The Challenge of Reproducible ML: An Empirical Study on The Impact of Bugs

Emilio Rivera-Landos, Foutse Khomh, Amin Nikanjam
SWAT Lab., Polytechnique Montreal
Quebec, Canada
Email: {emilio.rivera, foutse.khomh, amin.nikanjam}@polymtl.ca

2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS) | 978-1-6654-5813-9/21/$31.00 ©2021 IEEE | DOI: 10.1109/QRS54544.2021.00116

***Abstract*—Reproducibility is a crucial requirement in scientific research. When results of research studies and scientific papers have been found difficult or impossible to reproduce, we face a challenge which is called reproducibility crisis. Although the demand for reproducibility in Machine Learning (ML) is acknowledged in the literature, a main barrier is inherent non-determinism in ML training and inference. In this paper, we establish the fundamental factors that cause non-determinism in ML systems. A framework, `ReproduceML`, is then introduced for deterministic evaluation of ML experiments in a real, controlled environment. `ReproduceML` allows researchers to investigate software configuration effects on ML training and inference. Using `ReproduceML`, we run a case study: investigation of the impact of bugs inside ML libraries on performance of ML experiments. This study attempts to quantify the impact that the occurrence of bugs in a popular ML framework, PyTorch, has on the performance of trained models. To do so, a comprehensive methodology is proposed to collect buggy versions of ML libraries and run deterministic ML experiments using `ReproduceML`. Our initial finding is that there is no evidence based on our limited dataset to show that bugs which occurred in PyTorch do affect the performance of trained models. The proposed methodology as well as `ReproduceML` can be employed for further research on non-determinism and bugs.**

***Index Terms*—Reproducibility, ML experiments, ML frameworks, Bugs**

I. Introduction

The reproducibility of scientific results is an important topic not only in Machine Learning (ML) but in research in general. It has happened many times that one reads scientific publications showing interesting results but finds them difficult to reproduce. This is called the reproducibility crisis, a methodological phenomenon where results of research studies and scientific papers have been found to be difficult or impossible to reproduce. The reproducibility crisis in ML is acknowledged in the literature [1], [2] and many propositions for research guidelines have emerged by showcasing reproducible research [1], creating guidelines for research formulation [3], guidelines for reproducible research [4]–[7] or even by modelling research [8], [9] and its workflows [10], [11]. Pham *et al.* surveyed the literature to understand the state of reproducibility and repeatability in Software Engineering and Artificial Intelligence (AI), and found that only 19.5% of papers tried to use several identical runs when reporting their results [12]. Similarly, a recent analysis of code inclusion in Deep Learning (DL) papers found that 25.8% of them include code [13]. While code inclusion in a research paper goes one step further for reproducibility, it needs to be usable. A study by Gunderson [14] introduces degrees of reproducibility, and finds that most papers in AI are not fully reproducible. A similar study for ML applied in medicine shows a 9% reproducibility [15]. Other researchers have also called for better reproducibility in ML [2], [16] and DL [17], [18].

Reproducibility of the results is directly impacted by the underlying causes of variance: *Non-Determinism Introducing Factors* (NDIF). In the context of ML, we define NDIF as any change in the ML lifecycle that directly or indirectly affects the training or inference results due to non-deterministic behaviours, so are important to consider for reproducing an experiment. Borrowed from Pham *et al.* [12] and adapted with findings of Crane [19] and Guo *et al.* [20], we present seven main NDIF that cause non-determinism in ML: random seed, model definition, software versions, threading model, runtime, hardware and data. Each of these NDIF finds their non-determinism in three root causes: randomness, versioning, and floating point operations. Fig. 1 establishes a mapping between NDIF and their interactions in ML workflow.

When using a random number generator, the associated runtime will check to see if a special configuration was done which is called "seeding" the number generator. Providing a specific value for the seed should generate the same random number from the generator enhancing reproducibility. While a model is generally static in its definition, as an example non-determinism of a neural network architecture can be showcased by the presence of some layers that are subject to randomness effects, like Dropout methods [21]. The main reason why a threading model can affect non-determinism is the non-associativity of floating-point operations. Therefore, training a model with multiple threads can affect the output of floating point calculations resulting in non-deterministic behaviour. Software versions encompasses two aspects: (1) direct software dependencies of the program and (2) software versions that are available in the runtime. Each component of a runtime environment may execute with different configurations affecting the output. For example, a specific runtime might be built with support for double precision while another might only perform standard precision operations. The hardware used for a specific computation can yield different results in computational workloads, e.g., different instructions leading to inconsistent floating point precision. While using a specific dataset already split in training, validation and test sets, there

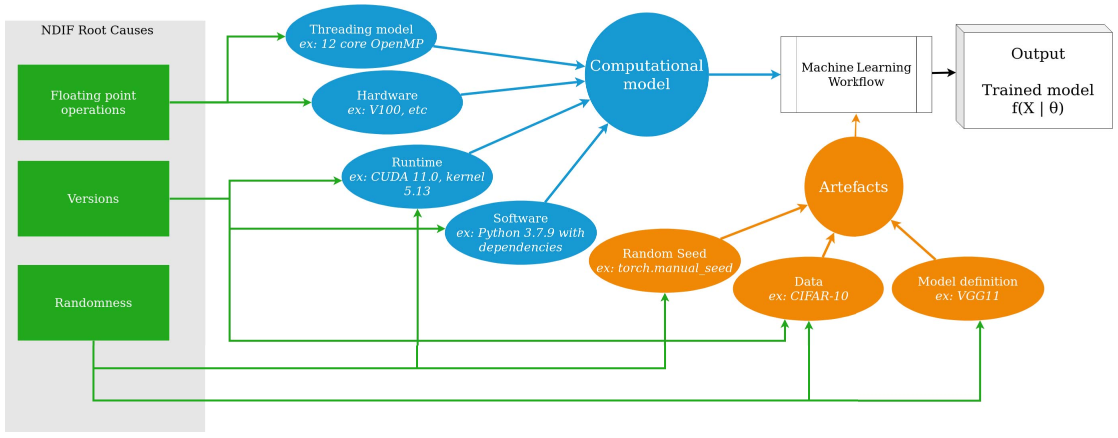

Fig. 1. Sources of Non-determinism Introducing Factors and interactions in ML workflow.

is another factor to take into account: the inner ordering of each of the splits. Ordering of data affects the convergence rate of Stochastic Gradient Descent (SGD) as an optimization method in DL [22].

On the other hand, widely used ML frameworks in research and development (like PyTorch, TensorFlow and Keras) are prone to bugs like other software. There are some empirical studies to understand bugs in ML but the focus of them is on bugs in client applications not bugs inside ML libraries. The only work that considers bugs in ML frameworks studied 202 bugs inside TensorFlow [23]. While root causes and symptoms are analyzed and compared to bugs in traditional software, the empirical impact of bugs on the ML process/model was not touched [23]. Quantifying the impact of defects in the process of ML allows us to make educated decisions on the versions used to develop and deploy a ML-based software systems. Moreover, understanding the effect of defects in frameworks on the ML models will allow us to gain understanding on the robustness of models. However, a prerequisite for such study is managing NDIF during ML training and inference being assured of the effect of a change in the framework or model. In fact, an exact duplication of experiments is highly unlikely, the goal is thus to reduce possible NDIF in order to obtain the same results *within the stated precision* [24]. Moreover, a proper calculation of the results with statistical analysis is paramount for reproducibility in ML.

In this paper, we first introduce `ReproduceML`, a toolset for minimizing the effects of NDIF in ML experiments (reproducible experiments). Then, we suggest a methodology to measure the impact of bugs in ML frameworks using `ReproduceML` answering the following research question: *Given a sample of fixed size of recent bugs in a given ML framework, does the occurrence of bugs affect the performance metrics of ML models?* Our contribution is two-fold: 1) introducing `ReproduceML`, a framework for controlling some NDIFs, i.e., data split and random state configuration, in ML experiments, and 2) leveraging `ReproduceML` to investigate the potential impact of bugs contained inside ML libraries, e.g. Pytorch, on training/inference of ML models. Although such NDIFs are well-known and should be managed in ML experimental designs, to the best of our knowledge, `ReproduceML` is the first toolset that offers automated and systematic control of such factors during ML experiments. It facilitates designing experiments and provides the user with low-variance measurements by controlling NDIFs. The framework is completely open source and available online[1].

The rest of this paper is organized as follows: Section II provides a brief summary of the related works. We introduce `ReproduceML` in Section III, a framework for reproducible training and inference in ML. In Section IV, we present application of `ReproduceML` to investigate the impact of bugs in ML frameworks. To do so, we describe our designed methodology from data collection to statistical analysis. In Section V, we apply our methodology on PyTorch and present the results including statistical analysis. In Section VI, we explain the main limitations and threats to validity of this study. We conclude the paper in Section VII by explaining potential future research directions.

II. Related works

Vanschoren *et al.* tried to standardize and centralize research results and processes for ML by constructing "experiment databases" [9], allowing researchers to explore and query results from ML findings. Building on this work, OpenML is created, a platform allowing for sharing datasets, implementations and results among ML researchers [25]. Using such platform has the benefit of improving reusability and reproducibility of previous ML research. In another work, authors set the same random seed in order to measure difference in performance between different random seeds, and between PyTorch and TensorFlow [17]. They show a 7% difference between the random seeds; which is similar to the aforementioned results. More interestingly, they look at the "Sensitivity to Data Ordering" and show that there is as much

[1]https://github.com/swatlab/ml-frameworks-evaluation

as 3% difference when changing the order in which the data was fed.

There is a promising tool for managing the ML workflow in an ecosystem called MLFlow [26]. Its main goal is to simplify the lifecycle of a ML project by wrapping over existing processes. Such encapsulation of the lifecycle, they argue, benefits the ML practitioner. Moreover, reproducibility is boasted as one of its four challenges addressed. MLFlow allows for tracking of experiment metrics and configuration via file configurations.

A survey was conducted by Isdahl comparing the existing ML "platforms". These platforms are ecosystems for managing the ML workflow in some way or another [27]. They rank these platforms by comparing their "out-of-the-box reproducibility". They show that most of these ecosystems offer some degree of reproducibility, but none of them achieves total reproducibility. Therefore, reproducibility is not yet completely managed by a single ML ecosystem and it is up to practitioners and researchers to make sure their work is reproducible.

Data Version Control (DVC) is a tool that aims to facilitate scientific experimentation in ML, both in terms of shareability and reproducibility [28]. They use a variety of techniques and tools to make sure experiments can be easily shared and reproduced. Our work, `ReproduceML`, resembles DVC closely as we also use configuration files to control reproducibility, notably for datasets. Our approach differs in that it acts more as a central database for collection and reproduction of runs. Actually, `ReproduceML` may benefit from DVC in order to verify that data stays the same between experiments. Moreover, we aim at ensuring that the *runtime* on which experiments are executed are tightly controlled, in order to collect measurements. Our work can benefit from DVC in order to make sure data stays the same between experiments.

III. The framework

The creation of `ReproduceML` is motivated by the need for reproducibility in ML and the lack of existing solutions at the time of creation of the framework. It was originally designed to reduce the amount of variance between multiple runs of ML experiments (training or inference).

At its core, `ReproduceML` is a training benchmarking framework developed by Python focused on *repeatability* and striving for *reproducibility*. As there are technical limitations, prohibiting complete reproducibility, we will use the term "reproducibility" to indicate both "repeatability" and "reproducibility". `ReproduceML` configures the necessary runtime mechanism for other existing ML frameworks in order to have reproducible results. It focuses on reproducible data splits and reproducible random state configuration where possible. Another goal of `ReproduceML` is to have a simple usage with as simple as possible opt-in mechanism: this is done by defining clear interfaces. It is primordial to consider that `ReproduceML` aims to address reproducibility at the data and randomness aspects of the ML workflow. The goal here is not to create an ecosystem to ensure reproducibility but rather to provide a framework for the specified parts: runtime configuration, data management and randomness seed archival. Other tools exist and their combined usage with this framework is what will allow empirical studies to achieve better reduction of confounding factors. `ReproduceML` performs the concept of "data versioning" but has some key differences to DVC [28]: our work is a benchmarking tool that integrates data versioning.

`ReproduceML` consists of two parts: a client and a server, each of which needs special configuration to achieve reproducibility. The client trains ML models in tightly controlled experiments which are given by a server. The server is responsible for giving the training and test data subsets in a deterministic fashion and to collect results from multiple clients. The server can be viewed as static, whereas the client is dynamic. Indeed, the server acts as a central repository for data collection, challenges and most importantly deterministic data serving to clients. Clients, on the other hand, are expected to widely vary in configuration, hence there are two key aspects to consider: (1) reproducible random seed initialization and processes during training and (2) communication with the server in order to collect data and send metrics. What makes this portion of the framework stand out from most other available frameworks is its focus on reproducibility rather than performance. We chose a client-server model where `ReproduceML`'s server is implemented as a metric collection center to allow reusability and clients are expected to widely vary in runtime configuration. In this section, we explain how `ReproduceML` deals with the essential parts of the ML workflow in order to have better reproducibility and repeatability than standard ML workflows.

A. Client

The client has the responsibility to configure the runtime using reproducibility mechanisms available in the dependencies. Therefore, the client is strongly tied to ML frameworks and their versions. The client uses the built-in mechanisms of dependencies and frameworks to correctly set the runtime. As an example, the core of the framework comes with support for PyTorch and interfaces with the runtime for model training. Interoperability with various frameworks is done by leveraging the interpreted nature of Python and its mechanism for dynamic imports. Using this, `ReproduceML` sets the several mechanisms to make sure the randomness is controlled.

Users are expected to import their models by inheriting from a base class or creating a class that contains the model. The training procedure is entirely controlled by the framework as the interface from the base class is representative of a typical DL training procedure. Currently, `ReproduceML` comes with two predefined models for image classification: VGG [29], and AlexNet [30] taken from the official implementation of Torchvision [31]. It also comes with a CNN inspired from VGG but adapted for working with 32x32 images, internally named `VGG-X`. When running a model with the framework, one specifies the runtime she wants to load and the model name, among other parameters. Therefore the goal is for a user to write her model with the ML frameworks of her choice, make sure it is supported by the framework and launch a client running procedure. Provided that the server is running, this setup will allow for reproducible runtime configuration done by the framework and the training sequence can begin.

B. Server

The server responsibility of `ReproduceML` is two-fold: (1) keeping metadata and metrics about experiments and (2) serving data to a client in a deterministic and reproducible fashion. The server controls the metadata by holding and safekeeping seed values for an experiment. In this context, an experiment is a unique identifier that is supplied by the client. Deterministic random seeds are generated when a new identifier is transmitted by a client and are saved in a file by the server. These seeds are used both by the client and the server to configure the randomness on their end. On the server-side, reproducible data splitting directly uses the random seed allowing for consistent serving of data. Moreover, the server collect metrics from runs by establishing a session with a simple protocol for communication. This protocol is followed by the current Python implementation provided in `ReproduceML`.

It is also possible to use the server standalone and simply leverage the communication protocol defined within `ReproduceML`, as the underlying protocol is TCP. Therefore `ReproduceML` could be used to make a simple metric collection server and dashboard for comparison based solely on the identifier. This allows for others to provide similar work of controlling randomness on the clients not using a Python runtime.

C. Communication

The client and the server communicate over TCP to exchange training data, seed information and metric collection. There are some rudimentary safeguards currently implemented in `ReproduceML`'s communication protocol to make sure data is correctly received and unchanged. Notably, as a precautionary action, both parts of the communication check data integrity when receiving information. Another precautionary action taken is by sending the value of the seed when the client asks for data for a specific run. The seed is compared to the server's version and if a mismatch is found, the server stops. Moreover, a checksum is used on both sides to make sure data integrity and ordering are preserved.

IV. Impact of bugs in ML frameworks: A case study

In this section, we describe a general methodology applied in order to measure the impact of *code changes* that intended to be bug fixes in a ML framework using `ReproduceML`. The main idea is to obtain versions of a framework before and after a change is introduced and measure the impact of the change based on a particular evaluation procedure.

We define a *change* as the difference between two accessible *versions* of a software artifact. Practically, a change is a difference between *versions* in a Version Control System (VCS), *i.e.,* a difference between two "commits" in `Git`. The presence and absence of a bug, by the process of bug fixing can be seen as a change: therefore this methodology applies directly to bugs; since the overall idea is to quantify the impact that the presence or absence of a bug has on ML model performance, we use two different versions of the framework. The first version containing the bug is denoted as "buggy" whereas the version that does not have the bug is denoted as "corrected". Generally, for an end-user of a library, the buggy and corrected versions would be separated by an official release of the software. However, while applying the methodology using official releases would be much faster and accessible, by doing so we would not measure *only* the bug presence, but also all the other changes that were bundled in the same version. Therefore, we opt not to use official releases. We rather compare the absence and presence of a bug by obtaining an artifact, *i.e.,* a build of version, with and without the defect's presence. Consequently, for each bug the key information is its buggy and corrected versions. In the current study, we propose to set the "buggy" and "corrected" versions as respectively, the *last version* that contained the bug and the version that contains the bug fix. Therefore, the bugs are separated by a single commit .

The general process consists of the following steps: (1) Data collection, (2) Data Filtering, (3) Artifact generation, (4) Evaluation (experimental runs) and (5) Statistical analysis. These steps are generic and can be applied to any ML framework, however special considerations might lead to small experimental differences from framework to framework. The methodology is designed to reduce the amount of confounding variables and non-determinism, and thereby variance. In the rest of this section, we will present the details of each step.

A. Data collection

The first step is to collect information about the past defects in the framework, *i.e.,* identify what bugs are present. There are various ways this can be done as a consequence of ML frameworks varying in size, software engineering complexity, documentation and release process. In either case of manual or systematic bug collection, the core information that needs to be available at the end of this step is: (1) **buggy** and **corrected** versions of each bug, (2) enough information to build an artifact for each version and (3) enough information on the understanding of the bug to apply a decision on the filtering process. As mentioned before, we take **buggy** and **corrected** versions to be before and after a bug fix is applied to the main branch of the project. In the following, we present specific methodological considerations taken in order to carry out an empirical study for the bugs in PyTorch.

The study analyzes the bugs on specific releases ranging from version 1.1.0 up to version 1.6.0. In the case of PyTorch, the software release process has been streamlined and changes are usually correctly reported in their **Release Notes** with corresponding bug fix information. Therefore, we use the bug fix sections of each release note to construct our corpus. We followed this methodology since the bugs that are reported by the maintainers are guaranteed to be coming from the framework and are definitely labelled as bugs. As PyTorch's repository is hosted on GitHub, it is possible to mine the issues to find the bugs since GitHub provides a web API to get information about issues opened and their labelling scheme. However, we found that in practice this approach leads to more complications, as a bug might be wrongly labelled by the triage system in place. Additionally, using an issue-mining approach is time dependent, as labels can be added and removed at any time. Following the guideline of never changing the content of an official release by using consistent semantic versioning principles allows us to mine the information directly from the release notes. We extracted 737 bugs following an automated approach to parsing of release notes published by the maintainers. This parsing yielded metadata such as the Pull Request number, links to description, etc. We call this set of all collected bugs as B_a.

B. Data Filtering

After having collected all the bug information, there is a necessary step of manual filtering, *i.e.,* creating a subset of B_a which is called B_f containing bugs that are relevant (independent of end-user) and "silent". Each project being managed differently, specific methods, adapted for each framework's development life-cycle, to list potential bugs to be studied are to be considered. However, for each framework, the following conditions for a specific bug to be examined share the same rejection criteria. For each bug $x \in B_a$:

1) Reject x if it causes a compilation error,
2) Reject x if it causes a runtime error, causing the application to exit (crash),
3) Reject x if it is caused by end-user code.

Our reasoning behind these criteria is to only study bugs that are *silent* and could easily be missed during the process of training ML models. In the first two cases, we therefore remove all non-silent bugs. Lastly, we do not consider bugs that would be opened due to errors by end users, as they do not constitute a bug of the ML framework itself. Bugs that comply with the such conditions are added to the set of issues to be examined, B_f. We left with 439 bugs at the end of this phase.

For the current study on PyTorch, we added the following constraints to limit the search of bugs. These constraints were applied in order to prioritize bugs that would have a clear impact on training and that would be easy to replicate. Thus, a manual filtering effort was made with the following additional criteria:

- Rejection criteria:
    1) The bug is on CPU, so running an experiment may take too much time in comparison to GPU,
    2) We cannot find an application that would be affected: the bug must have an impact on ML experiments: creating a ML model or using it for training/inference.
- Favourable acceptance:
    1) Related to gradients,
    2) Related to mathematical functions,

We use "favourable acceptance" instead of "acceptance": this is done to avoid false negatives. This effort was made by three persons possessing a working knowledge of ML and DL. This criterion still may lead to false positives and false negatives that is discussed in Section V-C. We analyzed 439 bugs out of which 115 met our criteria.

C. Artifacts generation

This step aims to create artifacts that can be used to measure the impact of the bug at both **buggy** and **corrected**. For each bug-revision pair, one needs to go through the entire build pipeline to produce an artifact that is suitable to be installed and evaluated. The build process between a bug's **buggy** and **corrected** revisions have to be identical, while the build and linking dependencies need to be tightly controlled. Ideally, build dependencies are identical as well as the build process. While this specific step is to-the-point, there are magnitude of precautions and limitations that one needs to take into account. Section VI discusses the precautions and limitations in more detail, notably on why a single build configuration environment is probably not enough. Depending on the framework under study, certain other precautions might apply. However, for each bug $x \in B_f$, we:

- Identify the build tool chain and configuration needed, namely T_x,
- Build a version x_b for the **buggy** version using T_x at revision **buggy**,
- Build a version x_c for the **corrected** version using T_x at revision **buggy** and **corrected**.

We successfully completed the build process for the buggy and corrected versions of 49 bugs out of 115. For the rest, we could not successfully run the entire build pipeline for all necessary versions due to issues in building and linking dependencies.

D. Evaluation (experimental runs)

This step aims to define the measurements of performance of ML models running on artifacts, for both **buggy** and **corrected** versions. The general idea is to train both versions on the same model, hyperparameters, data, randomness configuration, data dependencies and hardware and gather a series of measurements during training and evaluation: these are *experiment attributes*.

An *experiment* constitutes the whole metric collection process of a training procedure executed multiple times, each repetition being denoted as a *run*, for a specific version of the framework. In this context, a training procedure aims to recreate as closely as possible the process of learning used in modern ML; a run's output is a trained model that gives acceptable performance along with the metrics calculated against a test set. An *experiment* is uniquely characterized by the attribute in Table I. More specifically, a training procedure

TABLE I
ATTRIBUTES OF AN EXPERIMENT

Bug identifier	-	A user given name
Evaluation type	t	Either "**buggy**" or "**corrected**"
Model	m	Model used to train
Challenge	c	The dataset used to train the model
State	s	The random state used
Artifact	b	Specific build that showcases the bug
Software	d	Software dependencies used for the runtime
Epochs	t	Number of epochs used to train the model

(run) is the set of steps taken to train a model from scratch. In the case of neural nets, the training procedure consists of multiple training epochs over the same dataset and updating the weights. For each bug, we will conduct two experiments: one for the **buggy** and one for the **corrected**. We use the term *performance* as various measurements of a model's capability to correctly accomplish its task. Notably, for classification problems, metrics used are Accuracy, Precision, Recall and F1-score, among others.

For a single bug $x \in B_f$, let e_b and e_c be its *experiment* for respectively **buggy** (b) and **corrected** (c) versions. The only experimental attributes (see Table I) that differ between e_b and e_c are their *evaluation type* t and artifact b. The following procedure is done to collect metrics for an experiment consisting of N runs, it characterizes a model's performance given a specific set of experiment attributes:

- At the start of each run r_i, all random configurations must be reset to specific state s,
- Each r_i must consist of a training algorithm spreading over e_t epochs,
- At the end of each r_i, measure performance metrics on test set and record metrics,

An appropriate number of runs N is to be set for analysis: this depends on various factors, but we recommend at least 30. Hence, for each bug $x \in B_f$

- Choose experiment attributes e (Table I) representative instance of x
- Apply evaluation procedure with experiment attributes for e_b, the **buggy** version.
- Apply evaluation procedure with experiment attributes for e_c, the **corrected** version.

Table II shows a sample output for a single experiment. We attempted to run experiments with all 49 bugs and manually inspected the results. The goal was making sure that experiments were out of error (e.g., failure of training procedure) and conducted under the same experimental setup. This process led to the exclusion of 39 bugs and finally, we were able to complete our statistical analysis on 18 bugs.

E. Statistical analysis

In order to quantify the impact of each bug, two distributions are measured: the distributions of performance metrics when the bug is present (buggy) and when the bug is corrected (corrected). To reduce the variability, the same random seed is used for each experiment, that is both the **buggy** and **corrected** versions of a bug will be trained using exactly the same data and initial random seed. More generally, the **buggy** and **corrected** versions should run with identical NDIF.

TABLE II
METRICS COLLECTION SAMPLE

	0	1	...	N
accuracy	a_0	a_1	...	a_n
precision	p_0	p_1	...	p_n
recall	r_0	r_1	...	r_n
f1	f_0	f_1	...	f_n

In order to answer our research question, we define the following hypotheses:

H_0 For a training procedure, there are no differences in a metric between the buggy and the corrected version of a bug fix.

H_1 For a training procedure, there is a difference in a metric between the buggy and the corrected version of a bug fix.

We formulate the alternative hypothesis as a strict inequality since we have no prior knowledge of measurements, *i.e.,* we use a two-tailed test. We use Wilcoxon-Mann-Whitney test between the performance metrics of each version, *i.e.,* between the training procedure results using e_b and e_c. Wilcoxon-Mann-Whitney is a nonparametric test that measures the difference in mean ranks between two samples [32]. There are two versions of this test: one in which observations are paired and one that observations are not paired. Since all initial factors are identical for each run, ordering of the runs is not important. Furthermore, the samples are not considered paired, as each run contains the same random seed configuration and is reinitialized to its initial state. In this case, as the metrics for runs are not paired we use the non-paired Wilcoxon-Mann-Whitney's test: the *Wilcoxon-Mann-Whitney U-test* with a 95% confidence interval for measuring the impacts.

Regarding the experimental setup, one needs to train models repeatedly to get all the required performance metrics. However, using a traditional environment for training does not suffice. Indeed there are a plethora of factors affecting the performance of a ML model. In this study, it is imperative that the runtime and its dependencies be the same for both experiments of the same bug. Therefore, a consistent runtime needs to be used: we employ `ReproduceML`, presented in Section III in order to consistently collect the data needed for statistical analysis. While `ReproduceML` deals with the configuration of randomness and data across runs, there are other mechanisms that it cannot control, notably the runtime itself. Consequently, a virtual machine with predefined Docker [33] images provides a good compromise in runtime configuration and ease of configuration.

V. EXECUTION AND RESULTS

In this section, we present the results of application of our methodology on a set of bugs in PyTorch using `ReproduceML`.

TABLE III
SHARED ATTRIBUTES FOR OUR RUNS

Model	VGG-X (no Batch Normalization)
Challenge	CIFAR-10
State	*Depends on the bug*
Software	*Depends on the bug, but Docker image contains specifics*
Epochs	30

A. Experimental setup

In the following, we overview the overall experimental configurations. For a complete list of our hardware and software used throughout the sections, please see the online repository[2].

To build the various versions of PyTorch, we used Google Cloud Platform (GCP) virtual machines running *Docker* containers. We use two images to build the versions: the first is a container loaded with the tools needed to build PyTorch at Python 3.7.9 while the other is for the version 3.6.7. This dual setup is to make sure that we replicate a plausible build. For running the experiments, we leverage virtual machines for hardware control and Docker containers for software and runtime dependency control. The GCP virtual machine used is a `n1-standard-16` with virtualization on *Skylake* CPUs running Linux kernel version 5.4.0-1028-gcp, on Ubuntu 18.04.5 equipped with a *NVIDIA V100-SXM2-16GB* running driver 450.51.06 with `CUDA` 11.0. Experiments ran on Docker 19.03.13 accelerated by the nvidia-container-runtime.

Our current experimental runs use `VGG-X` from the default models provided by `ReproduceML` to run experiments on working on the CIFAR-10 dataset [34]. This choice is motivated by a good balance between complexity, time to train, and relative novelty of the model. We run our model for 30 epochs over 50 experimental runs for each of the **buggy** and **corrected** versions. The optimizer is set by the framework and uses SGD with a learning rate of 0.01 and a momentum of 0.5. We chose to experiment on a Docker container in order to provide a reliable and reproducible software configuration. The overhead induced by Docker is not noticeable and has been shown to be small when training models [35]. Table III shows the attributes we use for our experiments. The random state used varied from a bug to another, as we are not interested in measuring the stability for a single random seed. Within a single bug, the random seed stays identical for all the runs.

Since our challenge is CIFAR-10, the problem is multiclass. Therefore, there are multiple ways to calculate the accuracy, precision and recall. In this study, the scores reported are using the unweighted mean of scores of each metric. That is, we average the metric for each class, without weighing by the number of representatives in each class.

B. Results

We have been able to measure successfully 18 bugs and conduct the Wilcoxon-Mann-Whitney test on their buggy and corrected versions.

We present the results after having processed the log files emitted by our experimental setup. There were manual steps needed to parse and verify that results were clean of any mistakes. Notably, we verified that each experiment was conducted under the same experimental setup as described in Section IV. There are two bugs that did not complete the 50 runs mandated for both **buggy** and **corrected**, namely experiment `study-pr36832` for its buggy version completed 47 runs (and 50 for its corrected). The second is experiment `study-pr31433` which completed 43 runs for its corrected version and 50 for its buggy version. All other bugs completed 50 runs; we mark with "†" results that did not complete full 50 runs for both versions. We kept these two bugs for full disclosure.

As mentioned in the methodology, we use a Wilcoxon-Mann-Whitney non-paired test to indicate whether or not the results did show an impact. Table IV shows the p-values of the hypothesis test for all of our experiments. The Wilcoxon-Mann-Whitney test can accept a different number of samples in each population, but in this case, we choose to only compare the same number of samples in the population. That is, for the bugs that did not complete the full experiments, the values portrayed are for a Wilcoxon-Mann-Whitney test done with the minimal number of runs available by either buggy or corrected version.

We can see that only one bug did show in fact a statistically significant result for our experiment: `study-pr31433`. This bug's corrected version only completed 43 runs. Even when computing the p-value by using the maximum number of samples in each population (50 for buggy and 43 for corrected), we still obtain p-values less than 0.05 for each metric. We can see the distribution of each metric in Fig. 2. Interestingly, this bug is not related to our CNN: the bug fix is related to completely different files, *i.e.,* LSTMs. We categorize this value as an outlier as there is no reason to suspect there was a difference between the two versions. We can therefore say that with the current data, there is no evidence to reject H_0, thus we cannot conclude that studied bugs have an impact on model performance. We will discuss several aspects that introduced non-determinism in our experiments in Section VI.

We here report the standard deviation of our runs per metric. This allows us to observe the effects of NDIF on the runs. We can observe in Fig. 3 that the standard deviation for each metric is pretty small for all runs, indicating that some control of NDIF was done, without, however, quantifying the impact that is left to control.

C. Discussion

For each experiment, the standard deviation for all metrics across runs is situated between 0.2% and 0.51%, showing that controlling NDIF was done relatively well. We compare ourselves to work done by Pham *et al.* [12], in which we can observe that the larger models have a bigger standard deviation than smaller models when using a fixed seed. Notably, our model uses around 15 million parameters with the Dropout

[2]https://github.com/swatlab/ml-frameworks-evaluation

TABLE IV
P-VALUES OF THE HYPOTHESIS TEST FOR ALL EXPERIMENTS

	Metrics			
	Accuracy	Precision	Recall	F1
study-pr31167	0.62929	0.42190	0.58365	0.88217
study-pr31433†	**0.03320**	**0.00243**	**0.03508**	**0.01057**
study-pr31552	0.92581	0.58365	0.97525	0.73292
study-pr31584	0.58356	0.56486	0.62208	0.57893
study-pr32044	0.84956	0.39074	0.78009	0.48842
study-pr32062	0.46259	0.87673	0.45033	0.77481
study-pr32350	0.48821	0.55096	0.53270	0.65657
study-pr32541	0.63418	0.63185	0.61237	0.39074
study-pr32829	0.50135	0.21085	0.50589	0.39455
study-pr32831	0.66144	0.94230	0.65657	0.95327
study-pr32978	0.31404	0.29949	0.35740	0.44620
study-pr33017	0.14363	0.21849	0.12005	0.10014
study-pr35022	0.28821	0.51031	0.29629	0.32932
study-pr36820	0.20198	0.53270	0.20097	0.57422
study-pr36832†	0.30004	0.37223	0.26631	0.26958
study-pr37214	0.67396	0.72257	0.69181	0.69181
study-pr38945	0.40400	0.97525	0.36465	0.58365
study-pr39903	0.97249	0.74854	0.98625	0.93681

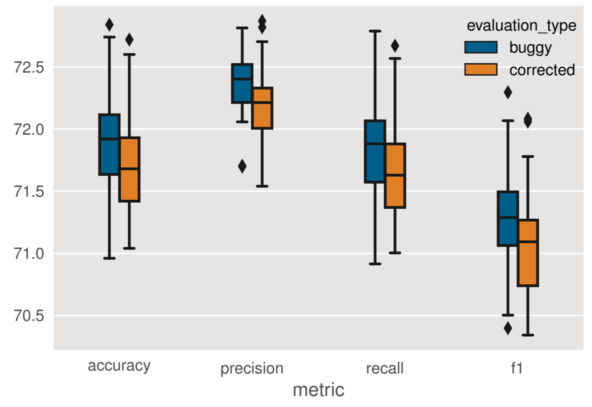

Fig. 2. Distribution of metric values for bug `study-pr31433`

effect encountered and still shows mean standard deviations around 0.3%, comparatively, ResetNet56 shows similar range, around 0.5% of standard deviation. While there is no hard evidence of the effect of size, *i.e.,* statistical test to show this, based on observations we think that bigger models have a higher potential of being affected by NDIF. It would be interesting to see how Pham *et al.* constructed their models, since they said it was reviewed multiple times [12]. However, at this time, their repository does not seem to contain their implementation[3].

During our experiments, there were 3 bugs that have been excluded for not using the same seed: these are the consequences of ungraceful shutdowns due to GCP specificities, leading to data loss of seed value. While these results were discarded for not using the same random seed, they did prove statistically significant impact. Consequently, we can argue that when using different random seeds, a statistical test such as Wilcoxon-Mann-Whitney might positively reject H_0.

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

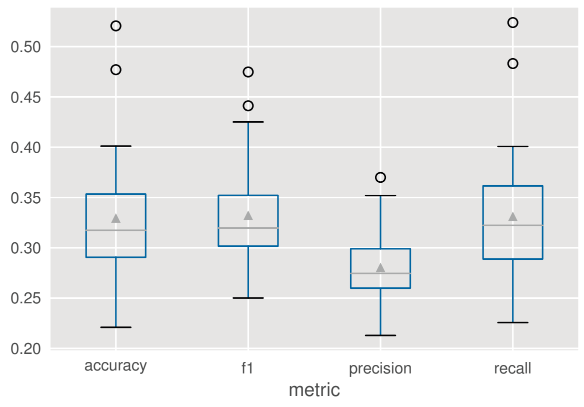

Fig. 3. Distribution of metric standard deviations across all experiments

VI. Limitations and Threats to Validity

This section exposes the different limitations that are present in our realization of the methodology from a theoretical and practical standpoint.

A. `ReproduceML` *framework*

To add a new ML framework, the main hurdles to usability is that `ReproduceML` is limited by the external ML frameworks' ability to control randomness and ensure reproducibility. Therefore, we strongly encourage ML framework developers and hardware makers to provide compatibility for reproducible workflows. One limitation of the current implementation is the backup mechanism and checkpoint system for seeds: there is a mechanism in place in order to save the values when the program ends or a signal is received, but this needs to be done more periodically, perhaps at each generation; unexpected errors can cause seed values to be lost in environments where programs can end at any moment. Moreover, there is no validation set generation, *i.e.,* data splits are made in "train" and "test". It is however trivial to add support for validation sets if one desires to use it as a metric collection and challenge database. There are multiple factors affecting the stochastic aspect of training a model that cannot be controlled by runtime configuration: *e.g.,* installed version dependencies and the runtime itself. These issues are meant to be addressed in future work or by other tools and techniques.

B. *Other NDIF*

There are multiple non-deterministic sources that have not been controlled. Our work is currently limited and does not offer any guarantee against multiple confounding variables such as floating point calculations, possible thread interactions and GPU models. These are however mitigated by the fact that we carry the computations by using the same hardware, runtime, software versions and threading model for our experiments. Consequently, the results for a specific bug, *i.e.,* between a buggy and a corrected experiment, use the same experimental attributes, thus reducing the effects of NDIF. Specifically for thread interactions, `ReproduceML` uses PyTorch's built-in mechanisms in order to control its thread interactions. So, our experiments use only one CPU core to do its work, thereby eliminating some thread interactions from sources of errors. However, there is not a guarantee that PyTorch's thread interactions are complete, and our work is based on their suggestions for reproducibility [36], [37].

While we use `ReproduceML` in order to control the random seed on runtime, there are still variations present in the runs. This indicates that additional precautions need to be taken when interpreting results. We believe that they are the result of NDIF at the implementation level, as results by Pham *et al.* show a similar range of values for a similar model to ours [12]. Our current work does not support detecting if floating point operations are the same between experimental runs. There has been some research on making float point operations more stable [38], but we are aware that our current framework values only hold for a specific architecture, which is why we disclose the hardware and software used in the study.

C. Triggering the bug

The first and most important concern in this study is validating if bugs were actually triggered during the execution of our runs. While the bugs could be triggered during manual execution, *e.g.,* by reproducing the bug with the code from the pull requests, this does not mean that when processing data in a training process this would happen. This is due to multiple factors, but the most important is that PyTorch is a big framework with multiple "backends"; here, the key is building the right version, with the right build parameters to make sure that the bug is triggered. Indeed, if the bug was not triggered, this study only means to demonstrate the variability induced by NDIF even by controlling the seed, for within bug experiments. It can also be used to show the variance between multiple random seeds.

D. Model and dataset used

One might wonder why only a variation of VGG was used throughout this experiment. Indeed, there is no way of knowing during training if the workload contains the bug, as it is silent. Therefore, instead of using multiple models to accomplish the same task, we use the same model. The current model, VGG-X, is intended to work on the CIFAR-10 dataset: traditionally, the model works on the ImageNet [39] dataset, but as we wanted to reduce NDIF by reusing the same data split, we did not succeed in time to provide fast and deterministic serving of the ImageNet dataset.

We adapted the original VGG model (PyTorch's implementation in Torchvision [31]), to work on 32x32 input images. While this does not change the results for our study, comparisons were made for the same model, which achieves acceptable good performance (around 70-74%) considering that comparisons should be made only between buggy and corrected versions, our comparison holds. The model should achieve better performance considering its size. We believe that the adaptation of the model to fit the 32x32 input did in fact cause some non-optimal configuration for the network. There is one aspect of the network that caused some potential non-determinism: the dropout layer. Two dropout layers with activation probability of 0.5 were in the network: this could have affected the determinism of the computation. This falls under the NDIF of "Model definition" which is not studied in this paper.

E. Build system

We built versions of PyTorch using a common build process and using the default parameters for build execution. As most software are distributed prebuild through various channels, the upside from this process is that we reduce the possibility of introducing multiple build-related errors, as the recommended configurations reflect most widespread use of the framework. There are, of course, differences that can affect the final build artifacts from the ones we have built, notably because there might be some changes in the specific tooling used. Indeed, in the aim of reducing confounding variables, a single build system configuration was chosen to build all the versions with two variants on the Python version. Each variant represents a *Docker* container with the dependencies used to build PyTorch according to their guidelines. Nonetheless, both containers use `CUDA` API 10.1.

The methodology prescribes using a build system in order to build **buggy** and **corrected** versions. However, great care must be taken when doing so: in the case of PyTorch, there are multiple computational "backends". These are library dependencies that exist in order to execute mathematical operations allowing different hardware and software configurations to run PyTorch. The limitation comes from the fact that some parts of the library can only be used, *i.e.,* compiled with one of these "backends".

For reproducibility, the best way to build a framework such as PyTorch would be to statically link every dependency instead of using dynamic dependencies as this would mitigate improper or deviation of the methodology. Indeed, when statically linking an assembly, there would be no question of which runtime on the framework dependencies would be used. This would allow relaxation of the current constraint of running within the *Docker* container. Configuring static linking, however, is not trivial and would require a lot of effort.

VII. Conclusion and Future works

In this paper, we first introduced `ReproduceML`, a tool set to minimize the effects on non-determinism in ML experiments. It performs both random-seed control and data split control, a primitive form of data versioning. Then, we demonstrated an application of `ReproduceML` to measure the impact of any "change" in a ML framework by using the commits as the revision upon which a build process would be initiated. We also explored the various factors that influence reproducibility in this context and devised the methodology and artifacts used in order to enhance reproducibility in the study. Finally, we conducted an empirical case study on PyTorch, as a popular DL framework, by extracting, filtering, building and evaluating the artifacts. We have collected 737 bugs from PyTorch's repository, labelled 439, filtered 115, built 49 bugs (for their buggy and corrected version) and conducted our statistical analysis on 18 of them. Based on these numbers and our dataset, we can not conclude that our sample of bugs from PyTorch has an impact on model performance. We demonstrated an application of `ReproduceML` by analyzing the effect of bugs but its applicability is not limited to such scope. With the current tooling presented in this work, several other empirical research on ML configurations can be made where controlled ML experiments are inevitable. A very first one is investigating a more comprehensive set of bugs in ML frameworks to evaluate their impact on ML application. Moreover, one may study how the dependencies' versions or deprecated APIs in ML frameworks can affect the performance metrics of ML models.

References

[1] J. Gardner, Y. Yang, R. Baker, and C. Brooks, "Enabling end-to-end machine learning replicability: A case study in educational data mining," *arXiv preprint arXiv:1806.05208*, 2018.

[2] B. K. Olorisade, P. Brereton, and P. Andras, "Reproducibility in machine learning-based studies: An example of text mining," 2017.

[3] F. Doshi-Velez and B. Kim, "Towards a rigorous science of interpretable machine learning," *arXiv preprint arXiv:1702.08608*, 2017.

[4] V. Stodden and S. Miguez, "Best practices for computational science: Software infrastructure and environments for reproducible and extensible research," *SSRN Electronic Journal*, 2013.

[5] S. M. Crook, A. P. Davison, and H. E. Plesser, "Learning from the past: Approaches for reproducibility in computational neuroscience," in *20 Years of Computational Neuroscience*, pp. 73–102, Springer New York, 2013.

[6] S. R. Piccolo and M. B. Frampton, "Tools and techniques for computational reproducibility," *GigaScience*, vol. 5, no. 1, p. doi: s13742–016–0135–4, 2016.

[7] A. Davison, "Automated capture of experiment context for easier reproducibility in computational research," *Computing in Science & Engineering*, vol. 14, pp. 48–56, Jul 2012.

[8] V. Stodden, "Beyond open data: A model for linking digital artifacts to enable reproducibility of scientific claims," in *Proceedings of the 3rd International Workshop on Practical Reproducible Evaluation of Computer Systems*, ACM, Jun 2020.

[9] J. Vanschoren, H. Blockeel, B. Pfahringer, and G. Holmes, "Experiment databases," *Machine Learning*, vol. 87, pp. 127–158, Jan 2012.

[10] F. Z. Khan, S. Soiland-Reyes, R. O. Sinnott, A. Lonie, C. Goble, and M. R. Crusoe, "Sharing interoperable workflow provenance: A review of best practices and their practical application in CWLProv," *GigaScience*, vol. 8, Nov 2019.

[11] S. Cohen-Boulakia, K. Belhajjame, O. Collin, J. Chopard, C. Froidevaux, A. Gaignard, K. Hinsen, P. Larmande, Y. L. Bras, F. Lemoine, F. Mareuil, H. Ménager, C. Pradal, and C. Blanchet, "Scientific workflows for computational reproducibility in the life sciences: Status, challenges and opportunities," *Future Generation Computer Systems*, vol. 75, pp. 284 – 298, 2017.

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

[13] C. Liu, C. Gao, X. Xia, D. Lo, J. Grundy, and X. Yang, "On the replicability and reproducibility of deep learning in software engineering," *arXiv preprint arXiv:2006.14244*, 2020.

[14] O. E. Gundersen and S. Kjensmo, "State of the art: Reproducibility in artificial intelligence.," in *AAAI*, pp. 1644–1651, 2018.

[15] F. Renard, S. Guedria, N. D. Palma, and N. Vuillerme, "Variability and reproducibility in deep learning for medical image segmentation," *Scientific Reports*, vol. 10, Aug 2020.

[16] T. Raeder, T. R. Hoens, and N. V. Chawla, "Consequences of variability in classifier performance estimates," in *2010 IEEE International Conference on Data Mining*, IEEE, Dec 2010.

[17] S. Jean-Paul, T. Elseify, I. Obeid, and J. Picone, "Issues in the reproducibility of deep learning results," in *2019 IEEE Signal Processing in Medicine and Biology Symposium (SPMB)*, pp. 1–4, IEEE, IEEE, Dec 2019.

[18] K. Khetarpal, Z. Ahmed, A. Cianflone, R. Islam, and J. Pineau, "Re-evaluate: Reproducibility in evaluating reinforcement learning algorithms," 2018.

[19] M. Crane, "Questionable answers in question answering research: Reproducibility and variability of published results," *Transactions of the Association for Computational Linguistics*, vol. 6, pp. 241–252, Dec 2018.

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

[21] A. Labach, H. Salehinejad, and S. Valaee, "Survey of dropout methods for deep neural networks," *arXiv preprint arXiv:1904.13310*, 2019.

[22] Q. Meng, W. Chen, Y. Wang, Z.-M. Ma, and T.-Y. Liu, "Convergence analysis of distributed stochastic gradient descent with shuffling," *Neurocomputing*, vol. 337, pp. 46–57, Apr 2019.

[23] L. Jia, H. Zhong, X. Wang, L. Huang, and X. Lu, "An empirical study on bugs inside tensorflow," in *International Conference on Database Systems for Advanced Applications*, pp. 604–620, Springer, 2020.

[24] "Artifact review and badging-version 1.1." ACM, https://www.acm.org/publications/policies/artifact-review-and-badging-current, 2020. Accessed: 2020-09-20.

[25] J. Vanschoren, J. N. van Rijn, B. Bischl, and L. Torgo, "Openml: Networked science in machine learning," *SIGKDD Explor. Newsl.*, vol. 15, p. 49–60, June 2014.

[26] M. Zaharia, A. Chen, A. Davidson, A. Ghodsi, S. A. Hong, A. Konwinski, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe, *et al.*, "Accelerating the machine learning lifecycle with mlflow.," *IEEE Data Eng. Bull.*, vol. 41, no. 4, pp. 39–45, 2018.

[27] R. Isdahl and O. E. Gundersen, "Out-of-the-box reproducibility: A survey of machine learning platforms," in *2019 15th International Conference on eScience (eScience)*, pp. 86–95, IEEE, Sep 2019.

[28] "DVC documentation." https://dvc.org/doc, 2020. Accessed: 2020-10-27.

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

[30] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[31] "pytorch/vision: Datasets, transforms and models specific to computer vision." https://github.com/pytorch/vision, 2017. Accessed: 2020-11-14.

[32] R. Bergmann, J. Ludbrook, and W. P. J. M. Spooren, "Different outcomes of the wilcoxon—mann—whitney test from different statistics packages," *The American Statistician*, vol. 54, pp. 72–77, Feb 2000.

[33] D. Merkel, "Docker: Lightweight linux containers for consistent development and deployment," *Linux J.*, vol. 2014, Mar. 2014.

[34] A. Krizhevsky, G. Hinton, *et al.*, "The cifar-10 dataset."

[35] P. Xu, S. Shi, and X. Chu, "Performance evaluation of deep learning tools in docker containers," in *2017 3rd International Conference on Big Data Computing and Communications (BIGCOM)*, IEEE, Aug 2017.

[36] Torch Contributors, "Reproducibility - pytorch 1.6.0 documentation." https://pytorch.org/docs/1.6.0/notes/randomness.html, 2020. Accessed: 2020-07-28.

[37] Torch Contributors, "Reproducibility - pytorch master documentation." https://pytorch.org/docs/1.1.0/notes/randomness.html, 2019. Accessed: 2020-07-28.

[38] U. Köster, T. J. Webb, X. Wang, M. Nassar, A. K. Bansal, W. H. Constable, O. H. Elibol, S. Gray, S. Hall, L. Hornof, *et al.*, "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," *arXiv preprint arXiv:1711.02213*, 2017.

[39] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.