CCF Transactions on High Performance Computing (2024) 6:94–111
https://doi.org/10.1007/s42514-023-00168-6

REGULAR PAPER

Quantitative evaluation of deep learning frameworks in heterogeneous computing environment

Zhengxian Lu[1] · Chengkun Du[1] · Yanfeng Jiang[1] · Xueshuo Xie[2,3] · Tao Li[1,2] · Fei Yang[4]

Received: 8 July 2023 / Accepted: 28 August 2023 / Published online: 8 September 2023

Abstract
Deep learning frameworks are powerful tools to support model training. They dispatch operators by mapping them into a series of kernel functions and launching these kernel functions to specialized devices such as GPUs. However, there is little known about the performance of dispatching and mapping mechanisms in different frameworks, although these mechanisms directly affect training time. This paper presents a performance evaluation in various frameworks by examining their kernel function efficiency and operator dispatching mechanisms. We introduce two evaluation metrics, device computing time (DCT) and device occupancy ratio (DOR), based on the device's active and idle states. To ensure comparable evaluation results, we propose a three-step verification method including hyper-parameter, model, and updating method equivalences. Due to inequivalent implementations in frameworks, we present an equivalence adjustment method based on the number of operators. Our evaluation results demonstrate the device utilization capability of five frameworks, namely PyTorch, TensorFlow 1, TensorFlow 2, MXNet, and PaddlePaddle, and reveal the potential for further optimizing the training performance of deep learning frameworks.

Keywords Deep learning framework · Performance evaluation · Device computing time · Device occupancy ratio

1 Introduction

In the realm of deep learning, frameworks like TensorFlow (Abadi et al. 2016) and PyTorch (Paszke et al. 2019) play a crucial role in bridging deep learning models and hardware platforms. These frameworks automatically execute user-defined training processes in heterogeneous computing systems by dispatching computing tasks, referred to as operators, on high-performance computing devices such as GPUs to alleviate the burden of model training. However, it is not trivial for frameworks to gain high performance from heterogeneous computing. During training, frameworks dispatch operators by mapping them into a series of kernel functions and launching these kernel functions to the device sequentially. Their operator mapping and dispatching mechanisms can significantly affect the training performance (Kim et al. 2017; Zhu et al. 2018). Besides, various optimization techniques are also adopted to accelerate the training. Through evaluation, we can better understand their mechanisms, aid in the more efficient use of frameworks, and explore further optimization opportunities.

However, recent studies have mainly evaluated deep learning frameworks by discussing their lack of support for

✉ Xueshuo Xie
xueshuoxie@nankai.edu.cn

✉ Tao Li
litao@nankai.edu.cn

Zhengxian Lu
luzx@mail.nankai.edu.cn

Chengkun Du
dck@mail.nankai.edu.cn

Yanfeng Jiang
yfjiang@mail.nankai.edu.cn

Fei Yang
yangf@zhejianglab.com

1 College of Computer, Nankai University, Tianjin 300350, China

2 Haihe Lab of ITAI Street, Tianjin 300350, China

3 State Key Laboratory of Computer Architecture, Institute of Computing Technology, Beijing 100190, China

4 Zhejiang Lab, Zhejiang 310058, China

specific device architectures (Trindade et al. 2019; Yang et al. 2021), communication issues in distributed training (Shi et al. 2018; Jäger et al. 2018; Shams et al. 2017), hardware resource consumption (Wu et al. 2018; Elshawi et al. 2021), and software engineering issues (Guo et al. 2019; Han et al. 2020a, b; Sun et al. 2021). While various deep learning frameworks have been included in benchmarking efforts, these efforts have focused on different evaluation targets, such as deep learning workload (Adolf et al. 2016; Zhu et al. 2018; Mattson et al. 2020a). Some studies have similar concerns to ours, but they are limited to discussing the kernel function computation efficiency of certain operators, such as convolutions (Shi et al. 2016; Kim et al. 2017). The performance gap among frameworks due to their different operator mapping and dispatching mechanisms remains unknown.

In this paper, we aim to evaluate and reveal the performance of different mechanisms among deep learning frameworks. The performance evaluation of frameworks has two limitations. (1) Both the operator mapping mechanism and operator dispatching mechanism in the framework can become performance bottlenecks. It is not possible to evaluate the mechanism that does not become the performance bottleneck by training time, leading to incomparable evaluation results. (2) Implementing a training process for each framework is necessary, but inequivalent training process implementations may lead to unfair evaluation results. For example, if one framework's training includes bias computations in convolutional layers while others do not, the evaluation will lead to misunderstandings that the framework is worse in performance. Consequently, to perform an evaluation of frameworks, there are two challenges: (1) how to compare the operator mapping and dispatching mechanisms among various frameworks; (2) how to locate and eliminate the inequivalent implementations of training process to ensure the comparability of evaluation.

We employ two key designs to address the challenges mentioned above. To tackle the first challenge, we divide the device state into two categories: active and idle states, to reflect the operator mapping and dispatching mechanism of the frameworks. Based on these states, we propose the device computing time (DCT) metric to measure the execution time of the kernel functions on the device and the device occupancy ratio (DOR) metric to reveal the degree of device starvation. Shorter DCT and higher DOR lead to less training time. For the second challenge, we propose a three-step equivalence validation method that includes hyper-parameter equivalence, model equivalence, and parameter updating equivalence. This validation method helps to locate the inequivalent implementations and verify the training process equivalence. To eliminate the discovered inequivalent implementations, we present an equivalence adjustment method based on the number of operators. By comparing the number of operators before and after the equivalence adjustment, we can determine whether to adjust the inequivalent implementations.

We carefully choose a convolutional neural network named ResNet (He et al. 2016) training on the CIFAR-10 (Krizhevsky et al. 2009) dataset and a Transformer named BERT (Devlin et al. 2019) training on the SQuAD (Rajpurkar et al. 2016) as our study cases. We evaluate five frameworks, i.e., PyTorch, MXNet, PaddlePaddle, TensorFlow 1, and TensorFlow 2, on NVIDIA RTX A6000 GPU. Our evaluation shows the advantages and disadvantages of frameworks that use different mechanisms. Furthermore, we also investigate how different hyper-parameters can affect training performance. We also demonstrated the results of applying our metrics to Huawei Atlas 300T. The contributions of this paper are as follows:

- We introduce evaluation metrics based on device states, namely DCT and DOR, to provide a comprehensive analysis of deep learning frameworks. These metrics reflect changes in the inner state of the device, facilitate the decoupling of training time, and reveal the effectiveness of mechanisms.
- We propose a three-step equivalence validation method and an equivalence adjustment method for evaluation comparability. Through validation and adjustment, we can identify any inequivalent implementations in training and eliminate them to obtain the equivalent training process across frameworks.
- We conduct a comprehensive evaluation of five deep learning frameworks and derive several conclusions. For instance, we find that PaddlePaddle outperforms PyTorch in certain cases due to its built-in operators, while PyTorch performs better than PaddlePaddle in large-batch training due to fewer device synchronizations.

2 Background

2.1 Deep learning training and frameworks

Generally, deep learning is to design a model and an objective function based on domain knowledge and then perform a training process to determine the model parameters by optimizing the objective function (Li et al. 2014). The most commonly used optimization method for deep learning models is an iterative algorithm called Stochastic Gradient Descent (SGD) (Sun et al. 2019). The training process with SGD consists of multiple *epochs*. In each epoch, the model is trained on the training data by randomly dividing the entire dataset into batches of the same size and updating the model parameters with these

Table 1 Information of deep learning frameworks

	Stars	Forks	Recent activities	Status	DEM
TensorFlow (Abadi et al. 2016)	172,265	87,980	High	Ongoing	S/E
PyTorch (Paszke et al. 2019)	64,151	17,753	High	Ongoing	E
PaddlePaddle (Ma et al. 2019)	19,851	5049	High	Ongoing	S/E
MXNet (Chen et al. 2015)	20,322	6872	Medium	Ongoing	S/E
DL4J[a]	12,826	4937	Medium	Ongoing	E
OneFlow (Yuan et al. 2021)	4610	534	Medium	Ongoing	E
MindSpore[b]	3386	623	Low	Ongoing	E
Caffe (Jia et al. 2014)	33,184	18,969	Low	Discontinued	S
CNTK (Seide and Agarwal 2016)	17,331	4380	Low	Discontinued	S
Theano (Al-Rfou et al. 2016)	9685	2509	Low	Discontinued	S
Caffe2[c]	8399	1998	Low	Discontinued	S

The community activities were obtained from GitHub on March 21, 2023. Recent activities are categorized by the total number of issues and pull requests in the past month: Low (less than 1), Medium (between 1 and 100), and High (above 100). The default execution mode (DEMs) is denoted as *S* for static, *E* for eager, and *S/E* for static before but eager now

[a]https://deeplearning4j.konduit.ai/

[b]https://www.mindspore.cn/

[c]https://caffe2.ai/

batches sequentially. The number of samples in each batch is referred to as the *batch size*. A training *step* processes a single batch and approaches the local optimum point through several stages, including data preprocessing, forward computing, backward propagation, and parameter updating (Rumelhart et al. 1986). Data preprocessing performs data augmentation and processes the samples into a format that can be fed into the model. The processed samples then undergo forward computing and backward propagation to obtain the gradients of the parameters, which are used to update the model parameters to approach the local minimum value of the objective function.

To execute the training process, deep learning frameworks should construct a computation graph that abstracts the calculations of the application (Abadi et al. 2016). A computation graph is a directed acyclic graph composed of nodes representing operators, such as convolutional layers and ReLU activation functions, and edges representing data dependencies. Frameworks have two execution modes to generate computation graphs: static deferred execution (Abadi et al. 2016) and eager execution (Paszke et al. 2019). In the static deferred execution mode, the framework explicitly constructs the computation graph to obtain global computation information. In the eager execution mode, the framework implicitly builds the computation graph by only reserving sub-graphs of intermediate results and releasing them when they are no longer referenced. For example, PyTorch adopts eager execution to simplify the construction and debugging of modules (Paszke et al. 2019). Conversely, MXNet employs a static execution mode to ensure that an optimized graph is executed on devices (Chen et al. 2015).

Frameworks may change their default execution mode alone with the version updates to ease the development and debugging for users. For instance, while TensorFlow 1 only supports static deferred execution, TensorFlow 2 adopts dynamic eager execution as the default mode. To bridge the performance gap, these frameworks also provide methods to generate static dataflow graphs from eager execution codes. Furthermore, the development status of some frameworks has also changed. For example, Theano and CNTK have ceased development, and Caffe2 became the backend of PyTorch to provide both development efficiency and production-ready capability. Table 1 exhibits community activity, status, and default execution mode of some popular frameworks.

2.2 Related works

Many efforts have conducted remarkable experimental comparisons of training times and hardware resource utilization (Shi et al. 2016; Kim et al. 2017; Mahmoud et al. 2019; Elshawi et al. 2021; Xie et al. 2023). These studies analyze the performance characteristics of frameworks by breaking down the training time into stage-wise or operator-wise execution time and examining the utilization of deep learning libraries, such as *cuDNN* and *fbcunn*. However, the usage of deep learning libraries is insufficient to represent the performance of the mechanisms implemented in frameworks. Liu et al. (2018) demonstrated that default configurations recommended by frameworks could significantly impact results and highlighted the importance of considering hyper-parameters and software configurations when benchmarking deep learning frameworks.

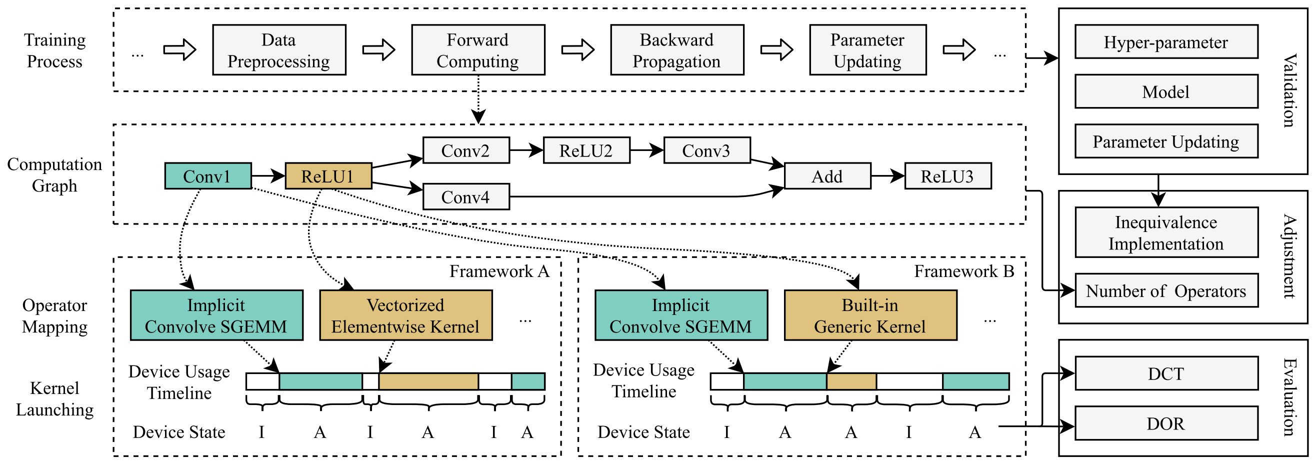


Fig. 1 Framework working mechanism and our methods. To dispatch operators like *Conv1* and *ReLU1* in the computation graph, frameworks map operators to kernel functions and launch them on the high-performance computing device

However, they do not consider the inequivalence implementations in frameworks.

Several researchers have investigated the performance of various frameworks on specific device architectures. For instance, Trindade et al. (2019) studied NUMA architectures, while Yang et al. (2021) focused on Xeon Phi CPU. In distributed training, studies have examined the performance of NVLink and Knights Landing (Shams et al. 2017), parallel communication overhead (Shi et al. 2018), and different communication approaches (Jäger et al. 2018). Recent efforts in software engineering have concentrated on development and deployment (Guo et al. 2019), dependency networks (Han et al. 2020a), popular topics discussed (Han et al. 2020b), and bug fixes (Sun et al. 2021; Makkouk et al. 2022) in frameworks. Additionally, several benchmarks aim to characterize deep learning workloads, such as DAWNBench (Coleman et al. 2019), Fathom (Adolf et al. 2016), TBD (Zhu et al. 2018), MLPerf (Mattson et al. 2020b; Reddi et al. 2020, 2021), and AIBench (Tang et al. 2021). While the above works provide excellent experimental benchmarks and analysis, our focus is primarily on the mechanisms within frameworks.

Remark The aforementioned efforts lack a comprehensive evaluation of mechanisms in frameworks. Besides, very few studies have discussed the inequivalence implementations in frameworks. These are precisely the problems that our work mainly solves.

3 Methodology

3.1 Overview

We propose three methods to obtain comparable evaluation results, as shown in Fig. 1. We adopt two metrics to reveal the framework working mechanism based on the device states during training. We validate the training process equivalence and identify any inequivalence implementation of frameworks by proposing a three-step validation method. Moreover, we adjust all the identified inequivalence implementations based on the number of operators.

- *Evaluation metrics by device state.*

  Deep learning frameworks transform the training process into a series of kernel functions and launch them on devices. The dispatching mechanism of operators and operator mapping methods in various frameworks can be assessed based on the device state. We measure the duration of the active state using device computing time (DCT) and reveal the proportion of idle state in the total time through device occupancy ratio (DOR). These two metrics can be seen as two optimization directions of training time, indicating the potential improvement space of frameworks.
- *Three-step equivalence validation.*

  Frameworks should execute the same training process to ensure the comparability of evaluation results. According to the composition of the training process, we propose a three-step validation method. Firstly, we establish hyper-parameter equivalence by setting each necessary hyper-parameters to the same value. Secondly, we ensure model equivalence by comparing output values of forward computing among frameworks. Finally, we verify parameter updating equivalence by checking that all parameter updating methods are equivalent.
- *Equivalence adjustment based on the number of operators.*

  After completing the three-step equivalence validation, we can find inequivalence implementations in frameworks. To determine if these inequivalence implementations will result in incomparability in evaluations, we compare the number of operators before and after equivalence adjustment. Additionally, we demonstrate our equivalence adjustment method using examples of momentum optimizer inequivalence and ℓ_2 regularization inequivalence.

3.2 Evaluation metrics by device state

Figure 1 illustrates how frameworks dispatch operators on devices. During training, frameworks map operators into kernel functions according to the computation graph and launch all kernel functions on the computing device. In operator mapping, frameworks either map an operator into the same kernel function, leveraging deep learning libraries like *cuDNN*, or map into different kernel functions. Besides, an inefficient operator dispatching can cause the device lying idle state. On the other hand, the device is in an active state when the device is running for computing one or more kernel functions. In Fig. 1, the active state and the idle state are represented by A and I, respectively. To shorten training time, frameworks should optimize training performance from two aspects: (1) to minimize the execution time of kernel functions, such as designing more efficient kernel functions and reducing unnecessary computation, to reduce the active time; (2) to improve the speed of operator dispatching and overlap the input/output and CPU (central processing unit) execution time spent with kernel function computing time to avoid the device being idle. Therefore, we can evaluate the operator mapping mechanism and the operator dispatching mechanism according to the device state.

Device computing time. We employ a metric called device computing time (DCT) to measure the time cost for the computing unit of the device to handle kernel functions during the sampling period; that is, the time the compute engine of the device is active. Specifically, we denote DCT within a sampling period that starts at time t_s and ends at time t_e as $T(t_s, t_e)$. This metric is defined as follows:

$$T(t_s, t_e) = \int_{t_s}^{t_e} D_{active}(t)dt, \tag{2}$$

where $D_{active}(t)$ represents the state of the device as in (3).

$$D_{active}(t) = \begin{cases} 1, \text{ if device is in active at time } t \\ 0, \text{ otherwise} \end{cases}. \tag{3}$$

Device occupancy ratio. We use a metric called device occupancy ratio (DOR) to reveal the degree of device starvation. Specifically, we denote DOR within a sampling period that starts at time t_s and ends at time t_e as $R(t_s, t_e)$. This metric is defined as follows:

$$R(t_s, t_e) = \frac{T(t_s, t_e)}{t_e - t_s}. \tag{4}$$

We have $R(t_s, t_e) \in (0, 1]$. The longer the device is idle, the closer $R(t_s, t_e)$ is to 0. On the other hand, if the device is active throughout the entire sample duration, $R(t_s, t_e)$ takes the maximum value of 1.

Analysis. The training time is a general metric for the overall performance evaluation. However, it cannot pinpoint the exact bottleneck in performance since it is difficult to determine whether the kernel functions computation or the operator dispatching is inefficient during training. To evaluate operator mapping and dispatching, we observe internal changes in device states during the training process. Given a certain computation graph, a short DCT represents efficient kernel functions, while a DOR closing to one suggests efficient operator dispatching. Since training time is the sum of active and idle time on the device, shorter DCT and higher DOR lead to less total training time. Therefore, we can indicate the further optimization direction within frameworks according to the continuous device state switching between active and idle.

3.3 Three-step equivalence validation

To ensure the comparability of performance evaluation, we should keep training process equivalence on all frameworks. As illustrated in Fig. 1, the training process consists of multiple successive training steps. The consistent training step can ensure the training process equivalence. A single step includes data preprocessing, forward computing, backward propagation, and parameter updating. Therefore, to ensure the training process equivalence, we need to guarantee that: (1) data preprocessing is consistent; (2) model operators and data dependencies are consistent; (3) parameter updating methods are consistent across all frameworks. We propose a three-step validation method, including hyper-parameter equivalence, model equivalence, and parameter updating equivalence, to verify the training process equivalence.

Hyper-parameter equivalence ensures consistent data preprocessing and assists in ensuring the same operators and parameter updating method. We validate four types of hyper-parameters: (1) dataset preprocessing, like *mean* for data standardization; (2) operators, e.g., *rate* for dropout layers; (3) parameter updating, such as *learning rate* and *weight decay*; (4) performance factors like the number of workers for data preprocessing in parallel. To accomplish the hyper-parameter equivalence validation, we begin by loading an identical configure file into all frameworks. We then explicitly pass all arguments when calling APIs to avoid different default arguments, such as *bias* for convolutional layers and *epsilon* for batch normalization layers. Finally, we consider API support when determining hyper-parameters because not all frameworks support every API. For example, some frameworks do not supply initialization techniques like *He* initialization (He et al. 2015), so we set convolution initializers as *Xavier Uniform* (Glorot and Bengio 2010) instead.

Model equivalence allows verifying the operators and data dependencies equivalence of models. However, obtaining computation graphs built by frameworks can be challenging, and different orders of floating-point calculation may result in varying outputs. To address these issues, we employ an indirect approach to validate model equivalence. (1) We first train a model with one framework for several epochs, save the parameters of the trained model into a binary file, and then load it into other frameworks so that all frameworks have identical model parameters. (2) We compare the model using *Open Neural Network Exchange (ONNX)*,[1] which is an open format for representing deep learning models. We export the model in each framework as an *ONNX* model, input the same data to these *ONNX* models, and check whether all the models output consistent results. (3) If there are discrepancies in output results among the models, we identify the cause of the difference and patch it. We repeat the validation process until all models produce consistent outputs. We record inequivalence implementations of frameworks causing the different outputs and decide whether to adjust these implementations in equivalence adjustment.

Parameter updating equivalence aims to ensure that the same parameters are obtained after updating. In parameter updating, a *transformation* function is first applied to gradients, followed by an *optimizer* function to update the parameters according to transformed gradients. In addition, a *regularization* function is used to prevent overfitting. However, frameworks may have different implementations of the same function. To address this issue, we validate the equivalence of the transformation, regularization, and optimizer functions. We compare these implementations based on the descriptions of these functions from their official documents. We also examine their source codes if necessary. We have discovered some inequivalence implementations, such as differences in momentum optimizer and ℓ_2 regularization between TensorFlow and PyTorch. We will use our equivalence adjustment method to adjust these implementations.

3.4 Equivalence adjustment based on the number of operators

Considering that the working mechanism of the framework is to continuously dispatch operators in the computation graph during training, it is important to keep the same number of operators across frameworks to maintain the comparability of evaluation results. We propose an equivalence adjustment method based on the number of operators to address each inequivalence implementation among frameworks. The proposed method involves four steps. (1) We count the number of operators for the inequivalence implementation in different frameworks and calculate the difference in operator counts as $count_{pre}$. (2) We modify the training process instead of changing the implementation in the framework to make the implementation equivalent. (3) We count the number of operators in different frameworks after modification and calculate their difference as $count_{post}$. (4) We apply the implementation with a smaller difference in the number of operators for performance evaluation. If $count_{pre} < count_{post}$, the modification will not be applied; otherwise, we will modify the inequivalence. Below we take momentum optimizer and ℓ_2 regularization as examples.

For momentum optimizer inequivalence, TensorFlow adopts (5) to update parameters, where w_t, $\varepsilon > 0$, and μ are the model parameters in iteration t, learning rate and momentum coefficient, respectively. g_t is the gradient given w_t. $v_0 = 0$.

$$\begin{aligned} v_{t+1} &= \mu v_t + \varepsilon g_{t+1} \\ w_{t+1} &= w_t - v_{t+1} \end{aligned}. \tag{5}$$

However, PyTorch employs the following equation:

$$\begin{aligned} v_{t+1} &= \mu v_t + g_{t+1} \\ w_{t+1} &= w_t - \varepsilon v_{t+1} \end{aligned}. \tag{6}$$

Equation (5) and (6) both require four operators, including twice multiplication and twice addition/subtraction. Therefore, We have $count_{pre} = 0$ for momentum optimizer inequivalence. We can modify (6)–(7) to make the two momentum optimizers equivalent.

$$\begin{aligned} v_{t+1} &= \frac{\varepsilon_t}{\varepsilon_{t+1}} \mu v_t + g_{t+1} \\ w_{t+1} &= w_t - \varepsilon_{t+1} v_{t+1} \end{aligned}. \tag{7}$$

We can adjust μ during training to modify the momentum optimizer. However, computing the new μ will introduce extra operators with $count_{post} = 2$. We have $count_{pre} < count_{post}$, and thus discard adjusting the momentum optimizer inequivalence.

For ℓ_2 regularization inequivalence, TensorFlow adds the weight penalties to the objective function and calculates the gradients of model weights to perform the regularization:

$$g_t = \frac{\partial L(w_t) + \lambda \sum w_t^2}{\partial w_t}, \tag{8}$$

where $L(w)$ is the objective function, and λ is the regularization coefficient. PyTorch adopts weight decay to implement ℓ_2 regularization, which can be represented as:

[1] https://onnx.ai/.

$$g_t = \frac{\partial L(w_t)}{\partial w_t} + \lambda w_t. \quad (9)$$

TensorFlow requires extra operators for the ℓ_2 weight penalty term $\sum w_t^2$, and we have $count_{pre} > 0$. We implement the weight decay method as (9) on TensorFlow by removing weight penalties ($\lambda = 0$) and adjusting the parameter updating process. After this modification, both TensorFlow and PyTorch have the same number of operators for ℓ_2 regularization implementation. Therefore, we have $count_{post} = 0$ and $count_{pre} > count_{post}$, and thus apply this modification to our performance evaluation.

4 Evaluation

4.1 Experimental setup

Application. We choose a convolution neural network called ResNet (He et al. 2016) and a transformer model named BERT (Devlin et al. 2019) as the workloads. ResNet uses residual modules to maintain gradients for deep layers, which is a common practice in many deep learning models. Additionally, ResNet is widely used as the backbone for other computer vision tasks. We choose an image classification task called CIFAR-10 (Krizhevsky et al. 2009), which is widely used to evaluate the classification capability of deep learning models. The ResNet architecture for CIFAR-10 employs a hyper-parameter n to control network size, as the number of convolutions and fully-connected layers can be computed by $2 + 6n$. Therefore, we can easily expand the model size and explore the impact of model size on performance based on ResNet and CIFAR-10. BERT is a classic Transformer model widely used in natural language processing (NLP). Considering that the Transformer is often used for different NLP tasks by fine-tuning, we choose SQuAD v1.1 (Rajpurkar et al. 2016) as the dataset and fine-tune the pre-trained BERT on it. Due to the memory limitations of the device, we only tested the BERT-base, which is the smaller model size of BERT. We examined its performance under six different batch sizes. Notably, our focus is not on neural network capabilities but rather on differences in training performance among frameworks.

Platform. Our experiments are conducted on hardware equipped with two Intel(R) Xeon(R) Gold 6248 CPUs and one NVIDIA RTX A6000 GPU. We select five frameworks, namely MXNet, PaddlePaddle, PyTorch, TensorFlow 1, and TensorFlow 2, based on the community activity and status as shown in Table 1. Our evaluation includes two major versions of TensorFlow because there are significant differences between TensorFlow 1 and TensorFlow 2 in default execution mode and APIs. We leverage *Docker*[2] to set up the development environment for each framework. For stability in framework performance, we ensure the consistent major versions, but adopt the official default version of *CUDA* and *cuDNN* instead of matching equal version numbers for all frameworks. Table 2 shows the version number of all the frameworks.

Table 2 Version information of frameworks

	Framework Version	CUDA Version	cuDNN Version
MXNet	1.8.0	11.0	8.0.4
PaddlePaddle	2.1.3	11.2	8.1.1
PyTorch	1.9.0	11.1	8.0.5
TensorFlow 1	1.15.1	11.2	8.1.1
TensorFlow 2	2.6.0	11.2	8.1.0

Optimization techniques. We apply two extra optimization techniques for all frameworks supporting them. (1) Choosing convolution algorithms by testing (*OpT* in short). This allows MXNet, PaddlePaddle, and PyTorch to benchmark and choose the fastest convolution algorithm prior to training. (2) Static deferred execution (*OpS*). MXNet, PaddlePaddle, and TensorFlow 2 can run eager execution programs under the static deferred execution mode. We also evaluate these two optimization techniques.

Metric. We use DCT and DOR to evaluate the performance of frameworks. We consider one training epoch as one sampling duration and mark each epoch's start and end timestamps using *NVIDIA Tools Extension Library (NVTX)*[3] to measure the training time per epoch. We use a framework independent tool named *NVIDIA Nsight System*[4] to gather *GPU Metrics*. We disable other features like CUDA tracing to avoid additional overhead. We obtain DCT according to *GR Active*, which is one of the *GPU Metrics* and represents the percentage of cycles the graphics/compute engine is active. We calculate DOR according to (4), using measured DCT and training time per epoch. The first training step includes initializations like memory allocation. Therefore, we train models for six epochs but remove the first epoch to suppress the impact of initialization. We obtain the final results by averaging results from the remaining five epochs. For the BERT model, since the number of step is large, we load all the dataset and shuffle samples, and then we train 3200 samples as one epoch.

Implementation. We use our three-step equivalence validation and equivalence adjustment methods to ensure comparability. We demonstrate in Tables 3 and 4 that all the frameworks can achieve comparable prediction accuracy by

[2] https://www.docker.com/.

[3] https://docs.nvidia.com/nvtx/.

[4] https://developer.nvidia.com/nsight-systems.

Table 3 ResNet accuracy (%) shown as "mean ± std"

	ResNet-20	ResNet-56	ResNet-110
MXNet	92.10 ± 0.24	93.71 ± 0.31	94.14 ± 0.17
PaddlePaddle	91.97 ± 0.06	93.53 ± 0.11	94.18 ± 0.18
PyTorch	91.94 ± 0.20	93.68 ± 0.17	94.05 ± 0.20
TensorFlow 1	91.95 ± 0.16	93.59 ± 0.03	94.07 ± 0.13
TensorFlow 2	91.94 ± 0.30	93.70 ± 0.20	93.96 ± 0.18
Origin	91.25	93.03	93.57 ± 0.16

Table 4 BERT-base results shown as "mean ± std"

	Exact match (%)	F1 score (%)
MXNet	81.26 ± 0.18	88.60 ± 0.21
PaddlePaddle	81.21 ± 0.10	88.55 ± 0.09
PyTorch	81.16 ± 0.21	88.61 ± 0.24
TensorFlow 1	81.49 ± 0.17	88.78 ± 0.14
TensorFlow 2	81.21 ± 0.12	88.53 ± 0.11
Origin	80.8	88.5

modifying all the inequivalence implementations. The validation accuracy results are obtained by repeating the model training five times. We can observe that they have comparable precisions with a difference of no more than 0.35%. The reason for outperforming the result in the original paper may be attributed to the use of different weight initialization, post-process, and convolution downsampling methods. We can also obtain the coincident accuracy curves during training.

4.2 Comprehensive evaluation results

4.2.1 Comprehensive comparison by DCT and DOR

Figure 2 shows the performance of different frameworks using DCT and DOR. We examined three ResNets with a batch size of 512 and a BERT with a batch size of 32. We draw a dashed line in Fig. 2 to represent the least training time among the five frameworks as the training time equals $T(t_s, t_e)/R(t_s, t_e)$. We can observe that only the frameworks with a DCT below a certain threshold and a relatively high DOR can achieve optimal training performance. This is because DCT reflects the minimum value that training time can achieve, while DOR indicates how much potential the framework can reach. Figure 2 shows that PyTorch achieves superior training performance for ResNet-20 due to its high kernel function efficiency and ability to fully utilize the device. On the other hand, MXNet achieves the highest training performance for ResNet-56 and ResNet-110 owing to its most efficient kernel functions. Similarly, PyTorch achieves the highest BERT training performance due to its most efficient DCT.

Remark Maximizing the training performance requires enabling the framework to fully utilize the GPU to increase DOR and efficiently utilize the GPU to improve DCT.

4.2.2 Framework evaluation by DCT

Figure 3 and 4 shows the DCT results of training ResNet and BERT, respectively. We can observe that PaddlePaddle can achieve the lowest DCT with a small batch size. As the batch size increases, the DCT of PyTorch gradually becomes smaller than that of PaddlePaddle. For the convolution neural network, this is because PaddlePaddle also adopts built-in convolution operators in addition to *cuDNN*. The built-in convolution implementation is more efficient than that in the deep learning library when the computing complexity is low. However, it fails to fully utilize the GPU with large batch sizes. For the Transformer model, the difference in DCT is mainly due to the different implementations of permuting and matrix multiplication operators between

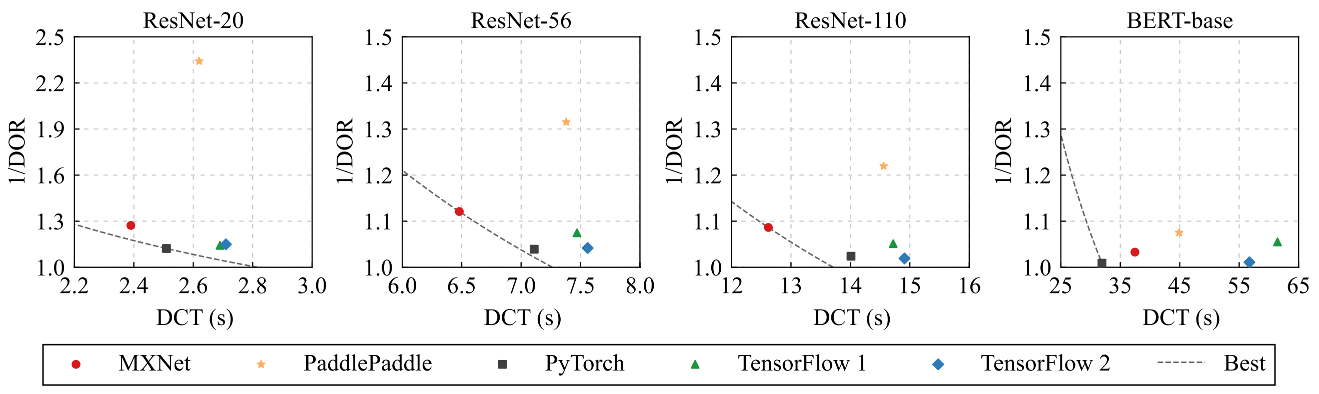


Fig. 2 Evaluation results of DCT(s) and 1/DOR. The dashed line denotes the best training time (computed by *DCT/DOR*) among these five frameworks

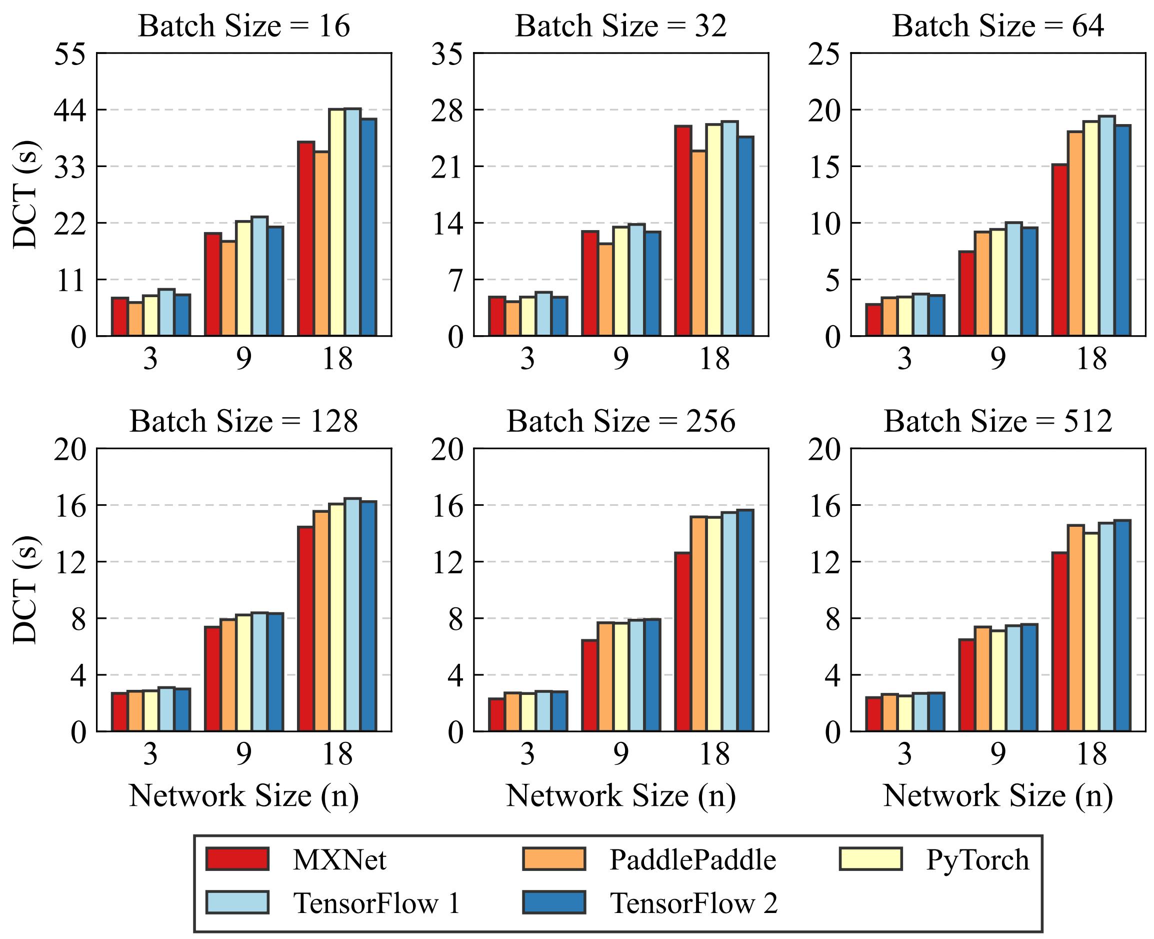


Fig. 3 Evaluation results of DCT(s) for ResNet-20 ($n = 3$), ResNet-56 ($n = 9$), and ResNet-110 ($n = 18$)

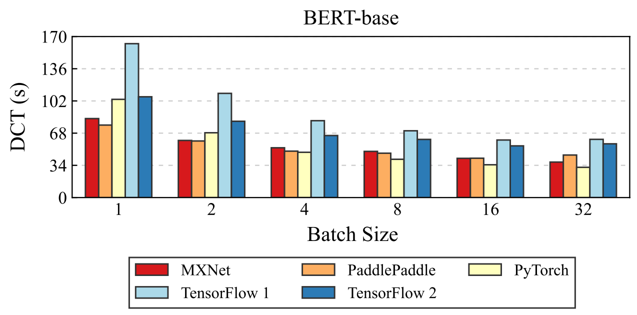


Fig. 4 Evaluation results of DCT(s) for BERT-base

the two frameworks. On the implementation of permuting, PyTorch adopts a "lazy" execution method, which only annotates tensors without explicitly executing permuting kernel functions. This method provides PyTorch with additional optimization space. For example, many libraries provide arguments on whether the matrix is transposed when executing GEMM. When conditions permit, GEMM can be used to complete tensor permuting, thereby reducing kernel function execution time. Especially when the batch size is large, the permuting operators have a greater time overhead, so PyTorch's DCT is better at large batch sizes. On the other hand, PaddlePaddle implements a kernel function named *MatrixColReduce* to assist in the backpropagation of matrix multiplication. Thus the computational efficiency of the matrix multiplication operator in PaddlePaddle is significantly better than PyTorch when the batch size is small.

From Fig. 3, we can also observe that there is a discernible difference in DCT between TensorFlow 1 and TensorFlow 2, particularly at small batch sizes. This is because the kernel functions mapped into by TensorFlow 1 and TensorFlow 2 are distinct for certain convolutions in forward computing and backward propagation. Hence, TensorFlow 2's kernel function computation efficiency outperforms TensorFlow 1's at lower workloads and performs comparably to TensorFlow 1 at higher workloads. We also observe that TensorFlow performs worse in terms of DCT compared to other frameworks. This is because TensorFlow does not provide the *OpT* setting and thus cannot choose the fastest convolution algorithm from various algorithms in *cuDNN*. For

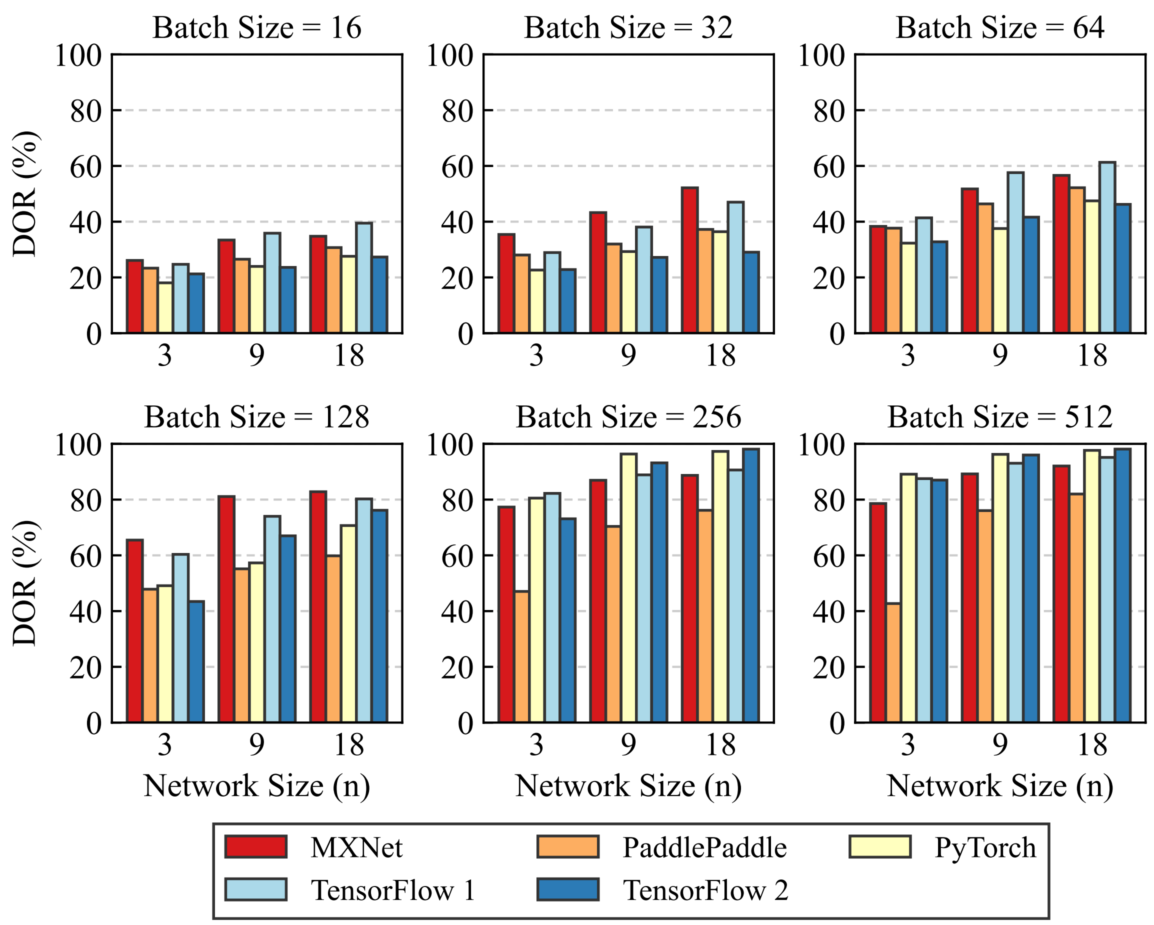


Fig. 5 Evaluation results of DOR(%) for ResNet-20 ($n = 3$), ResNet-56 ($n = 9$), and ResNet-110 ($n = 18$)

BERT in Fig. 4, the most kernel functions executed in Tensorflow 1, including matrix multiplication and element-wise operators, are based on *Eigen*. The matrix multiplication of TensorFlow 2 is implemented using *CUTLASS*,[5] while some operators are still based on *Eigen*.[6] Therefore, in some cases, the DCT of TensorFlow is poor.

With the large batch size, MXNet can leverage GPU more efficiently compared to other frameworks when training ResNet, as in Fig. 3. In addition to the use of *OpT*, there are two additional reasons: (1) MXNet employs a more efficient built-in element-wise kernel function for the ReLU operator; (2) MXNet does not need to clear gradients before backward propagation since MXNet writes into the gradients instead of accumulating them from zeros. In BERT training, the time cost of kernel functions for activation values and gradient clearing is relatively small. Due to PyTorch's "lazy" execution of permuting operators, PyTorch's DCT is better than MXNet at large batch sizes.

Remark PaddlePaddle has developed built-in kernel functions for convolutions to optimize performance resulting in higher device computing efficiency with small batch sizes. TensorFlow 2 achieves better DCT results for small batch sizes compared to its previous version. MXNet can utilize devices most efficiently through more efficient mechanisms like the built-in ReLU kernel function in convolution neural networks training. PyTorch can achieve the best DCT results while training Transformers at large batch sizes.

4.2.3 Framework evaluation by DOR

Figure 5 shows the DOR results of training ResNet with five frameworks at different batch sizes. We can observe that PyTorch has the lowest DOR for ResNet-20 training when the batch size is set to 16. This is because, with the eager execution mode, PyTorch cannot overlap its CPU control flow execution with GPU computing time. TensorFlow 2 can also not fully utilize the GPU for a lower

[5] https://github.com/NVIDIA/cutlass.

[6] https://eigen.tuxfamily.org.

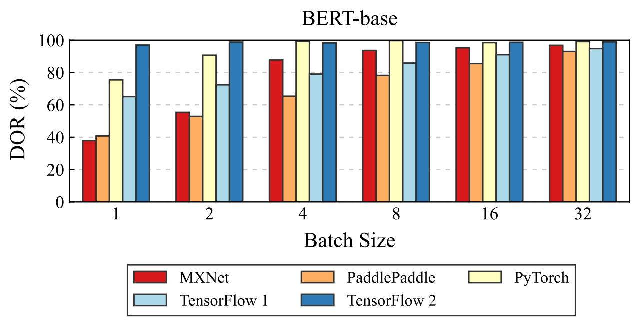


Fig. 6 Evaluation results of DOR(%) for BERT-base

workload. This could be due to the fact that the computation graph TensorFlow 1 built also includes the data fetch and preprocessing. TensorFlow 2 only converts the computation graph of a single training step to a static graph, resulting in high CPU execution overhead between steps. Besides, building the static computation graph through tracing will incur more time overhead at the beginning of each epoch. However, we can also observe that this situation is alleviated as the computational load increases. When the batch size is set to 512, it can be observed that PyTorch and TensorFlow 2 can better utilize GPU computation to hide the execution time overhead on the CPU. This situation also occurs in training models with higher computational loads, such as BERT-base. As shown in Fig. 6, for BERT-base, PyTorch and TensorFlow 2 can achieve higher DOR when the batch size is 1. Moreover, this advantage continues to exist as the batch size increa ses.

Figure 7 more clearly demonstrates the reasons for the difference in DOR by dividing the CPU-side execution time into six parts: data preprocessing, forward computation, gradient clearing, backward propagation, parameter updating, and others. Each part is further divided into two categories: those that overlap with GPU execution time (*-G* in Fig. 7) and those that do not overlap with GPU execution time (*-C*). TensorFlow 1 and TensorFlow 2 are not listed in the figure as they compile the entire step or epoch into a static graph. We can see that the gradient clearing and parameter updating stages have a low ratio of GPU usage, particularly at small batch sizes. This is because a large number of kernel functions generated from zero-setting operators during the gradient clearing stage have less computing time on the GPU. However, launching these kernel functions to the device requires significantly more CPU execution time than GPU computing time. Typically, frameworks execute kernel functions asynchronously. It means that the GPU computation in the forward pass stage can cover the CPU execution time in the gradient clearing stage with a sufficiently high workload. Similarly, frameworks can overlap the parameter updating stage and the subsequent data preprocessing stage with kernel functions launched in the backward propagation stage.

We can observe that PyTorch effectively hides the CPU overhead of the gradient update stage when using a batch size of 256. However, PaddlePaddle performs synchronization before each gradient clearing and parameter updating stage, leading to the lower DOR regardless of the large batch size. Finally, MXNet's main thread and threads responsible for dispatching operators are asynchronous. As a result, MXNet achieves the highest DOR when the computational

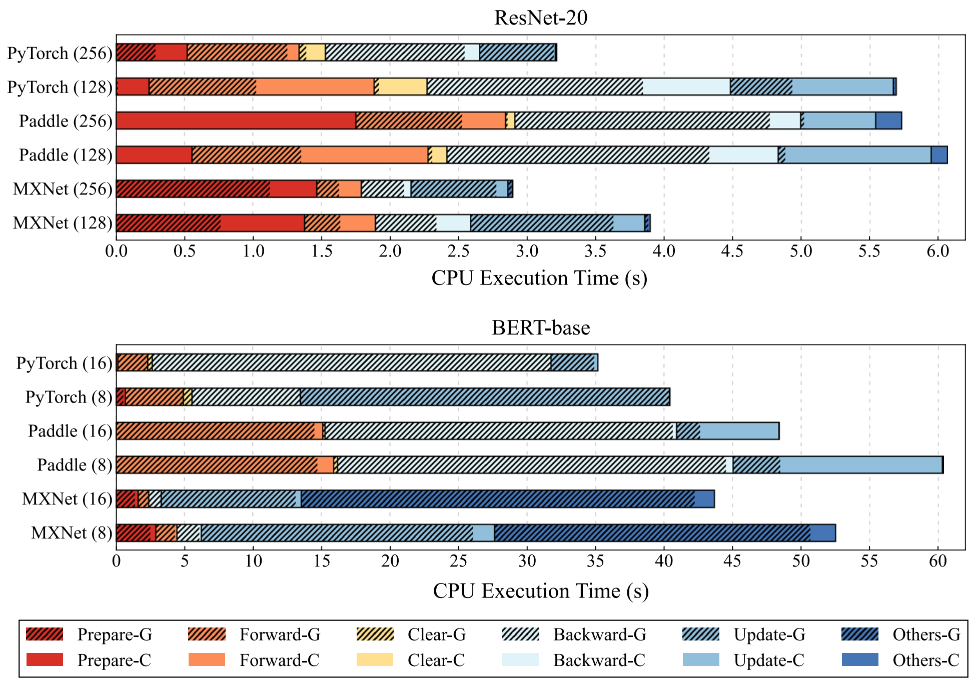


Fig. 7 Training time breakdown by stages for ResNet-20 and BERT-base training. We denote PaddlePaddle as "Paddle", and the labels for the *y*-axis are denoted as "*Framework (Batch size)*". For instance, "PyTorch (128)" represents the training time of PyTorch with a batch size of 128. "Prepare", "Forward", "Clear", "Backward", and "Update" represent the data preprocessing, forward pass, gradient clearing, backward propagation, and parameter updating stages, respectively. "-G" indicates the CPU execution time overlapped by GPU computing time, and "-C" otherwise

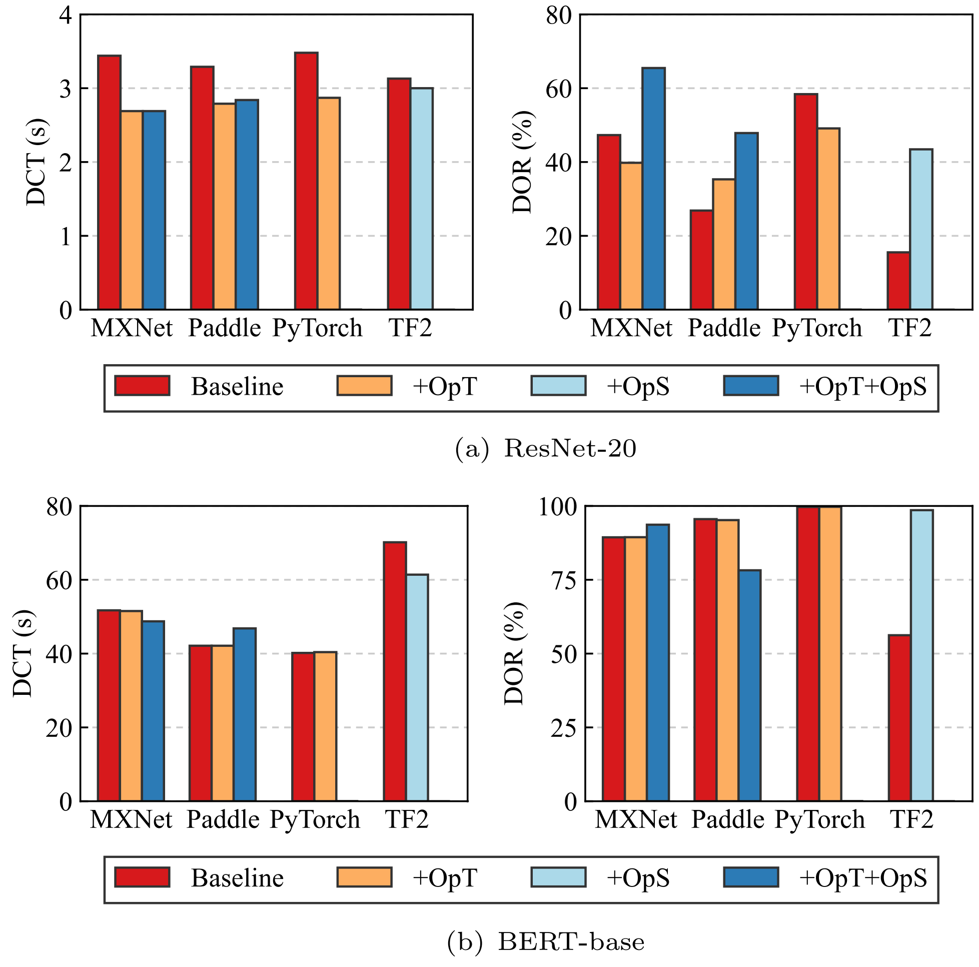


Fig. 8 The effect of optimization techniques in MXNet, PaddlePaddle (Paddle), PyTorch, and TensorFlow 2 (TF2) when training ResNet-20 and BERT-base. "Baseline" represents training without OpS and OpT. "+OpT" and "+OpS" denote training with OpT and OpS, respectively

load is small. However, in MXNet, the thread for dispatching operators synchronizes with the *stream* of the GPU, resulting in a lower DOR when the batch size is becoming larger, like 512.

Remark PyTorch underutilizes the device at a small computational load because of its eager execution mode. PaddlePaddle and MXNet have low DOR when the computational load is large, owing to their frequent synchronizations with the device and streams. For TensorFlow 2, the inefficient CPU execution between steps causes it to perform worse than the previous versions when the computational load is small. When the computational load is large, TensorFlow 2 effectively overlaps this CPU execution overhead with GPU computation, resulting in a better DOR than TensorFlow 1.

4.3 Evaluation of optimization techniques

Figure 8a shows the evaluation results of ResNet-20 training with the batch size set to 128. We can observe that turning on *OpT* can reduce DCT significantly in MXNet, PaddlePaddle, and PyTorch. This is because without *OpT*, the framework selects convolution algorithms and corresponding kernel functions through heuristic methods. The efficiency of heuristically selected kernel functions is slower than the kernel functions selected by running all possible convolution algorithms once. For MXNet and PyTorch, this optimization also deteriorates the DOR. This is because a faster convolutional algorithm reduces DCT, while their performance bottleneck comes from the CPU execution time of operator dispatching when the batch size is set to 128. In models without convolutional layers, *OpT* will not have an impact on training performance, as shown in Fig. 8b.

When training ResNet, we can also observe that *OpS* shortens the DCT only in TensorFlow 2, while the other two frameworks cannot achieve better DCT with static execution mode. It is because ResNet has limited space for graph optimizations. For example, batch normalization is applied immediately after each convolution and before activation in ResNet. As a result, the remapper optimization can not replace the subgraph of convolution and activation with an optimized fused kernel. TensorFlow establishes the entire training step as a graph instead of building a graph only for the neural network, which provides

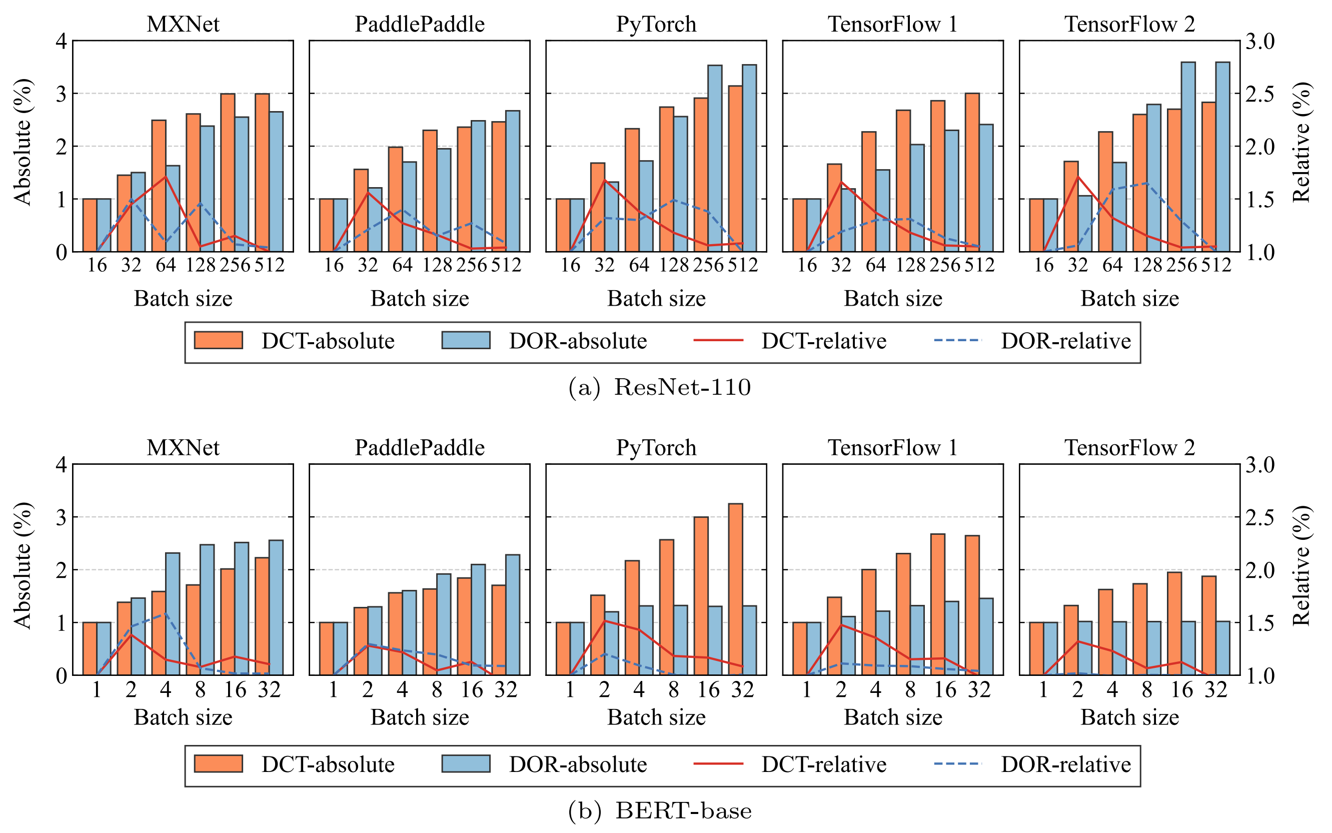


Fig. 9 Improvements of DCT and DOR for ResNet-110 training in PyTorch, PaddlePaddle, TensorFlow 1, and TensorFlow 2. Bars show improvement based on the performance at a batch size of 16. Lines show improvement relative to the performance at (batch size)/2 for a certain batch size

additional optimization space. For BERT's training tasks, both MXNet and TensorFlow 2 can find graph optimization space and reduce DCT. In PaddlePaddle, although some kernel functions have shortened their time, some operators, such as permuting, have added memory copy operators from device to device, resulting in an increase in DCT time. We can also observe the improvements in DOR for all three frameworks since static execution reduces the time cost of operator dispatching. One exception is PaddlePaddle when training BERT, as it synchronizes the device after the forward computing and backpropagations when applying *OpS*, resulting in a decrease in DOR, as analyzed in the previous section.

Remark Using *OpT* can improve GPU computing efficiency in any framework when training convolution neural networks. Besides, *OpS* can help frameworks focus less on improving the dispatching efficiency under eager execution mode. However, applying *OpS* will not improve the kernel function efficiency in MXNet and PaddlePaddle if a neural network has limited graph optimization space.

4.4 Performance at different hyper-parameters

4.4.1 Effect of the batch size

Figure 9 shows the DCT and DOR improvement obtained by the five frameworks at different batch sizes for ResNet-110. We can observe that the improvement trend of DCT is similar among all frameworks. One exception is MXNet, which uses different convolutional algorithms at different batch sizes. However, the improvement trend of DOR varies among frameworks. For example, we can observe that PyTorch, PaddlePaddle, and TensorFlow 2 still have a relative improvement above 1.25× at a batch size of 256.

The batch size increase can benefit the framework performance twofold. Firstly, due to the massive cores in the GPU architecture, a kernel function with insufficient computation required will result in the underutilization of the GPU. The batch size increase can enhance the kernel computing efficiency on GPU and reduces the DCT intensely. Since the computation required for a single kernel function to fully utilize computing cores in GPU is almost independent of the framework, similar DCT improvement trends can be observed in different frameworks. Secondly, the growth of

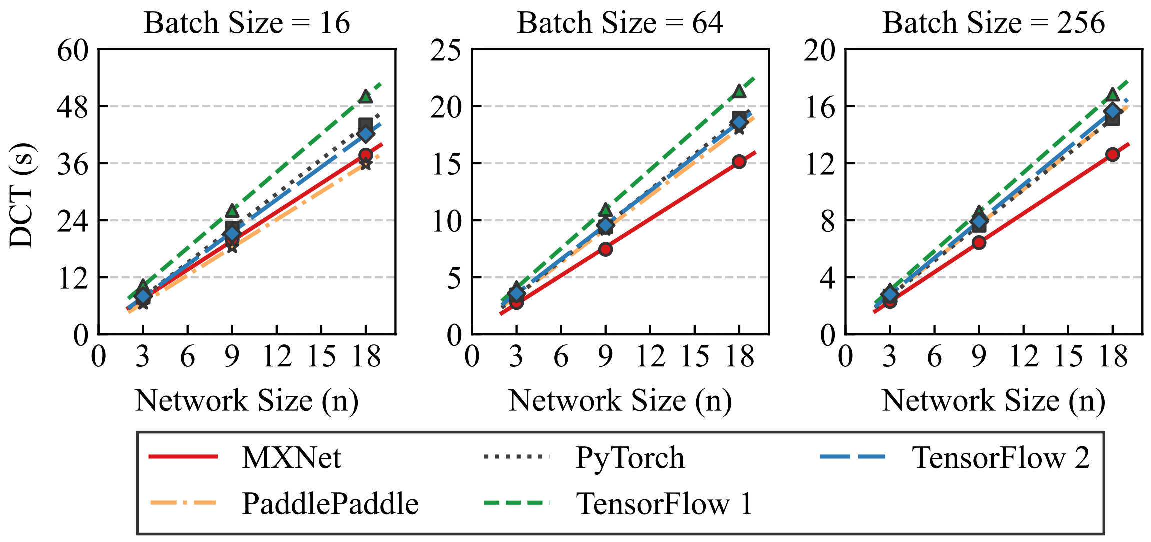


Fig. 10 The relationship between network size n and DCT at batch sizes of 16, 64, and 256, fitting with linear regression

batch size increases the computation requirement of operators in the forward computing and backward propagation stages. As the batch size increases, the time to execute kernel functions on GPU grows, thus masking more CPU execution time and improving DOR. Therefore, if a framework has a lower DOR at a small batch size, it will have a longer CPU execution time for dispatching operators and require a larger batch size to improve DOR.

Remark For all frameworks, the required batch size to achieve optimal DCT is similar. Besides, the more efficient the operators dispatching in a framework, the smaller the batch size required to achieve optimal performance.

4.4.2 Effect of the network size

By comparing different network sizes in Fig. 3, we can see that the DCT rises with the increase of the network size n. Figure 10 further shows the DCT results of all five frameworks at batch sizes 16, 64, and 256. The lines in the figure show the linear regression for modeling the relationship between network size n and DCT. We can observe that linear regression can fit the relationship between n and DCT well, regardless of the framework or batch size. Theoretically, DCT is linearly related to n since $6n + 2$ represents the number of operators in ResNet. When n increases by 1, the convolutional, batch normalization, and ReLU layers in ResNet increase by 6, making DCT grow by a fixed value. Therefore, DCT and n exhibit a linear relationship in all cases.

For DOR, we can observe in Fig. 5 that all frameworks improve with the increase in network size. There are two situations. (1) When the batch size is small, the kernel functions launched in forward computation and backward propagation are not enough to overlap with CPU execution time for gradient clearing and parameter updating. Increasing network size reduces the proportion of the data processing stage, which does not change with network sizes and has low DOR. (2) When the batch size is large, increasing network size helps more CPU execution time that can be overlapped to be overlapped, thereby improving DOR. Besides, for PaddlePaddle, the reason for DOR improvement is always the first situation due to its frequent synchronization with the device.

Remark There is a linear relationship between DCT and network size n for all frameworks. Besides, a larger network size is more advantageous in hiding CPU execution overhead.

4.5 Discussion

4.5.1 Summary

We summarize the above analysis in Table 5, including eight aspects: convolution kernel function computation efficiency (Conv in the table), ReLU kernel function computation efficiency (ReLU), *OpT*, *OpS*, gradient clearing (Grad clear), for which to synchronize (Sync target) and the frequency of synchronization (Sync freq). We also consider two cases: low workload (LW) and high workload (HW). These mechanisms are reflected in DCT and DOR and ultimately have an impact on training time (Time).

We also highlight the following conclusions. (1) We observe that the performance of using deep learning libraries is sometimes lower than built-in operators. This implies that

Table 5 Evaluation results with different mechanisms

	MXNet	PaddlePaddle	PyTorch	TensorFlow 1	TensorFlow 2
Conv (LW)	Avg	**Good**	Avg	Avg	**Good**
Conv (HW)	**Good**	Avg	**Good**	Avg	Avg
ReLU	**Good**	Avg	Avg	Avg	Avg
GEMM (LW)	Avg	**Good**	Avg	Poor	Avg
GEMM (HW)	Avg	Avg	**Good**	Poor	Avg
Lazy Permuting	No	No	**Yes**	No	No
OpT	**Yes**	**Yes**	**Yes**	No	No
OpS	**Yes**	**Yes**	No	/	**Yes**
Grad clear	**No**	Yes	Yes	**No**	**No**
Sync target	Stream	Device	Device	Device	Device
Sync freq	High	Medium	**Low**	**Low**	**Low**
DCT (LW)	Avg	**Good**	Avg	Avg	**Good**
DOR (LW)	**Good**	Avg	Poor	**Good**	Poor
DCT (HW)	**Good**	Avg	**Good**	Avg	Avg
DOR (HW)	Avg	Poor	**Good**	**Good**	**Good**
Time (LW)	**Good**	Avg	Poor	Avg	Poor
Time (HW)	**Good**	Poor	**Good**	Avg	Avg

Texts in bold indicate the superior performance or mechanisms

it is challenging to design a high-performance deep learning library that covers as many situations as possible. (2) DCT can be improved by testing all the convolution algorithms and selecting the fastest one. This indicates that it is challenging to choose the optimum convolution algorithm by predicting the performance. (3) Compared to the eager execution mode, the static execution mode does not significantly improve the computational efficiency of operators when training convolution neural networks like ResNet.

For different hyper-parameters, we first pointed out that some frameworks require larger batch sizes to achieve optimal DOR compared to other frameworks. Considering a too-large batch size can negatively impact the quality of the model (Goyal et al. 2017; Smith et al. 2020), the degree of parallelism for data parallelism in such frameworks may be more restricted. Secondly, we find that DCT can establish a good relationship with network size across all frameworks. Considering that the DOR is close to 1 under high workloads, it is hopeful to use DCT to estimate the minimum training time for larger networks.

4.5.2 Other devices

In this section, we apply our metrics to another hardware device to determine the optimization direction of model training. The hardware environment is equipped with two Intel (R) Xeon (R) Gold 5218R CPUs @ 2.10 GHz and one Atlas 300T. The Atlas 300T (NPU) is a specialized device developed by Huawei to accelerate deep learning computing. Huawei also developed deep learning frameworks that can perform training on NPUs. For now, they support PyTorch 1.8 and TensorFlow 2.6 running on NPUs. We use these two frameworks to perform ResNet training on CIFAR-10, collecting kernel function execution start and end times, as well as epoch time, through the dedicated tool named *msprof*. Due to support issues, we preprocess input data serially during training on TensorFlow 2.

The DCT and DOR results are shown in Table 6. Firstly, comparing the two frameworks, it can be observed that PyTorch has a lower DCT. This is mainly because PyTorch and TensorFlow 2 use different kernel functions to calculate Batch Normalization operators. The TensorFlow 2's DOR is significantly lower than PyTorch because it performs data preprocessing in a serial manner, resulting in a longer idle time for the NPU. Compared with the results obtained with the NVIDIA A6000, it can be seen that all DCTs have achieved more than twice the acceleration. The main reason is that NPU contains a large number of FP16 computing units, so the entire training process uses mixed precision training. In addition, both PyTorch and TensorFlow 2 have lower DOR results. This means that operator distribution

Table 6 ResNet training on Atlas 300T

	DCT	DOR
ResNet-20(PyTorch)	1.32s	5.59%
ResNet-56(PyTorch)	3.43s	6.81%
ResNet-110(PyTorch)	6.58s	7.06%
ResNet-20(TensorFlow 2)	1.40s	0.29%
ResNet-56(TensorFlow 2)	3.81s	0.77%
ResNet-110(TensorFlow 2)	7.42s	1.50%

has become the main bottleneck in training. Therefore, in order to further improve training performance, the next goal should be to maximize the operator distribution speed of the framework.

5 Conclusion

In this paper, we provide a method for evaluating deep learning frameworks, including evaluation metrics, equivalence validation, and equivalence adjustment. We introduce DCT and DOR to evaluate the kernel function computation efficiency and operator dispatching efficiency, respectively. DCT and DOR can decouple the impact of different mechanisms on training performance, enabling the effect of each mechanism to be effectively reflected in the evaluation results. We propose a three-step equivalence validation method, called hyperparameter, model, and parameter updating equivalence, to ensure all the frameworks execute the same computation graph and discover inequivalence implementations in frameworks. We further propose a method to adjust these inequivalence implementations based on the number of operators. We evaluate PyTorch, MXNet, PaddlePaddle, TensorFlow 1, and TensorFlow 2 and uncover the reasons for the performance gap among these frameworks, which can help better utilize frameworks and optimize training performance on heterogeneous computing. The source code can be found at https://github.com/LuZhengx/DLFrameBench.

Acknowledgements This work is partially supported by the National Natural Science Foundation (62272248), the National Key Research and Development Program of China (2018YFB2100300), the Open Project Fund of State Key Laboratory of Computer Architecture, Institute of Computing Technology, Chinese Academy of Sciences (CARCHA202108), the Natural Science Foundation of Tianjin of China (21JCZDJC00740, 21JCYBJC00760), the Key Research Project of Zhejiang Lab (2022PG0AC02), and the Key R &D Program of Zhejiang (2022C04006).

Data availability The data underlying this article are available in the article. The article includes citations for the datasets that were used. The code used in the experiments is open source on GitHub and can be accessed through https://github.com/LuZhengx/DLFrameBench.

Declarations

Conflict of interest All authors certify that they have no affiliations with or involvement in any organization or entity with any financial interest or non-financial interest in the subject matter or materials discussed in this manuscript.

References

Abadi, M., Barham, P., Chen, J., et al.: TensorFlow: a system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16), pp. 265–283. USENIX Association, Savannah, GA (2016)
Adolf, R., Rama, S., Reagen, B., et al.: Fathom: reference workloads for modern deep learning methods. In: 2016 IEEE International Symposium on Workload Characterization (IISWC), pp. 1–10 (2016). https://doi.org/10.1109/IISWC.2016.7581275
Al-Rfou, R., Alain, G., Almahairi, A., et al.: Theano: A Python Framework for Fast Computation of Mathematical Expressions. arXiv e-prints pp arXiv-1605 (2016)
Chen, T., Li, M., Li, Y., et al.: Mxnet: A Flexible and eFficient Machine Learning Library for Heterogeneous Distributed Systems. arXiv preprint arXiv:1512.01274 (2015)
Coleman, C., Kang, D., Narayanan, D., et al.: Analysis of dawnbench, a time-to-accuracy machine learning performance benchmark. ACM SIGOPS Oper. Syst. Rev. **53**(1), 14–25 (2019)
Devlin, J., Chang, M.W., Lee, K., et al.: Bert: Pre-training of deep bidirectional transformers for language understanding. In: Proceedings of NAACL-HLT. Association for Computational Linguistics, pp. 4171–4186 (2019)
Elshawi, R., Wahab, A., Barnawi, A., et al.: Dlbench: a comprehensive experimental evaluation of deep learning frameworks. Clust. Comput. **24**, 2017–2038 (2021)
Glorot, X., Bengio, Y.: Understanding the difficulty of training deep feedforward neural networks. In: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, JMLR Workshop and Conference Proceedings, pp. 249–256 (2010)
Goyal, P., Dollár, P., Girshick, R., et al.: Accurate, Large Minibatch sgd: Training Imagenet in 1 hour. arXiv preprint arXiv:1706.02677 (2017)
Guo, Q., Chen, S., Xie, X., et al.: An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms. In: 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 810–822. IEEE (2019)
Han, J., Deng, S., Lo, D., et al.: An empirical study of the dependency networks of deep learning libraries. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 868–878. IEEE (2020a)
Han, J., Shihab, E., Wan, Z., et al.: What do programmers discuss about deep learning frameworks. Empir. Softw. Eng. **25**, 2694–2747 (2020b)
He, K., Zhang, X., Ren, S., et al.: Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In: Proceedings of the IEEE International Conference on Computer Vision, pp. 1026–1034 (2015)
He, K., Zhang, X., Ren, S., et al.: Deep residual learning for image recognition. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 770–778 (2016)
Jäger, S., Zorn, H.P., Igel, S., et al.: Parallelized training of deep nn: comparison of current concepts and frameworks. In: Proceedings of the Second Workshop on Distributed Infrastructures for Deep Learning, pp. 15–20 (2018)
Jia, Y., Shelhamer, E., Donahue, J., et al.: Caffe: Convolutional architecture for fast feature embedding. In: Proceedings of the 22nd Acm International Conference on Multimedia, pp. 675–678 (2014)
Kim, H., Nam, H., Jung, W., et al.: Performance analysis of cnn frameworks for gpus. In: 2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 55–64. IEEE (2017)
Krizhevsky, A., Hinton, G., et al.: Learning Multiple Layers of Features From Tiny Images (2009)
Li, M., Andersen, D.G., Park, J.W., et al.: Scaling distributed machine learning with the parameter server. In: 11th USENIX

Symposium on Operating Systems Design and Implementation (OSDI 14), pp. 583–598 (2014)

Liu, L., Wu, Y., Wei, W., et al.: Benchmarking deep learning frameworks: Design considerations, metrics and beyond. In: 2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS), pp. 1258–1269. IEEE (2018)

Ma, Y., Yu, D., Wu, T., et al.: Paddlepaddle: an open-source deep learning platform from industrial practice. Front. Data Comput. **1**(1), 105–115 (2019)

Mahmoud, N., Essam, Y., Elshawi, R., et al.: Dlbench: an experimental evaluation of deep learning frameworks. In: 2019 IEEE International Congress on Big Data (BigDataCongress), pp. 149–156. IEEE (2019)

Makkouk, T., Kim, D.J., Chen, T.H.P.: An empirical study on performance bugs in deep learning frameworks. In: 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 35–46. IEEE (2022)

Mattson, P., Cheng, C., Diamos, G., et al.: Mlperf training benchmark. Proc. Mach. Learn. Syst. **2**, 336–349 (2020a)

Mattson, P., Reddi, V.J., Cheng, C., et al.: Mlperf: an industry standard benchmark suite for machine learning performance. IEEE Micro **40**(2), 8–16 (2020b)

Paszke, A., Gross, S., Massa, F., et al.: Pytorch: an imperative style, high-performance deep learning library. Adv. Neural Inf. Process. Syst. **32**, 8026 (2019)

Rajpurkar, P., Zhang, J., Lopyrev, K., et al.: SQuAD: 100,000+ questions for machine comprehension of text. In: Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing. Association for Computational Linguistics, Austin, TX, pp. 2383–2392 (2016). https://doi.org/10.18653/v1/D16-1264

Reddi, V.J., Cheng, C., Kanter, D., et al.: Mlperf inference benchmark. In: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), pp. 446–459. IEEE (2020)

Reddi, V.J., Cheng, C., Kanter, D., et al.: The vision behind mlperf: understanding AI inference performance. IEEE Micro **41**(3), 10–18 (2021)

Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning representations by back-propagating errors. Nature **323**(6088), 533–536 (1986)

Seide, F., Agarwal, A.: Cntk: microsoft's open-source deep-learning toolkit. In: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, pp. 2135–2135 (2016)

Shams, S., Platania, R., Lee, K., et al.: Evaluation of deep learning frameworks over different HPC architectures. In: 2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS), pp. 1389–1396. IEEE (2017)

Shi, S., Wang, Q., Xu, P., et al.: Benchmarking state-of-the-art deep learning software tools. In: 2016 7th International Conference on Cloud Computing and Big Data (CCBD), pp. 99–104. IEEE (2016)

Shi, S., Wang, Q., Chu, X.: Performance modeling and evaluation of distributed deep learning frameworks on Gpus. In: 2018 IEEE 16th Intl Conf on Dependable, Autonomic and Secure Computing, 16th Intl Conf on Pervasive Intelligence and Computing, 4th Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech), pp. 949–957. IEEE (2018)

Smith, S., Elsen, E., De, S.: On the generalization benefit of noise in stochastic gradient descent. In: International Conference on Machine Learning, PMLR, pp. 9058–9067 (2020)

Sun, S., Cao, Z., Zhu, H., et al.: A survey of optimization methods from a machine learning perspective. IEEE Trans. Cybern. **50**(8), 3668–3681 (2019)

Sun, X., Zhou, T., Wang, R., et al.: Experience report: investigating bug fixes in machine learning frameworks/libraries. Front. Comp. Sci. **15**, 1–16 (2021)

Tang, F., Gao, W., Zhan, J., et al.: Aibench training: Balanced industry-standard ai training benchmarking. In: 2021 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 24–35. IEEE (2021)

Trindade, R.G., Lima, J.V.F., Charão, A.S.: Performance evaluation of deep learning frameworks over different architectures. In: High Performance Computing for Computational Science—VECPAR 2018, pp. 92–104. Springer International Publishing, Cham (2019)

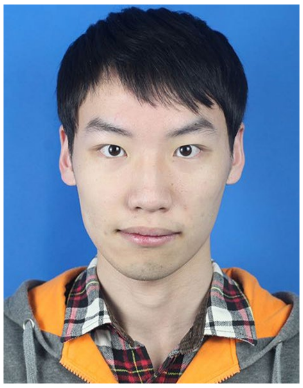
Wu, Y., Cao, W., Sahin, S., et al.: Experimental characterizations and analysis of deep learning frameworks. In: 2018 IEEE International Conference on Big Data (Big Data), pp. 372–377. IEEE (2018)

Xie, X., He, W., Zhu, Y., et al.: Performance evaluation and analysis of deep learning frameworks. In: Proceedings of the 2022 5th International Conference on Artificial Intelligence and Pattern Recognition. Association for Computing Machinery, New York, NY, USA, AIPR '22, pp. 38–44. https://doi.org/10.1145/3573942.3573948(2023)

Yang, C.T., Liu, J.C., Chan, Y.W., et al.: Performance benchmarking of deep learning framework on intel xeon phi. J. Supercomput. **77**, 2486–2510 (2021)

Yuan, J., Li, X., Cheng, C., et al.: Oneflow: Redesign the Distributed Deep Learning Framework from Scratch (2021). arXiv preprint arXiv:2110.15032

Zhu, H., Akrout, M., Zheng, B., et al.: Benchmarking and analyzing deep neural network training. In: 2018 IEEE International Symposium on Workload Characterization (IISWC), pp. 88–100. IEEE (2018)



Zhengxian Lu received his B.S. degree from Nankai University in 2020. Since 2020, he has been working toward his Ph.D. degree at the College of Computer Science, Nankai University. His main research interests include heterogeneous computing and systems for machine learning.

Chengkun Du received his B.S. degree from Nankai University in 2020 and his M.S. degree from Nankai University in 2023. He is currently working at Huawei Technologies Co., Ltd. His main research interests include deep learning and AI frameworks.

Yanfeng Jiang received his B.S. degree in 2021 from Chongqing University. He is now studying at Nankai University for a Ph.D. degree in computer science and technology. His main research interests are deep learning and systems.

Xueshuo Xie is currently an associate researcher at Haihe Lab of ITAI, received a postdoctoral in the College of Computer Science, Nankai University, received a Ph.D. degree in engineering from Nankai University in 2021, and received a B.S. and MA. SC degree from Shandong University, Jinan, China, in 2011 and 2014. He is currently working at the Intelligent Computing System Lab, College of Cyber-Science, Nankai University, Tianjin, China. His current research interests include IoT security, data-driven anomaly detection, and Blockchain.

Tao Li received his Ph.D. degree in Computer Science from Nankai University, China, in 2007. He works at the College of Computer Science, Nankai University, as a Professor. He is a Member of the IEEE Computer Society and the ACM and a distinguished member of the CCF. His main research interests include heterogeneous computing, Intelligent IoT, and Blockchain systems.

Fei Yang received a B.S. and M.S. degree in computer science from Shanghai Jiao Tong University in 2011 and 2014, and the Ph.D. degree in computer science from Eindhoven University of Technology, the Netherlands, in 2018. From 2019 to 2020, He worked as a research fellow at the Cyber Security Lab in the Department of Computer Science and Engineering at Nanyang Technological University, Singapore. Since 2020, he has worked at Zhejiang Lab as an advanced research specialist. His major research interest at Zhejiang Lab includes deep learning framework, distributed computing technique, and intelligent computing platform. He is the software architect of the Digital Reactor OS project.