Runtime Performance Prediction for Deep Learning Models with
Graph Neural Network

Yanjie Gao
Microsoft Research
China
yanjga@microsoft.com

Xianyu Gu
Microsoft Research
China
v-xianyg@microsoft.com

Hongyu Zhang
The University of Newcastle
Australia
hongyu.zhang@newcastle.edu.au

Haoxiang Lin*
Microsoft Research
China
haoxlin@microsoft.com

Mao Yang
Microsoft Research
China
maoyang@microsoft.com

ABSTRACT
Deep learning (DL) models have been widely adopted in many application domains. Predicting the runtime performance of DL models such as GPU memory consumption and training time is important for boosting development productivity and reducing resource waste because improper configurations of hyperparameters and neural architectures can result in many failed training jobs or unsatisfactory models. However, the runtime performance prediction of DL models is challenging because of the hybrid DL programming paradigm, complicated hidden factors within the framework runtime, fairly huge model configuration space, and wide differences among models. In this paper, we propose DNNPerf, a novel Graph Neural Network-based tool for predicting the runtime performance of DL models using Graph Neural Network. DNNPerf represents a DL model as a directed acyclic computation graph and incorporates a rich set of performance-related features based on the computational semantics of both nodes and edges. We also propose a new Attention-based Node-Edge Encoder to better encode the node and edge features. DNNPerf is evaluated on thousands of configurations of real-world and synthetic DL models to predict their GPU memory consumption and training time. The experimental results show that DNNPerf achieves accurate predictions, with an overall error of 13.684% for the GPU memory consumption prediction and an overall error of 7.443% for the training time prediction, outperforming all the compared methods.

CCS CONCEPTS
• Software and its engineering → Extra-functional properties.

KEYWORDS
deep learning, AutoML, graph neural network, runtime performance, performance prediction

1 INTRODUCTION
In recent years, deep learning (DL) has been widely adopted in many application domains such as computer vision [28], speech recognition [72], and natural language processing [15]. Like many traditional software systems, DL models are also highly configurable via a set of configuration options for hyperparameters (e.g., the batch size and the dropout) and neural architectures (e.g., the number of layers). To search for the optimal configuration of a DL model that satisfies specific requirements, developers usually run (e.g., via automated machine learning tools) a number of training jobs to explore diverse configurations.

Different model configurations may lead to different quality attributes (i.e., non-functional properties), among which runtime performance (e.g., GPU memory consumption and training time) is one of the most important quality attributes because it directly affects model quality and development productivity. Improper model configurations can unexpectedly degrade runtime performance and result in many failed jobs or unsatisfactory models. For instance, an overlarge batch size causes a job to exhaust GPU memory and raise an OOM (out-of-memory) exception. According to a recent empirical study on 4,960 DL job failures in Microsoft [75], 8.8% of the failed jobs were caused by GPU OOM, which accounts for the largest category in all DL specific failures. Another example is that a sophisticated neural architecture increases the computation load and makes the training time longer than expected, as a result the job has to be stopped early because of limited budget. Even worse, in the automated machine learning (AutoML) scenario, other tens or hundreds of jobs with the same batch size or similar neural architectures can experience the same issues and fail. Therefore, predicting the runtime performance of a DL model ahead of job execution is critical for boosting development productivity and reducing resource waste.

In literature, there is already much research work for estimating the runtime performance of programs [2, 3, 5, 12, 64, 67] and deployed systems [24–26, 56–58] using program analysis and machine learning (ML) techniques. Recently, some authors advanced the estimation effort to DL models [19, 33, 49, 73]. However, these works either cannot be directly applied or have limitations in precise prediction for the following challenges:

(1) DL frameworks provide a hybrid programming paradigm: developers only invoke high-level interfaces to construct DL models while low-level computational operations are implemented with proprietary NVIDIA CUDA, cuDNN, or cuBLAS APIs. Such a paradigm hides the internal implementation details and thus makes it hard to model the runtime performance accurately.

(2) There are many complicated hidden factors such as garbage collection and operator execution order within the framework runtime, which observably affect a model’s runtime...
performance. However, understanding and extracting such hidden factors can be very difficult, time-consuming, and error-prone because they are volatile and fast-changing with the rapid development of DL frameworks.

(3) The configuration space of a DL model is fairly huge and there exist wide differences among various kinds of DL models; therefore, it is challenging to design a general ML prediction model from limited samples with high accuracy.

In this paper, we propose DNNPerf, a novel Graph Neural Network-based tool for predicting the runtime performance of DL models with Graph Neural Network (GNN) [52]. Our key observation is that a DL model can be represented as a directed acyclic computation graph [22]. Each node denotes a computational operation called an operator (e.g., matrix addition), and an edge delivers a tensor and specifies the execution dependency between two nodes. The algorithmic execution of the model is then represented as iterative forward and backward propagation on such a computation graph. Node features include the operator type (e.g., Conv2D), hyperparameters (e.g., the kernel size), number of floating-point operations (FLOPs), sizes of input/output/weight/temporary tensors, etc. Edge features consist of the edge type (forward or backward) and size of the delivered tensor. Relevant parameters of target devices such as floating-point operations per second (FLOPs), memory capacity, and bandwidth are also extracted as node or edge features. Moreover, we propose a new Attention-based Node-Edge Encoder to better encode the node and edge features by adapting ideas of Graph Attention Network (GAT) [66] and Edge Enhanced Graph Neural Network (EGNN) [21].

We have implemented DNNPerf as a prediction model trained from the runtime performance data of various kinds of DL models. The full dataset includes: (1) 10,238 model configurations of ten real-world DL models (e.g., VGG [60], ResNet [28], and Vanilla RNN [50]) whose representative hyperparameters’ values are randomly generated within reasonable ranges; (2) 8,403 model configurations whose neural architectures are randomly synthesized by Neural Architecture Search (NAS) [47] algorithms. We explore two typical prediction tasks for GPU memory consumption and training time; however, our approach can be easily applied to other tasks such as predicting the inference time and GPU power consumption. The experimental results show that DNNPerf outperforms all five compared methods (e.g., BiRNN [5] and GBDT [43]) with an overall mean relative error of 13.684% for the GPU memory consumption prediction and an overall mean relative error of 7.443% for the training time prediction, confirming its effectiveness.

We summarize our main contributions as follows:

(1) We propose a novel Graph Neural Network-based approach for accurately predicting the runtime performance of deep learning models.

(2) We design a rich set of node and edge features to capture performance-related factors (e.g., compute, I/O, etc.). We also propose a novel Attention-based Node-Edge Encoder to better encode the computation graph of a DL model.

(3) We implement an end-to-end tool named DNNPerf and demonstrate its practical effectiveness through a set of experiments on thousands of configurations of real-world and synthetic DL models.

```python
from tensorflow.keras import layers, models

t = model = models.Sequential()

t.add(layers.Conv2D(filters=32, kernel_size=(3, 3),
activation='relu', input_shape=(224, 224, 3)))
t.add(layers.AveragePooling2D(pool_size=(2, 2)))
t.add(layers.Conv2D(filters=64, kernel_size=(3, 3),
activation='relu'))
t.add(layers.AveragePooling2D(pool_size=(2, 2)))
t.add(layers.Flatten())
t.add(layers.Dense(units=1000, activation='softmax'))
t.compile(optimizer='SGD',
loss='categorical_crossentropy')
t.fit(train_images, train_labels, batch_size, epochs=60)
```

Figure 1: A sample Keras MNIST model constructed with the Conv2D, AvgPool2D, Flatten, Dense, and Softmax operators.

Figure 2: Computation graph for training the above model.

## 2 BACKGROUND

### 2.1 Deep Learning Models and Computation Graphs

Being essentially mathematical functions, DL models are formalized by frameworks like TensorFlow [1] and PyTorch [45] as tensor-oriented computation graphs (i.e., directed acyclic graphs). The inputs and outputs of a computation graph or a graph node are tensors (multidimensional arrays of numerical values). Each node denotes a mathematical operation called an operator (e.g., matrix addition). An edge pointing from one output of node $A$ to one input of $B$ delivers a tensor and specifies their execution dependency. A DL model usually provides a set of configuration options for its hyperparameters (e.g., the batch size and the dropout) and neural architecture (e.g., the number of layers).

Figure 1 shows the snippet of a sample MNIST [14] training program written with TensorFlow’s Keras [11] API. The program constructs a sequential model with the framework built-in Conv2D (2D convolution with a $3 \times 3$ kernel size), AvgPool2D (2D average pooling with a $2 \times 2$ pool size), Flatten (collapsing the input tensor into one dimension), Dense (fully-connected layer with 64 units), and Softmax (normalizing “the probability distribution over
different classes” [22]) operators (lines 3–9). The above number of output channels (denoted by the variable # filters), kernel size, number of units, etc. are hyperparameters that control the training process. For training, the framework constructs a computation graph shown in Figure 2 with concrete hyperparameter values and then applies iterative forward and backward propagation on such a graph to learn the optimal learnable parameters (aka weights). Operators in the middle of Figure 2 (e.g., AvgPool12d_BP) are automatically generated by TensorFlow for backward propagation. Input data (Input_Data) is fed through the computation graph and manipulated by the developer-specified operators (on the left of Figure 2). Input labels (Input_Labels) and produced outputs of Dense are then propagated backward to compute the gradients of weights. Finally, an optimizer [34] updates the weights to minimize the loss (e.g., the mean squared error between actual and predicted outputs), marking the end of one training iteration.

2.2 Graph Neural Networks

The graph is a common data structure for representing elements and their relations, and it is widely used in data analysis. Recently, there are emerging requirements to apply DL techniques to learning from graph data. However, existing models such as Convolutional Neural Network (CNN) [39] and Recurrent Neural Network (RNN) [30] have limitations in handling graphs because of the irregularity caused by unequal node neighbors. Graph Neural Network (GNN) [52] is then proposed to efficiently address this issue.

Graph Convolutional Network (GCN) [35] is the first popular GNN, which learns localized spatial features by convolutional filters. Graph Attention Network (GAT) [66] uses masked self-attentional layers to address the shortcomings of prior GCN-based methods. GAT does not require costly matrix operations or knowing the graph structure upfront by specifying different weights to different nodes in a neighborhood. In some datasets, edge features also contain important graph information; however, they are not adequately utilized by GCN and GAT. Edge Enhanced Graph Neural Network (EGNN) [21] builds a framework for a family of new GNNs that can sufficiently exploit the features of edges (including both undirected and multi-dimensional ones). In our work, we adopt GNN and propose a novel Attention-based Node-Edge Encoder for better predicting the runtime performance of DL models.

3 DNNPERF: A GNN-BASED APPROACH

3.1 Problem Formulation

As mentioned before, a DL model \( M \) is represented as the following directed acyclic graph (DAG) [22]:

\[
M = (\{u_i\}_{i=1}^n, \{e_{ij} = (u_i, u_j)\}_{i,j \in [1,n]}, \{h_{pk}\}_{k=1}^m).
\]

Each node \( u_i \) is an operator such as the above Conv2D and AvgPool12d. A directed edge \( e_{ij} \) pointing from node \( u_i \) to \( u_j \) delivers an output tensor of \( u_i \) to \( u_j \) as input and also specifies the execution dependency. Each \( h_{pk} \) is a hyperparameter of some operator such as the batch size and input tensor shape, whose domain is denoted as \( B_k \). The model \( M \) is assumed to be deterministic without control flow operators (e.g., loops and conditional branches) or dynamic structural changes; hence, the execution flow and runtime performance across different training iterations should be identical. We use \( M(b_1 \in B_1, b_2 \in B_2, \cdots, b_m \in B_m) \) to denote one model configuration of \( M \). Then, \( \Delta_M \), the configuration space of \( M \), is defined as follows:

\[
\Delta_M = \{ M(b_1, b_2, \cdots, b_m) \mid b_k \in B_k \text{ for } k \in [1, m] \}.
\]

A runtime specification describes the environment execution, which currently contains the target device’s information such as floating-point operations per second (FLOPS) and memory capacity. All the actual runtime specifications constitute a space denoted by \( S \). Then, the runtime performance prediction tasks for \( M \) can be formulated as a family of real regression functions:

\[
f_i : \Delta_M \times S \rightarrow \mathbb{R}.
\]

Each \( f_i \) is a performance function, taking a model configuration and a runtime specification as its parameters. Hence, our GNN-based approach trains two typical performance functions that return the GPU memory consumption and training time, respectively.

3.2 Workflow

Figure 3 illustrates the workflow of DNNPerf. It accepts a DL model file, a model configuration specification, and a runtime specification as input and reports the runtime performance values. DNNPerf implements a front-end parser using the framework’s built-in model deserialization APIs, which is responsible for reading the input model file and reconstructing the corresponding computation graph. The model configuration specification includes hyperparameter values. The runtime specification currently contains the target device’s information (e.g., GPU FLOPS).

DNNPerf traverses the computation graph to automatically generate the node and edge features, strictly following the operator execution order that is predefined by reference to the framework implementations [38, 48]. It employs a novel Attention-based Node-Edge Encoder (ANEE) to embed the nodes and edges in several rounds (Section 3.4). To improve the prediction accuracy and reduce the amount of training data, DNNPerf further utilizes the semantics of both nodes and edges. The details are summarized in Table 1 and Table 2. Finally, DNNPerf aggregates the updated feature vectors and uses a multilayer perceptron (MLP) [27] to output the result.

3.3 Model Encoding

We refer to the framework source code, carefully identify hidden factors, and then design a set of effective performance-related features based on the computational semantics of both nodes and edges. First, DNNPerf uses an operator’s type, hyperparameters, tensor sizes, GPU memory consumption, and FLOPs to capture
Table 1: Node features.

<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operator ((h^r))</td>
<td>Operator Type</td>
<td>Type of the operator (e.g., Conv2D and AvgPool2D)</td>
</tr>
<tr>
<td>Hyperparameter ((h^p))</td>
<td>Hyperparameter</td>
<td>Type and value of each hyperparameter of the operator (e.g., kernel size and channel size)</td>
</tr>
<tr>
<td>Tensor ((h^d))</td>
<td>Weight Tensor Size</td>
<td>Sizes of weights (aka learnable parameters) and weight gradients (computed under backward propagation for updating weights)</td>
</tr>
<tr>
<td></td>
<td>In/Out Tensor Size</td>
<td>Sizes of inputs, outputs, and output gradients (computed under backward propagation for calculating weight gradients)</td>
</tr>
<tr>
<td></td>
<td>Temporary Tensor Size</td>
<td>Sizes of temporary variables used by the operator</td>
</tr>
<tr>
<td>Computation Cost ((h^c))</td>
<td>FLOPs</td>
<td>Number of floating-point operations of the operator</td>
</tr>
<tr>
<td>Device ((h^d))</td>
<td>FLOPS</td>
<td>Peak floating-point operations per second</td>
</tr>
<tr>
<td></td>
<td>Memory Capacity</td>
<td>Total amount of the device memory</td>
</tr>
</tbody>
</table>

Table 2: Edge features.

<table>
<thead>
<tr>
<th>Category</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Edge ((e^i))</td>
<td>Edge Type</td>
<td>Type of the edge (&quot;forward&quot; or &quot;backward&quot;)</td>
</tr>
<tr>
<td>Tensor ((e^t))</td>
<td>Tensor Size</td>
<td>Size of the delivered tensor</td>
</tr>
<tr>
<td>Device ((e^d))</td>
<td>Bandwidth</td>
<td>Bandwidth for accessing the delivered tensor</td>
</tr>
</tbody>
</table>

operator-level factors. Secondly, DNNPerf uses an edge’s type and size of the delivered tensor to capture the operator execution order, tensor liveness [19], and I/O costs. Lastly, some device features such as FLOPS, memory capacity, and bandwidth are also extracted and associated with the nodes and edges to capture runtime factors such as garbage collection, memory access, and distributed communication.

3.3.1 Initial Node Encoding. For each node of a computation graph (such as the graph in Figure 2), we associate it with a real-valued feature vector \(h \in \mathbb{R}^{N_1}\), where \(N_1\) is the number of designed node features. \(h\) is composed by several sub-feature vectors such that \(h = F(h^r \parallel h^p \parallel h^d \parallel h^e \parallel h^c)\):

(1) "\(\parallel\)" is the vector concatenation operation, and \(F\) is the function for raw feature encoding and vector shape alignment.
(2) \(h^r\) represents the operator type (e.g., Conv2D).
(3) \(h^p\) is a vector for encoding the operator’s hyperparameters, whose length equals the number of total hyperparameters.
(4) \(h^d\) stores the sizes of the weight, input, output, and temporary tensors.
(5) \(h^e\) denotes the computation cost. Currently, we use FLOPs.
(6) \(h^c\) encodes the device information such as GPU memory capacity and FLOPS, which affects the operator execution.

Details are listed in Table 1. For categorical features (e.g., operator type), we adopt the One-Hot encoding [70] strategy.

We take Conv2D, the 2D convolution operator [39], as an example to illustrate how to compute the initial values of \(h^r\) (i.e., FLOPs) and \(h^d\). Let \(sz\) be the size of the data type in bytes (e.g., 4 bytes for float data). The following hyperparameters are needed: kernel size (\(ks\)), stride (\(sd\)), padding (\(pad\)), and dilation (\(dil\)). We assume that each of them is an array of two positive integers that specify the height- and width-related information, respectively. Suppose that Conv2D receives an input tensor whose shape is \([N, C_i, H_i, W_i]\) (i.e., a batch of \(N\) input samples with height \(H_i\), width \(W_i\), and \(C_i\) input channels). Conv2D produces an output tensor whose shape is \([N, C_o, H_o, W_o]\). \(C_o\) is the number of output channels. The output height (\(H_o\)) and width (\(W_o\)) can be calculated as follows [62]:

\[
H_o = \left\lfloor \frac{H_i + 2 \times pad[0] - dil[0] \times (ks[0] - 1) - 1}{sd[0]} + 1 \right\rfloor, \\
W_o = \left\lfloor \frac{W_i + 2 \times pad[1] - dil[1] \times (ks[1] - 1) - 1}{sd[1]} + 1 \right\rfloor.
\]

Now we show how the FLOPs under forward propagation [8] and tensor sizes (in bytes) of Conv2D are computed:

\[
FLOPs(\text{Conv2D}) = 2 \times (C_i \times H_i \times W_i + 1) \times N \times C_o \times H_o \times W_o, \\
IT(\text{Conv2D}) = sz \times N \times C_i \times H_i \times W_i, \\
OT(\text{Conv2D}) = sz \times N \times C_o \times H_o \times W_o, \\
WT(\text{Conv2D}) = sz \times (C_i \times H_i \times W_i + 1) \times C_o.
\]

\(IT\), \(OT\), and \(WT\) are functions that return the sizes of the input, output, and weight tensors, respectively.

Currently, DNNPerf supports 70+ common types of DL operator.

3.3.2 Initial Edge Encoding. For each edge of a computation graph, we associate it with a real-valued feature vector \(e \in \mathbb{R}^{N_2}\), where \(N_2\) is the number of designed edge features. \(e\) is composed of several sub-feature vectors such that \(e = e^t \parallel e^d \parallel e^c\):

(1) \(e^t\) represents the edge type. Currently, its initial value is either "forward" or "backward" depending on which propagation stage the delivered tensor is used. \(e^t\) tries to capture the tensor liveness that affects the GPU memory consumption.
(2) \(e^d\) and \(e^c\) denote the size of the delivered tensor and device bandwidth, respectively. These two features affect the I/O access latency, which in turn affects the training time.

Details are listed in Table 2.

3.3.3 Normalization. Since feature values vary widely, DNNPerf scales the range of each feature to the interval \([0, 1]\) by MinMaxScaler [53]. DNNPerf does not use the neighbor-level normalization.
from EGNN [21] because the average node degree of a computation graph is much smaller than that of previously studied large graphs such as the social network graph. Suppose $g$ is the computation graph of a model configuration in the training dataset $DS$ and $u$ is a node of $g$. We use $h_{g,u}$ and $h_{g,u,i}$ to denote the initial feature vector of node $u$ and its $i$-th dimension, respectively. Then, the normalized node feature vector $\hat{h}_{g,u} = [\hat{h}_{g,u,1}, \cdots, \hat{h}_{g,u,N_l}]$ is produced as follows:

$$\hat{h}_{g,u,i} = \frac{h_{g,u,i} - \min S_i}{\max S_i - \min S_i}, \quad S_i = \{h_{g',u',i} | g' \in DS \land u' \in g'\}.$$  

Similarly, we use $e_{g,l}$ and $e_{g,l,i}$ to denote the initial feature vector of edge $l$ and its $i$-th dimension, respectively. Then, the normalized edge feature vector $\hat{e}_{g,l} = [\hat{e}_{g,l,1}, \cdots, \hat{e}_{g,l,N_l}]$ is produced as follows:

$$\hat{e}_{g,l,i} = \frac{e_{g,l,i} - \min S_i}{\max S_i - \min S_i}, \quad S_i = \{e_{g',l',i} | g' \in DS \land l' \in g'\}.$$  

### 3.4 Attention-based Node-Edge Encoder

To achieve high prediction accuracy while taking into account efficiency, we propose a novel Attention-based Node-Edge Encoder (ANEE) for multi-dimensional features, which is shown in the middle part of Figure 3.

Firstly, the original GCN and GAT models are restricted to only one-dimensional, real-valued edge features, which cannot capture the effects of an edge. In comparison, ANEE could capture both the node and edge features. Secondly, the operators of a DL model contain less fan-out, and the costs of operators are different. DNNPerf uses a global normalizer that could achieve better efficiency and accuracy. We also use weight sharing design for the encoder parameter matrix (e.g., $W_u$, $W_e$, and $W_m$) to reduce the number of parameters, which could benefit the training/inference time and GNN model size. Therefore, compared with EGNN [21], ANEE is lightweight and can achieve better efficiency.

ANEE performs multiple rounds of computation to encode the nodes and edges. Suppose $g$ is the computation graph of a model configuration, $u$ is a node, and $l = (u, u')$ is an edge pointing from the source node $u$ to the target node $u'$. We use $h_{g,u}^0$ and $e_{g,l}^0$ to denote the computed feature vectors of node $u$ and edge $l$ in the $i$-th ($i > 0$) round, respectively. $h_{g,u}^0$ and $e_{g,l}^0$ are the initial normalized feature vectors mentioned before.

First, ANEE computes a preliminary result of $h_{g,u}^i$ (denoted by $\hat{h}_{g,u}$) locally as follows:

$$\text{LeakyReLU}(x) = \max(0,x) - \alpha \times \max(0,-x),$$  
$$\hat{h}_{g,u}^i = \text{LeakyReLU}(W_u^i \cdot h_{g,u}^{i-1}).$$

The activation function LeakyReLU is the leaky version of Rectified Linear Unit (ReLU), $\alpha$ is the slope coefficient that defaults to 0.01, and $W_u$ is a parameter matrix.

Next, ANEE updates $e_{g,l}^i$ as follows:

$$e_{g,l}^i = \sigma (a^T \cdot (\hat{T}_{g,l}^e \parallel \hat{T}_{g,l}^a) \times W_e \times e_{g,l}^{i-1}).$$

$\sigma$ is the sigmoid activation function, and $W_e$ is a parameter matrix. The attention mechanism uses a single-layer feedforward neural network that is denoted by a weight vector $a \in \mathbb{R}^{2 \times N_l}$. $a^T$ is the transpose of $a$.

Then, ANEE gathers the information of $u$’s neighbors and the associated edges to compute the final $h_{g,u}^i$ as follows:

$$f(u', l') = \text{Softmax}(W_m \times e_{g,l'}) \times \hat{h}_{g,u}^i,$$

$$h_{g,u}^i = \text{LeakyReLU}(\sum_{l'=(u', u')} f(u', l')).$$

$u'$ is a neighbor node of $u$, $l'$ is their associated edge, and $W_m$ is a parameter vector. Softmax makes coefficients easily comparable across different features and normalizes them across all choices of features.

Finally, DNNPerf performs a global aggregation by summing up all the node feature vectors. The aggregated result will be fed to the predictor (an MLP layer) to generate the runtime performance value.

### 3.5 Loss Function

The GNN-based prediction of GPU memory consumption and training time can be reduced to a regression problem. We use the mean squared error (MSE) to design the following loss function:

$$L = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2.$$  

$N$ is the number of model configurations in the training set; $\hat{y}_i$ and $y_i$ are the real and predicted runtime performance values of the $i$-th model configuration, respectively. We actually tried mean relative error (MRE), but found no better results, so we adopted the more commonly used MSE.

### 4 EXPERIMENTAL SETUP

#### 4.1 Datasets

We collect 18,641 model configurations implemented with TensorFlow v1.13.1 and divide them into two groups. The first is an HPO (Hyperparameter Optimization) [13] dataset that includes 10,238 configurations of ten real-world models with various hyperparameter combinations. In this dataset, the configurations of five randomly selected DL models (LeNet [40], ResNet-V1 [28], Inception-V3 [61], Vanilla RNN [50], and LSTM [29]) are used for normal training, validation, and test, in a proportion of 70:10:20. The rest models (AlexNet [37], VGG [60], OverFeat [55], ResNet-V2 [28], and GRU [9]) are unseen to DNNPerf because their configurations do not exist in the training set and are only used to evaluate the generalization ability of DNNPerf (Section 5.2). These ten models are representative in areas of computer vision, speech recognition, and natural language processing, among which some (such as RNN, LSTM, and GRU) are also widely used in various DL for software engineering applications [23, 32, 42, 74]. The second is a NAS (Neural Architecture Search) [47] dataset that contains 8,403 configurations of NAS-searched models.

#### HPO dataset

We choose representative DL models from a TensorFlow public benchmark [59] and adopt a random strategy based on the provided APIs to generate the model configurations. We consider the batch size (16, 32, 48, 64, 80, 96, 112, and 128), number of input channels (1, 3, 5, 7, and 9), input height (224), and input width (224) as the hyperparameters. For VGG, we use different domains of the batch size (interval [16, 128]) and number of input channels (interval [1, 10]). For AlexNet, OverFeat, and LeNet, we...
To compare with the GNN-based model of DNNPerf, we consider Table 4.

Dense (4) number of units (128, 256, 512, 768, and 1,024) for (16, 32, 64, 96, 128, 160, 256, 512, and 1,024) and kernel size (1, 3, 5, 7, and 9); (2) number of output channels (1, 3, 5, 7, and 9); (3) kernel size (1×1, 3×3, and 5×5) for Conv2D; (3) kernel size (2×2) for MaxPool2D; (4) number of units (128, 256, 512, 768, and 1,024) for Dense. We use the "same" padding for each Conv2D operator to avoid shape errors. The cells also contain randomly generated operators as a hyperparameter whose domain is set to an interval [default × 0.5, default × 2]. For VGG model configurations, we randomly generate them from VGG-11, VGG-16, and VGG-19 models. Similarly, for ResNet-V1 and ResNet-V2 model configurations, we randomly generate them from ResNet-50, ResNet-101, ResNet-152, and ResNet-200 models. For Inception-V3, we assign random numbers to the Inception API's min_depth, depth_multiplier, and end_point parameters. The statistics of the HPO dataset is shown in Table 3.

NAS dataset. We use the NAS search space illustrated in Figure 4 to generate the model configurations. The model skeleton consists of combinations of operator cells (e.g., Cell Type 1, Cell Type 2, etc.). Each cell contains a diverse combination of different operators (e.g., Conv2D, Dense, and MaxPool2D) and hyperparameters. The hyperparameters include: (1) batch size (16, 32, 64, 80, 96, 128, and 1,024), input height (224), input width (224), and number of input channels (1, 3, 5, 7, and 9); (2) number of output channels (16, 32, 64, 96, 128, 160, 256, 512, and 1,024) and kernel size (1×1, 3×3, and 5×5) for Conv2D; (3) kernel size (2×2) for MaxPool2D; (4) number of units (128, 256, 512, 768, and 1,024) for Dense. We use the "same" padding for each Conv2D operator to avoid shape errors. The cells also contain randomly generated Dropout and MaxPool2D operators. The statistics of the NAS dataset is shown in Table 4.

We measure the runtime performance (execution time and peak GPU memory consumption) of three iterations and compute average values. The first few iterations are bypassed to make sure that the training has been stable.

4.2 Baselines

To compare with the GNN-based model of DNNPerf, we consider the following models as the baselines:

1. BiRNN (Bidirectional RNN) [3] is a two-layer bidirectional RNN [30] with the LSTM [29] cell. Node features include the name, type, and computation cost of the operator, which are computed in the same way as the initial node features of DNNPerf. We feed the node feature vectors to BiRNN as the input sequence according to the topological order of the computation graph. This baseline allows us to identify how effective is GNN in capturing the data flow information of the computation graph.

2. ARNN (Adjacency BiRNN) [3] is an extension to BiRNN [3]. The feature vector of a node is updated by computing the average of vectors of predecessors, successors, and itself. ARNN is a stronger baseline since it takes some structural information of a computation graph into account.

3. MLP (Multilayer Perceptron) [27] is a traditional machine learning model. Prior work [33] used it to predict the operator execution time. We evaluate MLP with the same node features that DNNPerf uses.

4. GBDT (Gradient Boost Decision Tree) [43] is also a traditional machine learning model. It was used in the work of Chen et al. [7] for encoding loop programs. GBDT uses the same features as MLP.

5. BRP-NAS (Binary Relation Predictor-based NAS) [18] is a graph convolutional network for the NAS dataset. It encodes only the graph topology without considering the runtime factors (e.g., compute and I/O) of nodes and edges.

We have implemented BiRNN, ARNN, MLP, and BRP-NAS with PyTorch v1.5.0 [45]; for GBDT, we used the built-in functionality of scikit-learn v0.20.3 [46]. After tuning these models, we select the following hyperparameter values. For BiRNN and ARNN, learning rate is 0.0001, layer size is 1, and hidden size is 512. For MLP, learning rate is 0.001, and the number of units of each layer is 512, 128, 16, and 1, respectively. For GBDT, the learning rate is 0.01, the max tree depth is 30, and the number of trees is 200. For BRP-NAS, the learning rate is 0.0001.

4.3 Implementation and Settings of DNNPerf

We have implemented DNNPerf with DGL (Deep Graph Library) [68] v0.4.3, which is a package built on top of PyTorch for easy implementation of graph neural networks.

We tune the hyperparameters of our GNN-based model on the validation dataset. The MLP uses hidden layers of 512, 128, and 16
units (the size of final output is 1). The number of message-passing rounds of ANEE is set to 3 (training time prediction) or to 1 (GPU memory consumption prediction). We use the Adam [34] optimizer with default parameter values ($\alpha = 0.9$ and $\beta = 0.999$), set batch size to 64 and initial learning rate to 0.0001, and train our model for 250 epochs (training time prediction) or 200 epochs (GPU memory consumption prediction). After tuning, our model scales to about 737.2 graphs per second during the training and about 1,800 graphs per second during the inference. These graphs have 73.7 nodes and 119.7 edges on average.

4.4 Evaluation Metrics
To assess the effectiveness of DNNPerf, we use the mean relative error (MRE) and root-mean-square error (RMSE):

$$\text{MRE} = \frac{\sum_{i=1}^{N} \hat{y}_{i} - y_{i}}{N} \times 100\%,$$
$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{N} (\hat{y}_{i} - y_{i})^2}{N}}.$$

$N$ is the number of data samples in the test set; $\hat{y}_{i}$ and $y_{i}$ are the predicted and real performance values of the $i$-th sample, respectively. We choose these two metrics because they are widely used to measure the accuracy of prediction models [19, 26, 33]. The smaller the error, the higher the prediction accuracy. For the prediction of GPU memory consumption and training time, the units of RMSE values are gigabyte (GB) and millisecond (ms), respectively.

5 EVALUATION
We evaluate our proposed approach by addressing the following Research Questions (RQs):

RQ1: How effective is DNNPerf in predicting runtime performance?
RQ2: How general is DNNPerf to unseen DL models?
RQ3: How effective is DNNPerf in the ablation study?

Our experiments are conducted on an Azure Standard ND24s virtual machine [44], which has 24 Intel Xeon E5-2690V4 (2.60 GHz, 35M Cache) vCPUs, 448 GB main memory, and 4 NVIDIA Tesla P40 (24 GB GDDR5X memory) GPUs, running Ubuntu 18.04.

5.1 RQ1: How effective is DNNPerf in predicting the runtime performance?
In this section, we compare DNNPerf with five baselines (Section 4.2) on the same test set. Table 5 shows the MRE and RMSE values of all the six approaches for predicting the runtime performance. On average (see the “Overall” rows), DNNPerf achieves 7.443% MRE and 58.5 RMSE for the training time prediction and 13.684% MRE and 1.806 RMSE for the GPU memory consumption prediction, which exceed the best MRE/RMSE values of the baselines: 17.982%/106.2 from MLP and 15.338%/119.7 edges on average.

More specifically, DNNPerf outperforms BiRNN by 13.363%/92.9, ARNN by 13.497%/55.4, MLP by 10.539%/47.7, GBDT by 23.57%/64.1, and BRP-NAS by 87.805%/276.9 in terms of MRE/RMSE for the training time prediction; DNNPerf outperforms BiRNN by 8.618%/1,011, ARNN by 11.813%/1,076, MLP by 7.776%/0.529, GBDT by 1.908%/0.407, and BRP-NAS by 19.97%/2.338 in terms of MRE/RMSE for the GPU memory consumption prediction. We also experiment on the HPO models and the NAS models separately. The results demonstrate that DNNPerf still excels each of the baselines and confirm its effectiveness. We think the reason is that DNNPerf captures richer and more diverse performance-related information from both the operators/hyperparameters and the neural architecture than the other approaches.

Table 6 shows the prediction results on the HPO dataset. The test model configurations are derived from five representative real-world models: LeNet, ResNet-V1, Inception-V3, Vanilla RNN, and LSTM, which are already seen to DNNPerf because the training dataset contains their configurations. We choose various hyperparameters and larger value ranges (Section 4.1) to bring the diversity of the runtime performance. DNNPerf achieves satisfactory precision and outperforms the baseline approaches in most cases by

<table>
<thead>
<tr>
<th>Model Name</th>
<th>Prediction of Training Time</th>
<th>MRE (%)</th>
<th>RMSE (ms)</th>
<th>DNNPerf</th>
<th>BiRNN</th>
<th>ARNN</th>
<th>MLP</th>
<th>GBDT</th>
<th>BRP-NAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MRE (%)</td>
<td>7.443</td>
<td>20.806</td>
<td>20.940</td>
<td>17.982</td>
<td>31.013</td>
<td>95.284</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RMSE (ms)</td>
<td>58.5</td>
<td>151.4</td>
<td>119.9</td>
<td>106.2</td>
<td>122.6</td>
<td>335.4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HPO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MRE (%)</td>
<td>7.813</td>
<td>21.560</td>
<td>22.220</td>
<td>19.405</td>
<td>33.113</td>
<td>86.715</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RMSE (ms)</td>
<td>51.9</td>
<td>107.7</td>
<td>95.2</td>
<td>94.7</td>
<td>120.4</td>
<td>294.3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NAS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RMSE (ms)</td>
<td>75.7</td>
<td>242.6</td>
<td>159.5</td>
<td>136.4</td>
<td>129.3</td>
<td>441.0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Model Name</th>
<th>Prediction of GPU Memory Consumption</th>
<th>MRE (%)</th>
<th>RMSE (GB)</th>
<th>DNNPerf</th>
<th>BiRNN</th>
<th>ARNN</th>
<th>MLP</th>
<th>GBDT</th>
<th>BRP-NAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RMSE (GB)</td>
<td>86.715</td>
<td>17.982</td>
<td>13.684</td>
<td>25.497</td>
<td>30.974</td>
<td>40.452</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HPO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RMSE (GB)</td>
<td>80.4</td>
<td>18.982</td>
<td>12.350</td>
<td>22.104</td>
<td>14.150</td>
<td>44.144</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NAS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RMSE (GB)</td>
<td>1.800</td>
<td>2.813</td>
<td>2.369</td>
<td>1.955</td>
<td>2.437</td>
<td>4.991</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 5: Overall experimental results.

Table 6: Experimental results on HPO models.
a large margin. The overall MRE/RMSE improvement over the baseline approaches ranges from 14.014%/23.4 to 60.548%/190 in predicting the training time, and from 3.992%/0.38 to 16.583%/3.151 in predicting the GPU memory consumption. We notice that the overall RMSE value of GBDT for the GPU memory consumption prediction is a little better than DNNPerf since GBDT has advantages on configurations of models whose operator types are relative single (e.g., LSTM and ResNet-V1). The experimental results demonstrate that DNNPerf is very stable to diverse hyperparameter options.

The NAS dataset exhibits a great diversity of neural architectures because the numbers of nodes and edges are broadly distributed. We divide the dataset into several subsets according to value ranges of the node number (internals [20, 29], [30, 39], [40, 49], and [50, 70]) and the edge number (internals [35, 49], [50, 64], [65, 79], and [80, 115]) separately. Then, we conduct one experiment for each subset and show the prediction results using line charts in Figures 5 and 6.

5.2 RQ2: How general is DNNPerf to unseen DL models?

In this section, we evaluate the generalization ability of DNNPerf, which is important for our tool to be used in practice. We construct a test set that includes model configurations of AlexNet, VGG, OverFeat, ResNet-V2, and GRU. These five models are unseen to DNNPerf; that is to say, the training set does not contain any configurations of them.

After tuning, our model scales to about 737.2 graphs per second during the training and about 1,800 graphs per second during the inference. These graphs have 73.7 nodes and 119.7 edges on average.

Table 7 shows the MRE and RMSE values, in which the overall results of DNNPerf outperform the baseline approaches. The improvement of MRE/RMSE ranges from 10.984%/40.4 to 83.509%/253.9 for the training time prediction task, and from 0.233%/0.421 to 12.472%/2.202 for the GPU memory consumption prediction task. BRP-NAS does not perform well because it is designed only for specific NAS search space without modeling the runtime performance-related features. Other methods do not encode the complete computation graph and hidden runtime factors (e.g., operator scheduling and tensor liveness), resulting in relatively poor results. In summary, the experimental results demonstrate that DNNPerf is achieves good generalization ability.

5.3 RQ3: How effective is DNNPerf in the ablation study?

In this section, we run additional experiments to study the impact of alternative design choices of DNNPerf such as the graph encoding and feature normalization methods:

1. DNNPerf-GAT, which replaces the ANEE layer (Section 3.4) by GAT [66]. Since GAT cannot perform local encoding on edge features through message passing, this experiment is to evaluate the effectiveness of ANEE.
2. DNNPerf-StandardScaler, which uses StandardScaler [54] to normalize features “by removing the mean and scaling to unit variance” [46]. This experiment is to evaluate which normalization method is more effective.
3. DNNPerf-NoTensorCost, which excludes the tensor size and computation cost features described in Table 1 and Table 2. This experiment is to evaluate the effectiveness of runtime performance-related features.
Table 7: Experimental results on unseen models.

<table>
<thead>
<tr>
<th>Model Name</th>
<th>Prediction of Training Time</th>
<th>Metrics</th>
<th>DNNPerf</th>
<th>BRNN</th>
<th>ARNN</th>
<th>MLP</th>
<th>GBDT</th>
<th>BRP-NAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall</td>
<td>MRE (%)</td>
<td>7.651</td>
<td>19.771</td>
<td>19.771</td>
<td>18.635</td>
<td>34.504</td>
<td>91.160</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RMSE (ms)</td>
<td>55.1</td>
<td>111.8</td>
<td>95.5</td>
<td>99.1</td>
<td>131.2</td>
<td>309.0</td>
<td></td>
</tr>
<tr>
<td>AlexNet</td>
<td>MRE (%)</td>
<td>11.942</td>
<td>22.545</td>
<td>18.394</td>
<td>21.633</td>
<td>85.414</td>
<td>291.101</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RMSE (ms)</td>
<td>13.6</td>
<td>27.1</td>
<td>28.6</td>
<td>26.5</td>
<td>90.4</td>
<td>264.6</td>
<td></td>
</tr>
<tr>
<td>VGG</td>
<td>MRE (%)</td>
<td>7.295</td>
<td>13.308</td>
<td>10.099</td>
<td>16.817</td>
<td>12.593</td>
<td>41.648</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RMSE (ms)</td>
<td>84.2</td>
<td>164.6</td>
<td>92.3</td>
<td>134.4</td>
<td>148.6</td>
<td>339.0</td>
<td></td>
</tr>
<tr>
<td>OverFeat</td>
<td>MRE (%)</td>
<td>7.462</td>
<td>20.735</td>
<td>24.892</td>
<td>24.892</td>
<td>58.530</td>
<td>65.409</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RMSE (ms)</td>
<td>24.3</td>
<td>90.7</td>
<td>77.7</td>
<td>77.6</td>
<td>178.6</td>
<td>154.4</td>
<td></td>
</tr>
<tr>
<td>ResNet-V2</td>
<td>MRE (%)</td>
<td>5.690</td>
<td>7.232</td>
<td>10.580</td>
<td>5.892</td>
<td>9.933</td>
<td>65.049</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RMSE (ms)</td>
<td>91.1</td>
<td>98.1</td>
<td>118.1</td>
<td>65.9</td>
<td>175.2</td>
<td>584.0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>RMSE (ms)</td>
<td>29.7</td>
<td>118.7</td>
<td>118.9</td>
<td>105.8</td>
<td>59.1</td>
<td>130.7</td>
<td></td>
</tr>
</tbody>
</table>

Table 8: Ablation study.

<table>
<thead>
<tr>
<th>Ablation Description</th>
<th>Training Time</th>
<th>GPU Memory Consumption</th>
</tr>
</thead>
<tbody>
<tr>
<td>DNNPerf</td>
<td>MRE (%)</td>
<td>RMSE (ms)</td>
</tr>
<tr>
<td>Overall</td>
<td>7.443</td>
<td>58.5</td>
</tr>
<tr>
<td>DNNPerf-GAT</td>
<td>11.693</td>
<td>101.3</td>
</tr>
<tr>
<td>DNNPerf-StandardScaler</td>
<td>11.763</td>
<td>56.0</td>
</tr>
<tr>
<td>DNNPerf-NoTensorCost</td>
<td>15.862</td>
<td>91.7</td>
</tr>
<tr>
<td>DNNPerf-ConcateEdge</td>
<td>7.993</td>
<td>60.5</td>
</tr>
<tr>
<td>DNNPerf-AvgReadout</td>
<td>20.884</td>
<td>100.9</td>
</tr>
</tbody>
</table>

Table 9: Training cost and accuracy of DNNPerf on five datasets of different sizes.

<table>
<thead>
<tr>
<th>Sample Size (Ratio)</th>
<th>Memory (GB)</th>
<th>Memory (MBC)</th>
<th>Time (ms)</th>
<th>Time (MBC)</th>
<th>Collection Time</th>
<th>Training Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>9964 (100%)</td>
<td>1.806</td>
<td>13.684</td>
<td>58.5</td>
<td>7.443</td>
<td>2.9h (20 collectors)</td>
<td>2.5h</td>
</tr>
<tr>
<td>4982 (50%)</td>
<td>1.974</td>
<td>16.496</td>
<td>96.4</td>
<td>16.244</td>
<td>2.9h (10 collectors)</td>
<td>1.5h</td>
</tr>
<tr>
<td>992 (10%)</td>
<td>2.635</td>
<td>20.547</td>
<td>118.5</td>
<td>18.504</td>
<td>2.9h (2 collectors)</td>
<td>0.8h</td>
</tr>
<tr>
<td>996 (10%)</td>
<td>2.637</td>
<td>23.610</td>
<td>130.2</td>
<td>18.518</td>
<td>2.9h (2 collectors)</td>
<td>0.8h</td>
</tr>
<tr>
<td>999 (5%)</td>
<td>2.722</td>
<td>28.439</td>
<td>144.1</td>
<td>24.058</td>
<td>2.9h (1 collector)</td>
<td>0.45h</td>
</tr>
</tbody>
</table>

(4) DNNPerf-ConcatEdge, which sums the node features and edge features separately, concatenates the summarized features into a vector, and then deliver the vector to MLP.

(5) DNNPerf-AvgReadout, which averages all the values of node features before passing the features to MLP. It and the above DNNPerf-ConcatEdge are used to evaluate the effectiveness of different global aggregation methods.

Table 8 shows the the overall prediction results of DNNPerf and other variants on the complete test set. Firstly, after replacing our ANEE encoder with GAT, the prediction accuracy is reduced in which the overall MRE/RMSE values increase by 4.25%/42.8 for the training time prediction and 3.373%/0.201 for the GPU memory consumption prediction. Nevertheless, the training throughput of DNNPerf-GAT improves some from 737.2 graphs/second to 847.3 graphs/second. We do not try EGNN [21] because it is much more costly than GAT in training (about 3X slower). Secondly, the MinMaxScaler normalization method outperforms StandardScaler in three out of four cases, showing its effectiveness and stability. Although the RMSE value of the training time prediction using StandardScaler is better, the difference is very small. Thirdly, we find that the features of tensor size and computation cost are obviously effective for the runtime performance prediction. After excluding them, the overall MRE/RMSE values increase by 8.419%/33.2 for the training time prediction and 10.73%/1.514 for the GPU memory consumption prediction. Finally, the results of DNNPerf-ConcateEdge and DNNPerf-AvgReadout show that concatenating the node and edge features and averaging the node features do not improve the prediction accuracy. The current global aggregation which sums up all the node features is more suitable to represent the accumulated runtime performance of the computation graph.

6 DISCUSSION

6.1 Extensibility of DNNPerf

Currently, DNNPerf supports representative real-world DL models and 70+ types of operator. Users can extend DNNPerf by incorporating new operators to support other models. To add a new operator, users need to formulate the features analytically based on its semantics and implement the feature extraction scripts. Furthermore, users should add training data that contains the new operator.

DNNPerf can be easily applied to predict other runtime performance metrics of both model training and inference, such as inference time, GPU utilization, and GPU power consumption. Users need to collect new training data with the metrics as the label. Note that for a prediction task associated with model inference, the front-end parser should remove operators under back propagation from the computation graph.

6.2 Training Cost of DNNPerf

A larger training dataset is generally helpful to increasing the accuracy of a prediction model; however, the training overhead also increases and may exceed what users can afford. In this section, we report the training cost and accuracy of DNNPerf on five datasets of different sizes in Table 9. The original training dataset contains 9,964 model configurations (excluding the testing and unseen configurations), from which other datasets are randomly generated.

Since the training data can be easily collected in parallel, the collection cost is not significant. For example, it spends only 2.91 hours to finish the collection of the original training dataset using 20 collectors, which is equal to that of collecting the smallest dataset (498 model configurations) using 1 collector. The more collectors, the less collection time. Training our GNN-based model does not take long either. For instance, the training with the original dataset completes after 2.5 hours. When shrinking the dataset, the training time is shortened obviously. Although the accuracy of DNNPerf decreases too, it is still acceptable because the MRE values even stay...
we apply Batch Normalization [31] to normalize the data before VGG-19 [60], and Inception-V3 [61]). The hyperparameter value
of ANEE is set to 2 and the
mechanism because the system is very large and complex. Instead,
feature to prevent the batch size information from disappearing.
the MLP layer of DNNPerf. We also enhance the input tensor size
implemented with PyTorch v1.5.1 on NVIDIA P100. To evaluate the generality of DNNPerf, we collect a total of 5,064 HPO (Hyperparameter Optimization) model configurations
which may reduce the effectiveness of DNNPerf. We mitigated this
warm-up ones), we notice that the training time across iterations is
highly non-linear. We consider the training time as a sum of the
and evaluate configurations of TensorFlow models but there are
Threats to Internal Validity: We carefully examine the framework
data source code to extract many performance-related features as the
such as the tensor size, computation cost (FLOPs) of the
framework runtime logs. We mitigated this threat by refining the feature extraction after carefully referring to the NVIDIA
development documentation, profiling the APIs using nvprof,
and analyzing the framework runtime logs.

Threats to External Validity: In reality, there could exist many
different kinds of DL models and fairly large configuration spaces,
which may reduce the effectiveness of DNNPerf. We mitigated this threat by:
(1) enlarging the training set with more diverse neural architectures and hyperparameter combinations; (2) supporting more types of DL operator. The experimental results confirm that DNNPerf is generally effective, even for configurations of unseen DL models (Section 5.2). Another threat is that we only collect and evaluate configurations of TensorFlow models but there are other DL frameworks such as PyTorch and MXNet [6]. However, since the frameworks take the same abstraction to represent models and adopt similar runtime implementations, we believe that our approach is general and DNNPerf can be adapted to support other frameworks.

7 RELATED WORK
Performance prediction for configurable systems. Many researchers focus on predicting the performance of configurable sys-
tems in a deployment environment [24–26, 51, 56–58, 63, 76]. For example, DeepPerf [26] uses less training data to train a deep sparse neural network but still achieves much higher prediction accuracy. It treats a configurable system as a black box and ignores its internal mechanism because the system is very large and complex. Instead, our work carefully analyses both the DL model and the framework
to extract performance-related features as much as possible.

### Table 10: Experimental results on the PyTorch HPO models.

<table>
<thead>
<tr>
<th>Model Name</th>
<th>Prediction of Time</th>
<th>Metrics</th>
<th>DNNPerf</th>
<th>BiRNN</th>
<th>ARNN</th>
<th>MLP</th>
<th>GBDT</th>
<th>BRP-NAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall</td>
<td></td>
<td>MRE</td>
<td>18.491</td>
<td>105.294</td>
<td>93.069</td>
<td>68.779</td>
<td>112.871</td>
<td>30.849</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RMSE</td>
<td>26.5</td>
<td>47.8</td>
<td>48.5</td>
<td>47.0</td>
<td>54.9</td>
<td>73.0</td>
</tr>
<tr>
<td>LeNet</td>
<td></td>
<td>MRE</td>
<td>22.711</td>
<td>104.180</td>
<td>90.075</td>
<td>36.515</td>
<td>20.3</td>
<td>12.757</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RMSE</td>
<td>20.7</td>
<td>22.2</td>
<td>21.9</td>
<td>21.6</td>
<td>20.3</td>
<td>21.6</td>
</tr>
<tr>
<td>ResNet-V1</td>
<td></td>
<td>MRE</td>
<td>11.407</td>
<td>69.028</td>
<td>63.477</td>
<td>33.249</td>
<td>69.835</td>
<td>18.401</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RMSE</td>
<td>18.5</td>
<td>28.1</td>
<td>27.6</td>
<td>26.8</td>
<td>27.2</td>
<td>27.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RMSE</td>
<td>32.5</td>
<td>60.3</td>
<td>60.8</td>
<td>58.9</td>
<td>76.4</td>
<td>130.3</td>
</tr>
<tr>
<td>VGG</td>
<td></td>
<td>MRE</td>
<td>59.628</td>
<td>248.064</td>
<td>199.222</td>
<td>230.410</td>
<td>250.220</td>
<td>46.494</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RMSE</td>
<td>42.8</td>
<td>83.2</td>
<td>84.0</td>
<td>83.2</td>
<td>93.2</td>
<td>96.3</td>
</tr>
</tbody>
</table>

### 6.4 Stability of Training Time Across Iterations
Practical model training usually lasts many iterations. Since it is an
iterative process, the training time across iterations should be
identical provided a model has no control-flow operators or dy-
namic structural changes (Section 3.1). To understand such stability, we actually conducted 16 experiments on the VGG, ResNet-V1, Inception-V3, and LSTM models under both TensorFlow and Py-
Torch. We trained every model configuration for an epoch of 1,000
iterations. From the collected data of each iteration (excluding the
warm-up ones), we notice that the training time across iterations is
fairly stable, with a relative standard error (i.e., standard error [69]
divided by the mean) ranging from 0.24% to 1.61%. Therefore, al-
though DNNPerf predicts the training time per iteration, we can
still derive the total training time by multiplying the iteration count.

### 6.5 Threats to Validity
**Threats to Internal Validity:** We carefully examine the framework
data source code to extract many performance-related features as the
features, such as the tensor size, computation cost (FLOPs) of the	hardware and software (Section 4.1). We randomly split the
framework and device options.

**Threats to External Validity:** In reality, there could exist many
different kinds of DL models and fairly large configuration spaces,
which may reduce the effectiveness of DNNPerf. We mitigated this threat by:
(1) enlarging the training set with more diverse neural architectures and hyperparameter combinations; (2) supporting more types of DL operator. The experimental results confirm that DNNPerf is generally effective, even for configurations of unseen DL models (Section 5.2). Another threat is that we only collect and evaluate configurations of TensorFlow models but there are other DL frameworks such as PyTorch and MXNet [6]. However, since the frameworks take the same abstraction to represent models and adopt similar runtime implementations, we believe that our approach is general and DNNPerf can be adapted to support other frameworks.

### 6.3 Generality of DNNPerf
Our GNN-based approach is general and can be adapted to other DL
frameworks such as PyTorch [45] and other devices such as NVIDIA
P100. To evaluate the generality of DNNPerf, we collect a total of
5,064 HPO (Hyperparameter Optimization) model configurations
implemented on NVIDIA P100 GPU.

Because the framework implementation and hyperparameters (e.g., the padding) of PyTorch are different from those of TensorFlow, we apply Batch Normalization [31] to normalize the data before the MLP layer of DNNPerf. We also enhance the input tensor size feature to prevent the batch size information from disappearing. The number of message-passing rounds of ANEE is set to 2 and the learning rate is set to 0.001.

The HPO dataset includes the configurations of six real-world DL
models (AlexNet [37], LeNet [40], ResNet-V1 [28], VGG-11, VGG-16,
VGG-19 [60], and Inception-V3 [61]). The hyperparameter value combinations are described in Section 4.1. We randomly split the full dataset for training (80%) and test (20%). We retrain DNNPerf (for 200 epochs) and baselines over the new dataset.

The experimental results in Table 10 show that the overall MRE/RMSE improvement over the baseline approaches ranges from
12.358%/20.5 to 86.79%/46.5 in predicting the training time, and from
0.328%/1.571 to 44.984%/2.132 in predicting the GPU memory
consumption. These results demonstrate that DNNPerf is general to
diverse framework and device options.

**Table 10: Experimental results on the PyTorch HPO models.**

<table>
<thead>
<tr>
<th>Model Name</th>
<th>Prediction of Time</th>
<th>Metrics</th>
<th>DNNPerf</th>
<th>BiRNN</th>
<th>ARNN</th>
<th>MLP</th>
<th>GBDT</th>
<th>BRP-NAS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Overall</td>
<td></td>
<td>MRE</td>
<td>18.491</td>
<td>105.294</td>
<td>93.069</td>
<td>68.779</td>
<td>112.871</td>
<td>30.849</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RMSE</td>
<td>26.5</td>
<td>47.8</td>
<td>48.5</td>
<td>47.0</td>
<td>54.9</td>
<td>73.0</td>
</tr>
<tr>
<td>LeNet</td>
<td></td>
<td>MRE</td>
<td>22.711</td>
<td>104.180</td>
<td>90.075</td>
<td>36.515</td>
<td>20.3</td>
<td>12.757</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RMSE</td>
<td>20.7</td>
<td>22.2</td>
<td>21.9</td>
<td>21.6</td>
<td>20.3</td>
<td>21.6</td>
</tr>
<tr>
<td>ResNet-V1</td>
<td></td>
<td>MRE</td>
<td>11.407</td>
<td>69.028</td>
<td>63.477</td>
<td>33.249</td>
<td>69.835</td>
<td>18.401</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RMSE</td>
<td>18.5</td>
<td>28.1</td>
<td>27.6</td>
<td>26.8</td>
<td>27.2</td>
<td>27.3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RMSE</td>
<td>32.5</td>
<td>60.3</td>
<td>60.8</td>
<td>58.9</td>
<td>76.4</td>
<td>130.3</td>
</tr>
<tr>
<td>VGG</td>
<td></td>
<td>MRE</td>
<td>59.628</td>
<td>248.064</td>
<td>199.222</td>
<td>230.410</td>
<td>250.220</td>
<td>46.494</td>
</tr>
<tr>
<td></td>
<td></td>
<td>RMSE</td>
<td>42.8</td>
<td>83.2</td>
<td>84.0</td>
<td>83.2</td>
<td>93.2</td>
<td>96.3</td>
</tr>
</tbody>
</table>
Performance prediction for DL models. Recently, there is some work on predicting the runtime performance of DL models [18, 19, 33, 41, 49, 73]. For example, Paleo [49] estimates the execution time from FLOPs, and DNNMem [19] pre-builds an analytic model for the GPU memory consumption estimation. However, these analytic approaches require lots of hand-crafted efforts and are specific to certain tasks. The work of [73] predicts the GPU utilization from the FLOPS, input data size, and number of convolutional layers, but it ignores many other affecting factors such as the neural architecture, operator execution order, I/O cost, etc. Predicting the learning performance (e.g., accuracy) of DL models has also attracted interests in the AutoML community [4, 10, 16–18, 20, 36, 65, 71]. BRP-NAS “proposes a GCN-based predictor for the end-to-end latency” [18] using the operator type and computation graph structure as features. However, its generalization ability of cross-model prediction is low. NPNet employs a Recurrent Neural Network (RNN) to predict the accuracy from the neural architecture, which “avoids time-consuming training to obtain true accuracy but with a slight drawback of regression error” [17]. Comparing to the prior work, DNNPerf captures not only operator-level features but also the computation graph information and hidden factors within the framework. Our general learning-based approach reduces hand-crafted efforts and achieves better prediction results.

8 CONCLUSION

In this paper, we have presented DNNPerf, a novel runtime performance prediction tool for deep learning models. DNNPerf adopts a GNN-based approach and systematically explores performance-related features derived from the semantics of the computation graph and hidden factors within the framework. Our experiments demonstrate that DNNPerf accurately predicts the GPU memory consumption and training time. DNNPerf is also effective and robust to the choices of hyperparameters and neural architectures, even to unseen models.

REFERENCES


