DLBricks: Composable Benchmark Generation to Reduce Deep Learning Benchmarking Effort on CPUs

Cheng Li, Abdul Dakkak
University of Illinois
Urbana-Champaign
Urbana, Illinois
{cli99,dakkak}@illinois.edu

Jinjun Xiong
IBM T. J. Watson Research Center
Yorktown Heights, New York
jinjun@us.ibm.com

Wen-mei Hwu
University of Illinois
Urbana-Champaign
Urbana, Illinois
w-hwu@illinois.edu

ABSTRACT

The past few years have seen a surge of applying Deep Learning (DL) models for a wide array of tasks such as image classification, object detection, machine translation, etc. While DL models provide an opportunity to solve otherwise intractable tasks, their adoption relies on them being optimized to meet target latency and resource requirements. Benchmarking is a key step in this process but has been hampered in part due to the lack of representative and up-to-date benchmarking suites.

This paper proposes DLBricks, a composable benchmark generation design that reduces the effort of developing, maintaining, and running DL benchmarks. DLBricks decomposes DL models into a set of unique runnable networks and constructs the original model's performance using the performance of the generated benchmarks. Since benchmarks are generated automatically and the benchmarking time is minimized, DLBricks can keep up-to-date with the latest proposed models, relieving the pressure of selecting representative DL models. We evaluate DLBricks using 50 MXNet models spanning 5 DL tasks on 4 representative CPU systems. We show that DLBricks provides an accurate performance estimate for the DL models and reduces the benchmarking time across systems (e.g. within 95% accuracy and up to 4.4× benchmarking time speedup on Amazon EC2 `c5.xlarge`).

CCS CONCEPTS

• Computing methodologies → Machine learning; **• General and reference → Performance**; **Evaluation**; **• Software and its engineering → Software maintenance tools**.

KEYWORDS

Deep Learning; Benchmarking; Performance Measurement

ACM Reference Format:
Cheng Li, Abdul Dakkak, Jinjun Xiong, and Wen-mei Hwu. 2020. DLBricks: Composable Benchmark Generation to Reduce Deep Learning Benchmarking Effort on CPUs. In *Proceedings of the 2020 ACM/SPEC International Conference on Performance Engineering (ICPE '20), April 20–24, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3358960.3379143

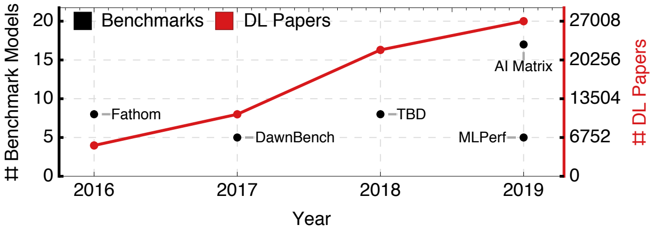


Figure 1: The number of DL models included in the recent published DL benchmark suites (Fathom [1], DawnBench [?], TBD [20], AI Matrix [19], and MLPerf [11]) compared to the number of DL papers published in the same year (using Scopus Preview [13]) .

1 INTRODUCTION

The recent progress made by Deep Learning (DL) in a wide array of applications, such as autonomous vehicles, face recognition, object detection, machine translation, fraud detection, etc. has led to increased public interest in DL models. Benchmarking these trained DL models before deployment is critical, as DL models must meet target latency and resource constraints. Hence there have been significant efforts to develop benchmark suites that evaluate widely used DL models [1, 11, 19, 20]. An example is MLPerf [11], which is formed as a collaboration between industry and academia and aims to provide reference implementations for DL model training and inference.

However, developing, maintaining, and running benchmarks takes a non-trivial amount of effort. For each DL task of interest, benchmark suite authors select a small representative subset (or one) out of tens or even hundreds of candidate models. Deciding on a representative set of models is an arduous effort as it takes a long debating process to determine what models to add and what to exclude. For example, it took over a year of weekly discussion to determine and publish MLPerf v0.5 inference models, and the number of models was reduced from the 10 models originally considered to 5. Figure 1 shows the gap between the number of DL papers [13] and the number of models included in recent benchmarking efforts. Given that DL models are proposed or updated on a daily basis [5, 8], it is very challenging for benchmark suites to be agile and representative of real-world DL model usage. Moreover, only public available models are considered for inclusion in benchmark suites. Proprietary models are trade secrets or restricted

ICPE '20, April 20–24, 2020, Edmonton, AB, Canada
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-6991-6/20/04…$15.00
https://doi.org/10.1145/3358960.3379143

by copyright and cannot be shared externally for benchmarking. Thus, proprietary models are not included or represented within benchmark suites.

To address the above issues, we propose DLBricks — a composable benchmark generation design that reduces the effort to develop, maintain, and run DL benchmarks. Given a set of DL models, DLBricks parses them into a set of atomic (i.e. non-overlapping) unique layer sequences based on the user-specified *benchmark granularity* (*G*). A *layer sequence* is a chain of layers. Two layer sequences are considered the *same* (i.e. not *unique*) if they are identical ignoring their weight values. DLBricks then generates unique *runnable networks* (i.e. subgraphs of the model with at most *G* layers that can be executed by a framework) using the layer sequences' information, and these networks form the representative set of benchmarks for the input models. Users run the generated benchmarks on a system of interest and DLBricks uses the benchmark results to construct a performance estimate on that system.

DLBricks leverages two key observations on DL inference: ❶ Layers are the performance building blocks of the model performance. ❷ Layers (considering their layer type, shape, and parameters, but ignoring the weights) are extensively repeated within and across DL models. DLBricks uses both observations to generate a representative benchmark suite, minimize the time to benchmark, and estimate a model's performance from layer sequences.

Since benchmarks are generated automatically by DLBricks, benchmark development and maintenance effort are greatly reduced. DLBricks is defined by a set of simple consistent principles and can be used to benchmark and characterize a broad range of models. Moreover, since each generated benchmark represents only a subset of the input model, the input model's topology does not appear in the output benchmarks. This, along with the fact that "fake" or dummy models can be inserted into the set of input models, means that the generated benchmarks can represent proprietary models without the concern of revealing proprietary models.

In summary, this paper makes the following contributions:

- We perform a comprehensive performance analysis of 50 state-of-the-art DL models on CPUs and observe that layers are the performance building blocks of DL models, thus a model's performance can be estimated using the performance of its layers (Section 2.1).
- We also perform an in-depth DL architecture analysis of the DL models and make the observation that DL layers with the same type, shape, and parameters are repeated extensively within and across models (Section 2.2).
- We propose DLBricks, a composable benchmark generation design that decomposes DL models into a set of unique runnable networks and constructs the original model's performance using the performance of the generated benchmarks (Section 3).
- We evaluate DLBricks using 50 MXNet models spanning 5 DL tasks on 4 representative CPU systems (Section 4). We show that DLBricks provides a tight performance estimate for DL models and reduces the benchmarking time across systems. The composed model latency is within 95% of the actual performance while up to 4.4× benchmarking speedup is achieved on the Amazon EC2 `c5.xlarge` system.

This paper is structured as follows. First, we detail two key observations that enable our design in Section 2. We then propose DLBricks in Section 3 and describe how it provides a streamlined benchmark generation workflow which lowers the effort to benchmark. Section 4 evaluates DLBricks using 50 models running on 4 systems. In Section 5 we describe different benchmarking approaches previously performed. We then describe future work in Section 6 before we conclude in Section 7.

Table 1: The 50 MXNet models [12] used for evaluation, including Image Classification (IC), Image Processing (IP), Object Detection (OD), Regression (RG) and Semantic Segmentation (SS) tasks.

ID	Name	Task	Num Layers
1	Ademxapp Model A Trained on ImageNet Competition Data	IC	142
2	Age Estimation VGG-16 Trained on IMDB-WIKI and Looking at People Data	IC	40
3	Age Estimation VGG-16 Trained on IMDB-WIKI Data	IC	40
4	CapsNet Trained on MNIST Data	IC	53
5	Gender Prediction VGG-16 Trained on IMDB-WIKI Data	IC	40
6	Inception V1 Trained on Extended Salient Object Subitizing Data	IC	147
7	Inception V1 Trained on ImageNet Competition Data	IC	147
8	Inception V1 Trained on Places365 Data	IC	147
9	Inception V3 Trained on ImageNet Competition Data	IC	311
10	MobileNet V2 Trained on ImageNet Competition Data	IC	153
11	ResNet-101 Trained on ImageNet Competition Data	IC	347
12	ResNet-101 Trained on YFCC100m Geotagged Data	IC	344
13	ResNet-152 Trained on ImageNet Competition Data	IC	517
14	ResNet-50 Trained on ImageNet Competition Data	IC	177
15	Squeeze-and-Excitation Net Trained on ImageNet Competition Data	IC	874
16	SqueezeNet V1.1 Trained on ImageNet Competition Data	IC	69
17	VGG-16 Trained on ImageNet Competition Data	IC	40
18	VGG-19 Trained on ImageNet Competition Data	IC	46
19	Wide ResNet-50-2 Trained on ImageNet Competition Data	IC	176
20	Wolfram ImageIdentify Net V1	IC	232
21	Yahoo Open NSFW Model V1	IC	177
22	AdaIN-Style Trained on MS-COCO and Painter by Numbers Data	IP	109
23	Colorful Image Colorization Trained on ImageNet Competition Data	IP	58
24	ColorNet Image Colorization Trained on ImageNet Competition Data	IP	62
25	ColorNet Image Colorization Trained on Places Data	IP	62
26	CycleGAN Apple-to-Orange Translation Trained on ImageNet Competition Data	IP	94
27	CycleGAN Horse-to-Zebra Translation Trained on ImageNet Competition Data	IP	94
28	CycleGAN Monet-to-Photo Translation	IP	94
29	CycleGAN Orange-to-Apple Translation Trained on ImageNet Competition Data	IP	94
30	CycleGAN Photo-to-Cezanne Translation	IP	96
31	CycleGAN Photo-to-Monet Translation	IP	94
32	CycleGAN Photo-to-Van Gogh Translation	IP	96
33	CycleGAN Summer-to-Winter Translation	IP	94
34	CycleGAN Winter-to-Summer Translation	IP	94
35	CycleGAN Zebra-to-Horse Translation Trained on ImageNet Competition Data	IP	94
36	Pix2pix Photo-to-Street-Map Translation	IP	56
37	Pix2pix Street-Map-to-Photo Translation	IP	56
38	Very Deep Net for Super-Resolution	IP	40
39	SSD-VGG-300 Trained on PASCAL VOC Data	OD	145
40	SSD-VGG-512 Trained on MS-COCO Data	OD	157
41	YOLO V2 Trained on MS-COCO Data	OD	106
42	2D Face Alignment Net Trained on 300W Large Pose Data	RG	967
43	3D Face Alignment Net Trained on 300W Large Pose Data	RG	967
44	Single-Image Depth Perception Net Trained on Depth in the Wild Data	RG	501
45	Single-Image Depth Perception Net Trained on NYU Depth V2 and Depth in the Wild Data	RG	501
46	Single-Image Depth Perception Net Trained on NYU Depth V2 Data	RG	501
47	Unguided Volumetric RG Net for 3D Face Reconstruction	RG	1029
48	Ademxapp Model A1 Trained on ADE20K Data	SS	141
49	Ademxapp Model A1 Trained on PASCAL VOC2012 and MS-COCO Data	SS	141
50	Multi-scale Context Aggregation Net Trained on CamVid Data	SS	53

2 MOTIVATION

DLBricks is designed based on two key observations presented in this section. To demonstrate and support these observations, we perform comprehensive performance and architecture analysis of state-of-the-art DL models. Evaluations in this section use 50 MXNet models of different DL tasks (listed in Table 1) and were run with MXNet (v1.5.1 MKL release) on a Amazon `c5.2xlarge` instance (as listed in Table 2). We focus on latency sensitive (batch size = 1) DL inference on CPUs.

2.1 Layers as the Performance Building Blocks

A DL model is a directed acyclic graph (DAG) where each vertex within the DAG is a layer (i.e. operator, such as convolution, batchnormalization, pooling, element-wise, softmax) and an edge represents the transfer of data. For a DL model, a *layer sequence* is defined as a simple path within the DAG containing one or more vertices. A *subgraph*, on the other hand, is defined as a DAG composed of one or more layers within the model (i.e. subgraph is a superset of layer sequence, and may or may not be a simple path). We are only interested in network subgraphs that are runnable within frameworks and we call these runnable subgraphs *runnable networks*.

DL models may contain layers that can be executed independently in parallel. The network made of these data-independent layers is called a *parallel module*. For example, Figure 2a shows the `VGG16` [14] (ID=17) model architecture. `VGG16` contains no parallel module and is a linear sequence of layers. `Inception V3` [15] (ID=9) (shown in Figure 2b), on the other hand, contains a mix of layer sequences and parallel modules.

DL frameworks such as TensorFlow, PyTorch, and MXNet execute a DL model by running the layers within the model graph. We explore the relation between layer performance and model performance by decomposing each DL model in Table 1 into layers. We define a model's *critical path* to be a simple path from the start layer to the end layer with the highest latency. For a DL model, we add all its layers' latency and refer to the sum as the *sequential total layer latency*, since this assumes all the layers are executed sequentially by the DL framework. Theoretically, data-independent paths within a parallel module can be executed in parallel, thus we also calculate the *parallel total layer latency* by adding up the layer latencies along the critical path. The critical path of both `VGG 16` (ID=17) and `Inception V3` (ID=9) is highlighted in red in Figure 2. For models that do not have parallel modules, the sequential total layer latency is equal to the parallel total layer latency.

For each of the 50 models, we compare both sequential and parallel total layer latency to the model's end-to-end latency. Figure 3 shows the normalized latencies in both cases. For models with parallel modules, the parallel total layer latencies are much lower than the model's end-to-end latency. The difference between the sequential total layer latencies and the models' end-to-end latencies are small. The normalized latencies are close to 1 with a geometric metric mean of 91.8% for the sequential case. This suggests the current software/hardware stack does not exploit parallel execution of data-independent layers or overlapping of layer execution, we verified this by inspecting the source code of popular frameworks such as MXNet, PyTorch, and TensorFlow.

The difference between a model's end-to-end latency and its sequential total layer latency is due to the complexity of model execution within DL frameworks and the underlying software/hardware stack. We identified two major factors that may affect this difference: framework overhead and memory caching. Executing a model within frameworks introduced an overhead that is roughly proportional to the number of the layers. This is because frameworks need to perform bookkeeping, layer scheduling, and memory management for model execution. Therefore, the measured end-to-end performance can be larger than the total layer latency. On the other hand, both the framework and the underlying software/hardware stack can take advantage of caching to decrease the latency of data-dependent layers. For memory-bound layers, this can achieve significant speedup and therefore the measured end-to-end performance can be lower than the total layer latency. Depending on which factor is dominant, the normalized latency can be larger or smaller than 1. Based on this, we formulate the ❶ observation:

> **Observation 1:** DL layers are the performance building blocks of the model performance, therefore, a model's performance can be estimated using the performance of its layers. Moreover, a simple summation of layer-wise latency is an effective approximation of the end-to-end latency given the current DL software stack (no parallel execution of data-independent layers or overlapping of layer execution) on CPUs.

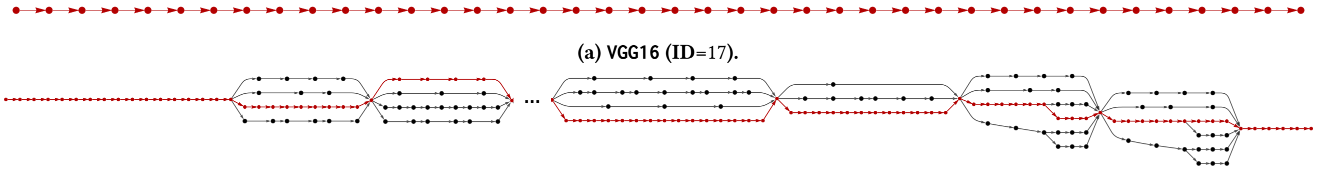
2.2 Layer Repeatability

From a model architecture point of view, a DL layer is identified by its type, shape, and parameters. For example, a convolution layer is identified by its input shape, output channels, kernel size, stride, padding, dilation, etc. Layers with the same type, shape, parameters (i.e. only differ in weights) are expected to have the same performance. We inspected the source code of popular frameworks and verified this, as they do not perform any special optimizations for weights. Thus in this paper we consider two layers to be the *same* if they have the same type, shape, parameters, ignoring weight values, and two layers are *unique* if they are not the same.

DL models tend to have repeated layers or modules (or subgraphs, e.g. Inception and ResNet modules). For example, Figure 4 shows the model architecture of `ResNet-50` with the ResNet modules detailed. Different ResNet modules have layers in common and ResNet modules $2, 4, 6, 8$ are entirely repeated within `ResNet-50`. Moreover, DL models are often built on top of existing models (e.g. transfer learning [17] where models are retrained with different data), using common modules (e.g. TensorFlow Hub [16]), or using layer bundles for Neural Architecture Search [7, 18]. This results in ample repeated layers when looking at a corpus of models. We quantitatively explore the layer repeatability within and across models.

Figure 5 shows the percentage of unique layers within each model in Table 1. We can see that layers are extensively repeated within DL models. For example, in `Unguided Volumetric Regression Net for 3D Face Reconstruction` (ID=47) which has 1029 layers, only 3.9% of the total layers are unique. We further look at the repeated layers within each model and Figure 6 shows their type distribution. As we can see Convolution, Elementwise, BatchNorm, and Norm are the most repeated layer types in terms of intra-model layer repeatability. If we consider all 50 models in Table 1, the total number of layers is $10,815$, but only $1,529$ are unique (i.e. 14% are unique).

We illustrate the layer repeatability across models by quantifying the similarity of any two models listed in Table 1. We use the Jaccard similarity coefficient; i.e. for any two models M_1 and M_2 the Jaccard similarity coefficient is defined by $\frac{|\mathcal{L}_1 \cap \mathcal{L}_2|}{|\mathcal{L}_1 \cup \mathcal{L}_2|}$ where $\mathcal{L}_1$ and $\mathcal{L}_2$ are the layers of M_1 and M_2 respectively. The results are shown in Figure 7. Each cell corresponds to the Jaccard similarity coefficient between the models at the row and column. As shown, models that share the same base architecture but are retrained using different data (e.g. `CycleGAN*` models with IDs $26 - 35$ and `Inception V1*` models with IDs $6 - 8$) have many common layers. Layers are



(a) VGG16 (ID=17).

(b) Inception V3 (ID=9).

Figure 2: The model architecture of VGG16 (ID=17) and Inception V3 (ID=9). The critical path is highlighted in red.

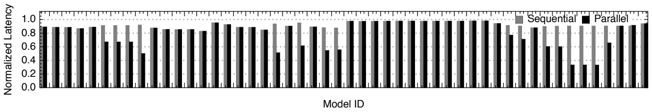


Figure 3: The sequential and parallel total layer latency normalized to the model's end-to-end latency using batch size 1 on c5.2xlarge in Table 2.

common across models within the same family (e.g. ResNet*) since they are built from the same set of modules (e.g. ResNet-50 is shown in Figure 4), or when solving the same task (e.g. the image classification task category). Based on this, we formulate the ❷ observation:

> **Observation 2:** Layers are repeated within and across DL models. This enables us to decrease the benchmarking time since only a representative set of layers need to be evaluated.

The above two observations suggest that if we can decompose models into layers, and then take the union of them to produce a set of representative runnable networks, then benchmarking the representative runnable networks is sufficient to construct the performance of the input models. Since we only look at the representative set, the total runtime is less than running all models directly, thus DLBricks can be used to reduce benchmarking time. Since layer decomposition elides the input model topology, models can be private while their benchmarks can be public. The next section (Section 3) describes how we leverage these two observations to build a benchmark generator while having a workflow where one can construct a model's performance based on the benchmarked layer performance. We further explore the design space of benchmark granularity and its effect on performance construction accuracy.

3 DESIGN

This section presents DLBricks, a composable benchmark generation design for DL models. The design is motivated by the two observations discussed in Section 2. DLBricks explores not only layer level model composition, but also sequence level composition where a *layer sequence* is a chain of layers. The *benchmark granularity* (G) specifies the maximum numbers of layers within any layer sequence in the output generated benchmarks. G is introduced to account for the effects of model execution complexity (e.g. framework overhead and caching as discussed in Section 2.1). Thus, a larger G is expected to increase the accuracy of performance construction. On the other hand, a larger G might decrease the layer repeatability across models. Therefore, a balance needs to be struck (by the user) between performance construction accuracy and benchmarking time speedup.

The design and workflow of DLBricks is shown in Figure 8. DLBricks consists of a benchmark generation workflow and a performance construction workflow. To generate composable benchmarks, one uses the *benchmark generator workflow* where: ① the user inputs a set of models (M_1, ..., M_n) along with a target benchmark granularity. ② The benchmark generator parses the input models into a representative (unique) set of non-overlapping layer sequences and then generates a set of runnable networks (S_1, ..., S_k) using these layer sequences' information. ③ The user evaluates the set of runnable networks on a system of interest to get each benchmark's corresponding performance (P_{S_1}, ..., P_{S_k}). The benchmark results are stored and ④ are used within the *performance construction workflow.* ⑤ To construct the performance of an input model, the performance constructor queries the stored benchmark results for the layer sequences within the model, and then ⑥ computes the model's estimated performance (P_{M_1}, ..., P_{M_k}). This section describes both workflows in detail.

3.1 Benchmark Generation

The benchmark generator takes a list of models $M_1, \ldots, M_n$ and a benchmark granularity G. The *benchmark granularity* specifies the maximum sequence length of the layer sequences generated. This means that when $G = 1$, each generated benchmark is a single-layer network, whereas when $G = 2$ each generated benchmark contains at most 2 layers.

To split a model with the specified benchmark granularity, we use FindModelSubgraphs (Algorithm 1). The FindModelSubgraphs takes a model and a maximum sequence length and iteratively generates a set of non-overlapping layer sequences. First, the layers in the model are sorted topologically and then calls the SplitModel function (Algorithm 2) with the desired begin and end layer offset. This SplitModel tries to create a runnable DL network (i.e. a valid DL network) using the range of layers desired, if it fails (e.g. a network which cannot be constructed due to input/output layer shape mismatch[1]), then SplitModel creates a network with the current layer and shifts the begin and end positions. The SplitModel returns a list of runnable DL networks (S_i, ..., S_{i+j}) along with the end position to FindModelSubgraphs. The FindModelSubgraphs terminates when no other subsequences can be created.

The benchmark generator applies the FindModelSubgraphs for each of the input models. A set of representative (i.e. *unique*) runnable DL networks ($S_1, \ldots, S_k$) is then computed. We say two sequences S_1 and S_2 are the same if they have the same topology along with the same node parameters (i.e. they are the same DL network modulo

[1] An example invalid network is one which contains a Concat layer, but does not have all of the Concat layer's required input layers.

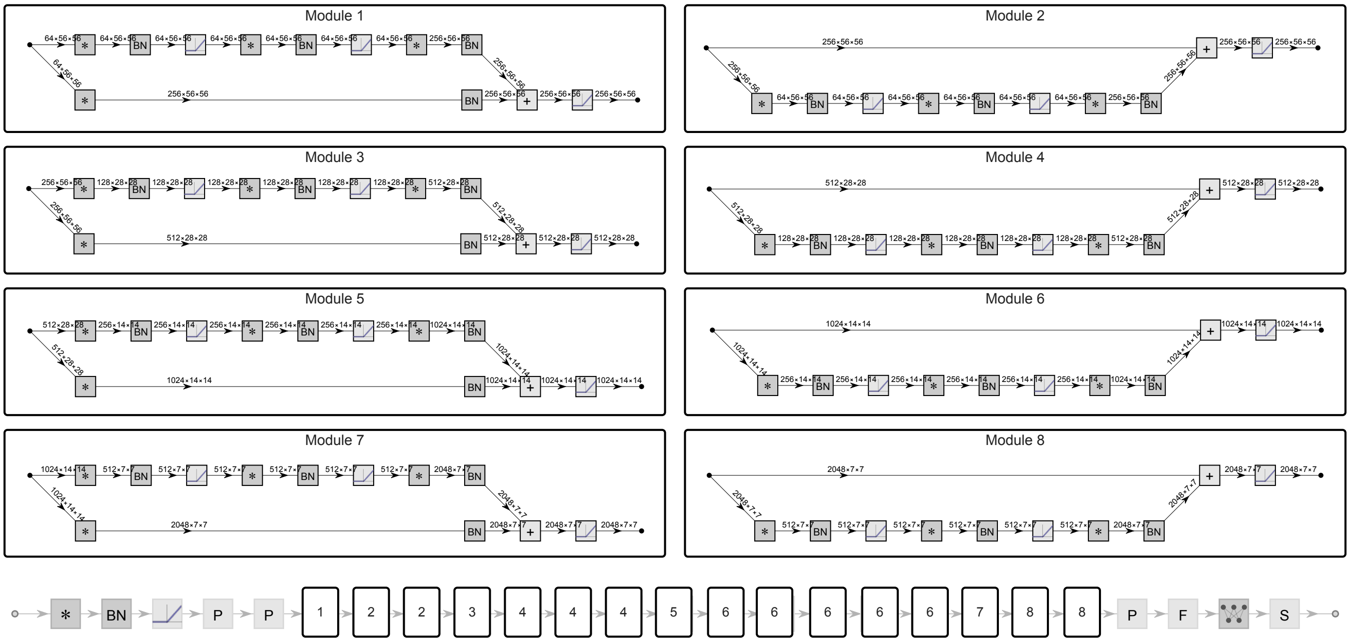


Figure 4: The ResNet-50 (ID=14) architecture. The detailed ResNet modules 1 − 8 are listed above the model graph.

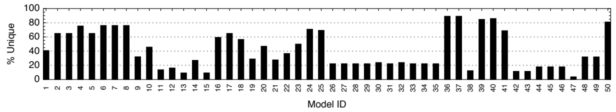


Figure 5: The percentage of unique layers in the models in Table 1, indicating that some layers are repeated within the model.

Algorithm 1 The `FindModelSubgraphs` algorithm.

```
Input: $M$ (Model), $G$ (Benchmark Granularity)
Output: $Models$
$begin \leftarrow 0, Models \leftarrow \{\}$
$verts \leftarrow$ TopologicalOrder(ToGraph($M$))
while $begin \leq$ Length($verts$) do
    $end \leftarrow$ Min($begin + G$, Length($vs$))
    $sm \leftarrow$ SplitModel($verts, begin, end$)
    $Models \leftarrow Models + sm$ ["models"]
    $begin \leftarrow sm$ ["end"] $+ 1$
end while
return $Models$
```

Algorithm 2 The `SplitModel` algorithm.

```
Input: $verts, begin, end$
Output: ⟨"models", "end"⟩                                  ▷ Hash table
$vs \leftarrow verts\,[begin : end]$
try
    $m \leftarrow$ CreateModel($vs$)                        ▷ Creates a valid model
    return ⟨"models" → $\{m\}$, "end" → $end$⟩ ▷ Hash table with keys: "model" and "end"
catch ModelCreateException
    $m \leftarrow$ {CreateModel($\{verts\,[begin]\}$)}      ▷ Creates a model with a single node
    $n \leftarrow$ SplitModel($verts, begin + 1, end + 1$)  ▷ Recrusively split the model
    return ⟨"models" → $m + n$ ["models"], "end" → $n$ ["end"]⟩
end try
```

the weights). The unique networks are exported to the frameworks' network format and the user runs them with synthetic input data based on each network's input shape. The performance of each network is stored (P_{S_i} …, P_{S_k}) and used by the performance construction workflow.

3.2 DL Model Performance Construction

DLBricks uses the performance of the layer sequences to construct an estimate to the end-to-end performance of the input model M. To construct a performance estimate, the input model is parsed and goes through the same process ① in Figure 8. This creates a set of layer sequences. The performance of each layer sequence is queried from the benchmark results (P_{S_i} …, P_{S_k}). DLBricks supports both sequential and parallel performance construction. Sequential performance construction is performed by summing up all of the resulting queried results, whereas parallel performance construction sums up the results along the critical path of the model. Since current frameworks exhibit a sequential execution strategy (from Section 2.1), sequential performance construction is used within DLBricks by default. Other performance construction can be easily added to DLBricks to accommodate different framework execution strategies.

4 EVALUATION

This section focuses on demonstrating DLBricks is valid in terms of performance construction accuracy and benchmarking time speedup. We explore the effect of benchmark granularity on the constructed performance estimation as well as the benchmarking time. We evaluated DLBricks with 50 DL models (listed in Table 1) using MXNet (v1.5.1 using MKL v2019.3) on 4 different Amazon EC2 instances. These systems are recommended by Amazon [2] for DL inference and are listed in Table 2. To maintain consistent CPU evaluation, the systems are configured to disable CPU frequency scaling, turbo-boosting, scaling-governor, and hyper-threading. Each benchmark is run 100 times and the 20th percentile trimmed mean is reported.

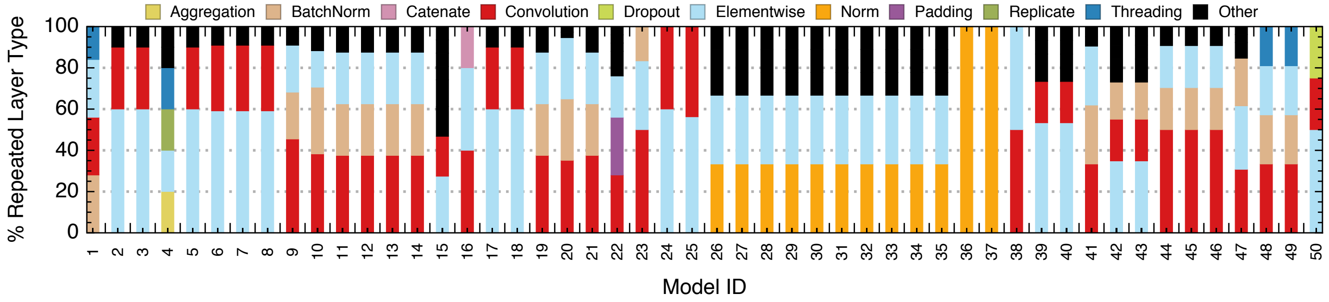


Figure 6: The type distribution of the repeated layers.

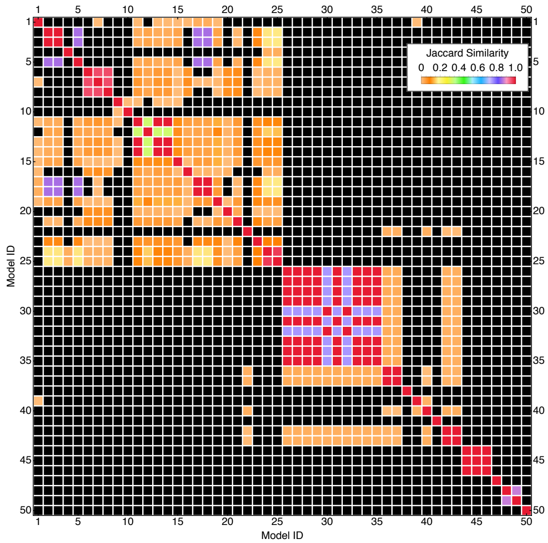


Figure 7: The Jaccard Similarity grid of the models in Table 1. Solid red indicates two models have identical layers, and black means there is no common layer.

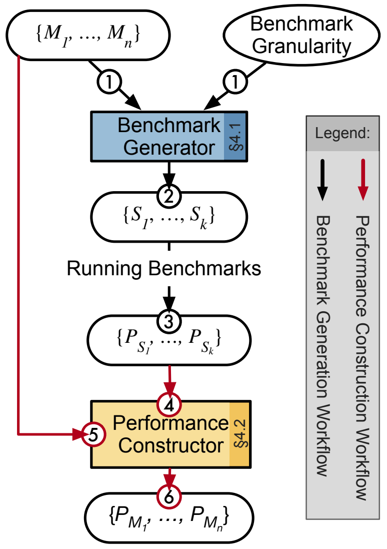


Figure 8: DLBricks design and workflow.

Table 2: Evaluations are performed on the 4 Amazon EC2 systems listed. The c5.* systems operate at 3.0GHz, while the c4.* systems operate at 2.9GHz. The systems are ones recommended by Amazon for DL inference.

Instance	CPUS	Memory (GiB)	\$/hr
c5.xlarge	4 Intel Platinum 8124M	8GB	0.17
c5.2xlarge	8 Intel Platinum 8124M	16GB	0.34
c4.xlarge	4 Intel Xeon E5-2666 v3	7.5GB	0.199
c4.2xlarge	8 Intel Xeon E5-2666 v3	15GB	0.398

4.1 Performance Construction Accuracy

We first ran the end-to-end models on the 4 systems to understand their performance characteristics, as shown in Figure 9. Then, using DLBricks, we constructed the latency estimate of the models based on the performance of their layer sequence benchmarks. Figure 10 shows the constructed model latency normalized to the model's end-to-end latency for all the models with varying benchmark granularity from 1 to 6 on c5.2xlarge. We see that the constructed latency is a tight estimate of the model's actual performance across models and benchmark granularities. E.g., for benchmark granularity $G = 1$, the normalized latency ranges between 82.9% and 98.1% with a geometric mean of 91.8%.

As discussed in Section 2.1, the difference between a model's end-to-end latency and its constructed latency is due to the combinational effect of model execution complexity such as framework overhead and caching, thus the normalized latency can be either below or above 1. At $G = 1$ (layer granularity model decomposition and construction), where a model is decomposed into the largest number of sequences, the constructed latency is slightly less accurate compared to other G values. Using the number of layers in Table 1 and the model end-to-end latency in Figure 9, we see no direct correlation between the performance construction accuracy, number of model layers, or end-to-end latency.

Figure 11 shows the geometric mean of the normalized latency (the constructed latency normalized to the end-to-end latency) of all the 50 models across systems and benchmark granularities. Model execution in a framework is system-dependent, thus the performance construction accuracy is not only model-dependent but also system-dependent. Overall, the estimated latency is within 5% (e.g. $G = 3, 5, 9, 10$) to 11% ($G = 1$) of the model end-to-end latency across systems. This demonstrates that DLBricks provides a tight estimate to input models' actual performance across systems.

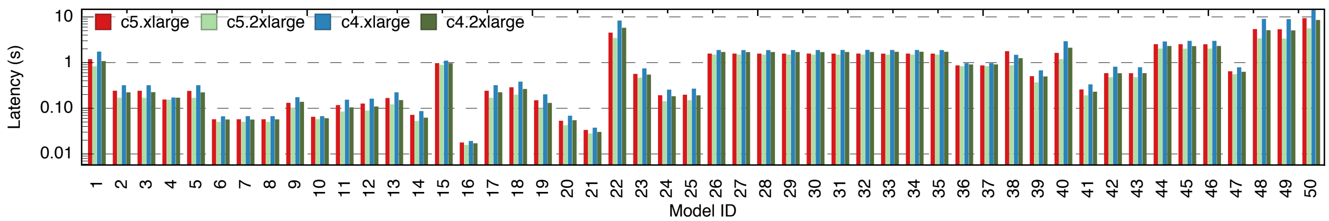
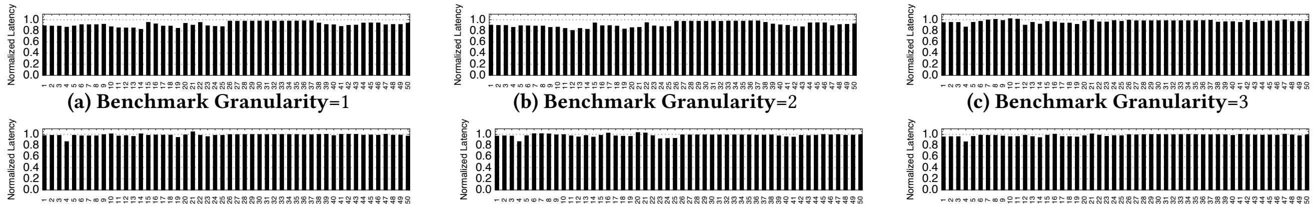


Figure 9: The end-to-end latency of all models in log scale across systems.



(a) Benchmark Granularity=1

(b) Benchmark Granularity=2

(c) Benchmark Granularity=3

(d) Benchmark Granularity=4

(e) Benchmark Granularity=5

(f) Benchmark Granularity=6

Figure 10: The constructed model latency normalized to the model's end-to-end latency for the 50 model in Table 1 on `c5.2xlarge`. The benchmark granularity varies from 1 to 6. Sequence 1 means each benchmark has one layer (layer granularity).

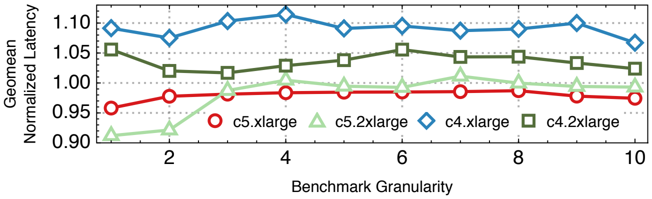


Figure 11: The geometric mean of the normalized latency (constructed vs end-to-end latency) of all the 50 models on the 4 systems with varying benchmark granularity from 1 to 10.

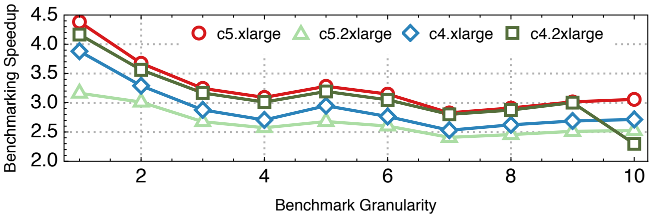


Figure 12: The speedup of total benchmarking time for the all the models across systems and benchmark granularities.

4.2 Benchmarking Time Speedup

DLBricks decreases the benchmarking time by only evaluating the unique layer sequences within and across models. Recall from Section 2.2 that for all the 50 models, the total number of layers is 10, 815, but only 1, 529 are unique (i.e. 14% are unique). Figure 12 shows the speedup of the total benchmarking time across systems as benchmark granularity varies. The benchmarking time speedup is calculated as the sum of the end-to-end latency of all models divided by the sum of the latency of all the generated benchmarks. Up to 4.4× benchmarking time speedup is observed for $G = 1$ on the `c5.xlarge` system. The speedup decreases as the benchmark granularity increases. This is because as the benchmark granularity increases, the chance of having repeated layer sequences within and across models decreases.

Figure 11 and Figure 12 suggest a trade-off exists between the performance construction accuracy and benchmarking time speedup and the trade-off is system-dependent. For example, while $G = 1$ (layer granularity model decomposition and construction) produces the maximum benchmarking time speedup, the constructed latency is slightly less accurate comparing to other G values on the systems. Since this accuracy loss is small, overall, $G = 1$ is a good choice of benchmark granularity configuration for DLBricks given the current DL software stack on CPUs.

5 RELATED WORK

To characterize the performance of DL models, both industry and academia have invested in developing benchmark suites that characterize models and systems. The benchmarking methods are either end-to-end benchmarks (performing user-observable latency measurement on a set of representative DL models [11, 19?]) or are micro-benchmarks [3, 4, 19] (isolating common kernels or layers that are found in models of interest). The end-to-end benchmarks target end-users and measure the latency or throughput of a model under a specific workload scenario. The micro-benchmark approach, on the other hand, distills models to their basic atomic operations (such as dense matrix multiplies, convolutions, or communication routines) and measures their performance to guide hardware or software design improvements [6]. While both approaches are valid and have their use cases, their benchmarks are manually selected and developed. As discussed, curating and maintaining these benchmarks requires significant effort and, in the case of lack of maintenance, these benchmarks become less representative of real-world models.

DLBricks complements the DL benchmarking landscape as it introduces a novel benchmarking methodology which reduces the effort of developing, maintaining, and running DL benchmarks. DLBricks relieves the pressure of selecting representative DL models and copes well with the fast-evolving pace of DL models. DLBricks automatically decomposes DL models into runnable networks and generates micro-benchmarks based on these networks. Users can specify the benchmark granularity. At the two extremes, when the granularity is 1 a layer-based micro-benchmark is generated, whereas when the granularity is equal to the number of layers within the model then an end-to-end network is generated. To our knowledge, there has been no previous work solving the same problem and we are the first to propose such a design.

Previous work [10] also decomposed DL models into layers, but uses the results to guide performance optimization. DLBricks focuses on model performance and aims to reduce benchmarking effort. DLBricks shares similar spirit to synthetic benchmark generation [9]. However, to the authors' knowledge, there has been no previous work on synthetic benchmark generation for DL.

6 DISCUSSION AND FUTURE WORK

Generating Overlapping Benchmarks. — The current design only considers non-overlapping layer sequences during benchmark generation. This may inhibit some types of optimizations (such as layer fusion). A solution requires a small tweak to Algorithm 1 where we increment the *begin* by 1 rather than the end index of the `SplitModel` algorithm (line 7). A small modification is also needed within the performance construction step to pick the layer sequence resulting in the smallest latency. Future work would explore the design space when generated benchmarks can overlap.

Adapting to Framework Evolution. — The current DLBricks design is based on the observation that current DL frameworks do not execute data-independent layers in parallel. Although DLBricks supports both sequential and parallel execution (assuming all data-independent layers are executed in parallel as described in Section 3.2), as DL frameworks start to have some support of parallel execution of data-independent layers, the current design may needs to be adjusted. To adapt DLBricks to this evolution of frameworks, one can adjust DLBricks to take user-specified parallel execution rules. DLBricks can then use the parallel execution rules to make a more accurate model performance estimation.

Future Work. — While this work focuses on CPUs, we expect the design to hold for GPUs as well. Future work would explore the design for GPUs. We are also interested in other use cases that are afforded by the DLBricks design — model/system comparison and advising for the cloud. For example, it is common to ask questions such as, *given a DL model which system should I use? or given a system and a task, which model should I use?* Using DLBricks, the system provider can curate a continuously updated database of the generated benchmarks results across its system offerings. The system provider can then perform a performance estimate of the user's DL model (without running it) and give suggestions as to which system to choose.

7 CONCLUSION

The fast-evolving landscape of DL poses considerable challenges in the DL benchmarking practice. While benchmark suites are under pressure to be agile, up-to-date, and representative, we take a different approach and propose a novel benchmarking design — aimed at relieving this pressure. Leveraging the key observations that layers are the performance building block of DL models and the layer repeatability within and across models, DLBricks automatically generates composable benchmarks that reduce the effort of developing, maintaining, and running DL benchmarks. Through the evaluation of state-of-the-art models on representative systems, we demonstrated that DLBricks provides a trade-off between performance construction accuracy and benchmarking time speedup. As the benchmark generation and performance construction workflows in DLBricks are fully automated, the generated benchmarks and their performance can be continuously updated and augmented as new models are introduced with minimal effort from the user. Thus DLBricks copes with the fast-evolving pace of DL models.

ACKNOWLEDGMENTS

This work is supported by the IBM-ILLINOIS Center for Cognitive Computing Systems Research (C3SR) - a member of the IBM Cognitive Horizon Network, and the Applications Driving Architectures (ADA) Research Center - one of the JUMP Centers co-sponsored by SRC and DARPA.

REFERENCES

[1] Robert Adolf, Saketh Rama, Brandon Reagen, Gu-yeon Wei, and David Brooks. 2016. Fathom: Reference workloads for modern deep learning methods. In *2016 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, IEEE, 1–10.

[2] Amazon. 2019. Recommended CPU Instances. docs.aws.com/dlami/latest/devguide/cpu.html. Accessed: 2019-10-17.

[3] Baidu. 2019. DeepBench. github.com/baidu-research/DeepBench.

[4] Soumith Chintala. 2019. ConvNet Benchmarks. github.com/soumith/convnet-benchmarks.

[5] Jeff Dean, David Patterson, and Cliff Young. 2018. A New Golden Age in Computer Architecture: Empowering the Machine-Learning Revolution. *IEEE Micro* 38, 2 (March 2018), 21–29.

[6] Shi Dong, Xiang Gong, Yifan Sun, Trinayan Baruah, and David Kaeli. 2018. Characterizing the Microarchitectural Implications of a Convolutional Neural Network (CNN) Execution on GPUs. In *Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering - ICPE '18*. ACM, ACM Press, 96–106.

[7] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. 2019. Neural Architecture Search: A Survey. *Journal of Machine Learning Research* 20, 55 (2019), 1–21.

[8] Kim Hazelwood and et al. 2018. Applied Machine Learning at Facebook: A Datacenter Infrastructure Perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, IEEE, 620–629.

[9] M.D. Hutton, J.S. Rose, and D.G. Corneil. 2002. Automatic generation of synthetic sequential benchmark circuits. *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.* 21, 8 (Aug. 2002), 928–940.

[10] Cheng Li, Abdul Dakkak, Jinjun Xiong, and Wen-Mei Hwu. 2020. Benanza: Automatic μBenchmark Generation to Compute "Lower-bound" Latency and Inform Optimizations of Deep Learning Models on GPUs. IEEE. The 34th IEEE International Parallel & Distributed Processing Symposium (IPDPS'20).

[11] MLPerf. 2019. MLPerf. github.com/mlperf.

[12] NeuralNetRepository 2019. Wolfram NeuralNet Repository. https://resources.wolframcloud.com/NeuralNetRepository/. Accessed: 2019-10-17.

[13] Scopus Preview. [n.d.]. Scopus Preview. https://www.scopus.com/. Accessed: 2019-10-17.

[14] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR* abs/1409.1556 (2014). arxiv.org/abs/1409.1556

[15] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the Inception Architecture for Computer Vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE, 2818–2826.

[16] TensorFlow Hub [n.d.]. TensorFlow Hub is a library for reusable machine learning modules . https://www.tensorflow.org/hub. Accessed: 2019-10-17.

[17] Karl Weiss, Taghi M. Khoshgoftaar, and DingDing Wang. 2016. A survey of transfer learning. *J Big Data* 3, 1 (May 2016), 9.

[18] Bichen Wu, Xiaoliang Dai, Peizhao Zhang, Yanghan Wang, Fei Sun, Yiming Wu, Yuandong Tian, Peter Vajda, Yangqing Jia, and Kurt Keutzer. 2018. FBNet: Hardware-aware Efficient ConvNet Design via Differentiable Neural Architecture Search. *CoRR* abs/1812.03443 (2018). arxiv.org/abs/1812.03443

[19] Wei Zhang, Wei Wei, Lingjie Xu, Lingling Jin, and Cheng Li. 2019. AI Matrix: A Deep Learning Benchmark for Alibaba Data Centers. *arXiv preprint arXiv:1909.10562* (2019).

[20] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Anand Jayarajan, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. 2018. Benchmarking and Analyzing Deep Neural Network Training. In *2018 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, IEEE, 88–100.