A Modular Benchmarking Infrastructure for High-Performance and Reproducible Deep Learning

Tal Ben-Nun, Maciej Besta, Simon Huber, Alexandros Nikolaos Ziogas, Daniel Peter, Torsten Hoefler
Department of Computer Science, ETH Zurich

***Abstract*—We introduce Deep500: the first customizable benchmarking infrastructure that enables fair comparison of the plethora of deep learning frameworks, algorithms, libraries, and techniques. The key idea behind Deep500 is its *modular* design, where deep learning is factorized into four distinct *levels*: operators, network processing, training, and distributed training. Our evaluation illustrates that Deep500 is *customizable* (enables combining and benchmarking different deep learning codes) and *fair* (uses carefully selected metrics). Moreover, Deep500 is *fast* (incurs negligible overheads), *verifiable* (offers infrastructure to analyze correctness), and *reproducible*. Finally, as the first distributed and reproducible benchmarking system for deep learning, Deep500 provides software infrastructure to utilize the most powerful supercomputers for extreme-scale workloads.**

***Index Terms*—Deep Learning, High-Performance Computing, Benchmarks, Distributed Deep Learning**

Deep500 Code: https://www.github.com/deep500/deep500

I. Introduction

Deep Learning (DL) has transformed the world and is now ubiquitous in areas such as speech recognition, image classification, or autonomous driving [3]. Its central concept is a Deep Neural Network (DNN), a structure modeled after the human brain. Thanks to rigorous training, DNNs are able to solve various problems, previously deemed unsolvable.

Recent years saw an unprecedented growth in the number of approaches, schemes, algorithms, applications, platforms, and frameworks for DL. First, DL computations can aim at inference or training. Second, hardware platforms can vary significantly, including CPUs, GPUs, or FPGAs. Third, operators can be computed using different methods, e.g., im2col [5] or Winograd [27] in convolutions. Next, DL functionalities have been deployed in a variety of frameworks, such as TensorFlow [14] or Caffe [21]. These functionalities may incorporate many parallel and distributed optimizations, such as data, model, and pipeline parallelism. Finally, DL workloads are executed in wildly varying environments, such as mobile phones, multi-GPU clusters, or large-scale supercomputers.

This richness of the DL domain raises an important question: **How can one ensure a leveled, fair ground for comparison, competition, and benchmarking in Deep Learning?** The key issue is that, due to the complex nature of DL workloads, there is no single metric by which one DNN or hardware is objectively better than another on all counts. This is an open question and there are multiple proposed metrics (e.g., throughput, time-to-solution), especially for hardware ranking for distributed DL. Thus, it is necessary to design an abstraction that supports these and future potential metrics for DL evaluation. Several benchmarks [9, 31, 50, 2, 19, 22, 13, 31] have been designed for DL, but they are either specialized for a given aspect (e.g., single-layer performance) or perform black-box tests, which hinder verifiability and reproducibility. Since DL is converging in terms of procedures, it is possible to design a white-box abstraction that covers key functionalities of the problem, enabling *arbitrary* metric measurement and full integration of the different software stacks (see Table I) for benchmarking.

We propose **Deep500**: a white-box benchmarking infrastructure that enables fair analysis and comparison of diverse DL workloads and algorithms. Deep500 is based on the following **five pillars**: ❶ **Customizability**, ❷ **Metrics**, ❸ **Performance**, ❹ **Validation**, and ❺ **Reproducibility**. ❶ "Customizability" indicates that Deep500 enables benchmarking of arbitrary combinations of DL elements, such as various frameworks running on different platforms, and executing custom algorithms. To achieve this, we design Deep500 to be a *meta-framework* that can be straightforwardly extended to benchmark any DL code. Table I illustrates how various DL frameworks, libraries, and frontends can be integrated in Deep500 to enable easier and faster DL programming. ❷ "Metrics" indicates that Deep500 embraces a complex nature of DL that, unlike benchmarks such as Top500 [15], makes a single number such as FLOPS an insufficient measure. To this end, we propose metrics that consider the accuracy-related

System	Operators		Networks				Training			Dist. Training			
	Sta	**Cus**	**Def**	**Eag**	**Com**	**Tra**	**Dat**	**Opt**	**Cus**	**PS**	**Dec**	**Asy**	**Cus**
(L) cuDNN	🟩 ✓	🟩 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗
(L) MKL-DNN	🟩 ✓	🟩 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗
(F) TensorFlow [1]	🟩 ✓	🟩 ✓	🟩 ✓	🟩 ✓	🟩 ◐	🟩 ◐	🟩 ✓	🟩 UR	🟩 ✓	🟩 ✓	🟩 ◐	🟩 ✓	🟩 ✗
(F) Caffe, Caffe2† [21]	🟩 ✓	🟩 ◐	🟩 ✓	🟩 ✗	🟩 ✗	🟩 ✗	🟩 ✗	🟩 UR	🟩 ✗	🟩 ✗	🟩 ✓	🟩 ✗	🟩 ✗
(F) [Py]Torch† [10, 35]	🟩 ✓	🟩 ✓	🟩 ✗	🟩 ✓	🟩 ✗	🟩 ✗	🟩 ✗	🟩 ✓	🟩 ✓	🟩 ✗	🟩 ✓	🟩 ✗	🟩 ✗
(F) MXNet [6]	🟩 ✓	🟩 ◐	🟩 ✓	🟩 ✗	🟩 ✗	🟩 ✗	🟩 ✓	🟩 UR	🟩 ◐	🟩 ✓	🟩 ✗	🟩 ✓	🟩 ✗
(F) CNTK [48]	🟩 ✓	🟩 ◐	🟩 ✓	🟩 ✗	🟩 ✗	🟩 ✗	🟩 ✓	🟩 UR	🟩 ◐	🟩 ✗	🟩 ✓	🟩 ✗	🟩 ✗
(F) Theano [4]	🟩 ✓	🟩 ✓	🟩 ✓	🟩 ✓	🟩 ✓	🟩 ✓	🟩 ✗	🟩 ◐	🟩 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗
(F) Chainer[MN] [44]	🟩 ✓	🟩 ✓	🟩 ✗	🟩 ✓	🟩 ✗	🟩 ✗	🟩 ✓	🟩 ◐	🟩 ✓	🟩 ✗	🟩 ✓	🟩 ✗	🟩 ✗
(F) Darknet [38]	🟩 ✓	🟩 ✗	🟦 ✓	🟦 ✗	🟦 ✗	🟦 ✗	🟦 ✗	🟦 ◐	🟦 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗
(F) DL4j [43]	🟩 ✓	🟩 ✓	🟦 ✓	🟦 ✗	🟦 ✗	🟦 ✗	🟩 ✓	🟩 UR	🟩 ◐	🟩 ✓	🟩 ✗	🟩 ✗	🟩 ✗
(F) DSSTNE	🟩 ✓	🟩 ✗	🟦 ✓	🟦 ✗	🟦 ✗	🟦 ✗	🟦 ✗	🟦 UR	🟦 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗
(F) PaddlePaddle	🟩 ✓	🟩 ✓	🟦 ✓	🟦 ✗	🟦 ✗	🟦 ◐	🟩 ◐	🟩 UR	🟩 ◐	🟩 ✓	🟩 ✗	🟩 ✓	🟩 ◐
(F) TVM [7]	🟩 ✓	🟩 ✓	🟩 ✓	🟩 ✗	🟩 ✓	🟩 ✓	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗
(E) Keras [8]	🟩 ✓	🟩 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟦 ✗	🟦 UR	🟦 ◐	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗
(E) Horovod [42]	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟩 ✗	🟩 ✓	🟩 ✗	🟩 ✗
(E) TensorLayer [14]	🟩 ✓	🟩 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟦 ✗	🟦 UR	🟦 ✗	🟩 ✗	🟩 ◐	🟩 ✗	🟩 ✗
(E) Lasagne	🟩 ◐	🟩 ✓	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟦 ✗	🟦 UR	🟦 ◐	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗
(E) TFLearn [11]	🟩 ◐	🟩 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟦 ✓	🟦 ✗	🟦 ✗	🟥 ✗	🟥 ✗	🟥 ✗	🟥 ✗
Integration within Deep500 [This work]	🟩 ✓*‡	🟩 ✓*	🟩 ✓*‡	🟩 ✓*‡	🟩 ✓	🟩 ✓*	🟩 ✓*‡	🟩 ✓*‡	🟩 ✓*	🟩 ✓*‡	🟩 ✓*‡	🟩 ✓*‡	🟩 ✓*

TABLE I: **An overview of DL frameworks, related systems that can be integrated within Deep500, and the advantages of such integration**. Each column is a specific feature/functionality; they are explained in more detail in Background (§ II). **Sta**: Standard Operators, **Cus**: Customizable (without full recompilation), **Def**: Deferred Execution Mode, **Eag**: Eager Execution Mode (also called "define-by-run"), **Com**: Network Compilation, **Tra**: Transformable, **Dat**: Dataset Network Integration, **Opt**: Standard Optimizers, **PS**: Parameter Server, **Dec**: Decentralized, **Asy**: Asynchronous SGD, **UR**: Update Rule Optimizers, ✓, ◐, ✗: A given system does offer a given feature, offers a given feature in a limited way, or does not offer it. **(L)**: a library, **(F)**: a framework, **(E)**: a frontend. †Caffe2 and PyTorch exist as a single repository of code, but execute as two separate frameworks. * Deep500 provides an isolated modular abstraction of a given feature. ‡Deep500 provides a reference implementation. Native system support for a category of features: 🟥 none, 🟦 partial, 🟩 full.

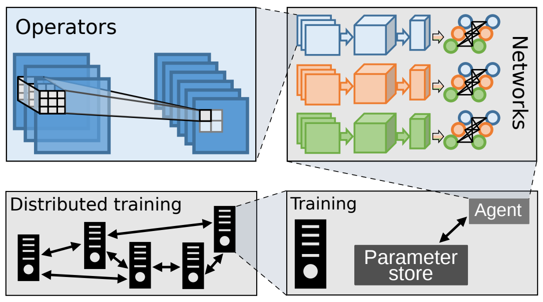


Fig. 1: Components in Deep Learning

aspects of DL (e.g., time required to ensure a specific test-set accuracy) and performance-related issues (e.g., communication volume). ❸ "Performance" means that Deep500 is the first DL benchmarking infrastructure that can be integrated with parallel and distributed DL codes. ❹ "Validation" indicates that Deep500 provides infrastructure to ensure correctness of aspects such as convergence. Finally, Deep500 embraces ❺ "Reproducibility" as specified in recent HPC initiatives [18] to help developing reproducible DL codes.

Table II compares Deep500 to other benchmarking infrastructures with respect to the offered functionalities. Deep500 is the only system that focuses on performance, accuracy, and convergence, while simultaneously offering a wide spectrum of metrics and criteria for benchmarking, enabling customizability of design, and considering a diversity of workloads.

Towards these goals, we contribute:

- the identification and analysis of challenges in high-performance reproducible benchmarking of deep learning,
- the design and implementation of the **Deep500 Benchmark**, a *meta-framework* for customizable, fast, validatable, and reproducible benchmarking of *extreme-scale* DL frameworks, applications, algorithms, and techniques,
- extensive evaluation, illustrating that Deep500 ❶ incurs negligible (<1%) overheads of benchmarking on top of tested systems, and ❷ vastly reduces development efforts to integrate and benchmark various elements of deep learning.

II. Background

We start with describing background on deep learning.

A. Deep Learning

We focus on *Deep Learning* (DL): a subclass of Machine Learning (ML) that uses Deep Neural Networks (DNNs) [3] for approximating certain complex functions. In this paper, we mostly discuss *supervised learning*, but Deep500 can be used to benchmark workloads for other tasks, such as unsupervised and reinforcement learning. A DNN is first *trained*: it is provided with various input data samples in a randomized order to minimize the difference (*loss*) between the obtained and the desired outcome (this difference is computed with some loss function ℓ). After training, a DNN is used to *infer* outcomes for given inputs.

Intuitively, a DNN is a composition of multiple functions called *operators*. Operators can range from fully-connected neural networks, through multi-dimensional convolution, to recurrent operators that maintain state. The process of evaluating a given operator for a given input data (referred to as a *sample*) is called *inference*. These operators are organized as a Directed Acyclic Graph (DAG) (Fig. 1, top right).

Formally, for an input dataset $S \subset (X \times Y) \sim \mathcal{D}$ of labeled samples (sampled from a data distribution $\mathcal{D}$), and a parametric model $f : X \to Y$ (denoted by $f(w, x)$), the goal is to minimize the *expected loss over the dataset*, i.e., find a *minimizing set of parameters* $w^* = \arg\min_w \mathbb{E}_{(x,y)} [\ell(f(w,x), y)]$, where ℓ is a certain norm function for assessing difference.

Training To minimize the expected loss, we use algorithms such as Stochastic Gradient Descent (SGD) [41] for training. In SGD, dataset elements are sampled at random in *minibatches* (data portions) of size B; usually $16 \leq B \leq 64$k [20]. In training, one iterates over the whole dataset (one such loop iteration is called an *epoch*) multiple times, and modifies minimizing parameters $w^{(t)}$ at iteration t according to the *average* gradient and possibly historical values of w. Algorithm 1 depicts such an SGD optimizer with a weight update rule U.

Algorithm 1 Minibatch Stochastic Gradient Descent [41]

```
1: for $t = 0$ to $\frac{|S|}{B} \cdot \#epochs$ do                 ▷ $|S|$: input dataset size
2:     $\vec{x}, \vec{y} \leftarrow$ Sample $B$ elements from $S$     ▷ $\vec{x}, \vec{y}$: samples of input data
3:     $w_{mb} \leftarrow w^{(t)}$           ▷ Load minimizing parameters from iteration $t$
4:     $\vec{z} \leftarrow \ell(w_{mb}, \vec{x}, \vec{y})$             ▷ Inference; $\vec{z}$ is a full minibatch.
5:     $g_{mb} \leftarrow \frac{1}{B}\sum_{i=0}^{B} \nabla\ell(w_{mb}, \vec{z}_i)$   ▷ Backpropagation; $\vec{z}_i$ is a sample.
6:     $\Delta w \leftarrow U(g_{mb}, w^{(0,\ldots,t)}, t)$                ▷ Apply an update rule
7:     $w^{(t+1)} \leftarrow w_{mb} + \Delta w$        ▷ Store updated minimizing parameters
8: end for
```

Distributed Training When distributing training among compute nodes, it is common to use data parallelism, i.e., partitioning across minibatches. The gradient average (Algorithm 1, line 5), necessary for descent, becomes a parallel reduction that is performed collectively (*allreduce* in MPI nomenclature). Data-parallel distributed training can be implemented in one of two general approaches: *decentralized*, using an allreduce operation; or *centralized*, where a (possibly sharded) "parameter server" (PS) governs optimization [12] by receiving individual gradients and broadcasting back new parameters (Algorithm 1, lines 3 and 5–7). Deep500 enables all these distributed schemes while vastly reducing development effort.

B. Frameworks

Many frameworks for training and inference exist; see Table I. According to GitHub [16], the three most popular DL frameworks are TensorFlow [1], Caffe [21], and PyTorch [10]. We use these three frameworks (except for Caffe, which we replace with Caffe2, an improved Caffe version written by the same authors) as use cases to demonstrate the flexibility of Deep500. Now, the frameworks can differ vastly, ranging from how operators are implemented and how extensible (*Customizable*) the frameworks are; to how DNNs are evaluated and trained. Some frameworks compute operators on-the-fly, i.e., as they are called (*Eager Execution*). Others construct a graph in advance (*Deferred Execution*) and modify it for high-level optimizations (*Transformable*), ahead-of-time *Network Compilation*, or employ *Dataset Integration* by adding data loading operators to the graph, enabling automatic pipelining of samples to accelerators and distributed storage integration. As for training, most frameworks support the aforementioned

Benchmark	Focus			Metrics									Criteria			Customizability			DL Workloads						Remarks
	Perf	Con	Acc	Tim	Cos	Ene	Util	Mem	Tput	Brk	Sca	Com	TTA	FTA	Lat	Clo	Ope	Inf	Ops	Img	Obj	Spe	Txt	RL	
DeepBench [40]	✓	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✓	✗	✓	✓	✗	✗	✗	✗	✗	Ops: Conv., GEMM, RNN, Allreduce
TBD [50]	✓	✗	✗	✗	✗	✗	✓	✓	✓	✗	✗	✗	✗	✗	✓	✓	✗	✗	✗	✓	✓	✓	✓	✓	+GANs
Fathom [2]	✓	✗	✗	✓	✗	✗	✗	✗	✓	✓	✓	✗	✗	✗	✓	✓	✗	✗	✗	✓	✗	✓	✓	✓	+Auto-encoders
DLBS [19]	✓	✗	✗	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗	✓	✗	✗	✗	✗	
DAWNBench [9]	✓	✓	✗	✓	✓	✗	✗	✗	✗	✗	✗	✗	✓	✗	✓	✗	✓	✗	✗	✓	✗	✗	✓	✗	
Kaggle [22]	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✓	✗	✗	✓	✓	✓	✓	✓	Varying workloads
ImageNet [13]	✗	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✓	✗	✗	✓	✗	✗	✓	✓	✗	✗	✗	
MLPerf [31]	✓	✓	✓	✓	✓	✓	✗	✗	✗	✗	✗	✗	✓	✓	✓	✓	✓	✗	✗	✓	✓	✓	✓	✓	
Deep500	✓	✓	✓	✓	✓	◐	✓	◐	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	◐	◐	◐	◐	

TABLE II: **An overview of available DL benchmarks**, focusing on the offered functionalities. **Perf**: Performance, **Con**: Convergence, **Acc**: Accuracy, **Tim**: Time, **Cos**: Cost, **Ene**: Energy, **Util**: Utilization, **Mem**: Memory Footprint, **Tput**: Throughput (Samples per Second), **Brk**: Timing Breakdown, **Sca**: Strong Scaling, **Com**: Communication and Load Balancing, **TTA**: Time to Accuracy, **FTA**: Final Test Accuracy, **Lat**: Latency (Inference), **Clo**: Closed (Fixed) Model Contests, **Ope**: Open Model Contests, **Inf**: Fixed Infrastructure for Benchmarking, **Ops**: Operator Benchmarks, **Img**: Image Processing, **Obj**: Object Detection and Localization, **Spe**: Speech Recognition, **Txt**: Text Processing and Machine Translation, **RL**: Reinforcement Learning Problems, ✓: A given benchmark does offer the feature. ◐: Planned benchmark feature. ✗: A given benchmark does not offer the feature.

Update Rule SGD, but some also provide other optimizers, or enable implementing arbitrary algorithms.

A complete DL framework provides: ❶ operator representation and implementations, ❷ network definition (connections between operators), ❸ schemes for loading datasets and for data augmentation (i.e., increasing variation in samples by perturbing data), ❹ inference and gradient computation, ❺ stochastic optimization (training), and ❻ distributed optimization and communication infrastructure. Elements ❶–❷ are partially standardized by initiatives such as ONNX [33] and NNEF [23] that define portable (up to framework limitations) file formats. Other elements are not standardized at all. Moreover, frameworks do not provide standardized metrics, such as accuracy and performance, which are absolutely necessary for scientific computing purposes, high-performance computing, and reproducibility. Table I illustrates the various DL frameworks, libraries, and frontends, and how Deep500 integrates them through its meta-framework design. These systems can then be analyzed using a set of carefully selected metrics.

C. Benchmarks

There exist preliminary efforts to benchmark DL and general ML. Table II analyzes the functionalities of these efforts. In general, only Deep500 focuses on performance, accuracy, *and* convergence, considering the five challenges of large-scale DL benchmarking discussed in the introduction.

D. Data Model and Format

We use the Open Neural Network Exchange (ONNX) [33] format to store DNNs reproducibly. ONNX provides a binary file format capable of describing an arbitrary DAG and standardizes a list of 118 common operators (as of version 1.3.0) used in DL and in general ML. Many popular frameworks provide and are actively improving interoperability with ONNX. Thus, we select ONNX as the basis of data format for Deep500. To use ONNX for reproducible training and to enable extensibility, we augment the ONNX with additional built-in operators and with support for user-defined operators.

III. Benchmarking Deep Learning: Challenges

We analyze challenges in benchmarking large-scale DL.

A. Motivation

We first describe motivating example use cases.

```
tf.layers.conv2d(
  inputs, filters,
  kernel_size, strides, padding,
  data_format, dilation_rate,
  activation, use_bias,
  kernel_initializer, bias_initializer,
  kernel_constraint, bias_constraint,
  kernel_regularizer,
  activity_regularizer,
  trainable, name, reuse
)
```

Listing 1: **TensorFlow: 19 parameters** to init 2D convolution.

```
cntk.layers.Convolution2D(
  filter_shape, num_filters,
  activation, init,
  pad, strides, bias, init_bias,
  reduction_rank, name
)
```

Listing 2: **CNTK: 10 parameters** to init 2D convolution.

Use Case 1 We observe that different frameworks come with significant differences between basic functionalities. For example, we present the initialization of 2D convolution in TensorFlow and CNTK in Listings 1–2. TensorFlow uses 19 parameters while CNTK needs 10. Thus, comparing fairly both frameworks in metrics such as runtime or accuracy is unclear ("Which parameters correspond to each other?"), and for some operators *impossible* due to implementation differences. Among the factors that vary between frameworks is data layout, which is unclear in the CNTK example. For the case of performance, we use the Adam SGD optimizer [24] as a second example. As TensorFlow provides general tensor operators (using the Eigen linear algebra library), it requires sequentially executing several short operations on the GPU (e.g. subtraction, division) to compute the optimizer update. Conversely, Caffe2 implements a specific "Adam" operator that performs the entire update using a single GPU kernel, drastically reducing invocation and GPU scheduling overheads. This *operation fusion* in Caffe2 is a common optimization technique, and Deep500 enables straightforward comparison of such TensorFlow and Caffe2 instances, both in an isolated environment and as an integrated part of training over existing datasets. In general, *this use case calls for an infrastructure that enables straightforward invocation of existing and custom implementations, in order to enable simple, maintainable comparison and benchmarking.*

Use Case 2 Constructing a complex DNN may be a significant time investment. Yet, today's DL frameworks do not enable a straightforward use of a network developed in a different framework. For example, networks designed in TensorFlow cannot easily be used in Caffe2 (e.g., due to the aforementioned differences in operators). *One would welcome a system that facilitates porting between different DNN formats, in order to develop DNN-related techniques independently, as well as reuse networks across frameworks.*

Use Case 3 No single framework provides all the required functionalities. Thus, one may be interested in extending a selected framework. Unfortunately, this is usually difficult and time-consuming. For example, implementing a second-order optimization, such as Stochastic L-BFGS [32], requires a training loop that is vastly different than that in Algorithm 1, which is the basis of many frameworks. Now, while some frameworks (e.g., TensorFlow) enable the creation of custom training loops (e.g., using optimized tensor operations), the CNTK `Learner` extension does not enable this straightforwardly. *An infrastructure for combining the best of different DL frameworks would be advantageous in such cases.*

Use Case 4 Many DL tools are distributed. In such cases, *to ensure scalability and high performance, one needs a system that can benchmark and analyze aspects related to distributed processing, such as the amount of communicated data.*

Others There are many other situations requiring a standard benchmarking platform for DL. They can be pictured by the following example questions: "What is the reduction in communication over the network, when a certain compression scheme is applied in training?", "How to illustrate performance and power advantages of using a novel ASIC for a particular class of DL workloads?", "How fast and accurate is a certain provably optimal operator?", "What is the advantage of using FPGAs for DL training?", "For a given DL workload, which one of the available machines will perform best?".

B. *Challenge 1: Customizability*

The first challenge emerging from the above examples is *customizability*: the ability to seamlessly and effortlessly combine different features of different frameworks and still be able to provide fair analysis of their performance and accuracy.

Deep500 enables customizability and interoperability with various DL codes through its *meta-framework* design. By incorporating both Python and C/C++ capabilities, we provide an infrastructure that can be straightforwardly extended to virtually any DL framework or arbitrary operator code.

C. *Challenge 2: Metrics*

Another challenge lies in a proper selection of metrics. On one hand, some metrics may simply be too detailed, for example the number of cache misses in 2D convolution implemented in TensorFlow or Caffe2. Due to the sheer complexity of such frameworks, this metric would probably not provide useful insights in potential performance regressions. On the other hand, other metrics may be too generic, for example simple runtime does not offer any meaningful details and does not relate to accuracy. Thus, one must select metrics that find the right balance between accuracy and genericness.

In Deep500, we offer carefully selected metrics, considering performance, correctness, and convergence in shared- as well as distributed-memory environments. ❶ Some metrics can test the performance of both the whole computation and fine-grained elements, for example *latency* or *overhead*. ❷ Others, such as *accuracy* or *bias*, assess the quality of a given algorithm, its convergence, and its generalization towards previously-unseen data. ❸ We also combine performance and accuracy (*time-to-accuracy*) to analyze the

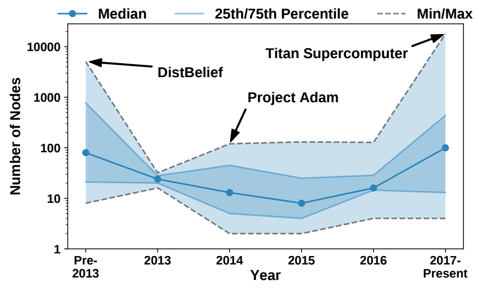


Fig. 2: Statistics of using compute nodes in distributed DL [3]

tradeoffs. ❹ Finally, we propose metrics for the distributed part of DL codes: *communication volume* and *I/O latency*.

D. *Challenge 3: Performance and Scalability*

Another unaddressed challenge is the design of distributed benchmarking of DL to ensure high performance and scalability. As DL datasets and training complexity continue to grow, large-scale distributed training is becoming an essential DL component [3] (Fig. 2). Every top competitor in DAWNBench [9] uses multiple multi-GPU nodes, and recently the entire Titan supercomputer (18,000 nodes) was used for a full 24 hours to perform distributed DL via meta-optimization (i.e., where the DNN structure may change) [47]. To deliver high-quality scalable distributed DL codes, one must be able to debug scalability issues, simultaneously preventing negative performance impact coming from the benchmarking infrastructure. Moreover, we need proper techniques to understand such scalability bugs in the context of DL workloads. As we show later (§ V), **the Deep500 benchmarking infrastructure potentially scales to thousands of cores and incurs negligible overheads over native performance**.

E. *Challenge 4: Validation*

A benchmarking infrastructure for DL must allow to *validate* results with respect to several criteria. As we discuss in § V, **Deep500 offers validation of convergence, correctness, accuracy, and performance**. Validation comes in the form of $\ell_1, \ell_2, \ell_\infty$ norms, but also in forms of heatmaps, to highlight regions of interest, or repeatability via a map of output variance. In addition, we provide gradient validation through numerical differentiation with similar metrics. We also test optimizers in similar ways, making sure that optimization trajectories do not diverge given the same inputs.

F. *Challenge 5: Reproducibility*

The final challenge in distributed DL benchmarking is the ability to *reproduce* or at least *interpret* [18] results. **In Deep500, we ensure these properties by using our interfaces and several careful design decisions**, described in § IV.

IV. Design and Implementation of Deep500

We now describe the purpose and design of Deep500, addressing the above-discussed challenges. *The core enabler* in Deep500 is *the modular design* that groups all the required functionalities into four *levels*: ❶ "Operators", ❷ "Network Processing", ❸ "Training", and ❹ "Distributed Training". Each level provides relevant abstractions, interfaces, reference implementations, validation procedures, and metrics. We illus-

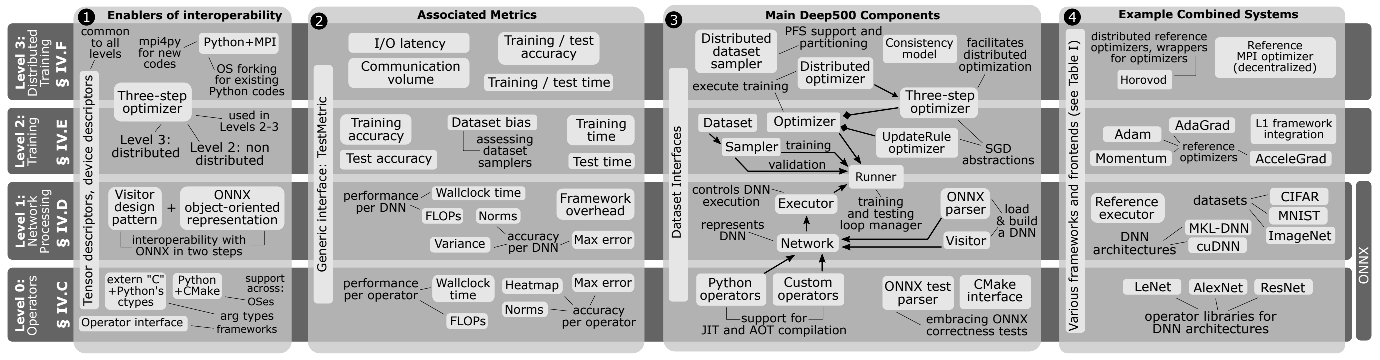


Fig. 3: The design of Deep500.

trate levels and their relationships in Fig. 1; the full design of the Deep500 meta-framework is shown in Fig. 3.

A. Intended Purpose and Roles

The Deep500 meta-framework is a benchmarking environment, and as such it is not meant to be a DL framework that provides optimized implementations of its own. Rather, Deep500 assumes high-performance frameworks exist. By abstracting the high-level aspects of DL (e.g., data loading) in a platform-agnostic manner, Deep500 enables the measurement and development of various metrics (performance, accuracy) in the different contexts of DL and distributed DL.

By taking the white-box approach, the user roles that Deep500 enables can be of a benchmark evaluator, or of an experimental scientist. In the former, one might use Deep500 and the various built-in metrics to choose hardware (or software) that performs best given a target workload. The latter role can use metrics and automatic integration with existing frameworks in order to empirically evaluate new operators, training algorithms, or communication schemes for DL. Since Deep500 provides reference code for nearly every concept, new methods can be validated against existing verified (yet slow) implementations.

Deep500 complements existing DL frameworks in assisting user efforts. DL frameworks and other integrations provide baselines for both performance and convergence, and all a user has to do is implement their own part, reusing the rest of the existing components. In the rest of this paper, we show that such extensions to the algorithms and metrics are simple to implement in a concise manner, and that the incurred overhead on performance is relatively low.

B. Common Components

We first describe elements common to all four levels.

Metrics Answering Use Cases 1, 3 and the resulting Challenge 2, Deep500 provides a general-purpose interface *to access and use metrics*. In particular, the `TestMetric` class contains methods to obtain the number of re-runs needed for a selected measurement (e.g., used in ensuring numerical stability), to make or summarize a measurement, and to generate a selected result (e.g., a plot file or a series of numbers).

Interoperability: DNN Format To read and write DNNs, we use the ONNX format, and thus need to interoperate with its Python package. As we show in more detail later (§ IV-D), Deep500 *converts the ONNX format to an object-oriented notation for easier interoperability*. To support this, we autogenerate code from the ONNX operator definition files. We also extend ONNX with new operations for computing loss functions and optimization, as well as distributed optimization. Finally, we embrace the ONNX correctness tests. This addresses Use Case 2 and Challenge 1: customizability and interoperability with various data formats. By using ONNX operators, we also address Use Case 1: Since ONNX standardizes a wide range of ML-related operators, Deep500 can be used to construct conforming DNNs between frameworks, such as TensorFlow and CNTK in the example.

Interoperability: Datasets and Networks Deep500 can download the MNIST [28], Fashion-MNIST [46], and CIFAR-10/100 [26] datasets on demand, as well as parse ImageNet [13]. Similarly, it facilitates access to DNN architectures (as ONNX files) for LeNet [29], ResNet [17] with varying depths, and Wide ResNet [49]. Facilitating access to various datasets enhances DL programmability and addresses Use Case 2.

Interoperability: Frameworks and Platforms Deep500 uses its own *descriptors* for *tensors* and *devices* to enable interoperability with frameworks and platforms (Use Case 1). Tensor descriptors (`tensordesc`), which are also C Application Binary Interface (ABI) compatible, extend the types given in ONNX by describing the data type in more detail (e.g., allowing bitsets, or including data layout types). These extensions enable each implemented framework to convert types back and forth from Deep500 tensor descriptors. Additionally, we provide extensible Device Descriptors (CPU, GPU, FPGA, etc.), for example used to identify the most advantageous compute device for a specific operator computation, addressing several use cases from "Others".

Interoperability: Distributed Training To facilitate distributed training and thus address Use Case 4 and Challenge 3 (performance and scalability), we use a Python interface with MPI, `mpi4py`, *to link with MPI*, and use OS forking to turn an existing Python application into an MPI-capable one, all while keeping the proper distributed DL semantics w.r.t. dataset sampling and distributed storage, as well as to the DNN model.

C. Level 0: Operators

Level 0 enables implementing, computing, and benchmarking individual operators, which are the building blocks of DNNs. An operator's functionality is general, and spans DNN layers (e.g., convolution) as well as training-related operations (e.g., distributed gradient accumulation).

Interfaces Deep500 Level 0 allows to integrate new custom operators with real datasets, networks, or frameworks, without having to implement other operators. For this, we provide the `CustomOperator` interface, available in Python (addressing high-level ML researchers and experimentation) or in C++ (addressing high-performance implementations). `CustomOperator` provides two functions, `forward(inputs)` and `backward(grad_inputs, fwd_inputs, fwd_outputs)`. To support the integration of arbitrary C++ code in existing frameworks and abstractions, we provide a runtime *compilation interface*. The compiler interface is a simple wrapper around CMake, which includes stub templates for each implemented DL framework. These templates include the custom C++ code to create a framework-compatible interface of an operator, which can be seamlessly used in the frameworks, even without the rest of Deep500. Using this abstraction, Deep500 supports both Just-In-Time (JIT) or Ahead-Of-Time (AOT) compilation of operators, enabling flexible benchmarking of high-performance code.

Example Use Case Listings 3–4 illustrate Deep500's interoperability with frameworks: they contain an example definition of a custom *median-pooling* operator in C++ and its straightforward registration as well as compilation for PyTorch.

```
template<typename T>
class MedianPooling : public deep500::CustomOperator {
public:
  void forward(const T *input, T *output) { /* Inference code */ }
  void backward(const T *nextop_grad, const T *fwd_input_tensor,
                const T *fwd_output_tensor, T *input_tensor_grad) {
    /* Backpropagation code */ }
};                                                    Operator definition

D500_REGISTER_OP(MedianPooling<DTYPE>); //Register a custom operator
D500_EXPORTED void *create_new_op(deep500::tensor_t *input_descs,
   int ninputs, deep500::tensor_t *output_descs, int noutputs) {
  //Create the actual operator object.
  return new MedianPooling<DTYPE>(/* ... */);
}                                                   Operator registration
```

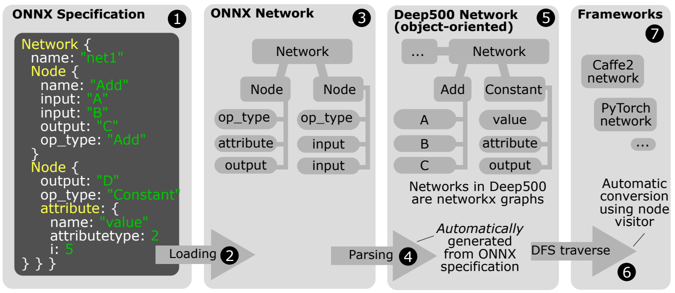
Listing 3: (§ IV-C) Defining a custom operator in Deep500 with C++.

```
import deep500 as d5
from deep500.frameworks import pytorch as d5pt
# Create an operator descriptor for compilation  Operator compilation
opdesc = d5.compile_custom_cppop('MedianPooling', 'mp.cpp',
   [d5.tensordesc(tf.float32, [256, 256])], # Input tensor shapes
   [d5.tensordesc(tf.float32, [128, 128])], # Output tensor shapes
   additional_definitions={'DTYPE': 'float'})
mycppop = d5pt.custom_op(opdesc) # Compiles operator for framework
```

Listing 4: (§ IV-C) Using a custom operator in Deep500 with Python.

Interoperability When an operator has more than one input or output, supporting a high-performance C++ implementation is complicated. Arrays incur overheads due to dynamic memory management and the code becomes less readable. To solve this issue, Deep500 uses *variable* arguments in the C++ interface, then exports a C *ABI-compatible* function (`extern "C"`, containing no defined arguments), and uses Python's dynamic library invocation interface (`ctypes`) to call the function *with unpacked arguments*. We also automatically convert native tensor types into C pointers, so that *any* operator is implemented *only once* for *all* frameworks.

To support custom C++ operators across frameworks *and* OSes (Deep500 supports Linux, Windows, and Mac OS), we use CMake. To enable JIT/AOT compilation of operations, we



```
Network {
  name: "net1"
  Node {
    name: "Add"
    input: "A"
    input: "B"
    output: "C"
    op_type: "Add"
  }
  Node {
    output: "D"
    op_type: "Constant"
    attribute: {
      name: "value"
      attributetype: 2
      i: 5
} } }
```

Fig. 4: An automatic transformation of an ONNX network to a Deep500 network.

wrap CMake process with a cross-platform Python interface that can accept multiple files for compilation.

Finally, our operator interface allows to convert native operators from frameworks into custom Deep500 operators for use in, e.g., other frameworks, see Listing 5.

```
import tensorflow as tf
from deep500.frameworks import tensorflow as d5tf
# ...Define A, B and C as TensorFlow tensors      Operator customization
op = d5tf.custom_op_from_native(tf.matmul,
    [d5tf.desc_from_tensor(A), d5tf.desc_from_tensor(B)],
    [d5tf.desc_from_tensor(C)])
# Deep500 can now use the operator interface to test `op`
```

Listing 5: (§ IV-C) Using a native operator as a custom one in Deep500 (Python).

Provided Implementations Deep500 provides reference implementations of all operators required for the DNNs in § IV-B.

Metrics One family of the associated metrics are performance per operator, for example wallclock time, FLOPs, or consumed energy. Another family are accuracy per operator: norms (e.g., $\ell_1, \ell_2, \ell_\infty$), variance in output, and 2D/3D heatmaps.

Validation Validation is enabled by two functions. First, `test_forward` tests operator correctness and performance. Second, `test_gradient` uses numerical differentiation (Jacobian matrix evaluation using finite differences) to provide automatic gradient checking, as well as measure the performance of the `backward` method.

D. Level 1: Network Processing

Level 1 is dedicated to the construction, modification, evaluation, and backpropagation of entire neural networks. Deep500 separates the network abstraction from file formats, operators, and training to enable a fair and extensible infrastructure upon which researchers can build their own graph transformations to optimize between operators.

Interfaces To represent a DNN, we use two classes: `Network` and `GraphExecutor`. `Network` defines the network structure and exposes a standard graph API that allows to add and `remove` nodes/edges, `fetch` node data contents, `feed` nodes with new values, and others. `GraphExecutor` controls the DNN execution. It provides two functions that enable inference and possibly backpropagation: `inference`, and `inference_and_backprop`.

To enable fine-grained measurements and support early exits, graph executors must be able to invoke certain `Events` at the right time. Events are user-specified "hooks" that are called at certain points during complex actions such as backpropagation and training. An example of an event "hook" is providing an early stopping condition. To enable benchmarking events with a selected metric, the same metric class can extend both the `TestMetric` and `Event` classes.

Interoperability While `Networks` can be created manually (node by node), Deep500 also provides a convenient interface to construct networks from ONNX files. This entails a non-trivial processing scheme, depicted in Fig. 4, in which an ONNX graph is first transformed to an intermediate, object-oriented representation. Deep500 then uses the *Visitor design pattern* to invoke `Network` construction by calling the right functions. An example construction is in Listing 6.

```python
class TensorflowVisitor(d5.OnnxBaseVisitor):
  # ... other definitions ...
  def visit_dropout(self, op: d5.ops.Dropout, network: TFNetwork):
    X = network.fetch_internal_tensor(op.i_data)
    ratio = op.ratio.get_value() if op.ratio else 0.5
    Y = tf.nn.dropout(X, ratio)
    network.feed_internal_tensor(op.o_output, Y)         Dropout visitor

  def visit_sub(self, op: d5.ops.Sub, network: TFNetwork):
    A, B = network.fetch_internal_tensors([op.i_A, op.i_B])
    C = tf.subtract(A, B)
    network.feed_internal_tensor(op.o_C, C)         Subtraction visitor

  def visit_mul(self, op: d5.ops.Mul, network: TFNetwork):
    A, B = network.fetch_internal_tensors([op.i_A, op.i_B])
    C = tf.multiply(A, B)
    network.feed_internal_tensor(op.o_C, C)      Multiplication visitor
```

Listing 6: An example DNN construction using the TensorFlow ONNX visitor.

Provided Implementations Deep500 implements a reference `Network` using the `networkx` Python graph library. `GraphExecutor` is implemented by a topological graph sort.

Metrics We adapt the metrics from Level 0 to full DNN execution. We also add the `FrameworkOverhead` metric, which measures the overall time for inference and backpropagation and compares it with the sum of running times of individual operators. This evaluates the impact from framework and hardware management (e.g., GPU kernel invocation latency).

Validation To validate the accuracy and performance of `Network` and `GraphExecutor`, we provide two functions: `test_executor`, and `test_executor_backprop` for inference and backpropagation, respectively.

E. Level 2: Training

Level 2 of Deep500 implements DNN training.

Interfaces The main Level 2 interfaces are `DatasetSampler` and `Optimizer`. First, `DatasetSampler` provides minibatches by sampling a given dataset, and can be extended to test different sampling schemes. For more performance, samplers can be implemented as custom operators in C++ and plugged into a DNN `Network` as native operators. Existing native sampler operators, such as `tf.Dataset` in TensorFlow, can also seamlessly be used. The second main interface, `Optimizer`, uses a `GraphExecutor` and a `DatasetSampler`, and can run any code as the training procedure. We provide two abstractions of SGD optimizers: `UpdateRuleOptimizer`, which runs an update rule akin to U in Algorithm 1, and `ThreeStepOptimizer`, a novel abstraction that facilitates *distributed* optimization. To facilitate automatic distribution of optimization, we divide `optimizer`'s execution into three steps: ❶ input sampling (Algorithm 1, line 2), ❷ adjusting parameters prior to inference (line 3), and ❸ applying an update rule (line 6).

Interoperability Implementing new optimizers in DL frameworks, especially ones that do not conform to simple update rules, is a notoriously hard task. This hinders testing new algorithms with good convergence guarantees [30] or non-SGD methods such as second-order optimization. Additionally, interfacing with datasets is not standardized across frameworks, with varying data augmentation methods, and minibatch sampling being hardcoded into frameworks. Deep500 alleviates these issues with the `ThreeStepOptimizer` interface.

Example Use Case Listing 7 illustrates the *full* implementation of AcceleGrad, a state-of-the-art DL optimizer [30], using Deep500's `ThreeStepOptimizer`. It is apparent that the code retains its algorithmic form.

```python
class AcceleGradOptimizer(d5.ThreeStepOptimizer):
  def new_input(self):                                          New input

    self.t = self.t + 1
    self.alpha_t = 1 if 0 <= self.t <= 2 else 1 / 4 * (self.t + 1)
    self.tau_t = 1 / self.alpha_t

  def prepare_param(self, param_name):                  Adjust parameters

    param = self.executor.network.fetch_tensors([param_name])[0]
    if not self.init:
      self.y[param_name] = param
      self.z[param_name] = param
      self.squares[param_name] = 0
    y = self.y[param_name]
    z = self.z[param_name]
    new_param = self.tau_t * z + (1 - self.tau_t) * y
    self.executor.network.feed_tensor(param_name, new_param)

  def update_rule(self, grad, old_param, param_name):
    squared_grad = self.squares[param_name]
    squared_grad += self.alpha_t ** 2 * np.linalg.norm(grad) ** 2
    eta_t = 2 * self.D / np.sqrt(self.G ** 2 + squared_grad)
    z_t = self.z[param_name]
    z_t2 = z_t - self.alpha_t * eta_t * grad
    y_t2 = old_param - eta_t * grad
    self.z[param_name] = z_t2
    self.y[param_name] = y_t2
    self.squares[param_name] = squared_grad
    adjusted_lr = self.lr / (self.eps + np.sqrt(squared_grad))
    self.init = False

    return old_param - adjusted_lr * grad                     Update rule
```

Listing 7: Implementation of AcceleGrad in Deep500.

Provided Implementations Deep500 provides many popular optimizers written in Python, such as Gradient Descent with learning rate schedule, momentum, Adam [24], and AdaGrad.

Metrics Deep500 provides two main metrics in Level 2: `TrainingAccuracy` (measures the training accuracy at every kth step) and `TestAccuracy` (measures the test accuracy at every kth epoch). Additionally, dataset samplers can be tested *individually* by running `test_sampler` with the `DatasetBias` metric, which collects a histogram of sampled elements w.r.t. corresponding labels.

Validation First, `test_optimizer` verifies the performance and correctness of *one step* of the optimizer (ensuring that an optimizer trajectory does not diverge from the Deep500 one). Second, `test_training` tests the convergence, performance, and the related tradeoff of the *overall* training.

F. Level 3: Distributed Training

A *cornerstone feature* of Deep500 is that *it distributes* DNN training *with virtually no effort from an API user*. The core enablers are the two class interfaces from Level 2: an update-rule optimizer and a three-step optimizer. The distributed MPI-based optimizer uses these classes to distribute new gradients and parameters before or after executing update rules.

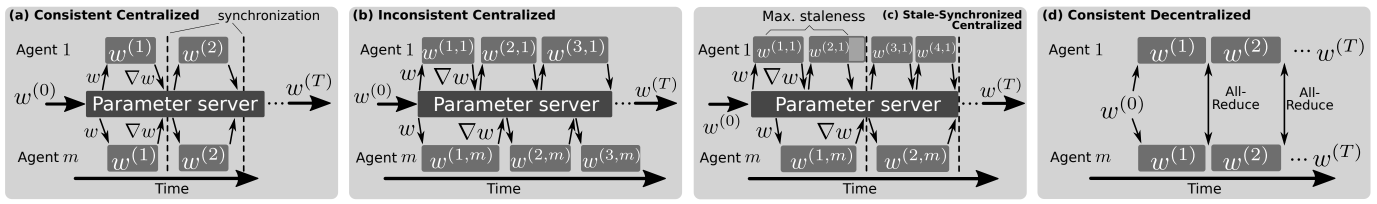

(a) Consistent Centralized
synchronization
Agent 1
Agent m
Parameter server
Time
(b) Inconsistent Centralized
Max. staleness
(c) Stale-Synchronized Centralized
(d) Consistent Decentralized
All-Reduce
All-Reduce

Fig. 5: Examples of distributed deep learning schemes ($w^{(x,m)}$ are minimizing parameters in iteration x belonging to agent m; see § II-A for more details).

Interfaces We provide two interfaces: `DistributedSampler` and `DistributedOptimizer`. The former refers to a potentially distributed data store. The latter refers to a specific subclass of `Optimizers` from Level 2, which support distributed communication. Still, there are no pre-defined constraints on *how* and *when* communication should occur, so this could potentially be a gradient-free optimizer, e.g., employing Genetic Algorithms [36, 47]. The three-step and update-rule optimizers from Level 2 also extend `DistributedOptimizer`, so implementing a custom optimizer based on these methods *automatically grants distribution capabilities*.

Example Use Case Testing for cluster-wide performance of different communication and parameter consistency schemes are notoriously hard tasks. In most codes, they require an almost total re-write of the program logic, depend on additional libraries (sometimes entailing *framework recompilation*), and not supported on all platforms, especially supercomputers. Due to the modularity of Deep500, the task is a matter of wrapping an optimizer with the right distributed scheme (Listing 8).

```python
gexec = # ... (the definition uses previous levels)
opt = d5ref.AdamOptimizer(gexec)
ds = d5ref.ShuffleDistributedSampler(dataset, batch_size)
# Collect metrics for different training schemes and topologies
ref = d5.test_training(d5ref.ConsistentDecentralized(opt), ds)
ps = d5.test_training(d5ref.ConsistentCentralized(opt), ds)
asgd = d5.test_training(d5ref.InconsistentCentralized(opt), ds)
hvd = d5.test_training(d5tf.HorovodDistributedOptimizer(opt), ds)
```

Listing 8: Testing cluster-wide performance of distributed training in Deep500.

Interoperability Deep500 also facilitates the construction of new communication methods, topologies, and interfaces. None of these features are natively supported by frameworks. For example, modifying a DNN graph to create pipeline parallelism across processes is impossible automatically in *any* of the frameworks, but can straightforwardly be done in Deep500.

Provided Implementations Deep500 implements different distributed SGD variants. This includes centralized and decentralized optimization (§ II-A), a variant with a globally-consistent model (synchronous SGD) [3], inconsistent model (asynchronous SGD, e.g., HOGWILD [37]), and stale-synchronous SGD, which enables inconsistency in the parameters up to a certain delay. Fig. 5 illustrates possible timelines of each method. These methods are, of course, compatible with all frameworks, as they use the MPI library separately. Listing 9 illustrates the achieved compatibility using the *full* implementation of the Consistent Decentralized optimizer.

As opposed to native distributed optimization, using reference implementations in conjunction with a GPU incurs a synchronous GPU-to-host copy prior to communicating, and vice versa. This could be further improved by custom C++ operators that implement a specialized `forward_cuda` method, e.g., using CUDA-aware MPI or GPUDirect.

```python
class ConsistentDecentralized(DistributedOptimizer):
  # self.base_optimizer is a `ThreeStepOptimizer` object
  def train(self, inputs):
    self.base_optimizer.new_input()
    for param in self.network.get_params():
      self.base_optimizer.prepare_param(param)
    output = self.executor.inference_and_backprop(inputs)
    gradients = self.network.gradient()
    for pname, grad_name in gradients:
      param, grad = self.network.fetch_tensors([pname, grad_name])
      grad = self.communication.sync_all(grad)
      param = self.base_optimizer.update_rule(grad, param, pname)
      self.network.feed_tensor(pname, param)
    return output
```

Listing 9: Consistent Decentralized reference optimizer in Deep500.

Metrics We provide two metrics: `CommunicationVolume` and `DatasetLatency`.

Validation We reuse the two validation functions from Level 2: `test_optimizer` and `test_training`. However, instead of an `Optimizer` and a `DatasetSampler`, we feed `DistributedOptimizer` and `DistributedSampler` classes.

V. Evaluation

Our key goal in this section is to show that **Deep500 enables detailed, accurate, and customizable benchmarking of DL codes while incurring negligible overheads**.

A. Methodology, Setup, Parameters

Neural Networks Operator dimensions and types for Level 0 tests were collected from the DeepBench [40] low-level benchmark. For convergence tests, we use ResNet-18 and 50 [17]. We use the small datasets MNIST [28] and CIFAR-10 [26], as well as the large-scale Imagenet [13] dataset, where the latter uses JPEG files packed in the `TFRecord` file format.

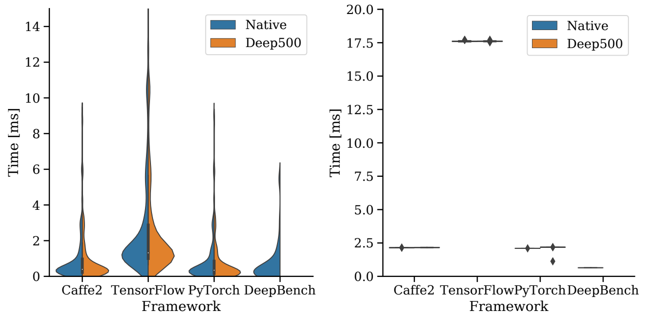
Experimental Setup and Architectures We use the CSCS Piz Daint supercomputer. Each XC50 compute node contains a 12-core HT-enabled Intel Xeon E5-2690 CPU with 64 GiB RAM, and one NVIDIA Tesla P100 GPU. The nodes communicate using Cray's Aries interconnect.

Evaluation Methodology To gather data for the non-distributed experiments (Levels 0–2), we run them 30 times and report median results and nonparametric 95% confidence intervals. We use 32-bit (single precision) floating point values for all DNN parameters and errors.

In all following benchmarks, Deep500 incurs certain overheads caused by additional data copying while conducting measurements and recording the outcomes. We expect that — as with any other benchmarking infrastructure — Deep500 users would switch off unnecessary benchmarking metrics and instrumentation for production runs and other performance-critical scenarios.

B. Level 0: Operators

We first investigate *performance and accuracy of operators* implemented with Deep500, PyTorch, TensorFlow, and Caffe2. We also consider NVIDIA *native results* obtained


(a) Convolution Performance (left: violin plot of all kernels, right: box plot of size $N = 16$, $C = 3$, $H = W = 224$, filter size 3×3).

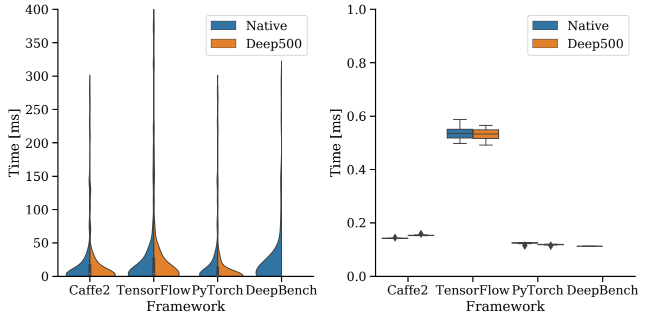

(b) Matrix Multiplication Performance (left: violin plot of all kernels, right: box plot of size $M = K = 2560$, $N = 64$).

Fig. 6: **Analysis of Deep500 Level 0:** Performance of operators implemented with Deep500 and selected frameworks, together with the DeepBench baseline.

with DeepBench. DeepBench provides 160 different matrix multiplication sizes and 94 convolution dimensions, typically found in DL workloads. We aggregate the results and present the running time distribution and accuracy of each framework.

We seek to show that **Deep500's Level 0 is reliable and fast.** Indeed, Fig. 6 shows that for all frameworks, *Deep500 operator runtime differs negligibly (within CIs) from the native frameworks*. Moreover, DeepBench can be used as a baseline for operator runtime (assuming all frameworks are implemented over the same low-level libraries, such as cuDNN), as it only calls a given kernel and outputs the resulting GPU runtime. This is not the case in other frameworks, as they contain management overhead and additional actions, such as copying tensors.

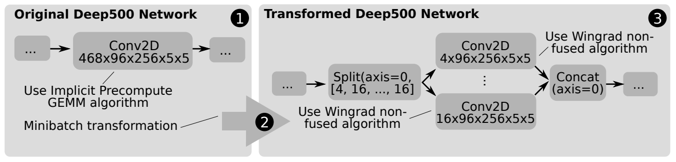
Since convolution and matrix multiplications vary widely in size and runtime, we choose two common problem sizes from DeepBench and present the results in the right-hand side of Figures 6a and 6b. The figures show that DeepBench is indeed faster than all frameworks, and the processing time varies between frameworks, where TensorFlow is the slowest and PyTorch is on average the fastest. The trends in a single example are similar to the overall results, however, TensorFlow and PyTorch over Deep500 are slightly faster than their native counterparts. Upon closer inspection, though, the runtime distributions are statistically indistinguishable.

As for correctness, the median (over the set of problem sizes) of the ℓ_∞ norms between Deep500 and the frameworks are ≈0.0007, ≈0.00068, and ≈0.00073 for TensorFlow, Caffe2, and PyTorch respectively.

C. Level 1: Networks

In the Level 1 analysis, we investigate the performance and accuracy benefits of *transforming a DNN convolution* by splitting input minibatches into smaller micro-batches, as Oyama et al. propose [34]. We apply the transformation on the network *independently of the framework* by solving an Integer Linear Program (ILP) to maximize performance and preserve memory utilization constraints. The Level 1 code then replaces convolutions with a split, followed by micro-batch convolution and concatenation operations, as illustrated in Fig. 7.

We use Deep500 to apply this transformation on both PyTorch and TensorFlow. Before the transformation, PyTorch suffered out-of-memory (OOM) errors for AlexNet [25] for


Original Deep500 Network
1
...
Conv2D 468x96x256x5x5
...
Use Implicit Precompute GEMM algorithm
Minibatch transformation
2
Transformed Deep500 Network
3
...
Split(axis=0, [4, 16, ..., 16]
Conv2D 4x96x256x5x5
Conv2D 16x96x256x5x5
Use Wingrad non-fused algorithm
Use Wingrad non-fused algorithm
Concat (axis=0)
...
Fig. 7: The illustration of the Microbatch transformation.

minibatch sizes of 468 or higher. Deep500's transformation decreases memory requirements, **eliminating OOM issues and enabling PyTorch to process a given dataset** in ≈225ms. Yet, the same optimization slows down TensorFlow from ≈350ms to ≈380ms, because splitting and concatenating nodes in TensorFlow incur additional memory copies. To alleviate this, one could modify the ONNX format to allow *referencing* sub-tensors instead of copying them.

D. Level 2: Training

In the Level 2 analysis, *we compare the performance and accuracy of training using TensorFlow, Caffe2, and Deep500 reference optimizers.*

Optimization Overhead We first measure the runtime of training in native TensorFlow and using the Deep500 TensorFlow integration. Apart from an instantiation overhead in the first epoch, Deep500 consistently incurs negligible (<1%) overhead, where both implementations take ≈243ms per epoch.

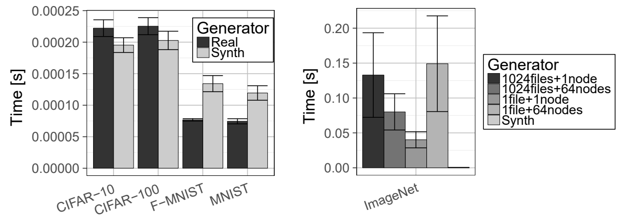


Fig. 8: The latency of loading various datasets.

Dataset Latency In Fig. 8, we measure the latency of loading data from different image processing datasets (`DatasetLatency` metric), which include reading the files, decoding data, and constructing minibatches of size 128. We test different formats (binary, `TFRecord`, POSIX `tar`) on one node and a distributed training setting with 64 nodes. For each dataset, we measure both data loading as well as synthetic data generation. In the left-hand side of the figure (using raw binary files), we see that for small datasets (MNIST, Fashion-MNIST), data loading is faster than allocating and generating synthetic data. This is because the dataset is already stored in memory, and not encoded. In larger datasets, such as CIFAR-10 and 100, new data is occasionally loaded from the filesystem, thus synthetic generation is faster.

For ImageNet, the figure (right-hand side) shows synthetic dataset generation is 2 orders of magnitude faster than loading images, as opposed to the other datasets. The main differences between ImageNet and the aforementioned datasets are the container (`TFRecord`) and image (encoded JPEG) format. To break down the image pipeline latency, we create an ImageNet dataset in a POSIX `tar` container with precomputed indexing (`IndexedTarDataset` in Deep500) and ingest it through two JPEG decoding pipelines (`PIL` and `libjpeg-turbo`). In Table III we see that using the `TFRecord` format and the TensorFlow pipeline is advantageous over the `tar` format. When using the POSIX format, both random file access and JPEG decoding play a role in slowdown. Even though for a single image `libjpeg-turbo` decodes images faster than the TensorFlow native decoder, the ratios between runtime of a minibatch and one image suggest that TensorFlow employs parallel decoding. Additionally, as opposed to the true random image selection in the `tar` format, TensorFlow uses pseudo-shuffling, where a buffer of (10,000) images is loaded into memory once and shuffled internally. This chunk-based loading reduces stochasticity, but, as the table shows, enables pipelining file I/O and in-memory shuffling.

Data Type	**Time [ms]**		
	Indexed `tar`		**`TFRecord`**
	`PIL`	`libjpeg-turbo`	Native Decoder
1 image (sequential)	10.04	2.80	7.43
1 image (shuffled)	51.19	34.60	9.50
128 images (sequential)	1,378.34	315.18	127.02
128 images (shuffled)	6,849.45	6,433.72	139.13

TABLE III: ImageNet decoding latency breakdown (median time).

For the distributed experiment, we test the ImageNet training set sharded to 1024 files (default) vs. 1 large file. In HPC, Parallel File Systems (PFS) generally prefer one segmented file rather than querying strings and inodes. Indeed, the latency of loading one file on a single node is lower than 1024. However, when using 64 nodes, we observe that surprisingly, 1024 files are ≈10% faster on Piz Daint. In all cases, the latency of loading a batch can be hidden by pipelining loading with DNN computation, a technique that is standard practice in large-scale ML.

Convergence In Fig. 9 and 10, we analyze the *convergence* of different native optimizers in Caffe2, Deep500, and of AcceleGrad [30], the Deep500 custom Python operator from Listing 7. While the AcceleGrad algorithm is short and descriptive, it exhibits lower performance than the native Caffe2 optimizers (≈1.6× slower). This can be attributed to the native Caffe2 weight update kernels, which are written specifically for GPUs. AcceleGrad also achieves comparable accuracy to similar algorithms (e.g., AdaGrad). Furthermore, while Deep500's Adam, which was directly translated from the original algorithm [24], is ≈5× slower than native, it still achieves high accuracy, even when the framework does not (Fig. 10).

In an attempt to understand this difference, we test the

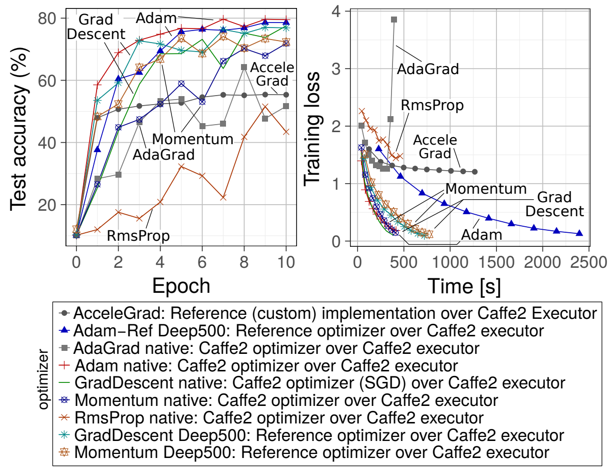


Fig. 9: The analysis of test accuracy vs. epoch number and training loss vs. elapsed time for different optimizers (assuming Caffe2, ResNet-18, CIFAR).

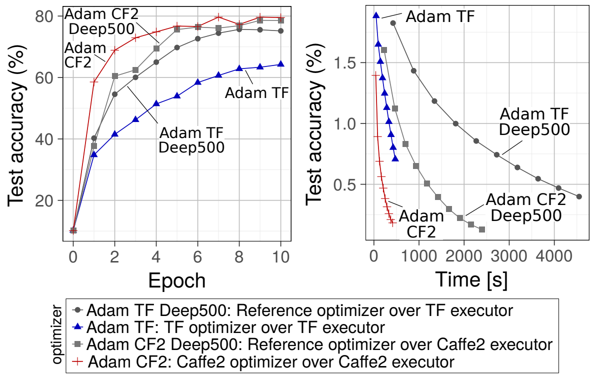


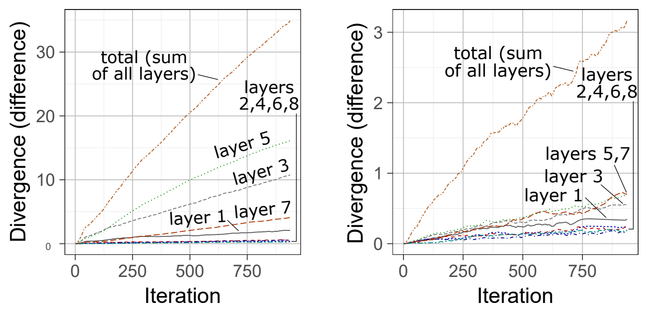
Fig. 10: The analysis of test accuracy vs. epoch number and training loss vs. elapsed time for different frameworks (assuming Adam, ResNet-18, CIFAR).

accuracy of the TensorFlow Adam optimizer on a smaller scale by comparing its trajectory with the Deep500 implementation. Fig. 11 shows the ℓ_2 and ℓ_∞ norms of the difference between the parameters, illustrating the chaotic divergence of deep learning, now easily visualized by Deep500. We observe that a single step of TensorFlow is faithful to the original algorithm, however, continuing training increases divergence, where some parameters (e.g., fully connected, layers 5,7) diverge faster than others (additive bias, layers 2,4,6,8).

The Deep500 reference optimizers are evidently slower (e.g., Figure 10), as they are unoptimized reference implementations. Here, our primary goal is *not* to offer tuned competitive codes, but instead to illustrate that Deep500 enables comparison and convergence of a multitude of optimizers.

E. Level 3: Distributed Training

Finally, we analyze Deep500's Level 3. *We compare distributed variants of SGD*, including TensorFlow's native parameter server (TF-PS), Horovod, as well as Deep500 reference implementations of centralized SGD (PSSGD), decentralized (DSGD), decentralized with a neighbor-based communication graph (DPSGD), asynchronous (ASGD), model-averaging (MAVG), DSGD with a Deep500 custom C++/MPI *allreduce* operator (CDSGD), and the *custom distributed communication* scheme SparCML [39], written as a custom Deep500 operator. All compared Deep500 implementations are distributed over the TensorFlow graph executor. We use ResNet-50 for strong and weak scaling, and up to 256 compute



(a) Divergence for the ℓ_2 norm. (b) Divergence for the ℓ_∞ norm.

Fig. 11: The difference (divergence) between DNN weights in the native optimization (Adam on TensorFlow) and the Deep500 Adam optimization (MNIST).

nodes. We use the `CommunicationVolume` metric in conjunction with mpiP [45] to collect communication statistics.

Fig. 12 (left) presents strong scaling results of the distributed implementations and competitors, compared with two baselines — TF-PS and Horovod — both measured using the TensorFlow Benchmark[1]. We use a minibatch size of 1,024 images on 8–64 nodes (since fewer nodes run out of memory and more nodes become ineffective). The figure shows that while *Python distributed optimizers provide reference results for **correctness*** analysis, *C++ operators can deliver **high-performance*** necessary for large-scale training, which are on-par with the state-of-the-art (Horovod). In particular, several effects can be seen: ❶ ASGD is centralized but does not use broadcast/gather operations. Consequently, despite being asynchronous, ASGD becomes slower the more worker nodes queue up to communicate. ❷ PSSGD, MAVG, and DSGD all start with similar epoch times, but as nodes increase, the decentralized versions (MAVG, DSGD) prove more efficient. ❸ DSGD written in C++, which uses direct CPU/GPU pointers, scales strongly up to 32 nodes and is almost an order of magnitude faster than its Python counterpart, which undergoes conversions to/from NumPy arrays.

In terms of communication volume, the collected metrics indicate that the reference DSGD and C++ DSGD exhibit the same communication volume, as expected. The number of PSSGD messages, however, scales linearly with the number of nodes. DPSGD communication volume remains constant with respect to the number of nodes, but usually converges slower and to a less accurate result [3]. As for SparCML's sparse allreduce (183 lines of C++ code), we see that while communication is greatly reduced (up to $2\times$ on 8 nodes), the running time is still high compared to the DSGD *allreduce* custom operator (23 lines of code), and increases with the number of participating nodes. This is both due to the reduced vector representation, which becomes denser with increasing nodes [39] (every allreduce step aggregates more sparse vectors with different indices), and due to the time it takes to filter the dense gradient to the sparse representation, which could potentially be optimized by using a CUDA custom operator.

In Fig. 12 (right), we study the weak scaling of the same implementations on 1–256 nodes. The baseline for this test is TF-PS, the default distributed implementation available in TensorFlow. Although simple, the *allreduce* operator (CDSGD) provides full DSGD, and is able to scale better than the PS architecture and Horovod, as in point ❷ above. This result is also non-trivial, since the parameters are stored on the GPUs, and they are copied automatically using Deep500 for use with MPI. Also observe that the native TensorFlow and Horovod implementations are missing results at 256 nodes. For TF-PS, the application crashed, whereas on Horovod the test ran but produced exploding (infinitely increasing) loss values, which is an indicator of incorrect gradient accumulation.

Overall, the plots show that **using Deep500, comparing multiple communication schemes is as easy as replacing an operator**. Deep500 facilitates the tradeoff analysis between different topologies, operator overhead (e.g., gradient sparsification), and optimizer quality (async. vs. sync. SGD); and enables benchmarking results on large node configurations.

VI. Related Work

Our work touches on various areas. We now discuss related works, briefly summarizing the ones covered in previous sections: DL frameworks in § II-B and Table I, DL data model and format in § II-D, and DL benchmarks in Table II.

DL Benchmarks The DL community has recently gained interest in benchmarking DL codes. Example benchmarks are DAWNBench [9], MLPerf [31], or DeepBench [40]; see Table II for a full list and analysis of their functionalities. *Deep500 is the only benchmark that addresses **the five challenges** described in § III: customizability, metrics, performance, validation, and reproducibility*.

DL Frameworks There exist many DL frameworks and related libraries as well as frontends [1, 21, 6, 35]. As we illustrate in Table I, none of them offers a full spectrum of functionalities. *Deep500 does not only enable benchmarking of these systems. On top of that – through its meta-framework design – it enables integrating arbitrary elements of the considered DL systems to **combine the best of different DL worlds***.

DL Data Formats Finally, we use and extend the established ONNX DNN format [33] with an object-oriented notation, new operations, and others. Thus, *Deep500 significantly improves interoperability between ONNX and DL frameworks*.

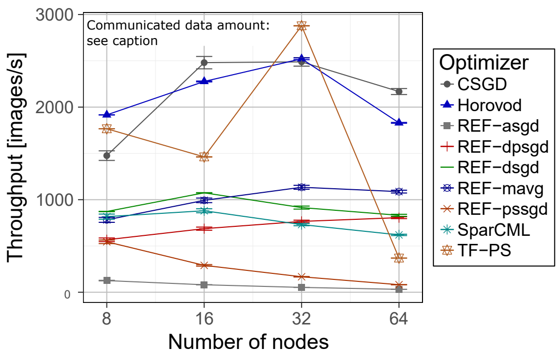
VII. Conclusion

Deep Learning (DL) has become ubiquitous in areas as diverse as speech recognition and autonomous driving. However, it is still unclear how to compare and benchmark the plethora of available DL algorithms and systems, especially in extreme-scale distributed environments.

To answer these questions, we introduce Deep500: a customizable infrastructure that enables detailed, accurate, fast, and fair benchmarking of DL codes. The essence of Deep500 is its *layered and modular* design that allows to *independently* extend and benchmark DL procedures related to simple operators, whole neural networks, training schemes, and distributed training. The principles behind this design can be reused to enable interpretable and reproducible benchmarking of extreme-scale codes in domains outside DL.

To ensure the best design decisions for Deep500, we analyze challenges in benchmarking complex and large-scale DL codes. To this end, we identify *five core challenges*: customizability, metrics, performance, validation, and reproducibility.

[1]https://www.github.com/tensorflow/benchmarks



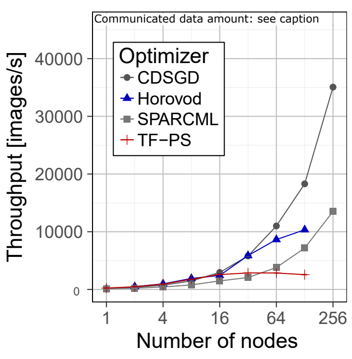


Fig. 12: **Scaling of Level 3**: Strong (left) and weak (right) scaling on Piz Daint and ImageNet. **Communicated data per node:** 0.952 GB (CDSGD), 0.951 GB (SparCML), 0.952 GB (REF-dsgd), 28.573 GB (REF-asgd), 1.904 GB (REF-dpsgd), 1.903 GB (REF-pssgd).

Through extensive evaluation, we illustrate that Deep500 satisfies these challenges. For example, it ensures identical accuracy while offering negligible ($<1\%$) performance overheads over native operators or whole DNNs in state-of-the-art DL frameworks, including TensorFlow, PyTorch, and Caffe2.

Finally, we predict that Deep Learning will become a part of computing as important as general dense or sparse linear algebra. Thus, we construct Deep500 such that it can be freely modified to ensure fair benchmarking, produce artifacts for DL papers, provide insightful analyses, and enable effective development of any future DL effort.

Acknowledgments

We thank Cédric Renggli, Dan Alistarh, Yosuke Oyama, and Kfir Y. Levy for valuable discussions; and Hussein Harake, Colin McMurtrie, and the whole CSCS team granting access to the Greina and Daint machines, and for their excellent technical support. This project has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 programme (grant agreement DAPP, No. 678880). T.B.N. is supported by the ETH Zurich Postdoctoral Fellowship and Marie Curie Actions for People COFUND program.

References

[1] M. Abadi et al. "Tensorflow: a system for large-scale machine learning." In: *OSDI*. 2016.
[2] R. Adolf et al. "Fathom: Reference Workloads for Modern Deep Learning Methods". In: *arXiv:1608.06581* (2016).
[3] T. Ben-Nun and T. Hoefler. "Demystifying Parallel and Distributed Deep Learning: An In-Depth Concurrency Analysis". In: *arXiv:1802.09941* (2018).
[4] J. Bergstra et al. "Theano: Deep learning on GPUs with Python". In: *NIPS, BigLearning Workshop, Granada, Spain*. 2011.
[5] K. Chellapilla, S. Puri, and P. Simard. "High Performance Convolutional Neural Networks for Document Processing". In: *ICFHR*. 2006.
[6] T. Chen et al. "Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems". In: *arXiv:1512.01274* (2015).
[7] T. Chen et al. "TVM: End-to-End Optimization Stack for Deep Learning". In: *arXiv:1802.04799* (2018).
[8] F. Chollet. *Keras*. 2015. URL: https://github.com/keras-team/keras.
[9] C. Coleman et al. "DAWNBench: An End-to-End Deep Learning Benchmark and Competition". In: *ML Systems Workshop @ NIPS* (2017).
[10] R. Collobert, S. Bengio, and J. Mariéthoz. *Torch: a modular machine learning software library*. Tech. rep. Idiap, 2002.
[11] A. Damien et al. *TFLearn*. 2016. URL: https://github.com/tflearn/tflearn.
[12] J. Dean et al. "Large scale distributed deep networks". In: *NIPS*. 2012.
[13] J. Deng et al. "ImageNet: A Large-Scale Hierarchical Image Database". In: *CVPR*. 2009.
[14] H. Dong et al. "TensorLayer: a versatile library for efficient deep learning development". In: *ACMMM*. 2017.
[15] J. J. Dongarra et al. *Top500 supercomputer sites*. 1994.
[16] GitHub. *Deep Learning Repositories on GitHub*. 2018. URL: https://github.com/topics/deep-learning.
[17] K. He et al. "Deep Residual Learning for Image Recognition". In: *CVPR*. 2016.
[18] T. Hoefler and R. Belli. "Scientific benchmarking of parallel computing systems". In: *SC*. 2015.
[19] HPE. *Deep Learning Benchmarking Suite*. 2018. URL: https://developer.hpe.com/platform/hpe-deep-learning-cookbook/home.
[20] X. Jia et al. "Highly Scalable Deep Learning Training System with Mixed-Precision: Training ImageNet in Four Minutes". In: *arXiv:1807.11205* (2018).
[21] Y. Jia et al. "Caffe: Convolutional architecture for fast feature embedding". In: *ACMMM*. 2014.
[22] Kaggle. *Data Science Competition Repository*. 2018. URL: https://www.kaggle.com.
[23] Khronos Group. *Neural Network Exchange Format*. 2018.
[24] D. P. Kingma and J. Ba. "Adam: A Method for Stochastic Optimization". In: *ICLR*. 2015.
[25] A. Krizhevsky, I. Sutskever, and G. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *NIPS*. 2012, pp. 1097–1105.
[26] A. Krizhevsky. "Learning Multiple Layers of Features from Tiny Images". MA thesis. 2009.
[27] A. Lavin and S. Gray. "Fast algorithms for convolutional neural networks". In: *CVPR*. 2016.
[28] Y. LeCun and C. Cortes. *The MNIST database of handwritten digits*. 1998.
[29] Y. LeCun et al. "Gradient-based learning applied to document recognition". In: *Proc. IEEE* (1998).
[30] Y. K. Levy, A. Yurtsever, and V. Cevher. "Online Adaptive Methods, Universality and Acceleration". In: *NIPS*. 2018, pp. 6500–6509.
[31] MLPerf. *A broad ML benchmark suite*. 2018. URL: https://www.mlperf.org.
[32] P. Moritz, R. Nishihara, and M. I. Jordan. "A Linearly-Convergent Stochastic L-BFGS Algorithm". In: *ICAIS*. 2016.
[33] Open Neural Network Exchange. *ONNX Github repository*. 2018. URL: https://www.github.com/onnx.
[34] Y. Oyama et al. "Accelerating Deep Learning Frameworks with Micro-batches". In: *IEEE International Conference on Cluster Computing (CLUSTER)*. 2018.
[35] A. Paszke et al. *Automatic differentiation in PyTorch*. Tech. rep. 2017.
[36] E. Real et al. "Regularized Evolution for Image Classifier Architecture Search". In: *arXiv:1802.01548* (2018).
[37] B. Recht et al. "Hogwild: A lock-free approach to parallelizing stochastic gradient descent". In: *NIPS*. 2011.
[38] J. Redmon. *Darknet: Open Source Neural Networks in C*. 2013–2018.
[39] C. Renggli, D. Alistarh, and T. Hoefler. "SparCML: High-Performance Sparse Communication for Machine Learning". In: *arXiv:1802.08021* (2018).
[40] B. Research. *DeepBench: Benchmarking Deep Learning operations on different hardware*. 2018. URL: https://github.com/baidu-research/DeepBench.
[41] H. Robbins and S. Monro. "A Stochastic Approximation Method". In: *The Annals of Mathematical Statistics* (1951).
[42] A. Sergeev and M. Del Balso. "Horovod: fast and easy distributed deep learning in TensorFlow". In: *arXiv:1802.05799* (2018).
[43] Skymind. *Deeplearning4j: Open-source distributed deep learning for the JVM*. 2016. URL: https://deeplearning4j.org.
[44] S. Tokui et al. "Chainer: a next-generation open source framework for deep learning". In: *LearningSys @NIPS*. 2015.
[45] J. S. Vetter and M. O. McCracken. "Statistical Scalability Analysis of Communication Operations in Distributed Applications". In: *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*. PPoPP '01. Snowbird, Utah, USA: ACM, 2001, pp. 123–132.
[46] H. Xiao, K. Rasul, and R. Vollgraf. "Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms". In: *arXiv:1708.07747* (2017).
[47] S. R. Young et al. "Evolving Deep Networks Using HPC". In: MLHPC'17. 2017.
[48] D. Yu et al. "An introduction to computational networks and the computational network toolkit". In: *Microsoft Technical Report MSR-TR-2014–112* (2014).
[49] S. Zagoruyko and N. Komodakis. "Wide Residual Networks". In: *arXiv:1605.07146* (2016).
[50] H. Zhu et al. "TBD: Benchmarking and Analyzing Deep Neural Network Training". In: IISWC, 2018.