POLITECNICO DI TORINO

Master Degree
in Mechatronic Engineering

Master Thesis

A Novel GNN-based Framework with Differentiable Pooling for High-Level Synthesis QoR Prediction

Politecnico di Torino

Supervisors:

Prof. Luciano Lavagno
Prof. Mihai Teodor Lazarescu
Dr. M. Usman Jamal

Candidate:

Ruize Chen

Academic Year 2024-2025

Abstract

High-Level Synthesis (HLS) enables rapid prototyping of application-specific hardware, allowing designers to develop hardware using C/C++ instead of hardware description languages (HDL). HLS directives (pragmas) provide optimization mechanisms for balancing performance and resource utilization. However, as the number of applied optimizations increases, the number of possible design configurations grows exponentially. Evaluating each design with HLS tools incurs significant computational and time costs, leading to slow and inefficient design space exploration. To accelerate this process, machine learning models, particularly graph neural networks (GNN), have been used to predict the quality of results (QoR) from pre-synthesis representations. However, existing methods still require improvements due to issues such as significant prediction errors caused by information loss during graph convolution and pooling.

To address these challenges, we propose a novel GNN-based framework incorporating differentiable pooling (DiffPool) to learn hierarchical representations of HLS designs. By capturing multi-level structural information, our method effectively reduces information loss while improving the accuracy of QoR predictions. Experimental results demonstrate that our model significantly reduces prediction errors in FPGA resource utilization compared to both conventional HLS estimation methods and existing learning-based models. Additionally, we conducted ablation studies to assess the impact of different components on model performance. The results indicate that, in addition to DiffPool, GATv2 and Global Attention further enhance the model's performance.

Acknowledgements

March is the rainy season in Torino. The raindrops dissolve into a gentle mist, softly outlining my memories and dreams from the two and a half years spent at Politecnico di Torino.

I would like to sincerely thank **Prof. Luciano Lavagno** and **Prof. Mihai Teodor Lazarescu**, who opened the door to academia for me and created wonderful research conditions along the journey.

My deepest gratitude goes to **M. Usman Jamal**, whose meticulous guidance and sincere concern deeply touched me. He is not only my teacher but will always remain my friend.

Thank you to my dear friends both in Europe and in China, you warmth and encouragement have given me endless joy and confidence.

Thank you Italy, you are the place where I really grew up and matured.

Finally, my heartfelt gratitude goes to my parents, my mother **Liu Yu** and father **Chen Hu**. No matter where I am, you will always give me endless support and infinite tolerance. You have made everything possible for me.

Farewell, Torino——until we meet again.

Contents

Acknowledgements 2

List of Tables 5

List of Figures 6

1 Introduction 7
- 1.1 Background and Motivation . . . 7
- 1.2 Non-Graph-Based Machine Learning Methods . . . 7
- 1.3 Graph-Based Machine Learning Methods . . . 8
- 1.4 Thesis Structure . . . 8

2 High Level Synthesis 9
- 2.1 Introduction to High-Level Synthesis (HLS) . . . 9
  - 2.1.1 Significance and HLS Tools . . . 9
  - 2.1.2 HLS vs. RTL Design . . . 9
- 2.2 Principles and Workflow of HLS . . . 10
  - 2.2.1 Control and Data Path Extraction . . . 10
  - 2.2.2 Scheduling and Binding . . . 11
  - 2.2.3 RTL Generation . . . 11
- 2.3 Common HLS Pragmas . . . 11
  - 2.3.1 Loop Pragmas (Unrolling and Pipelining) . . . 11
  - 2.3.2 Array Partition Pragmas . . . 12
  - 2.3.3 Function and Module Pragmas . . . 12
- 2.4 Quality of Results (QoR) in HLS . . . 13
- 2.5 Vitis HLS Workflow . . . 13
  - 2.5.1 Pre-synthesis C Validation . . . 14
  - 2.5.2 Post-synthesis RTL Verification . . . 14

3 Machine Learning 15
- 3.1 Definition of Machine Learning . . . 15
- 3.2 Categories of Machine Learning . . . 15
- 3.3 Basic Steps of Machine Learning . . . 17
- 3.4 Introduction to Deep Learning . . . 18
- 3.5 Introduction to PyTorch . . . 19

4 Graph Neural Networks for High-Level Synthesis 21
- 4.1 Fundamentals of Graph Theory . . . 21
- 4.2 Graph Representation in HLS Designs . . . 21

4.3 Message Passing and Aggregation for GNNs . . . 21
4.4 Variants of GNNs (GCN, GAT, GATv2, GIN, DiffPool) . . . 22
4.4.1 Graph Convolutional Networks (GCN) . . . 22
4.4.2 Graph Attention Networks (GAT) . . . 24
4.4.3 GATv2 . . . 25
4.4.4 Differentiable Pooling (DiffPool) . . . 25
4.5 Applications of GNNs in EDA and HLS . . . 26

5 Dataset and data preprocessing 30
5.1 Dataset overview . . . 30
5.2 Data Preprocessing . . . 31

6 Model Framework 33
6.1 Basic Model . . . 33
6.2 Proposed Model . . . 33

7 Experimental process and results 38
7.1 Verification and selection of ablation models . . . 38
7.2 Comparative experiments with baseline . . . 41
7.2.1 Resource Utilization Comparison . . . 41
7.2.2 Expanding to the overall comparison of latency . . . 41

8 Conclusion 44

List of Tables

5.1 Node Features Description 31
5.2 Edge Features Description 32
7.1 Validation loss comparison for different models. 40
7.2 Comparison of RMSE values (DSP, LUT, FF, BRAM) between Baseline 1 and our proposed model. 41
7.3 Comparison of resource utilization (DSP, LUT, FF, BRAM) and latency (RMSE) between Baseline 2 and our proposed model. 42

List of Figures

2.1 Illustration of the Vitis HLS flow. [2] 10
3.1 The workflow of cross-validation.[3] 18
3.2 How pytorch works.[17] 19
4.1 Graph data structure, adjacent matrix and general process of GCNs[8]. . 23
4.2 Application of attention mechanism on graph convolution [21]. 24
4.3 Work flow of Diff-pooling [19] 27
5.1 Overall data preprocessing pipeline 32
6.1 Architecture of basic model 34
6.2 The architecture of the proposed model 34
6.3 Visual data flow diagram of the proposed modell 35
6.4 Differentiable pooling nodes aggregation processl 36
7.1 Visual data flow diagram of the proposed modell 39
7.2 The comparison of baseline 1 model and our proposed model 41
7.3 The comparison of resource utilization for Baseline 2 model and our proposed model 42
7.4 The comparison of latency for Baseline 2 model and our proposed model 43

Chapter 1

Introduction

1.1 Background and Motivation

High-Level Synthesis (HLS) lets designers use high-level programming languages such as C, C++ or SystemC to make faster hardware design prototyping by automatically transforming them into hardware designs. Traditional hardware design process requires low-level coding like Verilog or VHDL and this process is time-consuming and not cost-effective. HLS helps designers cut this time by focusing on algorithmic optimizations and not on low-level hardware details[12].

HLS tools have pragmas, which are essentially synthesis directives. Pragma choices define the performance and use of hardware resources like LUTs, FFs, DSPs and BRAMs of an arbitrary algorithm. By coupling high-level languages and pragmas, HLS tools allow designers to do faster design space exploration, i.e. trade-offs between performance and hardware cost. Current HLS tools do not provide dependable quality-of-results (QoR) estimations, and thus designers are unable to take advantage of faster design space exploration as the actual implementation results are different from the estimated QoR from the tools. Hence, designers need a way to better estimate QoR from HLS tools.

With rise of use of Machine learning (ML) to solve problems in various domains, it has also been adapted in the field of electronic design automation (EDA). Particularly, graph-based approaches like graph neural networks (GNNs) have been applied successfully to different steps of electronic design automation flow [13], [14], [24]. GNNs have also been used to improve the quality of the estimate of results in HLS [20] [9]. The models learns from past designs to predict better QoR. This makes design space exploration faster adhering to the HLS concept of faster design prototyping. However, existing GNN models provide better QoR results than commercial HLS tool but not as accurate as post-implementation results. This work proposes a hierarchichal GNN model using Differential Pooling (DiffPool) to process both structurual and hierarchical input graph information to estimate post-implementation results given an arbitrary design.

1.2 Non-Graph-Based Machine Learning Methods

Early approaches for QoR prediction used non-graphical machine learning models. These methods extracted features from HLS synthesis reports. These features included resource counts, timing, and pipeline details. They used these features in regression models or multi-layer perceptrons. For example, Dai et al. (2018) [5] extracted features from HLS reports and trained models like linear regression, gradient boosting, and simple

neural networks to predict resource usage and timing.[15] use various models like Linear Regression, Random forest, Support Vector Machines (SVM), Artificial Neural Network (ANN) and an ensemble of the four models. However, these approach use time-consuming scheduling and binding steps as they require inputs extracted from HLS reports.

1.3 Graph-Based Machine Learning Methods

Recently, graph neural networks (GNNs) have been used to model hardware designs. These methods treat circuits or netlists as graphs. Graph aids in representing this complex information by establishing relationships between different objects explicitly. GNNs learn from these input graph structure and contextual information. GNNs pass information through connections (edges) in a graph. Hardware performance depends not only on the number of operations but also on how they are connected. Graphs capture this information. Early research represented C/C++ code as graphs. Nodes represented operations or basic blocks. Edges showed data/control dependencies. Researchers trained GNNs to predict QoR, including area, power and latency. GNNs are used to generate graph-level embeddings which summarizes an input graph and then these embeddings are feed to non-graph based models like MLPs to predict final objectives. For example, Jamal et al. (2023)[9] introduced a GNN model that takes a design as an input which is represented as a graph and predicts post-implementation quality-of-results. [20] proposed a GNN-based model to estimate quality of a design to enable faster design space exploration. Deng et al. [6] introduced an hop-wise attention based GNN model for scalable and generalizable circuit representation learning. Their hop-wise graph attention network (HOGA) reduced QoR prediction error by 47% compared to a standard GNN while training faster.

1.4 Thesis Structure

The thesis is composed of 8 chapters. Chapter 1 presents the background and motivation behind this work. Chapter 2 discusses typical HLS flow and common HLS pragmas. Chapter 3 and 4 describes Machine Learning and Graph Neural Network respectively. Chapter 5 details the dataset used in this work. Chapter 6 proposes different GNN based model to target HLS QoR prediction problem. Chapter 7 provides the experiments performed and comparison with other GNN-based approaches and Chapter 8 summarizes the achievements in this thesis.

Chapter 2

High Level Synthesis

2.1 Introduction to High-Level Synthesis (HLS)

High-Level Synthesis (HLS) is an automated design process that generates register-transfer level (RTL) hardware designs from a high-level behavioral description. In HLS, designers describe the desired functionality in a high-level language (such as C/C++ or SystemC) instead of writing low-level HDL code. The HLS tool then handles the detailed hardware implementation, creating cycle-accurate RTL circuits that realize the given behavior. This raises the design abstraction level and lets engineers focus on algorithmic functionality rather than hand-coding clock-cycle operations. HLS is sometimes called behavioral or C-based synthesis, reflecting its use of high-level programming constructs to describe hardware [16].

2.1.1 Significance and HLS Tools

HLS has grown popular because it can greatly shorten development time and time-to-market for complex digital systems. By working at a higher level of abstraction, designers can more easily explore different architectures and make changes without dealing with low-level details. This is especially useful for today's heterogeneous systems and FPGA designs, which demand fast design iterations and high performance. In recent years, many HLS tools have emerged. Examples include AMD Vitis HLS (before Vivado HLS), Intel HLS Compiler and Cadence Stratus. Each tool may support different input languages and apply various optimizations, but all share the goal of automatically producing hardware from high-level code. HLS tools allow hardware acceleration of algorithms by leveraging software-oriented design entry, making hardware design more accessible to engineers with a software background.[26]

2.1.2 HLS vs. RTL Design

Compared to traditional RTL design, HLS offers significantly higher design productivity. In a RTL-based flow, the designer manually defines the exact clock-cycle behavior and datapaths using HDLs, which is time-consuming. HLS automates this process by automatically transforming an input source code and generates the RTL. This productivity gain comes from HLS handling the scheduling, concurrency, and hardware mapping automatically. However, one trade-off is that the *quality of results* (QoR) – in terms of performance and resource usage – from HLS might not always match a post-implementation design.

Historically, the RTL approach could yield better optimized circuits (e.g. higher clock speeds or lower area), but the gap is continually narrowing as HLS tools improve. HLS is now considered a viable option for fast prototyping and designs with tight deadlines, where development speed is more critical than squeezing out every last bit of performance.

2.2 Principles and Workflow of HLS

The HLS design flow starts with a high-level functional specification (in C, C++, SystemC, etc.) and ends with a cycle-accurate hardware description (RTL). The process involves several key steps to bridge the gap between the high-level code and the hardware implementation.Figure 2.1 shows the general workflow. This process generally includes several key steps:

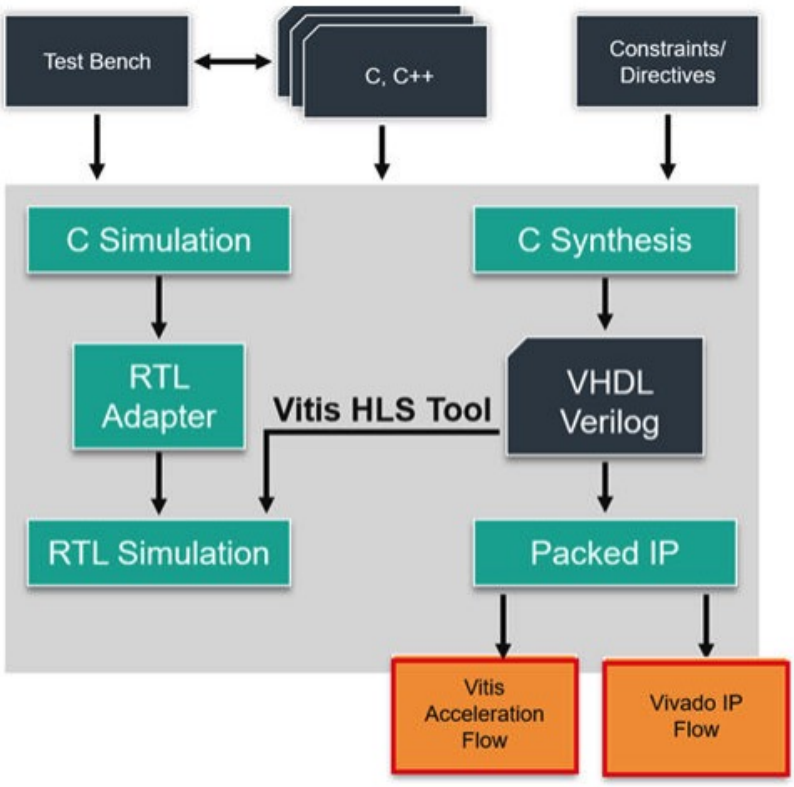

Figure 2.1. Illustration of the Vitis HLS flow. [2]

2.2.1 Control and Data Path Extraction

First, the HLS tool analyzes the input program to identify its control flow and data operations. It extracts the program's control structure (e.g., loops, conditionals) and builds a control-flow graph, which will become the finite state machine (FSM) that controls the hardware[11]. In parallel, the tool identifies the data path – the arithmetic and logic operations and data transfers needed to implement the algorithm. Essentially, HLS separates *what* computations happen from *when* they happen. The control flow (decision points, loop structure) is mapped into a hardware controller (FSM), while the computations (like additions, multiplications, memory accesses) form the datapath that

the FSM orchestrates. This separation of control and datapath is a fundamental step before scheduling the operations in time.

2.2.2 Scheduling and Binding

Next, the HLS tool schedules the operations into clock cycles and binds those operations to specific hardware resources. **Scheduling** determines *when* each operation will execute (in which clock cycle or state). The scheduler aims to meet timing constraints (like a target clock period) and maximize parallel execution where possible. For example, independent operations might be scheduled in the same cycle to execute concurrently if resources allow. **Binding** (and the related step of resource allocation) assigns each operation to a hardware resource (such as an adder, multiplier, or memory port). This means deciding which functional unit will perform each operation and whether certain units are time-multiplexed (shared) or duplicated. Allocation decides how many instances of each resource type to use, and binding maps the operations and variables to those instances.

2.2.3 RTL Generation

After scheduling and binding, the HLS tool generates the RTL code (usually in Verilog or VHDL) that corresponds to the decided micro-architecture. This RTL description includes a finite-state machine for the control flow and the datapath components (ALUs, multiplexers, registers, etc.) connected to perform the operations in the scheduled order. Essentially, the high-level algorithm is now realized as a cycle-by-cycle HDL design. For example, if the input was a C function, the tool produces an equivalent hardware module that implements that function, complete with handshaking or clock enable signals as needed.

2.3 Common HLS Pragmas

High-Level Synthesis (HLS) tools often allow the use of *pragmas* or *directives*, which are special instructions that guide the compiler's optimization decisions. Pragmas act as hints added as comments or attributes in the source code. They do not change the algorithm's functionality but influence how the hardware is generated. They help designers specify micro-architectural choices that the compiler might not automatically infer. For example, a pragma can direct the HLS tool to pipeline a loop or partition a memory array, modifying the performance and resource usage of the generated hardware. In general, HLS pragmas help control loop optimizations, concurrency, memory structures, and interface behaviors without modifying the core algorithm [11].

2.3.1 Loop Pragmas (Unrolling and Pipelining)

Loop pragmas determine how loops are implemented in hardware. *Loop unrolling* replicates the loop body to create multiple parallel iterations. For instance, unrolling a loop by a factor of 2 means the hardware will execute two iterations simultaneously, doubling throughput at the cost of increased resource usage. This exposes more parallelism across loop iterations and significantly speeds up computations, though it generally increases area since more hardware is active concurrently.

Loop pipelining allows a new loop iteration to begin before the previous one finishes. The HLS tool automatically inserts pipeline registers and schedules operations so that successive iterations overlap in time. This can achieve an initiation interval (II) of 1 cycle, meaning a new iteration starts every clock cycle, greatly improving throughput for larger loops. However, pipelining often requires resolving data dependencies (ensuring that later iterations do not read data too early) and may increase the number of registers (flip-flops) used for pipeline stages.

Together, unrolling and pipelining are powerful optimizations. Unrolling increases parallel hardware utilization per cycle, while pipelining keeps hardware busy every cycle by overlapping execution [18]. Designers often use both pragmas to achieve performance goals.

2.3.2 Array Partition Pragmas

Array partition pragmas control how arrays (or memories) are implemented. In high-level code, an array is usually stored as a single block of memory. In hardware, if multiple accesses to the array are needed in the same cycle (such as in an unrolled loop), this single memory block can become a bottleneck. An *array partition* pragma tells the HLS tool to split an array into multiple smaller memories or distribute it across several banks.

By partitioning an array, the design can perform multiple simultaneous accesses (one per memory partition) without conflicts. For example, partitioning an array of size 16 into two banks of size 8 allows two independent read/write operations per cycle, effectively doubling the memory bandwidth. This optimization is crucial for increasing parallelism when the algorithm frequently accesses arrays. The trade-off is that duplicating or banking memory may use more block RAMs (BRAMs) or registers. Pragmas can specify different partitioning methods, such as complete, block, or cyclic partitioning, based on the access pattern of the data.

2.3.3 Function and Module Pragmas

Function and module pragmas influence how functions or code blocks are implemented in hardware. One common directive is *function inlining*. By default, a function call in C/C++ can be compiled into a separate hardware module with its own internal logic, which the top-level hardware calls.

However, inlining a function expands its code in place at the call site. This exposes more optimization opportunities to the HLS tool, similar to how instruction-level parallelism can improve software performance. Inlining may allow operations inside the function to be scheduled in parallel with the caller. On the other hand, keeping functions as separate modules enables reuse when a function is called multiple times, reducing area usage by sharing one hardware implementation.

Pragmas allow users to force a function to inline or remain separate based on the design requirements. Additionally, module pragmas control interface protocols and resource allocation. For instance, a pragma can specify that a function should follow a specific interface protocol, such as `AXI4-Stream` or `AXI4-Lite`, for seamless integration with other IP cores.

Other pragmas can set limits on how many instances of a specific operation are used (e.g., limiting multipliers to a certain number, thereby enforcing resource sharing). Additionally, a pragma can bind an operation to a specific hardware implementation, such as using a DSP block for multiplication instead of lookup tables (LUTs).

In summary, function and module pragmas help structure the hardware design at a higher level. They control hierarchy, interfaces, and resource allocation, giving designers a way to guide the HLS tool's decisions. These pragmas help ensure that the generated RTL aligns with the desired architecture and integrates efficiently into the larger system.

2.4 Quality of Results (QoR) in HLS

In High-Level Synthesis (HLS), Quality of Results (QoR) is a key metric used to evaluate and optimize hardware designs. QoR generally encompasses various factors such as resource usage, timing, power consumption, and area. Below are some of the key resources and QoR parameters involved in HLS:

- **LUT (Look-Up Table)**: LUTs are fundamental components used to implement Boolean functions in digital circuits. In FPGA designs, LUTs are widely used for constructing logic gates and arithmetic units. The number and size of LUTs directly affect resource usage and performance. A higher LUT usage indicates more complex logic, but may lead to increased resource consumption and slower timing.

- **FF (Flip-Flop)**: FFs are the basic building blocks of sequential logic, used to store data and maintain state. The number and arrangement of FFs are crucial for timing control. More FFs typically lead to higher area and power consumption.

- **DSP (Digital Signal Processor)**: DSP units are specialized hardware components designed to efficiently perform arithmetic operations such as multiplication and accumulation. DSP usage accelerates computation-intensive tasks, but may increase resource consumption.

- **BRAM (Block RAM)**: BRAM provides on-chip memory for storing data. It is used to store intermediate results, caches, and FIFOs. Excessive BRAM usage may increase resource consumption, but its efficient use can significantly improve memory access speed.

- **Latency**: latency refers to the total number of clock cycles required for a hardware design to complete.

By understanding and optimizing these QoR parameters, designers can ensure that the HLS-generated design meets performance, resource, and power requirements. Optimizing for one QoR metric (such as performance) may often affect others (such as area and power), so careful trade-offs must be made during the design process.

2.5 Vitis HLS Workflow

Vitis HLS follows a structured workflow to ensure that the high-level design is functionally correct and that the generated RTL meets the required specifications. The workflow consists of two primary phases: *C-level validation* (before synthesis) and *RTL verification* (after synthesis).

2.5.1 Pre-synthesis C Validation

In this step, the original C/C++ code is validated to ensure the algorithm functions as expected. Designers write a test bench in C/C++ to exercise the high-level function with various inputs and verify the outputs. This step effectively simulates the algorithm in software.

Vitis HLS provides a complete C simulation environment, including support for bit-accurate data types such as arbitrary precision integers and fixed-point types. These allow designers to model hardware-specific behaviors precisely in C. The C simulation runs quickly and enables efficient debugging of functional issues. The primary goal is to detect and fix errors in the algorithm before committing to hardware generation.

For instance, if the design is an image filter, an engineer can run the C model on sample images to verify the correctness of the filtering process. This step does not involve any hardware yet; it is purely a check to confirm that the C code (which will later be synthesized) behaves correctly. Debugging at this stage is much easier than troubleshooting errors after RTL generation.

2.5.2 Post-synthesis RTL Verification

Once the C model is verified, the Vitis HLS tool synthesizes the C/C++ code into RTL. After synthesis, Vitis HLS generates Verilog/VHDL output along with reports on estimated performance and resource usage. The next step is to verify that the synthesized RTL design is functionally equivalent to the original C model.

Vitis HLS supports *C/RTL co-simulation* to facilitate this verification. In co-simulation, the same C test bench (or an automatically generated test bench) is used to apply input stimuli to the synthesized RTL model. The simulation runs the RTL design (often using a built-in simulator) and compares its outputs against the expected results obtained from the C simulation.

Essentially, the tool uses the C model's outputs as a reference and checks whether the RTL produces identical results, either cycle-accurately or at least frame-accurately in streaming applications.

If the RTL output does not match the expected results, the designer must debug the discrepancies. Possible issues include undefined behaviors in the original C code that manifest differently in hardware or pragmas that modify latency in a way that was not accounted for in the test bench.

Co-simulation ensures confidence in the final design by verifying that *"what you simulated in C is what you get in hardware."* Once the RTL passes co-simulation, Vitis HLS allows exporting the design as an IP core with standard interfaces, such as `AXI4-Stream` or `AXI4-Lite`, enabling integration into a larger FPGA system.

At this point, the typical FPGA design flow—*place-and-route* and *bitstream generation*—takes over. However, from an HLS perspective, the workflow has validated the design at both the algorithmic level and the RTL level.

Chapter 3

Machine Learning

Machine Learning (ML) is an important branch of artificial intelligence (AI). It analyzes data, learns from experience, and makes predictions based on existing data without the need for humans to write specific rules [1]. Traditional programming requires engineers to manually write code to define how each task is performed, while machine learning allows computers to automatically discover patterns in data and gradually optimize their performance. Today, machine learning has been widely used in various aspects, such as computer vision, natural language processing, audio processing, and even wider fields.

3.1 Definition of Machine Learning

Machine Learning (ML) is a technology that allows computers to discover patterns or regularities in large amounts of data without completing prediction tasks through specific programmatic instructions. The key elements of machine learning include data, features, models, training, evaluation, and optimization. Data quality directly affects model performance, while feature engineering and optimization methods can improve learning results. According to the learning method, machine learning is mainly divided into supervised learning (such as classification and regression, which rely on labeled data), unsupervised learning (such as clustering and dimensionality reduction, which are used to discover data structures), and reinforcement learning (intelligent agents learn optimal strategies through trial and error and reward mechanisms, which are applied to autonomous driving, robot control, etc.). It is widely used in computer vision, natural language processing, recommendation systems, financial analysis and other fields, and is an important foundation of artificial intelligence.

3.2 Categories of Machine Learning

The types of machine learning can be divided according to whether label information is required and the different training methods. The following will introduce three typical types:

- **Supervised Learning:** Supervised learning must be trained with corresponding labeled data. The purpose is to find the mapping relationship between input and output, and make predictions on new data based on this relationship. Supervised learning mainly includes: Classification: used to predict discrete category labels,

such as spam detection (determining whether an email is spam or normal), handwritten digit recognition (MNIST data set), and disease diagnosis (determining whether a tumor is benign or malignant). Regression: used to predict specific values, such as house price prediction, weather forecast, and stock market trend prediction. Common algorithms for supervised learning include tree models, logistic regression, decision trees, random forests, support vector machines, and neural networks. Today's popular deep learning networks such as convolutional neural networks, recurrent neural networks, and graph neural networks also belong to the supervised learning part. The task of this article is the regression task in supervised learning.

- **Unsupervised Learning:** Unsupervised learning does not require manual annotation of data for learning. This learning method will automatically discover hidden patterns in the data and complete the required tasks. Common tasks include clustering and dimensionality reduction. Clustering methods are used to group data according to similarities. For example, in market analysis, e-commerce platforms can divide customers into different groups based on user purchasing behavior in order to personalize product recommendations; in the medical field, researchers can use genetic data to classify symptoms and help discover new disease categories (new category discovery). Dimensionality reduction can reduce the dimension of data and retain useful information as much as possible. For example, principal component analysis (PCA) can convert high-dimensional data into low-dimensional representations or improve computational efficiency. In addition, if you want to visualize or understand, dimensionality reduction is also a very intuitive way. Unsupervised learning is also commonly used for anomaly detection. For example, banking systems use unsupervised learning models to analyze transaction patterns and discover possible credit card fraud. In addition, unsupervised learning is also widely used in feature learning, natural language processing (NLP), and image generation, such as stable diffusion, and has become a hot research direction in the field of AI. Although the results of unsupervised learning are usually not as intuitive and explainable as those of supervised learning, its feature of not requiring manual labeling greatly reduces the cost of training and has become the mainstream of research today.

- **Reinforcement Learning:** Unlike supervised learning, reinforcement learning does not rely on training data. It learns the optimal decision-making strategy through trial and error and reward mechanisms by continuously taking actions in the environment and observing the feedback from the environment to maximize long-term returns. Reinforcement learning has become a hot field in AI today because the concept of intelligent agents and their learning methods are closer to biological organisms. For example, in robot control, Boston Dynamics' robots learn how to move in different complex environments through reinforcement learning, such as going up and down stairs and gaining balance by jumping in ruins; in the field of autonomous driving, reinforcement learning algorithms can learn response strategies under simulated traffic conditions, allowing autonomous driving systems to make autonomous decisions in complex traffic environments. In sports events, AlphaGo developed by DeepMind defeated the world champion in the Go game through reinforcement learning. It learned the optimal strategy by playing against the intelligent agent itself and improved its Go game ability. In the financial field, reinforcement learning can continuously learn more favorable strategies through feedback such as return on investment to maximize returns. The core methods of reinforcement learning

include value-based algorithms (such as Q-learning), policy-based methods (such as REINFORCE), and model-based methods (such as AlphaZero). Different application scenarios correspond to different reinforcement learning methods. Although reinforcement learning has a high computational cost and the training process may be slow, it has great potential in scenarios that require long-term planning and complex decision-making, and has become one of the hot research directions in artificial intelligence.

3.3 Basic Steps of Machine Learning

The typical workflow for a machine learning project involves several key stages:

- **Data Collection and Preprocessing:** Data collection is the cornerstone and even the most important step of supervised learning. When doing computer vision or natural language processing related tasks, our data sources often come from the Internet or some databases and libraries. Engineers often use crawlers and other technologies to build databases that meet the task requirements. In our task, we selected the open source data set on the machine learning competition website kaggle. This data set is generated through three versions of Vitis HLS and is authentic and professional. The data preprocessing process is also an important part of determining data quality. Specific operations include cleaning data, removing noise, and normalizing data.

- **Feature Engineering:** In the training of traditional machine learning and graph neural networks (GNNs), feature engineering is a key link. Feature engineering will extract features that reflect the characteristics of the data from the original raw data, which will enable the model to learn better and make predictions. Therefore, selecting so-called good features is very important for the process of machine learning. For other deep learning methods (such as CNNs, RNNs, Transformer, etc.), feature engineering will be automatically generated through convolutional layers or encoders. For the graph neural network (GNNs) used in this article, we need to manually extract the initial features, and then the graph convolution layer automatically extracts deeper features through the graph structure and edge attributes.

- **Model Selection and Training:** After the data processing is completed, we need to build the corresponding training model. For the deep learning process, this process is similar to building blocks. It is necessary to select appropriate components (such as convolutional layer, pooling layer) according to the characteristics of the task and data. In addition, we also need to specify the corresponding optimizer, loss function, and training strategy. We often have multiple candidate models with different model structures and hyperparameters. We can select models and hyperparameters through verification methods. K-fold cross-validation is a good way. This method divides the data set into K parts, uses K-1 parts to train the model each time, and the remaining part is used as a test set. This process is repeated K times, using different parts as test sets each time, and finally taking the average result. This can reduce the risk of overfitting and ensure that the performance of the model does not depend on a specific data partition. After multiple verifications, we can select the model with the best performance through loss and use it as the final model.

- **Model testing and deployment:** After selecting the model, we will use a completely independent test set to test the model and compare it with other baseline models on the test set. After proving the superiority of the model, we will deploy the model to the actual environment. In actual applications, the model needs to process new data and make predictions. The deployment method can be local operation, cloud computing, or integrated into the application for real-time reasoning.

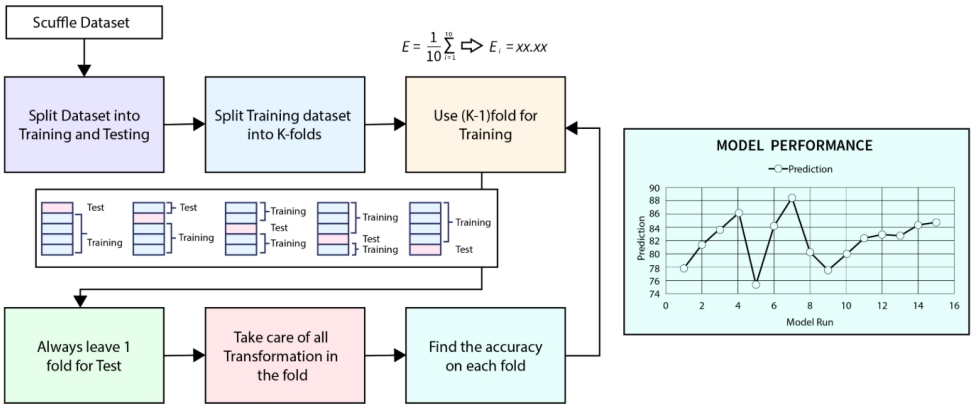

Figure 3.1. The workflow of cross-validation.[3]

3.4 Introduction to Deep Learning

- **What is deep learning?** Deep learning is a part of machine learning. It helps computers learn patterns from data. It builds multi-layer neural networks to do classification or prediction. Traditional machine learning needs human-designed features. Deep learning does not need this. It can extract features automatically. Because of this, deep learning works well in image recognition, speech recognition, and natural language processing.

- **How neural networks work** The key to deep learning is artificial neural networks (ANNs). These networks have many neurons. Each neuron gets input, does calculations, and sends the result to the next layer. The process has two parts: forward propagation and backpropagation. Forward propagation finds the output from the input. Backpropagation reduces errors by adjusting neuron weights. This makes the model more accurate. The model keeps learning from data.

- **Main deep learning models** Deep learning has different types of neural networks. Each type is used for different tasks. Convolutional Neural Networks (CNNs) are common for image processing. CNNs find edges and shapes in images. They help computers recognize objects and detect faces. Recurrent Neural Networks (RNNs) work with time series data. They are used in speech recognition and text analysis. RNNs remember past data, so they predict the next step well. This is useful in natural language tasks. Generative Adversarial Networks (GANs) are used for making images and style changes. A GAN has two parts: a generator and a discriminator. The generator creates new images. The discriminator checks if they are real. They compete and improve. This makes high-quality images. Variational Autoencoders

(VAEs) are another type of deep learning model. They help with data generation and noise reduction.

- **Application of deep learning** Deep learning is used in many areas. In medical diagnosis, it helps analyze CT scans and X-rays. It helps doctors find diseases and improve accuracy. In autonomous driving, it detects roads, people, and traffic signals. It helps cars make decisions. In recommendation systems, it studies user behavior. It suggests movies, music, and products. In finance, it predicts stock trends, scores credit, and finds fraud. This helps manage risk. Deep learning is also used in robot control, language translation, and voice assistants.

3.5 Introduction to PyTorch

PyTorch is an open-source deep learning framework developed by Facebook's AI Research Lab. It is designed for both research and production environments, providing a flexible and efficient platform for building deep learning models. PyTorch is known for its dynamic computation graph, making it highly flexible and intuitive for experimentation. Key

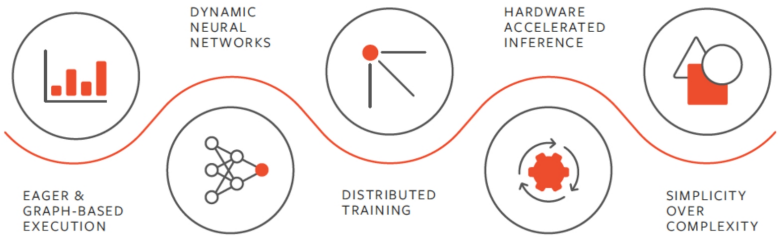

Figure 3.2. How pytorch works.[17]

features of PyTorch include:

- **Dynamic Computational Graphs:** PyTorch builds computational graphs dynamically during execution, allowing the model structure to change as the computation progresses. This is particularly useful for complex models and research.
- **Automatic Differentiation:** PyTorch provides an automatic differentiation library called `autograd`, which computes gradients for backpropagation, simplifying the training process.
- **GPU Acceleration:** PyTorch supports seamless integration with GPUs, which allows for faster computation, especially when dealing with large datasets and complex models.
- **TorchScript:** PyTorch allows models to be exported using TorchScript, which makes it possible to run the model in non-Python environments, such as production systems.

PyTorch has become one of the most popular frameworks for deep learning due to its flexibility, user-friendly interface, and strong support for research and production workflows.

It is also worth mentioning that PyTorch Geometric (PyG) is a deep learning library that specializes in processing graph structure data. It is built on PyTorch and can efficiently implement various graph neural network (GNN) models. PyG provides a wealth of graph data processing tools, model libraries (such as GCN, GAT, GIN, etc.) and flexible module combinations [7]. It can be easily applied to tasks such as node classification, link prediction, and graph classification. It is widely used in molecular structure prediction, recommendation systems, and electronic design automation. This project mainly processes graph data, so it mainly uses this deep learning library.

Chapter 4

Graph Neural Networks for High-Level Synthesis

4.1 Fundamentals of Graph Theory

A graph is a data structure consisting of a set of nodes (vertices) and edges that connect certain pairs of nodes [14]. Edges can be directed (with a source and target) or undirected, and can optionally carry weights or labels. In many use cases, each node and edge can have associated feature vectors describing their properties (e.g. an operation type for a node, or a dependency type for an edge). This means a graph can be represented not just by its topology but also by matrices of node features and edge features. Graphs provide a flexible way to model relationships in structured data, which is especially useful in representing circuits and programs in hardware design contexts.

4.2 Graph Representation in HLS Designs

In an HLS, an input program (in C/C++ or similar) is often converted into an intermediate representation (IR) that can be modeled as a graph. A common choice is the Control and Data Flow Graph (CDFG), which combines the control flow graph (CFG) of the program (representing basic blocks and their execution order) with the data flow graph (DFG) (representing operations and data dependencies). The merged CDFG captures both control and data dependencies in one structure. This has several advantages: for example, a CDFG derived from the program's IR (e.g. LLVM IR) naturally reflects loop hierarchies and can be annotated to show which loops or operations are parallelizable or share resources. In addition to control/data edges, HLS-specific pragmas (compiler directives for optimizations like pipelining or unrolling) can be represented in the graph. For instance, one approach constructs a graph where nodes represent operations (and possibly functions or loops) and edges encode control flow, data flow, function calls, and pragma relationships, so that the effect of HLS directives is included in the graph structure

4.3 Message Passing and Aggregation for GNNs

Graph Neural Networks (GNNs) are neural models that operate on graph-structured data, updating node representations by exchanging information along the graph's edges. The fundamental mechanism in most GNNs is message passing: at each layer, every node

gathers "messages" from its immediate neighbors, aggregates these incoming messages, and uses the result to update its own embedding (feature vector). For example, a simple GNN layer might sum or average the feature vectors of all neighbor nodes (and sometimes include the node's own previous features) to compute the new feature for the node. More advanced GNNs use learnable aggregation functions – such as attentional weighting of neighbors or learned combination via neural networks – but the core idea is similar: the node's new state is a function of its neighbors' states from the previous iteration. After stacking multiple such layers, each node's final embedding captures information from its k-hop neighborhood (if k layers are used). In summary, GNNs perform iterative neighbor-based feature aggregation, preserving both the features and the connectivity of the graph in the learned representations.

4.4 Variants of GNNs (GCN, GAT, GATv2, GIN, DiffPool)

There are many variants of GNNs. I will introduce several relevant to this article in detail.

4.4.1 Graph Convolutional Networks (GCN)

GCNs apply a convolution-like operation on graphs by having each node aggregate features from its neighbors, typically using a weighted sum or average [14]. In the classic GCN model, the neighbor features are averaged (with normalization by node degrees) and passed through a linear transform and nonlinearity to produce new node features. GCNs are simple and efficient, often achieving good performance with just 2–3 layers of propagation, as 4.1 shows. A known limitation of the original GCN formulation is that it is transductive, meaning the model is trained assuming a fixed graph – it cannot naturally generalize to completely new nodes unseen in training. Additionally, if one stacks many GCN layers, the risk of over-smoothing arises (node embeddings becoming indistinguishably similar), so GCNs typically work best with shallow depths. Overall, GCNs provide a strong baseline that emphasizes localized neighbor information averaging.

Here is the mathematical explanation:

Graph Representation: A graph is defined as:

$$G = (V, E) \tag{4.1}$$

where:

- V: The set of nodes (vertices).
- E: The set of edges.

The structure of the graph is represented using an adjacency matrix A of size $N \times N$, where:

$$A_{ij} = \begin{cases} 1, & \text{if an edge exists between nodes } i \text{ and } j, \\ 0, & \text{otherwise.} \end{cases} \tag{4.2}$$

This matrix captures the connectivity between nodes.

Each node v_i has a feature vector x_i, and the feature matrix for all nodes is:

$$X \in R^{N \times F} \tag{4.3}$$

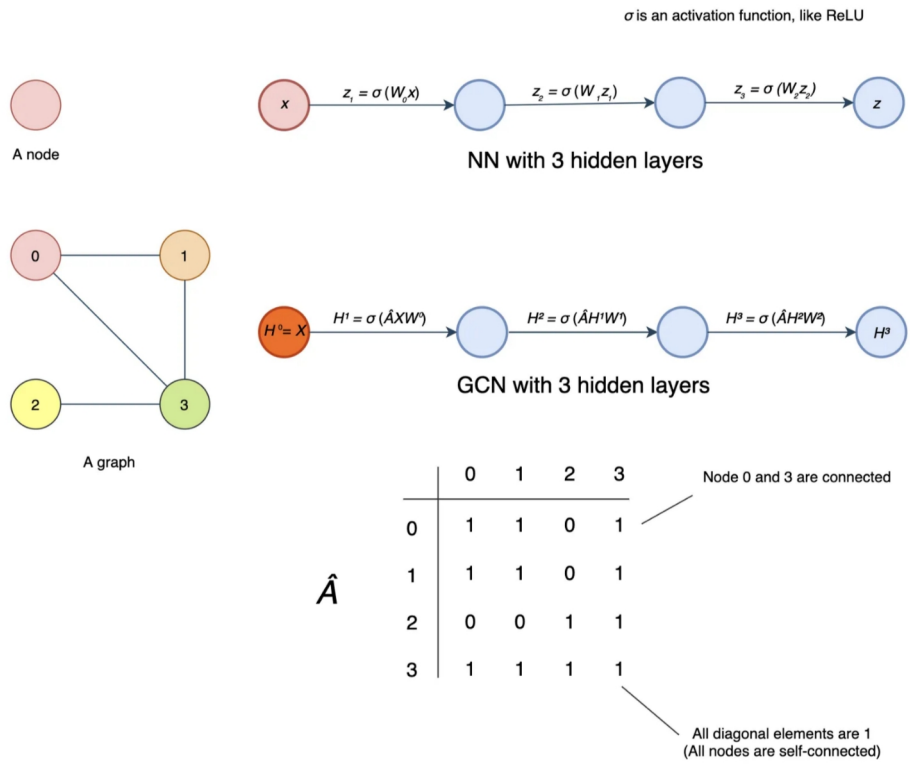

Figure 4.1. Graph data structure, adjacent matrix and general process of GCNs[8].

where:

- F is the number of input features per node.
- X stores node attributes and is updated at each layer of the GCN.

Forward Propagation in a Single-Layer GCN: GCN updates node features by aggregating information from neighboring nodes. The propagation rule for a single-layer GCN is:

$$H^{(l+1)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(l)}W^{(l)}\right) \tag{4.4}$$

where:

- $H^{(l)}$: The node feature matrix at layer l, with $H^{(0)} = X$.
- $W^{(l)}$: The trainable weight matrix at layer l, responsible for feature transformation.
- $\sigma(\cdot)$: The non-linear activation function, typically ReLU:

$$\sigma(x) = \max(0, x) \tag{4.5}$$

- $\tilde{A}$: The adjacency matrix with self-loops, where $\tilde{A} = A + I$, and I is the identity matrix. Adding self-loops ensures each node incorporates its own features during aggregation.
- $\tilde{D}$: The degree matrix, defined as:

$$\tilde{D}_{ii} = \sum_j \tilde{A}_{ij} \tag{4.6}$$

The degree matrix normalizes the adjacency matrix to balance feature aggregation.

The term $\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}$ ensures that feature propagation is normalized across nodes, preventing dominant nodes from overshadowing others.

Multi-Layer GCN: A deeper GCN is formed by stacking multiple layers of the above propagation rule:

$$H^{(L)} = \text{softmax}\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}H^{(L-1)}W^{(L-1)}\right) \tag{4.7}$$

where:

- L is the total number of layers.
- The softmax function is used for classification tasks, ensuring output probabilities sum to 1.

4.4.2 Graph Attention Networks (GAT)

GAT introduces a learnable attention mechanism to the aggregation process. Instead of treating all neighbor contributions equally or just by degree, a GAT layer computes an attention weight $\alpha_{(u,v)}$ for each edge (neighbor v to node u) based on the feature content, and uses these weights to form a weighted sum of neighbor features. In effect, each node can attend to different neighbors with different importance. The benefit is an increase in model expressiveness: the network can focus on the most relevant parts of a node's neighborhood for a given task [23]. Empirically, GATs often outperform GCNs on tasks where not all neighbors are equally informative. One weakness identified in the original GAT is that its attention mechanism was somewhat limited (static) – the ranking of neighbor importance did not depend on which node was querying (it was largely the same ordering for all) [4]. This can hinder GAT's ability to distinguish certain graph structures and was addressed by later improvements. The calculation mechanism of GAT is shown in the figure 4.2

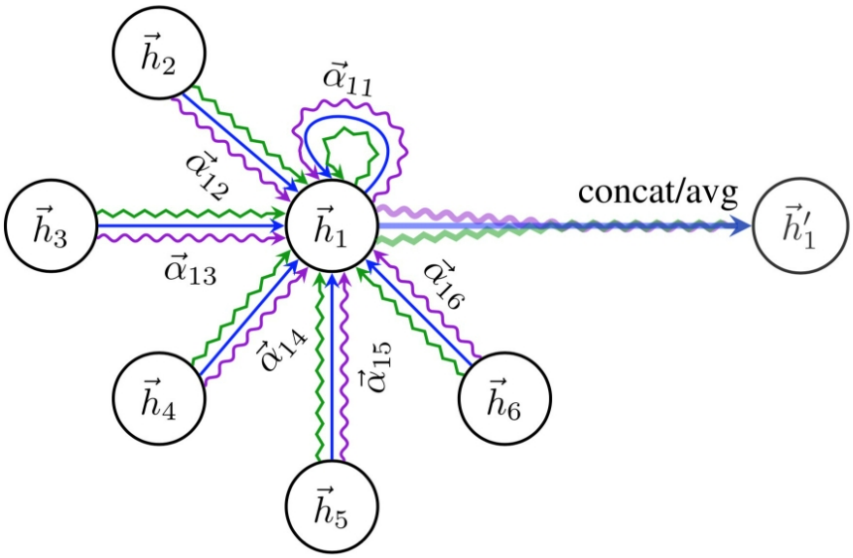

Figure 4.2. Application of attention mechanism on graph convolution [21].

Feature Transformation: Each node u has an initial feature vector h_u. All nodes undergo a linear transformation:

$$h'_u = Wh_u \tag{4.8}$$

where W is a trainable weight matrix.

Attention Score Computation: For each edge (u, v), the model computes an **attention score** e_{uv} based on node features:

$$e_{uv} = \text{LeakyReLU}\left(a^T \left[h'_u \parallel h'_v\right]\right) \tag{4.9}$$

where a is a **trainable attention vector** and $\parallel$ denotes **concatenation**.

Softmax Normalization: To ensure attention scores sum to 1 across neighbors:

$$\alpha_{uv} = \frac{\exp(e_{uv})}{\sum_{v' \in \mathcal{N}(u)} \exp(e_{uv'})} \tag{4.10}$$

Feature Aggregation: The final node feature update is computed as:

$$h_u^{(l+1)} = \sigma\left(\sum_{v \in \mathcal{N}(u)} \alpha_{uv} h'_v\right) \tag{4.11}$$

where σ is a non-linear activation (e.g., **ReLU**).

Multi-Head Attention: To improve stability, GAT applies K independent attention heads:

$$h_u^{(l+1)} = \sigma\left(\frac{1}{K}\sum_{k=1}^{K}\sum_{v \in \mathcal{N}(u)} \alpha_{uv}^{(k)} h_v'^{(k)}\right) \tag{4.12}$$

GAT enhances model flexibility by allowing nodes to prioritize important neighbors, unlike GCN which treats all neighbors equally.

4.4.3 GATv2

GATv2 is a revised version of the Graph Attention Network that fixes the static attention limitation. It modifies the internal computations of GAT to enable a more expressive, dynamic attention where the importance of an edge can truly depend on both the source and target nodes' features. The result is a model that is strictly more powerful in terms of attention expressiveness, while retaining the same order of computational complexity as the original GAT. In the paper that introduced GATv2, it was shown that this variant can fit patterns that vanilla GAT could not, and GATv2 achieved better accuracy than GAT on a variety of graph benchmarks without increasing the parameter count or runtime [4]. In summary, GATv2 strengthens the attention mechanism, making GNNs more capable of capturing complex relationships between connected nodes.

4.4.4 Differentiable Pooling (DiffPool)

Differentiable Pooling (DiffPool): Most GNNs propagate on the original graph structure ("flat" graphs), but DiffPool is a technique to introduce hierarchical graph representation learning Ying et al. (2018) proposed DiffPool as a differentiable graph pooling module that can be inserted into a GNN to gradually coarsen the graph [25]. DiffPool learns a soft assignment of nodes to a smaller set of clusters at certain layers, effectively compressing the graph in a learned way. This allows the network to create higher-level "summary" nodes (clusters) and build a hierarchy similar to pooling in CNNs, which is especially useful for graph-level tasks like classification of entire graphs. The advantage is improved capacity to capture long-range interactions and abstract patterns (it achieved

about 5–10% accuracy improvements on graph classification benchmarks by adding this hierarchy). However, DiffPool and similar pooling approaches come with challenges. They add extra parameters and computation for learning cluster assignments, and if the assignments are not perfect, the pooling can cause a loss of information – some fine-grained details of the original graph may be "averaged out" or lost in the coarser representation. In practice, DiffPool's soft clustering was observed to produce some noisy aggregations (indicating information loss in the coarsened graph) [22]. Thus, while DiffPool increases the modeling power for graph-level inference, it must be used carefully to balance granularity and scalability.
Here is the mathematical process:
Assignment Matrix Computation: Each node is softly assigned to a cluster using an assignment matrix S^l:

$$S^l = \text{softmax}(\text{GNN}_1, \text{pool}(A^l, X^l)) \tag{4.13}$$

where:

- $S^l \in R^{n_l \times n_{l+1}}$ contains cluster assignments.
- A^l is the adjacency matrix.
- X^l is the node feature matrix.
- GNN_1 extracts features for clustering.
- softmax ensures that assignments sum to 1.

Graph Coarsening: New coarsened node features X^{l+1} are computed as:

$$X^{l+1} = S^{lT} Z^l \tag{4.14}$$

where:

$$Z^l = \text{GNN}_1, \text{embed}(A^l, X^l) \tag{4.15}$$

The new coarsened adjacency matrix is:

$$A^{l+1} = S^{lT} A^l S^l \tag{4.16}$$

This updates the graph structure to match the compressed node set.And the following figure 4.3 shows the data flow of diff-pooling.

4.5 Applications of GNNs in EDA and HLS

GNNs have been increasingly applied in Electronic Design Automation (EDA) and HLS workflows, where many problems naturally involve graph-structured data (netlists, data flow graphs, etc.). Below are some key application areas along with research findings.

Graph Neural Networks (GNNs) have been increasingly applied in Electronic Design Automation (EDA) and High-Level Synthesis (HLS) workflows. Many problems in these domains involve graph-structured data, such as netlists and data flow graphs. Below are key applications of GNNs in these fields.

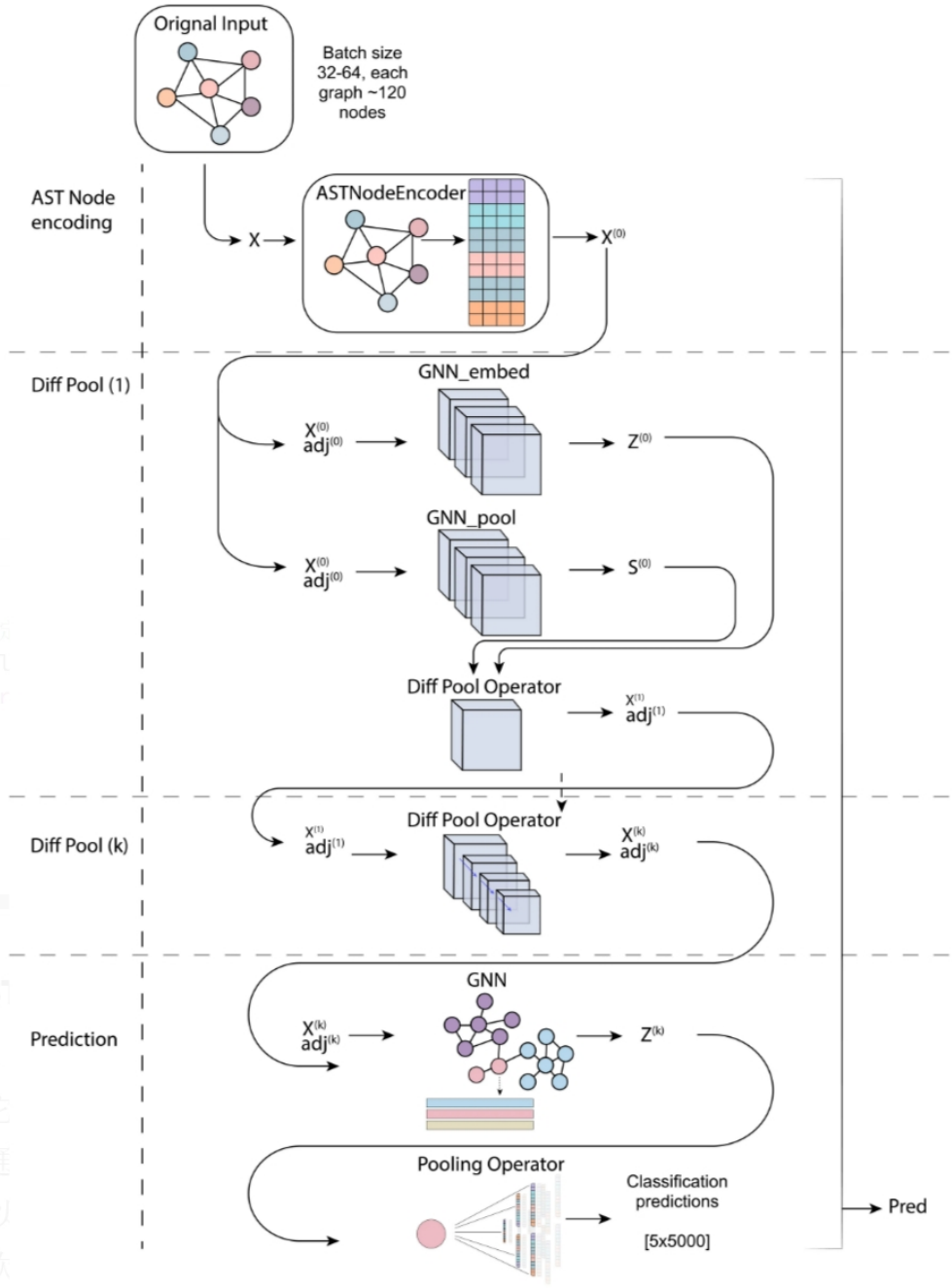

Figure 4.3. Work flow of Diff-pooling [19]

- **HLS Performance Prediction and Design Space Exploration** HLS Performance Prediction and DSE: GNN models are used to predict the quality-of-results of HLS designs (e.g. latency, area, power) given a particular code and pragma configuration. For example, a framework called GNN-DSE takes an HLS C/C++ kernel, explores possible pragma optimizations, represents each design variant as a graph (including control/data flow and pragmas), and predicts the FPGA performance (cycle count, resource usage) for each variant. This helps prune the design space by quickly identifying promising optimization combos without exhaustive synthesis runs. In one study, such a GNN-driven DSE could find near-optimal pragma settings

while avoiding dozens of slow HLS compilations. Similarly, GNNs have been used to estimate individual operation delays after HLS: Ustün et al. (2020) proposed D-SAGE, a GraphSAGE-based model on the HLS dataflow graph, which outperformed the HLS tool's own estimates for operation mapping and timing across all test cases [14]. These predictors enable faster design iterations by providing early feedback on performance and resource utilization.

- **Logic Synthesis and Resource Mapping** Graph-based learning has been applied to logic synthesis optimizations. The HLS or RTL netlist can be modeled as a graph (nodes as operations or gates, edges as connections), and GNNs can learn to predict outcomes of mapping or optimization. For instance, D-SAGE (above) not only predicted delays but also learned to classify which functional units map to which FPGA resource (LUT vs DSP, etc.), improving accuracy of resource mapping predictions. This kind of capability can guide synthesis tools to make better decisions. Another work called GRANNITE used a GNN (a sequential GCN) to predict average toggle rates (switching activity) for power estimation from an RTL netlist, achieving higher accuracy than conventional methods while running in seconds [14]. These examples show GNNs aiding different stages of logic synthesis and verification by leveraging the circuit graph structure.

- **Placement and Physical Design** Several EDA studies have employed GNNs in placement, routing, and floorplanning tasks. A notable example is Net^2, a graph attention model used to predict wirelength (net lengths) during placement planning [24]. By modeling the pre-placement netlist as a graph and using a GAT to estimate connection lengths, Net^2 could identify long critical interconnects with about 15% higher accuracy than prior heuristic approaches, and did so much faster (orders of magnitude speedup, e.g. 1000× faster in some cases). This helps designers estimate routing congestion and timing early on.

- **Analog Circuit Design and Optimization** Beyond digital circuits, GNNs have shown promise in analog and mixed-signal design where graph representations are natural (e.g., circuits as component graphs). CircuitGNN used a GCN to predict properties of circuit components (like predicting magnetic coupling effects in integrated inductors) by modeling the circuit schematic as a graph of components and connections. Another work combined GCN embeddings with an RL agent to perform transistor sizing (choosing device parameters in an analog circuit), where the GNN helped evaluate the design state for the agent[13]. These applications demonstrate that GNNs can learn complex device interactions and constraints in analog circuits, a realm traditionally governed by expert heuristics.

Across these applications, researchers are continuously improving GNN approaches. One trend is using hierarchical or multi-level GNN models to cope with large designs and to capture different levels of abstraction. For example, a recent hierarchical GNN model for HLS learned to aggregate information from inner-loop graphs to outer loops, achieving post-route QoR predictions with under 10% error on average and shrinking design-space exploration time to minutes. Another direction is combining GNNs with other AI techniques like reinforcement learning or evolutionary search to handle sequential decision problems in EDA. In floorplanning, as noted, GNNs embedded in RL have shown state-of-the-art results, and more generally, an "ML-assisted design flow" is envisioned where GNNs rapidly evaluate design choices so that search algorithms can find optimal

solutions faster. As these techniques mature, we expect GNNs to play an even larger role in accelerating EDA tasks, from high-level synthesis down to physical design.

Chapter 5

Dataset and data preprocessing

In this study, we try to explore the possibility and superiority of differentiable pooling (DiffPool) in resource utilization and latency prediction of HLS QoR in graph neural networks, and we evaluate and test our method on the open source dataset of Kaggle. Our research has the following highlights: 1. Pragma information will be directly embedded in the control data flow graph (CDFG) through vertex edges. 2. Explore the gains of the new components Graph Attention Network v2 (GATv2), Global attention pooling and Differentiable Pooling (Diffpool) for the model. We first use the k-fold cross-validation method and an ablation study idea to select the best model, which is the combined model architecture of Graph Attention Network v2 Convolutional Layer (GAT2) + Global attention pooling + Differentiable Pooling (Diffpool).

5.1 Dataset overview

The dataset used in this study comes from the "Machine Learning Contest for High-Level Synthesis" competition on the Kaggle platform [10]. The dataset contains 42 different programs or kernels, representing different computing modes. The target objectives are *LUTs*, *FFs*, *DSPs*, *BRAMs* and *Latency*. In addition, three key optimization instructions (pragmas) are involved in the dataset:

- **TILE**: Divide the loop into smaller blocks to optimize memory access and improve the efficiency of parallel execution.
- **PIPELINE**: Enhance instruction-level parallelism through loop pipelining.
- **PARALLEL**: Unroll the loop to increase parallelism, but may result in increased resource usage.

Each design point consists of a specific pragma configuration. There are a total of 14,135 different design points in the dataset. The labels of the designs in the training set are mainly obtained using AMD/Xilinx's Vivado HLS tool (versions are Vitis 2020.2 and Vivado HLS 2018). The labels of the designs in the test set are obtained using the latest version of the Vivado HLS tool (Vitis 2021.1). In addition, the dataset also provides a limited number of fine-tuning sets, whose labels are also obtained using Vitis 2021.1. The dataset includes the following files:

- **train_designs**: Contains all design data for training.
- **test.csv**: Provides the design configuration of the test set for model testing.
- **Graphs**: Stores the graph representation of each kernel.
- **Sources**: Source code files corresponding to each kernel.

5.2 Data Preprocessing

Our data preprocessing starts with the raw data on Kaggle which contains 42 different kernels and their various design configurations by using different set os synthesis directives. First, we filter out some design points through the filtering step, that is, we remove the design points with which don't have valid implementation. After that, we split into two set. i.e. train set and test set covering 35 kernels and 7 kernels respectively. In this case, the test set is all unseen kernels, which ensures the generalization and robustness of the model. The final size of training dataset is 12866, and the size of test dataset is 1269.

In the next step, we need to extract features for the training set and test set. Feature extraction includes node features and edge features. Node features include node type, block id, function id, bit width, opcode, etc. These features describe the node attributes of the data flow (CDFG). Edge features can represent the connection relationship between each node, including flow features and position features. The pragma information is embedded into input graph as done by [20]. After completing feature extraction, we one-hot compile all string features. In the compilation step, our test set and training set remain independent to prevent information leakage. The exact information about features is in table 5.1 and table 5.2.

Node Feature	**Range**	**Description**
Type	0-3	0: instruction, 1: variable, 2: constant, 3: pragma
Block ID	0-55	Unique identifier for each basic block in CDFGs
Function ID	0-7	Unique function identifier in control/data flow graphs
Bitwidth	8, 16, 32, 64	Extracted from 'text' or 'full text'
In/Out Degree	Constant	Computed from graph structure
Pragma Type	PIPE, TILE, PARA	For pragma nodes (type 3), extracted from 'text', needs encoding
Opcode	35 types	For instruction nodes (type 1), extracted from 'text', needs encoding
Opcode Category	9 types	For instruction nodes (type 1), extracted from library, needs encoding
Pragma Numeric	Constant	For pragma nodes (type 3), extracted from 'full text', e.g., `__TILE__L2=2`

Table 5.1. Node Features Description

Edge Feature	Range	Description
Flow	0-3	0: control, 1: data, 2: call, 3: pragma
Position	0-2	0: tile, 1: pipeline, 2: parallel (denotes edge ordering)

Table 5.2. Edge Features Description

After data processing, we convert the graph into a format suitable for PyTorch Geometric, pass it into the model, train and verify it on the training set, and when the model selection is completed, test and compare it on the unseen test set. The overall data preprocessing process follows the structure shown in Figure 1, which intuitively represents filtering, feature extraction, encoding, and model training.

The overall data preprocessing pipeline follows the structure illustrated in Figure 5.1, which visually represents the filtering, feature extraction, encoding, and model training steps.

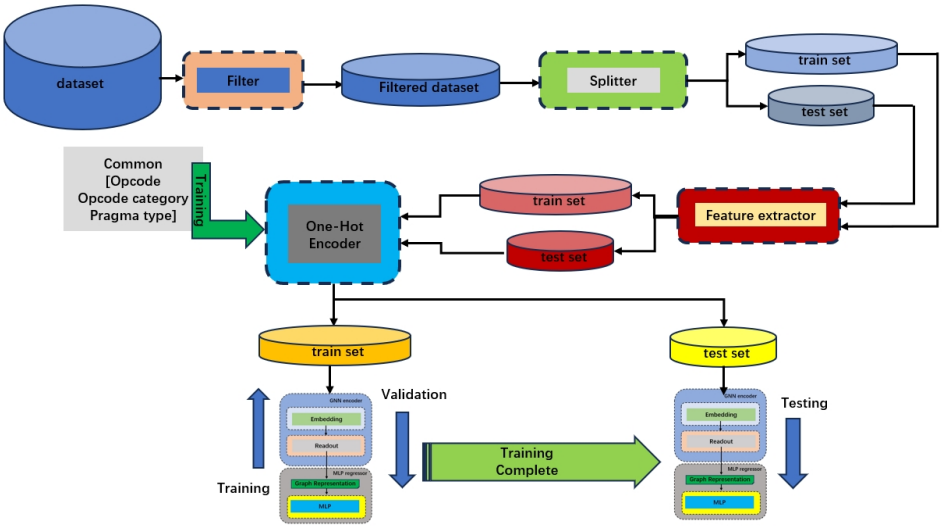

Figure 5.1. Overall data preprocessing pipeline

Chapter 6

Model Framework

6.1 Basic Model

Our basic model consists of two parts, namely the GNN encoder at the top of the model and the MLP regressor at the bottom. The GNN encoder is responsible for encoding the input graph data into a vector and reading it out as a graph representation. The MLP regressor decodes the graph representation vector and outputs the predicted value. The GNN encoder consists of two convolutional layers + activation function (ReLU) and a pooling layer, while the MLP regressor consists of two fully connected layers.Figure 6.1 shows the specific architecture of the basic model

We will innovate on the basic model, specifically replacing the convolutional layer and pooling layer of the GNN encoder. This process conforms to the logic of ablation and mainly includes the following combinations:

- **GCN + Mean pooling**
- **GATv2 + Mean pooling**
- **GATv2 + Global attention pooling**
- **GATv2 + Global attention pooling + Diffpool**

6.2 Proposed Model

After k-fold cross validation, we confirmed the superiority of the GATv2 + Global attention pooling + Diffpool model, and we use it as our proposed model. Figure 6.2 and 6.3 shows in detail the specific architecture and visual data flow of the proposed model.

The key components are:

- **GATv2: Improved Graph Attention** We have discussed the mathematical properties of GATv2 and its advantages in Chapter 4. In our case GATv2 refines the original Graph Attention Network by making the attention mechanism more flexible and permutation-invariant. It dynamically assigns different importance scores to each neighboring node, allowing the model to focus more on informative connections while reducing noise.

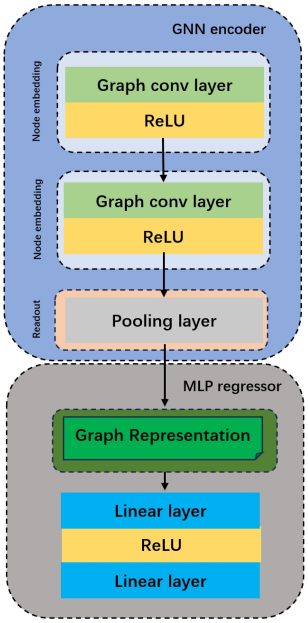

Figure 6.1. Architecture of basic model

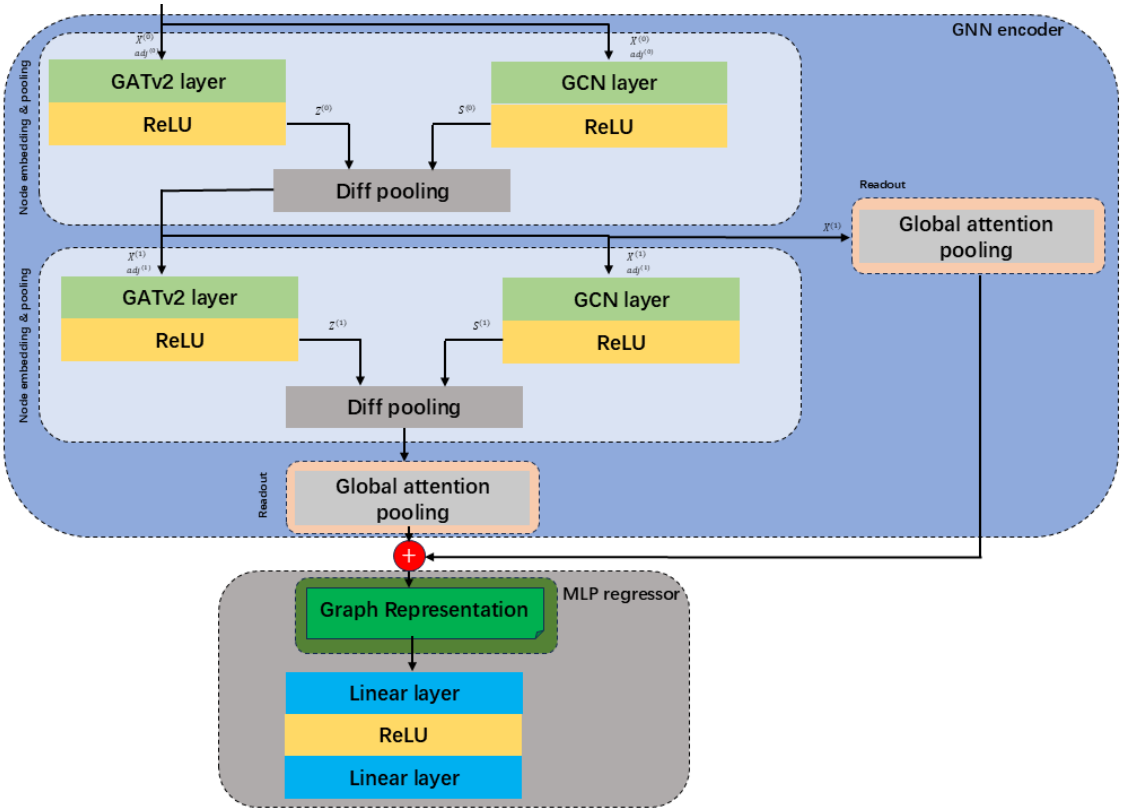

Figure 6.2. The architecture of the proposed model

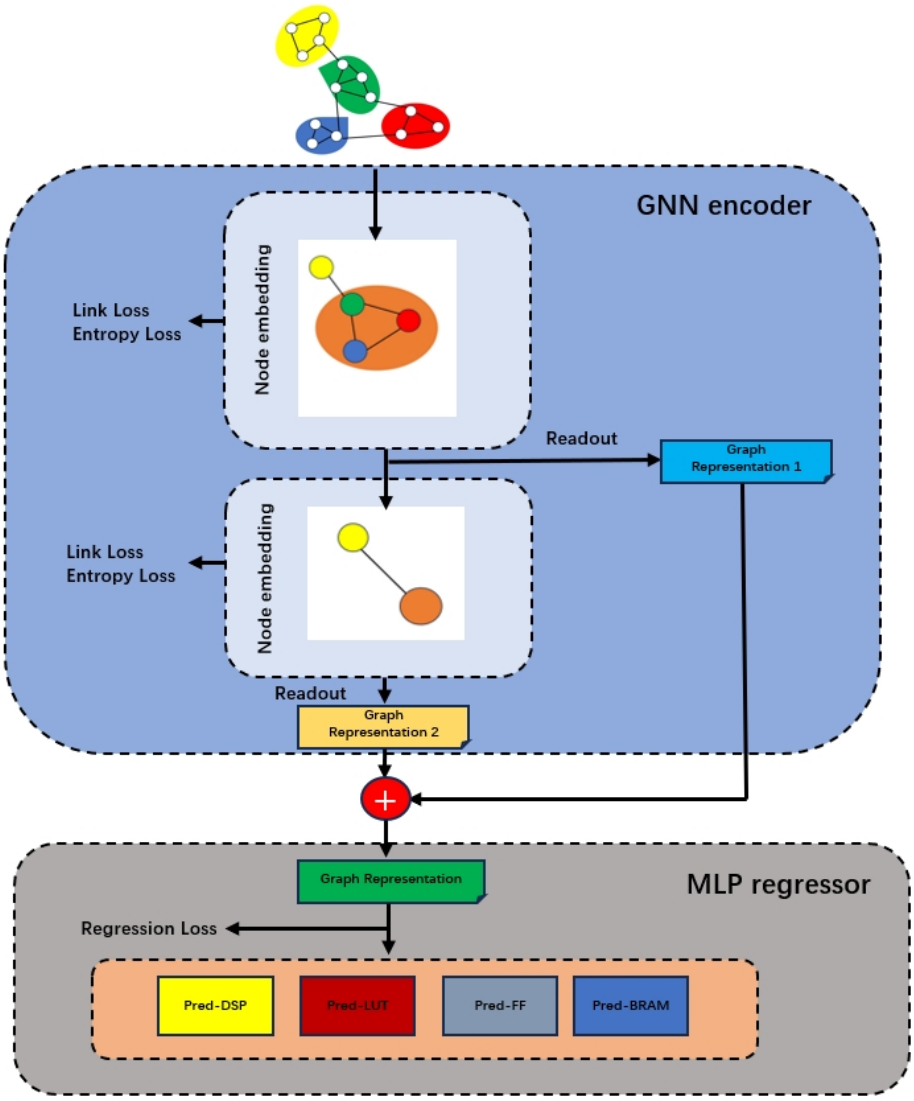

Figure 6.3. Visual data flow diagram of the proposed modell

- **Differentiable Pooling (DiffPool)** The model uses GATv2 and GCN as its core graph convolutional layers: GATv2 captures the importance of neighboring nodes through an attention mechanism, updating node features; meanwhile, GCN is used to generate the assignment matrix s assigning nodes to different clusters to simplify the graph structure. Based on this, the model applies hierarchical pooling using DenseDiffPool, which groups nodes through the learned assignment matrix s resulting in aggregated node features z and a simplified adjacency matrix adj. Each DenseDiffPool layer outputs updated z and adj and calculates link loss and entropy loss to ensure effective clustering while preserving the graph structure. Formally, clustering can be represented as:

$$z = s^T x, \quad \text{adj} = s^T A s \tag{6.1}$$

  Next, we combine the information from the two layers together, allowing the model to better handle multi-layer information.

  According to our formula, we know that in the process of DiffPool, nodes are always aggregated. When we visualized the adjacent matrix of the graph during the experiment, we found that the dimension of the matrix was constantly decreasing, and the distribution of the matrix also changed from scattered to concentrated, which shows that DiffPool is aggregating different nodes. At each layer of pooling, the graph will display different information. Figure 6.4 shows the details of this process.

Figure 6.4. Differentiable pooling nodes aggregation processl

- **Global Attention Pooling Computation:** Global Attention Pooling is a pooling method that uses the attention mechanism. In this task, we also call it a read out operation. Similar to Max Pooling and Mean Pooling, its function is to combine

and synthesize the information of each node to obtain a representation of the entire graph.

Unlike the simple read out operation, Global Attention Pooling can automatically train the different importance of each node information, thereby assigning attention weights, and finally obtaining the sum of the products of all nodes and their weights as a graph representation. In this way, the model can capture the key information that is really useful.

The method has two main steps. The graph G has N nodes, and each node has a feature vector $x_i \in R^d$. Each node is assigned an attention weight a_i:

$$a_i = \sigma\left(x_i W_1 + b\right) \tag{6.2}$$

where:

- x_i is the feature vector of node i.
- $W_1 \in R^{d\times 1}$ is a weight matrix learned by the model.
- b is a bias term.
- $\sigma(\cdot)$ is an activation function, like sigmoid or tanh, used to control the range of the weights.

Then, the final graph-level feature vector h_G is computed by combining node features and attention weights:

$$h_G = \sum_{i=1}^{N} a_i \cdot (x_i W_2) \tag{6.3}$$

where:

- $W_2 \in R^{d\times d'}$ is another weight matrix used to adjust the size of node features.
- a_i is the attention weight computed in the previous step.
- N is the total number of nodes in the graph.

- **MLP Regressor** The MLP regressor is like a decoder, which inputs the graph representation vector, passes through two fully connected layers and the corresponding nonlinear activation function layer, and outputs the QoR prediction of high-level synthesis (HLS), including resource utilization prediction of DSP, LUT, FF and BRAM, as well as latency prediction. It can capture the relationship between graph representation and target label.

Chapter 7

Experimental process and results

Our experiments are mainly divided into two parts. One is to verify and select the proposed model and the ablation model through 3-fold cross validation. The second experiment is to compare the performance of the proposed model with the baseline model. We also compare the best performing model with two baseline models.

7.1 Verification and selection of ablation models

Following are the candidate models:

- **Model 1:** GCN + Mean pooling
- **Model 2:** GATv2 + Mean pooling
- **Model 3:** GATv2 + Global attention pooling
- **Model 4:** GATv2 + Global attention pooling + Diffpool

For the four candidate models, we use 3-fold cross validation to evaluate the model. In our case, the training set will be divided into 3 parts, two of which are used for real training, and the remaining one is used for validation of each iteration. During the training process, we use RMSE as the loss function.

During training, we use the Adam optimizer with a learning rate of 0.001 for 100 epochs. For models using the attention mechanism, the number of heads is uniformly set to 4 to capture different aspects of the node representations. In addition, when we use diff-pool, there are 2 inner losses named ent-loss and link-loss, to limit the process for nodes aggregation.

We recorded the RMSE loss of each model for each fold in detail for further analysis and model selection, as the following table 7.1.And figure 7.1 shows the results more intuitively.

Based on the results, we can perform the following ablation analysis:

Impact of GATv2 Over GCN (Model 1 vs. Model 2)

- **Model 1 (GCN + Mean Pooling)** has the highest mean validation loss (0.1116), indicating that its ability to capture structural information is weaker.

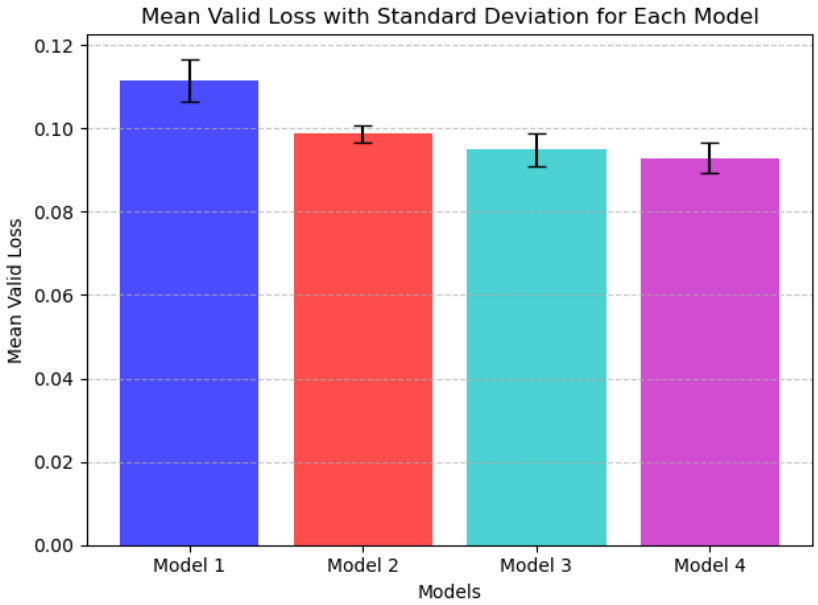

Figure 7.1. Visual data flow diagram of the proposed modell

Model	1st Fold	2nd Fold	3rd Fold	Mean ± Std
GCN + Mean Pooling (Model 1)	0.1173	0.1050	0.1125	0.1116 ± 0.0060
GATv2 + Mean Pooling (Model 2)	0.1014	0.0976	0.0970	0.0987 ± 0.0018
GATv2 + Global Attention Pooling (Model 3)	0.0960	0.0992	0.0897	0.0950 ± 0.0040
GATv2 + Global Attention Pooling + Diff-Pooling (Model 4)	0.0975	0.0928	0.0885	0.0929 ± 0.0045

Table 7.1. Validation loss comparison for different models.

- **Model 2 (GATv2 + Mean Pooling)** significantly improves performance (0.0987), showing that GATv2's attention mechanism enhances feature extraction, leading to better resource utilization prediction.

Effect of Global Attention Pooling (Model 2 vs. Model 3)

- **Model 3 (GATv2 + Global Attention Pooling)** further reduces the validation loss to 0.0950, demonstrating that global attention pooling improves the representation of the entire graph by learning node importance adaptively.
- This suggests that a learned attention-based aggregation is more effective than simple mean pooling, which treats all nodes equally.

Effect of Differentiable Pooling (Model 3 vs. Model 4)

- **Model 4 (GATv2 + Global Attention Pooling + Diff-Pooling)** achieves the best performance with the lowest mean validation loss (0.0929).
- The introduction of diff-pooling enables the model to capture hierarchical structures, leading to improved generalization.

Thus, the best model for resource utilization prediction in this study is **GATv2 + Global Attention Pooling + Diff-Pooling**, which will be used for further evaluation.

7.2 Comparative experiments with baseline

7.2.1 Resource Utilization Comparison

We first compare the performance of our model with a GNN-based QoR estimation model proposed in [9]. We call this model as Baseline 1. We mainly compare resource utilization i.e. LUTs, FFs, DSPs, and BRAMs, with this model.

Model	LUT RMSE	FF RMSE	DSP RMSE	BRAM RMSE
Baseline 1	0.164	0.141	0.203	0.205
Ours	**0.132**	**0.094**	**0.123**	**0.173**

Table 7.2. Comparison of RMSE values (DSP, LUT, FF, BRAM) between Baseline 1 and our proposed model.

Figure 7.2 shows the performance improvement more intuitively.

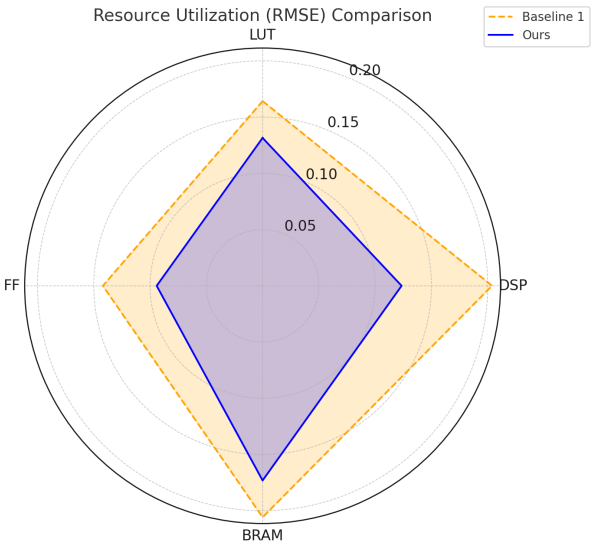

Figure 7.2. The comparison of baseline 1 model and our proposed model

For baseline 1, the model's pragma is not directly embedded in the control data flow graph (CDFG), but the pragma information is concatenated to the generated graph embeddings. In addition, the model does not use the DiffPool mechanism. It can be seen that the changes in our model have improved the overall model's performance in resource utilization QoR prediction.

7.2.2 Expanding to the overall comparison of latency

In order to make our model more accurate in predicting latency, we train another model that focuses on latency prediction with the same structure and hyperparameters in addition to the resource utilization prediction model. This model only has a single regression task.

Like resource utilization, we also need to normalize latency data as in [20]. We use the following formula latency normalization:

$$T_{\text{latency}} = \log_2 \left(\frac{\text{NormalizationFactor}}{\text{latency}} \right) \tag{7.1}$$

Finally, we restricted the range of the latency data to $(0, 12.06)$.

The work in [20] propose two GNN-based predictive models. One for estimation of LUTs, FFs, DSPs and Latency, and the other for BRAMs. This work is closest to ours as we also target exactly same predictive objectives in this work. We take the predictive models in [20] and train them to the best of our ability according to how they are proposed in the work. We call these models as Baseline 2 for quantitative comparison.

Model	**LUT RMSE**	**FF RMSE**	**DSP RMSE**	**BRAM RMSE**	**Latency RMSE**
Baseline 2	0.189	0.126	0.172	0.209	0.857
Ours	**0.132**	**0.094**	**0.123**	**0.173**	**0.8255**

Table 7.3. Comparison of resource utilization (DSP, LUT, FF, BRAM) and latency (RMSE) between Baseline 2 and our proposed model.

Figure 7.3 shows a radar chart comparing the resource utilization prediction RMSE of Baseline 2 and our proposed model. Figure 7.4 shows a bar chart comparing the latency prediction RMSE.

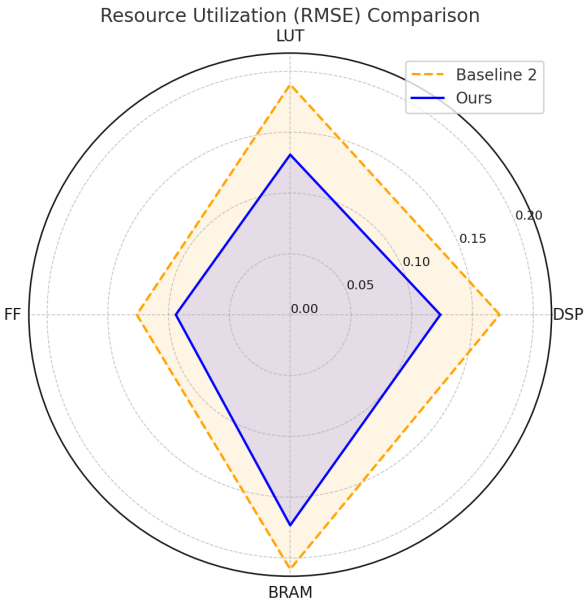

Figure 7.3. The comparison of resource utilization for Baseline 2 model and our proposed model

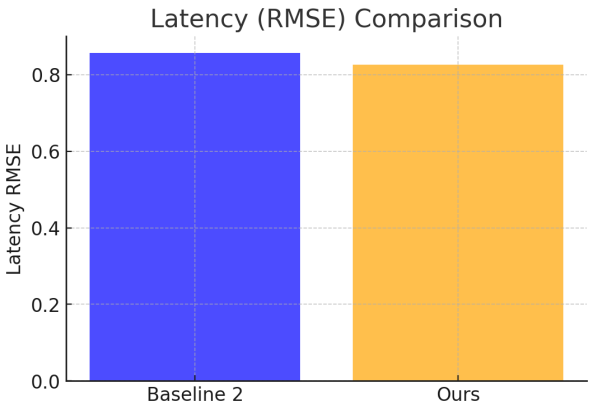

Figure 7.4. The comparison of latency for Baseline 2 model and our proposed model

These qualitative comparison clearly shows that our model outperforms both baselines in predicting resource utilization, and also latency. The inclusion of GATv2, differentiable pooling, and global attention pooling significantly improves the accuracy. These findings suggest that more advanced GNN architectures are critical for accurate QoR estimation in high-level synthesis, and the idea of hierarchical pooling holds promise for this task.

Chapter 8

Conclusion

High-level synthesis (HLS) helps designers create specialized hardware quickly. Designers can use C/C++ instead of hardware description language (HDL). HLS directives (pragma) let them balance performance and resource usage. However, when more optimizations are added, the number of design configurations increases rapidly. Checking each design with an HLS tool takes a lot of time and makes design space exploration slow. To speed up, machine learning models such as graph neural networks (GNNs) predict the quality of results (QoR) before running full synthesis. However, due to information loss in graph convolution and pooling, existing methods still have prediction errors with respect to the final implementations.

We propose a new GNN-based framework with differentiable pooling (DiffPool). We capture multi-level structures to reduce information loss and improve QoR prediction. Experiments show that our model reduces the prediction error of FPGA resources and latency compared to other graph neural network learning-based methods.

Bibliography

[1] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2020.

[2] AMD Xilinx Inc. Vitis high-level synthesis user guide. https://www.xilinx.com/support/documents/sw_manuals/xilinx2022_2/ug1399-vitis-hls.pdf, 2022. Accessed: 2024-03-16.

[3] Applied AI Course. Cross validation in machine learning. https://www.appliedaicourse.com/blog/cross-validation-machine-learning/, n.d. Accessed: 2025-03-29.

[4] Shaked Brody, Uri Alon, and Eran Yahav. How attentive are graph attention networks?, 2022.

[5] Steve Dai, Yuan Zhou, Hang Zhang, Ecenur Ustun, Evangeline F.Y. Young, and Zhiru Zhang. Fast and accurate estimation of quality of results in high-level synthesis with machine learning. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 129–132, 2018.

[6] Chenhui Deng, Zichao Yue, Cunxi Yu, Gokce Sarar, Ryan Carey, Rajeev Jain, and Zhiru Zhang. Less is more: Hop-wise graph attention for scalable and generalizable learning on circuits, 2024.

[7] Matthias Fey and Jan Eric Lenssen. Fast graph representation learning with pytorch geometric. *arXiv preprint arXiv:1903.02428*, 2019.

[8] Jonathan Hui. Graph convolutional networks (gcn) - pooling, 2023. Accessed: 2025-03-14.

[9] M. Usman Jamal, Zhuowei Li, Mihai T. Lazarescu, and Luciano Lavagno. A graph neural network model for fast and accurate quality of result estimation for high-level synthesis. *IEEE Access*, 11:85785–85798, 2023.

[10] Kaggle. Machine learning contest for high-level synthesis. https://www.kaggle.com/competitions/machine-learning-contest-for-high-level-synthesis/overview, 2025. access: 2025-03-15.

[11] Dirk Koch, Frank Hannig, and Daniel Ziener. *FPGAs for software programmers*. Springer, 2016.

[12] Sakari Lahti, Panu Sjovall, Jarno Vanne, and Timo Hämäläinen. Are we there yet? a study on the state of high-level synthesis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, PP:1–1, 05 2018.

[13] Yaguang Li, Yishuang Lin, Meghna Madhusudan, Arvind Sharma, Wenbin Xu, Sachin S Sapatnekar, Ramesh Harjani, and Jiang Hu. A customized graph neural network model for guiding analog ic placement. In *Proceedings of the 39th International Conference on Computer-Aided Design*, pages 1–9, 2020.

[14] Daniela Sánchez Lopera, Lorenzo Servadei, Gamze Naz Kiprit, Souvik Hazra, Robert Wille, and Wolfgang Ecker. A survey of graph neural networks for electronic design automation. In *2021 ACM/IEEE 3rd Workshop on Machine Learning for CAD (MLCAD)*, pages 1–6, 2021.

[15] Hosein Mohammadi Makrani, Farnoud Farahmand, Hossein Sayadi, Sara Bondi, Sai Manoj Pudukotai Dinakarrao, Liang Zhao, Avesta Sasan, Houman Homayoun, and Setareh Rafatirad. Pyramid: Machine learning framework to estimate the optimal timing and resource usage of a high-level synthesis design, 2019.

[16] Razvan Nane, Vlad-Mihai Sima, Christian Pilato, Jongsok Choi, Blair Fort, Andrew Canis, Yu Ting Chen, Hsuan Hsiao, Stephen Brown, Fabrizio Ferrandi, Jason Anderson, and Koen Bertels. A survey and evaluation of fpga high-level synthesis tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604, 2016.

[17] NVIDIA. What is pytorch? https://www.nvidia.com/en-us/glossary/pytorch/. Accessed: 2024-03-16.

[18] Stéphane Pouget, Louis-Noël Pouchet, and Jason Cong. Automatic hardware pragma insertion in high-level synthesis: A non-linear programming approach. *ACM Transactions on Design Automation of Electronic Systems*, 30(2):1–44, February 2025.

[19] Nathanael Ren. Solving ogb function name prediction using data augmentation and diffpool hierarchical pooling, 2024. Accessed: 15-Mar-2025.

[20] Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong. Automated accelerator optimization aided by graph neural networks. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, DAC '22, page 55–60, New York, NY, USA, 2022. Association for Computing Machinery.

[21] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks, 2018.

[22] Yu Wang, Liang Hu, Yang Wu, and Wanfu Gao. Graph multihead attention pooling with self-supervised learning. *Entropy*, 24(12):1745, 2022.

[23] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, 2021.

[24] Zhiyao Xie, Rongjian Liang, Xiaoqing Xu, Jiang Hu, Yixiao Duan, and Yiran Chen. Net2: A graph attention network method customized for pre-placement net length estimation. ASPDAC '21, page 671–677. Association for Computing Machinery, 2021.

[25] Rex Ying, Jiaxuan You, Christopher Morris, Xiang Ren, William L. Hamilton, and Jure Leskovec. Hierarchical graph representation learning with differentiable pooling, 2019.

[26] Zhiru Zhang, Y Fan, W Jiang, G Han, C Yang, and J Cong. High-level synthesis: From algorithm to digital circuit. In *ch. AutoPilot: A Platform-Based ESL Synthesis System*, pages 99–112. Springer Netherlands, 2008.