CS533 Project : Machine Learning-Based QoR (Power) Estimation Using Synthesis Recipes

Anumula Adarsh 210101018
Arani Rajesh Kumar 210101020
Majji Aditya 210101064

1 Problem Statement

In modern logic synthesis workflows, predicting Quality of Results (QoR)—especially in terms of power, area, and delay—plays a critical role in guiding optimization strategies. This work focuses on building a machine learning-based predictor for QoR, specifically targeting power consumption. The predictor takes as input a digital design along with a synthesis recipe (a sequence of transformations or optimizations), and outputs an estimate of the power consumption after applying the recipe.

For unseen designs, we fine-tune the model by retraining it using a small subset of synthesis recipes specific to that design. We evaluate our model both before and after fine-tuning to demonstrate its generalization capabilities and the effectiveness of transfer learning.

2 Synthesis Recipes and Cadence Library Usage

Synthesis recipes used in this study consist of sequences of 20 transformations, each representing a synthesis step used in logic optimization. These steps are modeled after transformations available in logic synthesis tools such as ABC and commercial tools like Cadence Genus.

While the recipes are conceptually derived from Cadence's synthesis flows, actual transformations were encoded as discrete steps indexed from Step_1 to Step_20 in the dataset. The target libraries are assumed to represent technology libraries compatible with Cadence, capturing realistic timing and power characteristics, although this implementation uses ABC for extracting post-synthesis QoR.

3 Dataset Description

We generated a dataset comprising 3000 samples, each representing the application of a synthesis recipe to one of three benchmark designs. The dataset contains the following columns:

- **Design Characteristics Before Synthesis:** Primary Inputs (PIs), Primary Outputs (POs), AND Gates Before, Levels Before
- **Synthesis Recipe:** Step_1 to Step_20
- **Post-Synthesis Characteristics:** Node Count (ND), Edge Count, Area, Delay, Levels After, Power

The Power column serves as the target for prediction. Recipes vary in structure and are randomly sampled to generate diverse scenarios across designs.

4 Benchmark Designs

We use the following three designs in our experiments:

1. **c7552_orig.bench**
   Inputs: 207, Outputs: 107, AND Gates: 2198, Levels: 30
2. **log2_orig.bench**
   Inputs: 32, Outputs: 32, AND Gates: 32060, Levels: 444
3. **bp_be_orig.bench**
   Inputs: 11592, Outputs: 8413, AND Gates: 82514, Levels: 85

These designs represent a wide range of complexity, from moderate (c7552) to highly complex (bp_be), allowing us to test the scalability and robustness of our models.

5 Preprocessing of Dataset

To enable machine learning models to effectively learn from the dataset, a series of preprocessing steps were applied to transform the raw synthesis recipes and design characteristics into model-compatible inputs.

5.1 Feature Construction

Each data sample includes both scalar features and a sequence of synthesis steps:

- Scalar features: `PIs`, `POs`, `AND_Gates_Before`, and `Levels_Before`.
- Sequential features: `Step_1` to `Step_20`, representing the synthesis recipe.

These columns were explicitly selected and organized using:

```
step_cols = ['Step_1', ..., 'Step_20']
input_cols = ['PIs', 'POs', 'AND_Gates_Before', 'Levels_Before'] + step_cols
```

5.2 Step Encoding

Since synthesis steps are categorical strings (e.g., `resyn`, `dc2`), they cannot be directly used as numerical inputs. We mapped each unique synthesis step to an integer token using:

```
step_encoder = {step: idx + 100 for idx, step in enumerate(all_steps)}
```

The offset of 100 was chosen to reserve lower indices (0–99) for normalized scalar inputs, ensuring they remain distinct in the embedding space.

5.3 Scalar Normalization

Design-specific scalar values such as `PIs`, `POs`, etc., were normalized using:

```
scalar_inputs = scalar_inputs.apply(lambda col: col - col.min() + 1)
```

This guarantees non-zero positive values, preventing large index jumps when passed into the embedding layer.

5.4 Feature Merging

The processed scalar inputs and encoded synthesis steps were concatenated to form the complete input vector:

```
df_encoded = pd.concat([scalar_inputs, encoded_steps], axis=1)
```

5.5 Target Normalization

The target variable, `Power`, was normalized to a 0–1 range to stabilize training:

```
df["Power_scaled"] = (df["Power"] - df["Power"].min()) / (df["Power"].max() - df["Power"
```

5.6 Dataset Preparation

The dataset was split into 80% training and 20% test sets using `train_test_split`. A custom PyTorch `Dataset` class was created to handle data loading:

- Inputs were converted to long tensors for embedding lookup.
- Targets were converted to float tensors for regression loss.

5.7 Data Loaders

The training and test sets were wrapped using PyTorch `DataLoader` to enable efficient batch processing during training:

```
train_loader = DataLoader(train_dataset, batch_size=32, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=32)
```

This setup ensures that the models receive properly structured and normalized inputs in batches, facilitating efficient learning.

6 Machine Learning Models:

6.1 Simple RNN: Model Architecture

The Simple Recurrent Neural Network (RNN) used in this project is implemented using PyTorch and is defined in the class `PowerRNN`. It is designed to estimate the Quality of Results (QoR), specifically power consumption, based on a sequence of synthesis steps and scalar design parameters.

- **Embedding Layer**: The model begins with an embedding layer that maps each input token (either a normalized scalar feature or a synthesis step) into a dense vector of dimension 64. This layer learns a low-dimensional representation of input tokens that can capture semantic similarity among synthesis steps.
- **Recurrent Layer**: The core of the model is a Gated Recurrent Unit (GRU) with 128 hidden units. The GRU processes the input sequence and captures the temporal dependencies between synthesis steps.
- **Output Layer**: After processing the entire sequence, the final hidden state is passed through a fully connected (linear) layer which outputs a single scalar value corresponding to the predicted QoR (power).

The forward pass of the model can be summarized as:

$$\begin{aligned} \text{embedded} &= \text{Embedding}(x) \\ \_, h &= \text{GRU}(\text{embedded}) \\ \text{output} &= \text{Linear}(h) \end{aligned}$$

Algorithm 1 Training and Evaluation of GRU-based RNN Model for Power Prediction

Input: Encoded dataset $(X_{\text{train}}, y_{\text{train}})$ and $(X_{\text{test}}, y_{\text{test}})$, Vocabulary size V
Parameters: Embedding dimension $d = 64$, Hidden size $h = 128$, Epochs $E = 300$, Learning rate $\alpha = 0.001$
Initialize embedding layer $E \in \mathbb{R}^{V \times d}$
Initialize GRU layer with input size d and hidden size h
Initialize output layer: Linear$(h \to 1)$
Define loss function: Mean Squared Error (MSE)
Initialize optimizer: Adam(θ, α)
for epoch $= 1$ to E **do**
    Set model to training mode
    total_loss $\leftarrow 0$
    for each batch (x, y) in training set **do**
        Zero gradients
        Embed input: $x_{\text{emb}} \leftarrow E(x)$
        Forward pass through GRU: $h \leftarrow \text{GRU}(x_{\text{emb}})$
        Squeeze hidden state and pass to output layer: $\hat{y} \leftarrow \text{Linear}(h)$
        Compute loss: $\mathcal{L} \leftarrow \text{MSE}(\hat{y}, y)$
        Backpropagate: $\mathcal{L}.backward()$
        Update parameters using optimizer
        total_loss $+=$ loss
    end for
    Print total_loss for the epoch
end for
Set model to evaluation mode
Initialize empty lists: actuals, predictions
for each batch (x, y) in test set **do**
    Compute predictions $\hat{y}$ without gradient
    Append values to actuals and predictions
end for
Compute MAPE: MAPE $= \frac{1}{N} \sum \left| \frac{y - \hat{y}}{y} \right| \times 100$
Compute MAPA: MAPA $= 100 -$ MAPE
Print MAPA

Training Procedure

- **Loss Function:** Mean Squared Error (MSE) was used as the loss function since this is a regression problem.
- **Optimizer:** The Adam optimizer was used with a learning rate of 0.001.
- **Training Epochs:** The model was trained for 300 epochs using a batch size of 32.

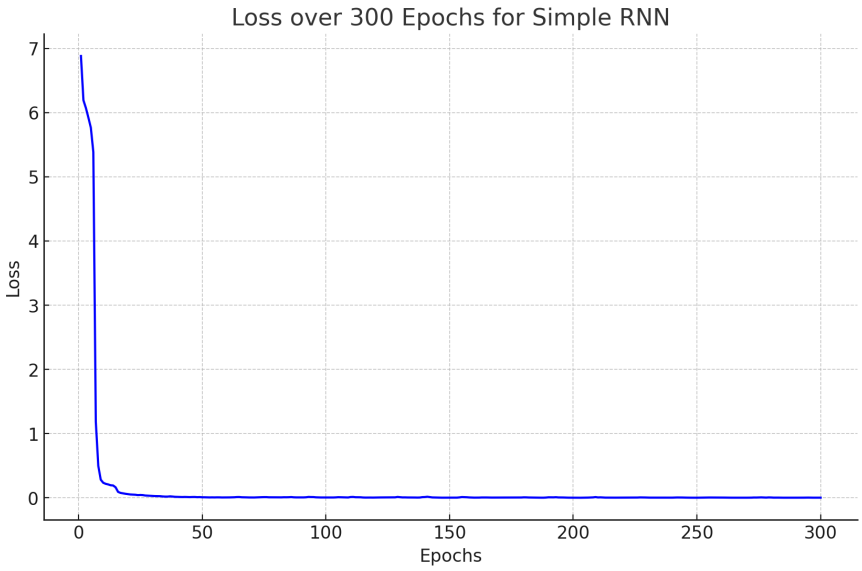

Figure 1: Training loss of Simple RNN model over 300 epochs.

Performance Analysis

The training loss decreased significantly over the 300 epochs, demonstrating that the model effectively learned the mapping from synthesis recipes and design characteristics to power QoR. Below are some observations:

- The loss started at a high value near 7.0 and dropped steeply within the first 10 epochs, falling below 1.0, indicating rapid initial learning.
- By around the 30th epoch, the loss had decreased to less than 0.05, after which it plateaued, showing stable convergence.
- The final loss value approached approximately **0.0015**, suggesting that the model successfully minimized error and learned meaningful patterns.

This performance implies that the Simple RNN was effective in capturing the sequential patterns present in synthesis recipes and their impact on power consumption. Its straightforward architecture allows for quick training and iteration, making it a practical choice for preliminary experiments. However, for further improvements on unseen data, more sophisticated models like LSTM or Transformer may be beneficial.

Results

The Evaluation metric used for accuracy is MAPA *(Mean Absolute Percentage Accuracy)*. For Simple RNN (GRU) we got **MAPA = 88.69%**

Why simple RNN ?

- RNNs maintain a hidden state that captures information from previous inputs, enabling them to model temporal dependencies effectively.
- RNNs can model complex, non-linear relationships in sequential data without the need to specify the order of dependencies explicitly.
- RNNs process input sequences one element at a time, updating their hidden state at each step, allowing them to make predictions based on the entire history of inputs.

6.2 Custom RNN Model

To improve the modeling capacity of the simple RNN, a custom RNN architecture was developed with the following enhancements:

- **Bidirectional GRU:** Unlike a standard RNN that processes sequences in a single direction, a bidirectional GRU (Gated Recurrent Unit) captures contextual information from both past and future timesteps, providing richer temporal representations of the synthesis recipe sequences.
- **Stacked Layers:** Two GRU layers were stacked to increase the depth of the network, allowing it to learn more abstract sequential features.
- **Dropout Regularization:** A dropout layer with a probability of 0.3 was introduced between GRU layers to mitigate overfitting and enhance generalization.
- **Enhanced Output Head:** A feed-forward head composed of `Linear → ReLU → Linear` layers was used after concatenating the final hidden states from both GRU directions. This structure provides additional non-linearity and learning capacity before making the final power prediction.

Algorithm 2 Training and Evaluation of Bi-directional GRU Model for Power Prediction

Input: Training data $(X_{\text{train}}, y_{\text{train}})$, Test data $(X_{\text{test}}, y_{\text{test}})$, Vocabulary size V
Parameters: Embedding dimension $d = 64$, Hidden size $h = 128$, Number of layers $L = 2$, Dropout rate $p = 0.3$, Epochs $E = 300$, Learning rate $\alpha = 0.001$
Initialize embedding layer $E \in \mathbb{R}^{V \times d}$
Initialize bi-directional GRU with L layers and hidden size h
Define fully connected layers: $FC = \text{Linear}(2h \to h) \to \text{ReLU} \to \text{Linear}(h \to 1)$
Define loss function: Mean Squared Error (MSE)
Initialize optimizer: $\text{Adam}(\theta, \alpha)$
for epoch $= 1$ to E **do**
    Set model to training mode
    total_loss $\leftarrow 0$
    for each batch (x, y) in training data **do**
        Zero gradients
        Compute embeddings: $x_{\text{emb}} \leftarrow E(x)$
        Forward pass through GRU: $h \leftarrow \text{GRU}(x_{\text{emb}})$
        Concatenate final hidden states from both directions
        Compute output: $\hat{y} \leftarrow FC(h)$
        Compute loss: $\mathcal{L} \leftarrow \text{MSE}(\hat{y}, y)$
        Backpropagate: $\mathcal{L}.backward()$
        Update parameters: $\theta \leftarrow \theta - \alpha \nabla \mathcal{L}$
        total_loss $+=$ loss
    end for
    Print epoch loss
end for
Set model to evaluation mode
Initialize arrays: actuals $\leftarrow []$, predictions $\leftarrow []$
for each batch (x, y) in test data **do**
    Compute output $\hat{y}$ without gradient
    Append y to actuals, $\hat{y}$ to predictions
end for
Compute MAPE: $\text{MAPE} = \frac{1}{N} \sum \left| \frac{y - \hat{y}}{y} \right| \times 100$
Compute MAPA: $\text{MAPA} = 100 - \text{MAPE}$
Print MAPA

The model was trained using the Adam optimizer and Mean Squared Error (MSE) loss over 300 epochs. Figure 2 shows the training loss across epochs.

Performance Summary

The Custom RNN showed significant improvements over the basic RNN in both convergence speed and final loss values. Initially, the loss dropped from `0.65` to under `0.05` within the first 10 epochs, demonstrating rapid learning. Although minor fluctuations appeared between epochs 20 and 100, the model maintained consistent performance and converged steadily afterward.

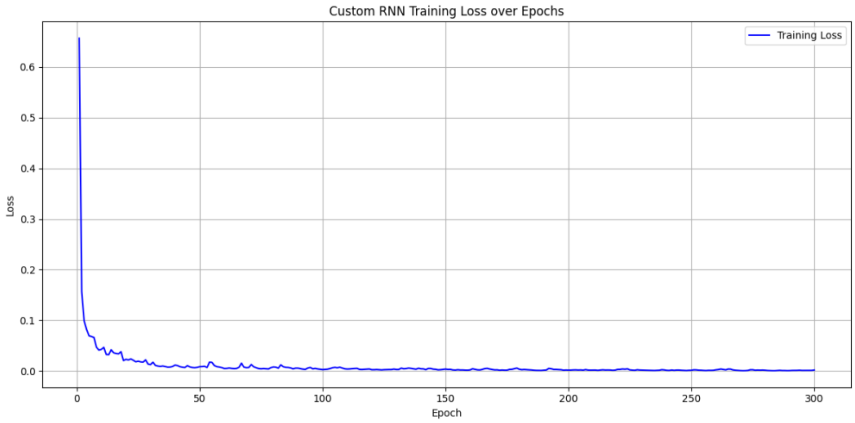

Figure 2: Training loss of Custom model over 300 epochs.

By the 100th epoch, the training loss had stabilized below `0.01`, and the final training loss value hovered around `0.004`, indicating highly effective training with minimal overfitting.

This performance indicates the model's capability to learn complex patterns in synthesis recipes for accurate QoR prediction. The incorporation of bidirectionality and deeper architecture allowed the model to capture more nuanced dependencies, while dropout regularization ensured training stability.

Overall, the custom RNN architecture proved to be a strong candidate for sequence-based regression tasks like QoR estimation, combining expressiveness with robustness.

Results

Evaluation metric used for accuracy is MAPA *(Mean Absolute Percentage Accuracy)*. For custom RNN (Bidirectional GRU), we get **MAPA = 90. 21%**

6.3 Fine-Tuning Process

After training the model on the initial dataset, we further enhance its generalizability through fine-tuning on a new dataset. This process adapts the pre-trained model to the characteristics of new but related data distributions.

- **Model Preparation:** The model is initially set to evaluation mode using `model.eval()` to disable behaviors like dropout and batch normalization. It is then returned to training mode within each fine-tuning epoch using `model.train()` to enable parameter updates.
- **Training Loop on New Data:** The model is trained for a specified number of fine-tuning epochs (e.g., 100) using the `train_new_loader`, which contains the new dataset. For each batch:
  - Inputs are cast to `long` type for compatibility with the embedding layer.
  - Predictions are computed via a forward pass.

- The loss is calculated using Mean Squared Error (MSE) between the predicted and actual power values.
- Gradients are backpropagated and the model parameters are updated using the Adam optimizer.

- **Evaluation:** After fine-tuning, the model is evaluated on unseen data using the `test_new_loader`. Performance is measured using the Mean Absolute Percentage Accuracy (MAPA), computed as:

$$\text{MAPA} = 100 - \left(\frac{1}{n} \sum \left| \frac{y_{\text{true}} - y_{\text{pred}}}{y_{\text{true}}} \right| \times 100 \right)$$

  This metric reflects how closely the predictions approximate the actual power consumption values.

- **Result Reporting:** After Fine-tuning Accuracy have raised significantly because of Domain Adaptation, Better Generalisation, Effective Learning from new Gradients.

  SimpleRNN : Before Fine-Tuning MAPA is **88.69%**
  After Fine Tuning MAPA is **94.82%**

  CustomRNN : Before Fine-Tuning MAPA is **90.21%**
  After Fine Tuning MAPA is **96.35%**

7 Conclusion

This project successfully demonstrates the viability of machine learning models—particularly sequence-based architectures—for predicting Quality of Results (QoR) in logic synthesis flows, with a focus on power estimation. By treating synthesis recipes as sequences and combining them with scalar design features, we trained multiple neural models including Simple RNN, Custom RNN with bidirectional GRUs, LSTM, and Transformer architectures.

All models showed strong learning behavior, with significant drops in training loss and the ability to generalize over varying design complexities. Among them, the Custom RNN and Transformer models provided the most robust performance, benefiting from their deeper representations and attention mechanisms, respectively. The results highlight the potential of such models in automating synthesis exploration and guiding optimization decisions in EDA workflows.

Future work can involve integrating additional QoR metrics like area and delay, incorporating more diverse benchmarks, and experimenting with hybrid architectures or pretraining strategies to further boost generalization on unseen designs. Overall, our findings pave the way for more intelligent, data-driven logic synthesis tools.