Towards Training Reproducible Deep Learning Models

Boyuan Chen
Centre for Software Excellence,
Huawei Canada
Kingston, Canada
boyuan.chen1@huawei.com

Mingzhi Wen
Huawei Technologies
Shenzhen, China
wenmingzhi@huawei.com

Yong Shi
Huawei Technologies
Shenzhen, China
young.shi@huawei.com

Dayi Lin
Centre for Software Excellence,
Huawei Canada
Kingston, Canada
dayi.lin@huawei.com

Gopi Krishnan Rajbahadur
Centre for Software Excellence,
Huawei Canada
Kingston, Canada
gopi.krishnan.rajbahadur1@huawei.com

Zhen Ming (Jack) Jiang
York University
Toronto, Canada
zmjiang@eecs.yorku.ca

ABSTRACT

Reproducibility is an increasing concern in Artificial Intelligence (AI), particularly in the area of Deep Learning (DL). Being able to reproduce DL models is crucial for AI-based systems, as it is closely tied to various tasks like training, testing, debugging, and auditing. However, DL models are challenging to be reproduced due to issues like randomness in the software (e.g., DL algorithms) and non-determinism in the hardware (e.g., GPU). There are various practices to mitigate some of the aforementioned issues. However, many of them are either too intrusive or can only work for a specific usage context. In this paper, we propose a systematic approach to training reproducible DL models. Our approach includes three main parts: (1) a set of general criteria to thoroughly evaluate the reproducibility of DL models for two different domains, (2) a unified framework which leverages a record-and-replay technique to mitigate software-related randomness and a profile-and-patch technique to control hardware-related non-determinism, and (3) a reproducibility guideline which explains the rationales and the mitigation strategies on conducting a reproducible training process for DL models. Case study results show our approach can successfully reproduce six open source and one commercial DL models.

CCS CONCEPTS

• **Software and its engineering → Empirical software validation**.

KEYWORDS

Artificial Intelligence, Deep Learning, Software Engineering, Reproducibility

ACM Reference Format:
Boyuan Chen, Mingzhi Wen, Yong Shi, Dayi Lin, Gopi Krishnan Rajbahadur, and Zhen Ming (Jack) Jiang. 2022. Towards Training Reproducible Deep Learning Models. In *44th International Conference on Software Engineering (ICSE '22), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3510003.3510163

ICSE '22, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9221-1/22/05…$15.00
https://doi.org/10.1145/3510003.3510163

1 INTRODUCTION

In recent years, Artificial Intelligence (AI) has been advancing rapidly both in research and practice. A recent report by McKinsey estimates that AI-based applications have the potential market values ranging from $3.5 and $5.8 trillion annually [11]. Many of these applications, which can perform complex tasks such as autonomous driving [34], speech recognition [24], and healthcare [29], are enabled by various Deep Learning (DL) models [46]. Unlike traditional software systems, which are programmed based on deterministic rules (e.g., if/else), the DL models within AI-based systems are constructed in a stochastic way due to the underlying DL algorithms, whose behavior may not be reproducible and trustworthy [26, 54]. Ensuring the reproducibility of DL models is vital for not only many product development related tasks such as training [50], testing [18], debugging [56] and legal compliance [2], but also facilitating scientific movements like open science [66, 67].

One of the important steps towards reproducible AI-based systems is to ensure the reproducibility of the DL models during the training process. A DL model is *reproducible*, if under the same training setup (e.g., the same training code, the same environment, and the same training dataset), the resulting trained DL model yields the same results under the same evaluation criteria (e.g., the same evaluation metrics on the same testing dataset) [56, 57]. Unfortunately, recent studies show that AI faces reproducibility crisis [37, 41], especially for DL models [32, 44, 48, 50, 56, 58, 63, 65]. In general, there are three main challenges associated with this:

- **Randomness in the software [61]**: Randomness is essential in DL model training like batch ordering, data shuffling, and weight initialization for constructing robust and high-performing DL models [14, 56]. However, randomness prevents the DL models from being reproduced. To achieve reproducibility in the training process, the current approach is to set predefined random seeds before the training process. Although this approach is effective in controlling the randomness, it has three drawbacks: (1) it might cause the training process to converge to local optimums and not able to explore other optimization opportunities; (2) it is non-trivial to select the appropriate seeds as there are no existing

techniques for tuning random seeds during the hyperparameter tuning process; (3) non-trivial manual efforts are needed to locate randomness introducing functions and instrument them with seeds for the imported libraries and their dependencies.

- **Non-determinism in the hardware [6]**: Training DL models requires intensive computing resources. For example, many matrix operations occur in the backward propagation, which consists of a huge amount of floating point operations. As GPUs have way more numbers of cores than CPUs, GPUs are often used for running DL training processes due to their ability to process multiple operations in parallel. However, executing floating point calculation in parallel becomes a source of non-determinism since the results of floating-point operations are sensitive to computation orders due to rounding errors [33, 56]. In addition, GPU specific libraries (e.g., CUDA [4] and cuDNN [27]) by default auto-select the optimal primitive operations based on comparing different algorithms of operations during runtime (i.e., the auto-tuning feature). However, the comparison results might be non-deterministic due to issues like floating point computation mentioned above [14, 56]. These sources of non-determinism from hardware need to be controlled in order to construct reproducible DL models. Case-by-case solutions have been proposed to tackle specific issues. For example, both Pytorch [55] and TensorFlow [22] provide configurations on disabling the auto-tuning feature. Unfortunately, none of these techniques have been empirically validated in literature. Furthermore, there is still a lack of a general technique which can work across different software frameworks.
- **Lack of systematic guidelines [56]**: Various checklists and documentation frameworks [19, 30, 53] have been proposed on asset management to support DL reproducibility. There are generally four types of assets to manage in machine learning (ML) in order to achieve model reproducibility: resources (e.g., dataset and environment), software (e.g., source code), metadata (e.g., dependencies), and execution data (e.g., execution results) [43]. However, prior work [23, 25, 43] shows these assets should not be managed with the same toolsets (e.g., Git) used for source code [23]. Hence, new version management tools (e.g., DVC [12] and MLflow [1]) are specifically designed for managing ML assets. However, even by adopting the techniques and suggestions mentioned above, DL models cannot be fully reproduced [56] due to problems mentioned in the above two challenges. A systematic guideline is needed for both researchers and practitioners in order to construct reproducible DL models.

To address the above challenges, in this paper, we have proposed a systematic approach towards training reproducible DL models. Our approach includes a set of rigorously defined evaluation criteria, a record-and-replay-based technique for mitigating randomness in software, and a profile-and-patch-based technique for mitigating non-determinism from hardware. We have also provided a systematic guideline for DL model reproducibility based on our experience on applying our approach across different DL models. Case studies on six popular open source and one commercial DL models show that our approach can successfully reproduce the studied DL models (i.e., the trained models achieve the exact same results under the evaluation criteria). To facilitate reproducibility of our study, we provide a replication package [21], which consists of the implementation of open source DL models, our tool, and the experimental results. In summary, our paper makes the following contributions:

- Although there are previous research work which aimed at reproducible DL models (e.g., [50, 56]), to the authors' knowledge, our approach is the first systematic approach which can achieve reproducible DL models during the training process. Case study results show that all the studied DL models can be successfully reproduced by leveraging our approach.
- Compared to existing practices for controlling randomness in the software (a.k.a., presetting random seeds [5, 56]), our record-and-replay-based technique is non-intrusive and incurs minimal disruption on the existing DL development process.
- Compared to the previous approach on verifying model reproducibility [56], our proposed evaluation criteria has two advantages: (1) it is more general, as it covers multiple domains (Classification and Regression tasks), and (2) it is more rigorous, as it evaluates multiple criteria, which includes not only the evaluation results on the testing dataset, but also the consistency of the training process.

Paper Organization. Section 2 provides background information associated with DL model reproducibility. Section 3 describes the details of our systematic approach to training reproducible DL models. Section 4 presents the evaluation of our approach. Section 5 discusses the experiences and lessons learned from applying our approach. Section 6 presents our guideline. Section 7 describes the threats to validity of our study and Section 8 concludes our paper.

2 BACKGROUND AND RELATED WORK

In this section, we describe the background and the related work associated with constructing reproducible DL models.

2.1 The Need for Reproducible DL models

Several terms have been used in existing literature to discuss the concepts of reproducibility in research and practice [5, 14, 17, 19, 40, 50, 56, 57]. We follow the similar definitions used in [56, 57], where a particular piece of work is considered as *reproducible*, if the same data, same code, and same analysis lead to the same results or conclusions. On the other hand, replicable research refers to that different data (from the same distribution of the original data) combined with same code and analysis result in similar results. In this paper, we focus on the reproducibility of DL models during the training process. The same training process requires the exact same setup, which includes the same source code (including training scripts and configurations), the same training and testing data, and the same environment.

Training reproducible DL models is essential in both research and practice. On one hand, it facilitates the open science movement by enabling researchers to easily reproduce the same results. Open science movement [66, 67] promotes sharing research assets in a transparent way, so that the quality of research manuscripts can be checked and improved. On the other hand, many companies are also integrating the cutting-edge DL research into their products. Having reproducible DL models would greatly benefit the product development process. For example, if a DL model is reproducible, the testing and debugging processes would be much easier

as the problematic behavior can be consistently reproduced [18]. In addition, many DL-based applications now require regulatory compliance and are subject to rigorous auditing processes [13]. It is vital that the behavior of the DL models constructed during the auditing process closely matches with that of the released version [2].

2.2 Current State of Reproducible DL Models

2.2.1 Reproducibility crisis. In 2018, Huston [41] mentioned it is very difficult to verify many claims published in research papers due to the lack of code and the sensitivity of training conditions (a.k.a., the reproducibility crisis in AI). Similarly, Gundersen and Kjensmo [37] surveyed 400 research papers from IJCAI and AAAI and found that only 6% of papers provided experiment code. Similarly, in software engineering research, Liu et al. [50] surveyed 93 SE research papers which leveraged DL techniques and only 10.8% of research discussed reproducibility related issues. Isdahl and Gundersen [44] surveyed 13 state of the art ML platforms and found the popular ML platforms provided by well-known companies have poor support for reproducibility, especially in terms of data. Instead of verifying and reporting the reproducibility of different research work, we focus on proposing a new approach which can construct reproducible DL models.

2.2.2 Efforts towards improving reproducibility. Various efforts have been devoted to improve the reproducibility of DL models:

(E1) Controlling Randomness from software. Liu et al. [50] found that the randomness in software could impact the reproducibility of DL models and only a few studies (e.g., [28, 36, 40]) reported using preset seeds to control the randomness. Similarly, Pham et al. [56] found that by controlling randomness in software, the performance variances in trained DL models decrease significantly. Sugimura and Hartl [64] mentioned that a random seed needs to be set as a hyperparameter prior to training for reproducibility. Determined.AI [5], a company that focuses on providing services for DL model training, also supports setting seeds for reproducing DL experiments. However, none of the prior studies discussed how to properly set seeds or the performance impact of different set of seeds. Compared to presetting random seeds, our record-and-replay-based technique to control the randomness in the software is non-intrusive and incurs minimal disruption on the existing DL development.

(E2) Mitigating non-determinism in the hardware. Pham et al. [56] discussed using environment variables to mitigate non-determinism caused by floating point rounding error and parallel computation. Jooybar et al [45] designed a new GPU architecture for deterministic operations. However, there has been a lack of thorough assessment of the proposed solutions. In addition, our approach mainly focuses on mitigating non-determinism on common hardware instead of proposing new hardware design.

(E3) Existing guidelines and best practices. To address the reproducibility crisis mentioned above, major AI conferences such as NeurIPS, ICML, and AAAI hold reproducibility workshops and advocate researchers to independently verify the results of published research papers as reproducibility challenges. Various documentation frameworks for DL models [30, 53] or checklists [19] have been proposed recently. These documentations specify the required information and artifacts (e.g., datasets, code, and experimental results) that are needed to reproduce DL models. Similarly, Ghanta et al [32] investigated AI reproducibility in production, where they mentioned many factors need to be considered to achieve reproducibility such as pipeline configuration and input data. Tatman et al. [65] indicated that high reproducibility is achieved by managing code, data, and environment. They suggest in order to reach the highest reproducibility, the runtime environment should be provided as hosting services, containers, or VMs. Sugimura and Hartl [64] built an end-to-end reproducible ML pipeline which focuses on data, feature, model, and software environment provenance. In our study, we mainly focus on model training with the assumption that the code, data, and environment should be consistent across repeated training processes. However, even with consistent assets mentioned above, it is still challenging to achieve reproducibility due to the lack of tool support and neglection of certain sources of non-determinism [32, 44, 48, 56, 58, 63, 65].

2.3 Industrial Assessment

Huawei is a large IT company, which provides many products and services relying on AI-based components. To ensure the quality, trustworthiness, transparency, and traceability of the products, practitioners in Huawei have been investigating approaches to training reproducible DL models. We worked closely with 20 practitioners, who are either software developers or ML scientists with Ph.D degrees. Their tasks are to prototype DL models and/or productionalize DL models. We first presented the current research and practices on verifying and achieving reproducibility in DL models. Then we conducted a two hour long semi-formal interview with these practitioners to gather their opinions on whether the existing work can help them address their DL model reproducibility issues in practice. We summarized their opinions below:

Randomness in the Software: Practitioners are aware that currently the most effective approach to control the randomness in the software is to set seeds prior to training. However, they are reluctant to adopt such practice due to the following two reasons: (1) *a variety of usage context:* for example, in software testing, they would like to reserve the randomness so that more issues can be exposed. However, after the issue is identified, they find it difficult to reproduce the same issue in the next run. Setting seeds cannot meet their needs in this context. (2) *Sub-optimal performance:* DL models often require fine-tuning to reach the best performance. Currently, the DL training relies on certain levels of randomness to avoid local optimums. Setting seeds may have negative impacts on the model performance. Although tools like AutoML [42] have been recently widely adopted for selecting the optimal hyperparameters, there are no existing techniques which incorporate random seeds as part of their tuning or searching processes.

Non-determinism in the Hardware: There are research and grey literature (e.g., technical documentations [14], blog posts [15]) describing techniques to mitigate the non-determinism in hardware or proposing new hardware architecture [45]. However, in an industrial context, adopting new hardware architecture is impractical due to the additional costs and the lack of evaluation and support. In addition, the mentioned approaches (e.g., setting environment variables) are not extensively evaluated on the effectiveness and

overhead. Hence, a systematic empirical study is needed before applying such techniques in practices.

Reproducibility Guidelines: They have already applied best practices to manage the assets (e.g., code and data) used during training processes by employing data and experiment management tools. However, they found the DL models are still not reproducible. In addition, they mentioned that existing techniques in this area does not cover all of their use cases. For example, existing evaluation criteria for DL model reproducibility works for classification tasks (e.g., [56]), but not for regression tasks, which are the usage contexts for many DL models within Huawei. Hence, they prefer a systematic guideline which standardizes many of these best practices across various sources and usage context so that they can promote and enforce them within their organizations.

Inspired by the above feedback, we believe it is worthwhile to propose a systematic approach towards training reproducible DL models. We will describe our approach in details in the next section.

3 OUR APPROACH

Here we describe our systematic approach towards reproducing DL models. Section 3.1 provides an overview of our approach. Section 3.2 to 3.6 explain each phase in detail with a running example.

3.1 Overview

There are different stages in the DL workflow [23]. The focus of our paper is training reproducible DL models. Hence, we assume the datasets and extracted features are already available and can be retrieved in a consistent manner.

Figure 1 presents the overview of our approach, which consists of five phases. (1) During the *Conducting initial training* phase, we prepare the training environment and conduct the training process twice to generate two DL models: $Model_{target}$ and $Model_{repro}$. (2) During the *Verifying model reproducibility* phase, the two DL models from the previous phase are evaluated on a set of criteria to check if they yield the same results. If yes, $Model_{target}$ is reproducible and the process is completed. We also will update the reproducibility guideline if there are any new mitigation strategies that have been introduced during this process. If not, we will proceed to the next phase. (3) During the *Profiling and diagnosing* phase, the system calls and function calls are profiled. Such data is used to diagnose and identify the root causes behind non-reproducibility. (4) During the *Updating* phase, to mitigate newly identified sources of non-determinism, the system calls that need to be intercepted by the record-and-replay technique are updated and the non-deterministic operations due to hardware are patched. (5) During the *Record-and-replay* phase, the system calls, which introduce randomness during training, are first recorded and then replayed. Two DL models, $Model_{target}$ and $Model_{repro}$, are updated with the DL models during the recording and replaying steps, respectively. These two updated DL models are verified again in Phase 2. This process is repeated until we have a reproducible DL model.

To ease explanation, in the rest of the section, we will describe our approach using LeNet-5 as our running example. LeNet-5 [47] is a popular open source DL model used for image classification. The dataset used for training and evaluation is MNIST [9], which consists of a set of 60,000 images for training, 10,000 images for testing. Each image is assigned a label representing the handwritten digits from 0 to 9.

3.2 Phase 1 - Conducting initial training

The objective of this phase is to train two DL models under the same experimental setup. This phase can be further broken down into the following three steps:

Step 1 - Setting up the experimental environment. In this step, we set up the experimental environment, which includes downloading and configuring the following experimental assets: the dataset(s), the source code for the DL model, and the runtime environment based on the required software dependencies and the hardware specifications [43]. Generally the experimental assets are recorded in documentations like research papers, reproducibility checklist [19], or model cards [53] and data sheets [30]. For our running example, documentations are from research papers [47, 56]. The code for LeNet-5 is adapted from a popular open source repository [49], and the MNIST dataset is downloaded from [9]. We further split the dataset into three parts: training, validation, and testing similar to the prior work [56]. In particular, we split the 10,000 images in testing into 7,500 images and 2,500 images. The 7,500 images are used for validation in the training process, and the 2,500 images are used to evaluate the final model, which are not exposed to the training process. We deploy the following runtime environment: for the software dependencies, we use Python 3.6 with TensorFlow 1.14 GPU version. For the hardware specification, we use a SUSE Linux Enterprise Server 12 machine with a Tesla-P100-16GB GPU. The GPU related libraries are CUDA 10.0.130 and CuDNN 7.5.1.

Step 2 - Training the target DL model. In this step, we invoke the training scripts to generate the target DL model, called $Model_{target}$. During the training process, we collect the following set of metrics: loss values, the training epochs, and the training time. This set of metrics is called $ProcessMetrics_{target}$. In our running example, we invoke the training scripts for LeNet-5 to construct the DL model and record its metrics.

Step 3 - Verifying assets and retraining. In this step, we first verify whether the experimental assets are consistent with the information provided in step 1. There are many approaches to verifying the experimental assets. For example, to verify the dataset(s), we check if the SHA-1 checksum is consistent. To verify the software environment, we check the software dependency versions by reusing the same environment (e.g., docker, VM) or simply checking all the installed software packages by commands like `pip list`. Once the assets are verified, we perform the same training process as step 2 to generate another DL model, named as $Model_{repro}$. We also record the same set of metrics, called as $ProcessMetrics_{repro}$, during the training process. The two DL models along with the recorded set of metrics will be used in the next phase for verifying model reproducibility. In our running example, we reuse the same experimental environment without modifying the source code and the datasets to ensure the asset consistency. Then we repeat the training process to collect the second LeNet-5 model and its metrics.

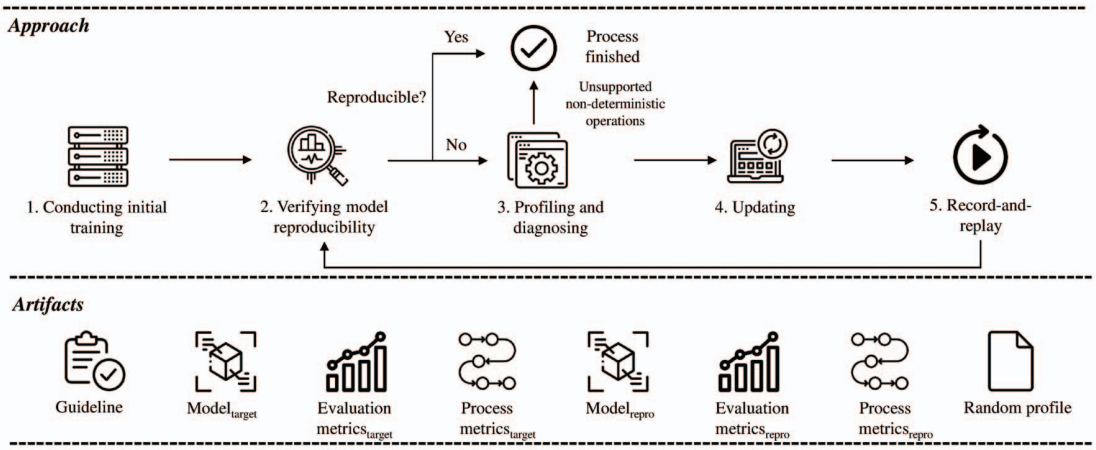

Figure 1: An overview of our approach.

3.3 Phase 2 - Verifying model reproducibility

The objective of this phase is to verify if the current training process is reproducible by comparing the two DL models against a set of evaluation criteria. This phase consists of the following three steps:

Step 1 - Verifying the reproducibility of the training results. In this step, we evaluate the two DL models, $Model_{target}$ and $Model_{repro}$ on the same testing dataset. Depending on the tasks, we use different evaluation metrics:

- *Classification tasks*: For classification tasks, we evaluate three metrics: the overall accuracy, per-class accuracy, and the prediction results on the testing dataset. Consider the total number of instances in testing datasets is N_{test}. The number of correctly labeled instances is $N_{correct}$. For label i, the number of instances are N_{test_i}. The correctly labeled instances of label i is $N_{correct_i}$. Hence, the overall accuracy is calculated as: $Overall\ accuracy = \frac{N_{correct}}{N_{test}}$. For each label i, the per-class accuracy is calculated as: $Per\text{-}class\ accuracy\ (label\ i) = \frac{N_{correct_i}}{N_{test_i}}$. In addition, we collect the prediction results for every instance in the testing dataset.
- *Regression tasks*: For regression tasks, we evaluate the Mean Absolute Error (MAE). The total number of instances in the testing dataset is N_{test} Consider for each instance, the true observed value is X_i and the predicted value is Y_i. MAE is calculated as: $MAE = \frac{\sum_{i=1}^{N_{test}} |Y_i - X_i|}{N_{test}}$. These metrics are called as $EvaluationMetrics_{target}$ and $EvaluationMetrics_{repro}$ for these two models, respectively. In our running example, we use evaluation metrics for classification tasks as LeNet-5 is used for image classification.

Step 2 - Verifying the reproducibility of the training process. In this step, we compare the collected metrics for $Model_{target}$ and $Model_{repro}$ (i.e., $EvaluationMetrics_{target}$ vs. $EvaluationMetrics_{repro}$ and $ProcessMetrics_{repro}$ vs. $ProcessMetrics_{target}$) by a Python script. For evaluation metrics, we check if $EvaluationMetrics_{target}$ and $EvaluationMetrics_{repro}$ are exactly identical. For process metrics, we check if the loss values during each epoch, and the number of epochs are the same.

Step 3 - Reporting the results. A DL model is reproducible if both the evaluation metrics and the process metrics are identical (except for the training time). If the DL models are not reproducible, we move on to the next phase.

In our running example, the two DL models emit different evaluation metrics. The overall accuracy for the two models are 99.16% and 98.64%, respectively. For per-class accuracy, the maximum absolute differences could be as large as 2.3%. Among the 2,500 prediction results, 48 of them are inconsistent. None of the loss values during the epochs are the same. The total number of training epochs are 50 as it is pre-configured. This result shows that the two DL models are not reproducible. Hence, we proceed to the next phase.

3.4 Phase 3 - Profiling and diagnosing

The objective of this phase is to identify the rationales on why the DL models are not reproducible through analysis of the profiled results. The output of this phase is a list of system calls that introduce software-related randomness and a list of library calls that introduce hardware related non-determinism. This phase consists of the following four steps:

Step 1 - Profiling. This step is further divided into two sub-steps based on the type of data, which is profiled:
Step 1.1 - Profiling system calls: After inspecting the documentation and the source code of the DL frameworks, we have found that the randomness from software can be traced to the underlying system calls. For example, in TensorFlow, the random number generator is controlled by a special file (e.g., `/dev/urandom`) in the Linux environment. When a random number is needed in the training, the kernel will invoke a system call to query `/dev/urandom` for a sequence of random bytes. The sequence of random bytes is then used by the random generation algorithm (e.g., the Philox algorithm [60]) to generate the actual random number used in the training process.
Step 1.2 - Profiling library calls: To mitigate the sources of non-determinism in the hardware, popular DL frameworks start to provide environment variables to enhance reproducibility. For example, in TensorFlow 2.1 and above, setting the environment variable

TF_CUDNN_DETERMINISTIC to be "true" could indicate the cuDNN libraries to disable the auto-tuning feature and use the deterministic operations instead of non-deterministic ones. However, there are still many functions that could introduce non-determinism even after the environment variable is set. In addition, lower versions of TensorFlow (e.g., 1.14), which does not support such configuration, are still widely used in practice. To address this issue, Nvidia has released an open source repository [15] to document the root causes of the non-deterministic functions and is currently working on providing patches for various versions of TensorFlow. Not all the operations could be made deterministic and ongoing efforts are being made [17]. Hence, to diagnose the sources of non-determinism in hardware, we perform function level profiling to check if any of the functions are deemed as non-deterministic. Different from profiling the system calls, which extracts call information at the kernel level, the goal of profiling the library calls is to extract all the invoked function calls at framework level (e.g., tensorflow.shape).

In our running example, we repeat the training process of LeNet-5 with the profiling tools. We use strace to profile the list of system calls invoked during the training process. strace exposes the interactions between processes and the system libraries and lists all the invoked system calls. We use cProfile, a C-based profiling tool, to gather the list of invoked functions at the framework level.

Step 2 - Diagnosing sources of randomness. In this step, we analyze the recorded data from strace to identify the set of system calls which can contribute to software-related randomness. We consult with the documentation of system calls and identify the list of system calls, which causes randomness. This list varies depending on the versions of the operating systems. For example, the system call getrandom is only used in later version of Linux kernel (version 3.17 and after). Prior to 3.17, only /dev/urandom is used. Hence, we have to not only search for the list of randomness introducing system calls in the strace data, but also checking if the function parameters contain "/dev/urandom". Figure 2(a) shows a snippet of the sample outputs from strace in our running example. Each line corresponds to one system call. For example, line 10 shows that the program from /usr/bin/python3 is executed with the script mnist_lenet_5.py and the return value is 0. The system call recorded at line 20 reads from "/dev/urandom", and system call (getrandom) recorded at line 51 is also invoked. Both of the two system calls introduce software-related randomness.

Step 3 - Diagnosing sources of non-determinism in hardware. In this step, we cross-check with the Nvidia documentation [15] to see if any of the library functions invoked during the training process triggers the non-determinism functions at the harware layer. If such functions exist, we check if there is a corresponding patch provided. If no such patch exists, we will document the unsupported non-deterministic operations and finish the current process. If the patch exists, we will move on to the next phase. Figure 2(b) shows a snippet of the sample outputs of cProfile for our running example. The functions softmax, weights, bias_add are invoked 3, 101, and 2 times, respectively. We find that bias_add leverages the CUDA implementation of atomicAdd(), which is commonly used in matrix operations. The behavior of atomicAdd() is non-deterministic because of the order of parallel computations is undetermined,

```
Line  System calls
10    execve("/usr/bin/python3", ["python3", "mnist_lenet_5.py"],
      0x7fffd6d52180 /* 25 vars */) = 0
...                                 ...
20    openat(AT_FDCWD, "/dev/urandom", O_RDONLY) = 4
...                                 ...
51    getrandom("\xfb\xc3\x44\xe2\x06\x65\x95\x70\xca\x48\x4b\xd3\x65
      \x9d\xcb\x8f", 16, 0) = 16
...                                 ...
100   exit_group(0) = ?
101   +++ exited with 0 +++
```

(a). The sample outputs of strace.

```
ncalls      Filename:lineno(function)
3           nn_ops.py:2876 (tf.nn.softmax)
101         base_layer.py:742 (weights)
...                                  ...
2           nn_ops.py:2627 (tf.nn.bias_add)
```

(b). The sample outputs of cProfile.

Figure 2: Sample output snippets from strace and cProfile.

which causes rounding error in floating point calculation [15, 56]. The other function calls do not trigger non-deterministic behavior.

3.5 Phase 4 - Updating

In this phase, we update our mitigation strategies based on the diagnosis results from the previous phase. This phase can be further broken down into two steps:

Step 1 - Updating the list of system calls for recording. For the randomness introducing functions, we will add them into the list of intercepted system calls for our record-and-replay technique, so that the return values of the relevant system calls can be successfully recorded (described in the next phase). In our running example, we will add the invocation of reading /dev/urandom and getrandom into the list of intercepted system calls to mitigate randomness in the software.

Step 2 - Applying the right patches for non-deterministic library calls. For the non-deterministic functions related to hardware, we check if there are existing patches that address such problems and integrate them into the training scripts. In our running example, after checking the documentation from the Nvidia repository [15], we found one patch, which replaces bias_add calls with _patch_bias_add. We then integrated the patch to the source code of the training scripts by adding these two lines of code: from tfdeterminism import patch and patch(). In this way, during the subsequent training process of LeNet-5, the non-deterministic functions will be replaced with the deterministic alternatives.

3.6 Phase 5 - Record-and-Replay

As explained in Section 2, presetting random seeds is not preferred by practitioners due to various drawbacks. There are libraries (e.g., numpy) which support the recording and replaying of random states through explicit API calls. However, this method is also intrusive and would incur additional costs we described before. More importantly, mainstream DL frameworks such as TensorFlow and PyTorch do not provide such functionality. Hence, we propose a record-and-replay technique (overview shown in Figure 3) to address these challenges. This phase has two steps:

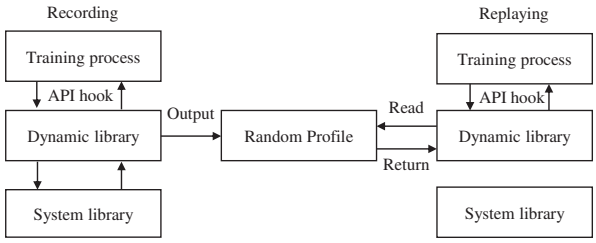

Figure 3: Our record-and-replay technique.

Step 1 - Recording. In this step, we record the random values returned by system calls during the training process. We will run the identical training process as in Phase 1 with the our recording technique enabled. We leverage the API hook mechanism to intercept the system calls by pointing the environment variable `LD_PRELOAD` to our self-implemented dynamic library. It tells the dynamic loaders to look up symbols in the dynamic library defined in `LD_PRELOAD` first. The functions of the dynamic library will be first loaded into the address space of the process. Our dynamic library implements a list of functions which have the same symbols of the randomness introducing system calls in the system libraries. These self-implemented functions will be loaded first and invoke the actual randomness introducing system calls to get the returned random bytes. The sequences of random bytes emitted by the system calls are then recorded into an user-defined object. These objects are then serialized and written into files called the random profile. We replace $Model_{target}$ and $ProcessMetrics_{target}$ with the DL model and the process metrics generated in this step.

In our running example, two types of system calls are intercepted (i.e., `getrandom` and the read of `/dev/urandom`) and the return values are successfully recorded. The outputted random profile is stored at a pre-defined path in the local file system called `urandom.conf` and `getrandom.conf`. For the process-related metrics, we collect the loss values for each epoch (e.g., the loss value of the first epoch is 1.062), the training time (106.9 seconds), and the number of training epochs (50).

Step 2 - Replaying. In this step, we repeat the same training process as the previous step while replaying the random values stored in the random profile by leveraging the API hook mechanism. As shown in Figure 3, our dynamic library will search for existing random profile. If such random profile exists, the recorded random bytes are used to to replace the random bytes returned by the system calls. We also replace $Model_{repro}$ and $ProcessMetrics_{repro}$ with the DL model and the process metrics generated in this step. In our running example, we compare the execution logs between our recording and replaying steps and verify that the same set of random numbers are generated in these two steps.

Once this phase is completed, the two updated DL models are sent to Phase 2 for verifying their reproducibility again. This process is repeated until the DL model is shown to be reproducible or we find certain sources of non-determinism that currently do not have existing solutions. For example, the function `tf.sparse.sparse_den_matmul` is noted in [15] that no solution has been released yet. The reasons for non-reproducibility should be included in the documentations along with released DL models.

Table 1: The DL models used in our case study.

Models	Datasets	# of Labels	Setup	Task
LeNet-1 [47] LeNet-4 [47] LeNet-5 [47]	MNIST [9]	10	All	Classification
ResNet-38 [39] ResNet-56 [39]	CIFAR-10 [3]	10		
WRN-28-10 [69]	CIFAR-100 [3]	100	G1-G10	
ModelX	Dataset X	-	G1-G5	Regression

In our running example, in terms of $Model_{repro}$ and $Model_{target}$, $EvaluationMetrics_{repro}$ and $EvaluationMetrics_{target}$ are identical (i.e., overall accuracy, the per-class accuracy, and prediction results on the testing datasets of two DL models are identical). Except for the training time, the $ProcessMetrics_{repro}$ and $ProcessMetrics_{target}$ are also identical. In conclusion, we consider the trained LeNet-5 models to be reproducible.

4 RESULTS

In this section, we evaluate our approach against open source and commercial DL models. Section 4.1 describes our case study setup. Section 4.2 presents the analysis of our evaluation results.

4.1 Case Study Setup

We have selected six commonly studied Computer Vision (CV) related DL models similar to prior studies [35, 38, 51, 52, 56]. The implementations of these models are adapted from a popular open source repository used by prior studies [35, 38, 49, 52].

Table 1 shows the details about the studied datasets and the models. The studied models are LeNet-1, LeNet-4, LeNet-5, ResNet-38, ResNet-56, and WRN-28-10. These models mainly leverage the Convolutional Neural Network (CNN) as their neural network architectures. We use popular open source datasets like MNIST, CIFAR-10, and CIFAR-100. The models and datasets have been widely studied and evaluated in prior SE research [31, 38, 51, 52]. For models in LeNet family, we train for 50 epochs. For models in ResNet family and WRN-28-10, we train for 200 epochs [56].

We also study ModelX used in a commercial system from Huawei. ModelX is a LSTM-based DL model used to forecast energy usages. ModelX is trained with the early-stopping mechanism and the epochs are not deterministic. The training process will automatically stop when the loss values have not improved for 5 epochs. The maximum number of epochs in the training is set to be 50. ModelX uses proprietary time-series data as their training and testing datasets and is deployed in systems, which are used by tens of millions of customers. Due to the company policy and review standards, we cannot disclose the detail design of the DL model. The implementation of other open source models is disclosed in our replication package [21].

For both open source and commercial models, we perform the training processes with different setups. In total, we have 16 different setups listed in Table 2:

- First, there are two general groups of setups, CPU-based and GPU-based, to assess whether our approach can address different sources of hardware-related non-determinism. For CPU-based

Table 2: The information of all the experiment setups. R&R represents Record-and-Replay.

ID	Hardware	Software	Seed	R & R	Patch
C1	CPU	TF1.14	-	-	-
C2			Yes	-	-
C3			-	Yes	-
C4		TF2.1	-	-	-
C5			Yes	-	-
C6			-	Yes	-
G1	GPU	TF1.14	-	-	-
G2			Yes	-	-
G3			-	-	Yes
G4			Yes	-	Yes
G5			-	Yes	Yes
G6		TF2.1	-	-	-
G7			Yes	-	-
G8			-	-	Yes
G9			Yes	-	Yes
G10			-	Yes	Yes

experiments (i.e., C1 - C6), we only train the models of the LeNet family and ResNet family, as the training for WRN-28-10 and the commercial project takes extremely long time (longer than a week) and not practical to use in field. For GPU-based experiments (i.e., G1 - G10), we conduct experiments on training all the aforementioned models. The CPU used for the experiments is Intel(R) Xeon(R) Gold 6278C CPU with 16 cores and the GPU we use is Tesla-P100-16GB. The GPU related libraries are (CUDA 10.0.130 and cuDNN 7.5.1), and (CUDA 10.1 and cuDNN 7.6) for TensorFlow 1.14 and TensorFlow 2.1, respectively. We use two sets of hardware related libraries due to compatibility issues mentioned in the official TensorFlow documentation [16].

- Then, within the same hardware setup, we also conduct experiments by varying the software versions. For open source models, we use both TensorFlow 1.14 and TensorFlow 2.1. We choose to carry out our experiments on these two TensorFlow versions as major changes [8] have been made from TensorFlow 1.X to TensorFlow 2.X and there are still many models which use either or both versions. Hence, we want to verify if our approach can work with both versions. For ModelX, we only use TensorFlow 1.14 as it currently only supports the TensorFlow 1.X APIs.
- For CPU-based experiments in a particular software version (e.g., TensorFlow 1.14), we have three setups: C1 is to run the training process without setting seeds and without enabling the record-and-replay technique. C2 is to run the training with seeds, whereas C3 is to run the training with record-and replay enabled. For GPU-based experiments in a particular software version (e.g., TensorFlow 1.14), we have five setups: G1 and G2 are similar to C1 and C2. G3 is to run the experiments with patching only to evaluate the variance related to software randomness. G4 and G5 are both running with patches, but configured with either setting seeds or enabling record-and-replay, respectively.

For each setup, we run the experiments 16 times similar to a prior study [56]. The training dataset is split into batches and fed into the training process; the validation dataset is used for evaluating the losses during training; and the testing dataset is used for evaluating the final models. In other words, the training and validation dataset are known to the trained DL models, while the testing data is completely new to the model to mimic the realistic field assessment.

We further divide the 16 runs of DL experiments into 8 pairs, each of which consists of two runs. We then compare the evaluation metrics from each pair of runs to verify reproducibility. For the setups with random seeds configured, we choose 8 most commonly used random seeds for each pair (e.g., 0 and 42) [10]. We collect the process and evaluation metrics as described in Section 3.3.

In addition, for each experiment, we also collect the running time for each of the above experiment to assess the runtime overhead incurred by our approach. We only focus on the experiments conducted on GPU, as GPU-based experiments are executed on a physical machine. CPU-based experiments are conducted on a virtual machine in the cloud environment, which can introduce large variances caused by the underlying cloud platform [62]. For example, comparing the time of G1 and G3 could reveal the performance impact on enabling deterministic patch for GPU. Comparing the time of G3 and G5 could reveal the overhead introduced through record-and-replay technique. To statistically compare the time differences, we perform the non-parametric Wilcoxon rank-sum test (WSR). To assess the magnitude of the time differences among different setups, we also calculate the effect size using Cliff's Delta [59].

Finally, as our approach also stores additional data (e.g., the recorded random profile during the store-and-replay phase), we evaluate the storage overhead brought by our approach by comparing the size of DL models with the size of random profiles.

4.2 Evaluation Analysis and Results

Here we evaluate if the studied models are reproducible after applying our approach. Then we study the time and storage overhead associated with our approach.

Reproducibility by applying our approach. The results show that, the six open source models can be successfully reproduced by applying our approach with default settings. In other words, all the predictions are consistent between the target model and the reproduced model. The default record-and-replay technique intercepts two types of randomness introducing system calls (i.e., the read of `/dev/urandom` and `getrandom`). The default patch is the version 0.3.0 of tensorflow-determinism released in PyPI for TensorFlow 1.14. For TensorFlow 2.1, we need to set the environment variable `TF_CUDNN_DETERMINISTIC` to "true". The results demonstrate the effectiveness of our approach on training reproducible DL models.

Unfortunately, ModelX under such default setup cannot be reproduced. While applying our approach, during the profiling and diagnosing phase, we found one library function (`unsorted_segment_sum`) invoked from ModelX, which cannot be mitigated by the default patch. We carefully examined the solutions described in [15] and discovered an experimental patch that could resolve this issue. We applied the experimental patch along with the record-and-replay technique and are able to achieve reproducibility for ModelX, i.e., all the predictions are consistent.

Overhead. We evaluate the overall time overhead incurred by our approach by comparing training time between the setup without

seed, record-and-replay, and patch against the setup with record-and-replay and patch (a.k.a., our approach). We only compare the training time among open source models, as ModelX adopts the early-stopping mechanism as described above (Section 4.1). As shown in Table 3, training takes longer when applying our approach than the setups without. This is mainly because patched functions adopt deterministic operations, which do not leverage operations (e.g., `atomicAdd`) that support parallel computation. The time overhead ranges from 24% to 114% in our experiments. Although our approach makes training on GPU slower, compared with training on CPUs, training on GPU with our approach is still much faster (e.g., training WRN-28-10 on CPU takes more than 7 days). We further evaluate the time overhead brought by patching and record-and-replay alone. We compare the setup with patching enabled against the setups without it (e.g., G1 vs. G3). We also compare the setup with record-and-replay, patching enabled with the setup with patching only (e.g., G3 vs. G5). The results show that the record-and-replay technique does not introduce statistical significant overhead ($p - value > 0.05$). In other words, patching is the main reason that our approach introduces the time overhead.

Table 3: Comparing the time and storage overhead. Time(O) represents the average training time (in hours) for original setup, and Time(R) represents the average training time (in hours) for the setup using our approach (Time(R)). The time is italicized if p-value is <0.001 and the effect size is large with (*). RP represents for Random Profile.

Model	Time(O)/Time(R)	Model Size	RP Size
LeNet-1	*0.017/0.023* (*)	35 KB	13 KB
LeNet-4	*0.019/0.027* (*)	224 KB	13 KB
LeNet-5	*0.021/0.028* (*)	267 KB	13 KB
ResNet-38	*1.243/1.561* (*)	4.8 MB	13 KB
ResNet-56	*1.752/2.179* (*)	7.6 MB	13 KB
WRN-28-10	*7.08/14.979* (*)	279 MB	13 KB
ModelX	-	675 KB	38 KB

Table 3 also shows the average size of trained DL models and the random profiles. The absolute storage sizes of the random profile are very small, ranging between 13 KB to 38 KB depending on the DL models. Compared to the size of the model, the biggest model is WRN-28-10 (279 MB). The random profile is only 0.005% of the model in terms of the size. When the model is less complex (e.g., LeNet-1), the additional cost becomes more prominent. In LeNet-1, the random profile incurs 37% additional storage. However, the total storage size when combining the model and the random profile for LetNet-1 is less than 50 KB, which is acceptable under most of the use cases.

Summary: *Case study results show that our approach can successfully reproduce all the studied DL models. Patching (i.e., replace non-deterministic operations from hardware with deterministic ones) incurs large time overhead as the trade-off for ensuring deterministic behavior. The record-and-replay technique does not incur additional time overhead in the training process with very small additional storage sizes.*

5 DISCUSSIONS

In this section, we conduct the variance analysis and discuss the lessons learnt when applying our approach.

5.1 Variance Analysis

To measure the variances introduced by different sources of non-determinism, we compare the evaluation metrics among different setups. Such analysis demonstrates the variances between our approach with the state-of-the-art techniques towards reproducing DL models. For example, variances caused by software are analyzed by comparing the evaluation metrics between each pair in G3, where patching is enabled to eliminate hardware non-determinism (i.e., the approach proposed by [15]). To measure the variances caused by hardware, we compare the evaluation metrics between each pair in G2 or G7, where the random seeds are preset to eliminate software randomness (i.e., the approach proposed by [56]). In addition to measuring the software variance and hardware variance, which result from applying two state-of-the-art techniques, we also show the variances incurred from the original setup with no preset seed, record-and-replay not enabled, and patching not enabled. The results of our approach, which incurs zero variances, are also listed in the table.

The detailed results are shown in Table 4. We only include the results for the six open source projects due to confidentiality reasons. Three evaluation metrics are used: overall accuracy, per-class accuracy, and the consistency of predictions. For each type of metric, we calculate the maximum differences and the standard deviations of the differences.

For example, for ResNet-38, the largest variance of overall accuracy in the original setup is 2.0%, while the largest variances introduced by software randomness and hardware non-determinism are 1.4% and 1.2%, respectively. For per-class accuracy, the largest variance in the original setup is 10.1%, while the largest variances introduced by software randomness and hardware non-determinism are 6.8% and 4.9%. For predictions, the largest number of inconsistent predictions in the original setup is 219, while the largest number of inconsistent predictions caused by software randomness and hardware non-determinism are 216 and 209, respectively.

In summary, the variances caused by software are generally larger than those caused by hardware, yet the variances caused by hardware are not negligible and need to be controlled in order to train reproducible DL models. The results demonstrate the importance and effectiveness of applying our approach for training reproducible DL models, as our approach is the only one that does not introduce any variances.

5.2 Generalizability in other DL frameworks

Other than the DL framework studied in Section 4.2, we have also applied our approach on another popular DL framework, PyTorch. Experiment results show that for common models such as LeNet-5 and ResNet-56 with PyTorch version 1.7, our approach can work out of the box. In the future, we also plan to experiment our approach on more DL frameworks and more DL models across different tasks.

Table 4: Comparing variances between our approach and the state-of-the-art techniques. Software variance refers to the technique for only controlling hardware non-determinism [15]. Hardware variance refers to the technique for only controlling software randomness [56]. Original variance refers to the variance caused by the original setup.

		Our Variance		Software Variance		Hardware Variance		Original Variance	
		Diff	SDev	Diff	SDev	Diff	SDev	Diff	SDev
Overall acc.	LeNet1	0	0	0.8%	0.2%	0	0	1.7%	0.3%
	LeNet4	0	0	0.7%	0.1%	0	0	0.8%	0.1%
	LeNet5	0	0	0.5%	0.1%	0	0	0.5%	0.1%
	ResNet38	0	0	1.4%	0.3%	1.2%	0.3%	2.0%	0.4%
	ResNet56	0	0	1.2%	0.3%	0.8%	0.2%	1.7%	0.3%
	WRN-28-10	0	0	1.4%	0.4%	1.7%	0.5%	2.4%	0.5%
Per-class acc.	LeNet1	0	0	3.7%	0.8%	0	0	4.8%	1.2%
	LeNet4	0	0	1.7%	0.3%	0	0	3.0%	0.6%
	LeNet5	0	0	2.3%	0.4%	0	0	2.5%	0.5%
	ResNet38	0	0	6.8%	1.2%	4.9%	0.9%	10.1%	1.9%
	ResNet56	0	0	6.8%	1.1%	5.3%	0.8%	10.5%	1.9%
	WRN-28-10	0	0	35.0%	5.0%	25.0%	3.0%	40.9%	7.8%
Predictions	LeNet1	0	0	48	14.1	0	0	50	17.04
	LeNet4	0	0	31	3.8	0	0	29	3.8
	LeNet5	0	0	28	3.5	0	0	26	3.5
	ResNet38	0	0	216	10.1	209	11.3	219	10.7
	ResNet56	0	0	198	8.6	188	8.8	198	8.0
	WRN-28-10	0	0	485	18.0	453	12.3	542	18.7

5.3 Documentations on DL Models

Mitchell et al. [53] proposed Model Cards to document ML models. A typical model card includes nine sections (e.g., Model Details and Intended Use), each of which contains a list of relevant information. For example, in the Model Details section, it suggests that the "Information about training algorithms, parameters, fairness constraints or other applied approaches, and features" should be accompanied with released models. Such a practice would help other researchers or practitioners to evaluate if the models can be reproduced. However, the current practice would still miss certain details. We share our experience below to demonstrate this point.

TensorFlow and Keras are two of the most widely used DL frameworks. Keras is a set of high level APIs designed for simplicity and usability for both software engineers and DL researchers, while TensorFlow offers more low level operations and is more flexible to design and implement complex network structures. There are two ways of using Keras and TensorFlow in DL training. The first way is to import Keras and TensorFlow separately by first calling `import keras` and then verify if the backend of Keras is TensorFlow. If yes, TensorFlow can be imported by `import tensorflow`. This way is referred to as *Keras_first*. The second way is to directly use the Keras API within TensorFlow by first importing TensorFlow. Then we use another import statement `from tensorflow import keras`. This way is referred to as *TF_first*. We conduct experiments to evaluate if the two different usage of APIs have an impact on training reproducible DL models. As a result, the following findings are presented:

- When training on CPUs, using *Keras_first* will lead to unreproducible results even after mitigating all the sources of non-determinism. This issue can be reproduced by using various Keras version from 2.2.2 to 2.2.5. On the contrary, using *TF_first* with the same setting will yield reproducible results. This issue does not exist in training on GPUs.
- While training with Keras version 2.3.0 and above, we are able to reproduce the results both for *Keras_first* and *TF_first* using our approach. However, the DL models trained using *Keras_first* and *TF_first* are not consistent with each other.

Both findings have been submitted as issue reports to the official Keras development team who suggested us to use newer versions of Keras instead [7, 20]. The findings highlight that not only the versions of dependencies, but also how the dependent software packages are used can impact the reproducibility of DL models. Unfortunately, existing DL model documentation frameworks like Model cards [53] do not specify how the software dependencies should be described. Hence, we suggest ML practitioners look into the approach adopted for traditional software projects like software bills of materials (SBOM) [68] for rigorously specifying software dependencies.

6 GUIDELINE

In this section, we propose a guideline for researchers and practitioners who are interested in constructing reproducible DL models. Our guideline consists of five steps:

(1) Use documentation frameworks such as `Model Cards` to document the details such as model training. Consider leveraging SBOM to document software dependencies. Ensure the documentation co-evolves with the model development process.
(2) Use asset management tools such as DVC [12] and MLflow [1] to manage the experimental assets used during training process. To mitigate the risks of introducing non-determinism from

assets, we suggest using virtualization techniques to provide a complete runtime environment.

(3) Use and document the appropriate evaluation criteria depending on the domain of the DL models. Some of these metrics (e.g., evaluation metrics) may be domain specific, whereas other metrics (e.g., process metrics) are general.

(4) Randomness in the software and non-determinism from hardware are two of the main challenges preventing the reproducibility of DL models. Use record-and-replay technique to mitigate sources of randomness in the software when presetting seed is not preferred. Use patching to mitigate the non-determinism from hardware if the overhead is acceptable.

(5) If DL models are still not reproducible by applying our approach, double check if the list of system calls which introduce randomness changes or if the deterministic operations are not currently supported by the hardware libraries. Document the unsupported non-deterministic operations and search for alternative operations on the same operation.

7 THREATS TO VALIDITY

External Validity. Currently, we focus on DL training using Python along with TensorFlow and Keras framework under Linux. We are currently working on extending our approach to support DL models developed in other DL frameworks and additional operating systems. In addition, we have applied our approaches on two popular domains of DL: classification and regression tasks. We plan to investigate other tasks such as Natural Language Processing and Reinforcement Learning. GPUs and CPUs are common and widely adopted hardware for DL training. Hence, in this paper, we choose to focus on evaluating the DL training on GPUs and CPUs. However, DL training on other hardware such as TPU and edge devices also might encounter reproducibility issues. We believe the idea of our approach can be applied in these contexts as well. Future work is welcomed to extend our approach to different platforms.

Internal Validity. When measuring the variances incurred by different sources of non-determinism, we control the other confounding factors to ensure internal validity. For example, when measuring the overall accuracy variance caused by randomness in software, we only compare the runs with patching enabled and with the same dependencies. In addition, in our evaluation, we repeat the model training process for at least 16 times for each setup to observe the impact of different non-deterministic factors.

Construct Validity. The implementation code for the DL models used in our case studies has been careful reviewed by previous researchers [35, 38, 52, 56]. Our record-and-replay technique for controlling the software factors work when low level random functions are dynamically linked and invoked.

8 CONCLUSIONS

Reproducibility is a rising concern in AI, especially in DL. Prior practices and research mainly focus on mitigating the sources of non-determinism separately without a systematic approach and thorough evaluation. In this paper, we propose a systematic approach to reproducing DL models through controlling the software and hardware non-determinism. Case studies on six open source and one commercial DL models show that all the models can be successfully reproduced by leveraging our approach. In addition, we present a guideline for training reproducible DL models and describe some of the lessons learned based on our experience of applying our approach in practice. Last, we provide a replication package [21] to facilitate reproducibility of our study.

REFERENCES

[1] 2021 (accessed August, 2021). *An open source platform for the machine learning lifecycle.* https://mlflow.org/

[2] 2021 (accessed August, 2021). *Assessment List for Trustworthy Artificial Intelligence (ALTAI) for self-assessment.* https://digital-strategy.ec.europa.eu/en/library/assessment-list-trustworthy-artificial-intelligence-altai-self-assessment

[3] 2021 (accessed August, 2021). *The CIFAR-10 and CIFAR-100 datasets.* https://www.cs.toronto.edu/~kriz/cifar.html

[4] 2021 (accessed August, 2021). *CUDA Toolkit.* https://developer.nvidia.com/cuda-toolkit

[5] 2021 (accessed August, 2021). *Determined AI Reproducibility.* https://docs.determined.ai/latest/topic-guides/training/reproducibility.html

[6] 2021 (accessed August, 2021). *Determinism in Deep Learning (S9911).* https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9911-determinism-in-deep-learning.pdf

[7] 2021 (accessed August, 2021). *Inconsistent results when using two styles of import statements - Issue 14672.* https://github.com/keras-team/keras/issues/14672

[8] 2021 (accessed August, 2021). *Migrate your TensorFlow 1 code to TensorFlow 2.* https://www.tensorflow.org/guide/migrate

[9] 2021 (accessed August, 2021). *The Mnist Database of handwritten digits.* http://yann.lecun.com/exdb/mnist/

[10] 2021 (accessed August, 2021). *Most common random seeds.* https://www.kaggle.com/residentmario/kernel16e284dcb7

[11] 2021 (accessed August, 2021). *Notes from the AI Frontier Insights from Hundreds of Use Cases.* https://www.mckinsey.com/featured-insights/artificial-intelligence/notes-from-the-ai-frontier-applications-and-value-of-deep-learning

[12] 2021 (accessed August, 2021). *Open-source Version Control System for Machine Learning Projects.* https://dvc.org/

[13] 2021 (accessed August, 2021). *Proposal for a REGULATION OF THE EUROPEAN PARLIAMENT AND OF THE COUNCIL LAYING DOWN HARMONISED RULES ON ARTIFICIAL INTELLIGENCE (ARTIFICIAL INTELLIGENCE ACT) AND AMENDING CERTAIN UNION LEGISLATIVE ACTS.* https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A52021PC0206

[14] 2021 (accessed August, 2021). *Reproducibility in Pytorch.* https://pytorch.org/docs/stable/notes/randomness.html

[15] 2021 (accessed August, 2021). *Tensorflow Determinism.* https://github.com/NVIDIA/framework-determinism

[16] 2021 (accessed August, 2021). *TensorFlow GPU Support.* https://www.tensorflow.org/install/source#gpu

[17] 2021 (accessed August, 2021). *Tensorflow RFC for determinism.* https://github.com/tensorflow/community/blob/master/rfcs/20210119-determinism.md

[18] 2021 (accessed August, 2021). *Testing for Deploying Machine Learning Models.* https://developers.google.com/machine-learning/testing-debugging/pipeline/deploying

[19] 2021 (accessed August, 2021). *The Machine Learning Reproducibility Checklist.* https://www.cs.mcgill.ca/~jpineau/ReproducibilityChecklist.pdf

[20] 2021 (accessed August, 2021). *Unreproducible results when directly import keras in CPU environment - Issue 14671.* https://github.com/keras-team/keras/issues/14671

[21] 2022 (accessed Feb, 2022). *The replication package.* https://github.com/nemo9cby/ICSE2022Rep

[22] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek Gordon Murray, Benoit Steiner, Paul A. Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: A System for Large-Scale Machine Learning. In *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016.* USENIX Association, 265–283.

[23] Saleema Amershi, Andrew Begel, Christian Bird, Robert DeLine, Harald C. Gall, Ece Kamar, Nachiappan Nagappan, Besmira Nushi, and Thomas Zimmermann. 2019. Software engineering for machine learning: a case study. In *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*, Helen Sharp and Mike Whalen (Eds.). IEEE / ACM, 291–300.

[24] Dario Amodei, Sundaram Ananthanarayanan, Rishita Anubhai, Jingliang Bai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse H. Engel, Linxi Fan, Christopher Fougner, Awni Y. Hannun, Billy Jun, Tony Han, Patrick LeGresley, Xiangang Li, Libby Lin, Sharan Narang, Andrew Y. Ng, Sherjil Ozair, Ryan Prenger,

Sheng Qian, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Chong Wang, Yi Wang, Zhiqian Wang, Bo Xiao, Yan Xie, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. 2016. Deep Speech 2 : End-to-End Speech Recognition in English and Mandarin. In *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016 (JMLR Workshop and Conference Proceedings)*.

[25] Amine Barrak, Ellis E. Eghan, and Bram Adams. 2021. On the Co-evolution of ML Pipelines and Source Code - Empirical Study of DVC Projects. In *28th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER 2021, Honolulu, HI, USA, March 9-12, 2021*. IEEE, 422–433.

[26] Miles Brundage, Shahar Avin, Jasmine Wang, Haydn Belfield, Gretchen Krueger, Gillian K. Hadfield, Heidy Khlaaf, Jingying Yang, Helen Toner, Ruth Fong, Tegan Maharaj, Pang Wei Koh, Sara Hooker, Jade Leung, Andrew Trask, Emma Bluemke, Jonathan Lebensbold, Cullen O'Keefe, Mark Koren, Theo Ryffel, J. B. Rubinovitz, Tamay Besiroglu, Federica Carugati, Jack Clark, Peter Eckersley, Sarah de Haas, Maritza Johnson, Ben Laurie, Alex Ingerman, Igor Krawczuk, Amanda Askell, Rosario Cammarota, Andrew Lohn, David Krueger, Charlotte Stix, Peter Henderson, Logan Graham, Carina Prunkl, Bianca Martin, Elizabeth Seger, Noa Zilberman, Seán Ó hÉigeartaigh, Frens Kroeger, Girish Sastry, Rebecca Kagan, Adrian Weller, Brian Tse, Elizabeth Barnes, Allan Dafoe, Paul Scharre, Ariel Herbert-Voss, Martijn Rasser, Shagun Sodhani, Carrick Flynn, Thomas Krendl Gilbert, Lisa Dyer, Saif Khan, Yoshua Bengio, and Markus Anderljung. 2020. Toward Trustworthy AI Development: Mechanisms for Supporting Verifiable Claims. *CoRR* abs/2004.07213 (2020). arXiv:2004.07213 https://arxiv.org/abs/2004.07213

[27] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. 2014. cuDNN: Efficient Primitives for Deep Learning. *CoRR* abs/1410.0759 (2014). arXiv:1410.0759 http://arxiv.org/abs/1410.0759

[28] Cédric Colas, Olivier Sigaud, and Pierre-Yves Oudeyer. 2018. How Many Random Seeds? Statistical Power Analysis in Deep Reinforcement Learning Experiments. *CoRR* abs/1806.08295 (2018). arXiv:1806.08295 http://arxiv.org/abs/1806.08295

[29] Andre Esteva, Alexandre Robicquet, Bharath Ramsundar, Volodymyr Kuleshov, Mark DePristo, Katherine Chou, Claire Cui, Greg Corrado, Sebastian Thrun, and Jeff Dean. 2019. A guide to deep learning in healthcare. *Nature medicine* 25, 1 (2019), 24–29.

[30] Timnit Gebru, Jamie Morgenstern, Briana Vecchione, Jennifer Wortman Vaughan, Hanna M. Wallach, Hal Daumé III, and Kate Crawford. 2018. Datasheets for Datasets. *CoRR* abs/1803.09010 (2018). arXiv:1803.09010 http://arxiv.org/abs/1803.09010

[31] Simos Gerasimou, Hasan Ferit Eniser, Alper Sen, and Alper Cakan. 2020. Importance-driven deep learning system testing. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 702–713.

[32] Sindhu Ghanta, Lior Khermosh, Sriram Subramanian, Vinay Sridhar, Swaminathan Sundararaman, Dulcardo Arteaga, Qianmei Luo, Drew Roselli, Dhananjoy Das, and Nisha Talagala. 2018. A systems perspective to reproducibility in production machine learning domain. (2018).

[33] David Goldberg. 1991. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Comput. Surv.* 23, 1 (1991), 5–48.

[34] Sorin Mihai Grigorescu, Bogdan Trasnea, Tiberiu T. Cocias, and Gigel Macesanu. 2020. A survey of deep learning techniques for autonomous driving. *J. Field Robotics* 37, 3 (2020), 362–386.

[35] Jiazhen Gu, Huanlin Xu, Haochuan Lu, Yangfan Zhou, and Xin Wang. 2021. Detecting Deep Neural Network Defects with Data Flow Analysis. In *51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops, DSN Workshops 2021, Taipei, Taiwan, June 21-24, 2021*.

[36] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018, Gothenburg, Sweden, May 27 - June 03, 2018*, Michel Chaudron, Ivica Crnkovic, Marsha Chechik, and Mark Harman (Eds.). ACM, 933–944. https://doi.org/10.1145/3180155.3180167

[37] Odd Erik Gundersen and Sigbjørn Kjensmo. [n.d.]. State of the Art: Reproducibility in Artificial Intelligence. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18)*.

[38] 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 *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*.

[39] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*. IEEE Computer Society, 770–778.

[40] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. 2018. Deep Reinforcement Learning That Matters. In *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18)*. AAAI Press.

[41] Matthew Hutson. 2018. Artificial intelligence faces reproducibility crisis. *Science (New York, N.Y.)* 359 (02 2018), 725–726. https://doi.org/10.1126/science.359.6377.725

[42] Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren (Eds.). 2019. *Automated Machine Learning - Methods, Systems, Challenges*. Springer. https://doi.org/10.1007/978-3-030-05318-5

[43] Samuel Idowu, Daniel Strüber, and Thorsten Berger. 2021. Asset Management in Machine Learning: A Survey. In *43rd IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2021, Madrid, Spain, May 25-28, 2021*. IEEE, 51–60.

[44] Richard Isdahl and Odd Erik Gundersen. 2019. Out-of-the-box reproducibility: A survey of machine learning platforms. In *2019 15th international conference on eScience (eScience)*. IEEE, 86–95.

[45] Hadi Jooybar, Wilson W. L. Fung, Mike O'Connor, Joseph Devietti, and Tor M. Aamodt. 2013. GPUDet: a deterministic GPU architecture. In *Architectural Support for Programming Languages and Operating Systems, ASPLOS 2013, Houston, TX, USA, March 16-20, 2013*. ACM, 1–12. https://doi.org/10.1145/2451116.2451118

[46] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. 2015. Deep learning. *Nat.* (2015).

[47] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. IEEE* (1998).

[48] Brian Lee, Andrew Jackson, Tom Madams, Seth Troisi, and Derek Jones. 2019. Minigo: A Case Study in Reproducing Reinforcement Learning Research. In *Reproducibility in Machine Learning, ICLR 2019 Workshop, New Orleans, Louisiana, United States, May 6, 2019*. OpenReview.net.

[49] Wei Li. 2017. cifar-10-cnn: Play deep learning with CIFAR datasets. https://github.com/BIGBALLON/cifar-10-cnn.

[50] Chao Liu, Cuiyun Gao, Xin Xia, David Lo, John C. Grundy, and Xiaohu Yang. 2020. On the Replicability and Reproducibility of Deep Learning in Software Engineering. *CoRR* abs/2006.14244 (2020). arXiv:2006.14244 https://arxiv.org/abs/2006.14244

[51] 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 *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*, Marianne Huchard, Christian Kästner, and Gordon Fraser (Eds.). ACM, 120–131.

[52] 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 *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.).

[53] Margaret Mitchell, Simone Wu, Andrew Zaldivar, Parker Barnes, Lucy Vasserman, Ben Hutchinson, Elena Spitzer, Inioluwa Deborah Raji, and Timnit Gebru. 2019. Model Cards for Model Reporting. In *Proceedings of the Conference on Fairness, Accountability, and Transparency, FAT* 2019, Atlanta, GA, USA, January 29-31, 2019*, danah boyd and Jamie H. Morgenstern (Eds.). ACM, 220–229.

[54] David Lorge Parnas. 2017. The real risks of artificial intelligence. *Commun. ACM* 60, 10 (2017), 27–31. https://doi.org/10.1145/3132724

[55] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Köpf, 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 *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 8024–8035.

[56] Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. [n.d.]. Problems and Opportunities in Training Deep Learning Software Systems: An Analysis of Variance. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*.

[57] Joelle Pineau, Philippe Vincent-Lamarre, Koustuv Sinha, Vincent Larivière, Alina Beygelzimer, Florence d'Alché Buc, Emily Fox, and Hugo Larochelle. 2020. Improving Reproducibility in Machine Learning Research (A Report from the NeurIPS 2019 Reproducibility Program). arXiv:2003.12206 [cs.LG]

[58] Edward Raff. 2019. A Step Toward Quantifying Independently Reproducible Machine Learning Research. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.).

[59] Jeanine Romano, Jeffrey D Kromrey, Jesse Coraggio, and Jeff Skowronek. 2006. Appropriate statistics for ordinal level data: Should we really be using t-test and

Cohen'sd for evaluating group differences on the NSSE and other surveys. In *annual meeting of the Florida Association of Institutional Research*, Vol. 13.

[60] John K. Salmon, Mark A. Moraes, Ron O. Dror, and David E. Shaw. 2011. Parallel random numbers: as easy as 1, 2, 3. In *Conference on High Performance Computing Networking, Storage and Analysis, SC 2011, Seattle, WA, USA, November 12-18, 2011*. ACM, 16:1–16:12. https://doi.org/10.1145/2063384.2063405

[61] Simone Scardapane and Dianhui Wang. 2017. Randomness in neural networks: an overview. *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.* 7, 2 (2017).

[62] Joel Scheuner, Jürgen Cito, Philipp Leitner, and Harald C. Gall. 2015. Cloud WorkBench: Benchmarking IaaS Providers based on Infrastructure-as-Code. In *Proceedings of the 24th International Conference on World Wide Web Companion, WWW 2015, Florence, Italy, May 18-22, 2015 - Companion Volume*. ACM, 239–242. https://doi.org/10.1145/2740908.2742833

[63] Peter Sugimura and Florian Hartl. 2018. Building a reproducible machine learning pipeline. *arXiv preprint arXiv:1810.04570* (2018).

[64] Peter Sugimura and Florian Hartl. 2018. Building a Reproducible Machine Learning Pipeline. *CoRR* abs/1810.04570 (2018). arXiv:1810.04570 http://arxiv.org/abs/1810.04570

[65] Rachael Tatman, Jake VanderPlas, and Sohier Dane. 2018. A practical taxonomy of reproducibility for machine learning research. (2018).

[66] Ruben Vicente-Saez and Clara Martinez-Fuentes. 2018. Open Science now: A systematic literature review for an integrated definition. *Journal of business research* 88 (2018), 428–436.

[67] Michael Woelfle, Piero Olliaro, and Matthew H Todd. 2011. Open science is a research accelerator. *Nature chemistry* 3, 10 (2011), 745–748.

[68] Curtis Yanko. 2021 (accessed August, 2021). *Using a Software Bill of Materials (SBOM) is Going Mainstream.* https://blog.sonatype.com/software-bill-of-materials-going-mainstream

[69] Sergey Zagoruyko and Nikos Komodakis. 2016. Wide Residual Networks. In *Proceedings of the British Machine Vision Conference 2016, BMVC 2016, York, UK, September 19-22, 2016*. BMVA Press. http://www.bmva.org/bmvc/2016/papers/paper087/index.html