Problems and Opportunities in Training Deep Learning Software Systems: An Analysis of Variance

Hung Viet Pham
University of Waterloo
Waterloo, ON, Canada
hvpham@uwaterloo.ca

Shangshu Qian
Purdue University
West Lafayette, IN, USA
shangshu@purdue.edu

Jiannan Wang
Purdue University
West Lafayette, IN, USA
wang4524@purdue.edu

Thibaud Lutellier
University of Waterloo
Waterloo, ON, Canada
tlutelli@uwaterloo.ca

Jonathan Rosenthal
Purdue University
West Lafayette, IN, USA
rosenth0@purdue.edu

Lin Tan
Purdue University
West Lafayette, IN, USA
lintan@purdue.edu

Yaoliang Yu
University of Waterloo
Waterloo, ON, Canada
yaoliang.yu@uwaterloo.ca

Nachiappan Nagappan
Microsoft Research
Redmond, WA, USA
nachin@microsoft.com

ABSTRACT

Deep learning (DL) training algorithms utilize nondeterminism to improve models' accuracy and training efficiency. Hence, multiple identical training runs (e.g., identical training data, algorithm, and network) produce different models with different accuracies and training times. In addition to these algorithmic factors, DL libraries (e.g., TensorFlow and cuDNN) introduce additional variance (referred to as implementation-level variance) due to parallelism, optimization, and floating-point computation.

This work is the first to study the variance of DL systems and the awareness of this variance among researchers and practitioners. Our experiments on three datasets with six popular networks show large overall accuracy differences among identical training runs. Even after excluding weak models, the accuracy difference is 10.8%. In addition, implementation-level factors alone cause the accuracy difference across identical training runs to be up to 2.9%, the per-class accuracy difference to be up to 52.4%, and the training time difference to be up to 145.3%. All core libraries (TensorFlow, CNTK, and Theano) and low-level libraries (e.g., cuDNN) exhibit implementation-level variance across all evaluated versions.

Our researcher and practitioner survey shows that 83.8% of the 901 participants are unaware of or unsure about any implementation-level variance. In addition, our literature survey shows that only 19.5±3% of papers in recent top software engineering (SE), artificial intelligence (AI), and systems conferences use multiple identical training runs to quantify the variance of their DL approaches. This paper raises awareness of DL variance and directs SE researchers to challenging tasks such as creating deterministic DL implementations to facilitate debugging and improving the reproducibility of DL software and results.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
ASE '20, September 21–25, 2020, Virtual Event, Australia
© 2020 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6768-4/20/09…$15.00
https://doi.org/10.1145/3324884.3416545

CCS CONCEPTS

• **Software and its engineering** → **Empirical software validation**; • **General and reference** → *Empirical studies.*

KEYWORDS

deep learning, variance, nondeterminism

ACM Reference Format:
Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. 2020. Problems and Opportunities in Training Deep Learning Software Systems: An Analysis of Variance . In *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20), September 21–25, 2020, Virtual Event, Australia.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3324884.3416545

1 INTRODUCTION

Deep learning is widely used in many fields including autonomous driving cars [19], diabetic blood glucose prediction [78], and software engineering [18, 20, 21, 52, 63, 87, 114, 117, 119]. DL training algorithms utilize nondeterminism to improve training efficiency and model accuracy, e.g., using shuffled batch ordering of training data to prevent overfitting and speed up training [49].

These *nondeterminism-introducing (NI)-factors* cause multiple *identical training runs*, i.e., training runs with the *same* settings (e.g., identical training data, identical algorithm, and identical network), to produce different DL models with significantly different accuracies and training times [57, 88, 101, 104].

For example, our experiments show that for 16 identical training runs of a popular DL network, LeNet5 [65], the accuracy of the resulting 16 models ranges from 8.6% to 99.0% —a large accuracy difference of 90.4%. Four of these identical training runs resulted in *weak models* (accuracy below 20%). Even if we exclude such models, the accuracy difference is still up to 10.8% with LeNet1 between the most accurate run (98.6%) and the least accurate run (87.8%).

One can eliminate the variance introduced by *algorithmic NI-factors* (e.g., shuffled batch ordering) using fixed random seeds. For example, with a fixed seed for batch ordering, multiple identical training runs will have the same batch ordering. We refer to these runs as *fixed-seed identical training runs*.

In addition to these algorithmic NI-factors, DL libraries (e.g., TensorFlow [13] and cuDNN [23]) introduce additional variance.

For example, by default, *core DL libraries* (e.g., TensorFlow or PyTorch [82]) perform data preprocessing in parallel for speed, which changes the order of training data, even if algorithmically the batch order is fixed. In addition, core DL libraries, by default, leverage *autotune* to automatically benchmark several modes of operation (i.e., underlying algorithms for computations such as addition) for each primitive cuDNN function (e.g., pooling and normalization) in its first call. The fastest mode of operation is then used for that cuDNN function in subsequent calls. Different identical runs sometimes use different modes of operation, introducing variance.

Our experiments (Section 5.2) show that these *implementation-level NI-factors* alone cause an overall accuracy difference of up to 2.9%. Specifically, we train the popular WideResNet-28-10 [112] (or WRN-28-10 for short) DL network for image classification 16 times using the same default training configuration (e.g., same CIFAR100 [61] training data, batch size, optimizer, and learning rate schedule), identical model selection criteria (i.e., selecting models with the lowest loss on a validation set), the same DL libraries (i.e., Keras 2.2.2, TensorFlow 1.14.0, CUDA 10.0, and cuDNN 7.6), and identical hardware (i.e., the same NVIDIA RTX 2080Ti GPU), while disabling all algorithmic NI-factors (i.e., using fixed random seeds to ensure identical initial weights, identical batch ordering, deterministic dropout layers, and deterministic data augmentation). This process generates 16 models. Since all algorithmic NI-factors are disabled, one may expect little variance across the 16 runs. However, we found that the accuracies of these 16 models vary between 77.3% and 80.2% (a 2.9% difference).

These implementation-level NI-factors and differences are often characteristics and consequences of the DL *software implementations*, which create unique challenges for SE researchers and practitioners (Section 8), while DL researchers have paid little attention to implementation-level NI-factors (the focus is on theoretical analyses of DL training [26, 27, 35, 58, 68, 90]).

To see whether such difference is known, we conduct a survey (Section 6), which surprisingly shows, that 83.8% of the 901 responded researchers and practitioners with DL experience are unaware of (63.4%) or unsure about (20.4%) any implementation-level variance! Of the 901 respondents, only 10.4% expect 2% or more accuracy difference across fixed-seed identical training runs.

We also perform a literature survey of 454 papers *randomly* sampled from recent top *SE*, *AI*, and *systems* conferences to understand the awareness and practices of handling DL system variance in research papers. Of 225 papers that train and evaluate DL systems, only 19.5±3% use multiple identical training runs to quantify the variance of their DL approaches.

The per-class accuracy exhibits a much larger difference among the 16 fixed-seed identical training runs. For example, in the previously described WRN-28-10 runs, for the "camel" class (all images with the ground-truth label of "camel"), the models' accuracy varies from 38.1% to 90.5% (a 52.4% difference).

In addition, there are large differences in convergence time. For example, the time to convergence of the 16 fixed-seed identical training runs of another popular network, ResNet56 [51], ranges from 2,986 to 7,324 seconds (a one hour and 12 minutes difference)—a 145.3% relative time difference. We also observe a discrepancy between the empirical per-class accuracy and convergence time and the corresponding estimates from the surveyed researchers and practitioners.

Thus, it is important to study and quantify the variance of DL systems, especially the implementation-level variance. On the one hand, some practitioners, whose primary goal is to obtain the best model, may be able to take advantage of the variance by running multiple identical runs to achieve their goal.

On the other hand, SE, AI, and systems researchers who propose new DL architectures and models that outperform existing ones may need to execute multiple identical runs to ensure the validity of their experiments.

For example, a recent research paper [64] proposed a new approach with a reported 0.8% accuracy improvement over the standard WRN-28-10. Our experiments show that this network's accuracy can vary by up to 2.9%. Therefore, the reported accuracy improvement may not be statistically significant when considering the aforementioned NI-factors in the training algorithm and the implementation. In other words, if one runs the two approaches again, the resulting mode of [64] may not outperform the WRN-28-10 model. At best, the comparison results still hold, but the current experiments fail to provide evidence to demonstrate the improvement given the possible variance.

There are existing theoretical analyses of DL training [26, 27, 35, 58, 68, 90] that study how well optimizers find good local optima given algorithmic NI-factors. Such work fails to study the nondeterminism in the underlying DL implementation.

To fill this gap, first, we systematically study and quantify the accuracy and time variance of DL systems across identical runs using 6 widely-used networks and 3 datasets. Second, we conduct a survey to ascertain whether DL variance and their implications are known to researchers and practitioners with DL experience. Third, we conduct a literature review of the most recent editions of top SE, AI, and systems conferences to understand the awareness and practices of coping with DL variance in research papers.

In this paper, we make the following contributions:

- ♦ **Finding 0**: A list of implementation-level NI-factors (parallel process, auto-selection of primitive operations, and scheduling and floating-point precision) and algorithmic NI-factors (nondeterministic DL layers, weight initialization approach, data augmentation approach, and batch ordering), and techniques that control these NI-factors to remove or reduce variance (Section 2).
- ♦ A DL variance study of 4,838 hours (over 6.5 months) of GPU time on 3 widely-used datasets (MNIST, CIFAR10, CIFAR100) with 6 popular models (LeNet1, LetNet4, LetNet5, ResNet38, ResNet56, and WideResNet-28-10) on three core DL libraries (TensorFlow, CNTK, and Theano):
  - **Finding 1**: The accuracy of models from 16 identical training runs varies by as much as 10.8%, even after removing weak models.
  - **Finding 2**: With algorithmic NI-factors disabled, DL model accuracy varies by as much as 2.9%—an accuracy difference caused solely by implementation-level nondeterminism.
  - **Finding 3**: Implementation-level NI-factors cause a per-class accuracy difference up to 52.4%, while the per-class difference is up to 100% with default settings (i.e., with algorithmic and implementation-level nondeterminism).

- **Finding 4**: Training time varies by as much as 145.3% (1 hour and 12 minutes) among fixed-seed identical training runs, while the training time difference is up to 4,014.8% with default settings.

♦ A researcher and practitioner survey, with 901 valid replies, reveals that:
- **Finding 5**: A large percentage of respondents are unaware of (31.9%) or unsure about (21.8%) *any variance* of DL systems; there is no correlation between DL experience and awareness of DL variance.
- **Finding 6**: Even more researchers and practitioners (83.8%) are unaware or uncertain of *implementation-level* nondeterminism in DL systems.
- **Finding 7**: Only 10.4% of respondents expect 2% or more accuracy difference across fixed-seed identical training runs.
- **Finding 8**: Most (77.7%) participants estimate the convergence time differences to be less than 10% across identical training runs, and the majority of respondents (84.5%) estimates a similar 10% or less convergence time difference among fixed-seed identical training runs.

♦ **Finding 9**: A literature survey of a random sample of 454 papers from the most recent editions of top SE (ICSE [7], FSE [4], and ASE [1]), AI (NeurIPS/NIPS [9], ICLR [11], ICML [6], CVPR [3], and ICCV [5]), and systems (ASPLOS [2], SOSP [10], and MLSys [8]) conferences, shows that 225 papers train and evaluate DL systems, only 19.5±3% of which use multiple identical training runs to evaluate their approaches.

♦ **Implications and suggestions** for researchers and practitioners, and raising awareness of DL variance (Section 8).

Code and data are available in a GitHub repository[1].

2 NONDETERMINISM-INTRODUCING (NI)-FACTORS

Many factors affect DL systems' results and training time. The first set of factors is the *input* to a system. Such input includes the training data and hyper-parameters (e.g., number of layers, dropout rate, optimizer, learning rate, batch size, and data augmentation method and settings). It is expected that models with different inputs perform differently, and there is a flurry of work on how to select the best input (e.g., hyper-parameter tuning) [17, 54, 111].

However, several factors (e.g., shuffled batch ordering) independent of the system's input affect the training and accuracy of the final models. We call these factors NI-factors. We divide NI-factors into two categories: (1) algorithmic NI-factors, which are introduced to improve the effectiveness of the training algorithm, and (2) implementation-level NI-factors, which are the byproduct of optimizations to improve DL implementations' efficiency.

2.1 Definitions

In this study, we define a *DL system* as the composition of a *DL algorithm* and a *DL implementation*. DL algorithm is the theory portion of DL and consists of model definition, hyper-parameters, and theoretical training process. DL implementation consists of *high-level DL libraries* (e.g., Keras), *core DL libraries* (e.g., TensorFlow and PyTorch), *low-level computation libraries* (e.g., cuDNN and CUDA), and hardware (e.g., GPU, CPU, and TPU). The DL training process spans across the DL algorithm and DL implementation.

2.2 Algorithmic NI-factors

The most common algorithmic NI-factors include nondeterministic DL layers (e.g., dropout layer), weight initialization, data augmentation, and batch ordering.

Nondeterministic DL Layers: DL architectures can contain nondeterministic layers. For example, dropout layers [99] are commonly used to prevent overfitting. They randomly set parts of the input tensor to zero during training and guide each neuron to be trained with different portions of the training data. *Dropout* tensors are chosen randomly on-the-fly during training, which means two identical runs could produce two different models with different accuracies and different training times.

Weight Initialization: Weight initialization [42, 50, 66, 90] is an important step in DL training [101, 104]. Random weight initialization samples the initial DL model weights from a predefined distribution. Goodfellow et al. [45] state that random initialization "breaks the symmetry" across all the weight tensors. This process helps similarly structured neurons learn different functions instead of repeating each other. Thus, they learn different aspects of the training data and help to increase the model's generalization. However, different initial weights may result in convergence to different local minima [36, 40, 67]. Therefore, random initialization can lead to variance in model accuracy across identical runs.

Data Augmentation: DL training algorithms also utilize the randomness in data augmentation to improve their effectiveness (i.e., produce more accurate models). Data augmentation [96] is an inexpensive method that randomly transforms the input to increase the input's diversity. It has been shown to improve the generalization of the final trained model. Randomly transforming the training data will result in nondeterministic identical training runs.

Batch Ordering: Random batch ordering also improves the generalization of DL models. It breaks up the order of the training data to prevent the model from quickly overfitting to a particular label [66]. Reordering training batches at each epoch results in nondeterministic identical training runs.

2.3 Implementation-level NI-factors

Implementation-level NI-factors are caused by libraries (e.g., TensorFlow, CUDA, and cuDNN). The most common implementation-level NI-factors are parallel computation, nondeterministic primitive operations, and rounding errors due to scheduling.

Parallel processes: Core DL libraries (e.g., TensorFlow and Pytorch) provide options to use multiple processes to improve the efficiency of DL systems. For example, the core libraries, by default, run the data preprocessing task in parallel to prepare the training data faster. However, due to the random completion order of parallel tasks, the order of training data may change and impact the optimization path of the training process, resulting in variance even if data preprocessing itself is deterministic.

[1] https://github.com/lin-tan/dl-variance

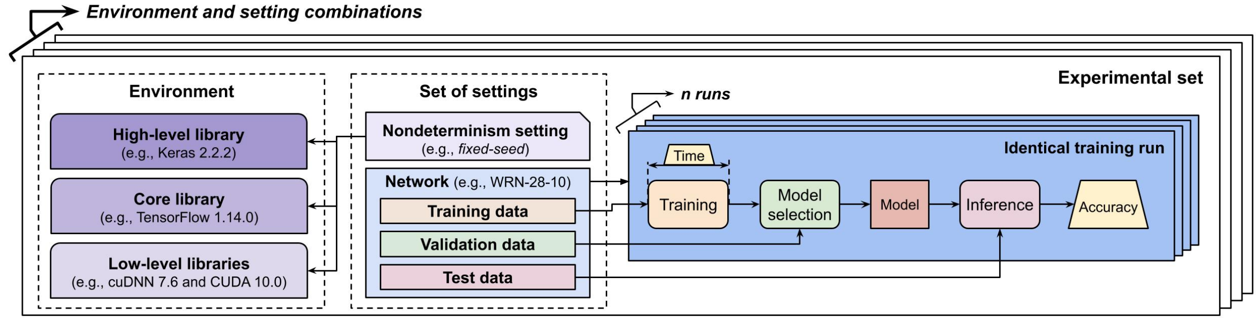

Figure 1: Overview of the experimental method

Auto-selection of primitive operations: Core DL libraries implement DL algorithms by leveraging the GPU-optimized DL primitives provided by low-level libraries (e.g., cuDNN and CUDA). When used with NVIDIA GPUs, cuDNN provides hardware-accelerated primitives for common DL operations such as forward and backward convolution, pooling, normalization, and activation layers. cuDNN provides several modes of operation (i.e., different computational algorithms) for primitive functions.

By default, core DL libraries enable *autotune*, which automatically benchmarks several modes of operation for each primitive cuDNN function in its first call. The fastest mode of operation is then used for that cuDNN function in subsequent calls. The exact mode of operation for each primitive used in each run changes depending on the dynamic benchmark result. Different identical runs sometimes use different modes of operation, introducing variance. Furthermore, some modes of operation are nondeterministic due to rounding errors introduced by scheduling (see below).

Scheduling and floating-point precision: GPU programming uses *warp* as a unit of computation. A warp consists of 32 parallel threads with concurrent memory access. Due to the limited precision (32-bits) of DL models, rounding errors are introduced at every step of floating-point calculation. In the GPU model, concurrent accesses to a single memory address must be serialized to prevent race conditions with no guaranteed order of access. Therefore, the rounding error introduced in each warp may vary across fixed-seed identical training runs due to the different accessing orders.

For example, matrix reduction operations such as `atomicAdd` (used in depthwise convolution layers) are affected by the varying serialization order. Since floating-point operations are not associative due to rounding errors [43], varying orders of additions may produce nondeterministic output ($A + B + C \neq B + C + A$).

2.4 Controlling NI-factors for determinism

Removing algorithmic NI-factors enables us to study the nondeterminism introduced by implementation-level NI-factors. In addition, deterministic training may be desirable for debugging and other purposes. Thus, we identify techniques that control algorithmic and implementation-level NI-factors to remove or reduce variance.

Controlling algorithmic NI-factors: All algorithmic NI-factors are controlled by pseudo-random number generators. Thus, we can control these NI-factors by fixing the random seeds at the beginning of each run to achieve algorithmic determinism across identical runs while still maintaining the pseudo-random characteristic within a single run. We defined this as fixed-seed identical training runs.

Controlling implementation-level NI-factors: To control these NI-factors, we need to take several steps. First, the DL system should not use multiple processes that cannot guarantee data order. For example, using more than one worker in the data generator to feed training data would shuffle the batch ordering even with fixed random seeds.

Second, the *autotune* mechanism should choose deterministic implementations of primitive operations only. For example, in TensorFlow 1.14.0, if the environment flag `TF_CUDNN_DETERMINISTIC` is set to 1, the autotune mechanism will not consider the nondeterministic modes for cuDNN primitive functions.

Third, since some operations (e.g., `atomicAdd`) are nondeterministic when used on a GPU due to nondeterministic serialization, the input of these operations should be serialized after all parallel executions (i.e., to ensure a deterministic ordering of input). Then, the operations should be executed on a single CPU thread.

Finally, one solution to achieve complete deterministic training is forcing the DL system to run completely in a serial manner (i.e., running on a single CPU thread). However, this option prevents DL systems to utilize the hardware efficiently and may be unrealistic, as many models would take months or years to train on a single CPU thread. As future work, deterministic multithreading [32] may be promising for more realistic deterministic DL systems.

A major goal of this paper is to quantify the variance introduced by implementation-level NI-factors.

3 EXPERIMENTAL METHOD

First, we extract the *default input* (i.e., training data, hyper-parameters, and optimizers) of a DL system from existing work (Section 4). Figure 1 shows an overview of our experimental method. We generate different *environments* that combine different versions of DL libraries (e.g., high-level libraries, core libraries, and low-level libraries). For all environments, the hardware is the same (details in Section 4). For example, one such environment includes Keras 2.2.2, TensorFlow 1.14.0, cuDNN 7.6, and CUDA 10.0. Each network is coupled with its default input. For example, the CIFAR 100 dataset (including training, validation, and test data) is used to train the WRN-28-10 network with stochastic gradient descent (SGD) optimizer in 50 epochs. Table 1 shows the corresponding default input for each network. Each network (including its default input) combined with one *nondeterminism setting* (details below) is defined as a set of *setting*. For example, one set of settings that we use is training the WRN-28-10 network with all algorithmic NI-factors disabled (i.e., fixed-seed nondeterminism setting). For each environment and

setting combination, we perform an experimental set and measure the accuracy and time variance across n identical training runs.

To ensure a valid study, we address one main challenge: to measure realistic variance, the experiments need to reflect the real usage of DL systems. We pick training from-scratch as our scenario for the training phase because it is a common and fundamental training scenario [14, 51, 94, 98, 103, 112, 122], i.e., we train a new DL model from the beginning, starting from randomly initialized weights.

In addition, we focus on studying the variance of models' overall accuracy, per-class accuracy, and training time, as these are common metrics that DL researchers and practitioners use, and there have been many techniques [14, 22, 38, 45, 103, 118] proposed to improve on these metrics.

Further, to make sure the accuracy and time that we observe are valid, we check that our results are equivalent to the ones reported in the original papers. Training inputs (e.g., training data, prepossessing methods, and optimizer) are chosen to match, as closely as possible, those reported or used in the authors' code. While a complete reproduction is often impossible, we reproduce previous work as faithfully as possible by using reported settings and ensuring at least one of our runs can reach almost identical accuracy to that of the original work.

Finally, we perform two statistical tests (Levene's test for variance and Mann Whitney U-test for mean) to ensure that we draw statistically significant conclusions.

3.1 Experimental sets of identical training runs

The training phase is an iterative process and after each iteration (i.e., epoch) a checkpoint of the model is stored. After the training finishes, the best checkpoint is selected based on a final *model selection* criterion. We focus on two common (77.5% of the respondents in our survey use one of these criteria) model selection criteria:

- **Best-loss selection criterion:** The final model is the checkpoint with the best (i.e., lowest) validation loss.
- **Best-accuracy selection criterion:** The final model is the checkpoint with the best (i.e., highest) validation accuracy.

Validation loss and accuracy are calculated on the validation set (i.e., unseen data, different from the training data, and used to tune the model). We report the test accuracy of the selected best model which is calculated on the test data (i.e., unseen data, different from the training and validation data).

In practice, the training runs would end if the selection metric (i.e., validation loss or accuracy) did not improve after a set number of epochs (i.e., patience). In this study, we instead run the training to a maximum number of epochs while storing the model checkpoints. Once the training is done, we select the model based on the selection criterion and then compute the training time as if the training had stopped at the best checkpoint. This is an estimation of training time without running a separate set of experiments for each criterion.

We define *identical training runs* as training runs executed with the *same environment* (i.e., hardware and DL libraries), the *same network architecture*, and the *same inputs* (i.e., training data, hyper-parameters, and optimizers). Each identical training run is followed by an inference run on the test data to compute the model accuracy.

An *experimental set* is a group of identical training runs. We make sure to avoid measurement bias [79] as much as we could by using the same machine along with Docker environments that are built from the same base image. The only changes across experimental sets are the DL library combinations and the set of settings (i.e., the nondeterminism-level, the network, and its default input). In each experimental set, we perform $n = 16$ *runs*.

3.2 Nondeterminism-level settings

We perform two categories of experiments with different nondeterminism-levels: the default and fixed-seed settings.

Default identical training runs are experiments that do not enforce determinism (i.e., none of the NI-factors are controlled). These are identical training runs with the default input (training data and hyper-parameters).

Fixed-seed identical training runs are experiments for which algorithmic NI-factors are disabled, i.e., we use the same random generator and the same seed. For example, with the TensorFlow core library, we set the global Python random seed, Python hash seed, Numpy random seed, and the TensorFlow random seed to be identical. Initializing all random number generators with identical seed disables all algorithmic NI-factors (i.e., dropout layers, initial weights, data augmentation, and batch ordering).

3.3 Metrics and measurements

To measure the variance across identical training runs, we measure a model's overall and per-class accuracy on the test set. The *overall accuracy* measures the portion of correct classifications that a model makes on test samples. The *per-class accuracy* splits the overall accuracy into separate classes based on the ground-truth class labels (i.e., the accuracy of the model for each class). For example, the MNIST dataset has 10 classes, so an MNIST model would have 10 per-class accuracy values (one for each digit). For all identical training runs, we measure the total training time as well as the number of epochs until convergence (i.e., until the checkpoints specified by the selection criterion). For each experimental set, the maximum difference shows the most extreme gap of model accuracy and training time between the best and the worst runs.

3.4 Statistical tests

Levene's test is a statistical test to assess the equality of variance of two samples. Specifically, when testing accuracy variance, the null hypothesis is that the accuracy variance of set A is equal to the accuracy variance of set B. If we find that p-value < 0.05 then we can confirm with 95% confidence the accuracy variance of set A is different from set B. Thus, if the accuracy variance of set A is smaller, then runs in set A are more stable than in B.

Mann Whitney U-test is a statistical test to assess the similarity of sample distributions. We run the U-test instead of the T-test because the U-test does not assume normality while the T-test does. For example, when comparing two sets of runs (A and B), the null hypothesis is that the accuracies of set A are similar to set B. If we find that p-value < 0.05, then we can confirm with 95% confidence that our alternative hypothesis is true which means set A has statistically different accuracies than set B. We compute the effect size as the Cohen's d [28] to check if the difference has a meaningful effect ($d = 0$: no effect and $d = 2$: huge effect [89]).

Table 1: Datasets, networks, and training settings

Dataset	#samples			Network		Settings	
	Train	Val	Test	Name	#parameters	#epochs	Optimizer
MNIST	60,000	7,500	2,500	LeNet1	7,206	50	SGD
				LeNet4	69,362		
				LeNet5	107,786		
CIFAR10	50,000	7,500	2,500	ResNet38	565,386	200	Adam
				ResNet56	857,706		
CIFAR100	50,000	7,500	2,500	WRN-28-10	36,536,884	200	SGD

4 EXPERIMENTAL SETTINGS

Datasets and models: We perform our experiments using three popular datasets: MNIST [65], CIFAR10 [61], and CIFAR100 [61]. We choose image classification architectures as they are often used as test subjects in recent SE papers that test [41, 71, 73, 85, 100, 105, 106, 109, 115, 120], verify [46, 83], and improve [39, 56, 74, 110, 113, 116] DL models and libraries. Table 1 shows each dataset with the corresponding numbers of instances in each subset (training, validation, and test). The training set is used to train the model. Following common practice [14, 24, 45, 51, 62, 97, 102], we use the validation set to select the best model. The test set is used to evaluate the final model.

We choose to experiment on LeNet [65], ResNet [51], and WideResNet [112] architectures as they are popular networks for image classification. In our literature review, of the 225 relevant papers, 64% of papers use or compare to one of these architectures. Table 1 also shows the number of trainable parameters for each network. Our networks are diverse in size, from 7,206 (LeNet1) to 36,536,884 parameters (WRN-28-10).

We reproduce previous work as faithfully as possible by using networks and settings recorded by previous work and ensuring some of our runs have a similar (within 1%) accuracy as that in the original work. We also use (when available) the original implementation of the approach from the author and the Keras implementation (if available) to reduce the risk of introducing new bugs.

We list the training configurations used in table 1. To ensure that the model will converge (i.e., training loss stops improving) within the maximum number of epochs (Table 1), we empirically choose a maximum number of epochs larger than the number of epochs to convergence using both selection criteria.

We cannot use networks from prior work [84, 85, 105] since they only provide pre-trained models and do not provide enough details for us to reproduce the training runs.

DL libraries: We use Keras version 2.2.2 [25] as our high-level library since it provides us with the ability to transparently switch between three DL core libraries (TensorFlow [13], CNTK [16], and Theano [91]). This ensures that the comparison across core libraries is fair and the least affected by our code. We perform our experiments with the official TensorFlow versions (including the latest) (1.10, 1.12, and 1.14), CNTK version (2.7), and Theano version (1.0.4). We pair each version of the core libraries with the officially supported low-level cuDNN and CUDA versions. For example, TensorFlow 1.12 supports cuDNN 7.3 to 7.6 coupled with CUDA 9.0, while TensorFlow 1.14 supports only cuDNN 7.4 to 7.6 coupled with CUDA 10.0. Since it is not practical to perform experiments on all library combinations, we use 11 library combinations for TensorFlow, one combination each for CNTK and Theano.

Table 2: Maximum differences of *overall* and *per-class* accuracy among *default* and *fixed-seed* identical training runs

Setting	Network	Overall(%)		Per-class(%)	
		Diff	SDev (SDevCI)	Diff	SDev (SDevCI)
Default	LeNet1	10.8	2.6 (2.0-3.8)	99.6	24.5 (19.0-35.2)
	LeNet4	10.6	2.6 (2.0-3.7)	**100.0**	24.7 (19.1-35.5)
	LeNet5	**90.4**	38.7 (30.0-55.6)*	**100.0**	44.5 (34.4-63.9)
	ResNet38	1.9	0.5 (0.4-0.7)	11.7	2.8 (2.2-4.1)
	ResNet56	2.1	0.6 (0.4-0.8)	11.9	2.8 (2.2-4.0)
	WRN-28-10	2.8	0.8 (0.6-1.1)	50.0	13.3 (10.3-19.1)
Fixed-seed	LeNet1	0.1	<0.1 (<0.1)	0.8	0.3 (0.2-0.4)
	LeNet4	0.5	0.1 (0.1-0.2)	1.9	0.6 (0.5-0.9)
	LeNet5	1.2	0.3 (0.2-0.4)	4.8	1.3 (1.0-1.9)
	ResNet38	2.7	0.6 (0.5-0.8)	12.2	3.3 (2.6-4.8)
	ResNet56	1.9	0.5 (0.4-0.7)	10.6	2.3 (1.8-3.3)
	WRN-28-10	**2.9**	0.7 (0.6-1.0)	**52.4**	16.3 (12.6-23.4)

* 4/16 runs produce weak models that have lower than 20% accuracy

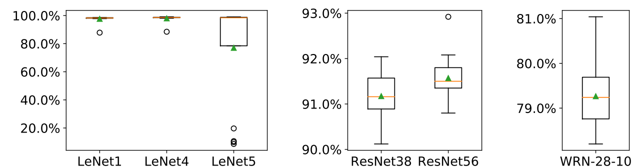

Figure 2: Boxplots of the overall accuracy for default identical runs with the largest overall accuracy difference

Infrastructure: We carry out all experiments on a machine with 56 cores, 384GB of RAM, and RTX 2080Ti graphic cards each with 11GB memory. To accommodate multiple combinations of libraries, we use Anaconda (4.4.10) with Python (3.6) and Docker (19.03).

5 RESULTS AND FINDINGS

We perform 2,304 identical training runs (144 experimental sets with 16 runs each) of six networks on three datasets, with two levels of nondeterminism, using three core libraries (TensorFlow, CNTK, and Theano) which is 4,838 hours (**over 6.5 months**) of GPU time.

5.1 RQ1: How much accuracy variance do NI-factors introduce?

To investigate the variance caused by NI-factors, we run 16 default identical training runs for each of the 66 experimental sets (i.e., combinations of 6 networks and 11 environments). Recall that default identical training runs are defined as training runs with the same default inputs where no NI-factors are disabled (Section 3.1).

To estimate the *extreme case*, we compute the maximum difference of accuracy (overall and per-class) between the least accurate and the most accurate default identical training runs of an experimental set while the standard deviation estimates the *average case*.

Table 2 (*Default*) shows results for RQ1. Columns *Diff* show the maximum differences of accuracy while columns *SDev* and *(SDevCI)* shows the standard deviation of accuracy among 16 identical training runs and corresponding confidence interval (i.e., with 90% confidence, the confidence interval would contain the population standard deviation). We only show the larger accuracy differences when using either selection criterion (best-loss or best-accuracy) as the results are similar between the criteria. Figure 2 shows the

boxplots of the overall accuracy of each network. The triangles represent the mean accuracy and the orange line is the median. Dots outside of the whiskers are outliers.

Across default identical training runs, the accuracy difference is as big as 10.8%, even after removing weak models. (**Finding 1**).

Specifically, in the LeNet5 default training experimental set (with TensorFlow 1.14.0, CUDA 10.0, cuDNN 7.5, and loss selection criterion), the most and least accurate runs have an overall accuracy of 99.0% and 8.6% respectively (a 90.4% difference). The worst model's accuracy is lower than random guesses (i.e., 10% because the MNIST dataset has 10 classes). This large accuracy difference is caused by the random initialization of the weights [101, 104]. Particularly, four runs do not improve much after training—with the final models' accuracies being 8.6%, 9.9%, 10.6%, and 19.7% (the outliers shown as circles in Figure 2). While the four runs produce weak models, they are faithful reproductions of training with widely-used networks and algorithms using realistic data and settings. The fact that 4 out of 16 runs fail to improve significantly, shows the *importance of reporting the variance* between multiple identical training runs so that *the DL approaches can be evaluated on not just their best accuracy, but also on how stable the training process is.*

If we exclude networks with such weak models, we still see an accuracy difference up to 10.8% with LeNet1 (the difference between 87.8% and 98.6%). For WRN-28-10, the largest difference is 2.8% (between 78.2% and 81.0%) respectively. Although these differences may seem small, researchers [64] report improvements of 0.8% when comparing against WRN-28-10 without accounting for NI-factors. At best, the comparison conclusions still hold, but the papers fail to provide evidence for that.

The per-class accuracy differences are even larger compared to the overall accuracy differences (Table 2, column *Default: Overall* versus column *Default: Per-Class*). On the least accurate run of LeNet5, the trained model fails completely on a single class (i.e., the prediction accuracy for the class digit "0" is 0%), while, for other runs, the highest prediction accuracy for the same class is 100%. Digit "0" has 261 test images (all classes have similar numbers) so such single-class failures are not due to insufficient instances or bias distribution of that class. A similar single-class failure happens for LeNet1 and LeNet4 training runs. The standard deviation is smaller for these networks (24.5% and 24.7% comparing to 44.5%) because only one run completely fails.

As another example, WRN-28-10 default identical training runs (using library combination TensorFlow 1.12.0, CUDA 9.0, cuDNN 7.6, and best-accuracy selection criteria) incur a maximum overall accuracy difference of 2.8%. With the same settings, the per-class accuracy difference is 50.0% (dropping from 72.7% to 22.7%) for the class "bee" (with 22 test samples). Per-class accuracy variance can be problematic for applications where the accuracy of specific classes is critical. For example, the accuracy variance of the pedestrian class of a self-driving car's object classification system could vary pedestrian prediction reliability. This, in turn, could endanger pedestrians, even if the overall variance of the model is small.

NI-factors cause a complete single-class failure, where the biggest per-class accuracy difference is 100% with a standard deviation of 44.5% (**Finding 3(a)**).

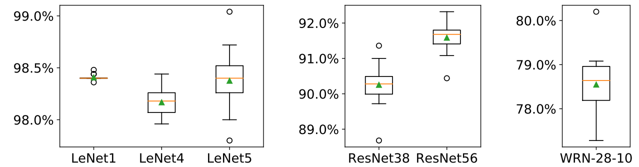

Figure 3: Boxplots of the overall accuracy for fixed-seed identical runs with the largest overall accuracy difference

5.2 RQ2: How much accuracy variance do implementation-level NI-factors cause?

Accuracy variance: We analyze nondeterminism introduced by implementation-level NI-factors by performing 66 experimental sets (i.e., combinations of 6 networks with 11 environments) of fixed-seed identical training runs (each with 16 runs). Recall that fixed-seed identical training runs are default identical training runs with algorithmic NI-factors disabled using fixed random seed initialization (Section 3.2).

Table 2 (*Fixed-seed*) shows the largest accuracy differences of the overall and per-class accuracy of all models (for any library combinations and selection criteria) with disabled algorithmic NI-factors (i.e., among fixed-seed identical training runs).

Implementation-level NI-factors cause accuracy differences as large as 2.9% (**Finding 2**), while per-class accuracy differences are up to 52.4% (**Finding 3 (b)**).

Among the fixed-seed identical training runs of WRN-28-10 (with TensorFlow 1.14.0, CUDA 10, cuDNN 7.6, and loss selection criterion), the most and the least accurate runs have an overall accuracy of 80.2% and 77.3% respectively. In the same experimental set, the implementation-level NI-factors cause a per-class accuracy difference of 52.4% (the "camel" class–with 21 test samples– has 90.5% and 38.1% accuracies in the most and least accurate run). All other classes have similar numbers of test samples so the large per-class accuracy difference is not due to insufficient instances or bias distribution classes.

The lack of complete failure caused by the random weight initialization (an algorithmic NI-factors) in LeNet training (Figure 3) indicates that training is more stable without algorithmic NI-factors.

When comparing the results of setting *Default* and *Fixed-seed* in Table 2, LeNet and ResNet56 have smaller overall and per-class accuracy differences among default identical training runs. While for ResNet38 and WRN-28-10, the accuracy differences among fixed-seed identical training runs are smaller. Levene's test cannot statistically confirm the significance (p-value > 0.05) of these differences in variance for all networks except for LeNet5 (where there are complete failures in identical training runs with default setting).

Table 2 (*Fixed-seed*) shows that except for ResNet38, the more complex a network is (i.e., more trainable parameters), the larger the accuracy (overall and per-class) variance exists across fixed-seed identical training runs. For more complex networks, the error introduced by nondeterminism might propagate further.

To demonstrate the importance of performing identical training runs when comparing different DL approaches, we consider a scenario where ResNet56 is a baseline approach to the CIFAR10 image classification problem and ResNet38 is the proposed improvement.

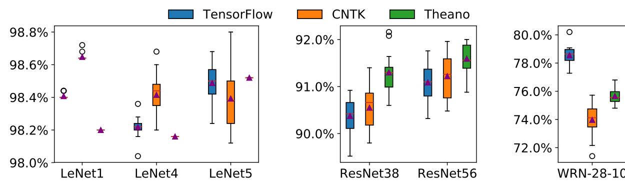

Figure 4: Boxplots of the overall accuracy of fixed-seed identical training runs with different core libraries

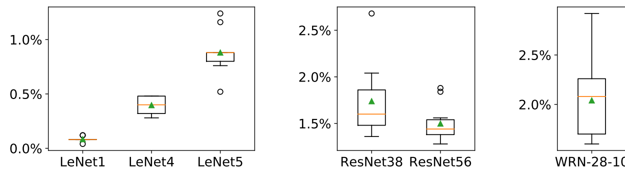

Figure 5: Boxplots of the overall accuracy difference of fixed-seed identical training runs with 11 low-level library version combinations for each network

Among 16 fixed-seed identical training runs, ResNet56 averages 91.2% in test accuracy while ResNet38 averages 90.3%. The U-test confirms (with p-value < 0.01) that ResNet56 has 0.9% higher test accuracy than ResNet38 with an effect size (Cohen's d) of 1.7 (very large effect). Hence, there is no improvement from the proposed technique (ResNet38) over the baseline (ResNet56). However, if each approach only runs once, in the most extreme case, ResNet56 accuracy is reported with its worse run (90.4%) and ResNet38 with its best run (91.4%), the researchers might have come to an invalid conclusion that ResNet38 has 1% higher accuracy than ResNet56. Researchers and practitioners should be aware of DL system variance, even with only implementation-level NI-factors, so they would perform multiple identical training runs when comparing approaches.

Different core libraries: We investigate if switching core libraries leads to different accuracy variance among fixed-seed identical training runs. Since it is prohibitively expensive to run all combinations of core and low-level library versions (our experiments' GPU time are already over 6.5 months), we compare the latest versions of core and low-level libraries at the time of the experiment (i.e., in addition, we run 12 more experiment sets – combinations of 6 models with 2 environments).

Figure 4 shows the boxplots of the overall accuracy of fixed-seed identical training runs for the experimental set of each network with the best-loss selection criterion across three different core libraries. The accuracy variance is similar across different core libraries. For example, for ResNet56 the accuracy difference with CNTK is 1.5% (between 91.8% and 90.3%) and 1.9% with TensorFlow. All core libraries are affected similarly by implementation-level NI-factors, as Levene's test cannot reject the null hypothesis that each core library has a different accuracy variance (p-value > 0.1).

Different low-level libraries versions: We analyze the overall accuracy differences of the 11 low-level library combinations (cuDNN and CUDA) with TensorFlow to see if there is still variance when switching versions of the low-level libraries. Figure 5 shows the boxplots of the overall accuracy differences of fixed-seed identical training runs when training each network with each of the 11 library combinations. All training runs are affected by implementation-level NI-factors, independently from the low-level libraries used. For example, with WRN-28-10, the largest overall accuracy difference is 2.9% (reported in Table 2). On average, across 11 experimental sets, the accuracy difference for this network is over 2% while the smallest accuracy difference is 1.6%.

Table 3: Running time to convergence differences among *default* and *fixed seed* identical training runs

Setting	Network	Time_{Loss} (seconds)			Time_{Acc} (seconds)		
		Diff	RelDiff	RelSDev	Diff	RelDiff	RelSDev
Default	LeNet1	27	24.5%	6.5%	41	47.2%	8.5%
	LeNet4	22	17.4%	4.7%	25	19.9%	4.0%
	LeNet5	155	3,940.7%*	54.2%	158	**4,014.8%***	55.1%
	ResNet38	434	21.4%	5.3%	2,953	133.2%	18.7%
	ResNet56	699	23.9%	6.0%	3,813	116.5%	17.7%
	WRN-28-10	2,333	12.9%	3.2%	**6,316**	46.0%	8.8%
Fixed-seed	LeNet1	17	14.3%	3.8%	18	14.4%	3.8%
	LeNet4	17	13.3%	3.6%	25	20.1%	6.2%
	LeNet5	31	25.8%	5.6%	37	30.0%	6.0%
	ResNet38	415	20.4%	4.4%	2,782	115.5%	15.9%
	ResNet56	467	16.4%	3.6%	4,338	**145.3%**	22.5%
	WRN-28-10	2,197	12.2%	2.9%	**5,625**	38.3%	10.1%

* 3/16 runs stuck at the first epoch

5.3 RQ3: How much training-time variance do NI-factors introduce?

We study the variance in overall training time to convergence of default identical training runs and fixed-seed identical training runs which is often the primary variance that researchers and practitioners care about. We measure training time to convergence with respect to best-loss and best-accuracy selection criteria.

Table 3 shows the analysis of the running time to convergence for default identical training runs and fixed-seed identical training runs. Time_{Loss} and Time_{Acc} denote the training time using two popular model selection criteria—best-loss and best-accuracy, respectively. For each selection criterion, the table shows the time difference between the slowest and the fastest runs (columns *Diff*). Since the running time is very different across networks, we compute the relative time difference (columns *RelDiff*) —the ratio of the time difference over the running time of the fastest. To give some indication of an average case, columns *RelSDev* show the relative standard deviation (i.e., coefficient of variation [37]) of the 16 runs.

Among default identical training runs, LeNet5 has the largest relative training time difference of 4,014.9% using the best-accuracy selection criterion. As discussed in RQ1, three runs fail to improve after the first epoch (3.9 seconds for the fastest), creating such a large time difference. However, since only three runs got stuck at the first epoch, the relative standard deviation is 55.1%.

The largest training time difference among default identical training runs is 6,316 seconds (1 hour and 40 minutes) for the WRN-28-10 network with the best-accuracy selection criterion (relative training time difference of 46.0%). Given how expensive DL training can be, 46.0% of training time difference could mean days or longer.

Among fixed-seed identical training runs, ResNet56 incurs the largest relative training time difference (145.3%) when using the best-accuracy selection criterion. This means that the deviation caused by random computation errors can lead to significantly different optimization paths, hence different convergence time.

Finding 4: Training time varies by as much as 145.3% (1 hour and 12 minutes) among fixed-seed identical training runs, while the training time difference is up to 4,014.8% with default identical training runs.

6 RESEARCHER AND PRACTITIONER SURVEY

We conduct a survey to understand if researchers and practitioners are (1) aware of the NI-factors and (2) if they correctly estimate how much impact NI-factors have on DL experiments.

6.1 Survey design and deployment

We conduct an anonymous online survey over a period of two weeks in February 2020. We target GitHub users who committed code to popular public DL projects under the topics `TensorFlow`, `PyTorch`, `CNTK`, `Theano`, `deeplearning`, and `neural-network`. We send 19,333 emails using Qualtrics services and receive 1,051 responses (5.4% response rate), 901 of which are valid. Many of the email addresses are from industry (364 from Microsoft, 833 from Google, and 80 from NVidia), and from academia (797 from U.S. universities).

We take the following steps to ensure our survey of 29 questions is valid and not biased. First, we conduct three rounds of in-person pilot studies with ten graduate students who have worked on DL projects and use their feedback to remove ambiguity and biases in our initial design. The pilot studies' participants do not participate in the actual survey.

Second, to ensure participant's understanding, we define important terms (e.g., deep learning, determinism, identical training runs, and fixed-seed identical training runs) in the context of our survey before the questions. For example, we give a clear definition of a deterministic DL system before survey questions: "*We define a system as deterministic if the system has identical accuracy or similar running time between multiple identical runs. In the case of a DL system, identical training runs have the same training dataset, data preprocessing method (e.g., same transformation operations), weight initializer (i.e., drawn from the same random distribution), network structure, loss function, optimizer, lower libraries, and hardware.*"

All questions and definitions are included in the GitHub repository whose link is provided in Section 1.

6.2 Survey results and findings

Participant Experience and Statistics: Of the 901 responses, 472 work in industry and 342 work in academia. Participants have an average work experience of 6.3 years and a maximum of 47 years. The average DL experience is 3.0 years. Over 68.6% learn AI formally (e.g., undergraduate and graduate school) and 32 are involved with 5 or more AI projects.

Awareness of NI-factors in DL systems: We ask Question 20: *"In your opinion, are DL systems deterministic?"* to gauge the awareness that participants have of the NI-factors (results in Figure 6).

Many respondents are unaware (31.9%) or uncertain (21.8%) of any variance of DL systems; and there is no correlation between DL experience and awareness of DL variance (**Finding 5**).

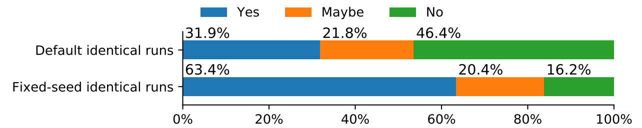

Figure 6: Distribution of responses to Question 20 (*Default identical runs*) and Question 26 (*Fixed-seed identical runs*)

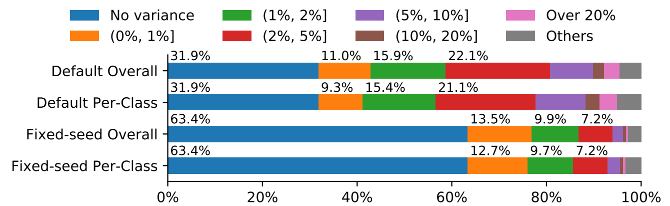

Figure 7: Estimation of *overall* and *per-class* accuracy difference across default and fixed-seed identical training runs

To measure the correlation between different factors, we use the Pearson correlation coefficient (r), a statistical indicator of linear correlation between two variables ($|r|$=0 means no correlation, while $|r|$=1 suggests a strong correlation). There is no correlation between awareness of DL variance and DL experience (r=0.03), DL educational background (r=0.04), or job position (r=0.02). These results suggest limited awareness of variance in DL systems regardless of experience and educational background.

Awareness of implementation-level NI-factors in DL systems: We design Questions 26: *"Do you expect fixed-seed identical DL training runs to be deterministic?"* to study how aware respondents are with implementation-level NI-factors (results in Figure 6).

Most (83.8%, 755 out of 901) of our surveyed researchers and practitioners are unaware of or unsure about implementation-level NI-factors (**Finding 6**).

There is no correlation between awareness of implementation-level NI-factors and DL experience (r=0.03), DL educational background (r=-0.01), or job position (r=0.06).

Estimate of accuracy difference: We ask participants who answered "Yes" or "Maybe" to Question 20 to answer Question 21: "*From your experience, in the worst case, by how much would you expect the final overall accuracy (e.g., in classification task) to vary in terms of absolute value between identical training runs?*". Also, after Question 26, we ask participants a similar Question 27 regarding fixed-seed identical training runs. Those who answer "Maybe" (i.e., unsure about DL system variance), we still ask them to estimate the magnitude of the variance. "Other" is an option to specify an explanation if no estimate is given.

Figure 7 shows participants' estimations of the overall and per-class accuracy differences across default identical training runs (*Default Overall* and *Default Per-Class*) and fixed-seed identical training runs (*Fixed-seed Overall* and *Fixed-seed Per-Class*). *No variance* indicates participants that are unaware of the nondeterminism of DL systems. Some participants choose "Others" and state that the accuracy difference depends on the task and network architecture.

Researchers and practitioners underestimate the magnitude of accuracy differences. Most (80.7%) responses estimate an accuracy difference across default identical training runs to be less than 5%.

Finding 2 indicates that the accuracy difference is up to 2.9% with implementation-level NI-factors alone. However, only 10.4% of respondents expect 2% or more accuracy difference across fixed-seed identical training runs, and they estimate similarly for per-class accuracy differences (**Finding 7**).

Estimate of training time difference: We ask participants to estimate how much the running time to convergence varies across default and fixed-seed identical training runs to see if their estimation matches the results from RQ3 (i.e., the convergence time differences are up to 4,014.8% among default identical training runs and up to 145.3% among fixed-seed identical training runs).

Most (77.7%) participants estimate the convergence time differences to be less than 10% across default identical training runs, and the majority of (84.5%) respondents estimate a similar 10% or less convergence time difference among fixed-seed identical training runs (**Finding 8**).

7 DL-TRAINING PAPER SURVEY

We conduct a literature survey to study the awareness of and the practice of handling DL variance in research papers.

Paper selection criteria and study approach: We extract research articles from the most recent top SE (ICSE'19, FSE'19, ASE'19), machine learning (NeurIPS/NIPS'19, ICLR'20, and ICML'19), computer vision (CVPR'19 and ICCV'19), and systems (SOSP'19, ASPLOS'19, MLSys'19) conferences. We focus on articles that were accepted for oral presentations (i.e., we exclude posters and spotlight articles), to keep the amount of manual examination realistic, considering that over 1,000 papers are accepted per year for conferences such as NeurIPS/NIPS. In total, 1,152 articles meet the above criterion. We split conferences into SE-systems-focused (SE and systems) and AI-focused (machine learning and computer vision) conferences to investigate whether AI papers are more likely to consider this variance in their evaluation.

Two authors independently check each of the 454 *randomly* sampled papers to see if it is relevant, i.e., papers that train DL models (89.3% of agreement). With 95% confidence, 28 out of 202 papers from SE-systems conferences (13.9±3%), versus 197 out of 252 papers from AI conferences (78.1±4%) are relevant.

Paper survey results: We present the survey result as follows.

Of the 225 relevant papers, only 19.5±3% use multiple identical training runs to evaluate their approaches (**Finding 9**): 25.0±4% for SE-systems conferences and 18.7±3% for AI conferences.

These results corroborate our online survey findings, indicating that researchers rarely consider (or have no clear solutions to measure) the impact of NI-factors. In addition, 33 papers in our sample use the same models we evaluated and report an accuracy improvement lower than the variance that we observed across multiple fixed-seed identical training runs (2.9%). Most (23) of these studies do not report validation using multiple identical training runs. Thus, the conclusions of these 23 studies are likely affected by the variance in multiple identical training runs. This is a conservative estimate as we use the implementation-level only variance (2.9%) instead of the overall variance (10.8%) as the criterion.

8 IMPLICATIONS, SUGGESTIONS, AND FUTURE WORK

Improving the stability of training implementations: Practitioners may need to control NI-factors or replay DL training deterministically to facilitate debugging, which are challenging tasks. As discussed in Section 2.4, algorithmic NI-factors are generally straightforward to control as they are introduced explicitly using pseudo-random number generators which can be seeded before each run. Practitioners may benefit from new methods (e.g., deterministic GPU [55]) to control implementation-level NI-factors, which are much harder to control because they are often the byproduct of optimization.

Research reproducibility and validity: Variance introduced by NI-factors reduces the reproducibility of DL-related experiments. Researchers should check if multiple identical training runs are needed to ensure the validity of their experiments and comparison.

It is nontrivial to determine the number of identical runs needed, which depends on the approaches and the baselines. One solution is iteratively performing more replication runs when comparing to a selected baseline. The replication process can stop when statistical tests (e.g., U-test) confirm the significance (e.g., p-value < 0.05) of the difference between the new approach and the baseline. If after a large number of replication runs (e.g., more than 30 runs[15]) the improvement is not statistically significant, then the variance might be large enough such that a statistically significant conclusion about the difference between the two techniques might not be possible. We are developing such a technique to help researchers and practitioners with this process, to reduce the manual effort to conduct valid experiments and replicate experiments.

Approaches to improve reproducibility suggested by the SE community [44, 75, 107] need to also consider training variance. New approaches such as efficient checkpointing may be desirable.

Transparency is important in making sure that research is reproducible and valid. Recently, the DL research community promoted sharing artifacts and results transparently[12, 34]. Since DL systems are nondeterministic, it is important to share the data from the replication runs as well. One solution is maintaining a centralized trusted database that stores these replication runs and provides authors of new approaches with baseline results that they can directly compare to without rerunning the baseline approaches. We are developing a tool that helps users to measure the variance of their approaches and facilitates the comparison across approaches. Users can upload their replication packages and results to a database, that is provided by our tool, to be curated for comparison.

Producing better models: When a DL model is the contribution (e.g., defect prediction [70] or program repair [69]), practitioners could leverage variance to obtain a more accurate model.

Less expensive training and variance estimation: Since DL training is expensive, an important research direction could be less-expensive variance estimation and training approaches such as software support for incremental training [47].

9 THREATS TO VALIDITY

External validity: Observed results might be different for other networks. We use 6 very popular networks with diverse complexity

(from 7,206 to 36,536,884 parameters) to mitigate this issue. We encourage others (and ourselves) to replicate and enrich our studies with different DL networks, DL training approaches, and DL libraries to build an empirical body of knowledge about variance in DL systems. We are building a replication tool to help the research community to share and replicate research experiments and results.

New algorithmic NI-factors can be added (e.g., new nondeterministic layers) so our list of algorithmic NI-factors could become incomplete in the future. For fixed-seed identical training runs, we ensure that all algorithmic NI-factors of the DL networks we evaluate are disabled.

Internal validity: Our implementation or the libraries we used might have bugs. This should be alleviated by that (1) all our code is reviewed by most authors, (2) our results show that variance exists for all versions of all libraries we evaluate, thus unlikely that they are caused by library bugs, (3) we focus on official releases of DL libraries, so our runs should still be representative of real DL usage, and (4) we analyze all results especially outliers to ensure there were no implementation bugs.

Multiple identical training runs might produce different models (i.e., models with different weights) with identical accuracy and running time. Our study focuses on accuracy and time variance, since these are the end results that users care about.

Construct validity: We ensure to target relevant participants in our survey by specifically inviting code contributors to DL projects and asking them to confirm that they work with DL. Respondents might not have wanted to show any perceived ignorance which could have biased their responses. However, the strength of the responses, 83.8% being unaware or uncertain about implementation-level nondeterminism in DL systems helps alleviate this issue.

10 RELATED WORK

Our paper is unique because we study and quantify DL variance caused by NI-factors and conduct surveys to check its awareness.

Variance study of reinforcement learning (RL): Closest to our work is the study [80] that measures the impact of some implementation-level NI-factors on RL experiments. We study the general DL variance, while they focus on RL variance only. In addition, we measure the awareness by conducting a survey and a literature review. Furthermore, while they focus on one network for one task (i.e., playing the Atari game Breakout) and one version of one core library (PyTorch), we study the impact of NI-factors using 6 networks trained on 3 datasets and multiple versions of three core libraries. Papers [29, 53] that investigates the impact of random seeds on RL are different from ours, since they only consider the impact of random seeds (i.e., algorithmic NI-factors), while we also study the variance caused by implementation-level NI-factors.

Awareness of the impact of nondeterminism: A recent literature survey [81] on 30 papers on the topic of text mining confirms the results of our literature survey. None of the 30 investigated papers report using different random seeds. Our literature survey investigates DL training in general (not just text mining papers) and examines 225 papers. Furthermore, our overall contribution is different since we also quantify the differences in training accuracy and time across identical training runs. Another paper [79] states that small changes in the experimental setup can generate measurement bias. It focuses on standard CPU computation benchmark [31] and does not study the nonderterminism of DL systems.

Anecdotal evidence of NI-factors: Some studies [59, 76, 77, 108] quantify the variance in their results caused by NI-factors. However, these are only anecdotal evidence and none attempts to systematically study the variance introduced by algorithmic and implementation-level NI-factors. In addition, our surveys show that awareness is still very low in the research community.

Nondeterminism in stochastic gradient descent (SGD): Much work investigates variance caused by SGD [26, 33, 35, 58, 68]. While these papers quantify nondeterminism caused by SGD, they ignore all other sources of nondeterminism described in Section 2.

Impact of weight initialization: Prior work [57, 88, 101, 104] measure the impact of different initial weights on models' training time. While such an issue is known, implementation-level NI-factors have not been studied and our surveys show that they are often not considered when evaluating DL systems.

Controlling implementation-level nondeterminism: Jooybar et al. [55] propose a hardware mechanism to introduce determinism in GPU-based algorithms. In our work, we do not focus on the hardware itself and measure the variance caused by NI-factors using popular GPUs without special hardware modifications.

DL System Benchmarking: Much work focuses on benchmarking DL systems. However, their target is finding the best networks [30, 86, 121], hardware [92, 121], hyper-parameters [72], and framework [48, 60, 72, 93, 95, 121]. Such approaches do not consider the impact of NI-factors on multiple identical training runs.

11 CONCLUSIONS

This work studies the variance introduced by nondeterminism in DL systems and the awareness of this variance among researchers and practitioners. We perform experiments on three datasets with six popular networks and find up to 10.8% accuracy differences among identical training runs when excluding weak models. Even with fixed seeds, the accuracy differences are as large as 2.9%. Our surveys show that 83.8% of surveyed researchers and practitioners are unaware of or unsure about implementation-level variance and only 19.5±3% of papers in recent relevant top conferences use multiple identical training runs to quantify the variance of their DL approaches. Thus, we raise the awareness of DL variance, for better research validity and reproducibility, more accurate models, deterministic debugging, new research on training stability, efficient training, and fast variance estimation.

ACKNOWLEDGMENTS

The authors thank Eric Horvitz, Ben Zorn, and Madan Musuvathi for their insightful comments and discussions. They thank Yitong Li for examining some research papers for the literature survey. The research is partially supported by NSF 2006688 and a Facebook research award. Any opinions, findings, and conclusions in this paper are those of the authors only and do not necessarily reflect the views of the sponsors.

REFERENCES

[1] 2019. *ASE '19: Proceedings of the 34th ACM/IEEE International Conference on Automated Software Engineering.*
[2] 2019. *ASPLOS '19: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems.*
[3] 2019. *CVPR'19: Proceedings of The IEEE Conference on Computer Vision and Pattern Recognition.*
[4] 2019. *ESEC/FSE '19: Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.*
[5] 2019. *ICCV'19: Proceedings of The IEEE International Conference on Computer Vision.* IEEE Press.
[6] 2019. *ICML'19: Proceedings of The International Conference on Machine Learning.*
[7] 2019. *ICSE '19: Proceedings of the 41st International Conference on Software Engineering.* IEEE Press.
[8] 2019. *MLSys'19: Proceedings of Machine Learning and Systems.*
[9] 2019. *NIPS'19: Proceedings of the 33rd Conference on Neural Information Processing Systems.*
[10] 2019. *SOSP '19: Proceedings of the 27th ACM Symposium on Operating Systems Principles.*
[11] 2020. *ICLR'20: Proceedings of The International Conference on Learning Representations.*
[12] 2020. *Reproducibility Challenge @ NeurIPS 2019.* https://reproducibility-challenge.github.io/neurips2019/
[13] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. TensorFlow: a System for Large-Scale Machine Learning.. In *OSDI*.
[14] Charu C Aggarwal. 2018. Neural networks and deep learning. In *Springer*.
[15] Andrea Arcuri and Lionel Briand. 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE*.
[16] James Bergstra, Frédéric Bastien, Olivier Breuleux, Pascal Lamblin, Razvan Pascanu, Olivier Delalleau, Guillaume Desjardins, David Warde-Farley, Ian Goodfellow, Arnaud Bergeron, et al. 2011. Theano: Deep learning on GPUs with Python. In *NIPS*.
[17] James Bergstra and Yoshua Bengio. 2012. Random search for hyper-parameter optimization. In *JMLR*.
[18] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. 2019. AutoFocus: Interpreting Attention-Based Neural Networks by Code Perturbation. In *ASE*.
[19] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. 2015. Deepdriving: Learning Affordance for Direct Perception in Autonomous Driving. In *ICCV*.
[20] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. Unblind Your Apps: Predicting Natural-Language Labels for Mobile GUI Components by Deep Learning. In *ICSE*.
[21] Jinyin Chen, Keke Hu, Yue Yu, Zhuangzhi Chen, Qi Xuan, Yi Liu, and Vladimir Filkov. 2020. Software Visualization and Deep Transfer Learning for Effective Software Defect Prediction. In *ICSE*.
[22] Liang-Chieh Chen, Maxwell Collins, Yukun Zhu, George Papandreou, Barret Zoph, Florian Schroff, Hartwig Adam, and Jon Shlens. 2018. Searching for efficient multi-scale architectures for dense image prediction. In *NIPS*.
[23] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning.
[24] François Chollet. 2017. Xception: Deep learning with depthwise separable convolutions. In *CVPR*.
[25] François Chollet et al. 2015. Keras. https://keras.io.
[26] Anna Choromanska, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, and Yann LeCun. 2015. The loss surfaces of multilayer networks. In *AISTATS*.
[27] Anna Choromanska, Yann LeCun, and Gérard Ben Arous. 2015. Open Problem: The landscape of the loss surfaces of multilayer networks. In *COLR*.
[28] J. Cohen. 1988. *Statistical Power Analysis for the Behavioral Sciences.*
[29] Cédric Colas, Olivier Sigaud, and Pierre-Yves Oudeyer. 2018. How Many Random Seeds? Statistical Power Analysis in Deep Reinforcement Learning Experiments. In *CoRR*.
[30] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. 2017. Dawnbench: An end-to-end deep learning benchmark and competition. In *Training*.
[31] Standard Performance Evaluation Corporation. 2006. SPEC CPU2006 Benchmarks. http://www.spec.org/cpu2006/.
[32] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. 2013. Parrot: a Practical Runtime for Deterministic, Stable, and Reliable Threads. In *SOSP*.
[33] Yann N Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. 2014. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *NIPS*.
[34] Jesse Dodge, Suchin Gururangan, Dallas Card, Roy Schwartz, and Noah A. Smith. 2019. Show Your Work: Improved Reporting of Experimental Results. In *EMNLP-IJCNLP*.
[35] Felix Draxler, Kambis Veschgini, Manfred Salmhofer, and Fred Hamprecht. 2018. Essentially No Barriers in Neural Network Energy Landscape. In *ICML*.
[36] Simon S Du, Jason D Lee, Yuandong Tian, Barnabas Poczos, and Aarti Singh. 2017. Gradient descent learns one-hidden-layer cnn: Don't be afraid of spurious local minima. In *arXiv preprint arXiv:1712.00779*.
[37] Brian Everitt. 2002. *The Cambridge dictionary of statistics.*
[38] Hao-Shu Fang, Guansong Lu, Xiaolin Fang, Jianwen Xie, Yu-Wing Tai, and Cewu Lu. 2018. Weakly and Semi Supervised Human Body Part Parsing via Pose-Guided Knowledge Transfer. In *CVPR*.
[39] Xiang Gao, Ripon Saha, Mukul Prasad, and Abhik Roychoudhury. 2020. Fuzz Testing based Data Augmentation to Improve Robustness of Deep Neural Networks. In *ICSE*.
[40] Rong Ge, Furong Huang, Chi Jin, and Yang Yuan. 2015. Escaping from saddle points—online stochastic gradient for tensor decomposition. In *COLT*.
[41] Simos Gerasimou, Hasan Ferit-Eniser, Alper Sen, and Alper Çakan. 2020. Importance-Driven Deep Learning System Testing. In *ICSE*.
[42] Xavier Glorot and Yoshua Bengio. 2010. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*.
[43] David Goldberg. 1991. What Every Computer Scientist Should Know about Floating-Point Arithmetic. *ACM Comput. Surv.* (1991).
[44] Jesús M González-Barahona and Gregorio Robles. 2012. On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. In *ESE*.
[45] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning.* MIT press.
[46] Divya Gopinath, Hayes Converse, Corina S. Păsăreanu, and Ankur Taly. 2019. Property Inference for Deep Neural Networks. In *ASE*.
[47] Hui Guan, Xipeng Shen, and Seung-Hwan Lim. 2019. Wootz: A Compiler-Based Framework for Fast CNN Pruning via Composability. In *PLDI*.
[48] Qianyu Guo, Sen Chen, Xiaofei Xie, Lei Ma, Qiang Hu, Hongtao Liu, Yang Liu, Jianjun Zhao, and Xiaohong Li. 2019. An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms. In *ASE*.
[49] Jeff Haochen and Suvrit Sra. 2019. Random Shuffling Beats SGD after Finite Epochs. In *ICML*.
[50] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*.
[51] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *CVPR*.
[52] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. 2018. Deep Learning Type Inference. In *ESEC/FSE*.
[53] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. 2018. Deep Reinforcement Learning that Matters. In *AAAI*.
[54] Ilija Ilievski, Taimoor Akhtar, Jiashi Feng, and Christine Annette Shoemaker. 2017. Efficient hyperparameter optimization for deep learning algorithms using deterministic rbf surrogates. In *AAAI*.
[55] Hadi Jooybar, Wilson WL Fung, Mike O'Connor, Joseph Devietti, and Tor M Aamodt. 2013. GPUDet: a deterministic GPU architecture. In *ASPLOS*.
[56] Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding Deep Learning System Testing Using Surprise Adequacy. In *ICSE*.
[57] YK Kim and JB Ra. 1991. Weight value initialization for improving training speed in the backpropagation network. In *IJCNN*.
[58] Bobby Kleinberg, Yuanzhi Li, and Yang Yuan. 2018. An Alternative View: When Does SGD Escape Local Minima?. In *ICML*.
[59] Yuriy Kochura, Sergii Stirenko, Oleg Alienin, Michail Novotarskiy, and Yuri Gordienko. 2018. Performance Analysis of Open Source Machine Learning Frameworks for Various Parameters in Single-Threaded and Multi-threaded Modes. In *CSIT*.
[60] Vassili Kovalev, Alexander Kalinovsky, and Sergey Kovalev. 2016. Deep learning with theano, torch, caffe, tensorflow, and deeplearning4j: Which one is the best in speed and accuracy?
[61] Alex Krizhevsky. 2009. Learning multiple layers of features from tiny images.
[62] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. Imagenet classification with deep convolutional neural networks. In *NIPS*.
[63] Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. DIRE: A Neural Approach to Decompiled Identifier Naming. In *ASE*.
[64] Alex Lamb, Jonathan Binas, Anirudh Goyal, Sandeep Subramanian, Ioannis Mitliagkas, Denis Kazakov, Yoshua Bengio, and Michael C. Mozer. 2019. State-Reification Networks: Improving Generalization by Modeling the Distribution of Hidden Representations. In *ICML*.
[65] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. 1998. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*.
[66] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. 2012. Efficient backprop. In *Neural networks: Tricks of the trade.*
[67] Jason D Lee, Max Simchowitz, Michael I Jordan, and Benjamin Recht. 2016. Gradient descent converges to minimizers. In *arXiv preprint arXiv:1602.04915*.

[68] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. 2018. Visualizing the Loss Landscape of Neural Nets. In *NIPS*.
[69] Yi Li, Wang Shaohua, and Tien N. Nguyen. 2020. DLFix: Context-based Code Transformation Learning for Automated Program Repair. In *ICSE*.
[70] Yi Li, Shaohua Wang, Tien N Nguyen, and Son Van Nguyen. 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. In *PACMPL*.
[71] Zenan Li, Xiaoxing Ma, Chang Xu, Chun Cao, Jingwei Xu, and Jian Lü. 2019. Boosting Operational DNN Testing Efficiency through Conditioning. In *ESEC/FSE*.
[72] Ling Liu, Yanzhao Wu, Wenqi Wei, Wenqi Cao, Semih Sahin, and Qi Zhang. 2018. Benchmarking deep learning frameworks: Design considerations, metrics and beyond. In *ICDCS*.
[73] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. 2018. DeepGauge: Multi-Granularity Testing Criteria for Deep Learning Systems. In *ASE*.
[74] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. 2018. MODE: Automated Neural Network Model Debugging via State Differential Analysis and Input Selection. In *ESEC/FSE*.
[75] Zaheed Mahmood, David Bowes, Tracy Hall, Peter CR Lane, and Jean Petrić. 2018. Reproducibility and replicability of software defect prediction studies. *ISE* (2018).
[76] Andrii Maksai and Pascal Fua. 2019. Eliminating exposure bias and metric mismatch in multiple object tracking. In *CVPR*.
[77] Luke Metz, Niru Maheswaranathan, Jeremy Nixon, C Daniel Freeman, and Jascha Sohl-Dickstein. 2019. Understanding and correcting pathologies in the training of learned optimizers. In *ICML*.
[78] Hrushikesh N. Mhaskar, Sergei V. Pereverzyev, and Maria D. van der Walt. 2017. A Deep Learning Approach to Diabetic Blood Glucose Prediction. In *Front. Appl. Math. Stat.*
[79] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter Sweeney. 2009. Producing wrong data without doing anything obviously wrong!. In *ACM SIGARCH Computer Architecture News*.
[80] Prabhat Nagarajan, Garrett Warnell, and Peter Stone. 2019. Deterministic Implementations for Reproducibility in Deep Reinforcement Learning. In *AAAI Workshop on Reproducible AI*.
[81] Babatunde K Olorisade, Pearl Brereton, and Peter Andras. 2017. Reproducibility in machine Learning-Based studies: An example of text mining. In *Reproducibility in Machine Learning Workshop, ICML*.
[82] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NIPS*.
[83] Brandon Paulsen, Jingbo Wang, and Chao Wang. 2020. ReluDiff: Differential Verification of Deep Neural Networks. In *ICSE*.
[84] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2019. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Commun. ACM*.
[85] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. 2019. CRADLE: Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries. In *ICSE*.
[86] Sanjay Purushotham, Chuizheng Meng, Zhengping Che, and Yan Liu. 2018. Benchmarking deep learning models on large healthcare datasets. In *J. Biomed. Inform.*
[87] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. 2018. Bug Synthesis: Challenging Bug-Finding Tools with Deep Faults. In *ESEC/FSE*.
[88] Tim Salimans and Durk P Kingma. 2016. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *NIPS*.
[89] Shlomo Sawilowsky, Jack Sawilowsky, and Robert J. Grissom. 2011. *Effect Size*.
[90] Andrew M. Saxe, James L. Mcclelland, and Surya Ganguli. 2014. Exact solutions to the nonlinear dynamics of learning in deep linear neural network. In *ICLR*.
[91] Frank Seide and Amit Agarwal. 2016. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *KDD*.
[92] Shayan Shams, Richard Platania, Kisung Lee, and Seung-Jong Park. 2017. Evaluation of deep learning frameworks over different HPC architectures. In *ICDCS*.
[93] Ali Shatnawi, Ghadeer Al-Bdour, Raffi Al-Qurran, and Mahmoud Al-Ayyoub. 2018. A comparative study of open source deep learning frameworks. In *ICICS*.
[94] Zhiqiang Shen, Zhuang Liu, Jianguo Li, Yu-Gang Jiang, Yurong Chen, and Xiangyang Xue. 2017. Dsod: Learning deeply supervised object detectors from scratch. In *ICCV*.
[95] Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. 2016. Benchmarking state-of-the-art deep learning software tools. In *CCBD*.
[96] Connor Shorten and Taghi M. Khoshgoftaar. 2019. A survey on Image Data Augmentation for Deep Learning. In *Journal of Big Data*.
[97] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. In *arXiv preprint arXiv:1409.1556*.
[98] Xinhang Song, Luis Herranz, and Shuqiang Jiang. 2017. Depth cnns for rgb-d scene recognition: Learning from scratch better than transferring from rgb-cnns. In *AAAI*.
[99] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. In *JMLR*.
[100] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. 2018. Concolic Testing for Deep Neural Networks. In *ASE*.
[101] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. 2013. On the importance of initialization and momentum in deep learning. In *ICML*.
[102] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *CVPR*.
[103] Nima Tajbakhsh, Jae Y Shin, Suryakanth R Gurudu, R Todd Hurst, Christopher B Kendall, Michael B Gotway, and Jianming Liang. 2016. Convolutional neural networks for medical image analysis: Full training or fine tuning?. In *IEEE Trans. Med. Imag.*
[104] Kok Keong Teo, Lipo Wang, and Zhiping Lin. 2001. Wavelet packet multi-layer perceptron for chaotic time series prediction: effects of weight initialization. In *ICCS*.
[105] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars. In *ICSE*.
[106] Yuchi Tian, Ziyuan Zhong, Vicente Ordonez, Gail Kaiser, and Baishakhi Ray. 2020. Testing DNN Image Classifier for Confusion & Bias Errors. In *ICSE*.
[107] Fabian Trautsch, Steffen Herbold, Philip Makedonski, and Jens Grabowski. 2018. Addressing problems with replicability and validity of repository mining studies through a smart data platform. In *ESE*.
[108] Nam Vo, Lu Jiang, Chen Sun, Kevin Murphy, Li-Jia Li, Li Fei-Fei, and James Hays. 2019. Composing text and image for image retrieval-an empirical odyssey. In *CVPR*.
[109] Huiyan Wang, Jingweiu Xu, Chang Xu, Xiaoxing Ma, and Jian Lu. 2020. DISSECTOR: Input Validation for Deep Learning Applications by Crossing-layer Dissection. In *ICSE*.
[110] Jingyi Wang, Guoliang Dong, Jun Sun, Xinyu Wang, and Peixin Zhang. 2019. Adversarial Sample Detection for Deep Neural Network through Model Mutation Testing. In *ICSE*.
[111] Steven R Young, Derek C Rose, Thomas P Karnowski, Seung-Hwan Lim, and Robert M Patton. 2015. Optimizing deep learning hyper-parameters through an evolutionary algorithm. In *MLHPC*.
[112] Sergey Zagoruyko and Nikos Komodakis. 2016. Wide Residual Networks. In *BMVC*.
[113] Hao Zhang and W. K. Chan. 2019. Apricot: A Weight-Adaptation Approach to Fixing Deep Learning Models. In *ASE*.
[114] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based Neural Source Code Summarization. In *ICSE*.
[115] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 2018. DeepRoad: GAN-Based Metamorphic Testing and Input Validation Framework for Autonomous Driving Systems. In *ASE*.
[116] Xiyue Zhang, Xiaofei Xie, Lei Ma, Xiaoning Du, Qiang Hu, Yang Liu, Jianjun Zhao, and Meng Sun. 2020. Towards Characterizing Adversarial Defects of Deep Learning Software from the Lens of Uncertainty. In *ICSE*.
[117] Gang Zhao and Jeff Huang. 2018. DeepSim: Deep Learning Code Functional Similarity. In *ESEC/FSE*.
[118] Hengshuang Zhao, Jianping Shi, Xiaojuan Qi, Xiaogang Wang, and Jiaya Jia. 2017. Pyramid scene parsing network. In *CVPR*.
[119] Yan Zheng, Xiaofei Xie, Ting Su, Lei Ma, Jianye Hao, Zhaopeng Meng, Yang Liu, Ruimin Shen, Yingfeng Chen, and Changjie Fan. 2019. Wuji: Automatic Online Combat Game Testing Using Evolutionary Deep Reinforcement Learning. In *ASE*.
[120] Husheng Zhou, Wei Li, Zelun Kong, Junfeng Guo, Yuqun Zhang, Lingming Zhang, Bei Yu, and Cong Liu. 2020. DeepBillboard: Systematic Physical-World Testing of Autonomous Driving Systems. In *ICSE*.
[121] H. Zhu, M. Akrout, B. Zheng, A. Pelegris, A. Jayarajan, A. Phanishayee, B. Schroeder, and G. Pekhimenko. 2018. Benchmarking and Analyzing Deep Neural Network Training. In *IISWC*.
[122] Rui Zhu, Shifeng Zhang, Xiaobo Wang, Longyin Wen, Hailin Shi, Liefeng Bo, and Tao Mei. 2019. ScratchDet: Training single-shot object detectors from scratch. In *CVPR*.