SC4000 Course Project
Google – Fast or Slow? Predict AI Model Runtime

Muhammad Zohaib Irfan — Tile Model
Zheng Andy — Layout Model

November 8, 2025

1 Competition Overview

The competition "Google – Fast or Slow? Predict AI Model Runtime" requires predicting which compilation configuration makes a TPU program execute fastest. Submissions rank the configuration indices for each program by predicted runtime (lower is better).

There are two types of compilation configurations: Tile and Layout. Each having its own pre-split train, validation and test datasets and evaluation metrics.

For tile, there is 1 dataset: tile:xla. The model needs to output the indices of the top 5 configurations with the lowest predicted runtime. The evaluation score is given by the (1-slowdown) incurred of the top-K predictions.

$$1 - \left(\frac{\text{The best runtime of the top-k predictions}}{\text{The best runtime of all configurations}} - 1 \right) = 2 - \frac{\min_{i \in K} y_i}{\min_{i \in A} y_i}$$

For layout, there are 4 datasets: layout:xla:random, layout:xla:default, layout:nlp:random, layout:nlp:default. The model needs to output the indices of all configurations sorted by predicted runtime in order. The evaluation score is given by Kendal Tau Correlation.

The final evaluation score for the competition is given by the average of the scores across all 5 datasets.

2 Tile Dataset: `.npz` Format

Each `.npz` file represents a single compiled graph with n nodes and m edges, compiled under c configurations (graph-level). The dataset contains the following keys:

- `node_feat` $\in \mathbb{R}^{n \times 140}$: float32 node features (topologically ordered)
- `node_opcode` $\in \mathbb{Z}^{n}$: opcode (int32) for each node
- `edge_index` $\in \mathbb{Z}^{m \times 2}$: directed edges $[u, v]$ meaning $u \leftarrow v$
- `config_feat` $\in \mathbb{R}^{c \times 24}$: per-configuration features
- `config_runtime`, `config_runtime_normalizers` $\in \mathbb{Z}^{c}$: runtime (ns) and baseline normalizer per config

We predict the best configuration by minimizing:

$$r_j^{\text{norm}} = \frac{\texttt{runtime}_j}{\texttt{normalizer}_j}.$$

3 Layout Dataset: `.npz` Format

Each `.npz` file represents a single compiled graph with n nodes and m edges, and nc configurable nodes, compiled under c configurations (node-level). The dataset contains the following keys:

- `node_feat` $\in \mathbb{R}^{n\times 140}$: float32 node features (topologically ordered)
- `node_opcode` $\in \mathbb{Z}^{n}$: opcode (int32) for each node
- `edge_index` $\in \mathbb{Z}^{m\times 2}$: directed edges $[u, v]$ meaning $u \leftarrow v$
- `node_config_ids` $\in \mathbb{Z}^{nc}$: indices for configurable nodes.
- `node_config_feat` $\in \mathbb{R}^{c\times nc\times 24}$: node configuration for each configuration. Specifically, for entry $[j, k]$, it gives the 24-dim feature vector for the configurable node with index k under configuration j.
- `config_runtime` $\in \mathbb{Z}^{c}$: runtime (ns)

4 Challenges of Layout Dataset

The main difference between the tile and layout dataset is that `node_config_feat` key being a 3D matrix, while the `config_feat` key is a 2D matrix. For some layout collections, there could be up to 100000 configurations per graph with more than 100 configurable nodes, making it infeasible to load the entire `node_config_feat` into memory.

Furthermore, the output needs to be a sorted list of all configurations based on predicted runtime, which is more challenging than just predicting the top-K configurations as in the tile dataset.

5 Layout Data Pre-processing

Due to limited memory, data must be compressed before training.

Firstly, nodes can be configurable or non-configurable. Non-configurable nodes have identical features across all configurations, Hence, we only extract the features of configurable nodes by using `node_config_ids` to index into `node_feat`, `node_opcode`.

Secondly, we only extract the first 1000 configurations for each layout collection. Although this may lead to loss of information, it significantly reduces the memory usage and time to load data. The test set keeps all the configurations since we need to output the sorted list of all configurations.

Thirdly, to enhance model convergence, normalization is applied to `node_feat`, `node_opcode`. For `config_runtime`, min-max scaling is applied.

6 Layout Model Architecture

For the layout dataset, we use a simple MLP architecture. The model takes in three inputs: `node_feat`, `node_opcode`, and `node_config_feat`.

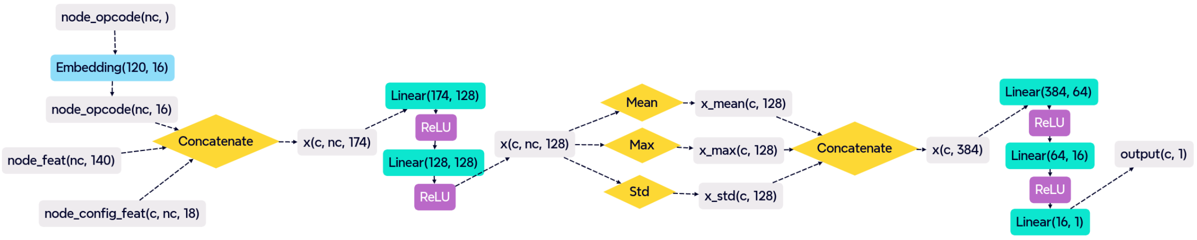


Figure 1: Layout model architecture: Note that node features only contains configurable nodes. Hence node_opcode and node_feat have length nc instead of n

`node_opcode` is first passed through an embedding layer to get a dense representation. Then, we concatenate all features. The concatenated features are passed through two linear layers with ReLU activation. Then, we extract the mean, max and standard deviation across all nodes. These statistics are concatenated before being passed through another two linear layers with ReLU activation. Finally, the output is passed through a linear layer to get the predicted runtime for each configuration.

7 Layout Training Setup

- **Optimizer:** Adam with learning rate $\text{lr} = 10^{-3}$ and weight decay $\text{wd} = 10^{-4}$
- **Scheduler:** Cosine Annealing
- **Loss function:** RankNet loss
- **Batch size:** 1
- **Epochs:** 200

8 Layout Analysis

The above setup achieved the best result. However, we have also experimented with other configurations.

For the loss function, we have experimented with MSE loss or L1 loss. However, the model would output equal runtimes for all configurations, leading to random ordering and poor performance on the leaderboard. This is likely due to the fact that the loss function only cares about minimizing the difference between predicted and actual runtimes, rather than the relative ordering of configurations. RankNet loss solves this issue by directly optimizing for the ordering.

For the model architecture, we have explored different MLP configurations, including varying the number of layers and hidden units. However, these changes did not lead to significant improvements in performance.

For the number of epochs, we found that increasing the number of epochs did yield better results. And adding an early stopper did not help.

9 Layout Model Limitations and Future Work

While the current layout model achieves reasonable performance, there are several limitations and areas for future improvement.

Firstly, the model does not fully utilize the graph structure of the data. Only node features are used, while edge information is ignored. In future work, we could explore graph neural networks (GNNs) to better capture the relationships between nodes and edges.

Secondly, the model currently only considers the first 1000 configurations due to memory constraints. Future work could investigate more efficient data loading and processing techniques to handle larger numbers of configurations.

Thirdly, hyperparameter tuning could be further optimized using techniques such as Bayesian optimization or grid search.

Fourthly, ensemble methods could be explored to combine predictions from multiple models.

Lastly, incorporating domain knowledge about TPU compilation and execution could help inform feature engineering and model design.

10 Feature Engineering (Tile Only)

For each program i and configuration j, we construct a feature vector:

$$x_{ij} = \underbrace{\texttt{config\_feat}_{ij}}_{\in\mathbb{R}^{24}} \oplus \underbrace{\phi(\texttt{node\_feat}, \texttt{node\_opcode}, \texttt{edge\_index})}_{\in\mathbb{R}^{8}} \in \mathbb{R}^{32},$$

where ϕ extracts graph-level summary statistics:

1. Node count n and feature dimension $F = 140$
2. Global mean and standard deviation across all node features
3. Opcode unique count and total count
4. Edge count m and density proxy $m/\max(1, n^2)$

Feature Standardization

Training-time standardization uses train-only statistics to prevent data leakage:

$$x' = \frac{x - \mu_{\text{train}}}{\sigma_{\text{train}} + \varepsilon}.$$

11 Model Architecture

11.1 Why an MLP?

Tile representations are tabular and low-dimensional after summarization. A multi-layer perceptron offers sufficient expressiveness without requiring graph message passing.

11.2 Network Definition

Let $x \in \mathbb{R}^{32}$. Our model consists of three layers:

$$\begin{aligned}
h_1 &= \mathrm{ReLU}(W_1 x + b_1), \quad W_1 \in \mathbb{R}^{256\times 32},\\
h_1 &= \mathrm{LayerNorm}(h_1),\\
h_2 &= \mathrm{ReLU}\big(W_2\,\mathrm{Dropout}(h_1; 0.1) + b_2\big), \quad W_2 \in \mathbb{R}^{128\times 256},\\
h_2 &= \mathrm{LayerNorm}(h_2),\\
\hat{y} &= W_3\,\mathrm{Dropout}(h_2; 0.1) + b_3, \quad W_3 \in \mathbb{R}^{1\times 128}.
\end{aligned}$$

11.3 Design Choices

LayerNorm: Runtime features vary significantly across programs. LayerNorm stabilizes hidden activations, improving training convergence for large-batch MLPs.

Dropout: The tile dataset is large, but many configuration features are correlated. Dropout (rate 0.1) mitigates co-adaptation and improves generalization to unseen graphs.

12 Training Setup

- **Optimizer:** AdamW with learning rate $\mathrm{lr} = 10^{-3}$ and weight decay $\mathrm{wd} = 10^{-5}$
- **Batch size:** 131,072
- **Epochs:** 5
- **Random seeds:** {42, 43, 44}

12.1 Log-Target Regression

Raw runtimes vary by orders of magnitude, leading to unstable gradients in direct regression. We predict:

$$y = \log(r_{ij}^{\mathrm{norm}}).$$

This transformation stabilizes variance, reduces the dominance of extreme runtimes, and empirically improves regret and ranking metrics.

12.2 Ensembling

For each seed s, model f_{θ_s} yields prediction $\hat{y}_s$. The final prediction averages across seeds:

$$\hat{y} = \frac{1}{3} \sum_{s\in\{42,43,44\}} \hat{y}_s.$$

Ensembling reduces variance, mitigates seed sensitivity, and improves ranking stability.

12.3 Per-Program Calibration

Raw predicted magnitudes differ across programs, but only *relative ordering within each file* matters for ranking. We apply per-program standardization:

$$\hat{y}'_{ij} = \frac{\hat{y}_{ij} - \overline{\hat{y}}_{i\cdot}}{\mathrm{std}(\hat{y}_{i\cdot}) + 10^{-12}}.$$

This enforces consistent scaling across programs and empirically improves Acc@1.

13 Results

13.1 Validation Performance

The model achieves the following metrics on the validation set:

Metric	Value
Acc@1	0.0814
Avg Regret	0.252924

13.2 Training Dynamics

Averaged across all three random seeds, the model exhibits stable convergence:

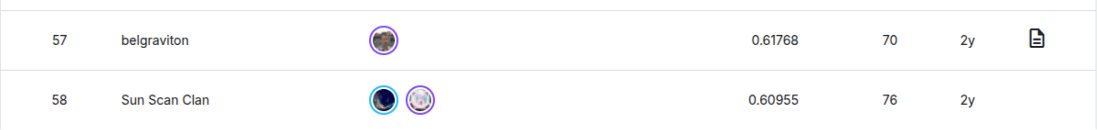
- **Train loss:** 0.45 → 0.21
- **Valid MSE:** 0.36 → 0.32

13.3 Ablation Study: Model Depth

We experimented with deeper architectures (4–6 layers). However, these models consistently achieved worse performance on the leaderboard compared to the 3-layer baseline. We therefore retain the 3-layer MLP as our final architecture.

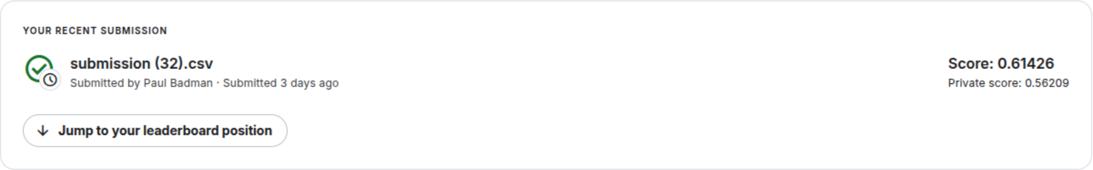
14 Overall Results

We combined the tile and layout submission csv files to create the final submission for the competition. This resulted in us achieving a final score of 0.61426 on the public leaderboard, ranking 58 out of 617 participants.



57	belgraviton	0.61768	70	2y
58	Sun Scan Clan	0.60955	76	2y

(a) Public leaderboard screenshot.



YOUR RECENT SUBMISSION

submission (32).csv
Submitted by Paul Badman · Submitted 3 days ago

Score: 0.61426
Private score: 0.56209

(b) Final score breakdown used for the submission.

Figure 2: Leaderboard and score breakdown.

15 Conclusion

In this report, we detailed our approach to predicting AI model runtimes for TPU programs using both tile and layout datasets. For the tile dataset, we employed feature engineering to summarize graph-level statistics, followed by a multi-layer perceptron architecture. For the layout dataset, we designed a specialized MLP model that processes configurable node features and utilizes RankNet loss to optimize configuration ordering. Despite complex data structures and memory constraints, we achieved a competitive score on the public leaderboard, demonstrating the effectiveness of data compression, normalization, feature engineering, and tailored model architectures. We also recognized the importance of experimentation, by exploring various loss functions, hyperparameters, and model depths to refine our approach. In future work, we aim to further enhance model performance by incorporating graph neural networks and more sophisticated data handling techniques.