Noname manuscript No.
(will be inserted by the editor)

Challenges and Practices of Deep Learning Model Reengineering: A Case Study on Computer Vision

Wenxin Jiang · Vishnu Banna · Naveen Vivek · Abhinav Goel · Nicholas Synovic · George K. Thiruvathukal · James C. Davis ·

Received: date / Accepted: date

Abstract *Context:* Many engineering organizations are reimplementing and extending deep neural networks from the research community. We describe this process as deep learning model reengineering.
Problem statement: Deep learning model reengineering — reusing, reproducing, adapting, and enhancing state-of-the-art deep learning approaches — is challenging for reasons including under-documented reference models, changing requirements, and the cost of implementation and testing. In addition, individual engineers may lack expertise in software engineering, yet teams must apply knowledge of software engineering and deep learning to succeed.
Related works: Prior work has characterized the challenges of deep learning model development, but as yet we know little about the deep learning model reengineering process and its common challenges. Prior work has examined on DL systems from

Wenxin Jiang
Purdue University, West Lafayette, IN, USA
E-mail: jiang784@purdue.edu

Vishnu Banna
Purdue University, West Lafayette, IN, USA
E-mail: vbanna@purdue.edu

Naveen Vivek
Purdue University, West Lafayette, IN, USA
E-mail: vivek@purdue.edu

Abhinav Goel
Purdue University, West Lafayette, IN, USA
E-mail: goel39@purdue.edu

Nicholas Synovic
Loyola University Chicago, Chicago, IL, USA
E-mail: jiang784@luc.edu

George K. Thiruvathukal
Loyola University Chicago, Chicago, IL, USA
E-mail: gkt@cs.luc.edu

James C. Davis
Purdue University, West Lafayette, IN, USA
E-mail: davisjam@purdue.edu

a "product" view, examining defects from projects regardless of the engineers' purpose. Our study is focused on reengineering activities from a "process" view, and focuses on engineers specifically engaged in the reengineering process.
Methodology: Our goal is to understand the characteristics and challenges of deep learning model reengineering. We conducted a case study of this phenomenon, focusing on the context of computer vision. Our results draw from two data sources: defects reported in open-source reeengineering projects, and interviews conducted with open-source project contributors and the leaders of a reengineering team. In the open-source data, we analyzed 348 defects from 27 open-source deep learning projects. Meanwhile, our reengineering team replicated 7 deep learning models over two years; we interviewed 2 practitioners and 6 reengineering team leaders to understand their experiences.
Results: Our results describe how deep learning-based computer vision techniques are reengineered, analyze the distribution of defects in this process, and discuss challenges and practices. We found that most defects (58%) are reported by reusers, and that reproducibility-related defects tend to be discovered during training (68% of them are). Our analysis shows that most environment defects (88%) are interface defects, and most of environment defects (46%) are caused by API defects. We also found that training defects have diverse impacts and root causes. We identified three main challenges in the DL reengineering process: model operationalization, portability of DL operations, and performance debugging. Integrating our quantitative and qualitative data, we proposed a novel reengineering workflow.
Future directions: Our findings inform several future directions, including: measuring additional unknown aspects of model reengineering; standardizing engineering practices to facilitate reengineering; and developing tools to support model reengineering and model reuse.

Keywords Empirical software engineering · Machine learning · Deep learning · Deep neural networks · Computer vision · Software reliability · Failure analysis · Bug study · Mixed methods · Case study

1 Introduction

Deep learning (DL) over neural networks achieves state-of-the-art performance on diverse tasks (Schmidhuber 2015), including games (Berner et al. 2019; Vinyals et al. 2019), language translation (Bahdanau et al. 2015; Wu et al. 2016), and computer vision (Ren et al. 2017; Redmon et al. 2016). After researchers demonstrate the potential of a DL approach in solving a problem, engineering organizations may incorporate it into their products. This software engineering task, of reusing, reproducing, and adapting state-of-the-art DL approaches, is challenging for reasons including mismatch between the needs of research and practice (Tatman et al. 2018; Hutson 2018), variation in DL libraries (Pham et al. 2020) and other environmental aspects (Unceta et al. 2020), and the high cost of model training and evaluation (Goyal et al. 2018). An improved understanding of the DL engineering process will help engineering organizations benefit from the capabilities of deep neural networks.

As illustrated in Figure 1, prior empirical studies have not fully explored the DL engineering process. These works have focused on understanding the characteristics of defects during DL development. These works consider defect characteristics and fix patterns, both in general (Humbatova et al. 2020; Sun et al. 2017; Zhang et al.

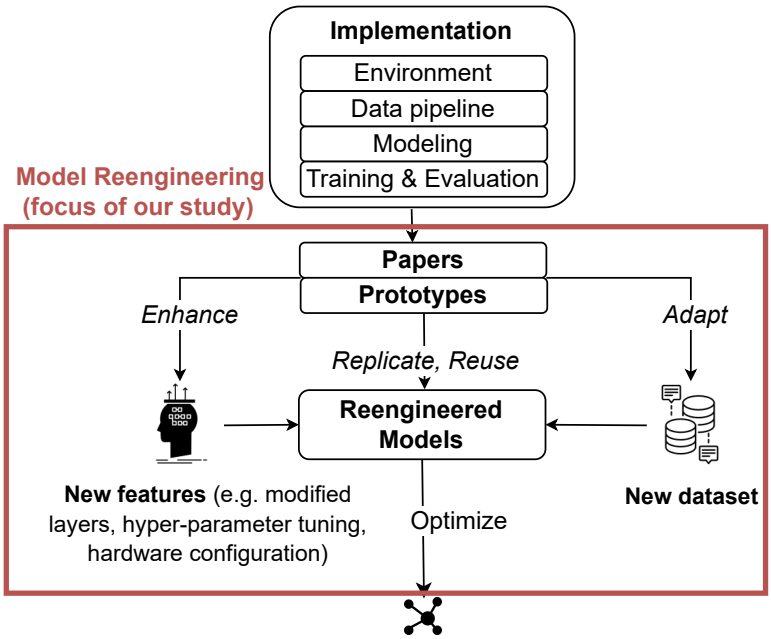

Fig. 1: High-level overview of a DL model development and application life cycle. Prior work may have accidentally captured reengineering activities, but it did not describe reengineering activities as a distinct activity and was generally from "product" view. We specifically focus on model reengineering activities and "process" view(red box).

We define DL *reengineering* as reusing, replicating, adapting or enhancing existing DL models.

2020a) and by DL framework (Zhang et al. 2018; Islam et al. 2019). In addition, these same works focus primarily on the "product" view of DL systems, which provides an incomplete view of software engineering practice.

Therefore, in this paper, we conducted a case study (Ralph et al. 2021) to examine the **Deep Learning reengineering process**: activities to *reuse, replicate, adapt, or enhance an existing DL model*. We used a mixed-methods approach and drew from two complementary data sources (Johnson and Onwuegbuzie 2004). First, to explore the characteristics, challenges, and practices of CV reengineering, we analyzed 348 defects from 27 open-source DL projects (§5.1). Second, we describe the qualitative reengineering experiences of two open-source engineers and a DL reengineering team (§5.2).

Combining these data, we report the challenges and practices of DL reengineering from a "process" view (§6). From our defect study, we observed that DL reengineering defects varied by DL stage (§6.1): environmental configuration is dominated by API defects (§6.4); the data pipeline and modeling stages are dominated by errors in assignment and initialization (§6.2); and errors in the training stage take diverse forms (§6.4). The performance defects discovered in the training stage are the most difficult to repair (§6.3). From our interview study we identified similar challenges, notably in model implementation and in performance debugging (§6.5). These problems often arose from a lack of portability, *e.g.,* to different

hardware, operating environment, or library versions. Interview subjects described their testing and debugging techniques to address these challenges. Synthesizing these data sources, we propose an idealized DL reengineering workflow §7. The difficulties we identify in DL reengineering suggest that researchers should investigate DL software testing, and that techniques to support the reuse of pre-trained models could mitigate many problems.

In summary, our main contributions are:

- We conducted the first study that takes a "process" view of DL reengineering activities (§5).
- We analyze 348 defects from 27 repositories in order to describe the characteristics of the defects in DL reengineering projects (§6.1–§6.4).
- We complement this quantitative failure study with qualitative data on reengineering practices and challenges (§6.5). Our subjects included open-source practitioners. However, due to recruiting difficulties, we also coordinated a two-year engineering effort by a student team to enrich this part of the study. To the best of our knowledge, this second approach is novel.
- We illustrate the challenges and practices of DL model reengineering with a novel reengineering process workflow. We propose future directions for practical and empirical research based on our results and analysis (§7).

2 Background and Related Work

2.1 Reengineering in Machine Learning and Deep Learning

Historically, *software reengineering* referred to replicating, understanding, or improving an existing implementation (Linda et al. 1996). This process arises from needs including optimization, adaptation, and enhancement (Jarzabek 1993; Byrne 1992; Tucker and Devon 2010). Today, we observe that ML and DL engineers are reusing, replicating, adapting, and enhancing existing models to understand the algorithms and improve their implementation (Amershi et al. 2019a; Alahmari et al. 2020). The maintainers of major ML frameworks, including TensorFlow and Pytorch, store reengineered models within official GitHub repositories (Google 2022; Meta 2022). Many engineering companies, including Amazon (Schelter et al. 2018), Google (Kim and Li 2020), and Meta (Pineau 2022) are engaged in forms of *and DL reengineering*. For example, Google sponsors the TensorFlow Model Garden which provides "a centralized place to find code examples for state-of-the-art models and reusable modeling libraries" (Kim and Li 2020). Many research papers use PyTorch because it is easy to learn and has been rapidly adopted in research community, but many companies prefer TensorFlow versions because of it provides better visualization and robust deployment (O'Connor 2023).

ML reengineering has received much attention in the research community (Hutson 2018; Pineau et al. 2020; Gundersen and Kjensmo 2018; Gundersen et al. 2018). Pineau *et al.* noted three needed characteristics for ML software: reproducibility, re-usability, and robustness. They also proposed two problems regarding reproducibility: an insufficient exploration of experimental variables, and the need for proper documentation (Pineau et al. 2020). Gundersen *et al.* surveyed 400 AI publications, and indicated that documentation facilitates reproducibility. They proposed a reproducibility checklist (Gundersen and Kjensmo 2018; Gundersen et al. 2018). Chen *et al.* highlights that the reproducibility of DL models is still a challenging task as of 2022 (Chen et al. 2022a). Consequently, the characteris-

tics of the reengineering process are significant for both practitioners (Villa and Zimmerman 2018; Pineau 2022) and researchers (MLR 2020; Ding et al. 2021).

The combination of the software, hardware, and neural network problem domains exacerbates the difficulty of deep learning reengineering. The DL ecosystem is evolving, and practitioners have varying software environments and hardware configurations (Boehm and Beck 2010). This variation makes it hard to reproduce and adapt models (Goel et al. 2020). Additionally, neural networks are reportedly harder to debug than traditional software, *e.g.,* due to their lack of interpretability (Bibal and Frénay 2016; Doshi-Velez and Kim 2017).

To facilitate the reengineering of DL systems, researchers advise the community to increase the level of portability and standardization in engineering processes and documentation (Gundersen and Kjensmo 2018; Pineau et al. 2020; Liu et al. 2020). Microsoft indicated that existing DL frameworks focus on runtime performance and expressiveness and neglect composability and portability (Liu et al. 2020). The lack of standardization makes finding, testing, customizing, and evaluating models a tedious task (Gundersen et al. 2018). These tasks require engineers to "glue" libraries, reformat datasets, and debug unfamiliar code — a brittle, time-consuming, and error-prone approach (Sculley et al. 2014).

To support DL engineers, we conducted a case study on the defects, challenges, and practices of DL reengineering.

2.2 Empirical Studies on Deep Learning Engineering Processes

DL models are being adopted across many companies. With the demand for engineers with DL skills far exceeding supply (Nahar et al. 2022), companies are looking for practices that can boost the productivity of their engineers. Google (Breck et al. 2017), Microsoft (Amershi et al. 2019a), and SAP (Rahman et al. 2019) have provided insights on the current state of the DL development and indicate potential improvements. Breck *et al.* indicated that it is hard to create reliable and production-level DL systems (Breck et al. 2017). Amershi *et al.* proposed the requirements of model customization and reuse, *i.e.,* adapting the model on different datasets and domains (Amershi et al. 2019a). Rahman *et al.* pointed out that knowledge transfer is one of the major collaboration challenges between industry and academia (Rahman et al. 2019). Our work identifies challenges and practices of knowledge transfer from academia to industry and supports creating reliable customized models.

In addition to views of the industry, there are empirical works on DL software engineering practices from academic perspectives. Zhang *et al.* illustrated the need for cross-framework differential testing and the demand for facilitating debugging and profiling (Zhang et al. 2019). Serban *et al.* discussed the engineering challenges and the support of development practices, and highlighted the importance of experiment management, feature management, prototyping, and testing (Serban et al. 2020). Lorenzoni *et al.* showed how DL developers could benefit from a traditional software engineering approach and proposed improvements in the ML development workflow (Lorenzoni et al. 2021).

Figure 1 illustrates how our work differs from previous empirical studies. Although there have been many software engineering studies on DL, they collect data from a "product" view of DL systems (Islam et al. 2019; Shen et al. 2021; Chen et al. 2022b; Lorenzoni et al. 2021). These works sample open-source defects or Stack Overflow questions and report on the features, functionalities, and

quality of DL software. These studies may accidentally capture reengineering activities, but they did not consider reengineering as a distinct activity. Some work mentioned specific reengineering activities, such as model customization (Amershi et al. 2019b) and knowledge transfer of DL technologies (Rahman et al. 2019). However, there is no work focusing on the reengineering activities yet. We conduct the first empirical study of DL software from a "process" view which focuses on the activities and processes involved in creating the software product. We specifically examine the DL reengineering activities and process by sampling the defects from open-source research prototypes and replications. We also collected qualitative data about the activities and process by interviewing open-source contributors and leaders of the student reengineering team.

2.3 Deep Learning Bugs and Fix Patterns

Prior work focused on bugs in the general DL development process, studying the defects, characteristics, and impacts. Islam *et al.* demonstrated DL bug characteristics from 5 DL libraries in GitHub and 2716 Stack Overflow posts (Islam et al. 2019). Humbatova *et al.* analyzed data from Stack Overflow and GitHub to obtain a taxonomy of DL faults (Humbatova et al. 2020). By surveying engineers and researchers, Nikanjam *et al.* specified eight design smells in DL programs (Nikanjam and Khomh 2021).

Furthermore, researchers conducted works on DL fix patterns (Sun et al. 2017; Islam et al. 2020). Sun *et al.* analyzed 329 bugs for their fix category, pattern, and duration (Sun et al. 2017). Islam *et al.* considered the distribution of DL fix patterns (Islam et al. 2020). Their findings revealed a distinction between DL bugs and traditional ones. They also identified challenges in the development process: fault localization, reuse of trained models, and coping with frequent changes in DL frameworks. Some development defects studied from prior work are "wrong tensor shape" (Humbatova et al. 2020) and "`ValueError` when performing `matmul` with TensorFlow" (Islam et al. 2019). However, typical reengineering bugs have not been well discovered, such as "user's training accuracy was lower than what was claimed" or "training on a different hardware is very slow" (Table 3).

Prior work considered the defects in the DL model development process, without distinguishing which part of the development process they were conducting. In this work, we focus on defects arising specifically during the DL model reengineering process (Figure 1). We use defect data from GitHub repositories. We also collect interview data, providing an unusual perspective — prior studies used data from Stack Overflow (Islam et al. 2019; Zhang et al. 2018; Humbatova et al. 2020), open-source projects (Zhang et al. 2018; Islam et al. 2019; Sun et al. 2017; Humbatova et al. 2020; Shen et al. 2021; Garcia et al. 2020), and surveys (Nikanjam and Khomh 2021).

3 Research Questions

To summarize the literature: DL reengineering is common in engineering practice, but is challenging for reasons including (under-) documentation, shifting software and hardware requirements, and unpredictable computational expense. Prior work has studied the problems of DL engineering, *e.g.,* considering the characteristics of DL defects (Zhang et al. 2020b) sometimes distinguished by DL framework (Zhang et al. 2018). However, this prior work has not distinguished between the activities

of DL development and DL reengineering, and has not examined DL reengineering specifically.

We define **DL model reengineering** as: *reusing, replicating, adapting, or enhancing an existing DL model*. DL reengineering challenges and practices have not been studied. We ask:

RQ1 *How do defects manifest in Deep Learning reengineering?*
RQ2 *What types of Deep Learning reengineering defects are frequent?*
RQ3 *What are the impacts of Deep Learning reengineering defects?*
RQ4 *What are the root causes of Deep Learning reengineering defects?*
RQ5 *What are the challenges and practices in Deep Learning reengineering?*

4 Model of Deep Learning Reengineering

4.1 Reengineering Concepts

To have a better understanding of DL reengineering defects, we chose to study the defect manifestation, types, impacts, and root causes separately, following prior work (Islam et al. 2019; Wang et al. 2017; Liu et al. 2014). We did some preliminary analysis on open-source projects to understand the characteristics of deep learning projects which involve reengineering activities. Based on our literature review and observation in preliminary analysis, here we introduce the relevant concepts specific to DL reengineering process — defect *reporters* (Table 1) and defect *manifestations* (Table 2).

Table 1 defines four different types of *Defect reporters* based on their reengineering activities. Prior work indicates that there are inconsistencies in the use of these terms (Gundersen and Kjensmo 2018). Our definitions are mainly based on prior work(Pineau et al. 2020; Li et al. 2020; Git 2020; Amershi et al. 2019a; Alahmari et al. 2020) and our observations in preliminary analysis.

Table 1: Reporter types in the DL reengineering ecosystem. The reporter types are determined by whether they use the same code, dataset, and algorithm compared to the upstream project.

Reporter Category	Description
Re-user	Uses same code and same dataset. This is the common-case behavior associated with pure re-use.
Adaptor	Adapts code to other tasks (different dataset) and finds inconsistency compared to expectations.
Enhancer	Adds new features (*e.g.,* layer modification, hyper-parameter tuning, multi-GPU training configuration).
Replicator	Uses same algorithm, data, and configuration, in distinct implementation (*e.g.,* TensorFlow vs. PyTorch).

Table 2 defines three types of *Defect manifestation*. We followed prior work studying the manifestation of defects (Liu et al. 2014). Manifestation provides an unusual view combining symptoms, root causes00, and relevant contexts. We believe studying defect manifestations can give us a unique perspective of reengineering process.

Table 2: Defect manifestations and relevant definitions, determined by the runnability of code and the data used to train the model.

Defect Category	Description
Basic defects	The code does not run (*e.g.,* it crashes, behaves very incorrectly, or runs out of memory).
Reproducibility defects	Using the same data, the code runs without basic defects, but does not match the documented performance (*e.g.,* accuracy, latency).
Evolutionary defects	The code and/or data has been changed to adapt to the user's needs. It runs without basic defects, but does not match the specification/desired performance.

4.2 Open-source Relationships

Based on the typical open-source software collaboration process (Fitzgerald 2006), we modeled the expected relationship between CV reengineering repositories — see Figure 2. This figure depicts the relationship between three types of projects:

- *Research Prototype*: an original implementation of a model.
- *Replications*: a replication and possible evolution of the research prototype.
- *User "fork"*: a GitHub fork, clone, copy, etc. where users try to reuse or extend a model from the previous two types.

We expect the existence of two actions, "*REUSE* (fork)" and "*REPORT (defect)*", happening between different project types. The down-stream projects reuse up-stream projects, and engineers open new issues when they encounter defects or identify necessary enhancements. Typically the issues are reported in upstream projects, the maintainers explain the necessary fix patterns for reengineers, and the defects are repaired in downstream ones. Because our goal is to study reengineering defects, we do not study "forks", but instead examine the up-stream projects: research prototypes and replications.

4.3 Reengineering Repository Types

We identified two types of repositories during our preliminary analysis: *zoo* and *solo* repository. A *Solo* repository is either the research prototype from the authors or an independent replication. *Zoo repositories* all contain implementations of several models. A *zoo repository* can be the combination of research prototypes and replications. For example, `TensorFlow Model Garden` contains different versions of YOLO (Google 2022). These *zoo repositories* have been widely reused and contain many relevant issues (Table 4).

5 Methodology

To answer our research questions, we used a case study approach (Perry et al. 2004; Runeson and Höst 2009) that combined two perspectives (Johnson and Onwuegbuzie 2004). For RQ1–RQ4, we analyzed DL reengineering defects in open-source reengineering projects. However, this data source may be constraining because GitHub issues are known to be limited in the breadth of issue types and the depth of detail (Aranda and Venolia 2009). To answer RQ5, we collected qualitative reengineering experiences from open-source contributors and the leaders of a DL

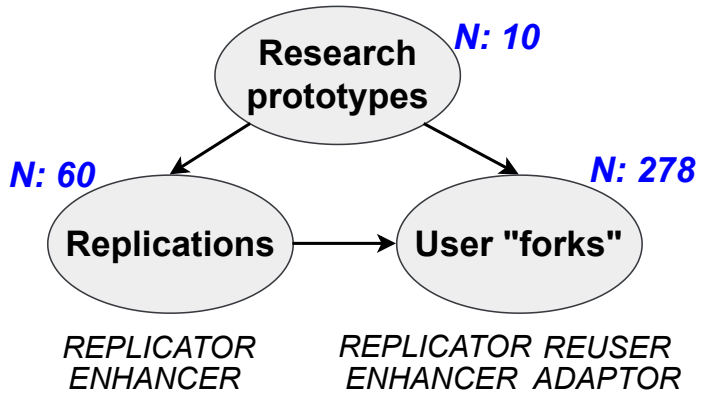

Fig. 2: Relationship between CV reengineering repositories. The capitalized texts indicate what types of reporters commonly open issues in the above projects.
Arrows indicate the dataflow of model reuse between upstream and downstream projects. *N*: the number of defects resolved in each type of the projects we analyzed.

reengineering team after they completed several projects. These two perspectives are complementary: the open-source defects gave us broad insights into some kinds of reengineering defects, while the DL reengineering team gave us deep insights into the reengineering challenges and process. Figure 3 shows the relationship between our questions and methods. To promote replicability and further analysis, all data is available in our artifact (§10).

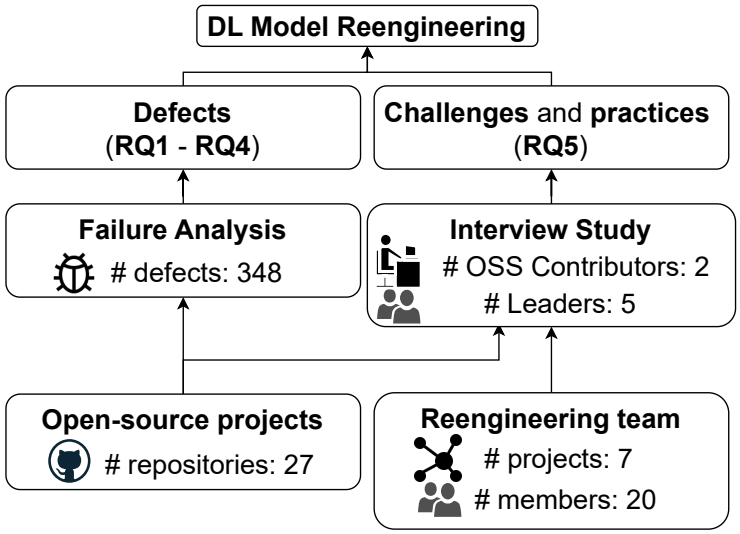

Fig. 3: Relationship of research questions to methodology. The failure analysis is conducted on open-source projects, while the interview study is conducted on both open-source contributors and reengineering team members.

In our case study, the phenomenon of interest was DL reengineering. We situated this study in the context of Computer Vision (CV). As a major application domain of DL techniques (Voulodimos et al. 2018; CVE 2021; Xu et al. 2021), CV serves as a microcosm of DL engineering processes (Amershi et al. 2019a; Thiruvathukal et al. 2022).[1] Our case study will thus provide findings for computer vision reengineering projects, which are important; and our results may generalize to other domains of deep learning.

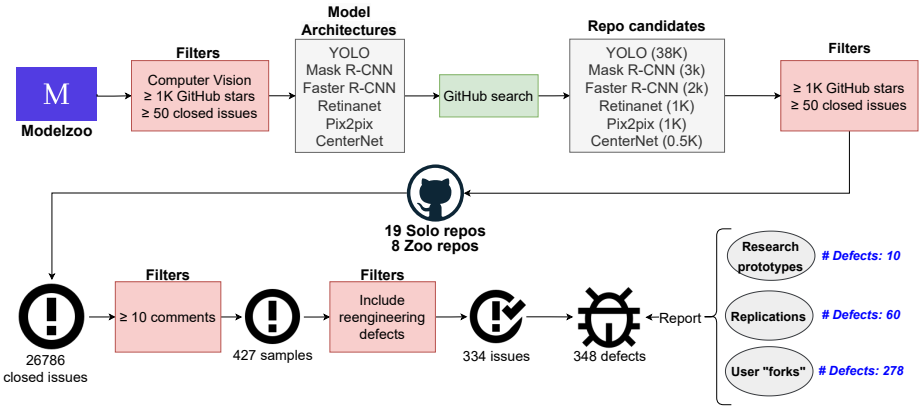

Fig. 4: Overview of defect collection and distribution of the collected defects in three project types, as well as the number of projects and defects we got in each step. The number of each model architecture after GitHub search (Most data were collected in 2021. Data associated with "Repo candidates" were collected in Feb, 2023)

5.1 Failure Analysis on Computer Vision Reengineering

Figure 4 shows the overview of our data collection in the failure analysis (bug study). We adapted previous failure analysis methods (Islam et al. 2019; Zhang et al. 2018; Wang et al. 2020; Eghbali and Pradel 2020) to identify and analyze defects in DL reengineering. In total, we examined 348 defects from 27 repositories (Table 4). The rest of this section discusses our selection and analysis method.

5.1.1 Repository Selection

To find sufficient DL reengineering defects in this target — research prototypes and replication projects — we chose to look at CV models with substantial popularity and GitHub issues. We proceeded as follows, along the top row of Figure 4:

1. We started by selecting popular CV models from ModelZoo (Jing 2021), a platform that tracks state-of-the-art DL models. We looked at relevant GitHub repository and chose the CV models with over 1K GitHub stars and 50 closed issues.
2. We searched for implementations of these model architectures on GitHub.

[1] We acknowledge that there are approaches to computer vision that do not leverage deep learning (Szeliski 2022; Forsyth and Ponce 2002). In this study, we focused on engineers applying deep learning to problems in computer vision.

3. For each of these architectures, we chose 2–5 projects implementing the same model (sorted by the number of stars; all having at least 1K GitHub stars and 50 closed issues (Borges and Valente 2018)). If there was only one implementation matching our criteria, we excluded that model in our analysis.

As shown in Table 4 (last 4 rows), for two architectures this process did not always yield both prototypes and replications. For `Pix2pix` we retrieved two prototypical implementations in different programming languages (*viz.* Lua and Python). For `CenterNet` we retrieved two prototypes for different model architectures that share the same model name. However, on inspection we found that these four repositories were actively reused, had many people engaged in the reengineering process, and had many reengineering defects. Therefore, we included the issues from these repositories in our analysis.

As shown in Figure 4, the same model architecture can be found in either *zoo repositories* or *solo repositories* during GitHub searching (§4.3). Most of the repositories (19/27) we identified during the GitHub searching are *solo repositories* which only implement a single model. Both *solo* and *zoo repositories* have DL reengineering defects reported by down-stream replicators or users. Therefore, we applied the same data collection methods to them and put the data together.

Overall, we examined 19 *Solo Repositories* and 8 *Zoo Repositories*.

5.1.2 Issue Selection

We applied two filters on issues in these repositories: (1) Issue status is *closed* with an associated *fix*, to understand the defect type and root causes (Tan et al. 2014); (2) Issue has ≥ 10 comments, providing enough information for analysis. We first used the two filters to filter the full set of issues in each repository, and then sampled the remainder.

Our goal was to analyze at least 10% of the issues for each reengineering project, but this was balanced against the wide range of qualifying issues for each repository. For example, `ultralytics/yolov5` has 279 qualifying issues while `yhenon/pytorch-retinanet` has only 4. We first conducted a pilot study on 5 solo projects which includes 69 defects.[2] The pilot study indicated that choosing the 20 most-commented issues would cover roughly 10% of the issues for these projects and give us plenty of data for analysis.

We checked and found that this approach would also work well for most of the zoo repositories. For two zoo repositories (`tensorflow/models` and `pytorch/vision`), the most-commented issues are controversial API change requests, not reengineering defects, so we randomly sampled 10% of the closed issues instead. For some smaller repositories, taking the top-20 qualifying issues consumed all available defects (Table 4).

For most of the selected repositories, we sorted the remaining issues by the number of comments and examined the 20 issues with the most comments. This sample constituted $\geq 10\%$ of their issues. For the zoo repositories from two major DL frameworks, `tensorflow/models` and `pytorch/vision`, the most-commented issues were API change requests, not defects. Here, we randomly sampled 10% of issues that met the first two filters.

[2] More details can be found in §5.1.3.

From these samples, we manually filtered out 93 non-defect issues, *e.g.,* development questions and feature requests. *78% (334/427)* of sampled issues included a reengineering defect. Table 3 provides four examples.

Table 3: Examples of reengineering defects included in our study, and non-reengineering defects excluded from the study. A reengineering defect refers to an error or flaw that occurs during the process of reusing, replicating, adapting, or enhancing an existing deep learning model.

Issue ID	**Type**	**Defect**	**Fix**
tensorflow/models #6043	Reeng.	The losses occasionally contained a NaN value when using a customized dataset.	Replace the empty sequence in tensor with zeros.
ultralytics/yolov3 #310	Reeng.	User training accuracy was lower than what was claimed by the replicator in the documentation.	Load the original checkpoint weights before training.
matterport/Mask_RCNN#1938	Reeng.	Training on the RTX 2080 Ti GPU with CUDA library version 9 is slow.	Fix the environment by following provided instructions and reduce the training epochs.
facebookresearch/detectron2#3225	Reeng.	Documentation does not seem to have been updated to reflect the new config files (`.py` rather than `.yaml`)	Add documentation for new training script to use existing configuration files.
rbgirshick/py-faster-rcnn #123	Non-reeng.	The user asked how to configure the repository with "CPU ONLY" mode.	N/A
tensorflow/models #1838	Non-reeng.	`pip install` does not work to install specific version of TensorFlow.	N/A

Table 4: The studied repositories and defects. The *Closed issues (qualified)* indicates the total closed issues and those matching two filters (closed, $\geq$ 10 comments). *Samples* are sampled from the qualified closed issues. After these selection criteria, we manually identified 334 GitHub issues from the samples that described at least one reengineering defect. Some GitHub issues contained multiple reengineering defects. †Repository `rwightman/pytorch-image-models` was converted to `huggingface/pytorch-image-models` after the study completed.

Repository	Type	Stars (K)	Forks (K)	Closed issues (qualified)	Samples	Reeng. issues (defects)
Zoo Repos						
tensorflow/models *(Google)*	Zoo	71.8	45.0	5560 (580)	58	51 (51)
facebookresearch/Detectron *(Facebook)*	Zoo	24.8	5.4	613 (27)	20	15 (15)
facebookresearch/detectron2 *(Facebook)*	Zoo	18.7	5.0	2605 (90)	20	12 (13)
open-mmlab/mmdetection	Zoo	17.1	6.1	4348 (155)	20	16 (16)
rwightman/pytorch-image-models†	Zoo	14.3	2.3	414 (11)	11	5 (5)
pytorch/vision *(Facebook)*	Zoo	10.2	5.3	1504 (139)	20	14 (14)
NVIDIA/DeepLearningExamples *(NVIDIA)*	Zoo	6.7	2.0	436 (22)	20	18 (18)
qubvel/segmentation_models	Zoo	3.5	0.8	167 (12)	12	8 (8)
YOLO						
ultralytics/yolov5	Solo (Proto.)	18.2	6.3	3795 (279)	20	19 (21)
ultralytics/yolov3	Solo (Repl.)	8.0	3.0	1671 (204)	20	17 (21)
qqwweee/keras-yolo3	Solo (Repl.)	6.9	3.4	226 (16)	16	13 (13)
eriklindernoren/PyTorch-YOLOv3	Solo (Repl.)	6.3	2.5	557 (26)	20	18 (19)
YunYang1994/tensorflow-yolov3	Solo (Repl.)	3.5	1.4	116 (3)	3	3 (4)
Mask R-CNN						
matterport/Mask_RCNN	Solo (Proto.)	20.9	10.2	783 (62)	20	15 (16)
CharlesShang/FastMaskRCNN	Solo (Repl.)	3.1	1.1	55 (2)	2	1 (1)
TuSimple/mx-maskrcnn	Solo (Repl.)	1.8	0.5	83 (7)	7	6 (6)
Faster R-CNN						
ShaoqingRen/faster_rcnn	Solo (Proto.)	2.5	1.2	53 (5)	5	4 (4)
rbgirshick/py-faster-rcnn	Solo (Repl.)	7.5	4.1	253 (33)	20	17 (17)
jwyang/faster-rcnn.pytorch	Solo (Repl.)	6.6	2.2	363 (34)	20	17 (18)
endernewton/tf-faster-rcnn	Solo (Repl.)	3.6	1.6	65 (15)	15	12 (12)
chenyuntc/simple-faster-rcnn-pytorch	Solo (Repl.)	3.4	1.0	267 (2)	2	1 (2)
Retinanet						
fizyr/keras-retinanet	Solo (Proto.)	4.2	2.0	1227 (90)	20	16 (18)
yhenon/pytorch-retinanet	Solo (Repl.)	1.8	0.3	78 (4)	4	2 (2)
pix2pix						
junyanz/pytorch-CycleGAN-and-pix2pix	Solo (Proto.)	16.2	4.9	847 (48)	20	15 (15)
phillipi/pix2pix	Solo (Proto.)	8.7	1.6	119 (13)	13	5 (5)
CenterNet						
xingyizhou/CenterNet	Solo (Proto.)	6.0	1.7	542 (17)	17	12 (12)
Duankaiwen/CenterNet	Solo (Proto.)	1.8	0.6	68 (2)	2	2 (2)
Total				**26786**	**427**	**334 (348)**

5.1.3 Issue Analysis: Instrument

We noticed that previous studies have proposed comprehensive taxonomies for DL defects. To test the reusability of existing taxonomies for DL reengineering defects, we did a pilot study on 69 issues from 5 repositories. Our goal was to achieve saturation in the pilot study. We made several changes to the instrument as we analyzed the first 3 repositories, but these tailed off in the 4th repository and we did not need to make any changes in the 5th repository. After this, we decided to finalize the instrument. Our pilot study indicated that DL reengineering defects are a subset of DL defects (as anticipated by Figure 1) and existing taxonomies can categorize DL reengineering defects.

We reorganized the existing taxonomies by distinguishing between four DL stages: environment, data pipeline, modeling, and training (Amershi et al. 2019a; MLO 2021). We reused a taxonomy of general programming defects (*e.g.,* interface defects, non-functional defects) from (Thung et al. 2012). To better characterize the defects, we revise our instrument by adding categories from other works, as shown in Table 5:

5.1.4 Issue Analysis: Process

Our classification and labeling process build on prior studies of DL defects (Islam et al. 2019; Humbatova et al. 2020). A graduate researcher first developed an instrument and led a pilot study on 69 defects from 5 repositories. To calibrate the analysis instrument, a team of 6 undergraduate researchers worked in pairs, and independently analyzed the same set of defects. One graduate student supervised the team, helped to address uncertainties, and refined the instrument based on the discussions. In our taxonomy, we consider the categories with less than five counted defects as *low frequency categories* to better present our results.

We analyzed the data from the pilot study twice, once using the original instrument and once using the improved instrument. After modifying the instrument to address ambiguities, we had two of the researchers re-annotated the data using the improved definitions and clarifications. Most of our annotators were undergraduate students who were unable to continue working on the project after the summer, so the rest half of our study was done by the most experienced rater, who sought a second opinion on a few uncertainties.

We used the Cohen's Kappa measure for inter-rater agreement (Cohen 1960). The kappa statistics can represent the inter-rater reliability (McHugh 2012). Using the original instrument, the Cohen's Kappa Value was 0.46 ("moderate"). The Kappa value for the modified instrument was 0.79 ("substantial"). We also measured half of the rest study and the Cohen's Kappa value was 0.70 ("substantial").

The data and scripts used to calculate the Cohen's Kappa value are available in §10.

5.2 Interview Study on Computer Vision Reengineering

To enrich our understanding of DL reengineering challenges and practices, we triangulated the failure analysis with an interview study of open-source contributors and a CV reengineering team.

5.2.1 Computer Vision Reengineering Team

Motivated by the method of measuring software experiences in the software engineering laboratory described by Valett and McGarry (1989), our lab organized a

Table 5: Taxonomy for defect impacts. The impacts were adapted from Islam et al. (2019) by distinguishing two types of *Bad Performance*: *Accuracy/Speed Below Expectations*, referring to the symptoms defined by Zhang et al. (2018). The *expectations* can be different from the documentation if the code or data change. We also added *Numerical Errors* based on Wardat et al. (2021). **Bold**: Changed categories.

Defect Impact Category	Description
Speed Below Expectations	The code runs but the training/inference time does not match the expectation.
Accuracy Below Expectations	The code runs but the evaluation results do not match the expected accuracy.
Numerical Error	The results are Inf, NaN or Zero which are caused by division (*i.e.,* division by zero returns not-a-number value), logarithm (*i.e.,* logarithm of zero returns -∞ that could be transformed into not-a-number); Or the results appear random for each running; Or floating point overflow.
Crash	The system stops unexpectedly.
Data Corruption	The data is corrupted as it flows through the model and causes unexpected outputs.
Hang	The system ceases to respond to inputs.
Incorrect functionality	The system behaves in an unexpected way without any runtime or compile-time error/warning.
Memory exhaustion	The system halts due to unavailability of the memory resources. This can be caused by, either the wrong model structure or not having enough computing resources to train a particular model.
Other	Other impacts that do not fall into one of the above categories.

CV reengineering team, focused on the replication of CV research prototypes. The goal of this team is to provide high-quality implementations of state-of-the-art DL models. Most team members are third- and fourth-year undergraduate students. Their work is supervised and approved by Google engineers over weekly sync-ups.

The team's industry sponsor (Google) uses our replication of 7 models in production in their ML applications and publishes them open-source in one of the most popular zoo repository we studied in §5.1, TensorFlow Model Garden. Each of the team's projects had 1-2 student leaders. The team leaders were fourth-year undergraduates, sometimes working for pay and sometimes for their senior thesis. Team leaders typically worked for 15-20 hours per week over their 2 years on the team. All team leaders contributed to at least two reengineering projects. All team members received team-specific training via our team's 6-week onboarding course on DL reengineering.

The completed models required 7,643 lines of code, measured using the cloc tool (AlDanial 2022). The estimated cost of the reengineering team's work was

Table 6: Taxonomy for defect types. We reused a taxonomy of general programming defects from Seaman et al. (2008), adding the "Configuration" category from Thung et al. (2012). For convenience, this table presents the combined taxonomy.

Defect Type Category	**Description**
Algorithm/method	An error in the sequence or set of steps used to solve a particular problem or computation, including mistakes in computations, incorrect implementation of algorithms, or calls to an inappropriate function for the algorithm being implemented.
Assignment/Initialization	A variable or data item that is assigned a value incorrectly or is not initialized properly or where the initialization scenario is mishandled (*e.g.,* incorrect publish or subscribe, incorrect opening of file).
Checking	Inadequate checking for potential defect conditions, or an inappropriate response is specified for defect conditions.
Data	Defects in specifying or manipulating data items, incorrectly defined data structure, pointer or memory allocation errors, or incorrect type conversions.(*i.e.,* Array, Linked List, Stack, Queue, Trees, Graphs)
External Interface	Defects in the user interface (including usability problems) or the interfaces with other systems. (e.g. API defects)
Internal Interface	Defects in the interfaces between system components, including mismatched calling sequences and incorrect opening, reading, writing, or closing of files and databases.
Logic	Incorrect logical conditions, including incorrect blocks, incorrect boundary conditions being applied, or incorrect expression.
Timing/optimization	Errors that will cause timing or performance problems.
Non-functional Defects	Includes non-compliance with standards, failure to meet non-functional requirements such as portability and performance constraints, and lack of clarity of the design or code to the reader.
Configuration (Thung et al. 2012)	Defects in non-code (*e.g.,* configuration files) that affects functionality.
Other	Other defects that do not fall into one of the above categories.

∼$105K ($15K/model): $40K on wages and $65K on computation ($5K for VM and storage, $60K for hardware accelerator rental [e.g., GPU, TPU]).

5.2.2 Data Collection

We recruited the participants from open-source model contributors (§5.1) and the CV reengineering team (§5.2.1). We sent out emails to the main contributor of

Table 7: Taxonomy for defect root causes. We adapted the taxonomy from (Humbatova et al. 2020) by reorganizing the categories into four DL stages. We distinguish the categories of data preprocessing and corrupt data (data flow bug) based on (Islam et al. 2019). **Bold**: changed/new categories.

DL Stage	Root Cause Category	Description
Data pipeline	**Data preprocessing**	If an input to the deep learning software is not properly formatted, cleaned, well before supplying it to the deep learning model.
	Corrupt data (data flow bug)	Due to the type or shape mismatch of input data after it has been fed to the DL model.
	Training data quality	Due to the complexity of the data and the need for manual effort to ensure a high quality of training data (*e.g.,* to label and clean the data, to remove the outliers).
Modeling	Activation function	Incorrectly selecting the activation function of neurons.
	Layer properties	Some layer's incorrect inner properties (*e.g.,* input/output shape, input sample size, number of neurons in it).
	Missing/Redundant/Wrong layer	Adding, removing or changing the type of a specific layer was needed to remedy the low accuracy of a network.
Training	Optimizer	The selection of an unsuitable optimization function for model training.
	Loss function	Wrong selection and calculation of the loss function.
	Evaluation	Problems caused by testing and validation (*e.g.,* bad choice of performance metrics)
	Hyper-parameters	Incorrectly tuning the hyperparameters (*e.g.,* learning rate, batch size, number of epochs) of a DL model.
	Training configuration	Wrong training scripts.
	Other training process	Other faults in the training process which do not fall into one of the above categories (*e.g.,* wrong management of memory resources, wrong post-processing of the output)
Environment	API defect	Caused by APIs, this includes API mismatch, API misuse, API change, *etc.*
	GPU Usage bug	Wrong usage of GPU devices while working with DL (*e.g.,* wrong reference to GPU device, failed parallelism, incorrect state sharing between subprocesses, faulty transfer of data to a GPU device).
	Wrong environment configuration	Incorrect setting of other configurations (*e.g.,* wrong operating systems, internal interface defects).
Other	**Insufficient/Incorrect documentation**	Engineers misunderstood the documentation or they cannot find correct or sufficient instructions.

each repository in Table 4. Overall, we interviewed the 6 leaders of the team after the team had been operating for 18 months, as well as 2 open-source contributors.

To design our interview, we followed the guideline of *framework analysis* which is flexible during the analysis process (Srivastava and Thomson 2009). A typical framework analysis include five steps: data familiarization, framework identification, indexing, charting, and mapping (Ritchie and Spencer 2002). Our interview follows a three-step process modeled on the *framework analysis* methodology:

Data Familiarization and Framework Identification Based on the reengineering challenges and practices we identified from our literature review §2 and the results of open-source failure analysis §5.1. We created a *thematic framework*, including the challenges and practices of bug identification and testing/debugging, and created a draft reengineering workflow as one theme of the reengineering practices.

Interview Design: We designed a semi-structured interview protocol with questions that follow our identified themes of DL reengineering. We conducted two pilot interviews and then revised our framework and interview protocol by clarifying our questions. The final interview protocol includes four main parts: demographic questions, reengineering process workflow, reengineering challenges, and reengineering practices. Table 8 indicates the three main themes of our interview protocol.

We recruited and interviewed 6 leaders in our team and 2 practitioners who are the major contributors for the open-source projects we studied in §5.1. These parts of the interviews were technical and focused, more closely resembling a structured interview than a semi-structured one. During the interview, we provide relevant themes and show the draft of the reengineering workflow in a slide deck.[3]

Analysis: The interview recordings were transcribed by a third-party service[4], and themes were extracted and mapped by the researcher who conducted the interviews. One of the transcripts was in Chinese and a researcher who is a native speaker listened to the recording. We compared those parts of each transcript side-by-side, and noted challenges or practices that were discussed by multiple leaders and extracted illustrative quotes. After all interviews, we recirculated our findings with the team leaders and they agreed with our analysis. By studying the chart and summaries, we were able to understand the larger picture and summarize common challenges.

6 Results and Analysis

6.1 RQ1: How do defects manifest in Deep Learning reengineering?

Finding 1: *Project types:* User "forks" has the most basic (73%) and reproducibility (79%) defects (Figure 5). *Reporter types:* Most defects (58%) are reported by re-users. Defects reported by replicators are rarely identified (<1%). Almost all reproducibility defects are reported by re-users (Figure 6). *DL stages:* 91% defects are reported during environment, training, and data pipeline. 68% reproducibility defects occur in the training stage (Figure 7).

[3] The final version of the reengineering workflow is shown in Figure 11.

[4] `https://www.rev.com/`

Table 8: Interview protocol addressing RQ5. We answer RQ5 by combining results from four kinds of questions: demographic questions, process workflow, challenges, effective practices. Details of demographic questions can be found in our artifact (§10). The interview is semi-structured. The questions in italics are questions that all participants were asked. The other questions are examples of follow-up questions.

Themes	Questions
Process	**Q1:** *Can you talk me through the process that your team follows to re-engineer a machine learning model from research paper/existing implementation/another engineer's project?*
	Q2: *Please take a look at our draft workflow. Can you tell me if you think this is an accurate process workflow?*
	Q3: *Would you like to add any back-edges in this diagram?*
	Q4: How does your team update new iterations of your model if it doesn't work for the first time?
Challenges	**Q5:** *Which parts do you think are challenging when re-engineering a model*
	Q6: Can you tell me about an error you found in TRAINING/MODELING/DATA PIPELINE?
	Q7: Can you describe any challenges you met when implementing TRAINING/MODELING/DATA PIPELINE?
	Q8: How do you address these challenges?
	Q9: Have you met any challenges when integrating all components?
	Q10: Can you think about 1-2 changes to the reengineering process that would make this process easier for you?
Practices	**Q11:** *How does your team work together to make the process more effective?*
	Q12: How do you decide an existing implementation is trustworthy?
	Q13: What do you find is helpful/problematic in a DL research paper?
	Q14: What do you find is helpful/problematic in the documentation of DL models?
	Q15: Are there existing tools (or other technologies) you found valuable/problematic for re-engineering?
	Q16: How do you determine the acceptable trade-off between the performance of the model (accuracy/speed) and the cost of your team (time/money, etc.)?

To understand the characteristics of DL model reengineering, we analyzed the distribution of defect manifestations in terms of reporter types and DL stages.

Project types: Most defects are basic defects (73%) located in user "forks". Most reproducibility defects (79%, 30/38) are identified in user "forks". Most of the basic defects in user "forks" are due to the misunderstanding of the implementation,

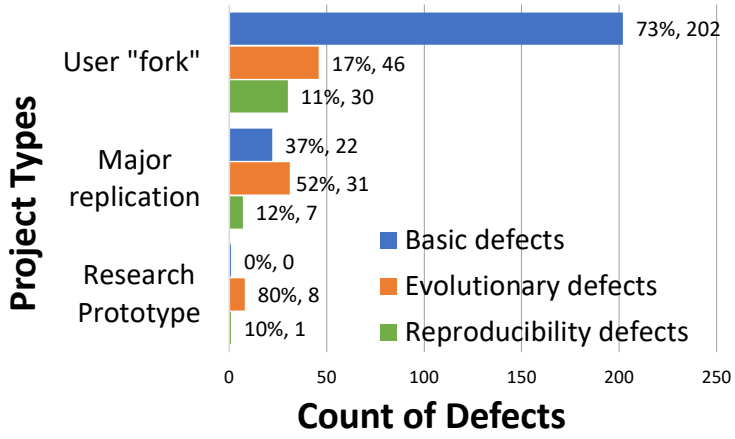

Fig. 5: Manifestation in different project types. Most defects (80%, 278/348) are identified in user "forks", among which 73% (202/278) are basic defects. In this and subsequent figures, the indicated percentages are calculated relative to the corresponding type.

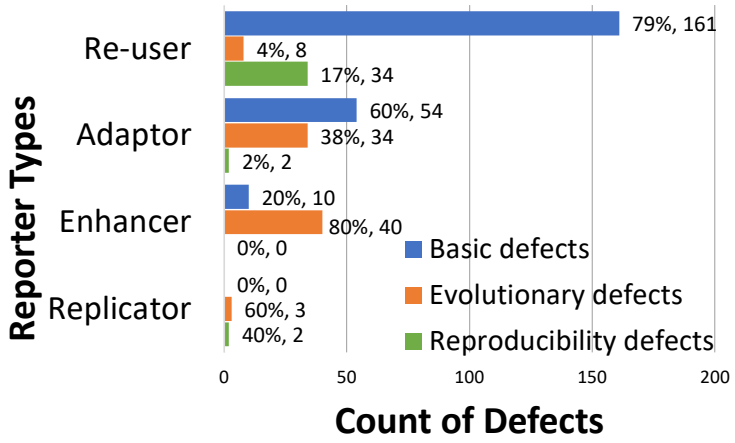

Fig. 6: Manifestation vs. Reporter types. 58% (203/348) of the defects are reported by re-users. Less than 1% are reported by replicators. Almost all reproducibility defects (89%, 34/38) are reported by re-users.

miscommunication, or insufficient documentation of the model(s) they are using. In the discussions, we saw the owners of the repositories often tell the re-users to read the documentation, while the re-users would remark that the documentation is confusing. This observation supports the suggestions from Gundersen *et al.* on documentation improvement (Gundersen and Kjensmo 2018) and indicates there is still a need for detailed documentation and tutorials, especially for the reengineering process (Pineau 2022).

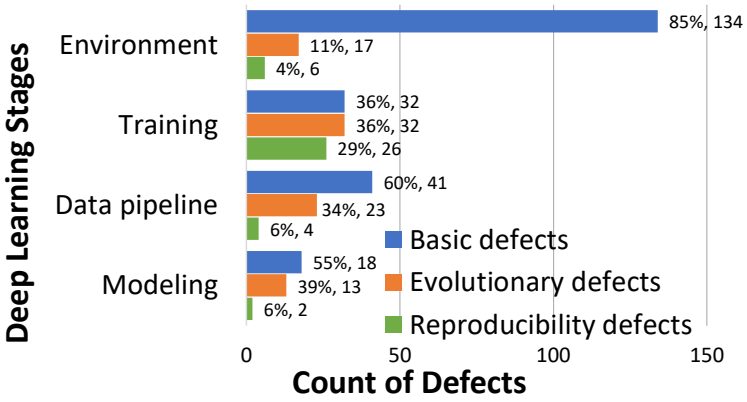

Fig. 7: Manifestation by DL stage. Most *reproducibility* (68%, 26/38) and *evolutionary* (38%, 32/85) defects occur in Training stage. Data pipeline also has many *evolutionary* (34%) defects.

Reporter types: Figure 6 indicates that almost all reproducibility defects are reported by re-users. This finding somewhat follows from our reporter type definitions — an adaptor uses a different dataset, and an enhancer adds new features to the model, so neither is likely to report a reproducibility defects. However, the absence of reproducibility defects from replicators was more surprising. Perhaps replicators are more experienced, and more likely to fix the problem themselves than to open an issue. Alternatively, perhaps there are simply far more re-users in this community.

DL stages: As indicated in Figure 7, 91% (315/348) of defects are reported during environment, training, and data pipeline, 26% (90/348) of defects are reported in the training stage and 20% (68/348) in the data pipeline stage. However, only 9% (33/348) of defects are reported in the modeling stage.

The majority (68%) of reproducibility defects we found are located in the training stage. Differently, 60% of defects in the data pipeline and 55% in the modeling stages are basic defects, which also means that they are easier to identify. This kind of manifestation can be easily found from either the error messages or visualization of the input data. For reproducibility efforts, it is less likely for re-users and replicators to encounter reproducibility defects because they are using the same data and model architectures between one another.

The data indicates that training stage is the most challenging, and data pipeline stage is the second in the reengineering process. Most of the training defects do not result in crashes, but lead to the mismatches between the reimplementation and specification/documented performances. The possible reason is that replicators can refer to the existing replications/prototypes, reuse the data pipeline, or use the same architecture (*e.g.,* backbones) when dealing with similar tasks.

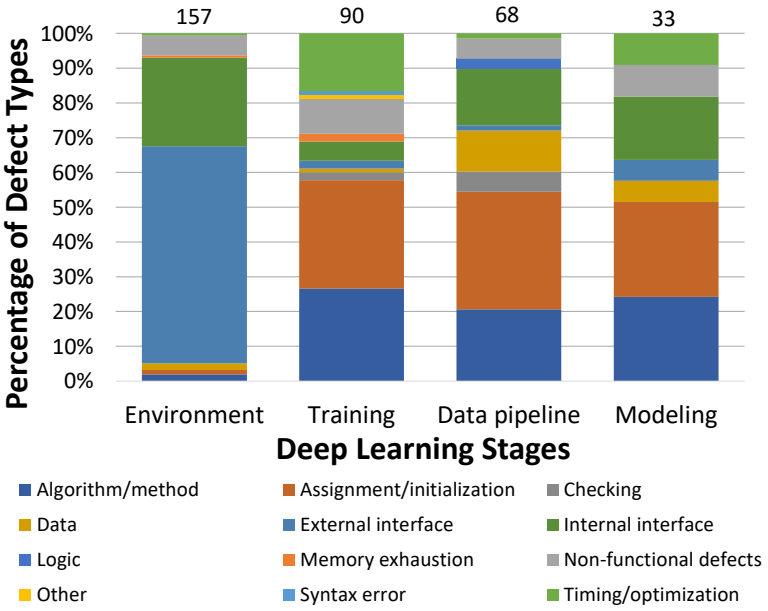

Fig. 8: Defect types by DL stage. Most Environment defects are *interface* defects. The Data Pipeline and Modeling stages have similar distributions oriented towards *assignment/initialization* defects. Training defects are diverse.

6.2 **RQ2: Frequent types of Deep Learning reengineering defects?**

Finding 2: Most Environment defects are *interface* defects (88%). The Data Pipeline and Modeling stages have similar distributions oriented towards *assignment/initialization* defects. Training defects are diverse. (Figure 8)

Here we consider the defect types by DL stage (Figure 8). 88% of the defects in the environment configuration are interface defects. These defects can be divided into external interface defects (62%, 98/157), *i.e.,* the user interface or the interfaces with other systems; and internal interface defects (25%, 40/157), *i.e.,* the interface between different system components, including files and datasets (Seaman et al. 2008). For the external environment, engineers have to setup the DL APIs and hardware correctly before running the model. However, there are often inconsistencies between different DL frameworks and hardware, leading to defects. For the internal environment, engineers need to set up the modules and configure the internal paths, but the documentation or tutorials of the model are sometimes incomprehensible and result in defects.

During training there are relatively more algorithm/method and timing/ optimization defects, compared to other stages. Engineers appear prone to make mistakes in algorithms and methods when adapting the model to their own datasets, or to fail to optimize the model in the training stage.

We observe that assignment/initialization defects account for 34% (23/68) in data pipeline stage and 27% (9/33) in the modeling stage. Internal interface de-

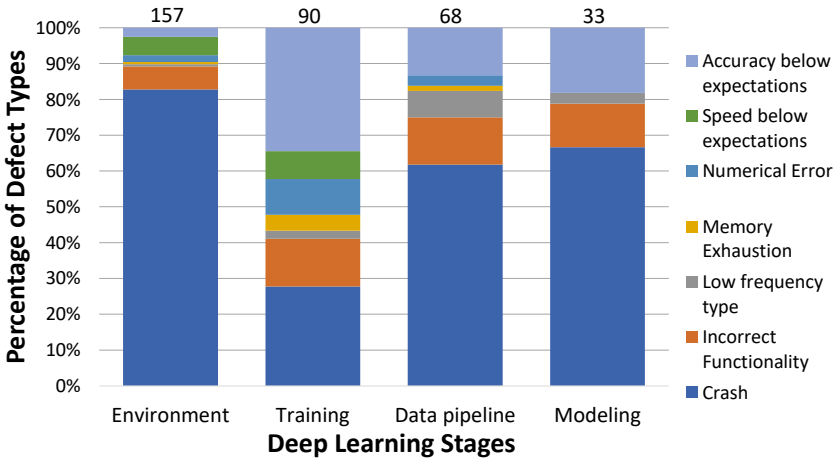

Fig. 9: Defect impacts (Islam et al. 2019; Zhang et al. 2018) by DL stage. Across most stages, the most frequent impact is *crash* (62-82%). In Training, *Accuracy below expectations* accounts for the largest proportion (34%) of defects.

fects also account for 16% (11/68) of the defects in the data pipeline. To fix assignment/initialization defects and internal interface problems, engineers only need to change the values or set up the modules and paths correctly. These are relatively simple defects and simple tool support could help.

6.3 RQ3: What are the impacts of Deep Learning reengineering defects?

Finding 3: Across most stages except training, the most frequent impact is *crash* (62-83%). Training is the most challenging stage where *Accuracy below expectations* accounts for the largest proportion (34%) of defects. (Figure 9)

Figure 9 shows the distribution of defect impacts in different DL stages. Our data shows that *Crash* is a common. 83% (130/157) defects result in crashes in environment, data pipeline, and modeling. Most crashes happen due to incorrect environment configuration, *e.g.,* internal interface, APIs, and hardware.

In contrast, 72% (65/90) of defects in the training stage do not result in crashes (34% lead to *Accuracy below expectations*). The training defects are more likely to result in lower accuracy and incorrect functionality which are harder to identify. Locating the defect could be more time-consuming because the fixers have to train the model for many iterations and compare the accuracy or other metrics to see whether the training works properly. Based on this, we believe that training is the most challenging stage.

Similar to the distribution shown in Figure 7, Figure 9 indicates that reproducibility and evolutionary defects (*e.g.,* lower accuracy and incorrect functionality) can be located in any of the four stages. For evolutionary defects, since the code or data has been changed based on the research prototypes or replications, the defects are more likely to be identified in the changed parts which can be easily found. Nevertheless, for reproducibility defects, especially for the reimplementation built from scratch, it is hard to debug.

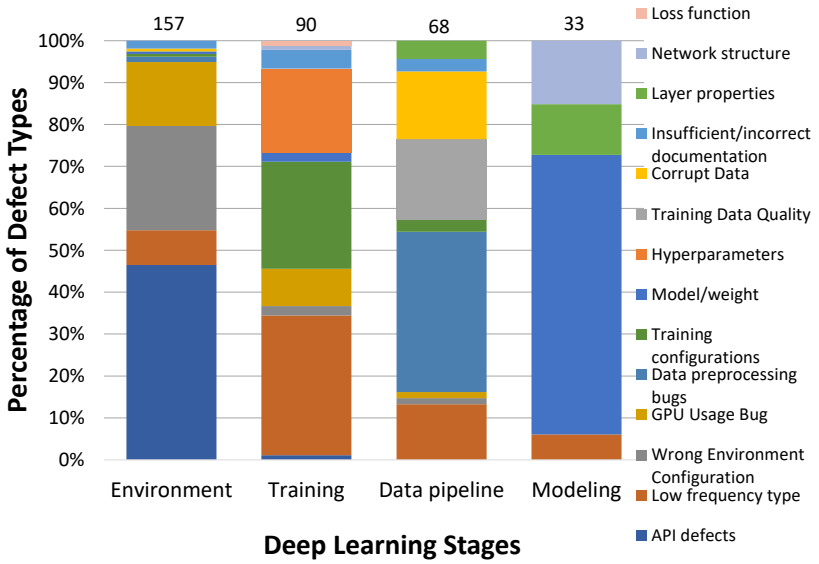

Fig. 10: Root causes by DL stage. In the Data Pipeline, *data preprocessing*, *corrupt data* and *data quality* predominate. Most Modeling defects are due to *model/weight operations* and *network structure*. Training defects have diverse causes.

6.4 **RQ4: What are root causes of Deep Learning reengineering defects?**

Finding 4: Most Environment defects are caused by API defects (46%, 73/157). In the Data Pipeline, *data preprocessing* defects predominate (38%, 26/68). Most Modeling defects are due to *model/weight operations* (67%, 22/33). Training defects have diverse causes. (Figure 10)

Figure 10 shows the distribution of root causes in different DL stages. To answer RQ4, we analyzed the distributions of defects and present our findings by DL stage.

Environment Figure 10 shows that most of the environment defects are caused by API defects (46%, 73/157), Wrong environment configuration (25%, 39/157), and GPU usage defects (15%, 24/157). Many reusers reported defects due to API changes and mismatches. Insufficient documentation can easily lead to misunderstandings and confusion. The portability of models is another problem, especially for the GPU configuration.

Data pipeline Figure 10 indicates three main root causes in the data pipeline: data preprocessing (38%, 26/68), training data quality (19%, 13/68), and corrupt data (16%, 11/68). Engineers are likely to have troubles in data processing and data quality. These defect types are especially frequent for adaptors who are using their own datasets. Datasets vary in format and quality compared to the benchmark CV datasets (*e.g.,* ImageNet (Krizhevsky et al. 2012), COCO (Lin et al. 2014)). Therefore, before adaptors feed the data into the model, they have to first address the dataset format, shape, annotations, and labels. Moreover, customized

datasets are less likely to have enough data to ensure a comparable level of accuracy, so it is necessary to use data augmentation in their data pipeline. However, as we observed, some engineers did not realize the significance of data augmentation and data quality in their reengineering tasks which lead to the lower accuracy.

Modeling The main root causes in the modeling stage are *Model/weight* (67%, 22/33), layer properties (12%, 4/33), and network structure (9%, 5/33). This cause represents the incorrect initialization, loading, or saving of models and weights. We observed that some reengineering work moves from one DL framework to another. Though some tools exist for the interoperability of models between DL frameworks (Liu et al. 2020), we did not see them in use, and engineers are still having troubles in the modeling.

Training There are multiple defect types contributing to training defects. The top two are training configurations and hyper-parameters. Training configurations include different training algorithms (*e.g.,* non-maximum suppression, anchor processing) and some specific parameters which are used to configure the training, but different from the hyper parameters. Putting together with hyper parameter tuning, these two causes are the pitfalls in the training stage. When engineers have different environment configurations or adapt the model to their own datasets, it is necessary to modify the training algorithms and tune the hyper parameters in a proper way so that the training results match their expectations.

6.5 RQ5: Challenges and practices in Deep Learning reengineering?

Finding 5: *Challenges*: The three main challenges in the DL reengineering process are model operationalization, portability of DL operations, and performance debugging. *Practices*: (Interface): Interface can help unify and automate model testing, especially testing of component integration. (Testing): Validation was a common emphasis. Team employed complementary testing techniques (*i.e.,* unit, differential, and visual testing). (Debugging): Comparing evaluation metrics after just 25% of training is a shortcut.

Our failure analysis shed some light on CV reengineering challenges, but little on the guidelines for ML reengineering requested in prior works (Rahman et al. 2019; Devanbu et al. 2020). In this section, we describe those aspects based on the experiences of our CV reengineering team. First, we describe three main reengineering challenges we found in our reengineering team. Then, in §6.5.2 we describe three practices we identified from interview study to mitigate those challenges, including interface, testing, and debugging.

6.5.1 Reengineering Challenges

We interviewed 2 open-source practitioners and 6 leaders of the team (§5.2). Our analysis identified three reengineering challenges within the workflow.

Challenge 1: Model Operationalization

> ***Practitioner 1****: "The only way to validate that your model is correct is to...train it, but the pitfall is that many people only write the code for the model without any training code...That, in my opinion, is the biggest pitfall."*

> ***Leader 2****: "So one challenge...digging through their code and figuring out what is being used and what isn't being used."*
>
> ***Leader 3****: "And in the paper, they...give a pseudo code for just iterating through all..., which they mentioned in the paper that is really inefficient. And they do mention that they have an efficient version, but they never explain how they do it [in the paper]."*
>
> ***Leader 5****: "A lot of the work is figuring out what they did from just the paper and the implementation, which is not exactly clear what they're doing."*

There may be multiple implementations or configurations of the target model. The leaders found it hard to distinguish between them, to identify the reported algorithm(s), and to evaluate their trustworthiness. First, some research prototypes are not officially published. To correctly replicate the model, it is hard to identify which implementation they could refer to. Moreover, the model may be different from the one in the paper. Leader 3 reported that the research prototype used a different but more efficient algorithm without any explanation. Even for the original prototypes, the leaders reported that the prototype's model or training configuration may differ from the documentation (*e.g.,* the research paper). These aspects made it hard for the reengineering team to understand which concepts to implement and which configuration to follow.

Challenge 2: Portability of DL Operations

> ***Practitioner 2****: "The biggest problem that we encounter now is that some methods will depend on certain versions of the software...When you modify the version, it will not be reproducible...For example, the early TensorFlow and later TensorFlow have different back propagation when calculation gradient. Later updated versions will have different results even when running with the same code...Running our current code with CUDA 11.2 and PyTorch 11. 9, the program can be executable, but the training will have issues....The inconsistency between PyTorch and Numpy versions could make the results the same every time. "*
>
> ***Leader 3****: "There was basically [a] one-to-one [mapping of] functions from DL_PLATFORM_1 [to our implementation in] DL_PLATFORM_2. But halfway through...we realized we needed to make it TPU friendly ...had to figure out a way to redesign...to work on TPU since the MODEL_COMPONENTS in DL_PLATFORM_1 were all dynamically shaped."*
>
> ***Leader 2****: "They don't talk about TPUs at all...If you're in a GPU environment, it's a lot easier also, but in order to convert it to TPU, we had to put some design strategies into place and test different possible prototypes."*
>
> ***Leader 6****: "The most challenging part for us is the data pipeline...a lot of the data manipulation...in the original C implementation is...hard [in] Python."*

Though engineers can refer to existing implementations, the conversion of operations between different DL frameworks is still challenging (Liu et al. 2020).

The leaders described four kinds of problems. (1) Different DL frameworks may use different names or signatures for the same behavior. (2) The "same" behavior may vary between DL frameworks, *e.g.,* the implementation of a "standard" optimizer.[5] (3) Some APIs are not supported on certain hardware (*e.g.,* TPU), and the behavior must be composed from building blocks. (4) Out of memory defects could happen in some model architectures when using certain hardware (*e.g.,* TPU). We opened the issue in a major DL framework and the maintainers are investigating.[6]

Challenge 3: Performance Debugging

> ***Practitioner 1****: "If your data format is complicated then the code around it would also be complicated and it would eventually lead to some hard to detect bugs."*

> ***Leader 1****: "You can take advantage of...differential testing but then someone shows out a probabilistic process to this...it becomes really difficult because the testing is dependent upon a random process, a random generation of information."*

> ***Leader 4****: "Sometimes it gets difficult when the existing implementation... [doesn't] have the implementation for the baseline. So we have to figure out by ourselves. Paper also doesn't have that much information about the baseline...if you're not able to even get the baseline, then going to the next part is hard..."*

The last main challenge we observed is matching the performance metrics of existing implementations, especially the original research prototype. These metrics include inference behavior (*e.g.,* accuracy, memory consumption) and training behavior (*e.g.,* time to train). Multiple leaders stated that performance debugging is difficult. Even after replicating the model, performance still varies due to hardware, hyper-parameters, and algorithmic stochasticity. Compared to Amershi *et al.'s* general ML workflow (Amershi et al. 2019a), during reengineering we find that engineers focus less on neural network design, and more on model analysis, operation conversions, and testing.

6.5.2 Reengineering Practices

We also summarized three reengineering practices based on our interview analysis:

Practice 1: Starting with the Interface

> ***Practitioner 2****: "We will define the API for the [our own] interface first. In such case, if something goes wrong during the component integration, we will easily localize the defect."*

> ***Leader 4****: "If you're using existing implementations, you should take time to familiarize with the existing things [code base] that we already have."*

[5] The reengineering team identified and disclosed three examples of this behavior to the owners of the DL framework they used. The documentation of a major DL framework was improved based on our reports.

[6] See `https://github.com/tensorflow/models/issues/10528`.

Leader 6*: "The DL_FRAMEWORK interface, for the MODEL_ZOO specifically, I think that's very important."*

Interface can help unify and automate model testing. Typically, an existing code base has a unified structure which includes the basic trainer and data loader of a model. The interface is essential if the reengineering team is required to use existing code base or model zoos, such as Banna *et al.* describe for the TensorFlow Model Garden (Banna et al. 2021). *Practitioner 2* indicated that implementing interface APIs first can help a lot on bug localization during component integration. Their team also combine agile method (Cohen et al. 2004) with the use of interface APIs which makes the team collaboration more effective. Our reengineering team shares similar experience by implementing interface API first. *Leader 4* indicated that team members have to get familiar with the code base and relevant interface APIs first before the real model implementation.

Practice 2: Testing

Leader 4*: "Unit testing was eas[ier]....because if you're just trying to check what you're writing, it is easier than always trying to differentiate your test and match it with some other model."*

Leader 1*: "You can load the...weights from the original paper. [It] might be a little bit difficult to do but you could do that [and] test the entire model by...comparing the outputs of each layer."'*

Based on the interviews and notes from weekly team meetings, we found that complementary testing approaches were helpful. We describe the team's three test methods: unit testing, differential testing, and visual testing.

Unit testing can ensure that each component works as intended. For the modeling stage, a pass-through test for tensor shapes should be done first, to check whether the output shape of each layer matches the input shape of the next. They also do a gradient test which calculates the model's gradient to ensure the output is differentiable.

Differential testing compares two supposedly-identical systems by checking whether they are input-output compatible (Mckeeman 1998). This is most applicable in the Modeling stage: the original weights/checkpoint can be loaded into the network, and the behavior of original and reengineered model can be compared. This technique isolates assessment of the model architecture from the stochastic training component (Pham et al. 2020).

Differential testing is also applicable in Environment and Training stages to ensure the consistency of each function. For example, when testing the loss function in the training stage, they generate random tensors, feed them into both implementations, and compare the outputs. The outputs should match, up to a rounding error.

Leader 4*: "Both...implementations, even though we are doing the same model, the way they organized are very different....Differential testing can get difficult."*

Leader 5*: "In the data pipeline, one of the big challenges is that the data pipeline is not deterministic, it's random. So it's hard to make test cases for it nor to see if it matches the digital limitation, because you can't do differential testing because it's random."*

However, differential testing does not apply to all DL components. For example, the data pipeline has complex probabilistic outputs, where they found it simpler to do *visual testing*. For data pipeline, each preprocessing operation can be tested by passing a sample image through the original pipeline and the reengineering pipeline. They visually inspect the output of two pipelines to identify noticeable differences. This approach is applicable in CV, but may not generalize to other DL applications.

Unit, differential, and visual testing are complementary. At coarser code granularities, automated differential tests are effective; at a fine-enough level of granularity the original and replicated component may not be comparable, and unit tests are necessary. The characteristics of a data pipeline are difficult to measure automatically, and visual testing helps here.

Practice 3: Performance Debugging

> ***Leader 1****: "I think the biggest thing is logging the accuracy after every single change...then you know what changes are hurting your accuracy and which changes are helping, which is immensely helpful."*

> ***Practitioner 1****: "If your data format is complicated, then the code around it would also be complicated and it would, eventually, lead to some hard to detect bugs."*

The most common way to detect reproducibility and evolutionary defects is to compare model evaluation metrics. This approach can be costly due to the resources required for training in a code-train-debug-fix cycle. However, as we noted from the team meetings, 70% of the final results may be achieved within 10-25% of training. They thus check whether evaluation metrics and trends are comparable after 25% of training, shortening the debugging cycle.

Beyond this approach, they agree with prior works that record-keeping help model training and management (Schelter et al. 2017; Vartak et al. 2016).

7 Discussion and Implications

7.1 Triangulating our findings into a reengineering workflow

The two prongs of our case study identified similar challenges in *model implementation and testing*, especially *performance debugging*, which we show below in Figure 11. Our failure analysis found that most reengineering defects are caused by DL APIs and hardware configuration (see Figure 7 and Figure 8), and we identified the similar challenge of *portability* in our CV reengineering team. Moreover, in the failure analysis we suggested that the training stage would be the most challenging because of the diversity of failure modes (cf. Figure 8, Figure 9, and Figure 10). This stage was also highlighted by our CV reengineering team as the *performance debugging* challenge. In addition, as we noticed from our failure analysis, engineers applied multiple testing strategies, among which the most common ones are unit testing, differential testing, and visual testing. To support the reengineering works, we provide detailed strategies and insights based on our reengineering experience.

However, our two data sources were not fully in agreement. The interviews with reengineering team leaders identified the *model operationalization* challenge, while the failure analysis data provided no direct evidence of this challenge. We believe that when engineers encounter model operationalization defects, they may not

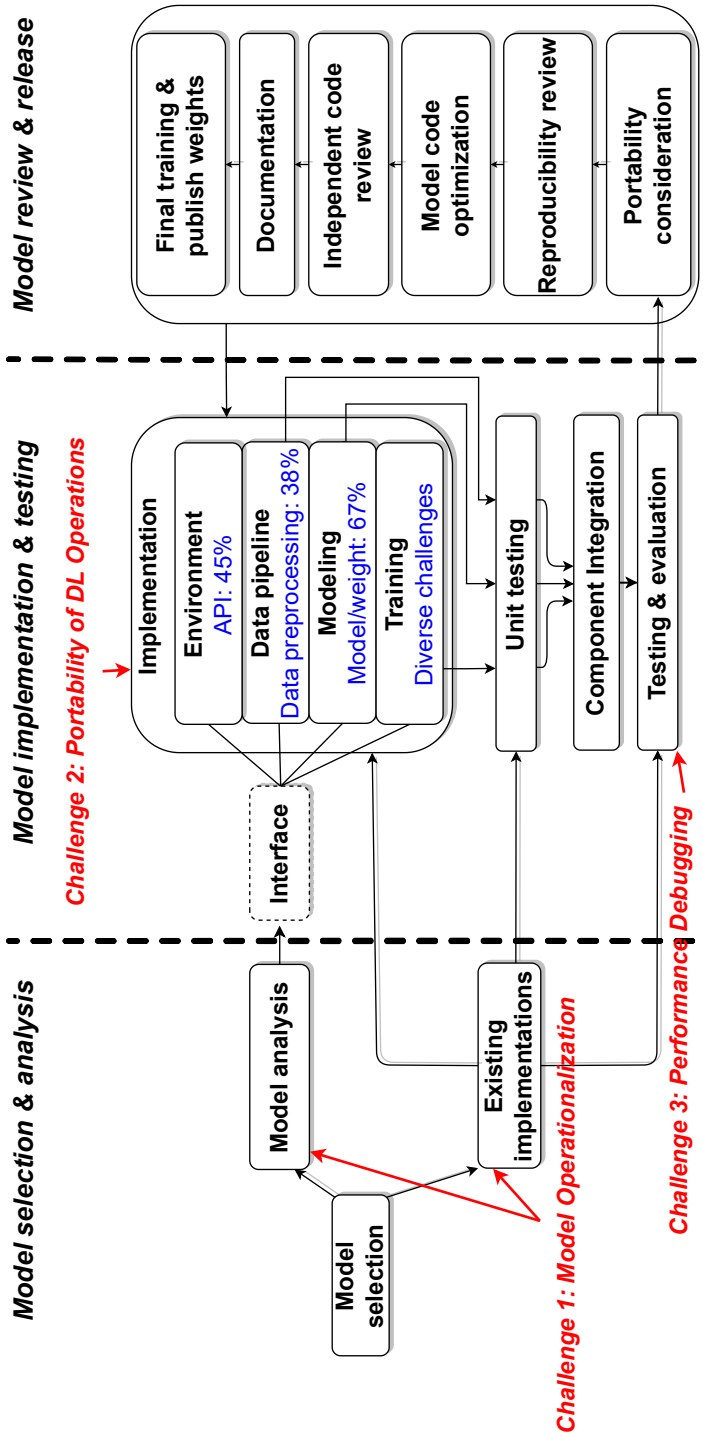

Fig. 11: Reengineering workflow, divided into three main stages. Forward edges denote the reengineering workflow. Dotted line denotes optional (interface). Back edges denote points where errors are often identified. Red arrows indicate the main location of each challenge. Blue text denotes the main root causes of failures in each DL stage (see Figure 10).

open an issue about it, or may be unable to identify the defect until testing. Thus the *operationalization* challenge may manifest as *performance debugging* defects.

To illustrate DL Reengineering challenges and practices, we developed a reengineering workflow based on our study on open-source projects and reengineering team. One leader of our reengineering team described the reengineering workflow their sub-team followed. The other leaders revised it until agreement. Then we revised it based on our sub-team approaches and observations of open-source bug reports.

The resulting workflow (see Figure 11 below) has three main stages: (1) Model selection and analysis; (2) Implementation and testing; and (3) Review and release. In the first stage, the team identifies a candidate model for the desired task (*e.g.,* low-power object recognition), and determines its suitability on the target DL framework and hardware. Existing implementations are examined as points of comparison. In the second stage, the components of the system are implemented, integrated, and evaluated, typically with different personnel working on each component. At the end of this stage, the model performance matches the paper to a tolerance of 1-3%. In the third stage, the model is tailored for different platforms, *e.g.,* servers or mobile devices, and appropriate checkpoint weights are published. We included our checklists for these tasks in our artifact. Note that this workflow is fairly linear, without the extensive iteration suggested by Amershi *et al.* for ML model development (Amershi et al. 2019a). As this workflow is focused on reengineering, the system requirements and many design elements were well understood; less iteration is necessary.

7.2 Comparison to Prior Works

Our general findings on the CV reengineering process match some results from prior works. For example, like prior bug studies on DL engineering (Islam et al. 2019; Humbatova et al. 2020), we observed a large percentage of API defects (21%, 74/348) within the CV reengineering process. In line with the results from Zhang *et al.* (Zhang et al. 2018), we also found basic defects are the most common manifestation.

However, we observed notably different proportions of defects, *e.g.,* by stage and by cause. We found a higher proportion of hyper-parameter tuning defects (5%, 18/348) in the DL reengineering process compared to the results of Islam *et al.*, who reported a proportion of less than 1% (Islam et al. 2019). Additionally, the results presented by Humbatova *et al.* shows that 95% of survey participants had trouble with training data (Humbatova et al. 2020). However, in our result, training data quality only accounts for 19% (13/68) of defects in data pipeline! This difference may arise from context: 58% (203/348) of the defects we analyzed were reported by re-users using benchmark datasets.

Qualitatively, our in-depth study of a CV reengineering team identified more detailed challenges in DL reengineering. For model configuration, we observed challenges in distinguishing models, identifying reported algorithm(s), and evaluating of trustworthiness. These were not mentioned in prior works (Islam et al. 2020). We also refined portability challenges into three different types: different names/signatures for the same behavior, inconsistent and undocumented behaviors, and different behavior on certain hardware. The latter two were not identified before (Zhang et al. 2019). Finally, we noted the importance — and difficulty — of performance debugging in DL reengineering.

We also compare the DL reengineering to other reengineering works in the software engineering literature (Singh et al. 2019; Bhavsar et al. 2020). We propose that the goal and focus are two main differences: First, the goal of many reengineering projects is to improve software performance or optimize the process, while the main goal we saw in DL reengineering was to support further software reuse and customization. Second, other reengineering studies focus on the maintenance/process aspect, while in our study we saw that the DL reengineering activities focus on the implementation/evolution aspect. One possible causal factor here is that the DL reengineering activities we observed were building on research products, while general software reengineering is revisiting the output from engineering teams.

7.3 Implications

Our empirical data motivates many directions for further research:

Empirical Studies Our case study identified several gaps in our empirical understanding of DL reengineering. *First*, to fully understand reengineering problems, it would be useful to investigate more deeply how experts solve specific problems. Our data indicated the kinds of problems being solved during CV reengineering, but we did not explore the problem solution strategies nor the design rationale. For example, reengineering defects include indications of expert strategies for choosing loss functions[7] and tuning hyper-parameters[8]. *Second*, our findings motivate further interviews and surveys for DL engineers, *e.g.,* to understand the reengineering process workflow (Figure 11) and to evaluate the efficiency benefits of our proposed strategies (among others (Serban et al. 2020)). *Third*, the difficulties we measured in DL model reengineering imply that reengineering is hard. Monte *et al.* showed that collections of pre-trained models (*e.g.,* ONNX model zoo (ONN 2019a) and Keras applications (Keras 2022)) may not be entirely consistent with the models they claim to replicate (Montes et al. 2022). Prior work also indicated that discrepancies do exist in the DL model supply chain (Jiang et al. 2022, 2023). These defects could result in challenges of *model operationalization* and *portability* (§6.5.1) and therefore impact the downstream DL pipelines (Xin et al. 2021; Gopalakrishna et al. 2022; Nikitin et al. 2022). There is no comprehensive study on how these discrepancies can affect downstream implementations and what are effective ways to identify them. To better support reengineering process and knowledge transfer, we envision future works to discover approaches to improve the transferability and auditability of deep learning systems.

Given the substantial prior research on DL failures, both quantitative and qualitative, we were surprised by the differences between our results and prior work that we described in §7.2. Although we can only conjecture, one possible explanation of these differences is that prior work takes a "product" view in sampling data (cf. §2.2), while our work's sampling approach takes a "process" view guided by the engineering activity. The sampling methods between our work and previous studies vary in what they emphasize. Prior work used keywords to search for issues, pull requests, and Stack Overflow questions (Islam et al. 2019; Zhang et al. 2018; Humbatova et al. 2020; Sun et al. 2017; Shen et al. 2021) which provide them a "product" view of the deep learning failures. In contrast, during the data col-

[7] See `https://github.com/ultralytics/yolov3/issues/2`.

[8] See `https://github.com/junyanz/pytorch-CycleGAN-and-pix2pix/issues/150`.

lection, we identified the engineering activities and process *first*, then categorized and analyzed the defects. If the results of a failure analysis differ based on whether data is sampled by a product or a process perspective, the implications for software failure studies are profound — most prior studies take a product view (Amusuo et al. 2022). This may bear further reflection by the empirical software engineering research community.

DL Software Testing Our results shows the DL-stage-related characteristics of reengineering defects (§6.1–§6.4) and difficulties of debugging (§6.5). We recommend future directions on debugging tools for each DL stage, especially Data Pipeline and Training.

Our analysis shows that most of the defects in data pipeline are due to the incorrect data pre-processing and low data quality. There have been many studies on data management (Kumar et al. 2017), validation for DL datasets (Breck et al. 2019), and practical tools for data format converting (Willemink et al. 2020). However, we did not observe much use of these tools in (open-source) practice, including in the commercially maintained zoos. It is unclear whether the gap is in adopting existing tools or in meeting the real needs of practitioners. In addition, creating large DL datasets is expensive due to high labor and time cost. Data augmentation methods are widely used in recent DL models to solve the related problems (*e.g.,* overfitting, lower accuracy) (Shorten and Khoshgoftaar 2019). Our results (§6.4) show that engineers often have data augmentation and data quality defects. Therefore, we recommend future studies on evaluating the quality of data augmentation. It would be of great help if engineers can test in advance whether the data augmentation is sufficient for training.

Researchers have been working on automated DL testing (Tian et al. 2018; Pei et al. 2017) and fuzzing (Zhang et al. 2021; Guo et al. 2018) tools for lowering the cost of the testing in the training stage. However, there are not many specific testing techniques for DL reengineering (Braiek and Khomh 2020). In our reengineering team (§6.5), we found performance debugging challenging, especially for reproducibility defects. We recommend software researchers explore end-to-end testing tools for reengineering tasks which can take advantage of existing model implementations. For example, new fuzzing technology can be developed to use adversarial inputs to test the correctness of training in the early stage by comparing to the original model which can eventually save lots of computational resources and time. To lower end-to-end costs, improved unit testing methods will also help. We recommend future directions on similar techniques of effective unit/differential testing for each DL stage to catch training bugs earlier, grounded in the stage-related characteristics of DL defects.

Enabling Model Reuse DL Model reuse may mainly happen in the modeling stage. Most modeling defects are due to the operations of pre-trained weights and models (Figure 10). Though there have been tools for the conversion of models between different DL frameworks, notably ONNX (ONN 2019a), converting models remains costly due to the rapidly increasing number of operators (Liu et al. 2020) and data types (ONN 2019b). Moreover, we found that engineers also struggle with data management and training configuration. Thus we recommend further investment in *end-to-end model conversion technology.*

When models cannot be reused automatically, DL reengineers must manually replicate the logic, resulting in a range of defects (Figure 10). Both the open-source

defects and team leaders mentioned mathematical defects, *e.g.,* sign mismatch or numerical instability. We suggest a domain-specific language for loss function mathematics, similar to how regular expressions support string parsing (Michael et al. 2019).

Standardized Practices As observed in the open-source defects, few of these reengineering efforts followed a standard process, causing novices to get confused. We recommend step-by-step guidelines for DL reengineering, starting with Figure 11. Though major companies provide some practices to standardize the DL model documentation, such as model card proposed by Google (Mitchell et al. 2019) and metadata extraction from IBM (Tsay et al. 2022), our analysis (Figure 10) and recent work (Jiang et al. 2023) indicate that existing implementations lack standardized documentation which results in the challenge of model analysis and reuse. Therefore, we recommend engineers develop and adhere to a *standard template*, *e.g.,* stating environment configuration and hyper-parameter tuning. We also envision further studies on automated tools to extract the training scripts, hyperparameters, and evaluation results from open-source projects to generate more standardized documentations.

8 Threats to Validity

Construct Threats In this paper we introduced the concept of *Deep Learning Reengineering*. This concept tailors the idea of software reengineering (Linda et al. 1996; Byrne 1992) to the reengineering tasks pertinent to DL. We used this concept as a criterion by which to select repositories and defects for analysis. Although this concept resembles traditional problems in reuse and porting, we believe reengineering is a useful conceptualization for the activities of software engineers working with DL technologies.

Internal Threats In measuring aspects of this concept, we relied on manual classification of GitHub issue data. Our methodology might bring potential bias in our results. The use of only one experienced rater in failure analysis and one analyzer in the interview study also increases the likelihood of subjective biases influencing the results. To mitigate subjectivity in the GitHub issue analysis, we adapted and applied existing taxonomies. Ambiguity was resolved through instrument calibration and discussion. Acceptable levels of interrater agreement were measured on a sample of the data.

We also agree that there could be some bias in the framework analysis of in the interview study. We considered the bias in our study design. To mitigate the bias in our interview data, we build a draft of reengineering workflow based on the knowledge of ML development workflow (Amershi et al. 2019a). Our observations from the failure analysis also let us tease out similarities and differences in the reengineering context. During the interview, we asked if the subjects have anything to add to our workflow. We also recirculated our findings with some of the interview subjects (team leaders) and they agreed with our analysis (Ritchie et al. 2013).

External Threats Our case study considered DL reengineering in the context of computer vision. Our findings may not generalize to reengineering in other DL applications. Within our case study, our failure analysis examined open-source CV models. Our data may not generalize to commercial practices; to mitigate this, we focused on popular CV models. Lack of theoretical generalization is a drawback of

our case study method; the trade-off is for a deeper look at a single context. We believe findings for CV are worth having, even if they do not generalize to all DL reengineering efforts. CV is a crucial technology with many applications (Wang et al. 2021; Garcia et al. 2020). As a point of comparison, we think that research on web applications contextualized to important frameworks like Rails (Yang et al. 2019) and Node.js (Chang et al. 2019) are worth having, even if not all software is a web application or if some web applications use other technologies.

Within our CV reengineering team study, our team had only two years of (undergraduate) experience in the domain. However, the corporate sponsor provided weekly expert coaching, and the results are now used internally by the sponsor. This provides some evidence that the team produced acceptable engineering results, implying that their process is worth describing in this work.

9 Conclusions

Software engineers are engaged in DL reengineering — getting DL research ideas to work well in their own contexts. Prior work mainly focus on the "product" view of DL systems, while our case study explored the "process" view of characteristics, pitfalls, and practices of DL model reengineering activities. We analyzed 348 CV reengineering defects and reported on a two-year reengineering effort. Our analysis shows that the characteristics of reengineering defects vary by DL stage, and that the environment and training configuration are the most challenging parts. Additionally, we identified the three challenges of DL reengineering: model operationalization, portability of DL operations, and performance debugging. We integrated our findings through a DL reengineering process workflow, annotated with practices, challenges, and frequency of defects. Our results inform future research directions on model reengineering, including tools for testing and reuse.

10 Data Availability (Reproducibility) and Research Ethics

Our artifact is at `https://github.com/Wenxin-Jiang/EMSE-CVReengineering-Artifact`. Human subjects work was approved by institutional IRB.

ACKNOWLEDGEMENT

This work was supported by Google and Cisco and by NSF awards #2107230, #2229703, #2107020, and #2104319.

References

(2019a) ONNX | Home. URL `https://onnx.ai/`

(2019b) Portability between deep learning frameworks – with ONNX. URL `https://blog.codecentric.de/en/2019/08/portability-deep-learning-frameworks-onnx/`

(2020) Managing labels. URL `https://docs.github.com/en/issues/using-labels-and-milestones-to-track-work/managing-labels`

(2020) Papers with Code - ML Reproducibility Challenge 2021 Edition. URL `https://paperswithcode.com/rc2021`

(2021) Being a Computer Vision Engineer in 2021. URL `https://viso.ai/computer-vision/computer-vision-engineer/`

(2021) Machine Learning Operations. URL `https://ml-ops.org/`

Alahmari SS, Goldgof DB, Mouton PR, Hall LO (2020) Challenges for the Repeatability of Deep Learning Models. IEEE Access

AlDanial (2022) cloc. URL `https://github.com/AlDanial/cloc`

Amershi S, Begel A, Bird C, DeLine R, Gall H (2019a) Software Engineering for Machine Learning: A Case Study. In: International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)

Amershi S, Begel A, Bird C, DeLine R, Gall H, Kamar E, Nagappan N, Nushi B, Zimmermann T (2019b) Software Engineering for Machine Learning: A Case Study. In: IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), DOI 10.1109/ICSE-SEIP.2019.00042

Amusuo P, Sharma A, Rao SR, Vincent A, Davis JC (2022) Reflections on software failure analysis. In: ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering — Ideas, Visions, and Reflections track (ESEC/FSE-IVR)

Aranda J, Venolia G (2009) The secret life of bugs: Going past the errors and omissions in software repositories. In: International Conference on Software Engineering (ICSE)

Bahdanau D, Cho KH, Bengio Y (2015) Neural machine translation by jointly learning to align and translate. In: International Conference on Learning Representations (ICLR)

Banna V, Chinnakotla A, Yan Z, Vegesana A, Vivek N, Krishnappa K, Jiang W, Lu YH, Thiruvathukal GK, Davis JC (2021) An experience report on machine learning reproducibility: Guidance for practitioners and TensorFlow model garden contributors. URL `https://arxiv.org/abs/2107.00821`

Berner C, Brockman G, Chan B, Cheung V, Debiak P, Dennison C, Farhi D, Fischer Q, Hashme S, Hesse C, Józefowicz R, Gray S, Olsson C, Pachocki J, Petrov M, Pinto HPdO, Raiman J, Salimans T, Schlatter J, Schneider J, Sidor S, Sutskever I, Tang J, Wolski F, Zhang S (2019) Dota 2 with Large Scale Deep Reinforcement Learning. arXiv preprint arXiv:191206680 URL `https://arxiv.org/abs/1912.06680`

Bhavsar K, Shah V, Gopalan S (2020) Machine learning: a software process reengineering in software development organization. International Journal of Engineering and Advanced Technology 9(2):4492–4500

Bibal A, Frénay B (2016) Interpretability of Machine Learning Models and Representations: an Introduction. In: European Symposium on Artificial Neural Networks

Boehm B, Beck K (2010) The changing nature of software evolution; The inevitability of evolution. In: IEEE Software

Borges H, Valente MT (2018) What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. In: Journal of Systems and Software (JSS), DOI 10.1016/j.jss.2018.09.016

Braiek HB, Khomh F (2020) On testing machine learning programs. Journal of Systems and Software (JSS) 164:110542

Breck E, Cai S, Nielsen E, Salib M, Sculley D (2017) The ML test score: A rubric for ML production readiness and technical debt reduction. In: 2017 IEEE International Conference on Big Data (Big Data), pp 1123–1132, DOI 10.1109/BigData.2017.8258038

Breck E, Polyzotis N, Roy S, Whang S, Zinkevich M (2019) Data Validation for Machine Learning. In: the Conference on Machine Learning and Systems (MLSys)

Byrne E (1992) A conceptual foundation for software re-engineering. In: Conference on Software Maintenance

Chang X, Dou W, Gao Y, Wang J, Wei J, Huang T (2019) Detecting atomicity violations for event-driven node. js applications. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, pp 631–642

Chen B, Wen M, Shi Y, Lin D, Rajbahadur GK, Ming Z, Jiang (2022a) Towards Training Reproducible Deep Learning Models. In: International Conference on Software Engineering (ICSE), pp 2202–2214, DOI 10.1145/3510003.3510163

Chen J, Liang Y, Shen Q, Jiang J (2022b) Toward Understanding Deep Learning Framework Bugs. URL http://arxiv.org/abs/2203.04026

Cohen D, Lindvall M, Costa P (2004) An introduction to agile methods. Advanced Computing 62(03):1–66

Cohen J (1960) A coefficient of agreement for nominal scales. Educational and psychological measurement 20(1):37–46

Devanbu P, Dwyer M, Elbaum S, Lowry M, Moran K, Poshyvanyk D, Ray B, Singh R, Zhang X (2020) Deep Learning & Software Engineering: State of Research and Future Directions. arXiv URL https://arxiv.org/abs/2009.08525

Ding Z, Reddy A, Joshi A (2021) Reproducibility. URL https://blog.ml.cmu.edu/2020/08/31/5-reproducibility/

Doshi-Velez F, Kim B (2017) Towards A Rigorous Science of Interpretable Machine Learning. URL https://arxiv.org/abs/1702.08608

Eghbali A, Pradel M (2020) No strings attached: an empirical study of string-related software bugs. In: International Conference on Automated Software Engineering (ASE)

Fitzgerald B (2006) The transformation of open source software

Forsyth DA, Ponce J (2002) Computer vision: a modern approach. prentice hall professional technical reference

Garcia J, Feng Y, Shen J, Almanee S, Xia Y, Chen QA (2020) A comprehensive study of autonomous vehicle bugs. In: International Conference on Software Engineering (ICSE), URL https://dl.acm.org/doi/10.1145/3377811.3380397

Goel A, Tung C, Lu YH, Thiruvathukal GK (2020) A Survey of Methods for Low-Power Deep Learning and Computer Vision. In: IEEE World Forum on Internet of Things (WF-IoT)

Google (2022) Tensorflow model garden. URL https://github.com/tensorflow/models

Gopalakrishna NK, Anandayuvaraj D, Detti A, Bland FL, Rahaman S, Davis JC (2022) "If security is required": Engineering and Security Practices for Machine Learning-based IoT Devices. In: International Workshop on Software Engineering Research & Practices for the Internet of Things (SERP4IoT)

Goyal P, Dollár P, Girshick R, Noordhuis P, Wesolowski L, Kyrola A, Tulloch A, Jia Y, He K (2018) Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. URL https://arxiv.org/abs/1706.02677

Gundersen OE, Kjensmo S (2018) State of the art: Reproducibility in artificial intelligence. AAAI Conference on Artificial Intelligence (AAAI)

Gundersen OE, Gil Y, Aha DW (2018) On reproducible AI: Towards reproducible research, open science, and digital scholarship in AI publications. AI Magazine

Guo J, Jiang Y, Zhao Y, Chen Q, Sun J (2018) DLFuzz: Differential Fuzzing Testing of Deep Learning Systems. In: European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)

Humbatova N, Jahangirova G, Bavota G, Riccio V, Stocco A, Tonella P (2020) Taxonomy of real faults in deep learning systems. In: International Conference on Software Engineering (ICSE)

Hutson M (2018) Artificial intelligence faces reproducibility crisis. American Association for the Advancement of Science 359(6377):725–726, DOI 10.1126/science.359.6377.725

Islam MJ, Nguyen G, Pan R, Rajan H (2019) A comprehensive study on deep learning bug characteristics. In: European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)

Islam MJ, Pan R, Nguyen G, Rajan H (2020) Repairing deep neural networks: fix patterns and challenges. In: International Conference on Software Engineering (ICSE)

Jarzabek S (1993) Software reengineering for reusability. In: International Computer Software and Applications Conference (COMPSAC)

Jiang W, Synovic N, Sethi R, Indarapu A, Hyatt M, Schorlemmer TR, Thiruvathukal GK, Davis JC (2022) An empirical study of artifacts and security risks in the pre-trained model supply chain. In: ACM Workshop on Software Supply Chain Offensive Research and Ecosystem Defenses (SCORED'22), p 105–114, DOI 10.1145/3560835.3564547, URL https://doi.org/10.1145/3560835.3564547

Jiang W, Synovic N, Hyatt M, Schorlemmer TR, Sethi R, Lu YH, Thiruvathukal GK, Davis JC (2023) An empirical study of pre-trained model reuse in the hugging face deep learning model registry. In: IEEE/ACM 45th International Conference on Software Engineering (ICSE'23)

Jing YK (2021) Model Zoo - Deep learning code and pretrained models. URL https://modelzoo.co/

Johnson RB, Onwuegbuzie AJ (2004) Mixed Methods Research: A Research Paradigm Whose Time Has Come. Educational Researcher

Keras (2022) Keras applications. URL https://keras.io/api/applications/

Kim J, Li J (2020) Introducing the model garden for tensorflow 2. URL https://blog.tensorflow.org/2020/03/introducing-model-garden-for-tensorflow-2.html

Krizhevsky A, Sutskever I, Hinton GE (2012) ImageNet Classification with Deep Convolutional Neural Networks. In: Advances in Neural Information Processing Systems (NeurIPS), vol 6, pp 84–90

Kumar A, Boehm M, Yang J (2017) Data Management in Machine Learning: Challenges, Techniques, and Systems. In: International Conference on Management of Data

Li R, Jiao Q, Cao W, Wong HS, Wu S (2020) Model Adaptation: Unsupervised Domain Adaptation without Source Data. Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition pp 9638–9647, DOI 10.1109/CVPR42600.2020.00966

Lin TY, Maire M, Belongie S, et al. (2014) Microsoft COCO: Common Objects in Context. In: European Conference on Computer Vision (ECCV)

Linda D, Rosenberg H, Hyatt LE (1996) Software Re-engineering. Software Assurance Technology Center

Liu Y, Xu C, Cheung SC (2014) Characterizing and detecting performance bugs for smartphone applications. In: Proceedings of the 36th International Conference on Software Engineering, ACM, Hyderabad India, pp 1013–1024, DOI 10.1145/

2568225.2568229, URL https://dl.acm.org/doi/10.1145/2568225.2568229
Liu Y, Chen C, Zhang R, Qin T, Ji X, Lin H, Yang M (2020) Enhancing the interoperability between deep learning frameworks by model conversion. In: European Software Engineering Conference/Foundations of Software Engineering (ESEC/FSE)
Lorenzoni G, Alencar P, Nascimento N, Cowan D (2021) Machine Learning Model Development from a Software Engineering Perspective: A Systematic Literature Review. arXiv URL https://arxiv.org/abs/2102.07574
McHugh ML (2012) Interrater reliability: the kappa statistic. Biochemia medica 22(3):276–282
Mckeeman WM (1998) Differential Testing for Software. Digital Technical Journal
Meta (2022) Torchvision. URL https://github.com/pytorch/vision
Michael LG, Donohue J, Davis JC, Lee D, Servant F (2019) Regexes are hard: Decision-making, difficulties, and risks in programming regular expressions. Proceedings - 2019 34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019 pp 415–426, DOI 10.1109/ASE.2019.00047, iSBN: 9781728125084
Mitchell M, Wu S, Zaldivar A, Barnes P, Vasserman L, Hutchinson B, Spitzer E, Raji ID, Gebru T (2019) Model Cards for Model Reporting. In: Proceedings of the Conference on Fairness, Accountability, and Transparency, ACM, Atlanta GA USA, pp 220–229, DOI 10.1145/3287560.3287596, URL https://dl.acm.org/doi/10.1145/3287560.3287596
Montes D, Peerapatanapokin P, Schultz J, Guo C, Jiang W, Davis JC (2022) Discrepancies among pre-trained deep neural networks: a new threat to model zoo reliability. In: European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE-IVR track).
Nahar N, Zhou S, Lewis G, Kästner C (2022) Collaboration Challenges in Building ML-Enabled Systems: Communication, Documentation, Engineering, and Process. In: International Conference on Software Engineering (ICSE)
Nikanjam A, Khomh F (2021) Design Smells in Deep Learning Programs: An Empirical Study. In: IEEE International Conference on Software Maintenance and Evolution (ICSME)
Nikitin NO, Vychuzhanin P, Sarafanov M, Polonskaia IS, Revin I, Barabanova IV, Maximov G, Kalyuzhnaya AV, Boukhanovsky A (2022) Automated evolutionary approach for the design of composite machine learning pipelines. Future Generation Computer Systems
O'Connor R (2023) Pytorch vs tensorflow in 2023. URL https://www.assemblyai.com/blog/pytorch-vs-tensorflow-in-2023/
Pei K, Cao Y, Yang J, Jana S (2017) DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In: Symposium on Operating Systems Principles (SOSP)
Perry D, Sim S, Easterbrook S (2004) Case studies for software engineers. In: International Conference on Software Engineering (ICSE)
Pham HV, Qian S, Wang J, Lutellier T, Rosenthal J, Tan L, Yu Y, Nagappan N (2020) Problems and Opportunities in Training Deep Learning Software Systems: An Analysis of Variance. In: International Conference on Automated Software Engineering (ASE), DOI 10.1145/3324884.3416545
Pineau J (2022) How the AI community can get serious about reproducibility. URL https://ai.facebook.com/blog/

how-the-ai-community-can-get-serious-about-reproducibility/
Pineau J, Vincent-Lamarre P, Sinha K, Lariviere V, Beygelzimer A (2020) Improving Reproducibility in Machine Learning Research. Journal of Machine Learning Research
Rahman S, River E, Khomh F, Guhneuc YG, Lehnert B (2019) Machine learning software engineering in practice: An industrial case study. arXiv preprint DOI 10.48550/arXiv.1906.07154
Ralph P, Ali Nb, Baltes S, Bianculli D, Diaz J, Dittrich Y, Ernst N, Felderer M, Feldt R, Filieri A, de França BBN, Furia CA, Gay G, Gold N, Graziotin D, He P, Hoda R, Juristo N, Kitchenham B, Lenarduzzi V, Martínez J, Melegati J, Mendez D, Menzies T, Molleri J, Pfahl D, Robbes R, Russo D, Saarimäki N, Sarro F, Taibi D, Siegmund J, Spinellis D, Staron M, Stol K, Storey MA, Taibi D, Tamburri D, Torchiano M, Treude C, Turhan B, Wang X, Vegas S (2021) Empirical Standards for Software Engineering Research. arXiv URL https://arxiv.org/abs/2010.03525
Redmon J, Divvala S, Girshick R, Farhadi A (2016) You Only Look Once: Unified, Real-Time Object Detection. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR)
Ren S, He K, Girshick R, Sun J (2017) Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. IEEE Transactions on Pattern Analysis and Machine Intelligence (TPAMI)
Ritchie J, Spencer L (2002) Qualitative data analysis for applied policy research. In: Analyzing qualitative data, Routledge, pp 187–208
Ritchie J, Lewis J, Nicholls CM, Ormston R, et al. (2013) Qualitative research practice: A guide for social science students and researchers. sage
Runeson P, Höst M (2009) Guidelines for conducting and reporting case study research in software engineering. Empirical Software Engineering (EMSE)
Schelter S, Boese JH, Kirschnick J, Klein T, Seufert S (2017) Automatically tracking metadata and provenance of machine learning experiments. In: Machine Learning Systems Workshop at NIPS
Schelter S, Biessmann F, Januschowski T, Salinas D, Seufert S, Szarvas G (2018) On Challenges in Machine Learning Model Management. Bulletin of the IEEE Computer Society Technical Committee on Data Engineering
Schmidhuber J (2015) Deep learning in neural networks: An overview. Neural Networks
Sculley D, Holt G, Golovin D, Davydov E, Phillips T, Ebner D, Chaudhary V, Young M (2014) Machine Learning : The High-Interest Credit Card of Technical Debt. In: NIPS Workshop on Software Engineering for Machine Learning (SE4ML)
Seaman CB, Shull F, Regardie M, Elbert D, Feldmann RL, Guo Y, Godfrey S (2008) Defect categorization: making use of a decade of widely varying historical data. In: Empirical Software Engineering and Measurement (ESEM)
Serban A, Blom KVD, Hoos H, Visser J (2020) Adoption and effects of software engineering best practices in machine learning. In: Empirical Software Engineering and Measurement (ESEM)
Shen Q, Ma H, Chen J, Tian Y, Cheung SC, Chen X (2021) A comprehensive study of deep learning compiler bugs. In: European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)

Shorten C, Khoshgoftaar TM (2019) A survey on Image Data Augmentation for Deep Learning. Journal of Big Data

Singh J, Singh K, Singh J (2019) Reengineering framework for open source software using decision tree approach. International Journal of electrical and computer engineering (IJECE) 9(3):2041–2048

Srivastava A, Thomson S (2009) Framework analysis: A qualitative methodology for applied policy research

Sun X, Zhou T, Li G, Hu J, Yang H, Li B (2017) An Empirical Study on Real Bugs for Machine Learning Programs. In: Asia-Pacific Software Engineering Conference (APSEC)

Szeliski R (2022) Computer vision: algorithms and applications. Springer Nature

Tan L, Liu C, Li Z, Wang X, Zhou Y, Zhai C (2014) Bug characteristics in open source software. Empirical Software Engineering (EMSE)

Tatman R, Vanderplas J, Dane S (2018) A Practical Taxonomy of Reproducibility for Machine Learning Research. In: Reproducibility in Machine Learning Workshop at ICML

Thiruvathukal GK, Lu YH, Kim J, Chen Y, Chen B (2022) Low-power Computer Vision: Improve the Efficiency of Artificial Intelligence

Thung F, Wang S, Lo D, Jiang L (2012) An empirical study of bugs in machine learning systems. In: International Symposium on Software Reliability Engineering (ISSRE)

Tian Y, Pei K, Jana S, Ray B (2018) DeepTest: automated testing of deep-neural-network-driven autonomous cars. In: International Conference on Software Engineering (ICSE)

Tsay J, Braz A, Hirzel M, Shinnar A, Mummert T (2022) Extracting enhanced artificial intelligence model metadata from software repositories. Empirical Software Engineering 27(7):176, DOI 10.1007/s10664-022-10206-6, URL `https://link.springer.com/10.1007/s10664-022-10206-6`

Tucker DC, Devon MS (2010) A Case Study in Software Reengineering. In: International Conference on Informatio (ITNG)n Technology: New Generations

Unceta I, Nin J, Pujol O (2020) Environmental adaptation and differential replication in machine learning. Entropy

Valett JD, McGarry FE (1989) A Summary of Software Measurement Experiences in the Software Engineering Laboratory. Journal of Systems and Software 9:137 – 148

Vartak M, Subramanyam H, Lee WE, Viswanathan S, Husnoo S, Madden S, Zaharia M (2016) Modeldb: a system for machine learning model management. In: the Workshop on Human-In-the-Loop Data Analytics

Villa J, Zimmerman Y (2018) Reproducibility in ML: why it matters and how to achieve it. URL `https://determined.ai/blog/reproducibility-in-ml`

Vinyals O, Babuschkin I, Czarnecki WM, Mathieu M, Dudzik A, Chung J, Choi DH, Powell R, Ewalds T, Georgiev P, et al. (2019) Grandmaster level in starcraft ii using multi-agent reinforcement learning. Nature

Voulodimos A, Doulamis N, Doulamis A, Protopapadakis E (2018) Deep Learning for Computer Vision: A Brief Review. Computational Intelligence and Neuroscience

Wang D, Li S, Xiao G, Liu Y, Sui Y (2021) An exploratory study of autopilot software bugs in unmanned aerial vehicles. In: ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engi-

neering (ESEC/FES), DOI 10.1145/3468264.3468559
Wang J, Dou W, Gao Y, Gao C, Qin F, Yin K, Wei J (2017) A comprehensive study on real world concurrency bugs in Node.js. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pp 520–531, DOI 10.1109/ASE.2017.8115663
Wang P, Brown C, Jennings JA, Stolee KT (2020) An Empirical Study on Regular Expression Bugs. International Conference on Mining Software Repositories (MSR)
Wardat M, Le W, Rajan H (2021) DeepLocalize: Fault Localization for Deep Neural Networks. In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp 251–262, DOI 10.1109/ICSE43902.2021.00034
Willemink MJ, Koszek WA, Hardell C, Wu J, Fleischmann D, Harvey H, Folio LR, Summers RM, Rubin DL, Lungren MP (2020) Preparing Medical Imaging Data for Machine Learning. Radiological Society of North America
Wu Y, Schuster M, Chen Z, Le QV, Norouzi M, Macherey W, Krikun M, Cao Y, Gao Q, Macherey K, et al. (2016) Google's neural machine translation system: Bridging the gap between human and machine translation. arXiv
Xin D, Miao H, Parameswaran A, Polyzotis N (2021) Production machine learning pipelines: Empirical analysis and optimization opportunities. In: Proceedings of the 2021 International Conference on Management of Data, pp 2639–2652
Xu S, Wang J, Shou W, Ngo T, Sadick AM, Wang X (2021) Computer Vision Techniques in Construction: A Critical Review. Archives of Computational Methods in Engineering
Yang J, Yan C, Wan C, Lu S, Cheung A (2019) View-centric performance optimization for database-backed web applications. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, pp 994–1004
Zhang R, Xiao W, Zhang H, Liu Y, Lin H, Yang M (2020a) An empirical study on program failures of deep learning jobs. In: International Conference on Software Engineering (ICSE)
Zhang T, Gao C, Ma L, Lyu M, Kim M (2019) An Empirical Study of Common Challenges in Developing Deep Learning Applications. In: International Symposium on Software Reliability Engineering (ISSRE)
Zhang X, Liu J, Sun N, Fang C, Liu J, Wang J, Chai D, Chen Z (2021) Duo: Differential Fuzzing for Deep Learning Operators. IEEE Transactions on Reliability
Zhang Y, Chen Y, Cheung SC, Xiong Y, Zhang L (2018) An empirical study on TensorFlow program bugs. International Symposium on Software Testing and Analysis (ISSTA)
Zhang Y, Ren L, Chen L, Xiong Y, Cheung SC, Xie T (2020b) Detecting numerical bugs in neural network architectures. European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)