A Graph Neural Network Surrogate Model for hls4ml

Dennis Plotnikov[1 2], Benjamin Hawks[1], and Nhan V. Tran[1]

[1]*Fermi National Accelerator Laboratory, Kirk & Pine St., Batavia, IL 60510, USA*
[2]*Department of Physics and Astronomy, The Johns Hopkins University, 3400 N. Charles St., Baltimore, MD 21218, USA*

July 2024

Abstract

Recent advancements in use of machine learning (ML) techniques on field-programmable gate arrays (FPGAs) have allowed for the implementation of embedded neural networks with extremely low latency. This is invaluable for particle detectors at the Large Hadron Collider, where latency and used area are strictly bounded. The hls4ml framework is a procedure that converts trained ML model software to a synthesis result to can be used on an FPGA. However, running the pipeline is a time-consuming procedure, and there is a strong risk of failure. In particular, it may not be possible to successfully convert a model into a synthesis result, or the resource consumption of the model may exceed the resources of the target FPGA. To aid with this development, we introduce wa-hls4ml, a surrogate model using a graph neural network to emulate the structure of the source models. The goal is to estimate the chance of success and resource consumption of a given model when passed through the hls4ml pipeline, without needing to run the pipeline.

1 Background

Physicists have been attempting to probe the smallest length scales, to attempt to make new fundamental discoveries about the universe. Experiments on this frontier require extreme data processing capabilities. In particular, at the Large Hadron Collider (LHC), the most powerful particle collider in the world, the rate of bunches of protons passing detectors is $\sim$ 40 MHz [1]. Storing data at this rate would be intractable, meaning that a vast majority of the data must be thrown out at the level of the detector.

Both the ATLAS and CMS detectors at the LHC are designed with a multilevel trigger system which only stores events containing interesting physical phenomena. The trigger system reduces the data volume down from 40 MHz to $\sim$ 1 kHz. The first step is a Level 1 (L1) trigger, which must reduce the data rate to $\sim$ 100 kHz. This L1 trigger is followed by the High-level trigger (HLT), which further reduces the data stream down another two orders of magnitude [1][2]. Both of these trigger layers choose which events should be discarded in real time. The HLT is able to make use of sophisticated and relatively long-running algorithms, whereas the L1 trigger has much stronger constraints on its algorithms. Since the throughput is stepped down so strongly in the L1 step, it is necessary to have a system that can effectively make a correct decision about the importance of an event, all in about 10 nanoseconds. This strong upper bound on potential latency of the trigger algorithm limits the sort of triggering decisions that can be made from a real time embedded system. Such algorithms, however, may not necessarily succeed in their task, and may either fail to discard an event containing only background (which slows down the final data analysis steps), or wrongly discard an event containing interesting physical data.

Machine learning (ML) has been widely applica-

ble for LHC event reconstruction [3] [4]. ML techniques are well-suited to performing analyses post-hoc, where the large time requirement of sophisticated models is not an issue. However, it is much more challenging to implement such a model for use in the L1 trigger. To achieve this, one can convert the ML model into a form that can be synthesized to hardware level, either with field-programmable gate arrays (FPGAs), or with application-specific integrated circuits (ASICs). This is achievable by training an ML model in the standard way, and then converting it to a hardware description language through the hls4ml process [5]. The hls4ml process is a pipeline which can convert a machine learning model to high-level synthesis (HLS) code, which can then be converted into a hardware description language like VHDL [6].

Using hls4ml, various kinds of ML models that can be converted onto FPGAs [7] [8] have been implemented. Additionally, the prototype "smart pixel" system uses the hls4ml pipeline to create designs for ASICs that can be embedded in the detector pixel itself, allowing for extremely space efficient and low-latency triggering, while also retaining the power of an ML-based solution [9].

However, this pipeline is not guaranteed to succeed. In particular, there are two modes of failure in which the pipeline cannot effectively be deployed onto an FPGA (or ASIC). Firstly, it is possible for the HLS synthesis step to fail, meaning that no meaningful hardware description code is produced. Secondly, even if HLS synthesis succeeds, the model may consume more resources than are available on the FPGA, or may violate a constraint necessary for the use case that the FPGA is being developed for. This raises an issue, as the time to train a model is significantly shorter than the time it takes to run the HLS synthesis, which in turn is a smaller time sink than the time required to perform the actual hardware synthesis. Thus, it would be advantageous to have a relatively fast estimation procedure to predict the success chance and resource consumption of a given model, without actually running the full hls4ml pipeline.

2 General Task and Framework

Our goal is to create a system, by which a given model can be fed in as input, and reasonable estimates of the synthesis resource usages can be given within a much shorter time than actually running the synthesis procedure. The creation of such "surrogate models" is common in fields of systems and industrial modeling. [10] In our case, the creation of a surrogate model largely involves finding a hidden transfer function which takes certain parameters of a model, and outputs the success chance and required resources of the resulting FPGA. The output does not have to be precise, and is only required to be a sufficiently close estimate. As can be seen, this is a good candidate problem for machine learning. ML techniques have been successful for surrogate models in general. [11] Additionally, ML techniques have been effectively applied to tasks within hardware-level design problems, such as PCB routing. [12] We have therefore implemented a neural network, named "wa-hls4ml", for the purpose of creating a fast and effective procedure for estimating the above parameters.

While a traditional multi-layer perceptron (MLP) neural network is a good approach for dealing with models of one type, there is a challenge when attempting to estimate models with varying numbers of layers, and varying connections between layers. It would be unfeasible to attempt to have individual MLP input features for every conceivable input type (in our case, any conceivable network shape) [13]. One solution is to use a graph neural network (GNN). A graph neural network takes input features not in the form of a simple n-dimensional vector, but as a graph representing the actual underlying structure of the data [14]. Graph neural networks have been widely used in the fields of social network analysis and chemical modeling, and have also seen great success in modeling systems in high-energy physics [15] [16]. In these domains, data of varying shapes and dimensions are common and must be robustly handled. The success of GNNs in these fields prompts us to consider them the best single solution to representing the varying structures of our input models.

The GNN framework allows us to accurately map a given neural network model into a representation compatible with ML. In particular, we create a graph, with nodes in the graph representing layers of the network, and edges representing feedforward connections between them. This allows the network to represent every different type of graph structure in a way that allows the different shapes of networks to correspond one-to-one with the differences in input data.

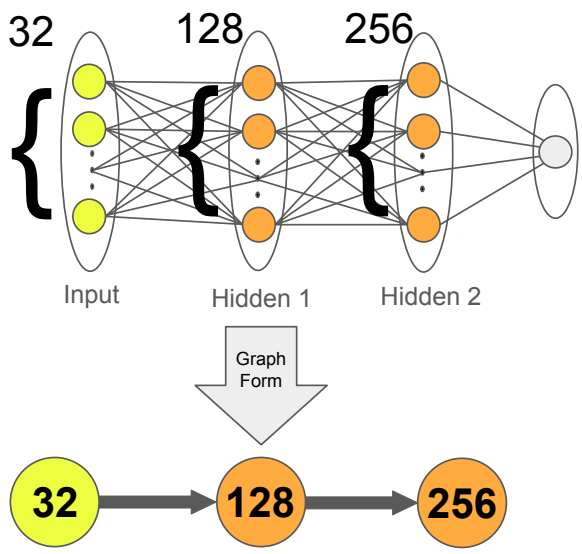

Figure 1: A simple two-hidden-layer feedforward neural network, represented as a three node directed graph.

2.1 GNN Theoretical Advantages

A GNN is expected to outperform a traditional MLP in two ways. In the first, a graph neural network, when trained on a sufficiently large and diverse dataset, is able to provide estimates on model shapes that it has not seen before (i.e. a particular graph structure that was not in its training set). In contrast, an MLP is necessarily unable to do this, as its input feature vector would have to be modified to account for the structure change, and would thus require retraining the entire model to account for this new data.

Secondly, the GNN should outcompete an MLP in terms of successful prediction on strongly heterogeneous data. Specifically, data sets which include a diverse selection of layer numbers and connections will be modeled effectively in a GNN arbitrary graph input. An MLP would need to instead have a feature vector accommodating the largest such graph, with entries of the vector being masked out if the datapoint does not contain as many layers as the largest model in the dataset. This is expected to contribute negatively to the predictive power of such a model, since it would effectively have to learn the relation between layers differently for any given pair of layers. A GNN would naturally have this represented in the data structure, allowing the connections between layers to act as inductive bias, and would not have to waste power in learning that relation.

3 Data

The datapoints used for the wa-hls4ml system are the models which can be put through the hls4ml pipeline. About 35000 models were created with varying parameters. Each of these were preinitialized with random weights, to simulate the structure of an actual neural network with that architecture. Since the particular weights themselves are not relevant to the hls4ml pipeline, this is sufficient to simulate actual models. For our sample dataset, the input neural networks contained either 1 hidden layer or 2 hidden layers each.

The models were then put into the hls4ml system, converting them through the HLS synthesis step. If the step succeeded, estimates for the resource consumption of the synthesis result (as produced by the HLS synthesis step) were then extracted. In the case of the synthesis failing, the resource estimates were all assigned a value of -1. The key parameters of the input and output of the process were then formatted into a CSV file to use for model training.

3.1 Parameters Modeled

For each layer of neural network in the input model, the number of nodes in that layer is saved as an input feature. A graph is then constructed from this data. Each node of the resultant graph is one of the layers (as previously mentioned), and is labeled with a node-level feature indicating the dimensionality of that layer in the input model. Since the output of the model is always a single feature, the output layer is not modeled.

A number of global/context features are used at the overall graph level, to indicate traits of the model's synthesis process as a whole. Specifically, the reuse factor of the model when put through synthesis, as well as the fixed-point precision are saved as these features. Additionally, the optimization strategy (resource-optimizing or latency-optimizing) is saved as a 1-hot encoded categorical feature.

The input data is generated by iterating over each of these features, and creating corresponding models

to allow the input data to effectively cover the search space. Models are generated for each number of dimensions between 32 and 256, in increments of 32. For each number of dimensions, every combination of fixed-point precision (from 2 to 16 in increments of 2) and reuse factor (increments vary based on the kind of model being checked). The type of synthesis strategy is not subject to this search space covering in the same way, since batches must be run separately for a different strategy choice.

For the corresponding output datapoints, the features used are estimates for the latency of the output, the interval between parallel steps, the number of flip-flops (FFs) used, number of lookup tables (LUTs) used, the consumption of the BRAM, and the number of DSP modules required. Additionally, the success or failure of the synthesis is saved as a binary 1/0 feature.

	Input	**Output**
Node-level	Nodes per layer	
Edge-level	Adjacency list	
Graph-level	Strategy Reuse factor Precision	Latency Interval FFs LUTs BRAM DSP Synth success

Table 1: All parameters currently modeled for this analysis.

4 Methods

In order to test the learning effectiveness of a GNN solution to this learning task, we have implemented a standard feedforward MLP to also attempt to model this problem. For the cases modeled, the MLP does not account for the graph structure of the data, and instead has a standard flattened feature vector. This feature vector has a separate feature for each dimension possible in the model. In our case, since we tested models with either one or two hidden layers, the feature vector has three rows corresponding to the three layer dimensions of the model.

Two activation functions are used in our neural networks. The first is the Leaky ReLU (LReLU) function, as follows: [17]

$$\mathrm{LReLU}(x) = \begin{cases} x & x \geq 0 \\ -0.01x & x < 0 \end{cases} \tag{1}$$

This allows us to utilize the experimentally validated approach of ReLU activation, while also avoiding the "dying ReLU" problem of vanishing gradients in standard ReLU. This activation is also used internally in attention computation. Similarly, we also make use of the Exponential Linear Unit (ELU), which is as follows: [18] [17]

$$\mathrm{ELU}(x) = \begin{cases} x & x > 0 \\ \exp(x) - 1 & x \leq 0 \end{cases} \tag{2}$$

ELU, just as with LReLU, retains much of the benefit of the ReLU function, while not being subject to the vanishing gradient issue. Furthermore, since it tends towards a plateau in the $x \to -\infty$ limit, and exhibits stronger nonlinearity, ELU activation can allow for a stronger convergence over only using LReLU.

4.1 Graph Neural Network Structure

The GNN structure chosen allows for arbitrary directed graph input, with each node and edge having some fixed number of features. For the purposes of our current tests, no edge features were given (besides the implicit edge feature of the adjacency list). We handle global (per-graph) features by incorporating them directly into the node features. During preprocessing, the global features are concatenated onto every per-node feature vector of the corresponding graph. This allows the global features to influence the entire rest of the GNN training.

Each of these independent feature vectors is passed through a shallow (one hidden layer) MLP layer, to translate each feature of the graph into a higher-dimensional embedding.

The per-node and per-edge feature vectors are then fed into a Graph Attention Network (GAT) layer. Specifically, the network used is a GATv2 layer.

This is a modified version of a graph attention procedure, in which an attention mechanism is applied during training, in order to weight edges which are more or less important to the results [19]. Since our input models may have layers that matter significantly more or less than others in terms of resource consumption (e.g. skip-layer connections), the attention mechanism will be able to handle this incongruity by learning on which edges are more or less valuable to the end result. The layer applies the following operator: [19] [20]

$$\mathbf{x}'_i = \sum_{j \in \mathcal{N}(i) \cup \{i\}} \alpha_{i,j} \mathbf{\Theta}_t \mathbf{x}_j \tag{3}$$

Where $\mathbf{\Theta}_t$ are trainable parameters, and $\alpha_{i,j}$ are the attention weights, calculated with

$$\frac{\exp\left(\mathbf{a}^\top \text{LReLU}\left(\mathbf{\Theta}_s \mathbf{x}_i + \mathbf{\Theta}_t \mathbf{x}_j + \mathbf{\Theta}_e \mathbf{e}_{i,j}\right)\right)}{\sum_{k \in \mathcal{N}(i) \cup \{i\}} \exp\left(\mathbf{a}^\top \text{LReLU}\left(\mathbf{\Theta}_s \mathbf{x}_i + \mathbf{\Theta}_t \mathbf{x}_k + \mathbf{\Theta}_e \mathbf{e}_{i,k}\right)\right)} \tag{4}$$

The GATv2 layer produces a new set of per-node embeddings, which are fed into the next layer. As a consequence of its operation, the GATv2 layer also produces an attention weight per-edge. As mentioned above, this value has great relevance for ensuring that potentially less relevant connections between layers are properly accounted for. These weights are thus extracted, and used meaningfully in further parts of the model.

The newly created per-node feature vector is then inputted into a graph convolution layer. This layer performs a non-trainable message-passing procedure, followed by a trainable single-layer linear transformation, on the input graph, producing a new set of per-node feature vectors.

The procedure is repeated again, with a second GATv2 layer. The edge attentions generated by the previous layer are fed into this layer as per-edge features, to allow the earlier attention information to inform the second layer. The node features are passed through this layer, and the resulting features are passed through a second non-trainable graph convolution layer. The resultant vector is normalized per-graph, which has been shown to be an effective means of improving convergence time and accuracy in GNNs [21].

The edge attentions produced by this second layer are concatenated onto the per-edge feature vector, essentially adding the attention output as an edge feature. The per-edge features are passed through a shallow MLP.

The node features outputted from the message-passing layer are then passed through a global pooling layer, using a combination of sum-pooling and multiplicative pooling. This reduces these features to a single global feature vector representing all node features. The same is done independently with the edge features with additive pooling, producing a single edge feature vector. The node feature vector and edge feature vector are then concatenated into a single large vector. This vector is then passed through another shallow MLP layer to produce a final output value. For the regression tasks, this value is used as-is. For the binary classification task (i.e. the synthesis success chance), the resultant value is treated as a logit and is passed through a sigmoid function.

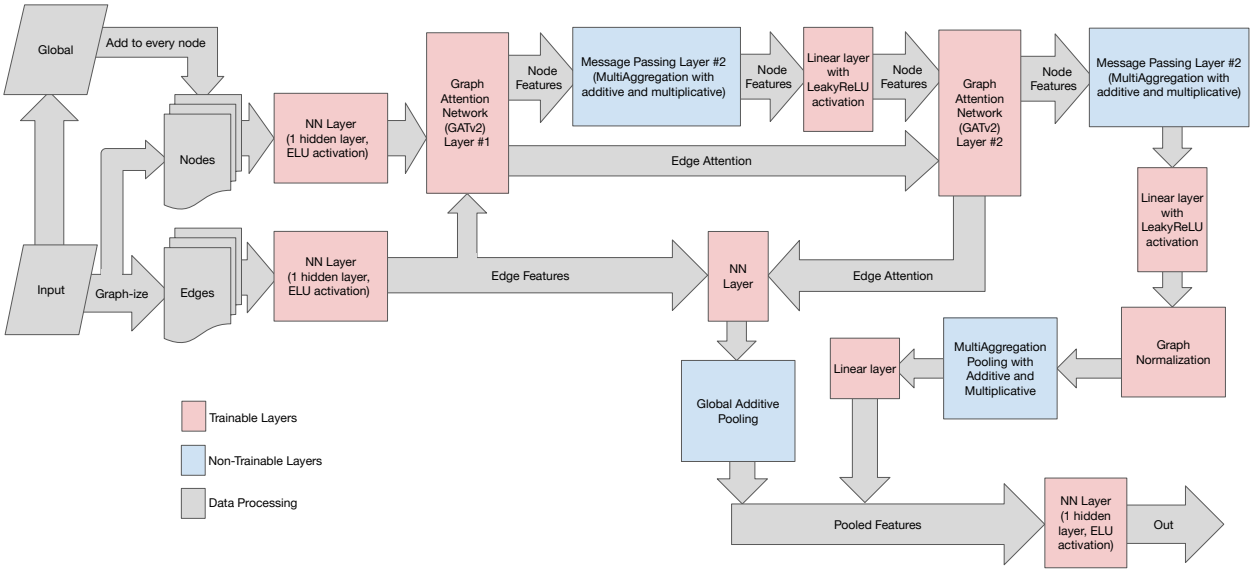

Figure 2: The overall structure of the GNN, with two GATv2 layers.

4.2 Comparison MLP Structure

The MLP used for comparison is a standard feed-forward neural network. The MLP has 5 hidden layers, which can be seen to approximately match the number of hidden layers in the GNN structure. The layers have a 20% dropout applied to them, to attempt to reduce overfitting, and especially to prevent overfitting to one or another structure of input model.

4.3 Losses and Activations

For classification training, the binary cross-entropy function is used. The output value, as described above, is passed through a sigmoid function, allowing the result value to be used as a confidence percentage of the synthesis succeeding.

The loss function used for regression training is the log cosh loss. Empirical testing showed that the L^2 norm produced effective results for some of the features, but failed to be effective for the BRAM and DSP parameters, so for the purposes of the comparisons here, all features were trained with the log cosh loss.

4.4 Implementation

The machine learning methods were implemented using the Pytorch library of machine learning tools. All GNN implementation components utilize the Pytorch Geometric library [20]. Data is read in from CSV files containing information about the graph structure, as well as all node values and global feature values.

For the GNN, the data are then converted into a graph format. For each data point, we create a node tensor, an edge tensor, and a global features vector. The node feature tensor is of the form $N \times F_N$, with N being the number of nodes in the data point (i.e. the number of layers in the input model), and F_N being the number of per-node features (for our tests here, this is just 1). The edge tensor is of the form $E \times (F_E + 2)$, with E being the number of edges (i.e. layer connections in the input model), and F_E being the number of edge features. The extra 2 comes from the representation of the graph, in the form of an adjacency-list, with the indices of the source and destination nodes being stored per-edge.

During preprocessing, the global feature vector is concatenated onto each node tensor, creating a tensor

of the form $N \times (F_N + F_G)$, where F_G is the number of global features. The adjacency list of the edge tensor is broken off, and self-loops are adding, creating tensors of $2 \times E$ and $(E+N) \times F_e$ respectively. These tensors are fed into the GNN layers.

For the GAT v2 model, we use the prebuilt PyTorch Geometric implementation, called `GATv2Conv`. The standard graph convolution layer which follows is implemented using the `SimpleConv` module, which is an implementation of a generalized non-trainable message-passing layer. We use the builtin `MultiAggregation` module for both message passing and to pool node features, combining multiplicative `MulAggregation` with mean `SumAggregation`. The two aggregations are combined via the `'proj'` combination method, which uses a trainable linear projection to combine the two into one feature vector. The `global_pool_add` built-in function was used to pool edge features.

The graph normalization layer is implemented with the prebuilt `GraphNorm` implementation [21]. This allows for accelerated training, and in testing appeared to additionally stabilize the final convergences.

For the MLP comparison model, the node features are instead just represented as individual numeric features in one flattened global feature vector per-event. In our testing case, this means that three dimensions of the feature vector are devoted to layer dimensions. Since some models do not have the same shape as others, nodes/hidden layers which are "missing" in some models are simply masked out with a value of -1. In particular, the middle dimension is masked out for all 1-layer models. This is then used as input for a 5 hidden layer MLP model. Both the Leaky ReLU and ELU functions are used as activation functions, with ELU being used as the first and last activations, and LReLU being used for the rest. This closely matches the use of losses in the training of the GNN, especially since the attention network structure makes use of the LReLU function internally.

5 Results

The model is tested using an 70-20-10 train-test-validation split. The random seed is consistant between the two model types, such that a direct comparison can be made.

5.1 Classification

For the binary classification, the confusion matrix is calculated, using a 50% result from the model as a threshold for a "success" classification.

For the classification task on our testing data, the precision of the GNN is about 99.71%, and the recall is 100% with respect to successes. For the MLP, the precision is about 99.69%, and the recall is again 100%. The confusion matrices are as follows:

GNN:

	Pred. Failure	Pred. Success
GT Failure	346	20
GT Success	0	6823

MLP:

	Pred. Failure	Pred. Success
GT Failure	345	21
GT Success	0	6823

Table 2: Comparison of the confusion matrices for the binary classification task, i.e. synthesis success or failure

5.2 Regression

The root-mean squared error (RMSE) is calculated for all regression values, as a means of assessing error rate. This result is calculated, as mentioned above, by first passing data points through the trained classification model, taking only those that score above 50% confidence in synthesis success, and performing the regression model on those data. The residuals for the output features are distributed as follows:

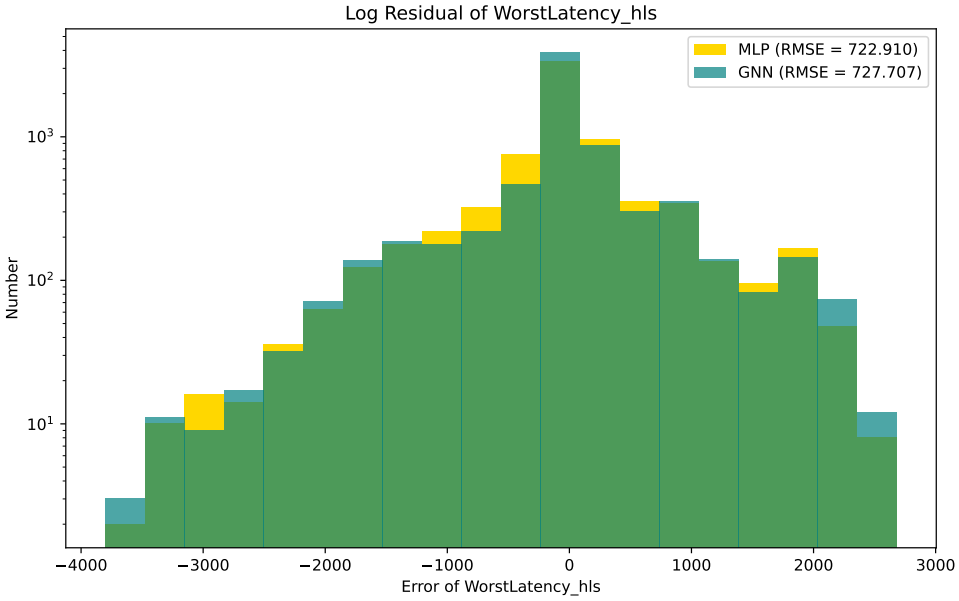

Figure 3: Comparison of residuals for the latency feature

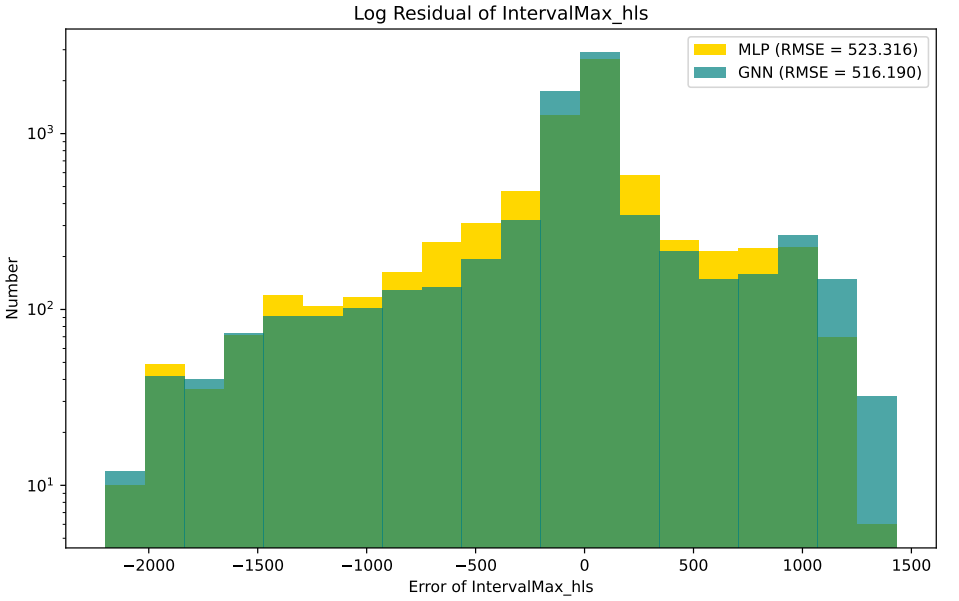

Figure 4: Comparison of residuals for the interval feature

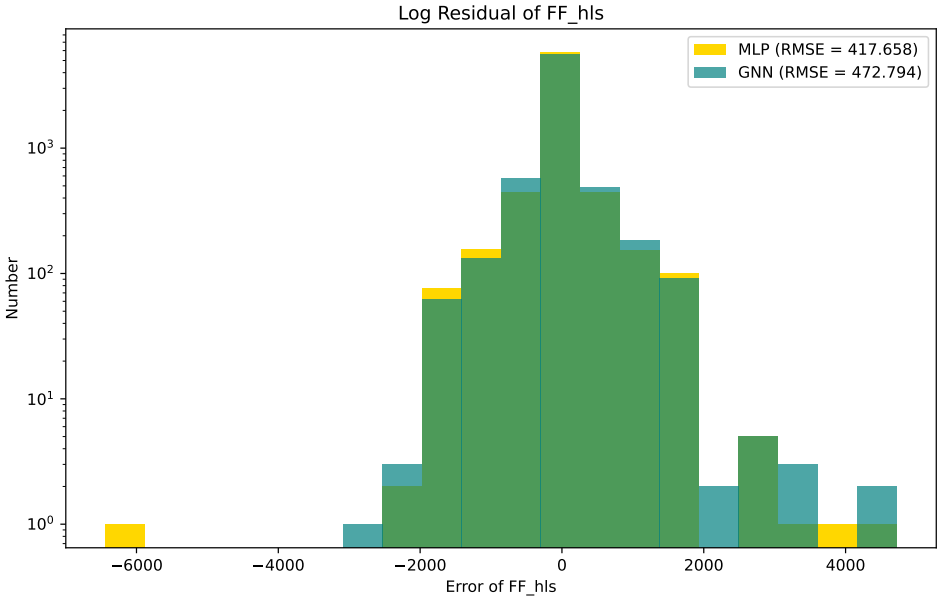

Figure 5: Comparison of residuals for the flip-flops feature

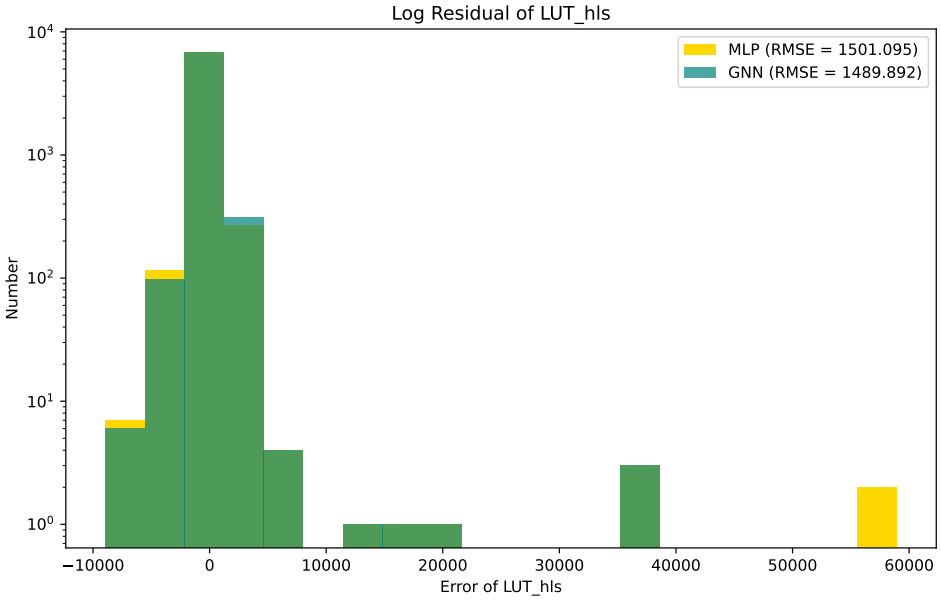

Figure 6: Comparison of residuals for the lookup tables feature

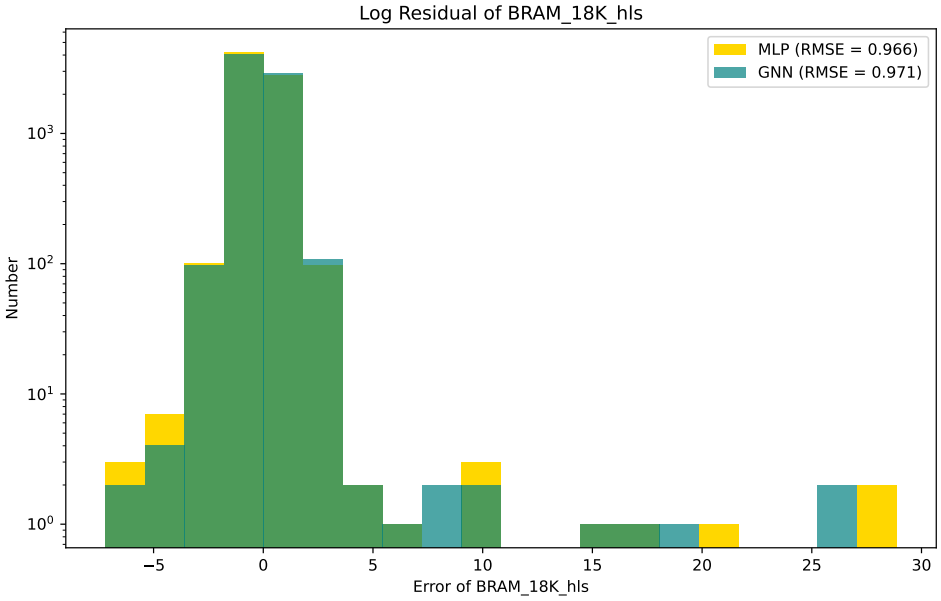

Figure 7: Comparison of residuals for the BRAM feature

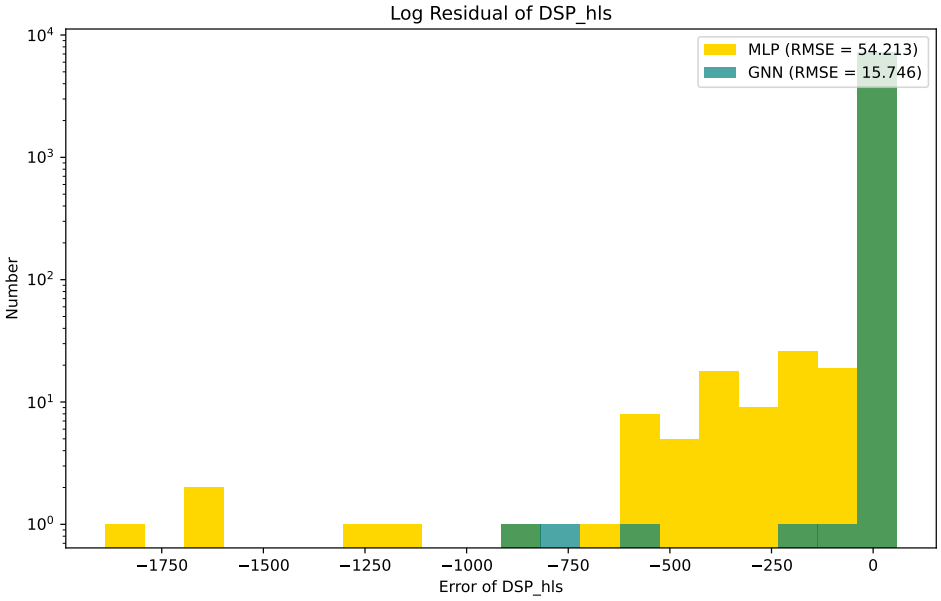

Figure 8: Comparison of residuals for the DSP feature

6 Discussion

The effectiveness of the GNN model evidently varies by parameter. The DSP, LUTs, and interval parameters were more effectively modeled by the GNN. The latency, flip-flops, and BRAM were predicted somewhat worse by the GNN compared to the control MLP.

The latency, interval, LUTs, and BRAM features were modeled only negligibly differently between the two models, with the differences in their RMS error being within $\sim 1\%$. This difference can largely be attributed to the specifics of the data, and are not likely to reflect any meaningful differences between the predictive power of the models. The DSP and flip-flops parameters showed bigger differences, indicating actual structural differences.

On this dataset, the DSP feature, which failed to be meaningfully predicted by the MLP even after hyperparameter optimization, was predicted by the GNN, with a fourfold decrease in RMS error. This shows that the more sophisticated GNN representation can succeed against a simpler MLP. This failure to converge is expected to improve with a larger dataset, as this effect may be caused by the MLP failing to escape from a local minimum.

The flip-flops appear to be estimated relatively weakly by the GNN, compared to the control MLP. This may be the result of those parameters correlating strongly between a global feature (e.g. precision) and the node-level feature, or in relation to their dependence on bit operations, an input feature we did not use for this analysis.

The classification task seems to have performed very well on both models, and differences between their performance appear to be no bigger than noise. However, this cannot necessarily be generalized, since the amount of failures in the dataset is dwarfed by the number of successes. In particular, the 100% recall for success may be caused by insufficiently many failure samples to test on. More data is needed to confirm the effectiveness of the models in the failure regime.

With the current limited set of data (only 1-hidden-layer and 2-hidden-layer models), the advantages of the graph representation are limited. More groundtruth data is required to achieve a larger breadth of model types. In particular, deep models and models with unusual connections could allow the GNN to learn much deeper information about the effects of different edges themselves.

The GNN is able to meet or exceed the performance of a traditional MLP in 5 out of the 6 tested regression tasks, which indicates promise on its ability to generalize model structure. However, the current data is insufficient, both in quantity and breadth, for making larger claims about the effectiveness of the GNN as a means of providing effective estimates for a generic input model. More testing needs to be performed to make certain that the success of this model on the testing dataset can be extrapolated to deeper and more complex architectures than in training data.

Further improvements on the GNN architecture, such as making global features represented in the message-passing layers themselves beyond simply being treated as node features, could potentially allow the model to more effectively understand the structure of input models as a whole.

7 Conclusion

We have introduced a machine-learning based surrogate model for hls4ml. The model is equipped with graph neural network architecture, to accurately emulate the structure of the input data, which themselves are neural network models. This model can be used to significantly decrease the turnaround time of an hls4ml production, by providing insight into which trained models will and will not be useable on a target FPGA/ASIC, rather than requiring the time-consuming synthesis process to be run to gain that same information. While the accuracy of the GNN model is still lacking in certain areas, this represents a distinct improvement in the effectiveness of hls4ml for design requiring fast turnaround, such as in the case of automated parameter testing. The use of a graph neural network architecture to accurately represent input neural networks could see utilization in other situations where the structure of machine learning models needs to be analyzed.

8 Code and Data Availability

The version of the code that was used for the present analysis, as well as graphing code for the

histograms (used to generate all of figures 3-8), can be found at `https://github.com/Dendendelen/wa-hls4ml-report-july2024`.The current version of the model (potentially updated since the version used for this analysis) can be found at `https://github.com/Dendendelen/wa-hls4ml`. The data (input models) used in this analysis are not yet publicly available while we work to refine the dataset.

9 Acknowledgements

This manuscript has been authored by Fermi Research Alliance, LLC under Contract No. DE-AC02-07CH11359 with the U.S. Department of Energy, Office of Science, Office of High Energy Physics.

This work was supported in part by the U.S. Department of Energy, Office of Science, Office of Workforce Development for Teachers and Scientists (WDTS) under the Science Undergraduate Laboratory Internships Program (SULI).

This work was supported in part by National Science Foundation (NSF) awards CNS-1730158, ACI-1540112, ACI-1541349, OAC-1826967, OAC-2112167, CNS-2100237, CNS-2120019, the University of California Office of the President, and the University of California San Diego's California Institute for Telecommunications and Information Technology/Qualcomm Institute. Thanks to CENIC for the 100Gbps networks.

References

[1] ATLAS Collaboration, Journal of Instrumentation **19**, P06029 (2024), arXiv:2401.06630 [hep-ex].

[2] CMS Collaboration, Journal of Instrumentation **15**, P10017 (2020), arXiv:2006.10165 [hep-ex, physics:physics].

[3] L.-G. Gagnon, Journal of Instrumentation **17**, C02026 (2022).

[4] M. J. Fenton et al., Communications Physics **7**, 139 (2024), arXiv:2309.01886 [hep-ex, physics:hep-ph].

[5] F. Fahim et al., (2021), arXiv: 2103.05579.

[6] J. Duarte et al., Journal of Instrumentation **13**, P07027 (2018), arXiv:1804.06913 [hep-ex, physics:physics, stat].

[7] J. Campos et al., ACM Trans. Reconfigurable Technol. Syst. **17**, 36:1 (2024).

[8] C. N. Coelho Jr. et al., Nature Machine Intelligence **3**, 675 (2021), arXiv:2006.10159 [hep-ex, physics:physics].

[9] B. Parpillon et al., (2024), arXiv: 2406.14860.

[10] R. Alizadeh, J. K. Allen, and F. Mistree, Research in Engineering Design **31**, 275 (2020).

[11] C. Angione, E. Silverman, and E. Yaneske, PLOS ONE **17**, e0263150 (2022).

[12] A. Plot, B. Goral, and P. Besnier, Machine Learning Techniques for Defining Routing Rules for PCB Design, in *2023 IEEE 27th Workshop on Signal and Power Integrity (SPI)*, pages 1–4, 2023, ISSN: 2835-0898.

[13] D. P. N. Nataraja and B. Ramesh, Machine Learning Algorithms for Heterogeneous Data: A Comparative Study, 2020.

[14] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, IEEE Transactions on Neural Networks **20**, 61 (2009).

[15] J. Zhou et al., Graph Neural Networks: A Review of Methods and Applications, 2021, arXiv:1812.08434 [cs, stat].

[16] H. Qu and L. Gouskos, Physical Review D **101**, 056019 (2020), arXiv:1902.08570 [hep-ex, physics:hep-ph].

[17] A. Paszke et al., PyTorch: An Imperative Style, High-Performance Deep Learning Library, 2019, arXiv:1912.01703 [cs, stat].

[18] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs), 2016, arXiv:1511.07289 [cs].

[19] S. Brody, U. Alon, and E. Yahav, How Attentive are Graph Attention Networks?, 2022, arXiv:2105.14491 [cs].

[20] M. Fey and J. E. Lenssen, Fast Graph Representation Learning with PyTorch Geometric, 2019, arXiv:1903.02428 [cs, stat].

[21] T. Cai et al., GraphNorm: A Principled Approach to Accelerating Graph Neural Network Training, 2021, arXiv:2009.03294 [cs, math, stat].