Final Project

CS533 ML for EDA

By
Ashwin Iyengar (244101007)
Devansh Upadhyay (244101015)
Saurav Chaudhary (244101048)
Vaibhav Wankar (244101061)

21 April 2025

Github Repo: QoR-predictor

Table of Contents

1. Assignment Overview 3

    1.1 Objective 3

    1.2 Our take 3

2. Implementation of the QoR predictor proposed by Animesh et al 4

    2.1 Data generation 4

        2.1.1 Problems faced 6

        2.1.2 Insights and Observations 6

        2.1.3 Insides of a PyTorch data object 7

    2.2 Training the Model 7

        2.2.1 Metrics used 8

        2.2.2 Insights and Observations 9

3. ABColabX 11

    3.1 Model Overview 11

    3.2 Deep Dive into the Model 12

        3.2.1 Input & Directory Structure 12

        3.2.2 Dataset Creation and Preprocessing 12

        3.2.3 Neural Network choice 13

    3.3 Training the Model 17

    3.4 Finetuning the Model 18

1. Assignment Overview

1.1 Objective

This assignment aims to develop a Quality of Results (QoR) predictor, specifically focused on estimating **delay**, for digital designs undergoing logic synthesis.
The predictor aims to estimate the delay outcome of a given design when synthesised with a specific synthesis recipe, before running an actual synthesis using the ABC tool.
A key aspect of this project is the ability to generalise to unseen designs. For such cases, the model should support fine-tuning by retraining on a small set of selected synthesis recipes associated with the new design. This step is critical to adapt the predictor and improve its estimation accuracy.
Finally, it should demonstrate the prediction accuracy by comparing the estimated delay values from the predictor against the actual delay results obtained from ABC synthesis runs. Proper evaluation metrics should be included to support the analysis.

1.2 Our take

We began by evaluating and implementing one of the state-of-the-art QoR predictors proposed by Animesh Basak Chowdhury et al. This gave us valuable insights into how such models work in practice and served as a solid foundation for our designs.
Building on these insights, we developed our custom model. Our model, namely ABColabx (fancy, isn't it?), takes an end-to-end approach, where all components are trained jointly, allowing them to learn and adapt in coordination.
Let's dive deeper into these models and explore the steps involved in their design.

2. Implementation of the QoR predictor proposed by Animesh et al

2.1 Data generation

OpenABC-D is a large-scale labelled dataset generated by synthesising open-source hardware IPs using a state-of-the-art logic synthesis tool, yosys-abc. They considered 29 open-source hardware IP designs collected from various sources (MIT-CEP, IWLS, OpenROAD, OpenPiton etc) and synthesised them with 1500 random synthesis flows (that they call *synthesis recipes*).

Each synthesis flow has a predefined length **L**. They preserved all AIGs: starting, intermediate and final AIGs with labels like number of nodes, longest path, sequence of atomic synthesis transformations (*rewrite*, *refactor*, *balance*, etc.), along with graph statistics, area and delay of final AIG.

Each synthesis run generates a .pt file containing an updated And-Inverter Graph (AIG), which forms the training data. These AIGs in PyTorch data format can be directly used in machine learning as labelled data.

We replicated these steps to create our dataset (i.e, collection of .pt files). We have generated .pt files for 4 designs, which are:

- simple_spi
- spi
- i2c
- ss_pcm

The reason to choose these designs was that they were related designs, and we wanted to keep generation time as low as possible.

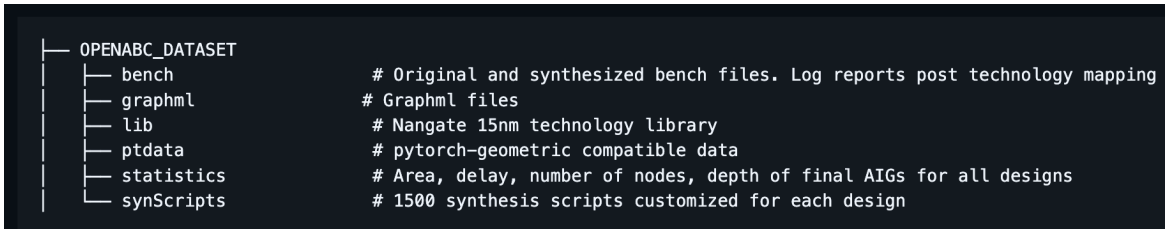

Fig 2.1: Directory structure for Data generation

From the official repository, all the necessary files for data generation were cloned.
The cloned datagen folder had the necessary automation and utilities scripts.

The following steps were followed:

Step 1: Run automate_synthesisScriptGen.py to generate a customised synthesis script for 1500 synthesis recipes. The reference scripts were kept under the synScripts folder.

Step 2: Run automate_bultkSynthesis.py to generate a shell script for a design. Run the shell script (synthesisBulk_<design_name>.sh)to perform the synthesis runs. Make sure yosys-abc is available in PATH. The shell script converts the previously generated synthesis file into a log file and zips it.

Step 3: Run automate_synbench2Graphml.py to generate a shell script for generating Graphml files. The shell script invokes andAIG2Graphml.py using 21 parallel threads processing data of each synthesis runs in sequence.

Step 4: Run the synthID2SeqMapping.py utility to generate synthID2Vec.pickle file containing numerically encoded data of synthesis recipes. (In the official document, this step was carried out at the end, i.e step 6, by step 4 needs pickle file, which is generated by this step)

Step 5: Run PyGDataAIG.py to generate PyTorch data for each graphml file of the format designIP_synthesisID_stepID.pt.

Step 6: Run collectAreaAndDelay.py and collectGraphStatistics.py to collect information about the final AIG's statistics. Post that, run pickleStatsForML.py, which will output synthesisStatistics.pickle file.

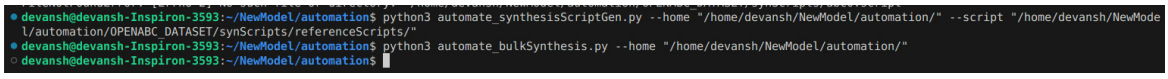

Fig 2.2: Steps 1 and 2

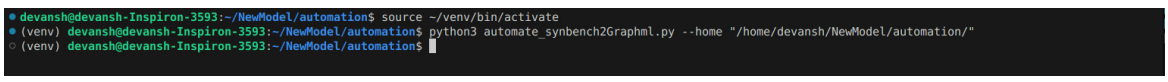

Fig 2.3: Step 3

Fig 2.4: Step 5

2.1.1 Problems faced

We ran GML generation shell script file, which gave permission denied error. Hence, we updated the permissions using the chmod+x <filename> command. The script takes an AIG circuit description in .bench format and converts it into a directed multigraph (MultiDiGraph) representation using NetworkX, then saves it as a .graphml file.

The ordering provided on GitHub is not correct. During step 4, we needed a pickle file, synthIDtoVec.pickle, which is generated in step 6, so we had to execute step 6 first.

2.1.2 Insights and Observations

As mentioned earlier, they have preserved all AIGs: starting, intermediate and final AIGs. And you can see that the bench files for all the steps were created and stored. See Fig. 5. Nomenclature is <design_name>_syn<id>_step<no>.pt.zip

For each design, we have synthesised 1500 such files. And the steps represent the state of an AIG of an operation.

So step 0 represents the original, unsynthesized AIG, the design immediately after conversion to AIG format. No synthesis transformations have been applied yet. Similarly, Step 20 refers to the post-synthesis AIG after applying 20 synthesis transformations. This is used to evaluate QoR (Quality of Result) improvements like node reduction or area minimisation.

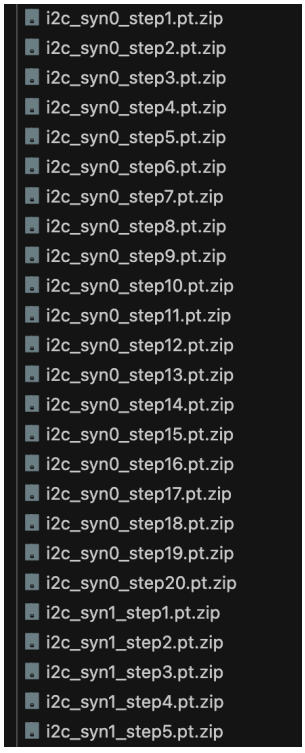

Fig 2.5: Generated pt files

```
Extracted data from the 'i2c_syn0_step0.pt' pt file:
 Data(and_nodes=1169, desName=[1], edge_index=[2, 2466], edge_type=[2466], longest_path=15, node_id=[1474], node_type=[1474], not_edges=11
88, num_inverted_predecessors=[1474], pi=177, po=128, stepID=[1], synID=[1], synVec=[20])
```

Fig 2.6: PyTorch data

2.1.3 Insides of a PyTorch data object

Figure 6 shows the data captured from each AIG. This includes edge_index=[2, 1992], node_id=[1474], node_type=[1474], num_inverted_predecessors=[1474], edge_type=[2466], longest_path=15, and_nodes=1169, pi=164, po=132, not_edges=1188, desName=[1], synVec=[18], synID=[1], stepID=[1].

For example, node_type [1474] - vector capturing node types of each node. There are three types of nodes as defined by the author: primary input, primary output, and internal node.

2.2 Training the Model

The GitHub code was generated on an older PyTorch version. Since the newer versions are not backwards compatible, a virtual environment was created.
All the necessary Python libraries were installed, which include torch, torch_geometric, numpy, pandas, scikit-learn, networkx, etc.

The following parameters were required to run train.py

- batch_size - input batch size for training (default: 64)
- lr - learning rate (default: 0.001)
- lp - Learning problem (QoR prediction: 1, Classification: 2)
- epochs - number of epochs to train (default: 80)
- dataset - Split strategy (set1/set2/set3, default: set1)
- rundir - Output directory path to store result
- datadir - Dataset directory containing processed dataset, train test split file csvs
- target - Target label (nodes/area/delay), default: "nodes"

The following specifications were provided while running the train function
python3 ./qor/SynthNetV1/train.py --datadir ./OPENABC-D --rundir ./OUTPUT/NETV1_set1 --dataset set1 --lp 1 --lr 0.001 --epochs 20 --target delay

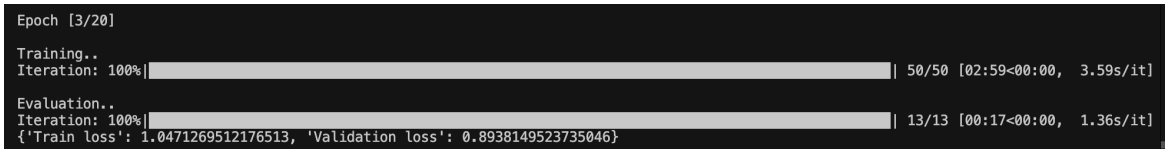

Fig 2.7a: Epoch 3

```
Epoch [20/20]

Training..
Iteration: 100%|| 50/50 [04:09<00:00,  4.99s/it]

Evaluation..
Iteration: 100%|| 13/13 [00:25<00:00,  1.93s/it]
{'Train loss': 0.9573566174507141, 'Validation loss': 1.0012487816810607}
```

Fig 2.7b: Epoch 20

```
(pyg171env) ashwiniyengar@Ashwins-Air Models_dobara % python3 ./qor/SynthNetV1/evaluate.py --datadir ./OPENABC-D/lp1 --rundir ./OUTPUT/NETV1_set1 --dataset
set1 --model "gcn-epoch-20-val_loss-1.001.pt" --target delay
Iteration: 100%| 63/63 [01:21<00:00,  1.30s/it]

Dataset type: train
MAPE (spi): 2.156796718703779
MAPE (simple_spi): 1.6895831055058335
MAPE (ss_pcm): 0.9852373351491375
MAPE (i2c): 9.185615828405698
Iteration: 100%| 32/32 [00:38<00:00,  1.19s/it]

Dataset type: test
MAPE (ss_pcm): 1.0679496946989824
MAPE (spi): 1.4549186344828027
MAPE (simple_spi): 3.2564184713243662
MAPE (i2c): 11.14286709927162
**********
Final run statistics
**********
Total Params: 74173
Training loss per sample:0.9606123180389404
Test loss per sample:0.9896522989273071
```

Fig 2.7c: Evaluation

```
DUMP_DIR = None
criterion = torch.nn.MSELoss()
```

Fig 2.8: Code snipper shows Loss function used

2.2.1 Metrics used

Loss function: Mean squared error (MSE)

MAPE was used as a metric of accuracy.

```
datasetDict =  {
    'set1' : ["train_data_set1.csv","test_data_set1.csv"],
    'set2' : ["train_data_set2.csv","test_data_set2.csv"],
    'set3' : ["train_data_mixmatch_v1.csv","test_data_mixmatch_v1.csv"]
}
```

OPENABC-D > lp1 > test_data_set1.csv > data

	fileName
1	spi_syn1071_step0.pt.zip
2	i2c_syn1004_step0.pt.zip
3	ss_pcm_syn1416_step0.pt.zip
4	simple_spi_syn1268_step0.pt.zip
5	simple_spi_syn1373_step0.pt.zip
6	ss_pcm_syn1382_step0.pt.zip
7	ss_pcm_syn1485_step0.pt.zip
8	spi_syn1495_step0.pt.zip
9	ss_pcm_syn1177_step0.pt.zip
10	i2c_syn1248_step0.pt.zip
11	spi_syn1111_step0.pt.zip
12	simple_spi_syn1402_step0.pt.zip
13	simple_spi_syn1257_step0.pt.zip
14	ss_pcm_syn1125_step0.pt.zip
15	ss_pcm_syn1200_step0.pt.zip
16	ss_pcm_syn1157_step0.pt.zip
17	ss_pcm_syn1239_step0.pt.zip
18	ss_pcm_syn1109_step0.pt.zip
19	i2c_syn1173_step0.pt.zip
20	spi_syn1451_step0.pt.zip
21	ss_pcm_syn1026_step0.pt.zip
22	ss_pcm_syn1298_step0.pt.zip
23	ss_pcm_syn1300_step0.pt.zip
24	i2c_syn1059_step0.pt.zip
25	i2c_syn1034_step0.pt.zip
26	ss_pcm_syn1276_step0.pt.zip
27	spi_syn1162_step0.pt.zip
28	simple_spi_syn1309_step0.pt.zip

OPENABC-D > train_data_set1.csv > data

	fileName
1	simple_spi_syn59_step0.pt.zip
2	spi_syn368_step0.pt.zip
3	spi_syn549_step0.pt.zip
4	i2c_syn591_step0.pt.zip
5	i2c_syn129_step0.pt.zip
6	i2c_syn811_step0.pt.zip
7	spi_syn375_step0.pt.zip
8	ss_pcm_syn925_step0.pt.zip
9	simple_spi_syn569_step0.pt.zip
10	ss_pcm_syn5_step0.pt.zip
11	ss_pcm_syn708_step0.pt.zip
12	ss_pcm_syn312_step0.pt.zip
13	ss_pcm_syn628_step0.pt.zip
14	spi_syn995_step0.pt.zip
15	simple_spi_syn882_step0.pt.zip
16	ss_pcm_syn870_step0.pt.zip
17	ss_pcm_syn42_step0.pt.zip
18	simple_spi_syn9_step0.pt.zip
19	ss_pcm_syn797_step0.pt.zip
20	simple_spi_syn512_step0.pt.zip
21	i2c_syn92_step0.pt.zip
22	spi_syn111_step0.pt.zip
23	simple_spi_syn407_step0.pt.zip
24	ss_pcm_syn996_step0.pt.zip
25	simple_spi_syn643_step0.pt.zip

Fig 2.9: Dataset

2.2.2 Insights and Observations

- While exploring the dataset used for testing, we noticed that it primarily contained all the step 0 AIGs—essentially, the raw, untransformed designs. This is evident from the GitHub repository. Since step 0 represents the original design before any synthesis transformations, the training process was essentially performed on the same initial AIG, repeated about 1500 times. Interestingly, there was no mechanism in the code to actually inject or apply different synthesis recipes during training.
- In Figure 2.9, we saw that set3 included mixed data, specifically the step 20 files, which are AIGs after synthesis transformations have been applied.

This felt inconsistent for a couple of reasons. First, testing on step 0 data means we're working with raw AIGs, without any indication of the recipes used—which limits the model's ability to learn recipe-specific behaviours. Second, if we switch and test on step 20 (transformed) AIGs, it implies the model is learning to predict QoR based on already-transformed data, defeating the original purpose of predicting QoR before synthesis.

PS: These observations are based solely on the implementation we explored—no additional experiments or validations were conducted beyond that

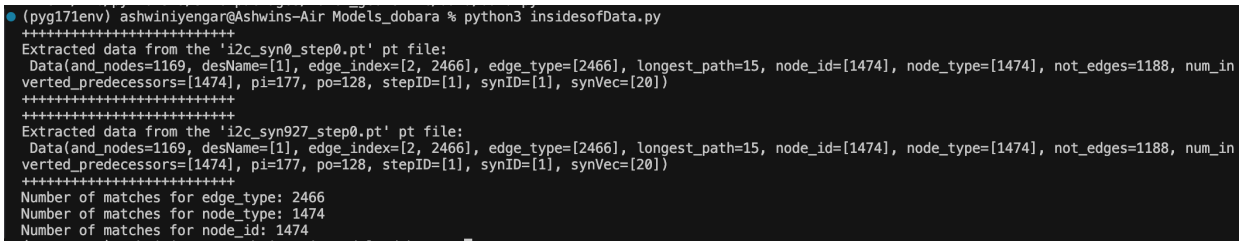

Fig 2.10: Insides of a PyTorch Data object

To verify whether different PyTorch data objects corresponding to the same design at step 0, but with different synthesis recipes, were distinct, we inspected the data objects closely. Specifically, we compared i2c_syn0_step0 and i2c_syn927_step0.

Our findings showed that these two data objects were identical. Key vectors such as edge_type, node_type, and node_id were the same, confirming that no recipe-specific variation was reflected in the data at step 0. This reinforced our earlier observation that the synthesis recipes weren't being incorporated into the model input, at least not at this stage.

This gave us a better understanding of the problem and helped us develop our custom model. Let’s have a look at that model.

3. ABColabX

3.1 Model Overview

It is an end-to-end deep learning model built to predict delay values based on a circuit's structure and its associated recipe. The model is designed to handle two types of inputs:

- An And-Inverter Graph (AIG), which represents the logic circuit structure. It's passed in as a PyTorch Geometric Data object that includes node features and graph connectivity.
- A recipe which is essentially a sequence describing the set of processing steps applied to the circuit.

What does the model do?

The model first encodes both inputs separately:

- The Graph Encoder captures structural and feature-based information from the AIG using a powerful variant of GraphSAGE. This helps the model learn rich representations of the circuit layout.
- The Recipe Encoder tokenises and encodes the sequence of recipe steps using embeddings and transformer-style layers, allowing it to understand the context within the recipe.

Once both parts are encoded, they are fused using a bidirectional cross-attention layer, which helps the model learn how the circuit structure and the recipe influence each other. This combined representation is then passed through a predictor — a simple neural network that outputs the final predicted delay.

How does it learns?

The model is trained using Mean Squared Error (MSE) as the primary loss function, which penalises predictions that are far off from the actual delay. During evaluation, we also compute the Mean Absolute Percentage Error (MAPE) to understand how close our predictions are to the actual values in relative terms, which gives a more intuitive sense of model performance.

Now let’s deep dive into the model.

3.2 Deep Dive into the Model

To better understand how our delay prediction model works, let's take a closer look at its building blocks — starting from how the data flows into the model to how it's processed and encoded before making a prediction.

3.2.1 Input & Directory Structure

The dataset used for training the model is organized in a clean and modular folder hierarchy under a top-level `data/` directory. This structure helps in keeping the different components of the dataset isolated and easy to manage:

- ptfiles/: Contains PyTorch Data objects (serialized .pt files) for each design, representing the And-Inverter Graph (AIG) structures. These are generated using the same data extraction method described in Section 2.1: Data Generation of Animesh's work. Only difference being, we just created one .pt file, i.e step 0.
- out_label/: Holds the output delay labels corresponding to each design, stored in CSV format as <design_name>.csv. Each file contains a single float value — the delay we're trying to predict.
- recipes/: Each design also comes with a recipe file, again as <design_name>.csv, which originally contained string-based instructions. These are later tokenised for input into our model.

3.2.2 Dataset Creation and Preprocessing

Before feeding data into the model, we need to convert it into a format that the neural network can process.

- Tokenisation of Recipes: Recipes start as strings describing operations. We apply a simple tokenisation strategy using a mapping like {'fsto': 0, 'rf': 1, 'rs': 2, 'rw': 3, 'b': 4,…} to convert these into integer tokens. This numerical format allows embedding layers to process them effectively.
- Feature Extraction from AIGs: From the PyTorch Data object (generated from the ptfiles), we extract a few key features:
  - node_types: Represents the logic gate type.
  - num_inverted_predecessors: Indicates the number of inverted predecessors for each node.

These features were chosen deliberately:

- node_types: give us semantic information about the number of PIs, POs and internal nodes our graph has — essential for understanding the circuit structure.
- num_inverted_predecessors: captures signal inversion patterns, which are crucial in determining the logical delay paths in digital circuits.

"Together, they form a concise and informative representation of the graph that is small enough to be tractable, yet rich enough to capture key architectural patterns influencing delay."

3.2.3 Neural Network choice

1. **Graph Encoder:** Captures structural and feature-based information from the AIG and helps the model learn rich representations of the circuit layout. We use a modified GraphSAGE architecture as our graph encoder, which processes the graph-structured AIG data.

models/graph_encoder.py: Contains the Graph encoder.

Why GraphSAGE?

- We tried GCN with different layer configurations. GraphSAGE proved to be better in terms of performance.

We believe that GraphSAGE performed better over traditional GCNs because,

- It performs neighbourhood aggregation, allowing the model to generalise better across graphs of different sizes.
- The use of residual connections (i.e., x = x1 + x2) stabilises training and helps gradients propagate deeper into the model.

Other layers include,

- BatchNorm layers help in faster convergence and reduce internal covariate shift.
- Dropout (at 30%) adds regularisation, reducing the chance of overfitting.

- Global Mean Pooling aggregates node-level features into a fixed-size graph-level embedding, essential for downstream fusion and prediction tasks.

2. **Recipe Encoder:** The Recipe Encoder focuses on interpreting the processing sequence or "recipe" associated with each design. These recipes originally provided as human-readable strings are tokenized and embedded into a numerical format that the model can learn from.

models/recipe_encoder.py: Contains the recipe encoder

From Tokens to Embeddings

- Each recipe starts as a sequence like “rs,rf,st,rwz,rw,st,...,rs,b,b,b”. These are first mapped to integer token IDs (using a map), so that we can embed them using a standard PyTorch nn.Embedding layer. This embedding layer learns a vector representation for each token, capturing its role in the context of delay prediction.

Positional Encoding

- Transformers have no built-in sense of order, so we inject sequence position information using Sinusoidal Positional Encoding. This method ensures that each token carries not only what it is, but also where it is in the sequence.
- This encoding is added to the embeddings, giving the model the ability to understand temporal dependencies in the sequence of operations.

“For example, certain key operation patterns, such as a chain of fsto operations followed by a fres, are often indicative of important synthesis behaviour, especially in lossless synthesis.
By applying sinusoidal positional encoding, we give the model the capability to not only recognize the presence of operations but also learn how their position influences outcome. If the model discovers that such patterns impact delay, the positional information allows it to harden those beliefs and use them meaningfully in prediction.”

Transformer Encoder

- At the heart of the Recipe Encoder is a multi-layer Transformer encoder, a powerful sequence model that can capture long-range dependencies and contextual relationships between operations in the recipe.
- Each token in the sequence attends to all others through multi-head self-attention, helping the model understand how each operation relates to the rest.

“This will ensure that the transformer will learn and recognise relations like b-rw or fsto-fres, which are proven to work better together.”

Sequence Aggregation

- Once the transformer processes the full sequence, we need to compress it into a fixed-size vector for downstream fusion. We do this using:
  - Adaptive Average Pooling, which reduces the sequence dimension into a single embedding.
  - Layer Normalisation, which helps stabilise training.

- This final output — one embedding per recipe — is passed to the Cross-Attention Fusion layer to be combined with the graph embedding.

3. **Cross-Attention Fusion:** Once we have the structural understanding from the Graph Encoder and the procedural insight from the Recipe Encoder, we need a way to blend these two perspectives. That’s where the Cross-Attention Fusion module comes in.

In many real-world systems, structure and operations are deeply intertwined — the same structure can behave differently under different synthesis strategies (recipes), and vice versa. So instead of simply concatenating or averaging these embeddings, we let them interact and influence each other’s representation through attention.

models/fusion_model.py: Contains fusion stages

Custom Bidirectional Cross-Attention

This module uses a custom Bidirectional Cross-Attention mechanism. Here's what it does:

- It lets the graph embedding attend to the recipe embedding, and vice versa.
- This helps both sides focus on the most relevant aspects of the other. For example:
  - The graph might attend to certain synthesis steps that affect its delay.
  - The recipe might emphasise parts of the structure it's transforming.
- This results in two refined representations one for the graph (g_attn) and one for the recipe (r_attn) — both now containing richer, context-aware information.

Once attention is applied, the model fuses the graph-aware recipe and recipe-aware graph embeddings using a simple average. Finally, the fused vector goes through a linear projection layer to match the expected shape and ensure trainability.

Why This Matters?

- Inter-modality learning: Each modality (graph/recipe) learns in the context of the other.
- Interpretability: You can inspect attention weights to see which recipe steps influence which graph patterns.
- Performance: In early tests, this kind of fusion leads to more expressive and accurate embeddings, improving prediction metrics like MAPE and MSE.

Final Model Flow Recap:

1. GraphEncoder → Understands the circuit's netlist structure.
2. RecipeEncoder → Understands the synthesis strategy.
3. CrossAttentionFusion → Merges the two perspectives using mutual attention.
4. Predictor → Outputs a delay estimate using fused information.

3.3 Training the Model

Once the model architecture is in place, the next step is training it effectively on real-world data. This involves preparing the input data, setting up the training pipeline, and tracking performance using suitable metrics.

train.py: Training script

Inputs to the Model

The training setup uses a dataset prepared using the following inputs:

1. PT Files: These are preprocessed PyTorch graph objects representing structural netlists from 7 distinct designs. They are stored in the ptfiles/ directory.
2. Recipe-Delay Pairs: A set of 1,500 randomly generated synthesis recipes was used, and their corresponding delay values were computed using the methods described in the Assignment 1. Each recipe-delay pair is stored in the recipes/ and out_label/ folders, respectively as described in Section 3.2.1. We also made sure to have different values.

Training Script – train.py

The training process is orchestrated via the train.py script, which includes:

- Data Preparation: The training data is processed as described in Section 3.2.2. Recipes are tokenized, and graph features are extracted (e.g., node_types, num_inverted_predecessors).
- Model Initialisation: The DelayPredictor model is instantiated with the Graph Encoder, Recipe Encoder, Cross-Attention Fusion module, and a predictor head.
- Optimiser: We use the AdamW optimiser, known for its robust handling of weight decay and generalisation.
- Loss Function: The model is trained using Mean Squared Error (MSE) loss, a standard metric for regression tasks that penalises larger deviations more severely.
- Model Checkpointing: During training, the model keeps track of the best performing state (based on validation loss) and saves it for later evaluation.

Evaluation

After training, the model is evaluated on the test and validation datasets using the following metrics:

- Mean Squared Error (MSE) – Measures average squared differences between predicted and actual delay values.
- Mean Absolute Percentage Error (MAPE) – Reflects percentage-based error, giving insight into relative prediction accuracy.
- R^2 Score – Indicates how well the model explains the variance in the data.

Together, these metrics provide a comprehensive picture of model performance both in absolute terms (MSE) and relative terms (MAPE), with R^2 offering a sense of generalisation.

3.4 Finetuning the Model

We adopted a targeted fine-tuning approach to enhance the model's adaptability to new designs and ensure its performance remains robust across diverse input distributions.

The process began by loading a previously trained and saved model checkpoint, this served as our base model, which had already learned general representations from the original training dataset. Instead of retraining from scratch, we leveraged this prior knowledge and adapted the model to a new set of unseen designs.

To support this, a new dataset was created, consisting of:

- New PT files corresponding to different circuit designs
- Associated recipes and delay values, curated and formatted as discussed in the earlier dataset preparation section

Once the data was prepared, we continued training the model using this fresh dataset. The idea was to let the model refine its understanding and better align with the characteristics of the new designs, while still retaining the generalisation ability it had acquired earlier.

To evaluate the effectiveness of finetuning, we captured the performance metrics before and after the finetuning phase. This included:

- Mean Squared Error (MSE)
- R^2 Score
- Mean Absolute Percentage Error (MAPE)

These metrics reflected the model's improvement after exposure to the new design space. Finetuning allowed the model to recalibrate without overfitting or forgetting previously learned patterns, showcasing its ability to scale and adapt efficiently.