TPU runtime prediction using GNNs

Abstract

This paper navigates the challenges of training Deep Learning Models for runtime prediction, particularly with limited hardware resources. A GNN architecture with residual connections and attention is trained on the TpuGraphs dataset. Despite resource constraints, the model showcases promising performance, achieving a notable 85.7% Ordered Pair Accuracy for a specific dataset. This research sheds light on the potential of GNNs for advancing computational graph optimization, emphasizing the need for further exploration in hyperparameter tuning to optimize efficiency in DLMs.

1 Introduction

Deep Learning model (DLM) training is still limited by hardware availability and cost in today's computing environments. For effective code generation, it would be helpful to measure the models' runtime and record the amount of hardware used while they were operating. Unfortunately, even the evaluation of runtime becomes difficult due to the complexity of modern processors like deep learning accelerators (also known as "Tensor Processing Units," or "TPUs") and the complexity of DLMs. To solve these issues, this paper trains a graph neural network to create a model that predicts runtime of DLMs on TPUs by leveraging a Google dataset, TpuGraphs [1].

Prior work [2] has employed Graph Neural Networks (GNN) to assess the runtime of two types of optimization compilers methods used in tensors using a similar dataset. The subsequent sections go deeper into the details of the characteristics of the dataset, implemented model architecture, experimental setup, and evaluations. This exploration seeks to provide insights into the potential of GNNs in Deep Learning model optimization.

2 Data

A tensor program can be represented as an acyclic and directed graph. A node in a graph represents a tensor operation. Each graph contains a set of nodes a set of edges and context. The graph can process one or more tensors operations into a single output, and an edge connects an output tensor from one node to an input tensor of another node. Meanwhile context is the information related to the general model.

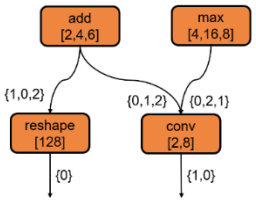

Figure 1. Example of a Graph

Each row of the dataset contains a graph tensor and its corresponding context which is a run time tensor expressed in seconds. Two distinct compiler optimizations separate the dataset. A compiler optimization is a process that looks to reduce the model run time. The first compiler optimization is “layout” where tensors are arranged in memory based on various feats like size and shape. The second one is “tile” which processes tensors by dividing them into smaller “tiles”. Tile optimization is sourced from an XLA (accelerated linear algebra) model meanwhile layout optimization is sourced from both XLA and NLP (natural language processing) models. Given that the layout method tensors can be arranged the dataset is subdivided into 2 different search strategies: the random split method, which involved randomly partitioning programs into sets, and the manual split method, which involved manually selecting the test set to minimize the subjective similarity of programs between the training and other two sets.

Layout Dataset	*Search strategy*	*Test rows*	*Train rows*	*Valid rows*
XLA	Default	8	61	7
	Random	8	69	7
NLP	Default	17	179	20
	Random	17	77	20

Table 1. Layout dataset content

TILES DATASET	*Test rows*	*Train rows*	*Valid rows*
XLA	832	5716	676

Table 2. Tiles dataset content

3 Model Design

Graph Neural Networks (GNNs) are a class of neural network architectures specifically designed to handle graph-structured data, where entities (nodes) and relationships (edges) form a complex network. This work employs a GNN architecture with residual connections tailored for the layout and tiles dataset. The implementation made use of the TensorFlow package and repurposed open-source code containing baseline models and functions for data processing. [1].

3.1 Traditional GNN

The input for the model is a custom embedding layer that represents the nodes of operations in the graph. Complex relationships can be represented with a higher embedding dimension. Consequently, the layer has a 2D shape defined by the embedding dimension hyperparameter and the number of unique operations of the original graph.

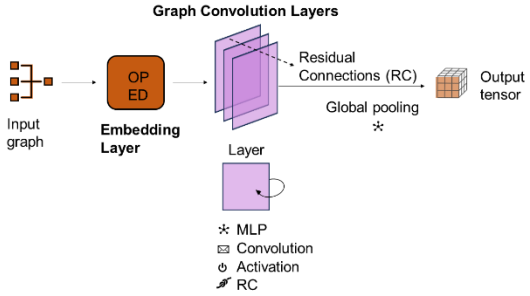

Figure 2. GNN architecture

Given the computation costs the embedding layer goes through only two graph convolution layers, which obtained good results in [2]. Each layer incorporates a multi-layer perceptron, "MLP" for processing features at the node level. Every MLP contains a bias term, regularization "l2reg", and activation function "Leaky Relu." The MLP output is a tensor that will go through a convolution operation (message passing), which is a linear combination of the features of neighbor nodes and will be transformed through a Leaky Relu function to add non-linearity. Subsequently, the architecture incorporates residual connections, which improve information flow across all layers by appending input features to each layer's output. Upon processing all layers, a global pooling operation (average pooling) occurs and applies l2reg. While the runtime is the expected output for each unique set of configurations, the competition required that the predictions be expressed as rankings so each set of configurations could be compared to others. Therefore, it is critical that the model output retains the graph's original shape; to accomplish this, the global pooling result is wrapped with a final MLP model to produce a tensor with the input graph's original shape which includes the resulting predicted runtime.

During the training process, the models incorporate a Listwise Maximum Likelihood Estimation "ListMLE" loss function which adjusts to the expected rankings outputs. The model implements the "ADAM" optimizer. Finally, to prevent overfitting, the architecture employs segment dropout, where edges are randomly sampled during training and early stopping based on the validation metric, ensuring efficient convergence.

3.2 Attention GNN

To capture more significant relationships between the data, a self-attention layer was added to the traditional model. The self-attention layer is applied before the convolution operation, and it computes the attention scores to calculate the relative importance of each element in the sequence based on the same set of features used in both as queries and the keys.

4 Model Evaluation

4.1 Hyperparameter tuning

Hyperparameter tuning was performed on both GNN models, focusing on preserving the best results while staying within memory limitations. The modified parameters were maximum nodes, maximum configurations, and batch size. The maximum nodes hyperparameter was used to down sample the number of operation nodes within a graph, thereby reducing the number of operations the model can learn from. The maximum configurations hyperparameter was used to down sample the number of configuration nodes, which were significantly more influential on the resulting run time than operation nodes. The batch size hyperparameter influenced the number of graphs used during a single training epoch, influencing the amount of time needed to iterate the entire training dataset. The parameters were initially set to values that would minimize the amount of RAM used during the training and testing process and were increased until the model reached peak performance within memory constraints.

4.2 Evaluation metrics

Two different methods were used in the model evaluation: The host competition's evaluation, which was found to be difficult to interpret for testing the model architecture and a personal evaluation for hyperparameter tuning. The host-competition's metric is the average of a metric per dataset. For the tiles dataset equation 1 was used, where K denotes the top-K configurations predicted by the model and A is all predicted configurations.

$$2 - \frac{\min K}{\min A}$$

Equation 1. Metric for Tiles dataset

Given the increased number of configurations per graph in the Layout dataset the Kendall Tau Correlation metric, described in equation 2, was chosen. This method compares the rankings of the configurations' runtime to the actual ranking. Where C is the number of concordant pairs, and D the number of discordant pairs.

$$(\mathrm{C} - \mathrm{D})/(\mathrm{C} + \mathrm{D})$$

Equation 2. Kendall Tau Correlation metric for Layout dataset

The personal evaluation was conducted using the training and validation dataset, as the testing data lacked labels. Two different metrics were employed to calculate results: ListMLE and Ordered Pair Accuracy (OPA) which determined the percentage of correctly ordered pairs and adjusts to rankings.

5 Results

Google Collab was initially chosen as the test environment with 12.7 GB of RAM, but multiple experiments in the training process showed it was not enough memory to process the graphs. Therefore, Kaggle was selected as test environment offering 27.9 GB RAM, 70 GB of CPU and 15 GB of GPU.

In terms of hyperparameters, the competition suggested using 1000, 500, and 200, respectively, for max nodes, max configurations, and batch size. However, the hyperparameters had to be lowered to 100, 20, and 10-100 due to the available resources. Once the model was running, epochs were trialed at 1-6, and early stopping was set between 2-4. With these hyperparameters, training took between 30 to 60 minutes, and testing took between 2 and 4 hours.

Model	*Batch size 10*	*Batch size 50*	*Batch size 100*
GNN	64.64	73.97	56.43
GNN + attention	67.14	**76.79**	68.22

Table 3. Percentage of OPA average for all validation datasets

Table 3 shows the OPA average across all datasets with the two different models: GNN and GNN + attention. Overall, GNN + attention performed better than the simple GNN architecture with a max average of 76.79%. Although the OPA results were close between the two models, the attention architecture was preferred, as it provided a more predictable loss behavior across different datasets as shown in figures 3 and 4. With non-attention models, the loss would move erratically between epochs, making it difficult to understand the behavior. Although the small number of max nodes and max configurations could have also caused this behavior.

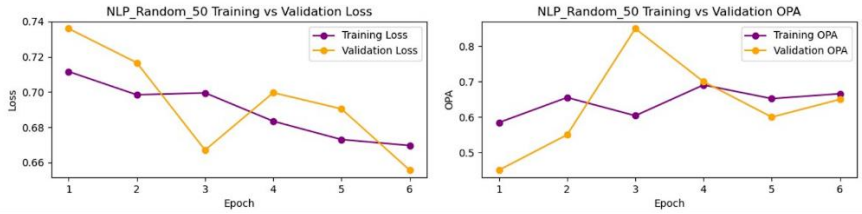

Figure 3. Training vs validation loss and OPA of GNN + attention

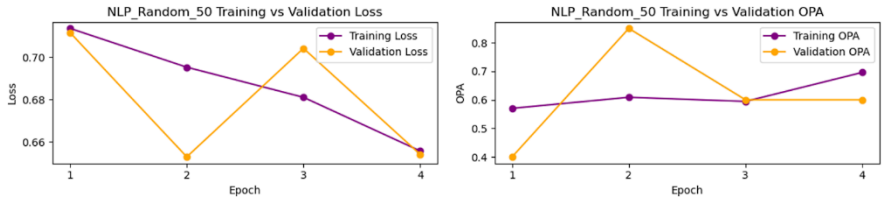

Figure 4. Training vs validation loss and OPA of GNN

Focusing on the batch size hyperparameter on the GNN + attention model, as shown in table 4, the best OPA was obtained with the XLA random dataset with a batch size of 50, which also obtained the best OPA on average across datasets. Overall random datasets performed better than the default ones suggesting the metric could have been impacted by the datapoints using during validation.

Batch size / dataset	*10*	***50***	*100*
XLA default	71.4	71.4	71.4
XLA random	57.1	**85.7**	71.4
NLP default	55	65	55
NLP random	85	85	75
Dataset avg	67.1	**76.8**	68.2

Table 4. OPA results of GNN + attention with different batch size

Regarding the competition metrics, the best-performing model had a batch size of 10 and was trained with 1 epoch obtaining a score of 23.18% as shown in table 5. This resulted in a placement of 239 out of 616 in the leaderboard.

	1 epoch	*6 epochs*
10 batches	**23.18**	23.06
50 batches	-	22.17
100 batches	17.58	85

Table 5. Competition Metric results using GNN+Attention

6 Conclusion

This study, presented a comprehensive approach to computational graph layout optimization, leveraging a Graph Neural Network. The exploration began with a detailed examination of the underlying data, comprising diverse computational graph layouts sourced from domains such as "xla" and "nlp." This GNN integrates an MLP with residual connections, enabling efficient message-passing and feature processing across nodes and edges as well as an attention layer to improve the model accuracy. The model's implementation showcased its flexibility in handling different graph structures and its ability to adapt to varying optimization scenarios.

The model is evaluated using OPA and ListMLE loss across various datasets, search strategies, and hyperparameters. Despite resource limitations, the model demonstrates promising performance, achieving a notable OPA of 85.7% on the XLP dataset with a random search strategy. However, challenges such as training time and RAM constraints necessitate further exploration of hyperparameter settings to improve model efficiency. Inconsistency in OPA and loss results across datasets and hyperparameters also suggests overfitting given a small training dataset.

Overall, the Kaggle dataset and competition style was aimed more toward rapid testing rather than an in-depth understanding of the material, as future work it would be interesting to reduce the graphs size to obtain better and smaller representations for GNNs. As computational graphs continue to play a pivotal role in various domains, this work opens new possibilities for enhancing efficiency and performance in critical graph-based applications.

References

[1] Mangpo Phothilimthana, Sami Abu-El-Haija, Bryan Perozzi, Walter Reade, Ashley Chow. (2023). Google - Fast or Slow? Predict AI Model Runtime. Kaggle. https://kaggle.com/competitions/predict-ai-model-runtime

[2] Kaufman, S., Phothilimthana, P., Zhou, Y., Mendis, C., Roy, S., Sabne, A., & Burrows, M. (2021). A learned performance model for tensor processing units. Proceedings of Machine Learning and Systems, 3, 387-400.