Prediction of the Resource Consumption of Distributed Deep Learning Systems

GYEONGSIK YANG, Department of Computer Science and Engineering, Korea University, South Korea
CHANGYONG SHIN, Department of Computer Science and Engineering, Korea University, South Korea
JEUNGHWAN LEE, Department of Computer Science and Engineering, Korea University, South Korea
YEONHO YOO, Department of Computer Science and Engineering, Korea University, South Korea
CHUCK YOO, Department of Computer Science and Engineering, Korea University, South Korea

The prediction of the resource consumption for the distributed training of deep learning models is of paramount importance, as it can inform a priori users how long their training would take and also enable users to manage the cost of training. Yet, no such prediction is available for users because the resource consumption itself varies significantly according to "settings" such as GPU types and also by "workloads" like deep learning models. Previous studies have aimed to derive or model such a prediction, but they fall short of accommodating the various combinations of settings and workloads together. This study presents *Driple* that designs graph neural networks to predict the resource consumption of diverse workloads. *Driple* also designs transfer learning to extend the graph neural networks to adapt to differences in settings. The evaluation results show that *Driple* can effectively predict a wide range of workloads and settings. At the same time, *Driple* can efficiently reduce the time required to tailor the prediction for different settings by up to 7.3×.

CCS Concepts: • **Computing methodologies** → **Distributed artificial intelligence**; *Machine learning*; • **Computer systems organization** → **Distributed architectures**.

Additional Key Words and Phrases: distributed deep learning, resource prediction, training time prediction, graph neural networks, transfer learning

ACM Reference Format:
Gyeongsik Yang, Changyong Shin, Jeunghwan Lee, Yeonho Yoo, and Chuck Yoo. 2022. Prediction of the Resource Consumption of Distributed Deep Learning Systems. *Proc. ACM Meas. Anal. Comput. Syst.* 6, 2, Article 29 (June 2022), 25 pages. https://doi.org/10.1145/3530895

1 INTRODUCTION

Currently, deep learning is employed in a wide variety of application fields, from computer vision [31, 43, 58] to natural language processing [25, 30, 33] to financial transactions [15, 26]. To support its increasingly diverse application fields, the deep learning model continuously evolves into a progressively larger one with the addition of more layers and parameters. Over the last ten years, the parameter size of deep learning models has become 2917× larger, resulting in much longer

Authors' addresses: Gyeongsik Yang, ksyang@os.korea.ac.kr, Department of Computer Science and Engineering, Korea University, Seoul, South Korea, 02841; Changyong Shin, cyshin@os.korea.ac.kr, Department of Computer Science and Engineering, Korea University, Seoul, South Korea, 02841; Jeunghwan Lee, jhlee21@os.korea.ac.kr, Department of Computer Science and Engineering, Korea University, Seoul, South Korea, 02841; Yeonho Yoo, yhyoo@os.korea.ac.kr, Department of Computer Science and Engineering, Korea University, Seoul, South Korea, 02841; Chuck Yoo, chuckyoo@os.korea.ac.kr, Department of Computer Science and Engineering, Korea University, Seoul, South Korea, 02841.

2476-1249/2022/6-ART29 $15.00
https://doi.org/10.1145/3530895

training times [19, 43]. Due to this excessive training time, the state-of-art distributed training (DT), which utilizes multiple nodes (workers), is receiving considerable attention.

Despite the advances in DT, users are still puzzled when they try to execute their deep learning models because it is unknown how to configure execution settings such as GPU type and the number of GPUs for their best interests and how long a model would take to be trained. Largely these questions are yet unanswered, which leaves users to come up with ad-hoc methods [27, 73].

The challenges posed by these questions come from multi-faceted complexities. One axis of complexities is related to the training settings. For example, there are many choices of devices for running deep learning models, such as 2080 Ti [4], Titan RTX [1], V100 [2], and A100 [5] GPUs. Also, the number of parameter servers (PSs) and workers can be configured. The GPUs of workers can be collocated in the same server and connected via PCIe [60]. Additionally, they can be placed in separate GPU servers communicating via 40 GbE or high-speed RDMA. These design choices in the training settings cause the training time and training efficacy to be highly different [21, 27, 53, 73]. The other axis of complexities is related to workloads, which means that the DT training time and resource consumption depend on the models, training datasets, and hyperparameters.

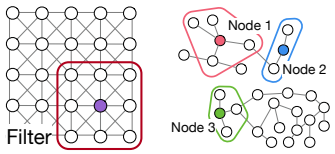
There have been a few efforts to address these challenges. Justus et al. [36] attempted to predict the execution time of commonly used layers in deep learning models, such as the convolution layer. Habitat [27] measured the time of executing a single iteration on a GPU and scaled it to that of another GPU via a pre-trained multi-layer perceptron. Lin et al. [46] profiled GPU and CPU utilization for DT and derived the time for training one mini-batch. These efforts are based on the cyclic repetitiveness of forward propagation and backward propagation against a dataset at every iteration. However, when a DT setting such as the number of workers is changed or a different hyperparameter is used in the workload, their prediction model needs to be rebuilt or re-derived. In other words, they fall short of accommodating the variety of settings and workloads together.

This study presents *Driple* that designs graph neural networks (GNNs) to predict the resource consumption of diverse workloads. First, we use a graph as a basic building block because most DT codes (workloads) are written with ML libraries (e.g., TensorFlow [13], PyTorch [51], Caffe [35], MXNet [20], and Theano [59]) that internally convert the codes to computational graphs. We exploit and augment this computational graph in order for any DT codes to be used as input to *Driple* so that *Driple* can encompass a broad spectrum of DT workloads. Second, we focus on predicting the resource consumption of the workloads because training time can be calculated from resource consumption [46]. The resource scheduler can also take advantage of the resource consumption [27] as well.

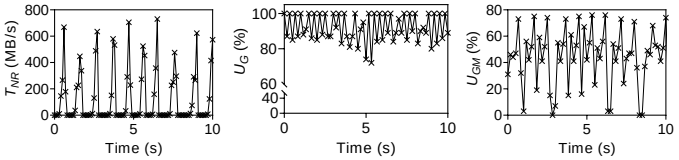
Furthermore, to adapt the prediction power of *Driple* for various settings, *Driple* designs transfer learning from a trained GNN to a new GNN for a new DT setting. Training a separate GNN for each setting requires considerable efforts in terms of the training dataset generation and the training time. To reduce such efforts, *Driple* re-uses the already-trained GNN for a new setting by scrutinizing the layers and parameters of the trained GNN.

Driple achieves the following contributions:

- This paper is the first attempt to design GNNs for the prediction of resource consumption in distributed deep learning.
- We suggest four key resources for prediction (i.e., GPU utilization, GPU memory utilization, network TX throughput, and network RX throughput). For each resource, *Driple* predicts the resource consumption amount, burst duration, and idle duration, which results in 12 prediction metrics.
- We show that graph-based *Driple* can achieve good accuracy in predicting resource consumption for the variety of DT workloads and DT settings.



(a) CNN on image. (b) GNN on graph.



(a) T_{NR}, workload 1. (b) U_G, workload 2. (c) U_{GM}, workload 3.

Fig. 1. Machine learning on different input types. See more details in §2.2.

Fig. 2. Resource consumption of three workloads. See more details in §2.3.

- With transfer learning, *Driple* reduces the required dataset and training time by up to 2.5× and 7.3×, respectively, while the prediction accuracy is maintained close to the model without transfer learning.

2 BACKGROUND AND MOTIVATION

2.1 Distributed Training

In DT, data parallel is one of the most widely used strategies because of its ease of implementation and training time effectiveness. Whenever a backward propagation is completed in each worker, the gradients generated per worker are gathered to update the model's parameters. This update requires network communications, which can either be synchronous or asynchronous. Also, there are two representative methods on how parameters are updated: PS and all-reduce [6]. PS is a dedicated node that collects gradients, calculates model parameters, and distributes the updated parameters to workers. All-reduce cooperatively gives and takes gradients among workers. Each worker locally updates its model parameters based on the gradients received from others. In this study, we focus on data parallel based on PS [53], considering its wide use.[1]

2.2 Graph Neural Network

Each DT workload is converted to a graph by machine learning libraries like TensorFlow. The input graph has a different size (i.e., a different number of nodes and edges). This varying size of the graphs makes it challenging to employ traditional machine learning algorithms, such as deep neural networks and convolutional neural networks (CNNs) because they work only with fixed-sized inputs (e.g., images, text, and numeric values) [64, 71]. Figure 1a shows an example of CNN applied to an image. An image of a fixed size (number of pixels) is given to a CNN. The CNN then aggregates information from a fixed number of pixels (called filter) and outputs a vector value that represents a section of the image known as "convolved feature" [43]. In Figure 1a, for example, a filter of 9 pixels yields one convolved feature from the image's lower-right corner. As the input and filter sizes are fixed, each input image results in the same number of convolved features.

On the other hand, in GNN, the information of a graph is primarily aggregated per node. This aggregated result is referred to as "node embedding." Specifically, node embedding is calculated as an aggregation of the features of neighbor nodes in a specific hop distance of each node. In Figure 1b, within one hop distance, node 1 has four neighbor nodes, but nodes 2 and 3 have two and three nodes, respectively. GNN's node embedding can aggregate a different number of neighbor nodes. Therefore, each GNN algorithm performs different methods for calculating node embeddings . In addition, unlike input images in CNN, GNN takes graphs that have a different number of nodes. To accommodate the diverse sizes, GNN creates "graph embedding," a fixed size vector that represents a graph. Various methods exist for creating graph embedding (see more details in §3.2).

[1]We believe that the approaches in this paper could be applied to other DT strategies as well (see §7).

Table 1. Amount and duration of Figure 2. See more details in §2.3.

	T_{NR} **(workload 1)**	U_G **(workload 2)**	U_{GM} **(workload 3)**
Burst	374.3 MB/s, 0.33 s	98.8%, 0.19 s	54.4%, 0.64 s
Idle	4.5 MB/s, 0.7 s	84.6%, 0.21 s	16.8%, 0.28 s

Moreover, GNN has novel hyperparameters, such as the number of layers in its model. A layer in GNN creates a node embedding by aggregating information from nearby nodes in a single hop. Also, the node embedding can aggregate neighbor nodes up to n hops away by stacking n layers. The ideal choice for the number of layers differs per the use-case of GNN [23, 64, 71].

The graph embedding is further used to make predictions in combination with traditional machine learning algorithms, such as multi-layer perceptron (MLP). Here, we design the prediction model of *Driple* based on GNN (§3.2) to predict the resource consumption of DT workloads over various DT settings. Similar to the existing deep learning algorithms, the GNN is trained by repetitive iterations (forward and backward propagations) against the training dataset.

2.3 Motivating Example

Here, we identify the resource consumption characteristics of DT workloads. We run three exemplar DT workloads and profile their resource consumption. The details of the workloads are as follows.

- **Workload 1**: NMT_Medium [40], Europarl dataset [41] (batch size 32, asynchronous training)
- **Workload 2**: DenseNet40_k12 [34], CIFAR-10 dataset [42] (batch size 512, synchronous training)
- **Workload 3**: Inception v3 [58], ImageNet dataset [24] (batch size 128, asynchronous training)

These workloads run with the DT setting of one PS and two workers, each pinned with a separate V100 GPU.[2] All PS and workers are packaged as individual containers. We focus on four resource types: 1) GPU utilization (U_G), 2) GPU memory utilization (U_{GM}), 3) network TX throughput (T_{NT}), and 4) network RX throughput (T_{NR}). These four resource types are chosen because previous studies have considered them useful in optimizing DT systems (§6). The measurements are conducted during 100 iterations.

Figures 2a, 2b, and 2c show 10 s snapshots of T_{NR}, U_G, and U_{GM} of workloads 1, 2, and 3, respectively. The x-axis of each graph represents time, and the y-axis indicates the amount of resource consumption. All three figures show cyclic patterns—repeatedly alternating data points of high and low resource consumption.

The fact that resource consumption shows cyclic patterns has been reported in previous studies (especially regarding U_{GM}) [38, 65, 66]. However, in lieu of the pattern itself, *Driple*'s focus is to predict values of resource consumption; so that deep learning model engineers or system researchers can fully manage their DT workloads with proper provisioning. Thus, we define metrics to quantify these cyclic patterns. We refer to data points of high and low resource consumptions as burst and idle points. They are quantified by two metrics—amount and duration. For example, the burst amount indicates the average amount of resources consumed by the burst data points. Also, the idle duration is the average period wherein the idle data points appear consecutively.

Table 1 shows the quantified values of the defined metrics (i.e., amounts and durations of burst and idle data points) of Figure 2. Between burst and idle points, resource consumption shows huge differences. For T_{NR}, the amount differs by 83×. Also, once T_{NR} is actively utilized, it is consumed over 0.33 s on average. Conversely, when it is idling, it idles for about 0.7 s (2.12× longer than the burst duration). In addition, significant differences are observed in U_G and U_{GM} as well. After all,

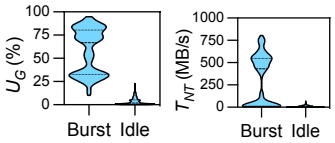
[2]We also profile the resource consumption on other GPUs (e.g., 2080 Ti and Titan RTX) but the results are omitted as they are similar to those shown in this section.

Table 2. Three factors of DT workloads. See more details in §2.4.

Model type	Model	Dataset	Hyperparameters
Image classification	AlexNet, GoogLeNet (Inception v1), Inception v3, Inception v4, ResNet101, ResNet101_v2, ResNet152, ResNet20, ResNet20_v2, ResNet32, ResNet32_v2, ResNet44, ResNet44_v2, ResNet50, ResNet50_v2, ResNet56, ResNet56_v2, ResNet110, ResNet110_v2, ResNet152, ResNet152_v2, VGG11, VGG16, VGG19, Overfeat, DenseNet100_k12, DenseNet100_k24, DenseNet40_k12	CIFAR-10, ImageNet	Batch size, parameter precision (floating-point), optimizer, data format, synchronization method
Natural language processing	NMT_Big, NMT_Medium, NMT_Small, Transformer, Transformer_AAN, Transformer_Big	Europarl	

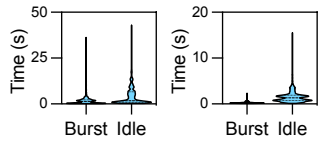
Table 3. Pattern ratio per resource. See more details in §2.4.

Pattern	U_G	U_{GM}	T_{NT}	T_{NR}
I	7.77%	9.26%	0%	0%
BIB	92.02%	90.63%	100%	100%
B	0.21%	0.11%	0%	0%



(a) U_G. (b) T_{NT}.

Fig. 3. Violin plots of resource consumption amounts. See more details in §2.4.



(a) U_G. (b) T_{NT}.

Fig. 4. Violin plots of resource consumption durations. See more details in §2.4.

we define 12 prediction targets—three metrics (i.e., burst duration, idle duration, and burst amount) each for four resource types (i.e., U_G, U_{GM}, T_{NT}, and T_{NR}). In the following subsections, we examine how sensitive the prediction targets are to DT workloads (§2.4) and to DT settings (§2.5).

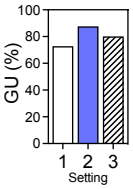
2.4 Sensitivity to DT Workloads

We profile resource consumption of various DT workloads with the DT setting in §2.3. *Driple* defines that a DT workload has three factors: 1) the model (including layers), 2) datasets used for training, and 3) hyperparameters for the training configurations.

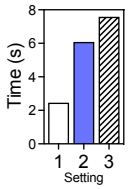
Table 2 presents the three factors of DT workloads we profile. It includes image classification [3] and natural language processing models [40]. For each model, the workload can have different datasets (e.g., CIFAR-10 and ImageNet for image classification models) and hyperparameters (e.g., 32 to 512 as batch size). We have obtained 832 measurements of the DT workloads based on the combinations of three factors in Table 2 by excluding outliers (e.g., results reporting out-of-memories on GPUs). For a systematical analysis, we use a k-means clustering algorithm that divides each data point into a certain number of clusters. With the k-means clustering algorithm, the data points of each measurement are classified into burst and idle data points.[3]

From the clustered data, we first examine whether the cyclic pattern is observed in DT workloads so that the resource consumption can be quantified by three patterns: "B," "I," and "BIB." "B" and "I" patterns indicate that the resource amount of the profiled data is largely burst or idle, respectively. Table 3 shows that the ratio of each pattern from the profiled data. We can see that the BIB pattern occurs most frequently over all types of resources and workloads—96% on average. Thus, we observe that the cyclic pattern (indicated by the BIB pattern) exists in most profiled DT workloads.

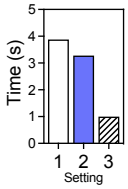
[3]We also used k-means clustering algorithm (e.g., Elbow method) to determine the optimal number of clusters that best partitions the given data. When applied to all the profiled data (3328 traces; four resources for 832 DT workloads), all data is divided into two clusters. So, we cluster the data into two clusters (i.e., burst and idle data points).



(a) burst amount.

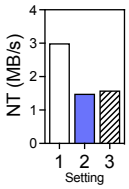


(b) burst duration.

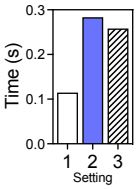


(c) idle duration.

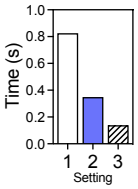
Fig. 5. Comparison of DT settings—U_G. See more details in §2.5.



(a) burst amount.



(b) burst duration.



(c) idle duration.

Fig. 6. Comparison of DT settings—T_{NT}. See more details in §2.5.

Second, we investigate the sensitivity of the metrics on DT workloads—the variance of the amount and duration by DT workloads. Figure 3 shows the distributions of the amount of burst and idle. We present the results of U_G and T_{NT} but omit the ones of U_{GM} and T_{NR} because the tendencies in the two are pretty similar to U_G and T_{NT}. In Figure 3, the distributions of burst amounts show dynamic ranges—the difference between maximum and minimum values—of U_G and T_{NT} are 84.9% (Figure 3a) and 799.1 MB/s (Figure 3b), respectively. On the other hand, the ranges for idle amounts are 87% shorter than burst amounts on average. These results indicate that the amount of burst varies highly, so very sensitive to DT workloads, while the idle amount varies relatively less.

Figure 4 shows the distributions of the durations of burst and idle. We first consider idle durations—the ranges are 42.6 s for U_G (Figure 4a) and 15.4 s for T_{NT} (Figure 4b). In terms of the burst durations, their ranges are 36 s and 2.2 s. By comparing the ranges of idle and burst durations, both idle and burst durations show high ranges for U_G (Figure 4a). For T_{NT}, the burst durations are 88% shorter than the idle durations. In summary, the burst durations are sensitive for U_G (and U_{GM}). Also, the idle durations are very sensitive for all types of resources to DT workloads to DT workloads.

2.5 Sensitivity to DT Settings

Here, we evaluate whether the prediction targets are sensitive to DT settings. DT setting is the underlying configuration to run DT workloads. In this study, DT setting is defined as a triple—{GPU, number of PSs and workers, network interconnects}. For example, the setting used in §2.3 and §2.4 is as follows: one GPU server, one PS and two workers, the two workers use a V100 GPU each (homogeneous), and V100 GPUs are connected by a PCIe bus. Let us refer to this setting as V100-P1W2/ho-PCIe (setting 1).

We profile and compare the DT workloads (Table 2) in different DT settings. Different DT settings include two separate GPU servers that are connected and communicated via 40 GbE. Each GPU server runs one PS and one worker; a total of two PSs and two workers. The first worker uses a 2080 Ti GPU, and the second uses a Titan RTX GPU (heterogeneous). Then, the settings of the first and second workers are 2080Ti-P2W2/he-40G (setting 2) and Titan RTX-P2W2/he-40G (setting 3), respectively.

With three DT settings, we measure the prediction targets of a DT workload—ResNet56 model trained by the CIFAR-10 dataset (asynchronous training, batch size 512). Figures 5 and 6 present the U_G and T_{NT} of the three settings: the x-axis represents the number of each setting, and the y-axis represents the resource consumption. We omit the results of U_{GM} and T_{NR}. Specifically, the burst amount, burst duration, and idle duration of U_G (Figure 5) show up to 1.2×, 3.1×, and 3.9× differences among the settings, respectively. The differences of T_{NT} (Figure 6) are 2×, 2.5×, and 6×. The results show that the prediction targets are highly sensitive to DT settings, given a workload.

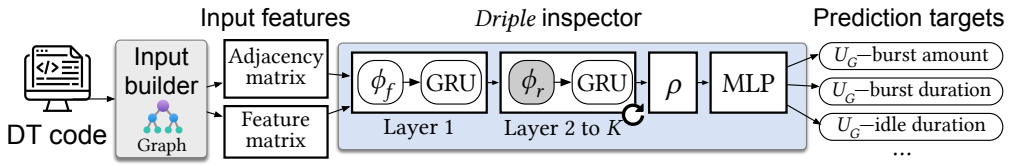


Fig. 7. Prediction workflow. See more details in §3.

We experiment with the other DT settings as well, and they also exhibit high sensitivity to DT settings. Overall, we find that the prediction targets are sensitive to both DT settings and DT workloads. This means that the prediction of resource consumption is extremely challenging because of the variety of the DT workloads as well as of DT settings.

3 *DRIPLE* DESIGN

In this section, we present *Driple* that predicts DT workloads. Figure 7 is the prediction flow that consists of "input builder" and "*Driple* inspector." The input builder extracts a computational graph from the DT code and builds input features by augmenting the graph with adjacency and feature matrices. *Driple* inspector is a GNN-based model that takes the input features. We first explain the input builder (§3.1) and *Driple* inspector (§3.2) in order. Then, we present the training process of the *Driple* inspector (§3.3).

3.1 Input Builder

3.1.1 Computational graph. Throughout this paper, we denote a computational graph as $G = (N, E)$, where N indicates a set of nodes, and E refers to a set of edges. Individual node (n) has its node feature (X_n) to express the characteristics and information on the node. The i-th node is expressed as n_i ($n_i \in N$). Each layer of the model is converted into multiple low-level operations (e.g., `Add` and `MatMul`), called "op." This op becomes n. Also, the dataset is expressed as an op (e.g., `VariableV2` in TensorFlow) that loads the data to start an iteration, so it is another n. The variables and constants necessary for each op become n (e.g., `Const`) as well. Each n has different features X_n (e.g., node type) because nodes are of different types such as computation (e.g., `MatMul`), dataset, and variables.

Also, hyperparameters of DT workload are converted as either nodes or features of nodes. For example, when the optimizer is changed (e.g., RMSProp and Momentum), the optimizer nodes in G are replaced into the nodes of the new optimizer. Also, batch size determines the amount of dataset to be fed. The batch size is reflected as n's X_n. Specifically, the ML library sets the batch size to `VariableV2`'s "tensor size" feature. As another example, the parameter precision in Table 2 indicates the type of floating-point operations. In addition, when the 16-bit floating-point operation is used, TensorFlow library changes the relevant n's X_n (e.g., `DT_HALF`). Other hyperparameters are similarly reflected in G.

We denote an edge connecting n_i and n_j is expressed as e_{ij}. Edges determine the execution order of ops (e.g., from the `VariableV2` op that loads data for an iteration to the last layers' ops). When G is created from the DT code, the deep learning library places it on devices (e.g., GPU) and executes the training following the edges, starting from the first n. For each node n of G, nodes connected through a single edge to n are called neighbors—$\mathcal{N}(n)$ is a set of neighbors.

3.1.2 Input features. Input builder produces two matrices from G: 1) adjacency matrix and 2) feature matrix in Figure 7. The adjacency matrix is for representing the nodes and edges of each graph. To be expressed as an adjacency matrix, the nodes in the graph are numbered. Given

Table 5. Features used in GNN-based optimizations. See more details in §3.1.2.

Studies	Node	Internal state
GDP [72]	Node type, previous nodes	Adjacent node IDs, output shape
HeteroG [69]	None	Tensor size
REGAL [49]	Internal required memory per node	Tensor size
Placeto [14]	Node placement	Tensor size
Driple	**Node type, grouped node size**	**Tensor size**

$G = (N, E)$, the size of the adjacency matrix is $m \times m$, where m is $|N|$. The element of the matrix of the i-th row and the j-th column has a value of one if the edge from n_i to n_j exists. If the edge does not exist, the value is zero. So, the adjacency matrix includes only the information on 1) the existence of nodes and 2) connectivity (edges) between the nodes.

The feature matrix includes X_n such as tensor size and node type. Note that the nodes in G have different types (ops) and numbers of features depending on the nodes. For example, X_n of `VariableV2` has tensor size, but `Conv2D` does not. Also, all nodes have node type as its X_n. However, in order for GNN to train graphs, each node should have the same features. Also, the number of features should not be too big because the number affects the training time and prediction accuracy [17, 64]. If the number of features in X_n is f, the feature matrix has the size of $m \times f$.

To decide which features to include in the X_n, we first checked the features used in previous GNN studies (Table 5). They used GNN for deep learning model optimizations, such as placing graph partitions between GPUs and graph optimizations (compiler). Note that these studies do not predict the resource consumption of DT workloads.

The features are classified into two categories: 1) features related to individual nodes and 2) features related to the internal states of the graph (including data transitions between nodes). We select two features as follows:

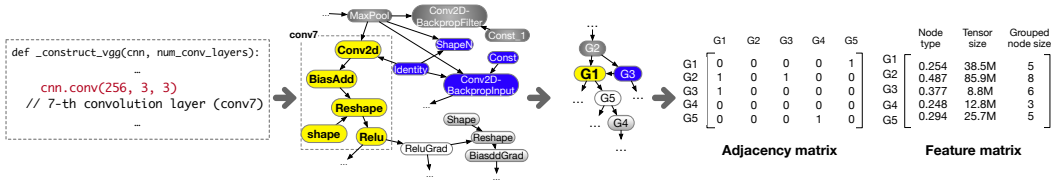
- Node type: Examples of node types include `Conv2D` and `L2Loss`. So, the node type in G is given as a text value. To take the node type as an input feature, we convert the type value into a number. Specifically, we apply frequency encoding [10] that sets the frequency of the node type that appears in a dataset.[4]
- Tensor size: The tensor size is important for computation and communication in DT. For the n that does not have tensor size as its X_n, we put zero as its tensor size.

In addition to the two features, we introduce another feature to reflect the "grouping" mechanism that *Driple* devises. The grouping mechanism is to group graph nodes into a graph with a fewer number of nodes, and it is to improve training efficiency; this will be explained in detail below (§3.1.3). This feature is called "grouped node size," which indicates how many nodes are grouped into that node.

3.1.3 Grouping. The purpose of grouping is two-fold. First, it aims to reduce the size ($|N|$ and $|E|$) of G. The graphs of the DT workloads in Table 2 have thousands of nodes and edges—the maximum node and edge numbers are 18758 and 28276 (transformer model), respectively (with average values of 2923 and 3982, respectively). These rather high numbers can make the prediction of GNN result in poor accuracy [69]. Also, because the number of edges is large, it requires too much time to calculate node embeddings; thereby, the total training time tends to increase exponentially.

Second, grouping enables batching in training graphs, which is crucial for the training speed. Batching means that a GNN is updated not per individual input data (a graph) but per set of data, called batch. For example, a batch size of 10 indicates that the model parameters are updated after

[4]Various encoding methods exist (e.g., one-hot encoding and integer encoding) for changing a categorical text value to a number value. We use frequency encoding, as it is more efficient than others in terms of GPU memory space and bias.



(a) DT workload code. (b) computational graph. (461 nodes and 606 edges) (c) grouping. (100 nodes and 203 edges) (d) input features.

Fig. 8. Example of input builder operations. See more details in §3.1.4.

10 graphs go through a GNN model. A problem here is that the number of layers in a GNN model differs as the number of nodes of a graph (m) changes (more details in §3.2). Thus, for batching, m should be identical for each graph within a batch. Obviously, input graphs have different m, so it is a challenge for *Dripple* to make batching work with grouping.

We design two grouping policies: uniform and proportional. Each policy partitions the input graphs in batches and sets the scale of "node grouping." Let M be the number of grouped nodes in a graph (e.g., 500 grouped nodes derived from 19758 original nodes). Uniform grouping then sets M to be identical across all batches.

Proportional grouping first sorts the input graphs in ascending order based on m. Then, the graphs are partitioned in the order of m by batch size. So, the first batch includes graphs with small numbers of nodes, while subsequent batches comprise the larger graphs. For each batch, we compute the average number of nodes (v) and set M to $log_{10}v$ nodes (empirically chosen considering the GPU memory size required to load a batch).

The input builder performs node grouping according to M. Node grouping is based on fluid communities algorithm [50]. The algorithm randomly selects M seed nodes in a graph and groups the remaining nodes around the seed nodes. The performance effect of node grouping on training is shown in §5.2.1.

3.1.4 Example. We present an example of input builder operations. Figure 8 shows the operations of the input builder on the VGG16 model. From the entire layers of the VGG16 model, we focus on the 7-th convolution layer (conv7). The line that defines the conv7 layer in the DT workload code (Figure 8a) generates multiple nodes in G. The nodes for mathematical operations in a forward propagation are shown in a dashed box of conv7 (Figure 8b). The other nodes are relevant variables, constants, and preceding operations. In total, 16 nodes are generated for conv7. Then, the mathematical operations in conv7 are merged into one group (G1) in Figure 8c.[5] The other nodes likewise form groups, such as G2 to G5. Finally, the input builder creates the adjacency and feature matrices (Figure 8d) which are the inputs of the *Dripple* inspector.

3.2 *Dripple* Inspector

Dripple inspector is designed based on GNN, taking adjacency and feature matrices as its input. The first part of the *Dripple* inspector is the multiple graph layers. A graph layer performs an update function (ϕ). We denote the first layer as ϕ_f and other layers ϕ_r as their update functions differ.

The first layer, ϕ_f, performs the update function of aggregating the X_{n_j} of n_j that belong to $\mathcal{N}(n_i)$ for every n_i. The aggregated value by ϕ_f becomes the node embedding of n_i, which we refer to as $h^1_{n_i}$. So, the node embedding of the first layer is formulated in Equation 1.

$$h^1_{n_i} = \phi_f = \texttt{AGGREGATE}^1(X_{n_j} | n_j \in \mathcal{N}(n_i)), i = 1, \ldots, m \quad (1)$$

[5]We show an example of uniform grouping here.

The next layers (e.g., k-th) perform two update steps (ϕ_r) per n_i: 1) ϕ_r aggregates the previous (k-1)-th layer's embedding ($h_{n_j}^{k-1}$) of n_j that belong to $\mathcal{N}(n_i)$ and 2) the aggregated embeddings are combined with the $h_{n_i}^{k-1}$ of the n_i. The way of combining differs from each GNN algorithm. Since there are multiple layers that perform ϕ_r, we denote the k-th updated embedding value of the n_i as $h_{n_i}^k$ in Equation 2 where k is a layer of ϕ_r.

$$h_{n_i}^k = \phi_r = \texttt{COMBINE}^k(h_{n_i}^{k-1}, \texttt{AGGREGATE}^k(h_{n_j}^{k-1} | n_j \in \mathcal{N}(n_i))), i = 1, \ldots, m \tag{2}$$

Suppose that the l-th G in the training dataset is G_l (G_l =(N_l, E_l) and $|N_l|$=m), and the value K is the number of layers in GNN. If K is set to be m (number of nodes in G_l), the node embeddings are aggregated m times; so the X_n of neighbors in m hops are aggregated and combined. Usually, GNN sets the number of GNN layers (K) similar to m or as half of m [23]. Empirically, we set K as half of m, showing the highest prediction accuracy in our experiment results (§5.2.2). The parameters of ϕ_r, such as weight and bias, are identical across the multiple (K-1 numbers of) ϕ_r layers. This is a common design choice to reduce the computational costs in training and inference [17, 64].

The calculations of Equations 1 and 2 depend on GNN algorithms. For example, graph convolution network (GCN) uses a normalized mean for the aggregation update function [39]. Also, graph attention network (GAT) deploys a weighted sum on $h_{n_i}^k$, with weights calculated by an attention mechanism that calculates the importance (weight) of X_n [61]. We test four GNN algorithms, i.e., GCN, GAT, graph isomorphism network (GIN) [67], and message passing neural network (MPNN) [28], and find that GCN results in the highest accuracy for prediction targets. Thus, we choose GCN for *Driple* inspector. The detailed experiments are explained in §5.2.2.

Additionally, we design *Driple* inspector with stacking gated recurrent units (GRUs) at the end of ϕ_f and ϕ_r, respectively, as in Figure 7. So, layer 1 of *Driple* inspector is a pair of ϕ_f and GRU. The subsequent layers are of pairs of ϕ_r and GRU. This is to prevent the over-smoothing problem caused by the information loss in previous layers as the number of layers increases [22, 23, 28, 45].

After having $h_{n_i}^K$ values go through GRU, the *Driple* inspector generates a graph embedding (h_{G_l}) for G_l. The creation of h_{G_l} is achieved by the graph readout layer (ρ). ρ layer performs the pooling function over the $h_{n_i}^K$ values of n_i belong to N_l in G_l. Equation 3 expresses the ρ layer.

$$h_{G_l} = \rho = \texttt{POOL}(h_{n_i}^K | n_i \in N_l) \tag{3}$$

Similar to ϕ_f and ϕ_r, various ρ layers of choice exist. There are simple pooling functions for converting the given $h_{n_i}^K$ values into a h_{G_l} through mathematical operations (mean, max, or sum). Also, sort is another simple pooling function that picks up a certain number of $h_{n_i}^K$ values in the descending (or ascending) order and returns them as final h_{G_l}. These simple functions are used widely because they are straightforward and lead to fast training [64]. In addition, ρ layer can be a separate neural network. For example, set2set [62] uses LSTM neural network [32]. Set2set views the $h_{n_i}^K$ values as time-series data and weighs the importance between $h_{n_i}^K$ values to generate h_{G_l}. Among these choices, we choose set2set. This design is based on the empirical evaluation for the prediction accuracy where *Driple* inspector is tested with simple functions and set2set (§5.2.2).

Finally, *Driple* inspector hands the created h_{G_l} from the ρ layer over to the MLP. Regarding MLP, we also evaluate and choose MLP of three fully connected (fc) layers. MLP produces the prediction of 12 targets for given G_l.

3.3 Training of *Driple* Inspector

For the *Driple* inspector's training, a dataset is required. To our knowledge, no datasets on the computational resources of DT are available. In particular, we require output features as prediction targets, which are not available as far as we know. Thus, we develop tools for creating datasets.

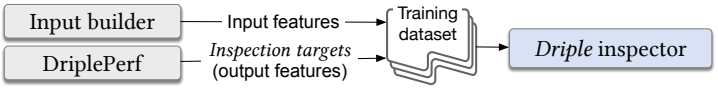


Fig. 9. Training workflow. See more details in §3.3.

The training dataset is generated per DT setting through the "input builder" and "DriplePerf." The input builder produces input features as explained in §3.1. DriplePerf creates the output features for the given DT code. As shown in Figure 9, the input features and prediction targets become the training dataset of a DT workload.

In conjunction with the input features from the input builder, DriplePerf creates the output features for a DT workload. The output features are the results of the DT training, so DriplePerf first measures the resource consumption (i.e., U_G, U_{GM}, T_{NT}, and T_{NR}) while executing the DT code. The measurement is formulated as data points of 2D coordinates—(measurement time, consumed resource amount). DriplePerf extracts the output features from the measurement results.

Similar to §2.4, DriplePerf runs k-means clustering algorithm for dividing the data points of the measurement data. The clustering algorithm categorizes each data point based on its y-axis value; so, the data point is categorized into two parts—burst and idle. Then, DriplePerf calculates the output features of the given graph as follows. First, the burst amount of each resource is calculated by the mean value of the consumed resource amount (y-axis) of burst data points. To calculate the burst duration and the idle duration, we count the consecutive burst and idle points appearing in each measurement. The period during which the consecutive burst points appear becomes a single burst duration. Also, the time wherein idle points appear is the idle duration. We formulate the normal distributions of burst amount, burst duration, and idle duration. From the distributions, we obtain the mean values as prediction targets.

Through the input builder and DriplePerf, a training dataset is generated for each DT workload in Table 2. Then, the *Driple* inspector is trained by multiple iterations against the dataset.

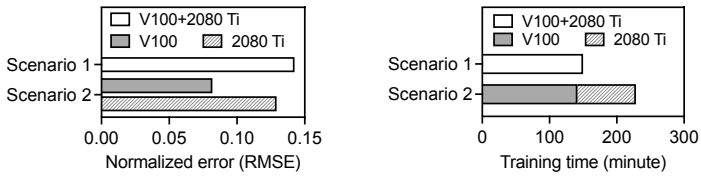
4 TRANSFER LEARNING OF *DRIPLE* INSPECTOR

Note that §3 suggests a *Driple* inspector to be trained for DT workloads given a DT setting. If the DT setting gets changed (such as the change of GPUs or addition of a worker), the *Driple* inspector has to be freshly trained again because the changed DT setting may significantly change resource consumption (§2.5). Considering that the number of possible DT settings is quite numerous, we explore whether transfer learning (TL) helps to cope with the variety of DT settings. In this section, we explain the necessity of TL in *Driple* (§4.1) and detailed designs for TL (§4.2).

4.1 Necessity of Transfer Learning

Before exploring TL, we investigate whether it is feasible to train a *Driple* inspector for multiple DT settings. In theory, a *Driple* inspector can be trained via a dataset of multiple DT settings. To test the theory, we profile resource consumption in two different DT settings with DriplePerf: 1) V100 GPU and 2) 2080 Ti GPU. The number of PSs and workers are identical for the two DT settings (one PS and two workers). Also, all PSs and workers are connected via PCIe.

Afterward, we evaluate two scenarios: scenario 1—a single inspector trained by a dataset composed of both V100 and 2080 Ti GPUs together (called V100+2080 Ti dataset), and scenario 2—two inspectors trained by a dataset of only V100 and a dataset of 2080 Ti, separately. In Figure 10a, the x-axis is the prediction error of the trained inspectors based on the average normalized error (root mean squared error) of the prediction targets. Figure 10a shows that the two inspectors for the scenario 2 show superior accuracy (170% and 110% in V100 and 2080 Ti, respectively) compared to



(a) prediction error. (b) training time.

Fig. 10. Comparison of inspectors based on the dataset composition. See more details in §4.1.

the inspector for the scenario 1. The results indicate that the two (or multiple) DT settings cannot be predicted accurately by a single inspector.

However, training a *Driple* inspector per DT setting requires considerable effort. When users may want to change DT settings according to the GPU type or number of GPUs, number of PS, number of workers, and network interconnect, generating a dataset for each DT setting would be burdensome. For example, in scenario 2 of Figure 10a, we find that more than 800 DT workloads are required to attain reasonable prediction accuracy just for the V100 model. So, it is too laborious to generate datasets for each DT setting.

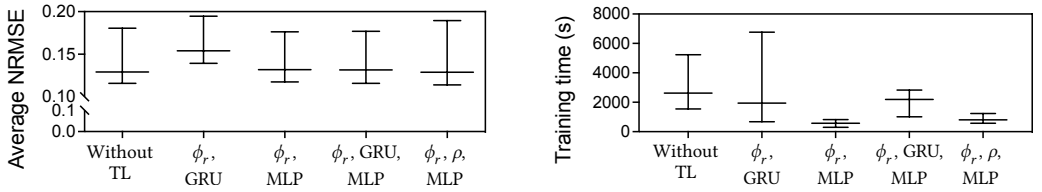
Second, the training time is a serious issue. Figure 10b shows the training time of the three inspectors for scenarios 1 and 2 above. The training time of the inspector for the scenario 1 is 149.8 minutes, while the scenario 2 requires 228.5 minutes in total, which is 52% longer than the scenario 1. However, a long training time can result in better prediction accuracy, as the V100 GPU and 2080 Ti GPU models in the scenario 2 (Figure 10a). This means that we need to find a trade-off to cope with the variety of DT settings. That is why we explore TL for *Driple*.

4.2 Transfer Learning Design for *Driple* Inspectors

TL is a training technique of deep learning models that leverages the knowledge of the pre-trained model to the target model [74]. To apply TL on *Driple* inspector training, we need to determine 1) the training dataset size and 2) which layers to reuse or how to update (fine-tune) parameters from the pre-trained inspector.

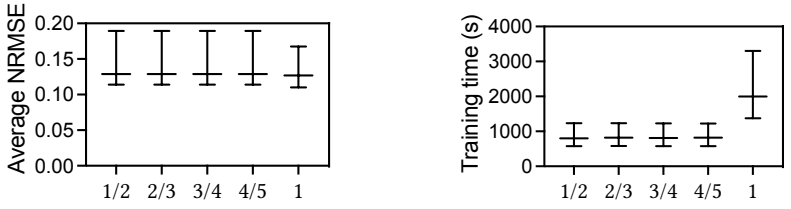
4.2.1 Training dataset size. TL is frequently used in ML models where the dataset is scarce. The size of the training dataset is empirically determined depending on where the TL is applied [16]. To determine the dataset size for *Driple*, we create datasets with various DT workloads (varying from 160 to 800 graphs). Afterward, we compare the training time and prediction accuracy between inspectors trained without TL and with TL. As the pre-trained model, *Driple* uses the inspector trained for V100 GPU with one PS and two workers connected by PCIe. We select it because it shows the highest prediction accuracy among many inspectors we trained.

We design TL from the pre-trained inspector to 13 different DT settings (to be explained in §5, Table 6). The training time and prediction accuracy differ per dataset size. We find that the training time of TL decreases as the training dataset size is smaller—on average, 17 minutes to 5 minutes for datasets of 640 and 160 DT workloads, respectively. So, the smaller the dataset, the better in terms of training time. For the prediction accuracy, we find that inspectors with TL maintain similar accuracy with those without TL when the training dataset size is at least 320. Thus, we set the number of DT workloads per training dataset as 320. Note that the dataset size required for training a *Driple* inspector without TL is 800 DT workloads (explained in §4.1, V100 model of scenario 2). Compared to that, the dataset of 320 DT workloads is the 2.5× improvement in the dataset size required for the pre-trained inspector.



(a) prediction error. (b) training time.

Fig. 11. Comparison by fine-tuning layer combinations. See more details in §4.2.2.



(a) prediction error. (b) training time.

Fig. 12. Comparison by ϕ_r partitioning ratios. See more details in §4.2.3.

4.2.2 Fine-tuning layers. TL tries to reuse the parameters of some layers in the pre-trained inspector. Simultaneously, other layers' parameters are newly updated against a new DT setting (new dataset). This update of the parameters is what we call fine-tuning.

To select which layers of a *Driple* inspector to fine-tune, we consider the prediction accuracy and training time. We experiment fine-tuning of TL from the pre-trained inspector (V100 GPU) to a *Driple* inspector for a new DT Setting—2080 Ti GPUs with one PS and two workers connected by PCIe. In the experiment, we vary the layers for fine-tuning.

Figure 11a shows the prediction accuracy over varying layers (the x-axis of Figure 11). For example, "ϕ_r, MLP" indicates that ϕ_r layers and MLP layers (explained in §3.2) are fine-tuned. When ϕ_r layers' fine-tuning is tested, the latter half layers ($K/2$ to K layers) are used for fine-tuning (to be explained in §4.2.3). For the prediction accuracy, we use normalized root mean squared errors (NRMSEs) in which errors of 12 prediction targets gained from each input graph are averaged. Figure 11a shows that "ϕ_r, GRU" has lower average accuracy over others—14% lower than others. Figure 11b shows the training time where "ϕ_r, MLP" has the shortest training time by 82% over without TL. So, we choose ϕ_r layers ($K/2$ to K layers) and MLP layers for fine-tuning.

4.2.3 ϕ_r partitioning. For TL, we devise ϕ_r partitioning to improve the training speed of *Driple* inspectors further. In deep learning, the parameters (e.g., weight and bias) of each layer have different values. For example, in CNN, multiple convolution layers for image processing exist, but their parameters in each layer are different. Because the convolution layers of CNN have different parameter values, it is easy to fine-tune parameters or layers selectively. However, in the case of the *Driple* inspector, weights and biases are identical across K-1 numbers of ϕ_r layers (as described in §3.2). So, unlike CNNs that can selectively fine-tune their convolution layers, the *Driple* inspector has no choice but to fine-tune all ϕ_r layers. This problem increases the training time significantly.

To reduce TL training time, ϕ_r partitioning partitions ϕ_r layers (layer 2 to K in Figure 7) into two types: 1) some ϕ_r layers having parameters from the pre-trained inspector (denoted as ϕ_r^s layers)

Table 6. Experiment cases. See more details in §5.1.

Name	GPU	DP topology	Network	# of GPU machines	Name	GPU	DP topology	Network	# of GPU machines
V100-P1w2/ho-PCIe	V100	PS1/w2/ homo	Co-located	1	2080Ti-P4w4/he-40G	2080Ti	PS4/w4/ hetero	40 GbE	2
V100-P2w2/ho-PCIe	V100	PS2/w2/ homo	Co-located	1	TitanRTX-P4w4/he-40G	Titan RTX			
2080Ti-P1w2/ho-PCIe	2080Ti	PS1/w2/ homo	Co-located	1	V100-P5w5/he-1G	V100	PS5/w5/ hetero	1 GbE	5
2080Ti-P1w3/ho-PCIe	2080Ti	PS1/w3/ homo	Co-located	1	2080Ti-P5w5/he-1G	2080Ti			
2080Ti-P2w2/he-PCIe	2080Ti	PS2/w2/ hetero	Co-located	1	V100-P5w10/he-1G	V100	PS5/w10/ hetero	1 GbE	5
TitanRTX-P2w2/he-PCIe	Titan RTX				2080Ti-P5w10/he-1G	2080Ti			
2080Ti-P2w2/he-40G	2080Ti	PS2/w2/ hetero	40 GbE	2					
TitanRTX-P2w2/he-40G	Titan RTX								

and 2) the remaining ϕ_r layers with the parameters to be fine-tuned on a target DT setting (denoted as ϕ_r^{FT} layers).

To determine the ratio between ϕ_r^s and ϕ_r^{FT} layers, we conduct the following experiments. In the experiments, we test the partitioning by changing the ratio between ϕ_r^s and ϕ_r^{FT}, and the prediction accuracy and training time are measured. The ratio 3/4 means that 3/4 of the ϕ_r layers are partitioned to be ϕ_r^{FT}. Figure 12 shows the experiment results where the x-axis is the ratio of ϕ_r^{FT}. In terms of prediction accuracy (Figure 12a), the median of average NRMSE for the ratio 1 is 2% lower than others. Figure 12b shows that the training time of the ratio 1/2 is the shortest—2.5× shorter than the ratio 1. Figures 12a and 12b give conflicting results, so we deliberately weigh the importance of the improvement of the training time over the loss in the prediction accuracy since the difference of the accuracy is small (i.e., 2%). Thus, we set the ratio between ϕ_r^s and ϕ_r^{FT} as 1/2.

5 EVALUATION

We implement the input builder and DriplePerf that works with TensorFlow.[6] We also implement *Driple* inspector and TL with PyTorch, leveraging open-source implementation of [23]. We release the core implementation of *Driple* at [9]. Based on the implementation, we conduct extensive experiments to evaluate the effectiveness of *Driple*.

5.1 Evaluation Settings

For experiments, we utilize six servers—one server of four V100 GPUs, three servers of three 2080 Ti GPUs and one Titan RTX GPU, and two servers of two V100 GPUs. One server of V100 GPUs is equipped with Intel Xeon Skylake processor (16 cores). The other servers are equipped with two Intel Xeon Silver 4210 processors (40 cores). All GPUs are connected with CPU by PCIe. Except for the server with four V100 GPUs, all servers are connected via 1 GbE and 40 GbE (through Intel X550T and NVIDIA ConnectX-5 NIC, each). We train the *Driple* inspectors for 14 different DT settings shown in Table 6, and after training, we have one inspector per DT setting. The columns "Name" of the table present the abbreviated name of the DT setting of the corresponding row. Also, the columns "# of GPU machines" show the number of GPU machines used for each DT setting. Except for the V100-P1w2/ho-PCIe (pre-trained inspector), all *Driple* inspectors are trained via TL.

[6]Note that we do not modify TensorFlow itself. Instead, we use the existing APIs to implement the input builder in obtaining computational graphs (e.g., `write_graph` [12]). Also, DriplePerf measures the output features outside of TensorFlow using NVML [7] and libpcap [8].

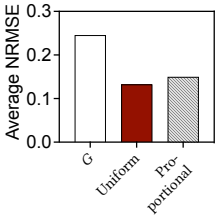
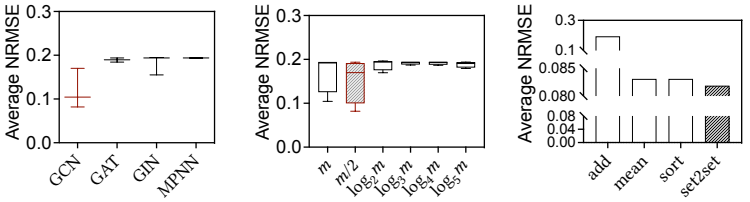


Fig. 13. Grouping policies comparison. See more details in §5.2.1.



(a) GNN algorithms. (b) number of layers. (c) ρ.

Fig. 14. Design choice of *Dripple* inspector structure. See more details in §5.2.2.

5.2 Experiments for the Design Choices of *Dripple*

Dripple has several design choices (e.g., the number of layers in *Dripple* inspector). We evaluate the choices through pertinent experiments to choose the proper ones. We explain experiments for design choices of input builder and *Dripple* inspector in the following subsections. As an evaluation metric, we first calculate the root mean squared error for each of the 12 prediction targets. Then, we normalize each root mean squared error (NRMSE) and average them, which is the average NRMSE. We use the average NRMSE as a metric for prediction accuracy because the units of the prediction targets are different (i.e., %, MB/s, and second).

5.2.1 Input builder. We evaluate the effect of node grouping. We use the V100-P1w2/ho-PCIe setting for the experiments. For the uniform grouping, we empirically set M as 100. For the proportional grouping, we set the number of graphs belonging to each batch as 32. When the average number of nodes of 32 graphs is v, we set M as $log_{10}v$. This decision is made empirically so that the largest graph of the dataset can be loaded into the V100 GPU memory after grouping.

Figure 13 shows the results of node grouping evaluation in terms of prediction accuracy. Without applying TL, we train *Dripple* inspectors with the three datasets: 1) without grouping policy (G), 2) uniform grouping, and 3) proportional grouping. Compared to the dataset without the grouping policy, proportional grouping shows 46% lower NRMSE on average. Also, when comparing proportional and uniform grouping, the uniform grouping results in 12% lower NRMSE.

This is because, without grouping, training can only be performed with a batch size of 1 (training one graph at an iteration). Without grouping, the number of graph layers (K) varies according to the input graph for the training (e.g., $m/2$), but each graph has a different m.

In other words, each graph requires the different K numbers of layers for its training. Thus, the training also should be performed based on a single graph (batch size of 1). However, it is known that training with batch size 1 takes the convergence of model parameters too long (overshooting problem) [57]. Grouping, in contrast, makes the batch size higher than 1, which leads to the 46% accuracy improvement in Figure 13. According to the evaluation results, *Dripple* chooses uniform grouping for the input builder.

5.2.2 Dripple inspector structure. *Dripple* inspector has a couple of design choices when designing its structure (Figure 14)[7]. The first is what GNN algorithm *Dripple* inspector uses. We compare four GNN algorithms: GCN, GAT, GIN, and MPNN. The size of the embedding and ρ are fixed as 64 and set2set, respectively. Figure 14a shows the distribution of the average NRMSE of the GNN algorithms. What stands out in Figure 14a is that GCN shows the lowest prediction error. In

[7]Here, we present the design choices for the GNN part. The design of the later part of the *Dripple* inspector is explained at Appendix (§A).

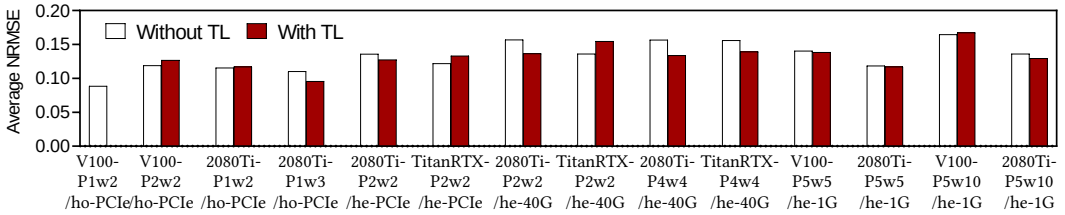


Fig. 15. Prediction accuracy comparison for TL. See more details in §5.3.

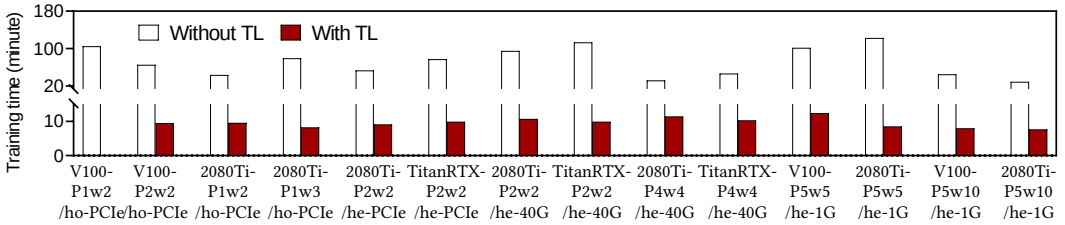


Fig. 16. Training time comparison for TL. See more details in §5.3.

comparison to others, its median prediction error is 45% lower. We also evaluate GNN algorithms with other DT settings (e.g., 2080Ti-P1w2/ho-PCIe), and the results exhibit similar tendencies overall. So, we choose GCN.

The second design choice is the number of layers in *Driple* inspector, K. We train *Driple* inspectors with the K of m, $m/2$, $log_2 m$, $log_3 m$, $log_4 m$, and $log_5 m$. The size of the embedding is fixed as 64. Figure 14b presents the average NRMSE according to K. As shown in Figure 14b, the average NRMSE is lowest when K is $m/2$, and it is 1.1× to 1.2× better than m and the average of log-based m values, respectively. So, we choose $m/2$.

The third design choice is the readout layer, ρ. We train *Driple* inspectors by changing ρ to three simple methods (i.e., add, mean, and sort) and set2set. For training, GNN, K, and the embedding size are set to GCN, $m/2$, and 64. As shown in Figure 14c, set2set shows the lowest errors—57.8%, 1.5%, and 1.5% lower than the simple methods. Thus, we choose set2set for ρ.

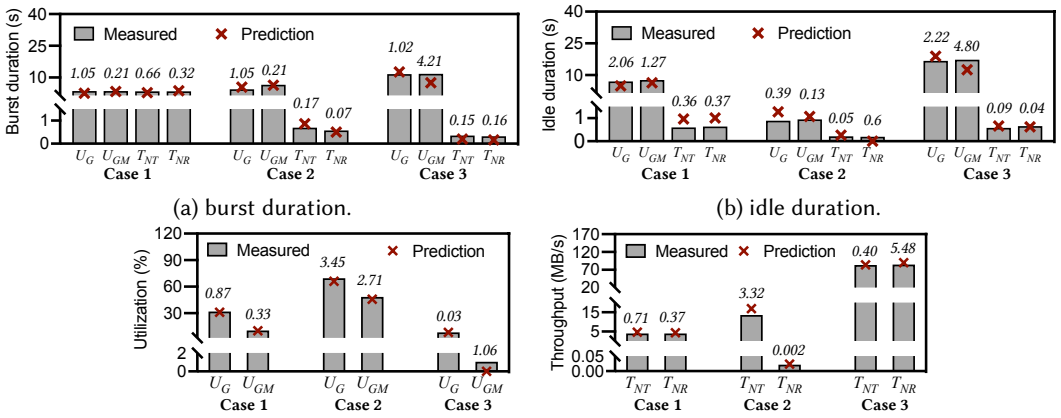
The fourth and last design choice for *Driple* inspector is the embedding size. We train *Driple* inspectors with three embedding sizes of 16, 32, 64 with GCN, $m/2$, and set2set. The NRMSE of 64 is the lowest—8% and 1% lower than for 16 and 32 sizes. So, we choose 64 as the embedding size.

5.3 TL Effectiveness

After the design choices are determined, we evaluate the effectiveness of training through TL by comparing *Driple* inspector trained "without TL" and "with TL." For experiments, we use the *Driple* inspector trained by V100-P1w2/ho-PCIe dataset as the pre-trained inspector. The reason is that it shows the highest prediction accuracy between the *Driple* inspectors trained without TL. We apply the TL techniques proposed in §4.2 to the pre-trained inspector to obtain 13 inspectors for 13 DT settings in Table 6.[8] For comparison, we separately and individually train 13 inspectors for 13 DT settings. We make the training stop when the prediction on the validation set does not improve for 1000 iterations, which means the convergence of the model parameters.

Figure 15 is the comparison of inspectors trained without TL and with TL. We use prediction accuracy (average NRMSE) as a metric. The first bar in Figure 15 is for the pre-trained inspector, so

[8]The first DT setting in Table 6 is used for a pre-trained inspector. So, we apply TL for the remaining 13 settings.



(a) burst duration. (b) idle duration.

(c) burst amount of U_G and U_{GM}. (d) burst amount of T_{NT} and T_{NR}.

Fig. 17. Prediction accuracy. See more details in §5.4.

it does not have the results for "with TL." Figure 15 shows that the prediction accuracy of inspectors with TL is similar to that of inspectors without TL. Although some inspectors with TL have better accuracy and others have worse than the inspectors without TL, the prediction accuracy is within a certain range, which we believe validates that TL is meaningful and applicable to training *Driple* inspectors for new DT settings. Regarding the question of how the accuracy of TL is better (e.g., 2080Ti-P4w4/he-40G in Figure 15), machine learning literature has reported that TL can achieve higher prediction accuracy because TL utilizes the pre-knowledge jointly with the knowledge of the training domain [68, 70]. So, Figure 15 can be interpreted in that the inspector with TL uses its own dataset (of a smaller scale) with the pre-trained inspector, which leads to better accuracy.[9]

Second, we also measure the training time of *Driple* inspectors. Figure 16 shows that the training time is much reduced by 7.3× on average. The detailed analysis finds that the improvement of training time comes from two techniques: 1) fine-tuning of selective layers (i.e., ϕ_r and MLP) and 2) ϕ_r partitioning. Due to the space limit, we omit the detailed analysis. In summary, Figures 15 and 16 indicate that *Driple* can reduce the training time significantly while maintaining the prediction accuracy close to the inspectors without TL.

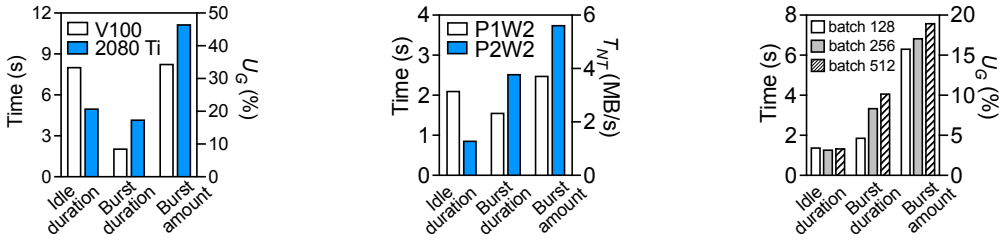
5.4 Prediction of Resource Consumption

The goal of this study is to predict the resource consumption of the variety of workloads and settings. We make prediction cases by randomly selecting workloads and settings. Then, we present the resource consumption of three cases to demonstrate the prediction power of *Driple*. All *Driple* inspectors except for the one used as the pre-trained inspector are trained through TL. As for accuracy, the measurement (ground-truth) and the predicted values for targets (burst duration, idle duration, and burst amounts) are compared. Three cases are as follows:

- **Case 1**: ResNet44 model, CIFAR-10 dataset, V100-P2w2/ho-PCIe (asynchronous training)
- **Case 2**: GoogLeNet model, ImageNet dataset, TitanRTX-P2w2/he-40G (synchronous training)
- **Case 3**: Transformer-AAN, Europarl dataset, V100-P2w2/ho-PCIe (asynchronous training)

Figure 17 presents the prediction results of three cases. Figures 17a and 17b present the burst and idle durations, respectively. Figure 17c shows the burst amount of U_G and U_{GM}. Also, Figure 17d presents the burst amount of T_{NT} and T_{NR}. The bars in the graphs are the measured values, and

[9]We analyze the prediction accuracy in terms of DT setting changes in Appendix (§B).



(a) GPU change. (b) PSs and workers change. (c) batch size change.

Fig. 18. *Driple* applications examples. See more details in §5.5.

the x-marks are the predicted values of *Driple* inspectors. Also, the numbers above bars represent mean absolute errors (MAEs) between the measured and prediction values.

For the burst duration (Figure 17a), predicted values are pretty similar to the measured values. Across the cases, the average MAEs of idle duration, burst duration, and burst amount are 1 s, 0.8 s, and 1.56%, respectively. To put the MAE numbers in perspective, we calculate the percentage errors (symmetric mean absolute percentage errors) [29]. On average, the percentage errors are 11%, 9%, 17%, and 15%, for U_G, U_{GM}, T_{NT}, and T_{NR}, respectively. The results indicate that *Driple* inspectors achieve reasonably good prediction accuracies, considering their training speed advantages.

5.5 *Driple* Applications

We present some applications wherein *Driple* benefits DT engineers. The benefits are described when the changes take place between 1) different GPUs, 2) different numbers of PSs and workers, and 3) batch size (hyperparameter). The first two are related to the DT setting changes, and the third is of the DT workload changes. We show that *Driple* can provide a few insights, including the total training time. We use *Driple* inspectors trained in §5.3 for prediction.

5.5.1 Effect of GPU changes. We experiment with a DT workload of ResNet56 model trained with CIFAR-10 dataset (asynchronous training). To see changes coming from different GPUs, we use two inspectors of different DT settings for the prediction, V100-P1w2/ho-PCIe and 2080Ti-P1w2/ho-PCIe. Figure 18a shows that the burst amount of U_G is higher in 2080 Ti than V100 (1.3×). In terms of the durations, the idle duration decreases by up to 49% when GPU is changed from V100 to 2080 Ti. At the same time, the burst duration increases by up to 1.6×. The ratio between the idle and burst durations are 3.9:1 and 1.2:1 in V100 and 2080 Ti GPUs, respectively. Considering these results, if the DT developer desires to utilize the GPU efficiently, 2080 Ti would be a better choice for this workload.

5.5.2 Effect of PS and worker changes. Figure 18b shows the prediction targets of T_{NT} when the number of PSs changes. The DT workload used in the experiments is VGG19 model trained with ImageNet dataset (asynchronous training). We predict the targets in two DT settings: V100-P1w2/ho-PCIe and V100-P2w2/ho-PCIe. The results show, when the number of PSs is increased from one to two, the burst amount of T_{NT} increases by 1.5×. The burst duration also increases (1.6×) while the idle duration decreases (0.4×). The increased and decreased rate of the burst and idle durations are similar, so the time per iteration is similar in both cases because the sum of burst and idle durations becomes the time per iteration. The results indicate that two PSs cause additional communication time between workers and PS, but the total training time remains similar. So, in this case, selecting one PS is better for the efficient training.

Table 7. Related work comparison. See more details in §6.

	Paleo [54]	Justus et al. [36]	Daydream [73]	Habitat [27]	*Driple*
Inspection metric	Computation time per iteration, communication time per iteration	Training time per batch	Training time per iteration	Training time per iteration	Burst amount, burst duration, idle duration of four resources.
Inspection method	Mathematical modeling	Neural network	Simulation on graph	MLP	GNN, TL
Inspection (input) scope	Two kinds of convolution layer (matrix multiplication and fast Fourier Transform)	Convolution, fc	Optimization on models (e.g., FusedAdam optimizer)	2D-convolution, LSTM matrix multiplication, linear	Any operations expressed by graph
Distributed training	No detailed models	No	All-reduce	No	Yes (data parallel, PS)
ML library	TensorFlow	TensorFlow	PyTorch	PyTorch	TensorFlow

5.5.3 Effect of batch size changes. Figure 18c presents the prediction targets of U_G when the batch size changes. The DT workload used is Overfeat model trained by ImageNet dataset (synchronous training). We change the batch size to 128, 256, and 512, while the DT setting is identical—2080Ti-P2w2/he-40G. The results show that the burst duration and burst amount increase by up to 2.2× and 1.2×, respectively, as the batch size increases. Conversely, the idle duration decreases up to 4%.

The results indicate the following: First, in terms of burst amount of U_G, it increases slightly (from 15.8% to 19%) as the batch size quadruples. Thus, large batch sizes like 512 can be handled well by 2080 Ti. Second, the time of an iteration (sum of burst and idle durations) increases with the batch size—1.7× increase (from 3.3 s for a batch size of 128 to 5.5 s for a batch size of 512). When the batch size increases from 128 to 512, each mini-batch has 4× more data; thus, the amount of input processed in a single iteration also increases by 4×. Although the data processed per iteration grows 4×, the time per iteration increases only 1.7×. If the epoch, the number of times the entire dataset is trained, is identical, the larger batch size (i.e., 512) can reduce the total training time.

6 RELATED WORK

Deep learning resource estimation. Several studies focused on estimating (or predicting) the resource of training. Table 7 summarizes the existing studies and compares them with *Driple*. First, existing studies focused on the training time (duration) of an iteration. So, they did not estimate the training time for burst or idle durations of individual resources. Paleo [54] provided communication time per iteration but yet focused on an iteration-level metric. Second, some studies used mathematical modeling, simulation, or machine learning (e.g., neural networks or MLPs) for prediction. However, such methods took fixed-size numerical values as inputs. As a result, their prediction scope is quite limited. For example, papers [27, 36, 54] estimated the training time of particular neural network layers like convolution or fc. Daydream [73] also performed simulations to estimate training time, but the focus was on the improvement of training time from model optimization techniques. On the other hand, *Driple* takes a whole graph (any DT code) as its input, so it can capture any operations described by the ML library.

DT system research. DT motivated a line of literature that focused on improving the utilization and training speed in GPU clusters. First, some studies addressed workload scheduling to improve U_G. They designed solutions based on cyclic U_{GM} patterns [65], past training traces [47], predictions on loss function convergence [52], or estimation models on system-wide goodput [55]. However, these studies are not based on per-workload and per-DT setting that could further enhance performance.

Second, several studies enhanced the communication of the DT because it can easily become a bottleneck for the entire training. For example, ByteScheduler [53] designed a packet scheduling

policy based on the dependency between computation and communication in DT. Several papers [37, 38, 44] suggested congestion avoidance techniques or in-network parameter processing based on the cyclic communication patterns (T_{NT} and T_{NR}). However, these studies did not use prediction in their techniques.

7 DISCUSSION

***Driple* on other DT strategies.** This study has shown the effectiveness of *Driple* through data parallel with PS because it is the most widely-used strategy. However, there are other DT strategies, such as all-reduce based data parallel [6] and model parallel [18]. Because *Driple* is based on GNNs and TL, which have generalizability [17, 49, 56] in its training, we believe *Driple* could be applied to other DT strategies as well. Nevertheless, at this point in time, we leave the application of *Driple* to other DT strategies for future work.

***Driple* on other deep learning libraries.** This study validates *Driple* with TensorFlow. *Driple* uses the graph extraction APIs from TensorFlow without any modifications. There is a question about whether *Driple* can work with other libraries such as PyTorch, Caffe, and MXNet. Note that these libraries also convert the DT codes to graphs and provide similar APIs to obtain the graphs (e.g., `torchsummary` for PyTorch). Thus, we believe *Driple* can work with other libraries. Moreover, there is an effort towards interoperability between the libraries, called ONNX [11], which provides a unified format for storing models and their graphs. By using ONNX, *Driple* should be able to work with various libraries.

Other resources of PSs and workers. This study concentrates on U_G, U_{GM}, T_{NT}, and T_{NR} of workers. However, there are other resources such as CPU utilization of workers or network throughput of PSs. We measure U_G, U_{GM}, T_{NT}, and T_{NR} of workers through DriplePerf. DriplePerf measures the metrics outside of the deep learning libraries by utilizing resource profilers, such as NVML and libpcap. So, DriplePerf can be extended to measure the other resources. In addition, the *Driple* inspector can predict other resources by adding them as prediction targets after the MLP (Figure 7), which is not complex at all. Thus, we believe that *Driple* can be extended to the other resources without much difficulty.

Selection of node features. For selecting node features of *Driple* inspector, we have surveyed (Table 5, §3.1) the features used in previous studies that used GNN. Except for the chosen three features of *Driple*, we exclude the other features used in previous studies due to the following reasons. First, we exclude previous nodes (the category of the ancestor node) and adjacent node IDs [72] because the adjacency matrix can reflect them. The two features are related to previous nodes, so the GNN model can indirectly know via the adjacency matrix and node type feature. Second, we exclude the node placement feature [14]. The node placement feature is the ID of the device where each node is placed. However, the *Driple* inspector is the prediction model of each worker, so all nodes belong to the identical device. So, *Driple* does not require the node placement feature.

Time of dataset generation. In this study, we reduce the required amount of dataset up to 2.5× in training a *Driple* inspector by using TL (§4.2.1). Accordingly, the time for dataset generation also reduces. Another possible method to further reduce the dataset generation time is finding a good number of iterations per DT workload. In this study, each dataset is generated by running each DT workload for 100 iterations and calculating the average of output features per iteration. However, we observe that the output features are quite regular per iteration, which means that the output features can be created with fewer iterations. We leave the validation of the relationship between the time reduction in dataset generation and prediction accuracy for future work.

Design choice reasoning. In this study, we present various experiments that evaluate design choices of *Driple* (§5.2). By changing the grouping policies, GNN algorithms, number of layers, and ρ, we choose the suitable designs for the *Driple* inspector. However, one might be curious why the

specific value results in higher accuracy than others. In machine learning, the design choices of prediction models typically correspond to an empirical trial-and-error process. Finding the reason for hyperparameter selection is actually the subject of other fields of machine learning, "explainable AI," which is used for understanding the effect of each design choice and model parameter in prediction [48, 63]. Thus, we believe this is beyond the scope of this study, and in the future, we plan to research the semantic interpretations of design choices applied in *Driple*.

Boundary of TL. We use TL for training *Driple* inspectors for various DT settings and show that TL improves the training times with a small size of datasets. Two possible research issues exist with regard to TL. The first is whether the accuracy and training time will sustain if the pre-trained inspector (i.e., V100-P1w2/ho-PCIe) is changed to another inspector. The second is whether the effect of TL will be consistent in the other DT settings not shown in Table 6.

In our design, the input and output features are similar between the pre-trained inspector and a new inspector to be trained. This kind of TL is known as homogeneous transfer that shows relatively stable performance and low complexity in training, in comparison to the TL cases that have entirely different features between the pre-trained and new inspectors [74]. Therefore, we think that the improvements through TL can sustain with another pre-trained inspector if the pre-trained inspector shows a high enough accuracy. We leave the validation of this assumption as our future work.

8 CONCLUSION

This study proposes "*Driple*," a resource prediction technique for a variety of DT workloads and DT settings. *Driple* creates a *Driple* inspector per DT setting that predicts 12 targets of the resource consumption of workloads. By leveraging GNN, *Driple* takes the DT code itself as input, and the prediction scope is not limited to specific workloads. Furthermore, *Driple* applies TL to training in order to create *Driple* inspectors for multiple DT settings effectively.

The evaluation results show that *Driple* predicts the targets of resource consumption accurately. Also, *Driple* improves training time by 7.3× while reducing the training dataset size by 2.5×. In addition, we show example application scenarios in which DT developers can see the effect of the batch size change or GPU change a priori.

ACKNOWLEDGMENTS

We thank our shepherd, Sergey Blagodurov, and the anonymous reviewers for their insightful comments that helped us to improve this study. This work was supported by Institute of Information & communications Technology Planning & Evaluation grant funded by the Korea government (Ministry of Science and ICT, MSIT) (2015-0-00280, (SW Starlab) Next generation cloud infra-software toward the guarantee of performance and security SLA). This research was also partly supported by Basic Science Research Program through the National Research Foundation of Korea funded by the Ministry of Education (NRF-2021R1A6A1A13044830), and a Korea University Grant.

REFERENCES

[1] 2020. NVIDIA Titan RTX is here. https://www.nvidia.com/en-us/deep-learning-ai/products/titan-rtx/ Accessed: 2022-01-02.
[2] 2020. NVIDIA V100 | NVIDIA. https://www.nvidia.com/en-us/data-center/v100/. Accessed: 2021-12-09).
[3] 2021. Benchmarks/scripts/tf_cnn_benchmarks · TENSORFLOW/benchmarks. https://github.com/tensorflow/benchmarks/tree/master/scripts/tf_cnn_benchmarks Accessed: 2021-09-25.
[4] 2021. Graphics reinvented: NVIDIA GeForce RTX 2080 Ti graphics card. https://www.nvidia.com/en-us/geforce/graphics-cards/rtx-2080-ti/ Accessed: 2021-12-07.
[5] 2021. NVIDIA A100 GPUs. https://www.nvidia.com/en-us/data-center/a100/ Accessed: 2021-12-21.
[6] 2021. NVIDIA Collective Communications Library (NCCL). https://developer.nvidia.com/nccl Accessed: 2022-01-03.

[7] 2021. NVML API Reference Guide :: GPU Deployment and Management Documentation. https://docs.nvidia.com/deploy/nvml-api/structnvmlUtilization__t.html#structnvmlUtilization__t Accessed: 2021-09-24.
[8] 2021. TCPDUMP & LIBPCAP. https://www.tcpdump.org/. Accessed: 2021-09-24.
[9] 2022. Driple. https://github.com/gsyang33/Driple. Accessed: 2022-04-07.
[10] 2022. A library of sklearn compatible categorical variable encoders. https://github.com/scikit-learn-contrib/category_encoders Accessed: 2022-01-13.
[11] 2022. ONNX. https://onnx.ai/. Accessed: 2022-04-06.
[12] 2022. tf.io.write_graph | TensorFlow Core. https://www.tensorflow.org/api_docs/python/tf/io/write_graph. Accessed: 2022-04-06.
[13] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: A system for large-scale machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.
[14] Ravichandra Addanki, Shaileshh Bojja Venkatakrishnan, Shreyan Gupta, Hongzi Mao, and Mohammad Alizadeh. 2019. Placeto: Learning generalizable device placement algorithms for distributed machine learning. *arXiv preprint arXiv:1906.08879* (2019).
[15] Dmitrii Babaev, Maxim Savchenko, Alexander Tuzhilin, and Dmitrii Umerenkov. 2019. E.T.-RNN: Applying deep learning to credit loan applications. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 2183–2190.
[16] Jayme Garcia Arnal Barbedo. 2018. Impact of dataset size and variety on the effectiveness of deep learning and transfer learning for plant disease classification. *Computers and electronics in agriculture* 153 (2018), 46–53.
[17] Peter W Battaglia, Jessica B Hamrick, Victor Bapst, Alvaro Sanchez-Gonzalez, Vinicius Zambaldi, Mateusz Malinowski, Andrea Tacchetti, David Raposo, Adam Santoro, Ryan Faulkner, et al. 2018. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261* (2018).
[18] Tal Ben-Nun and Torsten Hoefler. 2019. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)* 52, 4 (2019), 1–43.
[19] Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165* (2020).
[20] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. MXNet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).
[21] Yangrui Chen, Yanghua Peng, Yixin Bao, Chuan Wu, Yibo Zhu, and Chuanxiong Guo. 2020. Elastic parameter server load distribution in deep learning clusters. In *Proceedings of the 11th ACM Symposium on Cloud Computing*. 507–521.
[22] Kyunghyun Cho, Bart van Merrienboer, Çaglar Gülçehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. 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 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*, Alessandro Moschitti, Bo Pang, and Walter Daelemans (Eds.). ACL, 1724–1734.
[23] Gabriele Corso, Luca Cavalleri, Dominique Beaini, Pietro Liò, and Petar Veličković. 2020. Principal Neighbourhood Aggregation for Graph Nets. In *Advances in Neural Information Processing Systems*, Vol. 33. 13260–13271.
[24] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 248–255.
[25] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).
[26] Xiao Ding, Yue Zhang, Ting Liu, and Junwen Duan. 2015. Deep Learning for Event-Driven Stock Prediction. In *Proceedings of the 24th International Conference on Artificial Intelligence* (Buenos Aires, Argentina) *(IJCAI'15)*. AAAI Press, 2327–2333.
[27] X Yu Geoffrey, Yubo Gao, Pavel Golikov, and Gennady Pekhimenko. 2021. Habitat: A Runtime-Based Computational Performance Predictor for Deep Neural Network Training. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. 503–521.
[28] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. 2017. Neural message passing for quantum chemistry. In *International conference on machine learning*. PMLR, 1263–1272.
[29] Paul Goodwin and Richard Lawton. 1999. On the asymmetry of the symmetric MAPE. *International Journal of Forecasting* 15, 4 (1999), 405–408.
[30] David Harwath, Antonio Torralba, and James Glass. 2016. Unsupervised Learning of Spoken Language with Visual Context. In *Advances in Neural Information Processing Systems*, Vol. 29. Curran Associates, Inc.

[31] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 770–778.

[32] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[33] Jeremy Howard and Sebastian Ruder. 2018. Universal language model fine-tuning for text classification. *arXiv preprint arXiv:1801.06146* (2018).

[34] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. 2017. Densely Connected Convolutional Networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*.

[35] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional architecture for fast feature embedding. In *Proceedings of the 22nd ACM international conference on Multimedia*. 675–678.

[36] Daniel Justus, John Brennan, Stephen Bonner, and Andrew Stephen McGough. 2018. Predicting the computational cost of deep learning models. In *2018 IEEE international conference on big data (Big Data)*. IEEE, 3873–3882.

[37] Minkoo Kang, Gyeongsik Yang, Yeonho Yoo, and Chuck Yoo. 2020. TensorExpress: In-Network Communication Scheduling for Distributed Deep Learning. In *2020 IEEE 13th International Conference on Cloud Computing*. 25–27.

[38] Minkoo Kang, Gyeongsik Yang, Yeonho Yoo, and Chuck Yoo. 2021. Proactive Congestion Avoidance for Distributed Deep Learning. *Sensors* 21, 1 (2021), 174.

[39] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[40] Guillaume Klein, Yoon Kim, Yuntian Deng, Jean Senellart, and Alexander Rush. 2017. OpenNMT: Open-Source Toolkit for Neural Machine Translation. In *Proceedings of ACL 2017, System Demonstrations*. Association for Computational Linguistics, Vancouver, Canada, 67–72.

[41] Philipp Koehn. 2005. Europarl: A Parallel Corpus for Statistical Machine Translation. In *Proceedings of Machine Translation Summit X: Papers*. Phuket, Thailand, 79–86.

[42] Alex Krizhevsky. 2009. *Learning multiple layers of features from tiny images*. Technical Report.

[43] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet classification with deep convolutional neural networks. *Advances in neural information processing systems* 25 (2012), 1097–1105.

[44] ChonLam Lao, Yanfang Le, Kshiteej Mahajan, Yixi Chen, Wenfei Wu, Aditya Akella, and Michael Swift. 2021. ATP: In-network Aggregation for Multi-tenant Learning. In *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*. USENIX Association, 741–761.

[45] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2015. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493* (2015).

[46] Zheyu Lin, Xukun Chen, Hanyu Zhao, Yunteng Luan, Zhi Yang, and Yafei Dai. 2020. A Topology-Aware Performance Prediction Model for Distributed Deep Learning on GPU Clusters. In *2020 IEEE International Conference on Big Data (Big Data)*. IEEE, 2795–2801.

[47] Kshiteej Mahajan, Arjun Balasubramanian, Arjun Singhvi, Shivaram Venkataraman, Aditya Akella, Amar Phanishayee, and Shuchi Chawla. 2020. Themis: Fair and efficient GPU cluster scheduling. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. 289–304.

[48] Julia Moosbauer, Julia Herbinger, Giuseppe Casalicchio, Marius Lindauer, and Bernd Bischl. 2021. Explaining Hyperparameter Optimization via Partial Dependence Plots. In *Advances in Neural Information Processing Systems*, M. Ranzato, A. Beygelzimer, Y. Dauphin, P.S. Liang, and J. Wortman Vaughan (Eds.), Vol. 34. Curran Associates, Inc., 2280–2291.

[49] Aditya Paliwal, Felix Gimeno, Vinod Nair, Yujia Li, Miles Lubin, Pushmeet Kohli, and Oriol Vinyals. 2019. REGAL: Transfer learning for fast optimization of computation graphs. *arXiv preprint arXiv:1905.02494* (2019).

[50] Ferran Parés, Dario Garcia Gasulla, Armand Vilalta, Jonatan Moreno, Eduard Ayguadé, Jesús Labarta, Ulises Cortés, and Toyotaro Suzumura. 2018. Fluid Communities: A Competitive, Scalable and Diverse Community Detection Algorithm. In *Complex Networks & Their Applications VI*, Chantal Cherifi, Hocine Cherifi, Márton Karsai, and Mirco Musolesi (Eds.). Springer International Publishing, Cham, 229–240.

[51] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019), 8026–8037.

[52] Yanghua Peng, Yixin Bao, Yangrui Chen, Chuan Wu, and Chuanxiong Guo. 2018. Optimus: an efficient dynamic resource scheduler for deep learning clusters. In *Proceedings of the Thirteenth EuroSys Conference*. 1–14.

[53] Yanghua Peng, Yibo Zhu, Yangrui Chen, Yixin Bao, Bairen Yi, Chang Lan, Chuan Wu, and Chuanxiong Guo. 2019. A generic communication scheduler for distributed DNN training acceleration. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 16–29.

[54] Hang Qi, Evan R Sparks, and Ameet Talwalkar. 2017. Paleo: A performance model for deep neural networks. (2017).

[55] Aurick Qiao, Sang Keun Choe, Suhas Jayaram Subramanya, Willie Neiswanger, Qirong Ho, Hao Zhang, Gregory R. Ganger, and Eric P. Xing. 2021. Pollux: Co-adaptive Cluster Scheduling for Goodput-Optimized Deep Learning. In *15th*

USENIX Symposium on Operating Systems Design and Implementation (OSDI 21). USENIX Association, 1–18.
[56] Michael T Rosenstein, Zvika Marx, Leslie Pack Kaelbling, and Thomas G Dietterich. 2005. To transfer or not to transfer. In *NIPS 2005 workshop on transfer learning*, Vol. 898. 1–4.
[57] Sebastian Ruder. 2016. An overview of gradient descent optimization algorithms. *arXiv preprint arXiv:1609.04747* (2016).
[58] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2818–2826.
[59] The Theano Development Team, Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermueller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, et al. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv preprint arXiv:1605.02688* (2016).
[60] Yiltan Hassan Temuçin, AmirHossein Sojoodi, Pedram Alizadeh, and Ahmad Afsahi. 2021. Efficient Multi-Path NVLink/PCIe-Aware UCX based Collective Communication for Deep Learning. In *2021 IEEE Symposium on High-Performance Interconnects (HOTI)*. IEEE, 25–34.
[61] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations*.
[62] Oriol Vinyals, Samy Bengio, and Manjunath Kudlur. 2015. Order matters: Sequence to sequence for sets. *arXiv preprint arXiv:1511.06391* (2015).
[63] Xin Wang, Shuyi Fan, Kun Kuang, and Wenwu Zhu. 2021. Explainable Automated Graph Representation Learning with Hyperparameter Importance. In *Proceedings of the 38th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 10727–10737.
[64] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems* 32, 1 (2020), 4–24.
[65] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. 2018. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. 595–610.
[66] Wencong Xiao, Shiru Ren, Yong Li, Yang Zhang, Pengyang Hou, Zhi Li, Yihui Feng, Wei Lin, and Yangqing Jia. 2020. AntMan: Dynamic Scaling on GPU Clusters for Deep Learning. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*. 533–548.
[67] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *International Conference on Learning Representations*.
[68] Zhilin Yang, Ruslan Salakhutdinov, and William W Cohen. 2017. Transfer learning for sequence tagging with hierarchical recurrent networks. *arXiv preprint arXiv:1703.06345* (2017).
[69] Xiaodong Yi, Shiwei Zhang, Ziyue Luo, Guoping Long, Lansong Diao, Chuan Wu, Zhen Zheng, Jun Yang, and Wei Lin. 2020. Optimizing distributed training deployment in heterogeneous GPU clusters. In *Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies*. 93–107.
[70] Wei Ying, Yu Zhang, Junzhou Huang, and Qiang Yang. 2018. Transfer learning via learning to transfer. In *International conference on machine learning*. PMLR, 5085–5094.
[71] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. Graph neural networks: A review of methods and applications. *AI Open* 1 (2020), 57–81.
[72] Yanqi Zhou, Sudip Roy, Amirali Abdolrashidi, Daniel Wong, Peter C Ma, Qiumin Xu, Ming Zhong, Hanxiao Liu, Anna Goldie, Azalia Mirhoseini, et al. 2019. GDP: Generalized device placement for dataflow graphs. *arXiv preprint arXiv:1910.01578* (2019).
[73] Hongyu Zhu, Amar Phanishayee, and Gennady Pekhimenko. 2020. Daydream: Accurately Estimating the Efficacy of Optimizations for DNN Training. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. 337–352.
[74] Fuzhen Zhuang, Zhiyuan Qi, Keyu Duan, Dongbo Xi, Yongchun Zhu, Hengshu Zhu, Hui Xiong, and Qing He. 2020. A comprehensive survey on transfer learning. *Proc. IEEE* 109, 1 (2020), 43–76.

A DESIGN OF *DRIPLE* INSPECTOR STRUCTURE

The model structure of the *Driple* inspector (Figure 7) is composed of two parts. The first part (ϕ_f, ϕ_r, and ρ) is a GNN for producing graph embedding. The second part (MLP) is to predict resource consumption using the graph embedding generated from the first part. We present the design rationale as follows.

The design rationale of the first part is presented in §5.2.2. For the second part of the *Driple* inspector, we test various candidate structures. First, we test MLP as the MLP is the representative

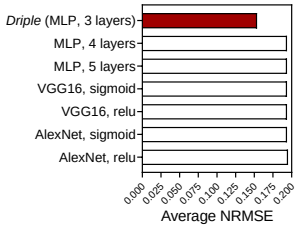
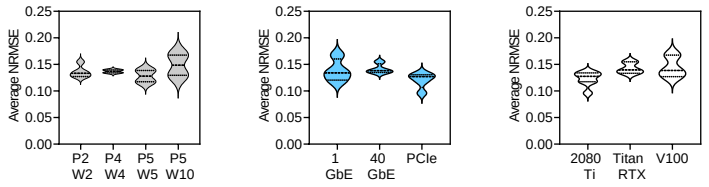


Fig. 19. Design comparison for *Driple* inspector. See more details in §A.



(a) # of PSs and workers change. (b) Network interconnect changes. (c) GPU type changes.

Fig. 20. Prediction accuracy of the *Driple* inspectors for DT setting changes. See more details in §B.

neural network structure to predict numerical values. We change the number of MLP layers from 4 to 7 but only show the results for 4 and 5 layers because the other layers show results identical to those of 5 layers. Second, we test convolution neural networks, such as VGG16 and AlexNet, because they have been proven to be accurate in many ML domains. We also change the optimizer of the convolutional neural networks as sigmoid and relu. Figure 19 shows the prediction accuracy between the candidate structures. We observe that the MLP of three layers produces the highest accuracy. This is why the MLP of three layers is chosen for the second part of the *Driple* inspector.

B PREDICTION ACCURACY ON DT SETTING CHANGES

We further analyze the relationship between the prediction accuracy and the DT setting changes in terms of 1) the change in the number of PSs and workers, 2) network interconnect types, and 3) GPU types. Figure 20a shows the accuracy when the PSs and workers increase. The median accuracies are in the order of P5W5, P2W2, P3W3, and P5W10; so, we do not observe any tendency between the prediction accuracy and the number of the PSs and workers.

Figure 20b analyzes the accuracy according to the network interconnect types. The median prediction error increases in the order of PCIe, 1 GbE, and 40 GbE. The GPUs in the experiments are connected through PCIe 3.0, which is faster than 1 GbE and 40 GbE. Although the networking speed is in the order of 1 GbE, 40 GbE, and PCIe, the increasing order prediction error is PCIe, 1 GbE, and 40 GbE. This also means that the networking speed is little correlated to the prediction accuracy. However, considering other characteristics, such that PCIe is the interconnect for co-located workers, and 1 GbE and 40 GbE are for distributed servers, we believe that identifying the precise relationship between the interconnects and prediction accuracy requires further research.

Lastly, in Figure 20c, the accuracies between three GPUs are compared. The median prediction error becomes higher (poorer accuracy) in the order of 2080 Ti, V100, and Titan RTX. The numbers of CUDA cores of 2080 Ti, V100, and Titan RTX are 4352, 5120, and 4608 each. Although V100 has the most CUDA cores, its prediction accuracy is in the middle. Again, this means that there is no specific correlation between the number of CUDA cores and the prediction accuracy. Given that the GPUs have so many other properties (e.g., GPU memory capacity, number of streaming processors, and number of CUDA cores), further investigation of the prediction accuracy is left as future work.

Received February 2022; revised March 2022; accepted April 2022