Performance Models for Deep Neural Network Training

Fall 2023, CS243 Final Project

Andrew Li
Harvard University
andrew_li@college.harvard.edu

Saketh Mynampati
Harvard University
sbmynampati@college.harvard.edu

***Abstract*—Performance models to predict program configuration runtimes and recommend optimal configurations play an important role for today's compilers, particularly in the compute-intensive machine learning domain. Performance models stand to provide cost and time improvements over traditional heuristic methods, greedy optimizers, and physical measurement for estimating a program's runtime. In this paper, we employ TpuGraphs, a performance prediction dataset for deep neural network (DNN) graph configurations, to build a learned cost model that accurately predicts and ranks the most optimal graph configurations. Our work encompasses motivated feature engineering to tackle the high-dimensional nature of DNN graphs, data exploration and visualization, and aggregate modeling using LightGBM. We show that our model substantially outperforms the baseline graph neural network by 59% on graph tiling optimization.**

I. Introduction

Hardware performance models are auxiliary tools that play a critical role in optimized code compilation. Compilers are presented with several choices when converting high-level human code to its low-level equivalent for execution. These choices involve non-trivial decisions for which an optimal and discrete strategy does not yet exist. Performance models serve to assist compilers or autotuners in recommending an optimal program configuration. The alternative is physical hardware simulation, which can be costly and time-consuming. Performance models are canonically implemented analytically with heuristic methods or greedy algorithms, but these approaches are inherently challenging due to the shear complexity of computer program execution and time-intensive to implement and run [1].

Compilers are of particular relevance to machine learning domains. With rapid advances in deep learning algorithms and growing real-world applications, training efficiency for ML models has become a paramount concern. Roughly speaking, the doubling time of ML training FLOPS has been 6 months, and deep neural network (DNN) parameter size has grown in a similar fashion [2], [3]. OpenAI's breakthrough GPT-3 large language model is estimated to have cost $4.6 million in compute, over a thousand GPUs, and 34 days to train [4].

ML models have been proposed to fill the role of a reliable and generalizable performance prediction model. These approaches are focused on piecemeal sub-programs such as basic assembly or a handful of specific tensor operations. In this paper, we leverage TpuGraphs, a recent performance dataset composed of complete DNN programs represented as graphs, to gain insight into different methods to optimize performance models to most accurately fit DNNs.

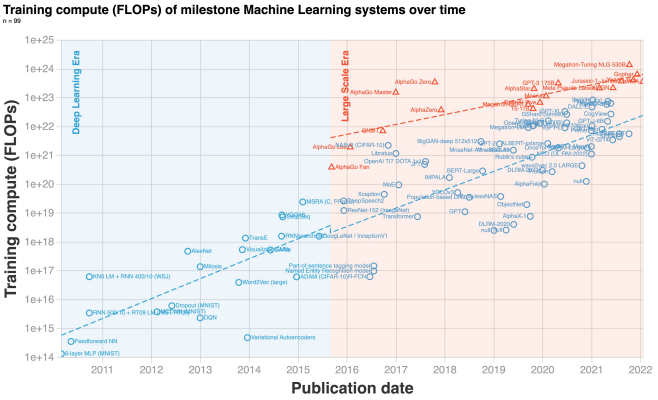

Fig. 1. Training compute has grown exponentially with time [2].

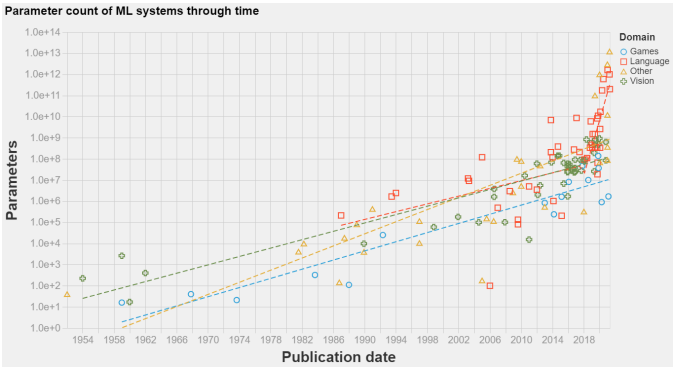

Fig. 2. Parameter counts of models have grown exponentially with time [3].

II. Data

The TpuGraphs dataset is one of the largest datasets of its kind, and the data is made up of different trained DNNs, each with multiple potential configurations. Representing nodes as tensor operations such as matrix multiplication and convolution, and edges as representations of data flow and depen-

dency between nodes, TpuGraphs contains 31 million graph-configuration pairs in total. It includes state-of-art models such as BERT, Inception, and ResNet, giving an accurate sample space of real-world models that a learned cost model might encounter. Some configurations are generated by Google's XLA compiler which has search space-enumerating and optimizer capabilities at the expense of long (roughly 6 hour) convergence times [5].

The performance model we seek to create optimizes over two major program attributes: layout and tiling.

These two factors are dependent open both graph-level properties and node-level attributes, which in turn can reveal the network type or subdivide the dataset further to make runtime prediction more accurate.

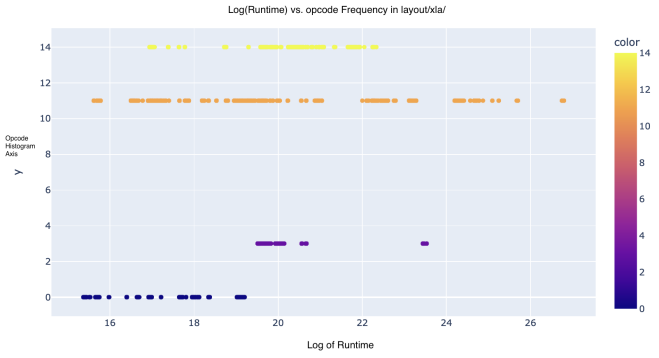

Fig. 3. Runtime distribution of graphs separated based on frequency of nodes with each given operation code. [1].

In the above figure, opcode acts as a proxy to view the network, but from a local node property instead of a global property, which indicates a way to compress the information further and segment the full dataset to make the original ranking task more specific to each type of graph.

A. Layout

Layout characterizes the physical storage of tensors in memory. For a given series and or combination of tensor operations that a DNN is made of, there exist several possible configurations in which they may be laid out. Reconfiguration is necessary in several cases, and the compiler is faced with several potential choices that it must optimize over. In Figure 3, a mismatch between the add and convolution output and input dimensions, respectively, necessitates layout changes.

Already tradeoffs are introduced, as the compiler must balance the copy node computational overhead with potential outsized improvements relative to other configurations. Another motivation for layout optimization is hardware specific memory architecture and padding, which can introduce varying optimal layout configurations (e.g. optimally utilizing a TPU accelerator's 128x128 Matrix-Multiplier Unit). Each layout configuration might require a special set of additional operations to be inserted, or memory contents to be copied over from one location to another. This creates a large search

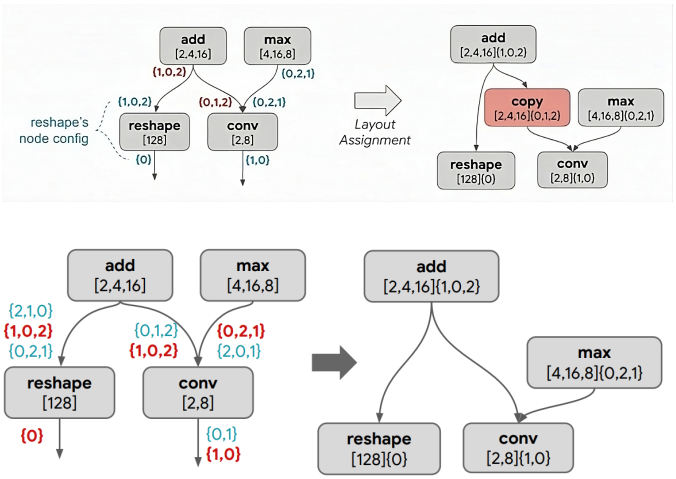

Fig. 4. Two possible layout re-configurations for a graph subsection. [1].

space for the compiler, from which optimal selection can lead to a large speedup [1].

B. Tile

Tile shape and size characterize how tensor operations fit into memory. Small tile sizes will underutilize any available highspeed memory such as scratchpad memory. Large tile sizes may cause memory overflow, particularly with large single matrix multiplications prevalent in DNNs, resulting in similar training slowdowns. The search space of per-node tile configuration is relatively small, so we can afford to enumerate all possible combinations in our data.

C. Data and Preprocessing

The dataset is divided into several sections. The first delineation is tile and layout data, with the respective data splits containing either tile or layout features. The tile data is straightforward, with samples consisting of models and their enumerated tile configurations and corresponding runtimes.

Within the layout data, there are two delineating variables resulting in four total datasets. First, the data is divided by model type: 'xla' consists of a diverse group of models from various domains while 'nlp' comprises NLP transformer models. Within each group, the data is divided by how it was generated: 'random' consists of models with randomly sampled configurations and 'default' consists of models whose configurations were programmatically enumerated by the XLA compiler.

The prediction target is the configuration's runtime normalized to a standard default configuration runtime and MinMaxScaled. We normalize runtimes to account for data generation on heterogeneous machines, and we scale the normalized scores because the prediction task concerns only relative configuration rankings such that the cost model can recommend the most optimal configuration.

A baseline model was developed alongside the dataset consisting of a simply graph neural network (GNN) with GCN

and GraphSAGE, both performing similarly. We successfully implemented the baseline model and verify its performance on smaller subtasks but will rely on the originally published evaluation metrics as benchmarks for the sake of limited computation resources.

D. Challenges

Several major challenges are inherent in this prediction task. The most difficult lie in the data's properties.

High dimensionality. The graph sample space is incredibly high-dimensional and diverse. Each model can have thousands of potential configurations arising from thousands of nodes and per-node attributes. Trying to generalize to unfamiliar programs and graphs is difficult, even with the size of the TpuGraphs dataset which, while composed of tens of millions of configurations, traces each sample back to only a handful of base models that carve out very unique data regimes (due to differing domain applications).

Modeling. To model such complex data, a model must 1) be expressive enough to capture underlying patterns, 2) be properly tuned so as to avoid overfitting and maximize accuracy, and 3) scale well to massive datasets that may exceed even the training machine's memory capacity. The data is sparse when accounting for the redundancy of models in the dataset (e.g. BERT is represented several times), and this poses further problems when trying to train complex models like GNNs. Even if such a model were developed, it's unclear whether its performance gains would warrant the use of an uninterpretable black box in production settings.

III. Design

We typically start with the high-level architecture of your system, and then describe the details of your design, described in enough relevant detail that a skilled system builder could replicate your work. It is also important to compare your design choices with alternative approaches to give us reasons on why you design your system in this way.

Neural networks are difficult to explain, prone to overfitting, reliant on quality and abundant data, and time-consuming to train and infer with, particularly when accounting for large memory requirements. In preliminary implementations of the baseline model, basic data loading and preprocessing accelerated memory consumption to over 32GB before training began. Hence, in this paper, we take a data and feature-centric approach in pursuit of not only a high-performing performance model but also meaningful and interpretable insights.

To this end, we forego a deep learning approach to the prediction task, and instead pursue deliberate feature engineering and analysis for prediction. For both tile and layout tasks, a smaller subslice of data is treated as the training data (1,000,000 configuration samples in the case of tile) to balance training and work iteration speed with sufficiently expressive models.

A. Tile

Tile data is relatively straightforward and low-dimensional. We use 23-dimensions of per-node tile configuration features relevant to the node's operation: input tile size, output tile size, and, in the case of convolution, convolution kernel bounds. Additionally, each configuration is associated with the averaged 140 node features across the entire graph, including shape dimeensions, broadcasting dimensions, window attributes such as size and padding in the case of convolutions, convolution striding, and edge padding characteristics. We use an $80/20\%$ train-test split.

Baseline results are obtained from simple, lasso, and ridge regularized regression with the penalty term α selected through a grid search. A high-performing light gradient-boosting machine (LightGBM) is trained on the data with hyperparameters tuned through randomized search using 4-fold cross-validation, mean-squared error scoring, and 25 iterations.

Gradient boosting, and specificaly LightGBM, was chosen due to a combination of favorable factors in the prediction problem. LightGBM lends itself well to high-dimensional data by performing inherent feature selection with best-split optimization and matches or outperforms deep learning and other gradient boosting approaches both in computational efficiency and prediction performance for tabular datasets due to leaf-wise tree growth. Additionally, LightGBM's split and gain scoring of features allows for more intuitive and explainable model decisions-making.

B. Layout

Layout data is more relevant in a graph-wide context, as unlike tile size, a single node's layout configuration will can directly impact the performance of neighboring nodes (or result in the creation of new nodes, as in the Figure 3 example). GNNs show the most promise in capturing the complexity of the graph layout interactions, but learning on aggregated graph features may still be sufficient for relatively strong performance due to the dimensionality of the data. We use a $90/10\%$ train test split due to the dataset's smaller size relative to tile (66,000 samples).

We compute the following features:

- Total padding generated per node
- Frequency of each layout node feature value relative to all feature values
- Number of convolution, dot product, and reshape nodes

Padding is computed as the sum of all additional padding generated by the nodes. Due to hardware specific memory alignment properties, node padding can have significant impacts on program efficiency. Layout node feature rate is relevant for the "default" dataset, as the configurations are automatically generated by XLA's greeedy, genetic algorithm. This implies that more frequent configuration settings should be associated with faster runtimes. Lastly, we consider the frequency of convolution, dot product, and reshape nodes as these tend to be the most compute intensive tensor operations and are the only configurable operations at all in the data.

These features are relevant because, roughly speaking, model runtime should correspond to the sum of the runtimes of its parts.

Like our approach with tile data, we train three varying linear regressions as baseline along with a LGBM model to predict the normalized and scaled configuration runtime.

IV. Evaluation

For systems work, this will often include the following subsections: (1) Experimental setup. Describe how you ran your experiments. What kinds of machine? How much memory? How many trials? How did you prepare the machine before each trial? (2) The experiments themselves, grouped by purpose. Include figures. (3) A summary of the experimental results. Some good evaluations are organized around performance hypotheses: statements that the experiments aim to support or disprove. It is important to discuss the implications of your results and why you see such results.

Experiments were run on Google Jupyter Notebook environments with a single Nvidia Tesla P100 GPU with 16gb VRAM and 30GB RAM. Machines used a 2-core Intel Xeon CPU. Experiments were run on fresh machines to confirm evaluation results at conclusion. First, we conducted exploratory data analysis (EDA) and visualiztion with our engineered features, and then proceeded to modeling. We test our models on the data and benchmark them against the baseline model performance.

A. Exploratory Data Analysis and Visualizations

After featurizing the tile data, we observe that within a model, there exists a strong degree of linear separation between configurations (Fig. 6), but the same does not hold as strongly for the full dataset configuration space (Fig. 5).

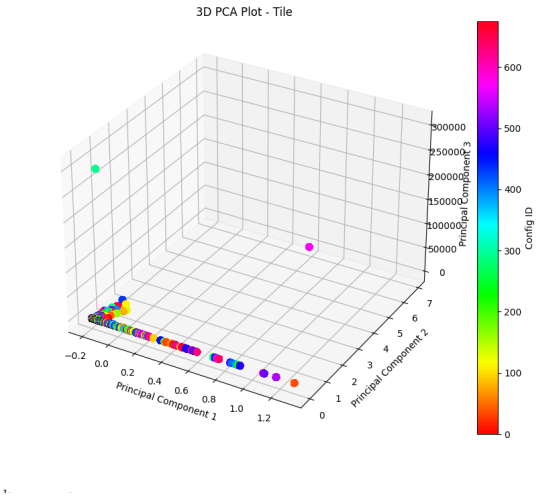

Fig. 5. PCA plot of tile features across all configurations and all models. The majority of variance lies along the first principal component.

B. Evaluation Metrics

The primary metric we use for tile evaluation is *Top-K Slowdown*, measuring the average slowdown of our recommended models relative to the true optimal configuration in the data.

$$Top{-}K\,Slowdown = 1{-}(\frac{\text{Best runtime of top-k predicted}}{\text{Best runtime of all configs}}{-}1) \tag{1}$$

We choose $K = 5$ for our evaluation—in real-world tiling use cases, the absolute cost of evaluating all tiling configurations is cheap.

When considering the layout collection, we use Kendal Tau Correlation, a ranking metric considering the entire ranked list of configurations. While tile size search space is small and we can afford to infer on every possible configuration, layout collections have a much larger optimization space. Compiler strategies hence use greedy and genetic algorithms for optimization, and require a "fitness" or "utility" function such as our performance model [7]. The full configuration ranking is necessary for these applications, and we evaluate as such. The formula is given in equation (2), where C is the number of concordant pairs and D is the number of discordant pairs between the true and predicted rankings.

$$\tau = \frac{C - D}{C + D} \tag{2}$$

C. Model Benchmarks

TABLE I
Evaluation Results

Dataset	Test Performance: Top-K Slowdown % and Kendal τ				
	Baseline	*OLS*	*Lasso*	*Ridge*	*LGBM*
Tile:XLA	9.1	43.6	45.7	52.5	3.7
Layout:XLA:Default	0.12	-0.0008	0.006	0.001	0.004

For the tile data, the unregularized linear regression (OLS), lasso, and ridge regressions all severely underperform the baseline model. The best of their top 5 ranked configurations underperforms the true optimal configuration by an average of $43.6 - 52.5\%$. The baseline appears serviceable in the absence of other models with a modest 9.1% average slowdown. Our LightGBM regressor achieves an average slowdown of only 3.7%, representing a 59% smaller average slowdown, even when trained on just one-tenth of the available data.

Additionally, incorporating the identity of operation codes for each node as a way to globally assess the given network's purpose didn't greatly affect slowdown, leading to a marginal/nearly negligible improvement to a 3.5% when training on a dataset of equal size. In a list of features that influenced model predictions the most, op-codes did not appear in the top ten.

For layout data, all non-baseline models perform extremely poorly. This is likely a function of data sparsity (there were less than 10 models represented among $> 60,000$ configurations) combined with the greatly increased dimensionality

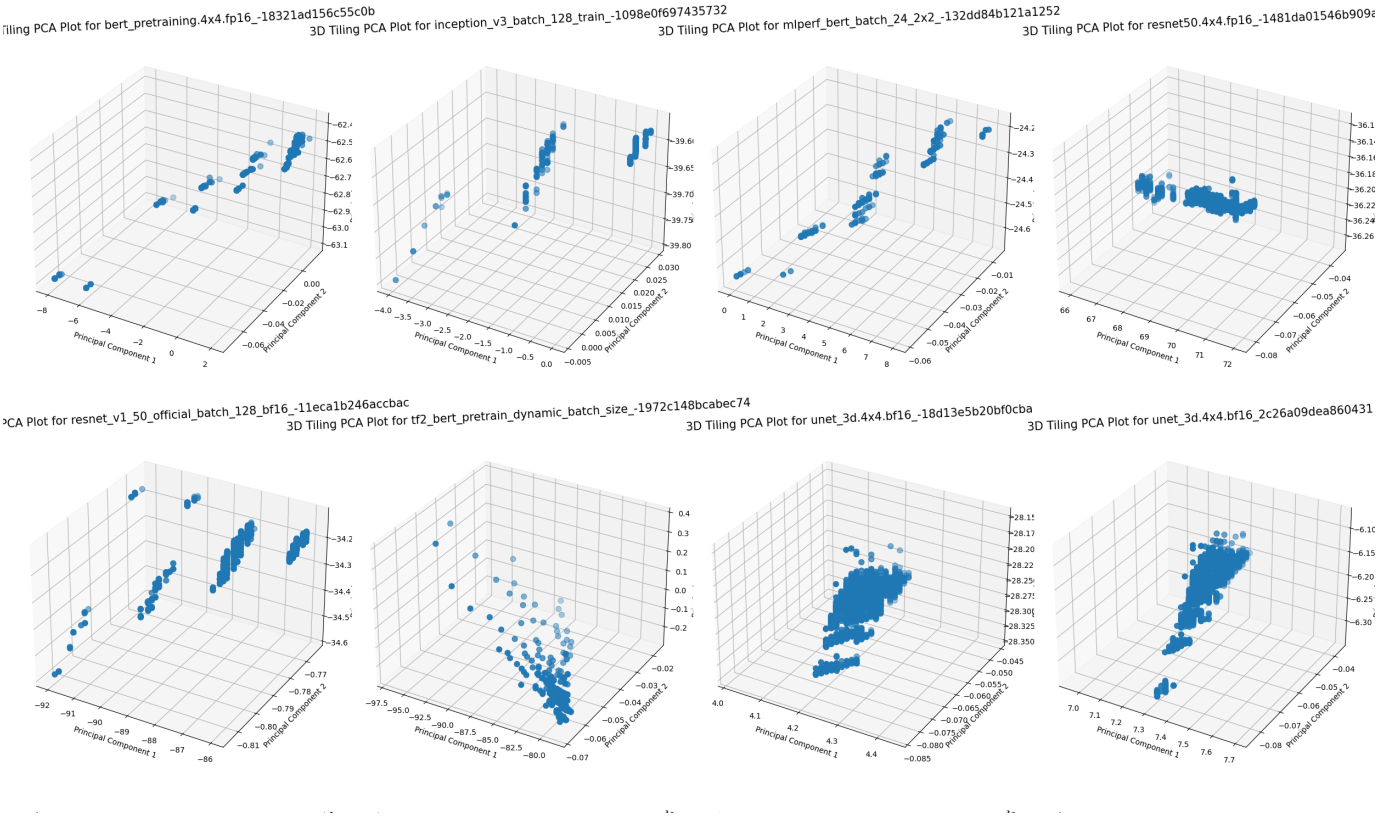

Fig. 6. PCA plot of tile features among configurations of the same model. All models exhibit a strong degree of linear separability in three dimensions.

associated with layout configurations. Although not immediately obvious why GNNs would outperform in low-data/high-dimensionality settings, it is clear that its expressivity and ability to capture more complex graph properties improves its performance over alternative models.

Incorporating opcode identity did seem to help over the baseline, leading to an increase in Kendall-Tau Correlation. However, further research would be needed to determine if this played a significant role in the improved performance, and if this effect would be scalable to larger datasets, as the layout data is relatively small for this application.

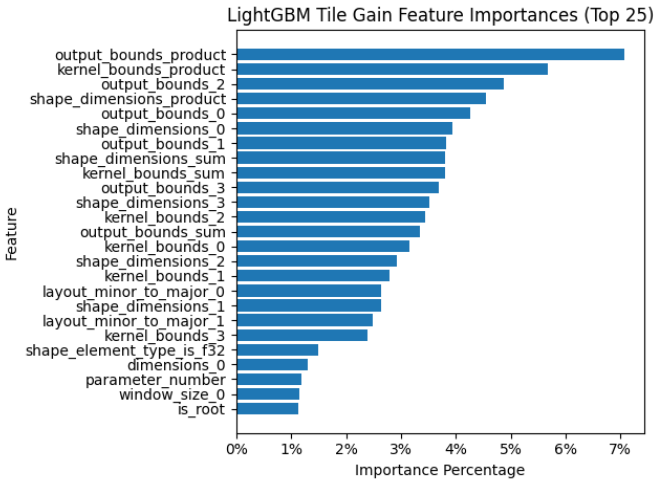

Fig. 7. Feature importance plot for LGBM regressor by gain.

D. Feature Importance

One advantage of LightGBM models is ease of feature importance interpretability. One such metric, gain, represents the reduction in loss resulting from adding a split point. By aggregating the gain for all features' split points, we can normalize and derive a metric for each feature's contribution to the model's behavior.

V. Related Work

The TpuGraphs dataset [1] was the original work that gave rise to this research. For further reading, Google's XLA compiler and Xtat autotuner are closely related to this domain of research and were used in enumerating the configuration search space for the TpuGraphs dataset [5]. Through the Kaggle competition based on this paper, several insights were made such as the value of opcode (operation type) [6] and

the ability of simpler models to perform well with aggregate graph feature engineering.

Additionally, some insights we confirmed, such as the lack of importance of non-configurable nodes, were drawn from previous competitive efforts. For example, another competitor determined, in the forums, that GraphSAGE representation wouldn't help with accuracy, so we did not explore the idea further [6]

VI. Conclusion

The initial data analysis yielded some key insights.

First, the data in the Tile set is very diverse, and as noted in the Kaggle Competition, very noisy; we found that global graph properties could explain large amounts of the variance in runtime, indicating some sort of clustering based on network type.

Additionally, the number of parameters were not conducive to reduction and any simple representation, aside from pruning non-configurable vertices; we sought to retian some properties by preserving the operation codes of such vertices in order to avoid losing the information they provide in the graph.

The Tile-based model performed well with and without this distinction, with engineering features replacing the GNNs used in [1], and the Layout model fared worse. Future research would involve improving the accuracy of and re-engineering features for, the Layout model.

Acknowledgments

Thank you to Minlan Yu, Weifan Jiang, and Mark Ting for your teaching, guidance, and an amazing semester!

References

[1] P. M. Phothilimthana et al., "TpuGraphs: A Performance Prediction Dataset on Large Tensor Computational Graphs." arXiv, 2023. doi: 10.48550/ARXIV.2308.13490.

[2] J. Sevilla, L. Heim, A. Ho, T. Besiroglu, M. Hobbhahn, and P. Villalobos, "Compute Trends Across Three Eras of Machine Learning," 2022 International Joint Conference on Neural Networks (IJCNN). IEEE, Jul. 18, 2022. doi: 10.1109/ijcnn55064.2022.9891914.

[3] J. Sevilla, "Parameter counts in Machine Learning," 2021.

[4] D. Narayanan et al., "Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM." arXiv, 2021. doi: 10.48550/ARXIV.2104.04473.

[5] P. M. Phothilimthana et al., "A Flexible Approach to Autotuning Multi-Pass Machine Learning Compilers," 2021 30th International Conference on Parallel Architectures and Compilation Techniques (PACT). IEEE, Sep. 2021. doi: 10.1109/pact52795.2021.00008.

[6] Abaojiang. (2023, September). Re: Detailed EDA [Discussion post]. Kaggle. https://www.kaggle.com/code/abaojiang/google-fast-or-slow-detailed-eda.

[7] P. M. Phothilimthana et al., "Google - Fast or Slow? Predict AI Model Runtime." Kaggle, 2023.