2023 IEEE/ACM 45th International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP) | 979-8-3503-0037-6/23/$31.00 ©2023 IEEE | DOI: 10.1109/ICSE-SEIP58684.2023.00039

Runtime Performance Prediction for Deep Learning Models with Graph Neural Network

Yanjie Gao[1], Xianyu Gu[1,3*], Hongyu Zhang[2], Haoxiang Lin[1†], Mao Yang[1]
[1]Microsoft Research, Beijing, China Email: {yanjga, haoxlin, maoyang}@microsoft.com
[2]Chongqing University, Chongqing, China Email: hyzhang@cqu.edu.cn
[3]Tsinghua University, Beijing, China Email: gxy21@mails.tsinghua.edu.cn

***Abstract*—Deep learning models have been widely adopted in many application domains. Predicting the runtime performance of deep learning models, such as GPU memory consumption and training time, is important for boosting development productivity and reducing resource waste. The reason is that improper configurations of hyperparameters and neural architectures can result in many failed training jobs or unsatisfactory models. However, the runtime performance prediction of deep learning models is challenging because of the hybrid programming paradigm, complicated hidden factors within the framework runtime, enormous model configuration space, and broad differences among models. In this paper, we propose DNNPerf, a novel ML-based tool for predicting the runtime performance of deep learning models using Graph Neural Network. DNNPerf represents a 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 for the node and edge features. DNNPerf is evaluated on thousands of configurations of real-world and synthetic deep learning models to predict their GPU memory consumption and training time. The experimental results show that DNNPerf achieves accurate predictions, with an overall error of 7.4% for the training time prediction and an overall error of 13.7% for the GPU memory consumption prediction, confirming its effectiveness.**

***Index Terms*—deep learning, AutoML, graph neural network, runtime performance, performance prediction**

I. Introduction

In recent years, deep learning (DL) has been widely adopted in many application domains, such as computer vision [1], speech recognition [2], and natural language processing [3]. 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 large 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 because it directly affects model quality and development productivity. Improper model configurations can unexpectedly degrade runtime performance or result in many failed jobs or unsatisfactory models. For instance, an overlarge batch size causes a job to exhaust all GPU memory and raise an *out-of-memory* (OOM) exception. According to a recent empirical study on 4,960 DL job failures collected from Microsoft [4], 8.8% of the failed jobs were caused by GPU OOM, which accounts for the largest category in all the DL specific failures. We also observed that a number of internal DL jobs inside Microsoft exhibited low GPU utilization and ran slowly because of too small batch sizes and other improper hyperparameter values. As a result, these jobs had to be early stopped before reaching our expectations due to over 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 the literature, there is already much research work for estimating the runtime performance of programs [5]–[10] and deployed systems [11]–[16] using program analysis and machine learning (ML) techniques. Recently, some authors advanced the estimation effort to DL models [17]–[20] using analytic or machine learning methods. However, these studies either cannot be directly applied or have limitations in precise prediction due to the following challenges:

1) DL frameworks provide a hybrid programming paradigm: developers invoke high-level interfaces only to construct DL models, while low-level computational operations are implemented with proprietary NVIDIA CUDA, cuDNN, and cuBLAS APIs. Such a paradigm hides the internal implementation details and thus makes it hard to model the runtime performance accurately.
2) Many complicated hidden factors within the framework runtime, such as garbage collection and operator execution order, affect the runtime performance of models observably. However, understanding and extracting these hidden factors could be very difficult, time-consuming, and error-prone because they are volatile and fast-changing with the rapid evolution of DL frameworks.
3) A DL model usually has an enormous configuration space, and there are broad differences among various

*Work performed during the internship at Microsoft Research.
†Corresponding author.

2832-7659/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE-SEIP58684.2023.00039

kinds of models. Therefore, it is challenging to design a general ML prediction model with high accuracy from limited samples.

In this paper, we propose DNNPerf, a novel ML-based tool for predicting the runtime performance of DL models with *Graph Neural Network* (GNN) [21]. 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 consist of, for example, the operator type (e.g., `Conv2D`), hyperparameters (e.g., the kernel size), number of floating-point operations (FLOPs), and sizes of input/output/weight/temporary tensors. Typical edge features include the edge type (forward or backward) and the 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 the ideas of Graph Attention Network (GAT) [23] and Edge Enhanced Graph Neural Network (EGNN) [24].

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 [25], ResNet [1], and Vanilla RNN [26]), whose representative hyperparameter values are randomly generated within reasonable ranges; (2) 8,403 model configurations, whose neural architectures are randomly synthesized by Neural Architecture Search (NAS) [27] algorithms. We explore two typical prediction tasks for training time and GPU memory consumption; however, our approach can be easily applied to other tasks, such as predicting inference time and GPU power consumption. The experimental results show that DNNPerf outperforms all the five compared methods (e.g., BiRNN [5] and GBDT [28]) with an overall mean relative error of 7.4% for the training time prediction and an overall mean relative error of 13.7% for the GPU memory consumption prediction. The improvement of MRE/RMSE ranges from 10.6%/47.7 ms to 87.8%/276.9 ms for the training time prediction task and from 1.9%/0.4 GB to 20.0%/2.3 GB for the GPU memory consumption prediction task, confirming the effectiveness of DNNPerf.

We summarize our main contributions as follows:

1) We propose a novel Graph Neural Network-based approach for predicting the runtime performance of deep learning models accurately.
2) We design a rich set of node and edge features to capture performance-related factors (e.g., compute and I/O). We also propose a novel Attention-based Node-Edge Encoder for the computation graph of a deep learning model.
3) We implement an end-to-end tool named DNNPerf and demonstrate its practical effectiveness through experiments on thousands of configurations of real-world and synthetic deep learning models.

```
1  from tensorflow.keras import layers, models
2  ...
3  model = models.Sequential()
4  model.add(layers.Conv2D(filters=32, kernel_size=(3, 3),
   ↪  activation='relu', input_shape=(224, 224, 3)))
5  model.add(layers.AveragePooling2D(pool_size=(2, 2)))
6  model.add(layers.Conv2D(filters=64, kernel_size=(3, 3),
   ↪  activation='relu'))
7  model.add(layers.AveragePooling2D(pool_size=(2, 2)))
8  model.add(layers.Flatten())
9  model.add(layers.Dense(units=1000, activation="softmax"))
10 model.compile(optimizer='SGD',
   ↪  loss="categorical_crossentropy")
11 model.fit(train_images, train_labels, batch_size,
   ↪  epochs=60)
```

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

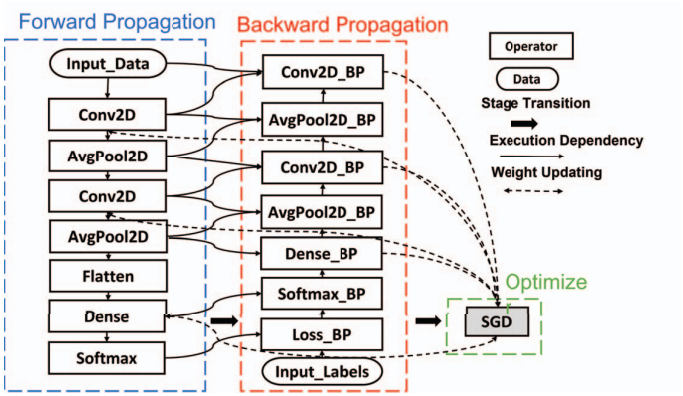

Fig. 2: Computation graph for training the above model.

II. Background

A. Deep Learning Models and Computation Graphs

Being essentially mathematical functions, DL models are formalized by frameworks like TensorFlow [29] and PyTorch [30] as tensor-oriented computation graphs (i.e., directed acyclic graphs). The inputs and outputs of a computation graph or a graph node are *tensors* (i.e., multidimensional arrays of numerical values). Each node denotes a mathematical operation called an operator (e.g., 2D convolution [31]). An edge pointing from one output of node A to one input of B delivers a tensor and specifies the execution dependency of such two nodes. 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 [32] training program written in the Keras [33] API. The program constructs a sequential model with the framework built-in `Conv2D` (2D convolution with a 3×3 kernel size), `AvgPool2D` (2D average pooling with a 2×2 pool size), `Flatten` (collapsing the input tensor into one dimension), `Dense` (fully-connected layer with 64 units), and `Softmax` (normalizing "the probability distribution over k different classes" [22]) operators (lines

3–9). The above number of output channels (denoted by the variable `filters`), kernel size, number of units, and others 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 (i.e., weights). The operators (nodes) in the middle of Figure 2 (e.g., `AvgPool2d_BP`) are automatically generated by TensorFlow for backward propagation. The input data (`Input_Data`) is fed through the computation graph and is manipulated by the developer-specified operators (on the left of Figure 2). The input labels (`Input_Labels`) and the produced outputs of `Dense` are then backward propagated to compute the gradients of the 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.

B. Graph Neural Networks

The graph is a common data structure for representing elements and their relations, widely used in data analysis. Recently, there have been emerging requirements to apply DL techniques to learning from graph data. However, existing models such as Convolutional Neural Network (CNN) [31] and Recurrent Neural Network (RNN) [35] have limitations in handling graphs because of the irregularity caused by unequal node neighbors. Graph Neural Network (GNN) [21] is then proposed to address this issue efficiently.

Graph Convolutional Network (GCN) [36] is the first popular GNN, which learns localized spatial features by convolutional filters. Graph Attention Network (GAT) [23] 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) [24] builds a framework for a family of new GNNs that can exploit the features of edges (including both undirected and multi-dimensional ones) sufficiently. In our work, we adopt GNN and propose a novel Attention-based Node-Edge Encoder for better predicting the runtime performance of DL models.

III. DNNPerf: A GNN-based Approach

A. Problem Formulation

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

$$\mathcal{M} = \langle \{u_i\}_{i=1}^{n}, \{e_{ij} = (u_i, u_j)\}_{i,j\in[1,n]}, \{hp_k\}_{k=1}^{m} \rangle .$$

Each node u_i is an operator such as the above `Conv2D` and `AvgPool2D`. 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 hp_k is a hyperparameter of some operator, such as the batch size and the input tensor shape, whose domain is denoted as B_k. Model $\mathcal{M}$ 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 $\mathcal{M}(b_1 \in B_1, b_2 \in B_2, \cdots, b_m \in B_m)$ to denote one *model configuration* of $\mathcal{M}$. Then, $\Delta_{\mathcal{M}}$, the *configuration space* of $\mathcal{M}$, is defined as follows:

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

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

$$f_i : \Delta_{\mathcal{M}} \times \mathcal{S} \to \mathbb{R} .$$

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

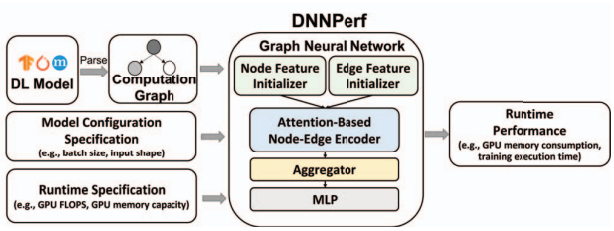

Fig. 3: Workflow of DNNPerf.

B. 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 runtime performance values. DNNPerf implements a front-end parser using the built-in model deserialization APIs of DL frameworks, which is responsible for reading the input model file and reconstructing the corresponding computation graph. The model configuration specification includes tuned hyperparameters and their values. The runtime specification contains the device information at present.

DNNPerf traverses the computation graph to generate the node and edge features automatically, strictly following an operator execution order predefined by reference to the framework implementations [37], [38]. It employs a novel Attention-based Node-Edge Encoder (ANEE) to embed the nodes and edges in several rounds (Section III-D). To improve prediction accuracy and reduce the amount of training data, DNNPerf further utilizes the node and edge semantics. The details are summarized in Tables I and II. Finally, DNNPerf

TABLE I
NODE FEATURES.

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

TABLE II
EDGE FEATURES.

Category	Name	Description
Edge (e^t)	Edge Type	Type of the edge ("forward" or "backward")
Tensor (e^d)	Tensor Size	Size of the delivered tensor
Device (e^r)	Bandwidth	Bandwidth for accessing the delivered tensor

aggregates updated feature vectors and uses a multilayer perceptron (MLP) [39] to output a result.

C. Model Encoding

We refer to the framework source code, identify hidden factors carefully, and 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, sizes of different tensors, and FLOPs to capture operator-level factors. Secondly, DNNPerf uses an edge's type and size of the delivered tensor to capture the operator execution order, tensor liveness [18], and I/O costs. Lastly, selected device features such as FLOPS, memory capacity, and bandwidth are extracted and associated with the nodes and edges to capture those runtime factors such as garbage collection, memory access, and distributed communication.

1) Initial Node Encoding: For each node of a computation graph (such as the one 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 the designed node features. h is composed of several sub-feature vectors such that $h = F(h^t \parallel h^p \parallel h^d \parallel h^c \parallel h^r)$:

1) "$\parallel$" is the vector concatenation operation, and F is the function for raw feature encoding and vector shape alignment.
2) h^t 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 all hyperparameters.
4) h^d stores the sizes of the weight, input, output, and temporary tensors.
5) h^c denotes the computation cost. Currently, we use FLOPs.
6) h^r encodes the device information affecting the operator execution, such as GPU memory capacity and FLOPS.

Details are listed in Table I. For categorical features (e.g., operator type), we adopt the One-Hot encoding [40] strategy.

We take `Conv2D`, the 2D convolution operator [31], as an example to illustrate how to compute the initial values of h^c (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 [41]:

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

We now show how the FLOPs under forward propagation [42] and tensor sizes (in bytes) of `Conv2D` are computed:

$$\begin{aligned} FLOPs(\texttt{Conv2D}) &= 2 \times (C_i \times H_i \times W_i + 1) \times N \\ &\quad \times C_o \times H_o \times W_o \,, \\ IT(\texttt{Conv2D}) &= sz \times N \times C_i \times H_i \times W_i \,, \\ OT(\texttt{Conv2D}) &= sz \times N \times C_o \times H_o \times W_o \,, \\ WT(\texttt{Conv2D}) &= sz \times (C_i \times H_i \times W_i + 1) \times C_o \,. \end{aligned}$$

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 operators.

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 the designed edge features. e is composed of several sub-feature vectors such that $e = e^t \parallel e^d \parallel e^r$:

1) e^t represents the edge type, whose initial value is either "forward" or "backward" at present, depending on which

propagation stage the delivered tensor is used. e^t tries to capture tensor liveness [18] that affects GPU memory consumption.

2) e^d denotes the size of the delivered tensor, while e^r denotes the device bandwidth. These two features affect I/O access latency, which in turn affects training time.

Details are listed in Table II.

3) Normalization: Since feature values vary widely, DNNPerf scales the range of each feature to the interval $[0, 1]$ by *MinMaxScaler* [43]. DNNPerf does not use the neighbor-level normalization from EGNN [24] because the average node degree of a computation graph is much smaller than that of previously studied large graphs such as social network graphs. 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 $\hat{h}_{g,u} = [\hat{h}_{g,u,1}, \cdots, \hat{h}_{g,u,N_1}]$ is produced as follows:

$$S_i = \{h_{g,u,i} \mid g \in DS \wedge u \in g\}\,,$$
$$\hat{h}_{g,u,i} = \frac{h_{g,u,i} - \min S_i}{\max S_i - \min S_i}\,.$$

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 $\hat{e}_{g,l} = [\hat{e}_{g,l,1}, \cdots, \hat{e}_{g,l,N_2}]$ is produced as follows:

$$S_i = \{e_{g,l,i} \mid g \in DS \wedge l \in g\}\,,$$
$$\hat{e}_{g,l,i} = \frac{e_{g,l,i} - \min S_i}{\max S_i - \min S_i}\,.$$

D. Attention-based Node-Edge Encoder

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

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 captures both the node and edge features. The operators of a DL model contain less fan-out, and their runtime performance may vary significantly. DNNPerf uses a global normalizer that achieves better efficiency and accuracy. We also use weight sharing for the encoder parameter matrices to reduce the number of parameters, which benefits the training/inference time and GNN model size. Therefore, compared with EGNN [24], 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 $l = (u_s, u_d)$ is an edge pointing from the source node u_s to the target u_d. We use $h^i_{g,u}$ and $e^i_{g,l}$ to denote the computed feature vectors of node u and edge l in the i-th ($i > 0$) round, respectively. $h^0_{g,u}$ and $e^0_{g,l}$ are the initial normalized feature vectors mentioned before.

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

$$\texttt{LeakyReLU}(x) = \max(0, x) - \alpha \times \max(0, -x)\,,$$
$$\overline{h}^i_{g,u} = \texttt{LeakyReLU}(\mathbf{W}_u \times h^{i-1}_{g,u})\,.$$

The activation function `LeakyReLU` is the leaky version of Rectified Linear Unit (ReLU), α is the slope coefficient that defaults to 0.01, and $\mathbf{W}_u$ is a parameter matrix.

Next, ANEE updates $e^i_{g,l}$ as follows:

$$e^i_{g,l} = \sigma(a^T \times (\overline{h}^i_{g,s} \parallel \overline{h}^i_{g,d}) \times \mathbf{W}_e \times e^{i-1}_{g,l})\,.$$

σ is the sigmoid activation function, and $\mathbf{W}_e$ is a parameter matrix. The attention mechanism uses a single-layer feedforward neural network that is denoted by a weight vector $a \in \mathbb{R}^{2\times N_1}$. a^T is the transpose of a.

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

$$f(u', l') = \texttt{Softmax}(\mathbf{W}_m \times e^i_{g,l'}) \times \overline{h}^i_{g,u'}\,,$$
$$h^i_{g,u} = \texttt{LeakyReLU}(\textstyle\sum_{l'=(u',u)} f(u', l'))\,.$$

u' is a neighbor node of u, l' is their associated edge, and $\mathbf{W}_m$ 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.

E. 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 a loss function:

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

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

IV. Experimental Setup

A. 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) [44] dataset, including 10,238 configurations of ten real-world models with various hyperparameter combinations. In this dataset, the configurations of five randomly selected DL models (LeNet [45], ResNet-V1 [1], Inception-V3 [46], Vanilla RNN [26], and LSTM [47]) are used for normal training, validation, and test, in a proportion of 70:10:20. The rest models (AlexNet [48], VGG [25], OverFeat [49], ResNet-V2 [1], and GRU [50]) are

TABLE III
STATISTICS OF THE HPO DATASET.

Model Name	Node #	Avg Node #	Edge #	Avg Edge #	Config #
LeNet	[18, 18]	18.0	[25,25]	25.0	800
ResNet-V1	[242, 796]	494.6	[413, 1,382]	854.7	622
Inception-V3	[12, 254]	95.2	[16, 449]	162.7	1,850
Vanilla RNN	[20, 56]	38.0	[28, 82]	55.0	1,280
LSTM	[20, 56]	38.0	[28, 82]	55.0	1,280
AlexNet (UNSEEN)	[28, 28]	28.0	[40, 40]	40.0	800
VGG (UNSEEN)	[38, 54]	46.7	[55, 79]	68.0	870
OverFeat (UNSEEN)	[28, 28]	28.0	[40, 40]	40.0	800
ResNet-V2 (UNSEEN)	[242, 796]	493.5	[413, 1,382]	852.9	656
GRU (UNSEEN)	[20, 56]	38.0	[28, 82]	55.0	1,280
Total	[12, 796]	102.9	[16, 1,382]	170.6	10,238

unseen by DNNPerf because their configurations do not exist in the training set and are only used to evaluate the generalization ability of DNNPerf (Section V-B). These ten models are representative in computer vision, speech recognition, and natural language processing, among which some (such as RNN, LSTM, and GRU) are also widely used in various deep-learning-for-software-engineering applications [51]–[54]. The second is a NAS (Neural Architecture Search) [27] dataset, containing 8,403 configurations of NAS-searched models.

HPO dataset. We choose representative DL models from the TensorFlow-Slim model library [55] 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 tuned hyperparameters. For VGG, we use different value domains of the batch size (interval $[16, 128]$) and number of input channels (interval $[1, 10]$). For AlexNet, OverFeat, and LeNet, we additionally take the number of output channels of their `Conv2D` operators as a tuned hyperparameter and set its domain to an interval $[\text{default} \times 0.5, \text{default} \times 2]$. For VGG model configurations, we generate them randomly from the VGG-11, VGG-16, and VGG-19 models. Similarly, for ResNet-V1 and ResNet-V2 model configurations, we generate them randomly from the ResNet-50, ResNet-101, ResNet-152, and ResNet-200 models. For Inception-V3, we assign random numbers to the `min_depth`, `depth_multiplier`, and `end_point` parameters of the Inception API. The statistics of the HPO dataset are shown in Table III.

NAS dataset. We use the NAS search space illustrated in Figure 4 to generate model configurations. The model skeleton consists of combinations of several operator cells (e.g., Cell Types 1–5). Each cell contains a diverse combination of different operators (e.g., `Conv2D`, `Dense`, and `MaxPool2D`) and hyperparameters. The tuned hyperparameters include: (1) batch size (16, 32, 48, 64, 80, 96, 112, and 128), 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

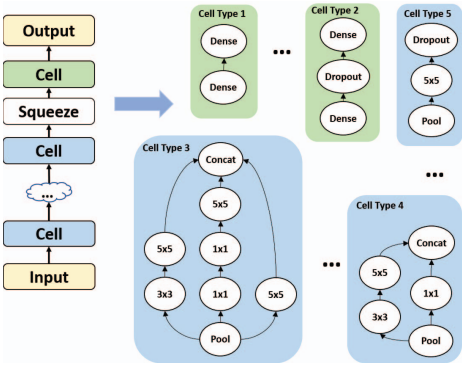

Fig. 4: Example neural architectures in the NAS dataset.

TABLE IV
STATISTICS OF THE NAS DATASET.

Node #	Total	[20, 29]	[30, 39]	[40, 49]	[50, 70]
Config #	8,403	499	4,445	2,974	485
Edge #	**Total**	**[35, 49]**	**[50, 64]**	**[65, 79]**	**[80, 115]**
Config #	8,403	2,071	4,183	1,797	352

`Dropout` and `MaxPool2D` operators. The statistics of the NAS dataset are shown in Table IV.

The first few iterations are bypassed until the training has been stable. Afterward, we measure the runtime performance (i.e., execution time and peak GPU memory consumption) of three iterations and compute the average values.

B. Baselines

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

1) BiRNN (Bidirectional RNN) [5] is a two-layer bidirectional RNN [35] with the LSTM [47] cell. Node features include the name, type, and computation cost of the operator, computed in the same way as the initial node features of DNNPerf. We feed the node feature vectors to BiRNN as an input sequence according to the topological order of the computation graph. This baseline allows us to identify how effective the GNN is in capturing the data flow information of a computation graph.
2) ARNN (Adjacency BiRNN) [5] is an extension of BiRNN [5]. The feature vector of a node is updated by computing the average of vectors of predecessors, successors, and itself. ARNN is a strong baseline because it takes some structural information of a computation graph into account.
3) MLP (Multilayer Perceptron) [39] is a traditional machine learning model. Prior work [19] used it to predict operator execution time. We evaluate MLP with the same node features that DNNPerf uses.
4) GBDT (Gradient Boost Decision Tree) [28] is also a traditional machine learning model. It was used by Chen et al. [56] for encoding loop programs. GBDT uses the same features as MLP.

5) BRP-NAS (Binary Relation Predictor-based NAS) [57] is a graph convolutional network for the NAS dataset. It encodes only graph topology, regardless of 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 [30]; for GBDT, we use the built-in functionality of scikit-learn v0.20.3 [58]. To tune these models, we select the following hyperparameter values. For BiRNN and ARNN, the learning rate is 0.0001, layer size is 1, and hidden size is 512. For MLP, the 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.

C. Implementation and Settings of DNNPerf

We have implemented DNNPerf with DGL (Deep Graph Library) [59] v0.4.3, 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 by grid search [60]. The MLP uses hidden layers of 512, 128, and 16 units; the size of the final output is 1. The number of message-passing rounds of ANEE is set to 3 (training time prediction) or 1 (GPU memory consumption prediction). We use the Adam [34] optimizer with default parameter values ($\alpha = 0.9$ and $\beta = 0.999$), set the batch size to 64 and the 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.

D. Evaluation Metrics

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

$$\text{MRE} = \frac{\sum_{i=1}^{N} \left| \frac{\hat{y}_i - y_i}{y_i} \right|}{N} \times 100\%,$$

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

N is the number of data samples in the test set; $\hat{y}_i$ and y_i are the predicted and realistic 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 [11], [18], [19]. 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 gigabytes (GB) and milliseconds (ms), respectively.

V. Evaluation

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

TABLE V
Overall experimental results.

Model Name	Prediction of Training Time						
	Metrics	DNNPerf	BiRNN	ARNN	MLP	GBDT	BRP-NAS
Overall	MRE (%)	**7.4**	20.8	20.9	18.0	31.0	95.2
	RMSE (ms)	**58.5**	151.4	113.9	106.2	122.6	335.4
HPO	MRE (%)	**7.8**	21.6	22.2	19.4	33.1	86.7
	RMSE (ms)	**51.9**	107.7	95.2	94.7	120.4	294.3
NAS	MRE (%)	**6.3**	18.4	16.9	13.5	24.4	122.3
	RMSE (ms)	**75.7**	242.6	159.5	136.4	129.3	441.0
Model Name	**Prediction of GPU Memory Consumption**						
	Metrics	**DNNPerf**	**BiRNN**	**ARNN**	**MLP**	**GBDT**	**BRP-NAS**
Overall	MRE (%)	**13.7**	22.3	25.5	21.5	15.6	33.7
	RMSE (GB)	**1.8**	2.8	2.9	2.3	2.2	4.1
HPO	MRE (%)	**13.2**	22.1	26.7	22.1	14.2	31.5
	RMSE (GB)	**1.8**	2.8	3.0	2.4	2.1	4.2
NAS	MRE (%)	**15.3**	22.8	21.7	19.4	20.2	40.5
	RMSE (GB)	**1.8**	2.8	2.4	2.0	2.4	4.0

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 [61], 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.

A. RQ1: How effective is DNNPerf in predicting the runtime performance?

In this section, we compare DNNPerf with five baselines (Section IV-B) on the same test set. Table V shows the MRE and RMSE values of all six approaches for predicting runtime performance. On average (see the "Overall" rows), DNNPerf achieves an MRE of 7.4% and an RMSE of 58.5 ms for the training time prediction and an MRE of 13.7% and an RMSE of 1.8 GB for the GPU memory consumption prediction, exceeding the best MRE/RMSE values of the baselines: 18.0%/106.2 from MLP and 15.6%/2.2 from GBDT. More specifically, DNNPerf outperforms BiRNN by 13.4%/92.9, ARNN by 13.5%/55.4, MLP by 10.6%/47.7, GBDT by 23.6%/64.1, and BRP-NAS by 87.8%/276.9 in terms of MRE/RMSE for the training time prediction; DNNPerf outperforms BiRNN by 8.6%/1.0, ARNN by 11.8%/1.1, MLP by 7.8%/0.5, GBDT by 1.9%/0.4, and BRP-NAS by 20.0%/2.3 in terms of MRE/RMSE for the GPU memory consumption prediction. We also experiment on the HPO and NAS models separately. The results demonstrate that DNNPerf still excels in the baselines and confirm its effectiveness. We think the reason is that DNNPerf captures richer and more diverse performance-related information from the operator, hyperparameter, and neural architecture than the other approaches.

Table VI 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 by DNNPerf because the training dataset contains their configurations. We choose various hyperparameters and larger value ranges

TABLE VI
EXPERIMENTAL RESULTS ON HPO MODELS.

Model Name	Prediction of Training Time						
	Metrics	DNNPerf	BiRNN	ARNN	MLP	GBDT	BRP-NAS
Overall	MRE (%)	**8.5**	28.8	32.0	22.5	27.6	69.0
	RMSE (ms)	**36.8**	89.3	94.0	74.4	60.2	226.8
LeNet	MRE (%)	**3.9**	27.2	15.5	14.6	43.5	69.2
	RMSE (ms)	**7.1**	64.5	32.7	23.2	44.6	87.0
ResNet-V1	MRE (%)	**4.1**	7.5	11.4	7.4	8.3	61.7
	RMSE (ms)	**60.0**	84.2	109.3	74.5	130.5	505.3
Inception-V3	MRE (%)	**12.7**	28.7	41.9	18.5	47.7	95.8
	RMSE (ms)	41.2	58.3	70.1	**31.3**	53.6	175.2
RNN	MRE (%)	**8.5**	41.3	41.0	36.0	21.1	77.3
	RMSE (ms)	**20.8**	91.4	91.4	77.0	36.8	154.6
LSTM	MRE (%)	**7.5**	27.7	29.4	26.2	7.8	29.9
	RMSE (ms)	38.5	126.9	131.4	117.4	**37.9**	183.7

Model Name	Prediction of GPU Memory Consumption						
	Metrics	DNNPerf	BiRNN	ARNN	MLP	GBDT	BRP-NAS
Overall	MRE (%)	**12.8**	24.3	30.5	18.1	16.8	63.2
	RMSE (GB)	1.5	2.7	3.2	1.8	**1.3**	4.6
LeNet	MRE (%)	**11.4**	19.9	20.1	15.8	15.1	46.5
	RMSE (GB)	**1.3**	2.6	2.2	1.6	1.6	3.5
ResNet-V1	MRE (%)	8.7	15.3	16.3	9.7	**7.4**	31.1
	RMSE (GB)	**1.6**	2.9	3.0	1.7	1.7	4.8
Inception-V3	MRE (%)	**14.8**	25.8	27.4	17.5	27.8	57.2
	RMSE (GB)	**1.2**	1.9	1.9	1.4	1.2	2.7
RNN	MRE (%)	**10.8**	16.8	22.2	17.1	16.7	114.5
	RMSE (GB)	**0.8**	1.1	1.5	0.9	0.9	4.7
LSTM	MRE (%)	14.7	36.0	54.6	24.9	**7.9**	46.9
	RMSE (GB)	2.1	4.2	5.5	2.8	**1.3**	6.5

(Section IV-A) to increase the diversity of runtime performance. DNNPerf achieves satisfactory precision and outperforms the baseline approaches in most cases by a large margin. The overall MRE/RMSE improvement over the baseline approaches ranges from 14.0%/23.4 to 60.5%/190 in predicting the training time and from 4.0%/0.3 to 50.4%/3.1 in predicting the GPU memory consumption. We notice that the overall RMSE value of GBDT for the GPU memory consumption prediction is slightly better than DNNPerf since GBDT has advantages on models whose operator types are relatively simple (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. DNNPerf still achieves satisfactory precision and outperforms all the baseline approaches. The MRE values of DNNPerf over different numbers of nodes range from 5.8% to 7.2% for the training time prediction and from 14.8% to 15.9% for the GPU memory consumption prediction; the RMSE values range from 25.4 to 154.5 and from 1.6 to 2.3, respectively. The MRE values of DNNPerf over different numbers of edges range from 5.9% to 7.0% for the training time prediction and from 14.8% to 16.6% for the GPU memory consumption prediction; the RMSE values range from 50.6 to 153.2 and from 1.6 to 2.0, respectively. The experimental results demonstrate that DNNPerf has good stability on diverse neural architectures.

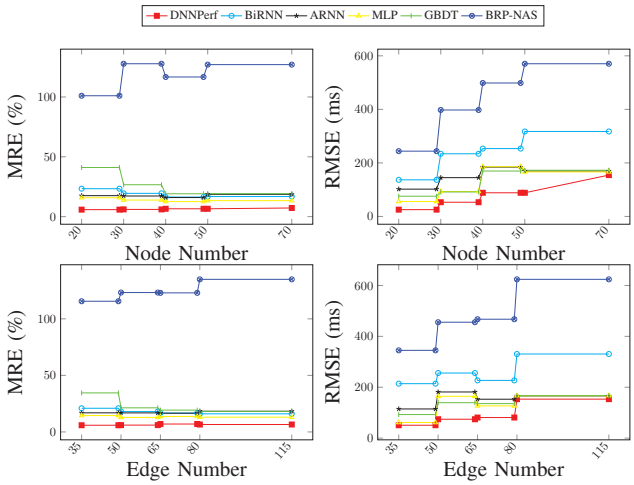

Fig. 5: Experimental results of the training time prediction on NAS models.

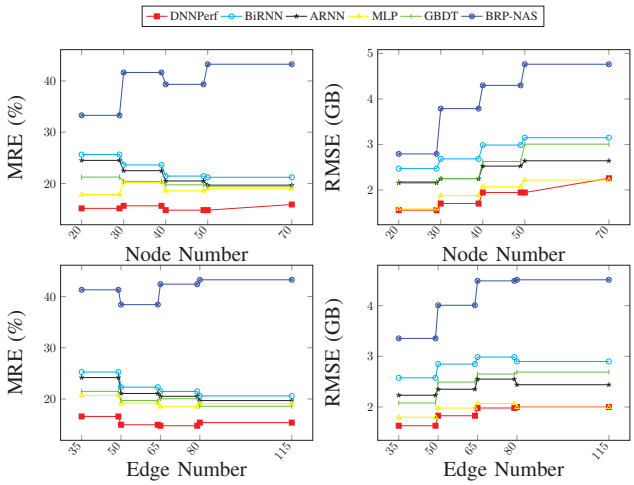

Fig. 6: Experimental results of the GPU memory consumption prediction on NAS models.

B. RQ2: How general is DNNPerf to unseen DL models?

In this section, we evaluate the generalization ability of DNNPerf, which is critical for its practical use. We construct a test set consisting of the model configurations of AlexNet, VGG, OverFeat, ResNet-V2, and GRU — these five models are *unseen* by DNNPerf, as the training set does not include any of their model configurations.

Table VII lists the values of MRE and RMSE, with the overall results of DNNPerf surpassing the baseline approaches. The improvement in MRE/RMSE ranges from 10.9%/40.4 to 83.5%/253.9 for the training time prediction task and from 0.2%/0.4 to 12.4%/2.2 for the GPU memory consumption prediction task. BRP-NAS does not perform well due to its design, which is limited to a specific NAS search space without modeling the runtime performance-related features. Other methods fail to encode the complete computation graph and hidden runtime factors (such as operator scheduling and tensor liveness), leading to inferior results. In summary, the

TABLE VII
EXPERIMENTAL RESULTS ON UNSEEN MODELS.

Model Name	Prediction of Training Time						
	Metrics	DNNPerf	BiRNN	ARNN	MLP	GBDT	BRP-NAS
Overall	MRE (%)	**7.7**	19.8	19.8	18.6	34.5	91.2
	RMSE (ms)	**55.1**	111.8	95.5	99.1	131.2	309.0
AlexNet	MRE (%)	**11.9**	22.5	18.4	21.6	85.4	291.1
	RMSE (ms)	**13.6**	27.1	28.6	26.5	90.4	264.6
VGG	MRE (%)	**7.3**	13.3	10.1	16.8	12.6	41.6
	RMSE (ms)	**84.2**	164.6	92.3	154.4	148.6	339.0
OverFeat	MRE (%)	**7.5**	20.7	24.9	16.2	58.5	65.4
	RMSE (ms)	**24.3**	90.7	77.9	77.6	178.6	154.4
ResNet-V2	MRE (%)	**5.7**	7.2	10.6	5.9	9.9	65.0
	RMSE (ms)	91.1	98.1	118.1	**65.9**	175.2	584.0
GRU	MRE (%)	**6.3**	28.2	28.7	26.0	15.2	29.3
	RMSE (ms)	**29.7**	118.7	118.9	105.8	59.1	130.7
Model Name	**Prediction of GPU Memory Consumption**						
	Metrics	**DNNPerf**	**BiRNN**	**ARNN**	**MLP**	**GBDT**	**BRP-NAS**
Overall	MRE (%)	**13.3**	21.6	25.7	23.1	13.5	23.5
	RMSE (GB)	**1.9**	2.8	3.0	2.6	2.3	4.1
AlexNet	MRE (%)	13.5	30.3	42.6	38.2	16.2	**11.2**
	RMSE (GB)	**1.3**	2.6	3.5	3.2	1.5	0.9
VGG	MRE (%)	**15.8**	17.9	17.9	16.8	16.5	29.0
	RMSE (GB)	**2.3**	3.2	2.6	2.4	3.0	4.6
OverFeat	MRE (%)	**13.6**	28.7	35.1	34.5	14.0	13.6
	RMSE (GB)	1.2	2.5	3.0	3.0	1.3	**1.1**
ResNet-V2	MRE (%)	**8.3**	14.3	15.1	9.6	8.4	30.2
	RMSE (GB)	**1.6**	2.8	2.9	1.7	1.8	4.7
GRU	MRE (%)	13.8	18.0	20.1	17.8	**12.1**	30.3
	RMSE (GB)	**2.3**	3.0	3.0	2.4	2.8	5.5

experimental results show that DNNPerf possesses strong generalization ability.

C. 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 replaces the ANEE layer (Section III-D) with GAT [23]. Since GAT cannot perform local encoding on edge features through message passing, this experiment aims to evaluate the effectiveness of ANEE.
2) DNNPerf-StandardScaler employs *StandardScaler* [62] to normalize features "by removing the mean and scaling to unit variance." [58] This experiment aims to evaluate which normalization method is more effective.
3) DNNPerf-NoTensorCost excludes the tensor size and computation cost features listed in Tables I and II. This experiment aims to evaluate the impact of runtime performance-related features.
4) DNNPerf-ConcatEdge computes the sum of the node and edge features separately, concatenates the results into a vector, and inputs the vector to MLP.
5) DNNPerf-AvgReadout calculates the average of all node features before feeding them into MLP. DNNPerf-AvgReadout and the above DNNPerf-ConcatEdge are used to evaluate the effectiveness of different global aggregation methods.

Table VIII shows 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 decreased: the overall MRE/RMSE values increase by 4.3%/42.8 for the training time prediction and 3.4%/0.2 for the GPU memory consumption prediction. Nevertheless, the training throughput and execution time of DNNPerf-GAT improved from 737.2 graphs/second and 7.42 ms to 847.3 graphs/second and 4.74 ms. We do not try EGNN [24] 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 slightly lower, the difference is very small. Thirdly, we find that the features of the tensor size and computation cost are noticeably effective for the runtime performance prediction. After excluding them, the overall MRE/RMSE values increase by 8.5%/33.2 for the training time prediction and 10.7%/1.5 for the GPU memory consumption prediction. Finally, the results of DNNPerf-ConcatEdge and DNNPerf-AvgReadout show that concatenating the node and edge features and averaging the node features do not improve prediction accuracy. The current global aggregation, which sums all the node features, is more suitable to represent the accumulated runtime performance of a computation graph.

TABLE VIII
ABLATION STUDY.

Ablation Description	Training Time		GPU Memory Consumption	
	MRE (%)	RMSE (ms)	MRE (%)	RMSE (GB)
DNNPerf	**7.4**	58.5	**13.7**	**1.8**
DNNPerf-GAT	11.7	101.3	17.1	2.0
DNNPerf-StandardScaler	11.8	**56.0**	15.0	2.0
DNNPerf-NoTensorCost	15.9	91.7	24.4	3.3
DNNPerf-ConcatEdge	8.0	60.5	15.2	1.9
DNNPerf-AvgReadout	20.9	100.9	19.8	2.4

VI. DISCUSSION

A. Extensibility of DNNPerf

Currently, DNNPerf has supported representative real-world DL models and 70+ types of operators. Users can extend its capabilities by incorporating new operators to support additional models. To add a new operator, users need to formulate the features analytically based on its semantics and implement the corresponding feature extraction scripts. Users must also provide training data that contains the new operator.

DNNPerf can be easily adapted to predict a variety of runtime performance metrics for both model training and inference, such as inference time, GPU utilization, and GPU power consumption. To achieve this, users need to collect new training data with the relevant metrics as labels. Note that for prediction tasks related to model inference, the front-end parser should exclude operators involved in backpropagation from the computation graph.

B. Training Cost of DNNPerf

A larger training dataset is generally helpful in 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 varying sizes in Table IX. The original training dataset comprises 9,964 model configurations

TABLE IX
TRAINING COST AND ACCURACY OF DNNPERF ON FIVE DATASETS OF DIFFERENT SIZES.

Sample Size (Ratio)	Memory RMSE (GB)	Memory MRE (%)	Time RMSE (ms)	Time MRE (%)	Collection Time	Training Time
9,964 (100%)	1.8	13.7	58.5	7.4	2.9 h (20 collectors)	2.5 h
4,982 (50%)	2.0	16.5	96.4	16.2	2.9 h (10 collectors)	1.5 h
1,992 (20%)	2.4	20.5	118.9	18.5	3.8 h (3 collectors)	1.0 h
996 (10%)	2.6	23.6	120.2	18.5	2.9 h (2 collectors)	30 min
498 (5%)	2.7	24.4	141.5	24.1	2.9 h (1 collector)	27 min

(excluding the testing and unseen ones), from which the other datasets are randomly generated.

The collection of training data is efficient, as it can be done in parallel, resulting in modest collection costs. For instance, we collected the original training dataset in just 2.9 hours using twenty collectors, which is equal to collecting the smallest dataset of 498 model configurations using one collector. The more collectors, the less collection time. In addition, training our GNN-based model is relatively quick, as we completed the training with the original dataset in 2.5 hours. Shrinking the dataset shortens the training time noticeably, but the accuracy of DNNPerf also decreases. However, the results are still acceptable because the MRE values remain within 25% even with only 498 training samples. Our findings indicate that DNNPerf can be easily adapted to a new deployment environment at a reasonable cost.

C. Generality of DNNPerf

Our GNN-based approach is general and can be adapted to other DL frameworks, such as PyTorch, as well as other devices, such as the NVIDIA P100. To assess the generality of DNNPerf, we have collected a total of 5,064 HPO (Hyperparameter Optimization) model configurations implemented using PyTorch v1.5.1 on a single 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 [63] to normalize the data prior to the MLP layer of DNNPerf. In addition, the input tensor size feature is enhanced to prevent the batch size information from disappearing. We set the number of message-passing rounds in ANEE to 2 and the learning rate to 0.001.

The HPO dataset comprises the configurations of six real-world DL models (AlexNet [48], LeNet [45], ResNet-V1 [1], VGG-11, VGG-16, VGG-19 [25], and Inception-V3 [46]). The specific combinations of hyperparameter values are outlined in Section IV-A. The full dataset was randomly split for training (80%) and testing (20%). We then retrained DNNPerf (for 200 epochs) and the baselines using the new dataset.

As shown in Table X, the experimental results show that the overall MRE/RMSE improvement over the baseline approaches ranges from 12.4%/20.5 to 94.4%/46.5 in predicting the training time and from 0.3%/1.6 to 45.0%/2.2 in predicting the GPU memory consumption. These findings indicate that DNNPerf can be adapted to various frameworks and devices.

D. Stability of the 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 dynamic structural changes (Section III-A). In order to analyze such stability, we conducted 16 experiments on the VGG, ResNet-V1, Inception-V3, and LSTM models under both TensorFlow and PyTorch. We trained every model configuration for an epoch of 1,000 iterations. Our analysis of the collected data from each iteration (excluding the warm-up ones) shows that the training time across iterations is fairly stable, with a relative standard error (i.e., standard error [64] divided by the mean) ranging from 0.2% to 1.6%. Therefore, although DNNPerf predicts the training time per iteration, we can still calculate the total training time by multiplying the iteration count.

TABLE X
EXPERIMENTAL RESULTS OF THE PYTORCH HPO MODELS ON ONE NVIDIA P100 GPU.

Model Name	Prediction of Training Time						
	Metrics	DNNPerf	BiRNN	ARNN	MLP	GBDT	BRP-NAS
Overall	MRE (%)	**18.491**	105.3	93.1	68.8	112.9	30.9
	RMSE (ms)	**26.5**	47.8	48.1	47.0	54.9	73.0
LeNet	MRE (%)	22.7	104.2	90.1	36.5	130.2	**12.8**
	RMSE (ms)	20.7	22.2	21.9	21.6	**20.3**	21.6
ResNet-V1	MRE (%)	**11.4**	69.0	63.5	53.2	69.8	18.4
	RMSE (ms)	**18.5**	28.1	27.6	26.8	27.2	27.3
Inception-V3	MRE (%)	**11.2**	8.7	8.6	10.7	14.2	66.0
	RMSE (ms)	**32.5**	60.3	60.8	58.9	76.4	130.3
VGG	MRE (%)	**39.6**	248.1	199.2	230.4	250.2	46.5
	RMSE (ms)	**42.8**	83.2	84.0	83.2	93.2	96.3
Model Name	**Prediction of GPU Memory Consumption**						
	Metrics	**DNNPerf**	**BiRNN**	**ARNN**	**MLP**	**GBDT**	**BRP-NAS**
Overall	MRE (%)	**14.6**	59.5	51.8	14.9	17.5	44.6
	RMSE (ms)	1.2	3.0	2.8	1.2	**0.8**	3.4
LeNet	MRE (%)	35.5	74.1	66.0	36.9	**17.5**	53.7
	RMSE (ms)	2.2	4.2	4.1	2.2	**0.6**	3.1
ResNet-V1	MRE (%)	**4.1**	26.5	17.4	4.5	4.9	8.9
	RMSE (ms)	0.6	3.0	2.0	**0.6**	0.6	1.1
Inception-V3	MRE (%)	**7.5**	7.6	15.5	9.6	9.0	53.9
	RMSE (ms)	**0.7**	0.8	1.1	0.9	0.9	4.0
VGG	MRE (%)	11.6	41.5	40.5	**11.0**	10.6	92.7
	RMSE (ms)	1.2	3.1	3.3	1.1	**1.0**	5.6

E. Threats to Validity

Threats to Internal Validity. We examine the framework source code carefully to extract many performance-related factors as the features, such as the tensor size, operator computation cost (FLOPs), and device bandwidth (Tables I and II). However, there exist hidden factors resulting in unexpected fluctuations in runtime performance. For instance, the proprietary NVIDIA CUDA, cuDNN, and cuBLAS APIs invoked by the operator implementations use unpredictable temporary GPU memory called *workspace* to boost runtime performance. We refine our feature extraction to mitigate this threat: we consult the NVIDIA development documentation, profile the APIs using *nvprof*, and analyze the framework runtime logs.

Threats to External Validity. In reality, there exist many different kinds of DL models and fairly large configuration spaces, which may reduce the effectiveness of DNNPerf. We mitigate this threat by (1) enlarging the training set with more diverse neural architectures and hyperparameter combinations; (2) supporting more types of DL operators. The experimental

results confirm that DNNPerf is generally effective, even on the configurations of unseen DL models (Section V-B). Another threat is that we collect and evaluate configurations of only TensorFlow [29] and PyTorch [30] models, but there are other DL frameworks such as MXNet [65]. However, since these frameworks use the same abstraction to represent models and have similar runtime implementations, we believe that our approach is general and can be adapted to support other frameworks.

VII. Related Work

Performance prediction for configurable systems. Many researchers focus on predicting the performance of configurable systems in a deployment environment [11]–[16], [66]–[68]. For example, DeepPerf [11] used less training data to train a deep sparse neural network but still achieved much higher prediction accuracy. It treated a configurable system as a black box and ignored its internal mechanism since the system was too large and complex. In contrast, our work analyses both DL models and framework implementations carefully to extract performance-related features as much as possible.

Performance prediction for DL models. Recently, there have been some studies on predicting the runtime performance of DL models [17]–[20], [57], [69]. For example, Paleo [17] estimated execution time from FLOPs, and DNNMem [18] pre-built an analytic model for GPU memory consumption estimation. However, these analytic approaches require extensive manual work and are limited to specific tasks. Daniel et al. [19] trained a DL model to predict the execution time of parts of a target model, which could then be combined to estimate the whole execution time. However, this approach ignored graph-level factors that could impact performance, such as kernel algorithm selection. Yeung et al. [20] predicted GPU utilization based on FLOPS, input data size, and number of convolutional layers, but did not consider other important factors such as neural architecture, operator execution order, and I/O cost. The AutoML community has also expressed interest in predicting the learning performance (e.g., predictive accuracy) of DL models [57], [70]–[77]. For instance, BRP-NAS "proposed a GCN-based predictor for the end-to-end latency," [57] using operator type and computation graph structure as features; however, its generalization ability of cross-model prediction is limited. PPP-Net employed a Recurrent Neural Network (RNN) to predict accuracy from the neural architecture, which "avoided time-consuming training to obtain true accuracy but with a slight drawback of regression error." [70] Compared to previous work, DNNPerf captures not only operator-level features but also computation graph information and hidden factors within frameworks. Our general GNN-based approach requires less manual effort and delivers improved prediction results.

VIII. Conclusion

In this paper, we have presented DNNPerf, a novel runtime performance prediction tool for deep learning models. We investigate performance-related features that are derived from both the semantics of computation graphs and the hidden factors of frameworks. By employing a GNN-based approach, DNNPerf encodes these features and accurately predicts training time and GPU memory consumption. DNNPerf is also effective and robust to various choices of hyperparameters and neural architectures, even to unseen models.

References

[1] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 770–778.

[2] W. Xiong, J. Droppo, X. Huang, F. Seide, M. Seltzer, A. Stolcke, D. Yu, and G. Zweig, "Achieving human parity in conversational speech recognition," *CoRR*, vol. abs/1610.05256, 2016. [Online]. Available: http://arxiv.org/abs/1610.05256

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186.

[4] R. Zhang, W. Xiao, H. Zhang, Y. Liu, H. Lin, and M. Yang, "An empirical study on program failures of deep learning jobs," in *Proceedings of the 42nd International Conference on Software Engineering*, ser. ICSE '20. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1159–1170.

[5] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations*, 2018.

[6] I. M. Verbauwhede, C. J. Scheers, and J. M. Rabaey, "Memory estimation for high level synthesis," in *Proceedings of the 31st Annual Design Automation Conference*, ser. DAC '94. New York, NY, USA: ACM, 1994, pp. 143–148.

[7] L. Unnikrishnan, S. D. Stoller, and Y. A. Liu, "Automatic accurate stack space and heap space analysis for high-level languages," Indiana University, Tech. Rep., 2000.

[8] V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine, "Parametric prediction of heap memory requirements," in *Proceedings of the 7th International Symposium on Memory Management*, ser. ISMM '08. New York, NY, USA: ACM, 2008, pp. 141–150.

[9] E. Albert, S. Genaim, and M. Gómez-Zamalloa, "Parametric inference of memory requirements for garbage collected languages," in *Proceedings of the 2010 International Symposium on Memory Management*, ser. ISMM '10. New York, NY, USA: ACM, 2010, pp. 121–130.

[10] D.-H. Chu, J. Jaffar, and R. Maghareh, "Symbolic execution for memory consumption analysis," in *Proceedings of the 17th ACM SIGPLAN/SIGBED Conference on Languages, Compilers, Tools, and Theory for Embedded Systems*, ser. LCTES 2016. New York, NY, USA: ACM, 2016, pp. 62–71.

[11] H. Ha and H. Zhang, "Deepperf: Performance prediction for configurable software with deep sparse neural network," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 1095–1106.

[12] J. Guo, K. Czarnecki, S. Apely, N. Siegmundy, and A. Wasowski, "Variability-aware performance prediction: A statistical learning approach," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '13. IEEE Press, 2013, p. 301–311.

[13] J. Guo, Y. Dingyu, N. Siegmund, S. Apel, A. Sarkar, P. Valov, K. Czarnecki, A. Wasowski, and H. Yu, "Data-efficient performance learning for configurable systems," *Empirical Software Engineering*, vol. 23, 06 2018.

[14] N. Siegmund, A. Grebhahn, S. Apel, and C. Kästner, "Performance-influence models for highly configurable systems," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2015. New York, NY, USA: Association for Computing Machinery, 2015, pp. 284–294.

[15] N. Siegmund, S. S. Kolesnikov, C. Kästner, S. Apel, D. Batory, M. Rosenmüller, and G. Saake, "Predicting performance via automated feature-interaction detection," in *2012 34th International Conference on Software Engineering (ICSE)*, 2012, pp. 167–177.

[16] N. Siegmund, M. Rosenmuller, C. Kastner, P. G. Giarrusso, S. Apel, and S. S. Kolesnikov, "Scalable prediction of non-functional properties in software product lines," in *2011 15th International Software Product Line Conference*, 2011, pp. 160–169.

[17] H. Qi, E. R. Sparks, and A. Talwalkar, "Paleo: A performance model for deep neural networks," in *Proceedings of ICLR*, 2017.

[18] Y. Gao, Y. Liu, H. Zhang, Z. Li, Y. Zhu, H. Lin, and M. Yang, "Estimating gpu memory consumption of deep learning models," in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2020. New York, NY, USA: Association for Computing Machinery, 2020, pp. 1342–1352.

[19] D. Justus, J. Brennan, S. Bonner, and A. S. McGough, "Predicting the computational cost of deep learning models," in *2018 IEEE International Conference on Big Data (Big Data)*, 2018, pp. 3873–3882.

[20] G. Yeung, D. Borowiec, A. Friday, R. Harper, and P. Garraghan, "Towards GPU utilization prediction for cloud deep learning," in *12th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 20)*. USENIX Association, Jul. 2020.

[21] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, and G. Monfardini, "The graph neural network model," *IEEE Transactions on Neural Networks*, vol. 20, no. 1, pp. 61–80, 2009.

[22] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[23] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *International Conference on Learning Representations*, 2018.

[24] L. Gong and Q. Cheng, "Exploiting edge features for graph neural networks," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2019, pp. 9203–9211.

[25] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1409.1556

[26] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Learning Representations by Back-Propagating Errors*. Cambridge, MA, USA: MIT Press, 1988, pp. 696–699.

[27] H. Pham, M. Guan, B. Zoph, Q. Le, and J. Dean, "Efficient neural architecture search via parameters sharing," in *Proceedings of the 35th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, J. Dy and A. Krause, Eds., vol. 80. PMLR, 10–15 Jul 2018, pp. 4095–4104.

[28] L. Mason, J. Baxter, P. Bartlett, and M. Frean, "Boosting algorithms as gradient descent," in *Proceedings of the 12th International Conference on Neural Information Processing Systems*, ser. NIPS '99. Cambridge, MA, USA: MIT Press, 1999, p. 512–518.

[29] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for Large-Scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 265–283.

[30] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, vol. 32. Curran Associates, Inc., 2019, pp. 8024–8035.

[31] Y. LeCun, B. Boser, J. Denker, D. Henderson, R. Howard, W. Hubbard, and L. Jackel, "Handwritten digit recognition with a back-propagation network," in *Advances in Neural Information Processing Systems*, D. Touretzky, Ed., vol. 2. Morgan-Kaufmann, 1990.

[32] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[33] F. Chollet *et al.*, "Keras," https://keras.io, 2015.

[34] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[35] J. J. Hopfield, "Neural networks and physical systems with emergent collective computational abilities," *Proceedings of the National Academy of Sciences*, vol. 79, no. 8, pp. 2554–2558, 1982.

[36] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," in *International Conference on Learning Representations (ICLR)*, 2017.

[37] R. M. Larsen and T. Shpeisman, "Tensorflow graph optimizations," 2019.

[38] PyTorch, "The topological sorting algorithm for the graph transformation subsystem," https://github.com/pytorch/pytorch/blob/v1.5.1/caffe2/core/nomnigraph/include/nomnigraph/Graph/TopoSort.h#L26, 2020.

[39] S. Haykin, *Neural Networks: A Comprehensive Foundation*. Prentice Hall, 1994.

[40] Wikipedia contributors, "One-hot — Wikipedia, the free encyclopedia," 2021, [Online; accessed 19-January-2021].

[41] TensorFlow, "The tf.layers.conv2d api," https://github.com/tensorflow/docs/blob/r1.13/site/en/api_docs/python/tf/layers/Conv2D.md, 2019.

[42] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, "cudnn: Efficient primitives for deep learning," *CoRR*, 2014. [Online]. Available: http://arxiv.org/abs/1410.0759

[43] Scikit-learn, "The sklearn.preprocessing.minmaxscaler api," https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html, 2022.

[44] M. Claesen and B. D. Moor, "Hyperparameter search in machine learning," *CoRR*, vol. abs/1502.02127, 2015. [Online]. Available: http://arxiv.org/abs/1502.02127

[45] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[46] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," *CoRR*, vol. abs/1512.00567, 2015. [Online]. Available: http://arxiv.org/abs/1512.00567

[47] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997.

[48] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1*, ser. NIPS '12. Red Hook, NY, USA: Curran Associates Inc., 2012, pp. 1097–1105.

[49] P. Sermanet, D. Eigen, X. Zhang, M. Mathieu, R. Fergus, and Y. LeCun, "Overfeat: Integrated recognition, localization and detection using convolutional networks," in *International Conference on Learning Representations (ICLR 2014)*. CBLS, April 2014.

[50] K. Cho, B. van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder–decoder for statistical machine translation," in *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Doha, Qatar: Association for Computational Linguistics, Oct. 2014, pp. 1724–1734.

[51] W. Ling, E. Grefenstette, K. M. Hermann, T. Kociský, A. W. Senior, F. Wang, and P. Blunsom, "Latent predictor networks for code generation," *CoRR*, vol. abs/1603.06744, 2016. [Online]. Available: http://arxiv.org/abs/1603.06744

[52] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, "Summarizing source code using a neural attention model," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 2073–2083.

[53] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep api learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 631–642.

[54] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *Proceedings of the 41st International Conference on Software Engineering*, ser. ICSE '19. IEEE Press, 2019, pp. 783–794.

[55] N. Silberman and S. Guadarrama, "Tensorflow-slim image classification model library," https://github.com/tensorflow/models/tree/master/research/slim/nets, 2016.

[56] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, H. Shen, M. Cowan, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated End-to-End optimizing compiler for deep learning," in *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. Carlsbad, CA: USENIX Association, Oct. 2018, pp. 578–594.

[57] L. Dudziak, T. Chau, M. Abdelfattah, R. Lee, H. Kim, and N. Lane, "Brp-nas: Prediction-based nas using gcns," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 10 480–10 490.

[58] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and Édouard Duchesnay, "Scikit-learn: Machine learning in python," *Journal of Machine Learning Research*, vol. 12, no. 85, pp. 2825–2830, 2011. [Online]. Available: http://jmlr.org/papers/v12/pedregosa11a.html

[59] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, Z. Huang, Q. Guo, H. Zhang, H. Lin, J. Zhao, J. Li, A. J. Smola, and Z. Zhang, "Deep graph library: Towards efficient and scalable deep learning on graphs," *CoRR*, vol. abs/1909.01315, 2019. [Online]. Available: http://arxiv.org/abs/1909.01315

[60] M. Feurer and F. Hutter, *Hyperparameter Optimization*. Cham: Springer International Publishing, 2019, pp. 3–33. [Online]. Available: https://doi.org/10.1007/978-3-030-05318-5_1

[61] Microsoft, "Nd-series virtual machines," https://docs.microsoft.com/en-us/azure/virtual-machines/nd-series, 2022.

[62] Scikit-learn, "The sklearn.preprocessing.standardscaler api," https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html, 2022.

[63] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML '15. JMLR.org, 2015, pp. 448–456.

[64] Wikipedia, "Standard error — Wikipedia, the free encyclopedia," http://en.wikipedia.org/w/index.php?title=Standard%20error&oldid=1064719702, 2021.

[65] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, "MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems," *CoRR*, vol. abs/1512.01274, 2015. [Online]. Available: http://arxiv.org/abs/1512.01274

[66] A. Sarkar, J. Guo, N. Siegmund, S. Apel, and K. Czarnecki, "Cost-efficient sampling for performance prediction of configurable systems (t)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015, pp. 342–352.

[67] E. Thereska, B. Doebel, A. X. Zheng, and P. Nobel, "Practical performance models for complex, popular applications," in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, ser. SIGMETRICS '10. New York, NY, USA: Association for Computing Machinery, 2010, pp. 1–12.

[68] Y. Zhang, J. Guo, E. Blais, and K. Czarnecki, "Performance prediction of configurable software systems by fourier learning," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '15. IEEE Press, 2015, pp. 365–373.

[69] Y.-C. Liao, C.-C. Wang, C.-H. Tu, M.-C. Kao, W.-Y. Liang, and S.-H. Hung, "Perfnetrt: Platform-aware performance modeling for optimized deep neural networks," in *2020 International Computer Symposium (ICS)*, 2020, pp. 153–158.

[70] J.-D. Dong, A.-C. Cheng, D.-C. Juan, W. Wei, and M. Sun, "Ppp-net: Platform-aware progressive search for pareto-optimal neural architectures," in *International Conference on Learning Representations (ICLR) Workshop*, 2018.

[71] T. Domhan, J. T. Springenberg, and F. Hutter, "Speeding up automatic hyperparameter optimization of deep neural networks by extrapolation of learning curves," in *Proceedings of the 24th International Conference on Artificial Intelligence*, ser. IJCAI '15. AAAI Press, 2015, p. 3460–3468.

[72] M. Gargiani, A. Klein, S. Falkner, and F. Hutter, "Probabilistic rollouts for learning curve extrapolation across hyperparameter settings," *CoRR*, vol. abs/1910.04522, 2019. [Online]. Available: http://arxiv.org/abs/1910.04522

[73] A. Klein, S. Falkner, J. T. Springenberg, and F. Hutter, "Bayesian neural networks for predicting learning curves," 2016.

[74] B. Baker, O. Gupta, R. Raskar, and N. Naik, "Practical neural network performance prediction for early stopping," *CoRR*, vol. abs/1705.10823, 2017. [Online]. Available: http://arxiv.org/abs/1705.10823

[75] T. Unterthiner, D. Keysers, S. Gelly, O. Bousquet, and I. Tolstikhin, "Predicting neural network accuracy from weights," 2021.

[76] Y. Xin, C. Xu, and H. Zhao, "Predicting future performance of convolutional neural networks in early training stages," Tech. Rep., 2019.

[77] D. Choi, H. Cho, and W. Rhee, "On the difficulty of dnn hyperparameter optimization using learning curve prediction," in *TENCON 2018 - 2018 IEEE Region 10 Conference*, 2018, pp. 0651–0656.