Synthesis QoR prediction

Praharsha Mahurkar
Department of ECE
University of California, San Diego
La Jolla, CA 92093
pmahurkar@ucsd.edu

Chinmay Samudra
Department of ECE
University of California, San Diego
La Jolla, CA 92093
csamudra@ucsd.edu

Ankur Sharma
Department of ECE
University of California, San Diego
La Jolla, CA 92093
ankursharma@ucsd.edu

Abstract

In the process of semiconductor chip development, Logical Synthesis plays a very important role. It translates the behavioral description of the circuit (which is specified using HDLs) to boolean gate level netlists. In the prevalent industry flows, every critical design IP is taken through multiple rounds of synthesis to determine the optimal gate level representation. Each of these rounds make use of a different synthesis "recipe". Furthermore, these rounds are time consuming due to design complexities and large tool run times. This adds to the total design cycle time and affects the chip's time to market. We explore an ML driven approach to predict the Quality of Result (QoR) of using a new synthesis recipe on a design IP. This approach aims to reduce the run time of the multiple design iterations.[1]

1 Introduction

Logic Synthesis translates the HDL circuit description coming from the RTL files to actual circuits built using standard cell gates. It outputs a gate level netlist. With ever increasing chip sizes, the time taken to Synthesize the designs increases. This time is mainly spent by the EDA tool (For eg. Cadence Genus) to read and optimize our design. The tool flow applies a sequence of synthesis transformations (i.e., a "synthesis recipe") to an and-inverter-graph (AIG) representation of a Boolean function to minimize its size or depth. This being a combinatorial problem, forms a good candidate for Machine Learning optimization.

Traditionally, the EDA tool chooses from a set of handcrafted synthesis recipes. The issue lies in the fact that the space of all synthesis recipes is vast and going through many recipes for every design is impractical. Hence we try various ML based approaches to find the best synthesis recipe for the given designs. This essentially entails predicting the QoR of using different recipes on the given designs and choosing one which maximizes the QoR (or minimizes the delay). Please note that we do not perform any sort of circuit reduction or optimization ourselves. That data is captured in the mapping from design recipe to QoR. Our algorithm simply learns what kind of delay a circuit can end up giving when optimized by a particular recipe.

We have tried three different supervised learning approaches to solve the problem at hand.

1. *Simple Linear Regression*: In this we try to predict the delay given the a) Design information such as number of AND and NOT gates, number of gates in the longest path and area of the path b) Synthesis recipe information. We use a simple linear model for prediction

2. *Neural Network based Regression*: We use the same features as above and try to predict the delay using a simple feed forward NN model with Backpropagation. Although straightforward to code, this approach took up a lot of our time since we frequently faced the vanishing gradient problem and had to make multiple amends to our model.

3. *Graph Convolutional Network*: In this approach, we try to predict the delay by reading in the AIG netlist (which is in a graph form - And-Invert-Graph) and passing it through layers of convolutional and simple feed forward network. This approach was suggested by the authors in [1] and our GCN model is inspired by their architecture.

For all the three approaches, we have used the dataset created by the authors in [1].

2 Related work

There is a growing academic and industry interest in using ML techniques in Electronic Design Automation [4]–[11]. Several problems in EDA (EDA), such as logic synthesis, placement and routing, and VLSI testing are combinatorial optimization problems that require sequential decision-making to achieve the target objective. Recent works [5]–[8] have shown that ML-guided exploration of this vast solution space can help us generate recipes that result in very good QoR. In one of these papers [5], the authors model the search of a good synthesis recipe as a classification problem which classifies the good and bad recipes. In some works [6]–[8] the task is solved by modeling Markov Decision Process (MDP) that is solved using reinforcement learning (RL).

However, the prior work is deficient on two fronts: (1) scalability to large designs; and (2) generalizability to previously unseen netlists. Our approach tries to tackle both these flaws and follows the work of the authors in [1] and [4] closely.

3 Problem Formulation

In hardware design cycle, synthesis is an important step, which gives the timing numbers. However, running synthesis for every design and every synthesis recipe is very time consuming and expensive. There is a need to predict the delay after training with some designs and recipes using number of AND gates, number of NOT gates, IP length, area as input parameters.

For predicting the delay, we need to minimize the mean square error:

$$MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y_i})^2 \tag{1}$$

4 Method/Approach

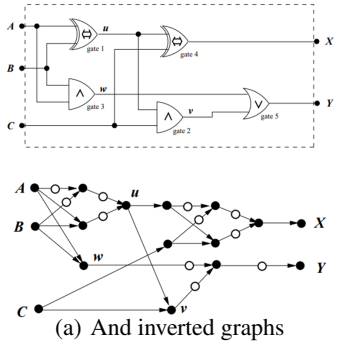
4.1 Dataset

We have used the dataset created by the authors in [1]. There are two versions of the dataset used:

1. *Tabular Data*: this was used for supervised learning in linear regression and NN regression. It consists of the design features of 42 open source designs after running 1500 synthesis recipes. These 1500 recipes are represented as unique vectors, each consisting of 19 numerical entries. The design features consist of number of AND gates, number of NOT gates, number of gates in the longest path of the IP (IP length), area of the IP and delay of the longest path of the IP (target variable). Figure 1b below shows how the way in which the tabular data is arranged.

We separate the data into training and testing by segregating the number of synthesis recipes of each design based on a normal distribution. We plot a normal curve with mean 300 and standard deviation of 25. We do this for all the 42 designs. This gave us the proportion of files to be extracted from an assortment of the 42 designs chosen randomly. This gave us a good testing dataset from total datapoints. The plot in Figure 1c shows the such a normal curve.

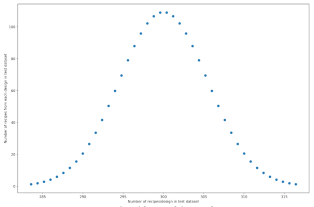
2. *Graph Data*: this was used for the GCN model. It consists of an AIG stored in a tabular format. This means that it is a literal graph with vertices and edges (vertices representing AND and NOT gates and edges representing the connection between them). Figure 1a depicts such an AIG.



(a) And inverted graphs

	design	recipe	ANDgates	NOTgates	ipLen	area	delay
0	Rocket_csr	0	6671	7234	22	4758.21	3369.65
1	Rocket_csr	1	6689	7159	19	4923.93	6041.40
2	Rocket_csr	2	6632	6791	21	5051.87	2811.36
3	Rocket_csr	3	6636	6813	21	5073.42	2920.17
4	Rocket_csr	4	6641	7132	20	4921.80	2879.31

(b) Dataframe snippet



(c) Normal distribution we use

Figure 1: Data representations

4.2 Linear regression

The problem requires multivariate linear regression, as output, delay, depends on a number of input features. The hypothesis function used is as follows:

$$h(x) = \theta_0 + \theta_1 * in_1 + \theta_2 * in_2 ... \theta_n * in_n \tag{2}$$

where $in_1, in_2 ... in_n$ are the input features.

However, the features are all of different ranges. This may have a bad impact on gradient descent algorithm. Therefore, feature normalization is required to bring them in same range. We used mean normalization for this: $in_1 = (in_1 - \mu)/\sigma$

We performed four experiments using various combinations of input features:

1. Linear regression 4 parameters(AND,NOT,IP Length ,Area): Here, we used four input features: number of AND gates, number of NOT gates, IP length and Area.
2. Linear regression using 5 parameters: Along with the above four parameters, we used synthesis recipe and encoded it using one hot encoding, as its categorical data
3. Linear regression using 5 parameters(Recipe as syn vectors):Instead of using one hot encoding on the synthesis recipes, we used syn_vector provided in the dataset
4. Linear regression using 6 parameters:In this experiment, we used number of AND gates, number of NOT gates, IP length, Area, synthesis recipes using syn_vector and encoded design names using one hot encoding

To find the most optimal values of thetas, we use mean square error as the cost function.

$$Costfunction = J = \frac{1}{n}\sum_{i=0}^{n}(h_\theta(x^i) - y^i)^2 \tag{3}$$

We iterate over this cost function multiple times using gradient descent. This gradient descent is performed for all thetas.

$$\theta_0 = \theta_0 - \alpha * \frac{dJ}{d\theta_0} = \theta_0 := \theta_0 - \alpha * \frac{1}{m}\sum_{i=0}^{m}(h_\theta(x^i) - y^i) * x_0^i \tag{4}$$

$$\theta_n = \theta_n - \alpha * \frac{dJ}{d\theta_n} = \theta_n := \theta_n - \alpha * \frac{1}{m}\sum_{i=0}^{m}(h_\theta(x^i) - y^i) * x_n^i \tag{5}$$

To predict the delay of unseen data (test data), we use the final theta values in the hypothesis function.

4.3 Neural Network based regression

We perform the same experiments by varying the inputs as given in the Linear Regression section above. We finally go with a combination of 65 inputs (4 parameters : AND gates, NOT gates, IP length, Area. 19 synthesis recipe vectors. 42 one hot encoded design names). We use a 5 hidden layered network. Each hidden layer uses 2048 neurons. We have implemented a simple *Backpropagation* algorithm, on an *MSE Loss*, using an *Adam Optimizer* updation. In many of our initial trials we faced vanishing gradient problem where the gradients and weights were tending to 0 and as a result the network was reporting a constant output. We needed numerous fixes including addition of *Kaiming weight initialization*, addition of *Batch Normalization*, tweaking hyperparameters and observing individual layer outputs to trace the problem to resolve it. The hyperparameters like learning rate, number of epochs, optimizer coefficients were finalized upon many training iterations. Figure 2 shows our NN architecture.

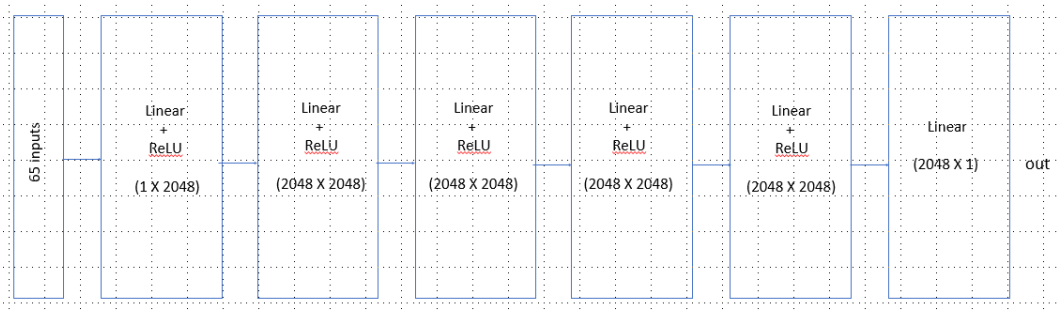


Figure 2: Neural Network architecture

4.4 Graph Convolution Network

The dataset provides graphical representation of the data in the pytorch files. Reading the data from these processed files is challenging because of the logistic requirements that the files are in an earlier version of pytorch and needed specific support from packages that are not easy to setup. Specifically, we need pytorch 1.7.1 with cuda 10.1. Apart from this, pytorch geometric has to be setup, which in turn needs pytorch-scatter, pytorch-sparse and pytorch-cluster with specific versions and python 3.8 (only) support. This specific environment took us some time to figure out. We also couldn't run this on datahub since it requires changing the default cuda version.

Once the issues with pytorch were resolved, we started implementing the graph convolution network. The network consists of 2 layers of GNN that expect 256 x256x3 dimensional adjacency matrices, three convolutional layers that take synthesis vector as an input. These three layers have distinct kernel sizes that we used. The objective here was to find the association between the various parameters in the synthesis recipe. The synthesis vector represents the synthesis recipe in a 20 element vector form. Once we obtain the results of the GCN and the three convolution layers, we add the outputs and convert the problem into a linear regression problem. The model now uses a set of 32 linear layers with ReLU non-linearity and ultimately a linear layer with 1 element at the output. The model is represented in Fig 3.

The code to process the pytorch files is also very involving. The data is in the form of .pt that have been zipped. So, to use them we unzip files and store the data in a temporary buffer for each batch. The size of the data makes it really hard to train the network especially without a GPU. We constantly ran into out of memory issues. To fix this, we reduced the size of data we were training on. Out of the 1500 synthesis recipes, we picked out 500 recipes for each design and then trained the model. This of course reduced the accuracy, but each epoch now took around 4 hours instead of 15+ hours it was taking earlier. We present our results for GCN based on a training for 10 epochs of this network.

To train the model, we used the mean squared error as the loss function and used adam optimizer with learning rate between 0.001 and 0.01. Another added functionality that we added to counteract the vanishing gradient problem was to use the scheduler ReduceLROnPlateau. Using the scheduler helped us decrease the problem of vanishing gradient to some extent. The code for the GCN was

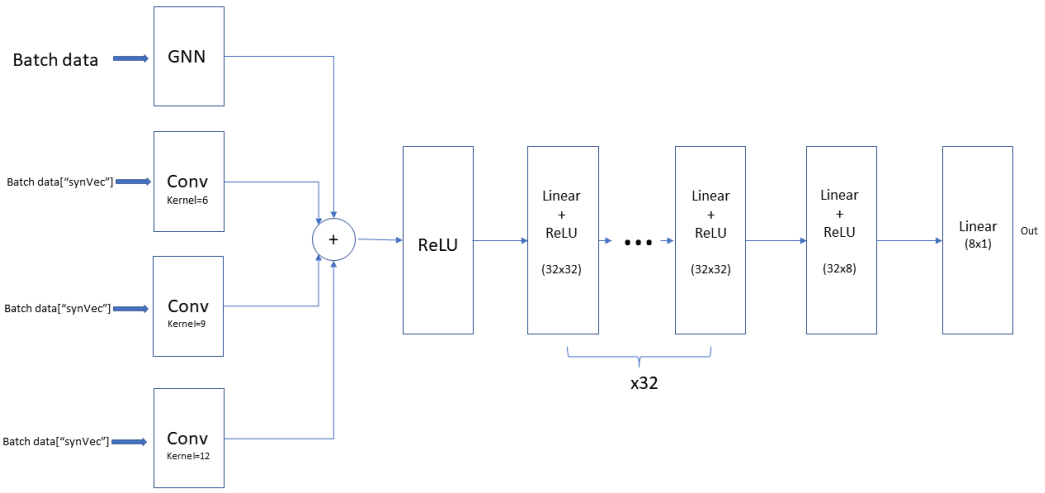


Figure 3: GCN architecture

inspired from the architecture that the authors used. We also used Xavier uniform initialization for the NN layers.

5 Results

5.1 Results of Linear Regression

Fig. 4(a) given below depicts the predicted and expected outputs for the test datapoints using 4 input features. We defined an accuracy function to check the delay outputs lying within 30% of the true value. It was observed that the model gives 17.26% accuracy

Fig. 4(b) given below depicts the predicted and expected outputs for the test datapoints using 5 input features with recipes depicted as syn vectors. We defined an accuracy function to check the delay outputs lying within 30% of the true value. It was observed that the model gives 18.26% accuracy.

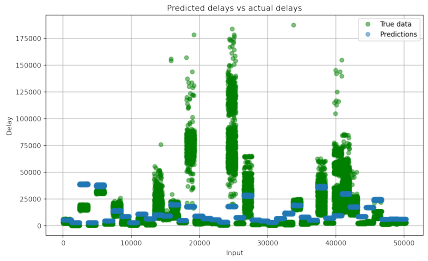
Fig. 4(c) given below depicts the predicted and expected outputs for the test datapoints using 6 input features with recipes depicted as syn vectors and design names encoded using one hot encoding. We defined an accuracy function to check the delay outputs lying within 30% of the true value. It was observed that the model gives 66.69% accuracy.

Fig. 4(d) given below depicts the predicted and expected outputs for the test datapoints using 5 input features with recipes encoded using one hot encoding. We defined an accuracy function to check the delay outputs lying within 30% of the true value. It was observed that the model gives 35.13% accuracy

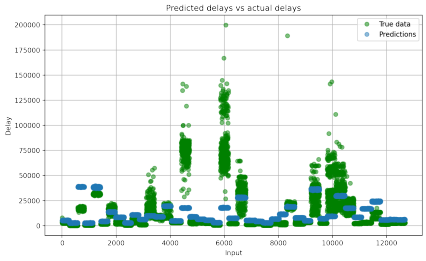
Fig. 5 shows the cost function vs number of iterations. We can see that the curve flattens out, which indicates convergence.

5.2 Results of NN

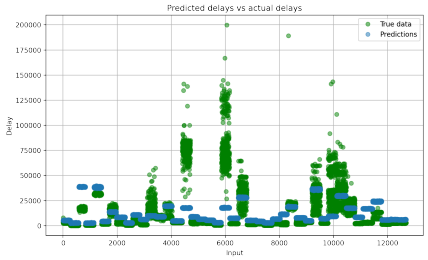
After training the network detailed in *Section 4.3* for 200 epochs, the lowest loss we get is 0.101. We define accuracy as the percentage of predicted outputs within 10ps of the expected outputs. Hence the calculated accuracy comes out to be 75%. Figure 6a shows the plot of difference between actual and predicted delays, against the row number of input in the dataset. It is clearly seen that difference is 0 for maximum points. Figure 6b shows the reduction in MSE loss as we progress in our epochs.



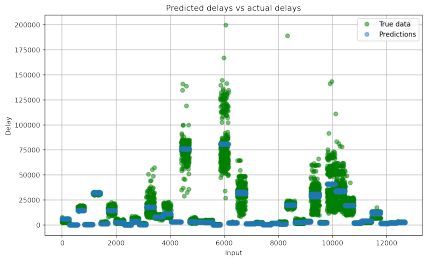
(a) Predicted delays vs actual delays 4 parameters



(b) Convergence of gradient descent algorithm 5 parameters



(c) Predicted delays vs actual delays 5 parameters



(d) Predicted delays vs actual delays 6 parameters

Figure 4: Results of Linear Regression

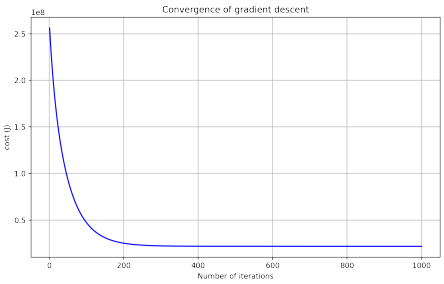


Figure 5: Convergence of gradient descent

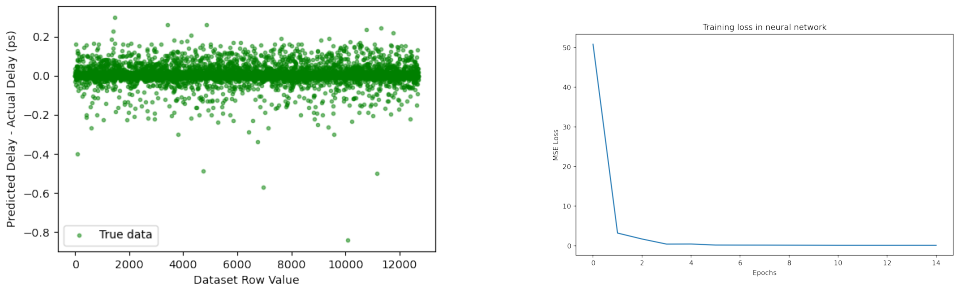
5.3 Results of GCN

After training the network for 10 epochs, we get the predicted vs actual delay for a few of the designs in our dataset as indicated in Fig 8. It indicates how closely our model is able to predict the data corresponding to the expected values. Fig. 7 shows the learning rate decreasing with the number of epochs as well.

From the image above it is clearly seen that the model is very well able to predict the values of the designs in similar range as the actual values in the test dataset.

6 Discussion

From our analysis of the three models used, vanishing gradients was a recurring problem that occured. To mend it, we changed the representation of our data. Instead of using the number of synthesis recipes, we used synthesis vectors that the authors had in the pytorch files. This brought to light a very interesting fact that the inputs of the linear regression model must be uncorrelated. We rectified this issue and vanishing gradient issue was overcome in the subsequent runs. We also note that our predictions are with a 10ps window of the real results for 80% of the test datapoints which is a



(a) Delay prediction with neural network (b) Training loss in neural network

Figure 6: Results of Neural Network

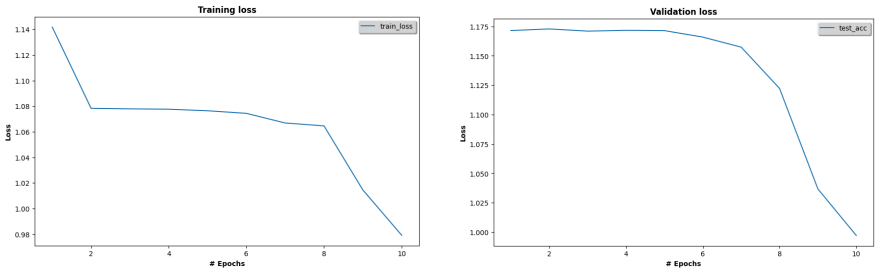


Figure 7: Training and validation loss values changing with each epoch

good result. To put this in perspective, for someone doing a synthesis run on a huge design can now successfully know within a 10ps window, the timing that their design will meet. If it is not acceptable, then the design should be changed immediately. This saves countless hours of actual synthesis runs for the design.

7 Individual contribution and Github repository

All members unanimously state that they contributed equally towards this project.

ECE-228 Final Project Repository

Acknowledgments

We are thankful to the Professor Yuanyuan Shi for giving us the opportunity to work towards this project. We thank our TA, Tawaana, who helped us narrow down our issues with vanishing gradients. She pointed us in right direction that helped us achieve our goal. Lastly, we want to thank Animesh Chowdhary, the author of [1]. He helped us navigate the dataset OpenABC-D. Without his help, it would have been very difficult to work with the dataset.

References

[1] Animesh Basak Chowdhury, Benjamin Tan, Ramesh Karri, and Siddharth Garg. (2021) *Openabc-d: A large-scale dataset for machine learning guided integrated circuit synthesis.* arXiv preprint arXiv:2110.11292, 2021a.

[2] Bower, J.M. & Beeman, D. (1995) *The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System.* New York: TELOS/Springer–Verlag.

[3] Hasselmo, M.E., Schnell, E. & Barkai, E. (1995) Dynamics of learning and recall at excitatory recurrent synapses and cholinergic modulation in rat hippocampal region CA3. *Journal of Neuroscience* **15**(7):5249-5262.

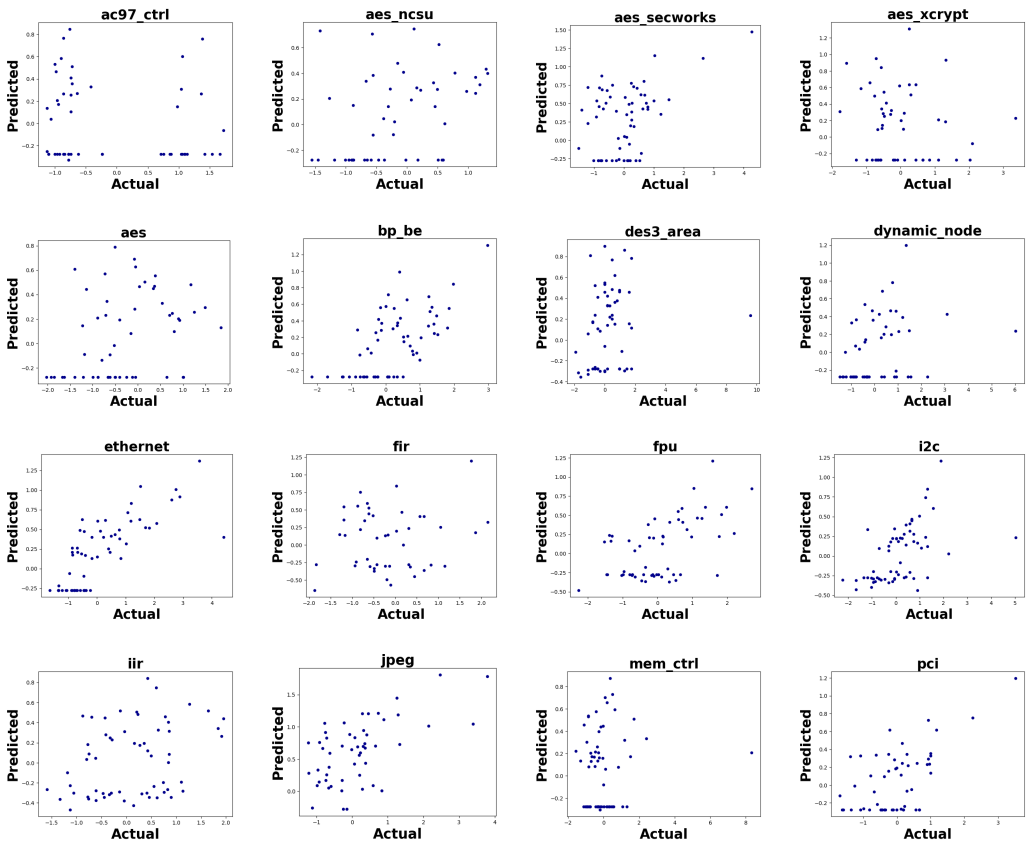


Figure 8: GCN results for predicted vs actual values of 16 different designs

[4] Animesh Basak Chowdhury, Benjamin Tan, Ramesh Karri, and Siddharth Garg. (2022) *Too Big to Fail? Active Few-shot Learning Guided Logic Synthesis.* arXiv preprint arXiv:2204.02368v1

[5] C. Yu et al. (2018) *Developing synthesis flows without human knowledge* ACM/IEEE Design Automation Conference, pp. 1–6.

[6] A. Hosny et al. (2020) *DRiLLS: Deep reinforcement learning for logic synthesis* IEEE Asia South Pacific Design Automation Conference, pp. 581–586.

[7] K. Zhu et al. (2020) *Exploring logic optimizations with reinforcement learning and graph convolutional network* ACM/IEEE Workshop on Machine Learning for CAD pp. 145–150.

[8] Y. V. Peruvemba et al. (2021) *Rl-guided runtime-constrained heuristic exploration for logic synthesis* ACM/IEEE International Conference On Computer Aided Design, pp. 1–9.

[9] A. Mirhoseini et al. (2021) *A graph placement methodology for fast chip design* Nature, vol. 594, no. 7862, pp. 207–212.

[10] W. L. Neto et al. (2019) *LSOracle: a logic synthesis framework driven by artificial intelligence: Invited paper* ACM/IEEE International Conference on Computer-Aided Design pp. 1–6.

[11] Y. Lin et al. (2020) *DREAMplace: Deep Learning Toolkit-Enabled GPU Acceleration for Modern VLSI Placement* IEEE Transactions Computer-Aided Design of Integrated Circuits and Systems, vol. 40, no. 4, pp. 748–761