ABSTRACT

KOH, DONGHYEON. Critical Path Delay Prediction for Evolving Netlists. (Under the direction of William Davis).

Power, performance, and area (PPA) are key metrics that determine the quality of integrated circuit (IC) designs. Accurate PPA estimation traditionally requires completing the full electronic design automation (EDA) flow, including placement and routing, which can be time-consuming and costly. Predicting PPA accurately at early design stages could enable faster design cycles and reduce wasted computational effort on suboptimal designs.

Recent approaches using deep neural networks (DNNs) with transfer learning have demonstrated potential for early PPA prediction, but suffer from limited accuracy—particularly for critical path delay—and often fail to generalize across diverse design spaces. Graph neural networks (GNNs) have emerged as a promising alternative, better capturing structural information inherent in netlist graphs. However, prior GNN-based models have been evaluated only on small-scale, pre-synthesis designs, without considering full EDA optimization or transfer learning scenarios.

This dissertation proposes a novel approach to predict post-route PPA using pre-floorplan data through transfer learning with GNNs. We evaluate the effectiveness of GNN models in generalizing across a wide variety of IC designs and demonstrate that transfer learning can achieve high accuracy in predicting critical path delay. In addition, we introduce the use of the Wasserstein distance as a measure of statistical dissimilarity between training and test datasets, enabling better understanding of dataset diversity and its impact on transfer learning performance.

Experimental results show that the proposed GNN-based framework significantly outperforms traditional DNN models in critical path delay prediction on large-scale designs.

Furthermore, Wasserstein distance is shown to be an effective predictor of model transferability, guiding the selection of training datasets to improve model generalization. These findings advance the state of early-stage PPA prediction and open up new possibilities for efficient, learning-driven IC design methodologies.

Critical Path Delay Prediction for Evolving Netlists

by
Donghyeon Koh

A dissertation submitted to the Graduate Faculty of
North Carolina State University
in partial fulfillment of the
requirements for the Degree of
Doctor of Philosophy

Electrical Engineering

Raleigh, North Carolina
2026

APPROVED BY:

Paul Franzon	Aydin Aysu
Yeonjong Shin	William Davis Chair of Advisory Committee

BIOGRAPHY

Donghyeon Koh received the BE degree in Electrical Engineering from the Sogang University in February 2019 and MS degree in Electrical and Computer Engineering from the North Carolina State University in May 2021. In August 2021, Donghyeon joined the Ph.D. program of the Department of Electrical and Computer Engineering at North Carolina State University.

ACKNOWLEDGEMENTS

Acknowledgments

Transferring to North Carolina State University to pursue a Ph.D. in Electrical Engineering has been one of the most defining and rewarding decisions of my academic journey. I am deeply grateful to my advisor, Dr. Rhett Davis, for offering me the opportunity to join the program and for his steadfast support throughout this endeavor. Without his initial encouragement, this dissertation would not have been possible.

I am equally thankful to Dr. Paul Franzon, Dr. Aydin Aysu, and Dr. Yeonjong Shin for serving on my dissertation committee. I deeply appreciate their time, thoughtful feedback, and meaningful contributions to my academic development.

I am especially grateful to Mr. Ryan Carey and Mr. Rajeev Jain, my mentors at Qualcomm Inc., whose advice and insights played a pivotal role in shaping my early research interests.

To my friends in MRC—Bharadwaj, Derek, Ziyu, Han, Priyank, and Josh—thank you for the camaraderie, laughter, and shared experiences that made this journey all the more memorable.

Finally, and most importantly, I would like to express my heartfelt gratitude to my family and friends. Your unwavering love, encouragement, and belief in me have been my greatest source of strength. This accomplishment is as much yours as it is mine.

TABLE OF CONTENTS

List of Tables . . . . . **ix**

List of Figures . . . . . **xi**

Chapter 1 INTRODUCTION **1**
1.1 Motivations 1
1.2 Research Contributions 2
1.3 Outline 4

Chapter 2 State-of-the-art **6**
2.1 Chapter Introduction 6
2.2 PPA Prediction with Machine Learning 7
2.3 Transfer Learning for PPA Prediction 9
2.4 Meta Learning 11

Chapter 3 Machine Learning Background **15**
3.1 Chapter Introduction 15
3.2 Neural Network Model 15
3.2.1 Model Structure 16
3.2.2 Initialization 18
3.2.3 Forward propagation 18
3.2.4 Backpropagation 19
3.2.5 Model Evaluation and Testing 19
3.2.6 Overfitting and Generalization 20
3.3 Graph Data Structure 21
3.3.1 Adjacency Matrix and Adjacency List 22
3.3.2 Directed Acyclic Graph 23
3.3.3 Directed Acyclic Graphs in Netlists and Timing Graphs 24
3.3.4 Graph Topology 26
3.4 Graph Neural Network Architecture 27
3.4.1 Graph Convolutional Network 28
3.4.2 Graph Attention Network [21] 29
3.4.3 Edge Graph Attention Network 30
3.5 Transfer Learning 31
3.5.1 GNN Transfer Learning 32
3.5.2 Meta Learning 34
3.6 Statistical Method 35
3.6.1 Relative Entropy 35
3.6.2 Wasserstein Metric 35
3.6.3 Rectified Linear Unit (ReLU) Activation Function 36

3.6.4 AdamW Optimization Algorithm . . . 36
3.6.5 Error Calculation Methods . . . 37

Chapter 4 PPA Prediction with GNN Transfer Learning with OpenABC-D dataset . . . 40
4.1 Chapter Introduction . . . 40
4.2 Dataset . . . 42
4.2.1 OpenABC-D Dataset: Large-Scale Pretraining via Open-Source Flow 42
4.2.2 Commercial-Flow Dataset: High-Fidelity Transfer Learning via Synopsys Flow . . . 45
4.2.3 Dataset Summary . . . 47
4.3 Model Structure and Transfer Learning Strategy . . . 47
4.4 Experiments . . . 49
4.4.1 Area Prediction Results . . . 49
4.4.2 Critical Path Delay (cdelay) Prediction Results . . . 49
4.5 Conclusion and Future Work . . . 50

Chapter 5 Accurate Few-shot PPA Prediction with Meta-Transfer Learning . . . 52
5.1 Chapter Introduction . . . 52
5.2 Datasets . . . 53
5.2.1 Features and Labels . . . 54
5.2.2 Relative Entropy and Wasserstein Distance . . . 55
5.3 Model Structure . . . 56
5.3.1 Base Model . . . 56
5.3.2 Outer Optimizations . . . 57
5.3.3 Inner Optimization . . . 58
5.4 Experiments . . . 58
5.4.1 Base Model . . . 60
5.4.2 Meta-transfer learning . . . 61
5.4.3 Ablation Study . . . 63
5.5 Conclusion and Future Work . . . 64

Chapter 6 Critical Path Delay Prediction for Evolving Netlists . . . 66
6.1 Chapter Introduction . . . 66
6.2 Dataset . . . 68
6.2.1 Features and Labels . . . 69
6.2.2 Graph Structure and Format . . . 71
6.2.3 Data Analysis . . . 74
6.3 Model Structure . . . 78
6.3.1 Base Model: GCN, GAT, and EGAT Architectures . . . 79
6.3.2 Guo-GNN: Timing-Engine Inspired GNN Architecture . . . 82
6.4 Experiments . . . 84

6.4.1 Base Model . . . 85
6.4.2 Transfer Learning . . . 89
6.5 Conclusion and Future Work . . . 95

Chapter 7 Conclusion and Future Work . . . 98
7.1 Conclusions . . . 98
7.2 Future Work . . . 100

References . . . 101

APPENDICES . . . 103
Appendix A Additional PPA Results . . . 104
A.1 Critical Path Delay Prediction for Evolving Netlists . . . 104
A.1.1 Graph Structure and Format . . . 104
A.1.2 Data Analysis . . . 106
A.1.3 Model Structure . . . 120
A.1.4 Experiments . . . 122
A.1.5 Selective Feature GNN . . . 133
Appendix B IEEE Conference Paper Submission . . . 142

AUTORSHIP STATEMENT

Contributions of Donghyeon Koh and coauthors are listed below for each chapter.

Chapter 1

1. Donghyeon Koh: sole author of Chapter 1

Chapter 2

1. Donghyeon Koh: sole author of Chapter 2

Chapter 3

1. Donghyeon Koh: sole author of Chapter 3

Chapter 4

1. Donghyeon Koh: Conceived and developed the methodology; carried out all analytical derivations; implemented all software; designed and executed simulations and applications; led manuscript preparation and revisions; conducted the literature review.

2. Dr. Davis: Provided statistical guidance on simulation design and interpretation; offered substantive editorial feedback and revisions; continuous advising on scope and execution.

3. Dr. Chowdhury and Dr. Siddharth: Supplied conceptual motivation via the OpenABC-D line of work and provided access to the associated dataset; offered technical input during the project.

4. Dr. Carey and Dr. Jain: Contributed editorial suggestions and ongoing practical guidance from an industry perspective.

Chapter 5

1. Donghyeon Koh: Designed the approach and experimental protocol; performed theoretical analysis; implemented code and ran studies; led drafting and revision of the chapter; completed the literature survey.

2. Dr. Davis: Helped frame the study in relation to Dr. Francisco work on transfer-learning PPA modeling; provided iterative feedback on analyses and writing; ongoing supervision and advice.

Chapter 6

1. Donghyeon Koh: Formulated the methods and analyses; derived all theoretical components; implemented the modeling pipeline; executed experiments; served as primary author and editor; performed the literature review.

2. Dr. Davis: Advised the project with reference to Dr. Lopera work on GNN-based timing estimation; contributed editorial review and strategic guidance throughout.

3. Dr. Shrestha and Dr. Savidis: Provided the dataset used in this chapter and offered domain-specific feedback.

Chapter 7

1. Donghyeon Koh: sole author of Chapter 7

LIST OF TABLES

Table 4.1 Graph Structure 44
Table 4.2 Transfer learning performance on area prediction. 49
Table 4.3 Transfer learning performance on critical path delay prediction. . . 50

Table 5.1 Features and Labels Scale Factor 54
Table 5.2 Wasserstein Distance between rocket design 56
Table 5.3 Pre-trained Base model evaluation 60
Table 5.4 Meta-transfer learning method evaluation by resetting layers 63
Table 5.5 Model evaluation for ablation study by not resetting the layers . . 64
Table 5.6 Model evaluation for ablation study by resetting the layers 64

Table 6.1 Pin-level Graph Structure 73
Table 6.2 Comparison of datasets used in this research 73
Table 6.3 Total Number of Gate per EDA stage 76
Table 6.4 Number of Gate Reduction between EDA stage 77
Table 6.5 Wasserstein distance on critical path delay decomposed by stage for each design 78
Table 6.6 Standard Deviation on critical path delay between stage for each design 79
Table 6.7 Comparison of GCN, GAT, EGAT, and [7] Model Performance . . 85
Table 6.8 Comparison on Critical Path Delay Prediction MAPE 88
Table 6.9 Statistics on Critical Path Delay Prediction MAPE 88
Table 6.10 Comparison of GCN, GAT, and EGAT Base Models before Retraining 89
Table 6.11 Comparison of GCN, GAT, and EGAT Retrained Models 90
Table 6.12 Critical Path Delay MAPE Comparison across GCN, GAT, and EGAT Before Retraining Models 92
Table 6.13 Critical Path Delay MAPE Comparison across GCN, GAT, and EGAT Retrained Models 93

Table A.1 Gate-level Graph Structure 105
Table A.2 Total Number of Gate for Post-Synthesis Stage 107
Table A.3 Total Number of Gate for Post-Floorplan Stage 108
Table A.4 Total Number of Gate for Post-Placement Stage 109
Table A.5 Total Number of Gate for Post-Cts Stage 110
Table A.6 Total Number of Gate for Post-Route Stage 111
Table A.7 Number of Gate Reduction between EDA stage for standard cell . 113
Table A.8 Wasserstein distance between each design and `ac97_ctrl`, using different normalization methods. 116
Table A.9 Wasserstein Distance of dataset [17] on critical path delay 120
Table A.10 Wasserstein Distance of dataset[17] normalized between 0 and 1 . . 121

Table A.11 Wasserstein Distance of dataset[17] normalized by Zscore 122
Table A.12 Wasserstein distance on PPA between stage for each design 123
Table A.13 Standard Deviation on PPA between stage for each design 124
Table A.14 Comparison on **arrival time / slack** prediction performance (R^2) and **runtime** (s). OpenROAD flow is the ground truth. We compare with results from [7] with the model recreation. 126
Table A.15 MAPE Comparison of GCN, GAT, EGAT, and [7] Model Performance 126
Table A.16 MAPE Comparison of GCN, GAT, and EGAT Base Models for Retraining 129
Table A.17 Area MAPE Comparison across GCN, GAT, and EGAT models . . 130
Table A.18 Critical Path Delay MAPE Comparison across GCN, GAT, and EGAT Models 133
Table A.19 Leakage Power MAPE Comparison across GCN, GAT, and EGAT models 134
Table A.20 Area MAPE Comparison across GCN, GAT, and EGAT Retrained Models 135
Table A.21 Critical Path Delay MAPE Comparison across GCN, GAT, and EGAT Retrained Models 135
Table A.22 Leakage Power MAPE Comparison across GCN, GAT, and EGAT Retrained Models 136
Table A.23 SF-GNN Graph Data Structure 136
Table A.24 Baseline Model Performance Comparison on Full Dataset 138
Table A.25 Critical Path Delay MAPE Comparison across GCN, GAT, EGAT, Guo-GNN [7], and SF-GNN for Train Dataset 139
Table A.26 Critical Path Delay Comparison across GCN, GAT, EGAT, Guo-GNN [7], and SF-GNN for Test Dataset 140

LIST OF FIGURES

Figure 3.1 Neural Network Structure 16
Figure 3.2 Adjacency matrix of directed graph 22
Figure 3.3 Adjacency list of directed graph 23
Figure 3.4 Predecessor and successor 24
Figure 3.5 Circuit Schematic 24
Figure 3.6 Gate focused graph representation 25
Figure 3.7 Pin focused graph representation 25
Figure 3.8 Transfer Learning Measurement [10] 33
Figure 3.9 Transfer Learning Node Classification [10] 33
Figure 3.10 Transfer Learning Graph Classification [10] 33

Figure 4.1 Transfer Learning Process 41
Figure 4.2 [3] benchmarks 43
Figure 4.3 [3] Data Distribution 45
Figure 4.4 [6] Data Distribution 46
Figure 4.5 Graph Neural Network architecture from OpenABC-D [3]. 47

Figure 5.1 Histogram on target Area 55
Figure 5.2 MAPE of 50 cross-validations on the area of CNN design 61
Figure 5.3 MAPE of 50 cross-validations on the area of three designs 62
Figure 5.4 MAPE of 50 cross-validations on leak power of three designs . . . 62

Figure 6.1 EDA Stage Algorithms with Predictions 69
Figure 6.2 Benchmark Comparison 70
Figure 6.3 Evolution of PPA and gate count across design stages for `ac97_ctrl` ([17]) 74
Figure 6.4 GNN Architecture 80
Figure 6.5 GNN Model Structure Details 81
Figure 6.6 GuoGNN Architecture [7] 82
Figure 6.7 GCN Cdelay Cross Validation Histogram 85
Figure 6.8 GAT Cdelay Cross Validation Histogram 86
Figure 6.9 EGAT Cdelay Cross Validation Histogram 87
Figure 6.10 Retrained GCN Cdelay Cross Validation Histogram 91
Figure 6.11 Retrained GAT Cdelay Cross Validation Histogram 91
Figure 6.12 Retrained EGAT Cdelay Cross Validation Histogram 91

Figure A.1 Critical Path Arrival Time and Total Area Relationship [17] . . . 112
Figure A.2 Critical Path Delay and Total Area Relationship [17] 114
Figure A.3 Leakage Power and Total Area Relationship [17] 114
Figure A.4 Other 16 designs Critical Path Delay and Total Area Relationship [17] 115

Figure A.5 Other 17 Designs Leakage Power and Total Area Relationship [17] 115
Figure A.6 [17] data PPA change through stage 117
Figure A.7 [17] data PPA change through stage 117
Figure A.8 [17] data PPA change through stage 117
Figure A.9 [17] data PPA change through stage 118
Figure A.10 [17] data PPA change through stage 118
Figure A.11 [17] data PPA change through stage 118
Figure A.12 [17] data PPA change through stage 119
Figure A.13 [17] data PPA change through stage 119
Figure A.14 [17] data PPA change through stage 119
Figure A.15 [17] data PPA change through stage 125
Figure A.16 GCN Area MAPE Cross Validation Histogram 127
Figure A.17 GAT Area MAPE Cross Validation Histogram 127
Figure A.18 EGAT Area MAPE Cross Validation Histogram 127
Figure A.19 GCN Pleak MAPE Cross Validation Histogram 128
Figure A.20 GAT Pleak MAPE Cross Validation Histogram 128
Figure A.21 EGAT Pleak MAPE Cross Validation Histogram 128
Figure A.22 Retrained GCN Area MAPE Cross Validation Histogram 131
Figure A.23 Retrained GAT Area MAPE Cross Validation Histogram 131
Figure A.24 Retrained EGAT Area MAPE Cross Validation Histogram 131
Figure A.25 Retrained GCN Pleak MAPE Cross Validation Histogram 132
Figure A.26 Retrained GAT Pleak MAPE Cross Validation Histogram 132
Figure A.27 Retrained EGAT Pleak MAPE Cross Validation Histogram 132
Figure A.28 SF-GNN Model Architecture Details 137

Chapter 1

INTRODUCTION

1.1 Motivations

Integrated circuit (IC) design is a complex and iterative process, where quality is often measured by three key metrics: power, performance, and area (PPA). However, accurate estimation of these metrics is typically available only at the end of the design cycle, following synthesis, placement, and routing. This late-stage feedback restricts opportunities for early-stage optimization and may lead to costly design revisions.

To address this, deep neural networks (DNNs) with transfer learning have been explored as a means of predicting PPA early in the design process. These methods leverage large training datasets from existing designs to adapt to new designs using only a few samples, offering promising results for early prediction [6]. Nevertheless, predictive fidelity—particularly for critical-path delay—varies markedly across designs, limiting robustness in practice. A likely cause is the difficulty of capturing netlist evolution across EDA stages: successive optimization passes alter graph topology, capacitive loads, and path constraints, inducing distributional shift between the features available at early stages and the timing labels realized post-route. Consequently, models calibrated on

pre-synthesis or pre-floorplan representations may underfit or misgeneralize at later stages unless they explicitly encode stage-conditioned structural and parasitic effects.

Recent work has shifted toward graph neural networks (GNNs), which offer a more natural representation of circuit structures. GNN-based models, such as those in [19] and [20], have shown high accuracy in predicting timing metrics like maximal arrival time or the longest path in abstract syntax trees. However, these studies are often limited to small, pre-synthesis designs and lack generalization to larger circuits. Moreover, none of these efforts have explored transfer learning in the context of GNNs.

This dissertation investigates the effectiveness of graph neural networks (GNNs) for transfer learning based prediction of critical-path delay in large, structurally diverse integrated-circuit designs. In addition, it examines the Wasserstein distance as a fast, interpretable proxy for dataset diversity and as a predictor of cross-design transferability. We further characterize netlist evolution across the EDA flow by tracking gate counts and cell type composition across stages. Collectively, these contributions aim to improve early stage PPA prediction and to enable more efficient, scalable IC design flows.

1.2 Research Contributions

This dissertation advances the state of the art through the following contributions:

1. **Stage-wise analysis of netlist evolution**

   We quantify structural drift across the EDA pipeline by tracking per-stage gate counts, standard-cell composition, and instance persistence. Motivated by prior reports that low critical path delay accuracy often stems from failing to model stage-induced transformations, we empirically verify that optimization introduces incremental, localized edits (e.g., buffering, replication, logic restructuring) rather

than wholesale netlist replacement. This supports the premise that the resulting evolution is amenable to GNNs: iterative message passing over the netlist graph can propagate and integrate these stage-to-stage changes.

2. **Comprehensive evaluation of various graph neural network (GNN) architectures for large-scale critical path delay prediction**

   We conduct an extensive comparison of multiple GNN architectures, such as Graph Convolutional Networks (GCNs), Graph Attention Networks (GATs), and Edge Graph Attention Networks (EGATs), in terms of their effectiveness in predicting critical path delay across a diverse range of large-scale integrated circuit designs. Using cross-validation, we quantify and compare the predictive accuracy, generalization ability, and computational cost of each GNN variant, offering practical insight into their suitability for timing analysis tasks.

3. **First application of transfer learning in GNN-based critical path delay prediction across heterogeneous IC designs**

   To the best of our knowledge, this is the first study to apply transfer learning to GNNs for the purpose of adapting delay prediction models across structurally diverse IC designs. We demonstrate that pretrained GNNs can be fine-tuned on new designs with minimal labeled data, thereby significantly reducing the time and computational resources required to build accurate predictors for unseen design configurations.

4. **Introduction of the Wasserstein distance as a dataset similarity measure for assessing transfer learning effectiveness in GNNs**

   We introduce the use of the Wasserstein distance (also known as Earth Mover's Distance) to quantify the statistical dissimilarity between training and target datasets.

This distance metric serves as a predictive indicator of transfer learning success, enabling informed selection of source designs that are more likely to yield high transfer performance when adapting GNN models to new tasks.

5. **Cross-tool generalization through transfer learning between open-source and commercial EDA flows**

   This work presents the first investigation into transferring GNN-based timing models across datasets generated from different EDA toolchains. Specifically, we train GNNs using data generated by open-source tools such as OpenROAD and OpenSTA, and subsequently fine-tune the models on designs produced by commercial tools like Synopsys Design Compiler and IC Compiler II. This cross-tool transfer pipeline demonstrates the feasibility of leveraging high-throughput open-source data to accelerate learning on more limited but high-accuracy commercial datasets.

1.3 Outline

The remainder of this dissertation is structured as follows.

Chapter 2 reviews existing literature on the application of machine learning techniques for predicting PPA metrics in IC design. Chapter 3 provides the necessary theoretical background on neural network (NN) models and graph-based data representations, establishing the foundation for the graph neural network (GNN) methodologies employed throughout this research.

Chapter 4 presents a study on early-stage PPA prediction using transfer learning techniques, leveraging the dataset introduced in [3]. Chapter 5 explores a meta-transfer learning framework based on few-shot learning, designed to enable accurate PPA prediction across a wide range of IC designs using limited labeled data. Chapter 6 investigates

the prediction of critical path delay in evolving netlists using a pre-floorplan dataset from [17], and evaluates the effectiveness of GNN-based transfer learning in this setting.

Finally, Chapter 7 summarizes the key contributions of this dissertation, discusses current limitations, and outlines potential directions for future research.

Chapter 2

State-of-the-art

2.1 Chapter Introduction

The increasing complexity of integrated circuit (IC) design has driven a growing interest in machine learning (ML) techniques as an alternative to traditional rule-based methods in electronic design automation (EDA). Predicting key quality-of-results (QoR) metrics such as power, performance, and area (PPA) early in the design flow remains a central challenge. Conventional approaches require full synthesis and place-and-route flows, which are computationally expensive and time-consuming. As a result, there is significant value in developing data-driven models that can accurately estimate PPA at early design stages using only partial design information.

Recent research has demonstrated the potential of ML-based models—including neural networks, graph neural networks (GNNs), and large language models—to approximate PPA metrics with varying levels of abstraction. However, these models often suffer from limited generalization across different design spaces and technology stacks, especially when trained on narrowly scoped datasets. Transfer learning has emerged as a promising solution, enabling knowledge reuse across related designs to improve prediction accuracy

while reducing data requirements. Additionally, advances in meta-learning and few-shot learning provide new frameworks for adapting models to unseen designs with minimal training overhead.

This chapter surveys the current state of the art in three key areas that form the foundation of this dissertation. Section 2.2 reviews machine learning models developed for PPA prediction, including those based on handcrafted features, RTL graphs, and post-synthesis netlists. Section 2.3 explores transfer learning methods applied to EDA and hardware modeling tasks. Section 2.4 introduces recent work in meta-learning, with a focus on techniques that enable fast adaptation to new designs under limited supervision. Collectively, this literature review provides both the context and motivation for the technical approaches developed in later chapters.

2.2 PPA Prediction with Machine Learning

In order to accurately estimating PPA metrics at early stages of the IC design flow, recent research has explored the use of machine learning models to approximate PPA metrics using abstract design representations such as register-transfer level (RTL) code, intermediate graphs, or analytical features. These data-driven approaches aim to significantly reduce turnaround time while maintaining predictive accuracy, especially in early design iterations where rapid feedback is essential. The following works represent key advances in ML-based PPA prediction, covering neural models trained on accelerator designs, RTL-level graph representations, placement-aware timing graphs, circuit-structured GNNs, and even large language models trained directly on hardware code.

Juracy et al. [9] address the challenge of fast and accurate PPA estimation for CNN hardware accelerators. They develop an analytical model calibrated using physical synthesis results, capturing architectural, technological, and workload-level features to estimate

area, power, and performance. Integrated into high-level design frameworks, their model enables rapid design-space exploration without requiring full synthesis. Experiments show average errors below 7%, demonstrating strong alignment with post-synthesis PPA and validating its effectiveness for early-stage exploration.

Lopera et al. [19] propose a GNN-based method to predict timing characteristics directly from RTL-level representations, eliminating the need for synthesis or STA. RTL designs are transformed into graphs encoding operator and control dependencies, and a GNN is trained to learn timing-sensitive features and propagate delay estimates. The model achieves a correlation of 0.82 with post-synthesis timing results and highlights timing-critical RTL lines, enabling early timing feedback with orders-of-magnitude faster runtime than traditional flows.

Guo et al. [7] introduce a timing-engine–inspired GNN model for predicting post-routing arrival times using only post-placement data. To avoid limitations like over-smoothing and shallow receptive fields, they design a levelized, asynchronous message-passing scheme that mimics STA's forward propagation. Edge-level supervision and graph partitioning further enhance generalization and scalability. Trained on Sky130 OpenROAD designs, their model achieves an R^2 of 0.95 on training and 0.90 on unseen circuits, while delivering over 2,600× speedup compared to full routing+STA.

Dong et al. [5] present CktGNN, a domain-specialized GNN for circuit-level prediction tasks. Unlike generic GNNs, CktGNN incorporates direction-aware message passing, edge-type sensitivity, and circuit-context embeddings to model signal flow, fanout, and timing dependencies. Applied to tasks such as delay prediction, area estimation, timing-criticality ranking, and design classification, CktGNN consistently outperforms baseline GNNs and classical machine-learning models. Across benchmarks, CktGNN improves delay-prediction accuracy by 8–15% over prior GNN architectures, achieves over 90% accuracy on timing-criticality ranking, and reduces prediction error by up to 12% on

area estimation tasks. These improvements highlight the importance of incorporating circuit-specific inductive biases into graph-based models for EDA.

Abdollahi et al. [1] propose RocketPPA, a large-language-model–based framework that predicts post-route PPA metrics directly from RTL code, bypassing the full RTL-to-GDSII flow. Fine-tuning CodeLLaMA with LoRA adapters and a regression head, the model learns to infer delay, power, and area from code-level patterns. Trained on 21,000 Verilog modules with signoff PPA labels, RocketPPA achieves 5–7% accuracy gains over LLM-only baselines and delivers 1,000×+ speedup over full flows. It supports early-stage pruning and generalizes across a wide range of RTL structures.

2.3 Transfer Learning for PPA Prediction

Transfer learning has emerged as a powerful tool for accelerating PPA prediction in IC design by reducing data requirements and training costs. Rather than training a model from scratch for each new design, transfer learning leverages knowledge acquired from related tasks or datasets—enabling generalization across designs, technology nodes, or architectural variants. Recent research spans regression-based models, graph-based frameworks, Bayesian optimization pipelines, and reinforcement learning agents, each integrating transfer learning to improve efficiency, scalability, or adaptability. Below, we summarize key contributions that highlight diverse applications of transfer learning for design-space exploration and modeling.

Francisco et al. [6] present a hybrid modeling framework that combines gradient boosting and neural networks to predict post-synthesis PPA metrics from synthesis-stage parameters. By fine-tuning a subset of model layers on new configurations, the authors demonstrate $> 95\%$ prediction accuracy using only 10–15 training samples. Their transfer-learning strategy enables 8–9× faster adaptation across design variants, making

it well-suited for rapid system-level design exploration.

Last et al. [11] propose a deep transfer learning pipeline for embedded memory PPA modeling. Their approach uses domain adaptation and feature normalization to mitigate distribution shifts between memory compilers and configurations. Results show that transfer learning reduces the number of required training samples by up to 90%, achieving sub-3% prediction error across diverse SRAM designs and technology nodes.

Zhang et al. [25] combine transfer learning with multi-objective Bayesian optimization to accelerate parameter tuning for circuit and system-level design. By leveraging priors learned from previous optimization tasks, their framework achieves near-identical Pareto-optimal solutions while using only 10–20% of the function evaluations required by standard Bayesian optimization, and reduces tuning time by approximately an order of magnitude when the source and target tasks share structural similarity. These results demonstrate that transfer-enhanced MOBO can substantially reduce computational cost while maintaining high-quality design trade-offs, enabling efficient reuse of historical optimization data across technology nodes and design families.

Wang et al. [24] introduce a GCN-RL framework for automated transistor sizing, where graph embeddings capture circuit topology and reinforcement learning selects device sizes. Compared to traditional optimization methods—including Bayesian Optimization, Evolution Strategies, and expert-designed heuristics—their method achieves up to 26% higher Figure of Merit (FoM) across various amplifier topologies and reduces the number of required simulations by $2\times$ to $5\times$ when transferred to new circuits or technology nodes. Through transfer learning, policies trained on one node or topology are efficiently adapted to others with minimal fine-tuning, enabling significant reductions in simulation time while preserving high performance across competing design objectives such as gain, power, and bandwidth.

Kooverjee et al. [10] provide a comprehensive study on transfer learning in GNNs,

showing that transfer effectiveness depends strongly on structural similarity between source and target graphs. When graph domains are well-aligned, transfer learning improves accuracy by up to 12.5% and reduces required training samples by 30–50% compared to training from scratch. They find that partial fine-tuning, such as freezing early GNN layers, often outperforms full fine-tuning in moderately similar domains, and that self-supervised pretraining further enhances generalization. However, in cross-domain settings with mismatched topology or node semantics, transfer can lead to performance degradation of up to 7%, underscoring the importance of domain-aware pretraining and architectural adaptation.

Vlcek et al. [23] propose ApproxGNN, a pretrained GNN framework for approximate computing design-space exploration. The model learns reusable embeddings for approximate arithmetic components and transfers them to system-level accelerators. This two-stage pipeline reduces prediction error by up to 54% with minimal fine-tuning and offers substantial speedup over traditional synthesis-based exploration.

2.4 Meta Learning

Few-shot learning aims to enable models to generalize from only a small number of training samples. Meta-learning extends this goal by training a model—referred to as a meta-learner—across many related tasks so that it can quickly adapt to a new task with minimal additional data or gradient steps. Meta-learning frameworks have been widely adopted across classification, regression, and reinforcement learning, and their principles form the basis for emerging methods in fast, data-efficient model adaptation.

Li et al. [12] introduce Meta-SGD, a gradient-based meta-learning method that learns not only an optimal initialization but also per-parameter learning rates and update directions. This enables rapid adaptation with a single gradient step, yielding faster

convergence and improved accuracy compared to MAML and other gradient-based baselines. On few-shot learning benchmarks, Meta-SGD achieves 95.93% accuracy on 5-way 1-shot Omniglot and 63.47% on 5-way 1-shot MiniImageNet, outperforming MAML by 2–4% across tasks while requiring fewer inner-loop updates. These results highlight the importance of jointly learning both initial parameters and learning dynamics for robust and efficient task adaptation.

Munkhdalai and Yu [15] propose MetaNet, a meta-learning framework that incorporates fast weight generation to enable single-step or few-step adaptation. MetaNet consists of a base learner for task prediction and a meta-learner that produces fast weights derived from task-specific gradients. This architecture supports rapid adaptation and strong generalization across few-shot tasks. On the Omniglot dataset, MetaNet achieves 98.95% accuracy on 5-way 1-shot and 99.35% on 5-way 5-shot classification, while on MiniImageNet, it reaches 49.21% on 5-way 1-shot and 63.84% on 5-way 5-shot, demonstrating competitive or state-of-the-art results at the time of publication. The dual-timescale learning mechanism positions MetaNet as an influential contribution in early meta-learning research.

Sun et al. [18] introduce Meta-Transfer Learning (MTL), which combines transfer learning from pretrained deep networks with meta-learning for efficient few-shot adaptation. The method introduces lightweight scale-and-shift parameters to modulate pretrained convolutional filters and employs hard-task meta-batching to emphasize challenging tasks during training. On the miniImageNet benchmark, MTL achieves 60.2% $\pm$ 1.8% accuracy for 5-way 1-shot and 74.3% $\pm$ 0.9% for 5-way 5-shot classification, outperforming variants without hard-task meta-batching by 1.0–1.2%. On the FC100 dataset, MTL reaches 43.6% $\pm$ 1.8% (1-shot) and 55.4% $\pm$ 0.9% (5-shot), with hard-task meta-batching further improving performance to 45.1% and 57.6%, respectively. These results demonstrate the benefits of merging transfer-learned representations with meta-learned

rapid adaptation mechanisms.

The state of the art on PPA prediction reveals several persistent limitations:

- **Limited accuracy in predicting critical–path delay.** Despite meaningful advances, many ML approaches struggle to achieve stable, high-fidelity critical path delay prediction across heterogeneous designs and flows.
- **Lack of a principled measure of inter-design distributional shift.** Most prior work does not quantify how far a target design distribution is from the source, complicating domain adaptation and making transfer performance difficult to anticipate.
- **Substantial data and compute requirements for pretraining.** State-of-the-art pipelines typically rely on large labeled corpora and long pretraining cycles, incurring significant tool runtime and engineering cost before transfer learning can be effective.

To address these gaps, this dissertation pursues the following research thrusts:

- **Comparative evaluation of GNN architectures** (e.g., GCN, GAT, EGAT, and timing-inspired variants) for large-scale critical path delay prediction, providing empirical evidence and practical guidance on architectural suitability for timing analysis.
- **Transfer learning for critical path delay prediction across heterogeneous IC designs** to reduce labeled-data requirements and training time while maintaining predictive accuracy under distribution shift.
- **Use of the Wasserstein distance** as a quantitative proxy for inter-design distributional distance, enabling informed data selection, curriculum design, and model assessment in domain-adaptation settings.

- **Cross-tool generalization** from open-source flows (e.g., OpenROAD/Yosys) to commercial toolchains (e.g., Synopsys ICC2), thereby lowering the cost of generating large pretraining datasets prior to transfer.

Collectively, these contributions aim to advance data-efficient, early-stage timing prediction and to establish principled strategies for architecture selection and domain adaptation in the presence of distribution shift.

Chapter 3

Machine Learning Background

3.1 Chapter Introduction

This chapter provides the foundational background relevant to the proposed research. It is divided into three main sections: neural network (NN) model training, graph-based data structures, and graph neural network (GNN) architecture. The first section introduces the architecture and learning algorithms of traditional neural networks, followed by an overview of graph neural networks (GNNs). The second section describes graph-structured representations commonly used in circuit modeling, with particular emphasis on directed acyclic graphs (DAGs), which form the core data structure in this work. The third section presents a common GNN architecture used in various research. Finally, we present the statistical methods to measure the dataset.

3.2 Neural Network Model

A neural network (NN) is a type of machine learning model designed to learn complex patterns from data. Inspired by the structure of biological neurons, NNs consist of

multiple interconnected layers of artificial neurons. Each neuron applies a learned transformation to its inputs and passes the result to the next layer. As illustrated in Figure 3.1, a typical feedforward NN is composed of three main components: the input layer, one or more hidden layers, and the output layer. The input layer receives external data, while the output layer produces the final predictions of the model. The number of neurons in the output layer depends on the nature of the task (e.g., regression or classification). Hidden layers serve as intermediate processing stages that enable the network to learn hierarchical representations of the data.

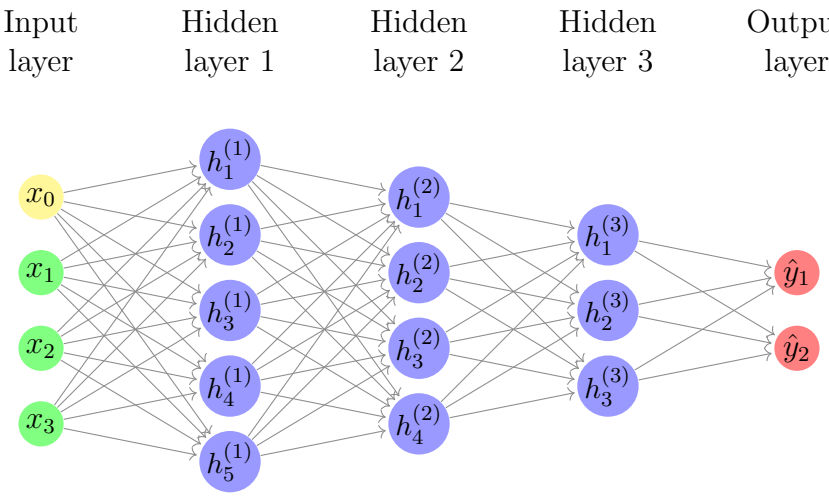


Figure 3.1: Neural Network Structure

3.2.1 Model Structure

A feedforward neural network (FFNN) is composed of an ordered sequence of layers, where each layer consists of a set of artificial neurons. These neurons are computational units that apply an affine transformation followed by a nonlinear activation function. The overall architecture typically includes the following components:

- **Input Layer**: This layer receives the raw input features from the dataset. Each neuron in the input layer corresponds to one feature. The input layer does not perform any transformation but simply passes the input to the next layer. It is denoted as $a^{(0)} = x$, where x is the input vector.

- **Hidden Layers**: One or more intermediate layers, known as hidden layers, perform the core computations of the network. Each hidden layer receives activations from the previous layer, applies a linear transformation using a weight matrix and bias vector, and then passes the result through a nonlinear activation function. These layers allow the network to learn complex, hierarchical representations of the input data.

- **Output Layer**: The final layer of the network produces the model's prediction. The number of neurons in this layer and the choice of activation function depend on the task type. For regression tasks, a single neuron with a linear or identity activation is commonly used; for classification, a softmax or sigmoid activation is often applied.

The architecture is fully connected, meaning each neuron in one layer is connected to every neuron in the next layer. The complexity and capacity of the network are determined by the number of layers (depth) and the number of neurons per layer (width).

The NN model learning adapts the network to have a better prediction of the problem. The NN learning algorithm is divided into four separate procedures: Initialization, forward propagation, backpropagation, and testing.

3.2.2 Initialization

In the initialization step, the model parameters (weights and biases) are assigned initial values before training begins. Poor initialization can lead to vanishing or exploding gradients, hindering effective learning. Common initialization techniques include Xavier initialization and He initialization, which are designed to maintain stable variance across layers.

3.2.3 Forward propagation

The forward propagation phase passes the input data through the network where each layer applies a transformation to the input data using its parameters and an activation function. The final layer, an output layer, produces a prediction that goes through a loss function to measure the difference between the predicted values and the actual values. At each layer ℓ, the neuron computes a weighted sum $z^{(\ell)}$ of its inputs:

$$z^{(\ell)} = W^{(\ell)} \cdot a^{(\ell-1)} + b^{(\ell)} \tag{3.1}$$

where $W^{(\ell)}$ is the weight matrix between layer ℓ and $\ell - 1$, $b^{(\ell)}$ is the bias vector, and $a^{(\ell-1)}$ is the activation from the previous layer. This value is then passed through a nonlinear activation function $\sigma^{(\ell)}$ to produce the output of the layer:

$$a^{(\ell)} = \sigma^{(\ell)}(z^{(\ell)}) \tag{3.2}$$

Equation 3.2 is applied to all hidden layers and the output layer, since the input layer does not have an activation function. Instead, the input is directly assigned as the activation of layer zero: $a^{(0)} = x$, where x represents the input data vector.

3.2.4 Backpropagation

Backpropagation is the process of computing the gradient of the loss function with respect to the network parameters. It proceeds from the output layer backward to the input layer using the chain rule of calculus in the direction that minimizes the loss function. For each layer ℓ, the gradient of the loss with respect to parameters θ^ℓ (including $W^{(\ell)}$ and $b^{(\ell)}$) is computed as:

$$\frac{\partial L}{\partial \theta^\ell} = \frac{\partial L}{\partial a^\ell} \cdot \frac{\partial a^\ell}{\partial z^\ell} \cdot \frac{\partial z^\ell}{\partial \theta^\ell}. \tag{3.3}$$

The gradient of the loss function L to the activation of the output layer a^L is first computed and propagated backward through the layers of the NN. For each layer, the gradient of the loss function to the activation of the layer $\partial L/\partial a^\ell$ is used to compute the gradient of the loss to the weighted sum of input z^ℓ. Then, the gradient of the loss to the layer parameter θ^ℓ is computed.

The parameters are then updated using an optimization algorithm such as stochastic gradient descent (SGD), Adam, or RMSProp:

$$\theta^\ell \leftarrow \theta^\ell - \alpha \cdot \frac{\partial L}{\partial \theta^\ell} \tag{3.4}$$

where α is the learning rate that controls the step size in the update.

3.2.5 Model Evaluation and Testing

The process of forward propagation, loss calculation, backpropagation, and parameter update continues for a fixed number of iterations (epochs) or until the model converges to a satisfactory solution.

After training, the performance of the trained model is evaluated using a separate

validation dataset to tune hyperparameters and monitor overfitting.

Once validated, the final model is tested on unseen data to assess its generalization ability. Common evaluation metrics include mean squared error (MSE), accuracy, or R-squared (R^2) score, depending on the specific task.

3.2.6 Overfitting and Generalization

One of the central challenges in training neural networks is achieving good generalization—that is, the ability of the trained model to perform well not only on the training data but also on unseen test data. A neural network that performs extremely well on the training set but poorly on new data is said to be overfitting. Overfitting occurs when the model captures noise, outliers, or spurious patterns in the training data that do not generalize to the underlying distribution of the task.

Overfitting is particularly common in deep networks with a large number of parameters relative to the size of the training dataset. These models have high capacity and can memorize training data rather than learn generalizable features. This leads to low training error but high test error.

To mitigate overfitting, several regularization techniques are commonly employed:

- **Early stopping**: Monitoring the model's performance on a validation set and halting training when performance degrades.
- **Dropout**: Randomly deactivating a subset of neurons during each training iteration to prevent co-adaptation.
- **Weight regularization**: Adding penalty terms to the loss function (e.g., L1 or L2 norms) to constrain the magnitude of model weights.

- **Data augmentation**: Expanding the training dataset with perturbed versions of existing examples to improve diversity.

In contrast, underfitting occurs when the model is too simple to capture the underlying patterns in the data, leading to poor performance on both training and test sets. A well-trained neural network should balance complexity and data fit in a way that minimizes generalization error.

Understanding the trade-off between underfitting and overfitting is essential for designing robust neural architectures, especially when working with limited or imbalanced datasets, as is often the case in electronic design automation (EDA) applications.

3.3 Graph Data Structure

A graph is a data structure that represents data by the relationships among objects and entities. Examples of data commonly modeled as graphs include images, text, and social networks. The graph expresses the data with a set of vertices (**V**) and a set of edges (**E**), which could be denoted as **G(V,E)**.

1. **Vertices**

   Vertices, also known as nodes, are a fundamental unit of a graph. They represent objects or entities in the data. Each vertex may be associated with a feature vector or label that encodes its attributes.

2. **Edges**

   Edges are the connections or relationships between pairs of vertices. An edge may be directed or undirected, depending on whether the relationship it models has a direction. The attributes, such as weights or labels, which encode additional information about the relationship, could be carried inside the edges.

$$\begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

Figure 3.2: Adjacency matrix of directed graph

3.3.1 Adjacency Matrix and Adjacency List

Both adjacency lists and adjacency matrices are methods used to represent the connectivity of nodes in a graph. Each method has its method for storing graph information followed by advantages and disadvantages.

The adjacency matrix is a square matrix used to represent the graph. It has a dimension of $|V| \times |V|$, where $|V|$ is the number of vertex in the graph. Each cell of the matrix represents a possible edge between two vertices. If there is an edge from vertex i to vertex j, the corresponding cell in the matrix is set to 1; otherwise, it is set to 0. For an undirected graph, the matrix is symmetric. However, in an adjacency matrix for a directed graph shown in 3.2, the matrix may not be symmetric since edges have different directions. This method is efficient for checking connections between vertices and is easy to compute since it is in matrix form. However, it is inefficient for storing large graph data due to its required dimension. Also, it is inefficient for removing and adding edges which requires updating the entire row or column.

The adjacency list is a collection of lists, where each list represents the neighbors of a particular vertex in the graph. For each vertex v, its adjacency list contains all the vertices that are adjacent to v. In a directed graph shown in 3.3, each edge is represented only once in the adjacency lists, as it is stored in the list of the source vertex. This method is efficient for storing large graphs and makes it easy to add or remove edges of a graph. However, compared to the adjacency matrix, it is inefficient for checking the connection between vertices and mathematical computation due to its form.

$$
\begin{aligned}
A: &\quad [C] \\
B: &\quad [C, D] \\
C: &\quad [A, B] \\
D: &\quad [B]
\end{aligned}
$$

Figure 3.3: Adjacency list of directed graph

3.3.2 Directed Acyclic Graph

A directed acyclic graph (DAG) is a type of graph with directed edges and does not contain any cycles. In other words, the DAG has the properties of a directed graph and an acyclic graph. A directed graph, also known as a digraph, has directed edges which indicate the relations between vertices are directional. An acyclic graph is a graph that does not contain any cycles. A cycle in a graph is a path that starts and ends at the same vertex, traversing one or more edges in the process. This indicates that no sequence of edges starts and ends at the same vertex without repeating any edges. These properties of DAG make predecessor and successor play an important part in the directed acyclic graph neural network (DAGNN) which will be covered later in this chapter.

The predecessor and successor of the vertex refer to the relations between vertex based on the directed edges. Figure 3.4 illustrates the relations between the predecessor and successor of a vertex. The predecessor of a vertex h_1 is a vertex with a directed edge pointing to vertex h_1 which are x_0 and x_1. The successor of a vertex h_1 is a vertex with a directed edge originating from vertex h_1 which is y_1. The predecessor and successor determine the direction of information flow and dependencies between the vertices which plays a crucial role in DAGNN.

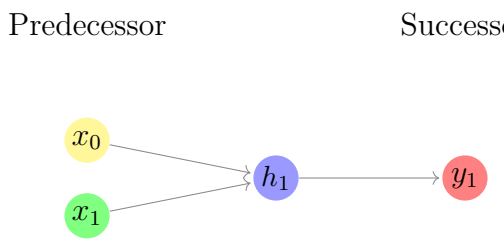


Figure 3.4: Predecessor and successor

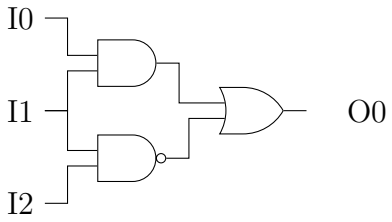


Figure 3.5: Circuit Schematic

3.3.3 Directed Acyclic Graphs in Netlists and Timing Graphs

There are several methods to map data to the graph. Consider Figure 3.5 which consists of simple AND, NAND, and OR logic gates. As shown in Figure 3.6 and 3.7, the logic schematic is mapped to two different graph representations. Figure 3.6 focuses on gate features of the logic schematic by expressing gates as vertices and edges as the connection between the gates. The vertices are labeled to represent the types of the gates. On the other hand, Figure 3.7 focuses on the pin features of the gate and their interconnections by expressing pins as vertices and the edges as the connection between pins and the gate process. The input and output pins are labeled to represent the attributes of the vertices.

And-inverter Graph

An And-Inverter Graph (AIG) is a type of directed acyclic graph (DAG) that represents Boolean functions using a combination of two-input AND gates and inverters. AIGs are widely used in digital logic synthesis, optimization, and formal verification due to

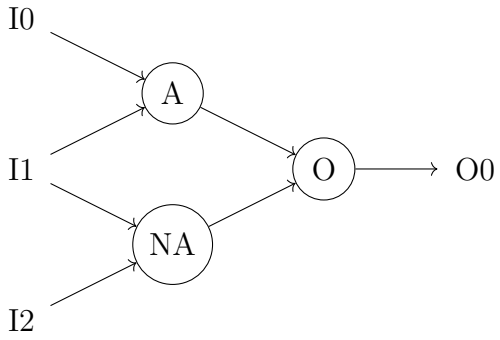


Figure 3.6: Gate focused graph representation

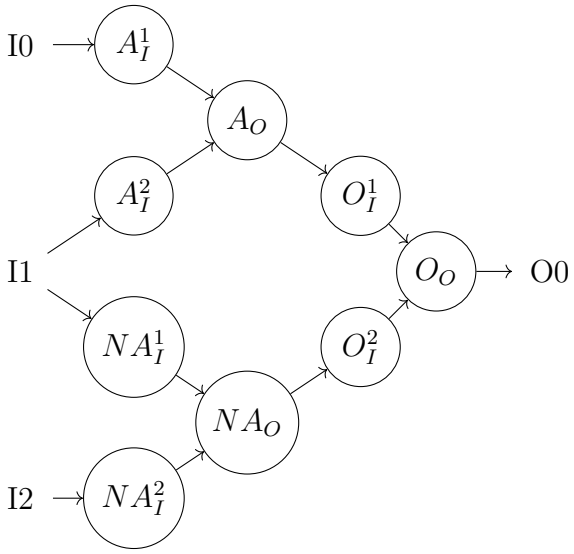


Figure 3.7: Pin focused graph representation

their canonical structure, simplicity, and suitability for structural hashing and rewriting techniques.

Formally, an AIG is defined as a DAG $G = (V, E)$ where each node $v \in V$ represents either a primary input (PI), a two-input AND gate, or a primary output (PO), and each edge $e \in E$ may carry an optional inversion marker (typically implemented as a single-bit flag). The use of only AND gates and inverters provides a normalized and structurally uniform representation of logic circuits, facilitating compact storage, efficient traversal, and algebraic manipulation.

Each non-terminal node in an AIG has exactly two incoming edges corresponding to the two operands of an AND gate. Inversion is modeled by annotating the edge rather than introducing separate NOT gates, allowing for compact expression of logical negation. This results in a minimal and consistent representation, which is especially advantageous for optimization tasks such as redundancy removal, logic resubstitution, and technology mapping.

AIGs are particularly effective in enabling fast equivalence checking, model checking, and SAT-based verification, and form the core data structure in tools such as AIGER [2], Gini logic library [4], and others. In machine learning applications, AIGs can serve as input graphs to graph neural networks (GNNs), where nodes represent logic elements and edge annotations capture logical polarity. Due to their acyclic nature and well-defined semantics, AIGs provide an excellent foundation for structure-aware learning tasks in EDA, such as power, performance, and area (PPA) prediction or logic path analysis.

3.3.4 Graph Topology

One of the fundamental advantages of GNNs is their ability to incorporate and exploit topological structure in the learning process. Unlike traditional neural networks, which assume that input features are independent or have a fixed spatial ordering, GNNs operate on graph-structured data where the relationships between entities are defined by irregular and arbitrary connectivity. This makes GNNs particularly well-suited for domains such as electronic design automation (EDA), where circuits are naturally represented as graphs of gates, nets, or timing paths.

In such settings, the topology of the graph captures critical structural information—for example, the number and direction of connections (fan-in, fan-out), depth of logic chains, or the presence of reconvergent paths. These factors have a significant influence on performance metrics such as delay, power, and signal integrity. By incorporating message passing between connected nodes, GNNs allow the network to learn context-aware representations, where each node's embedding reflects not only its own features but also the structure and attributes of its neighborhood. As a result, models that preserve and leverage graph topology can achieve better generalization and interpretability, especially for complex tasks like timing analysis or PPA prediction.

3.4 Graph Neural Network Architecture

The graph neural network (GNN) is a neural network architecture designed to operate on graph-structured data. The GNN architecture is divided into three separate procedures: node representation, message passing, and readout.

1. **Node representation**

   In the node representation procedure, GNN initializes every vertex with initial feature h_v^0, usually the same as the input graph data vertex feature x_v.

2. **Message passing**

   In the message passing procedure, each vertex aggregates information from its neighboring vertex and updates its representation based on the aggregated information. This process is performed over multiple layers of GNN, allowing vertices to gather and propagate information through the graph.

$$\begin{aligned} m_v^\ell &= G^\ell(h_u^{(\ell-1)} | u \in N(v)), \quad \ell = 1, \cdots, L \\ h_v^\ell &= F^\ell(h_v^{(\ell-1)}, m_v^\ell), \quad \ell = 1, \cdots, L \end{aligned} \tag{3.5}$$

   The Equation 3.5 expresses the mathematical representation of the message passing procedure. The aggregate message at vertex v at layer ℓ (m_v^ℓ) is defined by using the aggregation function G^ℓ to the neighborhood of vertex v denoted as $N(v)$ in the previous layer. This procedure is done for the number of layers L. Thus, the current layer representation is defined using the update function (F^ℓ) to the aggregated message and the vertex representation of the previous layer.

3. **Readout**

the final representations of the nodes are typically aggregated to produce a graph-level output. This final aggregation step, known as readout or graph pooling, summarizes the information learned from the entire graph and produces an output vector representing the entire graph. The mathematical representation of a readout procedure is as follows,

$$h_G = R(h_v^L, v \in V) \tag{3.6}$$

where h_G denotes the final graph-level representation which is defined using the readout function R.

In the traditional NN model, the sequential data is passed through the model where the connectivity pattern is predefined and unchangeable. The model operates from the input layer to the output layer where there is no direct interaction between vertices.

On the other hand, the GNN operates with graph-structured data where the connections are arbitrary and have different attributes. The connectivity pattern of the GNN model is determined by the structure of the input graph. The GNN aggregates the information from its neighbors, updates its representation, and then passes this updated information to its neighboring nodes.

3.4.1 Graph Convolutional Network

Graph Convolutional Network (GCN) is a type of Graph Neural Network (GNN) architecture that extends the concept of convolutional operations to graph-structured data. GCN uses convolutional operations for message passing and node representation learning on graphs to aggregate information from neighboring nodes and update node representations. The convolutional operation is a mathematical operation involving the element-wise

multiplication of a set of weights (kernel/filter) with a local region of input data, followed by summation. These operations extract features from input data while preserving spatial or structural relationships.

The convolutional operation of GCN occurs in the graph convolutional layer (GCL) used in the message passing procedure of GCN. The GCL aggregates information from neighboring nodes using weighted combinations of their feature vectors and updates the node representations accordingly. The mathematical expression of GCL in message passing procedure is as follows,

$$h_v^\ell = \sigma\left(\sum_{u \in N(v)} \mathbf{W} \cdot h_u\right), \quad \ell = 1, \cdots, L \tag{3.7}$$

where $\mathbf{W}$ represents the learnable weight matrix and σ represents the activation function applied element-wise to the sum. Thus, the aggregation is denoted as a weighted sum of the feature vectors.

3.4.2 Graph Attention Network [21]

Graph Attention Network (GAT) is a type of Graph Neural Network (GNN) architecture that extends the idea of convolutional operations to graphs using attention mechanisms. GAT enables nodes in a graph to selectively attend to their neighbors' features during message passing, allowing for more flexible and expressive representation learning. This process is done by computing attention weights for each edge to determine the importance of one node to the other.

$$e_{ij} = a(\mathbf{W}h_i, \mathbf{W}h_j) \tag{3.8}$$

Equation 3.8 is the attention coefficient which express the importance of node j's

features to node i. a and $\mathbf{W}$ represents the learnable attention vector and learnable weight matrix whereas h_i and h_j is the node features of node i and j. In order to express the coefficients comparable across the different nodes, we use the normalization equation which is shown in Equation 3.9

$$a_{ij} = \frac{\exp(e_{ij})}{\sum_{k \in N_i} \exp(e_{ik})} \tag{3.9}$$

where N_i is the neighborhood of node i in the graph. Depending on which attention mechanism a is used in Equation 3.8, the LeakyReLU nonlinearity might apply.

$$h'_i = \|_{k=1}^{K} \sigma \left(\sum_{j \in N_i} a_{ij}^k \mathbf{W}^k \cdot h_j \right) \tag{3.10}$$

Equation 3.10 express the node features of i which are updated by aggregating features from neighbors, weighted by the attention coefficients. a_{ij}^k is the normalized attention coefficients computed by the k^{th} attention mechanism a^k and $\mathbf{W}^k$ is the input linear transformation weight matrix corresponding to a_{ij}^k. σ is the activation function, and $\|$ as the concatenation function.

3.4.3 Edge Graph Attention Network

The Edge Graph Attention Network (EGAT) is the Graph Attention Network (GAT) designed to handle edge-specific features. EGAT incorporates edge features explicitly into the attention mechanism, enabling it to model graph data with edge-specific properties. This model is especially efficient in task where edge properties significantly affects the graph features.

Since the EGAT incorporates edge features explicitly, the attention coefficient could be expressed as follow:

$$e_{ij} = a(\mathbf{W}_{node}h_i, \mathbf{W}_{node}h_j, \mathbf{W}_{edge}e_{ij}) \tag{3.11}$$

where e_{ij} express the edge feature vector for edge (i, j) and $\mathbf{W}_{node}$ and $\mathbf{W}_{edge}$ are the learnable weight matrix for node and edge features. Thus, the node features of i which are updated by aggregating features from neighbors could be expressed as:

$$h'_i = \|_{k=1}^{K} \sigma \left(\sum_{j \in N_i} a_{ij}^k \mathbf{W}_{node}^k \cdot h_j \right) \tag{3.12}$$

3.5 Transfer Learning

Transfer learning is a machine learning (ML) technique where a model is trained on one task or dataset and reused or adapted for another related task or dataset. This technique allows the knowledge gained from solving one problem to be applied to a different but related problem. In a neural network, a pre-trained model trained on a large dataset is fine-tuned by a smaller dataset. This process adjusts the pre-trained model to a new task or dataset and requires less data and computation compared to training a model from scratch.

Transfer learning is a valid technique for predicting PPA. In [6], we applied transfer learning to the prediction of PPA for configurations of Rocket CPU core, using a base model trained with the default configuration. The retrained base model is evaluated in different Rocket Core configurations and OpenRisc 1200 with minimum samples. Another transfer learning research from [11], compared two scenarios with different source domains. The base model is trained on the source domain which consists of single or multiple compilers from the previous versions. Then, it is fine-tuned with predecessor or cross-tech scenarios having different distances from the source domain, implying the

closely related domain (predecessor) is better for data reduction when a more distanced domain (cross-tech) is better for generalization. The work in [25] proposes the fast design space exploration (DSE) framework with transfer learning and multi-objective Bayesian optimization by implementing RISC-8 CPU with TSMC 130nm and 65nm technology.

3.5.1 GNN Transfer Learning

GNNs with transfer learning leverage pre-trained GNN models on one graph-related task and fine-tune the learned representations to another task or dataset. In the fine-tuning process, the model uses a smaller dataset compared to the dataset used in the pre-training process. Also, it reduces the training time by starting with pre-trained weights. Several studies [10, 8] investigate GNNs with transfer learning method on natural language dataset.

In [10], the various GNNs have been used to attempt transfer learning on the citation network and Blood-Brain Barrier Penetration (BBBP) physiological dataset. GCN, GraphSage, and Graph Isomorphism Network (GIN) are used to measure the impact of transfer learning with Transfer Ratio, Jumpstart, and Asymptotic Performance.

Figure 3.8 illustrates the method used in [10] to measure the impact of fine-tuning. The Jumpstart and Asymptotic Performance measure the initial and final learned performance of the transfer learning over the base model. The Transfer Ratio measures the total cumulative performance of the transfer learner to the base learner.

Figure 3.9 and 3.10 illustrate the node classification and graph classification result (10 runs) from [10]. The results show the impact on transfer learning by comparing it with base model performance.

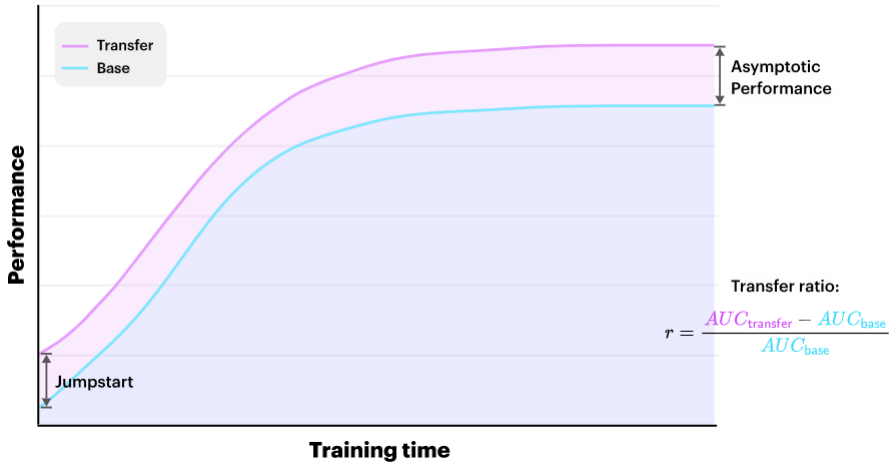


Figure 3.8: Transfer Learning Measurement [10]

Source Task → *MAG-Target* (Spectral Dist.)	Model	Transfer Ratio	Jumpstart	Asymptotic Performance
MAG-Source [Old layer]	GCN	0.042 ± 0.007	**0.228 ± 0.023**	0.003 ± 0.003
	G'SAGE	0.044 ± 0.041	**0.239 ± 0.018**	0.001 ± 0.010
	GIN	**0.183 ± 0.008**	0.226 ± 0.004	**0.024 ± 0.002**
MAG-Source	GCN	0.032 ± 0.011	**0.001 ± 0.002**	0.002 ± 0.003
	G'SAGE	0.046 ± 0.041	0.000 ± 0.001	0.003 ± 0.011
	GIN	**0.089 ± 0.007**	−0.001 ± 0.004	**0.009 ± 0.001**
Arxiv	GCN	0.021 ± 0.007	**0.001 ± 0.001**	**0.008 ± 0.002**
	G'SAGE	0.028 ± 0.040	**0.001 ± 0.001**	**0.007 ± 0.010**
	GIN	**0.048 ± 0.009**	−0.001 ± 0.004	0.005 ± 0.001

Figure 3.9: Transfer Learning Node Classification [10]

Source Task → *HIV-Target*	Model	Transfer Ratio	Jumpstart	Asymptotic Performance
HIV-Source	GCN	**0.065 ± 0.010**	0.148 ± 0.009	**0.031 ± 0.017**
	G'SAGE	0.030 ± 0.006	0.160 ± 0.016	0.011 ± 0.014
	GIN	0.033 ± 0.016	**0.186 ± 0.045**	**0.029 ± 0.040**
BBBP	GCN	0.036 ± 0.012	0.047 ± 0.016	0.026 ± 0.011
	G'SAGE	**0.048 ± 0.008**	**0.072 ± 0.011**	**0.035 ± 0.009**
	GIN	−0.059 ± 0.033	0.026 ± 0.046	−0.081 ± 0.075

Figure 3.10: Transfer Learning Graph Classification [10]

3.5.2 Meta Learning

Meta-learning is a machine learning (ML) technique that develops algorithms capable of learning from multiple tasks or datasets to improve their ability to generalize to new tasks or datasets more effectively and efficiently. One of the meta-learning approaches is using a few-shot learning approach which attempts to learn quickly from a minimum number of samples.

Meta-learning is a few-shot learning approach in which a model can be trained with multiple training algorithms (called learners or tasks) by combining them with a higher-level algorithm, called the meta-learner. This concept is very general and can be applied in many ways.

In Meta-SGD [12], meta-learning was used to solve few-shot learning problems by quickly choosing the initial weights and learning rate of a model. MetaNet [15] quickly mapped training gradients from a base model onto weights with a higher learning rate to solve one-shot learning problems. These works illustrate the promise of meta-learning for few-shot learning scenarios, but they are much more complex and difficult to apply than transfer learning.

Meta-transfer learning (MTL) applies meta-learning to transfer learning to create a simple few-shot learning approach. MTL re-trains the base model with meta-batches, which are a sequence of fast learners. In [18], this approach was used to accelerate the training of an image classifier and prevent overfitting.

3.6 Statistical Method

3.6.1 Relative Entropy

The relative entropy, known as Kullback-Leibler divergence (K-L divergence), is a statistical distance from the model to the target[14]. The K-L divergence measures how far the target is from the model, in units of the natural unit of information (nats). The mathematical representation of K-L divergence as where two probability distributions P and Q defined over the same domain $\mathcal{X}$, the KL divergence from Q to P is defined as follows.:

$$D_{\mathrm{KL}}(P\|Q) = \sum_{x\in\mathcal{X}} P(x)\log\left(\frac{P(x)}{Q(x)}\right) \tag{3.13}$$

3.6.2 Wasserstein Metric

Statistical distance is the distance between two statistical objects. In our case, the statistical object is considered as the distribution of PPA. The Wasserstein distance or earth mover's distance is an estimate of the optimal transport cost between two measures [22]. In other words, if two probability distributions are viewed as masses of earth, the Wasserstein distance is the minimum cost of transporting the mass from one distribution to make it equal to the other.

$$W_1(u,v) = \int_{-\infty}^{\infty} |F_u(x) - F_v(x)|dx \tag{3.14}$$

Equation 3.14 presents the Wasserstein distance calculation for random variables u and v in terms of their cumulative distribution functions F_u, F_v. This equation is valid only if it is performed in one dimension [16], which is sufficient for this simple analysis.

3.6.3 Rectified Linear Unit (ReLU) Activation Function

The Rectified Linear Unit (ReLU) is one of the most commonly used activation functions in modern neural networks. It introduces non-linearity into the model while maintaining computational efficiency. ReLU is defined as a piecewise linear function:

$$\text{ReLU}(x) = \max(0, x) \tag{3.15}$$

For any input $x \in \mathbb{R}$, the function outputs x if $x > 0$, and 0 otherwise. ReLU has become the default activation function in many deep learning architectures due to its simplicity, non-saturating behavior, and its ability to mitigate the vanishing gradient problem that often occurs with sigmoid or tanh activations.

However, ReLU is not without limitations. A known issue is the so-called "dying ReLU" problem, where neurons can become inactive and always output zero, particularly if large negative gradients cause the weights to update such that the neuron no longer activates. Despite this, ReLU remains highly effective and is often combined with techniques such as leaky ReLU or batch normalization to improve robustness and stability during training.

3.6.4 AdamW Optimization Algorithm

AdamW is an extension of the Adam optimization algorithm that incorporates a decoupled weight decay term to improve generalization [13]. Originally proposed to address the incorrect implementation of L2 regularization in standard Adam, AdamW separates the weight decay term from the gradient-based update, ensuring that regularization is applied correctly and consistently.

The standard Adam optimizer updates the parameters θ using estimates of the first and second moments of the gradients:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{3.16}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \tag{3.17}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{3.18}$$

$$\theta_{t+1} = \theta_t - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \tag{3.19}$$

In AdamW, the update rule is modified to include an explicit weight decay term:

$$\theta_{t+1} = \theta_t - \alpha \left(\frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} + \lambda \theta_t \right) \tag{3.20}$$

where λ is the weight decay coefficient and α is the learning rate. By decoupling the regularization from the adaptive gradient update, AdamW achieves better convergence and improved generalization performance, especially in settings where large learning rates or deep architectures are used. It has been widely adopted in training modern deep neural networks, including transformer-based models.

3.6.5 Error Calculation Methods

The common methods to calculate the accuracy of a GNN model are as follows: the mean of mean absolute percentage error (MAPE), mean absolute error (MAE) and R2 score. Each of these metrics offers different insights into model performance. MAPE is useful for percentage-based interpretation, MAE gives an absolute error scale, and R^2 provides a variance-based goodness-of-fit measure. For robust evaluation, especially in highly variable datasets such as those encountered in integrated circuit design, it is common to report multiple metrics in tandem.

Mean Absolute Percentage Error

MAPE measures the average magnitude of the prediction errors as a percentage of the true values. It is defined as:

$$MAPE = \frac{100\%}{N} \sum_{n=1}^{N} \left| \frac{A_n - P_n}{A_n} \right| \tag{3.21}$$

where A_n is the actual value and P_n is the predicted value. MAPE is scale-independent and intuitively interpretable, making it suitable for performance comparison across different datasets. However, it is sensitive to true values A_n, which can result in large percentage errors.

Mean Absolute Error

MAE calculates the average absolute difference between the predicted and true values:

$$MAE = \frac{100\%}{N} \sum_{n=1}^{N} |A_n - P_n| \tag{3.22}$$

where A_n is the actual value and P_n is the predicted value. MAE retains the same units as the target variable, providing a direct interpretation of prediction accuracy. This makes MAE more viable option when actual value and predicted value are near-zero values.

Coefficient of Determination

The R^2 score, or coefficient of determination, measures the proportion of variance in the target variable that is explained by the model. It is defined as:

$$R2 = 1 - \frac{\sum_{n=1}^{N} (A_n - P_n)^2}{\sum_{n=1}^{N} (A_n - A_{mean})^2} \tag{3.23}$$

where A_{mean} is the mean of the actual data A_n. An R^2 score of 1 indicates perfect prediction, whereas a score of 0 implies that the model performs no better than a naive mean predictor. Negative values suggest that the model performs worse than the baseline.

Chapter 4

PPA Prediction with GNN Transfer Learning with OpenABC-D dataset

4.1 Chapter Introduction

Commercial EDA tools such as Synopsys Design Compiler and Cadence Innovus are widely recognized for their accuracy and industrial reliability in performing synthesis, placement, routing, and timing analysis. However, these tools require significant computational resources and long runtimes, especially when applied to large-scale datasets or iterative design space exploration. In contrast, open-source tools such as OpenROAD and OpenSTA offer a lightweight and rapid alternative, enabling faster design turnaround. Despite their reduced accuracy and limited support for advanced technology nodes, these tools are highly suitable for generating large-scale datasets efficiently.

To balance the trade-off between speed and accuracy, this research proposes a transfer learning approach that leverages the scalability of open-source tools for pretraining and the accuracy of commercial tools for fine-tuning. Specifically, we train a graph neural network (GNN) model using a large dataset generated by open-source EDA flows and

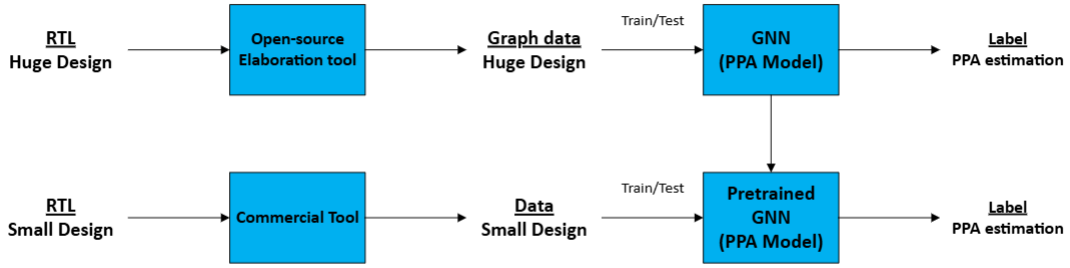


Figure 4.1: Transfer Learning Process

subsequently apply transfer learning to adapt the pretrained model to smaller datasets generated by commercial flows. This enables the model to generalize to high-fidelity PPA predictions while significantly reducing the time and computational cost of data generation and training. Figure 4.1 illustrates the proposed model training and transfer learning pipeline.

Our goal is to achieve a minimum of 95% prediction accuracy on PPA metrics (power, performance, and area) for commercial tool flows, while minimizing the overall time spent generating training data and training the model. This approach demonstrates the viability of combining high-throughput data generation from open-source tools with high-accuracy learning outcomes via domain adaptation.

The remainder of this chapter is organized as follows. Section 4.2 introduces the datasets used for both pretraining and transfer learning. Section 4.3 presents the architecture and configuration of the GNN model. Section 4.4 discusses experimental results and evaluation metrics. Finally, Section 4.5 summarizes key findings and outlines future directions.

4.2 Dataset

This research employs two complementary datasets to support both pretraining and transfer learning of the proposed GNN-based PPA prediction framework. The first dataset, introduced in [3], is derived from an open-source synthesis flow and is used to pretrain the model across a wide range of design structures and synthesis conditions. The second dataset, adapted from [6], is generated using a commercial ASIC flow and serves as the fine-tuning and validation target for transfer learning. Together, these datasets enable the evaluation of both generalization across designs and adaptation across toolchains and abstraction levels.

4.2.1 OpenABC-D Dataset: Large-Scale Pretraining via Open-Source Flow

The OpenABC-D dataset is generated using the open-source EDA tools OpenROAD v1.0 and Yosys v0.9, with ABC used as the backend optimizer. It consists of 27 open-source IP designs, each synthesized under 1,500 distinct synthesis recipes, where each recipe contains 20 logic transformation steps. This results in approximately 30,000 graph-structured samples per IP. The synthesis recipes are randomly sampled using a uniform distribution over all available ABC transformations and saved at intermediate and final optimization stages to capture a diverse range of structural variations. Figure 4.2 illustrates the benchmark suite used in constructing the dataset from [3].

Each resulting circuit is represented as an And-Inverter Graph (AIG), a directed acyclic graph (DAG) composed of two-input AND gates and inversion edges. These AIGs are saved in BENCH format and subsequently converted to GRAPHML using a custom netlist parser. The final dataset includes both graph topology and a rich set of features: two node-level features (gate type and number of inverted inputs) and one binary edge-

IP	Characteristics of Benchmarks						
	PI	PO	N	E	I	D	Function
spi [18]	254	238	4219	8676	5524	35	Serial peripheral interface
i2c[18]	177	128	1169	2466	1188	15	Bidirectional serial bus protocol
ss_pcm[18]	104	90	462	896	434	10	Single slot PCM
usb_phy[18]	132	90	487	1064	513	10	USB PHY 1.1
sasc[18]	135	125	613	1351	788	9	Simple asynch serial controller
wb_dma[18]	828	702	4587	9876	4768	29	Wishbone DMA/Bridge
simple_spi[18]	164	132	930	1992	1084	12	MC68HC11E based SPI interface
pci[18]	3429	3157	19547	42251	25719	29	PCI controller
wb_conmax[18]	2122	2075	47840	97755	42138	24	WISHBONE Conmax
ethernet[18]	10731	10422	67164	144750	86799	34	Ethernet IP core
ac97_ctrl[18]	2339	2137	11464	25065	14326	11	Wishbone ac97
mem_ctrl[18]	1187	962	16307	37146	18092	36	Wishbone mem controller
bp_be[19]	11592	8413	82514	173441	109608	86	Black parrot RISCV processor engine
vga_lcd[18]	17322	17063	105334	227731	141037	23	Wishbone enhanced VGA/LCD controller
des3_area[18]	303	64	4971	10006	4686	30	DES3 encrypt/decrypt
aes[18]	683	529	28925	58379	20494	27	AES (LUT-based)
sha256[20]	1943	1042	15816	32674	18459	76	SHA256 hash
aes_xcrypt[21]	1975	1805	45840	93485	36180	43	AES-128/192/256
aes_secworks[22]	3087	2604	40778	84160	45391	42	AES-128 (simple)
fir[20]	410	351	4558	9467	5696	47	FIR filter
iir[20]	494	441	6978	14397	8596	73	IIR filter
jpeg[18]	4962	4789	114771	234331	146080	40	JPEG encoder
idft[20]	37603	37419	241552	520523	317210	43	Inverse DFT
dft[20]	37597	37417	245046	527509	322206	43	DFT design
tv80[18]	636	361	11328	23017	11653	54	TV80 8-Bit Microprocessor
tiny_rocket[15]	4561	4181	52315	108811	67410	80	32-bit tiny riscv core
fpu[23]	632	409	29623	59655	37142	819	OpenSparc T1 floating point unit
picosoc[23]	11302	10797	82945	176687	107637	43	SoC with PicoRV32 riscv
dynamic_node[15]	2708	2575	18094	38763	23377	33	OpenPiton NoC architecture

Table 1: Open source IP characteristics (unoptimized). Primary Inputs (PI), Primary outputs (PO), 'Nodes (N), Edges (E), 'Inverted edges (I), Netlist Depth (D). Color code: Communication/Bus protocol, Controller, Crypto, DSP, Processor, Processor+control, Control+Communication

Figure 4.2: [3] benchmarks

level feature (original vs. inverted signal). All graphs are processed and stored using the PyTorch Geometric framework for efficient GNN integration. Table 4.1 summarizes the structure and features of each graph.

Table 4.1: Graph Structure

Type	**Feature**	**Type**	**Description**
Node	Node ID	str	Name of the Node ex) clk, reg[1]
	Node type	int	Encoded flag of the node in gate
	# Inverter Predecessor	int	Number of connected inverter as predecessor
Edge	Edge type	int	Encoded flag of the node in gate
Graph	Longest Path	int	Longest path in DAG
	# AND Nodes	int	Total number of AND type nodes
	# PI	int	Number of connected inverter as predecessor
	# PO	int	Number of connected inverter as predecessor
	# NOT Edges	float	Total number of NOT type edges
Type	**Label**	**Type**	**Description**
Graph	Nodes	int	Number of Nodes
	Area	float	Total area
	Critical Path	float	Cell area

Post-synthesis PPA metrics are collected using the NanGate 45nm technology library and the "5K_heavy" wireload model. Figure 4.3 illustrates the distribution of area and delay for the dataset. Notably, the area remains relatively stable across different synthesis recipes, while the delay exhibits significant variability—making it a suitable target for GNN-based learning.

The IPs span a broad range of functions, including bus protocols, cryptographic cores, digital signal processors, and system controllers. Compared to earlier benchmarks such as ISCAS or EPFL, the OpenABC-D designs are larger and more functionally diverse, ensuring coverage of real-world structural patterns (e.g., tree-like, balanced, or skewed topologies). Dataset generation required over 200,000 compute hours using server-grade

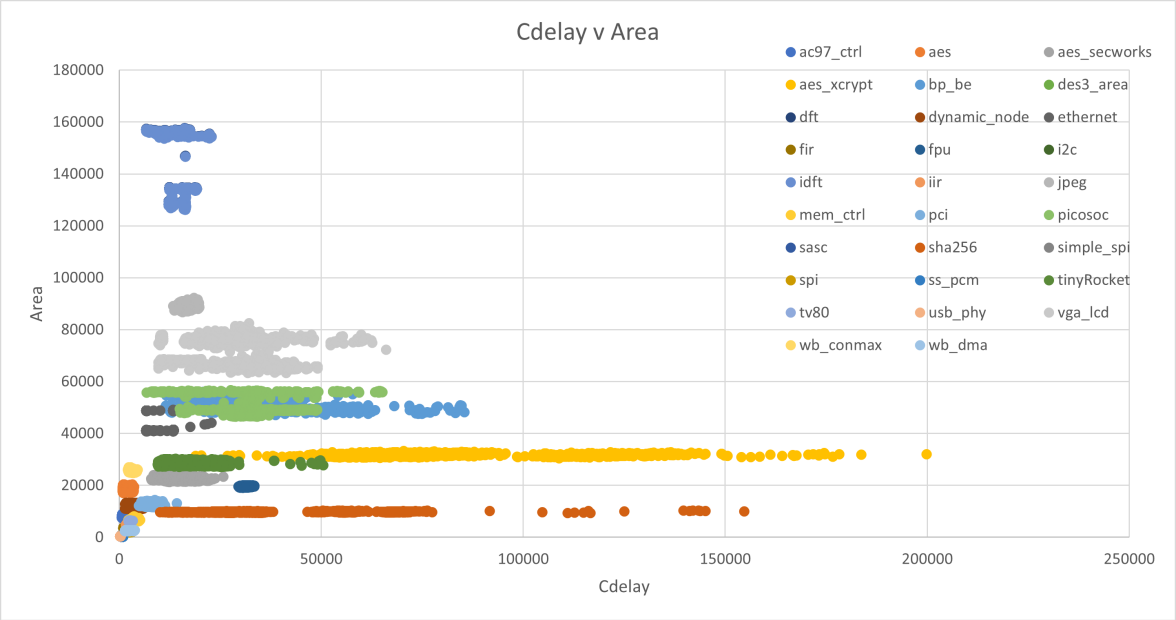


Figure 4.3: [3] Data Distribution

Intel processors, underscoring its scale and realism.

4.2.2 Commercial-Flow Dataset: High-Fidelity Transfer Learning via Synopsys Flow

The second dataset, adapted from [6], is generated using a commercial EDA toolchain comprising Synopsys Design Compiler and IC Compiler II. It targets the GlobalFoundries 22FDX/22FDSOI INVECAS process and provides high-accuracy post-route PPA labels. This dataset serves as the fine-tuning domain for evaluating transfer learning effectiveness.

It includes three widely-used IP blocks:

- **Rocket Core** – a parameterized RISC-V processor,
- **CNN Accelerator** – a deep learning hardware block,

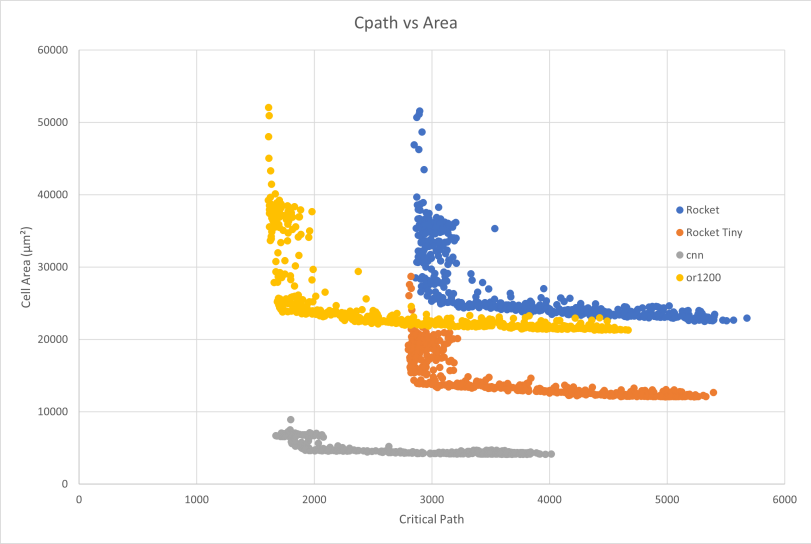


Figure 4.4: [6] Data Distribution

- **OpenRISC 1200 (or1200)** – an open-source RISC processor.

Each design is evaluated across 500 configurations, generated using Latin Hypercube Sampling (LHS) to ensure systematic coverage of the parameter space. The parameters span design aspects such as pipeline stages, ALU and memory subsystems, multiplier variants, and other RTL-level tuning knobs. Every sampled configuration undergoes full synthesis, placement, and routing, producing high-fidelity PPA labels: cell area, critical path delay, total power, and related technology-specific timing metrics.

The model input features consist of structural and synthesis-level descriptors extracted pre-routing (e.g., gate counts, net fanouts, pipeline depth, memory macros), while the post-route PPA results serve as prediction targets. Figure 4.4 illustrates the data distribution of area versus delay in this dataset. In contrast to the OpenABC-D dataset, the commercial dataset exhibits an inverse relationship: area tends to decrease as critical path delay increases, reflecting real trade-offs enforced by the backend optimization engine.

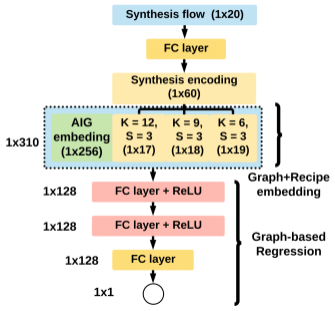


Figure 4.5: Graph Neural Network architecture from OpenABC-D [3].

4.2.3 Dataset Summary

By combining these two datasets—one offering breadth (scale and structural diversity) and the other offering depth (accuracy and industry realism)—this research enables robust evaluation of transfer learning strategies in GNN-based PPA prediction. The OpenABC-D dataset serves as a high-throughput, low-cost source for model pretraining, while the commercial dataset provides high-quality targets for fine-tuning and benchmarking.

4.3 Model Structure and Transfer Learning Strategy

The Graph Neural Network (GNN) architecture used in this research is based on the model introduced in the OpenABC-D framework [3], and is illustrated in Figure 4.5. This model is trained on a large-scale dataset comprising 27 open-source IP cores, each synthesized using 1,000 distinct synthesis recipes generated through the Yosys-ABC flow, as integrated within the OpenROAD toolchain. These recipes apply a wide range of structural transformations, producing a diverse population of and-inverter graphs (AIGs) that enhance the generalization capacity of the learned graph representations.

Each input graph represents a circuit's logic structure in AIG form, where nodes en-

code logic gate types (e.g., 2-input AND, inverter) and edges represent signal connections, including signal polarity. The GNN architecture first applies an AIG embedding layer, projecting the input features into a high-dimensional latent space. This is followed by multiple message-passing layers (based on either graph convolution or graph attention mechanisms), which iteratively refine node representations by aggregating information from their neighbors. The final prediction is produced through a stack of fully connected (FC) layers, which map the global graph embedding to target metrics such as area and critical path delay. The original OpenABC-D model is trained using a mean squared error (MSE) loss function for regression.

In this research, we adapt the pretrained GNN to commercial ASIC designs generated using the Synopsys ICC2 flow on the GlobalFoundries 22FDX process. To enable this domain adaptation, we investigate three transfer learning strategies:

1. **1-layer fine-tuning:** Only the final FC layer is retrained, while all earlier layers (including message-passing and embedding layers) are frozen. This strategy assesses how transferable the pretrained graph features are to a new technology stack.

2. **2-layer fine-tuning:** The last two FC layers are retrained, offering additional representational flexibility while preserving the core learned features.

3. **Embedding-frozen retraining (AIG_embed):** The initial AIG embedding layer (of size 1×256) is frozen, while all downstream layers are retrained. This configuration enables structural adaptation to the target domain while maintaining the semantic integrity of the base graph encoding.

These strategies are designed to test how well models trained on open-source data generalize to more complex commercial designs, and whether transfer learning can reduce the volume of required labeled data without sacrificing predictive accuracy.

4.4 Experiments

To evaluate the transferability of the OpenABC-D GNN, we conducted experiments using two datasets: the original open-source designs from [3] and a commercial dataset generated using the GlobalFoundries 22FDX/22FDSOI technology with Synopsys ICC2 implementation. Table 4.2 and Table 4.3 summarize the model's performance in predicting area and critical path delay (cdelay), respectively, under different transfer learning configurations.

4.4.1 Area Prediction Results

Table 4.2: Transfer learning performance on area prediction.

Phase	Configuration	Dataset	Train Samples	Test Samples	Area MAPE
Evaluate	Baseline	GlobalFoundries	-	7	0.6466
	Baseline	OpenABC-D [3]	-	25	0.3727
Retrain	1-layer	GlobalFoundries	20.25	6.75	**0.2780**
	2-layer	GlobalFoundries	20.25	6.75	0.3065
	AIG_embed	GlobalFoundries	20.25	6.75	0.3175

As shown in Table 4.2, the model trained solely on open-source data yields moderate performance on the GlobalFoundries dataset (MAPE = 0.6466). However, retraining just the final FC layer significantly reduces the error (to 0.2780), demonstrating the potential for efficient domain adaptation with minimal model modification. Notably, all transfer configurations substantially outperform the baseline, especially for area prediction.

4.4.2 Critical Path Delay (cdelay) Prediction Results

The delay prediction task proved more challenging. While retraining improves performance over baseline evaluation (reducing MAPE from 4.8062 to 1.7296 in the best case),

Table 4.3: Transfer learning performance on critical path delay prediction.

Phase	Configuration	Dataset	Train Samples	Test Samples	Cdelay MAPE
Evaluate	Baseline	GlobalFoundries	-	7	4.8062
	Baseline	OpenABC-D [3]	-	25	7.989
Retrain	1-layer	GlobalFoundries	20.25	6.75	**1.7296**
	2-layer	GlobalFoundries	20.25	6.75	3.8153
	AIG_embed	GlobalFoundries	20.25	6.75	2.184

the accuracy remains lower than desired. Interestingly, the 2-layer retraining configuration underperforms compared to the simpler 1-layer setup, suggesting that overfitting or instability may arise when modifying too many parameters with limited training data.

4.5 Conclusion and Future Work

The results from this study highlight both the potential and the limitations of applying transfer learning to GNN-based PPA prediction. In the case of area estimation, the pretrained OpenABC-D model demonstrated strong adaptability when fine-tuned on a relatively small number of samples from a commercial dataset. With only modest retraining (e.g., 1-layer adaptation), we achieved a nearly 57% reduction in error compared to the original evaluation result.

However, the critical path delay predictions were less stable. Although some transfer configurations showed marked improvements, the overall accuracy remained limited. This disparity suggests that the timing characteristics of commercial designs—especially those involving proprietary PDKs and tool flows—are less amenable to knowledge transfer from open-source data.

Moving forward, several improvements are planned. First, we aim to incorporate additional timing-sensitive features (e.g., slew, fanout, parasitic estimates) into the graph data features to enhance its predictive capacity. Second, we will explore the statistical

method (e.g., Relative entropy or Wasserstein metric) to measure the distance from a source domain to a target domain, to measure the fine-tuning performance of the GNN model. Finally, we plan to explore various domain adaptation methods other than transfer learning (e.g., Maximum mean discrepancy or Wasserstein distance minimization) to improve the GNNs' fine-tuning performance.

Chapter 5

Accurate Few-shot PPA Prediction with Meta-Transfer Learning

5.1 Chapter Introduction

The machine learning mechanism for predicting PPA in the early stage of designing a System-on-Chip (SoC) is critical considering the time it takes to process a chip. This could also prevent any possible flaw that will cause one to miss any of the PPA targets. Thus, we propose machine learning to initially generate and train the model and retrain the model using transfer learning to optimize the model to a specific design.

Machine learning with transfer learning is one of the methods for predicting the PPA. This method has been found useful in a number of researches [6, 9, 11]. The transfer learning method optimizes the trained model for a similar task. For the machine learning method, this process involves retraining specific layers or neurons with a similar design. This method requires fewer QoR, or samples, for retraining the model. Since this method is performed in different fields, the performance is varied depending on which field the model is being executed to predict the PPA.

However, the PPA model retrained with the transfer learning method could only predict the PPA of the design that is related to the design that is used when initially training the PPA model. Initial training of the PPA model consumes a large amount of time and the limited spectrum of a model to certain designs suggests a new model for the PPA prediction. Thus, in this paper, we discuss the meta-transfer learning model as a new model that could predict a design in which statistical distance is comparable far from the design used to initially train the model, using a minimum number of QoR as possible.

In the rest of the chapter, we organize as follows: we introduce the datasets which we will train the model with in section 5.2. In section 5.3., we present the model structure to be used in the research. Section 5.4 presents the result of the evaluated model and provides a conclusion in section 5.5.

5.2 Datasets

This work uses the same dataset from [6]. As in our previous work, we train a base model on the Rocket default configuration (rocket) and transfer the model to the tiny configuration (rocket-tiny). In this work, we also transfer the model to two designs that differ from the Rocket core more significantly: a convolutional neural network (CNN) accelerator and the OpenRisc 1200 (or1200) CPU. The dataset contains RTL code and synthesis results from a commercial 22 nm process and implementation flow. So as not to reveal trade-secret information, the data were obfuscated by randomly scaling and shifting each column by a different amount.

Table 5.1: Features and Labels Scale Factor

Type	Names	Scale Factor
Feature	Tclk	10000
	Fanout	50
	MaxTran	500
	Uncertainty	200
Target	Area	50000
	Cpath	10000
	Csw	200
	Pleak	0.003

5.2.1 Features and Labels

Our dataset has four features, which are RTL-synthesis constraints used by the implementation flow: clock period (Tclk), maximum fan-out at a logic node (Fanout), maximum transition time at a logic node (MaxTran), and clock uncertainty. We created 500 samples by varying these four constraints using a Latin hypercube and executing the synthesis flow. From the synthesis PPA results, we derived four more values for our label/target vectors: cell-area, critical path delay (Cpath), switched capacitance (Csw), and leakage power (Pleak). The features are normalized by dividing with scale factor represented in Table 5.1.

The difficulty of transferring a model from one design to another depends on how different the QoR distributions are. We illustrate the difference in the area metric between our designs with the histograms in Figure 5.1, showing 500 samples for each design. As can be seen the rocket and or1200 designs have similar area, while the rocket-tiny design is smaller, and the CNN design is smaller still. It is desirable to have a simple measure of the difference between two designs that can capture the difficulty of transferring a PPA model from one to another.

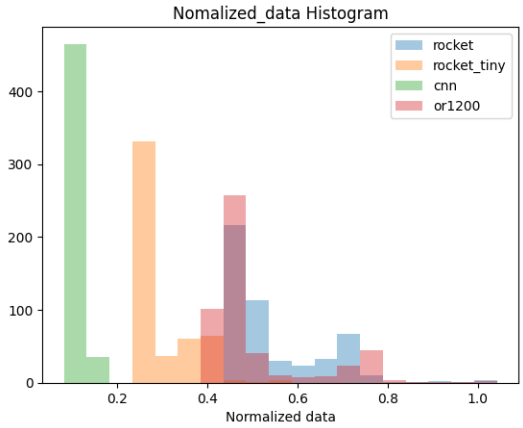


Figure 5.1: Histogram on target Area

5.2.2 Relative Entropy and Wasserstein Distance

Here we examine possible measure of statistical distance between the datasets. The relative entropy, known as Kullback-Leibler divergence (K-L divergence), is a commonly used statistical distance from a model to a target, in units of the natural unit of information (nats) [14]. K-L Divergence is defined for two random variables when their probability distributions are absolutely continuous. Probability distributions for continuous-valued samples are not absolutely continuous in general, and so additional methods such as kernel-density estimation must be applied to create probability density functions to enable K-L divergence calculation. We wanted to use a simpler measure that was easier to apply and gave a repeatable value for any dataset.

The Wasserstein distance or earth mover's distance is an estimate of the optimal transport cost between two measures [22]. In other words, if two probability distributions are viewed as masses of earth, the Wasserstein distance is the minimum cost of transporting the mass from one distribution to make it equal to the other.

$$W_1(u,v) = \int_{-\infty}^{\infty} |F_u(x) - F_v(x)|dx \tag{5.1}$$

Equation 5.1 presents the Wasserstein distance calculation for random variables u and v in terms of their cumulative distribution functions F_u, F_v. This equation is valid only if it is performed in one dimension [16], which is sufficient for this simple analysis.

Table 5.2 shows the statistical distance between the rocket default design, which we use to train our base model, and the other designs. The results indicate for all four designs, CNN design has the most distance from the rocket default configuration design followed by or1200 and rocket tiny designs.

Table 5.2: Wasserstein Distance between rocket design

Design	**Area**	**Critical Path**	**Switched Capacitance**	**Leakage Power**
rocket	0	0	0	0
rocket tiny	0.253	0.0074	0.1694	0.1536
cnn	0.4542	0.0816	0.3402	0.3092
or1200	0.0452	0.1055	0.0248	0.0431

5.3 Model Structure

In this section, we introduce the base model structure generated with PyTorch. The base model is trained and tested using two main algorithms: Outer Optimization and Inner Optimization.

5.3.1 Base Model

The base model is a neural network model structure with four linear layers the size of [128,256,512,256,64] with an input dimension of 6 and an output dimension of 1. The base model is trained which sets different weight paths for multiple targets. The activation function, also known as the transfer function, is used to determine the output of the

neural network. The base model uses the rectifier linear unit (ReLU) function as an activation function for the entire layers.

In the pre-training phase, the base model is trained with 300 samples from the rocket design and tested with 200 samples from the rocket design. The 1 will be used to train and test the model. This pre-trained model is saved to be used in the transfer learning and meta-transfer learning phase. Table 5.3 illustrates the results of the pre-training phase. In comparison with the result from [6] where the same pre-training has been executed with 400 training samples, illustrates similar MAPE for every target.

5.3.2 Outer Optimizations

The outer optimization is used in pre-training the base model and retraining the base model in transfer learning. Transfer learning is an optimization method for a problem similar to the base model. The weight and bias are stored in the base model which implies that for PPA approximation, certain numbers of layers could be shared to approximate the PPA of a different design. Considering the work in [6], the last three layers of an NN will be retrained and evaluated.

Algorithm 1 shows the five-step process of the outer optimization: Data load, Data normalization, Model load, Model train, and model test. The data is loaded by using the PyTorch method and normalized by the SCALEFACTOR that we set for each target variable. If the model is in the transfer learning or meta-transfer learning stage, the pre-trained model is loaded and the designated layers are frozen. The model starts to forward propagate and depending on whether the model uses the meta-transfer learning or not, goes through the training phase to get the prediction to the input. Then, it compares the prediction with the actual result to get the loss function. The loss function is back propagated for optimization. This process continues until the validation loss reaches its

lowest point.

5.3.3 Inner Optimization

The inner optimization is used in the process of meta-transfer training of the model. The meta-transfer learning method uses both the transfer learning method and the meta-batch method to optimize the model [18]. The process is divided into two parts: Inner Optimization and Outer Optimization mentioned in Algorithm 1.

In Algorithm 2, the single data from the datasets are used multiple times using the feedback loop. After loading the data, it is used to forward propagate to the base model. With the prediction and the actual value, the loss function is computed. Then, using the gradient of the transfer learning layer parameters and the loss function, we generate the weights and biases for the transfer learning layers. These weights and biases are feedback to the base model.

5.4 Experiments

In this section, we evaluate the model from Section 5.3 and 5.2. To explore few-shot learning performance, we used shuffle-split cross-validation, randomly using 5 re-training samples and 495 validation samples. We found that 50 splits were generally enough to get a MAPE value that was repeatable down to 0.01%. Table 5.3,5.4,5.6,5.5 shows the mean of denormalized MAPE for four target vectors. For comparison between the model, the total epochs executed is set to 5000.

For regularization, the AdamW algorithm from the PyTorch is used with a learning rate of 0.001 and weight-decay of 2.

Algorithm 1 Outer Optimization

function OUT-OPT($X, Y,$) ▷ X: knobs, Y: PPA
  $n_X \leftarrow$ numFeatures(X) ▷ Count number of features
  $i \leftarrow 0$
  while $i \leq n_X$ **do** ▷ Normalize X
    $X \leftarrow X$/SCALEFACTOR(X)
    $i \leftarrow i + 1$
  end while
  $n_Y \leftarrow$ numFeatures(Y)
  $i \leftarrow 0$
  while $i \leq n_Y$ **do** ▷ Normalize Y
    $Y \leftarrow X$/SCALEFACTOR(Y)
    $i \leftarrow i + 1$
  end while
  $m \leftarrow$ initializeModel()
  if $Transfer$ or $MetaTransfer$ **then**
    $m \leftarrow$ loadModel($m_{pre-trained}$) ▷ Load pre-trained model
  end if
  $X_{train}, X_{test}, Y_{train}, Y_{test} \leftarrow$ dataSplit(X,Y)
  $n_{train} \leftarrow$ countData(X_{train})
  $n_{test} \leftarrow$ countData(X_{test})
  while min(validlossList) or $length$(validlossList)=0 **do**
    if $MetaTransfer$ **then**
      $e_{train} \leftarrow$ IN-OPT(m, X_{train}, Y_{train})
    else
      $e_{train} \leftarrow \sum_{j \in 0,\ldots,n_{train}\}}$loss($m(X_{train,j}, Y_{train,j})$
    end if
    $m \leftarrow$ backpropagate(∇e_{train},m)
    if $MetaTransfer$ **then**
      $e_{valid} \leftarrow$ IN-OPT(m, X_{train}, Y_{train})
    else
      $e_{valid} \leftarrow \sum_{j \in 0,\ldots,n_{train}\}}$loss($m(X_{train,j}, Y_{train,j})$
    end if
    validlossList.insert(e_{valid})
  end while
  $e_{test} \leftarrow \sum_{j \in 0,\ldots,n_{test}\}}$loss($m(X_{test,j}, Y_{test,j})$
end function

Algorithm 2 Inner Optimization

function IN-OPT(m, X, Y) ▷ m: model, X: knobs, Y: PPA
  $i \leftarrow 0$
  $n \leftarrow$ countData(X)
  for $j = 0, 1, ..., n$ **do**
    while num_{steps} **do**
      $e_{in} \leftarrow$loss($m(X_j, Y_j)$)
      grad $\leftarrow$ sumGrad(e_{in},m) ▷ Computes and returns the sum of gradients of loss with respect to model m gradient enabled layers
      weight $\leftarrow$ ▷
      $m.weight(grad = true) \leftarrow$ weight ▷ Replace model's gradient true layer with weight
    end while
    $e \leftarrow e+$ loss($m(X_{train,j}, Y_{train,j})$
  end for
  return e
  Use Gradient descent to compute weights and bias for transfer layers
  Feedback transfer layers' weights and bias to base model
end function

Table 5.3: Pre-trained Base model evaluation

Design	Training samples	Test samples	Outer Opt	MAPE	area MAPE	cpath MAPE	Csw MAPE	Pleak MAPE
rocket	300	200	1500	0.022	0.0182	0.0105	0.0272	0.0321

5.4.1 Base Model

The base model is trained with Rocket Core Configuration design. From the total of 500 samples, 300 samples are used to train the model, and 200 samples are used to test the model. Table 5.3 illustrates the results of 1500 outer optimization. The mean of denormalized MAPE on four target vector illustrate less than 5% error rate. This is similar to the base model result that [6] has when it has been trained with 400 samples.

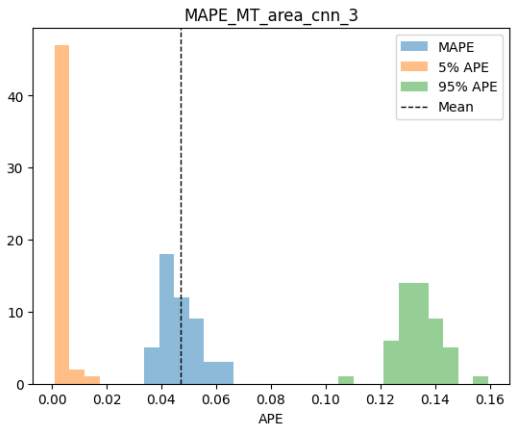


Figure 5.2: MAPE of 50 cross-validations on the area of CNN design

5.4.2 Meta-transfer learning

The 50 cross-validations using shuffle split are evaluated for meta-transfer learning. Figure 5.2 illustrates the histogram of meta-transfer learning for CNN design with their target as an area with their confidence interval and the mean of the MAPE. We could see in the figure that the results are symmetrical which implies the 50 cross-validations are performed as planned. In Figure 5.3, we compare the area histogram of three designs which is symmetrical and overlaps for having low MAPE. However, for Figure 5.4, the histograms of the three designs do not overlap which implies CNN design has higher leak power MAPE.

Table 5.4 illustrates the meta-transfer learning method evaluation results using 5 samples and 495 test samples. The total iteration was 25000 which costs time to simulate and may have resulted in overfitting. However, the denormalized MAPE for all four vectors became much lower. Especially when retraining the base model with CNN design for it shows almost no similarity compared with rocket core default configuration, the error rate decreased drastically, especially on the power leakage vector. Thus, the ablation study is required to analyze the effect of the meta-transfer learning method which differentiates

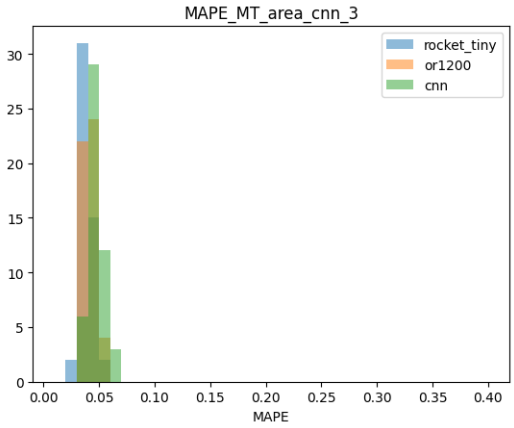


Figure 5.3: MAPE of 50 cross-validations on the area of three designs

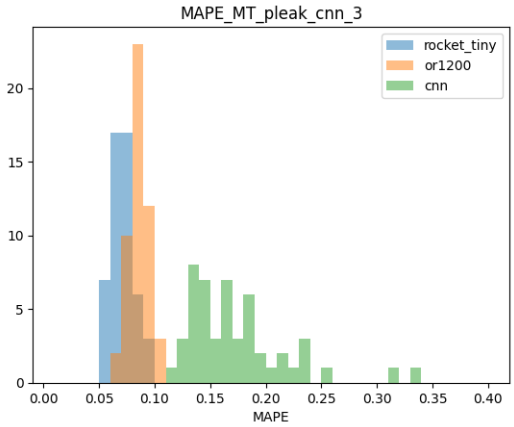


Figure 5.4: MAPE of 50 cross-validations on leak power of three designs

Table 5.4: Meta-transfer learning method evaluation by resetting layers

Method	Design	Inner Opt	Outer Opt	MAPE	area MAPE	cpath MAPE	Csw MAPE	Pleak MAPE
Transfer Learning	rocket tiny	0	25,000	0.0452	0.0364	0.0305	-	0.0754
Meta-transfer Learning	rocket tiny	50	500	0.0522	0.0382	0.0522	0.0472	0.0710
	cnn			0.0846	0.04702	0.0509	0.0665	0.1741
	or1200			0.0635	0.0416	0.0909	0.0357	0.0859

from the original transfer learning method.

5.4.3 Ablation Study

The ablation study has been processed to investigate the positive effect the meta-batch method had on the model. In Table 5.5 and 5.6, the total iterations have been set to 5000 with the difference being resetting and not resetting the layers for retraining. We found the Csw and Pleak data for the cnn design to be unrealistic, suggesting that there was a problem in their collection. This issue is apparent in the large MAPE values for these labels in Table 5.4. To avoid confusion and get a clearer picture of the benefits of the MTL approach, we omitted these values in Tables 5.5 and 5.6.

Comparing the transfer learning method with the meta-transfer learning method, the meta-transfer learning method reduced denormalized MAPE by 2.4% for not resetting the layers. On the contrary, the ablation study by resetting the layers showed transfer learning having better MAPE by 1.1%. Also, in order to explore the statistical distance between the designs, we compared the denormalized MAPE for each design. For designs with close statistical distance to the rocket design which has been used to train the base model, such as rocket tiny and or1200, both the transfer learning method and meta-transfer learning method showed low denormalized MAPE when the weights have not been reset as shown in Table 5.5. However, when the CNN accelerator IP block design is used to retrain the model, both methods showed low denormalized MAPE when the

Table 5.5: Model evaluation for ablation study by not resetting the layers

Method	Design	Inner Opt	Outer Opt	MAPE	area MAPE	cpath MAPE	Csw MAPE	Pleak MAPE
Transfer Learning	rocket tiny	0	5000	0.0485	0.0379	0.0422	0.04626	0.0678
	cnn			0.2054	0.0606	0.0806	-	-
	or1200			0.0772	0.0611	0.0733	0.0686	0.1059
Meta-transfer Learning	rocket tiny	10	500	0.0296	0.021	0.01751	0.0296	0.0505
	cnn			0.2348	0.0418	0.0520	-	-
	or1200			0.0482	0.0442	0.0225	0.036	0.0903

Table 5.6: Model evaluation for ablation study by resetting the layers

Method	Design	Inner Opt	Outer Opt	MAPE	area MAPE	cpath MAPE	Csw MAPE	Pleak MAPE
Transfer Learning	rocket tiny	0	5000	0.0606	0.045	0.0599	0.0544	0.0833
	cnn			0.1746	0.0729	0.1028	-	-
	or1200			0.0895	0.0592	0.1155	0.0525	0.1307
Meta-transfer Learning	rocket tiny	10	500	0.0814	0.0632	0.0848	0.0626	0.1152
	cnn			0.1308	0.0695	0.0862	-	-
	or1200			0.1010	0.0789	0.1115	0.0604	0.1532

weights have been reset as shown in Table 5.6.

5.5 Conclusion and Future Work

We modeled and compared meta-transfer learning and the transfer learning methods for the PPA modeling, which showed the possibility of PPA approximation on a wide range of designs with minimum QoR. The decrease in MAPE implies that under a specific condition, the model could be improved to predict the PPA of a design with various statistical distances from the rocket design. Thus, it reveals the potential for modeling the PPA with a few QoRs.

Even though we have shown the possibility of a meta-transfer learning method for a wide range of designs, the mean of denormalized MAPE is not within the 5% range requirement. We work to find appropriate total iterations and inner optimization iterations. Additionally, the designs that were experimented on contain different corners. We were only able to experiment on one of them due to a time shortage. Therefore, we will

research what condition is needed to generate the appropriate model depending on the statistical distance between the initial design and the retraining design.

Chapter 6

Critical Path Delay Prediction for Evolving Netlists

6.1 Chapter Introduction

The primary objective of this research is to develop a predictive model for power, performance, and area (PPA) in integrated circuit (IC) designs exhibiting significant structural variation. The proposed approach combines graph neural networks (GNNs) with transfer learning to enable accurate estimation of PPA metrics across diverse design spaces. While earlier studies have explored PPA prediction using traditional neural networks (NNs), these models often suffer from limited generalization. Specifically, many omit key stages of the electronic design automation (EDA) process—such as placement and routing—restricting their applicability to small or partially optimized designs. Consequently, their predictive capabilities diminish when extended to more complex or heterogeneous circuits. In this work, we target the prediction of post-route PPA metrics using only pre-floorplan information, as provided in the dataset from [17]. Accurate prediction at this early stage of the EDA flow has the potential to significantly accelerate design-space ex-

ploration by identifying suboptimal candidates before incurring the high computational cost of full-flow implementation.

Although recent efforts have made progress in this direction, their success remains limited. Figure 6.1 illustrates the state-of-the-art for EDA stages. For instance, [19] employs a GNN model to predict post-synthesis timing from RTL-level representations, reporting an R^2 score of 0.82 and a maximum error of 3.79 ns—results obtained on relatively small-to-medium logic modules. Similarly, [7] introduces a timing-engine–inspired GNN that predicts post-routing slack using only post-placement data. Their model, trained on 21 designs synthesized and routed via OpenROAD, achieves an average R^2 score of 0.9493 on training circuits and 0.8957 on unseen test circuits, while providing over 2,600$\times$ speedup compared to the full routing and static timing analysis (STA) flow. In contrast, [1] leverages a large language model (LLM) to predict post-route PPA from pre-floorplan Verilog modules, achieving performance in the range of 50–70% accuracy at a 10% threshold and 70–80% at a 20% threshold. While promising, these results still fall short of the precision and generalization needed for high-confidence, early-stage evaluation in large-scale design scenarios.

Building upon our prior work in transfer learning for NN-based PPA prediction, we observed notable limitations in modeling critical path delay. A likely cause of this limitation is the evolution of the netlist across EDA stages due to successive optimization passes. To overcome these challenges, we propose the use of GNNs, which are better suited to capture the inherent structural properties of circuit netlists. Graph-structured representations provide a natural and scalable way to encode netlists of varying size and topology for use with GNNs . Also, GNNs are particularly suitable for tracking EDA-induced netlist changes, since message passing encodes dependencies along signal paths and aggregates them hierarchically, mirroring transformations in timing and connectivity across stages. By applying transfer learning to GNN models, our goal is to achieve improved

generalization and predictive performance across a wide range of design configurations, with a target accuracy of approximately 90%.

To further enhance transferability, we introduce a dataset selection strategy based on the Wasserstein distance (also known as earth mover's distance), which quantifies the statistical dissimilarity between training and test sets. This metric enables us to identify and control for distributional shifts across datasets, thereby improving model robustness and mitigating negative transfer. By estimating the optimal transport cost between design distributions, the Wasserstein distance provides a principled means to assess dataset diversity and its impact on prediction accuracy.

The remainder of this chapter is organized as follows. Section 6.2 describes the datasets used for pretraining and transfer. Section 6.3 presents the GNN architecture and transfer learning methodology. Experimental results and evaluation are provided in Section 6.4, followed by concluding remarks in Section 6.5.

6.2 Dataset

This research utilizes and compares three datasets to investigate the scalability and transferability of GNNs for predicting critical path delay in integrated circuit (IC) designs. These datasets are:

- An RTL-level dataset from [19], which models timing using intermediate RTL graph representations.
- A large-scale, post-route dataset from [17], following the EDA-schema property graph standard.
- A self-curated dataset derived from the EDA-schema format and converted into PyTorch Geometric-compatible format for GNN training.

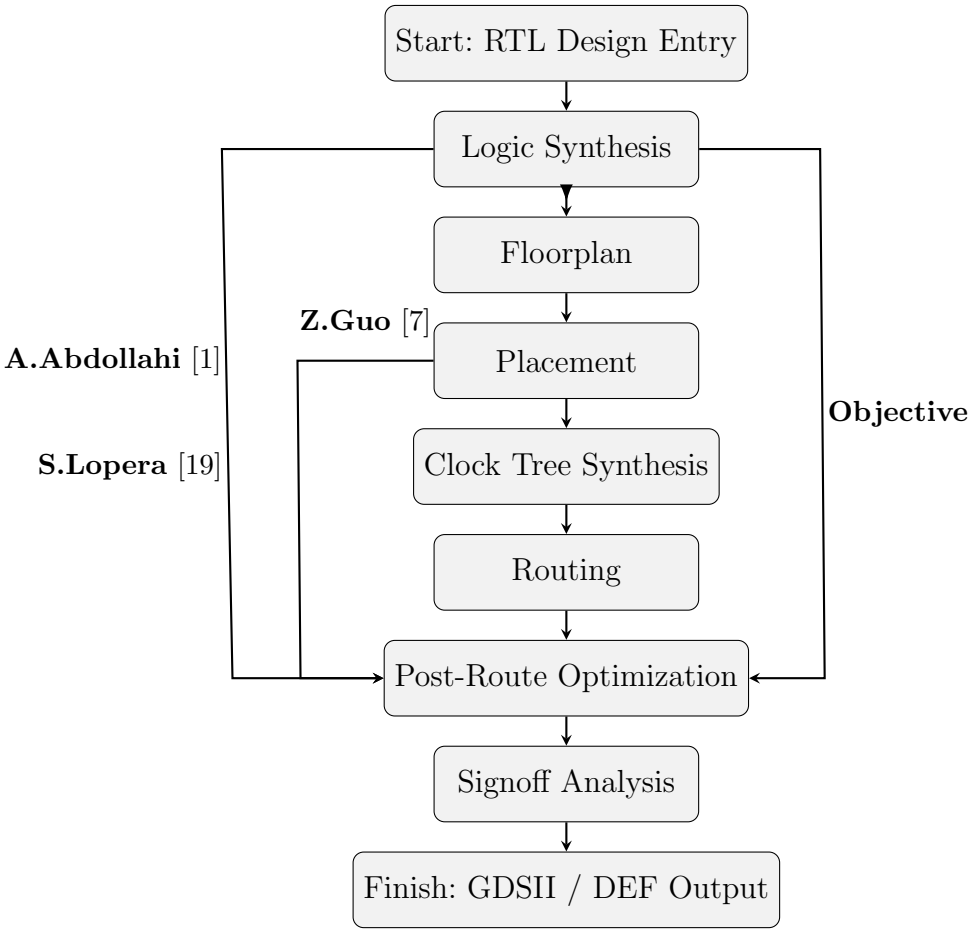


Figure 6.1: EDA Stage Algorithms with Predictions

The datasets vary in abstraction level, complexity, and feature richness, enabling an in-depth comparison of GNN performance across early-stage and signoff-stage design representations.

6.2.1 Features and Labels

We evaluate the generalization capacity of the GNN architectures originally proposed in [19] by scaling them to much larger designs using the dataset from [17]. Figure 6.2a and Figure 6.2b visually compare the relative circuit complexity of the datasets, showing

significant increases in I/O count, logic depth, and node connectivity in the designs from [17].

50 Hardware Adders	# PI	# PO	# Nodes	# Edges	Depth
Brent-Kung 4-bit	2	2	124	148	20
Kogge-Stone 8-bit	2	2	227	272	36
Sklansky 16-bit	2	2	704	881	72
...	...	...	...	...	...
Ripple Carry 32-bit	2	2	1221	1427	296
Vect. Brent-Kung 64-bit	2	2	2851	3495	54
6 ISCAS'85	**# PI**	**# PO**	**# Nodes**	**# Edges**	**Depth**
c17	5	2	52	58	18
c432	36	7	1294	1459	106
...	...	...	...	...	...
c1355	41	32	1586	1802	66
c1908	33	25	2246	2563	92

(a) [19] benchmarks

Circuits	No. of inputs	No. of outputs	Median no. of gates	No. of flip-flops	Median no. of nets	Median no. of timing paths
ac97_ctrl	84	48	13124	2211	6153	2827
aes_core	259	129	20884	530	13673	1195
des3_area	240	64	3977	64	2347	116
des3_perf	234	64	100455	8808	56818	16287
ethernet	96	115	69541	10544	35649	20220
fpu	70	40	23228	643	14370	1669
i2c	19	14	1053	129	485	196
mem_ctrl	115	152	9431	1051	5081	3422
pci	162	207	21612	3220	10769	3770
sasc	16	12	792	118	340	180
simple_spi	16	12	1007	131	473	5311
spi	47	45	3091	229	1631	493
ss_pcm	19	9	640	87	294	109
systemcaes	260	129	9565	670	5630	1436
systemcdes	132	65	2722	190	1343	446
tv80	14	32	7497	364	4620	3008
usb_funct	128	121	14792	1740	7963	159
usb_phy	15	18	674	108	275	141
vga_lcd	89	109	106047	17059	52511	42493
wb_dma	217	215	4241	521	1996	1136
Total						

(b) [17] benchmarks

Figure 6.2: Benchmark Comparison

RTL Dataset (Lopera et al.)

The dataset introduced by [19] uses register-transfer level (RTL) intermediate representations extracted prior to logic synthesis. These RTL designs are converted into graph structures where nodes correspond to RTL operations (e.g., registers, ALUs, control elements), and edges represent control or data dependencies. This representation allows

for early-stage prediction of timing characteristics such as maximum arrival time without requiring physical implementation. Ground-truth timing labels are generated through post-synthesis static timing analysis (STA). The dataset comprises small-to-medium logic modules synthesized and timed using the OpenROAD flow. While the dataset enables fast evaluation, its abstraction limits its generalization to physically optimized large-scale designs.

EDA-schema Dataset (Shrestha et al.)

The dataset from [17] follows the EDA-schema format, a property-graph-based data model designed for digital design automation. It is generated from 20 benchmark circuits across 48 design iterations each, resulting in a total of 960 fully-implemented physical designs. All designs are synthesized, placed, and routed using the OpenROAD toolchain with the SkyWater 130nm PDK. The circuits include large, open-source RTL designs processed through full EDA flow stages: post-placement, post-CTS, and post-routing.

The dataset was originally stored in MongoDB JSON format, capturing heterogeneous graph entities such as gate-level instances, pins, nets, and timing paths. Due to the query latency of MongoDB and the need for efficient ML processing, we converted the data into PyTorch Geometric-compatible format by parsing the JSON records into GraphML and then transforming them into PyG data objects.

6.2.2 Graph Structure and Format

Pin-level Graph: In the pin-to-node graph, each node represents a standard-cell pin or port, enriched with instance, fan-in/fan-out, and type information. Edge attributes include slew values. Labels remain consistent with gate-level representations. The formulation of this graph structure is the same data structure as [19], [17], and [7], but with

different node, edge, and graph features.

Table 6.1 summarizes the node-, edge-, and graph-level features extracted from the EDA-schema dataset in **Pin-level Graph** format. Collectively, these features capture logical connectivity, selected physical attributes, and the timing constraints used in the flow.

Node features. The Node ID records the pin identifier (e.g., A, B, X, Y, PI, PO), which is encoded as a categorical flag spanning 66 pin types in Flag. The Node type encodes the standard-cell class as a categorical flag on 434 library cells. The Instance field stores the cell instance identifier (e.g., `_123461_`) as it appears at the RTL/gate level. The #Inputs and #Outputs record the fan-in and fan-out counts, respectively, thereby characterizing local netlist connectivity.

Edge features. The Delay attribute reports rise/fall timing between connected pins. For intra-cell arcs (fan-in $\rightarrow$ fan-out within a cell), this quantity corresponds to the cell arc delay; for interconnect edges (e.g., PI $\rightarrow$ fan-in or fan-out $\rightarrow$ PO), it reflects the pin transition (slew). All edge-level timing values are extracted at the post-synthesis stage using OpenSTA.

Graph features. Graph-level attributes encode the global constraints and settings used to generate each design point (e.g., clock period, aspect ratio, maximum utilization, maximum skew, maximum fan-out, and maximum clock-network capacitance), as well as summary indicators such as the longest path length.

Labels (targets). As shown in Table 6.1, edge-level labels provide post-route rise/-fall timing between pins, while graph-level labels report PPA metrics—total area, worst (critical-path) arrival time, and total leakage power—at four implementation stages: floor-plan, placement(place), clock-tree synthesis (CTS), and routing(route).

To better understand the structural and functional differences among the datasets used in this work, Table 6.2 presents a side-by-side comparison across key characteristics,

Table 6.1: Pin-level Graph Structure

Type	Feature	Type	Description
Node	Node ID	str	Name of pin (e.g., A, X, Y)
	Flag	int	Encoded pin flag
	Node type	str	Cell type or primary input/output
	Instance	str	Instance ID (e.g., _123461_)
	# Inputs	int	Number of connected input pins
	# Outputs	int	Number of connected output pins
Edge	Delay	float	Rise and fall time between pins
Graph	Longest Path	int	Max DAG path length
	Clock period	float	Timing constraint
	Aspect ratio	float	Placement constraint
	Max utilization	float	Routing constraint
	Max skew	float	Max allowed skew
	Max fanout	float	Fanout constraint
	Max clock cap	float	Max clock capacitance
Type	**Label**	**Type**	**Description**
Edge	Edge_label	float	Rise and fall transition time between pins
Graph	Area	float	Total area (per stage)
	Cpath	float	Critical path delay (per stage)
	Pleak	float	Total leakage power (per stage)

including abstraction level, design size, feature types, and graph representations for [19], [7], and this research.

Table 6.2: Comparison of datasets used in this research

Aspect	Lopera et al. [19]	Guo et al. [7]	Converted Dataset (This Research)
Abstraction Level	RTL Intermediate Representation	Post-placement Physical Design	RTL Intermediate Representation
EDA Stage	Post-synthesis	Post-route	Post-route
Graph Representation	RTL Operator Graph	Property Graph Schema (Deep Graph Library)	PyTorch Geometric Graph (GraphML → PyG)
Number of Designs	Small-to-medium RTL modules	21 Benchmarks	960 physical designs (20 IPs × 48 runs)
Node Features	Operator type, role	Gate/cell/pin attributes, fan-in/out, area, leakage	Gate/cell/pin attributes, fan-in/out, area, leakage
Edge Features	Data/control flow	Slew, look-up table (LUT)	Slew, signal type (inverted/original)
Graph-level Features	None	None	Clock constraints, placement specs
Labels	Critical path delay (arrival time)	Edge-level net delay, Slack	Area, delay, leakage power

We also experimented on **gate-level Graph** which graph structure is available in Appendix A.1.1.

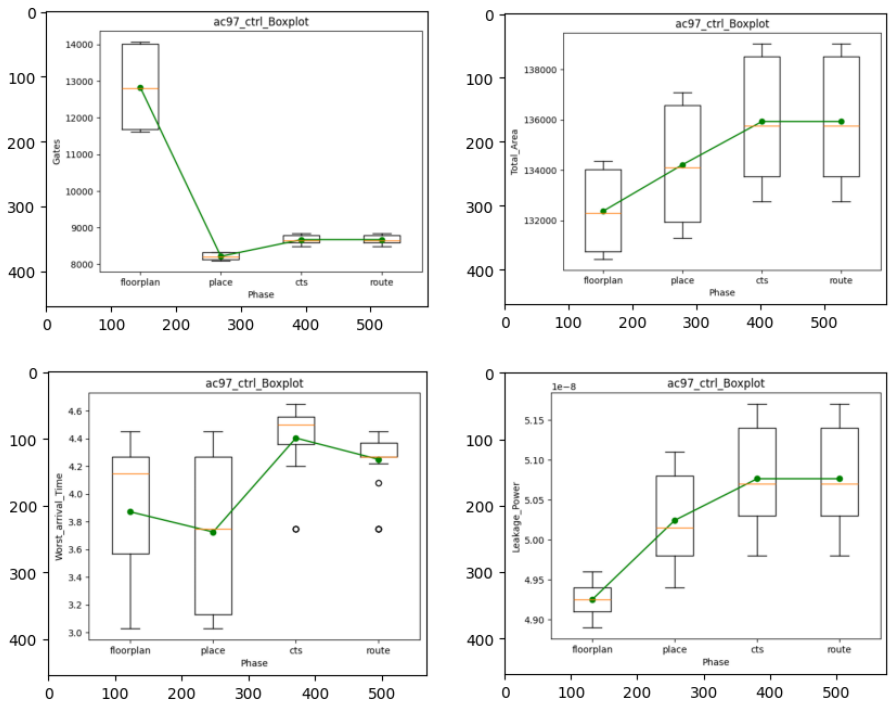


Figure 6.3: Evolution of PPA and gate count across design stages for `ac97_ctrl` ([17])

6.2.3 Data Analysis

Figure 6.3 illustrates how power, performance, and area (PPA) metrics, along with the total number of gates, evolve across these stages for the `ac97_ctrl` design. The observed trends indicate that while the total cell area and leakage power tend to vary inversely with the gate count as the design progresses through the stages, the critical path delay exhibits no clear correlation with either area or gate count. This non-monotonic behavior underscores the inherent complexity of accurately predicting delay, especially as it is influenced by layout-dependent effects such as congestion, routing topology, and clock skew. These results highlight the challenges of modeling delay using purely structural features, particularly in multi-stage timing prediction tasks. The change of PPA through EDA stages for all 20 designs are available in Appendix A.1.2.

Gate-Count and Composition Evolution Across the EDA Flow

We analyze the dataset of [17] to assess whether stage-wise optimization in the EDA flow induces netlist changes that can be plausibly captured by GNN message passing. Our analysis tracks (i) the total gate count at each implementation stage and (ii) the persistence of standard-cell types and instance identifiers across stages. Table 6.3 reports the aggregate number of gates per stage, while Table 6.4 quantifies the reductions observed as the flow progresses. We observe that most structural change occurs during Floorplan and Placement; by contrast, the cumulative reduction from Synthesis to Route remains below 17%, and the reduction from Floorplan to Route is below 19%. Thus, more than 80% of gates present at Synthesis persist through Route, indicating that optimization reshapes the netlist incrementally rather than wholesale. These observations support the modeling assumption that a GNN, via iterative message passing over the netlist graph, can propagate and encode the localized, stage-to-stage transformations produced by the EDA flow.

Appendix A.1.2 provides detailed per EDA stage gate statistics, including total gate counts at each EDA stage and a breakdown by standard-cell type (e.g., `buf`, `inv`) indicating which gates decrease or persist across stages.

Quantifying Structural Change via Wasserstein Distance and Standard Deviation Across EDA Stages

To evaluate the generalization capability of GNN models across structurally diverse IC designs, we employ the Wasserstein distance—also known as the Earth Mover's Distance—as a statistical measure of dissimilarity between design instances. Specifically, we compute the Wasserstein distance between the critical path delay distributions of various benchmark circuits from the dataset provided in [17]. The goal is to quantify how "far

Table 6.3: Total Number of Gate per EDA stage

Design	Synth	Floorplan	Place	Cts	Route
ac97_ctrl	8838	12828.8333	8215.375	8672.375	8672.375
des3_area	2893	3967.6667	2481.875	2496.875	2496.875
aes_core	15861	20775.3333	13985.125	14127.75	14127.75
des3_perf	72299	98611.1667	64702.1667	66772.0417	66772.0417
fpu	16041	22935.3333	15081.375	15234.5417	15234.5417
mem_ctrl	6583	9260.1667	6043.2917	6260.2917	6260.2917
i2c	668	1038.8333	597.125	625.4583	625.4583
ethernet	47167	67519.6667	45415.5417	47252.7083	47252.7083
pci	14767	21166.8333	13971.3333	14240.7083	14240.7083
simple_spi	630	991	575	603.3333	603.3333
sasc	483	785.6667	438	466.4167	466.4167
spi	2138	3050.5	1872.375	1928.7083	1928.7083
ss_pcm	400	633.8333	373.125	385.875	385.875
systemcdes	1773	2704.8333	1612.3333	1640.6667	1640.6667
systemcaes	6538	9370.6667	6248.25	6370.375	6370.375
tv80	5435	7437	4876.6667	4941.2917	4941.2917
usb_funct	10422	14608	9683.7083	9723.8333	9723.8333
vga_lcd	72017	101458.5	68025.5	71783.9583	71783.9583
usb_phy	418	671.6667	381	394	394
wb_dma	2897	4131.1667	2690.3333	2807.6667	2807.6667

apart" two designs are in terms of their timing behavior, thereby identifying test cases that are significantly different from the training distribution.

The dataset introduced in [17] provides multi-stage design data capturing the progression of each circuit through four key phases of the physical design flow: floorplan, place, cts, and route. These stages are generated using the OpenROAD flow with the SkyWater 130nm PDK. Table 6.5 illustrates the measure of Wasserstein distance between EDA stages. Table 6.6 shows the average change in standard deviation between each EDA stages. The highlighted values in each table indicate the designs with the largest Wasserstein distance and standard deviation in 20 designs, suggesting a high degree of

Table 6.4: Number of Gate Reduction between EDA stage

Design	Synth2floorplan	Floorplan2place	Place2cts	Cts2route	Synth2route		
					Gates	Synth (%)	Route (%)
ac97_ctrl	840	4838.8333	0	0	848	9.5949	9.7782
des3_area	451	1542.2917	0	0	467.625	16.164	18.7284
aes_core	2053	6971.9167	0	0	2057.5833	12.9726	14.5641
des3_perf	8651	34964.1667	0	0	8652	11.967	12.9575
fpu	1519	8434.4583	0	0	1540.125	9.6012	10.1094
mem_ctrl	697	3417.4167	0	0	740.25	11.2449	11.8245
i2c	74	450.6667	0	0	79.8333	11.9511	12.764
ethernet	4380	24791.0833	0	0	4438.4167	9.41	9.3929
pci	1574	7991.3333	0	0	1591.5	10.7774	11.1757
simple_spi	55	421	0	0	60	9.5238	9.9448
sasc	45	350.6667	0	0	48	9.9379	10.2912
spi	313	1245.875	0	0	333.375	15.5928	17.2849
ss_pcm	28	267.5833	0	0	33.75	8.4375	8.7464
systemcdes	178	1113.1667	0	0	181.3333	10.2275	11.0524
systemcaes	700	3532.8333	0	0	700.1667	10.7092	10.991
tv80	625	2629.9583	0	0	627.9583	11.554	12.7084
usb_funct	1184	5391.375	0	0	1205.375	11.5657	12.3961
vga_lcd	8308	37789.2917	0	0	8347.7917	11.5914	11.629
usb_phy	37	291.6667	0	0	38	9.0909	9.6447
wb_dma	322	1560.625	0	0	326.4583	11.2688	11.6274

statistical deviation in timing behavior.

Table 6.5 identifies eight designs—`ac97_ctrl`, `ethernet`, `pci`, `spi`, `systemcaes`, `systemcdes`, `tv80`, and `vga_lcd`—that exhibit the largest cumulative Wasserstein distance in critical-path delay across implementation stages. Likewise, Table 6.6 highlights eight designs—`ac97_ctrl`, `ethernet`, `fpu`, `mem_ctrl`, `pci`, `spi`, `tv80`, and `vga_lcd`—with the largest cumulative change in per-stage standard deviation.

Because the standard deviation is a widely used and easily interpretable measure of dispersion in statistical monitoring, we designate the latter set (the standard-deviation–ranked top eight) as our held-out test dataset. These designs are excluded from the training phase to assess how well a model trained on the remaining, less variable designs generalizes to circuits that undergo greater stage-to-stage variability. The Wasserstein-ranked top eight are then used to analyze transfer-learning behavior and error sensitivity as a function of

distributional shift across stages.

Complete per-stage computations of the Wasserstein distance and standard deviation for all PPA metrics—total area, critical-path delay, and leakage power—are provided in Appendix A.1.2. Additionally, The wasserstein distance calculation for all PPA (Total area, critical path delay, and leakage power) in post-route stage is available in Appendix A.1.2.

Table 6.5: Wasserstein distance on critical path delay decomposed by stage for each design

Design	floorplan2place	place2cts	cts2route	sum
ac97_ctrl	0.1829	0.6785	0.1544	**1.0158**
aes_core	0.006	0.429	0.11	0.545
des3_area	0.0044	0.0271	0	0.0315
des3_perf	0	0.0062	0	0.0062
ethernet	5.3927	0.2431	0.1054	**5.7413**
fpu	0.1021	0.1708	0.2804	0.5533
i2c	0.0735	0.0325	0.01	0.116
mem_ctrl	0.2875	0.0933	0.1513	0.5321
pci	0.4065	0.2102	0.0946	**0.7112**
sasc	0	0.1531	0.0208	0.174
simple_spi	0	0	0	0
spi	1.015	1.1163	0.2619	**2.3931**
ss_pcm	0	0.3744	0.0258	0.4002
systemcaes	0.8354	0	0	0.8354
systemcdes	2.7702	0.2044	0.0737	**3.0483**
tv80	0.1708	0.6629	0.3025	**1.1363**
usb_funct	0	0.6092	0.07	0.6792
usb_phy	0	0.3246	0.015	0.3396
vga_lcd	10.1162	0.1742	0.085	**10.3754**
wb_dma	0.1502	0.1877	0.0604	0.3983

6.3 Model Structure

This section introduces the graph neural network (GNN) models used in this research. All models are implemented using PyTorch Geometric and Deep Graph Library (DGL). Our goal is to evaluate the transferability of GNN-based delay predictors across designs

Table 6.6: Standard Deviation on critical path delay between stage for each design

Design	floorplan2place	place2cts	cts2route	sum
ac97_ctrl	0.4172	0.6269	0.1057	**1.1498**
aes_core	0.0161	0.6404	0.1892	0.8457
des3_area	0.0151	0.1356	0	0.1507
des3_perf	0	0.0165	0	0.0165
ethernet	8.8163	0.7048	0.288	**9.8091**
fpu	0.4106	0.419	0.8485	**1.6782**
i2c	0.116	0.1245	0.0265	0.267
mem_ctrl	0.4435	0.439	0.3281	**1.2106**
pci	0.4051	0.4662	0.1654	**1.0367**
sasc	0	0.2724	0.0297	0.3021
simple_spi	0	0	0	0
spi	0.8035	0.6766	0.5492	**2.0293**
ss_pcm	0	0.3127	0.0187	0.3314
systemcaes	0.5461	0	0	0.5461
systemcdes	0.3375	0.3433	0.0886	0.7695
tv80	0.2969	0.4888	0.3403	**1.126**
usb_funct	0	0.5455	0.0622	0.6078
usb_phy	0	0.3311	0.015	0.3461
vga_lcd	15.519	0.8107	0.2854	**16.6152**
wb_dma	0.3186	0.3584	0.0691	0.7461

with significant structural variations. To this end, we pretrain GNN models on a large dataset comprising 15 designs and fine-tune them using transfer learning on a smaller dataset of 5 statistically distant designs. The evaluation is performed using three standard regression metrics: mean absolute percentage error (MAPE), mean absolute error (MAE), and coefficient of determination (R^2 score).

6.3.1 Base Model: GCN, GAT, and EGAT Architectures

Our base architecture follows a conventional GNN pipeline consisting of a graph embedding stage, a message-passing backbone, and fully connected (FC) layers for final regression. Specifically, we compare the performance of three architectures: Graph Convolutional Networks (GCN), Graph Attention Networks (GAT), and Edge-featured Graph Attention Networks (EGAT).

Each model comprises three message-passing layers (MPL), whose internal structures

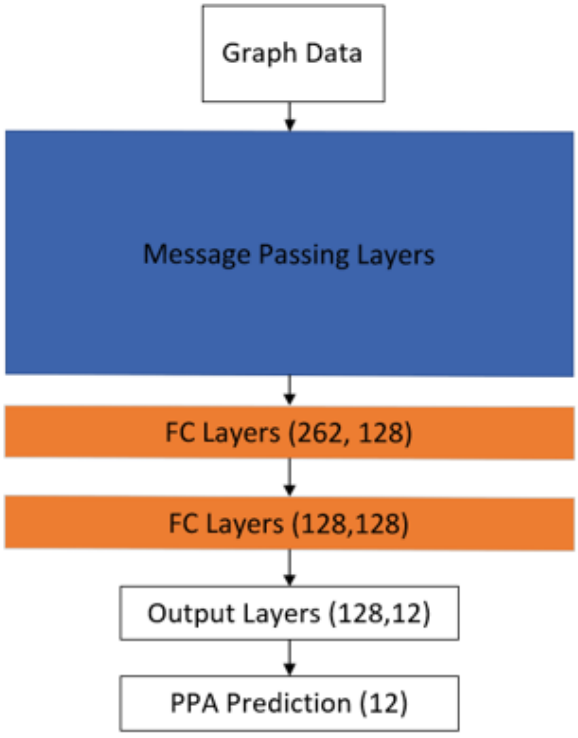
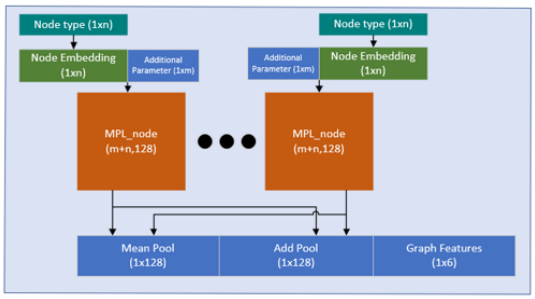


Figure 6.4: GNN Architecture

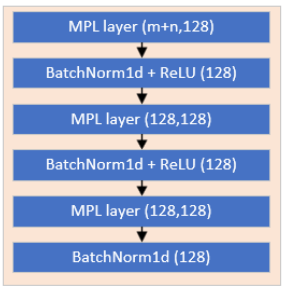
vary depending on the type of GNN used. GCN uses Graph Convolutional Layers (GCL), GAT employs Graph Attention Layers (GAL), and EGAT incorporates Edge-featured Graph Attention Layers (EGAL), which explicitly account for edge attributes in the attention mechanism. These MPL layers are followed by two fully connected layers (FC) for regression.

As shown in Figure 6.4, the GNN model consists of *Message passing layer* and*Fully Connected (FC) layer*. This GNNs aims to predict total area, critical path delay, and leakage power of four different stages.

Figure 6.5a illustrates the *Message passing layer* in details. The input graph is first passed through a node embedding layer to generate initial node and edge embeddings. The 66 different node feature (n variable) with 0 additional feature (variable m) to form



(a) Message-Passing Layer (MPL) Block



(b) Node-wise MPL Embedding with Feature

Figure 6.5: GNN Model Structure Details

a concatenated layer. The embedded features will be passed through *MPL_node* layers to execute message-passing.

These embeddings are processed to the message-passing block shown in Figure 6.5b, which includes alternating MPL layers, batch normalization, and rectified liner unit activation function. The output of the *MPL_node* layers with updated node and edge representation will be processed to global pooling layers (*mean pooling* and *sum pooling*) to aggregate information to desired dimensionality. This aggregated information will concatenate with graph-level features to be passed onto the *FC layers*. We also experimented on **gate-level Graph** which GNN structure is mentioned in Appendix A.1.3.

This design closely parallels the architecture proposed by [19], which uses GNNs to predict post-synthesis timing from RTL intermediate representations. However, in our research, we extend this framework to support post-route delay prediction using physically realized netlists and larger, more complex design graphs. In contrast to RTL graph model from [19], which operates on higher-abstraction operator-level nodes, our graphs are derived from post-routing timing paths, which offer more granular physical information. Our model also integrates graph-wide features and richer per-node descriptors, making it more suitable for low-level timing prediction.

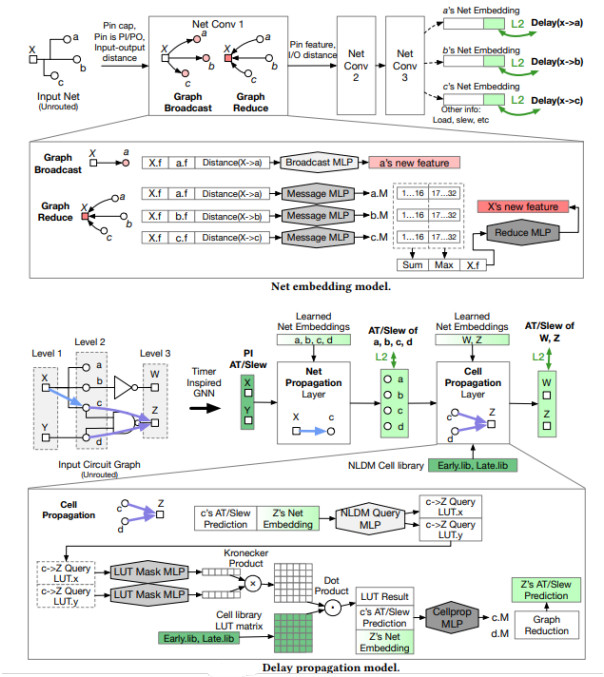


Figure 6.6: GuoGNN Architecture [7]

6.3.2 Guo-GNN: Timing-Engine Inspired GNN Architecture

To provide a stronger baseline for comparison, we implement the GNN architecture introduced by Guo et al. in [7], which is explicitly inspired by static timing analysis (STA) engines. This model is referred to as Guo-GNN in our experiments.

Net Embedding Model

The net-level module estimates post-routing interconnect delay (e.g., Elmore-style) from placement-time information by treating each routed net as a tree rooted at its driver and spanning all sink pins. Because the tree topology depends on the spatial configuration of pins (e.g., through Steiner-point insertions near clusters), the model must aggregate statistics across all pin locations on the net. To this end, it operates on a bidirectional graph that includes both net edges and their reversals, and applies a small stack of net-convolution layers. Each layer alternates a "broadcast" pass—propagating

information from the driver to sinks by concatenating driver, sink, and edge features and transforming them with a learned MLP—with a "reduction" pass that flows from sinks back to the driver using dual aggregation channels (sum and max). The final node embeddings are used to predict net delay, and the representation intentionally leaves unsupervised latent dimensions to carry quantities such as capacitive load and slew that will be consumed by downstream propagation.

Delay Propagation Model

The path-level timing stage performs levelized arrival-time and slew computation on a directed acyclic graph formed by interleaving net arcs with cell arcs. After assigning topological levels to pins, the model performs a single asynchronous update per pin by sweeping level-by-level and alternating net-propagation layers with cell-propagation layers. While the net layers mirror the broadcast/reduction mechanics from the net embedding model, cell propagation additionally requires evaluating library look-up tables (LUTs) for cell-arc delays and slews. To handle the high-dimensional LUTs efficiently, the approach learns interpolation coefficients along each LUT axis, then combines them with a Kronecker product to form a coefficient matrix that is applied to the corresponding LUT via a dot product. The resulting cell-arc messages are aggregated (sum and max) and used to update node features, from which per-pin arrival time and slew are predicted.

Overall behavior

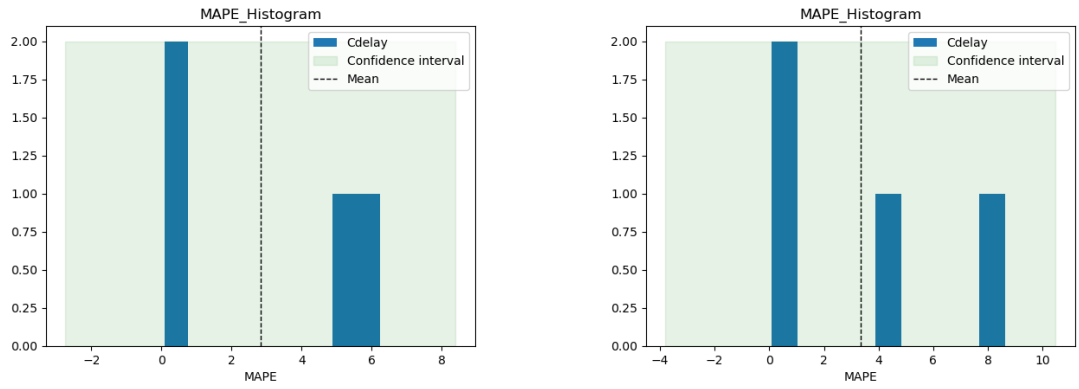
Together, the two stages implement a timing-engine–inspired GNN: the net embedding module can stand alone for net-delay prediction, while the propagation stage integrates interconnect and cell effects to recover pin-level arrivals and slews. By executing a single, levelized (asynchronous) update per pin—rather than stacking many generic GCN layers—the model closely mimics the computation order of static timing analysis, enlarges the effective receptive field without over-smoothing, and improves efficiency.

In effect, the network learns the procedural steps of STA with neural function approximators, enabling accurate pre-routing slack prediction from placement-time inputs. The Guo-GNN illustrates auxiliary supervision on intermediate values (delay, net load) improves convergence and interpretability, as it mirrors the inner workings of STA tools. The evaluation of Guo-GNN model with its original dataset is introduced in Appendix A.1.4. This experiment is executed to confirm that the performance of the model is trained properly as intended.

We replicate this architecture and train it on the dataset from [17] in order to predict post-route critical path delay with pre-floorplan graph data. By comparing Guo-GNN with our base GCN/GAT/EGAT models, we evaluate how well different message-passing strategies align with real timing behavior.

6.4 Experiments

In this section, we evaluate the models introduced in Sections 6.3. To estimate average generalization performance, we apply cross-validation to the GCN, GAT, and EGAT architectures. Unless otherwise noted, optimization follows the configuration used in Section 5.4: AdamW with a learning rate of 10^{-3}. For Guo-GNN, we adhere to the authors' original settings and use Adam with a learning rate of 5×10^{-4}. We also conducted experiments on a **gate-level** graph representation; detailed results are provided in Appendix A.1.4. For both base training and transfer learning, we employ early stopping based on the validation loss with a maximum of 100 epochs, and report the checkpoint that achieves the lowest validation loss. The models are evaluated using mean absolute percentage error (MAPE) and the coefficient of determination (R^2).



(a) Train Dataset (b) Test Dataset

Figure 6.7: GCN Cdelay Cross Validation Histogram

6.4.1 Base Model

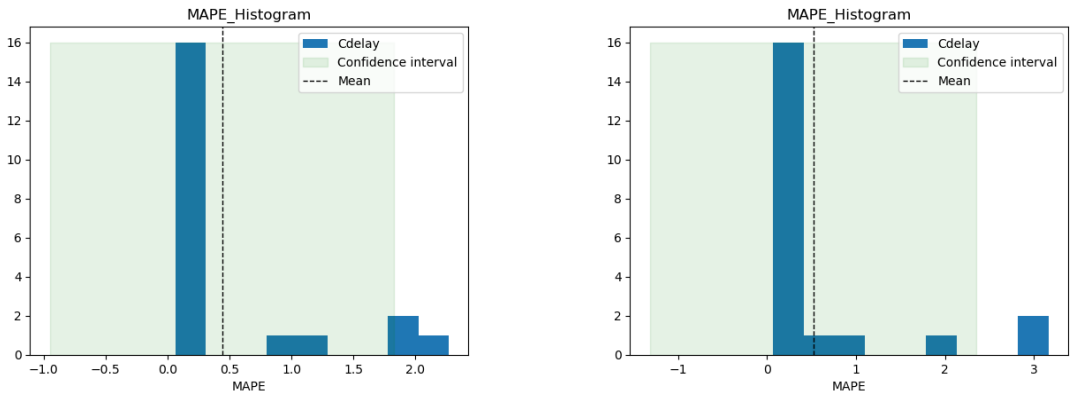
1. Basic Model Structure:

Table 6.7: Comparison of GCN, GAT, EGAT, and [7] Model Performance

Model	Splits	Epochs	Train Samples	Test Samples	Train MAPE	Test MAPE	Train R2	Test R2
GCN	4	78.75	580	380	2.8217	3.3358	-108.6242	-105.5818
GAT	40	65.75	580	380	0.544133	0.567914	-4.8913	-4.1758
EGAT	8	68.25	580	380	0.060038	0.065473	0.9946	0.9944
Guo-GNN [7]	1	26819	580	380	0.8353	0.8219	0.2427	0.2803

Table 6.7 summarizes the average training and testing performance of four GNN models—GCN, GAT, EGAT, and Guo-GNN. The GCN exhibits catastrophic performance, with a test MAPE of 3.3358 and strongly negative R^2 values (-108.62 train, -105.58 test), highlighting that standard graph convolution is insufficient for modeling fine-grained timing behavior without attention or edge-awareness. Figure 6.7 represents the histogram of MAPE in cross validation splits for GCN with train and test dataset.

The GAT model performs reasonably well, with most of its train MAPE being 0.544133 and test MAPE being 0.567914. However, in few splits, its MAPE shows drastic increase



(a) Train Dataset (b) Test Dataset

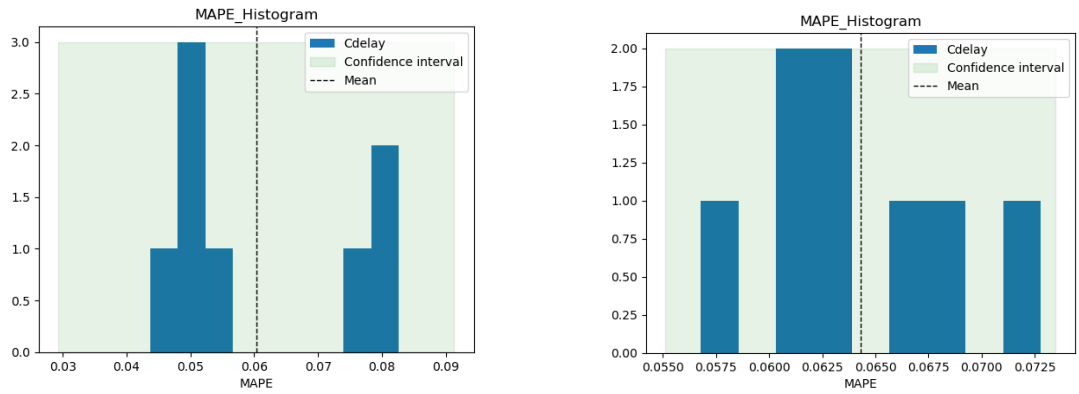
Figure 6.8: GAT Cdelay Cross Validation Histogram

in both train and test dataset. Also, its R^2 values (-4.8913 train, -4.1758 test) show that its performance is less stable when applied to fine-grained pin-level graph structures. This unstable stable of GAT could be seen in Figure 6.8 which shows the histogram of MAPE in cross validation splits for GAT with train and test dataset.

the EGAT model demonstrates the strongest overall performance, achieving a train MAPE of 0.0600 and a test MAPE of 0.0655, with corresponding R^2 values of 0.9946 and 0.9944. These results indicate that incorporating edge-level features (e.g., slew, net delay) within the attention mechanism enables EGAT to capture timing-critical relationships more effectively. Figure 6.9 represents the histogram of MAPE in cross validation splits for EGAT with train and test dataset.

2. Comparison with Guo-GNN:

The Guo-GNN model [7], which employs a timing-engine–inspired message-passing scheme, attains a test MAPE of 0.8219 and an R^2 of 0.2803. Although it outperforms GCN, its accuracy is substantially below that of GAT and EGAT, indicating that topology constrained message passing alone is less effective than attention-based mechanisms on structurally heterogeneous netlists. A principal reason is a feature–stage mismatch:



(a) Train Dataset (b) Test Dataset

Figure 6.9: EGAT Cdelay Cross Validation Histogram

Guo-GNN is designed to predict post-route slack and arrival from post-placement inputs, whereas our setting requires modeling the pre-floorplan → post-placement trajectory. As discussed in Section 6.3.2, the published **Net Embedding Model** and **Delay Propagation Model** operate on placement-resolved net trees and cell-library contexts and thus do not natively accommodate pre-floorplan signals—leading to degraded MAPE and R^2 on both train and test. To bridge this gap, we hypothesize that Guo-GNN must be augmented with a front-end net-embedding and delay-propagation stack tailored to pre-floorplan features (e.g., aspect ratio, utilization, clock period) and with the graph features used to generate our dataset. We instantiate and evaluate this idea in Appendix A.1.5.

3. Per design analysis:

Table 6.8 reports the per-design average performance on the training and test sets, using mean absolute percentage error (MAPE) as the evaluation metric. Notably, the `i2c` and `simple_spi` designs achieve a MAPE of 0 across GCN, GAT, and EGAT. This outcome suggests that, for these instances, the message-passing models are able to reproduce the labeled critical-path delay exactly. A plausible explanation is that these designs ex-

Table 6.8: Comparison on Critical Path Delay Prediction MAPE

	Statistics		Train Dataset				Test Dataset			
Design	**Standard Deviation**	**Wasserstein Distance**	**GCN**	**GAT**	**EGAT**	**Guo-GNN**	**GCN**	**GAT**	**EGAT**	**Guo-GNN**
ac97_ctrl	1.1498	1.0158	0.6007	0.0964	0.0328	0.9021	0.6021	0.0992	0.0369	0.9022
aes_core	0.8457	0.545	0.198	0.1385	0.0192	0.0077	0.1986	0.1375	0.0196	0.0077
des3_area	0.1507	0.0315	0.0879	0.0557	0.0273	0.0083	0.0873	0.0573	0.0243	0.0083
des3_perf	0.0165	0.0062	0.2555	0.4294	0.0196	0	0.2514	0.4295	0.019	0.0043
ethernet	9.8091	5.7413	16.1706	2.4598	0.2672	0	16.6544	2.543	0.328	0.9676
fpu	1.6782	0.5533	0.0016	0.0311	0.0019	0	0.0023	0.0311	0.0021	0
i2c	0.267	0.116	0	0	0	0.9588	0	0	0	0.9589
mem_ctrl	1.2106	0.5321	0.2982	0.0988	0.0391	0.9045	0.2974	0.0992	0.042	0.9044
pci	1.0367	0.7112	0.2751	0.0386	0.01	0.9236	0.2741	0.0388	0.0126	0.9236
sasc	0.3021	0.174	0.0217	0.015	0.0117	0.7492	0.023	0.0157	0.0137	0.7493
simple_spi	0	0	0	0	0	0.8602	0	0	0	0.8601
spi	2.0293	2.3931	0.0853	0.0967	0.0714	0.9477	0.0684	0.0799	0.0501	0.9478
ss_pcm	0.3314	0.4002	0.1281	0.0887	0.0671	0.0732	0.1223	0.0861	0.0728	0.0732
systemcaes	0.5461	0.8354	0.1452	0.0957	0.0272	0.9052	0.1417	0.0956	0.0269	0.9052
systemcdes	0.7695	3.0483	0.0112	0.0137	0.0081	0.8686	0.0121	0.0149	0.0105	0.8687
tv80	1.126	1.1363	0.1087	0.0941	0.0371	0.125	0.1062	0.0961	0.04	0.125
usb_funct	0.6078	0.6792	0.8385	0.3031	0.0916	0	0.8477	0.2997	0.1021	0
usb_phy	0.3461	0.3396	0.0856	0.0845	0.0547	0.1176	0.0797	0.0849	0.0659	0.1176
vga_lcd	16.6152	10.3754	39.3582	6.5613	0.3606	0	42.5623	7.3086	0.4179	0.9316
wb_dma	0.7461	0.3983	0.1318	0.0583	0.0451	0.6057	0.1405	0.0591	0.0397	0.6058
Avg.			2.9401	0.538	0.0596	0.4479	3.1236	0.5788	0.0662	0.5431

Table 6.9: Statistics on Critical Path Delay Prediction MAPE

	Statistics		Train Dataset				Test Dataset			
Design	**Standard Deviation**	**Wasserstein Distance**	**GCN**	**GAT**	**EGAT**	**Guo-GNN**	**GCN**	**GAT**	**EGAT**	**Guo-GNN**
ac97_ctrl	6	6	4	8	-	6	4	6	-	8
aes_core	9	-	8	5	-	-	8	5	-	-
des3_area	-	-	-	-	-	-	-	-	-	-
des3_perf	-	-	7	3	-	-	7	3	-	-
ethernet	2	2	2	2	2	-	2	2	2	1
fpu	4	-	-	-	-	-	-	-	-	-
i2c	-	-	-	-	-	1	-	-	-	2
mem_ctrl	5	-	5	6	8	5	5	7	7	7
pci	8	8	6	-	-	3	6	-	-	5
sasc	-	-	-	-	-	-	-	-	-	-
simple_spi	-	-	-	-	-	8	-	-	-	-
spi	3	4	-	7	4	2	-	-	6	3
ss_pcm	-	-	-	-	5	-	-	-	4	-
systemcaes	-	7	-	-	-	4	-	-	-	6
systemcdes	-	3	-	-	-	7	-	-	-	-
tv80	7	5	-	-	-	-	-	8	8	-
usb_funct	-	-	3	4	3	-	3	4	3	-
usb_phy	-	-	-	-	6	-	-	-	5	-
vga_lcd	1	1	1	1	1	-	1	1	1	4
wb_dma	-	-	-	-	7	-	-	-	-	-

perience minimal stage-to-stage variation (e.g., negligible structural or parasitic changes) or that compensating effects in the flow render the effective delay invariant, yielding a prediction task that is comparatively well conditioned.

Table 6.9 ranks the eight most challenging designs (highest MAPE) for each GNN, based on the values in Table 6.8. While the resulting orderings do not align perfectly with

either the Wasserstein-distance criterion or the cumulative change in standard deviation, they consistently identify `ethernet` and `vga_lcd` as outliers with the worst errors. This is consistent with their large inter-stage distributional shift and dispersion (Wasserstein distance > 5.7413 and cumulative change in standard deviation > 9.8091), indicating that pronounced stage-wise drift tends to coincide with degraded predictive accuracy, even though the relationship is not strictly monotonic across all designs.

6.4.2 Transfer Learning

Base Model Performance (Before Transfer Learning)

Table 6.10: Comparison of GCN, GAT, and EGAT Base Models before Retraining

Model	Epochs	Train Samples	Test Samples	Train MAPE	Test MAPE
GCN	97	576	384	0.0221	0.5453
GAT	75	576	384	0.0219	0.0448
EGAT	76	576	384	0.0090	0.2313

Table 6.10 reports model performance before transfer learning. In this setting, GAT attains the strongest test accuracy (MAPE = 0.0448), followed by **EGAT** (MAPE = 0.2313). GCN performs substantially worse (MAPE = 0.5453), indicating limited expressiveness for fine-grained timing prediction under the pin-level representation. Notably, EGAT achieves the lowest training error (MAPE = 0.0090), reflecting its capacity to capture precise edge-aware relationships in the training domain.

The magnitudes in Table 6.10 are markedly better than the cross-validated averages in Table 6.7 and the one-shot transfer results in Table 6.11 for three main reasons. First, the evaluation protocol differs: Table 6.10 uses a fixed train/test split tailored to the pin-level setting, whereas Table 6.7 aggregates across multiple splits and design granu-

larities, making it more sensitive to heavy-tailed errors from outlier designs (e.g., `vga_lcd`, `ethernet`). Second, the data regime is more favorable: the base-model results leverage hundreds of labeled samples per model, while the transfer-learning setting in Table 6.11 employs only a single fine-tuning example (one-shot), which necessarily increases variance and degrades generalization. Third, representational alignment matters: at the pin level, attention mechanisms (GAT/EGAT) better exploit directional and edge-specific cues than convolutional aggregation (GCN), yielding sharper fits and lower test MAPE under a stable split with early stopping (Section 6.4). Collectively, these factors explain why the base-model numbers in Table 6.10 appear substantially stronger than both the cross-validated averages and the one-shot transfer outcomes.

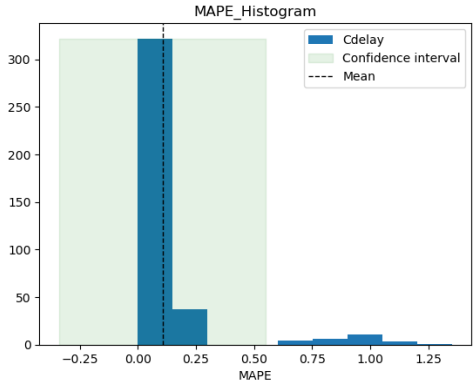
Retrained Model Performance (After Transfer Learning)

Table 6.11: Comparison of GCN, GAT, and EGAT Retrained Models

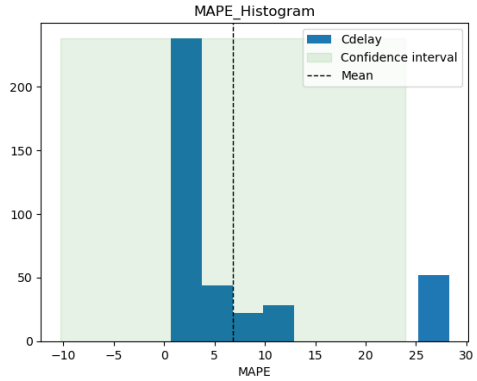
Model	Splits	Epochs	Train Samples	Test Samples	Train MAPE	Test MAPE
GCN	384	67.8	1	383	0.0363	1.9421
GAT	384	100	1	383	0.0000	0.0740
EGAT	384	95.5	1	383	0.1442	0.1915

Table 6.11 summarize the average performance of GCN, GAT, and EGAT after transfer learning. After retraining with a single sample (one-shot transfer), GAT maintains strong generalization, achieving the lowest test MAPE (0.0740). GCN, however, exhibits severe overfitting: despite a low training MAPE (0.0363), its test MAPE increases dramatically to 1.9421. EGAT shows moderate adaptability, with a test MAPE of 0.1915, suggesting that although EGAT captures detailed edge-level information well during full training, it may rely on more data than a single example for effective transfer.

Figure 6.10, 6.11, and 6.12 represents the histogram of MAPE in cross validation splits for GCN, GAT, and EGAT retrained models.

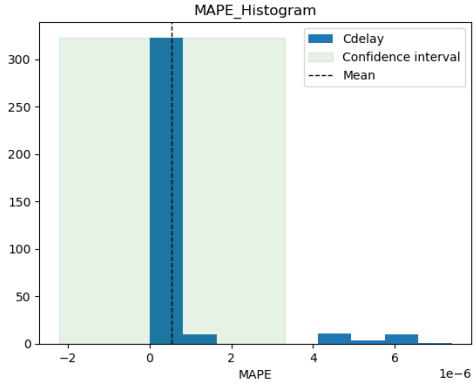


(a) Train Dataset

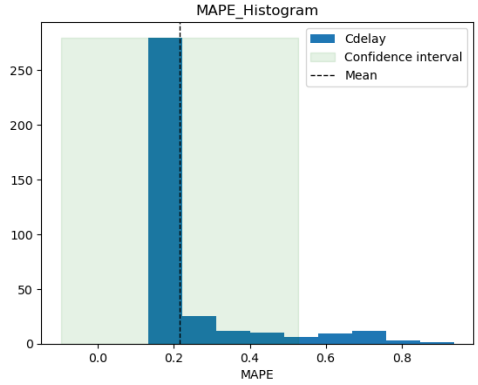


(b) Test Dataset

Figure 6.10: Retrained GCN Cdelay Cross Validation Histogram

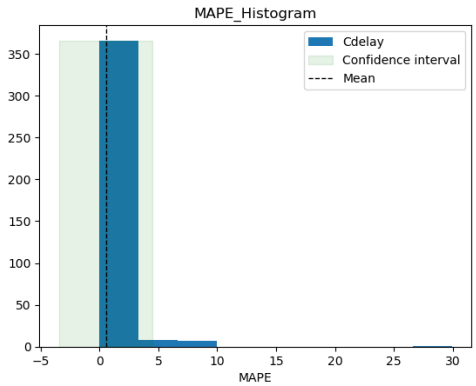


(a) Train Dataset

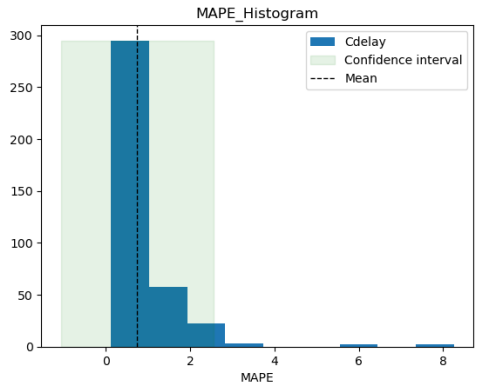


(b) Test Dataset

Figure 6.11: Retrained GAT Cdelay Cross Validation Histogram



(a) Train Dataset



(b) Test Dataset

Figure 6.12: Retrained EGAT Cdelay Cross Validation Histogram

3. Per design analysis:

Table 6.12: Critical Path Delay MAPE Comparison across GCN, GAT, and EGAT Before Retraining Models

Dataset	Design	GCN	GAT	EGAT
Train	Samples	576	576	576
Test	Samples	384	384	384
Train	aes_core	0.0207	0.0206	0.018
	des3_area	0.0311	0.031	0.0189
	des3_perf	0.0226	0.0244	0.0245
	i2c	0	0	0
	sasc	0.0154	0.0154	0.007
	simple_spi	0	0	0
	ss_pcm	0.09	0.0887	0.033
	systemcaes	0.032	0.0319	0.0218
	systemcdes	0.0097	0.0093	0.009
	usb_funct	0.1205	0.1205	0.0654
	usb_phy	0.0842	0.0843	0.0265
	wb_dma	0.0454	0.0453	0.0271
Train	Avg. Train	0.0363	0.0363	0.0193
Test	ac97_ctrl	0.274	0.0371	0.0554
	ethernet	0.3677	0.3527	1.8126
	fpu	0.0724	0.0021	0.009
	mem_ctrl	0.0525	0.0492	0.0725
	pci	0.0108	0.0108	0.0149
	spi	0.0822	0.06	0.0959
	tv80	0.1979	0.0466	0.0387
	vga_lcd	16.7481	0.5007	3.1149
Test	Avg. Test	2.2257	0.1324	0.6518

Tables 6.12 and 6.13 report per-design accuracy for the pretrained and retrained models in transfer learning, with performance measured by mean absolute percentage error (MAPE). The results reveal substantial variability across designs. In particular, `i2c` and `simple_spi` attain a MAPE of 0 for GCN, GAT, and EGAT, indicating that the message-passing models exactly reproduce the labeled critical-path delay for these in-

Table 6.13: Critical Path Delay MAPE Comparison across GCN, GAT, and EGAT Retrained Models

Dataset	**Design**	**GCN**	**GAT**	**EGAT**
Train	Samples	1	1	1
Test	Samples	383	383	383
Train	Avg. Train	0.1060	5.45E-07	0.5282
Test	ac97_ctrl	0.6959	0.0622	0.0857
	ethernet	16.8267	0.7126	1.7703
	fpu	0.1336	0.0038	0.007
	mem_ctrl	0.3987	0.0657	0.0553
	pci	0.2342	0.0196	0.0174
	spi	0.147	0.1075	0.0786
	tv80	0.2476	0.057	0.0449
	vga_lcd	35.662	0.6845	3.8792
Test	Avg. Test	6.7932	0.2141	0.7423

stances. A plausible explanation is that these designs exhibit minimal stage-to-stage variation (e.g., negligible structural or parasitic changes), or that compensating effects across the flow render the effective delay invariant, yielding a comparatively well-conditioned prediction task.

By contrast, `vga_lcd` consistently shows the largest test error among all designs—especially under one-shot retraining. For example, EGAT's test MAPE increases from 3.1149 (base model) to 3.8792 after retraining, suggesting reduced transferability when the target design is structurally dissimilar to the training set. To interrogate this behavior, we quantify inter-design distributional differences using the Wasserstein distance, which measures discrepancies between the distributions of timing and structural features across netlists. Designs such as `vga_lcd` and `ethernet` exhibit notably larger Wasserstein distances relative to designs like `ac97_ctrl` or `fpu`, indicating that they occupy underrepresented regions of the design space in the retraining corpus; correspondingly, models adapted on statistically distant designs show diminished accuracy when evaluated on these outliers.

These findings support the hypothesis that the Wasserstein distance is a useful proxy

for estimating model generalization limits in transfer-learning scenarios. Incorporating this metric into the pipeline—e.g., for Wasserstein-guided training-set selection, loss weighting based on feature divergence, or curriculum design that progressively reduces distributional mismatch—can provide a principled basis for more robust retraining and improved cross-design transfer.

Overall Interpretation

These results lead to the following observations:

- **GAT outperforms GCN and EGAT in transfer settings**, demonstrating strong robustness across data regimes.
- **GCN performs poorly** under both full training and transfer learning, highlighting the limitations of convolutional aggregation for modeling timing behavior in fine-grained circuit graphs.
- **EGAT provides state-of-the-art accuracy under full supervision** but is moderately affected during one-shot retraining due to its reliance on edge-level features.
- **Guo-GNN improves over GCN but does not surpass attention-based GNNs**, reinforcing the effectiveness of attention mechanisms for timing prediction tasks.

Importantly, our analysis also reveals that transfer learning effectiveness varies across designs depending on their statistical proximity to the source distribution. We use the Wasserstein distance to quantify this distance between designs, offering a principled way to estimate how well knowledge transfers between domains. For example, the `vga_lcd` design, which shows the highest critical path delay MAPE after transfer (3.8792 with EGAT), also has one of the largest Wasserstein distances from the training designs.

This supports the hypothesis that statistical dissimilarity—measured via Wasserstein distance—can serve as a proxy for transferability, and suggests that model degradation is predictable when transferring to structurally distant designs.

These insights collectively support the view that attention-based GNNs—particularly GAT—offer the best balance of accuracy and generalization for PPA prediction under both full-data and few-shot settings.

6.5 Conclusion and Future Work

This chapter evaluated four graph neural network (GNN) models—GCN, GAT, EGAT, and Guo-GNN—on the task of critical path delay prediction using graph representations extracted from post-routing circuit designs. The models were assessed both under full training (base model) and transfer learning (one-shot retraining), using metrics such as mean absolute percentage error (MAPE) and R^2 score.

In the base model evaluation, where models are trained on the full dataset, the results in Table 6.7 show that EGAT achieves the highest overall accuracy, with a test MAPE of 0.0655 and R^2 score of 0.9944. This confirms that incorporating edge-level features (e.g., slew, net delay) significantly improves the model's ability to capture timing-critical patterns. GAT also performs strongly achieving a test MAPE of 0.5679, while GCN demonstrates severe underperformance, with a test MAPE of 3.3358 and highly negative R^2 values, indicating poor fit and lack of generalization. The Guo-GNN model outperforms GCN but falls short of EGAT and GAT, suggesting that while STA-inspired message passing provides moderate gains, attention-based GNNs remain more effective in this domain.

In the transfer learning experiments (Tables 6.10 and 6.11), models were retrained with only one training sample, testing their capacity to generalize with extremely limited

data. Under this constraint, GAT maintains exceptional performance, achieving the lowest test MAPE of 0.0740, demonstrating that attention mechanisms are highly effective even under low-data regimes. In contrast, GCN suffers from significant overfitting, with its test MAPE rising sharply to 1.9421, despite achieving low training error. This underscores its limited expressiveness and inability to generalize when data is scarce. EGAT delivers a reasonable trade-off between precision and adaptability, with a test MAPE of 0.1915, suggesting that while edge-featured models are effective under full supervision, they may require more data to adapt reliably during transfer.

A key contribution of this work is the integration of the Wasserstein distance to explain variations in transfer performance. By computing the statistical distance between training and test designs, we observed a strong correlation between high Wasserstein distances and reduced predictive accuracy—particularly for designs like `vga_lcd`, which consistently yielded high MAPE after transfer. This finding highlights the importance of dataset diversity and distributional alignment in enabling successful transfer learning. It also suggests that Wasserstein distance may serve as a practical guide for data selection, model weighting, or curriculum-based retraining strategies in GNN-based EDA applications.

Overall, the experimental results reveal the following key insights:

- GAT is the most consistent and generalizable model, performing well in both full-data and low-data regimes.
- EGAT offers state-of-the-art accuracy under full supervision, but its performance degrades moderately under transfer learning, likely due to its reliance on edge-level features.
- GCN is insufficient for timing prediction in fine-grained netlists, especially when data is limited.

- Guo-GNN demonstrates improved stability over GCN, but does not outperform attention-based models in structurally diverse datasets.
- Wasserstein distance is a valuable metric for estimating cross-design transferability.

These findings strongly support the adoption of attention-based GNN architectures—particularly GAT—for post-routing timing prediction and transfer learning in circuit design automation. Their ability to maintain predictive accuracy across varying data regimes makes them highly promising for practical deployment, especially in early-stage design exploration where labeled data is scarce.

Chapter 7

Conclusion and Future Work

This dissertation explored multiple deep learning methodologies for Power, Performance, and Area (PPA) prediction in digital integrated circuit design, with an emphasis on generalization across structurally and statistically diverse designs. The research was structured around three core investigations: (1) transfer learning using open-source GNNs trained on the OpenABC-D dataset, (2) meta-transfer learning applied to wide-coverage PPA modeling using minimal QoR samples, and (3) critical path delay prediction using a variety of GNN architectures with post-routing graph representations.

7.1 Conclusions

Transfer Learning from OpenABC-D

The first study demonstrated that a GNN model pretrained on the OpenABC-D dataset can be effectively adapted to commercial designs with minimal retraining. In particular, area prediction benefited significantly from transfer learning, achieving up to a 57% reduction in error through simple one-layer fine-tuning. However, critical path delay prediction proved less robust, with only marginal improvements observed. This suggests

that certain timing-sensitive characteristics in commercial designs—such as parasitics and proprietary PDK-specific effects—may not transfer well from open-source training data alone. Future directions for this work include incorporating additional timing features (e.g., slew, fanout, congestion) and applying domain adaptation methods to better align open-source and commercial data distributions.

Meta-Transfer Learning for Few-Shot PPA Prediction

The second study investigated meta-transfer learning as a means to generalize across a wide range of design topologies using very few training samples. Results showed that meta-learning models—when properly tuned—could approximate PPA across statistically distant designs, particularly when architectural similarity or QoR sparsity was preserved. While this approach demonstrated the potential to support few-shot learning scenarios, denormalized MAPE remained above the 5% target in most cases. Furthermore, experiments were limited to a single design corner due to time and computational constraints. Ongoing work focuses on tuning meta-optimization parameters and extending the evaluation to multiple corners and technology nodes.

Graph Neural Network Architectures for Critical Path Delay

The final study benchmarked several GNN architectures—GCN, GAT, EGAT, and Guo-GNN—on post-routing timing prediction tasks. GAT consistently emerged as the most generalizable model, achieving high accuracy in both full-data and one-shot transfer regimes. EGAT provided state-of-the-art accuracy under full supervision, but its reliance on edge features reduced its adaptability in data-scarce scenarios. GCN was shown to be inadequate for timing prediction in fine-grained netlists, especially when limited data was available.

A key contribution of this work was the integration of the **Wasserstein distance** as a metric for assessing statistical distance between training and test designs. The results revealed a strong correlation between large Wasserstein distances and degraded prediction

accuracy, particularly for designs like `vga_lcd` that were structurally distant from the training domain. This metric offers practical utility in guiding data selection, estimating transfer risk, and designing retraining curricula for future GNN applications in EDA.

7.2 Future Work

Across the three research threads, several common themes and challenges emerged:

- **Attention-based GNNs (especially GAT)** provide robust performance across both low- and high-data regimes, and are the most suitable choice for real-world deployment.
- **Edge-feature–aware architectures** (e.g., EGAT) achieve high fidelity but require richer supervision or additional training samples to transfer effectively.
- **Transferability is tightly linked to dataset similarity**, which can be meaningfully quantified using the Wasserstein metric.
- **Few-shot and meta-learning methods are promising** but require careful tuning of learning rates, task sampling strategies, and architectural consistency to achieve target-level MAPE.

Future work will focus on several promising directions: integrating adversarial domain adaptation techniques, expanding training datasets to cover more corners and layout conditions, using Wasserstein-guided curriculum strategies, and developing unified multi-task GNNs for simultaneous prediction of area, power, and delay. Ultimately, the goal is to create scalable, explainable, and efficient GNN-based frameworks capable of early-stage PPA estimation across diverse design spaces.

REFERENCES

[1] A. Abdollahi, M. Kamal, and M. Pedram. Rocketppa: Ultra-fast llm-based ppa estimator at code-level abstraction. *arXiv e-prints*, pages arXiv–2503, 2025.

[2] A. Biere, K. Heljanko, and S. Wieringa. AIGER 1.9 and beyond. Technical Report 11/2, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria, 2011.

[3] A. B. Chowdhury, B. Tan, R. Karri, and S. Garg. Openabc-d: A large-scale dataset for machine learning guided integrated circuit synthesis. *arXiv preprint arXiv:2110.11292*, 2021.

[4] S. Cotton. go-air/gini: Sapeur, Jan. 2019. URL `https://doi.org/10.5281/zenodo.2553490`.

[5] Z. Dong, W. Cao, M. Zhang, D. Tao, Y. Chen, and X. Zhang. Cktgnn: Circuit graph neural network for electronic design automation. *arXiv preprint arXiv:2308.16406*, 2023.

[6] L. Francisco, P. Franzon, and W. R. Davis. Fast and accurate ppa modeling with transfer learning. In *2021 ACM/IEEE 3rd Workshop on Machine Learning for CAD (MLCAD)*, pages 1–6, 2021. doi: 10.1109/MLCAD52597.2021.9531109.

[7] Z. Guo, M. Liu, J. Gu, S. Zhang, D. Z. Pan, and Y. Lin. A timing engine inspired graph neural network model for pre-routing slack prediction. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 1207–1212, 2022.

[8] X. Han, Z. Huang, B. An, and J. Bai. Adaptive transfer learning on graph neural networks. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, pages 565–574, 2021.

[9] L. R. Juracy, A. de Morais Amory, and F. G. Moraes. A fast, accurate, and comprehensive ppa estimation of convolutional hardware accelerators. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 69(12):5171–5184, 2022. doi: 10.1109/TCSI.2022.3204932.

[10] N. Kooverjee, S. James, and T. Van Zyl. Investigating transfer learning in graph neural networks. *Electronics*, 11(8):1202, 2022.

[11] F. Last and U. Schlichtmann. Feeding hungry models less: Deep transfer learning for embedded memory ppa models : Special session. In *2021 ACM/IEEE 3rd Workshop on Machine Learning for CAD (MLCAD)*, pages 1–6, 2021. doi: 10.1109/MLCAD52597.2021.9531299.

[12] Z. Li, F. Zhou, F. Chen, and H. Li. Meta-sgd: Learning to learn quickly for few-shot learning, 2017.

[13] I. Loshchilov and F. Hutter. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101*, 2017.

[14] R. McElreath. *Statistical rethinking: A Bayesian course with examples in R and Stan*. Chapman and Hall/CRC, 2018.

[15] T. Munkhdalai and H. Yu. Meta networks, 2017.

[16] A. Ramdas, N. García Trillos, and M. Cuturi. On wasserstein two-sample testing and related families of nonparametric tests. *Entropy*, 19(2):47, 2017.

[17] P. Shrestha, A. Aversa, S. Phatharodom, and I. Savidis. Eda-schema: A graph datamodel schema and open dataset for digital design automation. In *Proceedings of the Great Lakes Symposium on VLSI 2024*, pages 69–77, 2024.

[18] Q. Sun, Y. Liu, T.-S. Chua, and B. Schiele. Meta-transfer learning for few-shot learning, 2019.

[19] D. Sánchez Lopera, I. Subedi, and W. Ecker. Using graph neural networks for timing estimations of rtl intermediate representations. In *2023 ACM/IEEE 5th Workshop on Machine Learning for CAD (MLCAD)*, pages 1–6, 2023. doi: 10.1109/MLCAD58807.2023.10299859.

[20] V. Thost and J. Chen. Directed acyclic graph neural networks, 2021.

[21] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, and Y. Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[22] C. Villani et al. *Optimal transport: old and new*, volume 338. Springer, 2009.

[23] O. Vlcek and V. Mrazek. Approxgnn: A pretrained gnn for parameter prediction in design space exploration for approximate computing, 2025. URL `https://arxiv.org/abs/2507.16379`.

[24] H. Wang, K. Wang, J. Yang, L. Shen, N. Sun, H.-S. Lee, and S. Han. Gcn-rl circuit designer: Transferable transistor sizing with graph neural networks and reinforcement learning. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[25] Z. Zhang, T. Chen, J. Huang, and M. Zhang. A fast parameter tuning framework via transfer learning and multi-objective bayesian optimization. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 133–138, 2022.

APPENDICES

Appendix A

Additional PPA Results

This appendix shows all additional PPA results from Chapter 5 and 6

A.1 Critical Path Delay Prediction for Evolving Netlists

A.1.1 Graph Structure and Format

Gate-level Graph: Each node in the gate-level graph corresponds to a gate or port, with features describing connectivity, area, leakage power, and functional role. Edges capture signal flow. Labels include total area, leakage power, and critical path delay.

Table A.1 summarizes the node-, edge-, and graph-level features extracted from the EDA-schema dataset in **Gate-level Graph** format. Collectively, these features capture logical connectivity, selected physical attributes, and the timing constraints used in the flow.

Node features The Node ID records the gate identifier (e.g., _123461_). The Node type encodes the standard-cell class as a categorical flag on 434 library cells, which is encoded as a categorical flag spanning in Flag. The #Inputs and #Outputs record the fan-in and fan-out counts, respectively, thereby characterizing local netlist connectivity.

Table A.1: Gate-level Graph Structure

Type	Feature	Type	Description
Node	Node ID	str	Instance ID (e.g., _123461_)
	Flag	int	Encoded gate role flag
	Node type	str	Standard Cell Name (e.g., sky130_fd_sc_hd__a32o_4)
	# Inputs	int	Number of connected input
	# Outputs	int	Number of connected output
	Cell area	float	Area of the gate instance
	Cell leakage power	float	Leakage power of the cell
Graph	Longest Path	int	Max DAG path length
	Clock period	float	Design constraint
	Aspect ratio	float	Floorplan setting
	Max utilization	float	Placement density limit
	Max skew	float	Clock skew constraint
	Max fanout	float	Maximum allowed fan-out
	Max clock cap	float	Clock network capacitance limit

Type	Label	Type	Description
Graph	Area	float	Total cell area
	Arrival time	float	Worst case delay (critical path)
	Leakage Power	float	Total leakage power

The Cell area and Cell leakage power illustrates the cell area and cell leakage power from the library file in sky130_fd_sc_hd technology.

Graph features Graph-level attributes encode the global constraints and settings used to generate each design point (e.g., clock period, aspect ratio, maximum utilization, maximum skew, maximum fan-out, and maximum clock-network capacitance), as well as summary indicators such as the longest path length.

Labels (targets) As shown in Table 6.1, graph-level labels report PPA metrics—total area, worst (critical-path) arrival time, and total leakage power—at four implementation stages: floorplan, placement(place), clock-tree synthesis (CTS), and routing(route).

A.1.2 Data Analysis

Table A.2, A.3, A.4, A.5, and A.6 illustrates the detailed total number of gate at each EDA stage with a breakdown by standard-cell type (e.g., `buf`, `inv`).

Table A.7 illustrates the detailed total number of gate that decreased at each EDA stage with a breakdown by standard-cell type (e.g., `buf`, `inv`).

Wasserstein Metrics for Post-Route Stage

We designate `ac97_ctrl` as the reference design and calculate the pairwise Wasserstein distance between it and each of the remaining designs in *post-route* stage. This allows us to systematically select test designs that are statistically distant from the training set. Table A.8 presents the computed Wasserstein distances under three normalization strategies: raw values, min-max normalization, and Z-score normalization. Each normalization method offers a different perspective on distributional similarity, with Z-score normalization emphasizing relative deviation and min-max scaling bounding values to a common range.

The highlighted values in Table A.8 indicate the designs with the largest Wasserstein distance from `ac97_ctrl`, suggesting a high degree of statistical deviation in timing behavior. Figure A.1 illustrates the relationship between critical path delay and total cell area across the dataset in the post-route stage (using raw, unnormalized values). While some correlation patterns emerge in a few designs, the overall distribution further highlights the complexity of delay prediction, especially when training and test domains differ significantly. The additional detailed plot on the relationship between critical path delay and total cell area is available in Appendix A.1.2 along with the plot of the relationship between the leakage power and total cell area.

Figure A.2 and A.3 shows the data distribution of the dataset provided by [17]. Figure

Table A.2: Total Number of Gate for Post-Synthesis Stage

Design	dfxtp	a32oi	o41a	or2	conb	o21a	mux4	maj3	a2111oi	o41ai	nor2	a22oi	and4	a2bb2oi	buf	dlxtn	o21ai	o22a	a41o	a32o	a21o	o32a
ac97_ctrl	1991	0	33	378	1	58	288	14	1	0	99	3	63	0	856	0	21	6	0	51	1644	1
des3_area	64	13	189	96	0	126	0	0	6	0	19	20	135	0	472	0	5	82	26	190	19	33
aes_core	530	0	5	3718	0	537	0	0	17	0	141	7	2473	0	2054	0	46	629	14	19	772	88
des3_perf	8808	8	12	1244	0	71	0	0	10	5	2570	1662	4011	13	8652	0	32	775	4	1714	663	16
fpu	643	5	49	1801	1	792	45	932	29	13	721	28	214	11	1539	48	271	76	40	53	569	33
mem_ctrl	761	0	11	422	2	140	0	2	2	1	189	48	185	1	763	0	47	25	6	22	545	8
i2c	11	0	0	58	1	25	0	0	0	1	27	1	15	0	82	0	4	0	0	45	25	0
ethernet	9305	3	7	2866	1	616	2364	25	2	0	381	33	168	5	4458	0	118	107	0	278	10005	38
pci	1864	2	0	1039	2	397	268	0	1	0	206	13	176	6	1675	0	78	124	0	109	2727	10
simple_spi	107	0	3	30	0	9	14	0	0	2	13	0	8	0	60	0	7	2	0	3	88	2
sasc	106	0	8	16	0	33	16	0	0	0	7	0	6	0	49	0	6	10	0	1	51	0
spi	0	0	0	69	1	82	42	10	0	0	23	1	96	4	335	0	123	114	22	1	197	3
ss_pcm	87	0	0	71	0	3	2	0	0	0	1	0	5	0	33	0	1	0	0	0	36	0
systemcdes	0	11	1	127	0	79	24	0	0	0	26	10	76	3	180	0	18	78	0	11	135	17
systemcaes	0	0	7	1251	0	503	0	0	1	3	41	27	18	0	701	0	12	176	2	64	723	3
tv80	364	3	10	658	0	219	48	26	7	2	184	17	187	0	628	0	54	85	4	46	423	14
usb_funct	1740	0	0	1211	2	273	76	201	2	0	235	9	202	0	1247	0	78	159	1	68	819	2
vga_lcd	16754	6	6	3191	2	853	4361	3	0	0	215	11	99	3	8362	0	151	116	0	605	16517	45
usb_phy	108	0	1	32	0	12	0	1	0	0	11	0	14	0	41	0	8	2	0	11	30	0
wb_dma	427	0	2	109	0	26	0	0	2	0	61	0	44	0	328	0	5	10	1	9	474	7
sum	43670	51	344	18387	13	4854	7548	1214	80	27	5170	1890	8195	46	32515	48	1085	2576	120	3300	36462	320

Design	nand4	nor4	nand2	dfrtp	a211o	nor3	or4	nand3	and2	xor2	a41oi	a21oi	a21bo	a21boi	a2111o	or3	and3	dfstp	o32ai	inv	xnor2	a2bb2o
ac97_ctrl	0	0	79	212	74	18	133	7	2272	16	0	5	5	7	0	20	343	8	0	104	26	1
des3_area	15	3	30	0	71	9	371	15	342	46	3	5	1	0	15	56	343	0	1	7	65	0
aes_core	51	24	148	0	79	37	594	69	1090	262	1	50	106	1	14	860	639	0	6	175	553	52
des3_perf	389	536	1134	0	6734	347	5825	195	10508	1164	1	47	28	5	56	863	9386	0	2	3440	1026	343
fpu	33	25	836	0	189	93	385	134	2113	490	6	369	250	58	32	273	319	0	3	757	1707	56
mem_ctrl	14	16	135	199	208	39	416	16	1417	28	1	114	87	26	4	113	219	91	2	168	58	32
i2c	0	0	16	94	8	7	44	4	73	0	0	12	6	3	0	21	17	24	0	28	1	15
ethernet	8	10	298	1138	329	87	386	31	12261	68	0	102	97	18	10	321	797	101	4	205	61	55
pci	2	6	187	1244	231	36	151	12	3220	58	0	64	46	11	1	79	314	112	1	146	116	33
simple_spi	0	0	21	24	15	4	10	2	139	2	0	3	8	1	0	8	24	0	0	13	4	4
sasc	1	1	14	11	2	7	25	1	66	5	0	3	3	3	0	3	9	1	0	14	4	1
spi	3	0	46	213	38	1	62	7	327	3	1	116	11	20	32	12	54	16	1	20	26	6
ss_pcm	0	0	1	0	13	1	1	0	64	0	1	1	0	0	0	19	48	0	0	11	0	1
systemcdes	7	7	32	189	59	22	61	14	209	45	0	31	11	2	6	93	54	1	2	41	63	28
systemcaes	1	1	55	670	234	2	106	2	869	286	0	10	44	23	5	14	450	0	0	29	197	8
tv80	7	10	123	0	233	38	244	30	917	47	1	70	112	17	16	106	303	0	2	74	66	40
usb_funct	3	11	224	0	467	66	282	15	1405	111	0	77	64	9	0	96	604	0	0	291	321	51
vga_lcd	0	1	136	305	561	55	164	14	18058	28	0	79	85	11	2	174	864	0	5	85	46	44
usb_phy	0	0	17	0	2	6	7	3	50	0	0	6	4	2	1	6	23	0	1	17	0	2
wb_dma	4	4	24	93	131	17	78	6	759	14	1	25	17	4	1	21	145	1	1	42	1	3
sum	538	655	3556	4392	9678	892	9345	577	56159	2673	16	1189	985	221	195	3158	14955	355	31	5667	4341	775

Table A.3: Total Number of Gate for Post-Floorplan Stage

Design	dfxtp	a32oi	o41a	or2	conb	o21a	mux4	maj3	a2111oi	o41ai	nor2	decap	a22oi	and4	a2bb2oi	buf	dlxtn	o21ai	o22a	a41o	a32o	a21o	o32a
ac97_ctrl		0	33	378	1	58	288	14	1	0	99	367	3	63	0	16	0	21	6	0	51	1644	1
des3_area	64	13	189	96	0	126	0	0	6	0	19	199.3333333	20	135	0	21	0	5	82	26	190	19	33
aes_core	530	0	5	3718	0	537	0	0	17	0	141	441.6666667	7	2473	0	1	0	46	629	14	19	772	88
des3_perf	8808	8	12	1244	0	71	0	0	10	5	2570	1007.666667	1662	4011	13	1	0	32	775	4	1714	663	16
fpu	643	5	49	1801	1	792	45	932	29	13	721	488.3333333	28	214	11	20	48	271	76	40	53	569	33
mem_ctrl	761	0	11	422	2	140	0	2	2	1	189	303.3333333	48	185	1	66	0	47	25	6	22	545	8
i2c	11	0	0	58	1	25	0	0	0	1	27	102.3333333	1	15	0	8	0	4	0	0	45	25	0
ethernet	9305	3	7	2866	1	616	2364	25	2	0	381	845	33	168	5	78	0	118	107	0	278	10005	38
pci	1864	2	0	1039	2	397	268	0	1	0	206	474	13	176	6	101	0	78	124	0	109	2727	10
simple_spi	107	0	3	30	0	9	14	0	0	2	13	98	0	8	0	5	0	7	2	0	3	88	2
sasc	106	0	8	16	0	33	16	0	0	0	7	89	0	6	0	4	0	6	10	0	1	51	0
spi	0	0	0	69	1	82	42	10	0	0	23	177.6666667	1	96	4	22	0	123	114	22	1	197	3
ss_pcm	87	0	0	71	0	3	2	0	0	0	1	75	0	5	0	5	0	1	0	0	0	36	0
systemcdes	0	11	1	127	0	79	24	0	0	0	26	168	10	76	3	2	0	18	78	0	11	135	17
systemcaes	0	0	7	1251	0	503	0	0	1	3	41	310.6666667	27	18	0	1	0	12	176	2	64	723	3
tv80	364	3	10	658	0	219	48	26	7	2	184	267	17	187	0	3	0	54	85	4	46	423	14
usb_funct	1740	0	0	1211	2	273	76	201	2	0	235	387	9	202	0	63	0	78	159	1	68	819	2
vga_lcd	16754	6	6	3191	2	853	4361	3	0	0	215	1047.666667	11	99	3	54	0	151	116	0	605	16517	45
usb_phy	108	0	1	32	0	12	0	1	0	0	11	80.33333333	0	14	0	4	0	8	2	0	11	30	0
wb_dma	427	0	2	109	0	26	0	0	2	0	61	202.3333333	0	44	0	6	0	5	10	1	9	474	7
sum	43670	51	344	18387	13	4854	7548	1214	80	27	5170	7131.333333	1890	8195	46	481	48	1085	2576	120	3300	36462	320

Design	nand4	nor4	nand2	dfrtp	a211o	nor3	or4	nand3	and2	xor2	a41oi	a21oi	a21bo	a21boi	a2111o	or3	and3	dfstp	o32ai	inv	xnor2
ac97_ctrl	0	0	79	212	74	18	133	7	2272	16	0	5	5	7	0	20	343	8	0	104	26
des3_area	15	3	30	0	71	9	371	15	342	46	3	5	1	0	15	56	343	0	1	7	65
aes_core	51	24	148	0	79	37	594	69	1090	262	1	50	106	1	14	860	639	0	6	175	553
des3_perf	389	536	1134	0	6734	347	5825	195	10508	1164	1	47	28	5	56	863	9386	0	2	3440	1026
fpu	33	25	836	0	189	93	385	134	2113	490	6	369	250	58	32	273	319	0	3	757	1707
mem_ctrl	14	16	135	199	208	39	416	16	1417	28	1	114	87	26	4	113	219	91	2	168	58
i2c	0	0	16	94	8	7	44	4	73	0	0	12	6	3	0	21	17	24	0	28	1
ethernet	8	10	298	1138	329	87	386	31	12261	68	0	102	97	18	10	321	797	101	4	205	61
pci	2	6	187	1244	231	36	151	12	3220	58	0	64	46	11	1	79	314	112	1	146	116
simple_spi	0	0	21	24	15	4	10	2	139	2	0	3	8	1	0	8	24	0	0	13	4
sasc	1	1	14	11	2	7	25	1	66	5	0	3	3	3	0	3	9	1	0	14	4
spi	3	0	46	213	38	1	62	7	327	3	1	116	11	20	32	12	54	16	1	20	26
ss_pcm	0	0	1	0	13	1	1	0	64	0	1	1	0	0	0	19	48	0	0	11	0
systemcdes	7	7	32	189	59	22	61	14	209	45	0	31	11	2	6	93	54	1	2	41	63
systemcaes	1	1	55	670	234	2	106	2	869	286	0	10	44	23	5	14	450	0	0	29	197
tv80	7	10	123	0	233	38	244	30	917	47	1	70	112	17	16	106	303	0	2	74	66
usb_funct	3	11	224	0	467	66	282	15	1405	111	0	77	64	9	0	96	604	0	0	291	321
vga_lcd	0	1	136	305	561	55	164	14	18058	28	0	79	85	11	2	174	864	0	5	85	46
usb_phy	0	0	17	0	2	6	7	3	50	0	0	6	4	2	1	6	23	0	1	17	0
wb_dma	4	4	24	93	131	17	78	6	759	14	1	25	17	4	1	21	145	1	1	42	1
sum	538	655	3556	4392	9678	892	9345	577	56159	2673	16	1189	985	221	195	3158	14955	355	31	5667	4341

Table A.4: Total Number of Gate for Post-Placement Stage

Design	dfxtp	a32oi	o41a	or2	conb	o21a	mux4	maj3	a2111oi	o41ai	nor2	a22oi	and4	a2bb2oi	buf	dlxtn	o21ai	o22a	a41o	a32o	a21o	o32a
ac97_ctrl	1991	0	33	378	1	58	288	14	1	0	99	3	63	0	233.375	0	21	6	0	51	1644	1
des3_area	64	13	189	96	0	126	0	0	6	0	19	20	135	0	60.875	0	5	82	26	190	19	33
aes_core	530	0	5	3718	0	537	0	0	17	0	141	7	2473	0	178.125	0	46	629	14	19	772	88
des3_perf	8808	8	12	1244	0	71	0	0	10	5	2570	1662	4011	13	1055.166667	0	32	775	4	1714	663	16
fpu	643	5	49	1801	3	792	45	932	29	13	721	28	214	11	577.375	48	271	76	40	53	569	33
mem_ctrl	761	0	11	422	8	140	0	2	2	1	189	48	185	1	217.2916667	0	47	25	6	22	545	8
i2c	11	0	0	58	2	25	0	0	0	1	27	1	15	0	10.125	0	4	0	0	45	25	0
ethernet	9305	3	7	2866	2	616	2364	25	2	0	381	33	168	5	2705.541667	0	118	107	0	278	10005	38
pci	1864	2	0	1039	8	397	268	0	1	0	206	13	176	6	873.3333333	0	78	124	0	109	2727	10
simple_spi	107	0	3	30	0	9	14	0	0	2	13	0	8	0	5	0	7	2	0	3	88	2
sasc	106	0	8	16	0	33	16	0	0	0	7	0	6	0	4	0	6	10	0	1	51	0
spi	0	0	0	69	1	82	42	10	0	0	23	1	96	4	69.375	0	123	114	22	1	197	3
ss_pcm	87	0	0	71	0	3	2	0	0	0	1	0	5	0	6.125	0	1	0	0	0	36	0
systemcdes	0	11	1	127	0	79	24	0	0	0	26	10	76	3	19.33333333	0	18	78	0	11	135	17
systemcaes	0	0	7	1251	0	503	0	0	1	3	41	27	18	0	411.25	0	12	176	2	64	723	3
tv80	364	3	10	658	0	219	48	26	7	2	184	17	187	0	69.66666667	0	54	85	4	46	423	14
usb_funct	1740	0	0	1211	20	273	76	201	2	0	235	9	202	0	490.7083333	0	78	159	1	68	819	2
vga_lcd	16754	6	6	3191	9	853	4361	3	0	0	215	11	99	3	4363.5	0	151	116	0	605	16517	45
usb_phy	108	0	1	32	0	12	0	1	0	0	11	0	14	0	4	0	8	2	0	11	30	0
wb_dma	427	0	2	109	0	26	0	0	2	0	61	0	44	0	121.3333333	0	5	10	1	9	474	7
sum	43670	51	344	18387	54	4854	7548	1214	80	27	5170	1890	8195	46	11475.5	48	1085	2576	120	3300	36462	320

Design	nand4	nor4	nand2	dfrtp	a211o	nor3	or4	nand3	and2	xor2	a41oi	a21oi	a21bo	a21boi	a2111o	or3	and3	dfstp	o32ai	inv	xnor2	a2bb2o
ac97_ctrl	0	0	79	212	74	18	133	7	2272	16	0	5	5	7	0	20	343	8	0	104	26	1
des3_area	15	3	30	0	71	9	371	15	342	46	3	5	1	0	15	56	343	0	1	7	65	0
aes_core	51	24	148	0	79	37	594	69	1090	262	1	50	106	1	14	860	639	0	6	175	553	52
des3_perf	389	536	1134	0	6734	347	5825	195	10508	1164	1	47	28	5	56	863	9386	0	2	3440	1026	343
fpu	33	25	836	0	189	93	385	134	2113	490	6	369	250	58	32	273	319	0	3	757	1707	56
mem_ctrl	14	16	135	199	208	39	416	16	1417	28	1	114	87	26	4	113	219	91	2	168	58	32
i2c	0	0	16	94	8	7	44	4	73	0	0	12	6	3	0	21	17	24	0	28	1	15
ethernet	8	10	298	1138	329	87	386	31	12261	68	0	102	97	18	10	321	797	101	4	205	61	55
pci	2	6	187	1244	231	36	151	12	3220	58	0	64	46	11	1	79	314	112	1	146	116	33
simple_spi	0	0	21	24	15	4	10	2	139	2	0	3	8	1	0	8	24	0	0	13	4	4
sasc	1	1	14	11	2	7	25	1	66	5	0	3	3	3	0	3	9	1	0	14	4	1
spi	3	0	46	213	38	1	62	7	327	3	1	116	11	20	32	12	54	16	1	20	26	6
ss_pcm	0	0	1	0	13	1	1	0	64	0	1	1	0	0	0	19	48	0	0	11	0	1
systemcdes	7	7	32	189	59	22	61	14	209	45	0	31	11	2	6	93	54	1	2	41	63	28
systemcaes	1	1	55	670	234	2	106	2	869	286	0	10	44	23	5	14	450	0	0	29	197	8
tv80	7	10	123	0	233	38	244	30	917	47	1	70	112	17	16	106	303	0	2	74	66	40
usb_funct	3	11	224	0	467	66	282	15	1405	111	0	77	64	9	0	96	604	0	0	291	321	51
vga_lcd	0	1	136	305	561	55	164	14	18058	28	0	79	85	11	2	174	864	0	5	85	46	44
usb_phy	0	0	17	0	2	6	7	3	50	0	0	6	4	2	1	6	23	0	1	17	0	2
wb_dma	4	4	24	93	131	17	78	6	759	14	1	25	17	4	1	21	145	1	1	42	1	3
sum	538	655	3556	4392	9678	892	9345	577	56159	2673	16	1189	985	221	195	3158	14955	355	31	5667	4341	775

Table A.5: Total Number of Gate for Post-Cts Stage

Design	dfxtp	a32oi	o41a	or2	conb	o21a	mux4	maj3	a2111oi	o41ai	nor2	a22oi	and4	a2bb2oi	buf	dlxtn	o21ai	o22a	a41o	a32o	a21o	o32a
ac97_ctrl	1991	0	33	378	1	58	288	14	1	0	99	3	63	0	235.375	0	21	6	0	51	1644	1
des3_area	64	13	189	96	0	126	0	0	6	0	19	20	135	0	61.875	0	5	82	26	190	19	33
aes_core	530	0	5	3718	0	537	0	0	17	0	141	7	2473	0	191.0833333	0	46	629	14	19	772	88
des3_perf	8808	8	12	1244	0	71	0	0	10	5	2570	1662	4011	13	1423.041667	0	32	775	4	1714	663	16
fpu	643	5	49	1801	3	792	45	932	29	13	721	28	214	11	599.875	48	271	76	40	53	569	33
mem_ctrl	761	0	11	422	8	140	0	2	2	1	189	48	185	1	240.625	0	47	25	6	22	545	8
i2c	11	0	0	58	2	25	0	0	0	1	27	1	15	0	13.79166667	0	4	0	0	45	25	0
ethernet	9305	3	7	2866	2	616	2364	25	2	0	381	33	168	5	2898.708333	0	118	107	0	278	10005	38
pci	1864	2	0	1039	8	397	268	0	1	0	206	13	176	6	948.7083333	0	78	124	0	109	2727	10
simple_spi	107	0	3	30	0	9	14	0	0	2	13	0	8	0	6	0	7	2	0	3	88	2
sasc	106	0	8	16	0	33	16	0	0	0	7	0	6	0	5	0	6	10	0	1	51	0
spi	0	0	0	69	1	82	42	10	0	0	23	1	96	4	81.04166667	0	123	114	22	1	197	3
ss_pcm	87	0	0	71	0	3	2	0	0	0	1	0	5	0	8.125	0	1	0	0	0	36	0
systemcdes	0	11	1	127	0	79	24	0	0	0	26	10	76	3	28.33333333	0	18	78	0	11	135	17
systemcaes	0	0	7	1251	0	503	0	0	1	3	41	27	18	0	445.125	0	12	176	2	64	723	3
tv80	364	3	10	658	0	219	48	26	7	2	184	17	187	0	81.625	0	54	85	4	46	423	14
usb_funct	1740	0	0	1211	20	273	76	201	2	0	235	9	202	0	493.5	0	78	159	1	68	819	2
vga_lcd	16754	6	6	3191	9	853	4361	3	0	0	215	11	99	3	4717.625	0	151	116	0	605	16517	45
usb_phy	108	0	1	32	0	12	0	1	0	0	11	0	14	0	6.333333333	0	8	2	0	11	30	0
wb_dma	427	0	2	109	0	26	0	0	2	0	61	0	44	0	133	0	5	10	1	9	474	7
sum	43670	51	344	18387	54	4854	7548	1214	80	27	5170	1890	8195	46	12618.79167	48	1085	2576	120	3300	36462	320

Design	nand4	nor4	nand2	dfrtp	a211o	nor3	or4	nand3	and2	xor2	a41oi	a21oi	a21bo	clkbuf	a21boi	a2111o	or3	and3	dfstp	o32ai	inv	xnor2	a2bb2o
ac97_ctrl	0	0	79	212	74	18	133	7	2272	16	0	5	5	455	7	0	20	343	8	0	104	26	1
des3_area	15	3	30	0	71	9	371	15	342	46	3	5	1	14	0	15	56	343	0	1	7	65	0
aes_core	51	24	148	0	79	37	594	69	1090	262	1	50	106	129.6666667	1	14	860	639	0	6	175	553	52
des3_perf	389	536	1134	0	6734	347	5825	195	10508	1164	1	47	28	1702	5	56	863	9386	0	2	3440	1026	343
fpu	33	25	836	0	189	93	385	134	2113	490	6	369	250	130.6666667	58	32	273	319	0	3	757	1707	56
mem_ctrl	14	16	135	199	208	39	416	16	1417	28	1	114	87	193.6666667	26	4	113	219	91	2	168	58	32
i2c	0	0	16	94	8	7	44	4	73	0	0	12	6	24.66666667	3	0	21	17	24	0	28	1	15
ethernet	8	10	298	1138	329	87	386	31	12261	68	0	102	97	1644	18	10	321	797	101	4	205	61	55
pci	2	6	187	1244	231	36	151	12	3220	58	0	64	46	194	11	1	79	314	112	1	146	116	33
simple_spi	0	0	21	24	15	4	10	2	139	2	0	3	8	27.33333333	1	0	8	24	0	0	13	4	4
sasc	1	1	14	11	2	7	25	1	66	5	0	3	3	27.41666667	3	0	3	9	1	0	14	4	1
spi	3	0	46	213	38	1	62	7	327	3	1	116	11	44.66666667	20	32	12	54	16	1	20	26	6
ss_pcm	0	0	1	0	13	1	1	0	64	0	1	1	0	10.75	0	0	19	48	0	0	11	0	1
systemcdes	7	7	32	189	59	22	61	14	209	45	0	31	11	19.33333333	2	6	93	54	1	2	41	63	28
systemcaes	1	1	55	670	234	2	106	2	869	286	0	10	44	88.25	23	5	14	450	0	0	29	197	8
tv80	7	10	123	0	233	38	244	30	917	47	1	70	112	52.66666667	17	16	106	303	0	2	74	66	40
usb_funct	3	11	224	0	467	66	282	15	1405	111	0	77	64	37.33333333	9	0	96	604	0	0	291	321	51
vga_lcd	0	1	136	305	561	55	164	14	18058	28	0	79	85	3404.333333	11	2	174	864	0	5	85	46	44
usb_phy	0	0	17	0	2	6	7	3	50	0	0	6	4	10.66666667	2	1	6	23	0	1	17	0	2
wb_dma	4	4	24	93	131	17	78	6	759	14	1	25	17	105.6666667	4	1	21	145	1	1	42	1	3
sum	538	655	3556	4392	9678	892	9345	577	56159	2673	16	1189	985	8316.083333	221	195	3158	14955	355	31	5667	4341	775

Table A.6: Total Number of Gate for Post-Route Stage

Design	dfxtp	a32oi	o41a	or2	conb	o21a	mux4	maj3	a2111oi	o41ai	nor2	a22oi	and4	a2bb2oi	buf	dlxtn	o21ai	o22a	a41o	a32o	a21o	o32a
ac97_ctrl	1991	0	33	378	1	58	288	14	1	0	99	3	63	0	235.375	0	21	6	0	51	1644	1
des3_area	64	13	189	96	0	126	0	0	6	0	19	20	135	0	61.875	0	5	82	26	190	19	33
aes_core	530	0	5	3718	0	537	0	0	17	0	141	7	2473	0	191.0833333	0	46	629	14	19	772	88
des3_perf	8808	8	12	1244	0	71	0	0	10	5	2570	1662	4011	13	1423.041667	0	32	775	4	1714	663	16
fpu	643	5	49	1801	3	792	45	932	29	13	721	28	214	11	599.875	48	271	76	40	53	569	33
mem_ctrl	761	0	11	422	8	140	0	2	2	1	189	48	185	1	240.625	0	47	25	6	22	545	8
i2c	11	0	0	58	2	25	0	0	0	1	27	1	15	0	13.79166667	0	4	0	0	45	25	0
ethernet	9305	3	7	2866	2	616	2364	25	2	0	381	33	168	5	2898.708333	0	118	107	0	278	10005	38
pci	1864	2	0	1039	8	397	268	0	1	0	206	13	176	6	948.7083333	0	78	124	0	109	2727	10
simple_spi	107	0	3	30	0	9	14	0	0	2	13	0	8	0	6	0	7	2	0	3	88	2
sasc	106	0	8	16	0	33	16	0	0	0	7	0	6	0	5	0	6	10	0	1	51	0
spi	0	0	0	69	1	82	42	10	0	0	23	1	96	4	81.04166667	0	123	114	22	1	197	3
ss_pcm	87	0	0	71	0	3	2	0	0	0	1	0	5	0	8.125	0	1	0	0	0	36	0
systemcdes	0	11	1	127	0	79	24	0	0	0	26	10	76	3	28.33333333	0	18	78	0	11	135	17
systemcaes	0	0	7	1251	0	503	0	0	1	3	41	27	18	0	445.125	0	12	176	2	64	723	3
tv80	364	3	10	658	0	219	48	26	7	2	184	17	187	0	81.625	0	54	85	4	46	423	14
usb_funct	1740	0	0	1211	20	273	76	201	2	0	235	9	202	0	493.5	0	78	159	1	68	819	2
vga_lcd	16754	6	6	3191	9	853	4361	3	0	0	215	11	99	3	4717.625	0	151	116	0	605	16517	45
usb_phy	108	0	1	32	0	12	0	1	0	0	11	0	14	0	6.333333333	0	8	2	0	11	30	0
wb_dma	427	0	2	109	0	26	0	0	2	0	61	0	44	0	133	0	5	10	1	9	474	7
sum	43670	51	344	18387	54	4854	7548	1214	80	27	5170	1890	8195	46	12618.79167	48	1085	2576	120	3300	36462	320

Design	nand4	nor4	nand2	dfrtp	a211o	nor3	or4	nand3	and2	xor2	a41oi	a21oi	a21bo	clkbuf	a21boi	a2111o	or3	and3	dfstp	o32ai	inv	xnor2	a2bb2o
ac97_ctrl	0	0	79	212	74	18	133	7	2272	16	0	5	5	455	7	0	20	343	8	0	104	26	1
des3_area	15	3	30	0	71	9	371	15	342	46	3	5	1	14	0	15	56	343	0	1	7	65	0
aes_core	51	24	148	0	79	37	594	69	1090	262	1	50	106	129.6666667	1	14	860	639	0	6	175	553	52
des3_perf	389	536	1134	0	6734	347	5825	195	10508	1164	1	47	28	1702	5	56	863	9386	0	2	3440	1026	343
fpu	33	25	836	0	189	93	385	134	2113	490	6	369	250	130.6666667	58	32	273	319	0	3	757	1707	56
mem_ctrl	14	16	135	199	208	39	416	16	1417	28	1	114	87	193.6666667	26	4	113	219	91	2	168	58	32
i2c	0	0	16	94	8	7	44	4	73	0	0	12	6	24.66666667	3	0	21	17	24	0	28	1	15
ethernet	8	10	298	1138	329	87	386	31	12261	68	0	102	97	1644	18	10	321	797	101	4	205	61	55
pci	2	6	187	1244	231	36	151	12	3220	58	0	64	46	194	11	1	79	314	112	1	146	116	33
simple_spi	0	0	21	24	15	4	10	2	139	2	0	3	8	27.33333333	1	0	8	24	0	0	13	4	4
sasc	1	1	14	11	2	7	25	1	66	5	0	3	3	27.41666667	3	0	3	9	1	0	14	4	1
spi	3	0	46	213	38	1	62	7	327	3	1	116	11	44.66666667	20	32	12	54	16	1	20	26	6
ss_pcm	0	0	1	0	13	1	1	0	64	0	1	1	0	10.75	0	0	19	48	0	0	11	0	1
systemcdes	7	7	32	189	59	22	61	14	209	45	0	31	11	19.33333333	2	6	93	54	1	2	41	63	28
systemcaes	1	1	55	670	234	2	106	2	869	286	0	10	44	88.25	23	5	14	450	0	0	29	197	8
tv80	7	10	123	0	233	38	244	30	917	47	1	70	112	52.66666667	17	16	106	303	0	2	74	66	40
usb_funct	3	11	224	0	467	66	282	15	1405	111	0	77	64	37.33333333	9	0	96	604	0	0	291	321	51
vga_lcd	0	1	136	305	561	55	164	14	18058	28	0	79	85	3404.333333	11	2	174	864	0	5	85	46	44
usb_phy	0	0	17	0	2	6	7	3	50	0	0	6	4	10.66666667	2	1	6	23	0	1	17	0	2
wb_dma	4	4	24	93	131	17	78	6	759	14	1	25	17	105.6666667	4	1	21	145	1	1	42	1	3
sum	538	655	3556	4392	9678	892	9345	577	56159	2673	16	1189	985	8316.083333	221	195	3158	14955	355	31	5667	4341	775

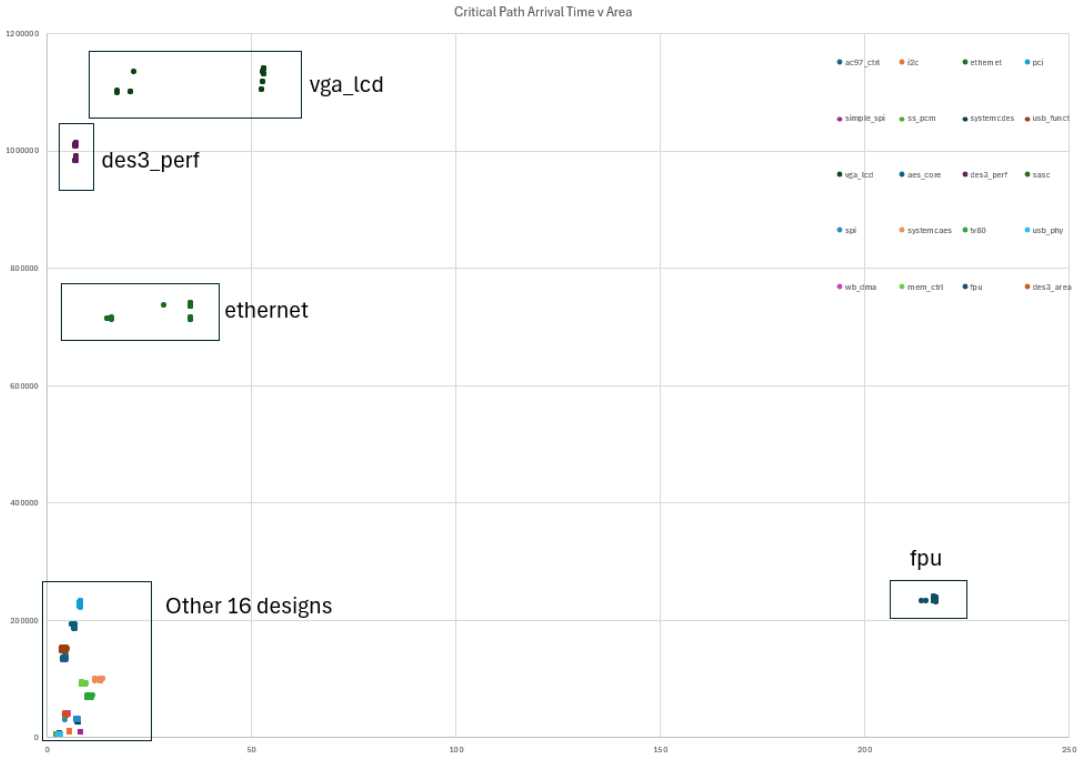


Figure A.1: Critical Path Arrival Time and Total Area Relationship [17]

Table A.7: Number of Gate Reduction between EDA stage for standard cell

Design	Synth2floorplan	Floorplan2place					Synth2route		
	buf	conb	decap	buf	inv	tap	conb	buf	inv
ac97_ctrl	840	1	367	2.875	4.125	4463.833	1	842.875	4.125
des3_area	451	0	199.3333	16.625	0	1326.333	0	467.625	0
aes_core	2053	0	441.6667	0.583333	4	6525.667	0	2053.583	4
des3_perf	8651	0	1007.667	1	0	33955.5	0	8652	0
fpu	1519	1	488.3333	0	20.125	7925	1	1519	20.125
mem_ctrl	697	2	303.3333	40.25	1	3070.833	2	737.25	1
i2c	74	1	102.3333	4.833333	0	342.5	1	78.83333	0
ethernet	4380	1	845	55.70833	1.708333	23887.67	1	4435.708	1.708333
pci	1574	2	474	13.79167	1.708333	7499.833	2	1587.792	1.708333
simple_spi	55	0	98	5	0	318	0	60	0
sasc	45	0	89	3	0	258.6667	0	48	0
spi	313	1	177.6667	19.375	0	1047.833	1	332.375	0
ss_pcm	28	0	75	4.75	1	186.8333	0	32.75	1
systemcdes	178	0	168	1.791667	1.541667	941.8333	0	179.7917	1.541667
systemcaes	700	0	310.6667	0	0.166667	3222	0	700	0.166667
tv80	625	0	267	1.75	1.208333	2360	0	626.75	1.208333
usb_funct	1184	2	387	18.25	1.125	4983	2	1202.25	1.125
vga_lcd	8308	2	1047.667	35.5	2.291667	36701.83	2	8343.5	2.291667
usb_phy	37	0	80.33333	1	0	210.3333	0	38	0
wb_dma	322	0	202.3333	4.291667	0.166667	1353.833	0	326.2917	0.166667
sum	32034	13	7131.333	230.375	40.16667	140581.3	13	32264.38	40.16667

A.2 shows the relationship between area and the critical path delay, while Figure A.3 shows the relationship between area and the leakage power. Figure A.4 and A.5 are added to analyze the clustered designs in each plot. These plots help us divide the dataset as mentioned in 6.2.

Table A.9, A.10, and A.11 shows the Wasserstein distance between each design compared with ac97_ctrl design. Table A.9 is the Wasserstein distance without normalizing the PPA metrics while Table A.10 and A.11 are the Wasserstein distance with minmax feature scaling and Z-score normalization. From these three tables, we chose the five designs with the highest value of Wasserstein distance: aes_core, des3_perf, fpu, ethernet, and vga_lcd. These designs will be used to test the GNN trained with the other 15 designs.

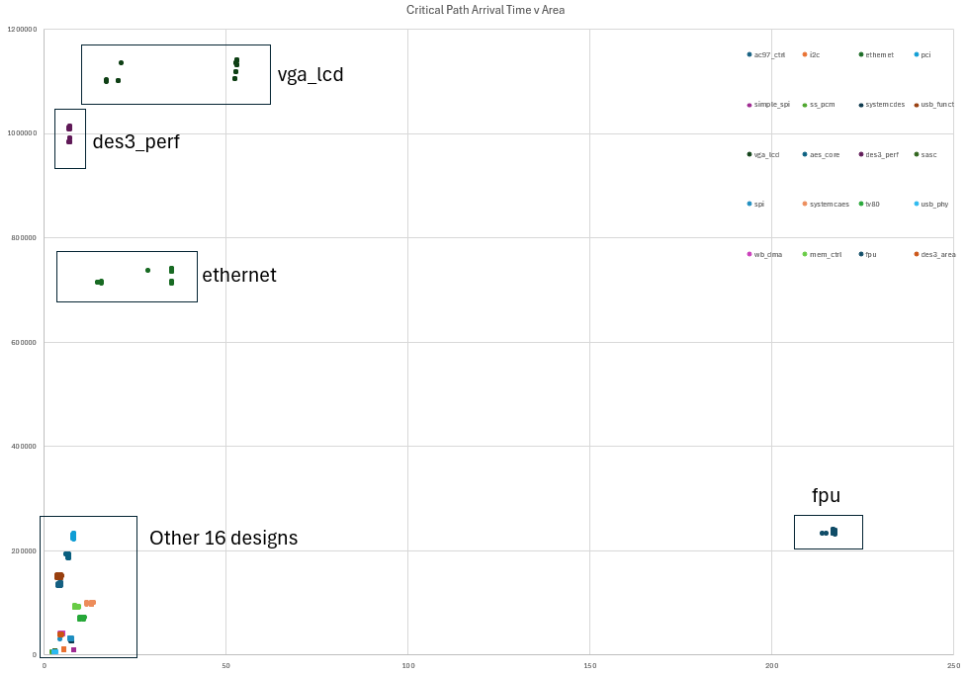


Figure A.2: Critical Path Delay and Total Area Relationship [17]

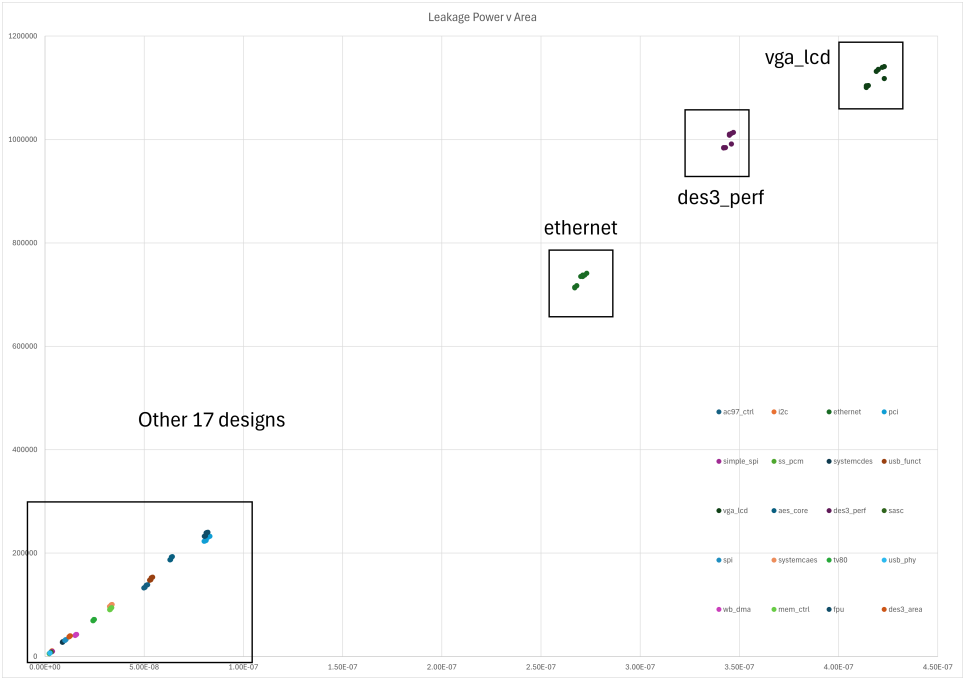


Figure A.3: Leakage Power and Total Area Relationship [17]

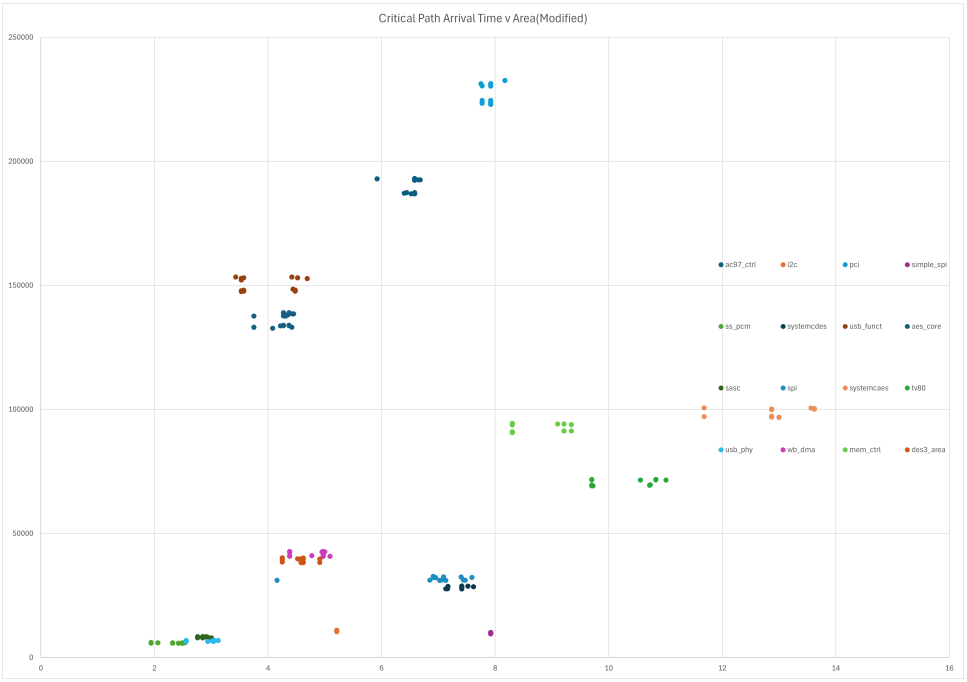


Figure A.4: Other 16 designs Critical Path Delay and Total Area Relationship [17]

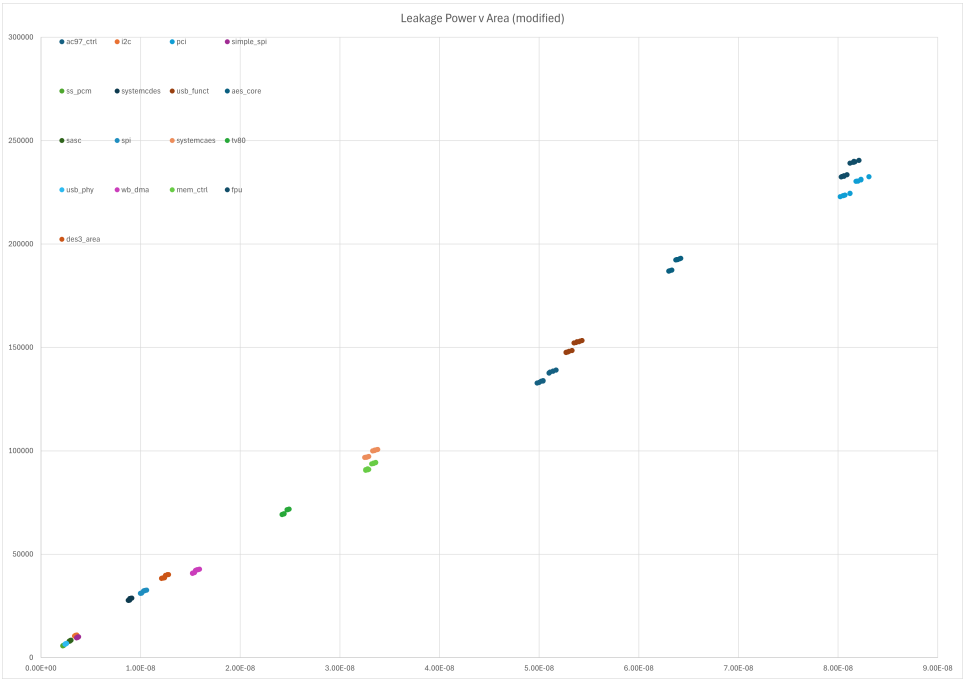


Figure A.5: Other 17 Designs Leakage Power and Total Area Relationship [17]

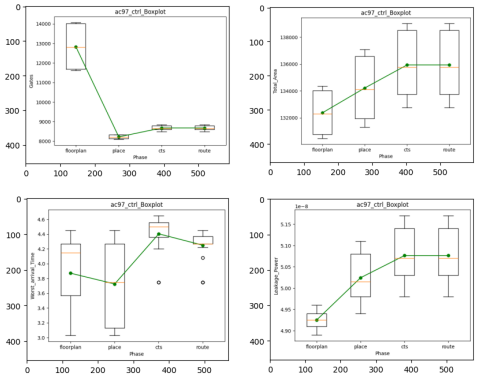
Table A.8: Wasserstein distance between each design and `ac97_ctrl`, using different normalization methods.

Design	Raw	Min-max Normalization	Z-score Normalization
ac97_ctrl	0.0000	0.0000	0.0000
des3_area	0.3100	0.6900	0.4480
aes_core	2.2550	**9.2810**	0.3550
des3_perf	2.6470	**11.2150**	0.4690
fpu	**212.6010**	**8.1060**	0.3360
mem_ctrl	4.4040	3.3280	0.8220
i2c	0.9610	1.0900	NaN
ethernet	**25.5260**	4.3180	0.4810
pci	3.6570	1.9560	0.6370
simple_spi	3.6700	2.5560	1.3580
sasc	1.4080	2.7950	0.5330
spi	2.8100	1.8660	0.4260
ss_pcm	1.8980	3.9610	0.3600
systemcdes	3.1510	3.2360	0.2850
systemcaes	8.5350	3.8950	0.3110
tv80	6.1540	5.2380	0.5240
usb_funct	0.3230	4.1160	0.6490
vga_lcd	**38.6760**	5.8350	0.5180
usb_phy	1.4540	4.4980	0.7130
wb_dma	0.6010	0.5540	0.3790

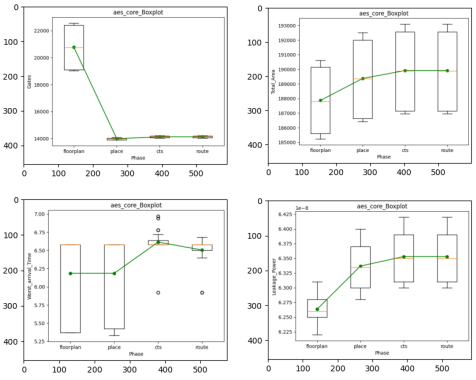
Wasserstein Metrics for Structural Changes Across EDA Stages

From Figure A.6 to A.15 illustrates the average change of PPA through EDA stages of 20 designs in [17]. The figures are plotted in box plot format to imply the overall changes of 48 samples in each design.

In addition to statistical distance, we also analyze the contribution of PPA for each physical design stage—floorplanning, placement, clock tree synthesis (CTS), and routing. Table A.12 shows the change of total area, critical path delay, and leakage power measured by the Wasserstein distance method between stages for each design, rounded to four

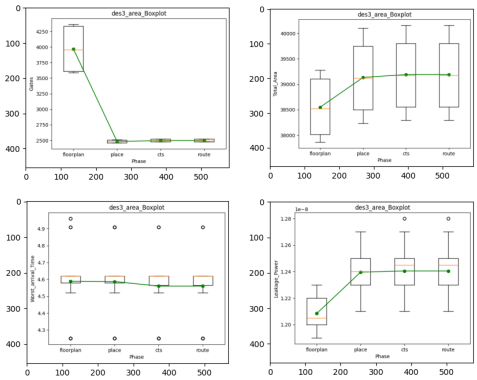


(a) ac97_ctrl PPA change through stages

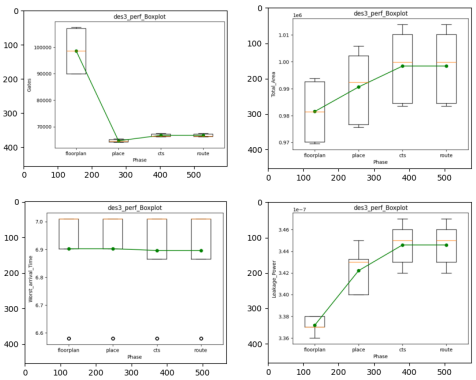


(b) aes_core PPA change through stages

Figure A.6: [17] data PPA change through stage

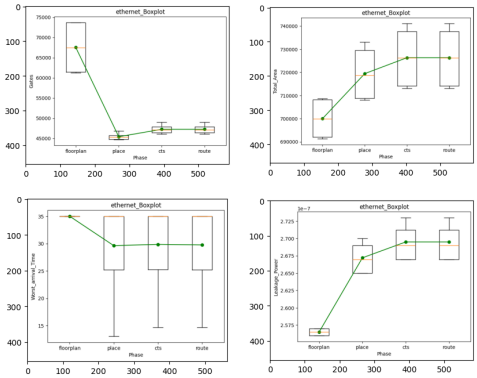


(a) des3_area PPA change through stages

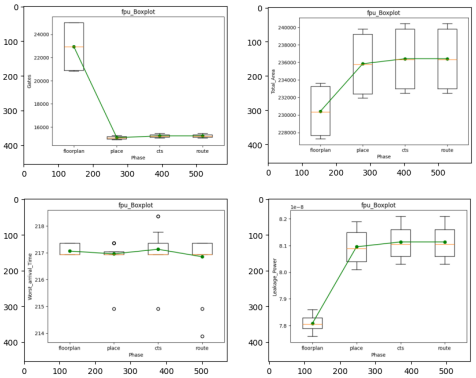


(b) des3_perf PPA change through stages

Figure A.7: [17] data PPA change through stage

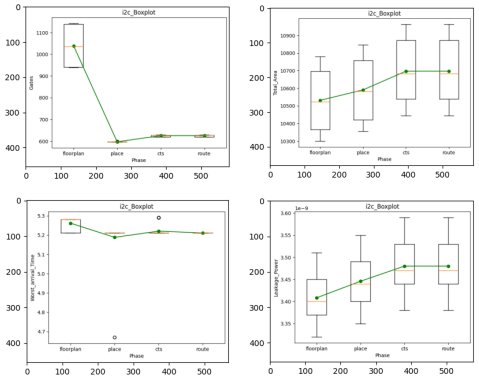


(a) ethernet PPA change through stages

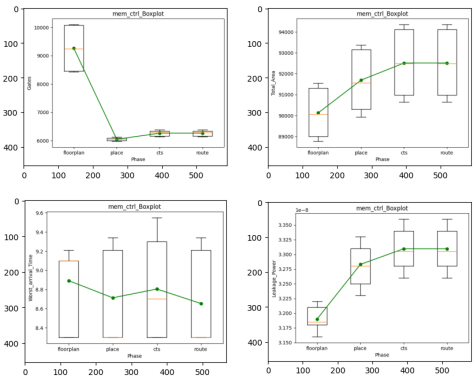


(b) fpu PPA change through stages

Figure A.8: [17] data PPA change through stage

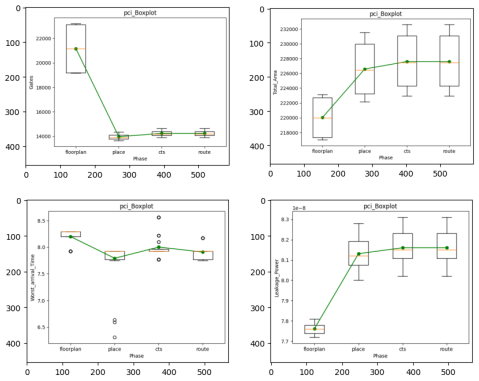


(a) i2c PPA change through stages

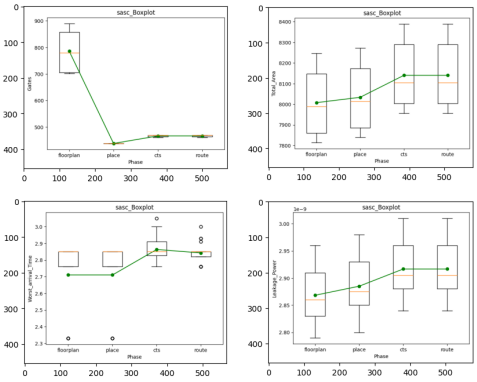


(b) mem_ctrl PPA change through stages

Figure A.9: [17] data PPA change through stage

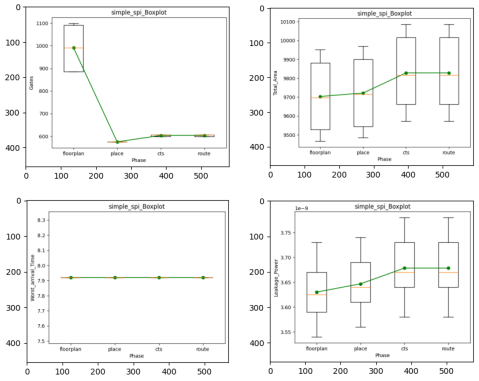


(a) pci PPA change through stages

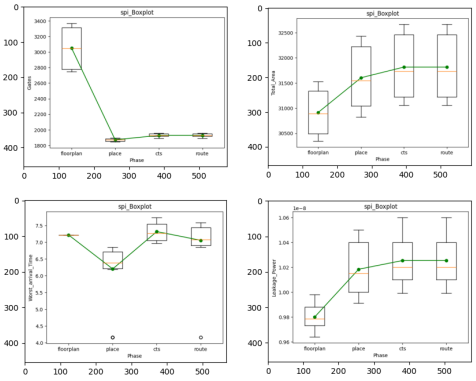


(b) sasc PPA change through stages

Figure A.10: [17] data PPA change through stage

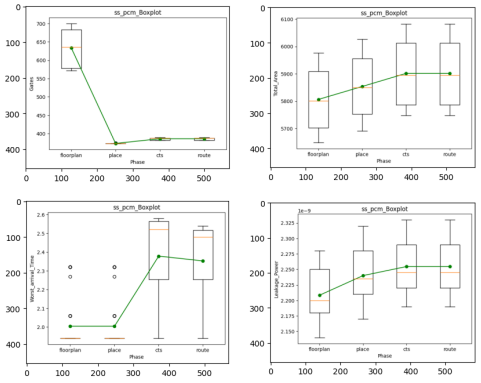


(a) simple_spi PPA change through stages

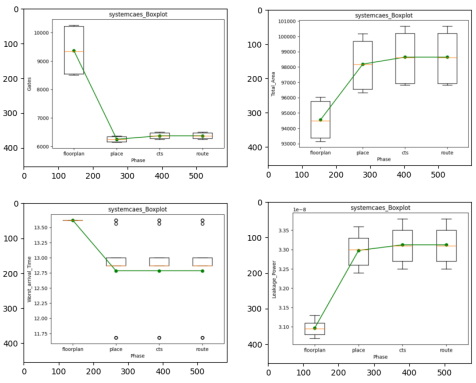


(b) spi PPA change through stages

Figure A.11: [17] data PPA change through stage

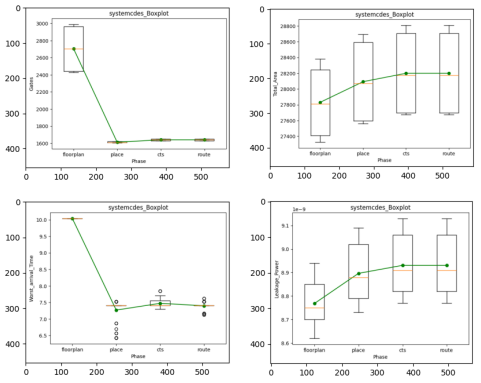


(a) ss_pcm PPA change through stages

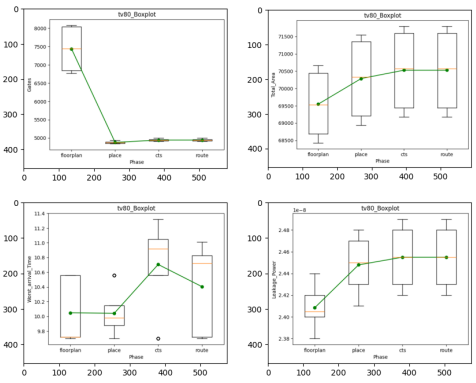


(b) systemcaes PPA change through stages

Figure A.12: [17] data PPA change through stage

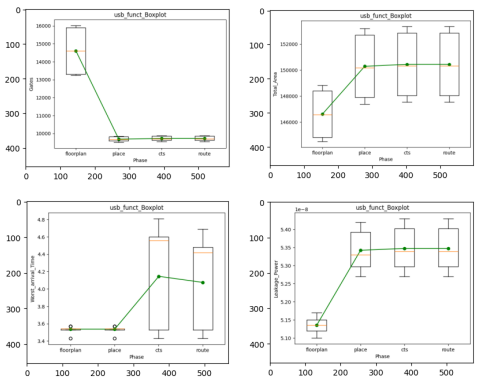


(a) systemcdes PPA change through stages

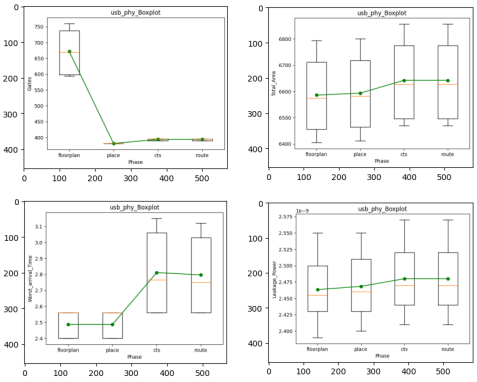


(b) tv80 PPA change through stages

Figure A.13: [17] data PPA change through stage



(a) usb_funct PPA change through stages



(b) usb_phy PPA change through stages

Figure A.14: [17] data PPA change through stage

Table A.9: Wasserstein Distance of dataset [17] on critical path delay

Design	Area	Critical Path	Leakage Power
ac97_ctrl	0.0	0.0	0.0
des3_area	96743.25	0.31	0.001
aes_core	53976.417	2.255	0.001
des3_perf	862485.667	2.647	0.001
fpu	100477.792	**212.601**	0.001
mem_ctrl	43426.125	4.404	0.001
i2c	125236.292	0.961	0.001
ethernet	590410.667	**25.526**	0.001
pci	91657.584	3.657	0.001
simple_spi	126104.959	3.67	0.001
sasc	127793.251	1.408	0.001
spi	104117.542	2.81	0.001
ss_pcm	130031.875	1.898	0.001
systemcdes	107733.709	3.151	0.001
systemcaes	37286.542	8.535	0.001
tv80	65404.917	6.154	0.001
usb_funct	14507.5	0.323	0.001
vga_lcd	985359.875	**38.676**	0.001
usb_phy	129291.125	1.454	0.001
wb_dma	94156.834	0.601	0.001

decimal places. Table A.13 shows the average standard deviation change in total area, critical path delay, and leakage power between stages for each design, rounded to four decimal places.

A.1.3 Model Structure

As shown in Figure 6.4, the model comprises a stack of message-passing layers (MPL) followed by fully connected (FC) readout heads. The objective is to predict three PPA targets—total area, critical-path delay, and leakage power—at four implementation stages (floorplan, placement, CTS, and routing).

Table A.10: Wasserstein Distance of dataset[17] normalized between 0 and 1

Design	Area	Critical Path	Leakage Power
ac97_ctrl	0.0	0.0	0.0
des3_area	0.036	0.69	0.688
aes_core	0.073	**9.281**	9.281
des3_perf	0.051	**11.215**	11.214
fpu	0.054	**8.106**	8.133
mem_ctrl	0.044	3.328	3.33
i2c	0.069	1.09	1.081
ethernet	0.038	4.318	4.318
pci	0.026	1.956	1.956
simple_spi	0.075	2.556	2.541
sasc	0.097	2.795	2.79
spi	0.04	1.866	1.862
ss_pcm	0.076	3.961	3.954
systemcdes	0.05	3.236	3.242
systemcaes	0.051	3.895	3.898
tv80	0.047	5.238	5.241
usb_funct	0.039	4.116	4.116
vga_lcd	0.063	5.835	5.836
usb_phy	0.087	4.498	4.491
wb_dma	0.027	0.554	0.552

Figure 6.5a details the MPL block. The input graph (gate- or pin-level) is first processed by node and edge encoders to obtain initial embeddings. In the **Gate-level graph** representation, the node descriptor consists of three primary node features (dimension n) and two auxiliary features (dimension m); these are concatenated to form the initial node embedding. The embeddings are then updated by a sequence of *MPL_node* blocks (Figure 6.5b), each comprising a message-passing layer, batch normalization, and a rectified linear unit (ReLU) nonlinearity. After L such updates, node embeddings are aggregated using permutation-invariant global pooling (mean and sum) to produce a graph-level representation.

Table A.11: Wasserstein Distance of dataset[17] normalized by Zscore

Design	Area	Critical Path	Leakage Power
ac97_ctrl	0.0	0.0	0.0
des3_area	0.083	0.448	0.235
aes_core	0.121	0.355	0.147
des3_perf	0.123	0.469	0.259
fpu	0.096	0.336	0.187
mem_ctrl	0.076	0.822	0.192
i2c	0.182	Nan	0.216
ethernet	0.076	0.481	0.167
pci	0.066	0.637	0.156
simple_spi	0.2	1.358	0.2
sasc	0.223	0.533	0.21
spi	0.071	0.426	0.205
ss_pcm	0.179	0.36	0.218
systemcdes	0.072	0.285	0.22
systemcaes	0.079	0.311	0.12
tv80	0.088	0.524	0.147
usb_funct	0.076	0.649	0.148
vga_lcd	0.166	0.518	0.264
usb_phy	0.197	0.713	0.222
wb_dma	0.074	0.379	0.171

The pooled representation is concatenated with graph-level features (e.g., design constraints) and passed to the FC layers, which serve as readout heads. The FC stack maps the fused representation to the desired outputs; in our implementation, a shared trunk is followed by task-specific heads that emit stage-conditioned predictions for area, critical-path delay, and leakage power.

A.1.4 Experiments

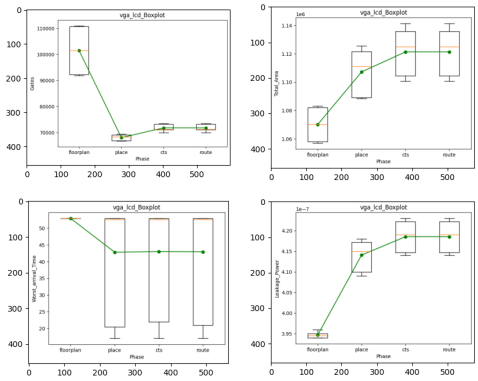
Here are the entire results of different graph structures for each GNN model, mentioned in Section 6.4.

Design	Area				Cpath				Pleak			
	floorplan2place	place2cts	cts2route	sum	floorplan2place	place2cts	cts2route	sum	floorplan2place	place2cts	cts2route	sum
ac97_ctrl	1847.4583	1715.5	0	3562.9583	0.1829	0.6785	0.1544	1.0158	9.92E-10	5.21E-10	0	1.51E-09
aes_core	1504.5417	535.5	0	2040.0417	0.006	0.429	0.11	0.545	7.33E-10	1.63E-10	0	8.96E-10
des3_area	584.75	56.125	0	640.875	0.0044	0.0271	0	0.0315	3.12E-10	8.33E-12	0	3.21E-10
des3_perf	9133.6667	7778.625	0	16912.2917	0	0.0062	0	0.0062	5.04E-09	2.38E-09	0	7.42E-09
ethernet	19459.625	6896	0	26355.625	5.3927	0.2431	0.1054	5.7413	1.07E-08	2.29E-09	0	1.30E-08
fpu	5410.2083	575.7083	0	5985.9167	0.1021	0.1708	0.2804	0.5533	2.87E-09	1.83E-10	0	3.05E-09
i2c	59.5	106.3333	0	165.8333	0.0735	0.0325	0.01	0.116	3.75E-11	3.42E-11	0	7.17E-11
mem_ctrl	1567.25	814.4167	0	2381.6667	0.2875	0.0933	0.1513	0.5321	9.29E-10	2.67E-10	0	1.20E-09
pci	6547.6667	1011.2083	0	7558.875	0.4065	0.2102	0.0946	0.7112	3.69E-09	2.92E-10	0	3.98E-09
sasc	26	106.5417	0	132.5417	0	0.1531	0.0208	0.174	1.67E-11	3.17E-11	0	4.83E-11
simple_spi	18.3333	106.6667	0	125	0	0	0	0	1.67E-11	3.17E-11	0	4.83E-11
spi	688.875	211.5417	0	900.4167	1.015	1.1163	0.2619	2.3931	3.84E-10	7.00E-11	0	4.54E-10
ss_pcm	47.2083	47.7083	0	94.9167	0	0.3744	0.0258	0.4002	3.17E-11	1.46E-11	0	4.63E-11
systemcaes	3637.9583	458.4583	0	4096.4167	0.8354	0	0	0.8354	2.01E-09	1.46E-10	0	2.16E-09
systemcdes	261.7083	106.375	0	368.0833	2.7702	0.2044	0.0737	3.0483	1.28E-10	3.42E-11	0	1.63E-10
tv80	739.3333	242.375	0	981.7083	0.1708	0.6629	0.3025	1.1363	3.96E-10	7.08E-11	0	4.67E-10
usb_funct	3681.625	151	0	3832.625	0	0.6092	0.07	0.6792	2.08E-09	5.00E-11	0	2.13E-09
usb_phy	7.6667	48.6667	0	56.3333	0	0.3246	0.015	0.3396	5.00E-12	1.17E-11	0	1.67E-11
vga_lcd	37125.1667	14107.5	0	51232.6667	10.1162	0.1742	0.085	10.3754	1.94E-08	4.46E-09	0	2.39E-08
wb_dma	1088.9167	440.2083	0	1529.125	0.1502	0.1877	0.0604	0.3983	6.17E-10	1.29E-10	0	7.46E-10

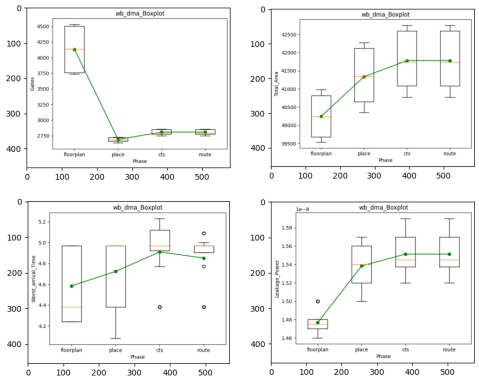
Table A.12: Wasserstein distance on PPA between stage for each design

Design	Area				Cpath				Pleak			
	floorplan2place	place2cts	cts2route	sum	floorplan2place	place2cts	cts2route	sum	floorplan2place	place2cts	cts2route	sum
ac97_ctrl	776.6341	272.8507	0	1049.4848	0.4172	0.6269	0.1057	1.1498	4.06E-10	9.12E-11	0	4.97E-10
aes_core	441.5212	28.2799	0	469.8011	0.0161	0.6404	0.1892	0.8457	2.17E-10	4.84E-11	0	2.66E-10
des3_area	179.3024	0.3307	0	179.6331	0.0151	0.1356	0	0.1507	1.09E-10	2.76E-11	0	1.37E-10
des3_perf	3141.1066	675.1546	0	3816.2612	0	0.0165	0	0.0165	1.93E-09	4.84E-10	0	2.41E-09
ethernet	3141.0862	1212.1786	0	4353.2648	8.8163	0.7048	0.288	9.8091	1.84E-09	4.55E-10	0	2.29E-09
fpu	721.0146	27.3473	0	748.3618	0.4106	0.419	0.8485	1.6782	4.07E-10	3.73E-11	0	4.44E-10
i2c	3.6856	14.1205	0	17.8061	0.116	0.1245	0.0265	0.267	4.33E-12	8.12E-12	0	1.25E-11
mem_ctrl	375.0345	164.9467	0	539.9812	0.4435	0.439	0.3281	1.2106	2.21E-10	4.71E-11	0	2.68E-10
pci	1083.6389	105.3411	0	1188.98	0.4051	0.4662	0.1654	1.0367	6.38E-10	4.93E-11	0	6.87E-10
sasc	0	12.7082	0	12.7082	0	0.2724	0.0297	0.3021	4.71E-12	3.73E-12	0	8.44E-12
simple_spi	0.4714	13.9124	0	14.3838	0	0	0	0	4.71E-12	8.98E-12	0	1.37E-11
spi	175.0677	36.4669	0	211.5345	0.8035	0.6766	0.5492	2.0293	1.01E-10	4.73E-11	0	1.48E-10
ss_pcm	2.9577	10.7954	0	13.7531	0	0.3127	0.0187	0.3314	4.71E-12	4.98E-12	0	9.70E-12
systemcaes	404.5545	62.1778	0	466.7324	0.5461	0	0	0.5461	2.59E-10	4.98E-11	0	3.09E-10
systemcdes	60.5678	14.1504	0	74.7182	0.3375	0.3433	0.0886	0.7695	3.10E-11	6.40E-12	0	3.74E-11
tv80	240.7974	4.6441	0	245.4415	0.2969	0.4888	0.3403	1.126	1.21E-10	4.55E-11	0	1.66E-10
usb_funct	670.2545	19.7927	0	690.0472	0	0.5455	0.0622	0.6078	3.74E-10	5.00E-11	0	4.24E-10
usb_phy	0.4714	11.0855	0	11.5569	0	0.3311	0.015	0.3461	5.00E-12	3.73E-12	0	8.73E-12
vga_lcd	5658.163	2243.8613	0	7902.0243	15.519	0.8107	0.2854	16.6152	3.45E-09	7.63E-10	0	4.21E-09
wb_dma	192.5891	54.4059	0	246.995	0.3186	0.3584	0.0691	0.7461	1.55E-10	5.38E-11	0	2.08E-10

Table A.13: Standard Deviation on PPA between stage for each design



(a) usb_funct PPA change through stages



(b) usb_phy PPA change through stages

Figure A.15: [17] data PPA change through stage

Guo-GNN [7] Model Evaluation

We evaluate the model provided by [7] to confirm the performance of the model is trained properly as intended by [7]. Table A.14 shows the results of Guo-GNN [7] model with dataset provided by [7] with results from the paper [7] as comparison. The R2 score results are similar, while the MAPEs are unstable. This implies the MAPE is not a suitable error calculation method for node-level or edge-level prediction in GNN.

Base Model

Gate-level Graph

Table A.15 illustrates the **Gate-level Graph** and **Pin-level Graph** results for GCN, GAT, and EGAT. Within the **Gate-level Graph** graph representation, the GAT model achieves remarkably high performance, reporting a train MAPE of 0.0770 and a test MAPE of 0.0770, along with R^2 values exceeding 0.99 in both training and test sets. This indicates near-perfect learning and generalization, suggesting that attention-based mechanisms are highly effective when node-level features and connectivity are gate-driven. In contrast, GCN performs poorly in this setting, with a test MAPE of 0.7800 and severely

Table A.14: Comparison on **arrival time / slack** prediction performance (R^2) and **runtime** (s). OpenROAD flow is the ground truth. We compare with results from [7] with the model recreation.

	Benchmark	OpenROAD (R^2)	Results from [7] (R^2)			Model Evaluation		
			Full	w/ Cell	w/ Net	Full (R^2)	Full (MAE)	Full (MAPE)
Train	blabla	1.0000	**0.9616**	0.3133	0.8949	0.9681	1.1450	0.0907
	usb_cdc_core	1.0000	**0.9751**	0.9529	0.9475	0.9846	0.1654	7268.3999
	BM64	1.0000	**0.9766**	0.9528	0.9146	0.9696	0.6834	0.1072
	salsa20	1.0000	**0.9624**	0.1709	0.8943	0.9296	1.7510	0.1842
	aes128	1.0000	0.8627	0.9081	**0.9487**	0.8659	1.4414	0.1339
	wbqspiflash	1.0000	**0.9721**	0.9711	0.9695	0.9698	0.2505	2047.8689
	cic_decimator	1.0000	**0.9840**	0.9692	0.9777	0.9860	0.1216	0.0596
	aes256	1.0000	0.8488	0.8621	**0.9477**	0.8292	1.7862	0.1436
	des	1.0000	**0.9922**	0.9879	0.9769	0.9943	0.5308	0.08123
	aes_cipher	1.0000	**0.9825**	0.8828	0.9288	0.9075	0.8004	753.3615
	picorv32a	1.0000	**0.9683**	0.9439	0.9008	0.8808	1.3656	24225.2481
	zipdiv	1.0000	**0.9753**	0.9768	0.9888	0.9702	0.3522	3681.4211
	genericfir	1.0000	0.9588	**0.9673**	0.9022	0.8513	0.5792	1180.7214
	usb	1.0000	**0.9784**	0.9645	0.9645	0.9871	0.1747	0.0900
	Avg. Train	1.0000	**0.9493**	0.9231	0.9184	0.9353	0.7962	2796.9937
Test	jpeg_encoder	1.0000	0.8820	0.7916	**0.9133**	0.0446	3.6178	759.1270
	usbf_device	1.0000	0.9252	0.8853	**0.9330**	0.6337	1.7876	8417.9834
	aes192	1.0000	0.8605	0.7928	**0.9338**	0.8530	1.6940	0.1403
	xtea	1.0000	**0.9135**	0.8207	0.8722	0.9874	0.3582	0.0883
	spm	1.0000	**0.9257**	0.9157	0.8724	0.9882	0.0936	0.07637
	y_huff	1.0000	**0.9255**	0.8565	0.8078	0.8378	0.8060	0.1536
	synth_ram	1.0000	**0.8566**	0.6473	0.8078	0.9633	1.0238	0.1263
	Avg. Test	1.0000	**0.8957**	0.8523	0.8513	0.7583	1.3401	1311.0993

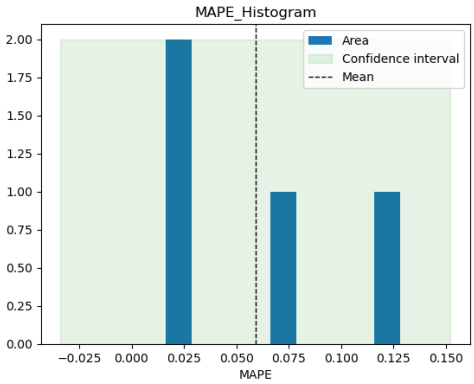
Table A.15: MAPE Comparison of GCN, GAT, EGAT, and [7] Model Performance

Graph Structure	**Model**	**Splits**	**Epochs**	**Train Samples**	**Test Samples**	**Train MAPE**	**Test MAPE**	**Train R2**	**Test R2**
Gate-level Graph	GCN	4	61.5	580	380	0.7800	0.7800	-4.9430	-2.7623
	GAT	4	58.5	580	380	0.0770	0.0770	0.9922	0.9933
Pin-level Graph	GCN	4	78.75	580	380	2.8217	3.3358	-108.6242	-105.5818
	GAT	40	65.75	580	380	0.544133	0.567914	-4.8913	-4.1758
	EGAT	8	68.25	580	380	0.060038	0.065473	0.9946	0.9944
	Guo-GNN [7]	1	26819	580	380	0.8353	0.8219	0.2427	0.2803

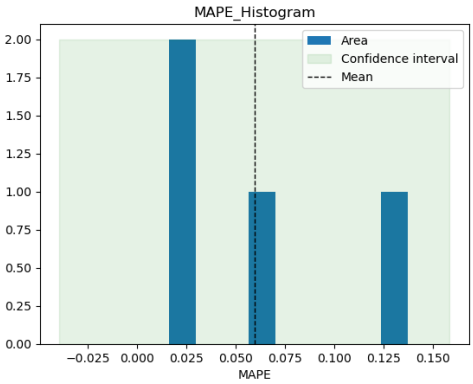
negative R^2 scores (-4.9430 training, -2.7623 testing), indicating a failure to learn meaningful signal from the data.

Figure A.16, A.17, and A.18 represents the histogram of MAPE in cross validation splits for **Pin-level Graph** GCN, GAT, and GCN models. It shows the accuracy in MAPE form for predicting total area.

Figure A.19, A.20, and A.21 represents the histogram of MAPE in cross validation splits for **Pin-level Graph** GCN, GAT, and GCN models. It shows the accuracy in

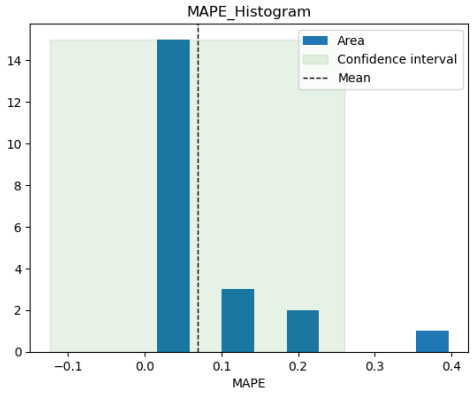


(a) Train Dataset

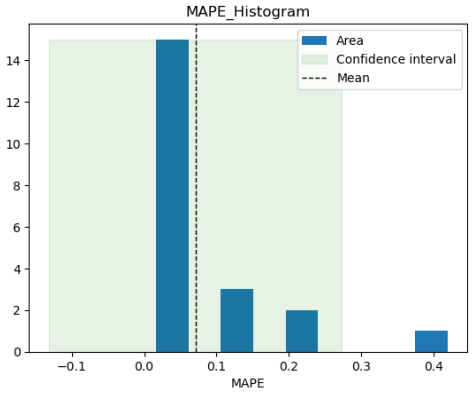


(b) Test Dataset

Figure A.16: GCN Area MAPE Cross Validation Histogram

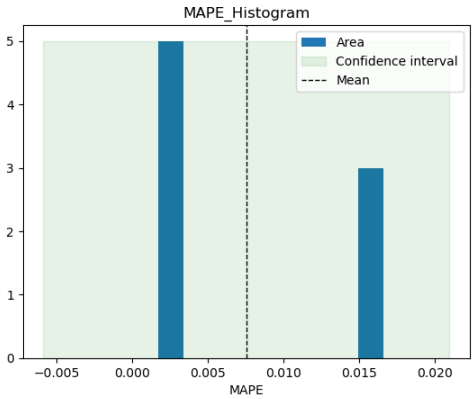


(a) Train Dataset

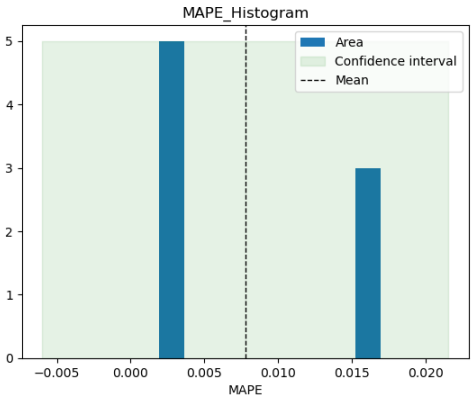


(b) Test Dataset

Figure A.17: GAT Area MAPE Cross Validation Histogram

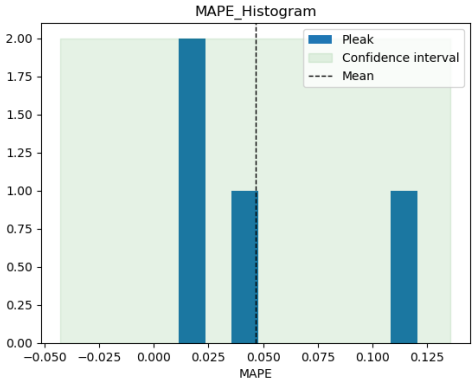


(a) Train Dataset

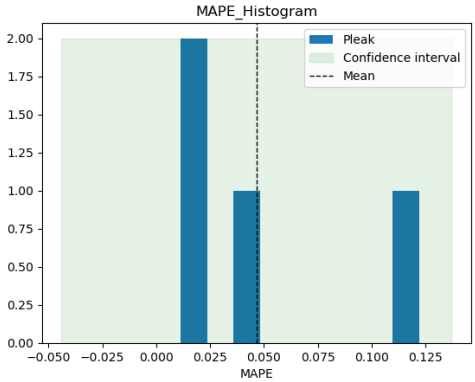


(b) Test Dataset

Figure A.18: EGAT Area MAPE Cross Validation Histogram

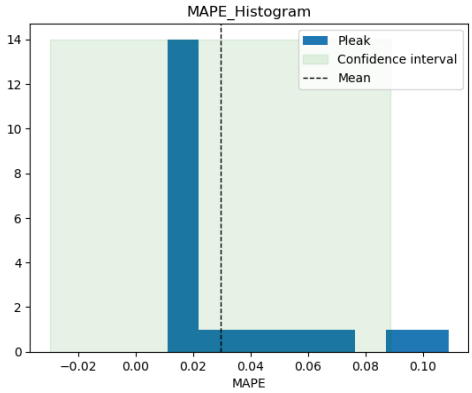


(a) Train Dataset

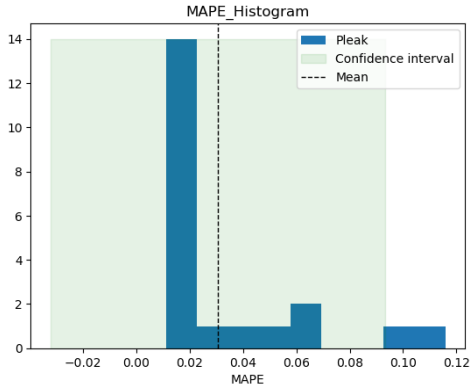


(b) Test Dataset

Figure A.19: GCN Pleak MAPE Cross Validation Histogram

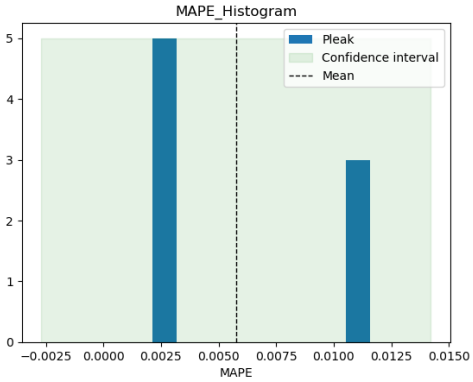


(a) Train Dataset

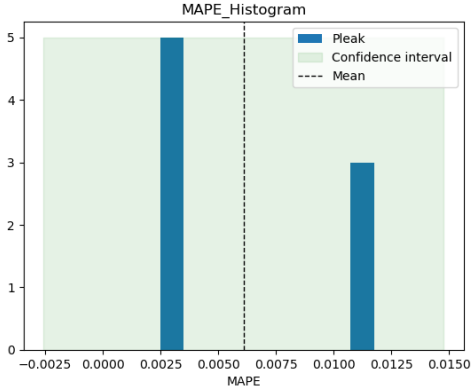


(b) Test Dataset

Figure A.20: GAT Pleak MAPE Cross Validation Histogram



(a) Train Dataset



(b) Test Dataset

Figure A.21: EGAT Pleak MAPE Cross Validation Histogram

MAPE form for predicting leakage power.

Transfer Learning

Table A.16: MAPE Comparison of GCN, GAT, and EGAT Base Models for Retraining

	Model	Epochs	Train Samples	Test Samples	Train MAPE	Test MAPE
Gate-level Graph	GCN	88	576	384	0.0387	0.1321
	GAT	5	576	384	0.0384	1.7466
Pin-level Graph	GCN	97	576	384	0.0221	0.5453
	GAT	75	576	384	0.0219	0.0448
	EGAT	76	576	384	0.0090	0.2313

Gate-level Graph

Table A.16 reports base–model performance for GCN, GAT, and EGAT under two graph granularities—gate-level and pin-level—which serve as initializations for subsequent transfer-learning experiments.

Under the **gate-level** representation, the GCN achieves strong fit and generalization (train MAPE = 0.0387, test MAPE = 0.1321). This suggests that when nodes aggregate gate semantics and connectivity, the inductive bias of convolutional neighborhood aggregation is well aligned with the task and available features. By contrast, GAT under the same setting yields a substantially higher test error (test MAPE = 1.7466), indicating that the added attention parameterization does not confer an advantage at this granularity—potentially due to over-parameterization, suboptimal attention weighting on coarse features, or optimization instability at the given data scale.

Table A.17, A.18, and A.19 illustrates the accuracy of PPA per designs for the pretrained model to be used in transfer learning. These tables illustrates the accuracy is measured in mean absolute percentage error (MAPE) format per design in the dataset.

Table A.20, A.21, and A.22 illustrates the accuracy of PPA per designs for the re-

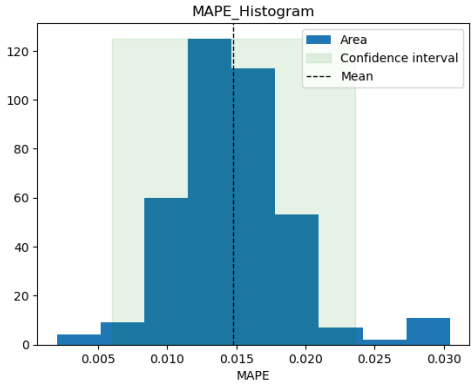
Table A.17: Area MAPE Comparison across GCN, GAT, and EGAT models

		Gate-level Graph		Pin-level Graph		
Dataset	**Design**	**GCN**	**GAT**	**GCN**	**GAT**	**EGAT**
Train	Samples	576	576	576	576	576
Test	Samples	384	384	384	384	384
Train	aes_core	0.0145	0.0145	0.0145	0.0145	0.0014
	des3_area	0.0182	0.0181	0.0182	0.0182	0.0021
	des3_perf	0.0122	0.0122	0.0126	0.0122	0.0103
	i2c	0.0162	0.0161	0.0162	0.0161	0.0013
	sasc	0.0173	0.0172	0.0173	0.0173	0.0018
	simple_spi	0.0172	0.0172	0.0172	0.0172	0.0015
	spi	0.0172	0.0172	0.0172	0.0172	0.0015
	ss_pcm	0.0186	0.0189	0.019	0.019	0.0012
	systemcaes	0.0166	0.0166	0.0166	0.0166	0.0009
	systemcdes	0.0166	0.0166	0.0167	0.0166	0.0014
	usb_funct	0.0163	0.0164	0.0163	0.0163	0.0011
	usb_phy	0.02	0.02	0.02	0.02	0.0011
	wb_dma	0.0184	0.0184	0.0184	0.0184	0.0017
Train	Avg. Train	0.0169	0.0169	0.0169	0.0169	0.0021
Test	ac97_ctrl	0.0177	0.0216	0.0575	0.0176	0.0097
	ethernet	0.0441	0.2421	0.031	0.0156	0.0223
	fpu	0.029	0.0397	0.9201	0.0146	0.0044
	mem_ctrl	0.0167	0.017	0.017	0.017	0.0032
	pci	0.0161	0.1561	0.016	0.016	0.0051
	spi	0.0191	0.0192	0.0188	0.0191	0.0028
	tv80	0.0158	0.0159	0.0159	0.0159	0.0024
	vga_lcd	0.0537	0.0846	0.019	0.0132	0.0391
Test	Avg. Test	0.0265	0.0745	0.1369	0.0161	0.0111

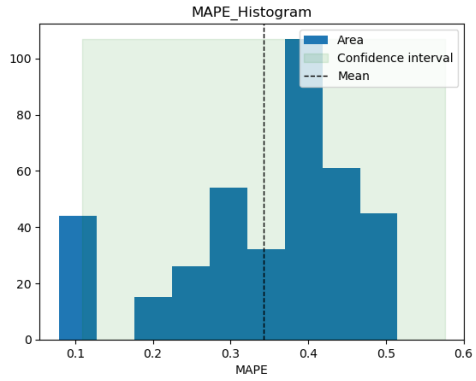
trained model to be used in transfer learning. The model is mentioned in section 6.4.2. The accuracy is measured in mean absolute percentage error (MAPE) format.

Figure A.22, A.17, and A.18 represents the histogram of MAPE in cross validation splits for **Pin-level Graph** GCN, GAT, and GCN models. It shows the accuracy in MAPE form for predicting area.

Figure A.25, A.26, and A.27 represents the histogram of MAPE in cross validation splits for **Pin-level Graph** GCN, GAT, and GCN models. It shows the accuracy in

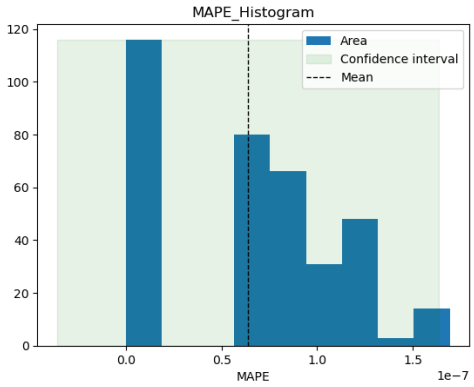


(a) Train Dataset

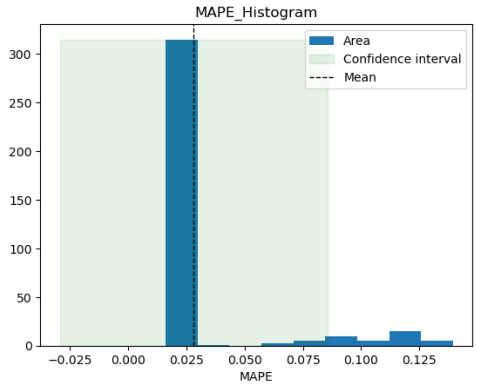


(b) Test Dataset

Figure A.22: Retrained GCN Area MAPE Cross Validation Histogram

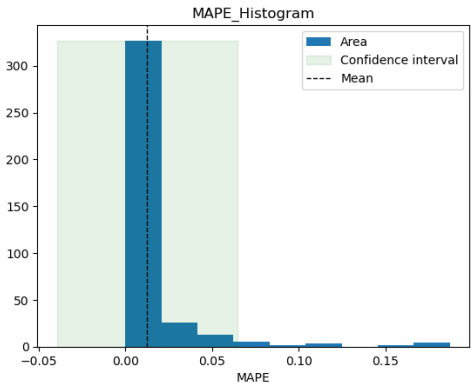


(a) Train Dataset

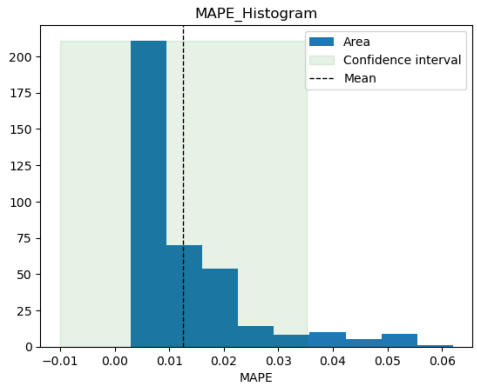


(b) Test Dataset

Figure A.23: Retrained GAT Area MAPE Cross Validation Histogram

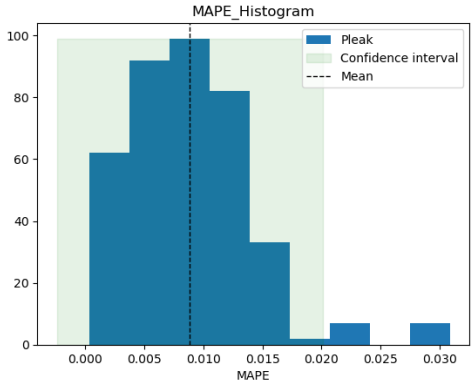


(a) Train Dataset

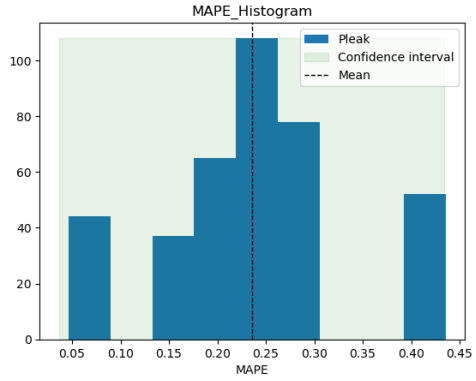


(b) Test Dataset

Figure A.24: Retrained EGAT Area MAPE Cross Validation Histogram

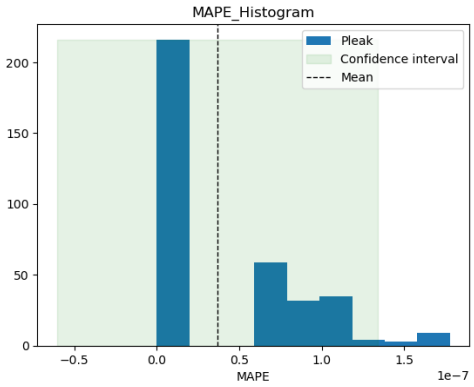


(a) Train Dataset

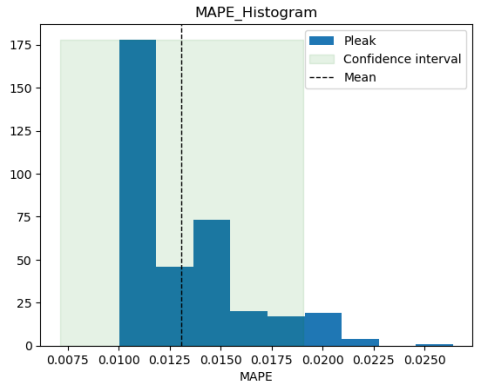


(b) Test Dataset

Figure A.25: Retrained GCN Pleak MAPE Cross Validation Histogram

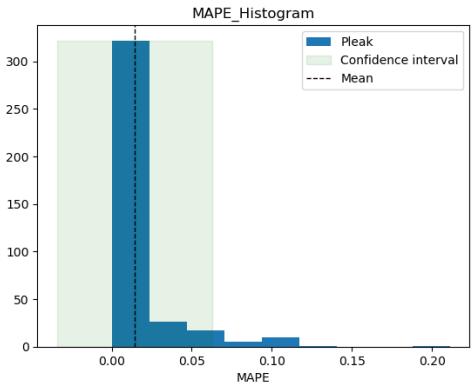


(a) Train Dataset

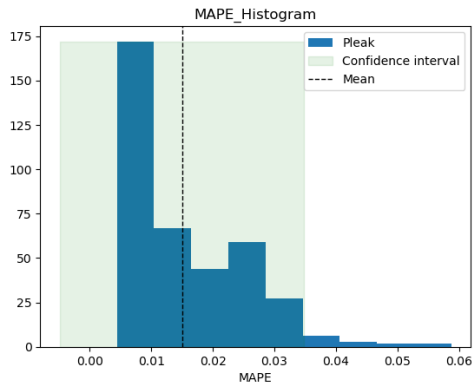


(b) Test Dataset

Figure A.26: Retrained GAT Pleak MAPE Cross Validation Histogram



(a) Train Dataset



(b) Test Dataset

Figure A.27: Retrained EGAT Pleak MAPE Cross Validation Histogram

Table A.18: Critical Path Delay MAPE Comparison across GCN, GAT, and EGAT Models

		Gate-level Graph		Pin-level Graph		
Dataset	**Design**	**GCN**	**GAT**	**GCN**	**GAT**	**EGAT**
Train	aes_core	0.0206	0.0191	0.0207	0.0206	0.018
	des3_area	0.0311	0.0291	0.0311	0.031	0.0189
	des3_perf	0.0244	0.023	0.0226	0.0244	0.0245
	i2c	0	0	0	0	0
	sasc	0.0154	0.0149	0.0154	0.0154	0.007
	simple_spi	0	0	0	0	0
	spi	0	0	0	0	0
	ss_pcm	0.0824	0.0867	0.09	0.0887	0.033
	systemcaes	0.0319	0.0306	0.032	0.0319	0.0218
	systemcdes	0.0093	0.0089	0.0097	0.0093	0.009
	usb_funct	0.1205	0.1205	0.1205	0.1205	0.0654
	usb_phy	0.0835	0.085	0.0842	0.0843	0.0265
	wb_dma	0.0453	0.0428	0.0454	0.0453	0.0271
Train	Avg. Train	0.0357	0.0354	0.0363	0.0363	0.0193
Test	ac97_ctrl	0.0376	0.1351	0.274	0.0371	0.0554
	ethernet	0.3575	9.4485	0.3677	0.3527	1.8126
	fpu	0.0044	0.0996	0.0724	0.0021	0.009
	mem_ctrl	0.0492	0.0509	0.0525	0.0492	0.0725
	pci	0.011	0.1196	0.0108	0.0108	0.0149
	spi	0.06	0.0586	0.0822	0.06	0.0959
	tv80	0.0432	0.0484	0.1979	0.0466	0.0387
	vga_lcd	0.4943	4.0122	16.7481	0.5007	3.1149
Test	Avg. Test	0.1321	1.7466	2.2257	0.1324	0.6518

MAPE form for predicting leakage power.

A.1.5 Selective Feature GNN

In Section 6.4.1, we trained and evaluated Guo-GNN on the dataset of [17]. The results indicate a feature–stage mismatch: the **Net Embedding Model** and **Delay Propagation Model** in Guo-GNN are instantiated for placement-resolved net trees and cell-library contexts, and therefore do not natively accommodate pre-floorplan inputs. To address

Table A.19: Leakage Power MAPE Comparison across GCN, GAT, and EGAT models

		Gate-level Graph		Pin-level Graph		
Dataset	**Design**	**GCN**	**GAT**	**GCN**	**GAT**	**EGAT**
Train	aes_core	0.0062	0.0062	0.0062	0.0062	0.0011
	des3_area	0.0138	0.0138	0.0138	0.0138	0.0036
	des3_perf	0.0039	0.0039	0.0046	0.0039	0.0041
	i2c	0.0163	0.0163	0.0163	0.0163	0.0013
	sasc	0.016	0.0158	0.016	0.016	0.0019
	simple_spi	0.0149	0.0149	0.0149	0.0149	0.0019
	spi	0.0149	0.0149	0.0149	0.0149	0.0019
	ss_pcm	0.0169	0.0169	0.0169	0.017	0.0009
	systemcaes	0.0123	0.0124	0.0123	0.0123	0.0018
	systemcdes	0.0111	0.0117	0.011	0.0111	0.0018
	usb_funct	0.0095	0.0101	0.0095	0.0095	0.0023
	usb_phy	0.0175	0.0179	0.0174	0.0174	0.0016
	wb_dma	0.0132	0.0133	0.0132	0.0132	0.0027
Train	Avg. Train	0.0128	0.0129	0.0129	0.0128	0.0021
Test	ac97_ctrl	0.0132	0.0415	0.0401	0.0109	0.0201
	ethernet	0.0589	0.0641	0.0224	0.008	0.0459
	fpu	0.0069	0.0616	0.152	0.0068	0.0024
	mem_ctrl	0.0107	0.0107	0.0109	0.0107	0.0048
	pci	0.0149	0.0295	0.0092	0.0092	0.0074
	spi	0.0167	0.0166	0.0165	0.0167	0.0056
	tv80	0.0102	0.0101	0.0418	0.0102	0.0022
	vga_lcd	0.0804	0.0977	0.5275	0.0075	0.0917
Test	Avg. Test	0.0265	0.0415	0.1026	0.01	0.0225

this, we adapted Guo-GNN to consume **pre-floorplan** features and re-evaluated it in a head-to-head comparison against GCN, GAT, EGAT, and the original (unmodified) Guo-GNN.

Selective Feature GNN Model Structure

The proposed Selective Feature Graph Neural Network (SF-GNN) builds upon the foundational GNN architecture described in Section 6.3, maintaining the same high-level structure depicted in Figure 6.4. However, unlike the standard message-passing layers

Table A.20: Area MAPE Comparison across GCN, GAT, and EGAT Retrained Models

		Pin-level Graph		
Dataset	**Design**	**GCN**	**GAT**	**EGAT**
Train	Samples	1	1	1
Test	Samples	383	383	383
Train	Avg. Train	0.01477	6.36E-08	0.0127
Test	ac97_ctrl	0.1263	0.0279	0.0095
	ethernet	0.2346	0.0583	0.0267
	fpu	1.2503	0.0159	0.0075
	mem_ctrl	0.071	0.0188	0.0041
	pci	0.0651	0.0248	0.0058
	spi	0.0403	0.0215	0.0031
	tv80	0.1256	0.0173	0.0026
	vga_lcd	0.8189	0.0406	0.0416
Test	Avg. Test	0.3415	0.0281	0.0126

Table A.21: Critical Path Delay MAPE Comparison across GCN, GAT, and EGAT Retrained Models

		Pin-level Graph		
Dataset	**Design**	**GCN**	**GAT**	**EGAT**
Train	Samples	1	1	1
Test	Samples	383	383	383
Train	Avg. Train	0.1060	5.45E-07	0.5282
Test	ac97_ctrl	0.6959	0.0622	0.0857
	ethernet	16.8267	0.7126	1.7703
	fpu	0.1336	0.0038	0.007
	mem_ctrl	0.3987	0.0657	0.0553
	pci	0.2342	0.0196	0.0174
	spi	0.147	0.1075	0.0786
	tv80	0.2476	0.057	0.0449
	vga_lcd	35.662	0.6845	3.8792
Test	Avg. Test	6.7932	0.2141	0.7423

(MPL) employed in models such as GCN, GAT, and EGAT, SF-GNN introduces a novel MPL block design and node-level embedding mechanism. These customizations are illustrated in Figure A.28, which presents both the SF-GNN message-passing block and the feature-aware node-wise aggregation module.

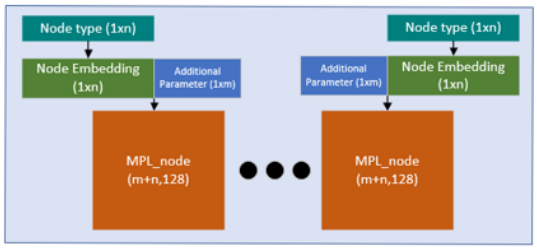
Table A.22: Leakage Power MAPE Comparison across GCN, GAT, and EGAT Retrained Models

		Pin-level Graph		
Dataset	**Design**	**GCN**	**GAT**	**EGAT**
Train	Samples	1	1	1
Test	Samples	383	383	383
Train	Avg. Train	0.0089	3.68E-08	0.0145
Test	ac97_ctrl	0.0472	0.0138	0.0118
	ethernet	0.3564	0.0117	0.0274
	fpu	0.3744	0.0085	0.0076
	mem_ctrl	0.0766	0.0129	0.004
	pci	0.0621	0.0123	0.0065
	spi	0.0575	0.0214	0.0056
	tv80	0.0722	0.0122	0.0026
	vga_lcd	0.833	0.0119	0.0543
Test	Avg. Test	0.2349	0.0131	0.015

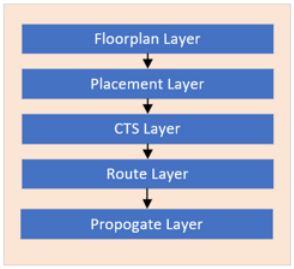
Table A.23: SF-GNN Graph Data Structure

Type	**Feature**	**Type**	**Description**
Node	Node type	float	[Fanout/Fanin, primary input/output flag, clock flag]
	Capacitance	float	[rise capacitance, fall capacitance] from cell library
	Cell	float	[Cell index, Cell area, Cell width, Cell height, Cell leakage power] from cell library
	Pin Location	float	[Pin Width, Pin Height] from cell library
Edge	ef	float	Cell Delay measure in pre-floorplan stage [rise delay, fall delay]
	rise_index	float	Rise delay index from library file
	fall_index	float	Fall delay index from library file
	rise_matrix	float	Rise delay matrix from library file
	fall_matrix	float	Fall delay matrix from library file
Graph	Longest Path	int	Max DAG path length
	Clock period	float	Timing constraint
	Aspect ratio	float	Placement constraint
	Max utilization	float	Routing constraint
	Max skew	float	Max allowed skew
	Max fanout	float	Fanout constraint
	Max clock cap	float	Max clock capacitance
Type	**Label**	**Type**	**Description**
Graph	Arrival time	float	Post-route Stage Critical path delay

SF-GNN is conceptually inspired by the timing-engine–aware Guo-GNN model proposed in [7], but differs in several aspects. While Guo-GNN executes a fixed two-stage net embedding followed by a single cell propagation phase, SF-GNN incorporates both



(a) SF-GNN MPL Block



(b) SF-GNN Node-wise Feature Embedding

Figure A.28: SF-GNN Model Architecture Details

Net Embedding and **Cell Propagation** in every message-passing layer. This iterative formulation allows the model to progressively refine local timing information across the topology while capturing higher-order dependencies between cell interactions.

In addition, SF-GNN includes enhanced graph-level features specifically engineered to support wire capacitance estimation. Rather than relying solely on raw node types and adjacency, the model selectively incorporates engineered attributes, which mimic physical design flow, design into the embedding and message propagation process. This feature-aware design enables SF-GNN to specialize in capturing both logical and physical phenomena underlying critical path delay.

SF-GNN Model Evaluation

We evaluate the proposed Selective-Feature GNN (SF-GNN) on the same benchmark suite described in Section 6.2, following the Pin-level graph representation and the

critical-path delay prediction task from [17]. Owing to intermittent backpropagation errors encountered in the Deep Graph Library (DGL) during preliminary experiments, we restricted the study to a stable subset comprising 12 designs, each with 48 samples. While this constraint reduces statistical coverage relative to our full corpus, it nevertheless provides a controlled setting for head-to-head comparisons among GNN architectures.

Table A.24 reports baseline performance for four established models—GCN, GAT, EGAT, and Guo-GNN—trained and evaluated on the full dataset. These results serve as a reference point for contextualizing SF-GNN on the reduced subset. As expected, attention-based models dominate: EGAT achieves the strongest average performance (test MAPE = 0.0655, test R^2 = 0.9944) followed by GAT.In contrast, vanilla GCN struggles on this task (test MAPE = 3.3358, large negative R^2), and Guo-GNN provides moderate gains but does not surpass attention-based approaches.

Table A.24: Baseline Model Performance Comparison on Full Dataset

Model	Splits	Epochs	Train Samples	Test Samples	Train MAPE	Test MAPE	Train R2	Test R2
GCN	4	78.75	580	380	2.8217	3.3358	-108.6242	-105.5818
GAT	40	65.75	580	380	0.5441	0.5679	-4.8913	-4.1758
EGAT	8	68.25	580	380	0.0600	0.0655	0.9946	0.9944
Guo-GNN [7]	1	26819	580	380	0.8353	0.8219	0.2427	0.2803

To isolate the effect of feature selection and stage-wise timing propagation in SF-GNN, we then retrained all models on the 12-design subset (432 train 144 test) under identical data splits and hyperparameters (Table A.25 and Table A.26). On the training split, SF-GNN attains the lowest MAPE across models (0.0389 vs. EGAT's 0.0693), a relative reduction of $\approx 3.04\%$ compared to EGAT. Per-design results further show that SF-GNN is competitive or superior on several small and medium designs (e.g., `simple_spi`, `tv80`) where the gating mechanism can emphasize predictive node/edge attributes.

On the held-out test split shown on Table A.26, results are mixed. EGAT remains

Table A.25: Critical Path Delay MAPE Comparison across GCN, GAT, EGAT, Guo-GNN [7], and SF-GNN for Train Dataset

Design	**GCN**	**GAT**	**EGAT**	**Guo-GNN [7]**	**SF-GNN**
Splits	40	40	40	1	1
Train	432	432	432	432	432
Test	144	144	144	144	144
ac97_ctrl	5.6943	6.0763	0.0346	0.9022	0.0335
aes_core	6.621	4.9667	0.0204	0.0077	0.0194
des3_area	1.7925	1.2579	0.0266	0.0083	0.0238
des3_perf	27.2481	21.2436	0.02	-	-
fpu	1.2438	0.619	0.0021	-	-
sasc	0.1624	0.1466	0.0123	0.7492	0.0151
simple_spi	0	0	0	0.8602	0.001
ss_pcm	0.42	0.5401	0.0719	0.0732	0.0599
tv80	2.4797	2.8232	0.0385	0.125	0.0275
usb_funct	5.8656	15.3637	0.1017	0	0.0583
usb_phy	0.3824	0.5084	0.0675	0.1176	0.1115
vga_lcd	555.8543	408.966	0.4359	-	-
Avg. Train	50.647	38.5426	0.0693	0.3159	0.0389

the strongest overall (average test MAPE = 0.0721), while SF-GNN achieves competitive accuracy on a few instances but underperforms on others (average test MAPE = 0.6449). The discrepancy between SF-GNN's strong training accuracy and its test-set behavior suggests overfitting under the current data budget, especially for designs with distributional characteristics that deviate substantially from the training subset. For example, `vga_lcd` exhibits substantially higher error for all models; SF-GNN's test MAPE (0.8068) is higher than EGAT (0.4444) and accompanies a negative R^2, indicating poor generalization. These outcomes are consistent with the broader observation in Section 6.5 that designs with large statistical distance from the training distribution (as measured by the Wasserstein metric) are more difficult to predict reliably after transfer.

Table A.26: Critical Path Delay Comparison across GCN, GAT, EGAT, Guo-GNN [7], and SF-GNN for Test Dataset

Design	**GCN**		**GAT**		**EGAT**		**Guo-GNN [7]**	**SF-GNN**	
Splits	40		40		40		1	1	
Train	432		432		432		432	432	
Test	144		144		144		144	144	
	MAPE	R2	MAPE	R2	MAPE	R2	MAPE	MAPE	R2
ac97_ctrl	5.6926	-30901	6.067	-23374	0.0374	-0.1569	-	-	-
aes_core	6.612	-291344	4.9724	-87994	0.0225	-0.8568	-	-	-
des3_area	1.7994	-4049	1.2575	-2240	0.0284	0.0293	-	-	-
des3_perf	27.3336	-1855701	21.2126	-1176286	0.0204	0.1275	0.0043	0.1512	-32.1664
fpu	1.2443	-252161	0.6189	-205974	0.0021	-0.3375	0	0.9767	-74417.1406
sasc	0.1643	-105	0.1448	-75	0.013	0.1309	-	-	-
simple_spi	0	1	0	1	0	1	-	-	-
ss_pcm	0.4181	-35	0.56	-45	0.079	0.1661	-	-	-
tv80	2.4706	-6149	2.8282	-5523	0.0405	0.1739	-	-	-
usb_funct	5.8193	-4236	15.4153	-23561	0.1059	0.1291	-	-	-
usb_phy	0.3798	-31	0.5061	-47	0.0721	0.1243	-	-	-
vga_lcd	521.9693	-3401646	414.8952	-1218126	0.4444	0.0401	0.9316	0.8068	-5.5455
Avg	47.8253	-487196	39.0399	-228604	0.0721	0.0475	0.3119	0.6449	-24818.2841

Discussion. Two patterns emerge. First, SF-GNN's training behavior is strong: selective gating helps the model fit relevant structural/physical features and reduces training error substantially relative to EGAT. Second, SF-GNN's generalization is less consistent under the present data budget, as evidenced by (i) worse average test MAPE than EGAT and (ii) strongly negative R^2 on outlier designs such as `vga_lcd`. This suggests that the current gating configuration may over-specialize to the limited subset; combined with the reduced sample size (12 designs $\times$ 48 samples), this increases the risk of overfitting.

A plausible explanation is distributional mismatch. As shown in Section 6.5, designs with large Wasserstein distance from the training set tend to be harder to transfer to. SF-GNN's errors concentrate on those statistically distant designs, whereas its accuracy is competitive on designs closer to the training distribution. This observation motivates two practical remedies: (i) Wasserstein-guided sampling (prioritizing or weighting training instances to reduce distributional divergence to target designs) and (ii) regularization of the gating module (e.g., dropout on gated features, mixup across designs) to reduce

over-specialization.

Summary and Limitations

In summary, SF-GNN demonstrates promising in-sample accuracy—improving average training MAPE by $\approx 43.7\%$ relative to EGAT on the 12-design subset—while exhibiting mixed out-of-sample performance under a constrained data regime. The primary limitations are (i) restricted statistical coverage due to DGL backpropagation issues, which prevented large-scale cross-validation, and (ii) a lack of systematic tuning of the gating regularization. Future work will address these issues by (1) scaling up cross-validation over more designs and seeds, (2) incorporating Wasserstein-guided data selection and loss weighting, and (3) ablations on the gating depth, sparsity regularization, and stage-wise timing features to improve robustness on outlier designs (e.g., `vga_lcd`).

Appendix B

IEEE Conference Paper Submission

An Efficient PPA Approximation Modeling using Meta-Transfer Learning

Donghyeon Koh, *PhD Student, NC State University*

***Abstract*—Machine learning (ML) has been widely used in predicting the power, performance, and area (PPA) of the System-on-Chip (SoC). ML model can predict the PPA of a chip which allows designers to predict the PPA of the SoC in the early stage of design and prevent mistakes that might be caused in the process. A variety of methods has been developed using ML to predict PPA, such as convolutional neural networks, transfer learning, and meta-learning. However, the ML model is still limited to predicting the PPA of a particular design and requires a large amount of Quality of Results (QoR) to train the model.**

In this paper, we propose a machine learning model using meta-transfer learning with few-shot learning to predict PPA. This method uses the machine learning (ML) method to train the initial model to predict the PPA of a design with high accuracy. After training the initial model, we retrain the model with a different design with as few QoR as possible.

***Index Terms*—PPA, Machine Learning, Power, Performance, Area, Few-shot learning, Transfer learning, Electronic design automation, System on Chip**

I. Introduction

The machine learning mechanism for predicting PPA in the early stage of designing a System-on-Chip (SoC) is critical considering the time it takes to process a chip. This could also prevent any possible flaw that will cause one to miss any of the PPA targets. Thus, we propose machine learning to initially generate and train the model and retrain the model using transfer learning to optimize the model to a specific design.

Machine learning with transfer learning is one of the methods for predicting the PPA. This method has been found useful in a number of researches [1]–[3]. The transfer learning method optimizes the trained model for a similar task. For the machine learning method, this process involves retraining specific layers or neurons with a similar design. This method requires fewer QoR, or samples, for retraining the model. Since this method is performed in different fields, the performance is varied depending on which field the model is being executed to predict the PPA.

However, the PPA model retrained with the transfer learning method could only predict the PPA of the design that is related to the design that is used when initially training the PPA model. Initial training of the PPA model consumes a large amount of time and the limited spectrum of a model to certain designs suggests a new model for the PPA prediction. Thus, in this paper, we discuss the meta-transfer learning model as a new model that could predict a design in which statistical distance is comparable far from the design used to initially train the model, using a minimum number of QoR as possible.

In the rest of the paper, we organize as follows: we describe different machine learning types, such as transfer learning, meta-learning, and meta-transfer learning in section II and present the model structure in section IV. In section III, we introduce the datasets which we will train the model with. Section V presents the result of the evaluated model and provides a conclusion in section VI.

II. Related Work

A. Transfer Learning

Machine learning with transfer learning has been widely used for PPA approximation. The purpose of transfer learning is to learn and optimize the model for a similar task. The basic theory of transfer learning is to initially train the model with a large QoR. Then, we retrain the part of the model with a new QoR that is similar to the initial training datasets. This process will reduce the time the model requires to predict the PPA of a similar design. The research done in [1] illustrated designing a neural network (NN) model with transfer learning to predict PPA by retraining with as few QoR as possible. The work in [2] used a transfer learning approach to reduce the amount of target domain data required by PPA models of memory compilers. In work [3], the design space exploration approach for a convolutional neural network is used to optimize the physical synthesis of hardware accelerators.

B. Meta Learning

The meta-learning is one of the machine learning methods that learn from other learning algorithms. There are a variety of mechanisms for meta-learning and we decided to focus on few-shot learning which is to learn from the few samples as possible. The MetaNet introduced in [4] learns in a meta space and shifts the parameters and bias of the neural network with few-shot learning. Another work in [5], proposes Meta-SGD which could adapt to a new task in just one step adaptation. These papers mention that Meta Learning could adapt to a new task with few-shot learning.

C. Meta-transfer Learning

Meta-transfer learning (MTL) uses the advantage of both the transfer learning method and the meta-learning method. This method speeds up the process it takes for the neural network to converge and prevent overfitting [6]. The scale and shifting are used to fine-tune the model. In [6], the hard task (HT) meta-batch method re-samples the hard samples depending on the past failed tasks with low validation accuracy. This is derived

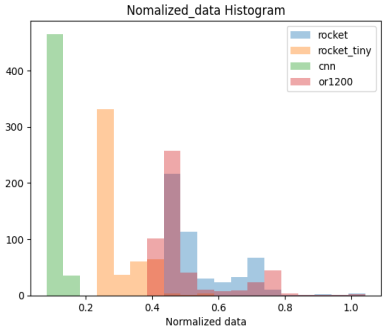


Fig. 1. Histogram on target Area

from the conventional meta-batch which re-samples randomly which is comparably inefficient. The HT meta-batch and fine-tuning are the two main methods to meta-transfer learning.

III. Datasets

The datasets that the model will be trained on contain Rocket Core Configuration (rocket), Rocket Core Configuration with tiny configuration (rocket tiny), convolutional neural network (CNN) accelerator IP blocks, and OpenRisc 1200 (or1200). The designs contain RTL codes and Synthesis results that are driven by GlobalFoundaries 22FDx/22FDSOI INVECAS-based Synopsys ICC2 Implementation Flow.

In the aspect of statistical distance between the designs, Rocket Core Configuration and OpenRisc 1200 are closer compared to CNN accelerator IP blocks. Figure 1 illustrates the statistics of the designs on the target area. This difference in statistical distance between the designs is critical in model optimization which will cause high denormalized MAPE. Thus, we need to analyze the statistical distances between the designs.

A. Relative Entropy

The relative entropy, known as Kullback-Leibler divergence (K-L divergence), is a statistical distance from the model to the target [7]. The K-L divergence measures how far the target is from the model, in units of the natural unit of information (nats). However, this approach is inappropriate for the normalization that we used in the research.

K-L Divergence is defined for two random variables when their probability distributions are absolutely continuous. Probability distributions for datasets are not absolutely continuous in general. Thus, we need another method to measure the statistical distance between the designs.

B. Wasserstein Distance

Wasserstein distance or earth mover's distance is the distance function to estimate the optimal transport cost between two measures. The optimal transport cost is the cost of transporting one unit of mass to the other. If the cost is defined in terms of distance, the Wasserstein distance could be used to measure the statistical distance between two measures [8]. Thus, we apply the Wasserstein distance between the designs to obtain the cost of an optimal transport plan to convert one distribution to the other.

$$W_1(u,v) = \int_{-\infty}^{\infty} |F_u(x) - F_v(x)|dx \tag{1}$$

The Equation 1 presents the Wasserstein distance equation for two probability mass functions. F_u, F_v represents the cumulative distribution function of the probability mass function u, v. The Equation 1 is valid Wasserstein distance only if it is performed in one dimension [10].

Table I shows the statistical distance between the designs. The results indicate for all four designs, CNN design has the most distance from the rocket default configuration design followed by or1200 and rocket tiny designs.

TABLE I
Wasserstein Distance between rocket design

Model	Area	Critical Path	Switched Capacitance	Leakage Power
rocket tiny	0.253	0.0074	0.1694	0.1536
cnn	0.4542	0.0816	0.3402	0.3092
or1200	0.0452	0.1055	0.0248	0.0431

C. Model Parameters

The four synthesis parameters are used to affect the gate-level netlist. Clock period (Tclk), maximum fan-out at logic node (Fanout), maximum transition time at a logic node (MaxTran), and clock uncertainty. We created 500 samples by varying the four synthesis parameters using Latin hypercube (LHC) and performed the synthesis. From the result of the synthesis, we used four variables as our target vectors: Leakage power (Pleak), critical path delay (Cpath), area, and switched capacitance (Csw).

IV. Model Structure

In this section, we introduce the base model structure generated with PyTorch. The base model is trained and tested using two main algorithms: Outer Optimization and Inner Optimization.

A. Mean Absolute Percentage Error

The accuracy of a model is measured by the mean of denormalized mean absolute percentage error (MAPE):

$$MAPE = \frac{100\%}{N}\sum_{i=1}^{N}\left|\frac{A_n - P_n}{P_n}\right| \tag{2}$$

where A_n is the actual value and P_n is the predicted value.

To investigate the behavior of the model, the 50 shuffle split cross-validation is executed. Table II,III,IV,V shows the mean of denormalized MAPE for four target vectors. For comparison between the model, the total epochs executed is set to 5000.

For regularization, the AdamW algorithm from the PyTorch is used, implemented from [9]

B. Base Model

The base model is a neural network model structure with four linear layers the size of [128,256,512,256,64] with an input dimension of 6 and an output dimension of 1. The base model is trained which sets different weight paths for multiple targets. The activation function, also known as the transfer function, is used to determine the output of the neural network. The base model uses the rectifier linear unit (ReLU) function as an activation function for the entire layers.

In the pre-training phase, the base model is trained with 300 samples from the rocket design and tested with 200 samples from the rocket design. The 1 will be used to train and test the model. This pre-trained model is saved to be used in the transfer learning and meta-transfer learning phase. Table II illustrates the results of the pre-training phase. In comparison with the result from [1] where the same pre-training has been executed with 400 training samples, it illustrates similar for every target MAPE.

C. Outer Optimizations

The outer optimization is used in pre-training the base model and retraining the base model in transfer learning. Transfer learning is an optimization method for a problem similar to the base model. The weight and bias are stored in the base model which implies that for PPA approximation, certain numbers of layers could be shared to approximate the PPA of a different design. Considering the work in [1], the last three layers of an NN will be retrained and evaluated.

Algorithm 1 shows the five-step process of the outer optimization: Data load, Data normalization, Model load, Model train, and model test. The data is loaded by using the PyTorch method and normalized by the SCALEFACTOR that we set for each target variable. If the model is in the transfer learning or meta-transfer learning stage, the pre-trained model is loaded and the designated layers are frozen. The model starts to forward propagate and depending on whether the model uses the meta-transfer learning or not, goes through the training phase to get the prediction to the input. Then, it compares the prediction with the actual result to get the loss function. The loss function is back propagated for optimization. This process continues until the validation loss reaches its lowest point.

D. Inner Optimization

The inner optimization is used in the process of meta-transfer training of the model. The meta-transfer learning model uses both the transfer learning method and meta-batch method to optimize the model [6]. The model process is divided into two parts: Inner Optimization and Outer Optimization mentioned in Algorithm 1.

In Algorithm 2, the single data from the datasets are used multiple times using the feedback loop. After loading the data, it is used to forward propagate to the base model. With the prediction and the actual value, the loss function is computed. Then, using the gradient of the transfer learning layer parameters and the loss function, we generate the weights and biases for the transfer learning layers. These weights and biases are feedback to the base model.

Algorithm 1 Outer Optimization

function OUT-OPT($X, Y,$) ▷ X: knobs, Y: PPA
  $n_X \leftarrow$ numFeatures(X) ▷ Count number of features
  $i \leftarrow 0$
  while $i \leq n_X$ **do** ▷ Normalize X
    $X \leftarrow X$/SCALEFACTOR(X)
    $i \leftarrow i + 1$
  end while
  $n_Y \leftarrow$ numFeatures(Y)
  $i \leftarrow 0$
  while $i \leq n_Y$ **do** ▷ Normalize Y
    $Y \leftarrow X$/SCALEFACTOR(Y)
    $i \leftarrow i + 1$
  end while
  $m \leftarrow$ initializeModel()
  if $Transfer$ or $MetaTransfer$ **then**
    $m \leftarrow$ loadModel($m_{pre-trained}$) ▷ Load pre-trained model
  end if
  $X_{train}, X_{test}, Y_{train}, Y_{test} \leftarrow$ dataSplit(X,Y)
  $n_{train} \leftarrow$ countData(X_{train})
  $n_{test} \leftarrow$ countData(X_{test})
  while min(validlossList) or $length$(validlossList)=0 **do**
    if $MetaTransfer$ **then**
      $e_{train} \leftarrow$ IN-OPT(m, X_{train}, Y_{train})
    else
      $e_{train} \leftarrow \sum_{j \in 0,...,n_{train}}$loss($m(X_{train,j}, Y_{train,j})$
    end if
    $m \leftarrow$ backpropagate(∇e_{train},m)
    if $MetaTransfer$ **then**
      $e_{valid} \leftarrow$ IN-OPT(m, X_{train}, Y_{train})
    else
      $e_{valid} \leftarrow \sum_{j \in 0,...,n_{train}}$loss($m(X_{train,j}, Y_{train,j})$
    end if
    validlossList.insert(e_{valid})
  end while
  $e_{test} \leftarrow \sum_{j \in 0,...,n_{test}}$loss($m(X_{test,j}, Y_{test,j})$
end function

Algorithm 2 Inner Optimization

function IN-OPT(m, X, Y) ▷ m: model, X: knobs, Y: PPA
  $i \leftarrow 0$
  $n \leftarrow$ countData(X)
  for $j = 0, 1, ..., n$ **do**
    while num_{steps} **do**
      $e_{in} \leftarrow$loss($m(X_j, Y_j)$
      grad $\leftarrow$ sumGrad(e_{in},m) ▷ Computes and returns the sum of gradients of loss with respect to model m gradient enabled layers
      weight $\leftarrow$ ▷
      $m.weight(grad = true) \leftarrow$ weight ▷ Replace model's gradient true layer with weight
    end while
    $e \leftarrow e+$ loss($m(X_{train,j}, Y_{train,j})$
  end for
  return e
  Use Gradient descent to compute weights and bias for transfer layers
  Feedback transfer layers' weights and bias to base model
end function

TABLE II
PRE-TRAINED BASE MODEL EVALUATION USING 300 TRAINING SAMPLES

Design	Training samples	Test samples	Outer Opt	MAPE	area MAPE	cpath MAPE	Csw MAPE	Pleak MAPE
rocket	300	200	1500	0.022	0.0182	0.0105	0.0272	0.0321
rocket [1]	400	100	-	0.026	0.029	0.0089	-	0.0345

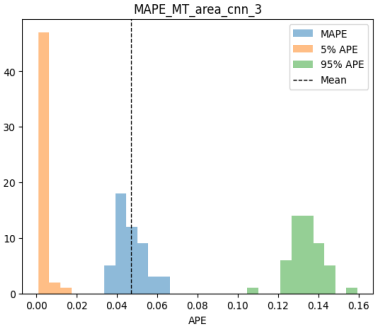


Fig. 2. MAPE of 50 cross-validations on the area of CNN design

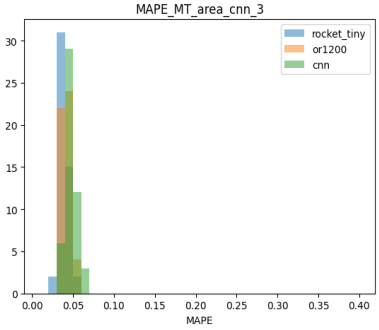


Fig. 3. MAPE of 50 cross-validations on the area of three designs

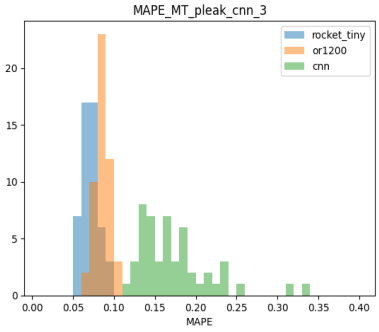


Fig. 4. MAPE of 50 cross-validations on leak power of three designs

V. EXPERIMENTS

In this section, we evaluate the model from Section IV and III.

A. Base Model

The base model is trained with Rocket Core Configuration design. From the total of 500 samples, 300 samples are used to train the model and 200 samples are used to test the model. Table II illustrates the results of 1500 outer optimization. The mean of denormalized MAPE on four target vector illustrate less than 5% error rate. This is similar to the base model result that [1] has when it has been trained with 400 samples.

B. Meta-transfer learning

The 50 cross-validations using shuffle split are evaluated for meta-transfer learning. Figure 2 illustrates the histogram of meta-transfer learning for CNN design with their target as an area with their confidence interval and the mean of the MAPE. We could see in the figure that the results are symmetrical which implies the 50 cross-validations are performed as planned. In Figure 3, we compare the area histogram of three designs which is symmetrical and overlaps for having low MAPE. However, for Figure 4, the histograms of the three designs do not overlap which implies CNN design has higher leak power MAPE.

Table III illustrates the meta-transfer learning model evaluation results using 5 samples and 495 test samples. The total iteration was 25000 which costs time to simulate and may have resulted in overfitting. However, the denormalized MAPE for all four vectors became much lower. Especially when retraining the base model with CNN design for it shows almost no similarity compared with rocket core default configuration, the error rate decreased drastically, especially on the power leakage vector. The result from [1] shows 4.52% overall accuracy when retrained with rocket tiny design. In his paper, he states that the leakage power component shows the least accuracy but does not give an exact error rate. Thus, the ablation study is required to analyze the effect of the meta-transfer learning model which differentiates from the original transfer learning model.

C. Ablation Study

The ablation study has been processed to investigate the positive effect the meta-batch method had on the model. In Table IV and V, the total iterations have been set to 5000 with the only difference being the meta-transfer learning method. Comparing the transfer learning model with the meta-transfer learning model, the meta-transfer learning model showed lower denormalized MAPE for rocket tiny and or1200 designs.

TABLE III
META-TRANSFER LEARNING MODEL EVALUATION USING 5 TRAINING SAMPLES AND 495 TEST SAMPLES BY RESETTING LAYERS

Design	Inner Opt	Outer Opt	MAPE	area MAPE	cpath MAPE	Csw MAPE	Pleak MAPE
rocket tiny			0.0522	0.0382	0.0522	0.0472	0.0710
cnn	50	500	0.0846	0.04702	0.0509	0.0665	0.1741
or1200			0.0635	0.0416	0.0909	0.0357	0.0859
rocket tiny [1]	0	-	0.0452	0.0364	0.0305	-	0.0754
rocket tiny$^{\pm}$ [1]			0.0439	0.0344	0.0404		0.0643

$^{\pm}$ Retrained with 15 samples

TABLE IV
MODEL EVALUATION FOR ABLATION STUDY BY NOT RESETTING THE LAYERS

Model	Design	Inner Opt	Outer Opt	MAPE	area MAPE	cpath MAPE	Csw MAPE	Pleak MAPE
Transfer Learning	rocket tiny	0	5000	0.0485	0.0379	0.0422	0.04626	0.0678
	cnn			0.2054	0.0606	0.0806	0.1701	0.5103
	or1200			0.0772	0.0611	0.0733	0.0686	0.1059
Meta-transfer Learning	rocket tiny	10	500	0.0296	0.021	0.01751	0.0296	0.0505
	cnn			0.2348	0.0418	0.0520	0.0908	0.7548
	or1200			0.0482	0.0442	0.0225	0.036	0.0903

TABLE V
MODEL EVALUATION FOR ABLATION STUDY BY RESETTING THE LAYERS

Model	Design	Inner Opt	Outer Opt	MAPE	area MAPE	cpath MAPE	Csw MAPE	Pleak MAPE
Transfer Learning	rocket tiny	0	5000	0.0606	0.045	0.0599	0.0544	0.0833
	cnn			0.1746	0.0729	0.1028	0.1241	0.3987
	or1200			0.0895	0.0592	0.1155	0.0525	0.1307
Meta-transfer Learning	rocket tiny	10	500	0.0814	0.0632	0.0848	0.0626	0.1152
	cnn			0.1308	0.0695	0.0862	0.0815	0.2858
	or1200			0.1010	0.0789	0.1115	0.0604	0.1532

However, for CNN design, the denormalized MAPE increased when compared with transfer learning. This is caused by the statistical distance between the designs that we work to withstand for optimizing the neural network for a wide range of designs.

To explore the statistical distance between the designs, we have compared the results between the model retraining from the trained weight and the model retraining after resetting the transfer learning layers' weight. For designs with close statistical distance to the rocket design which has been used to train the base model, such as rocket tiny and or1200, both the transfer learning model and meta-transfer learning model showed low denormalized MAPE when the weights have not been reset as shown in Table V. However, when the CNN accelerator IP block design is used to retrain the model, both models showed low denormalized MAPE when the weights have been reset as shown in Table IV.

VI. CONCLUSION AND FUTURE WORK

In this paper, we modeled and compared meta-transfer learning and the transfer learning methods for the PPA modeling which showed the possibility of PPA approximation on a wide range of designs with minimum QoR. The decrease in MAPE implies, that under a specific condition, the model could be improved to predict the PPA of a design with various statistical distances from the rocket design. Thus, revealing the potential for modeling the PPA with few QoRs.

Even though we have shown the possibility of a meta-transfer learning method for a wide range of designs, the mean of denormalized MAPE is not within the 5% range requirement. We work to find appropriate total iterations and inner optimization iterations. Also, the designs that were experimented on contain different corners. We were only able to experiment on one of them due to a time shortage. Therefore, we will research what condition is needed to generate the appropriate model depending on the statistical distance between the initial design and the retraining design.

REFERENCES

[1] L. Francisco, P. Franzon and W. R. Davis, "Fast and Accurate PPA Modeling with Transfer Learning," 2021 ACM/IEEE 3rd Workshop on Machine Learning for CAD (MLCAD), 2021, pp. 1-6, doi: 10.1109/MLCAD52597.2021.9531109.

[2] F. Last and U. Schlichtmann, "Feeding Hungry Models Less: Deep Transfer Learning for Embedded Memory PPA Models : Special Session," 2021 ACM/IEEE 3rd Workshop on Machine Learning for CAD (MLCAD), 2021, pp. 1-6, doi: 10.1109/MLCAD52597.2021.9531299.

[3] L. R. Juracy, A. d. M. Amory and F. G. Moraes, "A Fast, Accurate, and Comprehensive PPA Estimation of Convolutional Hardware Accelerators," in IEEE Transactions on Circuits and Systems I: Regular Papers, 2022, doi: 10.1109/TCSI.2022.3204932.

[4] Munkhdalai, Tsendsuren, and Hong Yu. "Meta networks." International Conference on Machine Learning. PMLR, 2017.

[5] Li, Zhenguo, et al. "Meta-sgd: Learning to learn quickly for few-shot learning." arXiv preprint arXiv:1707.09835 (2017).

[6] Sun, Qianru, et al. "Meta-transfer learning for few-shot learning." Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition. 2019.

[7] McElreath, Richard. Statistical Rethinking: A Bayesian Course with Examples in R and Stan. , 2016. Internet resource.

[8] Villani, Cédric. Optimal transport: old and new. Vol. 338. Berlin: springer, 2009.

[9] Loshchilov, Ilya, and Frank Hutter. "Decoupled weight decay regularization." arXiv preprint arXiv:1711.05101 (2017).

[10] Ramdas, Aaditya, Nicolás García Trillos, and Marco Cuturi. "On Wasserstein two-sample testing and related families of nonparametric tests." Entropy 19.2 (2017): 47.