred to additional, new hardware platforms for further training for three epochs.

The results, illustrated in Figure 7.3, demonstrate the superior adaptability and performance of our enhanced model on new hardware with sparse training samples. On the hi3519A-nnie12-int8 and atlas300-acl-fp16 platform, our model equipped with the structured pre-training achieves 70% and 29% Acc(10%) respectively, while GNN and GNN-NF achieves 0%, when training on just 32 samples. The results also highlight the importance of our structured pre-training strategy, increasing the accuracy of the LLM-GNN model by up to 50 percentage-point. These results underscore the critical roles of both LLMs and our structured-pre-training strategy in enhancing model adaptability, proving essential for the deployment of learned performance modeling in dynamic real-world applications.

New Deep Learning Architecture: This experiment evaluates our model's ability to quickly adapt to new DL architectures with sparse training samples. We compared four models: a baseline GNN, GNN-NF, and two variants of our model, utilizing either Llama3-8B or Mistral7B as the base LLM. Each model was trained on ten different DL model families and then evaluated on an new architecture, Vision Transformer (ViT), using only 32 training samples. As shown in Table 7.1, our approach significantly outperforms the GNN baselines, demonstrating rapid adaptation and superior performance when predicting the behavior of a new DL architecture.



(a) hi3519A-nnie12-int8



(b) atlas300-acl-fp16

Figure 7.3: Adaptability experiment demonstrating model transfer ability across different hardware platforms. We compared four models: a GNN baseline, enhanced GNN baseline with additional node features (GNN-NF) and two variants of our model-Llama3-8B (with and without the structured pre-training). Each model was trained on eight hardware configurations, followed by a transfer of learned weights to fine-tune on a new unseen hardware platform (hi3519A-nnie12-int8 or atlas300-acl-fp16) with a varying number of training samples. Our model with the structured pre-training outperformed both the GNN baselines and our model variant without the structured pre-training.

7.6 Discussion and Conclusion

This paper has investigated the integration of GNNs and LLMs to enhance the accuracy and adaptability of DL performance prediction. Our empirical evaluations have demonstrated that graph embeddings, derived from GNNs, are more effective inputs for LLMs than traditional text-based representations, leading to significant improvements in both accuracy and computational efficiency. Additionally, we have proposed a structured pre-training strategy that enables model adaptation to new hardware environments with minimal retraining, further enhancing the practicality and efficacy of our approach. We believe that our research offers a promising direction for advancing the field of DL performance prediction and its applications in various stages of the DL system stack.

7.7 Appendix

7.7.1 Environment setup

All experiments were conducted on hardware featuring AMD EPYC 7402 processors with two sockets (24 cores per socket), 512 GB DDR4-3200 RAM, and a 4 x NVIDIA A100 GPU with 40 GB HBM. Our software environment included Python libraries such as PyTorch 2.2.1, torch-geometric 2.5.3, transformers 4.41.0, and peft 0.10.1, running on CUDA version 12.1.

7.7.2 Evaluation Metrics

To assess the accuracy of our performance prediction models, we use the following two primary metrics:
Mean Absolute Percentage Error (MAPE): This metric quantifies the average of the absolute percentage differences between each predicted value and its corresponding actual value. It is defined mathematically as:

$$\mathrm{MAPE} = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{y_i - y_i'}{y_i}\right| \times 100\%$$

Here, y_i represents the actual value and y_i' represents the predicted value. MAPE is a non-negative number, where a smaller value indicates a more accurate model.

Accuracy within a delta threshold (Acc(δ)): This metric measures the percentage of predicted values that are within a specified percentage (δ) of the actual values. It is defined as:

$$\mathrm{Acc}(\delta) = \frac{1}{n}\sum_{i=1}^{n}\mathrm{pos}\left(\delta - \left|\frac{y_i - y_i'}{y_i}\right|\right) \times 100\%$$

where $\mathrm{pos}(z)$ is a function that returns 1 if $z \leq 0$ and 0 otherwise. A higher ACC(δ) value reflects better predictive performance of the model.

These metrics are widely used in previous works[63, 88, 89], providing a reliable means to compare the effectiveness of different models in terms of both overall error magnitude and the proportion of predictions within acceptable error thresholds.

7.7.3 Graph to Text Dataset

Graph to Text: Sample Prompt

Q: Summarise the graph

A: The graph contains 42 nodes and 40 edges. The batch size is 8. The graph has 12.1 GFLOPs, 0.33 billion parameters, and 0.4 billion MACs.
Layer Stats Summary:
Layer: convolution, Implemented: 20 times, FLOPs: 12.5 billion, MACs: 32.3 million, Parameters: 35.8 million
Layer: relu, Implemented: 17 times, FLOPs: 0, MACs: 0, Parameters: 3.3 million
Layer: max pooling, Implemented: 1 times, FLOPs: 0, MACs: 0, Parameters: 335.6 thousand Layer: addition, Implemented: 1 times, FLOPs: 0, MACs: 0, Parameters: 865

7.7.4 Limitations and Future Work

While our model effectively leverages static prompting to enhance performance prediction, exploring diverse prompting strategies could further optimize its adaptability and effectiveness across various scenarios.

Future research will explore several avenues to enhance the current model's robustness and applicability. We plan to extend our methodology to additional DL performance datasets such as TPU Graphs [94], allowing us to validate and refine our approach across a wider range of DL architectures and hardware configurations.

7.7.5 Additional Experiment: Graph Embedding Projection

This experiment investigates the effectiveness of different graph embedding projection techniques, essential for communicating the graph structural information from the GNN encoder to the LLM. It allow gradient flow from the LLM back to the GNN encoder, thereby enhancing the learning feedback loop.

Table 7.6: Performance comparison for Single vs. Multi Embedding -Projection methods

Type	MAPE ↓	ACC(10%) ↑	TTT ↓	Max Token Length
Single Proj.	**12.61**	**52.83**	**0.23**	512
Multi Proj.	13.66	48.50	2.32	2048

Setting: For this experiment, we utilized the same dataset and the same Llama3-8B and GIN encoder as DL representation experiment explained in Section 7.5.1. Each model was trained over 10 epochs 3 times to ensure stability and convergence of results.

Results: As result shown in Table 7.6, our proposed architecture, the single projection technique where the DL graph is projected as a single input embedding to LLM (g_1) demonstrated superior performance compared to the multi-projection method, which attempts to capture the graph structure as multiple embeddings from g_1 to $g_{H_{GNN}}$. This finding suggests that maintaining a focused, singular projection of graph features into the LLM not only preserves essential structural details but also enhances computational efficiency. This single embedding approach resulted in MAPE, higher ACC(10%), and reduced TTT. These results validate the importance of optimizing graph projection methods to enhance the interplay between GNN encodings and LLM capabilities for performance prediction tasks.

7.7.6 GNN Pre-training Hyper-parameters

Table 7.7: Hyper-parameters used for GNN pre-training.

Hyperparameter	**Value**
Number of Hidden Units	1024
Number of Features	44
Number of Layers	5
Learning Rate (lr)	0.0005
Weight Decay	0.00
Mask Rate	0.5
Drop Edge Rate	0.0
Maximum Epochs	500
Encoder Type	GIN
Decoder Type	GIN
Activation Function	PReLU
Loss Function	SCE
Use of Scheduler	No
Batch Size	128
Alpha_l	2
Replace Rate	0.1
Normalization Type	BatchNorm
Optimizer	Adam
Input Dropout	0.2
Attention Dropout	0.1

Table 7.8: Performance comparison of the GNN baseline against our models with different base LLMs (Llama3-8B and Mistral-7B) using a multi-platform performance prediction dataset. Both our models utilize GNN pre-training and graph-text adaptation. The results demonstrate that our approach outperforms the baseline, highlighting the effectiveness of the integrated method.

	MAPE ↓			**Acc (10%) ↑**		
Platforms	**GNN**	**Llama3-8B**	**Mistral-7B**	**GNN**	**Llama3-8B**	**Mistral-7B**
cpu-openppl-fp32	**10.48**	12.57	12.22	**58.94**	54.91	56.26
hi3559A-nnie11-int8	7.55	6.24	**5.38**	73.19	80.72	**88.15**
gpu-T4-trt7.1-fp32	**9.32**	10.00	9.69	**60.87**	56.52	58.74
gpu-T4-trt7.1-int8	18.10	15.17	**14.05**	27.90	**47.85**	46.78
gpu-P4-trt7.1-fp32	**9.75**	10.81	9.91	**60.97**	53.58	58.89
gpu-P4-trt7.1-int8	13.75	12.55	**12.05**	36.68	**48.93**	48.83
hi3519A-nnie12-int8	7.13	6.94	**5.96**	77.53	81.01	**85.02**
atlas300-acl-fp16	14.41	11.38	**9.47**	47.76	59.62	**68.05**
mul270-neuware-int8	**26.18**	26.88	28.31	21.61	30.77	**33.70**
Average	12.96	12.50	**11.89**	51.72	57.10	**60.49**

8 Conclusion

This thesis has presented a comprehensive approach to DL performance prediction, addressing key challenges in adapting models to diverse hardware environments, handling sparse labeled data, and ensuring scalability for large architectures such as LLMs. By integrating graph-based representations, semi-supervised learning, and multimodal learning, the research demonstrates that accurate, framework-agnostic, and efficient performance prediction is achievable without extensive profiling overhead.

A primary contribution is the development of a framework-agnostic performance prediction model that generalizes across TensorFlow, PyTorch, and ONNX. By abstracting DL architectures into a unified graph representation, the model captures computational dependencies and enables precise estimation of latency, memory consumption, and energy usage. The ability to provide accurate predictions across multiple frameworks eliminates the need for software-specific profiling, a crucial advancement given the increasing heterogeneity in DL development pipelines.

Another key innovation is the introduction of semi-supervised learning methods to mitigate the impact of sparse labeled data on performance prediction. Traditional supervised learning approaches rely on extensive profiling datasets, which are difficult to obtain for new hardware and models. Using graph auto-encoders for unsupervised representation learning, the proposed method significantly reduces dependence on labeled data while preserving predictive accuracy. This approach is particularly beneficial in emerging AI hardware environments, where profiling data is often scarce.

Beyond standard model architectures, this thesis shows that LLMs require specialized performance prediction strategies due to their hierarchical and repetitive structures. While graph-based approaches are effective for conventional workloads, LLMs benefit from tree-based performance prediction models, which improve both accuracy and efficiency in resource estimation. This tailored approach significantly improves prediction speed without compromising reliability, making it feasible to assess LLM performance without costly hardware execution.

One of the most forward-looking contributions of this thesis is the integration of multimodal learning to enhance adaptability to new and unseen hardware environments. By combining GNNs with LLMs, the research demonstrates that performance prediction models can generalize across hardware architectures with minimal retraining. By incorporating DL graph modality and textual descriptions, this approach allows predictive models to quickly adapt to evolving computing landscapes.

These contributions establish a foundation for accurate, scalable, and hardware-aware performance prediction in DL. By structuring model representations, leveraging both labeled and unlabeled data, and integrating adaptable learning frameworks, this thesis provides a pathway toward efficient and sustainable AI deployments. The methodologies significantly reduce the reliance on manual profiling and empirical testing, accelerating model deployment while optimizing computational resource allocation. These findings are not only relevant for immediate practical applications but also set the stage for future advancements as DL architectures and hardware platforms continue to evolve.

9 Limitations & Future Works

While this thesis advances DL performance prediction, certain limitations remain. A key constraint is that the proposed models are evaluated primarily for single-GPU performance prediction. As DL workloads scale, particularly with LLMs such as LLaMA 3.1 405B - FP8, single-GPU execution is no longer feasible. These models require multi-node, multi-GPU configurations, where factors such as interconnect bandwidth, communication overhead, and synchronization costs significantly impact performance. The current approach does not account for these multi-device interactions, limiting its applicability to large-scale distributed inference and training.

To address this limitation, future work should focus on extending performance prediction models to multi-GPU and multi-node settings. This requires benchmarking distributed DL workloads using libraries such as vLLM[1], which facilitate efficient serving of large-scale models across multiple devices. Incorporating multi-node execution profiles, GPU interconnect characteristics, and memory access patterns will enable the predictive framework to model distributed execution overhead and inter-device communication constraints more effectively.

Additionally, integrating hardware-aware features such as PCIe and NVLink bandwidth, tensor parallelism strategies, and memory partitioning efficiency will improve the adaptability of the performance prediction model. These enhancements will allow more accurate resource allocation strategies, optimizing system throughput and energy efficiency in large-scale AI workloads.

Another critical direction is improving the integration of GNN and LLM-based prediction models. Future research should explore various prompting techniques such as few-shot learning, chain-of-thought reasoning, and meta-prompting to refine model accuracy and generalization across diverse hardware settings.

Another promising direction is the development of intelligent performance model agents that function as a distributed, multi-agent system for automated performance optimization. These agents could analyze DL models at different levels to identify bottlenecks, inefficient operations, and resource constraints. For instance, one agent could specialize in profiling computational inefficiencies, while another could focus on suggesting architectural improvements or optimizing hyperparameters. After adjustments, an

[1]https://docs.vllm.ai/en/v0.5.1/serving/distributed_serving.html

agent could re-evaluate the optimized model across various hardware configurations, ensuring efficient deployment.

This agent-based approach would automate performance optimization and enable continuous learning, allowing models to adapt dynamically to evolving hardware environments with minimal human intervention. A real-time feedback mechanism could further refine predictions, as agents learn from interactions with both the model and hardware, progressively enhancing their recommendations.

These advancements will significantly improve adaptability, scalability, and efficiency, making performance prediction models more robust for next-generation DL deployments.

10 List of Papers

Papers included in this dissertation:

- **Karthick Panner Selvam**, and Mats Brorsson. "Performance Analysis and Benchmarking of a Temperature Downscaling Deep Learning Model," in 2023 31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP), Naples, Italy, 2023, pp. 1-8, doi: 10.1109/PDP59025.2023.00010.

- **Karthick Panner Selvam**, and Mats Brorsson. " DIPPM: A Deep Learning Inference Performance Predictive Model Using Graph Neural Networks," Euro-Par 2023: Parallel Processing. Euro-Par 2023. Lecture Notes in Computer Science, vol 14100. Springer, Cham. DOI: 10.1007/978-3-031-39698-4_1.

- **Karthick Panner Selvam**, and Mats Brorsson. “Can Semi-Supervised Learning Improve Prediction of Deep Learning Model Resource Consumption?” International Journal of Advanced Computer Science and Applications(IJACSA), 15(6), 2024. DOI: 10.14569/IJACSA.2024.0150610. *This research was also presented at the Machine Learning for Systems Workshop co-located with the 37th NeurIPS Conference, 2023, in New Orleans, LA, USA*

- **Karthick Panner Selvam**, and Mats Brorsson. "Can Tree-Based Model Improve Performance Prediction for LLMs?." ARC-LG workshop at 51st International Symposium on Computer Architecture (2024). hdl.handle.net/10993/62222.

- **Karthick Panner Selvam**, Phitchaya Mangpo Phothilimthana, Sami Abu-El-Haija, Bryan Perozzi and Mats Brorsson. "Can LLMs Enhance Performance Prediction for Deep Learning Models?" in the 2024 WANT Workshop at the International Conference on Machine Learning (ICML). https://openreview.net/pdf?id=bpS4vaOg7q. *This research was also presented at the 2024 KDD Ph.D. Consortium.*

Papers not included in this dissertation:

- **Karthick Panner Selvam**, and Mats Brorsson. "Performance Modeling of Weather Forecast Machine Learning for Efficient HPC," 2022 IEEE 42nd International Conference on Distributed Computing Systems (ICDCS), Bologna, Italy, 2022, pp. 1268-1269, doi: 10.1109/ICDCS54860.2022.00127.

- Raul Ramos-Pollan, Freddie Kalaitzis, **Karthick Panner Selvam** "Uncertainty and Generalizability in Foundation Models for Earth Observation," 2024. arXiv: 2409.08744 [cs.CV]. `https://arxiv.org/abs/2409.08744`. This work accepted at NeurIPS 2024 Foundation Models for Science Workshop.

- **Karthick Panner Selvam**, Raul Ramos-Pollan, Freddie Kalaitzis, "Rapid Adaptation of Earth Observation Foundation Models for Segmentation," 2024. arXiv: 2409.09907 [cs.CV]. `https://arxiv.org/abs/2409.09907`. This work accepted at ESA-NASA International Workshop on AI Foundation Model for Earth Observation 2025.

- Andrii Krutsylo, **Karthick Panner Selvam**, Freddie Kalaitzis, and Raul Ramos-Pollan. "Remote Sensing Segmentation with Foundation Models (on a Budget)," 2024 American Geophysical Union (AGU24).

- Mats Brorsson and **Karthick Panner Selvam** "Accelerate and Scale Your AI Deployment Through Automated Infrastructure Selection and Management" 2025 NVIDIA GTC 2025 (Poster)

Bibliography

[1] A. Steiner, A. S. Pinto, M. Tschannen, D. Keysers, X. Wang, Y. Bitton, A. Gritsenko, M. Minderer, A. Sherbondy, S. Long, S. Qin, R. Ingle, E. Bugliarello, S. Kazemzadeh, T. Mesnard, I. Alabdulmohsin, L. Beyer, and X. Zhai, “Paligemma 2: A family of versatile vlms for transfer,” 2024. [Online]. Available: https://arxiv.org/abs/2412.03555

[2] A. Q. Jiang, A. Sablayrolles, A. Roux, A. Mensch, B. Savary, C. Bamford, D. S. Chaplot, D. de las Casas, E. B. Hanna, F. Bressand, G. Lengyel, G. Bour, G. Lample, L. R. Lavaud, L. Saulnier, M.-A. Lachaux, P. Stock, S. Subramanian, S. Yang, S. Antoniak, T. L. Scao, T. Gervet, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, “Mixtral of experts,” 2024. [Online]. Available: https://arxiv.org/abs/2401.04088

[3] A. Pagnoni, R. Pasunuru, P. Rodriguez, J. Nguyen, B. Muller, M. Li, C. Zhou, L. Yu, J. Weston, L. Zettlemoyer, G. Ghosh, M. Lewis, A. Holtzman, and S. Iyer, “Byte latent transformer: Patches scale better than tokens,” 2024. [Online]. Available: https://arxiv.org/abs/2412.09871

[4] M. Z. Irshad, M. Comi, Y.-C. Lin, N. Heppert, A. Valada, R. Ambrus, Z. Kira, and J. Tremblay, “Neural fields in robotics: A survey,” 2024. [Online]. Available: https://arxiv.org/abs/2410.20220

[5] C. Wang, M. S. Pritchard, N. Brenowitz, Y. Cohen, B. Bonev, T. Kurth, D. Durran, and J. Pathak, “Coupled ocean-atmosphere dynamics in a machine learning earth system model,” 2024. [Online]. Available: https://arxiv.org/abs/2406.08632

[6] M. Alexe, E. Boucher, P. Lean, E. Pinnington, P. Laloyaux, A. McNally, S. Lang, M. Chantry, C. Burrows, M. Chrust, F. Pinault, E. Villeneuve, N. Bormann, and S. Healy, “Graphdop: Towards skilful data-driven medium-range weather forecasts learnt and initialised directly from observations,” 2024. [Online]. Available: https://arxiv.org/abs/2412.15687

[7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” in *Proceedings of the 31st International*

Conference on Neural Information Processing Systems, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 6000–6010.

[8] M. M. Ahsan, S. Raman, Y. Liu, and Z. Siddique, "A comprehensive survey on diffusion models and their applications," 2024. [Online]. Available: https://arxiv.org/abs/2408.10207

[9] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, "Improving language understanding by generative pre-training," 2018.

[10] N. Bouhali, H. Ouarnoughi, S. Niar, and A. A. El Cadi, "Execution time modeling for cnn inference on embedded gpus," in *Proceedings of the 2021 Drone Systems Engineering and Rapid Simulation and Performance Evaluation: Methods and Tools Proceedings*, ser. DroneSE and RAPIDO '21. New York, NY, USA: Association for Computing Machinery, 2021, p. 59–65.

[11] Z. Lu, S. Rallapalli, K. Chan, S. Pu, and T. L. Porta, "Augur: Modeling the resource requirements of convnets on mobile devices," *IEEE Transactions on Mobile Computing*, vol. 20, no. 2, pp. 352–365, 2021.

[12] L. Bai, W. Ji, Q. Li, X. Yao, W. Xin, and W. Zhu, "Dnnabacus: Toward accurate computational cost prediction for deep neural networks," 2022.

[13] D. Justus, J. Brennan, S. Bonner, and A. McGough, "Predicting the computational cost of deep learning models," in *2018 IEEE International Conference on Big Data (Big Data)*. Los Alamitos, CA, USA: IEEE Computer Society, dec 2018, pp. 3873–3882.

[14] H. Qi, E. R. Sparks, and A. Talwalkar, "Paleo: A performance model for deep neural networks," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[15] M. Sponner, B. Waschneck, and A. Kumar, "Ai-driven performance modeling for ai inference workloads," *Electronics*, vol. 11, no. 15, 2022.

[16] C.-C. Wang, Y.-C. Liao, M.-C. Kao, W.-Y. Liang, and S.-H. Hung, "Toward accurate platform-aware performance modeling for deep neural networks," *SIGAPP Appl. Comput. Rev.*, vol. 21, no. 1, p. 50–61, jul 2021.

[17] C. Yang, Z. Li, C. Ruan, G. Xu, C. Li, R. Chen, and F. Yan, "PerfEstimator: A Generic and Extensible Performance Estimator for Data Parallel DNN Training," in *2021 IEEE/ACM International Workshop on Cloud Intelligence (CloudIntelligence)*, May 2021, pp. 13–18.

[18] L. L. Zhang, S. Han, J. Wei, N. Zheng, T. Cao, Y. Yang, and Y. Liu, “Nn-meter: Towards accurate latency prediction of deep-learning model inference on diverse edge devices,” in *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys ’21. New York, NY, USA: Association for Computing Machinery, 2021, p. 81–93.

[19] S. Kaufman, P. Phothilimthana, Y. Zhou, C. Mendis, S. Roy, A. Sabne, and M. Burrows, “A learned performance model for tensor processing units,” in *Proceedings of Machine Learning and Systems*, A. Smola, A. Dimakis, and I. Stoica, Eds., vol. 3, 2021, pp. 387–400.

[20] L. Dudziak, T. Chau, M. S. Abdelfattah, R. Lee, H. Kim, and N. D. Lane, “Brp-nas: Prediction-based nas using gcns,” in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS’20. Red Hook, NY, USA: Curran Associates Inc., 2020.

[21] L. Liu, M. Shen, R. Gong, F. Yu, and H. Yang, “Nnlqp: A multi-platform neural network latency query and prediction system with an evolving database,” in *Proceedings of the 51st International Conference on Parallel Processing*, ser. ICPP ’22. New York, NY, USA: Association for Computing Machinery, 2023.

[22] T. Kipf and M. Welling, “Variational graph auto-encoders,” *NIPS Workshop on Bayesian Deep Learning*, 2016.

[23] Z. Hou, X. Liu, Y. Cen, Y. Dong, H. Yang, C. Wang, and J. Tang, “Graphmae: Self-supervised masked graph autoencoders,” in *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2022, pp. 594–604.

[24] K. P. Selvam and M. Brorsson, “ Performance Analysis and Benchmarking of a Temperature Downscaling Deep Learning Model ,” in *2023 31st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. Los Alamitos, CA, USA: IEEE Computer Society, Mar. 2023, pp. 1–8. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/PDP59025.2023.00010

[25] K. Panner Selvam and M. Brorsson, “Dippm: A deep learning inference performance predictive model using graph neural networks,” in *Euro-Par 2023: Parallel Processing*. Springer Nature Switzerland, 2023, pp. 3–16.

[26] K. P. Selvam and M. Brorsson, “Can semi-supervised learning improve prediction of deep learning model resource consumption?” *International Journal of Advanced Computer Science and Applications*, vol. 15, no. 6, 2024. [Online]. Available: http://dx.doi.org/10.14569/IJACSA.2024.0150610

[27] K. Panner Selvam and M. Brorsson, "Can tree-based model improve performance prediction for llms?" in *ARC-LG Workshop at 51st International Symposium on Computer Architecture (ISCA)*, 2024.

[28] K. P. Selvam, P. M. Phothilimthana, S. Abu-El-Haija, B. Perozzi, and M. Brorsson, "Can llms enhance performance prediction for deep learning models?" in *Proceedings of the 2024 WANT Workshop at the International Conference on Machine Learning (ICML)*, 2024. [Online]. Available: https://openreview.net/pdf?id=bpS4vaOg7q

[29] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, "Lora: Low-rank adaptation of large language models," 2021.

[30] A. Bulat and G. Tzimiropoulos, "Lasp: Text-to-text optimization for language-aware soft prompting of vision language models," 2023.

[31] Y. Wang, "Single image super-resolution with u-net generative adversarial networks," in *2021 IEEE 4th Advanced Information Management, Communicates, Electronic and Automation Control Conference (IMCEC)*, vol. 4, 2021, pp. 1835–1840.

[32] J. Leinonen, D. Nerini, and A. Berne, "Stochastic super-resolution for downscaling time-evolving atmospheric fields with a generative adversarial network," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 59, no. 9, pp. 7211–7223, 2021.

[33] O. Ronneberger, P. Fischer, and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," in *Medical Image Computing and Computer-Assisted Intervention – MICCAI 2015*, N. Navab, J. Hornegger, W. M. Wells, and A. F. Frangi, Eds. Cham: Springer International Publishing, 2015, pp. 234–241.

[34] G. Bing, L. Michael, D. Peter, C. Matthew, N. Thomas, A. Markus, and B.-N. Tal, "Report on a survey of MAELSTROM applications and ML tools and architectures. Deliverable 2.1." MEAELSTROM EuroHPC project., Tech. Rep., 2021. [Online]. Available: https://www.maelstrom-eurohpc.eu/deliverables

[35] M. Brorsson, "Roadmap analysis of technologies relevant for ML solutions in W&C. Deliverable 3.2." MEAELSTROM EuroHPC project., Tech. Rep., 2021. [Online]. Available: https://www.maelstrom-eurohpc.eu/deliverables

[36] Y. Sha, D. J. G. Ii, G. West, and R. Stull, "Deep-Learning-Based Gridded Downscaling of Surface Meteorological Variables in Complex Terrain. Part I: Daily Maximum and Minimum 2-m Temperature," *Journal of Applied Meteorology and Climatology*, vol. 59, no. 12, pp. 2057–2073, Dec. 2020. [Online]. Available: https://journals.ametsoc.org/view/journals/apme/59/12/jamc-d-20-0057.1.xml

[37] T. Ben-Nun, M. Besta, S. Huber, A. N. Ziogas, D. Peter, and T. Hoefler, "A modular benchmarking infrastructure for high-performance and reproducible deep learning," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2019, pp. 66–77.

[38] C. Yang, Y. Wang, T. Kurth, S. Farrell, and S. Williams, "Hierarchical roofline performance analysis for deep learning applications," in *Intelligent Computing*, K. Arai, Ed. Cham: Springer International Publishing, 2021, pp. 473–491.

[39] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P. Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, "Mlperf inference benchmark," in *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, 2020, pp. 446–459.

[40] C. A. Coleman, D. Narayanan, D. Kang, T. Zhao, J. Zhang, L. Nardi, P. D. Bailis, K. Olukotun, C. Ré, and M. A. Zaharia, "Dawnbench : An end-to-end deep learning benchmark and competition," 2017.

[41] S. Narang and G. Diamos, "Deepbench: Benchmarking deep learning operations on different hardware (2017)," 2017. [Online]. Available: https://github.com/Baidu-Research/DeepBench

[42] H. Zhu, M. Akrout, B. Zheng, A. Pelegris, A. Jayarajan, A. Phanishayee, B. Schroeder, and G. Pekhimenko, "Benchmarking and analyzing deep neural network training," in *2018 IEEE International Symposium on Workload Characterization (IISWC)*, 2018, pp. 88–100.

[43] Y. Wang, C. Yang, S. Farrell, Y. Zhang, T. Kurth, and S. Williams, "Time-based roofline for deep learning performance analysis," in *2020 IEEE/ACM Fourth Workshop on Deep Learning on Supercomputers (DLS)*, 2020, pp. 10–19.

[44] N. Ding and S. Williams, "An instruction roofline model for gpus," in *2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2019, pp. 7–18.

[45] D. Ernst, G. Hager, J. Thies, and G. Wellein, "Performance engineering for real and complex tall & skinny matrix multiplication kernels on gpus," *The International Journal of High Performance Computing Applications*, vol. 35, no. 1, pp. 5–19, 2021. [Online]. Available: https://doi.org/10.1177/1094342020965661

[46] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *International Conference on Learning Representations*, 12 2014.

[47] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. M. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language models are few-shot learners," in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS'20. Red Hook, NY, USA: Curran Associates Inc., 2020.

[48] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," in *2021 IEEE/CVF International Conference on Computer Vision (ICCV).* Los Alamitos, CA, USA: IEEE Computer Society, oct 2021, pp. 9992–10 002.

[49] T. Elsken, J. H. Metzen, and F. Hutter, "Neural architecture search: A survey," *J. Mach. Learn. Res.*, vol. 20, no. 1, p. 1997–2017, mar 2021.

[50] Y. Gao, Y. Liu, H. Zhang, Z. Li, Y. Zhu, H. Lin, and M. Yang, "Estimating GPU memory consumption of deep learning models," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, Nov. 2020, pp. 1342–1352.

[51] G. X. Yu, Y. Gao, P. Golikov, and G. Pekhimenko, "Habitat: A Runtime-Based Computational Performance Predictor for Deep Neural Network Training," in *Proceedings of the 2021 USENIX Annual Technical Conference (USENIX ATC'21)*, 2021.

[52] Y. Gao, X. Gu, H. Zhang, H. Lin, and M. Yang, "Runtime performance prediction for deep learning models with graph neural network," in *ICSE '23.* IEEE/ACM, May 2023, the 45th International Conference on Software Engineering, Software Engineering in Practice (SEIP) Track.

[53] B. Li, T. Patel, S. Samsi, V. Gadepally, and D. Tiwari, "Miso: Exploiting multi-instance gpu capability on multi-tenant gpu clusters," in *Proceedings of the 13th Symposium on Cloud Computing*, ser. SoCC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 173–189.

[54] J. Roesch, S. Lyubomirsky, L. Weber, J. Pollock, M. Kirisame, T. Chen, and Z. Tatlock, "Relay: A new ir for machine learning frameworks," in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*,

ser. MAPL 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 58–68.

[55] W. L. Hamilton, R. Ying, and J. Leskovec, “Inductive representation learning on large graphs,” in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS’17. Red Hook, NY, USA: Curran Associates Inc., 2017, p. 1025–1035.

[56] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, “Graph Attention Networks,” *International Conference on Learning Representations*, 2018, accepted as poster.

[57] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” in *International Conference on Learning Representations (ICLR)*, 2017.

[58] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, “How powerful are graph neural networks?” in *International Conference on Learning Representations*, 2019.

[59] L. N. Smith, “Cyclical learning rates for training neural networks,” in *2017 IEEE Winter Conference on Applications of Computer Vision (WACV)*, 2017, pp. 464–472.

[60] Y. Gao, Y. Liu, H. Zhang, Z. Li, Y. Zhu, H. Lin, and M. Yang, “Estimating gpu memory consumption of deep learning models,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, p. 1342–1352.

[61] L. Bai, W. Ji, Q. Li, X. Yao, W. Xin, and W. Zhu, “Dnnabacus: Toward accurate computational cost prediction for deep neural networks,” 2022.

[62] E. Gianniti, L. Zhang, and D. Ardagna, “Performance prediction of gpu-based deep learning applications,” in *2018 30th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD)*, 2018, pp. 167–170.

[63] L. Liu, M. Shen, R. Gong, F. Yu, and H. Yang, “Nnlqp: A multi-platform neural network latency query and prediction system with an evolving database,” in *51 International Conference on Parallel Processing - ICPP*, ser. ICPP ’22. Association for Computing Machinery, 2022.

[64] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, “Graph attention networks,” in *International Conference on Learning Representations*, 2018.

[65] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” in *International Conference on Learning Representations*, 2017.

[66] H. Qi, E. R. Sparks, and A. Talwalkar, "Paleo: A performance model for deep neural networks," in *International Conference on Learning Representations*, 2017.

[67] D. Velasco-Montero, J. Fernández-Berni, R. Carmona-Galán, and Rodríguez-Vázquez, "Previous: A methodology for prediction of visual inference performance on iot devices," *IEEE Internet of Things Journal*, vol. 7, no. 10, pp. 9227–9240, 2020.

[68] E. Cai, D.-C. Juan, D. Stamoulis, and D. Marculescu, "*NeuralPower*: Predict and deploy energy-efficient convolutional neural networks," in *Proceedings of the Ninth Asian Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M.-L. Zhang and Y.-K. Noh, Eds., vol. 77. Yonsei University, Seoul, Republic of Korea: PMLR, 15–17 Nov 2017, pp. 622–637.

[69] M. Fey and J. E. Lenssen, "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[70] R. Wightman, "Pytorch image models," https://github.com/rwightman/pytorch-image-models, 2019.

[71] Z. Liu, H. Mao, C. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, "A convnet for the 2020s," in *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2022, pp. 11 966–11 976. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CVPR52688.2022.01167

[72] G. Huang, Z. Liu, L. V. D. Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, jul 2017, pp. 2261–2269. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CVPR.2017.243

[73] M. Tan and Q. Le, "EfficientNet: Rethinking model scaling for convolutional neural networks," in *Proceedings of the 36th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. PMLR, 09–15 Jun 2019, pp. 6105–6114. [Online]. Available: https://proceedings.mlr.press/v97/tan19a.html

[74] M. Tan, B. Chen, R. Pang, V. Vasudevan, M. Sandler, A. Howard, and Q. V. Le, "Mnasnet: Platform-aware neural architecture search for mobile," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2019, pp. 2815–2823. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CVPR.2019.00293

[75] A. Howard, M. Sandler, B. Chen, W. Wang, L. Chen, M. Tan, G. Chu, V. Vasudevan, Y. Zhu, R. Pang, H. Adam, and Q. Le, “Searching for mobilenetv3,” in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. Los Alamitos, CA, USA: IEEE Computer Society, nov 2019, pp. 1314–1324. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICCV.2019.00140

[76] W. Yu, C. Si, P. Zhou, M. Luo, Y. Zhou, J. Feng, S. Yan, and X. Wang, “Metaformer baselines for vision,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2023.

[77] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[78] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, “ImageNet Large Scale Visual Recognition Challenge,” *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[79] Z. Chen, L. Xie, J. Niu, X. Liu, L. Wei, and Q. Tian, “Visformer: The vision-friendly transformer,” in *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*. Los Alamitos, CA, USA: IEEE Computer Society, oct 2021, pp. 569–578. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICCV48922.2021.00063

[80] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, J. Uszkoreit, and N. Houlsby, “An image is worth 16x16 words: Transformers for image recognition at scale,” *ICLR*, 2021.

[81] L. van der Maaten and G. Hinton, “Visualizing data using t-sne,” *Journal of Machine Learning Research*, vol. 9, no. 86, pp. 2579–2605, 2008. [Online]. Available: http://jmlr.org/papers/v9/vandermaaten08a.html

[82] T. Chen and C. Guestrin, “Xgboost: A scalable tree boosting system,” in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD ’16. New York, NY, USA: Association for Computing Machinery, 2016, p. 785–794. [Online]. Available: https://doi.org/10.1145/2939672.2939785

[83] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “Bert: Pre-training of deep bidirectional transformers for language understanding,” 2019.

[84] A. Conneau, K. Khandelwal, N. Goyal, V. Chaudhary, G. Wenzek, F. Guzmán, E. Grave, M. Ott, L. Zettlemoyer, and V. Stoyanov, “Unsupervised cross-lingual representation learning at scale,” 2020.

[85] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, “Roberta: A robustly optimized bert pretraining approach,” 2019.

[86] P. He, J. Gao, and W. Chen, “Debertav3: Improving deberta using electra-style pre-training with gradient-disentangled embedding sharing,” 2023.

[87] V. Sanh, L. Debut, J. Chaumond, and T. Wolf, “Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter,” 2020.

[88] Y. Yi, H. Zhang, R. Xiao, N. Wang, and X. Wang, “Nar-former v2: Rethinking transformer for universal neural network representation learning,” 2023.

[89] K. Panner Selvam and M. Brorsson, “Can semi-supervised learning improve prediction of deep learning model resource consumption?” in *Machine Learning for Systems Workshop at 37th NeurIPS Conference, 2023, New Orleans, LA, USA*. [Online]. Available: https://openreview.net/forum?id=C4nDgK47OJ

[90] A. Faiz, S. Kaneda, R. Wang, R. Osi, P. Sharma, F. Chen, and L. Jiang, “Llmcarbon: Modeling the end-to-end carbon footprint of large language models,” 2024.

[91] H. Hanpeng, S. Junwei, Z. Juntao, P. Yanghua, Z. Yibo, L. Haibin, and W. Chuan, “CDMPP: A Device-Model Agnostic Framework for Latency Prediction of Tensor Programs,” in *Proceedings of the Nineteenth EuroSys Conference*, 2024.

[92] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample, “Llama: Open and efficient foundation language models,” 2023.

[93] W. L. Hamilton, R. Ying, and J. Leskovec, “Inductive representation learning on large graphs,” 2018.

[94] P. M. Phothilimthana, S. Abu-El-Haija, K. Cao, B. Fatemi, M. Burrows, C. Mendis, and B. Perozzi, “Tpugraphs: A performance prediction dataset on large tensor computational graphs,” 2023.

[95] G. Team, T. Mesnard, C. Hardin, R. Dadashi, S. Bhupatiraju, S. Pathak, L. Sifre, M. Rivière, M. S. Kale, J. Love, P. Tafti, L. Hussenot, P. G. Sessa, A. Chowdhery, A. Roberts, A. Barua, A. Botev, A. Castro-Ros, A. Slone, A. Héliou, A. Tacchetti, A. Bulanova, A. Paterson, B. Tsai, B. Shahriari, C. L. Lan, C. A. Choquette-Choo, C. Crepy, D. Cer, D. Ippolito, D. Reid, E. Buchatskaya, E. Ni, E. Noland, G. Yan, G. Tucker, G.-C. Muraru, G. Rozhdestvenskiy, H. Michalewski, I. Tenney, I. Grishchenko, J. Austin, J. Keeling, J. Labanowski, J.-B. Lespiau, J. Stanway, J. Brennan, J. Chen, J. Ferret, J. Chiu, J. Mao-Jones, K. Lee, K. Yu, K. Millican, L. L.

Sjoesund, L. Lee, L. Dixon, M. Reid, M. Mikuła, M. Wirth, M. Sharman, N. Chinaev, N. Thain, O. Bachem, O. Chang, O. Wahltinez, P. Bailey, P. Michel, P. Yotov, R. Chaabouni, R. Comanescu, R. Jana, R. Anil, R. McIlroy, R. Liu, R. Mullins, S. L. Smith, S. Borgeaud, S. Girgin, S. Douglas, S. Pandya, S. Shakeri, S. De, T. Klimenko, T. Hennigan, V. Feinberg, W. Stokowiec, Y. hui Chen, Z. Ahmed, Z. Gong, T. Warkentin, L. Peran, M. Giang, C. Farabet, O. Vinyals, J. Dean, K. Kavukcuoglu, D. Hassabis, Z. Ghahramani, D. Eck, J. Barral, F. Pereira, E. Collins, A. Joulin, N. Fiedel, E. Senter, A. Andreev, and K. Kenealy, "Gemma: Open models based on gemini research and technology," 2024.

[96] K. Singhal, S. Azizi, T. Tu, S. S. Mahdavi, J. Wei, H. W. Chung, N. Scales, A. Tanwani, H. Cole-Lewis, S. Pfohl, P. Payne, M. Seneviratne, P. Gamble, C. Kelly, A. Babiker, N. Schärli, A. Chowdhery, P. Mansfield, D. Demner-Fushman, B. Agüera Y Arcas, D. Webster, G. S. Corrado, Y. Matias, K. Chou, J. Gottweis, N. Tomasev, Y. Liu, A. Rajkomar, J. Barral, C. Semturs, A. Karthikesalingam, and V. Natarajan, "Large language models encode clinical knowledge," *Nature*, vol. 620, no. 7972, pp. 172–180, Aug. 2023. [Online]. Available: https://www.nature.com/articles/s41586-023-06291-2

[97] X. Wayne, Z. Kun, and L. Junyi, "A survey of large language models," 2023.

[98] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu, A. H. Awadallah, R. W. White, D. Burger, and C. Wang, "Autogen: Enabling next-gen llm applications via multi-agent conversation," 2023.

[99] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu, J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder: may the source be with you!" 2023.

[100] C. Cummins, V. Seeker, D. Grubisic, M. Elhoushi, Y. Liang, B. Roziere, J. Gehring, F. Gloeckle, K. Hazelwood, G. Synnaeve, and H. Leather, "Large language models for compiler optimization," 2023.

[101] G. Jawahar, M. Abdul-Mageed, L. V. S. Lakshmanan, and D. Ding, "Llm performance predictors are good initializers for architecture search," 2023.

[102] B. Perozzi, B. Fatemi, D. Zelle, A. Tsitsulin, M. Kazemi, R. Al-Rfou, and J. Halcrow, “Let your graph do the talking: Encoding structured data for llms,” 2024.

[103] Z. Liu, X. He, Y. Tian, and N. V. Chawla, “Can we soft prompt llms for graph learning tasks?” in *Companion Proceedings of the ACM on Web Conference 2024*, ser. WWW ’24. ACM, May 2024. [Online]. Available: http://dx.doi.org/10.1145/3589335.3651476

[104] H. Lee, S. Lee, S. Chong, and S. J. Hwang, “Help: Hardware-adaptive efficient latency prediction for nas via meta-learning,” in *Advances in Neural Information Processing Systems (NeurIPS)*, 2021.

[105] Y. Zhou, S. Roy, A. Abdolrashidi, D. Wong, P. Ma, Q. Xu, H. Liu, M. P. Phothilimtha, S. Wang, A. Goldie, A. Mirhoseini, and J. Laudon, “Transferable graph optimizers for ml compilers,” in *Proceedings of the 34th International Conference on Neural Information Processing Systems*, ser. NIPS ’20. Red Hook, NY, USA: Curran Associates Inc., 2020.

[106] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, “How powerful are graph neural networks?” 2019.

[107] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” 2017. [Online]. Available: https://arxiv.org/pdf/1706.03762.pdf

[108] L. Xu, H. Xie, S.-Z. J. Qin, X. Tao, and F. L. Wang, “Parameter-efficient fine-tuning methods for pretrained language models: A critical review and assessment,” 2023.

[109] A. Q. Jiang, A. Sablayrolles, A. Mensch, C. Bamford, D. S. Chaplot, D. de las Casas, F. Bressand, G. Lengyel, G. Lample, L. Saulnier, L. R. Lavaud, M.-A. Lachaux, P. Stock, T. L. Scao, T. Lavril, T. Wang, T. Lacroix, and W. E. Sayed, “Mistral 7b,” 2023. [Online]. Available: https://arxiv.org/abs/2310.06825

[110] N. Reimers and I. Gurevych, “Sentence-bert: Sentence embeddings using siamese bert-networks,” in *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing*. Association for Computational Linguistics, 11 2019. [Online]. Available: https://arxiv.org/abs/1908.10084