Nondeterministic Impact of CPU Multithreading on Training Deep Learning Systems

Guanping Xiao*†⋆, Jun Liu*, Zheng Zheng‡, Yulei Sui§
*College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, China
†State Key Laboratory of Novel Software Technology, Nanjing University, China
‡School of Automation Science and Electrical Engineering, Beihang University, China
§School of Computer Science, University of Technology Sydney, Australia
{gpxiao, 161730324}@nuaa.edu.cn, zhengz@buaa.edu.cn, yulei.sui@uts.edu.au

***Abstract*—With the wide deployment of deep learning (DL) systems, research in reliable and robust DL is not an option but a priority, especially for safety-critical applications. Unfortunately, DL systems are usually nondeterministic. Due to software-level (e.g., randomness) and hardware-level (e.g., GPUs or CPUs) factors, multiple training runs can generate inconsistent models and yield different evaluation results, even with identical settings and training data on the same implementation framework and hardware platform. Existing studies focus on analyzing software-level nondeterminism factors and the nondeterminism introduced by GPUs. However, the nondeterminism impact of CPU multithreading on training DL systems has rarely been studied. To fill this knowledge gap, we present the first work of studying the variance and robustness of DL systems impacted by CPU multithreading. Our major contributions are fourfold: 1) An experimental framework based on VirtualBox for analyzing the impact of CPU multithreading on training DL systems; 2) Six findings obtained from our experiments and examination on GitHub DL projects; 3) Five implications to DL researchers and practitioners according to our findings; 4) Released the research data (https://github.com/DeterministicDeepLearning).**

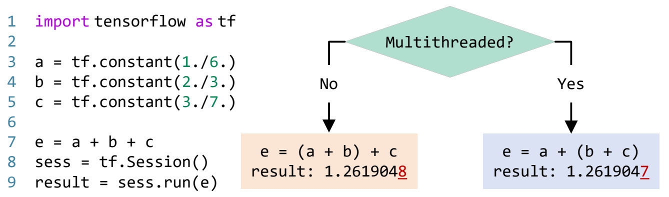
***Index Terms*—Deep learning systems, CPU multithreading, nondeterminism factors, training variance, empirical study**

I. Introduction

Deep learning (DL) is widely used in various domains, such as computer vision [1], natural language processing [2], virtual assistants [3], and many tasks in software engineering [4]–[10]. A reliable DL system is crucial, especially for safety-critical applications, such as self-driving cars [11]–[13] and medical diagnosis [14]–[16].

For reproducibility and stability purposes of using DL systems, it is expected that DL systems can have a *deterministic* behavior in identical training runs, where each training is conducted using identical data, experimental settings, neural networks on the same implementation framework and hardware platform. Such deterministic behavior is defined as follows: under a fixed software-level and hardware-level experimental condition, multiple training runs produce the same model and result in identical evaluation results.

Unfortunately, DL systems are usually *nondeterministic*, i.e., multiple identical training runs can generate inconsistent models and yield different evaluation results. This is due to the factors introduced by software-level and hardware-level nondeterminism [17], [18]. For software-level factors, model training settings (e.g., weight initialization [19], shuffled batch ordering [20]) and implementation frameworks (e.g., TensorFlow [21], PyTorch [22], and Keras [23]) are the primary two factors. For example, training settings in user code and several operations in DL frameworks often rely on random generators (e.g., Python random and Numpy random) [19], [24]–[26]. The variance introduced by such randomness has a major impact on the software-level nondeterminism.


```
import tensorflow as tf

a = tf.constant(1./6.)
b = tf.constant(2./3.)
c = tf.constant(3./7.)

e = a + b + c
sess = tf.Session()
result = sess.run(e)
```

```
e = (a + b) + c
result: 1.2619048
```
```
e = a + (b + c)
result: 1.2619047
```
Fig. 1. Floating-point difference impacted by different computing orders.

In addition to software-level factors, hardware environments, such as GPU or CPU platforms for training DL systems, can also introduce nondeterminism [17], mainly due to parallel computations. The nondeterminism factor introduced by GPU to the training of DL systems has been investigated by existing work [17], [18].

Although GPUs have a powerful concurrent computations ability, CPUs have several advantages such as high memory capacity [27], [28], usefulness in mobile systems [29] and some extreme environments, e.g., space and defense industries [30]. Therefore, training DL systems on CPUs is increasingly popular in research and industry fields [31]. For example, using HPC with CPU cluster to training DL systems [32], [33].

Fig. 1 shows an example of floating point difference in TensorFlow on a CPU platform. If we perform execution under the CPU multithreaded environment[1], the expression such as `e = a + b + c` can be computed in two different ways: (1) `a + b` is computed first following a sequential order, obtaining the result 1.2619048 or (2) `b + c` is computed first under the multithreading scenario, which produces a different result 1.2619047, due to floating point imprecision. Multithreading is quite common in modern DL systems. TensorFlow by default

⋆Corresponding author: Guanping Xiao.

[1]This simple example is used to illustrate different results calculated by different orders, and it is not necessarily executed in parallel under a real setting.

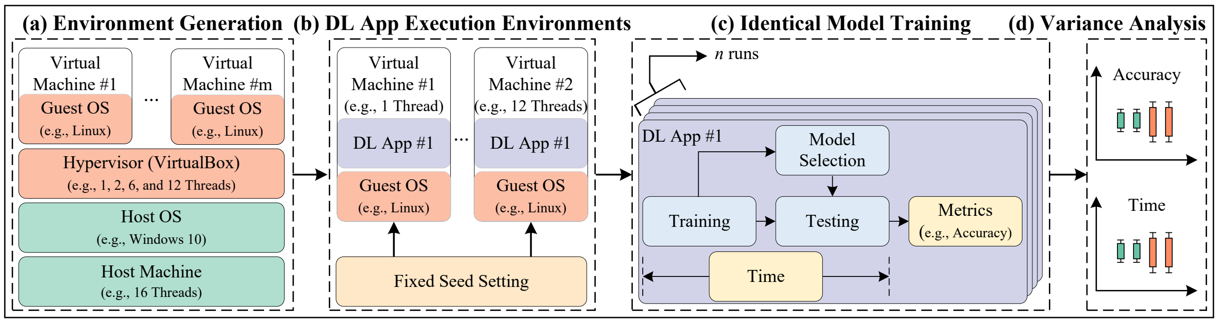

(a) Environment Generation
Virtual Machine #1
Guest OS (e.g., Linux)
...
Virtual Machine #m
Guest OS (e.g., Linux)
Hypervisor (VirtualBox) (e.g., 1, 2, 6, and 12 Threads)
Host OS (e.g., Windows 10)
Host Machine (e.g., 16 Threads)
(b) DL App Execution Environments
Virtual Machine #1 (e.g., 1 Thread)
DL App #1
Guest OS (e.g., Linux)
Virtual Machine #2 (e.g., 12 Threads)
DL App #1
Guest OS (e.g., Linux)
Fixed Seed Setting
(c) Identical Model Training
n runs
DL App #1
Model Selection
Training
Testing
Metrics (e.g., Accuracy)
Time
(d) Variance Analysis
Accuracy
Time

Fig. 2. Overview of our experimental framework.

uses multiple threads to execute independent operations [34]. The parallel execution of independent operations enables users to exploit CPU parallelism to conduct fast model training [35].

Existing studies focus on optimizing DL training efficiency on CPU platforms [36]. However, the nondeterministic impact of CPU multithreading on training DL systems has rarely been studied. The adverse impact of CPU multithreading on the nondeterminism (i.e., effectiveness variance) in training DL systems remains unknown. Floating-point rounding errors are well known by practitioners, but their impact on training DL systems is often overlooked. For example, one answer from StackOverflow stated that [37], *"However those differences are extremely small and will not effect the overall result in most cases."*

In this paper, we aim to conduct a systematic study to understand the nondeterministic impact on training DL systems under CPU multithreading. We evaluated five GitHub DL projects (TABLE IV), covering two types of fundamental neural network architectures (CNNs and RNNs), and three popular DL frameworks (TensorFlow, PyTorch, and Keras). After controlling software-level nondeterminism factors by setting fixed seed for random generators, for the TensorFlow projects (voicy-ai/DialogStateTracking [38], zjy-ucas/ChineseNER [39]) and PyTorch project (castorini/honk [40]), multiple identical training runs are able to produce deterministic evaluation results under the same number of CPU threads (Section IV-A). However, the results become nondeterministic when performing identical training using a different number of CPU threads. Moreover, for fixed-seed identical training runs without model selection (i.e., using the model trained after each epoch), CPU multithreading causes accuracy variance as large as 8.56%. Such accuracy variance can be minimized by using model selection techniques (e.g., select the model with the best validation accuracy). This implies that the impact of CPU multithreading on the effectiveness variance in training DL systems is substantial and not negligible. It is necessary to develop some mitigation techniques to minimize or eliminate the nondeterminism during model training.

In addition, our findings (Section IV-B) on time variance in training DL systems show that imposing more CPU threads does not necessarily mean we can train the model faster. For the examined five projects, four projects achieved the fastest training efficiency with only 2 CPU threads. Increasing more threads will not get faster training. Only one project (awni/ecg [41]) shows that the training time can be shortened when increasing the number of threads (e.g., from 1 to 12). This reveals that optimizing training efficiency on CPU platforms can not simply rely on the number of threads used [35].

To study whether DL developers are aware of hardware requirements for training and evaluating their DL projects, we examined 245 open-source DL projects from GitHub (Section II) to investigate the proportions of projects that have specified software requirements (e.g., dependency libraries and versions) and hardware requirements (e.g., detailed GPU or CPU models). The results (Section IV-C) show that among the 245 examined projects, 90.6% of them provide software requirements, while only 22.0% mention hardware requirements for model training and effectiveness evaluation.

Therefore, it is important to study the nondeterministic impact of hardware CPU multithreading on training DL systems. On the one hand, when developing DL systems, practitioners should not only focus on a model's performance during training, but also need to pay attention to the effectiveness variance under different CPU multithreaded environments, e.g., training with different numbers of CPU threads. On the other hand, since CPU multithreading may impact the effectiveness variance in training DL systems, researchers should mention CPU models used for the training when releasing their DL projects, if the training is performed on CPU platforms.

Attempts to fill this gap, we design an experimental framework (Fig. 2) based on Oracle VM VirtualBox [42], to generate four types of virtual machines (VMs) with different numbers of CPU threads (i.e., 1, 2, 6, and 12). We conduct three categories of identical training runs: (1) the default runs with model selection, (2) fixed-seed runs with model selection and (3) runs without model selection, under different multithreaded environments (VMs). Our study mainly focuses on answering the following three research questions (RQs).

- RQ1: What's the impact of CPU multithreading on the effectiveness variance (e.g., accuracy) in training DL systems?
- RQ2: What's the impact of CPU multithreading on the time variance in training DL systems?
- RQ3: How many collected DL projects in GitHub clearly mentioned software and hardware requirements?

In summary, the paper has the following key contributions:

- To the best of our knowledge, it is the first work to systematically study the nondeterministic impact of CPU multithreading on training DL systems.

TABLE I
NUMBER OF SEARCHED PROJECTS IN GITHUB

Keyword	#Stars	#Projects	Time Frame
deep learning	>200	985	up to 19/01/21
neural network	>200	625	up to 03/02/21

- We develop an experimental framework to understand and evaluate a wide range of CPU multithreaded environments based on VirtualBox. We characterize and quantify the impact on the training of DL systems. The experiment results show that our framework can effectively measure variances under different CPU multithreaded environments.
- We present six empirical findings and summarize five implications for DL researchers and practitioners.

II. DATA COLLECTION AND AGGREGATION

To collect our research data, we manually search projects with more than 200 stars in GitHub using two keywords, i.e., *deep learning* and *neural network*. After initial filtering, we gather a total of 1,610 projects, as shown in TABLE I. Then, we manually examine each project to select only the DL projects. Later, we examine the neural network architectures, implementation frameworks, and programming languages used in each DL project. The data collection and aggregation are described as follows.

Data Clean. We first check whether a project is a collection of source code and documents for educational purposes. Such a project is removed from the dataset. The projects that have been archived or deprecated by the developers are also removed. Moreover, we remove library projects, such as Python packages which can be installed using `pip install` command, since these projects are mainly developed as frameworks for further development. Furthermore, if a project only contains source code for loading pre-trained models but does not include training code, it will also be excluded from our dataset.

Examine DL Projects. To survey the popularity of neural networks, implementation frameworks, and programming languages used in DL development, we manually examine the README.md files, source code files, and external references (e.g., papers) in each repository. Note that we only record basic neural networks and the language used for model construction and training. For example, VGG16 [43] is multi-layered convolutional neural networks (CNNs). For the project that uses VGG16, we will treat its basic neural networks as CNNs. Besides, for recurrent neural networks (RNNs), we label all the variant RNN architectures as RNNs, such as long short-term memory (LSTM) and gated recurrent units (GRUs).

Examine Requirements. We manually examine the specified software and hardware requirements for each project. For software requirements, DL developers usually provide the "requirements.txt" file in their repositories or mention the required dependencies in the README.md file. If the prerequisite of software dependencies is found in a project, we consider the project has software requirements. For hardware requirements, we relax the conditions such that the hardware requirements of a project is considered as provided if the project mentions the tested hardware devices (e.g., a specific CPU or GPU model) or a minimum required size of DRAM or GPU RAM needed for execution.

TABLE II
DISTRIBUTION OF NEURAL NETWORKS, FRAMEWORKS AND LANGUAGES IN THE 245 COLLECTED PROJECTS

Network	#Projects	%Projects	Framework	#Projects	%Projects	Language	#Projects	%Projects
CNN	143	58.4	TensorFlow	92	37.5	Python	220	89.8
RNN	50	20.4	PyTorch	58	23.7	Lua	8	3.3
CNN/RNN	15	6.1	Keras (TensorFlow)	34	13.9	C++	7	2.9
GAN	13	5.3	Caffe	20	8.2	JavaScript	3	1.2
Others	24	9.8	Theano	11	4.5	Others	7	2.8
			Others	30	12.2			

Two authors independently conduct the manual data cleaning and examination. To ensure the consistency of results, we perform cross-checks to reach a consensus for conflicting cases. The examination and discussion process spent us two months. After these steps, we finally collected 245 DL projects. The statistic results of neural networks, DL frameworks, and languages used in these projects are shown in TABLE II. We can see that CNNs and RNNs are the dominant networks in DL development. These two neural networks account for 84.9% among all the examined projects. For DL frameworks, TensorFlow (including Keras using TensorFlow backend) and PyTorch are the most popular frameworks (75.1%). Moreover, Python (89.8%) is the most popular programming language used during DL development. The dataset of the collected DL projects is available at https://github.com/DeterministicDeepLearning.

III. OUR EXPERIMENTAL FRAMEWORK

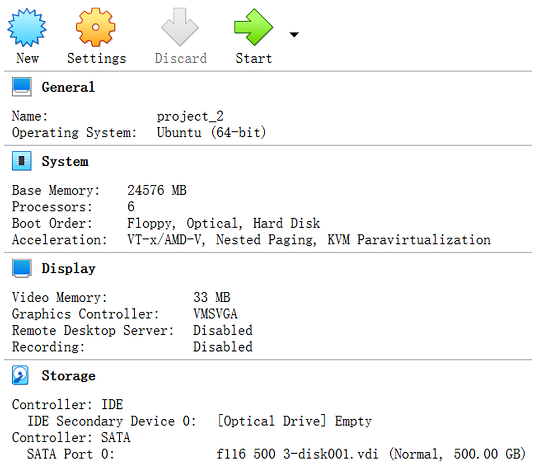
A. Overview

Fig. 2 shows our experimental framework, which consists of four parts, i.e., environment generation, DL app execution environments, identical model training, and variance analysis. The details of each part are described as follows.

Environment Generation. To emulate x86 hardware environments with different numbers of CPU threads, we use VirtualBox, an open-source type-2 hypervisor [44]. VirtualBox is executed within a host operating system (OS), which provides full virtualization for x86 virtualization. This means that guest OSs can be run in isolation within their own environment. Such virtualization can be seen as almost a complete simulation of the real hardware system, because it allows unmodified guest OSs to be run upon it [45]. Details of the infrastructures are as follows.

- Host Machine: The host machine is a workstation with an Intel Core i9-9900K CPU @ 3.60GHz (8 cores with 16 threads), 64 GB memory, 512 GB SSD and 2 TB HDD storage, and an Nvidia RTX 2080Ti GPU card.
- Host OS: The OS of the host machine is Windows 10 Pro (20H2).
- Hypervisor: We installed VirtualBox version 6.1.18 in the host OS to generate VMs.
- Guest OS: The guest OS installed in VMs is Ubuntu 18.04.5 LTS Desktop version [46], a popular Linux distribution.

We use VirtualBox to generate four types of VMs, all of which have the same memory size (i.e., 24 GB) and the same



```
General
Name:               project_2
Operating System:   Ubuntu (64-bit)

System
Base Memory:     24576 MB
Processors:      6
Boot Order:      Floppy, Optical, Hard Disk
Acceleration:    VT-x/AMD-V, Nested Paging, KVM Paravirtualization

Display
Video Memory:             33 MB
Graphics Controller:      VMSVGA
Remote Desktop Server:    Disabled
Recording:                Disabled

Storage
Controller: IDE
  IDE Secondary Device 0:    [Optical Drive] Empty
Controller: SATA
  SATA Port 0:               f116 500 3-disk001.vdi (Normal, 500.00 GB)
```

Fig. 3. A VM setting with 6 threads, 24 GB memory, and 500 GB storage.

storage size (i.e., 500 GB stored in the HDD), but different in the number of threads (i.e., 1, 2, 6, and 12). Fig. 3 shows an example setting of the VM with 6 threads. Note that the host CPU (i9-9900K) has 8 physical cores. If hyper-threading is enabled in BIOS, VirtualBox can manage a maximum number of 16 threads (logical CPUs).

DL App Execution Environments. To provide an isolated and independent execution environment, each DL project and its dependencies are installed and run within its own VM. To obtain a clean installation of a guest OS for each DL project, we create one VM and install Ubuntu 18.04.5. Then, we use the *importing and exporting Virtual Machines* feature to create a new VM with a clean installed Ubuntu 18.04.5 for the examined DL project. Therefore, one DL project will have at least one VM, and the number of CPU threads can be adjusted by VirtualBox.

Besides, to speed up the experiments, we execute two VMs at the same time. The resource allocation criterion of concurrent execution is that the total resources of concurrently executed VMs are less than those of the host machine. For example, two VMs (i.e., one has 2 CPU threads, and another has 6 CPU threads) can be run in parallel. When the VMs are executing, we do not perform any CPU-bound operations in the host machine.

Moreover, following the same setting in [18] and [47], we use fixed seed (i.e., same random generator and the same seed) to control software-level nondeterministic factors, as shown in TABLE III. Fixed-seed settings can disable the nondeterminism introduced in places such as initial weights, dropout layers, data augmentation, and batch ordering [18].

Identical Model Training. To study the impact of CPU multithreading (i.e., 1, 2, 6, and 12 threads) on the nondeterminism in training DL systems, we perform three categories of identical training runs as follows.

- Default identical training with model selection (*Default w/ MS*) that does not enforce software-level determinism (i.e., none of the random generators are controlled). The training runs are done under different multithreading environments with the model selection method used in the examined project.

TABLE III
FIXED-SEED SETTINGS FOR TENSORFLOW AND PYTORCH PROJECTS

TensorFlow (CPU)		PyTorch (CPU)	
Setting	SEED	Setting	SEED
os.environ['PYTHONHASHSEED']=str(SEED)	1	torch.manual_seed(SEED)	1
random.seed(SEED)	1	random.seed(SEED)	1
np.random.seed(SEED)	1	np.random.seed(SEED)	1
tf.set_random_seed(SEED)	1		

- Fixed-seed identical training with model selection (*Fixed w/ MS*) that is set to fixed seed to control software-level nondeterminism factors. The training runs are done under different multithreaded environments with the model selection method.
- Fixed-seed identical training without model selection (*Fixed w/o MS*) that has a fixed-seed setting under different multithreaded environments but without any model selection technique.

For each experiment set (i.e., one type of identical training runs of one DL project in one CPU multithreaded environment), we perform n runs ($n >= 30$), according to the time spent on training a model. Since we are performing training runs in a single-CPU platform, some DL projects are too slow to be trained in such an environment. Thus, it is hard to perform more identical runs for such projects.

The evaluation metrics mainly used are accuracy and training time. The accuracy is defined as the proportion of correct inferring cases by the model among the total number of cases input to the model. For the training time, we measure the time interval from the beginning of program execution to the end of model testing. The time used for data loading and processing should also be considered. Since we are performing training runs in different multithreaded environments, data operations may consume much time such as those in a single-threaded environment. Note that the evaluation metrics include but are not limited to the above metrics. According to specific DL projects, we can also use other metrics, e.g., recall, precision, and F1-score.

Variance Analysis. Multiple identical training runs of one DL project under one CPU multithreaded environment generate multiple models. In this part, we analyze the variance of the results produced by the trained models by using statistical methods, such as boxplots and statistics test, i.e., Mann–Whitney U test [49]. We study the variance by comparing the results obtained from different multithreaded environments to quantify the nondeterminism impact of CPU multithreading on training DL systems. For example, when comparing two sets of runs ($S1$ and $S2$), the null hypothesis is that the accuracies of sets $S1$ and $S2$ have similar distributions. Suppose we got a p-value which is less than a given significance level $\alpha = 0.05$. In that case, we can reject the null hypothesis with 95% confidence and infer that the alternative hypothesis is true, i.e., the accuracies of sets $S1$ and $S2$ are statistically different. In this study, we use R language [50] to perform the Mann–Whitney U test.

B. Projects for Experiments and Training Settings

To cover the most popular neural networks (i.e., CNNs and RNNs) and DL frameworks (i.e., TensorFlow, Keras,

TABLE IV
EXAMINED GITHUB DL PROJECTS

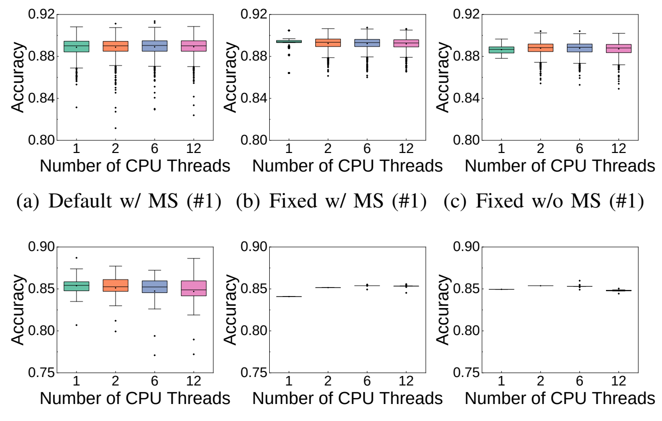
No.	Project	Framework	Framework Version	Neural Network	Application Domain	Commit	#Stars
#1	saurabhmathur96/clickbait-detector [48]	Keras (TensorFlow)	1.2.1 (0.12.1)	CNN	Clickbait Headlines Detection	1b6ab1b	467
#2	awni/ecg [41]	Keras (TensorFlow)	2.1.6 (1.8.0)	CNN	Medical (Heart) Diagnosis	c97bb96	403
#3	voicy-ai/DialogStateTracking [38]	TensorFlow	1.14.0	RNN (MemN2N)	Chat Bot	a102672	228
#4	zjy-ucas/ChineseNER [39]	TensorFlow	1.2.0	RNN	Chinese Named Entity Recognition	48e1007	1,470
#5	castorini/honk [40]	PyTorch	1.4.0	CNN	Keyword Spotting	c3aae75	389

and PyTorch) described in Section II, five DL projects in the collected dataset are selected for our experiments, as shown in TABLE IV. A project is selected if (1) it provides training and testing samples; (2) the project is developed using any DL framework with a CPU backend. Thus, we can perform the experiments with minimal changes in source code. Otherwise, we have to change the used library APIs, due to differences between CPU and GPU backends (e.g., Keras); and (3) the project has different framework versions and diverse application domains. In our evaluation, we only select five projects because it is very time-consuming to perform more experiments. The training settings of these projects, including the details of training samples, epochs, optimizations, model selection methods, evaluation metrics, and the number of identical training runs, are shown in TABLE V. In the following, we briefly introduce the examined DL projects.

- Project #1: *clickbait-detect* is a CNN-based model for detecting clickbait headlines [48]. Its implementation framework is Keras (using TensorFlow backend). For this project, we use the default training dataset and settings, such as settings of epochs and early stopping (ES, with a patience of 2), as shown in TABLE V. Since the time to train one model is about 30 seconds, we perform 1,000 identical training runs for each experiment set.
- Project #2: *ecg* is a medical (heart) diagnosis tool for cardiologist-level arrhythmia detection and classification in ambulatory electrocardiograms [41]. To accelerate the experiments, we create a smaller dataset from the original one and perform 30 identical training runs.
- Project #3: *DialogStateTracking* is a dialog system (chatbot) [38]. It implements two types of network architectures, i.e., hybrid code networks (HCN [51]) and end-to-end memory networks (MemN2N [52], an RNN architecture). We choose the MemN2N model for the experiment and perform task 3 (options in the model). All the training settings (TABLE V) are the same as the default settings in the project and 30 identical training runs are performed.
- Project #4: *ChineseNER* is a tool for Chinese named entity recognition based on RNNs [39]. Unlike other examined projects, the default evaluation metric is F1-score, since the recognition is a multi-label classification task. Similarly, we create a small dataset by reducing samples from the original dataset to accelerate the experiments and perform 30 identical training runs.
- Project #5: *honk* is a PyTorch reimplementation of CNNs for keyword spotting [40]. The model is used to recognize “command triggers” in speech-based interfaces [53], such as “Hey Siri”. In the experiments, we use three words *follow*, *learn*, and *visual*, as the keywords for training the model and we perform 30 runs.

TABLE V
TRAINING SETTINGS OF EXAMINED PROJECTS

No.	#Train	#Val	#Test	#Epochs	Optimizer	Model Selection	Metric	#Runs
#1	8,787	2,930	2,930	20	Adam	ES (2)	Accuracy	1,000
#2	205	51	64	30	Adam	ES (5) + Best Validation Loss	Accuracy	30
#3	9,855	9,978	10,018	200	Adam	Best Validation Accuracy	Accuracy	30
#4	10,439	1,188	2,386	30	Adam	Best Validation F1-score	F1-score	30
#5	4,617	477	596	30	SGD	Best Validation Accuracy	Accuracy	30



(a) Default w/ MS (#1) (b) Fixed w/ MS (#1) (c) Fixed w/o MS (#1)

(d) Default w/ MS (#2) (e) Fixed w/ MS (#2) (f) Fixed w/o MS (#2)

Fig. 4. Boxplots of accuracy variance of projects #1 and #2.

Note that all the training settings presented in TABLE V can produce models with the convergence of training loss. Since our goal is to evaluate the variance (i.e., nondeterminism) in identical training runs of DL systems impacted by CPU multithreading, our training settings may not generate the best training results. Besides, for reproducing and further investigation, we publicly release all the corresponding VMs of these projects, which can be accessed at https://github.com/DeterministicDeepLearning.

IV. RESULTS AND ANALYSIS

A. RQ1: Impact of CPU Multithreading on Effectiveness Variance (e.g., accuracy) in Training DL Systems

We first give the experiment results of projects #1 and #2, which are developed using Keras with the TensorFlow backend (TABLE IV). As shown in Fig. 4, for these two projects, we use boxplots to illustrate the accuracy distributions of the final testing results from the three categories of identical training runs (described in Section III-A). The results of Mann–Whitney U test between two accuracy variances from two different environments are presented in TABLE VI.

We can observe from Fig. 4 (a) and (d) that when performing default identical training runs with model selection (i.e., without controlling software-level nondeterminism factors), both projects have large accuracy variances across

TABLE VI
p-VALUES OF MANN–WHITNEY U TEST FOR FIG. 4

Project #1											
Default w/ MS				Fixed w/ MS				Fixed w/o MS			
#Threads	2	6	12	#Threads	2	6	12	#Threads	2	6	12
1	0.686	0.337	0.237	1	0.751	0.945	0.156	1	**<0.001**	**<0.001**	**<0.001**
2		0.573	0.452	2		0.858	0.133	2		0.912	**0.031**
6			0.851	6			0.178	6			**0.048**
Project #2											
Default w/ MS				Fixed w/ MS				Fixed w/o MS			
#Threads	2	6	12	#Threads	2	6	12	#Threads	2	6	12
1	0.717	0.455	0.174	1	**<0.001**	**<0.001**	**<0.001**	1	**<0.001**	**<0.001**	**<0.001**
2		0.595	0.344	2		**<0.001**	**<0.001**	2		**<0.001**	**<0.001**
6			0.674	6			**<0.001**	6			**<0.001**

Note: values shown in **bold** represent the p-value is less than a give significance level $\alpha = 0.05$.

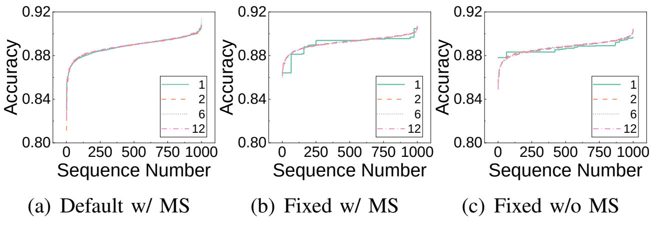


Fig. 5. Sorting of accuracy variance of project #1.

four environments, i.e., nondeterminism exists in multiple identical training runs. The results are intuitive, since different runs generate different randomness. This implies that when reporting the effectiveness of one DL system, multiple runs are necessary to be performed to obtain an average score. The maximum accuracy difference of project #1 is 9.97% in a 2-threaded environment, while that of project #2 is 11.43% under a 12-threaded environment.

Although nondeterminism exists in default identical runs across all the environments, the Mann–Whitney U test results show that such variances have no statistically significant difference, i.e., the p-value is larger than a given significance level $\alpha = 0.05$. Thus, we can not reject the null hypothesis that the accuracy variances of default runs have similar distributions in different environments. It seems that under software-level nondeterminism factors, the impact of CPU multithreading on the effectiveness variance in training DL systems can not be observed.

However, these two projects manifest differently for fixed-seed identical training runs, i.e., the 2nd and the 3rd categories of runs (with and without model selection). For project #1, Fig. 4 (b) and (c) demonstrate that accuracy variances still exist during multiple runs, although such variances are smaller than those from default runs. The Mann–Whitney U test results (TABLE VI) show that the accuracy variances generated from fixed-seed identical runs with model selection (early stopping) have no significant difference in different CPU multithreaded environments. Without model selection, the accuracy in a single-threaded environment is statistically different from that under multithreaded environments (i.e., 2, 6, and 12 threads). Note that the average accuracy scores (Fig. 4 (c)) obtained from different environments are very similar, i.e., 88.65%, 88.79%, 88.77%, and 88.72%, respectively. Although the accuracy scores are similar in different environments, it can be observed from Fig. 5 that for the fixed-seed runs, a few accuracy scores keep occurring in single-threaded environments.

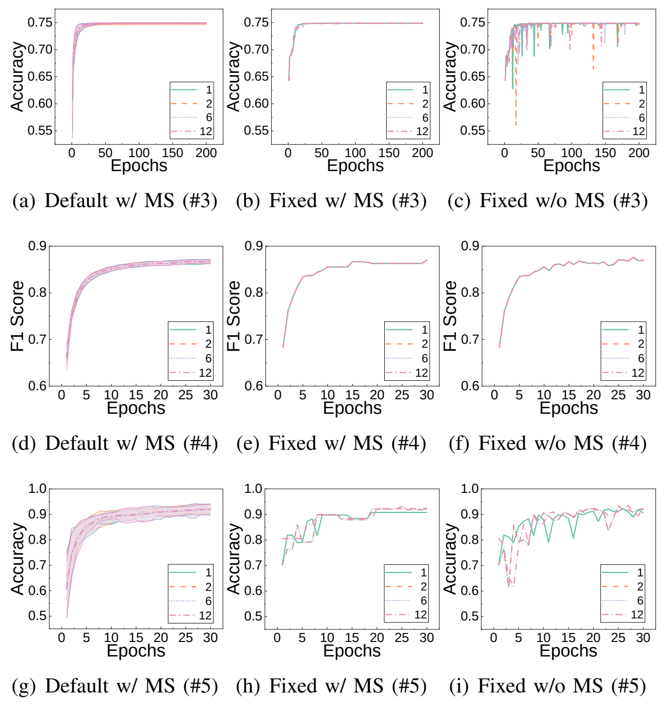


Fig. 6. Accuracy variance of projects #3-5 (F1-score for project #4). Shaded areas represent values within one standard deviation of the average score. The absence of a shaded area illustrates identical results across all identical runs.

For example, as shown in Fig. 5 (b) and (c), the accuracy scores 89.39% and 88.33%, have occurred 334 and 358 times, respectively in the 1,000 fixed-seed identical training runs with and without model selection.

For project #2, Fig. 4 (e) and (f) present that after controlling random generators with fixed seeds (TABLE III), the accuracy scores obtained from multiple runs in single-threaded and 2-threaded environments are deterministic. For example, for fixed-seed runs with model selection (Fig. 4 (e)), the accuracy scores in single-threaded and 2-threaded environments are 84.08% and 85.16%, respectively. However, accuracy variances occurred when increasing the number of CPU threads (e.g., 6 and 12). The training result can not be reproduced in a deterministic manner with multiple training runs. The statistic testing results in TABLE VI show that the accuracy scores from fixed-seed identical runs (both with and without model selection) are statistically different in different environments. This implies that CPU multithreading can cause accuracy variance in training DL systems.

Finding #1: For the examined two Keras (with TensorFlow backend) projects, the accuracy variances obtained from default identical training runs (without controlling random generators) have no significant difference in different CPU multithreaded environments. The nondeterminism factors cause accuracy differences as large as 11.43%. However, after controlling random generators, we can observe the impact of CPU multithreading on the accuracy variance (e.g., different training accuracy). Besides, although variance still exists in fixed-seed identical runs, some accuracy scores keep occurring in a single-threaded environment.

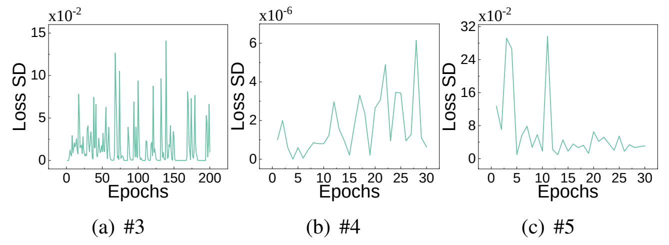


Fig. 7. Standard deviation (SD) calculated from training loss of fixed-seed runs of projects #3-5.

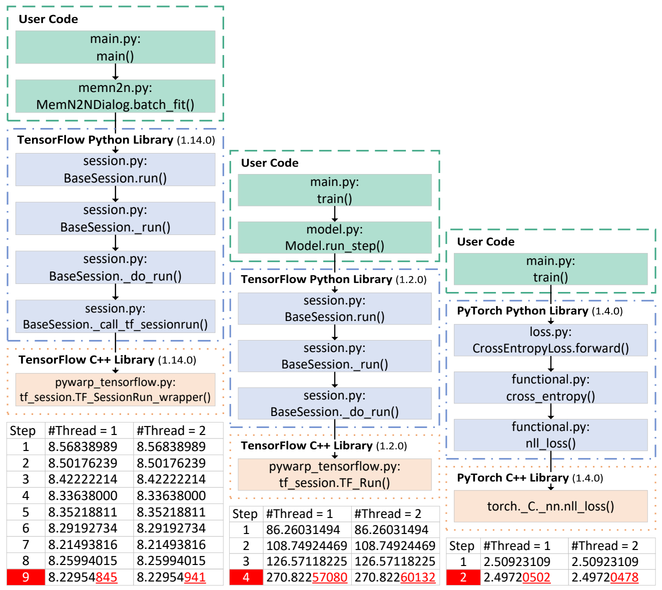
Next, we present the experimental results for projects #3-4 and #5, which are developed using TensorFlow and PyTorch, respectively. Fig. 6 presents accuracy variance obtained from three categories of identical training runs under different CPU multithreaded environments. Note that for TensorFlow and PyTorch projects, we present the evaluation results by the model trained after each epoch.

Similar to the default runs of the Keras projects, we can see from Fig. 6 (a), (d), and (g) that without setting fixed seeds, the training results are nondeterministic, i.e., multiple runs generate multiple models with different results. It seems that there is no difference in the accuracy variances obtained from different environments, because the shaded areas (i.e., values within one standard deviation (SD) of the average score) obtained from different environments are overlapped.

Moreover, it can be observed from Fig. 6 that when performing fixed-seed identical training runs, all three projects produce deterministic evaluation results within the same CPU multithreaded environment. It is interesting that for project #4, the evaluation results (F1-score) are identical (at a certain precision, i.e., six decimal places) in the four multithreaded environments. However, after examining the training loss of models generated from different environments, it is found that although the evaluation results of project #4 are identical, the loss values trained in single-threaded and multithreaded environments (i.e., 2, 6, and 12 threads) are different. The SD value of training loss of project #4 is not zero but is significantly smaller than that of projects #3 and #5 (i.e., differ by 4 orders of magnitude), as depicted in Fig. 7.

Finding #2: Similar to the Keras projects, for the examined TensorFlow and PyTorch projects, default identical training runs produce multiple models with different evaluation results. However, fixed-seed identical training runs produce deterministic evaluation results in the environment with the same number of CPU threads. The deterministic evaluation results of one project can be different across different environments.

To analyze which program functions introduce value differences, we manually trace the value of training loss in the source code. Fig. 8 (a), (b), and (c) show the tracing results in the first training epoch for projects #3-5, respectively. Note that the functions of TensorFlow's Python library between projects #3 and #4 are different, due to different TensorFlow versions. All the tracing results dive into the C++ library functions of TensorFlow and PyTorch. The training loss values generated from different environments are identical in a few beginning steps within the first training epoch but become different after more execution steps. It should be noted that the input samples for each step are identical in different environments. This implies that the rounding error (i.e., floating point imprecision) would be accumulated along with the training time.



User Code
main.py: main()
memn2n.py: MemN2NDialog.batch_fit()
TensorFlow Python Library (1.14.0)
session.py: BaseSession.run()
session.py: BaseSession._run()
session.py: BaseSession._do_run()
session.py: BaseSession._call_tf_sessionrun()
TensorFlow C++ Library (1.14.0)
pywarp_tensorflow.py: tf_session.TF_SessionRun_wrapper()
User Code
main.py: train()
model.py: Model.run_step()
TensorFlow Python Library (1.2.0)
session.py: BaseSession.run()
session.py: BaseSession._run()
session.py: BaseSession._do_run()
TensorFlow C++ Library (1.2.0)
pywarp_tensorflow.py: tf_session.TF_Run()
User Code
main.py: train()
PyTorch Python Library (1.4.0)
loss.py: CrossEntropyLoss.forward()
functional.py: cross_entropy()
functional.py: nll_loss()
PyTorch C++ Library (1.4.0)
torch._C._nn.nll_loss()

Step	#Thread = 1	#Thread = 2
1	8.56838989	8.56838989
2	8.50176239	8.50176239
3	8.42222214	8.42222214
4	8.33638000	8.33638000
5	8.35218811	8.35218811
6	8.29192734	8.29192734
7	8.21493816	8.21493816
8	8.25994015	8.25994015
9	8.22954845	8.22954941

(a) #3

Step	#Thread = 1	#Thread = 2
1	86.26031494	86.26031494
2	108.74924469	108.74924469
3	126.57118225	126.57118225
4	270.82257080	270.82260132

(b) #4

Step	#Thread = 1	#Thread = 2
1	2.50923109	2.50923109
2	2.49720502	2.49720478

(c) #5

Fig. 8. Loss value differences in user and framework code for projects #3-5.

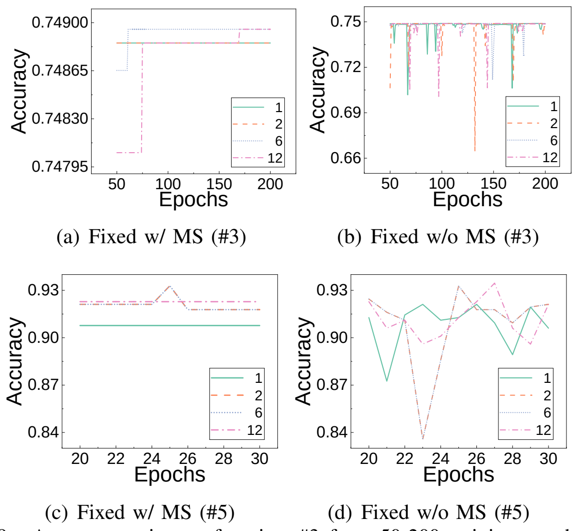


Fig. 9. Accuracy variance of project #3 from 50-200 training epochs and project #5 from 20-30 training epochs.

Finding #3: The rounding errors of floating-point numbers come from low-level implementations of DL frameworks, i.e., the C++ library. Under different CPU multithreaded environments, the rounding errors are accumulated during the training process. After some training steps, the values of training loss become different.

In the following, we focus on the projects (i.e., #3 and #5) that have different evaluation results in different CPU multithreaded environments, as shown in Fig. 9. For project

TABLE VII
MAXIMUM ACCURACY DIFFERENCE OF PROJECTS #3 AND #5

#Threads	Fixed w/ MS		Fixed w/o MS	
	#3 (Epochs 61–74)	#5 (Epoch 25)	#3 (Epoch 132)	#5 (Epoch 23)
1	0.748852066	0.907718122	0.748852066	0.921140969
2	0.748852066	0.932885885	0.664703534	0.835570455
6	0.748951887	0.932885885	0.748951887	0.835570455
12	0.748053504	0.922818780	0.748652426	0.895973146
Max Diff	0.000898383	0.025167763	0.084248353	0.085570514

#3, the accuracy scores from 50 to 200 epochs are presented, while the epoch range for project #5 is from 20 to 30. We can observe that fixed-seed identical training runs with model selection (i.e., best validation accuracy) can produce a more stable accuracy than the fixed-seed runs without model selection. The result is expected since the accuracy will only be updated after an epoch with a better validation accuracy.

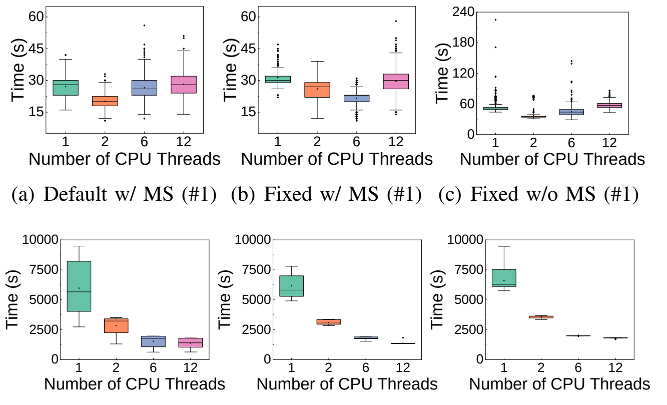
To further examine the maximum accuracy difference in the training process, we calculate the SD of the accuracy scores obtained from different environments. TABLE VII shows the maximum accuracy difference and the corresponding epoch(s) of projects #3 and #5. For fixed-seed runs with model selection, the max accuracy difference of project #3 obtained from different CPU multithreaded environments is very small (i.e., 0.0898%). In contrast, the max accuracy difference of project #5 is 2.52%. However, both projects have large accuracy differences without model selection, i.e., 8.42% and 8.56%, respectively. The result implies that the impact of CPU multithreading is substantial and not negligible.

Finding #4: For fixed-seed identical training runs without model selection, CPU multithreading causes accuracy differences as large as 8.56%. Using model selection can minimize the impact of CPU multithreading on training DL systems. The max accuracy differences are range from 0.0898% to 2.52% with model selection.

B. RQ2: Impact of CPU Multithreading on Time Variance in Training DL Systems

In this RQ, we present the results of time variance in training DL systems under different CPU multithreaded environments. Note that as described in Section III-A, the training time in our experiments is measured as the time interval from the beginning of program execution to the end of model testing.

Fig. 10 shows the time variance of projects #1 and #2 obtained from three categories of identical training runs. The Mann–Whitney U test results between two time-variances from two different environments are presented in TABLE VIII. For project #1, the testing results show that the time variances are significantly different in different environments for all three categories of identical training runs. We can observe from Fig. 10 (a) and (c) that for default runs and fixed-seed runs without model selection, using 2 CPU threads can achieve a faster training time, while for fixed-seed runs with model selection, using 6 threads would be better. However, for project #2, the training time of all the three categories of identical runs decreases with the increasing number of threads up to 12, i.e., the more CPU threads, the faster training efficiency.



(a) Default w/ MS (#1) (b) Fixed w/ MS (#1) (c) Fixed w/o MS (#1)
(d) Default w/ MS (#2) (e) Fixed w/ MS (#2) (f) Fixed w/o MS (#2)

Fig. 10. Boxplots of training time variance of projects #1 and #2.

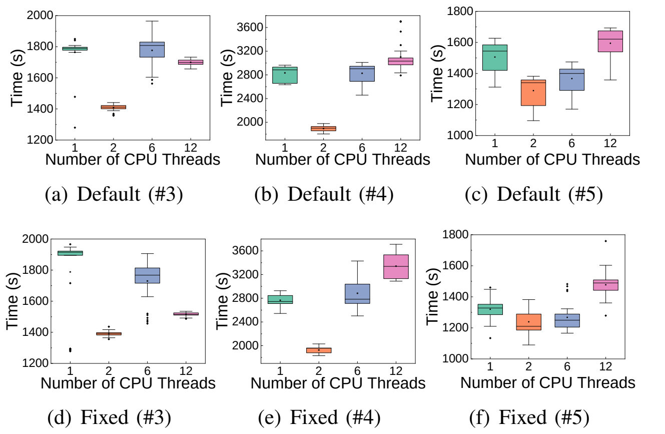
TABLE VIII
p-VALUES OF MANN–WHITNEY U TEST FOR FIG. 10

Project #1											
	Default w/ MS				Fixed w/ MS				Fixed w/o MS		
#Threads	2	6	12	#Threads	2	6	12	#Threads	2	6	12
1	**<0.001**	**0.004**	**<0.001**	1	**<0.001**	**<0.001**	**<0.001**	1	**<0.001**	**<0.001**	**<0.001**
2		**<0.001**	**<0.001**	2		**<0.001**	**<0.001**	2		**<0.001**	**<0.001**
6			**<0.001**	6			**<0.001**	6			**<0.001**
Project #2											
	Default w/ MS				Fixed w/ MS				Fixed w/o MS		
#Threads	2	6	12	#Threads	2	6	12	#Threads	2	6	12
1	**<0.001**	**<0.001**	**<0.001**	1	**<0.001**	**<0.001**	**<0.001**	1	**<0.001**	**<0.001**	**<0.001**
2		**<0.001**	**<0.001**	2		**<0.001**	**<0.001**	2		**<0.001**	**<0.001**
6			**0.044**	6			**<0.001**	6			**<0.001**

Note: values shown in **bold** represent the p-value is less than a give significance level $\alpha = 0.05$.

Fig. 11 shows the training-time variances of projects #3-5. The results of Mann–Whitney U test are presented in TABLE IX. Note that the time spent for fixed-seed identical training runs with and without model selection is the same. Since there is no early stopping used in these projects, all the epochs would be executed. We can observe from Fig. 11 that for the three projects, different environments generate different training times. Besides, using 2 CPU threads can achieve the best training efficiency. For projects #4 and #5, using 12 threads for model training produce the worst training efficiency.

More threads used for training may lead to the thread over-subscription issue, which can significantly affect the training time [35]. For example, threading settings can increase the training time of TensorFlow workloads running on CPUs in a substantial way [54]. In TensorFlow 1.x, training time of a model on a CPU backend relies on the threading model parameters in class ConfigProto, i.e., (`inter_op_parallelism_threads`, `intra_op_parallelism_threads`) [35], [54]. The first parameter (inter-op) denotes the number of threads used for parallelism between independent operations, while the second parameter (intra-op) specifies the number of threads used within an individual op for parallelism. If these parameters are unset by users, they will be initialized as the number of logical CPU cores by default. Therefore, in our experiments, we have four default threading parameters, i.e., (1,1), (2,2), (6,6), and (12,12). However, the recommended setting for the inter-op parameter is 2 [55], or a small value, e.g., ranging from 1 to 4 [35], depending on specified DL projects.



(a) Default (#3) (b) Default (#4) (c) Default (#5)

(d) Fixed (#3) (e) Fixed (#4) (f) Fixed (#5)

Fig. 11. Boxplots of training time variance of projects #3-5.

TABLE IX
p-VALUES OF MANN–WHITNEY U TEST FOR FIG. 11

Default											
Project #3				Project #4				Project #5			
#Threads	2	6	12	#Threads	2	6	12	#Threads	2	6	12
1	**<0.001**	0.267	**<0.001**	1	**<0.001**	0.652	**<0.001**	1	**<0.001**	**<0.001**	**<0.001**
2		**<0.001**	**<0.001**	2		**<0.001**	**<0.001**	2		**<0.001**	**<0.001**
6			**<0.001**	6			**<0.001**	6			**<0.001**
Fixed											
Project #3				Project #4				Project #5			
#Threads	2	6	12	#Threads	2	6	12	#Threads	2	6	12
1	**<0.001**	**<0.001**	**<0.001**	1	**<0.001**	0.223	**<0.001**	1	**<0.001**	**<0.001**	**<0.001**
2		**<0.001**	**<0.001**	2		**<0.001**	**<0.001**	2		0.225	**<0.001**
6			**<0.001**	6			**<0.001**	6			**<0.001**

Note: values shown in **bold** represent the p-value is less than a give significance level $\alpha = 0.05$.

Finding #5: For the examined five DL projects, only one project shows that the training time would decrease along with the increasing number of CPU threads. For most projects, using 2 threads would be a better choice to obtain a faster training efficiency.

C. RQ3: Proportions of DL Projects that Mentioned Software and Hardware Requirements

After examining software and hardware requirements of the 245 collected GitHub DL projects (described in Section II), we can see from TABLE X that 222 projects (90.6%) have listed software requirements. In comparison, only 54 projects (22.0%) have mentioned hardware requirements. For software requirements, the results are expected since DL developers mainly rely on DL libraries/frameworks to implement DL models [56].

We further conduct statistics of hardware types mentioned in the 54 projects. Note that some projects mentioned both tested GPU and CPU environments. Thus there exists an overlapping between the statistics results of GPU and CPU environments. It is found that the most frequent type of tested hardware environments is GPU, i.e., 51 projects (94.4%) have mentioned GPU devices and GPU-related memory required to be used for model training. Comparatively, 8 projects (14.8%) mentioned CPU-related devices. It seems that DL practitioners are unaware of reporting their tested hardware environments for model training and evaluation. However, according to our findings, it is not a good practice since CPU multithreading can cause effectiveness variance in training DL systems.

TABLE X
PROPORTIONS OF SOFTWARE AND HARDWARE REQUIREMENTS OF THE 245 COLLECTED GITHUB DL PROJECTS

Provided	Software Requirements		Hardware Requirements	
	#Projects	%Projects	#Projects	%Projects
Yes	222	90.6	54	22.0
No	23	9.4	191	78.0

Finding #6: Among the 245 collected GitHub DL projects, 90.6% of them provide software-level requirements, while only 22.0% mention hardware-level requirements. For the projects that have hardware requirements (54 projects), 94.4% present tested GPU environments, while 14.8% list tested CPU environments.

V. IMPLICATIONS FOR DL RESEARCHERS AND PRACTIONERS

Based on our empirical findings, we provide the following five implications for DL researchers and practitioners.

- **Implication #1**: When training DL systems on CPU platforms, to obtain deterministic training results, developers should control software-level nondeterminism factors (e.g., random generators). Besides, practitioners should pay attention to the effectiveness variance under different CPU multithreaded environments.
- **Implication #2**: The effectiveness variance introduced by CPU multithreading could be huge, in particular when training DL systems without using model selection techniques. To minimize the impact of CPU multithreading on the effectiveness variance in training DL systems, developers are suggested to use model selection methods (e.g., best validation loss/accuracy). Moreover, to achieve deterministic DL systems, it is necessary to develop mitigation techniques to eliminate the inconsistency of accuracy scores introduced by CPU multithreading.
- **Implication #3**: Using more CPU threads for model training does not necessarily indicate that we can achieve faster training efficiency. DL models with a default setting (e.g., using all CPU threads) do not often take full advantage of computing capability of the underlying hardware [55]. For obtaining a promising training time, developers are suggested to use some auto-tuning tools (e.g., TensorTuner [35]) to fine-tune the built-in threading configuration in DL frameworks (e.g., TensorFlow).
- **Implication #4**: When optimizing CPU parallelism for training DL systems, developers should pay attention to the potential impact on the effectiveness variance introduced by CPU multithreading. For example, it would be necessary to consider the task as a multi-objective optimization problem, i.e., minimize the training time and maximize the effectiveness (e.g., accuracy).
- **Implication #5**: Developers are suggested to mention hardware environments used for model training and evaluation when releasing a DL project. In particular, if DL models are trained on CPU platforms, developers should provide detailed CPU models, since the number of threads used can affect model training and evaluation.

VI. Threats to Validity

Internal Validity. The empirical data collection and examination can be considered as a threat to internal validity. To reduce this threat, we conduct data examination independently and carefully by two authors. The results are cross-checked to reach a consensus for the final reporting ones.

The use of virtual machines (i.e., VirtualBox) to emulate CPU multithreaded environments can be another threat to internal validity, since VMs may not fully represent the real multithreaded environments provided by CPUs. Using VMs to emulate different software and hardware environments for software testing is quite common in practice [57]–[59]. However, we acknowledge that the potential impact of VMs on the effectiveness of our findings can not be ignored. For example, creating VMs in different CPU models or hosting OSs may produce different environments, thus leading to different results. Besides, running VMs in parallel for the experiments could affect the training time.

Moreover, there exist other nondeterminism factors introduced by CPUs, such as instructions in different CPU architectures [60]. Such nondeterminism factors are interesting to be explored in the future, e.g., comparing identical training runs of DL systems on different CPU architectures.

External Validity. The generalization of our findings is the main threat to external validity. First, to collect DL projects, we used two keywords (i.e., *deep learning* and *neural network*) to search GitHub projects. Besides, for the experiments, we examined five DL projects with limited versions of DL frameworks. Therefore, the finding results are valid for our limited experiments. Our experimental framework is applicable for any DL project that can be trained on CPU platforms. We plan to examine more projects using the experimental framework to consolidate our findings in the future further.

Construct Validity. The threat to construct validity comes from the selected metrics (i.e., accuracy and training time) for the experiments. These two metrics are the commonly-used metrics for evaluating effectiveness and training efficiency. However, using other evaluation metrics may have different finding results. In our future work, we plan to use more evaluation metrics for the experiments.

VII. Related Work

Variance Analysis in Deep Learning. Nagarajan *et al.* [17] analyze some nondeterminism factors (e.g., GPU, environment, initialization, and minibatch) to the reproducibility in reinforcement learning. Pham *et al.* [18] study the variance in training general DL systems introduced by different nondeterminism factors (e.g., software-level and GPU nondeterminism). They conduct a survey and a literature review to measure the awareness of such variance in identical training runs. Different from their studies, our work focuses on investigating the nondeterministic impact of CPU multithreading on training DL systems. Bahrampour *et al.* [61] and Kochura *et al.* [62] study the training efficiency impacted by different CPU multithreaded environments. Guo *et al.* [63] study the accuracy variance of DL systems impacted by different DL frameworks and deployment platforms (e.g., web browsers and mobile devices). Our study investigates the impact of CPU multithreading on the variance (effectiveness and training time) in multiple identical training runs of DL systems, i.e., to quantify the deterministic or nondeterministic behavior introduced by CPU multithreading.

Optimizing Deep Learning on CPU Platforms. Hasabnis *et al.* [35] propose an auto-tuning tool, i.e., TensorTuner, to optimize the training efficiency of TensorFlow on CPU platforms by setting the optimal built-in threading model parameters. Kalamkar *et al.* [32] present optimization techniques for improving DL recommender systems' training efficiency on CPU cluster architectures. Awan *et al.* [64] analyze the characterization of CPU- and GPU-based DL training on modern architectures. They find that CPU-based optimizations (e.g., OpenMP [65]) can significantly improve DL training efficiency for CPU platforms. Wang *et al.* [33] analyze the key design features (e.g., scheduling and operator implementation) in DL frameworks and further propose guides for tuning DL framework parameters to maximize parallelism performance on CPU platforms. Our work focuses on studying the impact of CPU multithreading on the effectiveness variance in training DL systems. Our findings show that different CPU multithreaded environments can also lead to different effectiveness variances. Therefore, when optimizing training efficiency on CPU platforms, the effectiveness variance should also be considered. We refer this survey [36] to the readers who are interested in the topic of optimizing DL on CPU platforms.

VIII. Conclusion

In this work, we conducted an empirical study to analyze the nondeterministic impact of CPU multithreading on training DL systems. We proposed an experimental framework based on VirtualBox to generate a wide range of CPU multithreaded environments (i.e., 1, 2, 6, and 12 threads). We performed experiments on five GitHub DL projects, covering two neural networks (i.e., CNNs and RNNs) and three DL implementation frameworks (i.e., TensorFlow, PyTorch, and Keras). Finally, we investigated the proportions of projects that specified software and hardware requirements in the collected 245 GitHub DL projects. Based on the experiments, we presented six empirical findings and summarized five implications for DL researchers and practitioners. We believe this study can facilitate understanding the nondeterminism in training DL systems under CPU multithreaded environments. In the future, we plan to conduct experiments on more DL projects.

Acknowledgment

We thank the anonymous reviewers for their valuable comments and suggestions. This work was supported in part by the National Natural Science Foundation of China under Grants 62002163 and 61772055, Natural Science Foundation of Jiangsu Province under Grant BK20200441, Open Research Fund of State Key Laboratory of Novel Software Technology under Grant KFKT2020B20, and Australian Research Council under Grant DP200101328.

References

[1] R. Gao and K. Grauman, "On-demand learning for deep image restoration," in *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2017, pp. 1086–1095.

[2] Y. Goldberg, "A primer on neural network models for natural language processing," *Journal of Artificial Intelligence Research*, vol. 57, pp. 345–420, 2016.

[3] R. Haeb-Umbach, S. Watanabe, T. Nakatani, M. Bacchiani, B. Hoffmeister, M. L. Seltzer, H. Zen, and M. Souden, "Speech processing for digital home assistants: Combining signal processing with deep-learning techniques," *IEEE Signal Processing Magazine*, vol. 36, no. 6, pp. 111–124, 2019.

[4] X. Du, Z. Zheng, G. Xiao, and B. Yin, "The automatic classification of fault trigger based bug report," in *Proceedings of the IEEE 28th International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, 2017, pp. 259–265.

[5] X. Wan, Z. Zheng, F. Qin, Y. Qiao, and K. S. Trivedi, "Supervised representation learning approach for cross-project aging-related bug prediction," in *Proceedings of the IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2019, pp. 163–172.

[6] G. Xiao, X. Du, Y. Sui, and T. Yue, "Hindbr: Heterogeneous information network based duplicate bug report prediction," in *Proceedings of the IEEE 31st International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2020, pp. 195–206.

[7] Y. Sui, X. Cheng, G. Zhang, and H. Wang, "Flow2vec: value-flow-based precise code embedding," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–27, 2020.

[8] W. Hua, Y. Sui, Y. Wan, G. Liu, and G. Xu, "Fcca: Hybrid code representation for functional clone detection using attention networks," *IEEE Transactions on Reliability*, vol. 70, no. 1, pp. 304–318, 2020.

[9] W. Wang, Y. Zhang, Y. Sui, Y. Wan, Z. Zhao, J. Wu, P. Yu, and G. Xu, "Reinforcement-learning-guided source code summarization via hierarchical attention," *IEEE Transactions on Software Engineering*, 2020.

[10] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "Deepwukong: Statically detecting software vulnerabilities using deep graph neural network," *ACM Transactions on Software Engineering and Methodology*, vol. 30, no. 3, pp. 1–33, 2021.

[11] B. Huval, T. Wang, S. Tandon, J. Kiske, W. Song, J. Pazhayampallil, M. Andriluka, P. Rajpurkar, T. Migimatsu, R. Cheng-Yue *et al.*, "An empirical evaluation of deep learning on highway driving," *arXiv preprint arXiv:1504.01716*, 2015.

[12] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang *et al.*, "End to end learning for self-driving cars," *arXiv preprint arXiv:1604.07316*, 2016.

[13] A. I. Maqueda, A. Loquercio, G. Gallego, N. García, and D. Scaramuzza, "Event-based vision meets deep learning on steering prediction for self-driving cars," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2018, pp. 5419–5427.

[14] R. Fakoor, F. Ladhak, A. Nazi, and M. Huber, "Using deep learning to enhance cancer diagnosis and classification," in *Proceedings of the International Conference on Machine Learning (ICML)*, vol. 28. ACM, 2013.

[15] D. Shen, G. Wu, and H.-I. Suk, "Deep learning in medical image analysis," *Annual Review of Biomedical Engineering*, vol. 19, pp. 221–248, 2017.

[16] J. De Fauw, J. R. Ledsam, B. Romera-Paredes, S. Nikolov, N. Tomasev, S. Blackwell, H. Askham, X. Glorot, B. O'Donoghue, D. Visentin *et al.*, "Clinically applicable deep learning for diagnosis and referral in retinal disease," *Nature Medicine*, vol. 24, no. 9, pp. 1342–1350, 2018.

[17] P. Nagarajan, G. Warnell, and P. Stone, "Deterministic implementations for reproducibility in deep reinforcement learning," *arXiv preprint arXiv:1809.05676*, 2018.

[18] H. V. Pham, S. Qian, J. Wang, T. Lutellier, J. Rosenthal, L. Tan, Y. Yu, and N. Nagappan, "Problems and opportunities in training deep learning software systems: An analysis of variance," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 771–783.

[19] T. Salimans and D. P. Kingma, "Weight normalization: A simple reparameterization to accelerate training of deep neural networks," *arXiv preprint arXiv:1602.07868*, 2016.

[20] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop," in *Neural Networks: Tricks of the Trade*. Springer, 2012, pp. 9–48.

[21] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2016, pp. 265–283.

[22] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga *et al.*, "Pytorch: An imperative style, high-performance deep learning library," in *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, vol. 32, 2019.

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

[24] Y. Kim and J. Ra, "Weight value initialization for improving training speed in the backpropagation network," in *Proceedings of the IEEE International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1991, pp. 2396–2401.

[25] K. K. Teo, L. Wang, and Z. Lin, "Wavelet packet multi-layer perceptron for chaotic time series prediction: effects of weight initialization," in *Proceedings of the International Conference on Computational Science (ICCS)*. Springer, 2001, pp. 310–317.

[26] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," in *Proceedings of the International Conference on Machine Learning (ICML)*. PMLR, 2013, pp. 1139–1147.

[27] D. Budden, A. Matveev, S. Santurkar, S. R. Chaudhuri, and N. Shavit, "Deep tensor convolution on multicores," in *Proceedings of the International Conference on Machine Learning (ICML)*. PMLR, 2017, pp. 615–624.

[28] Y. E. Wang, G.-Y. Wei, and D. Brooks, "Benchmarking tpu, gpu, and cpu platforms for deep learning," *arXiv preprint arXiv:1907.10701*, 2019.

[29] C.-J. Wu, D. Brooks, K. Chen, D. Chen, S. Choudhury, M. Dukhan, K. Hazelwood, E. Isaac, Y. Jia, B. Jia *et al.*, "Machine learning at facebook: Understanding inference at the edge," in *Proceedings of the IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2019, pp. 331–344.

[30] P. Blacker, C. Bridges, and S. Hadfield, "Rapid prototyping of deep learning models on radiation hardened cpus," in *Proceedings of the NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. IEEE, 2019, pp. 25–32.

[31] Y. Zou, X. Jin, Y. Li, Z. Guo, E. Wang, and B. Xiao, "Mariana: Tencent deep learning platform and its applications," *Proceedings of the VLDB Endowment*, vol. 7, no. 13, pp. 1772–1777, 2014.

[32] D. Kalamkar, E. Georganas, S. Srinivasan, J. Chen, M. Shiryaev, and A. Heinecke, "Optimizing deep learning recommender systems' training on cpu cluster architectures," *arXiv preprint arXiv:2005.04680*, 2020.

[33] Y. E. Wang, C.-J. Wu, X. Wang, K. Hazelwood, and D. Brooks, "Exploiting parallelism opportunities with deep learning frameworks," *ACM Transactions on Architecture and Code Optimization*, vol. 18, no. 1, pp. 1–23, 2020.

[34] "tensorflow/direct_session.cc at master · tensorflow/tensorflow," 2021. [Online]. Available: https://github.com/tensorflow/tensorflow/blob/master/tensorflow/core/common_runtime/direct_session.cc

[35] N. Hasabnis, "Auto-tuning tensorflow threading model for cpu backend," in *Proceedings of the IEEE/ACM Machine Learning in HPC Environments (MLHPC)*. IEEE, 2018, pp. 14–25.

[36] S. Mittal, P. Rajput, and S. Subramoney, "A survey of deep learning on cpus: opportunities and co-optimizations," *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[37] "Understanding tensorflow inter/intra parallelism threads - Stack Overflow," 2021. [Online]. Available: https://stackoverflow.com/questions/47548145/understanding-tensorflow-inter-intra-parallelism-threads

[38] "Github - voicy-ai/Dialogstatetracking: Dialog State Tracking using End-to-End Neural Networks," 2021. [Online]. Available: https://github.com/voicy-ai/DialogStateTracking

[39] "Github - zjy-ucas/ChineseNER: A neural network model for Chinese named entity recognition," 2021. [Online]. Available: https://github.com/zjy-ucas/ChineseNER

[40] "GitHub - castorini/honk: PyTorch implementations of neural network models for keyword spotting," 2021. [Online]. Available: https://github.com/castorini/honk

[41] "Github - awni/ecg: Cardiologist-level arrhythmia detection and classification in ambulatory electrocardiograms using a deep neural network," 2021. [Online]. Available: https://github.com/awni/ecg

[42] "VirtualBox," 2021. [Online]. Available: https://www.virtualbox.org/

[43] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv preprint arXiv:1409.1556*, 2014.

[44] E. Bauman, G. Ayoade, and Z. Lin, "A survey on hypervisor-based monitoring: approaches, applications, and evolutions," *ACM Computing Surveys*, vol. 48, no. 1, pp. 1–33, 2015.

[45] D. Beserra, F. Oliveira, J. Araujo, F. Fernandes, A. Araújo, P. Endo, P. Maciel, and E. D. Moreno, "Performance evaluation of hypervisors for hpc applications," in *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE, 2015, pp. 846–851.

[46] "Enterprise Open Source and Linux — Ubuntu," 2021. [Online]. Available: https://ubuntu.com/

[47] "Reproducibility — PyTorch 1.8.1 documentation," 2021. [Online]. Available: https://pytorch.org/docs/stable/notes/randomness.html

[48] "Github - saurabhmathur96/clickbait-detector: Detects clickbait headlines using deep learning," 2021. [Online]. Available: https://github.com/saurabhmathur96/clickbait-detector

[49] N. Nachar *et al.*, "The mann-whitney u: A test for assessing whether two independent samples come from the same distribution," *Tutorials in Quantitative Methods for Psychology*, vol. 4, no. 1, pp. 13–20, 2008.

[50] R Core Team, *R: A Language and Environment for Statistical Computing*, R Foundation for Statistical Computing, Vienna, Austria, 2013. [Online]. Available: http://www.R-project.org/

[51] J. D. Williams, K. Asadi, and G. Zweig, "Hybrid code networks: practical and efficient end-to-end dialog control with supervised and reinforcement learning," *arXiv preprint arXiv:1702.03274*, 2017.

[52] S. Sukhbaatar, A. Szlam, J. Weston, and R. Fergus, "End-to-end memory networks," *arXiv preprint arXiv:1503.08895*, 2015.

[53] R. Tang and J. Lin, "Honk: A pytorch reimplementation of convolutional neural networks for keyword spotting," *arXiv preprint arXiv:1710.06554*, 2017.

[54] "Guide to TensorFlow* Runtime optimizations for CPU," 2021. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/guide-to-tensorflow-runtime-optimizations-for-cpu.html

[55] "Tips to Improve Performance for Popular Deep Learning Frameworks on CPUs," 2021. [Online]. Available: https://software.intel.com/content/www/us/en/develop/articles/tips-to-improve-performance-for-popular-deep-learning-frameworks-on-multi-core-cpus.html

[56] J. Han, S. Deng, D. Lo, C. Zhi, J. Yin, and X. Xia, "An empirical study of the dependency networks of deep learning libraries," in *Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 868–878.

[57] I. Chana and A. Rana, "Empirical evaluation of cloud-based testing techniques: a systematic review," *ACM SIGSOFT Software Engineering Notes*, vol. 37, no. 3, pp. 1–9, 2012.

[58] S. Winter, O. Schwahn, R. Natella, N. Suri, and D. Cotroneo, "No pain, no gain? the utility of parallel fault injections," in *Proceedings of the IEEE/ACM 37th IEEE International Conference on Software Engineering (ICSE)*, vol. 1. IEEE, 2015, pp. 494–505.

[59] D. Cotroneo, L. De Simone, and R. Natella, "Run-time detection of protocol bugs in storage i/o device drivers," *IEEE Transactions on Reliability*, vol. 67, no. 3, pp. 847–869, 2018.

[60] V. M. Weaver, D. Terpstra, and S. Moore, "Non-determinism and overcount on modern hardware performance counter implementations," in *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2013, pp. 215–224.

[61] S. Bahrampour, N. Ramakrishnan, L. Schott, and M. Shah, "Comparative study of deep learning software frameworks," *arXiv preprint arXiv:1511.06435*, 2015.

[62] Y. Kochura, S. Stirenko, O. Alienin, M. Novotarskiy, and Y. Gordienko, "Performance analysis of open source machine learning frameworks for various parameters in single-threaded and multi-threaded modes," in *Proceedings of the Conference on Computer Science and Information Technologies (CSIT)*, 2017, pp. 243–256.

[63] Q. Guo, S. Chen, X. Xie, L. Ma, Q. Hu, H. Liu, Y. Liu, J. Zhao, and X. Li, "An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 810–822.

[64] A. A. Awan, H. Subramoni, and D. K. Panda, "An in-depth performance characterization of cpu-and gpu-based dnn training on modern architectures," in *Proceedings of the Machine Learning on HPC Environments (MLHPC)*, 2017, pp. 1–8.

[65] "Home - OpenMP," 2021. [Online]. Available: https://www.openmp.org