# Improving the Reliability of Deep Learning Software Systems

by

Viet Hung Pham

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2022

**Examining Committee Membership**

The following served on the Examining Committee for this thesis. The decision of the Examining Committee is by majority vote.

| | |
|---|---|
| External Examiner: | Baishakhi Ray<br>Professor, Dept. of Computer Science,<br>Columbia University |
| Supervisor(s): | Lin Tan<br>Professor, Electrical & Computer Engineering,<br>University of Waterloo |
| | Yaoliang Yu<br>Professor, David R. Cheriton School of Computer Science,<br>University of Waterloo |
| Internal Member: | Jimmy Lin<br>Professor, David R. Cheriton School of Computer Science,<br>University of Waterloo |
| | Chengnian Sun<br>Professor, David R. Cheriton School of Computer Science,<br>University of Waterloo |
| Internal-External Member: | Patrick Lam<br>Professor, Electrical & Computer Engineering,<br>University of Waterloo |

## Author's Declaration

## Statement of Contributions

As the lead author of all the contributions of this thesis I was responsible for (1) creating the study concepts, (2) developing, implementing, and evaluating prototypes, (3) performing data collection and analysis, (4) drafting and submitting manuscripts, and (5) presenting the work at conferences. My coauthors provided feedback at each step of the research process and on manuscript drafts.

**Research presented in Chapter 3**: Dr. Thibaud Lutellier helped revising the manuscripts and double-checking related work, Weizhen Qi helped verifying the detected bugs. Carmen Kwan helped collecting evaluation models from GitHub, and Yitong Li helped reproducing and validating the experimental results. Dr. Lin Tan provided feedback and supervision in all parts of the research.

**Research presented in Chapter 4**: Shangshu Qian helped designing, performing, and analyzing the developers and researchers survey. Jiannan Wang helped producing some figures for the manuscripts. Dr. Thibaud Lutellier helped designing and organizing the paper survey. Jonathan Rosenthal helped double-checking related work. Shangshu Qian, Jiannan Wang, Dr. Thibaud Lutellier, Jonathan Rosenthal, and Yitong Li helped examining research paper for the paper survey. Dr. Eric Horvitz, Dr. Benjamin Zorn, and Dr. Madan Musuvathi provided feedback on the manuscript. Dr. Mijung Kim helped code the tool DEVIATE. Dr. Lin Tan, Dr. Yaoliang Yu and Dr. Nachiappan Nagappan provided feedback and supervision in all parts of the research.

**Research presented in Chapter 5**: Dr. Todd Mytkowicz helped debugging the experimental code to run on the cluster. Dr. Ece Kamar, Dr. Emre Kiciman, Dr. Lin Tan, and Dr. Benjamin Zorn provided feedback and supervision in all parts of the research.

**Citations:**

- Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, Lin Tan, *CRADLE: Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries*, Proceeding of The 41st ACM/IEEE International Conference on Software Engineering, 2019 [182].
- Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, Nachiappan Nagappan, *Problems and opportunities in training deep learning software systems: an analysis of variance*, Proceeding of The 35th IEEE/ACM International Conference on Automated Software Engineering, 2020 (ACM SIGSOFT Distinguished Paper Award) [183].

- Hung Viet Pham, Mijung Kim, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan, *DEVIATE: A Deep Learning Variance Testing Framework*, Proceeding of The 36th IEEE/ACM International Conference on Automated Software Engineering, 2021 [181].

## Abstract

For the last decade, deep learning (DL) has emerged as a new effective machine learning approach that is capable of solving difficult challenges. Due to their increasing effectiveness, DL approaches have been applied widely in commercial products such as social media platforms and self-driving cars. Such widespread application in critical areas means that mistakes caused by bugs in such DL systems would lead to serious consequences. Our research focuses on improving the reliability of such DL systems.

At a high level, the DL systems development process starts with labeled data. This data is then used to train the DL model with some training methods. Once the model is trained, it can be used to create predictions for some unlabeled data in the inference stage. In this thesis, we present testing and analysis techniques that help improve the DL system reliability for all stages.

In the first work, CRADLE, we improve the reliability of the DL system inference by applying differential testing to find bugs in DL libraries. One key challenge of testing DL libraries is the difficulty of knowing the expected output of DL libraries given an input instance. We leverage equivalent DL libraries to overcome this challenge. CRADLE focuses on finding and localizing bugs in DL software libraries by performing cross-implementation inconsistency checking to detect bugs, and leveraging anomaly propagation tracking and analysis to localize faulty functions that cause the bugs. CRADLE detects 12 bugs in three libraries (TensorFlow, CNTK, and Theano), and highlights functions relevant to the causes of inconsistencies for all 104 unique inconsistencies.

Our second work is the first to study the variance of DL systems training and the awareness of this variance among researchers and practitioners. Our experiments show large overall accuracy differences among identical training runs. Even after excluding weak models, the accuracy difference is 10.8%. In addition, implementation-level factors alone cause the accuracy difference across identical training runs to be up to 2.9%. Our researcher and practitioner survey shows that 83.8% of the 901 participants are unaware of or unsure about any implementation-level variance. This work raises awareness of DL training variance and directs SE researchers to challenging tasks such as creating deterministic DL implementations to facilitate debugging and improving the reproducibility of DL software and results.

DL systems perform well on static test sets coming from the same distribution as training sets but may not be robust in real-world deployments because of the fundamental assumption that the training data represents the real- world data well. In cases where the training data misses samples from the real-world distribution, it is said to contain blindspots. In practice, it is more likely a training dataset contains weakspots (i.e., a weaker form of blindspots, where the training data contains some samples that represent the real world but it does not contain enough). In the third work, we propose a new procedure to detect weakspots in training data and to improve the DL system with minimum labeling effort. This procedure leverages the variance of the DL training process to detect highly varying data samples that could indicate the weakspots. Metrics that measure such variance can also be used to rank new samples to prioritize the labeling of additional training data that can improve the DL system accuracy when applied to the real world. Our evaluation shows that, in scenarios where the weakspots are severe, our procedure improves the model accuracy on weakspot samples by 25.2% requiring 2% of additional training data. This is an improvement of 4.5 percentage points compared to the traditional single model metric with the same amount of additional training data.

## Acknowledgements

I would like to particularly thank my supervisor, Dr. Lin Tan: "Thank you for accepting me into your group, helping and supporting me during my Ph.D study. I learned and grew so much under your supervision and mentorship." I also would like to thank my other supervisor, Dr Yaoliang Yu: "Thank you for giving me advice and supporting me during my Ph.D study!" I could not have wished for better supervisors!

I would like to thank my examination committee members: Dr. Baishakhi Ray, Dr. Jimmy Lin, Dr. Chengnian Sun, Dr. Patrick Lam, and Dr. Meiyappan Nagappan—for their support, insightful discussion and valuable feedback on my research.

Thanks to all the students in our research group at the University of Waterloo who made going to the lab every day a pleasure: Dr. Song Wang, Dr. Edmund Wong, Dr. Thibaud Lutellier, Yuan Xi, Moshi Wei, Yitong Li, and Tejvinder Toor—thank you very much for the support, friendship, and joy.

I also want to thank the research team at Purdue University. While I have not met anyone in person, we had great online conversations and I learned a lot from them: Shangshu Qian, Danning Xie, Jiannan Wang, Dr. Mijung Kim, and Jonathan Rosenthal.

I want to thank all my other collaborators during my Ph.D. study: Dr. Nachiappan Nagappan, Dr. Benjamin Zorn, Dr. Ece Kamar, Dr. Emre Kiciman, Dr. Todd Mytkowicz, Dr. Jiahao Chen, Dr. Sameena Shah, Dr. Xiangyu Zhang, Dr. Michael W. Godfrey, Dr. Tung Nguyen, Dr. Tam The Nguyen, Dr. Yanghee Kim, Dr. Sherry Marx, Zeou Hu, Jungwon Kim, Weizhen Qi, and Phong Minh Vu. I really enjoyed working with them, and have learned a lot from them.

## Dedication

This is dedicated to my parents, my wife and my children. Thank you all for always being there for me, through thick and thin!

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

For the last decade, Deep learning (DL) has emerged as a new effective machine learning approach that is capable of solving difficult challenges such as image processing [55], speech recognition [105], and natural language processing [98, 222]. Due to their increasing effectiveness, DL approaches have been applied widely in many domains, including aircraft collision avoidance systems [117], Alzheimer's disease diagnosis [149], diabetic blood glucose prediction [160], autonomous driving cars [43], romance storytelling [121, 122], and software engineering [39, 44, 45, 102, 131, 189, 250, 255, 257] Bugs in such systems can cause disastrous consequences, e.g., a software bug in Uber's self-driving car DL system has resulted in the death of a pedestrian [73].

**Thesis statement:** *The work in this thesis aim to* ***improve the reliability of deep learning software systems*** *implementations and applications for all stages of development* ***from the software engineering perspective****.*

Figure 1.1 shows the high-level view of DL system development which starts with ***labeled data***. This data is then used to train the DL model with some ***training*** methods. Once the DL model is trained, it can be used to create predictions for some unlabeled data in the ***inference*** stage. In this thesis, we present testing and analysis techniques that help improve the DL system reliability for all stages: *detecting bugs in the inference stage, analyzing the variance of DL training, and detecting and mitigating weakspots in DL training data.*

In the first work, CRADLE, we improve the reliability of the DL system during inference by applying differential testing to find bugs in DL libraries (Chapter 3). In an effort to apply differential testing to the training process, our baseline analysis reveal a large variance in the training process. This led to our second work, which is the first to study the variance of

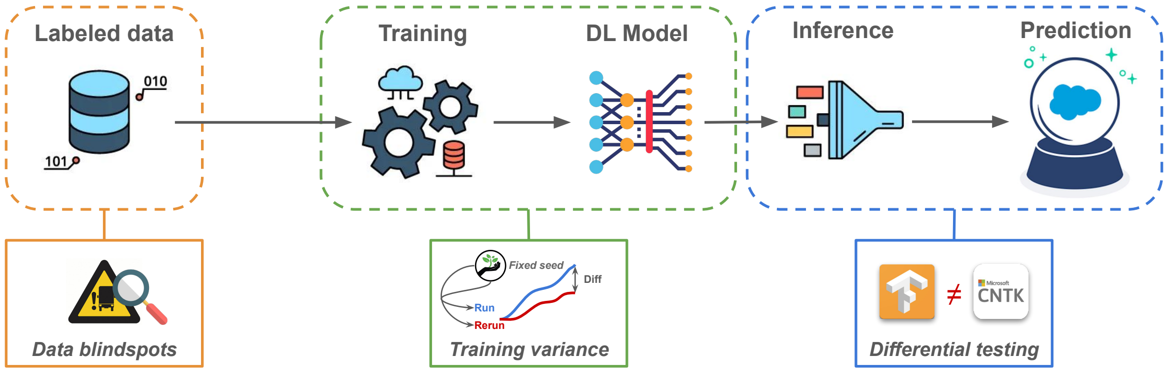


Figure 1.1: The overview of the three stages (labeled data, training, and inference) of deep learning development and the contributions of the thesis on all three of these stages.

DL systems training and the awareness of this variance among researchers and practitioners (Chapter 4). The insight from the variance analysis inspire our third work, in which we propose a new procedure to detect weakspots in training data and to improve the DL system with minimum labeling effort (Chapter 5).

The work in this thesis detects 12 bugs in popular DL libraries (TensorFlow, CNTK, and Theano), raises awareness of large DL training variance caused by software implementation alone, and improves model trained with data containing weakspots by up to 25.2% with only 2% of additional data.

## 1.1 Differential Testing at the Inference Stage

In this work, we apply differential testing to find bugs in the inference code of DL libraries. One key challenge of testing DL system is the difficulty to know the expected output given an input instance. The multiple DL library implementations of the same functionality provide us with a unique opportunity to apply differential testing to detect inconsistencies and then find bugs in DL libraries.

To automatically detect and localize such inconsistencies across DL backends, we propose and implement a novel approach—*CRADLE*. Given a DL model and its input data, CRADLE (1) uses two distance metrics to compare the output of a model on different backends to detect inconsistent output, and (2) identifies the location of the inconsistency by tracking the anomaly propagation through the execution graph. By identifying the spike in the magnitude of the difference between two backends, CRADLE points out the inconsistent functions in the backend that introduces the inconsistency, which should be very useful for developers understand the bug and debug it.

Our results show that CRADLE detects 12 bugs (9 have been fixed by developers) in DL software that cause inconsistencies for 28 out of 30 models, 3 of which are previously unknown bugs, 2 of which have already been confirmed by developers. CRADLE highlights functions relevant to the causes of inconsistencies for all 104 unique inconsistencies. CRADLE's median end-to-end running time is less than 5 minutes, suggesting that CRADLE is practical.

**Summary:** This work supports the thesis by proposing **a new differential testing approach to detect and localize bugs in DL libraries**. This technique detects 12 bugs in DL software that cause inconsistencies for 28 out of 30 models and highlights functions relevant to the causes of inconsistencies for all 104 unique inconsistencies.

## 1.2 An Analysis of Variance of Deep Learning Software Systems Training

In an attempt to apply differential testing to the training process of DL systems, we first measure the variance of such a process as a baseline. The variance is much larger than expected and our developer survey shows a lack of awareness of such variance among the communities. Thus, this work performs an analysis of the training variance of DL systems to raise awareness in the communities.

Specifically, DL training algorithms utilize nondeterminism to improve training efficiency and model accuracy. These *nondeterminism-introducing (NI)-factors* cause multiple *identical training runs*, i.e., training runs with the *same* settings (e.g., identical training data, identical algorithm, and identical network), to produce different DL models with significantly different accuracies and training times [120, 195, 221, 229]. One can eliminate the variance introduced by *algorithmic NI-factors* using fixed random seeds.

However, in addition to these algorithmic NI-factors, DL libraries (e.g., TensorFlow [26] and cuDNN [48]) introduce additional variance. For example, core DL libraries, by default, leverage *autotune* to automatically benchmark several modes of operation (i.e., underlying algorithms for computations such as addition) for each primitive cuDNN function (e.g., pooling and normalization) in its first call. The fastest mode of operation is then used for that cuDNN function in subsequent calls. Different identical runs sometimes use different modes of operation (due to different benchmark results), introducing variance. These *implementation-level NI-factors* alone can cause an overall accuracy difference of up to 2.9%.

These implementation-level NI-factors and differences are often consequences of the DL *software implementations*. DL researchers have paid little attention to implementation-level NI-factors. Their focus is on theoretical analyses of DL training [51, 52, 68, 123, 142, 198].

To see whether such differences are known, we conduct a survey, which surprisingly shows that 83.8% of the 901 responding researchers and practitioners with DL experience are unaware of (63.4%) or unsure about (20.4%) any implementation-level variance! Of the 901 respondents, only 10.4% expect 2% or more accuracy difference across fixed-seed identical training runs. We also perform a literature survey of 454 papers *randomly* sampled from recent top *SE*, *AI*, and *systems* conferences to understand the awareness and practices of handling DL system variance in research papers. Of 225 papers that train and evaluate DL systems, only 19% use multiple identical training runs to quantify the variance of their DL approaches.

The analysis and surveys raise awareness of DL training variance to improve the reproducibility of DL software and results. However, we also contribute by developing a tool DEVIATE that (1) consistently measures variance with minimal user efforts, and (2) provides appropriate statistical tests. DEVIATE replicates multiple training runs with little to no input from the users. DEVIATE *automatically* analyzes the DL system source code, extracts important metrics (such as accuracy, loss...) that are monitored by the authors, and stores those metrics in a consistent format. DEVIATE performs popular statistical tests (such as confidence interval of standard deviation, MannWhitney U-test, and Cohen's $d$ effect size) and provides users with a report of how well the proposed technique performs in comparison to the selected baselines.

**Summary:** This work supports the thesis by performing **the first study of the variance of DL systems training caused by the implementation and the awareness of this variance among researchers and practitioners**. This study shows that the implementation-level factors alone cause the accuracy difference across identical training runs to be up to 2.9% and the survey shows that 83.8% of the 901 participants are unaware of or unsure about any implementation-level variance. The publication of the study and surveys raises awareness of DL training variance while the tool improves the reproducibility of DL software and results.

## 1.3 Mitigating Blindspot in Deep Learning Training Data

Since DL is data-driven, it suffers similar problems to other data-driven machine learning techniques. One such problem is that the training dataset is not representative of the real-world application. In cases where the training data misses samples from the real world distribution, it is said to contain blindspots. In practice, it is more likely a training dataset contains weakspots (i.e., a weaker form of blindspots, where the training data contains some samples that represent the real world but, not enough).

We propose a new procedure to detect weakspots in training data and improve the DL system with minimum labeling effort. This procedure leverages the variance of the DL training process to detect highly varying data samples that could indicate weakspots. This procedure is not reliant on strategic dropout layers or specific neural network architecture (e.g., convolutional neural network, recurrent neural network, reinforcement learning) [81], and thus can be applied to any deep learning model as a black box method. Metrics that measure such variance can also be used to rank new samples to prioritize the labeling of additional training data that can improve the DL system accuracy when applied to the real world. Our evaluation shows that, in scenarios where the weakspots are severe, our procedure improves the model accuracy on weakspot samples by 25.2% requiring 2% of additional training data. This is an improvement of 4.5 percentage points compared to the traditional single model metric with the same amount of additional training data. Our qualitative analysis shows that by inspecting the unseen samples with high uncertainty across multiple models with the help of attention visualization methods such as Grad-CAM [201], the developers could gather insights into why the model fails to generalize, and how to improve the model.

**Summary:** This work supports the thesis by proposing **a new procedure to detect and mitigate blindspots in deep learning training data**. This new procedure improves the holdout test accuracy by up to 25.2% with only 2% added training data. Our qualitative analysis shows that inspecting the unseen samples with high uncertainty across multiple models could help developers to gather insights into why the model fails to generalize and how to improve the model.

## 1.4 Publications

Earlier versions of the work presented in this thesis have been published in the following papers (listed in chronological order). In these studies, I proposed novel ideas, collected data, conducted experiments, analysed the results, and completed the manuscript.

- Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, Lin Tan, *CRADLE: Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries*, Proceeding of The 41st ACM/IEEE International Conference on Software Engineering, 2019 [182].
- Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, Nachiappan Nagappan, *Problems and opportunities in training deep learning software systems: an analysis of variance*, Proceeding of The 35th IEEE/ACM International Conference on Automated Software Engineering, 2020 (ACM SIGSOFT Distinguished Paper Award) [183].
- Hung Viet Pham, Mijung Kim, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan, *DEVIATE: A Deep Learning Variance Testing Framework*, Proceeding of The 36th IEEE/ACM International Conference on Automated Software Engineering, 2021 [181].

The following papers (listed in chronological order) were published as the follow up work to the above mentioned publications. These studies explore other software engineering approach or aspect to improve the reliability of DL software systems. As a coauthor I gave feedback at each step of the research process and helped implementing the experiments and revising the manuscripts.

- Jiannan Wang, Thibaud Lutellier, Shangshu Qian, **Hung Viet Pham**, and Lin Tan, *EAGLE: Creating Equivalent Graphs to Test Deep Learning Libraries*, Proceeding of The 44th ACM/IEEE International Conference on Software Engineering, 2022 [236].
- Shangshu Qian, **Hung Viet Pham**, Thibaud Lutellier, Zeou Hu, Jungwon Kim, Lin Tan, Yaoliang Yu, Jiahao Chen, and Sameena Shah, *Are My Deep Learning Systems Fair? An Empirical Study of Fixed-Seed Training*, Proceeding of The 35th Conference on Neural Information Processing Systems, 2021 [186].

The following papers (listed in chronological order) were published in parallel to the above mentioned publications. These studies are not directly related to this thesis, but explore the usages of machine learning to improve the reliability of DL and general software.

- Danning Xie, Yitong Li, Mijung Kim, **Hung Viet Pham**, Lin Tan, Xiangyu Zhang, Michael W. Godfrey, *DocTer: Documentation-Guided Fuzzing for Testing Deep Learning API Functions*, Proceeding of The ACM SIGSOFT International Symposium on Software Testing and Analysis, 2022 [240].

- Thibaud Lutellier, **Hung Viet Pham**, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan, *CoCoNuT: Combining Context-Aware Neural Translation Models Using Ensemble for Program Repair*, Proceeding of The ACM SIGSOFT International Symposium on Software Testing and Analysis, 2020 [150].

# Chapter 2

# Related Work

## 2.1 Machine Learning and Deep Learning System Testing and Debugging

### 2.1.1 Testing Machine Learning (ML) Libraries

Recently, automatic testing of ML libraries is an active area of research [70, 72, 218]. Srisakaokul et al. [218] detect inconsistencies between multiple implementations of common ML algorithms (i.e., kNN or Naive Bayes (NB)). This approach uses majority votes to estimate the expected output. However, it requires many implementations of the same algorithm (19 kNNs and 7 NBs used) with the assumption that most of them are correctly implemented. In contrast, CRADLE performs pairwise comparisons, which as shown by our experiments, detects inconsistencies without knowing the expected output and works with a minimum of two implementations. Another major difference is that Srisakaokul et al. define deviation based on the inconsistency of top-1 classifications without comparing them to the ground truth. CRADLE, on the other hand, define inconsistency as deviations in predicted ranks of the ground-truth label because we want to focus on inconsistent implementations that affect the performance of DL models on real world validation dataset. Dwarakanath et al. [72] test ML libraries by applying transformations on the training and testing data to detect inconsistencies. However, they were only able to identify artificially injected bugs. Dutta et al. [70] used fuzzing to test probabilistic programming systems. None of these techniques performs error localization.

### 2.1.2 Benchmarking DL Libraries

Liu et al. [147] observe that the same DL algorithm with identical configurations, such as training set and learning rate, produce different execution time and accuracy when trained with different low-level DL libraries. However, that work aims to benchmark DL libraries, not to detect or localize inconsistency bugs, as it does not compare the exact same model on different backends. Since each model is re-trained on each backend and the training process contains non-determinism (e.g., the seed for the optimization function), small accuracy differences are expected. DL libraries have been compared in the literature [58, 71, 207, 208, 211]. However, the prior work focuses on performance comparison only and does not detect or localize non-performance bugs in DL libraries.

### 2.1.3 Adversarial Testing of DL Models

Much recent work focuses on testing DL models [84,109,118,129,151,167,168,174,179,225, 231, 239]. Many techniques generate adversarial examples [129, 167, 168, 174, 225]. Some work [109, 118, 239] verifies DL software. DeepXplore [179] introduces neuron coverage to measure testing coverage in CNN models. These approaches are orthogonal to our work as they test the correctness of DL models, while we test the correctness of the implementations of models in the DL software libraries.

### 2.1.4 Differential Testing and Inconsistency Detection

Differential testing [158] consists of testing whether different compilers produce the same results. Much work uses differential testing to find bugs in compilers by comparing the output of multiple compilers [210, 244, 245] or different compiler optimization levels [134, 244]. Inconsistency detection has been used in other domains such as cross-platform [77, 115], web browsers [53, 190–192] or document readers [128]. Our work is a new application of differential testing and inconsistency detection for DL software, which has its unique challenges such as identifying bug-triggering inconsistencies (Section 3.3.1). In addition, we localize the inconsistencies to the faulty functions.

### 2.1.5 Debugging and Fault Localization

We are not aware of prior work that localizes inconsistency bugs in DL libraries, despite the large volume of debugging and fault localization work for general software bugs [107, 114, 161,162,166,173,178,248]. While these approaches could be used to debug DL networks, applying such techniques to localize faulty functions in DL networks may have unique challenges such as scalability, and hence may remain as future work.

## 2.2 Variance of Deep Learning Training Process

### 2.2.1 Variance Study of Reinforcement Learning (RL)

Closest to our study of DL training variance is the study [165] that measures the impact of some implementation-level NI-factors on RL experiments. We study the general DL variance, while they focus on RL variance only. In addition, we measure the awareness by conducting a survey and a literature review. Furthermore, while they focus on one network for one task (i.e., playing the Atari game Breakout) and one version of one core library (PyTorch), we study the impact of nondeterminism-introducing factors(NI-factors) using 6 networks trained on 3 datasets and multiple versions of three core libraries. Papers [57,103] that investigates the impact of random seeds on RL are different from ours, since they only consider the impact of random seeds (i.e., algorithmic NI-factors), while we also study the variance caused by implementation-level NI-factors.

### 2.2.2 Awareness of The Impact of Nondeterminism

A recent literature survey [171] on 30 papers on the topic of text mining confirms the results of our literature survey. None of the 30 investigated papers report using different random seeds. Our literature survey investigates DL training in general (not just text mining papers) and examines 225 papers. Furthermore, our overall contribution is different since we also quantify the differences in training accuracy and time across identical training runs. Another paper [164] states that small changes in the experimental setup can generate measurement bias. It focuses on standard CPU computation benchmark [59] and does not study the nonderterminism of DL systems.

### 2.2.3 Anecdotal Evidence of NI-factors

Some studies [124,155,159,234] quantify the variance in their results caused by NI-factors. However, these only provide anecdotal evidence and none attempt to systematically study the variance introduced by algorithmic and implementation-level NI-factors. In addition, our surveys show that awareness is still very low in the research community.

### 2.2.4 Nondeterminism in Stochastic Gradient Descent (SGD)

Much work investigates variance caused by SGD [51, 61, 68, 123, 142]. While these papers quantify nondeterminism caused by SGD, they ignore all other sources of nondeterminism described in Section 4.2.

### 2.2.5 Impact of Weight Initialization

Prior work [120, 195, 221, 229] measures the impact of different initial weights on models' training time. While such an issue is known, implementation-level NI-factors have not been studied and our surveys show that they are often not considered when evaluating DL systems.

### 2.2.6 Controlling Implementation-Level Nondeterminism

Jooybar et al. [116] propose a hardware mechanism to introduce determinism in GPU-based algorithms. In our work, we do not focus on the hardware itself and measure the variance caused by NI-factors using popular GPUs without special hardware modifications.

### 2.2.7 Deep Learning System Benchmarking

Much work focuses on benchmarking DL systems. However, their target is finding the best networks [58,185,259], hardware [207,259], hyper-parameters [147], and framework [96,125, 147, 208, 212, 259]. Such approaches do not consider the impact of NI-factors on multiple identical training runs.

## 2.3 Blindspots in Machine Learning Training Data

### 2.3.1 Blindspots of Machine Learning Systems

Machine learning systems' blindspots are caused by the dataset shift [112,187]. Approaches have been proposed to discover dataset shift, including using guided exploration with feedback from an oracle [132], or harnessing human input [33]. However, these approaches all work with a single model which have the problem of unreliable uncertainty measure.

There has been other approaches that address the dataset shift, including weighting of training instances [213], online learning of prediction models [41], and learning models robust to adversarial actions [62, 94, 228]. These approaches either assume some access to the model, availability of original data, or the model can be adaptively retrained.

### 2.3.2 Out-of-distribution detection

Out-of-distribution (OOD) detection is an active area of research that focuses on exploring techniques that can detect OOD test samples that were not seen during training. Specifically, there are two distribution shift settings that OOD literature focuses on the covariate shift and semantic shift [242].

At a high level, our N-model metrics are designed to detect OOD samples affected by covariate shifts. Prior work in the sensory anomaly detection category [29, 32, 36, 54, 67, 113, 170, 176, 184, 238, 243] is designed to detect covariate shifts. However, since our N-model metrics target the specific subpopulation shift, it differs from prior sensory anomaly detection work which detects perturbations and small defects in adversarial defense [196], forgery recognition of biometrics and artworks [170, 176, 184, 238], image forensics [67, 113, 243], and industrial inspection [32, 36, 54].

Outlier detection [28, 35, 106] can also detect covariate shifts. But, in contrast with our N-model work—whose in-distribution is defined using the training data, outlier detection requires all observations and defines "in-distribution"as the majority of those observations. This requirement cannot be met in our open world setting where it is not possible to obtain all observations.

Much prior work such as semantic anomaly detection [64, 110, 143], novelty detection [156, 157, 172, 193], open set recognition [63, 76, 148, 199, 217, 241], and OOD detection [85, 104, 108] focuses on semantic shift detection, where the OOD datasets (i.e., OOD test samples) in their benchmark do not have label overlapping with the ID dataset (i.e., training samples). This problem setting differs from our subpopulation shift setting where the test data contains the same set of labels as the training data.

### 2.3.3 Active Learning

Active learning research has proposed various approaches for selectively learning about known unknowns during training (e.g., uncertainty sampling [141, 204], query by committee [206], expected model change [205], expected error reduction [261], expected variance reduction [253]) However, traditional active learning approaches depend on using a single model uncertainty measurement which could be unreliable.

Recently Deep Active Learning has been proposed to select known unknowns during training. However they are also leverage a single model approach [31,101,169,188]. Gal et al [81] propose an estimation approach using dropout layers and Monte Carlo sampling to estimate the uncertainty information of Bayesian convolutional neural networks (Bayesian CNN). However this technique is limited to the use of dropout layers placed at strategic locations, which could reduce test accuracy when applied to some network architectures.

# Chapter 3

# CRADLE: Cross-Backend Validation to Detect and Localize Bugs in Deep Learning Libraries

## 3.1 Motivation

Deep learning (DL) is widely used in many domains, including aircraft collision avoidance systems [117], Alzheimer's disease diagnosis [149], autonomous driving cars [43], and romance storytelling [121, 122]. Bugs in such systems can cause disastrous consequences, e.g., a software bug in Uber's self-driving car DL system has resulted in the death of a pedestrian [73].

Users of DL systems have a diverse range of background, including people with little technical backgrounds, e.g., singers/songwriters have used DL to compose music [8]. The pervasive use of DL systems requires them to be highly reliable.

Unfortunately, DL algorithms are complex to understand and use. Average users do not know all the details of DL algorithms. High-level DL Application Programming Interfaces (APIs) have been developed to enable users to build DL systems without knowledge of the inner working of neural networks. These high-level APIs rely on lower-level libraries that implement DL algorithms.

Figure 3.1 presents the structure of typical DL libraries. Developers write code using high-level library APIs (e.g., Keras [49] API). These APIs invoke low-level libraries that implement specific DL algorithms. Low-level libraries such as TensorFlow (Google) [26],

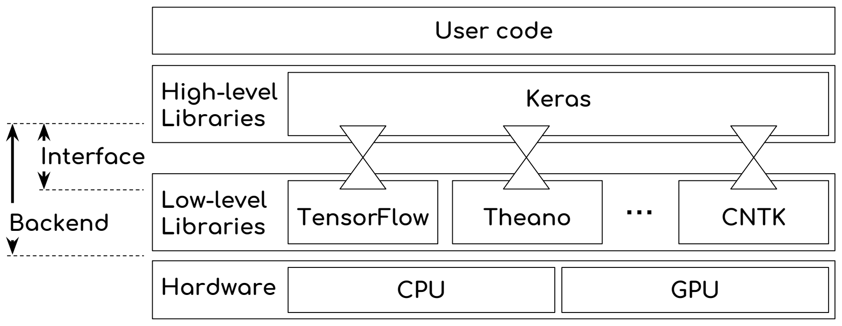


Figure 3.1: Overview of DL libraries.

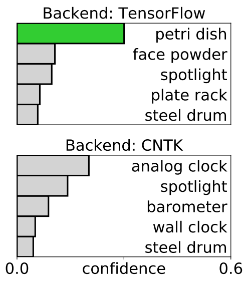
Theano [37], and CNTK (Microsoft) [200], implement the same algorithms, e.g., convolutional neural network (CNN) and recurrent neural network (RNN). Low-level libraries use different input formats and provide different APIs, while a high-level library allows users to seamlessly switch among different low-level libraries. The components that invoke low-level libraries are referred to as the *interfaces* between the high-level libraries and the low-level libraries. Each interface and low-level library, referred to as a *backend*, provides an implementation of DL algorithms. The backend trains and tests DL models. A DL *model* contains a DL network and parameters (also known as *weights*).

Keras [49] is the most popular high-level library for deep learning [10]. Keras has been used to implement neural networks in critical domains, including aircraft collision avoidance systems [117], inflammatory bowel disease diagnosis [79], chemical reaction predictions [130], medical imaging [42, 230], air quality control [80] and computer network security [65].

Like any software, DL backends and high-level libraries contain bugs, which are particularly challenging to find and fix [147, 179]. One key challenge is that it is difficult for developers to know the *expected output* given an input instance. DL backends implement DL models that use complex networks and mathematical formula. Thus, it is hard for humans to produce the expected output of a DL backend given an arbitrary input instance, if possible at all. For example, given an input image of digit '1' (*ground truth* '1'), and a digit classification model, the expected output of that model on that image is not necessarily '1', as it is common for a model to misclassify due to its limitations (100% classification accuracy is rarely achieved). Existing DL testing work [129, 167, 168, 174, 179, 225] focuses on generating input instances that make the ground truth and the model output disagree so that DL users and builders can improve the model.

Models must be implemented by backend libraries. If the backend libraries fail to faith-

(a) Input image "Petri dish"



(b) Top-5 InceptionResNetV2

```
-  return (x-mean)/(C.sqrt(var)+epsilon)*
     gamma+beta
+  return (x-mean)/ C.sqrt(var +epsilon)*
     gamma+beta
```

(c) Bug fix in `batch_normalization` in the CNTK backend.

Figure 3.2: A bug found by CRADLE in the CNTK backend, which has been fixed after we reported it.

fully implement a model (e.g., due to a bug in the backend), the output from the backend can be wrong even if the model is correct, and vice versa. An incorrectly-implemented DL backend may cause the aforementioned digit classification model to output '9' for the same image of '1', even if the expected output of the DL model is '7'. Alternatively, the DL backend may output '1' accidentally matching the ground truth. The wrong outputs could mislead DL users and builders in their debugging and fixing process. The output masks the implementation bug, which makes it challenging to be detected.

There has been little attention to testing the correctness of the models' *implementations*. Instead many techniques [129, 167, 168, 174, 179, 225] test the correctness of the *models* assuming that the backend implementation is correct. Both the model and the backend implementation need to be correct for DL algorithms to produce a correct output. The critically important task of testing DL backend implementation is challenging since the expected output of the backend is hard to obtain.

The multiple implementations (i.e., the DL backends) of the same functionality (i.e., the same DL algorithm) provide us a unique opportunity to detect inconsistencies among these implementations to find bugs in DL backend libraries. For example, if the *same* CNN model—which is the same CNN network with identical weights—behaves differently when running on the two CNN implementations (e.g., TensorFlow and CNTK), one of the CNN implementations is likely to be incorrect, without knowing the expected output.

Figure 3.2 shows a bug that causes two backends to be inconsistent. The input image (Figure 3.2a) is manually labeled as a petri dish (the ground truth) in ImageNet (a popular dataset of manually labeled images) [194]. Figure 3.2b shows the classification results of this image by the pre-trained model, InceptionResNetV2 [223], on Keras 2.2.0 with TensorFlow and CNTK backends respectively. While the model with TensorFlow backend classifies the image as a petri dish correctly as its first choice, the same model with CNTK classifies the image as an analog clock, with petri dish not in the top-5.

Once an inconsistency is detected, a big challenge is to identify the faulty functions among the many functions in the DL backend libraries. For example, one run that exposes the inconsistency bug in Figure 3.2 contains 781 invocations of backend functions. Following the complex invocation path of the InceptionResNetV2 model, it is difficult for developers to tease out that the `batch_normalization` function is faulty.

To automatically detect and localize such inconsistencies across DL backends, we propose and implement a novel approach—*CRADLE*. Given a DL model and its input data, CRADLE (1) uses two distance metrics to compare the output of a model on different backends to detect inconsistent output, and (2) identifies the location of the inconsistency by tracking the anomaly propagation through the execution graph. By identifying the spike in the magnitude of the difference between two backends, CRADLE points out the inconsistent functions in the backend that introduce the inconsistency, which should be very useful for developers to debug and understand the bug.

Including the example in Figure 3.2, CRADLE identifies 580 images (out of a 5,000 random sample from ImageNet) that trigger inconsistent classifications for InceptionResNetV2 model. CRADLE then successfully localizes the faulty function (`batch_normalization`). After we reported this bug in the interface, developers have fixed the bug since Keras 2.2.1. Figure 3.2c shows the fix. The batch normalization formula was implemented incorrectly in CNTK backend's function `batch_normalization`: it should take the square root of `(var + epsilon)` instead of the square root of `var`.

To evaluate the effectiveness of CRADLE, we answer the following research questions:

- RQ1: Can CRADLE detect bugs and inconsistencies in deep learning backends?
- RQ2: Can CRADLE localize the source of inconsistencies?
- RQ3: What is CRADLE's detection and localization time?

In this chapter work, we make the following contributions:

- A new approach to test DL software by cross-checking multiple *implementations* of the same model to detect inconsistencies and bugs;
- The first approach to localize the faulty function of a cross-model inconsistency, using anomaly propagation tracking and analysis; and
- An evaluation of the testing and localization technique on 30 DL models, 11 datasets (including ImageNet, MNIST, Udacity challenge 2, and KGS Go game), and 15 Keras versions (including the latest version).

Our results show that CRADLE detects **12 bugs** (9 have been fixed by developers) in DL software that cause inconsistencies for 28 out of 30 models, 3 of which are previously unknown bugs, 2 of which have already been confirmed by developers (RQ1). CRADLE highlights functions relevant to the causes of inconsistencies for all 104 unique inconsistencies (RQ2). CRADLE's median end-to-end running time is less than 5 minutes, suggesting that CRADLE is practical (RQ3).

## 3.2 Background

A DL *network* is a structure (i.e., a graph) that contains nodes or *layers* that are stacked to perform a specific task (e.g., regression or classification). Each layer represents a specific low-level transformation (e.g., convolution, pooling, etc.) of the input data with specific parameters (e.g., *weights*).

Each layer maps to a function invocation that converts weight and the input data to output. While multiple layers in a network can have the same type, the operation performed is generally different because the parameters of these layers are different. In a traditional program, this is analogous to the same methods/functions, defined in one specific place in the source code, are called many times with different input parameters. Similarly, in a DL network, the same *layer type* can be called several times (i.e., in multiple layers) with different input parameters (i.e., weights). Given one input instance, a model maps to an *execution graph* of those low-level functions (i.e., layers).

As a DL network generally consists of more than two layers, there are many intermediate layers. Each intermediate layer produces an internal state that is fed to the next layers. We call such states *hidden states* because they are internal: normal users have no access to them.

To obtain the correct weights for each layer, the network needs to be trained on a *training set*. We call this phase the training phase. Once the training phase is over, the weights (or parameters) of each layer are fixed and do not change, and the model can be used in the inference phase. A *validation set* is a set of input, different from the training set, that is used to tune a model. In this work, we use it as input to the models because we know the ground-truth labels of such input.

A *pre-trained* model is a network that had been trained (and saved) in prior work. Its network structure and weights are fixed and do not change. In the context of this work, a trained model also refers to a pre-trained model. While the training phase is often non-deterministic (e.g., the weights of the network can be initialized randomly), a pre-trained

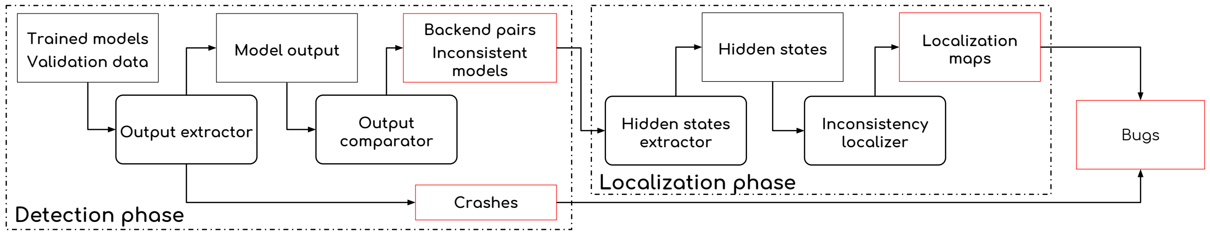


Figure 3.3: Overview of CRADLE. Red boxes indicate CRADLE outputs.

model is expected to behave deterministically in the inference phase because the weights of each layer do not change.

## 3.3 Approach

In this section, we describe how CRADLE detects and localizes inconsistencies among multiple backends. Recall that a backend consists of low-level libraries and the interface to high-level libraries (e.g., Keras). For example, the TensorFlow backend contains the TensorFlow library, the interface between Keras and TensorFlow, and the GPU computation library Nvidia CUDA invoked by TensorFlow.

### 3.3.1 Overview and Challenges

Figure 3.3 shows the two phases of CRADLE: the detection phase and the localization phase. The detection phase takes pre-trained DL models and their corresponding validation data as input. We focus only on the inference stage because of the non-deterministic nature of DL training.

CRADLE runs a pre-trained model using multiple DL backends. Specifically, the *Output extractor* feeds the validation set to the trained model as input and extracts the sets of output from the model on multiple backends. In general, we represent the output as a matrix of numbers. If a DL backend crashes during this extraction stage, the failure is recorded and later reviewed and reported. Otherwise, the *Output comparator* performs pairwise comparisons of the output for each model evaluated on different backends to detect inconsistencies.

Once an inconsistency is detected, CRADLE performs the localization phase. Specifically, the *Hidden states extractor* records hidden states of each inconsistent model on different backends. These hidden states are fed to the *Inconsistency localizer*, which produces localization maps where significant spikes in deviations propagating between hidden states on different backends are highlighted, indicating faulty locations.

To detect and localize cross-backend inconsistencies and bugs effectively, we need to address two main challenges:

**1. How to determine if a model's outputs with two backends are inconsistent?** Since different backends optimize the computational process differently, the results of the same calculation are almost always slightly different [88]. A naive approach that expects the output to be identical will detect inconsistencies for practically all models on all backends, which will not be useful for identifying bugs in DL systems. As shown by our experiment, Theano and CNTK backends always output slightly different values (the differences vary from $10^{-5}$ to less than $10^{-10}$).

It is difficult to know how big of a difference indicates a bug-revealing inconsistency, due to the diversity of models, DL tasks, and datasets. It is not possible to have a single threshold to distinguish between bug-revealing inconsistencies and uninteresting inconsistencies for all models and datasets. For example, LeNet1, a model performing a simple image classification task has an average top-1 confidence level of 95%. This means that for this model, a small variation (e.g., a change in confidence level from 95% to 80%) is unlikely to make the label change. On the other hand, Betago is a model performing a complex task (i.e., playing Go). For this model, the average top-1 confidence level is only 60%. In this case, the same output variation (from 60% to 45%) might change the predicted label. Therefore, different models need different thresholds. Determining the correct threshold is a challenging problem as it depends on many parameters (e.g., dataset, model structure, training, etc.).

To address this challenge of identifying bug-revealing inconsistencies without the need for complex hard-coded heuristics, we use two distance metrics (refer to later sections for details) that emphasize the deviation between the output of both backends and the ground truth. These metrics effectively differentiate bug-revealing inconsistent runs from consistent runs and uninteresting inconsistent runs.

For these metrics, we compare the differences of outputs against the ground-truth instead of comparing individual outputs directly to the expected output. Recall that it is difficult to obtain the expected output as explained in the Introduction. We cannot directly compare the output of one backend to the ground truth to detect bugs because when one backend produces a wrong label it does not necessarily indicate a bug in the backend, as it is common for DL models to produce incorrect labels for some inputs (e.g., due to the limitation of the algorithm/model, not a bug in the implementation).

**2. How to precisely localize the source of an inconsistency?** After an inconsistency is detected, the internal source of the inconsistency is often challenging to localize, due to the complexity of DL backends. For example, one run that exposes the inconsistency bug in Figure 3.2 contains 781 invocations of backend functions that have complex mathematical connections. We propose a novel localization and visualization method that localizes faulty functions in the backend library which introduces inconsistencies by analyzing internal input and output of these backend functions and localizing the error spikes that propagate through the execution graph.

### 3.3.2 Detection Phase

In the detection phase, CRADLE identifies pairs of backends that are inconsistent for a specific model.

**Output extractor** takes as input a pre-trained model and its corresponding validation instances. It loads the provided weights (no training required) and performs classification or regression tasks using the loaded models. It produces the model output using all backends under test for each input instance. For example, comparing 5,000 validation instances and one associated model on 3 different backends will generate 15,000 output vectors. During this phase, CRADLE detects crashes on specific backends and we report them to developers.

**Output comparator** loads previously stored output matrices and performs pair-wise comparisons for each given validation instance to detect inconsistencies. These pair-wise comparisons are between a specific pair of backends using a particular model, its associated validation data, and a particular Keras version. The *Output comparator* then groups inconsistencies into unique inconsistencies. We use two metrics to compare a pair of backends—the Class-based distance for classification and the MAD-based distance for regression.

A straightforward metric to use is top-k accuracy on the entire validation set. Top-k accuracy calculates the portion of correct instances—an instance's ground-truth label is within the top-k predicted labels—among the total number of instances classified. Top-k accuracy could fail to identify certain inconsistencies. For example, the Dog species classification model, affected by the presented *Batch Normalization* bug, induces inconsistency between Tensorflow and CNTK. However, when run on those backends, the model has identical top-1 (29.9%) and top-5 (64.4%) accuracies.

To overcome this problem, we calculate the proportion of inconsistent input instances over the validation set. Because of the way inconsistent input instances are detected, we will not aggregate inconsistencies in the same way as top-k accuracy metric. In the following sections, we introduce Class-based and MAD-based distances as the ways to measure the severity of inconsistent instances. Once we have the severities of all validation instances between a pair of backends, we can apply two thresholds to see if that pair of backends is inconsistent.

***Class-based distance*** is specific to classification models. It calculates the distance between two classifications based on the relative distances of the ground-truth label ranks in the output matrices. Here, we leverage the mapping between the syntax of the model output (the output vector) and its semantic meaning (the classification). Without this mapping, it would be difficult to come up with a universal metric and threshold that could work across different model configurations (e.g., the output vector size of a classifier can vary from 1000 for ImageNet models to 1 for binary classifiers).

A classification model with $N$ classes outputs a vector of size $N$ containing confidence level $p_i$ corresponding to class $C_i$, where $0 < i \leq N$. Confidence level $p_i$ shows how confident the model is in predicting class $C_i$ as the correct label for that input instance. Given an output vector of a classification model as $Y$ and the ground-truth label $C$ of the input, we calculate the score of classification $\sigma_{C,Y}$ as:

$$\sigma_{C,Y} = \begin{cases} 2^{k-rank_{C,Y}} & \text{if } rank_{C,Y} \leq k \\ 0 & \text{otherwise} \end{cases} \tag{3.1}$$

$rank_{C,Y}$ is the rank of the ground-truth label $C$ in the classification $Y$. For example $rank_{C,Y} = 1$ if $C$ is predicted as top-1 in classification $Y$. The score $\sigma_{C,Y}$ emphasizes on classifications that predict ground-truth label with higher rank. We consider $rank_{C,Y}$ larger than k(i.e., ground-truth label $C$ not in top-k) not interesting and give it zero score.

Given the confidence level output of the same model on a different backend as $Y'$, the Class-based distance $\text{D\_CLASS}_{C,Y,Y'}$ is calculated as the absolute difference between two scores $\sigma_{C,Y}$ and $\sigma_{C,Y'}$:

$$\text{D\_CLASS}_{C,Y,Y'} = |\sigma_{C,Y} - \sigma_{C,Y'}| \tag{3.2}$$

We define our Class-based metric based on the top-k rankings with $k = 5$. For example, in Figure 3.2, $\sigma_{petridish,Y_{TF}} = 2^{5-1} = 16$ as the rank of petri dish label by the TensorFlow backend $rank_{petridish,Y_{TF}}$ is 1. Similarly, $\sigma_{petridish,Y_{CN}} = 0$ because petri dish is not in CNTK's top-5 for that image. Then the Class-based distance $\text{D\_CLASS}_{petridish,Y_{TF},Y_{CN}}$ is 16. If another backend generates the ground-truth label in rank 3, then its $\sigma_{petridish,Y}$ is 4, and $\text{D\_CLASS}_{petridish,Y_{TF},Y}$ is 12. The maximum value of $\text{D\_CLASS}_{C,Y,Y'}$ is 16, and the minimum is 0 with $k = 5$.

***Mean absolute deviation (MAD)-based distance*** is a metric that could be used for both classification and regression models. However, the main purpose of the MAD-based distance is detecting inconsistencies in regression models where our Class-based distance would not work.

Given two predicted vectors $Y$ and $Y'$ of size $N$ for a pair of backends using a model and an input instance, we first calculate the Mean Absolute Distance (MAD), $\delta_{O,Y}$ and $\delta_{O,Y'}$, between the two output vectors and the ground-truth vector $O$. $\delta_{O,Y}$ is calculated as followed:

$$\delta_{O,Y} = \frac{1}{N}\sum_{i=1}^{N} |Y_i - O_i| \tag{3.3}$$

The MAD-based distance $\text{D\_MAD}_{O,Y,Y'}$ is calculated as:

$$\text{D\_MAD}_{O,Y,Y'} = \frac{|\delta_{O,Y} - \delta_{O,Y'}|}{\delta_{O,Y} + \delta_{O,Y'}} \tag{3.4}$$

MAD is used here (instead of the more common Euclidean distance) because it does not inflate due to outliers.

For example, Dave-2 [47] is a model that outputs the steering angle (measured in radian) of a car given a dashboard camera image as input. For a given input image I, the recorded (ground-truth) steering angle is $O = 0.0$. Using the same image as input, Dave-2 outputs $Y = 0.4$ and $Y' = -0.1$ using two different backends. We have $\delta_{OY,} = |0.4 - 0.0| = 0.4$ and $\delta_{O,Y} = |-0.1 - 0.0| = 0.1$. We can then calculate $\text{D\_MAD}_{O,Y,Y'}$ as $|0.4 - 0.1|/(0.4 + 0.1) = 0.6$. The MAD-based metric produces values between 0 and 1.

Before we can use this metric with classification models, we first need to convert the ground-truth labels to one-hot vectors. In multi-class classification, a one-hot vector is a vector of all zero except the value at the ground-truth label index is 1. This vector indicates a perfect classification with 100% confidence in the ground-truth label.

***Identifying Inconsistencies:*** Given a model (and its validation set), two backends, and one version of Keras, we consider this pair of backends *inconsistent* if at least $p$% of validation input instances cause the distance between those two sets of output to be larger than a given threshold $T$ ($T_C$ denotes the threshold for the Class-based metric and $T_M$ for MAD-based metric). We call such input instances *inconsistency-triggering*.

For Class-based metric with $k = 5$, using threshold $T_C = 16$ is the most strict. This means that an input instance is considered inconsistency-triggering if one backend ranks the ground-truth label top-1, while the other ranks it outside of the top-5. Using threshold $T_C = 1$ means that an input instance is inconsistency-triggering if there is any difference

Table 3.1: Example of inconsistencies found using the Class-based metric. TF is TensorFlow and CN is CNTK.

| | | | | Inconsistency pattern | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| Id | Keras | Backends | Model | 16 | 15-8 | 7-4 | 3-2 | 1 | 0 |
| 1 | 2.2.2 | TF-CN | Xception | 10 | 202 | 147 | 100 | 85 | 4456 |
| 2 | 2.2.2 | TF-CN | NASNetLarge | 5 | 132 | 86 | 77 | 65 | 4635 |
| 3 | 2.2.1 | TF-CN | Xception | 10 | 202 | 147 | 100 | 85 | 4456 |
| 4 | 2.2.1 | TF-CN | NASNetLarge | 5 | 132 | 86 | 77 | 65 | 4635 |

in the top-5 labels of the two backends and the ground-truth label is in the top-5 of at least one backend (e.g., if one backend ranks the ground truth label in the top-5, while the other backend ranks it outside of the top-5). In Figure 3.2, the petri dish image is an inconsistency-triggering input instance.

Similarly, for MAD-based metric, using $T_M = 1$ is the most strict. For example, with the Dave-2 model, an input image is *inconsistency-triggering* with $T_M = 1$ if it causes one backend to predict an angle matching the recorded angle exactly, while causing the other to predict a different angle. On the other hand, using $T_M = 0$ means that we consider any input image inconsistency-triggering.

The stricter the thresholds are the fewer inconsistencies wil be detected. However, the detected inconsistencies will be more severe (higher $T_C$ or $T_M$ means each inconsistency-triggering instance is more severe, while higher $p$ means more output instances are inconsistent). If covering all inconsistencies is the priority, lower and more relaxed thresholds should be used (e.g., the recommended thresholds in Section 3.4). However, if finding severe bugs that significantly affect models' accuracies is the priority, then stricter settings would ensure that those severe bugs will be found given less inspection effort.

***Identifying Unique Inconsistencies:*** Table 3.1 shows four examples of inconsistencies. These inconsistencies are identified using the Class-based metric. Column '7–4' is the number of validation input instances that cause the two backends to have Class-based distances of 7, 6, 5, or 4. Inconsistency in row one (inconsistency 1) indicates that the model Xception is inconsistent between TensorFlow and CNTK (Keras 2.2.2) on its associated ImageNet validation set where 10 input instances trigger a Class-based distance of 16, 202 instances trigger distances in the range of 15–8, etc.

The same inconsistencies may exist in different Keras versions (different interface versions in the backend). To avoid finding duplicate inconsistencies, the output comparator also automatically groups certain inconsistencies together into unique inconsistencies based on inconsistency patterns.

An *inconsistency pattern* is the distribution of the distances over the entire validation data. It expresses the characteristics of the inconsistencies. Table 3.1 shows two unique inconsistency patterns: pattern 1 (for inconsistencies 1 and 3) and pattern 2 (for inconsistencies 2 and 4).

Since the range of MAD-based metric is between 0 and 1, we choose 5 equal sized bins between 0 and 1 to calculate the inconsistency patterns. Similar to Class-based metric, the number in bin 0.6-0.8 is the number of input instances that trigger the MAD-based distance $0.6 \leq \text{D\_MAD} < 0.8$ for each pairwise comparison.

### 3.3.3 Localization Phase:

Given each unique inconsistency, the *Hidden states extractor* and the *Inconsistency localizer* produce a localization map. A localization map is an execution deviation graph of two implementations (backends), which highlights inconsistent executions (hidden states) of a function (layer type), pointing to potential faulty functions in one of the backends.

Recall that an execution of a model produces one execution graph (Section 3.2). Each execution graph contains connected layers, where the output of one layer is the input of subsequent layers. Given a model and an input instance, there is one execution graph for each implementation of libraries. An *execution deviation graph* is a graph that represents the differences between two execution graphs of the same model. Since both execution graphs are from executions of the same model, they have the same structure i.e., the network structure. Thus, the execution deviation graph also has that same structure but contains the deviation between each pair of layer type executions. We describe the deviation calculation below.

For each unique inconsistency, we only perform localization on the most inconsistent input instance. *The most inconsistent input instance* triggers the largest Class-based distance (classification tasks) or MAD-based distance (regression tasks) between the output of two backends.

**Hidden states extractor** produces execution graphs in a similar way to the *Output extractor* described previously. Both execute the model on validation input instances to extract output. However, the latter also retrieves the intermediate function output (hidden state) of each hidden layer (internal execution) in the model. Hidden states are presented as vectors of floating point numbers.

**Inconsistency localizer** produces a localization map for each unique inconsistency by first extracting the execution deviation graphs. It does this by calculating the mean absolute

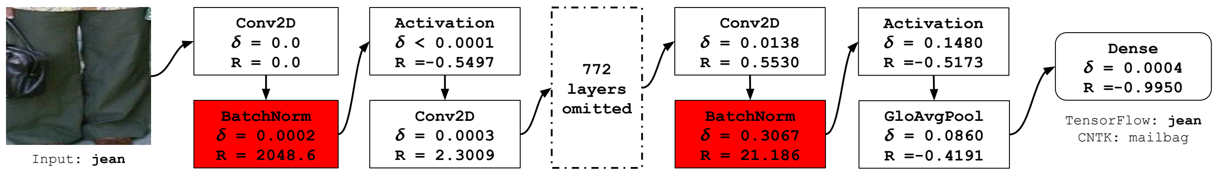


Figure 3.4: Batch normalization bug's localization map for InceptionResNetV2 between TensorFlow and CNTK with Keras 2.2.0.

deviation (MAD) between each pair of corresponding hidden state from two executions of the same layer type on two different backends. It is important not to confuse the usages of MAD here to the MAD-based metrics mentioned previously. Here, MAD is used to calculate the distances between corresponding intermediate outputs of hidden layers to represent the internal deviations of two execution graphs. Given the intermediate states $S_L$ and $S'_L$ of layer $L$ executed on two backends, the deviation is calculated using Equation 3.3 as $\delta_{S_L,S'_L}$.

Due to the sequential nature of a model, a noticeable MAD deviation at a particular layer does not indicate inconsistency at that layer as deviation can propagate through the execution graph and get amplified along the way. Ideally, we want to localize the source of the inconsistency. To do this, the *Inconsistency localizer* calculates the rate of change in deviation between consecutive function executions. Finally, it generates the localization maps by highlighting functions in the execution deviation graph that have inconsistent executions.

To calculate the rate of change, we first need to calculate the MAD deviation for all executions (layers output) in the set $pre(L)$ as $\delta_{S_l,S'_l}$ with $l \in pre(L)$ ($pre(L)$ is the set of inbound layers which hidden states are the input to layer $L$). We calculate the representative deviation of inbound executions, $\delta_{pre}$, simply as the maximum deviation:

$$\delta_{pre} = \max_{l \in pre(L)} (\delta_{S_l,S'_l}) \tag{3.5}$$

The rate of change in deviations at layer $L$ is:

$$R_L = \frac{\delta_{S_L,S'_L} - \delta_{pre}}{\delta_{pre} + \epsilon} \tag{3.6}$$

We use a smoothing constant $\epsilon = 10^{-7}$ to prevent $R_L = \infty$ in the case where $\delta_{pre} = 0$ (e.g., $L$ is the first layer).

We call $R_L$ the inconsistency introduction rate of a layer $L$, i.e., how much diversion layer $L$ (executions of a pair of function implementations) introduces due to inconsistent implementations. $R_L$ values of all layers provide an overall picture of how the inconsistency is introduced through the model so that we can localize the function that is the source of the inconsistency. To generate the localization map, we overlay the MAD and $R_L$ values for each layer on the model structure graph (e.g., maps in Figure 3.4). A node, representing a layer $L$, shows the layer type (i.e., low-level transformation function), the MAD value $\delta$, and the inconsistency introduction rate $R_L$. We select the third quantile of $R_L$ distribution of all nodes in each map as the highlighting threshold. We highlight a node red if its $R_L$ is higher than this threshold.

## 3.4 Datasets and experimental settings

**Trained Models and Datasets:** To evaluate CRADLE, we collect 11 public datasets and 30 DL models that are pre-trained from these datasets. Table 3.2 lists the datasets.

We collected the models by looking for pre-trained models compatible with Keras from prior work and GitHub. To avoid low-quality models (e.g., class projects and simple demos), we only examine repositories with at least two stars. Overall, we collected 13 ImageNet [194] models (Xception, VGG16-19, ResNet50, InceptionV3, InceptionResNetV2, MobileNetV1-V2, DenseNet121-169-201, NASNetLarge-Mobile [49]), 3 self-driving models used in previous work (DaveOrig-Norminit-Dropout [47,179]), 3 MNIST models (LeNet1-4-5 [137]), and various models trained for other tasks (Thai number detector – ThaiMnist [9], Go game player – Betago [3], anime faces recognition – AnimeFaces [2], cat and dog classifiers – CatDog(Basic, Augmented) [1,4], dog species classifier – Dog [6], gender detection – Gender [5], Pokemon classifier – Pokedex [7], and GTSRB traffic sign recognition – TrafficSigns(1, 2, 3) [11–13]). We use provided validation dataset for each model to run our experiment. For ImageNet, we use a random sample of 5,000 images from over 80,000 provided cropped validation images.

**Experimental settings:** We run CRADLE on 15 versions of Keras (2.0.5–2.2.2). For the low-level libraries, we use the latest versions of CNTK (2.5.1), Theano (1.0.1), and TensorFlow (1.7.0). For regression models, i.e., Dave variants, we only use the MAD-based metric because the Class-based metric does not apply. For the classification models, we use both Class and MAD-based metrics. Some models are not supported with older versions of Keras and result in crashes. Since the crash is the expected behavior, we do not consider them as bugs and exclude those runs from our experiment.

We vary the thresholds ($T_C$, $T_M$, and $p$) and found the optimal setting (covering the most inconsistency without false positives and false negatives) for Class-based metric are $T_C = 8$ and $p = 0\%$ and for MAD-based metric are $T_M = 0.2$ and $p = 0\%$. We use cross-validation with 80-20% of models to confirm that the thresholds consistently perform across all 5 folds. These are the thresholds we use in RQ1 and RQ2.

**Hardware and Infrastructure:** We utilize multiple Anaconda environments to switch between multiple versions of Keras and different backends. We run all experiments on an Intel Xeon E5-2695 machine with 128 GB of RAM and two Nvidia Titan XP GPUs. For the performance analysis, we run the output extraction step utilizing a single GPU.

## 3.5 Results

### 3.5.1 RQ1: Can CRADLE Detect Bugs and Inconsistencies in Deep Learning Backends?

CRADLE detects **12 bugs** in DL software for 28 out of 30 models that cause **104 unique inconsistencies**. The 12 bugs (9 have been fixed) consist of 7 inconsistency bugs (3 previously unknown, 2 out of 3 have already been confirmed by developers, e.g., the bug in Figure 3.2 has been fixed by developers after we reported it), and 5 crash bugs that crash either Keras or one of the backend libraries. None of the 12 bugs is detected by the test cases that come with Keras (including the interface), which does simple unit and integration testing. The results demonstrate that cross-backend inconsistencies are frequent and CRADLE is effective in detecting them.

Our approach does not report false inconsistencies as it is a dynamic approach: for each inconsistency, we have inputs that trigger two backends to disagree. Theoretically speaking, some true inconsistencies may indicate a false bug, as our approach may identify uninteresting inconsistencies (e.g., natural computation difference explained in Section 3.3.1). In our experiment, all 12 bugs are real (i.e., no false bugs detected).

**Inconsistencies and inconsistency-triggering input:** Using the Class-based metric on classification tasks and the MAD-based metric on regression tasks, CRADLE detected a total of 361 inconsistencies. Based on the inconsistency patterns, CRADLE automatically groups the inconsistencies into 104 unique inconsistencies (Section 3.3.2).

Table 3.2 shows the number of inconsistencies found by CRADLE for each dataset and pair of backends. For example, CRADLE detects '21(54)' inconsistencies between the

Table 3.2: Number of inconsistencies found by CRADLE. The numbers outside and (inside) brackets are the unique and (total) number of inconsistencies respectively. TF is TensorFlow, TH is Theano, and CN is CNTK.

| Dataset | Instances | # of Inconsistencies | | |
|---|---|---|---|---|
| | | TH-TF | TF-CN | CN-TH |
| ImageNet | 5,000 | 10(34) | 21(54) | 18(46) |
| Driving | 5,614 | | 3(9) | 3(12) |
| MNIST | 10,000 | | 3(9) | 3(12) |
| Thai MNIST | 1,665 | | 1(3) | 1(4) |
| KGS Go game | 12,288 | 2(14) | 3(12) | 3(15) |
| Anime Faces | 14,490 | 1(5) | | 1(6) |
| Dogs VS Cats | 832 | | 2(6) | 2(8) |
| Dog species | 835 | | 3(8) | 3(9) |
| Faces | 466 | 2(14) | 3(8) | 6(15) |
| Pokedex | 1,300 | 1(14) | 1(3) | 2(15) |
| GTSRB sign | 12,630 | 2(14) | 2(5) | 2(7) |
| Total | | 18(95) | 42(117) | 44(149) |
| | | | 104(361) | |

two backends TensorFlow and CNTK triggered by 13 ImageNet models. Here '21(54)' indicates that CRADLE detects 54 inconsistencies which map to 21 unique inconsistencies corresponding to 21 unique inconsistency patterns. Table 3.1 shows two such patterns (the first and second rows).

On average, these inconsistencies are triggered by 21.9% of input instances in a dataset (22.2% for classification tasks and 13.9% for regression tasks). Figure 3.5 provides examples of inconsistency-triggering inputs. The image of a groom was identified correctly by TensorFlow but incorrectly as an Indian elephant by the faulty Theano. In some extreme cases, the faulty TensorFlow backend *accidentally* labels an image of bananas "correctly" while CNTK identifies it as tennis balls.

**Inconsistency bugs:** We use CRADLE to localize the source function of all 104 detected unique inconsistencies (detailed localization results are in Section 3.5.2). We find that they are caused by 7 bugs in the backend libraries (Table 3.3). Some bugs have the same root inconsistency because they are either different bugs in the same function or affect several backends which required multiple fixes to multiple backends. For example, in addition to the batch normalization bug we presented earlier, we found another bug in the batch normalization function affecting an older version of Keras.

We manually check the fault localization maps for each cluster of inconsistencies and

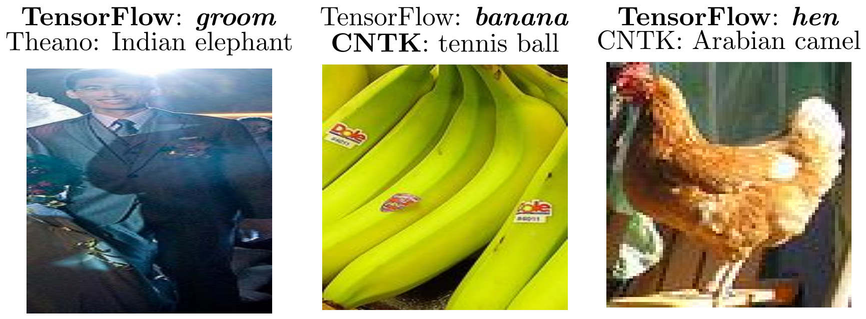


Figure 3.5: Inconsistency-triggering inputs for the pooling bug (left column), the padding bug (middle column), and the batch normalization bug (right column). Correct backends are bold.

Table 3.3: Bugs found by CRADLE. '# Inc. bugs' indicates the number of inconsistency bugs per root inconsistency.

| Root inconsistency | Localized layers (functions) | Affected backends | # Affected models | # Inc. bugs |
|---|---|---|---|---|
| Batch normalization | BatchNomalization | CNTK | 11 | 2 |
| Padding scheme | Conv2D, DepthwiseConv2D, SeparatableConv2D | TensorFlow, Theano | 15 | 2 |
| Pooling scheme | AveragePooling2D | Theano | 3 | 1 |
| Parameter organization | Trainable convolution | CNTK, Theano | 18 | 2 |

confirm whether it indicates a bug. If we find a corresponding bug fixing commit in a more recent version, we consider the bug has been fixed by developers. If not, we consider it previously unknown. Once two authors agree that it is a bug, we report it to developers.

If the same invocation of functions is identified for multiple bugs that are triggered by the same model in the same pair of backends across successive Keras versions (which affect the interface code between Keras and low-level libraries), we consider them one unique bug. However, if the bugs are in nonconsecutive versions, and the inconsistency pattern changes for some versions of Keras, this indicates that the issue was partially fixed (or a new bug introduced) in some Keras versions, then we consider them different bugs (e.g., the new inconsistency is likely to be a regression bug).

In addition to the batch normalization bug in Figure 3.2, we detail two additional confirmed bugs that CRADLE found.

***Padding scheme bugs:*** Padding artificially increases the size of an input image so that a kernel function can be applied to all the pixels of the original image and produces an

```
-   if padding == 'same':
-     th_avg_pool_mode = 'average_inc_pad'
-   elif padding == 'valid':
-     th_avg_pool_mode = 'average_exc_pad'
  ...
-                   mode=th_avg_pool_mode)
+                   mode='average_exc_pad')
```

Figure 3.6: Pooling scheme bug fix in `pool2d` in Theano backend.

output of the same shape as the input. The `SAME` padding scheme behaves inconsistently across backends when applied on different combination of odd or even sizes of input and kernel. This creates a shift in the input that propagates through the model and caused the model to sometimes completely miss some of the shapes it was trained to recognize. Eventually, it results in inconsistencies between Theano or TensorFlow (depends on the different combination of input and kernel sizes) and the other two backends. The middle column of Figure 3.5 shows an example of input images revealing this bug. Although it has not been fixed yet in the interface source code, this bug has been confirmed to be a significant problem because various models (i.e., ResNet50, MobileNet, NASNetsLarge-Mobile, and MobileNetV2) have been updated by their developers to include workarounds that makes their models consistent across backends.

***Pooling scheme bug:*** This bug in Theano backend causes Gender, InceptionRes-NetV2, and InceptionV3 models to misbehave. In Keras 2.1.4 and earlier, the 2D pooling layer in Theano interface determined the average pooling scheme based on the padding scheme. If the padding is `SAME`, it used the pooling `average_inc_pad` scheme which includes padding in the average calculation. However, if there is no padding, then they use the `average_exc_pad` scheme. This creates inconsistencies for models that use the Average-Pooling layer with `SAME` padding. Figure 3.6 presents the fix where `average_exc_pad` is used regardless of the padding scheme.

**Crashes bugs:** Excluding crashes caused by unsupported models, we encounter 86 crashes out of 1173 possible runs. We identified 3 Keras bugs (happened with all backends) and 2 specific backend bugs. In total, 4 of the crash bugs have already been fixed and a workaround has been added to the crashing model to address the last issue. They are often caused by incorrect shapes (e.g., incorrect weight or convolution kernel shapes).

**Comparison between Class-based metric and top-k accuracy:** One alternative to our Class-based metric is top-k accuracy. To measure its effectiveness in detecting inconsistencies, we integrate it into CRADLE by calculating the top-k accuracy differences between pairs of backends. A pair is considered inconsistent if the accuracy difference is larger than a threshold $T_{AC}$. We vary $k$ (1 to 5) and accuracy threshold $T_{AC}$ (between 0% and 50%).

Using $T_{AC} = 0\%$ and $k = 1$, the accuracy metric detects the most number of inconsistencies (305) but still misses 35 inconsistencies found by our Class-based metric. These are 35 valuable test cases that developers could use to test, localize, and fix detected bugs. In addition, our Class-based metric enables the generation of inconsistency patterns which help remove duplicates to reduce 340 detected inconsistencies to 98 unique inconsistencies. This reduction is not possible with top-k accuracy. The results show that our Class-based metric is more effective than top-k accuracy.

**MAD-based metric usage for classification models:** To demonstrate the usefulness of our Class-based metric, we compare the ability of both metrics to detect unique inconsistencies for classification models.

Using the MAD-based metric for classification tasks, CRADLE can only find 10 unique inconsistencies, 4 of which are inconsistent in confidence level but do not trigger inconsistent classifications. On the other hand, with the Class-based metrics, CRADLE correctly identifies 98 unique inconsistencies in classification models, including all inconsistencies correctly found using the MAD-based metric. These results show that Class-based metric help CRADLE find more inconsistencies with no false positives.

### 3.5.2 RQ2: Can CRADLE Localize The Source of Inconsistencies?

For each of the 104 unique inconsistencies, CRADLE generates a localization map for the most inconsistent input instance (Section 3.3.3). By focusing on the first localized inconsistent execution and executions with high inconsistency introduction rates in each map, we manually cluster the 104 unique inconsistencies into 7 bugs. CRADLE's localization maps enable us to do this clustering. This manual process takes 1–2 hours per bug. A technique to automatically cluster unique inconsistencies based on the first localized function executions or similarity between localization maps remains as future work.

Overall, CRADLE highlights executions that are relevant to the causes of inconsistencies for all 104 unique inconsistencies. For 4 of the bugs, the first localized inconsistent executions are exactly the executions of faulty functions that were fixed by developers. This suggests that the localization technique is effective in pinpointing the faulty functions, which should help developers to understand and fix the bugs. For example, the

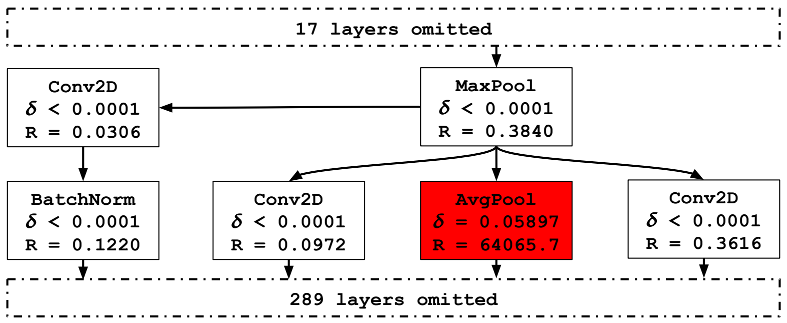


Figure 3.7: Pooling scheme bug's localization map for model InceptionV3 between TensorFlow and Theano with Keras 2.1.4 on the "groom" input image in Figure 3.5.

reduction is 13 to 1 in one case, meaning that the developers only need to examine one function instead of 13 functions with complicated formulae and interactions to understand and fix the bug. When we consider all (instead of only the first) localized inconsistent executions, the faulty methods are invoked in one of the localized inconsistent executions for 5 of the bugs. For the fifth bug, this represents a reduction of 22–44% in the number of functions to examine. For the remaining 2 bugs, the localized inconsistent executions are related to the bug fixes. In fact, the localized executions helped us tremendously in understanding the bugs so that we were able to write good bug reports.

Figure 3.4 shows a part of a localization map for the batch normalization bug (for the unique inconsistency involving InceptionResNetV2, TensorFlow and CNTK backends, and Keras 2.2.0). The input image shown is the most inconsistent input instance for this unique inconsistency. The Dense box shows the output: "jean" from TensorFlow, and "mailbag" from CNTK, while the ground truth is "jean". The map includes 781 invocations of backend functions. For presentation purposes, we omit 772 invocations. Each box represents an invocation of a neural network function. The arrows indicate the flow of data. Function names are indicated in each box, while $\delta$ is the MAD distance between the hidden states (defined in Equation 3.3), and $R$ is the inconsistency introduction rate (defined in Equation 3.6). In this example, executions of function `batch_normalization` are localized as faulty (shown in red). The white boxes indicate executions with low or negative $R$ (i.e., they are unlikely the source of inconsistency). This map correctly highlights the earliest invocation of the function `batch_normalization` as the source of inconsistency. We examine localization maps for the other affected models (e.g., InceptionV3, DenseNets (121, 169, 201)) and notice that they all point to the `batch_normalization` function. We reported this bug to developers and it has been fixed in Keras 2.2.1.

Figure 3.7 shows a section of the localization map highlighting the faulty executions for pooling scheme bug with model InceptionV3 between TensorFlow and Theano on Keras 2.1.4. The first highlighted execution indicates correctly the source of this inconsistency as the function `average_pooling`. We look at the source code of `average_pooling` which points to the faulty `pool2d` function in the Theano backend. Figure 3.6 shows the fix (for Keras 2.1.5) in the Theano backend source code where the average pooling scheme is set to `average_exc_pad` regardless of the padding scheme.

### 3.5.3 RQ3: What Is CRADLE's Detection and Localization Time?

We measure the execution time of CRADLE on the latest version of Keras (2.2.2) using all 30 models. Overall, CRADLE's detection and localization time is quite reasonable with a typical end-to-end execution time lower than 5 minutes.

The running times of *Output extractor* and *Hidden states extractor* dominate the model execution times, which depend on the model complexity, validation dataset size, and performance of the backend. The extractor is slow in rare cases, e.g., nearly 10 hours with the large NASNetLarge model containing over 1,000 layers. However, the typical running time is within minutes with the median of less than 2 minutes.

The *Output comparator* and *Inconsistency localizer* are much faster with median running time less than 20 seconds and maximum less than 5 minutes. The running time is independent of the backend implementation; it depends on the dataset size and the model complexity respectively.

## 3.6 Limitations and Threats to Validity

Since we focus on detecting bug-revealing inconsistencies, CRADLE may miss inconsistencies that cause internal errors but not failures (i.e., incorrect external behaviors). This is our design choice to avoid detecting too many false alarms.

We assume that the same algorithms are implemented with similar specifications in all backends due to the interchangeability of DL backends. In theory, it is possible for our technique to find false positive inconsistencies because of this assumption. However, our results show that the inconsistencies found by our approach almost always indicate real bugs because 11 of them have already been confirmed or fixed by developers.

Our approach might not be generalizable to other models or DL libraries. To mitigate this threat, we use 30 models extracted from different GitHub projects and evaluate our approach on Keras, the most popular high-level DL library [10], and three popular backends. Our approach of detecting and localizing inconsistencies should be applicable to other models and libraries with little work.

It is possible that some complex DL systems contain non-deterministic layers so that, given the same input, the output might be slightly different. To mitigate this issue, we make sure none of the layers contains intentional sources of randomness and we apply two metrics that are designed to be robust even in the existence of small inconsistencies.

Our approach uses pre-trained models. We made this our design choice as we believe that those pre-trained models that are used by real users are likely to cause bugs that developers care. Alternatively, we could use dummy models or mutated models to test backends in order to find more bugs, which remains as future work.

## 3.7 Summary

We propose CRADLE, a new approach to find and localize bugs in the implementations of DL models by cross-checking multiple backends. We evaluate CRADLE on three backends and 30 pre-trained models and find 12 bugs and 104 unique inconsistencies in the backends for 28 models. This work calls for attention for testing DL implementations and not just DL models. In the future, we plan to design approaches to identify bugs even if they do not cause observable differences in backends. It is also conceivable to expand the set of trained models with mutants for CRADLE to find more bugs.

# Chapter 4

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

## 4.1 Motivation

Deep learning is widely used in many fields including autonomous driving cars [43], diabetic blood glucose prediction [160], and software engineering [39, 44, 45, 102, 131, 189, 250, 255, 257]. DL training algorithms utilize nondeterminism to improve training efficiency and model accuracy, e.g., using shuffled batch ordering of training data to prevent overfitting and speed up training [97].

These *nondeterminism-introducing (NI)-factors* can cause multiple *identical training runs*, i.e., training runs with the *same* settings (e.g., identical training data, identical algorithm, and identical network) to produce different DL models with different accuracies and training times [120, 195, 221, 229].

For example, our experiments show that for 16 identical training runs of a popular DL network, LeNet5 [138], the accuracy of the resulting 16 models ranges from 8.6% to 99.0% —a large accuracy difference of 90.4%. Four of these identical training runs resulted in *weak models* (accuracy below 20%). Even if we exclude such models, the accuracy difference is still up to 10.8% with LeNet1 between the most accurate run (98.6%) and the least accurate run (87.8%).

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

In addition to these algorithmic NI-factors, DL libraries (e.g., TensorFlow [26] and cuDNN [48]) introduce additional variance. For example, by default, *core DL libraries* (e.g., TensorFlow or PyTorch [175]) perform data preprocessing in parallel for speed, which changes the order of training data, even if algorithmically the batch order is fixed. Furthermore, core DL libraries, by default, leverage *autotuning* to automatically benchmark several modes of operation (i.e., underlying algorithms for computations such as addition) for each primitive cuDNN function (e.g., pooling and normalization) in its first call. The fastest mode of operation is then used for that cuDNN function in subsequent calls. Different identical runs sometimes use different modes of operation, introducing variance.

Our experiments (Section 4.5.2) show that these *implementation-level NI-factors* alone cause an overall accuracy difference of up to 2.9%. Specifically, we train the popular WideResNet-28-10 [247] (or WRN-28-10 for short) DL network for image classification 16 times using the same default training configuration (e.g., same CIFAR100 [126] training data, batch size, optimizer, and learning rate schedule), identical model selection criteria (i.e., selecting models with the lowest loss on a validation set), the same DL libraries (i.e., Keras 2.2.2, TensorFlow 1.14.0, CUDA 10.0, and cuDNN 7.6), and identical hardware (i.e., the same NVIDIA RTX 2080Ti GPU), while disabling all algorithmic NI-factors (i.e., using fixed random seeds to ensure identical initial weights, identical batch ordering, deterministic dropout layers, and deterministic data augmentation). This process generates 16 models. Since all algorithmic NI-factors are disabled, one may expect little variance across the 16 runs. However, we found that the accuracies of these 16 models vary between 77.3% and 80.2% (a 2.9% difference).

These implementation-level NI-factors and differences are often characteristics and consequences of the DL *software implementations*, which create unique challenges for SE researchers and practitioners (Section 4.8). Meanwhile, DL researchers have paid little attention to implementation-level NI-factors (the focus is on theoretical analyses of DL training [51, 52, 68, 123, 142, 198]).

To see whether such differences are known, we conduct a survey (Section 4.6), which surprisingly shows that 83.8% of the 901 responded researchers and practitioners with DL experience are unaware of (63.4%) or unsure about (20.4%) any implementation-level variance! Of the 901 respondents, only 10.4% expect 2% or more accuracy difference across fixed-seed identical training runs.

We also perform a literature survey of 454 papers *randomly* sampled from recent top

*SE*, *AI*, and *systems* conferences to understand the awareness and practices of handling DL system variance in research papers. Of 225 papers that train and evaluate DL systems, only 19.5±3% use multiple identical training runs to quantify the variance of their DL approaches.

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

In addition, there are large differences in convergence time. For example, the time to convergence of the 16 fixed-seed identical training runs of another popular network, ResNet56 [100], ranges from 2,986 to 7,324 seconds (a one hour and 12 minutes difference)—a 145.3% relative time difference. We also observe a discrepancy between the empirical per-class accuracy and convergence time and the corresponding estimates from the surveyed researchers and practitioners.

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

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

For example, a recent research paper [133] proposed a new approach with a reported 0.8% accuracy improvement over the standard WRN-28-10. Our experiments show that this network's accuracy can vary by up to 2.9%. Therefore, the reported accuracy improvement may not be statistically significant when considering the aforementioned NI-factors in the training algorithm and the implementation. In other words, if one runs the two approaches again, the resulting mode of [133] may not outperform the WRN-28-10 model. At best, the comparison results still hold, but the current experiments fail to provide evidence to demonstrate the improvement given the possible variance.

There are existing theoretical analyses of DL training [51,52,68,123,142,198] that study how well optimizers find good local optima given algorithmic NI-factors. Such work fails to study the nondeterminism in the underlying DL implementation.

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

In this chapter, we make the following contributions:

- ⧫ **Finding 0**: A list of implementation-level NI-factors (parallel processing, auto-selection of primitive operations, and scheduling and floating-point precision) and algorithmic NI-factors (nondeterministic DL layers, weight initialization approach, data augmentation approach, and batch ordering), and techniques that control these NI-factors to remove or reduce variance (Section 4.2).

- ⧫ A DL variance study of 4,838 hours (over 6.5 months) of GPU time on 3 widely-used datasets (MNIST, CIFAR10, CIFAR100) with 6 popular models (LeNet1, LetNet4, LetNet5, ResNet38, ResNet56, and WideResNet-28-10) on three core DL libraries (TensorFlow, CNTK, and Theano):

  - **Finding 1**: The accuracy of models from 16 identical training runs varies by as much as 10.8%, even after removing weak models.
  - **Finding 2**: With algorithmic NI-factors disabled, DL model accuracy varies by as much as 2.9%—an accuracy difference caused solely by implementation-level nondeterminism.
  - **Finding 3**: Implementation-level NI-factors cause a per-class accuracy difference up to 52.4%, while the per-class difference is up to 100% with default settings (i.e., with algorithmic and implementation-level nondeterminism).
  - **Finding 4**: Training time varies by as much as 145.3% (1 hour and 12 minutes) among fixed-seed identical training runs, while the training time difference is up to 4,014.8% with default settings.

- ⧫ A researcher and practitioner survey, with 901 valid replies, reveals that:

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

(84.5%) estimates a similar 10% or less convergence time difference among fixed-seed identical training runs.

- **Finding 9**: A literature survey of a random sample of 454 papers from the most recent editions of top SE (ICSE [20], FSE [17], and ASE [14]), AI (NeurIPS/NIPS [22], ICLR [24], ICML [19], CVPR [16], and ICCV [18]), and systems (ASPLOS [15], SOSP [23], and MLSys [21]) conferences, shows that 225 papers train and evaluate DL systems, only 19.5±3% of which use multiple identical training runs to evaluate their approaches.

- **Implications and suggestions** for researchers and practitioners, and raising awareness of DL variance (Section 4.8).

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

## 4.2 Nondeterminism-Introducing (NI)-Factors

Many factors affect DL systems' results and training time. The first set of factors is the *input* to a system. Such input includes the training data and hyper-parameters (e.g., number of layers, dropout rate, optimizer, learning rate, batch size, and data augmentation method and settings). It is expected that models with different inputs perform differently, and there is a flurry o f work on how to select the best input (e.g., hyper-parameter tuning) [38, 111, 246].

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

### 4.2.1 Definitions

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

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

*computation libraries* (e.g., cuDNN and CUDA), and hardware (e.g., GPU, CPU, and TPU). The DL training process spans across the DL algorithm and DL implementation.

### 4.2.2 Algorithmic NI-factors

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

**Nondeterministic DL Layers:** DL architectures can contain nondeterministic layers. For example, dropout layers [219] are commonly used to prevent overfitting. They randomly set parts of the input tensor to zero during training and guide each neuron to be trained with different portions of the training data. *Dropout* tensors are chosen randomly on-the-fly during training, which means two identical runs could produce two different models with different accuracies and different training times.

**Weight Initialization:** Weight initialization [87, 99, 139, 198] is an important step in DL training [221, 229]. Random weight initialization samples the initial DL model weights from a predefined distribution. Goodfellow et al. [91] state that random initialization "breaks the symmetry" across all the weight tensors. This process helps similarly structured neurons learn different functions instead of repeating each other. Thus, they learn different aspects of the training data and help to increase the model's generalization. However, different initial weights may result in convergence to different local minima [69, 83, 140]. Therefore, random initialization can lead to variance in model accuracy across identical runs.

**Data Augmentation:** DL training algorithms also utilize the randomness in data augmentation to improve their effectiveness (i.e., produce more accurate models). Data augmentation [214] is an inexpensive method that randomly transforms the input to increase the input's diversity. It has been shown to improve the generalization of the final trained model. Randomly transforming the training data will result in nondeterministic identical training runs.

**Batch Ordering:** Random batch ordering also improves the generalization of DL models. It breaks up the order of the training data to prevent the model from quickly overfitting to a particular label [139]. Reordering training batches at each epoch results in nondeterministic identical training runs.

### 4.2.3 Implementation-Level NI-factors

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

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

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

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

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

For example, matrix reduction operations such as `atomicAdd` (used in depthwise convolution layers) are affected by serialization order. Since floating-point operations are not associative due to rounding errors [89], varying orders of additions may produce nondeterministic output ($A + B + C \neq B + C + A$).

### 4.2.4 Controlling NI-factors for Determinism

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

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

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

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

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

Finally, one solution to achieve complete deterministic training is forcing the DL system to run completely in a serial manner (i.e., running on a single CPU thread). However, this option prevents DL systems to utilize the hardware efficiently and may be unrealistic, as many models would take months or years to train on a single CPU thread. As future work, deterministic multithreading [60] may be promising for more realistic deterministic DL systems.

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

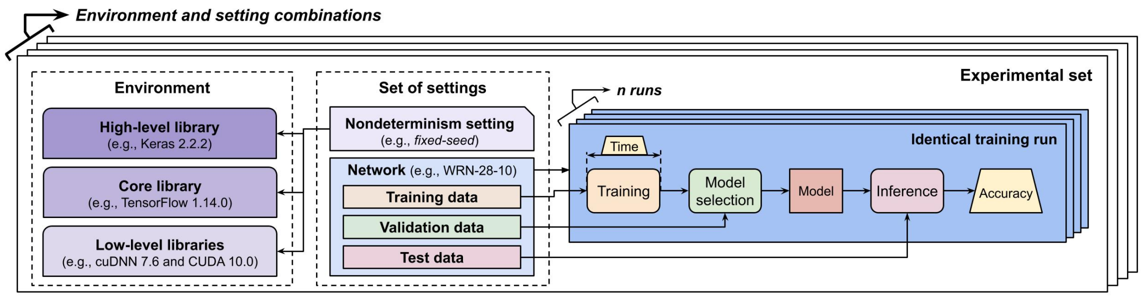


Figure 4.1: Overview of the experimental method

## 4.3 Experimental Method

First, we extract the *default input* (i.e., training data, hyper-parameters, and optimizers) of a DL system from existing work (Section 4.4). Figure 4.1 shows an overview of our experimental method. We generate different *environments* that combine different versions of DL libraries (e.g., high-level libraries, core libraries, and low-level libraries). For all environments, the hardware is the same (details in Section 4.4). For example, one such environment includes Keras 2.2.2, TensorFlow 1.14.0, cuDNN 7.6, and CUDA 10.0. Each network is coupled with its default input. For example, the CIFAR 100 dataset (including training, validation, and test data) is used to train the WRN-28-10 network with stochastic gradient descent (SGD) optimizer in 50 epochs. Table 4.1 shows the corresponding default input for each network. Each network (including its default input) combined with one *nondeterminism setting* (details below) is defined as a set of *setting*. For example, one set of settings that we use is training the WRN-28-10 network with all algorithmic NI-factors disabled (i.e., fixed-seed nondeterminism setting). For each environment and setting combination, we perform an experimental set and measure the accuracy and time variance across $n$ identical training runs.

To ensure a valid study, we address one main challenge: to measure realistic variance, the experiments need to reflect the real usage of DL systems. We pick training from-scratch as our scenario for the training phase because it is a common and fundamental training scenario [27, 100, 209, 216, 226, 247, 260], i.e., we train a new DL model from the beginning, starting from randomly initialized weights.

In addition, we focus on studying the variance of models' overall accuracy, per-class accuracy, and training time, as these are common metrics that DL researchers and practitioners use, and there have been many techniques [27, 46, 75, 91, 226, 256] proposed to improve on these metrics.

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

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

### 4.3.1 Experimental Sets of Identical Training Runs

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

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

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

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

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

An *experimental set* is a group of identical training runs. We make sure to avoid measurement bias [164] as much as we could by using the same machine along with Docker environments that are built from the same base image. The only changes across experimental sets are the DL library combinations and the set of settings (i.e., the nondeterminism-level, the network, and its default input). In each experimental set, we perform $n = 16$ *runs*.

### 4.3.2 Nondeterminism-Level Settings

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

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

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

### 4.3.3 Metrics and Measurements

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

Table 4.1: Datasets, networks, and training settings

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

### 4.3.4 Statistical Tests

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

**Mann Whitney U-test** is a statistical test to assess the similarity of sample distributions. We run the U-test instead of the T-test because the U-test does not assume normality while the T-test does. For example, when comparing two sets of runs (A and B), the null hypothesis is that the accuracies of set A are similar to set B. If we find that p-value $< 0.05$, then we can confirm with 95% confidence that our alternative hypothesis is true which means set A has statistically different accuracies than set B. We compute the effect size as the Cohen's $d$ [56] to check if the difference has a meaningful effect ($d = 0$: no effect and $d = 2$: huge effect [197]).

## 4.4 Experimental Settings

**Datasets and models:** We perform our experiments using three popular datasets: MNIST [138], CIFAR10 [126], and CIFAR100 [126]. We choose image classification architectures as they are often used as test subjects in recent SE papers that test [86, 146, 151, 182, 220, 231, 232, 235, 251, 258], verify [93, 177], and improve [82, 119, 152, 237, 249, 254] DL models and libraries. Table 4.1 shows each dataset with the corresponding numbers of instances in each subset (training, validation, and test). The training set is used to train the model. Following common practice [27, 50, 91, 100, 127, 215, 224], we use the validation set to select the best model. The test set is used to evaluate the final model.

We choose to experiment on LeNet [138], ResNet [100], and WideResNet [247] architectures as they are popular networks for image classification. In our literature review, of the 225 relevant papers, 64% of papers use or compare to one of these architectures. Table 4.1 also shows the number of trainable parameters for each network. Our networks are diverse in size, from 7,206 (LeNet1) to 36,536,884 parameters (WRN-28-10).

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

We list the training configurations used in table 4.1. To ensure that the model will converge (i.e., training loss stops improving) within the maximum number of epochs (Table 4.1), we empirically choose a maximum number of epochs larger than the number of epochs to convergence using both selection criteria.

We cannot use networks from prior work [180, 182, 231] since they only provide pretrained models and do not provide enough details for us to reproduce the training runs.

**DL libraries:** We use Keras version 2.2.2 [49] as our high-level library since it provides us with the ability to transparently switch between three DL core libraries (TensorFlow [26], CNTK [37], and Theano [200]). This ensures that the comparison across core libraries is fair and the least affected by our code. We perform our experiments with the official TensorFlow versions (including the latest) (1.10, 1.12, and 1.14), CNTK version (2.7), and Theano version (1.0.4). We pair each version of the core libraries with the officially supported low-level cuDNN and CUDA versions. For example, TensorFlow 1.12 supports cuDNN 7.3 to 7.6 coupled with CUDA 9.0, while TensorFlow 1.14 supports only cuDNN 7.4 to 7.6 coupled with CUDA 10.0. Since it is not practical to perform experiments on all library combinations, we use 11 library combinations for TensorFlow, one combination each for CNTK and Theano.

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

Table 4.2: Maximum differences of *overall* and *per-class* accuracy among *default* and *fixed-seed* identical training runs

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

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

## 4.5 Results and Findings

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

### 4.5.1 RQ1: How Much Accuracy Variance Do NI-factors Introduce?

To investigate the variance caused by NI-factors, we run 16 default identical training runs for each of the 66 experimental sets (i.e., combinations of 6 networks and 11 environments). Recall that default identical training runs are defined as training runs with the same default inputs where no NI-factors are disabled (Section 4.3.1).

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

Table 4.2 (*Default*) shows results for RQ1. Columns *Diff* show the maximum differences of accuracy while columns *SDev* and *(SDevCI)* shows the standard deviation of accuracy

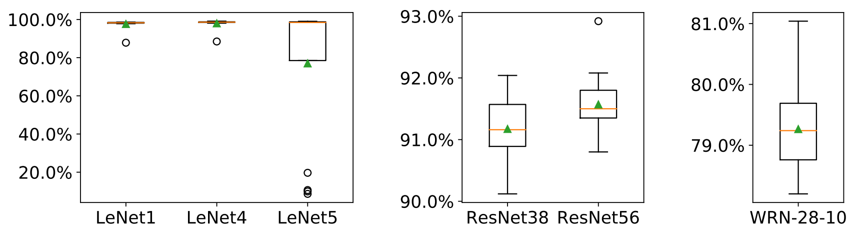


Figure 4.2: Boxplots of the overall accuracy for default identical runs with the largest overall accuracy difference

among 16 identical training runs and corresponding confidence interval (i.e., with 90% confidence, the confidence interval would contain the population standard deviation). We only show the larger accuracy differences when using either selection criterion (best-loss or best-accuracy) as the results are similar between the criteria. Figure 4.2 shows the boxplots of the overall accuracy of each network. The triangles represent the mean accuracy and the orange line is the median. Dots outside of the whiskers are outliers.

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

Specifically, in the LeNet5 default training experimental set (with TensorFlow 1.14.0, CUDA 10.0, cuDNN 7.5, and loss selection criterion), the most and least accurate runs have an overall accuracy of 99.0% and 8.6% respectively (a 90.4% difference). The worst model's accuracy is lower than random guesses (i.e., 10% because the MNIST dataset has 10 classes). This large accuracy difference is caused by the random initialization of the weights [221, 229]. Particularly, four runs do not improve much after training—with the final models' accuracies being 8.6%, 9.9%, 10.6%, and 19.7% (the outliers shown as circles in Figure 4.2). While the four runs produce weak models, they are faithful reproductions of training with widely-used networks and algorithms using realistic data and settings. The fact that 4 out of 16 runs fail to improve significantly, shows the *importance of reporting the variance* between multiple identical training runs so that *the DL approaches can be evaluated on not just their best accuracy, but also on how stable the training process is.*

If we exclude networks with such weak models, we still see an accuracy difference up to 10.8% with LeNet1 (the difference between 87.8% and 98.6%). For WRN-28-10, the largest difference is 2.8% (between 78.2% and 81.0%) respectively. Although these differences may seem small, researchers [133] report improvements of 0.8% when comparing against WRN-28-10 without accounting for NI-factors. At best, the comparison conclusions still hold, but the papers fail to provide evidence for that.

The per-class accuracy differences are even larger compared to the overall accuracy differences (Table 4.2, column *Default: Overall* versus column *Default: Per-Class*). On the least accurate run of LeNet5, the trained model fails completely on a single class (i.e., the prediction accuracy for the class digit “0” is 0%), while, for other runs, the highest prediction accuracy for the same class is 100%. Digit “0” has 261 test images (all classes have similar numbers) so such single-class failures are not due to insufficient instances or bias distribution of that class. A similar single-class failure happens for LeNet1 and LeNet4 training runs. The standard deviation is smaller for these networks (24.5% and 24.7% comparing to 44.5%) because only one run completely fails.

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

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

### 4.5.2 RQ2: How Much Accuracy Variance Do Implementation-Level NI-factors Cause?

**Accuracy variance:** We analyze nondeterminism introduced by implementation-level NI-factors by performing 66 experimental sets (i.e., combinations of 6 networks with 11 environments) of fixed-seed identical training runs (each with 16 runs). Recall that fixed-seed identical training runs are default identical training runs with algorithmic NI-factors disabled using fixed random seed initialization (Section 4.3.2).

Table 4.2 (*Fixed-seed*) shows the largest accuracy differences of the overall and per-class accuracy of all models (for any library combinations and selection criteria) with disabled algorithmic NI-factors (i.e., among fixed-seed identical training runs).

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

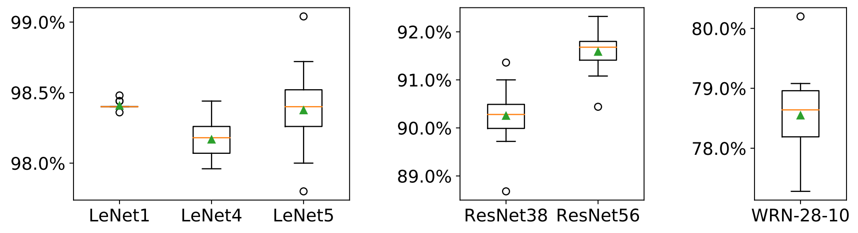


Figure 4.3: Boxplots of the overall accuracy for fixed-seed identical runs with the largest overall accuracy difference

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

The lack of complete failure caused by the random weight initialization (an algorithmic NI-factors) in LeNet training (Figure 4.3) indicates that training is more stable without algorithmic NI-factors.

When comparing the results of setting *Default* and *Fixed-seed* in Table 4.2, LeNet and ResNet56 have smaller overall and per-class accuracy differences among default identical training runs. While for ResNet38 and WRN-28-10, the accuracy differences among fixed-seed identical training runs are smaller. Levene's test cannot statistically confirm the significance (p-value $>$ 0.05) of these differences in variance for all networks except for LeNet5 (where there are complete failures in identical training runs with default setting).

Table 4.2 (*Fixed-seed*) shows that except for ResNet38, the more complex a network is (i.e., more trainable parameters), the larger the accuracy (overall and per-class) variance will be across fixed-seed identical training runs. For more complex networks, the error introduced by nondeterminism might propagate further.

To demonstrate the importance of performing identical training runs when comparing different DL approaches, we consider a scenario where ResNet56 is a baseline approach to the CIFAR10 image classification problem and ResNet38 is the proposed improvement. Among 16 fixed-seed identical training runs, ResNet56 averages 91.2% in test accuracy while ResNet38 averages 90.3%. The U-test confirms (with p-value $<$ 0.01) that ResNet56 has 0.9% higher test accuracy than ResNet38 with an effect size (Cohen's d) of 1.7 (very

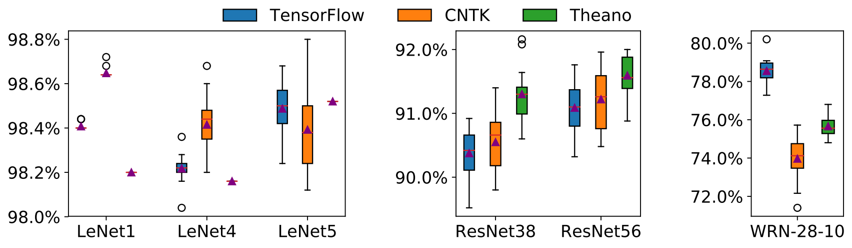


Figure 4.4: Boxplots of the overall accuracy of fixed-seed identical training runs with different core libraries

large effect). Hence, there is no improvement from the proposed technique (ResNet38) over the baseline (ResNet56). However, if each approach only runs once, in the most extreme case, ResNet56 accuracy is reported with its worst run (90.4%) and ResNet38 with its best run (91.4%), the researchers might have come to an invalid conclusion that ResNet38 has 1% higher accuracy than ResNet56. Researchers and practitioners should be aware of DL system variance, even with only implementation-level NI-factors, so they must perform multiple identical training runs when comparing approaches.

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

Figure 4.4 shows the boxplots of the overall accuracy of fixed-seed identical training runs for the experimental set of each network with the best-loss selection criterion across three different core libraries. The accuracy variance is similar across different core libraries. For example, for ResNet56 the accuracy difference with CNTK is 1.5% (between 91.8% and 90.3%) and 1.9% with TensorFlow. All core libraries are affected similarly by implementation-level NI-factors: Levene's test cannot reject the null hypothesis that each core library has a different accuracy variance (p-value $> 0.1$).

**Different low-level libraries versions:** We analyze the overall accuracy differences of the 11 low-level library combinations (cuDNN and CUDA) with TensorFlow to see if there is still variance when switching versions of the low-level libraries. Figure 4.5 shows the boxplots of the overall accuracy differences of fixed-seed identical training runs when training each network with each of the 11 library combinations. All training runs are affected by implementation-level NI-factors, independently from the low-level libraries used.

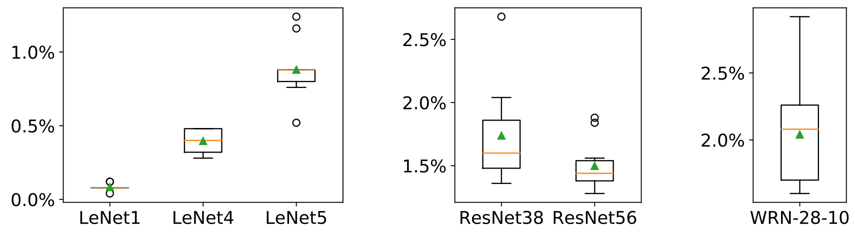


Figure 4.5: Boxplots of the overall accuracy difference of fixed-seed identical training runs with 11 low-level library version combinations for each network

For example, with WRN-28-10, the largest overall accuracy difference is 2.9% (reported in Table 4.2). On average, across 11 experimental sets, the accuracy difference for this network is over 2% while the smallest accuracy difference is 1.6%.

### 4.5.3 RQ3: How Much Training-Time Variance Do NI-factors Introduce?

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

Table 4.3 shows the analysis of the running time to convergence for default identical training runs and fixed-seed identical training runs. $\text{Time}_{Loss}$ and $\text{Time}_{Acc}$ denote the training time using two popular model selection criteria—best-loss and best-accuracy, respectively. For each selection criterion, the table shows the time difference between the slowest and the fastest runs (columns *Diff*). Since the running time is very different across networks, we compute the relative time difference (columns *RelDiff*)—the ratio of the time difference over the running time of the fastest. To give some indication of an average case, columns *RelSDev* show the relative standard deviation (i.e., coefficient of variation [74]) of the 16 runs.

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

Table 4.3: Running time to convergence differences among *default* and *fixed seed* identical training runs

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

* 3/16 runs stuck at the first epoch

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

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

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

## 4.6 Researcher and Practitioner Survey

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

### 4.6.1 Survey Design and Deployment

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

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

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

All questions and definitions are included in the GitHub repository whose link is provided in Section 4.1.

### 4.6.2 Survey Results and Findings

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

**Awareness of NI-factors in DL systems:** We ask Question 20: *"In your opinion, are DL systems deterministic?"* to gauge the awareness that participants have of the NI-factors (results in Figure 4.6).

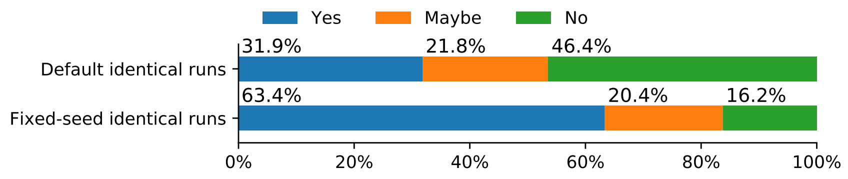


Figure 4.6: Distribution of responses to Question 20 (*Default identical runs*) and Question 26 (*Fixed-seed identical runs*)

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

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

**Awareness of implementation-level NI-factors in DL systems:** We design Questions 26: *"Do you expect fixed-seed identical DL training runs to be deterministic?"* to study how aware respondents are with implementation-level NI-factors (results in Figure 4.6).

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

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

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

Figure 4.7 shows participants' estimations of the overall and per-class accuracy differ-

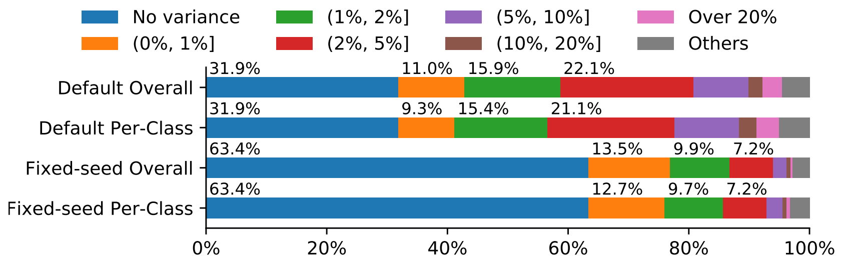


Figure 4.7: Estimation of *overall* and *per-class* accuracy difference across default and fixed-seed identical training runs

ences across default identical training runs (*Default Overall* and *Default Per-Class*) and fixed-seed identical training runs (*Fixed-seed Overall* and *Fixed-seed Per-Class*). *No variance* indicates participants that are unaware of the nondeterminism of DL systems. Some participants choose "Other" and state that the accuracy difference depends on the task and network architecture.

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

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

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

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

## 4.7 DL-Training Paper Survey

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

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

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

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

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

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

## 4.8 Implications, Suggestions, and Future Work

**Improving the stability of training implementations:** Practitioners may need to control NI-factors or replay DL training deterministically to facilitate debugging, which are challenging tasks. As discussed in Section 4.2.4, algorithmic NI-factors are generally straightforward to control as they are introduced explicitly using pseudo-random number generators which can be seeded before each run. Practitioners may benefit from new methods (e.g., deterministic GPU [116]) to control implementation-level NI-factors, which are much harder to control because they are often byproducts of optimization.

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

It is nontrivial to determine the number of identical runs needed, this depends on the approaches and the baselines. One solution is iteratively performing more replication runs when comparing to a selected baseline. The replication process can stop when statistical tests (e.g., U-test) confirm the significance (e.g., p-value $< 0.05$) of the difference between the new approach and the baseline. If after a large number of replication runs (e.g., more than 30 runs [30]) the improvement is not statistically significant, then the variance might be large enough such that a statistically significant conclusion about the difference between the two techniques might not be possible. We propose such a technique (Section 4.10) to help researchers and practitioners with this process, to reduce the manual effort to conduct valid experiments and replicate experiments.

Approaches to improve reproducibility suggested by the SE community [90, 154, 233] need to also consider training variance. New approaches such as efficient checkpointing may be desirable.

Transparency is important in making sure that research is reproducible and valid. Recently, the DL research community promoted sharing artifacts and results transparently [25, 66]. Since DL systems are nondeterministic, it is important to share the data from the replication runs as well. One solution is maintaining a centralized trusted database that stores these replication runs and provides authors of new approaches with baseline results that they can directly compare to without rerunning the baseline approaches. We are developing a tool that helps users to measure the variance of their approaches and facilitates the comparison across approaches. Users can upload their replication packages and results to a database, that is provided by our tool, to be curated for comparison.

**Producing better models:** When a DL model is the contribution (e.g., defect prediction [145] or program repair [144]), practitioners could leverage variance to obtain a more accurate model.

**Less expensive training and variance estimation:** Since DL training is expensive, an important research direction could be less-expensive variance estimation and training approaches such as software support for incremental training [95].

## 4.9 Threats to Validity

**External validity:** Observed results might be different for other networks. We use 6 very popular networks with diverse complexity (from 7,206 to 36,536,884 parameters) to mitigate this issue. We encourage others (and we will strive) to replicate and enrich our studies with different DL networks, DL training approaches, and DL libraries to build an empirical body of knowledge about variance in DL systems. We are building a replication tool to help the research community to share and replicate research experiments and results.

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

**Internal validity:** Our implementation or the libraries we used might have bugs. This should be alleviated by that (1) all our code is reviewed by most authors of the paper [183], (2) our results show that variance exists for all versions of all libraries we evaluate, thus is unlikely to be caused by library bugs, (3) we focus on official releases of DL libraries, so our runs should still be representative of real DL usage, and (4) we analyze all results, especially outliers to ensure there were no implementation bugs.

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

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

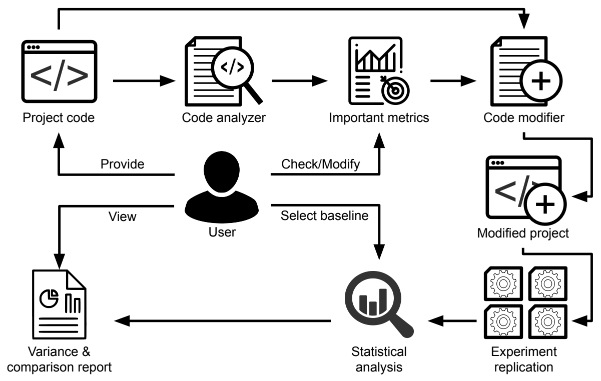


Figure 4.8: The overview of DEVIATE.

## 4.10 DEVIATE: A Deep Learning Variance Testing Framework

As suggested, DL researchers and practitioners must perform their DL training experiments multiple times with identical settings to measure the variance of their proposed approaches and then compute the statistical test when comparing to the baseline approaches. With the tested results, DL practitioners can make informed choices of techniques when applying DL properly and effectively in their applications.

While replicating identical DL experiments and measuring their variance may appear to be straightforward, this task has many research and engineering challenges. First, it requires extra effort to automatically monitor important metrics (e.g., epoch, accuracy, and loss) in training source code and record them in an organized manner among identical experiment runs. Since different users tend to use different coding styles in writing source code (e.g., a simple `for` loop or a dedicated method), automatically identifying the code location that monitors training metrics is challenging. Second, once the results from multiple experiment runs for both the proposed approach and baselines are ready, it is nontrivial to know which statistical tests to use for analyzing the variance among multiple runs of individual approaches and evaluating the improvements over baseline approaches with the variance taken into account.

We introduce the DEVIATE framework [2] which addresses these challenges. Figure 4.8 shows the overview of DEVIATE. The *code analyzer* automatically processes the training source code and extracts the variables where important metrics such as epoch, accuracy,

[2]https://https://github.com/lin-tan/DEVIATE

and loss, are assigned. DEVIATE supports an option for users to double-check the extracted metrics and select only the relevant metrics if necessary. The *code modifier* then injects the logging code into the original source code to record these metrics. DEVIATE then systematically replicates the experiments multiple times with the most identical setting possible. Once the experiments are completed, DEVIATE performs statistical tests to analyze the variance of the user's proposed approach as well as the comparison of the proposed approaches with baselines. Hence, DEVIATE enables the reproducibility of the proposed approach and ensures the validity of comparison results.

DEVIATE *automatically* analyzes the DL system source code, extracts important metrics (such as accuracy, loss...) that are monitored by the authors, and stores those metrics in a consistent format. DEVIATE performs popular statistical tests (such as confidence interval of standard deviation, MannWhitney U-test, and Cohen's $d$ effect size) and provides users with a report of how well the proposed technique performs in comparison to the selected baselines.

To demonstrate the effectiveness of DEVIATE, we perform a case study with adversarial training [153]. We found that apart from the standard training approaches, other training approaches can also have large variance. For example, with the standard training procedure, our analysis (Section 4.5) reports a difference up to 1.9% in accuracy among 16 identical training runs of ResNet56 network with fixed random seeds. However, with a more complex adversarial training process [153] that uses Fast Gradient Signed Method (FGSM) [92], DEVIATE measures a difference of robustness (accuracy on adversarial samples) and effectiveness (accuracy on natural samples) up to 5.1% and 2.0% in accuracy, respectively, among only 8 identical training runs with fixed random seeds.

## 4.11 Summary

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

# Chapter 5

# Mitigating Blindspots in Deep Learning Training Data: The N-model Approach

## 5.1 Motivation

Deep learning (DL) has been used in many practical tasks such as image processing [55], speech recognition [105], and natural language processing [222]. One reason for DL's popularity in recent years is its ability to learn from a large amount of data and perform tasks such as image classification even better than humans [99].

However, DL still suffers from the same problems as any other data-driven machine learning algorithm. One such problem is that the training dataset is not representative of the real-world application. For example, the training data for a gesture detection model may include samples of normal hand without tattoos, but does not contain a single sample of hands with a tattoo. Such a training dataset problem (i.e., blindspots in training dataset) can cause the trained model to be less robust when applied to the real world [227]. The robustness issues may lead to disproportionate error rates for different demographic groups causing fairness concerns [40], risks in safety-critical applications and unreliable behavior that may break user trust.

One possible way to detect such a blindspot problem is by looking at how uncertain the trained model is when predicting an unseen sample. Specifically, there are two kinds of test errors that a predictive model would make, the *known unknowns* and *unknown*

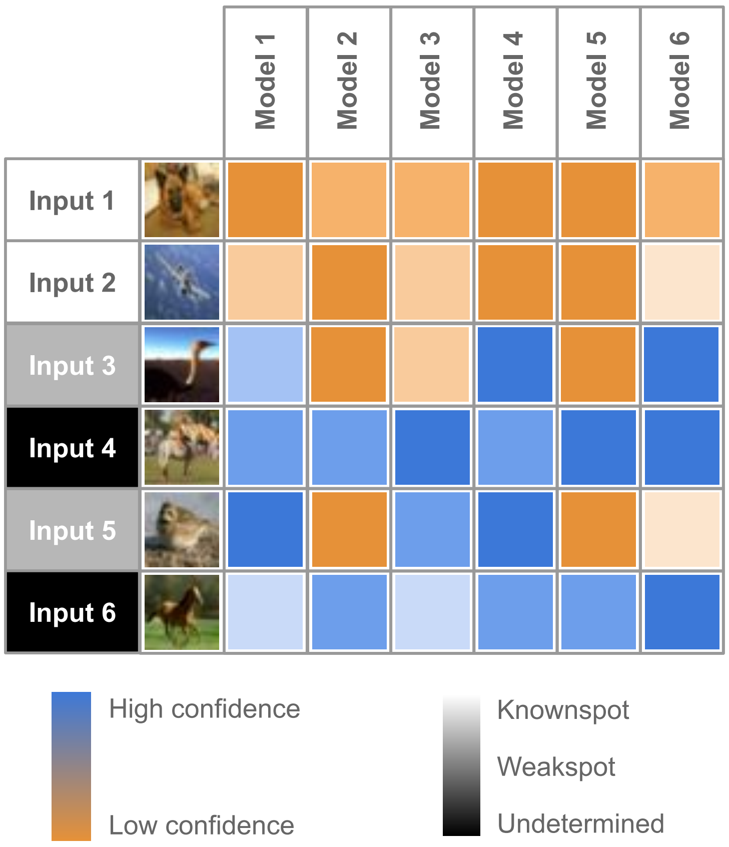


Figure 5.1: A demonstration on how N models can provide more information to select useful additional sample when compare to a single model. Each square shows the highest confidence level of the corresponding model's prediction for corresponding input.

*unknowns* [34]. A known unknown is an input that the model is uncertain about (i.e., low-confidence prediction). Figure 5.1 shows examples of such known unknown errors. Specifically, with respect to Model 1, a test Input 1 is a known unknown if Model 1 is uncertain about its prediction (i.e., low-confidence prediction which is presented with the orange color box). This means, a low-confidence prediction (i.e., known unknown) would indicate a blindspot sample (i.e., a sample that cannot be generalized from the training data). We call such samples *knownspots*. An unknown unknown, on the other hand, is an input that the model makes an unknowingly incorrect prediction with high confidence. Figure 5.1 shows examples of some high-confidence prediction. Specifically, with respect to Model 1, Inputs 3 to 6 have high confidence (presented with blue color boxes). However, a high-confidence prediction might also indicate that the unseen sample can be generalized from the training data. We call samples with high-confidence predictions *undetermined* since, without ground truth labels, it is not possible to know if high-confidence samples can be helpful in improving the model. In general, based on predictions from the Model 1 (a single model), Input 1 and 2 from Figure 5.1 are considered useful additional data to improve such a model.

In general, it is not possible to detect or address the "strong" blindspots (e.g., gesture detection training data does not contain a single sample of hands with a tattoo) by just analyzing the trained model, since the model only has access to the same flawed training data. However, if the blindspots are "weak", the predicted confidence can sometimes indicate weakspot samples as known unknowns. Here, we consider a blindspot as weak (or weakspot) when there are weak signals that can help the model to learn. For example, these weak signals could be a few samples of the weakspot subcategories in the training data (e.g., gesture detection training data contains only a few samples of hands with tattoo) or similar features within a class that a model can learn when a subcategory is missing completely (e.g., ray fish samples have the similar water background as some other fish samples). However, due to the stochastic nature of the optimization process, if we only analyze a single trained model, depending on the optimization path, such weakspot samples might still be considered as undetermined (e.g., Input 3 in Figure 5.1 is considered undetermined if we only analyse Model 1 on its own). Instead, we should analyze across multiple training runs. Then the variance of the confidence when predicting an unseen samples can indicate weakspot samples. For example, Input 3 in Figure 5.1 would represent a weakspot since it is considered as known unknown for Model 2, 3, and 5 while it is not considered as known unknown for Model 1, 4, and 6.

Intuitively, for each training run, the optimizer would select a different local optimum. From our observation, there are test samples that are correctly predicted consistently across multiple models (i.e., with a low variance of confidence level across multiple models) which are well generalized from the training data. This observation is consistent with recent research of the core set [203]. However, some samples are predicted with large variance in confidence (inconsistent samples). These samples could belong to the underrepresented subcategories (e.g., the "ray" fish in the example above) and each local optimum has to make compromises to focus on a few of those samples. Such samples could indicate weakspots (i.e., missing subcategories) in the training dataset. This is consistent with prior work which shows that training DL models with different random seeds produce different models that perform differently [183].

Building on this result, we propose a more general approach that is not reliant on strategic dropout layers or specific neural network architecture (e.g., convolutional neural network, recurrent neural network, reinforcement learning) [81], and thus can be applied to any deep learning model as a black box method. Our N model approach uses the stochastic process of training to estimate the uncertainty of the prediction by computing the variance of the confidence across the sample of multiple models (i.e., trained with different seeds) instead of measuring the uncertainty using only one model. Our approach can detect weakspots because of the stochastic nature of the DL optimizer (i.e., stochastic gradient descent) and the nonconvex nature of the optimization problem (e.g., image classification) leading to weakspot samples to be unknown unknown on one training run but known unknown on another.

To evaluate the proposed N model approach, we follow prior work [196] and first create candidate holdout training datasets that could contain weakspots of different severity by removing (i.e., holding-out) all or a portion of data samples of specific subcategories (e.g., "ray" samples from "fish" category). We then further analyze those candidate holdout datasets to measure the severity of the introduced weakspots. We found that removing certain subcategories such as "maple tree" from the "tree" category does not affect the generalization of the trained model when classifying "maple tree" test samples (accuracy drop of only 4.6%). However, when removing "mushroom" samples from the "fruit and veggie" category, the trained model can not predict well the "mushroom" sample (accuracy drop of 70.1%). Such analysis indicates that the holdout dataset of "mushroom" containts a more severe weakspot when compared to the holdout dataset of "maple tree" (Finding 0).

We then perform experiments to answer the following research questions:

- Can an input-sample score with N models better indicate a dataset weakspot sample than a traditional score?
- Can incorporating the multiple models analysis in the development process improve the DL model with retraining?
- Can analysis of samples with high variance reveal the characteristic of the blindspots?

The experiments show that our N model approach is able to detect weakspot samples better than a single model approach (Finding 1). By applying retraining, our N model approach improves the holdout test accuracy by up to 25.2% with only 2% added training data (Finding 2). In our qualitative analysis, we find that by inspecting the unseen samples with high uncertainty across multiple models with the help of attention visualization methods such as Grad-CAM [201], the developers could gather some insight into why the model fails to generalize and how to improve the model.

In this chapter, we make the following contributions:

- A new procedure that leverages the variance of training N models to detect and mitigate the weakspots in training data
- A set of training datasets with artificially induced weakspots along with the analysis of their characteristics **(Finding 0)**
- An evaluation of the N model ranking metrics that shows:
  - **Finding 1.a**: N model metrics are effective at predicting weakspot samples in scenarios where the weakspots in the training data are severe.
  - **Finding 1.b**: N model metrics are better indicators of blindspot samples comparing to the single model metric on average and in the worst case.
- An evaluation of the N model retraining procedure that shows:
  - **Finding 2.a**: The ranking based on the variance of predictive confidence enables retraining to gain the most holdout accuracy in scenarios where the original training data have a significant weakspot (up to 25.2% with just 2% of additional data).
  - **Finding 2.b**: The ranking based on the variance of predictive confidence does not affect the test accuracy in other classes after retraining.
- A qualitative analysis of the "ray" weakspot scenario that shows the benefit of investigating high variance input samples to reveal insights that help DL developers improve their training dataset and model.

## 5.2 Approach

Our N model approach leverages multiple models to rank and select the most useful unlabeled samples that can provide useful information for DL deverlopers to improve their training dataset. Additionally, these samples can be added to the original training data to improve the original models with additional training. Our approach applies both uncertainty based and consistency based ranking to select the least amount of additional data (reducing the labeling cost) while improving the original model the most.

### 5.2.1 Ranking Unlabeled Samples

During the DL model development, one way that the developers could improve their model is to examine validation samples that perform badly (e.g., incorrectly classified with very low model confidence on the ground-truth label). By analyzing such input samples, the developers would have some ideas on how to improve the model. For example, one might realise that the badly performing input samples might be important use cases that are under-represented in the training data. By collecting more inputs of such use cases, the model might perform better after retraining. Since the manual effort of examining the badly performing samples is costly, the higher problematic samples are ranked in the examination priority list, the less manual effort would be needed.

**Single model confidence score:** One traditional ranking metric is the confidence level of the predicted label. A low confidence level might indicate that the model has difficulty recognizing similar samples. However, low confidence levels could be due to many reasons, including limitations of the model. If we aim to detect samples that indicate a training data problem, a single score of a single model might not be the best metric.

**Average of N models confidence score:** As mentioned previously, DL model training is not deterministic. Different training runs of different random seeds will produce different models with different predictions. This is because the optimizer selects different optima each run due to different starting conditions. Hence, a more consistent metric to indicate a problematic sample is the average confidentce level across N models from the N different training runs.

**Variance of N models confidence score:** From our observation, there are a set of samples that is correctly predicted consistently across multiple models (consistent samples). This observation is consistent with recent research on core sets [203]. However, there are samples that are predicted correctly by only a few models (inconsistent samples). These samples could be "harder samples" to predict and each local optimum makes compromises to focus on a few of those samples. A metric that measures the variance of confidence levels across N models could be a good metric to indicate such "harder samples".

### 5.2.2 Investigating Interesting Unlabeled Samples

One way to benefit from ranking scores is to manually investigate highly ranked samples to find useful insights on how to improve the training dataset (e.g., those examples might indicate an under-represented population data that is missing from the training dataset). In this work, we evaluate such a usecase of the ranking scores by computing the predictive power of the holdout sample (i.e., how well they detect the weakspots that are artificially introduced.)

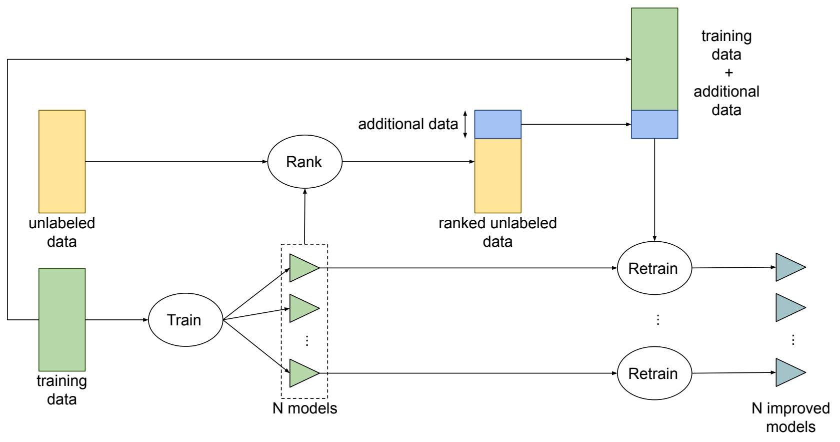


Figure 5.2: The retraining process to improve a DL system with additional training data

### 5.2.3 Retraining to Improve DL Model

Another less labor intensive way to benefit from the ranking scores is to apply them, as acquisition functions, to a retraining process that is similar to the traditional active learning procedure. Figure 5.2 shows the overview or our retraining process. The original training data (green box) is used to train the initial models. The ranking of a separate unlabeled data (yellow box) is computed based on the trained models. In the case of the average confidence score acquisition function, the lower the score (i.e., average confidence levels), the higher rank the sample will be, thus would be likely to be selected in the retraining step. In the case of the confidence score standard deviation acquisition function, the higher the score (i.e., the variance), the higher rank the sample will be. For the N model procedure, these rankings are shared across all original models, however, for the single model procedure, the ranking is specific to each individual model. Once the unlabeled data is ranked, a portion of the top ranked unlabeled (blue box) samples are selected to be labeled and added to the original training data. This new extended training data is then used to train the original models with additional iterations to obtain the final models.

In both processes (single model and N model ranking), the retraining process will create multiple models. In the case of the single model ranking, these models represent multiple identical runs of the same retraining algorithm. We evaluate these using their average to show the performance on average of the single model ranking metric. In the case of the N model ranking, one of the models can be selected as the improved production system or they can be used in an ensemble. In this work, we calculate the average accuracy across the models to show the improvement of the new dataset with added data.

Table 5.1: Coarse-classes and corresponding sub-classes

| Coarse-class | Sub-classes | | | | |
|---|---|---|---|---|---|
| aquatic mammal | beaver | dolphin | otter | seal | whale |
| fish | aquarium fish | flatfish | ray | shark | trout |
| flower | orchid | poppy | rose | sunflower | tulip |
| fruit and veggie | apple | mushroom | orange | pear | sweet pepper |
| insect | bee | beetle | butterfly | caterpillar | cockroach |
| medium mammal | fox | porcupine | possum | raccoon | skunk |
| people | baby | boy | girl | man | woman |
| reptile | crocodile | dinosaur | lizard | snake | turtle |
| small mammal | hamster | mouse | rabbit | shrew | squirrel |
| tree | maple tree | oak tree | palm tree | pine tree | willow tree |

## 5.3 Experimental Settings

**Data set and blindspot scenarios construction** To evaluate our approach of improving DL system when there exist weakspots in the training data, we first need to create artificial scenarios where such weakspots are injected into the training data. We first select an original dataset that represent the complete real-world. To create a training set with an induced weakspot, following prior work [196], we artificially remove samples of a specific holdout subclass (e.g., ray fish images) which belong to a coarse class (e.g., fish images) from the original training dataset. The test set simulates the real-world application and is the same across all scenarios. This way we can evaluate and compare the effect of various training data issues on the same test set.

Holding out a subset of the training data does not guarantee the creation of weakspots since the remaining samples could provide enough information for the model to generalize. For example, a DL model could learn features from adult faces and generalize well with faces of boys and girls. So, to validate that each holdout dataset indeed contains weakspots, we train a DL model on each holdout dataset and measure the performance of the model on the holdout test set. For example, we would remove a portion of the training images of the ray fish to create a holdout dataset of the ray fish. We then evaluate the trained model on a holdout test set containing only test images of the ray fish. If the amount of removed ray images from the training data correlate with the drop in the model accuracy on the test holdout set, we successfully inject weakspots in the holdout scenarios of ray fish.

Since it is difficult to get a perfect dataset that represents a real-world distribution, we use an existing dataset as an alternative to a real-world distribution. In this case, we select CIFAR100 [126] dataset for its popularity and manageable size as well as its hierarchical label structure, which facilitates the creation of realistic weakspots in the training datasets. ImageNet is another popular dataset with hierarchical labels, but since it is much more expensive to train ImageNet models, using such a larger dataset would reduce the number of scenarios that we could explore. CIFAR100 has 60,000 images, but with 20 coarse-classes, each has 5 sub-classes for a total of 100 sub-classes. Of the 20 coarse-classes, we select 10 coarse-classes that visually represent their sub-classes. For example, we remove the coarse-class "large man made outdoor things" which contains "bridge", "castle", "house", "road", "skyscraper" because these sub-classes are not visually similar. Table 5.1 shows the 10 selected coarse-classes and their sub-classes.

**Training and retraining settings** For each scenario, we train 25 models with the same training data but with different random seeds. We then evaluate the 25 models using the test sets.

We use a popular ResNet18 network as our experimental DL architecture because of its faster training time without too much accuracy compromise. While we believe the results should generalize, extending the approach to other DL architectures remains as future work.

The ResNet18 models are trained initially with 150 iterations, and later in the retraining step, are trained with additional 50 iterations. The optimizer is started at the correct training iteration, so that the correct learning rate according to the scheduler is selected to prevent the optimization process from updating the parameters too aggressively, which can create sub optimal retrained models.

**Hardware and software** The experiments are run on a cluster with Tesla V100 GPUs. We use Python 3.6 with Pytorch 1.4 to train and retrain the models. Overall, it took over a week on the 32 GPU cluster to run all experiments.

## 5.4 Results and Findings

First, we perform experiments to analyze the characteristic of the artificial weakspot scenarios RQ0). Second, we investigate if input-sample scores with N models better indicate a dataset weakspot sample than a traditional score. Third, we perform retraining using those ranking scores to see which scores provide the best improvement given the same resources. Finally, we discuss an usecase of analyzing the samples with high variance to reveal insight that can help developers improve their training data.

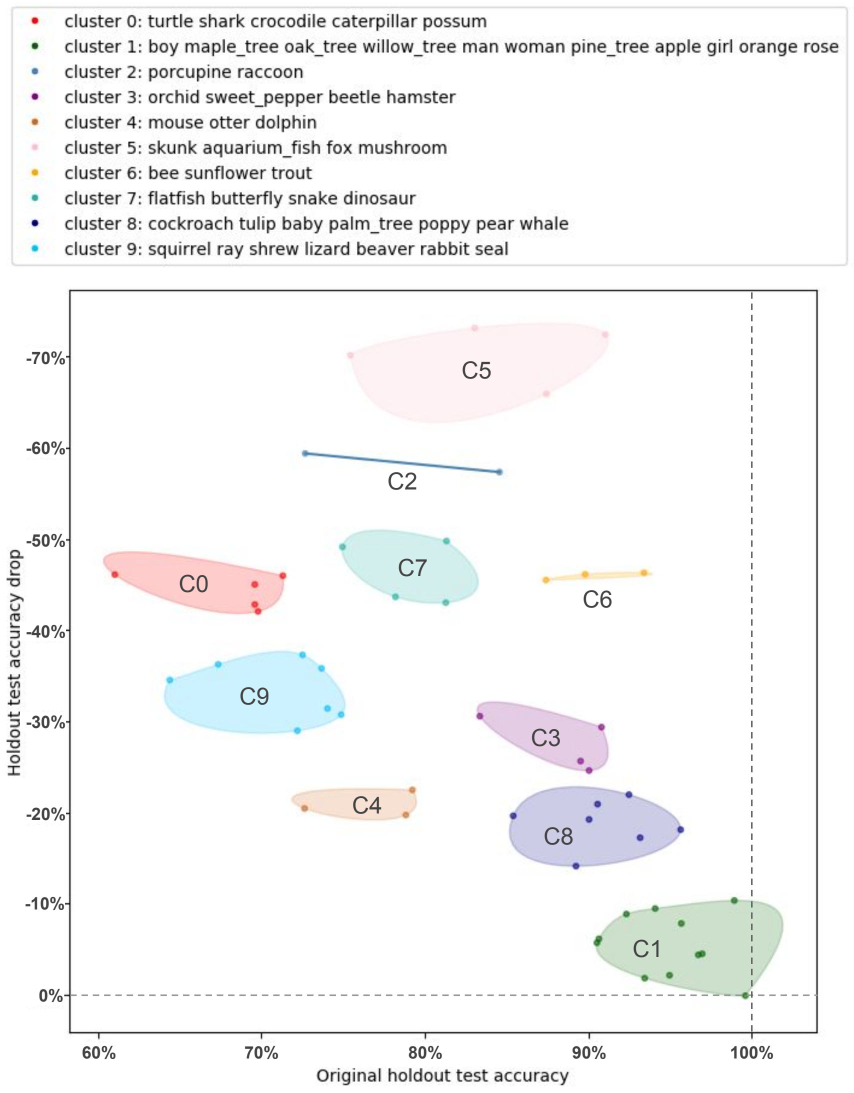


Figure 5.3: The clustering of 50 sub-classes based on the holdout accuracy drop and original holdout accuracy when they are used as 100% holdout sub-classes.

### 5.4.1 RQ0: Are All Holdout Scenarios Blindspot Scenarios?

To evaluate our approach of improving DL systems that suffered from blindspots in the training data, we first create holdout scenarios by removing training samples of sub-classes in the CIFAR-100 dataset. Specifically, we create 11 holdout scenarios (varying in the holdout portion from 0% to 100% with interval of 10%) for each of the selected 50 sub-classes, for a total of 550 scenarios.

For each holdout scenario, we train 25 ResNet18 models to predict the ten coarse labels. We then calculate the average accuracy of these trained models on the holdout test set.

Figure 5.3 shows the clustering of the 50 sub-classes with respect to their holdout test accuracy. The x-axis represents the original holdout test accuracy (i.e., holdout portion of 0%) and the y-axis represents how much the original accuracy drops in the complete holdout scenario (i.e., holdout portion of 100%). We notice that the removal of some sub-classes does not affect the holdout test accuracy as much as some other sub-classes. For example, sub-classes of cluster 1 have a small drop in holdout accuracy of less than 10%. This result makes sense since human faces are quite general across age groups and genders, hence, the DL model generalizes well with the remaining sub-classes. That means removing all training images of girl from the people coarse-class does not greatly reduce the trained model holdout test accuracy, as they are generalized well enough to recognize a girl as a person even if they have only been trained on images of boy, man, woman, and baby. These holdout scenarios do not represent a blindspot scenario in our study, as the missing samples do not induce much loss in the models' performance.

On the other hand, cluster 5 has a very large drop in accuracy: around 70%. Sub-classes such as mushroom are quite different from other sub-classes such as apple, orange, pear, sweet pepper in the fruit and vegetable coarse class. That means removing training samples of the mushroom reduces the model's ability to recognize the images of mushroom as fruit and vegetable. These kinds of holdout scenarios represent blindspot scenarios where the missing samples have a significant impact on the model accuracy and these would be scenarios where we want to focus our effort on improving.

**Finding 0**: Different holdout scenarios with different holdout classes have different blindspot severity. Removing all sample of subclasses in cluster C1 induces at most a 10% accuracy drop. However, removing all samples of subclasses(such as mushroom) in cluster C5 induces a large accuracy drop of up to 70%.

### 5.4.2 RQ1: Can Input Sample Scores with N Models Better Indicate a Dataset Blindspot Sample than Traditional Single Model Scores?

We hypothesise that the ranking scores could be used to rank the unseen samples so that the more related samples to the weakspots appear higher than the unrelated ones. To test

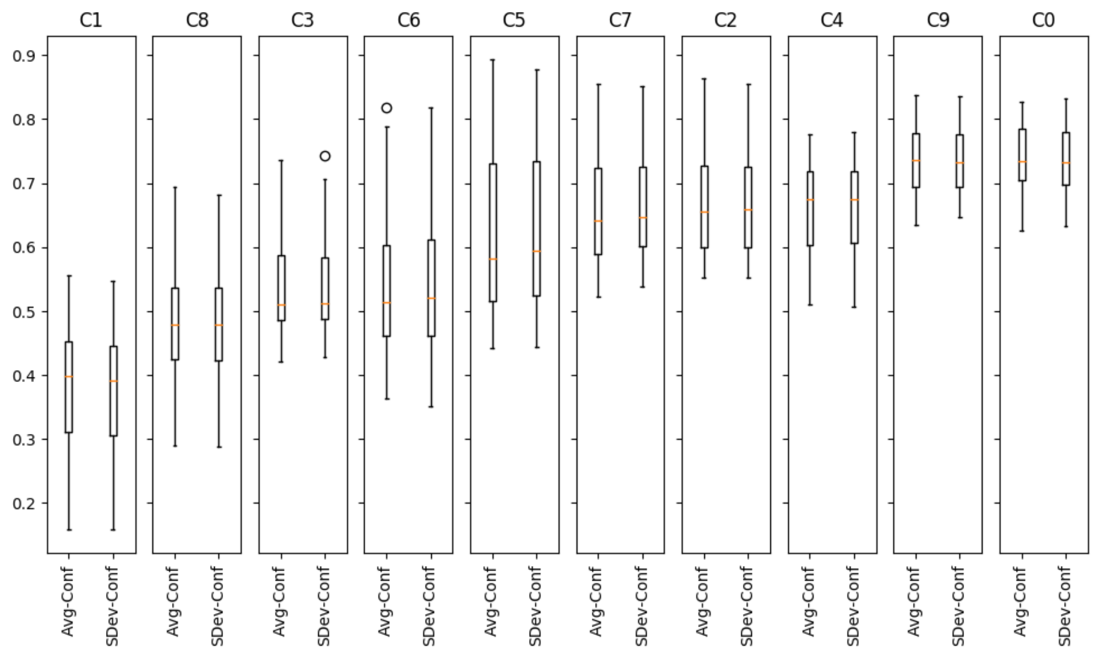


Figure 5.4: Box plot of AUC when using N models metrics to detect blindspot samples

this hypothesis, we apply the ranking scores to our artificially generated weakspot scenarios where we know the holdout samples are the injected weakspots. We then measure how well the ranking scores can detect such holdout samples (i.e., assigning higher ranking to holdout samples when compare to other sub-classes) by computing the Area Under the Receiver Operator Characteristic Curve (AUC).

Specifically, in this RQ, we investigate if N model metrics such as the variance and the average of predicted confidence levels across N models can indicate a weakspot samples and if they are better than the single model metric. Figure 5.4 shows the box plot of the AUC for each cluster of holdout classes when using the N model metrics—the variance (*SDev-Conf*) and the average (*Avg-Conf*) of predicted confidence levels across N models. The clusters are sorted with increasing effectiveness (i.e., AUC) of the metrics.

For clusters C9 and C0, the N model metrics are quite effective in detecting blindspot samples (with median AUC of more than 70%). This is a good indication that N model metrics can be used to indicate if a test sample represents a blindspot or not, as C9 and C0 are two clusters of holdout classes where there exists a blindspot in the training data (i.e., we see a large drop of 40-50% in accuracy when those samples are removed from the training data—refer to RQ0 for details).

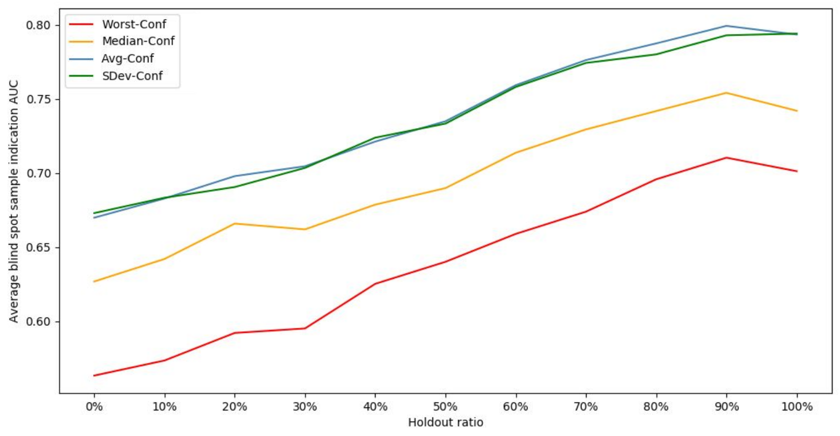


Figure 5.5: Average AUC across all holdout classes in cluster C0 when using different metrics

On the other hand, for clusters C1 and C8, the metrics are not effective in predicting the blindspot samples. This is consistent with the fact that for scenarios in clusters C1 and C8, the accuracy drops are small (less than 10%), indicating that the blindspot does not exist in the training data. Without a blindspot in the training data, it is understandable that the metrics would not be able to indicate the holdout samples.

One main observation that we have is the effectiveness of our N model metrics in predicting holdout samples seems to correlate to the original accuracy of the holdout sub-classes. This is shown by the matching of the clusters order in Figure 5.4 and the x-axis order of the clusters in Figure 5.3. We hypothesize that if the model is less accurate on a sub-class (e.g., in cluster C0), the N model training process is more likely making compromises and enables our N model based metrics to perform better. For more accurate sub-classes (e.g., in cluster 1), the variance is more likely smaller, which makes our N model metrics less effective. In this case, cluster C5 has the largest accuracy drop in Figure 5.3. However the N model ranking on scenarios in this cluster is average, which is correlated with their average holdout sub-classes' original accuracy.

**Finding 1.a**: N model metrics are effective at predicting blindspot samples in scenarios where the blindspots in the training data are severe (e.g., in clusters C0 and C9). On the other hand, they are less effective in scenarios where the severity of the blindspot are very low (e.g., in clusters C1 and C8).

To see if the N model metrics are better in predicting the blindspot samples, we compare

Table 5.2: Average holdout accuracy increase after retraining with 2% additional data for scenarios with 100% holdout ratio

| Cluster | Random | 1-model | N-models | |
|---|---|---|---|---|
| | | Conf | Avg-Conf | SDev-Conf |
| C1 | 0.4% | **1.2%** | 0.9% | 1.1% |
| C8 | 2.6% | 5.9% | **6.1%** | 5.2% |
| C3 | 3.1% | 6.3% | **7.0%** | 5.9% |
| C6 | 10.7% | 20.0% | **22.7%** | 20.7% |
| C5 | 7.6% | 20.7% | 24.7% | **25.2%** |
| C7 | 3.7% | 9.7% | **11.3%** | 10.9% |
| C2 | 9.0% | 18.5% | 20.4% | **21.2%** |
| C4 | 1.4% | 3.7% | **3.8%** | 3.4% |
| C9 | 2.0% | 5.2% | **6.3%** | 5.4% |
| C0 | 3.5% | 6.7% | **7.2%** | 6.6% |

the AUC of N model metrics with the AUC of single model metric. Figure 5.5 shows the average AUC across all holdout classes of cluster C0 with different holdout ratio for both N model metrics (Avg-Conf and SDev-Conf) and single model metric. Because there are 25 models, we show the average case (Median-Conf) and worst case (Worst-Conf) of the single model metric. The N model metrics have on average 5% better AUC than the average single model metric and over 10% better AUC than the worse single model metric. One could say that our method could be less effective than the best single model, however in practice it is not possible to know which is the best model without knowing the ground truth. So in this case, our N model metrics out perform the single model in most cases. The figure also shows that the more severe the blindspot (i.e., higher holdout ratio), the better the metrics at predicting blindspot samples.

**Finding 1.b**: N model metrics are better indicators of blindspot samples compared to the single model metric on average and in the worst case.

### 5.4.3 RQ2: Can Incorporating Multiple Model Analysis in The Development Process Improve The DL Model with Retraining?

One way to improve a DL system is to add additional training data. However, labeling training samples is an expensive process. Instead, we can use the metrics we evaluate in RQ2 to rank and prioritize examples to be labeled, so that we can improve the DL model with retraining with additional data using minimum labeling effort (more details in section 5.2.3)

For this RQ we use random selection as the baseline acquisition function (i.e., we rank the unseen samples to add with random scores). For the single model, each of the N models has its own acquisition function based on its predicted confidence of the unseen samples (Col. Conf). Finally, the two acquisition functions based on N models are the average predicted confidence (Col. Avg-Conf) and the standard deviation of predicted confidence (Col. SDev-Conf) across N models. Table 5.2 shows the average improvement from before retraining (over the $N = 25$ models) to after retraining with just additional 2% of data in the scenario where all of the samples of the holdout sub-class are excluded from the original training data. The improvement is shown as the average absolute holdout sample test accuracy gain after retraining across holdout classes in each cluster.

Overall, cluster C5, C6, C2, and C7 gain the most (between 11.3% and 25.2%) after retraining since they suffer the most when the holdout classes are removed (between 40% and 70%). For all clusters except one (i.e., C1), N models acquisition functions out perform random and single model functions. Especially for C5, the variance of the predicted confidence acquisition function improve the models on average almost 5% better than the single model one.

For cluster C1, where the training data does not have a significant weakspot (i.e., the holdout accuracy drop is the smallest among all clusters), the N model ranking metric can still improve the model with performance similar to the performance of the single model acquisition function. This indicates that the original training data, even though missing samples from an entire sub-class, does not contain significant weakspots: the models were able to be generalized from the samples of the remaining sub-class.

Figure 5.6 shows the holdout test accuracy gain after retraining with different portions of additional training data when using the standard deviation of the predicted confidence for sample ranking. The portion of additional data is in relation to the size of the original training data, however, in total, the available additional data is 10% of the original training data. This means that the maximum gain the retraining can achieve is with 10% of additional data (i.e., 100% of the available additional data). The gain trend shows that with only 20% of available additional data (i.e., 2% of additional data), the retraining with the N model ranking metric can achieve nearly the maximum potential of the whole available additional data.

**Finding 2.a**: The ranking based on the variance of predictive confidence enables retraining to gain the most holdout accuracy in scenarios where the original training data have a significant weakspot (up to 25.2% with just 2% of additional data).

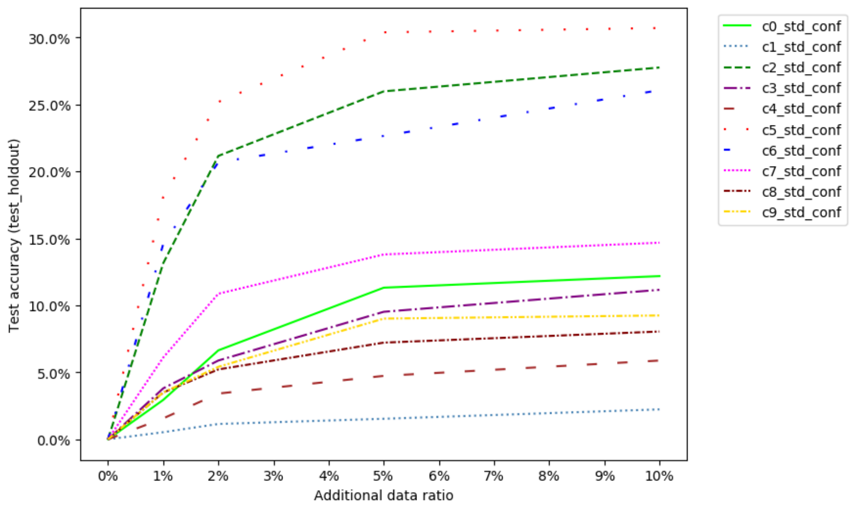


Figure 5.6: The holdout test accuracy gain after retraining using SDev-Conf ranking with different amounts of additional data in scenario with 100% holdout ratio

Figure 5.7 shows the non-holdout test accuracy gain after retraining with different portions of additional training data when using the standard deviation of the predicted confidence for sample ranking. This result shows that the N model variance metric improves the holdout accuracy significantly with just 2% additional training data, furthermore it does not affect the accuracy of non-holdout classes. At 5% of additional data, there is some small negative effect of at most 0.4% decrease in non-holdout classes' accuracy for some clusters (e.g., C6 and C4)

**Finding 2.b**: The ranking based on the variance of predictive confidence does not affect the test accuracy in other classes after retraining.

### 5.4.4 The "ray" Fish: A Use-Case of Leveraging Analysis of Samples with High Variance

To understand whether detailed analysis of the sample with high variance can give some insight on the nature of the weakspots, we apply a state of the art visualization tool Grad-CAM [202] to visualize the variance across multiple training runs. Specifically, Grad-CAM leverages the gradient information in the hidden layer to create a heat map that highlights important features which contribute to the decision made by the DL models.

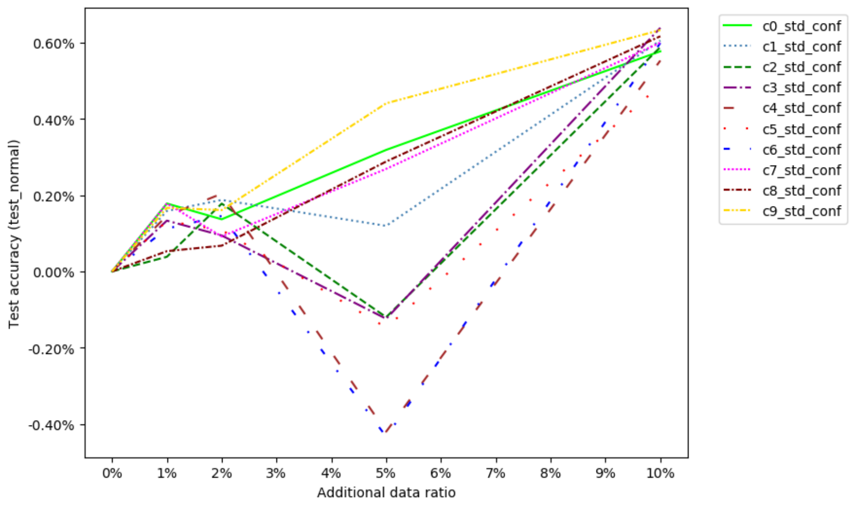


Figure 5.7: The non-holdout test accuracy gain after retraining using SDev-Conf ranking with different amounts of additional data with 100% holdout ratio

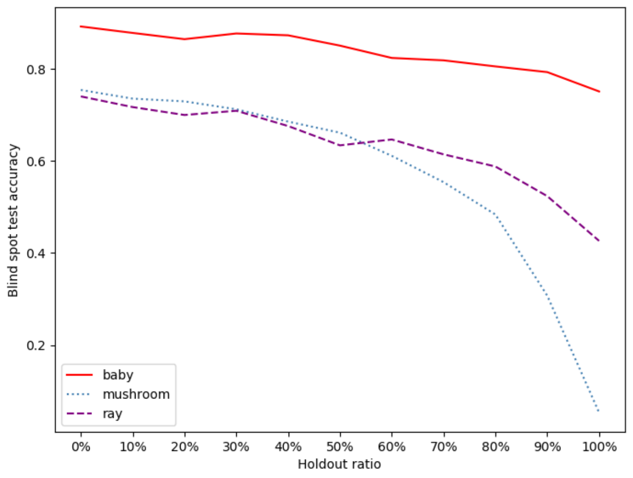


Figure 5.8: The holdout test accuracy drop as the holdout ratio is increased

In our experiments, one of the most interesting holdout sub-classes is the “ray” in the “fish” class. Comparing to other types of fish such as “aquarium fish”, “flatfish”, “shark”, or “trout”, the “ray” fish has very different shape and looks distinctively different. Thus,

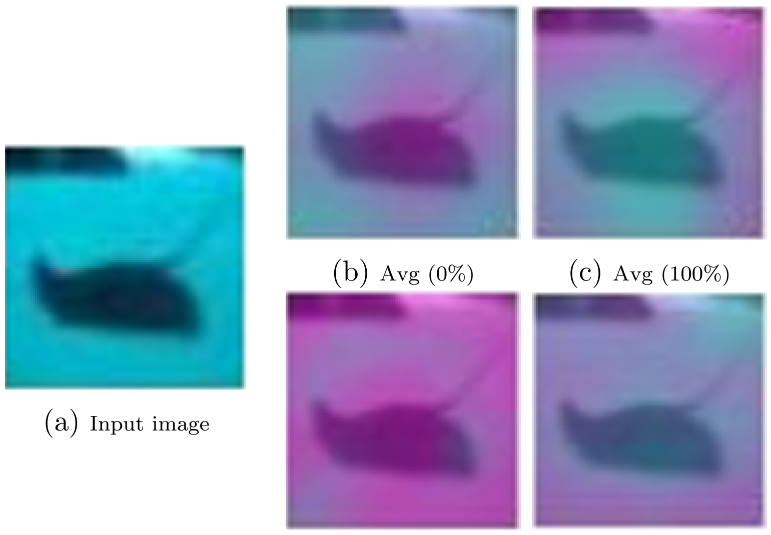

(a) Input image (b) Avg (0%) (c) Avg (100%) (d) Max-diff(0%) (e) Max-diff(100%)

Figure 5.9: Original input images of a “ray” fish and the average and maximum difference GradCAM for 0% and 100% holdout ratio.

when we perform the holdout experiment, we expect the accuracy to drop as we remove more and more examples of “ray” images from the training data. However, as shown in figure 5.8, the “ray” holdout test accuracy drops initially similar to the holdout class “mushroom”, but at 50% holdout ratio, unlike “mushroom”, the “ray” accuracy stops reducing and only drops to 43%, even when the ratio reaches 100% (i.e., there is no “ray” images in the training data). This finding leads us to suspect that the models may have used a non-“ray” related feature to classify the “ray” images as “fish”.

To investigate whether our suspicion is correct, we apply Grad-CAM to extract the feature heat map of “ray” fish in the test set for each trained model. We then compute the average and maximum difference heat map for those “ray” images. Figure 5.9 show the heat maps for the same image in figure 5.9a for models that have access to the full training dataset and models that were trained on training dataset with “ray” images completely removed. In the Grad-CAM images (figures 5.9b, 5.9c,5.9d,5.9e), the purple color indicates high values (i.e., high feature focus or high variance of feature focus) and the cyan color indicate the low values (i.e., low feature focus or low variance of feature focus).

Figure 5.9b shows the average Grad-CAM heat map across multiple models trained with a full dataset (i.e., 0% holdout ratio). In this heat map, the purple area is on top of the ray fish, indicating that the models are focusing on the actual fish to make the classification.

However, Figure 5.9c shows that the models focus more on the water surrounding the fish (i.e., indicating by the purple area around the fish) when there is no “ray” images in the training data (i.e., 100% holdout ratio). This explains the higher-than-expected accuracy (over 40%) even though no “ray” images are used in training, because the model learns to use the water to predict a “ray” image as a fish.

Figure 5.9d shows the maximum difference between Grad-CAM heat maps across multiple models trained with a full dataset. The whole image is covered in similar colors indicating that the differences between focus of different models are small. However, figure 5.9e shows that, when no “ray” images are used to train the model, the models are in more conflict on whether to use the fish to predict the label (indicated by the more distinct purple ring around the “ray” fish).

This is a use case where these detailed analysis of average and maximum difference between model focuses gives us more information and potentially can help DL model developers detect and pinpoint a weakspot problem with their training dataset.

## 5.5 Summary

DL systems can work well in the real world. However, they can suffer from the data blindspot problem where the training data misses samples from the real world distribution. It is difficult to detect such blindspots since the model may consider these samples unknown unknowns (i.e., it unknowingly classifies such samples incorrectly with high confidence). However, in practice, a weaker form of blindspot can also be a problem (i.e., the training data contains some samples that represents the real world but it does not contain enough). Our N models procedure leverages the variance of the DL training process to detect such data weakspots and help with the selection of additional training instances to improve the DL system with minimum labeling effort. Our evaluation shows that, in scenarios where the weakspots are severe, our procedure improves the model accuracy on weakspot samples by 25.2% requiring 2% of additional training data. This is an improvement of 4.5 percentage points compared to the traditional single model metric with the same amount of additional training data.

# Chapter 6

# Future Work

The projects in this thesis could be further supported and expanded with the following future work. Section 6.1 details future work on DL software systems debugging, Section 6.2 lists some possible exploration of DL systems automatic repair. Finally, Section 6.3 proposes potential directions to perform variance aware hyperparameter optimization.

## 6.1 Deep Learning Software Systems Testing and Debugging

CRADLE can detect real bugs in DL libraries by applying differential testing, however, it makes several compromises. Specifically, it requires more than one equivalent DL library to work. One direction that can remove such limitation is to utilize other equivalent concepts such as equivalent model/network, equivalent optimizations, or equivalent data formats to expand the differential testing that CRADLE can do. For example, in compiler testing, differential testing was able to detect bugs in optimization code by introducing dead branches to create equivalent programs [135, 136, 252]. One can apply similar concepts to add "dead branches" to DL models (i.e., the portion of the models that does not impact the final outcome but does interact with optimization code such as pruning or compression).

Another CRADLE's drawback is that it only tests inference code due to nondeterministic training [183], which makes differential testing challenging, because one can not tell if the detected inconsistency is caused by the variance induced by the software implementation or a software bug. One obvious solution is to enable deterministic training. However, deterministic training does not support some networks and configurations. Another potential solution to this challenge is to test partial training steps (e,g., only backpropagation—gradient computation—is enabled but no weight updating).

CRADLE can detect real DL libraries bugs that affect real models with real inputs, however, this limits the number of bugs CRADLE can detect. Our other work [240] is able to detect many critical DL libraries bugs by fuzzing DL libraries Application Programming Interface (API) inputs that conform or violate the extracted constraint from the API documents. However, such bug-triggering inputs are only for one internal API that might not trigger a bug in a real DL system even if such systems contain the buggy API. One possible improvement is to generate bug-triggering inputs to a whole DL system given the specific API bug-triggering input to a particular buggy API. This could be done by applying optimization [163] to the model input such that the input to the buggy API is as close to the API bug-triggering input as possible.

## 6.2 Deep Learning Systems Automatic Repair

Bugs detected by CRADLE require the attention of a developer to investigate and fix. An automatically generated quick fix would be beneficial to keep the DL functional until a proper fix is developed. Since DL systems are networks of mathematical formulas, one can create a quick fix by modifying the network (e.g., adding a dense layer after the buggy layer) and applying some fine-tuning so that the different libraries would produce matching outputs. For example, for the batch normalization bug that CRADLE found, one could add additional dense layers after the faulty CNTK's batch normalization layers. Once these additional layers are fine-tuned with the output of TensorFlow's model as the optimization goal, the additional layers would act as a temporary patch until a fix can be added to the source code of the faulty library.

## 6.3 Variance Aware Deep Learning Hyperparameter Optimization

Recently, the rise of complex and computationally expensive DL models with many hyperparameters has resulted in a resurgence of research on hyperparameter optimization (HPO). It is widely acknowledged that optimized hyperparameters often provide better models than the default settings [78]. Such HPO and the recent AutoML approaches often base their selection on the evaluations of the candidate settings. Given the cost of training DL models, such approaches usually perform a single evaluation run for each candidate setting. However, the variance caused by DL libraries [183, 186] could potentially cause the selection to be suboptimal and in turn, reduce the effectiveness of the optimization methods. The research community could benefit from a study that quantifies how much variance would the implementation introduce to approaches such as HPO or AutoML. If the variance is significant, developing variance-aware HPO or AutoML approaches could be the next step. One could integrate the observed variance into an optimization approach such as Bayesian optimization. One could optimize this further by developing estimation methods to approximate the variance and mitigate the cost of having to run multiple expensive training runs.

# Chapter 7

# Conclusion

This thesis proposes techniques that help improving the DL system reliability for all stages of the development process: detecting bugs in the inference stage, analyzing the variance of DL training, and detecting and mitigating blindspots in DL training data.

In Chapter 3, we propose CRADLE, which that improves the reliability of the DL system inference by applying differential testing to find bugs in DL libraries. CRADLE focuses on finding and localizing bugs in DL software libraries by performing cross-implementation inconsistency checking to detect bugs, and leveraging anomaly propagation tracking and analysis to localize faulty functions that cause the bugs. CRADLE detects 12 bugs in three DL libraries, and highlights functions relevant to the causes of inconsistencies for all 104 unique inconsistencies

In Chapter 4, we present the first study of the variance of DL systems training and the awareness of this variance among researchers and practitioners. The study shows implementation-level factors alone cause the accuracy difference across identical training runs to be up to 2.9%. The survey shows that 83.8% of the 901 participants are unaware of or unsure about any implementation-level variance. The study and surveys aim to raise awareness of DL training variance while the tool improves the reproducibility of DL software and results.

In Chapter 5, we evaluate a new procedure to detect weakspots in training data and to improve the DL system with minimum labeling effort. The result shows that this procedure improves the model accuracy on weakspot samples by 25.2% while requiring the same 2% of additional training data.

Collectively, these projects contribute to improve the reliability of deep learning software systems.

# References

[1] Cat vs. Dog Models. https://github.com/rajshah4/image_keras, 2017.

[2] Anime Face Dataset. http://www.nurs.or.jp/~nagadomi/animeface-character-dataset, 2018.

[3] BetaGo: AlphaGo for the Masses. https://github.com/maxpumperla/deep_learning_and_the_game_of_go, 2018.

[4] Dog Model. https://github.com/humayun/dl-dataday-workshop, 2018.

[5] Gender Model. https://github.com/oarriaga/face_classification/, 2018.

[6] Model Weights File of Dog Project. https://github.com/humayun/dl-dataday-workshop/blob/master/code/dog_project/saved_models/weights.best.from_scratch.hdf5, 2018.

[7] Pokedex. https://github.com/Robert-Alonso/Keras-React-Native-Pokedex, 2018.

[8] Taryn Southern's new album is produced entirely by AI. https://www.digitaltrends.com/music/artificial-intelligence-taryn-southern-album-interview/, 2018.

[9] Thai Handwriting Number. https://kittinan.github.io/thai-handwriting-number/, 2018.

[10] The Data Incubator. https://github.com/thedataincubator/data-science-blogs/blob/master/output/DL_libraries_final_Rankings.csv, 2018.

[11] TrafficSigns1 Model. https://github.com/jaeoh2/CoreML-Traffic-Sign-Classifier, 2018.

[12] TrafficSigns2 Model. https://github.com/inspire-group/advml-traffic-sign, 2018.

[13] TrafficSigns3 Model. https://github.com/MidnightPolaris/gtsdb_cnn, 2018.

[14] *ASE '19: Proceedings of the 34th ACM/IEEE International Conference on Automated Software Engineering*, 2019.

[15] *ASPLOS '19: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2019.

[16] *CVPR'19: Proceedings of The IEEE Conference on Computer Vision and Pattern Recognition*, 2019.

[17] *ESEC/FSE '19: Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019.

[18] *ICCV'19: Proceedings of The IEEE International Conference on Computer Vision*. IEEE Press, 2019.

[19] *ICML'19: Proceedings of The International Conference on Machine Learning*, 2019.

[20] *ICSE '19: Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019.

[21] *MLSys'19: Proceedings of Machine Learning and Systems*, 2019.

[22] *NIPS'19: Proceedings of the 33rd Conference on Neural Information Processing Systems*, 2019.

[23] *SOSP '19: Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019.

[24] *ICLR'20: Proceedings of The International Conference on Learning Representations*, 2020.

[25] Reproducibility challenge at neurips 2019, 2020.

[26] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. TensorFlow: a System for Large-Scale Machine Learning. In *OSDI*, volume 16, pages 265–283, 2016.

[27] Charu C Aggarwal. Neural networks and deep learning. In *Springer*, 2018.

[28] Charu C. Aggarwal and Philip S. Yu. Outlier detection for high dimensional data. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, page 3746, New York, NY, USA, 2001. Association for Computing Machinery.

[29] Naveed Akhtar and Ajmal S. Mian. Threat of adversarial attacks on deep learning in computer vision: A survey. *IEEE Access*, 6:14410–14430, 2018.

[30] Andrea Arcuri and Lionel Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE*, 2011.

[31] Nabiha Asghar, Pascal Poupart, Xin Jiang, and Hang Li. Deep active learning for dialogue generation. In *Proceedings of the 6th Joint Conference on Lexical and Computational Semantics (*SEM 2017)*, pages 78–83, Vancouver, Canada, August 2017. Association for Computational Linguistics.

[32] Deegan J. Atha and Mohammad Reza Jahanshahi. Evaluation of deep learning approaches based on convolutional neural networks for corrosion detection. *Structural Health Monitoring*, 17:1110 – 1128, 2018.

[33] Joshua Attenberg, Panos Ipeirotis, and Foster Provost. Beat the machine: Challenging humans to find a predictive model's unknown unknowns. *J. Data and Information Quality*, 6(1), 2015.

[34] Joshua Attenberg, Panos Ipeirotis, and Foster Provost. Beat the machine: Challenging humans to find a predictive model's unknown unknowns. *J. Data and Information Quality*, 6(1), March 2015.

[35] Irad Ben-Gal. *Outlier Detection*, pages 131–146. Springer US, Boston, MA, 2005.

[36] Paul Bergmann, Kilian Batzner, Michael Fauser, David Sattlegger, and Carsten Steger. The mvtec anomaly detection dataset: A comprehensive real-world dataset for unsupervised anomaly detection. *Int. J. Comput. Vision*, 129(4):10381059, apr 2021.

[37] James Bergstra, Frédéric Bastien, Olivier Breuleux, Pascal Lamblin, Razvan Pascanu, Olivier Delalleau, Guillaume Desjardins, David Warde-Farley, Ian Goodfellow, Arnaud Bergeron, et al. Theano: Deep learning on GPUs with Python. In *NIPS 2011, BigLearning Workshop, Granada, Spain*, volume 3, pages 1–48. Citeseer, 2011.

[38] James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. In *JMLR*, 2012.

[39] Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. Autofocus: Interpreting attention-based neural networks by code perturbation. In *ASE*, 2019.

[40] Joy Buolamwini and Timnit Gebru. Gender shades: Intersectional accuracy disparities in commercial gender classification. In Sorelle A. Friedler and Christo Wilson, editors, *Proceedings of the 1st Conference on Fairness, Accountability and Transparency*, volume 81 of *Proceedings of Machine Learning Research*, pages 77–91. PMLR, 2018.

[41] Nicol Cesa-Bianchi and Gbor Lugosi. *Prediction, learning, and games.* Cambridge University Press, 2006.

[42] Ken Chang, Niranjan Balachandar, Carson Lam, Darvin Yi, James Brown, Andrew Beers, Bruce Rosen, Daniel L Rubin, and Jayashree Kalpathy-Cramer. Distributed Deep Learning Networks Among Institutions for Medical Imaging. *Journal of the American Medical Informatics Association*, 2018.

[43] Chenyi Chen, Ari Seff, Alain Kornhauser, and Jianxiong Xiao. Deepdriving: Learning Affordance for Direct Perception in Autonomous Driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730, 2015.

[44] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. Unblind your apps: Predicting natural-language labels for mobile gui components by deep learning. In *ICSE*, 2020.

[45] Jinyin Chen, Keke Hu, Yue Yu, Zhuangzhi Chen, Qi Xuan, Yi Liu, and Vladimir Filkov. Software visualization and deep transfer learning for effective software defect prediction. In *ICSE*, 2020.

[46] Liang-Chieh Chen, Maxwell Collins, Yukun Zhu, George Papandreou, Barret Zoph, Florian Schroff, Hartwig Adam, and Jon Shlens. Searching for efficient multi-scale architectures for dense image prediction. In *NIPS*, 2018.

[47] Z. Chen and X. Huang. End-to-End Learning for Lane Keeping of Self-Driving Cars. In *2017 IEEE Intelligent Vehicles Symposium (IV)*, pages 1856–1860, June 2017.

[48] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning, 2014.

[49] François Chollet et al. Keras. `https://keras.io`, 2015.

[50] François Chollet. Xception: Deep learning with depthwise separable convolutions. In *CVPR*, 2017.

[51] Anna Choromanska, Mikael Henaff, Michael Mathieu, Gérard Ben Arous, and Yann LeCun. The loss surfaces of multilayer networks. In *AISTATS*, 2015.

[52] Anna Choromanska, Yann LeCun, and Grard Ben Arous. Open problem: The landscape of the loss surfaces of multilayer networks. In *COLR*, 2015.

[53] Shauvik Roy Choudhary. Detecting Cross-browser Issues in Web Applications. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1146–1148, New York, NY, USA, 2011. ACM.

[54] Wen-Hsuan Chu. Neural batch sampling with reinforcement learning for semi-supervised anomaly detection. Master's thesis, Carnegie Mellon University, Pittsburgh, PA, May 2020.

[55] Dan Ciresan, Ueli Meier, and Jrgen Schmidhuber. Multi-column deep neural networks for image classification. In *CVPR*, 2012.

[56] J. Cohen. *Statistical Power Analysis for the Behavioral Sciences*. 1988.

[57] Cédric Colas, Olivier Sigaud, and Pierre-Yves Oudeyer. How many random seeds? statistical power analysis in deep reinforcement learning experiments. In *CoRR*, 2018.

[58] Cody Coleman, Deepak Narayanan, Daniel Kang, Tian Zhao, Jian Zhang, Luigi Nardi, Peter Bailis, Kunle Olukotun, Chris Ré, and Matei Zaharia. DAWNBench: An End-to-End Deep Learning Benchmark and Competition. *Training*, 100(101):102, 2017.

[59] Standard Performance Evaluation Corporation. SPEC CPU2006 benchmarks. `http://www.spec.org/cpu2006/`, 2006.

[60] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. Parrot: a practical runtime for deterministic, stable, and reliable threads. In *SOSP*, 2013.

[61] Yann N Dauphin, Razvan Pascanu, Caglar Gulcehre, Kyunghyun Cho, Surya Ganguli, and Yoshua Bengio. Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. In *NIPS*, 2014.

[62] Dennis Decoste and Bernhard Schölkopf. Training invariant support vector machines. *Machine learning*, 46(1-3):161–190, 2002.

[63] Akshay Raj Dhamija, Manuel Günther, and Terrance Boult. Reducing network agnostophobia. In S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 31. Curran Associates, Inc., 2018.

[64] Christopher P. Diehl and John B. Hampshire Ii. Real-time object classification and novelty detection for collaborative video surveillance. In *In Proceedings of the International Joint Conference on Neural Networks*, pages 2620–2625, 2002.

[65] Nguyen Ngoc Diep. Intrusion Detection Using Deep Neural Network. *Southeast Asian Journal of Sciences*, 5(2):111–125, 2017.

[66] Jesse Dodge, Suchin Gururangan, Dallas Card, Roy Schwartz, and Noah A. Smith. Show your work: Improved reporting of experimental results. In *EMNLP-IJCNLP*, 2019.

[67] Brian Dolhansky, Russ Howes, Ben Pflaum, Nicole Baram, and Cristian Canton Ferrer. The deepfake detection challenge (dfdc) preview dataset, 2019.

[68] Felix Draxler, Kambis Veschgini, Manfred Salmhofer, and Fred Hamprecht. Essentially no barriers in neural network energy landscape. In *ICML*, 2018.

[69] Simon S Du, Jason D Lee, Yuandong Tian, Barnabas Poczos, and Aarti Singh. Gradient descent learns one-hidden-layer cnn: Don't be afraid of spurious local minima. In *arXiv preprint arXiv:1712.00779*, 2017.

[70] Saikat Dutta, Owolabi Legunsen, Zixin Huang, and Sasa Misailovic. Testing probabilistic programming systems. 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, 2018*, pages 574–586, 2018.

[71] Suvajit Dutta, BCS Manideep, Shalva Rai, and V Vijayarajan. A Comparative Study of Deep Learning Models for Medical Image Classification. In *IOP Conference Series: Materials Science and Engineering*, volume 263, page 042097. IOP Publishing, 2017.

[72] Anurag Dwarakanath, Manish Ahuja, Samarth Sikand, Raghotham M. Rao, R. P. Jagadeesh Chandra Bose, Neville Dubash, and Sanjay Podder. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, pages 118–128, New York, NY, USA, 2018. ACM.

[73] Amir Efrati. Uber Finds Deadly Accident Likely Caused by Software Set to Ignore Objects on Road. *The information*, 2018.

[74] Brian Everitt. *The Cambridge dictionary of statistics*. 2002.

[75] Hao-Shu Fang, Guansong Lu, Xiaolin Fang, Jianwen Xie, Yu-Wing Tai, and Cewu Lu. Weakly and semi supervised human body part parsing via pose-guided knowledge transfer. In *CVPR*, 2018.

[76] Zhen Fang, Jie Lu, Anjin Liu, Feng Liu, and Guangquan Zhang. Learning bounds for open-set learning, 06 2021.

[77] Mattia Fazzini and Alessandro Orso. Automated Cross-platform Inconsistency Detection for Mobile Apps. In *Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering*, ASE 2017, pages 308–318, Piscataway, NJ, USA, 2017. IEEE Press.

[78] Matthias Feurer and Frank Hutter. *Hyperparameter Optimization*. 2019.

[79] Diego Fioravanti, Ylenia Giarratano, Valerio Maggio, Claudio Agostinelli, Marco Chierici, Giuseppe Jurman, and Cesare Furlanello. Phylogenetic Convolutional Neural Networks in Metagenomics. *BMC bioinformatics*, 19(2):49, 2018.

[80] Brian S Freeman, Graham Taylor, Bahram Gharabaghi, and Jesse Thé. Forecasting Air Quality Time Series Using Deep Learning. *Journal of the Air & Waste Management Association*, pages 1–21, 2018.

[81] Yarin Gal and Zoubin Ghahramani. Bayesian convolutional neural networks with bernoulli approximate variational inference, 2016.

[82] Xiang Gao, Ripon Saha, Mukul Prasad, and Abhik Roychoudhury. Fuzz testing based data augmentation to improve robustness of deep neural networks. In *ICSE*, 2020.

[83] Rong Ge, Furong Huang, Chi Jin, and Yang Yuan. Escaping from saddle pointsonline stochastic gradient for tensor decomposition. In *COLT*, 2015.

[84] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. AI 2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Security and Privacy (SP), 2018 IEEE Symposium on*, 2018.

[85] Andreas Geiger, Philip Lenz, and Raquel Urtasun. Are we ready for autonomous driving? the kitti vision benchmark suite. In *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3354–3361, 2012.

[86] Simos Gerasimou, Hasan Ferit-Eniser, Alper Sen, and Alper akan. Importance-driven deep learning system testing. In *ICSE*, 2020.

[87] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *AISTATS*, 2010.

[88] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, March 1991.

[89] David Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 1991.

[90] Jesús M González-Barahona and Gregorio Robles. On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. In *ESE*, 2012.

[91] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning.* MIT press, 2016.

[92] Ian Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. In *ICLR*, 2015.

[93] Divya Gopinath, Hayes Converse, Corina S. Păsăreanu, and Ankur Taly. Property inference for deep neural networks. In *ASE*, 2019.

[94] Thore Graepel and Ralf Herbrich. Invariant pattern recognition by semi-definite programming machines. In S. Thrun, L. Saul, and B. Schölkopf, editors, *Advances in Neural Information Processing Systems*, volume 16. MIT Press, 2004.

[95] Hui Guan, Xipeng Shen, and Seung-Hwan Lim. Wootz: A compiler-based framework for fast cnn pruning via composability. In *PLDI*, 2019.

[96] Qianyu Guo, Sen Chen, Xiaofei Xie, Lei Ma, Qiang Hu, Hongtao Liu, Yang Liu, Jianjun Zhao, and Xiaohong Li. An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms. In *ASE*, 2019.

[97] Jeff Haochen and Suvrit Sra. Random shuffling beats SGD after finite epochs. In *ICML*, 2019.

[98] Hua He, Kevin Gimpel, and Jimmy Lin. Multi-perspective sentence similarity modeling with convolutional neural networks. In *Proceedings of the 2015 Conference on Empirical Methods in Natural Language Processing*, pages 1576–1586, Lisbon, Portugal, September 2015. Association for Computational Linguistics.

[99] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *ICCV*, 2015.

[100] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *CVPR*, 2016.

[101] Tao He, Xiaoming Jin, Guiguang Ding, Lan Yi, and C. Yan. Towards better uncertainty sampling: Active learning with multiple views for deep convolutional neural network. *2019 IEEE International Conference on Multimedia and Expo (ICME)*, pages 1360–1365, 2019.

[102] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In *ESEC/FSE*, 2018.

[103] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. In *AAAI*, 2018.

[104] Dan Hendrycks and Kevin Gimpel. A baseline for detecting misclassified and out-of-distribution examples in neural networks. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[105] Geoffrey Hinton, Li Deng, Dong Yu, George Dahl, Abdel rahman Mohamed, Navdeep Jaitly, Andrew Senior, Vincent Vanhoucke, Patrick Nguyen, Tara Sainath, and Brian Kingsbury. Deep neural networks for acoustic modeling in speech recognition. *Signal Processing Magazine*, 2012.

[106] Victoria Hodge and Jim Austin. A survey of outlier detection methodologies. *Artif. Intell. Rev.*, 22(2):85126, oct 2004.

[107] Renáta Hodován and Ákos Kiss. Modernizing Hierarchical Delta Debugging. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*, pages 31–37. ACM, 2016.

[108] Xiaowei Huang, Daniel Kroening, Wenjie Ruan, James Sharp, Youcheng Sun, Emese Thamo, Min Wu, and Xinping Yi. A survey of safety and trustworthiness of deep neural networks: Verification, testing, adversarial attack and defence, and interpretability, 2020.

[109] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety Verification of Deep Neural Networks. In *International Conference on Computer Aided Verification*, pages 3–29. Springer, 2017.

[110] Haroon Idrees, Mubarak Shah, and Ray Surette. Enhancing camera surveillance using computer vision: a research note, 2018.

[111] Ilija Ilievski, Taimoor Akhtar, Jiashi Feng, and Christine Annette Shoemaker. Efficient hyperparameter optimization for deep learning algorithms using deterministic rbf surrogates. In *AAAI*, 2017.

[112] Jing Jiang and ChengXiang Zhai. A two-stage approach to domain adaptation for statistical classifiers. In *CIKM*, 2007.

[113] Liming Jiang, Zhengkui Guo, Wayne Wu, Zhaoyang Liu, Ziwei Liu, Chen Change Loy, Shuo Yang, Yuanjun Xiong, Wei Xia, Baoying Chen, Peiyu Zhuang, Sili Li, Shen Chen, Taiping Yao, Shouhong Ding, Jilin Li, Feiyue Huang, Liujuan Cao, Rongrong Ji, Changlei Lu, and Ganchao Tan. Deeperforensics challenge 2020 on real-world face forgery detection: Methods and results, 2021.

[114] James A Jones and Mary Jean Harrold. Empirical Evaluation of the Tarantula Automatic Fault-Localization Technique. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 273–282. ACM, 2005.

[115] M. E. Joorabchi, M. Ali, and A. Mesbah. Detecting Inconsistencies in Multi-platform Mobile Apps. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 450–460, Nov 2015.

[116] Hadi Jooybar, Wilson WL Fung, Mike O'Connor, Joseph Devietti, and Tor M Aamodt. Gpudet: a deterministic gpu architecture. In *ASPLOS*, 2013.

[117] Kyle D Julian, Jessica Lopez, Jeffrey S Brush, Michael P Owen, and Mykel J Kochenderfer. Policy Compression for Aircraft Collision Avoidance Systems. In *Digital Avionics Systems Conference (DASC), 2016 IEEE/AIAA 35th*, pages 1–10. IEEE, 2016.

[118] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *International Conference on Computer Aided Verification*, pages 97–117. Springer, 2017.

[119] Jinhan Kim, Robert Feldt, and Shin Yoo. Guiding deep learning system testing using surprise adequacy. In *ICSE*, 2019.

[120] YK Kim and JB Ra. Weight value initialization for improving training speed in the backpropagation network. In *IJCNN*, 1991.

[121] Jamie Ryan Kiros. Recurrent Neural Network that Generates Little Stories About Images. https://github.com/ryankiros/neural-storyteller, 2018.

[122] Ryan Kiros, Yukun Zhu, Ruslan Salakhutdinov, Richard S Zemel, Antonio Torralba, Raquel Urtasun, and Sanja Fidler. Skip-Thought Vectors. *arXiv preprint arXiv:1506.06726*, 2015.

[123] Bobby Kleinberg, Yuanzhi Li, and Yang Yuan. An alternative view: When does SGD escape local minima? In *ICML*, 2018.

[124] Yuriy Kochura, Sergii Stirenko, Oleg Alienin, Michail Novotarskiy, and Yuri Gordienko. Performance analysis of open source machine learning frameworks for various parameters in single-threaded and multi-threaded modes. In *CSIT*, 2018.

[125] Vassili Kovalev, Alexander Kalinovsky, and Sergey Kovalev. Deep learning with theano, torch, caffe, tensorflow, and deeplearning4j: Which one is the best in speed and accuracy? 2016.

[126] Alex Krizhevsky. Learning multiple layers of features from tiny images. 2009.

[127] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet Classification with Deep Convolutional Neural Networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[128] Tomasz Kuchta, Thibaud Lutellier, Edmund Wong, Lin Tan, and Cristian Cadar. On the Correctness of Electronic Documents: Studying, Finding, and Localizing Inconsistency Bugs in PDF Readers and Files. *Empirical Software Engineering*, pages 1–34, 2018.

[129] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial Examples in the Physical World. *arXiv preprint arXiv:1607.02533*, 2016.

[130] Sunyoung Kwon and Sungroh Yoon. DeepCCI: End-to-End Deep Learning for Chemical-Chemical Interaction Prediction. In *Proceedings of the 8th ACM International Conference on Bioinformatics, Computational Biology, and Health Informatics*, pages 203–212. ACM, 2017.

[131] Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Dire: A neural approach to decompiled identifier naming. In *ASE*, 2019.

[132] Himabindu Lakkaraju, Ece Kamar, Rich Caruana, and Eric Horvitz. Discovering blind spots of predictive models: Representations and policies for guided exploration. *ArXiv*, abs/1610.09064, 2016.

[133] Alex Lamb, Jonathan Binas, Anirudh Goyal, Sandeep Subramanian, Ioannis Mitliagkas, Denis Kazakov, Yoshua Bengio, and Michael C. Mozer. State-reification networks: Improving generalization by modeling the distribution of hidden representations. In *ICML*, 2019.

[134] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler Validation via Equivalence Modulo Inputs. In *ACM SIGPLAN Notices*, volume 49, pages 216–226. ACM, 2014.

[135] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *PLDI*, 2014.

[136] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. In *OOPSLA*, 2015.

[137] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-Based Learning Applied to Document Recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998.

[138] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, 1998.

[139] Yann A LeCun, Léon Bottou, Genevieve B Orr, and Klaus-Robert Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, 2012.

[140] Jason D Lee, Max Simchowitz, Michael I Jordan, and Benjamin Recht. Gradient descent converges to minimizers. In *arXiv preprint arXiv:1602.04915*, 2016.

[141] David D. Lewis and William A. Gale. A sequential algorithm for training text classifiers, 1994. active learning roots.

[142] Hao Li, Zheng Xu, Gavin Taylor, Christoph Studer, and Tom Goldstein. Visualizing the loss landscape of neural nets. In *NIPS*, 2018.

[143] Li-Jia Li, Gang Wang, and Li Fei-Fei. Optimol: automatic online picture collection via incremental model learning. In *2007 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2007.

[144] Yi Li, Wang Shaohua, and Tien N. Nguyen. DLfix: Context-based code transformation learning for automated program repair. In *ICSE*, 2020.

[145] Yi Li, Shaohua Wang, Tien N Nguyen, and Son Van Nguyen. Improving bug detection via context-based code representation learning and attention-based neural networks. In *PACMPL*, 2019.

[146] Zenan Li, Xiaoxing Ma, Chang Xu, Chun Cao, Jingwei Xu, and Jian Lü. Boosting operational dnn testing efficiency through conditioning. In *ESEC/FSE*, 2019.

[147] Ling Liu, Yanzhao Wu, Wenqi Wei, Wenqi Cao, Semih Sahin, and Qi Zhang. Benchmarking Deep Learning Frameworks: Design Considerations, Metrics and Beyond. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018.

[148] Si Liu, Risheek Garrepalli, Thomas G. Dietterich, Alan Fern, and Dan Hendrycks. Open category detection with PAC guarantees. *CoRR*, abs/1808.00529, 2018.

[149] Siqi Liu, Sidong Liu, Weidong Cai, Sonia Pujol, Ron Kikinis, and Dagan Feng. Early Diagnosis of Alzheimer's Disease with Deep Learning. In *Biomedical Imaging (ISBI), 2014 IEEE 11th International Symposium on*, pages 1015–1018. IEEE, 2014.

[150] Thibaud Lutellier, Hung Viet Pham, Lawrence Pang, Yitong Li, Moshi Wei, and Lin Tan. Coconut: Combining context-aware neural translation models using ensemble for program repair. In *ISSTA*, 2020.

[151] 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. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 120–131, New York, NY, USA, 2018. ACM.

[152] Shiqing Ma, Yingqi Liu, Wen-Chuan Lee, Xiangyu Zhang, and Ananth Grama. Mode: Automated neural network model debugging via state differential analysis and input selection. In *ESEC/FSE*, 2018.

[153] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. In *ICLR*, 2018.

[154] Zaheed Mahmood, David Bowes, Tracy Hall, Peter CR Lane, and Jean Petrić. Reproducibility and replicability of software defect prediction studies. *ISE*, 2018.

[155] Andrii Maksai and Pascal Fua. Eliminating exposure bias and metric mismatch in multiple object tracking. In *CVPR*, 2019.

[156] Markos Markou and Sameer Singh. Novelty detection: a reviewpart 1: statistical approaches. *Signal Processing*, 83(12):2481–2497, 2003.

[157] Markos Markou and Sameer Singh. Novelty detection: A reviewpart 2: Neural network based approaches. *Signal Process.*, 83(12):24992521, dec 2003.

[158] William M McKeeman. Differential Testing for Software. *Digital Technical Journal*, 10(1):100–107, 1998.

[159] Luke Metz, Niru Maheswaranathan, Jeremy Nixon, C Daniel Freeman, and Jascha Sohl-Dickstein. Understanding and correcting pathologies in the training of learned optimizers. In *ICML*, 2019.

[160] Hrushikesh N. Mhaskar, Sergei V. Pereverzyev, and Maria D. van der Walt. A deep learning approach to diabetic blood glucose prediction. In *Front. Appl. Math. Stat.*, 2017.

[161] Ghassan Misherghi and Zhendong Su. HDD: Hierarchical Delta Debugging. In *Proceedings of the 28th international conference on Software engineering*, pages 142–151. ACM, 2006.

[162] Seokhyeon Moon, Yunho Kim, Moonzoo Kim, and Shin Yoo. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation (ICST)*, pages 153–162. IEEE, 2014.

[163] A. Mordvintsev, Christopher Olah, and Mike Tyka. Inceptionism: Going deeper into neural networks. 2015.

[164] Todd Mytkowicz, Amer Diwan, Matthias Hauswirth, and Peter Sweeney. Producing wrong data without doing anything obviously wrong! In *ACM SIGARCH Computer Architecture News*, 2009.

[165] Prabhat Nagarajan, Garrett Warnell, and Peter Stone. Deterministic implementations for reproducibility in deep reinforcement learning. In *AAAI Workshop on Reproducible AI*, 2019.

[166] Lee Naish, Hua Jie Lee, and Kotagiri Ramamohanarao. A Model for Spectra-based Software Diagnosis. *ACM Transactions on software engineering and methodology (TOSEM)*, 20(3):11, 2011.

[167] Nina Narodytska and Shiva Prasad Kasiviswanathan. Simple Black-Box Adversarial Attacks on Deep Neural Networks. In *CVPR Workshops*, pages 1310–1318, 2017.

[168] Anh Nguyen, Jason Yosinski, and Jeff Clune. Deep Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 427–436, 2015.

[169] Hieu T. Nguyen and Arnold Smeulders. Active learning using pre-clustering. In *Proceedings of the Twenty-First International Conference on Machine Learning*, ICML '04, page 79, New York, NY, USA, 2004. Association for Computing Machinery.

[170] Kristin Nixon, Valerio Aimale, and Robert Rowe. *Spoof Detection Schemes*, pages 403–423. 10 2007.

[171] Babatunde K Olorisade, Pearl Brereton, and Peter Andras. Reproducibility in machine learning-based studies: An example of text mining. In *Reproducibility in Machine Learning Workshop, ICML*, 2017.

[172] Guansong Pang, Chunhua Shen, Longbing Cao, and Anton Van Den Hengel. Deep learning for anomaly detection: A review. *ACM Comput. Surv.*, 54(2), mar 2021.

[173] Mike Papadakis and Yves Le Traon. Metallaxis-FL: Mutation-based Fault Localization. *Software Testing, Verification and Reliability*, 25(5-7):605–628, 2015.

[174] Nicolas Papernot, Patrick McDaniel, Ananthram Swami, and Richard Harang. Crafting Adversarial Input Sequences for Recurrent Neural Networks. In *Military Communications Conference, MILCOM 2016-2016 IEEE*, pages 49–54. IEEE, 2016.

[175] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In *NIPS*, 2019.

[176] Keyurkumar Patel, Hu Han, and Anil K. Jain. Secure face unlock: Spoof detection on smartphones. *IEEE Transactions on Information Forensics and Security*, 11(10):2268–2283, 2016.

[177] Brandon Paulsen, Jingbo Wang, and Chao Wang. Reludiff: Differential verification of deep neural networks. In *ICSE*, 2020.

[178] Spencer Pearson, José Campos, René Just, Gordon Fraser, Rui Abreu, Michael D Ernst, Deric Pang, and Benjamin Keller. Evaluating and Improving Fault Localization. In *Proceedings of the 39th International Conference on Software Engineering*, pages 609–620. IEEE Press, 2017.

[179] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18. ACM, 2017.

[180] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Commun. ACM*, 2019.

[181] Hung Viet Pham, Mijung Kim, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. Deviate: A deep learning variance testing framework. In *ASE*, 2020.

[182] Hung Viet Pham, Thibaud Lutellier, Weizhen Qi, and Lin Tan. Cradle: Cross-backend validation to detect and localize bugs in deep learning libraries. In *ICSE*, 2019.

[183] Hung Viet Pham, Shangshu Qian, Jiannan Wang, Thibaud Lutellier, Jonathan Rosenthal, Lin Tan, Yaoliang Yu, and Nachiappan Nagappan. Problems and opportunities in training deep learning software systems: An analysis of variance. In *35th International Conference on Automated Software Engineering*, ASE '20, pages 771–783, New York, 2020. IEEE/ACM.

[184] Gngr Polatkan, Sina Jafarpour, Andrei Brasoveanu, Shannon Hughes, and Ingrid Daubechies. Detection of forgery in paintings using supervised learning. In *2009 16th IEEE International Conference on Image Processing (ICIP)*, pages 2921–2924, 2009.

[185] Sanjay Purushotham, Chuizheng Meng, Zhengping Che, and Yan Liu. Benchmarking deep learning models on large healthcare datasets. In *J. Biomed. Inform.*, 2018.

[186] Shangshu Qian, Hung Viet Pham, Thibaud Lutellier, Zeou Hu, Jungwon Kim, Lin Tan, Yaoliang Yu, Jiahao Chen, and Sameena Shah. Are my deep learning systems fair? an empirical study of fixed-seed training. In *NeurIPS*, 2021.

[187] Joaquin Quionero-Candela, Masashi Sugiyama, Anton Schwaighofer, and Neil D. Lawrence. *Dataset Shift in Machine Learning*. The MIT Press, 2009.

[188] Hiranmayi Ranganathan, Hemanth Venkateswara, Shayok Chakraborty, and Sethuraman Panchanathan. Deep active learning for image classification. In *2017 IEEE International Conference on Image Processing (ICIP)*, pages 3934–3938, 2017.

[189] Subhajit Roy, Awanish Pandey, Brendan Dolan-Gavitt, and Yu Hu. Bug synthesis: Challenging bug-finding tools with deep faults. In *ESEC/FSE*, 2018.

[190] Shauvik Roy Choudhary, Mukul R Prasad, and Alessandro Orso. X-PERT: a Web Application Testing Tool for Cross-Browser Inconsistency Detection. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 417–420. ACM, 2014.

[191] Shauvik Roy Choudhary, Mukul R. Prasad, and Alessandro Orso. CrossCheck: Combining Crawling and Differencing to Better Detect Cross-browser Incompatibilities in Web Applications. In *Proceedings - IEEE 5th International Conference on Software Testing, Verification and Validation, ICST 2012*, pages 171–180, 04 2012.

[192] Shauvik Roy Choudhary, Husayn Versee, and Alessandro Orso. WEBDIFF: Automated Identification of Cross-browser Issues in Web Applications. In *Proceedings of*

*the 2010 IEEE International Conference on Software Maintenance*, ICSM '10, pages 1–10, Washington, DC, USA, 2010. IEEE Computer Society.

[193] Lukas Ruff, Jacob Kauffmann, Robert Vandermeulen, Gregoire Montavon, Wojciech Samek, Marius Kloft, Thomas Dietterich, and Klaus-Robert Mller. A unifying review of deep and shallow anomaly detection. *Proceedings of the IEEE*, PP:1–40, 02 2021.

[194] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*, 115(3):211–252, 2015.

[195] Tim Salimans and Durk P Kingma. Weight normalization: A simple reparameterization to accelerate training of deep neural networks. In *NIPS*, 2016.

[196] Shibani Santurkar, Dimitris Tsipras, and Aleksander Madry. BREEDS: benchmarks for subpopulation shift. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[197] Shlomo Sawilowsky, Jack Sawilowsky, and Robert J. Grissom. *Effect Size*. 2011.

[198] Andrew M. Saxe, James L. Mcclelland, and Surya Ganguli. Exact solutions to the nonlinear dynamics of learning in deep linear neural network. In *ICLR*, 2014.

[199] Walter J. Scheirer, Anderson de Rezende Rocha, Archana Sapkota, and Terrance E. Boult. Toward open set recognition. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(7):1757–1772, 2013.

[200] Frank Seide and Amit Agarwal. CNTK: Microsoft's Open-Source Deep-Learning Toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 2135–2135. ACM, 2016.

[201] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *ICCV*, pages 618–626. IEEE Computer Society, 2017.

[202] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. Grad-cam: Visual explanations from deep networks via gradient-based localization. In *ICCV*, pages 618–626. IEEE Computer Society, 2017.

[203] Ozan Sener and Silvio Savarese. Active learning for convolutional neural networks: A core-set approach. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[204] Burr Settles. Active learning literature survey. Computer Sciences Technical Report 1648, University of Wisconsin–Madison, 2009.

[205] Burr Settles, Mark Craven, and Soumya Ray. Multiple-instance active learning. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems*, volume 20. Curran Associates, Inc., 2008.

[206] H. S. Seung, M. Opper, and H. Sompolinsky. Query by committee. In *Proceedings of the Fifth Annual Workshop on Computational Learning Theory*, COLT '92, page 287294, New York, NY, USA, 1992. Association for Computing Machinery.

[207] Shayan Shams, Richard Platania, Kisung Lee, and Seung-Jong Park. Evaluation of deep learning frameworks over different hpc architectures. In *ICDCS*, 2017.

[208] Ali Shatnawi, Ghadeer Al-Bdour, Raffi Al-Qurran, and Mahmoud Al-Ayyoub. A Comparative Study of Open Source Deep Learning Frameworks. In *Information and Communication Systems (ICICS), 2018 9th International Conference on*, pages 72–77. IEEE, 2018.

[209] Zhiqiang Shen, Zhuang Liu, Jianguo Li, Yu-Gang Jiang, Yurong Chen, and Xiangyang Xue. Dsod: Learning deeply supervised object detectors from scratch. In *ICCV*, 2017.

[210] Flash Sheridan. Practical Testing of a C99 Compiler Using Output Comparison. *Software: Practice and Experience*, 37(14):1475–1488, 2007.

[211] S. Shi, Q. Wang, P. Xu, and X. Chu. Benchmarking State-of-the-Art Deep Learning Software Tools. In *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, pages 99–104, Nov 2016.

[212] Shaohuai Shi, Qiang Wang, Pengfei Xu, and Xiaowen Chu. Benchmarking state-of-the-art deep learning software tools. In *CCBD*, 2016.

[213] Hidetoshi Shimodaira. Improving predictive inference under covariate shift by weighting the log-likelihood function. *Journal of Statistical Planning and Inference*, 90(2):227–244, October 2000.

[214] Connor Shorten and Taghi M. Khoshgoftaar. A survey on image data augmentation for deep learning. In *Journal of Big Data*, 2019.

[215] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *arXiv preprint arXiv:1409.1556*, 2014.

[216] Xinhang Song, Luis Herranz, and Shuqiang Jiang. Depth cnns for rgb-d scene recognition: Learning from scratch better than transferring from rgb-cnns. In *AAAI*, 2017.

[217] Enrico Sorio, Alberto Bartoli, Giorgio Davanzo, and Eric Medvet. Open world classification of printed invoices. In *Proceedings of the 10th ACM Symposium on Document Engineering*, DocEng '10, page 187190, New York, NY, USA, 2010. Association for Computing Machinery.

[218] Siwakorn Srisakaokul, Zhengkai Wu, Angello Astorga, Oreoluwa Alebiosu, and Tao Xie. Multiple-implementation testing of supervised learning software. In *Proc. AAAI-18 Workshop on Engineering Dependable and Secure Machine Learning Systems (EDSMLS)*, 2018.

[219] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. In *JMLR*, 2014.

[220] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. Concolic testing for deep neural networks. In *ASE*, 2018.

[221] Ilya Sutskever, James Martens, George Dahl, and Geoffrey Hinton. On the importance of initialization and momentum in deep learning. In *ICML*, 2013.

[222] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. Sequence to sequence learning with neural networks. In *NIPS*, 2014.

[223] Christian Szegedy, Sergey Ioffe, and Vincent Vanhoucke. Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning. In *AAAI Conference on Artificial Intelligence*, 02 2016.

[224] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *CVPR*, 2016.

[225] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing Properties of Neural Networks. *arXiv preprint arXiv:1312.6199*, 2013.

[226] Nima Tajbakhsh, Jae Y Shin, Suryakanth R Gurudu, R Todd Hurst, Christopher B Kendall, Michael B Gotway, and Jianming Liang. Convolutional neural networks for medical image analysis: Full training or fine tuning? In *IEEE Trans. Med. Imag.*, 2016.

[227] Rohan Taori, Achal Dave, Vaishaal Shankar, Nicholas Carlini, Benjamin Recht, and Ludwig Schmidt. Measuring robustness to natural distribution shifts in image classification. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[228] Choon Teo, Amir Globerson, Sam Roweis, and Alex Smola. Convex learning with invariances. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems*, volume 20. Curran Associates, Inc., 2008.

[229] Kok Keong Teo, Lipo Wang, and Zhiping Lin. Wavelet packet multi-layer perceptron for chaotic time series prediction: effects of weight initialization. In *ICCS*, 2001.

[230] Kim-Han Thung, Pew-Thian Yap, and Dinggang Shen. Multi-Stage Diagnosis of Alzheimers Disease with Incomplete Multimodal Data via Multi-Task Deep Learning. In *Deep Learning in Medical Image Analysis and Multimodal Learning for Clinical Decision Support*, pages 160–168. Springer, 2017.

[231] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *ICSE*, 2018.

[232] Yuchi Tian, Ziyuan Zhong, Vicente Ordonez, Gail Kaiser, and Baishakhi Ray. Testing dnn image classifier for confusion & bias errors. In *ICSE*, 2020.

[233] Fabian Trautsch, Steffen Herbold, Philip Makedonski, and Jens Grabowski. Addressing problems with replicability and validity of repository mining studies through a smart data platform. In *ESE*, 2018.

[234] Nam Vo, Lu Jiang, Chen Sun, Kevin Murphy, Li-Jia Li, Li Fei-Fei, and James Hays. Composing text and image for image retrieval-an empirical odyssey. In *CVPR*, 2019.

[235] Huiyan Wang, Jingweiu Xu, Chang Xu, Xiaoxing Ma, and Jian Lu. Dissector: Input validation for deep learning applications by crossing-layer dissection. In *ICSE*, 2020.

[236] Jiannan Wang, Thibaud Lutellier, Shangshu Qian, Hung Viet Pham, and Lin Tan. Cradle: Cross-backend validation to detect and localize bugs in deep learning libraries. In *ICSE*, 2022.

[237] Jingyi Wang, Guoliang Dong, Jun Sun, Xinyu Wang, and Peixin Zhang. Adversarial sample detection for deep neural network through model mutation testing. In *ICSE*, 2019.

[238] Di Wen, Hu Han, and Anil K. Jain. Face spoof detection with image distortion analysis. *IEEE Transactions on Information Forensics and Security*, 10(4):746–761, 2015.

[239] Matthew Wicker, Xiaowei Huang, and Marta Kwiatkowska. Feature-Guided Black-Box Safety Testing of Deep Neural Networks. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 408–426. Springer, 2018.

[240] Danning Xie, Yitong Li, Mijung Kim, Hung Viet Pham, Lin Tan, Xiangyu Zhang, and Michael W. Godfrey. Docter: Documentation-guided fuzzing for testing deep learning api functions. In *ISSTA*, 2022.

[241] Hu Xu, Bing Liu, Lei Shu, and P. Yu. Open-world learning and application to product classification. In *The World Wide Web Conference on - WWW '19*. ACM Press, 2019.

[242] Jingkang Yang, Kaiyang Zhou, Yixuan Li, and Ziwei Liu. Generalized out-of-distribution detection: A survey, 2021.

[243] Pengpeng Yang, Daniele Baracchi, Rongrong Ni, Yao Zhao, Fabrizio Argenti, and Alessandro Piva. A survey of deep learning-based source image forensics. *Journal of Imaging*, 6(3), 2020.

[244] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and Understanding Bugs in C Compilers. In *ACM SIGPLAN Notices*, volume 46, pages 283–294. ACM, 2011.

[245] Takahide Yoshikawa, Kouya Shimura, and Toshihiro Ozawa. Random Program Generator for Java JIT Compiler Test System. In *Quality Software, 2003. Proceedings. Third International Conference on*, pages 20–23. IEEE, 2003.

[246] Steven R Young, Derek C Rose, Thomas P Karnowski, Seung-Hwan Lim, and Robert M Patton. Optimizing deep learning hyper-parameters through an evolutionary algorithm. In *MLHPC*, 2015.

[247] Sergey Zagoruyko and Nikos Komodakis. Wide residual networks. In *BMVC*, 2016.

[248] Andreas Zeller and Ralf Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[249] Hao Zhang and W. K. Chan. Apricot: A weight-adaptation approach to fixing deep learning models. In *ASE*, 2019.

[250] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. Retrieval-based neural source code summarization. In *ICSE*, 2020.

[251] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems. In *ASE*, 2018.

[252] Qirun Zhang, Chengnian Sun, and Zhendong Su. Skeletal program enumeration for rigorous compiler testing. In *PLDI*, 2017.

[253] Tong Zhang. The value of unlabeled data for classification problems. In *Proceedings of the Seventeenth International Conference on Machine Learning*, pages 1191–1198. Morgan Kaufmann, 2000.

[254] Xiyue Zhang, Xiaofei Xie, Lei Ma, Xiaoning Du, Qiang Hu, Yang Liu, Jianjun Zhao, and Meng Sun. Towards characterizing adversarial defects of deep learning software from the lens of uncertainty. In *ICSE*, 2020.

[255] Gang Zhao and Jeff Huang. Deepsim: Deep learning code functional similarity. In *ESEC/FSE*, 2018.

[256] Hengshuang Zhao, Jianping Shi, Xiaojuan Qi, Xiaogang Wang, and Jiaya Jia. Pyramid scene parsing network. In *CVPR*, 2017.

[257] Yan Zheng, Xiaofei Xie, Ting Su, Lei Ma, Jianye Hao, Zhaopeng Meng, Yang Liu, Ruimin Shen, Yingfeng Chen, and Changjie Fan. Wuji: Automatic online combat game testing using evolutionary deep reinforcement learning. In *ASE*, 2019.

[258] Husheng Zhou, Wei Li, Zelun Kong, Junfeng Guo, Yuqun Zhang, Lingming Zhang, Bei Yu, and Cong Liu. Deepbillboard: Systematic physical-world testing of autonomous driving systems. In *ICSE*, 2020.

[259] H. Zhu, M. Akrout, B. Zheng, A. Pelegris, A. Jayarajan, A. Phanishayee, B. Schroeder, and G. Pekhimenko. Benchmarking and analyzing deep neural network training. In *IISWC*, 2018.

[260] Rui Zhu, Shifeng Zhang, Xiaobo Wang, Longyin Wen, Hailin Shi, Liefeng Bo, and Tao Mei. Scratchdet: Training single-shot object detectors from scratch. In *CVPR*, 2019.

[261] Xiaojin Zhu, John Lafferty, and Zoubin Ghahramani. Combining active learning and semi-supervised learning using gaussian fields and harmonic functions. In *ICML 2003 workshop on The Continuum from Labeled to Unlabeled Data in Machine Learning and Data Mining*, pages 58–65, 2003.