Optimizing Training Cost for Scalable Graph Processing

Han Hoang gihoang@ucsd.edu
Joshua Li jol029@ucsd.edu
Lindsey Kostas lkostas@qti.qualcomm.com
Dhiman Sengupta dhimseng@qti.qualcomm.com

Abstract

The DE-HNN model is a state-of-the-art graph neural network designed to predict congestion using circuit netlist representations, significantly outperforming other hypergraph models in demand regression tasks. However, its high computational demands limit scalability and efficiency, posing a significant barrier to practical deployment. To address this, our research focuses on optimizing the training cost of DE-HNN, aiming to balance computational demands and model performance through systematic tuning and analysis of trade-offs. The optimized configuration consists of 4 layers and 8 dimensions, combined with Early Stopping and Cyclical Learning Rate scheduling. This configuration achieved substantial reductions in runtime (89.32%) and peak memory usage (38.83%), with only a minor average performance drop of less than 6%. These results highlight the effectiveness of combining architecture simplifications, dynamic learning rate scheduling, and Early Stopping to achieve an optimal balance between computational efficiency and predictive performance, ultimately enabling faster and more resource-efficient design feedback for chip designers.

1 Introduction

In chip design, traditional place-and-route (PnR) methods are beset by inefficiencies, as their iterative refinement of layouts can be both time-consuming and labor-intensive. By contrast, data-driven chip design optimization offers a more efficient alternative, leveraging machine learning to predict resource bottlenecks early on and inform design decisions that minimize costly design iterations. Central to this approach is the netlist, a hypergraph representation of a circuit's connectivity, where nodes represent components (e.g., logic gates) and hyperedges represent electrical connections. By analyzing and optimizing resource demand through the netlist, this method directly enhances floorplanning—the process of arranging components on a 2D chip canvas—by providing insights that minimize congestion while optimizing Power, Performance, and Area (PPA) and meeting design constraints.

DE-HNN (Luo et al. (2024)) is a state-of-the-art hypergraph neural network designed to predict congestion in chip design via demand regression. It outperforms other models by effectively capturing long-range dependencies through hierarchical virtual nodes, which aggregate node features within partitioned graph neighborhoods and propagate information efficiently, enabling more robust predictions.

While DE-HNN delivers high performance, its scalability and practicality are limited by its significant computational cost. This motivated our research to focus on exploring strategies that **optimize the training cost** of DE-HNN, in terms of *memory usage* or *training runtime* while **preserving prediction quality**.

2 Methodology

2.1 Environment Setup

All experiments were conducted on a UCSD DSMLP cloud system with NVIDIA RTX A5000 GPU (24 GB VRAM), using PyTorch 2.2.2 and CUDA 12.2.

2.2 Dataset Description

The training dataset includes netlists [1-5], while netlist 6 is used for validation and testing. The netlists vary in the number of nodes, nets (hyperedges), and edges (connections between nodes and nets). Each netlist provides structural, spectral, and representation characteristics as input features, with node and net demand as target variables.

Table 1: Feature Table

Feature Category	Feature Name	Description
Structural Features	Node Degree	The number of edges connected to a node.
	Net Degree	The number of hyperedges connected to a net.
	Edge Connection	If there is an edge connecting a node to a net.
	Sink/Source Node	If the node is a sink or source in relation to its net.
	Number of Virtual Nodes	Total count of virtual nodes present in the graph.
Spectral Features	Top 10 Eigenvalues (Node)	The largest eigenvalues of the graph's adjacency matrix for nodes.
	Top 10 Eigenvectors (Node)	Corresponding eigenvectors for nodes.
	Top 10 Eigenvalues (Net)	The largest eigenvalues of the graph's adjacency matrix for nets.
	Top 10 Eigenvectors (Net)	Corresponding eigenvectors for nets.
Representation Features	Nodes Representations	Learned embeddings of individual nodes and their neighbors.
	Nets Representations	Learned embeddings of nets.
	Virtual Nodes Representations	Learned embeddings of virtual (aggregated) nodes.
Target Variables	Node Demand	The demand associated with each node.
	Net Demand	The demand associated with each net.

Table 2: Dataset Characteristics

Netlist	# Nodes	# Nets	# Edges
1	797,938	821,523	2,950,019
2	923,355	954,144	3,459,373
3	604,921	627,036	2,358,662
4	671,284	696,983	2,532,180
5	459,495	468,888	1,942,114
6	810,812	830,308	3,107,242

2.3 Training and Evaluation Strategy

Minimize mean squared error (MSE) in the regression training set. Performance is monitored on the validation set during training, and final evaluation is performed on the test set to assess generalization to unseen data.

2.4 Optimization Strategy

We applied **iterative optimization** by first using early stopping, followed by architecture adjustments based on Grid Search, and finally incorporating a dynamic learning rate, while using a fixed random seed throughout for weight initialization to control the effects of each optimization component.

2.4.1 Early Stopping (ES)

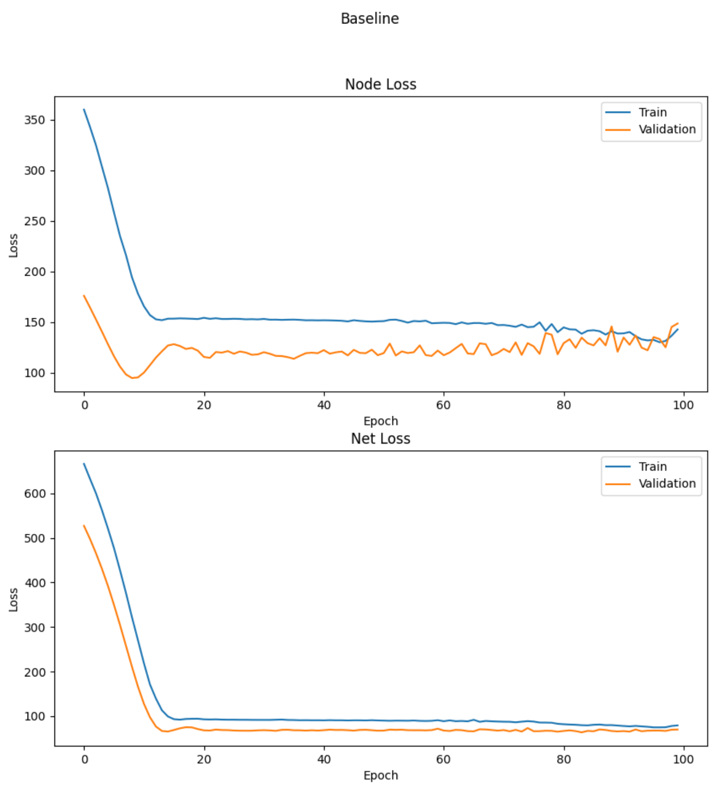


Figure 1: Loss Curves: Baseline Model

- **Description**: A custom condition using 3 parameters: *patience*, *tolerance*, and *min-epochs*. Early Stopping is triggered when validation loss exceeds a *tolerance*-defined range around the average loss of the last *patience* epochs, but only after at least *min-epochs* have passed, preventing overfitting while ensuring a minimum number of training epochs.
- **Justification**: The decision to use early stopping was informed by the analysis of the baseline model's loss curve (Figure 1). The baseline model's loss curve showed a pattern of overfitting: while training loss decreased, validation loss plateaued and then began to increase after several epochs. This divergence, a common indicator of overfitting, was accompanied by oscillations in the validation loss. The oscillations observed in the validation loss further highlight the model's sensitivity to the higher complexity or variance in the validation set. Early stopping addresses this by halting training when the validation loss stops improving, ensuring the model does not overfit to the training data or become overly sensitive to noise in the validation set.
- **Implementation Details**: Validation loss was monitored during training. The parameters were set as follows: *patience* = 5 epochs, *tolerance* = 0.1, and *min-epochs* = 10. These values were chosen based on observations that the model typically converged by 15 epochs, with validation loss fluctuations stabilizing within 5 epochs.

2.4.2 Architecture Adjustments (AA)

- **Description**: We use Grid Search in the next optimization stage to systematically explore combinations of two hyperparameters—number of layers and dimensions—and identify cost-accuracy trade-offs.
- **Justification**: Grid Search is computationally expensive, so it was implemented after Early Stopping to reduce training costs. The purpose of this stage is to validate our hypothesis that reducing the number of layers and dimensions lowers memory usage and runtime while identifying the architecture that maximized validation accuracy.
- **Implementation Details**: The parameter grid consists of [2, 3, 4] for the number of layers and [8, 16, 32] for the number of dimensions. Early Stopping is applied to all Grid Search experiments.

2.4.3 Dynamic Learning Rate (DLR)

- **Description**: We used Cyclical Learning Rate (CLR) scheduling (Smith (2015)) as our dynamic learning rate strategy. CLR oscillates the learning rate between a predefined minimum and maximum in a cyclical pattern, rising from the minimum to the maximum and then decreasing back to the minimum within each cycle.
- **Justification**: Adding a dynamic learning rate in the last optimization stage ensures efficient use of computational resources and avoids premature optimization. By first addressing architectural inefficiencies and overfitting, the dynamic learning rate can focus on improving convergence without being hindered by suboptimal configurations. Cyclical Learning Rate scheduling offers several advantages over other dynamic learning rate methods, such as step decay or exponential decay. By oscillating the learning rate instead of monotonically decreasing it, CLR prevents the model from getting stuck in suboptimal local minima and improves convergence speed. Additionally, CLR eliminates the need for manual tuning of decay schedules or complex heuristics, making it particularly effective for navigating the complex loss surfaces often encountered in deep neural network training.
- **Implementation Details**: For this study, the Cyclical Learning Rate scheduler was implemented with the following parameters:
  - Minimum Learning Rate: 0.001
  - Maximum Learning Rate: 0.01
  - Step Size: 5 iterations (half a cycle)
  - Mode: Triangular (linear increase and decrease within each cycle)

  These parameters were chosen to ensure faster convergence without risking divergence or unnecessary training iterations. Notably, the model trained with a maximum learning rate of 0.01 (e.g., Early Stop with 4 layers and 8 dimensions) showed no signs of loss divergence, validating the stability of this configuration.

2.5 Baseline Model

We used the unmodified full DE-HNN model with virtual nodes as our baseline. Due to the resource constraint of our project (see 2.1), the number of layers in our baseline model is 3 and the number of dimensions is 32 instead of 64 as stated in the original paper (Luo et al. (2024)). The model is trained for 100 epochs, and achieved a **133 MSE Node loss**, **67.5 MSE Net loss**, **5.05 minutes runtime**, and **22134 MBs peak memory** on the test set.

3 Results

3.1 Iterative Optimization

3.1.1 Early Stopping (ES)

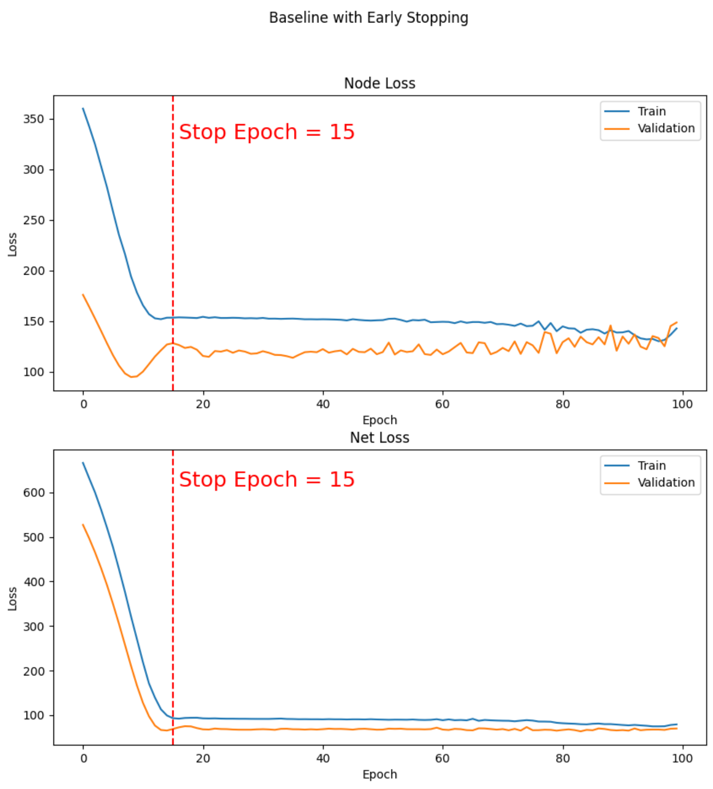
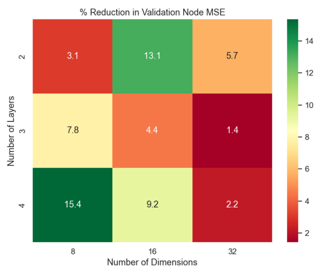


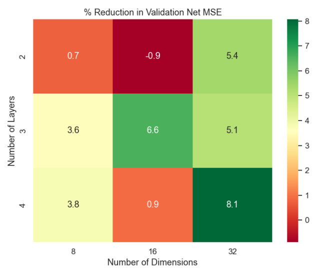
Figure 2: Loss Curves: Baseline Model with Early Stopping

With Early Stopping, the baseline model achieved a **126.5 MSE Node Loss**, **65.33 MSE Net loss**, **0.79 minutes runtime**, and **22134 MBs peak memory** on the test set. This represents an 4.9% reduction in Node loss, 3.17% reduction in Net loss, and a 84.37% decrease in runtime, with no change in peak memory usage. The training **stopped at epoch 15**, compared to the original 100 epochs, which explains the significant reduction in training runtime. These results confirm that the model was overfitting and benefited from fewer training iterations, improving both efficiency and performance.

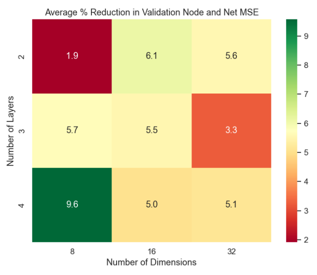
3.1.2 Architecture Adjustments (AA)



(a) Training Time (Minutes)

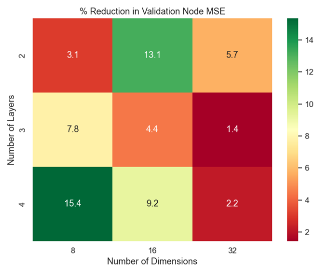


(b) Peak Memory Usage (MBs)

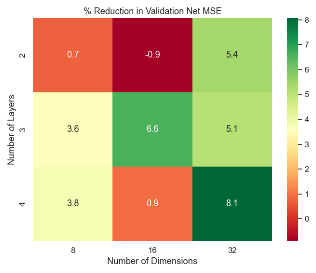


(c) Average Reduction

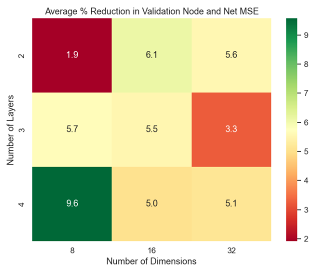
Figure 3: Reduction in Training Cost (in %)



(a) Validation Node MSE

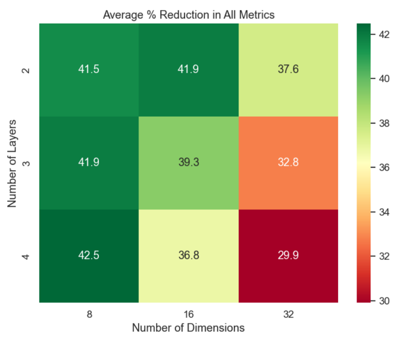


(b) Validation Net MSE

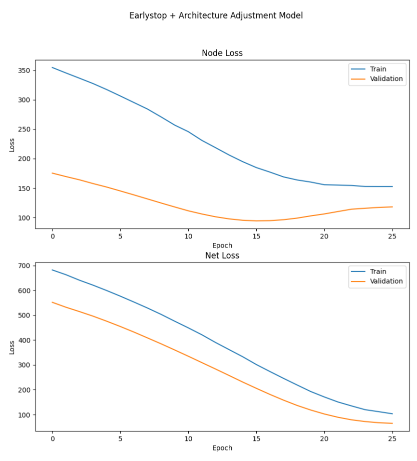


(c) Average Reduction

Figure 4: Reduction in Model Performance (in %)



(a) Average Reduction in All Metrics (in %)



(b) Loss Curves: Grid Search Optimal Model

Figure 5: Optimizing via Grid Search

We conducted Grid Search experiments with Early Stopping and identified the optimal architecture as having **4 layers** and **8 dimensions**. While some architectures, like 4 layers with 32 dimensions, performed better in peak memory usage and net validation error, the 4-layer, 8-dimension model provided the best overall improvement in both validation error and training efficiency as shown in Figure 5a.

The Grid Search optimal model achieved a **118 MSE Node Loss**, **65 MSE Net loss**, **1.15 minutes runtime**, and **13539 MBs peak memory** on the test set. This corresponds to an 11.27% reduction in Node loss, 3.65% reduction in Net loss, 77.19% decrease in runtime, and 38.83% reduction in peak memory usage. Training **stopped at epoch 25**, adding 10 more epochs compared to the Baseline Early Stop model, which slightly decreased the runtime improvement but resulted in an overall more effective model.

3.1.3 Dynamic Learning Rate (DLR)

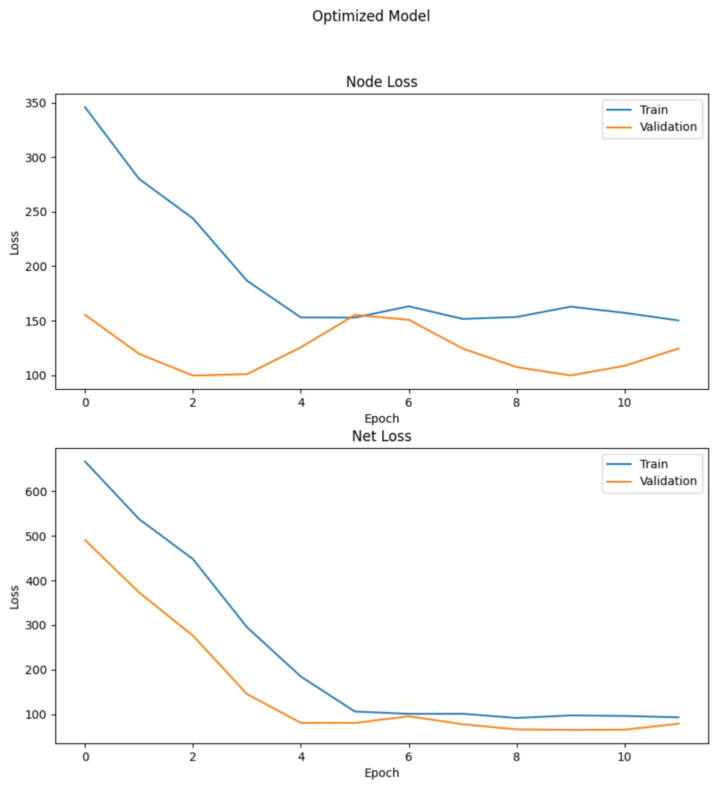


Figure 6: Loss Curves: Optimized Model

The *optimized model* resulting from this study is the same model produced during the final optimization stage using DLR scheduling (see Figure 6).

DLR with Cyclical Learning Rate (CLR) scheduling reduced the number of training epochs by 89%, with less than a 6% average performance drop. Training **stopped at epoch 11**, compared to epoch 25 for the Grid Search optimal model and epoch 100 for the baseline model. By combining CLR with early stopping in an efficient model architecture, convergence speed was significantly accelerated, enabling the model to stabilize in far fewer epochs while avoiding overfitting. The CLR model achieved a **124.48 MSE Node loss**, **79.34 MSE Net loss**, **0.54 minutes runtime**, and **13539 MB peak mem-**

ory usage. This corresponds to a 6.4% reduction in validation Node Loss but also a 17.6% increase in Net loss compared to the baseline model. This discrepancy between the changes in validation Node loss and Net loss can be attributed to the distinct loss surfaces of the two objectives, where CLR's oscillatory updates may better navigate the Node loss surface but struggle to find an optimal minimum on the potentially more complex Net loss surface. Despite the trade-off in Net loss, CLR significantly improved training efficiency by drastically reducing iterations and maintaining the benefit of reduced peak memory usage.

3.2 Overall Improvements

Table 3: Reduction in Training Cost and Performance from Training Adjustments

Adjustments	**Node Loss (MSE)**	**Net Loss (MSE)**	**Runtime**	**Memory**
ES[a]	4.9%	3.17%	84.37%	0%
ES[a]+ AA[b]	11.27%	3.65%	77.19%	38.83%
ES[a]+ AA[b]+ DLR[c]	6.4%	-17.6%	89.32%	38.83%

[*] Comparison is based against the baseline metrics
[a] Early Stopping
[b] Architecture Adjustment (Grid Search: 4 layers, 8 dimensions)
[c] Dynamic Learning Rate (Cyclical Learning Rate)

The optimization process in this study resulted in significant performance gains across multiple stages, as summarized in Table 3.

The baseline model served as the starting point, with subsequent optimization stages build upon one another to achieve substantial improvements in the model's runtime and memory usage. These improvements were achieved while keeping the performance metrics relatively close to the baseline, demonstrating the effectiveness of the systematic optimization strategy.

Notably, the final optimized model achieved a balance between computational efficiency and predictive performance, with the largest gains observed in training efficiency. Despite a relatively small performance drop (i.e., 6% on average), the final model demonstrated significant improvements in resource utilization, reducing runtime by 89.32% and memory usage by 38.83%.

4 Conclusion

Our results demonstrate that DE-HNN is a highly expressive model with a fast convergence rate. Through iterative optimization, we achieved maximum performance with minimal training using a simplified configuration **(4 layers, 8 dimensions)**, coupled with the scheduling of the Cyclical Learning Rate and our custom Early Stopping condition. Compared to the baseline model, the optimized DE-HNN achieved significant improvements in both **runtime (89.32% reduction)** and **memory usage (38.83% reduction)**, with a less than **6% average performance drop**. These results highlight the trade-off between computational efficiency and predictive performance, which is often necessary in real-world applications where resource constraints are critical.

The key findings of this study are as follows:

- DE-HNN performs better on the test set with fewer training iterations, demonstrating that the model can achieve high performance with minimal computational cost.
- Simpler configurations (e.g., 4 layers, 8 dimensions) outperform more complex ones (e.g., 4 layers, 32 dimensions) both in terms of training cost and model performance.
- While simpler models may require slightly more iterations to converge, the additional training time is negligible compared to the substantial gains in runtime and memory efficiency.
- Cyclical Learning Rate scheduling accelerates convergence, enabling the Early Stopping condition to terminate training earlier and reduce the number of training iterations.

Overall, the optimization process demonstrates the effectiveness of combining Early Stopping, Architecture Adjustments, and Dynamic Learning Rate scheduling in a systematic and cost-effective way to find the optimal balance between computational efficiency and predictive performance.

5 Discussion

To improve the robustness of our findings, future work could involve applying the optimization techniques across multiple random seeds and averaging the results. This approach would help mitigate the variability introduced by stochastic processes, providing a more reliable evaluation of the optimization strategies. By addressing the inherent randomness in weight initialization and training dynamics, this step would ensure that the observed improvements are consistent and not artifacts of specific experimental conditions.

Additionally, the trade-off between a slight performance drop (e.g., less than 6%) and substantial computational gains highlights the need for careful consideration of application-specific requirements when deploying the optimized model.

Our work could also be extended to other architectures or datasets to validate its scalability and generalizability. For instance, applying our optimization techniques to specialized deep learning architectures in fields like computer vision or to datasets with unique characteristics could offer deeper insights into the practical applicability of our study. Additionally, investigating the performance of our optimization strategy in transfer learning scenarios or with pre-trained models could uncover promising new research directions.

References

Luo, Zhishang, Truong Son Hy, Puoya Tabaghi, Donghyeon Koh, Michael Defferrard, Elahe Rezaei, Ryan Carey, Rhett Davis, Rajeev Jain, and Yusu Wang. 2024. “DE-HNN: An effective neural model for Circuit Netlist representation.” *arXiv preprint arXiv:2404.00477*

Smith, Leslie N. 2015. “No More Pesky Learning Rate Guessing Games.” *CoRR* abs/1506.01186. [Link]

Appendices

A.1 Project Proposal

A.1.1 Problem Statement

Graph Neural Network (GNN) are computationally expensive, especially for large-scale graphs. In congestion prediction, GNNs are trained on circuit netlist representations containing billions of components, making training time a significant bottleneck and limits scalability and usefulness of the model in the chip design process. As chip designs become more complex, the need to optimize the runtime of GNN model training becomes increasingly important. Optimized model training runtime will result in a faster chip design cycle in which quicker adjustments can be made, ultimately accelerating the overall development and optimization of the chip.

A.1.2 Background Information

The DE-HNN model is a type of GNN that provides a good framework for optimizing chip design by capturing long-range interactions in dense graphs. However, De-HNN is computationally expensive due to the dual update of nodes and nets features and the addition of virtual nodes. The high number of message passing operations causes a significant bottleneck that limits the scalability of DE-HNN to larger and more complex graphs.

Previous work on the DE-HNN model has primarily emphasized enhancing predictive performance, often overlooking the critical issue of computational efficiency. However, in real-world applications, the value of a more accurate model must be weighed against the practical constraints of computational cost.

The goal is to improve runtime efficiency without compromising the predictive power of the model. Our quarter 2 project aims to achieve this goal by investigating specific modifications to the DE-HNN model. Key areas of exploration include:

- **Propagation Interval Optimization:** Change the frequency of propagation steps while maintaining the model's ability to capture long-range dependencies. Our intuition is that by decreasing the propagation interval, we lower our number of message passing operations and cut down computational training cost.
- **Partitioning Optimization:** Change the number of partitioned neighborhoods while preserving the expressiveness of the model. Our intuition is that by decreasing the number of graph partitions, we decrease the number of constructed

virtual nodes and lower our number of message passing operations, resulting in lower computational training cost.

This investigation directly addresses a deficiency identified during the Q1 Project by focusing on computational efficiency rather than predictive performance. By building on the insights gained from the DE-HNN re-implementation and leveraging existing methodologies from GNN optimization, the objectives of our quarter 2 projects are:

- Enhance the scalability of DE-HNN for large, dense hypergraphs.
- Provide a framework for balancing accuracy and efficiency in hypergraph-based models.
- Establish a roadmap for adapting DE-HNN to real-world chip design applications, where computational constraints are a critical factor.

The proposed work not only fills a gap in prior research but also extends the applicability of DE-HNN, making it a more practical tool for congestion modeling in chip design. These advancements have the potential to impact other domains requiring scalable hypergraph neural networks.

A.1.3 Deliverables

The primary output will be a comprehensive report detailing the methods, experiments, and results of modifying the DE-HNN model. The report will include the following.

- **Problem Context:** An explanation of computational bottlenecks in the model.
- **Methodology:** Descriptions of the modifications and tests we performed on the DE-HNN model, including rationale and implementation specifics for each optimization we tried.
- **Benchmarking and Analysis:** An evaluation of the impact of these modifications on the model runtime, using average runtime as the primary performance metric. This will include comparison with the baseline DE-HNN implementation.

The report will also feature visualizations including graphs comparing the runtime performance of the baseline DE-HNN and the modified versions we will be testing, accuracy metrics to ensure that runtime optimizations do not significantly degrade the baseline model's performance, and graph statistics to analyze how structural properties may influence the effectiveness of the modifications.

A.2 De-HNN Model Architecture

A.2.1 Update Functions

One key difference between the De-HNN model architecture and other types of message-passing GNNs lie in the way it aggregates and updates cells and nets features.

Let a directed hypergraph $H = (V, \sum)$ where V is a set of nodes and $\sum$ is a set of hyperedges, the update function for a cell is given by:

$$m^{\ell}(v) = \sum_{\sigma' \in \mathscr{L}(v)} \mathrm{MLP}_1^{\ell}(M^{\ell-1}(\sigma'))$$

Where:

- $m^{\ell}(v)$ is the node feature of some ℓ GNN layer.

- MLP is a Multilayer perceptron.

- σ is a net in the set of nets $\sum$.

- $M^{\ell-1}$ is the net feature of the previous GNN layer.

- $\mathscr{L}(v)$ is the set of nets that node v is in.

This function denotes that a node feature gets updated by the aggregated sum of the MLP-transformed net features of the nets the node belongs to.

And the update function for a net is given by:

$$M^{\ell}(\sigma) = \mathrm{MLP}_3^{\ell}[m^{\ell}(v_{\sigma}) \bigoplus (\sum_{v' \in S_{\sigma}} \mathrm{MLP}_2^{\ell}(m^{\ell}(v')))]$$

Where:

- $M^{\ell}(\sigma)$ is the net feature of some ℓ GNN layer.

- MLP is a Multilayer perceptron.

- $m^{\ell}(v_{\sigma})$ is the driver cell of the net—cell that establishes incoming connections from other nets.

- $m^{\ell}(v_{\sigma})$ is the sink cell of the net—cell(s) other than the driver node from the same net.

- $\bigoplus$ is the vector concatenate operation.

This function denotes that a net feature gets updated by the MLP-transformed concatenate vector representation of the driver node features and the aggregated sum of all MLP-transformed features of the sink nodes.

Explanation. The cell features are first transformed by an MLP in a GNN layer, and then used to update the net features in the graph. De-HNN performs a dual update for both cells and nets, with information passing through the GNN layers sequentially from cells to nets. This dual update mechanism is key to De-HNN's ability to capture long-range interactions effectively, as it dynamically adjusts the directed hypergraph properties for contextual relevance, enabling the model to learn distant node features with minimal information dilution.

A.2.2 Aggregated Nodes (Virtual Nodes)

A key difference between the De-HNN model and other message-passing GNNs is its use of aggregated or "virtual nodes". In De-HNN, the graph is first partitioned into k number of neighborhoods using the Metis partitioning method. For each neighborhood, a virtual node is created by aggregating the features of all the nodes within that neighborhood. Information is then propagated through these virtual nodes, rather than directly between all individual nodes in the graph.