Google – Fast or Slow? Predict AI Model Runtime

Solution Documentation

Date: December 14, 2025

Task: Graph Neural Networks / Runtime Prediction

---

1. Introduction

The objective of the "Fast or Slow? Predict AI Model Runtime" competition is to predict the runtime ranking of different configuration layouts for AI models executed on Tensor Processing Units (TPUs). The data represents computational graphs (tensors and operations) and various compilation configurations. The goal is not necessarily to predict the absolute execution time in nanoseconds, but to correctly rank the configurations from fastest to slowest (optimizing for latency).

Our team approached this problem in three phases:

1. **Data Exploration:** Discovering the characteristics of the dataset.
2. **Baseline Phase:** Developing a gradient-boosting approach (XGBoost) using aggregated graph statistics to establish a performance floor.
3. **Deep Learning Phase:** Implementing a Graph Neural Network (GNN) using GraphSAGE to capture the topological dependencies of the computational graphs, which yielded our best performance.

2. Exploratory Data Analysis (EDA)

The dataset consists of .npz files representing compiler graphs. Each file contains:

- **Node Features:** Attributes of the operations in the graph.
- **Edge Indices:** Adjacency matrix representing data flow.
- **Node Opcodes:** The specific operation type (e.g., Convolution, Add, MatMul).
- **Config Runtimes:** The target variable (execution times for different configurations).
- **Config Features:** Specific compilation parameters applied to the graph.

Data Statistics:

An initial scan of the dataset revealed a high variance in graph sizes.

- **Node Counts:** Ranged from very small subgraphs (~20 nodes) to large networks (>40,000 nodes).
- **Configs per Graph:** Some graphs had thousands of potential configurations, making pairwise ranking computationally expensive.

This variance necessitated data sampling strategies during training to prevent memory overflows, specifically limiting the number of graphs processed per epoch or the number of configurations compared per graph.

3. Solution Phase 1: Baseline (XGBoost)

To establish a baseline, we treated the problem as a standard regression task by flattening the graph structure into tabular features.

3.1 Feature Engineering

Since XGBoost cannot natively handle graph topologies, we aggregated node and edge information into global statistics:

- **Graph-Level Features:** Total number of nodes (n_nodes) and edges (n_edges).
- **Configuration Features:** The dataset provides feature vectors for every configuration. For 3D configuration features (node-specific configs), we calculated the **mean** and **standard deviation** across nodes to flatten them into 1D vectors.
- **Target:** log_runtime (Log-transformed runtime to reduce skewness).

3.2 Model Architecture

We utilized the XGBRegressor with the following hyperparameters:

- **Objective:** reg:squarederror
- **Estimators:** 1000
- **Max Depth:** 6
- **Learning Rate:** 0.05

3.3 Training Strategy

We used GroupShuffleSplit to ensure that configurations belonging to the same graph were not split between training and validation sets. This prevents data leakage, as the model must generalize to unseen graph structures.

3.4 Baseline Results

The model predicted the runtime for each configuration. To generate a submission, we sorted the predicted runtimes for a given graph to produce a ranking.

- **Local Validation (Kendall's Tau):** ~0.50
- **Kaggle Public Score: 0.1575**

While the model captured basic trends, the "flattening" of the graph structure resulted in a significant loss of information regarding how data flows between operations, limiting the model's ability to distinguish between subtle configuration changes.

4. Solution Phase 2: Graph Neural Network

To better capture the relational data inherent in computational graphs, we migrated to a deep learning approach using PyTorch Geometric (PyG). We selected **GraphSAGE** (Graph Sample and Aggregate) as our architecture due to its efficiency in handling large graphs and inductive learning capabilities.

4.1 Data Representation

We implemented a custom TileDataset class to handle the loading of the 'Tile' dataset subset.

- **Inputs:**
    - node_feat: Numeric features of operations.
    - node_opcode: Categorical operation types.
    - edge_index: Connectivity of the graph.
    - config_feat: The compilation configuration vector to be evaluated.
- **Target:** Normalized Runtime. We scaled runtimes using MinMax scaling per graph to make the target distribution consistent across different graph sizes.

4.2 Model Architecture

Our model consists of two main branches that merge into a final predictor:

1. **Graph Encoder (SAGEConv):**
    - **Embedding Layer:** Converts node_opcode (integer ids) into dense vectors (dim=32).
    - **Convolutional Layers:** Three SAGEConv layers with ReLU activation. These layers aggregate information from neighboring nodes to generate a rich representation of the graph structure.
        - Layer sizes: [Input -> 64, 64 -> 128, 128 -> 64].
    - **Pooling:** We utilized global_mean_pool to aggregate all node embeddings into a single Graph Embedding vector.
2. **Prediction Head (Dense MLP):**
    - The Graph Embedding is concatenated with the Configuration Features.
    - This combined vector is passed through a Multi-Layer Perceptron (MLP):
        - Linear (Input -> 128) + LeakyReLU + Dropout (0.2)
        - Linear (128 -> 64) + LeakyReLU
        - Linear (64 -> 1)
    - **Output:** A scalar score representing the predicted efficiency of the configuration.

4.3 Loss Function

We implemented a custom Pairwise Ranking Loss.

Since the goal is ranking, regression on absolute runtime is insufficient.

- We sample pairs of configurations (i, j) for the same graph.
- If Runtime_i < Runtime_j (i is faster), the model should predict Score_i < Score_j.
- Loss Calculation: ReLU(margin - sign(target_diff) * (pred_i - pred_j))
- We added an L1 Regularization term (MAE Loss) to help stabilize the training.

Loss = Loss_{ranking} + 0.1 \times Loss_{L1}

4.4 Training Strategy

- **Optimizer:** AdamW (lr=1e-3, weight_decay=1e-4).
- **Scheduler:** CosineAnnealingLR.

- **Memory Management:** Due to the massive number of configurations per graph, we implemented a MAX_RANKING_SAMPLES cap (2000 pairs) inside the training loop to prevent GPU OOM (Out of Memory) errors.

4.5 Final Results

- **Kaggle Public Score: 0.1857**

The GNN approach successfully leveraged the graph topology, resulting in a performance increase over the XGBoost baseline.

5. Challenges and Constraints

One team member was unable to contribute to the project. As a result, the remaining team members had to cover all aspects of the pipeline.

Impact on Results: We could not perform extensive fine tuning on model parameters, and because of that our final score could be affected.

6. AI Use

We used the help of AI in these tasks:

- Documentation
- Gathering ideas for model architecture

7. Conclusion

We successfully developed a ranking system for AI model runtimes. We demonstrated that while regular ML methods provide a reasonable starting point, structure-aware models could represent data relations better - like in this specific case.