Titre: Title:	Towards Debugging and Testing Deep Learning Systems
Auteur: Author:	Houssem Ben Braiek
Date:	2019
Type:	Mémoire ou thèse / Dissertation or Thesis
Référence: Citation:	Ben Braiek, H. (2019). Towards Debugging and Testing Deep Learning Systems [Mémoire de maîtrise, Polytechnique Montréal]. PolyPublie. https://publications.polymtl.ca/3959/

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Towards Debugging and Testing Deep Learning Systems

HOUSSEM BEN BRAIEK
Département de génie informatique et génie logiciel

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
Génie informatique

Juillet 2019

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Towards Debugging and Testing Deep Learning Systems

présenté par **Houssem BEN BRAIEK**
en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*
a été dûment accepté par le jury d'examen constitué de :

Ettore MERLO, président
Foutse KHOMH, membre et directeur de recherche
Christopher J. PAL, membre

DEDICATION

To my family

ACKNOWLEDGEMENTS

Firstly, I would like to express my deepest gratitude to my supervisor, Dr. Foutse Khomh, for his great guidance, encouragement and patience that he provided along my master's studies. Secondly, I also would like to thank Dr. Bram Adams who gave me co-guidance to accomplish my first conference paper, and all members (professors and students) in the software engineering team of the Department of Computer Engineering. In addition, I would like to acknowledge my parents who are always supporting me throughout my studies.

Finally, I would like to thank my committee members, Dr. Ettore Merlo, Dr. Christopher Pal, and Dr. Foutse Khomh for evaluating my master's thesis.

RÉSUMÉ

Au cours des dernières années, l'apprentissage profond, en anglais *Deep Learning (DL)* a fait d'énormes progrès, en atteignant et dépassant même parfois le niveau de performance des humains pour différentes tâches, telles que la classification des images et la reconnaissance vocale. Grâce à ces progrès, nous constatons une large adoption du DL dans des applications critiques, telles que la conduite autonome de véhicules, la prévention et la détection du crime, et le traitement médical. Cependant, malgré leurs progrès spectaculaires, les systèmes de DL, tout comme les logiciels traditionnels, présentent souvent des comportements erronés en raison de l'existence de défauts cachés ou d'inefficacités. Ces comportements erronés peuvent être à l'origine d'accidents catastrophiques. Ainsi, l'assurance de la qualité des logiciels (SQA), y compris la fiabilité et la robustesse, pour les systèmes de DL devient une préoccupation majeure. Les tests traditionnels pour les modèles de DL consistent à mesurer leurs performances sur des données collectées manuellement ; ils dépendent donc fortement de la qualité des données de test qui, souvent, n'incluent pas de données d'entrée rares, comme en témoignent les récents accidents de voitures avec conduite autonome (exemple Tesla/Uber).

Les techniques de test avancées sont très demandées pour améliorer la fiabilité des systèmes de DL. Néanmoins, les tests des systèmes de DL posent des défis importants, en raison de leur nature non-déterministe puisqu'ils suivent un paradigme axé sur les données (la tâche cible est apprise statistiquement) et leur manque d'oracle puisqu'ils sont conçus principalement pour fournir la réponse. Récemment, les chercheurs en génie logiciel ont commencé à adapter des concepts du domaine du test logiciel tels que la couverture des cas de tests et les pseudo-oracles, pour résoudre ces difficultés. Malgré les résultats prometteurs obtenus de cette rénovation des méthodes existantes de test logiciel, le domaine du test des systèmes de DL est encore immature et les méthodes proposées à ce jour ne sont pas très efficaces. Dans ce mémoire, nous examinons les solutions existantes proposées pour tester les systèmes de DL et proposons quelques nouvelles techniques. Nous réalisons cet objectif en suivant une approche systématique qui consiste à : (1) étudier les problèmes et les défis liés aux tests des logiciels de DL ; (2) souligner les forces et les faiblesses des techniques de test logiciel adaptées aux systèmes de DL ; (3) proposer de nouvelles solutions de test pour combler certaines lacunes identifiées dans la littérature, et potentiellement aider à améliorer l'assurance qualité des systèmes de DL.

Notre analyse des méthodes de test proposées pour les systèmes de DL montre que la plupart

d'entre elles se concentrent sur le test de la fiabilité et de la robustesse du modèle de DL, en générant automatiquement des scénarios de tests, par exemple, de nouvelles entrées transformées à partir de celles d'origine. Ces techniques font toutes l'hypothèse que les programmes d'entraînement des modèles de DL sont adéquats et corrects (sans bug). Une hypothèse qui n'est pas toujours valide car les programmes de DL comme tout autre programme peuvent contenir des incohérences et des bugs. De ce fait, nous proposons dans ce mémoire, un guide pratique et sa boîte à outils connexe basée sur TensorFlow pour détecter les problèmes dans les programmes d'entraînement de systèmes de DL. Notre évaluation empirique démontre que la vérification automatisée d'un ensemble d'heuristiques et d'invariants reflétant la santé du processus d'entraînement, peut être un moyen efficace pour tester et déboguer systématiquement les programmes réels de DL et les mutants des programmes de DL. De plus, les générateurs automatiques de données de test actuels optimisent la création de scénarios de tests par rapport un objectif particulier (comme couvrir les comportements majeurs et mineurs des modèles de DL). Deux limitations importantes de ces générateurs sont : (1) le manque de variabilité des transformations de données lors de l'utilisation d'optimiseurs basés sur les gradients (2) et l'aveuglement du fuzzing aléatoire qui ne garantit pas l'atteinte de l'objectif. Dans ce mémoire, nous proposons une approche de test basée sur les méta-heuristiques afin de permettre l'optimisation de l'objectif de test à travers divers cas de test générés à partir d'une grande variété des transformations. Notre évaluation sur des modèles de DL appliqués à la vision par ordinateur montre l'efficacité de notre approche novatrice qui augmente la couverture du modèle testé, trouve plusieurs comportements rares dans les cas particuliers, et surpasse Tensorfuzz, un outil existant de fuzzing guidé par la couverture, pour détecter les défauts latents introduits pendant le déploiement.

ABSTRACT

Over the past few years, Deep Learning (DL) has made tremendous progress, achieving or surpassing human-level performance for different tasks such as image classification and speech recognition. Thanks to these advances, we are witnessing a wide adoption of DL in safety-critical applications such as autonomous driving cars, crime prevention and detection, and medical treatment. However, despite their spectacular progress, DL systems, just like traditional software systems, often exhibit erroneous corner-cases behaviors due to the existence of latent defects or inefficiencies, and which can lead to catastrophic accidents. Thus, software quality assurance (SQA), including reliability and robustness, for DL systems becomes a big concern.

Traditional testing for DL models consists of measuring their performance on manually collected data; so it heavily depends on the quality of the test data that often fails to include rare inputs, as evidenced by recent autonomous-driving car accidents (*e.g.*, Tesla/Uber). Advanced testing techniques are in high demand to improve the trustworthiness of DL systems. Nevertheless, DL testing poses significant challenges stemming from the non-deterministic nature of DL systems (since they follow a data-driven paradigm ; the target task is learned statistically) and their lack of oracle (since they are designed principally to provide the answer). Recently, software researchers have started adapting concepts from the software testing domain such as test coverage and pseudo-oracles to tackle these difficulties. Despite some promising results obtained from adapting existing software testing methods, current software testing techniques for DL systems are still quite immature. In this thesis, we examine existing testing techniques for DL systems and propose some new techniques. We achieve this by following a systematic approach consisting of : (1) investigating DL software issues and testing challenges; (2) outlining the strengths and weaknesses of the software-based testing techniques adapted for DL systems; and (3) proposing novel testing solutions to fill some of the identified literature gaps, and potentially help improving the SQA of DL systems.

Our review of existing testing techniques for DL systems show that most of them focuses on testing the reliability and the robustness of the DL model using automatically generated test cases, *e.g.*, using new inputs transformed from the original inputs. These techniques assume that training programs are adequate and bug-free. An assumption that is not always true since DL programs like any software program can contain inconsistencies and bugs. Therefore, in this thesis, we propose a practical guide and its related TensorFlow-based toolkit for detecting issues in DL training programs. Our empirical evaluation demonstrates that

the automated verification of well-known heuristics and invariants reflecting the sanity of the learning process can be an effective way to systematically test and debug real world and mutants DL programs. In addition, the input generators, which have been involved in DL model testing, optimize the creation of test cases towards a particular test objective (*e.g.*, covering major and minor DL model's behaviors). Two important limitations of these existing automated generators are : (1) the lack of input transformations variability when using gradient-based optimizers and (2) the blindness of random fuzzing that do not guarantee reaching the objective. In this thesis, we propose a search-based testing method that relies on population-based metaheuristics to explore the search space of semantically-preserving metamorphic transformations. Using a coverage-based fitness function to guide the exploration process; it aims to ensure a maximum diversity in the generated test cases. Our evaluation on computer-vision DL models shows that it succeeds in boosting the neuronal coverage of DNNs under test, finding multiple erroneous DNN behaviors. It also outperforms Tensorfuzz, which is an existing coverage-guided fuzzing tool, in detecting latent defects introduced during the deployment.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ACRONYMS AND ABBREVIATIONS

AI	Artificial Intelligence
ML	Machine Learning
DL	Deep Learning
DNN	Deep Neural Network
FNN	Feedforward Neural Network
CNN	Convolutional Neural Network
TF	TensorFlow
DAG	Directed Acyclic Graph
SQA	Software Quality Assurance
SBST	Search-Based Software Testing
HPC	High Performance Computing
GPU	Graphic Processing Unit

CHAPTER 1 INTRODUCTION

In software engineering, software quality assurance (SQA) assembles standards, procedures, and activities that can be involved in different phases of the software development process in order to guarantee a high quality of the delivered software product. Depending on the application domain of the software and its critical aspect, SQA defines the required policies and activities that must be accomplished to validate the conformance of the software in-development to its corresponding quality standard requirements. For example, a flight simulator would have much higher defect tolerance than software for an actual airplane. In a broader sense, SQA incorporates and implements software testing methodologies to identify errors, bugs, or missing requirements. Testing activities assist the development process in reducing beforehand the number of bugs, and consequently, prevents defects from reaching final users. They have demonstrated their effectiveness in detecting bugs and improving the trustworthiness of software systems. In recent years, software systems have made remarkable progress in integrating more and more intelligent processing and autonomous decision-making. These intelligent systems rely heavily on machine learning algorithms; which poses some additional testing challenges stemming from their non-deterministic and data-sensitive nature. This thesis examines these challenges in depth, discusses the adapted software testing approaches proposed by the research community to circumvent them, and proposes some novel testing solutions that make it possible to fill some of the gaps identified in the literature. The goal is to improve, practically, the testing of ML software systems.

1.1 The promise of Deep Learning

Recent advances in computing technologies and the availability of huge volumes of data have sparked a new machine learning (ML) revolution, where almost every day a new headline touts the demise of human experts by ML models on some task. Deep Neural Networks (DNNs), which represent the backbone of deep learning, play a significant role in this revolution. These sophisticated learning models are composed of multiple interconnected layers to learn relevant features at multiple levels of abstraction, allowing the construction of complex functions between the inputs and their corresponding outputs. Thus, there is no more need to extract high-level, abstract features from raw data, which requires time and effort from human experts. Having both, a large amount of training data and high-performance computing resources is sufficient to construct and fit state-of-the-art DNNs that are able to exploit the unknown patterns in the raw data distribution in order to discover useful hierarchy of

features and learn powerful prediction models.

1.2 DNN-based Software Paradigm

In the earliest days of Artificial Intelligence (AI), intelligent software applications aimed to automate routine tasks without human intervention. The field of AI focuses on solving problems that are intellectually difficult for humans but relatively straightforward for computers when a formal specification and a list of mathematical rules are well defined. Nowadays, DNN-based software systems are considered to be the next generation of AI software [1], thanks to their innovative development paradigm, where the program's logic is inferred automatically from data. They aim to automate tasks that could be intuitively performed by human beings, but for which the solver logic is hard to describe formally; *e.g.*, recognizing spoken words in vocal speech or faces in numerical images.

Deep Learning (DL) is an emerging machine learning methods based on artificial neural networks, that requires a diverse background expertise; including mathematical reasoning, statistical algorithms, and software engineering. After realizing the power of DNN-based models, both academics and large corporations have invested a lot of time and resources in the development of open source libraries and tools that contribute to the democratization and the adoption of this prominent technology. In 2015, Google released its powerful library *TensorFlow* [2] to the public, under the Apache 2.0 open source license, Baidu, Facebook, Microsoft and Amazon, and research labs, such as Yoshua Bengio's MILA lab, quickly followed suit or had done so already. Thanks to these open source libraries, companies and individuals can now leverage state-of-the-art DL algorithms and build powerful DNN-based software systems with minimal overhead. Nowadays, DL libraries assist practitioners throughout the whole life cycle of their DNN systems. During the model engineering phase, DL engineers have to tune their DNN models by trying different configurations and assessing their impacts on the quality of the model. This task has become easier using DL libraries that provide different configuration choices as ready-to-use features. These DL libraries are often cross-language and cross-platform solutions that allow the migration from one software environment to another, which eases the deployment of models. Regarding hardware environment migrations, they support an optimization approach, named quantization [3], allowing to reduce the computational cost and the size of DNN models to meet requirements in terms of model footprint and energy consumption (*e.g.*, running inference on cell phones as TFLite [4]). The DNN quantization mainly consists in casting the DNN involving tensors (*i.e.*, multi-dimensional data arrays) into numerical representations with fewer bits of digital memory.

1.3 DNN-based Software Testing

DNN-based software systems are being increasingly deployed in large-scale applications touching critical aspects of our daily lives; from finance, energy, to health and transportation. Their reliability is therefore of paramount importance. However, the fact that their behavior is inferred from data makes them difficult to test as there is no reference oracle. In fact, DNN-based software fall into the category of non-testable programs that are mainly built to determine the answer [5]. The traditional way to test DNN-based software consists of evaluating statistically the performance of its core DNN model in predicting correct answers on a new sample of data that was kept unseen to the model during the training. However, this technique relies heavily on the quality of test data, *i.e.*, their ability to contain the main characteristics of training data distribution and to cover relatively rare input data. The improvement of testing data quality is based on collecting and labeling more input data, which is a high cost task requiring a great deal of human time and effort. Moreover, the traditional test based on model performance exclusively considers the resulting predict outcomes and target ones; it can therefore be affected by coincidental correctness issues, *i.e.*, hidden errors in a program that somehow by coincidence do not result in test failures. In fact, a DNN prediction model encapsulates a trained complex function mapping the inputs to the outputs, that includes a sequence of both linear and non-linear mathematical operations, which becomes long and complex as the model capacity increases. As this complexity grows, single elements of learned parameters have either small or no contribution to the predict output that becomes insensitive to certain features' ranges of values. Therefore, DNN-based software may contain hidden defects while still displaying an acceptable overall accuracy on test data samples. These hidden defects can result in incorrect behaviors when the model is deployed in the field; causing severe accidents and losses, as illustrated by the Google car accident [6] and the Uber car crash [7].

Advanced testing methodologies are needed to guarantee the reliability and robustness of in-production DNN-based systems. As a response to this need, researchers have adapted multiple software testing approaches used for testing traditional programs without a specified oracle. One common denominator of these testing techniques is the extension of human-crafted oracles by automated partial oracles. These approaches rely on feedback information from internal DNN states to generate synthetic inputs triggering new DNN state. Their objective is to find test cases that are resilient to coincidental correctness and allow exposing potential model misconceptions and latent implementation defects. However, such testing practices for DNN-based systems are model-based testing that generate new test inputs to detect potential inconsistencies in the behavior of the DNN model, assuming implicitly that

the training program is bug-free and that its configuration is consistent and optimal. However, this assumption is not always valid as shown by Zhang [8], who investigated bugs that occurred in deep learning training programs. Besides, the automatic testing approaches require a data generation module that is able to infer new synthetic inputs from original data set. Actually, there are two methods used for this module that already have limitations: (1) Guided Fuzzing is based on random input mutations, which can be inefficient because it is a kind of blind generation hoping to achieve the goal of triggering new DNN behaviors; (2) gradient-based generation consists of computing the gradient of the objective w.r.t the input, then, applying gradient-based transformations that take as input the original input and its gradient to infer new synthetic input having high chances to satisfy the objective. However, gradient-based transformations lack variability and flexibility. As an example in computer-vision models, it is possible to use effectively the gradient in imperceptible pixels' perturbations but there is no way to infer optimized parameters for image-based affine transformations such as rotation or translation.

1.4 Research Statement

Existing DL testing methods rely on model evaluation, through automated data generation using either a gradient-based optimizer or a guided fuzzing process. To the best of our knowledge, researchers have not yet leveraged search-based approaches involving metaheuristics as gradient-free optimizer. In addition, most of these DL testing methods focus on testing the core model, assuming that the training program is bug-free and that the DNN is properly configured and well trained. In this thesis, we study the potential bugs and the challenges of testing DNN-based programs and propose automated verification routines for detecting training issues, as well as a search-based approach to improve the generation of test data that increases the DNN coverage and the diversity of inputs. The purpose is to expose DNN models' inconsistencies and find potential hidden defects.

1.5 Thesis Overview

In this thesis, we follow a systematic approach that consists of : (1) investigating DL software issues and testing challenges; (2) outlining the strengths and weaknesses of the software-based testing techniques adapted for DL systems; (3) proposing novel testing solutions to fill some of the identified literature gaps, and potentially help improving the SQA of DL systems. Next, we describe each step in detail:

1. *A comprehensive review of DNN-based software testing.* We analyze former research

works that proposed testing techniques for ML program, including its main components(*i.e.*, data and model). We organize the testing techniques in two categories based on the intention behind the techniques, *i.e.*, techniques that aim to detect conceptual and implementation errors in data, and techniques that focus on ensuring correct conception and implementation of ML models. Therefore, we discuss the fundamental concepts behind each proposed techniques, explaining the types of errors they can identify while also outlining their limitations.

2. *TFCheck: A TensorFlow Library for Detecting Training Issues in Neural Network Programs.* We examine training issues in DNN programs and introduce a catalog of verification routines that can be used to detect the identified issues, automatically. Then, we describe our implementation of the suggested detection mechanisms in a TensorFlow-based library named TFCheck, which can be used for monitoring and debugging Tensorflow-based DNN models.

3. *DeepEvolution: A Search-based testing approach for DNN models.* We design a novel model testing technique that can be seen as a renovation or adaptation of Search-based Software Testing (SBST) for DNN models. It aims to detect inconsistencies and potential defects related to the functionality of DNN models, during two important phases in their lifecycle: (1) model engineering and (2) model deployment. Then, we assesses the effectiveness of DeepEvolution in testing computer-vision DNN models.

1.6 Thesis Contribution

In this thesis, we carry out a deep study on DNN-based software issues and software testing approaches renovated and adapted to test them, with aim of proposing more advanced solutions that fill gaps in the existing literature. Our contributions are as follows:

- Based on a detailed review of current existing testing approaches for DL programs, we summarize and explain challenges that should be addressed when testing DL programs. Then, we identify gaps in the literature related to testing DL systems.

- We provide a practical guide and its related TensorFlow-based library outlining verification routines for training issues, that developers can use to detect and correct errors in their DNN training programs. Our evaluation shows that our library is effective in detecting training issues through a case study with real-world, mutants, and synthetic training programs.

- We construct, DeepEvolution, the first SBST approach for testing DNN models that succeeds in boosting the neuronal coverage, generate diverse synthetic testing inputs that uncover corner-cases behaviors and expose multiple latent quantization defects. DeepEvolution also outperforms TensorFuzz, which is an existing coverage-guided fuzzing tool, in terms of the number of the detected quantization defects. These results open up a promising area of research into effective SBST approaches for testing DL systems.

1.7 Organisation of the Thesis

The rest of this thesis is organised as follows:

- Chapter 2 presents the fundamental concepts and methods related to deep learning and software testing that are needed to understand our research work.
- Chapter 3 outlines a comprehensive review in the area of DL programs testing.
- Chapter 4 presents our guide of verification routines for DNN training programs and its corresponding TensorFlow-based library.
- Chapter 5 presents our search-based software testing approach specialized for DNN models and its concrete instantiation for testing computer-vision models.
- Chapter 6 summarises and concludes the thesis, and discusses future work.

CHAPTER 2 BACKGROUND

Chapter Overview Section 2.1 introduces the concepts and methods of Deep learning. Section 2.2 describes the concepts and methods of software testing specialized for non testable programs (*i.e.*, without oracle). Finally, Section 2.3 concludes the chapter.

2.1 Deep Learning

Conventional machine learning algorithms have been constructed to recognize the hidden patterns in the data and learn the relationship between the input features and the target output. Thus, they require feature engineering methods and considerable domain expertise to perform the extraction of a vector of useful features from raw data that will be fed to the ML algorithm in order to detect patterns in the input and learn the statistical model. However, human-crafted features require huge time and effort when the task to learn is complex and its input data are characterized by high-dimensionality. Therefore, the manual feature engineering is insufficient to learn effective prediction models that can be generalized to new input examples when working with high-dimensional data. Deep learning (DL) overcomes this limitation by mixing the learning of both high-level features and patterns in one sophisticated statistical model. Indeed, deep neural networks (DNNs), which are the backbone behind DL, use a cascade of multiple representation levels, where higher-level abstractions are defined in terms of lower-level ones. This powerful computational model, including multiple processing layers, is able to exploit unknown patterns in the raw data distribution in order to identify relevant hierarchy of features and learn complex mapping functions for a target prediction. However, in the earliest days of DNNs, it was difficult to train them properly, because of their huge number of parameters. In the last decade, the amount of available training data has increased and the advent of general purpose GPUs has given rise to parallel and high performance algebraic computation resources. Because of this progress, researchers are now able to train DNNs on large and high dimensional datasets 10 or more times faster. In addition, the DNN architectures have grown in size over time and have made major advances in solving problems that have resisted the best attempts of the ML community for many years. Our research work focuses, particularly, on feed-forward neural networks, which are the subject of the next section.

2.1.1 Feed Forward Neural Network

Feedforward neural network (FNN) architecture is the quintessential and the most used DL model. The objective of FNNs is to learn the mapping of a fixed-size input (for example, an image) to a fixed-size output (for example, a probability for each label). Apart from the input and output layers, FNN contains a cascade of multiple intermediate layers, which are called hidden layers because the ground truth data does not provide the desired values for these layers. The name feedforward arises from the fact that information flows through the processing layers in a feed-forward manner, *i.e.*, from input layer, through hidden layers and finally to the output layer without any feedback connection in which intermediate outputs could fed back into previous ones. Last, neural terminology appears in FNN because they are inspired from neuroscience. Each hidden layer contains a set of computation units, called neurons, that perform a weighted sum of their inputs from previous layers and pass the result through an activation function. The latter is a non-linear function that allows adding non-linearity in the approximated mapping function in order to be insensitive to irrelevant variations of the input. The strength of this complex model lies in the universal approximation theorem. A feed-forward network with a single hidden layer containing a finite number of neurons can approximate any continuous function, under mild assumptions on the activation function. However, this does not indicate how much neurons are required and the number can evaluate exponentially with respect to the complexity of formulating the relationships between inputs and outputs. That is why, researchers have constructed Deep FNNs that contain several hidden layers with different width (*i.e.*, number of neurons) to solve problems such as image and speech recognition, requiring both high selectivity and invariance in their input-output mapping functions. For example, Deep FNN for image recognition, should be selective to learn relevant features that are important for discrimination such as car wheels, but should be invariant to irrelevant aspects such as orientation or illumination of the car.

Dense Neural Network

Dense Neural Networks can be seen as regular feedforward neural networks (see Figure 2.1), where the layers are fully connected (densely) by neurons, meaning that all the neurons in a layer are connected to those in the next layer. A Dense Neural Network accepts an input feature as a single vector, then transforms it through a series of vector-valued hidden layers. Neurons in a single layer are completely independent and do not share any connection; each neuron performs a vector-to-scalar linear calculation follow by a non-linear activation to provide its own state. The output layer is the last fully connected layer that represents one

continuous output in regression settings, or a vector of class scores in classification settings.

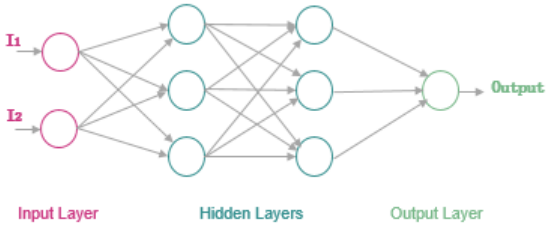

Figure 2.1 Dense Neural Network

Convolutional Neural Network

Convolutional Neural Network (CNN) represents a particular type of feedforward network that is designed to process data in the form of multiple arrays, such as 2D images and audio spectrograms, or 3D videos. It has achieved many practical successes in detection, segmentation and recognition of objects and regions in images; hence, it has recently been widely adopted by the computer-vision community. Figure 2.2 shows the architecture of a CNN that takes advantage of the spatial and temporal dependencies in the multi-dimensional input data. The CNN's specialized layers have neurons arranged in 3 dimensions: width, height, and depth. Three main types of those specialized layers transform the 3D input volume to a 3D output volume of neuron activations: Convolutional Layer, Activation Layer, and Pooling Layer.

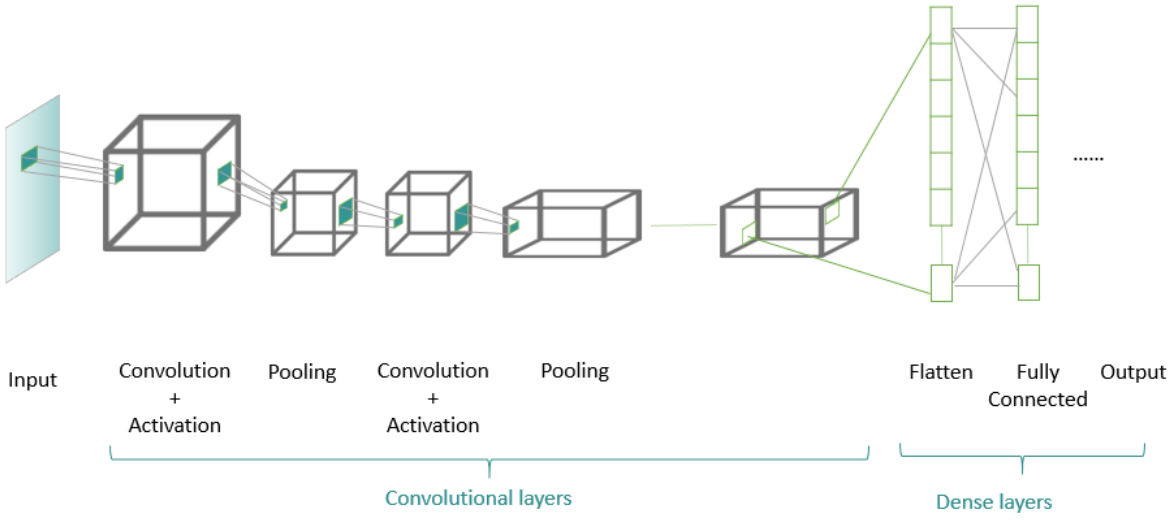

Figure 2.2 Convolutional Neural Network

Convolutional Layer : The main building block of this type of transformation layer is the convolution. A convolution provides a 2D feature map, where each unit is connected to local regions in the input data or previous layer's feature map through a multi-dimensional parameter called a filter or kernel. Indeed, those filters play the role of feature detectors. The feature map is produced by sliding the filter over the input data, then computing products between the filter entries and the local input region at each spatial position, to infer the corresponding feature map response. Different filters are performed in a layer and resulting feature maps are stacked in 3D volumes of output neurons. These separate filters aim to detect distinctive local motifs in the same local regions. However, all units in one feature map share the same filter, because motifs are generally invariant to location and it is useful to attempt finding it at different spatial positions. In addition, this assumption could reduce significantly the number of parameters used in convolutional layers through parameter sharing scheme. Therefore, a densely connected layer provides learning features from all the combinations of the features of the previous layer, whereas a convolutional layer relies on consistent features with a small repetitive field.

Activation Layer : As in any FNN layer, we use an activation function to add non-linearity to the computed value. The activation layer applies the activation function on extracted feature map as an element wise operation (*i.e.*, per feature map output). The resulting activation map indicates the state of each neuron, *i.e.*, active or not. For example, ReLU, which stands for Rectified Linear Unit, is widely used in CNNs that simply replaces the negative input values by zero; otherwise, it keeps the positive value as it is.

Pooling Layer : The pooling layer ensures spatial pooling operation to reduce the dimensionality of each feature map and to retain the most relevant information by creating an invariance to small shifts and distortions. Those spatial pooling operations can be of different types : Max, Average, Sum, etc... They define a neighbouring spatial window size, then, they take the maximum element from that window in the activation map if it is max pooling. Besides, they can compute the average or sum of all elements in that window for, respectively, average or max pooling. Apart from robustness to irrelevant variance, neighbouring pooling makes the input representations (*i.e.*, resulting activation maps) smaller and more manageable, hence, it reduces the number of parameters and internal computations. This helps shortening the training time and controlling the overfitting.

The architecture of CNN can be seen as two principal stages : (1) multiple stack of convolution, activation, and pooling that ensure the detection of relevant features from the input data; (2) the final activation map is flatten to be a vector of features and is fed to a dense

neural network that performs the prediction on top of these extracted features to estimate the labels' scores or the predict value.

2.1.2 DNN-based Model Engineering

Regardless of the chosen DNN model, the training algorithm aims is to drive the approximated mapping function to match the training distribution data. The learning aspect resides in the model fitting which is an iterative process during which little adjustments are made repeatedly, with the aim of refining the model until it predicts mostly the right outputs. Generally, an FNN is trained on input data through optimization routines using gradient learning, to create a better model or probably the best fitted one (*i.e.*, this could happen with a convex objective function or advanced update steps). The principal components of a DNN model are:

- **Parameters**, represent weights and biases used by neurons to make their internal calculations. The training program gradually adjusts these inner variables on its own through successive training iterations to build its logic and form its best function approximation.

- **Activations**, represent non-linear functions that are added following linear computation, in order to add non-linearity capacity in the computation of neuron outputs. Fundamentally, they indicate if a neuron should be activated or not given linear computation's results, *i.e.*, to achieve this goal, they assess how much the information received by a neuron could contribute to estimate the predicted outcome.

- **Loss Function**, consists of a mathematical function, which given a real value provides an estimation of the model learning performance, by evaluating the sum or the average of the distance between predicted outputs and the actual outcomes. If the model's predictions are perfect, the loss is zero; otherwise, the loss is greater than zero.

- **Regularization**, consists in techniques that penalize the model's complexity to prevent overfitting. Simple regularization techniques consist in adding parameters' $L2$ or $L1$ norms to the loss, which results in smaller overall weight values. A more advanced regularization technique is *dropout*; it consists in randomly selecting a percentage of neurons from training during an iteration and removing them. Doing this prevents models with high learning capacity from memorizing the peculiarities of the training data, which would have resulted in complex models that overfit the training data, and hence perform poorly on unseen data. To guarantee a model's generalization ability,

two parallel objectives should be reached: (1) build the best-fitted model *i.e.*, lowest loss and (2) keep the model as simple as possible *i.e.*, strong regularization.

- **Optimizers** adjust the parameters of the model iteratively (reducing the objective function) in order to : (1) build the best-fitted model *i.e.*, lowest loss; and (2) keep the model as simple as possible *i.e.*, strong regularization. The most used optimizers are based on gradient descent algorithms, which minimize gradually the objective function by computing the gradients of loss with respect to the model's parameters, and update their values in direction opposite to the gradients until a local or the global minimum is found. The objective function has to be globally continuous and differentiable. It is desirable that this function be also strictly convex and has exactly one local minimum point, which is also the global minimum point. As DNN's objectives are not convex functions, many gradient descent variants, that use stochastic gradient or momentum gradient descent, have a high probability of finding reasonably good solutions anyway, even though these solutions are not guaranteed to be global minimums.

- **Hyperparameters** are model's parameters that are constant during the training phase and which can be fixed before running the fitting process. Training a model with pre-fixed hyperparameters consists of identifying a specific point in the space of possible models. In fact, choosing in prior hyperparameters, such as the number of layers and neurons in each layer, the learning rate for gradient descent optimizer, or the regularization rate allows to define a subset of the model space to search. It is recommended to tune hyperparameters in order to find the composition of hyper-parameters that helps the optimization process find the best-fitted model. The most used search methods are (1) *Grid search*, which explores all possible combinations from a discrete set of values for each hyperparameter; (2) *Random search* which samples random values from a statistical distribution for each hyperparameter, and (3) *Bayesian optimization* which chooses iteratively the optimal values according to the posterior expectation of the hyperparameter. This expectation is computed based on previously evaluated values, until converging to an optimum.

To solve a problem using a DL solution, DL engineer tries to collect representative data that incorporate the knowledge required to perform the target task. If the associated labels are not available for the gathered inputs, he should collaborate with domain experts for labeling. The manually labeled data is divided into three different datasets: *training dataset*, *validation dataset*, and *testing dataset*. After readying these data sets, DL engineer designs and configures the DNN-based model by choosing the architecture, setting up initial hyperparameters values, and selecting variants of mathematical components that include activation

functions, loss functions, regularization terms, and gradient-based optimizers. Generally, DNN's design should consider statistic-based requirement (*e.g.*, expected prediction performance), data complexity, and best practices or guidelines from other works that addressed similar problems. After preparing the DL system ingredients (*i.e.*, data and model design), the training process starts and systematically evolves the decision logic learning towards effectively resolving the target task. Indeed, training a DNN-based model using an optimizer consists in gradually minimizing the loss measure plus the regularization term with respect to the training dataset. Once the model's parameters are estimated, hyperparameters are tuned by evaluating the model performance on the *validation dataset*, and selecting the next hyperparameters values according to a search-based approach that aims to optimize the performance of the model. This process is repeated using the newly selected hyperparameters until a best-fitted model is obtained. Last, this best-fitted model is tested using the testing dataset to verify if it meets the statistic-based requirement. Therefore, for traditional software, a human developer needs to understand the specific task, identify a set of algorithmic operations to solve the task, and program the operations in the form of source code for execution. However, DL software automatically distills the computational solution of a specific task from the training data; so the DNN-based software decision logic is encoded in the trained DNN model, consisting of the DNN architecture (*i.e.*, computation flow and layers' connection) and parameters (*i.e.*, weights and biases).

2.1.3 DNN-based Software Development

Deep learning algorithms are often proposed in pseudo-code formats that include jointly scientific formula and algorithmic rules and concepts. Indeed, the DNN-based software are implemented as traditional software using programming languages such as Python, but they represent a kind of scientific software including algebraic computation and automatic differentiation. Given the growth of training data amount and the increasing DNN architecture depth, all algebraic computations should be implemented in a way that support parallel and distributed execution in order to leverage the power of high-performance computing hardware architectures. This enables the large-scale training of DNNs with hundreds of billions of parameters on hundreds of billions of example records using many hundreds of processing units. DL systems require a lot of computation and involve quite a lot of parallel computing and software engineering effort to transfer such ideally designed DL algorithm to in-production DL software.

Fortunately, with the recent boom in DL applications, several DL software libraries such as Theano [9], Torch [10], Caffe [11], and Tensorflow [2] have been released to assist ML

practitioners in developing and deploying state-of-the-art DNNs and to support important research projects and commercial products involving DL models. These libraries reduce the big gap between well-defined theoretical DL algorithms and compute-intensive, parallel DL software. Nowadays, a DL engineer often leverages an existing DL framework to encode the designed DNN into a DL program. He can leverage ready-to-use features and routines offered by DL libraries to help developers to build solutions for designing, training and validating DL models for real-world, complex problems. In addition, DL libraries play an important role in DNN-based software development because they provide optimized implementations of highly intensive algebraic calculations, which take full advantage of distributed hardware infrastructures in the form of high-performance computing (HPC) clusters, parallelized kernels running on high-powered graphics processing units (GPUs), or the merge of these two technologies to breed new clusters of multiple GPUs.

In our research work, all the implementations proposed are TensorFlow-based solutions; so we detail furthermore the design and the features of TensorFlow (TF), which is an open source deep learning library released by Google. As shown in Figure 2.3, TF is based on a Directed Acyclic Graph (DAG) that contains nodes, which represent mathematical operations, and edges, which encapsulate tensors (*i.e.*, multidimensional data arrays). This dataflow graph offers a high-level of abstraction to represent all the computations and states of the DNN model, including the mathematical operations, the parameters and their update rules, and the input preprocessing. Thus, the dataflow graph establishes the communication between defined sub-computations; so it makes it easy to partition independent computations and execute them in parallel on multiple distributed devices. Given a TF dataflow graph, the TF XLA compiler is able to generate an optimized and faster code for any targeted hardware environment including conventional CPUs or higher-performance graphics processing units (GPUs), as well as Google's custom designed ASICs known as Tensor Processing Units (TPUs). Figure 2.4 shows that a TF program uses lazy evaluation or deferred execution and is composed of two principal phases: (1) a construction phase that assembles a graph, including variables and operations. It creates a session object to construct the computation graph that represents the DNN model's DAG; (2) an execution phase that uses a session to execute operations and evaluate results in the graph. Indeed, the session object provides a connection between the code and the DAG on execution. The training process consists of multiple consecutive *session.run()* calls to execute the training operation with the batches of data. During the training phase, the model's variables, which hold in-memory the weights and biases, are updated continuously. However, tensors resulting from intermediate computations such as activations (forward pass) or gradients (backward pass) do not survive past a single execution of the graph. During the testing phase, it is a simple inference of predictions

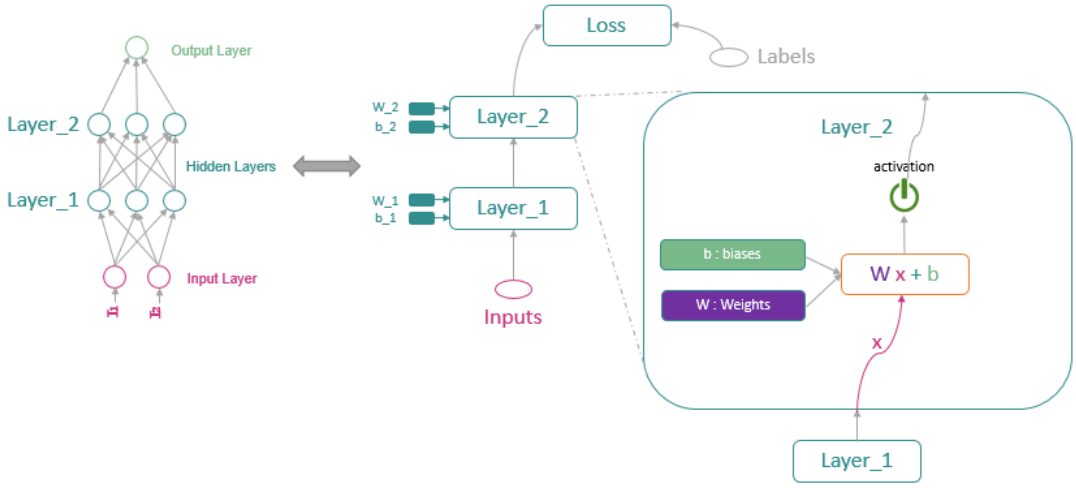

Figure 2.3 Tensorflow Computational Graph

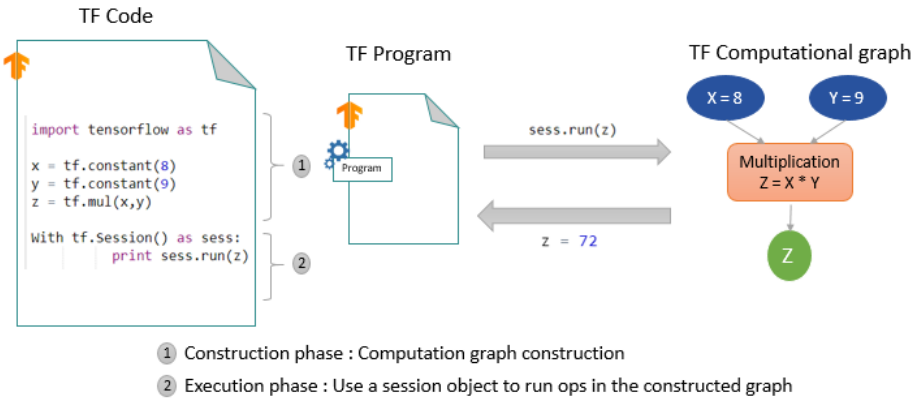

Figure 2.4 TensorFlow Program Execution

using session object to feed the testing data to the model and fetch the predictions in order to compute metrics such as accuracy or error rate.

2.2 Software Testing

Software testing consists in assessing the program internal states and outputs in order to find potential erroneous behaviors or bugs. A software bug refers to an imperfection in a program that causes a discordance between the existing and the required behaviors of the system. To track software bugs, we often need a test oracle, which allows distinguishing between the correct and incorrect obtained behaviors of the program under test. Test oracles could be automated using different techniques including modelling, specifications, and contract-driven development. Otherwise, human beings, who are often aware of informal specifications

and implicit knowledge, remain the last resort to construct test oracle information. Thus, software testing activities are designed to reveal bugs, including test oracle identification, test adequacy evaluation, test input generation.

2.2.1 Derived Test Oracle

A derived test oracle consists of a partial oracle built from multiple sources of information including program executions, program properties, and other implementations. When it is well-defined based on methods introduced below, a derived test oracle is able to distinguish a program's correct behavior from an incorrect behavior. Moreover, those automated partial oracle can be improved, over time, to become more effective in detecting inconsistencies.
We detail some methods that have been adapted for DNN-based software:

Metamorphic Testing

Metamorphic testing [12] is a derived test oracle that allows finding erroneous behaviors by detecting violations of identified metamorphic relations (MR). The first step is to construct MRs that relate inputs in a way that the relationship between their corresponding outputs becomes known in prior, so the desired outputs for generated test cases can be expected. For example, a metamorphic relation for testing the implementation of the function $\sin(x)$ can be the transformation of input x to $\pi - x$ that allows examining the results by checking if $\sin(x)$ differs from $\sin(\pi - x)$. The second step is to leverage those defined MRs in order to generate automatically partial oracle for follow-up test cases, *i.e.*, genuine test inputs could be transformed with respect to one MR, allowing the prediction of the desired output. Therefore, any significant differences between the desired and the obtained output would break the relation, indicating the existence of inconsistencies in the program execution. As the example of $\sin(x)$, breaking the relation between x and $\pi - x$ stating that $\sin(x) = \sin(\pi - x)$ indicates the presence of an implementation bug without needing to examine the specific values computed through execution. Figure 2.5 illustrates a metamorphic testing workflow. The efficiency of the metamorphic testing depends on the used MRs. MRs can be identify manually from specific properties of the algorithm implemented by the program under test, or inferred automatically by tracking potential relationships among the output of test cases run that hold across multiple executions. However, automatically generated MRs need to be analyzed by an expert to ensure that they are correct considering the domain knowledge. Experts can also help provide more generic formulations.

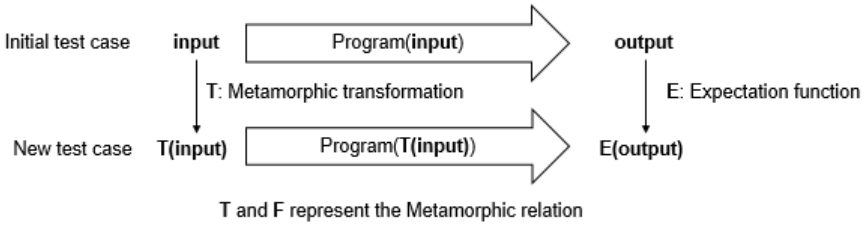

Figure 2.5 Illustration of Metamorphic Testing

Differential Testing

The differential testing [13] creates a partial test oracle that detects erroneous behaviors by observing whether similar programs yield different outputs regarding identical inputs. The intuition behind this approach is that any divergence between programs' behaviors, solving the same problem, on the same input data indicates the presence of a bug in one of them. It is quite related to N-version programming that aims to produce, independently, alternative program versions of one specification, so that if these multiple program versions return different outputs on one identical input, then a "voting" mechanism is leveraged to identify the implementations containing a bug. Davis and Weyuker [14] discussed the application of differential testing for 'non-testable' programs. The testing process consists of building at first multiple independent programs that fulfill the same specification, but which are implemented in different ways, *e.g.*, different developers' team, or different programming language. Then, we provide the same test inputs to those similar applications, which are considered as cross-referencing oracles, and watch differences in their execution to mark them as potential bugs (see Figure 2.6).

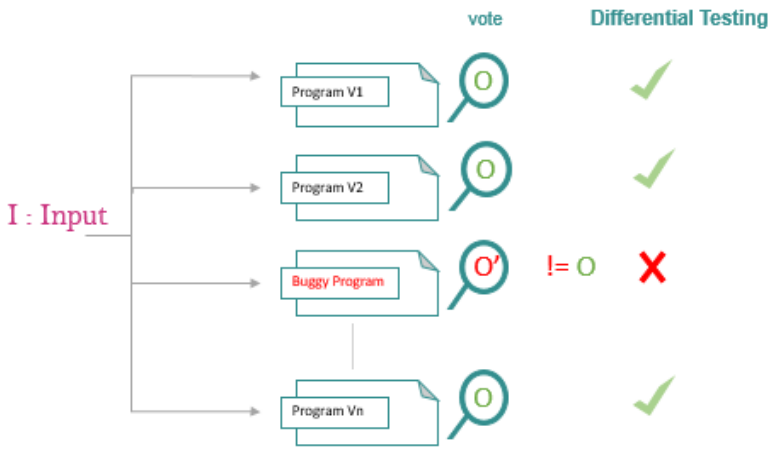

Figure 2.6 Illustration of Differential Testing

Invariant Detection

A derived test oracle can be formulated as a set of invariants that allow aligning an incorrect execution against the expected execution [15]. Indeed, those invariants can be identified from white-box inspection of the program under test; otherwise, they can be inferred automatically from the program execution trace, but human intervention is needed to filter the found invariants, *i.e.*, retaining the correct ones and discarding the rest. Thus, the defined invariants are instantiated by binding their variables to the program's intermediates values; so they can serve as test oracle to capture program behaviors and check the program correctness. Figure 2.7 illustrates a testing workflow using the program's invariants.

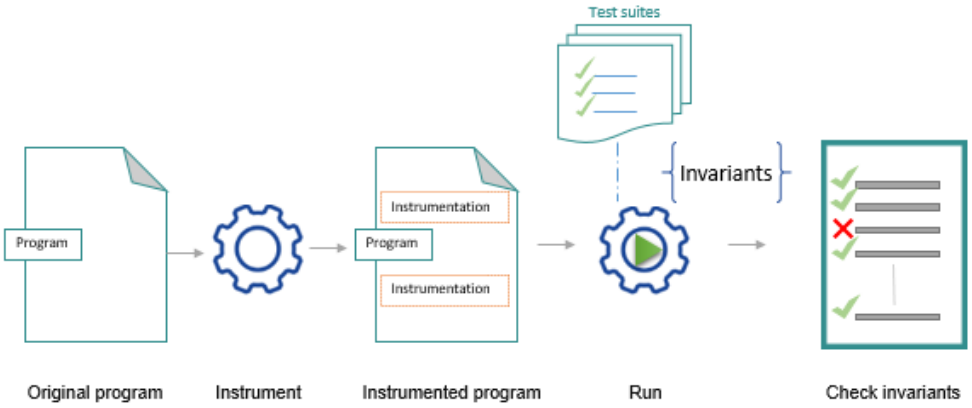

Figure 2.7 Illustration of Invariant Detection

To automate the testing process, we need to apply partial oracles capable of evaluating the program behavior and its outputs given inputs, but not only that, we need to automate the generation of input data for the program under test, that have better chances to expose bugs. Nevertheless, it is important to begin with introducing criteria that help testers evaluating the quality of the running test cases in terms of bugs detection ability.

2.2.2 Test Adequacy Evaluation

Test adequacy evaluation consists of assessing the fault-revealing ability of existing test cases. It is based on different adequacy criteria that estimate if the generated test cases are 'adequate' enough to terminate the testing process with confidence that the program under test is implemented properly. Popular test adequacy evaluation methods are code coverage and mutation testing.

Code coverage : it measures the proportion of the program's source code that is executed by test cases. It helps assessing the amount of internal logic triggered when testing the program. For example, one can use statement coverage (*i.e.*, run each statement at least one time), branch coverage (*i.e.*, try each branch from any decision point at least once), and path coverage (*i.e.*, test the different possible sequence of decisions from entry to the program exit). Indeed, test cases achieving high coverage are more likely to uncover the hidden bugs because the portions of code that have never been executed have a higher probability of containing defects and to work improperly.

Mutation Testing : it is a test adequacy evaluation that assesses the effectiveness of the test cases in revealing faults intentionally introduced into the source code of mutants [16]. Indeed, test cases should be robust enough to kill the mutant code (*i.e.*, the mutant failed the test). A mutation is a single syntactic change that is made to the program code; so each mutant differs from the original version by one mutation. The latter can be categorized into 3 types: (1) statement mutation: changes done to the statements by deleting or duplicating a line of code; (2) decision mutation: one can change arithmetic, relational, and logical operators; (3) value mutation: values of primary variables are modified such as changing a constant value to much larger or smaller values. The ratio of killed mutants against all the created mutants is called the mutation score, which measures how much the generated test cases are efficient in detecting injected faults.

Test adequacy criteria can be adopted to guide the test generation toward optimizing the testing effort and producing effective test cases that enhance the chances of detecting defects. The test data generation process is the subject of the next sub-section.

2.2.3 Test Data Generation

Generating test inputs for software is a crucial task in software testing. In the earliest days, test data generation was a manual process mainly driven by the tester, which inspects the specification and the code of the program. However, this traditional practice is costly and laborious since it requires a lot of human time and effort. Researchers have worked to automate the test inputs generation process, but automation in this area is still limited. In fact, exhaustive enumeration of all possible inputs is infeasible for large size programs and previous attempts to automate the generation of inputs for real-world software applications have proven that it is an undecidable problem [17]. The spectrum of test inputs generation techniques includes black-box (functional), white-box (structural), and grey-box (combination of structural and functional) testing. Functional testing is an approach where tests are

created directly from program requirements. In the following, we present well-known black-box techniques used to design test cases that reduce the testing effort and eliminate the need for exhaustive test inputs generation (which is not feasible).

Equivalence class partitioning [18] : It is based on partitioning of the input domain into finite number of equivalence classes in a way that all inputs of an equivalence class are processed in an equivalent way by the program under test. Using this technique, a tester can select only a given number of inputs from each equivalence class as a representative sample of that class. The assumption is that if a test case in a given equivalence class did not detect a particular type of defect then no other test case based on that class would detect the defect. Therefore, it allows testers to cover a large domain of inputs with a subset of test inputs with a high probability of detecting potential bugs.

Boundary value analysis [19] : It focuses on inputs that are above and below the edges of equivalence classes. Experience shows that test cases considering these boundaries are often more valuable in revealing defects. Thus, boundary value analysis consists of selecting inputs close to edges, so that both of the upper and lower bounds of an equivalence class are covered by test cases. This technique can strengthen the use of equivalence class partitioning.

Besides, structural testing is the type of test derived from knowledge of the internal structure of the code. The generation of test inputs that achieve high code coverage can be a hard problem considering the size and complexity of the software under test. A random testing would often fail to trigger particular states of the program, which results in low code coverage. Researchers have shown that both dynamic symbolic execution and search-based techniques overcome the limitations of random testing [20] [21]; so they are able to reach high or at least acceptable code coverage in reasonable time.

Dynamic Symbolic Execution (DSE) [20] : It is mainly based on a static analysis of the code. The testing process consists of running the program with given test inputs and performing a symbolic execution in parallel, to collect symbolic constraints obtained from the static analysis of the execution traces. Then, DSE uses a solver that generates new tests that aim to satisfy uncovered expressions or path conditions. Following this process, DSE has been found to be accurate and effective in generating automatically test inputs that achieve high code coverage.

Search-based Software Testing (SBST) [21]: It formulates the code coverage criteria as a fitness function, which can compare and contrast candidate solutions from the

space of possible inputs in terms of the covered portions of the code. Using this fitness function, SBST leverages metaheuristic search techniques, such as Genetic Algorithm, to drive the search into potentially promising areas of the input space; generating effective test cases and increasing the code coverage.

Since we propose a search-based approach for DNN models, we detail furthermore the concept of metaheuristics [22]. Metaheuristics represent computational approaches that resolve an optimization problem by iteratively attempting to ameliorate a candidate solution with respect to a fitness function. They require only few or no assumptions on the properties of both the objective function and the inputs search space. However, they do not provide any guarantee of finding an optimal solution. Their applicability in structural testing is suitable as these problems frequently encounter competing constraints and require near optimal solutions, and these metaheuristics seek solutions for combinatorial problems at a reasonable computational cost.

2.3 Chapter Summary

In this chapter, we briefly introduce the key concepts and methodologies that are related to DNN-based software systems and software testing, in order to ease the understanding of the different testing approaches presented in this thesis.

In the following chapter, we perform a comprehensive review and discussion of the research works that exist in the area of deep learning testing.

CHAPTER 3 A COMPREHENSIVE REVIEW : DNN-BASED SOFTWARE TESTING

Nowadays, we are witnessing a wide adoption of DL models in a wide range of areas, from finance, energy, to health and transportation. Many people are now interacting with systems based on DL every day, *e.g.*, voice recognition systems used by virtual personal assistants like Amazon Alexa or Google Home. Given this growing importance of DL-based systems in our daily life, it is becoming utterly important to ensure their reliability. Recently, software researchers have started adapting concepts from the software testing domain (*e.g.*, code coverage, mutation testing, or property-based testing) to help DL engineers detect and correct faults in DL programs. However, the shift in the development paradigm induced by DL makes it difficult to reason about the behavior of software systems with DL components, resulting in systems that are intrinsically challenging to test and verify, given that they do not have (complete) specifications or even source code corresponding to some of their critical behaviors. In fact, some DL programs rely on proprietary third-party libraries such as Intel Math Kernel Library (MKL) and NVIDIA CUDA Deep Neural Network library (cuDNN) for many critical operations. A defect in a DL program may come from its training data, program code, execution engine, or third-party frameworks. Compared with traditional software, the potential testing space of a DL programs is multi-dimensional. Thus, current existing software development techniques must be revisited and adapted to this new reality.

This chapter reviews current existing testing practices for DL programs. First, we identify and explain challenges that should be addressed when testing DL programs. Next, we report existing solutions found in the literature for testing DL programs. Finally, we identify gaps in the literature related to the testing of DL programs that have been exploited by our research work.

Chapter Overview Section 3.1 introduces the challenges and issues when engineering DL systems. Section 3.2 presents research trends in DL application testing. Section 3.3 describes the gaps and opportunities in the literature that have been exploited by our research work. Section 3.4 concludes the chapter.

3.1 DL System Engineering: Challenges and Issues

DL systems are data-sensitive applications whose decision logic is learned from the training data and generalized to the future unseen data. To begin with, the data is one of the most

important component in DL system because the training program distills the computation solution for the target task from the given inputs. Thus, low data quality in terms correctness and representativeness will result in a poor model that could not solve the target problem and perform the task. The DNN model is designed and configured, taking into account requirements, data complexity, as well as the problem domain. Then, the training algorithm tunes the parameters to fit the input data through an iterative process, during which the performance of the model is assessed continuously. Even if the learning algorithm is well defined using pseudo-code and mathematical notations, the DL software is often written as traditional software (*e.g.*, in Python) using eventually DL library or framework; so this establishes a gap to transfer such ideally designed DL models from the theory to a real-world DL application implemented on top of DL frameworks. Thus, a misconception of the neural network model or a buggy implementation of the training program will also result in a poor model that could not capture relevant information from the data and infer the mapping relation between the input features and the predicted output.

Therefore, there are are two main source of faults in DNN-based programs : the data and the model. For each of these two dimensions (*i.e.*, data and model), there can be errors both at the conceptual and implementation levels, which makes fault location very challenging. Approaches that rely on tweaking variables and watching signals from execution are generally ineffective because of the exponential increase of the number of potential faults locations. In fact, if there are n potential faults related to data, and m potential faults related to the model, we have $n \times m$ total possible faults locations when data is fed in the model. This number grows further when the model is implemented as code, since more types of faults can be introduced in the program at that stage. Systematic and efficient testing practices are necessary to help DL engineers detect and correct faults in DL programs. In this section, we discuss the challenges and issues that could encounter DL engineers when realizing and testing their DL systems.

3.1.1 Data Engineering: Challenges and Issues

In the following, we present potential issues that can occur in data, both at design and implementation levels.

Conceptual issues Once data is gathered, cleaning data tasks are required to ensure that the data is consistent, free from redundancy and given a reliable starting point for statistical learning. Common cleaning tasks include: (1) removing invalid or undefined values (*i.e.*, Not-a-Number, Not-Available), duplicate rows, and outliers that seems to be too different

from the mean value); and (2) unifying the variables' representations to avoid multiple data formats and mixed numerical scales. This can be done by data transformations such as normalization, min-max scaling, and data format conversion. This pre-processing step allows to ensure a high quality of raw data, which is very important because decisions made on the basis of noisy data could be wrong. In fact, recent sophisticated DL models endowed by a high learning capacity are highly sensitive to noisy data [23]. This brittleness makes model training unreliable in the presence of noisy data, which often results in poor prediction performances [24]. A good illustration of this issue is provided by McDaniel et al. [25], who showed that the nonlinearity and high learning capacity of DNN models allow them to construct boundary decision that conforms more tightly to their training data points. In the absence of carefully crafted regularization techniques, these DNNs tend to overfit their training data instead of learning robust high-level representations.

Conventional ML algorithms(*e.g.*, logistic regression, support vector machine, decision tree, etc...) require feature engineering model methods and considerable domain expertise to perform the extraction of a vector of semantically useful features from raw data that allow the DL algorithm detect relevant patterns and learn an explainable mapping function.

In fact, the performance of these machine learning algorithms depends heavily on the representation of the data they are given. Pertinent features that describe the structures inherent in the data entities need to be selected from a large set of initial variables in order to eliminate redundant or irrelevant ones. This selection is based on statistical techniques such as correlation measures and variance analysis. Afterwards, these features are encoded to particular data structures to allow feeding them to a chosen DL model that can handle the features and recognize their hidden related patterns in an effective way. The identified patterns represent the core logic of the model. Changes in data (*i.e.*, the input signals) are likely to have a direct impact on these patterns and hence on the behavior of the model and its corresponding predictions. Because of this strong dependence on data, DL models are considered to be data-sensitive or data-dependent algorithms. It is also possible to define features manually. However, these human-crafted features often require a huge amount of time and effort, and like any informal task, they can be subject to human errors and misunderstandings. A poor selection of features can impact a DL system negatively. Sculley et al. [26] report that unnecessary dependencies to features that contribute with little or no value to the model quality can generate vulnerabilities and noises in a DL system. Examples of such features are : *Epsilon Feature*, which are features that have no significant contribution to the performance of the model, *Legacy Feature*, which are features that lost their added information value on model accuracy improvement when other more rich features are included in the model, or *Bundled Features*, which are groups of features that are integrated to a DL system

simultaneously without a proper testing of the contribution of each individual feature.
In the case of deep learning, feature inference is done automatically through a cascade of multiple representation levels, where higher-level abstractions are defined in terms of lower-level ones. This powerful computational model includes multiple processing layers that allow to uncover unknown patterns in a raw data distribution in order to identify relevant hierarchy of features and lean complex mapping function for a target prediction. However, recent work [27] shows that DNN, without carefully tuned hyperparameters and healthy training process, tend to learn superficial regularity instead of learning high-level abstract features that are less exposed to overfit the data.

Implementation issues To process data as described above, DL engineers implement data pipelines containing components for data transformations, validation, enrichment, summarization, and–or any other necessary treatment. Each pipeline component is separated from the others, and takes in a defined input, and returns a defined output that will be served as input data to the next component in the pipeline. Data pipelines are very useful in the training phase as they help process huge volume of data. They also transform raw data into sets of features ready-to-be consumed by the models. Like any other software component, data pipelines are not immunized to faults. There can be errors in the code written to implement a data pipeline. Sculley et al. [28] identified two common problems affecting data pipelines:

- *Pipelines jungles* which are overly convoluted and unstructured data preparation pipelines. This appears when data is transformed from multiple sources at various points through scraping, table joins and other methods without a clear, holistic view of what is going on. Such implementation is prone to faults since developers lacking a good understanding of the code are likely to make mistakes. Also, debugging errors in such code is challenging.

- *Dead experimental code paths* which happens when code is written for rapid prototyping to gain quick turnaround times by performing additional experiments simply by tweaks and experimental code paths within the main production code. The remnants of alternative methods that have not been pruned from the code base could be executed in certain real world situations and create unexpected results.

Furthermore, DNN model training rely on mini-batch stochastic optimizers to estimate the model's parameters, another data-related component is required in addition to data pipelines, *i.e.*, the Data Loader. This component is responsible of the generation of the batches that are used to assess the quality of samples, during the training phase. It is also prone to errors.

3.1.2 Model Engineering: Challenges and Issues

Next, we present potential issues that can occur in DNN models, both at design and implementation levels.

Conceptual issues One key assumption behind the training process of supervised DL models is that the *training dataset*, the *validation dataset*, and the *testing dataset*, which are sampled from manually labeled data, are representative samples of the underlying problem. Following the concept of Empirical Risk Minimization (ERM), the optimizer allows finding the fitted model that minimizes the empirical risk; which is the loss computed over the training data assuming that it is a representative sample of the target distribution. The empirical risk can correctly approximates the *true risk* only if the training data distribution is a good approximation of the true data distribution (which is often out of reach in real-world scenarios). The size of the training dataset has an impact on the approximation goodness of the true risk, *i.e.*, the larger is a training data, the better this approximation will be. However, manual labeling of data is very expensive, time-consuming and error-prone. Training data sets that deviate from the reality induce erroneous models. The configuration of model includes the hyperparameters that control its capacity (such as number of hidden layers or the depth of decision tree) and also control the behavior of the training algorithm (such as the number of epochs or learning rate). A poor choice of hyperparameters, often results in models with poor performance. For example, inappropriate capacity may result in the model capturing irrelevant information, *i.e.*, noise (overfitting) or missing important data features (underfitting).

Implementation issues DL algorithms are often proposed in pseudo-code formats that include jointly scientific formula and algorithmic rules and concepts. When it comes to implementing DL algorithms, DL engineers sometimes have difficulties understanding these formulas, rules, or concepts. Moreover, because there is no 'test oracle" to verify the correctness of the estimated parameters (*i.e.*, the computation results) of a DL model, it is difficult to detect faults in the learning code. Also, DL algorithms often require sophisticated numerical computations that can be difficult to implement on recent hardware architectures that offer high-throughput computing power. Weyuker [5] identified three distinct sources of errors in scientific software programs such as DL programs.

1. **The mathematical model used to describe the problem**
   A DL program is a software implementation of a statistical learning algorithms that requires substantial expertise in mathematics (*e.g.*, linear algebra, statistics, multivari-

ate analysis, measure theory, differential geometry, topology) to understand its internal functions and to apprehend what each component is supposed to do and how we can verify that they do it correctly. Non-convex objectives can cause unfavorable optimization landscape and inadequate search strategies. Model mis-specifications or a poor choice of mathematical functions can lead to undesired behaviors. For example, many DL algorithms require mathematical optimizations that involve extensive algebraic derivations to put mathematical expressions in closed-form. This is often done through informal algebra by hand, to derive the objective or loss function and obtain the gradient. Generally, we aim to adjust model parameters following the direction that minimizes the loss function, which is calculated based on a comparison between the algorithmic outputs and the actual answers. Like any other informal tasks, this task is subject to human errors. The detection of these errors can be very challenging, in particular when randomness is involved. Errors in stochastic computations can persist indefinitely without detection, since some of them may be masked by the distributions of random variables and may require writing customized statistical tests for their detection.

To avoid overfitting, DL engineers often add a regularization loss (*e.g.*, norm L_2 penalty on weights). This regularization term which has a simple gradient expression can overwhelm the overall loss; resulting in a gradient that is primarily coming from it. Which can make the detection of errors in the real gradient very challenging. Also, non-deterministic regularization techniques, such as dropout in neural network can cause high-variance in gradient values and further complicate the detection of errors, especially when techniques like numerical estimation are used.

2. **The program written to implement the computation**
   The program written to implement a mathematical operation can differ significantly from the intended mathematical semantic when complex optimization mechanisms are used. Nowadays, most DL programs leverage rich data structures (*e.g.*, data frames) and high performance computing power to process massive data with huge dimensionality. The optimization mechanisms of these DL programs is often solved or approximated by linear methods that consists of regular linear algebra operations involving for example multiplication and addition operations on vectors and matrices. This choice is guided by the fact that high performance computers can leverage parallelization mechanisms to accelerate the execution of programs written using linear algebra. However, to leverage this parallelism on Graphics Processing Unit (GPU) platforms for example, one has to move to higher levels of abstraction. The most common abstractions used in this case are tensors, which are multidimensional arrays with some related operations.

Most DL algorithms nowadays are formulated in terms of matrix-vector, matrix-matrix operations, and tensor-based operations (to extract a maximum of performance from the hardware). Also, DL models are more and more sophisticated, with multiple layers containing huge number of parameters each. For such models, the gradient which represents partial derivatives that specify how the loss function is altered through individual changes in each parameter is computed by grouping the partial derivatives together in multidimensional data structures such as tensors, to allow for more straight forward optimized and parallelized calculations. This large gap between the mechanics of the high performance implementation and the mathematical model of one DL algorithm makes the translation from scientific pseudo-code to highly-optimized program difficult and error-prone. It is important to test that the code representations of these algorithms reflect the algorithms accurately.

3. **Features of the environment such as round-off error**
   The computation of continuous functions such as gradient on discrete computational environments like a digital computer incur some approximation errors when one needs to represent infinitely many real numbers with a finite number of bit patterns. These numerical errors of real numbers' discrete representations can be either overflow or underflow. An overflow occurs when numbers with large magnitude are approximated as $+\infty$ or $-\infty$, which become not-a-number values if they are used for many arithmetic operations. An underflow occurs when numbers near zero are rounded to zero. This can cause the numerical instability of functions such as division (*i.e.*, division by a zero returns not-a-number value) or logarithm (*i.e.*, logarithm of zero returns $-\infty$ that could be transform into not-a-number by further arithmetic).
   Hence, it is not sufficient to validate a scientific computing algorithm theoretically since rounding errors can lead to the failure of its implementation. Rounding errors on different execution platforms can cause instabilities in the execution of DL models if their robustness to such errors is not handled properly. Testing for these rounding errors can help select adequate mathematical formulations that minimize their effect.

When implementing DL programs, developers often rely on third-party libraries for many critical operations. These libraries provide optimized implementations of highly intensive computation functions, allowing DL developers to leverage high-performance computing resources with GPU support. However, a misuse of these libraries can result in faults that are hard to detect. For example, a copy-paste of a routine that creates a neural network layer, without changing the corresponding parameters (*i.e.*, weights and biases variables) would result in a network where the same parameters are shared between two different lay-

ers, which is problematic. Moreover, this error can remain in the code unnoticed for a long time. Which is why it is utterly important to test that configuration choices do not cause faults or instabilities in DL programs. In the following we explain the three main categories of libraries that exist today and discuss of their potential misuse.

- **High-level DL libraries:** A high-level DL library emphasizes the ease of use through features such as state-of-the-art supervised learning algorithms, unsupervised learning algorithms, and evaluation methods (*e.g.*, confusion matrix and cross-validation). It serves as an interface and provides a high level of abstraction, where DL algorithms can be easily configured without hard-coding. Using such library, DL developers can focus on converting a business objective into a DL-solvable problem. However, DL developers still need to test the quality of data and ensure that it is conform to the requirements of these built-in functions, such as input data formats and types. Moreover, a poor configuration of these provided algorithms could result in unstable or misconceived models. For example, choosing a sigmoid as activation functions can cause the saturation of neurons and consequently, slowing down the learning process. Therefore, after finishing the configuration, developers need to set up monitoring routines to watch internal variables and metrics during the execution of the provided algorithms, in order to check for possible numerical instabilities, or suspicious outputs.

- **Medium-level DL libraries:** Medium-level libraries provide machine learning or deep learning routines as ready-for-use features, such as numerical optimization techniques, mathematical function and automatic differentiation capabilities, allowing DL developers to not only configure the pre-defined DL algorithms, but also to use the provided routines to define the flow of execution of the algorithms. This flexibility allows for easy extensions of the DL models using more optimized implementations. However, this ability to design the algorithm and its computation flow through programming variables and native loops increases the risk of faults and poor coding practices, as it is the case for any traditional program.

- **Low-level DL libraries:** Contrary to high level and medium level libraries, this family of libraries do not provide any pre-defined DL feature, instead, they provide low level computations primitives that are optimized for different platforms. They offer powerful N-dimensional arrays, which are commonly used in numerical operations and linear algebra routines for a high level of efficiency. They also help with data processing operations such as slicing and indexing. DL developers can use these libraries to build a new DL algorithm from scratch or highly-optimized implementations of particular algorithms for specific contexts or hardwares. However, this total control

on the implementation is not without cost. Effective quality assurance operations such as testing and code reviews are required to ensure bug free implementations. DL developers need strong backgrounds in mathematics and programming to be able to work efficiently with these low level libraries.

The amount of code that is written when implementing a DL program depends on the type of DL library used. The more a developer uses high-level features from libraries, the more he has to write glue code to integrate incompatible components, which are putted together into a single implementation. Sculley et al. [28] observed that the amount of this glue can account for up to 95% of the code of certain DL programs. This code should be tested thoroughly to avoid faults.

3.2 Research Trends in DL System Testing

In this section, we analyze research works that proposed testing techniques for ML programs. We organize the testing techniques in two categories based on the intention behind the techniques, *i.e.*, techniques that aim to detect conceptual and implementation errors in data, and techniques that focus on ensuring correct conception and implementation of ML models. In each of these categories, we divide the techniques in sub-groups based on the concepts used in the techniques. Next, we discuss the fundamental concepts behind each proposed techniques, explaining the types of errors that can be identified using them while also outlining their limitations.

3.2.1 Approaches that aim to detect conceptual and implementation errors in data

The approaches proposed in the literature to test the quality of data addresses both conceptual and implementation issues, therefore, we discuss these two aspects together in this section. The most common technique used to test the quality of data is the analysis-driven data cleaning that consists of applying analytical queries (*e.g.*, aggregates, advanced statistical analytic, etc.) in order to detect errors and perform adequate transformations.
In this approach, aggregations such as sum or count and central tendencies such as mean, median or mode are used to verify if each feature's distribution matches expectation. For example, one can check that features take on their usual set or range of values, and the frequencies of these values in the data.
After this initial verification, advanced statistical analyses such as hypothesis testing and correlation analysis are applied to verify correlations between pair of features and to assess

the contribution of each feature in the prediction or explanation of the target variable. The benefit of each feature can also be estimated by computing the proportion of its explained variance with respect to the target output or by assessing the resulting accuracy of the model when removing it in prior to the fitting process. Besides, when assessing the contribution of each feature to the model, it is recommended to take into account the added inference latency and RAM usage, more upstream data dependencies, and additional expected instability incurred by relying on that feature. It is important to consider whether this cost is worth paying when traded off against the provided improvement in model quality.

Recent research work by Krishnan et al. [29] remarked that these aggregated queries can sometimes diminish the benefits of data cleaning. They observed that cleaning small samples of data often suffices to estimate results with high accuracy. They also observed that the power of statistical anomaly detection techniques rapidly deteriorates in the high-dimensional feature-spaces.

This comes from the fact that the aforementioned analysis-driven data cleaning operations require data queries in order to calculate the aggregate values and correlation measures. Indeed, performing many data queries and cleaning operation on the entire dataset could be impractical with huge amount of training datasets that likely contain dirty records. Moreover, ML developers often face difficulties to establish the data cleaning process. To address these issues, Krishnan et al. proposed ActiveClean [23], an interactive data-cleaning framework for ML models that allows ML developers to improve the performance of their model progressively as they clean the data. The framework has an embedding process that firstly samples a subset of likely dirty records from training data using a set of optimizations, which includes importance weighting and dirty data detection. Secondly, the ML developer is interactively invited to transform or remove each data instance from the selected batches of probably dirty data. Finally, the framework updates the model's parameters and continues the training using partially cleaned data. This process is repeated until no potential dirty instances could be detected. With ActiveClean, developers are still responsible for defining data cleaning operations. The framework only decides where to apply these operations. Recently, Krishnan et al. [30] proposed a full-automated framework, BoostClean, to establish a pipeline of data transformations that allow cleaning efficiently the data in order to fit well the model. BoostClean finds automatically the best combination of dirty data detection and repair operations by leveraging the available clean test labels in order to improve model accuracy. It selects this ensemble from extensible libraries : (1) pre-populated general detection functions, allow identifying numerical outliers, invalid and missing values, checking whether a variable values match the column type signature, and detecting effectively text errors in string-valued and categorical attributes using word embedding; (2) a pre-populated set of

simple repair functions that can be applied to records identified by a detector's predicate, such as impute a cell value with one central tendency (*i.e.*, mean, median and mode value), or discard a dirty record from the dataset. Thus, the boosting technique, which combines a set of weak learners and estimates their corresponding weights to spawn a single prediction model, is applied to solve the problem of detecting the optimal sequence of repairs that could best improve the ML model by formulating it as an ensembling problem. It consists of generating a new model trained on input data with new additional cleaned features and selecting the best collection of models that collectively estimate the predict label. Krishnan et al. evaluated their proposed framework on 8 ML datasets from Kaggle and the UCI repository which contained real data errors. They showed that BoostClean can increase the absolute prediction accuracy by 8-9% over the best non-ensemble alternatives, including statistical anomaly detection and constraint-based techniques.

By automating the selection of cleaning operations, BoostClean significantly simplifies the data cleaning process, however, this framework is resource consuming as it requires the use of multiple models and boosting techniques. Moreover, the evaluation of the embedded cleaning process results on new datasets is challenging because the creation of the cleaning data pipeline is driven by pure statistical analysis.

Hynes et al. [31] inspired by code linters, which are well-known software quality tools, introduced data linter to help ML developers track and fix issues in relation to data cleaning, data transformation and feature extraction. The data linter helps reduce the human burden by automatically generating issues explanations and building more sophisticated human-interactive loop processing. First, it inspects errors in training datasets such as scale differences in numerical features, missing or illegal values (*e.g.*, NaN), malformed values of special string types (*e.g.*, dates), and other problematic issues or inefficiencies discussed in Section 3.1. The inspection relies on data's summary statistics, individual items inspection, and column names given to the features. Second, given the detected errors and non-optimal data representations, it produces a warning, a recommendation for how to transform the feature into a correct or optimal feature, and a concrete instance of the lint taken directly from the data. Data linter guides its users in their cleaning data and features engineering process through providing actionable instructions of how individual features can be transformed to increase the likelihood that the model can learn from them. The main strength of this tool resides in the semi-automated data engineering process and the fact that it can be applied to all statistical learning models and several different data types. The proposed data linter has the ability to infer semantic meaning/intent of a feature based on its metadata as a complement to statistical analysis with the aim of providing specific and comprehensible feature engineering recommendations to ML developers. For example, using a data linter, a developer can

automatically discern whether a string or a numeric data represents a zip code or not. An information that is difficult to obtain by simply inspecting raw values.
In addition to that, a Data loader is often required for systems that rely on mini-batch stochastic optimizers to estimate the model's parameters. To test the reliability of this component, the following best practices are often used: (1) Shuffling of the dataset before starting the batches generation. This action is recommended to prevent the occurrence of one single label batch (*i.e.*, sample of data labeled by the same class) which would negatively affect the efficiency of mini-batch stochastic optimizers in finding the optimal solution. In fact, a straightforward extraction of batches in sequence from data ordered by label or following a particular semantic order, can cause the occurrence of one single label batch. (2) Checking the predictor/predict inputs matching. A random set of few inputs should be checked to verify if they are correctly connected to their labels following the shuffle of data; (3) reduce class imbalance. This step is important to keep the class proportions relatively conform to the totality of training data.

3.2.2 Approaches that aim to detect conceptual and implementation errors in ML models

As discussed in Section 3.1, errors in ML models can be due to conceptual mistakes when creating the model or implementation errors when writing the code corresponding to the model. In the following, we discuss testing approaches that focus on these two aspects, separately.

Approaches that aim to detect conceptual errors in ML models

Approaches in this category assume that the models are implemented into programs without errors and focus on providing mechanisms to detect potential errors in the calibration of the models. These approaches can be divided in two groups: black-box and white-box approaches [32]. Black-box approaches are testing approaches that do not need access to the internal implementation details of the model under test. These approaches focus on ensuring that the model under test predicts the target value with a high accuracy, without caring about its internal learned parameters. White-box testing approaches on the other hand take into account the internal implementation logic of the model. The goal of these approaches is to cover a maximum of specific spots (*e.g.*, neurons) in models. In the following, we elaborate more on approaches from these two groups.

A: Black-box testing approaches for ML models The common denominator to black-box testing approaches is the generation of adversarial data set that is used to test the ML models. These approaches leverage statistical analysis techniques to devise a multidimensional random process that can generate data with the same statistical characteristics as the input data of the model. More specifically, they construct generative models that can fit a probability distribution that best describes the input data. These generative models allows to sample the probability distribution of input data and generate as many data points as needed for testing the ML models. Using the generative models, the input data set is slightly perturbed to generate novel data that retains many of the original data properties. The advantage of this approach is that the synthetic data that is used to test the model is independent from the ML model, but statistically close to its input data. One prominent type of generative model are Generative adversarial networks (GANs) [33]. Over the last decade, GANs have proved to outperform other types of generative models such as variational auto-encoders (VAEs) [34] or restricted Boltzmann machines (RBMs) [35]. GANs are generative learning approaches that assemble two separate DNNs, a generator and discriminator, with opposing or adversarial objectives. The discriminator is trained to distinguish between original and synthetic samples, while, the generator is trained to fool the discriminator through realistic synthetic data. Although GANs have been successfully trained on high dimensional continuous data to generate diverse and realistic examples, they cannot be applied when original datasets are discrete such as words, characters, or bytes. Recently, the boundary-seeking GAN (BGAN) [36] has been proposed as a unified generative learning method that enables the generation of discrete data and improves the stability of the training on continuous data. A key characteristic of BGANs is that they reinterpret the generator objective as a distance instead of divergence. In fact, most common difference measures used by GANs come from the family of f-divergences (such as the KL-divergence), while BGANs define a new objective for the generator that represents a simple distance between log-probabilities of discriminator's outputs. This way, the generator learns to minimize this distance in order to make the discriminator outputs both real and fake labels with the same probability. GANs and BGANs have been successfully used to change data contexts and infer realistic situations from samples of training data, in order to test the robustness of the representations learned by models [37].

When ML models are involved in security-sensitive applications, their robustness against non obvious and potentially dangerous manipulation of inputs should be tested. ML models can be vulnerable to malicious adaptive adversaries that manipulate their input data; causing them to diverge from the training data. In fact, the training data cannot cover the entire inputs or features space, especially, when dealing with high-dimensional data. Which makes

it difficult to approximate the real decision boundaries, since ML algorithms learn by minimizing the empirical risk on training data. This phenomenon becomes more nuanced for sophisticated models with high capacity, such as DNNs, which are able to draw complex decision boundaries with original shapes in order to conform more tightly to the data points and reduce the error as much as possible. This complexity creates vulnerabilities that can be exploited by adversaries. Adversarial machine learning is an emerging area where various techniques are being used to find adversarial regions where models exhibit erroneous behaviors. Several mechanisms exist for the creation of adversarial examples. Goodfellow et al. [38] proposed a gradient-based perturbation that makes small modifications to drive the mutated input into ambiguous and vulnerable regions of the input space. It consists of computing, first, the gradient of the loss with respect to the input, to determine a suitable direction of changes. Then, it perturbs inputs towards this direction using a prefixed step size to control the magnitude of the perturbations. Engstrom et al. [39] introduced affine transformations (such as translations and rotations) that can fool DNN-based vision models. Contrary to gradient-based perturbations, those affine transformations do not require any complicated optimization technique, but they are easy to find using a few black-box queries on the target model. Recent research work [40] shows that adversarial examples can be found through simple guess-and-check of naturally-occurring situations related to the application domain. This requires human effort and time, to produce realistic inputs. Recent results [41] [42] have shown the effectiveness of adversarial examples in misleading DNN-based systems; corrupting their integrity. They show how malicious inputs can be added to the training data to improve the resilience of the models. The erroneous behavior of a model can also be used to understand the root cause of some security vulnerabilities and generate countermeasures as shown in [42].

To help protect DNNs against adversarial attacks, Wang et al. [43] leveraged model mutation testing to detect adversarial examples at runtime. Using the following mutation operators, *i.e.*, Gaussian Fuzzing (GF), Weight Shuffling (WS), Neuron Switch (NS), and Neuron Activation Inverse (NAI), they randomly mutated DNNs and compute the Label Change Ratio (LCR) of both adversarial data and genuine data. They observed that adversarial examples have significantly higher LCR under model mutation than original examples. Leveraging this observation, they proposed to use LCR to decide at runtime whether an input is adversarial or genuine. Model mutation testing can also be used to assess the quality of adversarial synthetic data and generate adversarial examples that are more subtle and effective for adversarial training.

Conceptually, ML models identify high-level abstract features and patterns in the data and encode semantic information about how these set of features or patterns are related to the

target outcome. In practice, ML models are exposed to multiple potential issues that can prevent them from learning the optimal mapping function, including inappropriate configuration (*e.g.*, poor regularization) and implementation errors in the ML algorithm. It is challenging to identify the resulting incorrect behaviors of the model because the input space is large and manually labeled test data can only cover a small fraction of this space, leaving many corner-cases untested. Inspired by adversarial evasion attacks, researchers have started proposing testing approaches for ML models based on automated partial oracles that are inferred from applying data transformations, including imperceptible perturbations and affine transformations, on an initial human created oracle. Thanks to these automated partial oracles, it is now possible to generate large sets of test inputs automatically [44] [45].

One major limitation of these black-box testing techniques is the representativeness of the generated adversarial examples. In fact, many adversarial models that generate synthetic images often apply only tiny, undetectable, and imperceptible perturbations, since any visible change would require manual inspection to ensure the correctness of the model's decision. This can result in strange aberrations or simplified representations in synthetic datasets, which in turn can have a hidden knock-on effects on the performance of a ML model when unleashed in a real-world setting. These black-box testing techniques that rely only on adversarial data (ignoring the internal implementation details of the models under test) often fail to uncover different erroneous behaviors of the model, even after performing a large number of tests. This is because the generated adversarial data often fail to cover the possible behaviors of the model adequately. An outcome that is not surprising given that the adversarial data are generated without considering information about the structure of the models. To help improve over these limitations, ML researchers have developed the white-box techniques described below, which use internal structure specificities to guide the generation of more relevant test cases.

B: White-box testing approaches for ML models Pei et al. proposed DeepXplore [44], the first white-box approach for systematically testing deep learning models. DeepXplore is capable of automatically identifying erroneous behaviors in deep learning models without the need of manual labelling. The technique makes use of a new metric named *Neuron Coverage (NC)*, which estimates the amount of neural network's logic explored by a set of inputs. This neuron coverage metric computes the rate of activated neurons in the neural network. It was inspired by the code coverage metrics used for traditional software systems. The approach circumvent the lack of a reference oracle, by using differential testing. DeepXplore leverages a group of similar deep neural networks that solve the same problem. Perturbations are introduced in inputs data to create many realistic visible differences (*e.g.*, different light-

ing, occlusion, etc.) and automatically detect erroneous behaviors of deep neural networks under these circumstances. Applying differential testing to deep learning with the aim of finding a large number of difference-inducing inputs while maximizing neuron coverage can be formulated as a joint optimization problem. DeepXplore performs gradient ascent to solve efficiently this optimization problem using the gradient of the deep neural network with respect to the input. Its objective is to generate test data that provokes a different behavior from the group of similar deep neural networks under test in order to ensure a high neuronal coverage. We noticed that domain-specific constraints are added to generate data that is valid and realistic. In the end of the testing process, the generated data are kept for future training, to have more robustness in the model.

Ma et al. [46] generalized the concept of *neuron coverage* by proposing DeepGauge, a set of multi-granularity testing criteria for Deep Learning systems. DeepGauge measures the testing quality of test data (whether it being genuine or synthetic) in terms of its capacity to trigger both major function regions as well as the corner-case regions of DNNs(Deep Neural Networks). It separates DNNs testing coverage in two different levels.
At the neuron-level, the first criterion is k-multisection neuron coverage, where the range of values observed during training sessions for each neuron are divided into k sections to assess the relative frequency of returning a value belonging to each section. In addition, the authors insist on the need for test inputs that are enough different from training data distribution to cover rare neurons' outputs. They introduced the concept of neuron boundary coverage to measure how well the test datasets can push activation values to go above and below a pre-defined bound (*i.e.*, covering the upper boundary and the lower boundary values). Their design intentions are complementary to Pei et al. in the sense that the k-multisection neuron coverage could potentially help to cover the main functionalities provided by DNN. However, the neuron boundary coverage could relatively approximate corner-cases DNN's behaviors. The neuron-level coverage criteria are : (1) *K-multisection Neuron Coverage (KMNC)*: the ratio of covered k-multisections of neurons; (2)*Neuron Boundary Coverge (NBC)*: the ratio of covered boundary region of neurons; (3)*Strong Neuron Activation Coverage (SNAC)*: the ratio of covered hyperactive boundary region.
At the layer-level, the authors leveraged recent findings that empirically showed the potential usefulness of discovered patterns within the hyperactive neurons that are often activated, *i.e.*, they render outputs larger than the prefixed threshold. On the one hand, each layer allows DNN to characterize and identify particular features from input data and its main function is in large part supported by its top active neurons. Therefore, regarding the effectiveness in discovering issues, test cases should go beyond these identified hyperactive neurons in each layer. On the other hand, DNN provide the predicted output based on pattern recognized

from a sequence features, including simple and complex ones. These features are computed by passing the summary information through hidden layers. Thereby, the combinations of top hyperactive neurons from different layers characterize the behaviors of DNN and the functional scenarios covered. Intuitively, test data sets should trigger other patterns of activated neurons in order to discover corner-cases behaviors. The layer-level coverage criteria are : (1)*Top-k Neuron Coverage (TKNC)*: the ratio of neurons in top-k hyperactived state on each layer; (2)*Bottom-k Neuron Coverage (BKNC)*: the ratio of neurons in top-k hypoactived state on each layer.
In their empirical evaluation, Ma et al. showed that DeepGauge scales well to practical sized DNN models (*e.g.*, VGG-19, ResNet-50) and that it could capture erroneous behavior introduced by four state-of-the-art adversarial data generation algorithms (*i.e.*, Fast Gradient Sign Method (FGSM) [38], Basic Iterative Method (BIM) [47], Jacobian-based Saliency Map Attack (JSMA) [48] and Carlini/Wagner attack (CW) [41]). Therefore, a higher coverage of their criteria potentially plays a substantial role, in improving the detection of errors in the DNNs. These positive results show the possibility to leverage this multi-level coverage criteria to create automated white-box testing frameworks for neural networks.

Indeed, the DNN coverage criteria allows assessing the diversity of DNN neuronal behavior triggered by testing inputs. This can be used to guide the generation of synthetic data that exhibit novel neurons' state and enhance the diversity of inputs. Evaluations [44] [46] have shown that the proposed DNN coverage criteria are correlated with the adversarial examples, which indicates that the unfamiliar inputs are more likely to trigger erroneous behaviors. Therefore, increasing the neuronal coverage promotes the diversity of the generated inputs, which likely results in more effective test cases.

Building on the pioneer work of Pei et al., Tian et al. proposed DeepTest [45], a tool for automated testing of DNN-driven autonomous cars. In DeepTest, Tian et al. expanded the notion of neuron coverage proposed by Pei et al. for CNNs (Convolutional Neural Networks), to other types of neural networks, including RNNs (Recurrent Neural Networks). Moreover, instead of randomly injecting perturbations in input image data, DeepTest focuses on generating realistic synthetic images by applying realistic image transformations like changing brightness, contrast, translation, scaling, horizontal shearing, rotation, blurring, fog effect, and rain effect, etc. They also mimic different real-world phenomena like camera lens distortions, object movements, different weather conditions, etc. They argue that generating inputs that maximize neuron coverage cannot test the robustness of trained DNN unless the inputs are likely to appear in the real-world. They provide a neuron-coverage-guided greedy search technique for efficiently finding sophisticated synthetic tests which capture different realistic image transformations that can increase neuron coverage in a self-driving car DNNs.

To compensate for the lack of a reference oracle, DeepXplore used differential testing. However, DeepTest leverages metamorphic relations (MRs) to create a test oracle that allows it to identify erroneous behaviors without requiring multiple DNNs or manual labeling. Tian et al. defined metamorphic relations between the car behaviors across the proposed image-based transformations. Since it is hard to specify in prior the correct steering angle for each transformed image, they assume that the predicted angle for a transformed scene driving is correct if it varies from its genuine one to less than λ times the mean squared error produced by the original data set. λ is a configurable parameter that helps to strike a balance between the false positives and false negatives.

DeepRoad [37] continued the same line of work as DeepTest, designing a systematic mechanism for the automatic generation of test cases for DNNs used in autonomous driving cars. Data sets capturing complex real-world driving situations is generated and Metamorphic Testing is applied to map each data point into the predicted continuous output. However, DeepRoad differentiates from DeepTest in the approach used to generate new test images. DeepRoad relies on a Generative Adversarial Network (GAN)-based method to provide realistic snowy and rainy scenes, which can hardly be distinguished from original scenes and cannot be generated by DeepTest using simple affine transformations. Zhang et al. argue that DeepTest synthetic image transformations, such as adding blurring/fog/rain effect filters, cannot simulate complex weather conditions. They claim that DeepTest's produced road scenes may be unrealistic, because simply adding a group of lines over the original images cannot reflect the rain condition or mixing the original scene with the scrambled "smoke" effect does not simulate the fog. To solve this lack of realism in generated data, DeepRoad leveraged a recent unsupervised DNN-based method (*i.e.*, UNIT) which is based on GANs and VAEs, to perform image-to-image transformations. UNIT can project images from two different domains (*e.g.*, a dry driving scene and a snowy driving scene) into a shared latent space, allowing the generative model to derive the artificial image (*e.g.*, the snowy driving scene) from the original image (*e.g.*, the dry driving scene). Evaluation results show that the generative model used by DeepRoad successfully generates realistic scenes, allowing for the detection of thousands of behavioral inconsistencies in well-known autonomous driving systems.

Despite the relative success of DeepXplore, DeepTest, and DeepRoad, in increasing the test coverage of neural networks, Ma et al. [49] remarked that the runtime state space is very large when each neuron output is considered as a state, which can lead to a combinatorial explosion. To help address this issue, they proposed DeepCT, which is a new testing method that adapts combinatorial testing (CT) techniques to deep learning models, in order to reduce the testing space. CT [50] has been successfully applied to test traditional software requiring

many configurable parameters. It helps to sample test input parameters from a huge original space that are likely related to undetected errors in a program. For example, the t-way combinatorial test set covers all the interactions involving t input parameters, in a way that expose efficiently the faults under the assumption of a proper input parameters' modeling. In DeepCT, K-way CT is adapted to allow for selecting effectively samples of neuron interactions inside different layers with the aim of decreasing the number of test cases. Given the initial test data sets. DeepCT generates some DNN-related K-way coverage criteria using constraint-based solvers (*i.e.*, by linear programming using the CPLEX solver [51]). Next, a new test data is generated by perturbing the original data within a prefixed value range, while ensuring that previously generated CT coverage criteria are satisfied on each layer. Ma et al. [49] conducted an empirical study, comparing the 2-way CT cases with random testing in terms of the number of adversarial examples detected. They observed that random testing was ineffective even when a large number of tests were generated. In comparison, DeepCT performed well even when only the first several layers of the DNN were analyzed, which shows some usefulness for their proposed CT coverage criteria in the context of adversarial examples detection and local-robustness testing.
However, even though solvers like CPLEX represents the state-of-the-practice, their scalability remains an issue. Hence, the effectiveness of the proposed DeepCT approach on real-world problems using large and complex neural networks remains to be seen.

Sun et al. [52] examined the effectiveness of the neuron coverage metric introduced by DeepXplore and report that a 100% neuron coverage can be easily achieved by a few test data points while missing multiples incorrect behaviors of the model. To illustrate this fact, they showed how 25 randomly selected images from the MNIST test set yield a close to 100% neuron coverage for an MNIST classifier. Thereby, they argue that testing DNNs should take into account the semantic relationships between neurons in adjacent layers in the sense that deeper layers use previous neurons' information represented by computed features and summarize them in more complex features. To propose a solution to this problem, they adapted the concept of Modified Condition/Decision Coverage (MC/DC) [53] developed by NASA. The concepts of "decision" and "condition" in the context of DNN-based systems correspond to testing the effects of first extracted less complex features, which can be seen as potential factors, on more complex features which are intermediate decisions. Consequently, they specify each neuron in a given layer as a decision and its conditions are its connected input neurons from the previous layer.
They propose a testing approach that consists of a set of four criteria inspired by MC/DC and a test cases generator based on linear programming (LP). As an illustration of proposed criteria, we detail their notion of *(Sign-Sign(SS)* coverage, which is very close to the spirit of

MC/DC. Since the neurons' computed outputs are numeric continuous values, the *SS* coverage could not catch all the interactions between neurons in successive layers. Since the changes observed on a neuron's output can be either a sign change or a value change, they added three additional coverage criteria to overcome the limitations of *SS*: Value-Sign Coverage, Sign-Value Coverage, and Value-Value Coverage. These three additional criteria allow detecting different ways in which changes in the conditions can affect the models' decision.

Sun et al. [54] also applied concolic testing [55] to DNNs. Concolic testing combines concrete executions and symbolic analysis to explore the execution paths of a program that are hard to cover by blind test cases generation techniques such as random testing. The proposed adaptation of concolic testing to DNNs leverages state-of-the-art coverage criteria and search approaches. The authors first formulate an objective function that contains a set of existing DNN-related coverage requirements using Quantified Linear Arithmetic over Rationals (QLAR). Then, their proposed method incrementally finds inputs data that satisfy each test coverage requirement in the objective. Iterating over the set of requirements, the concolic testing algorithm finds the existing test input that is the closest data point to satisfy the current requirement following an evaluation based on concrete execution. Relying on this found instance, it applies symbolic execution to generate a new data point that satisfies the requirement and adds it to the test suite. The process finishes by providing a test suite that helps reaching a satisfactory level of coverage. To assess the effectiveness of their proposed approach, they evaluated the number of adversarial examples that could be detected by the produced test suites.

Guo et al. [56] observed that traditional coverage-guided fuzzing techniques (which consist in randomly mutating input data to generate new inputs that may increase the coverage criteria and eventually help uncover erroneous model's behavior) often fail to find relevant input data because of the huge size of the input data space. Finding corner-cases regions through a blind mutation of an input corpus is hard and expensive when the input data space is huge. To solve this problem, they proposed DL Fuzz, a differential fuzzing testing framework where the maximization of neuron coverage and prediction difference between original and mutated inputs is formulated as a joint optimization problem, that can be solved efficiently using gradient-based ascent optimizers.

Starting with a random list of data inputs, DL Fuzz iteratively computes the gradient of the optimization objective with respect to each input data and applies it as perturbations to the input data in order to obtain new mutated test input. Then, the $L2$ distance between the original input and the mutated input is evaluated to guarantee that the performed perturbation is invisible to humans and that the two inputs share the same prediction out-

put. Finally, all the generated inputs that contributed to increase the neuron coverage are kept into a maintained input data corpus. DL Fuzz expands DeepXplore Neuron's coverage measure with additional neuron selection strategies such as : the selection of neurons that are frequently activated during model executions, and the prioritization of neurons with top high-value weights. The assumption being that these neurons may have a bigger influence on the model's behavior.

Kim et al. [57] proposed a fine-grained test adequacy metric, named Surprise Adequacy (SA) that quantifies how much surprising a given input is to the DNN under test with respect to the training data. The intuition behind this criterion is that effective test inputs should be sufficiently surprising compared to the training data. The surprise of an input is quantitatively measured as behavioural differences observed in a given input relatively to the training data. Kim et al. defined two concrete instances of their SA metric, given DNN's activations trace (AT) that represent a vector of neurons' activations : (1) Likelihood-based SA which uses Kernel Density Estimation (KDE) to estimate the probability density of each activation in AT, and computes the relative likelihood of new input's activation values with respect to estimated densities; and (2) Distance-based SA which uses the Euclidean distance between a given input's AT and the nearest AT of training data in the same class. Kim et al. evaluated their SA metrics using state-of-the-art adversarial generators (*i.e.*, FGSM, BIM, JSMA, or CW), DeepXplore gradient-based generator, and DeepTest combined transformation generator. They report that SA can capture the relative surprise of synthetic inputs. Using SA to generate test inputs, they improved the classification accuracy of DNN's models against adversarial examples by up to 77.5%.

Approaches that aim to detect errors in ML code implementations

Given the stochastic nature of most ML algorithms and the absence of oracles, most existing testing techniques are inadequate for ML code implementations. As a consequence, the ML community have resorted to numerical testing, property-based testing, metamorphic testing, mutation testing, coverage-guided fuzzing testing, and proof-based testing techniques to detect issues in ML code implementations. In the following, we present the most prominent techniques.

Numerical-based testing: Finite-difference techniques Most machine learning algorithms are formulated as optimization problems that can be solved using gradient-based optimizers, such as gradient descent or L-BFGS (*i.e.*, Limited-memory Broyden Fletcher Goldfarb Shanno algorithm). The correctness of the objective function gradient that are

computed with respect to the model parameters, is crucial. In practice, developers often check the accuracy of gradients using a finite difference technique that simply consists in performing the comparison between the analytic gradient and the numerical gradient. The analytic gradient can be either manually inferred and hard-coded by the developer or automatically generated by the automatic differentiation component of a DL library. However, the numerical gradient can be estimated relying on finite difference approximations. Nevertheless, because of the increasing complexity of models' architectures, this technique is prone to errors. To help improve this situation, Karpathy [58] have proposed a set of heuristics to help detect faulty gradients.

a)Use of the centered formula Instead of relying on the traditional gradient formula, Karpathy recommends using the centered formula from Equation 3.1 which is more precise. The taylor expansion of the numerator indicates that the centered formula has an error in the order of $O\left(h^2\right)$, while the standard formula has an error of $O\left(h\right)$.

$$\frac{df\left(x\right)}{dx} = \frac{f\left(x+h\right) - f\left(x-h\right)}{2h} \tag{3.1}$$

b) Use of relative error for the comparison As mentioned above, developers perform gradient checking by computing the difference between the numerical gradient f'_n and the analytic gradient f'_a. This difference can be seen as an absolute error and the aim of the gradient checking test is to ensure that it remains below a pre-defined fixed threshold. With deep neural networks, it can be hard to fix a common threshold in advance for the absolute error. Karpathy recommends fixing a threshold for relative errors. So, for a deep neural network's loss that is a composition of ten functions, a relative error 3.2 of $1 \exp^{-2}$ might be acceptable because the errors build up through backpropagation. Conversely, an error of $1 \exp^{-2}$ for a single differentiable function likely indicates an incorrect gradient.

$$\frac{|f'_a - f'_n|}{max(|f'_a|, |f'_n|)} \tag{3.2}$$

c) Use of double precision floating point Karpathy recommends to avoid using single precision floating point to perform gradient checks, because it often causes high relative errors even with a correct gradient implementation.

d) Stick around active range of floating point To train complex statistical models, one needs large amounts of data. So, it is common to opt for mini-batch stochastic gradient descent and to normalize the loss function over the batch. However, if the back-propagated

gradient is very small, additional divisions by data inputs count will yield extremely smaller vales, which in turn can lead to numerical issues. As a solution to this issue, Karpathy recommends computing the difference between values with minimal magnitude, otherwise one should scale the loss function up by a constant to bring the loss to a denser floats range, ideally on the order of 1.0 (where the float exponent is 0).

e) Use only few random data inputs The use of fewer data inputs reduces the likelihood to cross kinks when performing the finite-difference approximation. These kinks refer to non-differentiable parts of an objective function and can cause inaccuracies in the numerical gradients. For example, let's consider an activation function $ReLU$ that has an analytic gradient at the zero point, *i.e.*, it is exactly zero. The numerical gradient can compute a non-zero gradient because it might cross over the kink and introduce a non-zero contribution. Karpathy strongly recommends to perform the gradient checking for a small sample of the data and infer the correctness of gradient for an entire batch, because it makes this sanity-check fast and more efficient in practice.

f) Check only few dimensions Recent statistical models are more and more complex and may contain thousands parameter with millions of dimensions. The gradients are also multi-dimensional. To mitigate errors, Karpathy recommends checking a random sample of the dimensions of the gradient for each separate model's parameter, while assuming the correctness of the remaining ones.

g) Turn off dropout/regularization penalty Developers should be aware of the risk that the regularization loss overwhelms the data loss and masks the fact that there exists a large error between the analytic gradient of data loss function and its numerical one. Such circumstance can result in a failure to detect an incorrect implementation of the loss function or its corresponding gradient using the finite-difference approximation technique. Karpathy recommends turning off regularization and checking the data loss alone, and then the regularization term, independently. Moreover, recent regularization techniques applied to deep neural networks such as dropout, induce a non-deterministic effect when performing gradient check. Thereby, to avoid errors when estimating the numerical gradient, it is recommended to turn them off. An alternative consists in forcing a particular random seed when evaluating both gradients.

Property-based testing Property-based testing is a technique that consists in inferring the properties of a computation using the theory and formulating invariants that should

be satisfied by the code. Using the formulated invariants, test cases are generated and executed repeatedly throughout the computation to detect potential errors in the code. Using property-based testing, one can ensure that probability laws hold throughout the execution of a model. For example, one can test that all the computed probability values are non-negatives. For a discrete probability distribution, as in the case of a classifier, one can verify that the probabilities of all events add up to one. Also, marginalization can be applied to test probabilistic classifiers. Karpathy [58] recommend the verification of the following properties to test DL systems.

Initial random loss : when training a neural network classifier, turn off the regularization by setting its corresponding strength hyperparameter to zero and verify that initial softmax loss is equal $-\log\left(\frac{1}{N_c}\right)$ with N_c: number of label classes. One expect a diffuse probability of $\frac{1}{N_c}$ for each class.

Overfitting a tiny dataset : Keeping the regularization term turned off, extract a sample portion of data, (one or two examples inputs from each class) in order to ensure that the training algorithm can achieve efficiently zero loss. Breck et al. also recommend watching the internal state of the model on small amounts of data with the aim of detecting issues like numerical instability that can induce invalid numeric values like NaNs or infinities.

Regularization role : increase the regularization strength and check if the data loss is also increasing.

Another testing technique that shares the same philosophy as property-based testing is metamorphic testing.

Metamorphic testing Murphy et al. [59] introduced metamorphic testing to ML in 2008. They defined several Metamorphic Relationships (MRs) that can be classified into six categories (*i.e.*, additive, multiplicative, permutative, invertive, inclusive, and exclusive). The performed transformations include adding a constant value to numerical attributes; multiplying numerical attributes by a constant value; permuting the order of inputs; reversing the order of inputs; removing a portion of inputs; adding additional instances. In fact, the defined MRs are quite generic and can be applied for different types of machine learning algorithms (ranking, supervised classifiers, unsupervised clustering, etc...). Murphy et al. [60] manually assessed the effectiveness of their defined MRs on three well-known ML applications: Marti-Rank, SVM-Light (Support Vector Machine with a linear kernel), and PAYL [60], and concluded that they can be used to test ML applications efficiently. Xie et al. [61] proposed

MRs specialized for testing the implementations of supervised classifiers. The MRs are based on five types of transformations: (1) application of affine transformations to input features;(2) permutation of the order of labels or features; (3) addition of uninformative and informative new features; (4) duplication of some training instances; and (5) removal of arbitrary classes or instances. The evaluation of these new MRs was conducted on the implementation of k-Nearest Neighbors (kNN) and Naive Bayesian (NB) from Weka [62]. Using the MRs, the authors were able to uncover defects in the implementation of NB provided by Weka. In 2011, Xie et al. [63] further evaluated their MRs using mutation testing. They found that their proposed MRs were able to reveal 90% of the injected faults in Weka (injected by MuJava [64]).

Recent research works [65] have investigated the application of metamorphic testing to more complex machine learning algorithms such as SVM with non-linear kernel and deep residual neural networks (ResNET). For SVM, they applied transformations such as : changing features or instances orders, linear scaling of the features. For deep learning models, since the features are not directly available, they proposed to normalise or scale the test data, or to change the convolution operations order. They used MutPy(*i.e.*, a tool for python code mutation) to mutate the training program and simulate implementation bugs. Their MRs were able to find 71% of the injected faults.

Mutation testing Ma et al. [66] proposed DeepMutation that adapts mutation testing [16] to DNN-based systems with the aim of evaluating the test data quality in terms of its capacity to detect faults in the programs. To build DeepMutation, Ma et al. defined a set of source-level mutation operators to mutate the source of a ML program by injecting faults. These operators allow injecting faults in the training data (using data mutation operators) and the model training source code (using program mutation operators). After the faults are injected, the ML program under test is executed, using the mutated training data or code, to produce the resulting mutated DNNs. The data mutation operators are intended to introduce potential data-related faults that could occur during data engineering (*i.e.*, during data collection, data cleaning, and–or data transformation). Program mutation operators' mimic implementation faults that could potentially exist in the model implementation code. These mutation operators are semantic-based and specialized for DNNs' code. Training models from scratch following source-level mutations is very time-consuming since advanced deep learning systems require often tens of hours and even days to learn the model's parameters. Moreover, manually designing specific mutation operators using information about faults occurring in real-world DNNs systems is challenging, since it is difficult to imagine and simulate all possible faults that occur in real-world DNNs. To circumvent the cost of multiple re-execution

and fill in the gap between real-world erroneous models and mutated models, the authors define model-level mutation operators to complement source-level mutation operators. These operators directly change the structure and the parameters of neural network models to scale the number of resulted mutated models for testing ML programs in an effective way, and for covering more fine-grained model-level problems that might be missed by only mutating training data and–or programs. Once the mutated models are produced, the mutation testing framework assesses the effectiveness of test data and specify its weaknesses based on evaluation metrics related to the killed mutated models count. DL engineers can leverage this technique to improve data generation and increase the identification of corner-cases DNN behaviors.

Coverage-Guided Fuzzing Odena and Goodfellow [67] developed a coverage-guided fuzzing framework specialized for testing neural networks. Coverage-guided fuzzing has been used in traditional software testing to find critical errors. For DL code, the fuzzing process consists of handling an input corpus that evolves through the execution of tests by applying random mutation operations on its contained data and keeping only interesting instances that allow triggering new program behavior. Iteratively, the framework samples an input instance from testing data corpus and mutates it in a way that constrains the difference between the mutated input and its original version to have a user-configurable L_∞ norm. This ensures that mutated instances remain associated with the same label as the original input. Then, it checks whether the corresponding state vector is meaningfully different from the previous ones using a fast approximate nearest neighbor algorithm based on a pre-specified distance. Each input data that is relatively far from the existing nearest neighbor, is added to the test cases set. The framework, entitled TensorFuzz, is implemented to test TensorFlow-based neural network models automatically. The effectiveness of the proposed testing approach was assessed using three known issues in neural networks' implementations. Results show that TensorFuzz surpasses random search in : (1) finding NaNs values in neural network models trained using numerical instable cross-entropy loss, (2) generating divergences in decision between the original model that is encoded with 32-bit floating point real and its quantized version that is encoded with only 16-bit, and (3) surfacing undesirable behavior in character level language RNN models.

Proof-based testing Selsam et al. [68] proposed to formally specify the computations of ML programs and construct formal proofs of written theorems that define what it means for the programs to be correct and error-free. Using the formal mathematical specification of a ML program and a machine-checkable proof of correctness representing a formal certificate

that can be verified by a stand-alone machine without human intervention, ML developers are able to find errors in the code. Using their proposed approach, Selsam et al. analyzed a ML program designed for optimizing over stochastic computation graphs, using the interactive proof assistant *Lean*, and reported it to be bug-free. However, although this approach allows to detect errors in the mathematical formulation of the computations, it cannot help detect execution errors due to numerical instabilities, such as the replacement of real numbers by floating-point with limited precision.

3.3 Discussion

In the following, we detail the identified gaps in the literature that represent the motivations of our proposed testing approaches, which exploit these missing spots.

3.3.1 Search-based approach for testing DNN-based models

As detailed in white-box software testing methods for DNNs 3.2.2, TensorFuzz [67] and DeepHunter [69] implement coverage-guided fuzzing that consists in continuously applying mutations on corpus of inputs, triggering uncovered DNN's states, in order to enhance DNN's coverage and generate diverse synthetic inputs. The main limitation of coverage-guided fuzzing is that it relies on random fuzzing of original inputs, through a blind generation, hoping to maximize the coverage criteria. Besides, DeepXplore [44] and DLFuzz [56] formulate the generation of inputs triggering uncovered neurons as an optimization problem that can be solved using a differential process through gradient-based data transformations. Although those gradient-based transformations allowed for efficient data perturbations, they lack the applicability for several complex data transformations. As an example from computer vision applications, gradient-based transformation can perform imperceptible pixels' perturbations, but it cannot infer optimal parameters for affine transformations such as rotation or translation.

To improve over these limitations of existing white-box testing approaches, we intend to propose a search-based approach for testing DNN-based models. Indeed, search-based approach formulates the coverage criteria as an objective function to optimize through inputs generation, but it relies on metaheuristic-based optimizers that are gradient-free and can be applied to explore the parameters' space of a wide variety of transformations thanks to their flexibility (as it does not required prior assumptions). Indeed, we think that metaheuristic-based optimizers could be an effective solution for automating the generation on synthetic testing inputs for DNN models, since they show promising results in the security evaluation for DNN-based systems. The security evaluation is based on attacking the DNN-based systems

through constructing malicious inputs crafted to mislead the system, and hence, corrupt its integrity. Therefore, security experts analyze the potential causes of adversarial examples and the future countermeasures that might mitigate them. Sharif et al. [70] leveraged the Particle Swarm Optimization (PSO) algorithm in an impersonation attack on the face recognition model *face++*. This attack consists in adversary seeking to find a face recognized as another different face. They use PSO to explore a sub-space of perturbations restricted to pairs of glasses with smooth color variations, to ensure the realism of the attack. Results show that the PSO-based eyeglass frames printing could successfully fool *face++*. Alzantot et al. [71] recently reported about GenAttack, a gradient-free optimizer that uses Genetic Algorithms (GAs) to apply imperceptible perturbations on inputs. Alzantot et al. conducted a series of experimentation and report that GenAttack can successfully fool state-of-the-art image recognition models with significantly fewer queries.
Even if former white-box testing approaches do not aim to generate inputs increasing the odds that the model fools and misclassifies them, they show that increasing the neuronal coverage can enhance the diversity of inputs and revealing erroneous DNN's behaviors. Therefore, our search-based testing approach aims to generate automatically diverse test inputs through triggering the maximum of internal DNN logic, with the objective of uncovering corner-cases behaviors and potential defects.

3.3.2 Property-based testing for DNN training programs

As described in property-based testing paragraph 3.2.2, the properties leveraged to test the DL system are quite coarse-grained and allow DL engineers to validate if the training program does the minimum work or not. For stochastic algorithm, Roger et al. [72] infer invariants' assertions from the properties of MCMC samplers to detect errors in their concrete implementations. Using the formulated invariants as derived oracle test, they develop test cases that should be executed repeatedly throughout the computation to detect inconsistencies in their stochastic program. Indeed, DNN training program is written in a modular way using the computational routines provided by DL libraries as ready-for-use features. Assuming that those provided features are bug-free, the training program may have bugs and errors in its principal component that defines the main flow of the program and set up the configuration of the DNN. Thus, inferring verification routines based on invariants and heuristics, which are emerged from long DL research experiences and explained by fundamental statistical and mathematical concepts, could indicate the presence of inappropriate configuration or implementation errors. Hence, the automation of those routines can be a derived test oracle for DNN training programs.

3.4 Chapter Summary

Recently, researchers have started to develop new testing techniques to help DL engineers debug and test DL programs. In this chapter, we explain the main sources of faults in a DL program. Next, we review testing techniques proposed in the literature to help detect these faults both at the model and implementation levels; explaining the context in which they can be applied as well as their expected outcome. We also identify some gaps in the literature related to the testing of DL programs. In the following chapters, we propose our solutions to fill the identified gaps in the literature and improve the software quality of DL programs.

CHAPTER 4 TFCHECK: A TENSORFLOW LIBRARY FOR DETECTING TRAINING ISSUES IN NEURAL NETWORK PROGRAMS

Given the rise of deep learning adoption in safety-critical systems, researchers have proposed many model-based testing approaches to help improve the reliability of DL applications. These approaches consist of evaluating the model performance in terms of prediction ability regarding manually labeled data and–or automatically generated data. The objective is to check for inconsistencies in the behavior of the model under test; so whenever inconsistencies are uncovered, the training set is augmented with miss-classified test data in order to help the model learn the properties of corner-cases on which it performed poorly. This process is repeated until a satisfactory performance is achieved. These proposed ML testing approaches assume that the ML model is trained adequately, *i.e.*, the training program is bug-free and numerically stable. They also assume that the training algorithm and the model hyper-parameters are optimal in the sense that the model has the adequate capacity to learn the patterns needed to perform the targeted task, adequately. However, bugs may exist in ML training code and these bugs can invalidate some of these assumptions. In fact, Zhang et al. [8] investigated bugs in neural networks training programs built on TensorFlow [73] and reported multiple bug occurrences. They also identified five challenges related to bug detection and localization. One of these challenges is coincidental correctness. A coincidental correctness occurs when a bug exists in a program, but by coincidence, no failure is detected. Coincidental correctness can be caused by undefined values such as NaNs and Infs, induced by numerically unstable functions. Finding training input data to expose these issues can be challenging. Also, a bug in the implementation of a neural network can result in saturated or untrained neurons that do not contribute to the optimization, preventing the model from learning properly. Furthermore, when a neural network makes mistakes on some adversarial data, gathering more data is not a panacea. The neural network model may not have the appropriate capacity to learn patterns from these noisy data or may miss good regularization to avoid overfitting the noise. Detecting all these issues requires effective verification mechanisms.

In this chapter, we introduce a list of verification routines to help DL engineers detect and correct errors in their ML training programs. Since these verification routines can be difficult to adopt for beginners and since their manual application can be very time-consuming, we developed TFCheck, a TensorFlow library that implements the proposed verification routines to help DL engineers monitor and debug TensorFlow-based training programs. To assess the effectiveness of TFCheck at detecting DNN training issues automatically, we conducted a

case study using real-world, mutants, and synthetic training programs. The results of this case study show that using TFCheck, DL engineers can successfully detect training issues in a DNN code implementation.

Chapter Overview Section 4.1 presents some common issues experienced by developers when training ML models alongside some verification mechanisms. Section 4.2 describes the structure of our TensorFlow based library TFCheck and its utilization. Section 4.3 reports about a case study aimed at evaluating TFCheck. Finally, Section 4.5 concludes the chapter.

4.1 DNN model training pitfalls

Many issues can prevent a proper training of a DNN, which prevent the fitting process from finding the best-fitted model. Indeed, DL engineers rely on statistical learning measures to assess the goodness of fit of a model to its training data. If these measures such as loss value or error rate are less than particular thresholds, it indicates an occurrence of bugs in the training program, a poor configuration, or a misconception of the model. However, some training issues can be subtle and hard to detect because the resulting DNN model can provide acceptable or even high performance measures. Below, we elaborate on some of these training issues, explaining their potential causes. Then, we propose mechanisms for their automatic detection. For simple training issues, our verification routines could detect their presence earlier during the training, which allow DL engineers to save time. Regarding subtle issues, we argue that our verification routines can warn about their existence through smooth heuristic-based checks with tunable thresholds. In this section, the training issues are organized in three groups based on the main problematic component of the DNN model: Parameter-related issues, Activation-related issues, and Optimization-related issues.

4.1.1 Parameters related issues

DNN parameters represent the weights and biases of its layers, which values are unknown, but they are estimated during the training process. We discuss three different types of issues that are related to these parameters and their corresponding verification routines that could indicate the non-optimality of the approximated values.

Untrained Parameters. The most basic and common issue in relation to model's parameters occurs when DL engineers forget to connect the different branches of a DNN or build the layer with incorrect inputs. A DNN with less layers than necessary can still be trained and it would likely converge. However, it will not have enough learning capacity to reach

high prediction ability. Therefore, it is important to check that all defined parameters are modified appropriately.

Verification Routine: This issue can be detected by storing each actual tensor's parameter and comparing it with the tensor's parameters obtained after the execution of training operations, in order to make sure that each parameter is getting optimized and differs from its default value.

Poor Weight Initialization. The initialization of weights values should be done carefully. Following the recommendations of DL researchers, weights need to be initialized in a way that breaks the symmetry between hidden neurons of the same layer, because if hidden units of the same layer share the same input and output weights, they will compute the same output and receive the same gradient, hence performing the same update and remaining identical, thus wasting capacity. In other words, there is no source of asymmetry between neurons if their weights are initialized to be the same. The initial random weights values should be small enough to neither diverge while the gradients explode, nor very close to zero, so the gradients vanish quickly.

Verification Routine: One can verify that there are significant differences between the parameter's values by computing the variance of each parameter's values and checking if it is not equal or very close to 0.

Parameters' Values Divergence. Weights can diverge to $+/-\infty$ if the initial values or the learning rate are too high and there is a lack of–or–insufficient regularization, because the back-propagation updates process would push the weights to becoming higher and higher, until reaching ∞ values (this is caused by overflow rounding precision). In addition to that, biases also risk divergence in the sense that they can become huge in certain situations where features could not explain enough the predicted outcome and might not be useful in differentiating between available classes. This problem occurs more likely when the distribution of classes is very imbalanced.

Verification Routine: One can verify that parameters' values are not diverging by computing their 75th percentiles (upper quartiles) and verifying that they are less than a predefined threshold, during each training iteration.

Parameters' Unstable Learning. The use of several hidden layers can cause unstable learning situations such as: (1) Layer's parameters changing rapidly in an unstable way; preventing the model from learning relevant features. The intuition is that the parameters

are a part of the estimated mapping function, so we risk overfitting the current processed batch of data when we try to adapt strongly the parameters in order to fit this batch; (2) Layer's parameters changing slowly; making it difficult to learn useful features from data. These learning issues are strongly related to the unstable gradient phenomenon. Indeed, we mentioned that the computation of the gradients with respect to earlier layers contains a product of terms from all the later layers. This backpropagation of gradient tend to establish significantly different learning speeds in the layers. To ensure stability, advanced mechanisms or adequate hyperparameters choices are needed to establish relatively similar learning speed with respect to all neural network's layers, through balancing out the computed gradients with respect to the layers' parameters as much as possible. In fact, the parameters' updates are computed by an optimizer using not only the gradient and the learning rate. It also contains other hyper-parameters such as momentum coefficients to provide adaptive gradient steps. As suggested by Bottou [74], it is useful to compare the magnitude of parameter gradients to the magnitude of the parameters themselves with the aim of verifying that the magnitude of parameter updates over batches represent something like 1% of the magnitude of the parameter, not 50% or 0.001%.

Verification Routine: This issue can be detected by comparing the magnitude of parameters' gradients to the magnitude of the parameters themselves. More specifically, following Bottou's recommendation to keep the parameter update ratio around 0.01 (*i.e.*, -2 on base 10 logarithm), one can compute the ratio of absolute average magnitudes of these values and verify that this ratio doesn't diverge significantly from the following predefined thresholds :

$$-4 < \log_{10}\left(\frac{mean(abs(updates))}{mean(abs(parameters))}\right) < -1$$

Using this verification routine, one can recognize the layers where updates seem to be unstable or stalled.

4.1.2 Activation related issues

Activation represents the intermediate computation that introduces non-linearity to filter the information computed by the previous layer. We detail the following three categories of problems that are related to these activations. In addition, we propose the routines that could be automated in order to detect their presence.

Activations out of Range. Activation functions have specific output ranges. One common mistake is to implement a mathematical function or to apply a wrong existing function

for a layer assuming inaccurate outputs ranges. It is necessary to make sure that activation's range of values stays consistent with what is expected theoretically from their corresponding function (*e.g.*, sigmoid's outputs are between $[0, 1]$ and tanh's outputs are between $[-1, 1]$).

Verification Routine: To detect activation out of range issues, one can verify if the activation values exceed known theoretical range of values. This verification routine is particular useful to find computation mistakes when a new activation function is implemented from scratch.

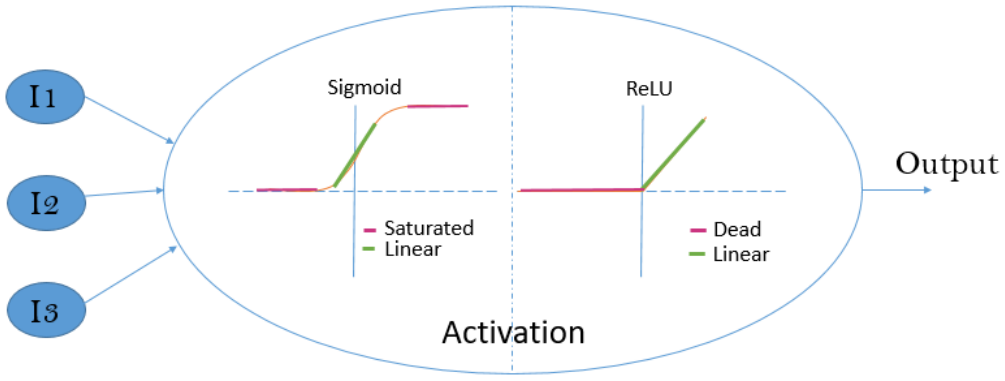

Figure 4.1 Illustration of Sigmoid Saturation and Dead ReLU

Neuron Saturation. Bounded activation functions with a sigmoidal curve, such as sigmoid or tanh, exhibit smooth linear behaviour for inputs within the active range and become very close to either the lower or the upper asymptotes for relatively large positive and negative inputs. The phenomenon of neuron saturation occurs when a neuron returns almost only values close to the asymptotic limits of the activation functions (see Figure 4.1). In such case, any change on weights of this neuron will not have a noticeable influence on the output of the activation function. As a result, the training process may stagnate with stable parameters, preventing the training algorithm from refining them.

Verification Routine: To detect neuron saturation issues in NNs, one can use the single-valued saturation measure ρ_B proposed by Rakitianskaia and Engelbrecht [75]. This measure is computed using the outputs of an activation function and is applicable to all bounded activation functions. It is independent of the activation function output range and allows a direct statistical measuring of the degree of saturation between NNs. ρ_B is bounded and easy to interpret: it tends to 1 as the degree of saturation increases and tends to zero otherwise. It contains a single tunable parameter, the number of bins B that converges for $B \geq 10$, *i.e.*, it means splitting the interval of activation outputs into B equal sub-intervals. Thus, $B = 10$ can be used without any further tuning. Given a bounded activation function g, ρ_B

is computed as the weighted mean presented in Equation 4.1.

$$\rho_B = \frac{\sum_{b=1}^{B} |\bar{g}'_b| N_B}{\sum_{b=1}^{B} N_B} \tag{4.1}$$

Where, B is the total number of bins, $\bar{g}'_b$ is the scaled average of output values in the bin b within the range $[-1, 1]$, N_b is the number of outputs in bin b. Indeed, this weighted mean formula turns to a simple arithmetic mean when all weights are equal. Thus, if $\bar{g}'_b$ is uniformly distributed in $[-1, 1]$, the value of ρ_B will be close to 0.5, since absolute activation values are considered, thus all $\bar{g}'_b$ values are squashed to the $[0, 1]$ interval. For a normal distribution of $\bar{g}'_b$, the value of ρ_B will be smaller than 0.5. The higher the asymptotic frequencies of $\bar{g}'_b$, the closer ρ_B will be to 1.
This verification routine can be automated by storing for each neuron its last O outputs' values in a buffer of a limited size. Then, it proceeds by computing its ρ_B metric based on those recent outputs. If the neuron corresponding value tends to be 1, the neuron can be considered as saturated. After checking all neurons for saturation, one can compute the ratio of saturated neurons per layer to alert DL engineers about layers with saturation ratios that surpass a predefined threshold.

Dead ReLU. ReLU stands for rectified linear unit, and is currently the most used activation function in deep learning models, especially CNNs. In short, ReLU is linear (identity) for all positive values, and zero for all negative values. Contrary to other bounded activation functions like sigmoid or tanh, ReLU does not suffer from the saturation problem, because the slope does not saturate when x gets large and the problem of vanishing gradient is less observed when using ReLU as activation function. However, the fact that they are null for all negative values increases the risk of "dead ReLUs". A ReLU neuron is considered "dead" when it always outputs zero (see Figure 4.1). Such neurons do not have any contribution in identifying patterns in the data nor in class discrimination. Hence, those neurons are useless and if there are many of them, one may end up with completely frozen hidden layers doing nothing. This problem often occurs when whether the learning rate is too high or there is a large negative bias. Recent ReLU variants such as Leaky ReLU and ELU are recommended as good alternatives when lower learning rates do not solve the problem.

Verification Routine: A given neuron is considered to be dead if it only returns zero or almost zero value as output. Hence, dead ReLUs issues can be detected by storing the last obtained outputs for each neuron in a limited size buffer and checking if almost all the stored values are zero or closer to zero. Whenever we find zero or close to zero values, we can mark the neuron as dead. By computing the ratio of dead neurons per layer, one can detect layers

with a high number of dead neurons with respect to a predefined threshold aimed at warning DL engineers about layers containing a high number of dead neurons.

4.1.3 Optimization related issues

The optimization process involved in DNN training aims to minimize the loss, *i.e.*, empirical risk with respect to training data, which is generally solved using an iterative gradient descent optimizer. Next, we present six kinds of issues that are related to the optimization, while providing verification routines to catch them earlier.

Unable to fit a small sample. Given a small data set, a DNN model should be able to fit it without any issue, because even simple models should be able to fit a tiny subset of data, otherwise, there is likely a poor configuration or a software defect. In such circumstance, there is no need to try training the DNN on large data for multiple days running.

Verification Routine: One can verify that the optimization mechanism is working well by performing the training process over a controlled sample data set. Following the recommendations of Karpathy [58], one can turn off regularization techniques and train the DNN on a tiny subset of data (*i.e.*, a few data points for each class), and then verify that it achieves zero loss under these circumstances. A failure to achieve this performance would signal a minimization problem.

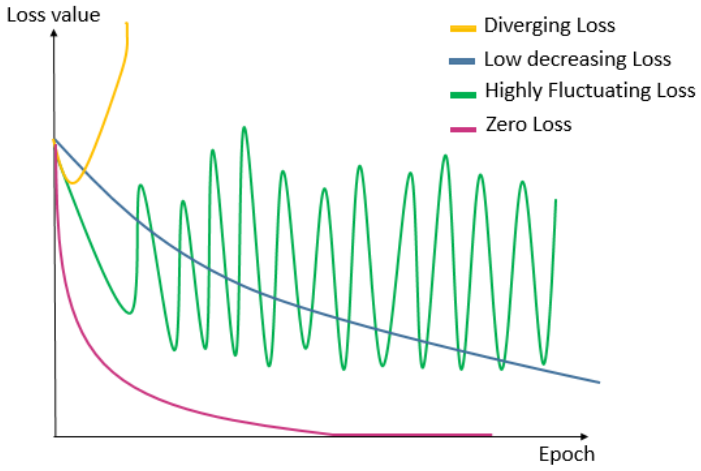

Figure 4.2 Illustration of loss minimization issues

Zero loss. The training process of DNNs relies on data minimization to adapt optimally the DNN parameters, in a way that fits well the training distribution, while being capable

of generalization on unseen data provided from the same distribution. When the loss minimization process reaches a zero value (see Figure 4.2), it is an indication that the model overfits the training data, and that its prediction ability could be low on unseen data. The optimization process aims to learn the high-level patterns in order to ensure a good differentiation between the classes, but it could never get all outcomes correct with a confidence of 100% for each predicted value (*i.e.*, a zero loss value).

Verification Routine: To detect zero loss issues, Goodfellow et al. [76] recommend to verify the training program directly, as it is (including regularization), checking that it sufficiently fits the controlled data sample, while keeping its loss different from zero.

Slow or Non decreasing loss. After multiple training iterations, if the model still have non-decreasing or a very slow decreasing loss (see Figure 4.2), it is likely to display a poor performance because of a low learning capacity. This low learning capacity issue could be due to a deficiency of the optimizer or an implementation mistake.

Verification Routine: One can verify that the loss is smoothly decreasing by computing the decreasing loss rate using Equation 4.2 and verifying that this rate is mostly higher than a predefined threshold.

$$loss\_rate = \frac{current\_loss\_value}{previous\_loss\_value} \tag{4.2}$$

Diverging loss. Due to a high learning rate or inadequate loss function, the minimization process can transformed into a maximization process, in the sense that the loss value can start increasing wildly at a certain point (see Figure 4.2).

Verification Routine: To detect diverging loss issues, one should keep track of the loss to make sure that it doesn't start increasing after a certain number of iterations. This verification can be done automatically by initializing and updating a reference variable containing *lowest_loss_value*. Using this variable, one can compute the absolute loss rate (following Equation 4.3) to check if the last computed losses are diverging from this previously obtained minimum value.

$$abs\_loss\_rate = \frac{current\_loss\_value}{lowest\_loss\_value} \tag{4.3}$$

Loss fluctuations. A highly fluctuating loss during the training phase of a DNN may indicate an inappropriate learning rate that prevents the convergence of gradient-based optimizers to the minimum and keeps jumping it with relatively large updates. Besides, mini-batch stochastic gradient-based optimizers risk encountering a fluctuating loss when the batch size is relatively small, so the loss would be noisy without an evolution trend during multiple

iterations.

Verification routine. Fluctuations in the loss functions can be identified by the presence of successive loss increases and decreases (see Figure 4.2); so the loss rate (from Equation 4.2) alternates between values that are either higher and lower than 1.

Unstable Gradients. Due to poor weights initialization and bad choices of hyperparameters, DNNs can be exposed to the unstable gradient problem, which could manifest in the form of vanishing or exploding gradients as described below.

(1) *Vanishing Gradient:* In this case, the gradient tends to have smaller values when it is back-propagated through the hidden layers of the DNN. This causes the gradient to be almost zero in the earlier layers and consequently would be transformed to undefined values such as Not-a-Number (NaN) caused by underflow rounding precision during discrete executions on hardware. The problem of vanishing gradient can leads to the stagnation of the training process and eventually causing a numerical instability. As an illustration, we take the example of a DNN configured to have sigmoid as activation function and a randomly initialized weight using a Gaussian distribution with a zero mean and a unit standard deviation. The sigmoid function returns a maximum derivative value of 0.25 (*i.e.*, the derivative of sigmoid when the input is equal to zero) and the absolute value of the weights product is less than 0.25 since they belong to a limited range between $[-1, 1]$. Generally, the gradient of a given hidden layer in a NN would be computed by the sum of products of all the gradients belonging to the deeper layers and the weights assigned to each of the links between them. As a result, the deeper the gradient is back-propagated over the hidden layers, the more there are products of several terms that are less or equal to 0.25. Hence, it is apparent that earlier hidden layers (*i.e.*, closer to the input layer) would have very less gradient and would be almost stagnant with less weights changes during the training process.

(2) *Exploding Gradient:* The exploding gradient phenomenon can be encountered when, inversely, the gradient with respect to the earlier layers diverges and its values become huge. As a consequence, this could result in the appearance of $-/+\infty$ values. Returning to the previous DNN example, the same NN can suffer from exploding gradients in case the parameters are large in a way that their products with the derivative of the sigmoid keep them on the higher side until the gradient value explodes and eventually becomes numerically unstable.

Verification Routine: One can detect unstable gradient issues by examining the first and the last gradients, which correspond respectively to the last, and the first hidden layer (to make the computation cheaper). More specifically, one can check if their 25^{th} percentiles (lower quartiles) are not NaN or less than a minimum threshold and their 75^{th} percentiles (upper quartiles) are not Inf or higher then a maximum threshold; this could indicate the

presence of diminishing or exploding values through backpropagation. Softer tests can be performed by comparing the ratio of absolute average magnitudes of the last gradient by the first gradient to some predefined tunable thresholds (see the Inequation 4.4), indicating the maximum tolerable growth or decline of back-propagated gradients.

$$min\_threshold < \log_{10}\left(\frac{mean(abs(last\_gradients))}{mean(abs(first\_gradient))}\right) < max\_threshold \tag{4.4}$$

4.2 TFCheck : A TensorFlow library for Testing Neural Networks programs

To assist users in debugging and testing the code implementation of their TF NN programs, we developed a library implementing the issues detection routines described in Section 4.1. We choose to focus on TF programs because of the popularity of TF in the ML community. Nevertheless, the ideas implemented in this library could be adapted for other frameworks. In the following, we describe each module in more details.

4.2.1 TensorFlow-based Implementation

TensorFlow already has an official debugging tool (TFDBG) [77] specialized for TF programs that are difficult to debug with general-purpose debuggers such as Python's pdb because of the TF computation-graph paradigm. Indeed, TFDBG offers features such as inspection of the computational graph, addition of conditional breakpoints and real-time view on internal tensor values of running TF graphs. However, it is not practical for our case since it adds a huge overhead on computation time, as it handles each execution step of the graph, to allow debugging the issues and pinpoint the exact graph nodes where a problem first surfaced. In fact, our verification routines do not require breaking the execution flow of the training program. They need to be instantiated with the values of intermediate computations after each step to validate the satisfaction of defined properties. In addition to that, TFDBG is not popular in the TF community because of its complexity of use. Thus, TF users often rely on summary operations to write graph variables and operations values out and leverage the Tensorboard tool to interactively display those values and the data flow structure of the running computation graph in a user-friendly Web interface [78]. They use named scope to define hierarchy on the nodes in a graph to have names like "dense_layer/weight", "dense_layer/bias", and "dense_layer/Relu". This is useful to collapse the variables from one scope during the visualization of multiple layers. We intend to build our TFCheck library following this established naming convention and the simple use of session object that provide a connection between the Python code and the DAG on execution. Through

this object, we can simply fetch any tensor value and evaluation results, by running it with the tensor name as parameter, using input data required for any prior computation and the names of variables and operations defined by a TF user following the creation of its DAG. TFCheck performs verification routines on the values of tensors fetched before and after running training operations. These verifications do not require to break neither the feed-forward nor the backward pass. Therefore, TFCheck introduces the verification routines before and after training operation runs, while keeping the training pass to be executed in an atomic way.

4.2.2 Setting Up the Testing Session

The testing of a DNN training program can be more complicated than for a traditional programs because of its non-deterministic aspect. Indeed, it is difficult to investigate and detect the training issues when the program exhibits a novel behavior and returns different output for each execution. However, the testing task can be much easier when we guarantee that the ML program always produces the same outputs given the same inputs. To avoid stochastic ML program's execution, we analyzed the different aspects of ML implementations that could result in non-deterministic behaviors.

Randomness. The stochastic nature of the training program is induced by the facts that the model parameters are initialized randomly, mini-batch optimizers select random batches of training data at each iteration until convergence. This results in the final parameters being slightly different between multiple execution times, and recent regularization techniques such as dropout, introduces randomness when it eliminates a number of neurons from training with respect to pre-defined probability.

Parallelism. TF relies heavily on parallelism to achieve high performance through multicores CPU and GPU. Multi-threading execution makes it extremely hard to get perfectly reproducible results. To illustrate this point, suppose that two parallel operations (noted a and b) are executed simultaneously by two threads, depending on how fast each thread runs, operation a may finish before or after operation b. That's not a problem as long as the outputs of these operations are used in a deterministic order, but if they are pushed to the same queue, then the order of the items may change at each single run, and consequently, the downstream operations also changes. Although mathematically the result should be the same using real numbers, program may not return the exact same results since the execution environments perform computation using floats with limited precision as illustrated in

Figure 4.3. During training, these tiny differences will accumulate and generate different outputs in the end. To make ML program deterministic, we first turn off multi-threading execution on CPU and GPU. However, this would be done only for the testing phase since the single thread program execution would be much slower in terms of running time.

To ensure the reproducibility of the DNN training program under test, the *setup()* method in our TFCheck library turns off almost all the potential sources of variability by fixing the seeds for all the computational libraries used, such as TF, Numpy and Python built-in random library. TFCheck can also deactivate the parallelism with GPU depending on the usage of the tester, *i.e.*, whether he prefers more the determinism or the rapidity of execution of the different verification routines in a parallel environment.

4.2.3 Monitoring the Training Program Behavior

TFCheck performs the verifications described in Section 4.1 using values of tensors fetched before running training operations and the results obtained after running the training operations. Thus, given the number of verifications that should be performed once in a while, between the *session.run()* calls, we use the monitored session and hooks mechanism to handle all this additional processing injected in the training program. This allows us to keep the code of our library elegant, maintainable, and easy to extend. To develop and use session hooks in our library, we need to perform two steps:

1. Create one or more Hooks that inherits from the *SessionRunHook* class and implement methods required to perform the check, essentially we implement *before_run* and *after_run* that execute pre-and post-processing to each *session.run()* call.

2. Create a *MonitoredSession* and attach to it the implemented session hooks by setting the hooks parameter to a list of session hooks instances.

TFCheck implements all the verification routines described in Section 4.1 as well as a *Hooks* module that contains the different session hooks for the different verification routines. Those hook objects access values obtained for activations, parameters, and gradients w.r.t parameters, with the aim of applying the verification routines on them. Since the training program runs consecutive *session.run()* calls, most tensors such as activations and gradients do not survive past a single execution of the graph and the parameters of the model such as weights and biases are updated continuously. Therefore, those hook objects generally store the last values of chosen parameters or previously obtained value to enable the execution of the necessary comparisons.

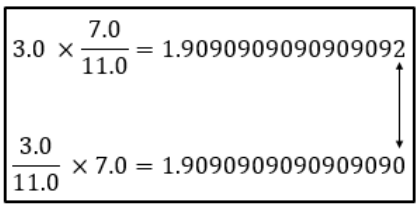

Figure 4.3 Example of result difference caused by operators order change

4.2.4 Logging for any Detected Issue

Once TFCheck finds an issue, it reacts depending on the configuration defined by the tester. The current available configurations are : "log meaningful warning message explaining the encountered issue" or "stop the training process and raise an exception". In fact, the issues, indicating bugs, such as activations out of range or untrained parameters, can be configured to raise an exception. However, the issues, resulting from poor choice of hyperparameters or misconception of the model, can be logged with their corresponding metrics, such as amount of saturated or dead neurons in the layer, to the tester who can make the decision of stopping the training or not and if the training is done, adopting the trained model or not. Logs and exceptions contain meaningful messages including the computed metrics that indicate the presence of the issue such as the amount of dead neurons and the layer name where an issue is spotted alongside its corresponding parameter or gradient. Those messages aim to help testers understand the issues of their training programs and assist them in identifying their root causes.

4.3 Evaluation of TFCheck

To assess the effectiveness of TFCheck at detecting DNN training issues. We replicate the training of 4 buggy TensorFlow programs identified by Zhang et al. [8] and of 7 mutants generated and verified by Dwarakanath et al. [65]. Besides, we create 4 synthetic training codes that imitate known issues to complement the evaluation of TFCheck. Table 4.1 summarizes the results and following, we describe in detail the issues found and the TFCheck verification routines that detected them.

4.3.1 Real-world Training Programs

In the empirical study on TF Bugs [8], We found programs in which the reported bugs are related to an Incorrect Model Parameter or Structure (IPS). We choose this type of bugs because they often manifest themselves during the training phase of the models, and therefore allow for early detection. These bugs are mainly caused by inappropriate modeling

Table 4.1 Overview on the Tested Neural Networks

DNN	Issue	Fired Checks
IPS-4	Inadequate Loss function	UPL[1](slow), Non-decreasing Loss
IPS-5	Inefficient Optimization	UPL(high), Exploding Gradient
IPS-15	Poor weight Initialization	Unbreaking Symmetry, Exploding Weights
IPS-17	High learning rate	UPL(high), Diverging Loss
Mut-29	Incorrect loss function	Zero Loss
Mut-30	Incorrect regularization term	UPL(high), Loss Fluctuations
Mut-31	Incorrect regularization term	UPL(high), Loss Fluctuations
Mut-32	Incorrect regularization term	UPL(high), Loss Fluctuations
Mut-43	Incorrect learning rate schedule	UPL, Vanishing Gradient
Mut-44	Incorrect learning rate schedule	UPL, Vanishing Gradient
Mut-45	Incorrect learning rate schedule	UPL, Vanishing Gradient
Synth-1	Deep NN with Sigmoid	Saturated Neurons
Synth-2	Huge negative biases	Dead Neurons
Synth-3	Disconnected layers	Untrained parameters
Synth-4	Deactivated layers	Activation out of range

configurations that lead to erroneous behaviors during the training phase. Zhang et al. claim that the major symptom of these bugs is low effectiveness, *i.e.*, low accuracy. However, we believe that TFCheck can generate more fine-grained feedbacks whenever such issue is encountered by ML developers; which will help for its early detection and for understanding its root cause.

To verify our hypothesis, we replicate these buggy TF programs. Then, their executions using a monitored training, incorporated with TFCheck hooks allowed us to detect the existing issues in 4 programs.

In IPS 4, the neural network use inadequate loss function that, first, triggers an *unstable parameters learning* issue; because parameters seem to be changing slowly and the model seems unable to learn patterns. Over time, this caused the *non-decreasing of the loss* issue and its stagnation at a relatively high value.

In IPS 5, the predict variable is a continuous variable and the loss function is MSE. However, the sample training data used contained outputs that are relatively big and the use of gradient descent optimizer with a high learning rate (*i.e.*, 0.1) caused the *unstable parameters learning* issue in the sense that weights changed wildly with relatively high update steps, then, the *exploding of gradients* issue occured.

In IPS 15, the weights are poorly initialized in a way that contain a constant value, which is not small enough to avoid diverging through backpropagation of gradients. Therefore, TFCheck displayed the *Unbreaking Symmetry* issue from the first iteration. Then, it triggered

the *exploding weights* issue when one of the DNN's weights starts containing huge values.
In IPS 17, the learning rate was high in a way that caused the *unstable parameters learning* issue, with remarkable bigger update weights ratio and, then, this ended up turning into a *diverging loss* issue.
These results suggest that TFCheck can successfully help DL engineers detect issues in their training programs resulting from misconception or poor choices of their DNN's configuration spots.

4.3.2 Mutants of Training Programs

Dwarakanath et al. [65] created mutants of training programs that represent a TF training program with one bug. They adopted a systematic approach of changing a line of code, at once, in the original source code and repeatedly generated multiple source code files with intentionally induced faults. To perform a large-scale generation, they used the MutPy tool [79], from the Python Software Foundation, to generate the mutants. From their mutants analysis and dataset, we extracted non-crashing mutants that contain a subtle bug in the training code in order to see if TFCheck can trigger adequate warnings for each issue.
In Mutant-29, a random change of the mathematical operator in the loss function causes its deficiency. TFCheck was able to kill this mutant during the testing process when it triggered the *Zero Loss* issue warning, indicating that the loss reached a zero value, which is a suspicious value. In fact, if we continue the training execution, the mutant will even return negative values.
In Mutant-30-31-32, different deformations in the regularization term causes its defectiveness in a way that it becomes a non-regularization term that would guide the optimizer to increase the value of weights in opposite with regularization objective. TFCheck identified this training issue by, first, alerting for the *Unstable parameters learning* issue caused by high weights' updates steps and second, triggering a warning related to the presence of the *high fluctuations on the loss* issue; showing that the optimizer encounters a difficulty finding the minimum under the given circumstances.
In Mutant-43-44-45, some intentionally induced faults in the formula that generates the schedule of different learning rates enhance the weirdness of computed values and the deficiency of the corresponding training session. For this mutant, TFCheck was able to detect those incorrect learning rate schedules through the resulting *unstable parameters learning* verification routine. Besides, TFCheck alerted also for the *vanishing gradient* issue when the gradients became smaller and smaller, to adapt the update step given those high learning rates.
These results show that TFCheck can successfully detect training issues in DNN

programs caused by simple coding mistakes.

4.3.3 Synthetic Training Programs

We complemented the evaluation of TFCheck using synthetic code examples.
For the saturation neurons issue, we constructed a fully connected NN with ten layers including each 100 neurons with Sigmoid as activation function for the MNIST digits classification (*i.e.*, input images 28×28 and 10 predicted classes). This neural network risks the saturation from early training iterations as explained in the work of Glorot and Bengio [80].
For the problem of *dead neurons*, we also implement a ReLU-based fully connected NN that contains a biased initializer that introduces huge negative bias in random neurons. This NN is at risk of dead neurons, since their linear computation could give almost zero or less, because of the huge negative bias.
Inspired from a buggy TF code snippet found in github[2], we developed a training program that contains disconnected layers from others in the computation graph; so they did not depend on the training operation. TFCheck successfully detected these known mistakes (it reported about the existence of parameters that are permanently untrained by the DNN during the training session).
Regarding activation layer issues, we removed the activation function from the layer definition in order to mimic a situation where the outputs of a newly designed activation function are not as expected. TFCheck triggered the *activation are out of their valid range* warning. **These results further reinforce previous findings that TFCheck can successfully detect training issues in NN programs.**

4.4 Discussion

The operations implemented in our proposed library to test and debug the training programs of neural networks can have some side effects that developers should take into account.

- The monitored execution of the training program includes different verification routines that perform calculations and verify conditions on the intermediary computations done by the DNN during a single training iteration. This allows detecting training issues that indicate the presence of bugs in the code or inconsistencies in the training algorithm configuration. As a consequence, the additional pre- and post-computations represent an overhead and, additionally, slow down the training execution of TensorFlow code

[2] `https://gist.github.com/Thenerdstation/1c333b33e587859e37476109b44491c1#file-convnet-py`

because it has been already compiled and optimized to run continuously on the target machine without cut-offs.

- The elaboration of preventive measures to guarantee the reproduction of same results during multiple executions of the training adversely affects the performance of the tests in terms of running time, because it forces the execution to be done sequentially on CPU only. Moreover, the use of fixed seeds could increase the risk of coincidental correctness because the fixed seed controls the stochastic computations and could hinder the detection of issues that did not show up for the particular random values.

Table 4.2 reports the average time required by the execution of the training program when using TFCheck (*i.e.*, the training program is under test) and when TFCheck is not used. The evaluation was done using a standard implementation of the popular convolutional neural network, LeNet.

Table 4.2 Comparison of training time (with/without verification routines)

	Normal	**Under Test**
Average time for one training iteration (seconds)	0.11	0.96

To mitigate the impact of this overhead, we recommend using TFCheck in a testing process as follows:

1. Setting up of the testing process (this step includes fixing of seed and sampling the training data)

2. Execution of the training for a reasonable number of iterations on the selected data. In our experiments, we were able to detect the issue after a few epochs.

3. In case an issue is detected, depending on the user configuration, the testing execution raises an exception or just displays a warning. In both cases, our comprehensive message makes it possible to predict potential causes, and therefore, the DL engineer adjusts the configuration or review some parts of code. Then, he re-iterates the process from the second step to verify if the issue still appears.

This testing workflow can be done multiple times with different sample data and initial seeds, but they are kept the same during the process to ensure the re-detection of the issue found if it is not fixed by the corrections done. There is trade-offs between the testing cost and the testing effectiveness like any other testing approach. This is controlled through choosing the

sample data size and the max number of iterations. Last, once the testing session is over, DL engineer can start training on the whole available data and higher number of epochs with more confidence on its training program quality.

4.5 Chapter Summary

In this chapter, we focus on testing DL training programs. We introduce a list of verification routines that can be used by DL engineers to detect issues in DNN training programs. We implemented these verification routines in an automated testing library for DNN training programs developed using TensorFlow. We assessed the effectiveness of our library at debugging DL training programs through a case study with synthetic mutants and real world neural network programs. Obtained results show that using our library, developers can successfully detect training issues in their DNN program implementations.

CHAPTER 5 DEEPEVOLUTION: A SEARCH-BASED TESTING APPROACH FOR DNN MODELS

Although DNN-based software have proven useful and effective in many fields and applications, their widespread adoption in large-scale and critical systems such as self-driving cars or aircraft collision-avoidance systems sheds the light on the importance and urgency of improving the quality assurance of DNN models in production. A human-crafted test oracle is hindered by the high cost of collecting and labeling data. Thus, systematic testing approaches are needed to generate a partial test oracle of synthetic data aimed at improving the reliability and robustness of DNN-based systems in different phases of their life cycle.

Indeed, during the model engineering phase, DL engineers have to tune their models by selecting appropriate DNN architecture, appropriate loss function, optimization method, and–or pre-fixed hyperparameters values. These configuration choices have a significant impact on the performance and reliability of the resulting model. DL engineers, therefore, need to assess the impact of their configuration choices, carefully. The effectiveness of this assessment depends on the quality of testing data to trigger both the major functionalities of the model (regular data) and the minor functionalities, *i.e.*, corner-cases (rare data). Thus, high-quality testing data should come from the same distribution as the training data, in order to verify that the DNN learned the principal functionalities, given regular inputs and normal situations. Besides, they also need to be different enough from the training data; this allows pushing intermediate computations outputs to their bounds' regions and exposing potential DNN inconsistencies.

Regarding the model deployment phase, the quantization [3] process is performed to fit the trained model into constrained environments (in terms of resources, such as computation capacity, storage, and power), when migrating from a high-performance training platform to embedded systems and cell phones. Studies [81] [82] have shown that full precision with 32-bit floating-point for parameters may not be necessary to maintain similar level of DNN performance. However, a post-deployment testing phase is required to assess the effect of quantization on the reliability and the robustness of the model. Indeed, as the DNN becomes deeper, the approximated mapping function includes longer computation sequence of both linear and non-linear operations. In such situations, the predicted outcomes become insensitive to small changes in parameters' elements; so the likelihood of coincidental correctness increases, which could result in the same level of overall accuracy between the two versions of the model. Therefore, the challenge is to generate testing inputs that are resilient to

this phenomenon and, hence, are capable of checking for the existence of inconsistencies and unexpected behaviors in the in-production model.

As described in Chapter 3, previous research studies on well-known DNNs and popular datasets show that the neuronal coverage criteria can enhance the diversity of generated inputs and provide meaningful guidance to explore the internal DNN logic and uncover corner-cases DNN behaviors, but white-box testing specialized for DNN-based software is still at its early stage.

In this chapter, we propose DeepEvolution, the first Search-based Software Testing (SBST) approach specialized for DNNs models. DeepEvolution aims to detect inconsistencies and potential defects related to the functionality of DNN-based models. To achieve this goal, we leverage metamorphic relations, as pseudo-oracle technique, to additionally distinguish faulty predictions from correct ones, for the synthetic generated data. We establish a fitness function exclusively based on DNN coverage to extend the investigation on the effectiveness of neuronal coverage in providing meaningful guidance to test cases generation. DeepEvolution relies on nature-inspired metaheuristics algorithms to explore the search space of semantic-preserving metamorphic mutations and to generate semantically preserved synthetic inputs that increase the neural network's internals coverage; guiding the testing process in finding corner-cases behaviors and exposing hidden issues and inconsistencies. We assessed the effectiveness of DeepEvolution through case studies with popular image recognition models trained on both MNIST and CIFAR-10 datasets. Results show that (i) DeepEvolution succeeds in finding relevant metamorphic transformations that allow triggering uncovered neurons' values and, consequently, boosting the neuronal coverage of DNN under test. (ii) Using DeepEvolution, we were able to uncover major and corner-case regions in the models; allowing us to find multiple erroneous model behaviors. (iii) Data generated using DeepEvolution allowed us to detect potential quantization defects in the models.

Chapter Overview Section 5.1 presents the design of our search-based approach. Section 5.2 describes the implementation of our TensorFlow-based library and its utilization. Section 5.3 reports about a case study aimed at evaluating DeepEvolution. Finally, Section 5.6 concludes the chapter.

5.1 DeepEvolution : Testing Workflow

In this section, we detail DeepEvolution's workflow that describes how to conduct DNN testing with different activities (*e.g.*, test generation, test execution, and test evaluation).

To begin with, we present the key concepts and activities that we use in our approach

specialized for testing DNNs.

Test Oracle: DeepEvolution adopts metamorphic testing as pseudo-oracle technique to circumvent the lack of a reference oracle for DNNs. It defines a metamorphic relation between a set of semantic preserving transformations and the identity function as follow-up test because the expected output of a transformed input should be the same as its genuine one. Thus, DNN fails the test when it triggers erroneous behaviour yielding another outcome value. The key is to find parametric input transformations for the target application domain and validate that these transformations preserve the semantic of inputs sampled from the data distribution, under the circumstance of keeping their parameters within a predefined range of values.

Test Adequacy Evaluation: DeepEvolution uses white-box coverage measures for DNN that assess how much a given test input engenders novelty in terms of triggered neurons. The test adequacy of generated test cases is evaluated with respect to two levels. The first level is to measure the amount of uncovered neurons or activations' regions that are triggered by a given test input. This enhances the diversity of generated test inputs to cover major and minor behaviors learned by the DNN, as a consequence, it increases the chances of finding erroneous behaviors. The second level is to estimate how far is the DNN's state obtained when running the generated test case from that state obtained when running the original test case. This allows to generate new synthetic inputs that are semantically equivalent with their original version, but different enough to exhibit a different DNN's state. As the first level is global, the application of multiple successive mutations on one input could converge quickly to zero when no new uncovered neurons or regions are triggered. This hinders its guidance aspect since it will not be able to assess the difference between mutated inputs, and therefore will not provide useful feedback to improve the generation process. The second level is quite local focusing on the behavioral changes induced by the applied transformation and is able to compare the candidate solutions inferred for one original inputs even if no more new neurons are covered.

Input Test Generation: DeepEvolution encapsulates a search-based approach to generate synthetic inputs from the existing test data. It defines a transformation vector that contains a value for each parameter (*i.e.*, including the neutral value) needed to apply the supported metamorphic transformations. Instead of searching in the space of inputs, DeepEvolution's search process consists of exploring the space of transformation vectors towards prominent directions where there are subtle and interesting transformations that have high chances to expose potential inconsistencies. Thus, there is no optimal

or sub-optimal data point to find. The objective is to find useful transformations that are able to provide effective test cases. To do that, DeepEvolution relies on population-based metaheuristic that maintains a set of candidate solutions and iteratively evolves them to produce new derived candidates that are strong and probably better than their predecessors in terms of fitness value. The fitness function should reflect the measure of adequacy evaluation of the mutated input resulting from applying a given transformation. Thus, it ensures that from one generation of candidates to another, DeepEvolution maximizes the chances of uncovering new transformations that are both enough different from the old ones to exhibit new DNN's behaviours and enough similar to those with high fitness values to keep the search processing in prominent directions and find more relevant corner-cases. However, sine the metamorphic transformation vector contains parameters related to the application of multiple transformations, it is probable that their application at once to the input could lead to meaningless mutated inputs, even if each semantic preserving transformation's parameters are separately verified with respect to a valid interval of values. To reduce this risk of meaningless inputs, DeepEvolution requires an appropriate similarity measure to be defined in a way that a given mutated input is rejected if its similarity with its genuine version is less than a pre-tuned threshold.

Figure 5.1 presents an overview of the design of DeepEvolution which is composed of the following components.

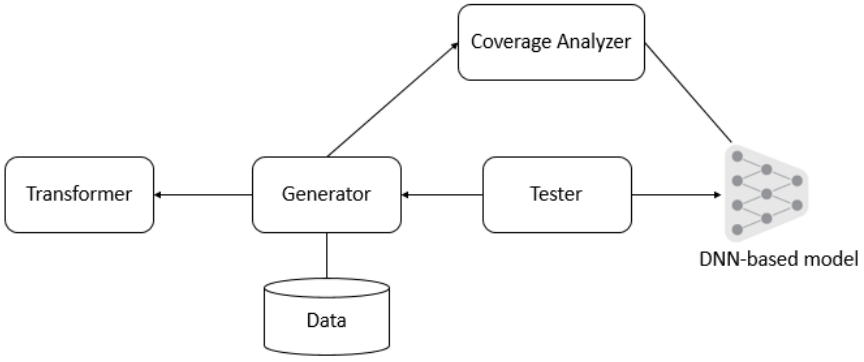

Figure 5.1 Overview design of DeepEvolution

Transformer: It contains the metadata and the processing logic for the metamorphic transformations. Thus, it exposes an interface providing multiple operations to create a random metamorphic transformation, to check transformation's elements bounds and clip

them within the acceptable range, and to apply the transformation on a given input data.

Coverage Analyzer: It is in charge of assessing how much a given test input engenders novelty with respect to the chosen coverage criteria. To do that, it stores neurons' data and fetches the activations' values during the testing process to keep track of the new triggered DNN's states with respect to both the current global coverage value and the DNN's state resulting from the corresponding original input.

Generator: It is responsible for test data generation guided by coverage criteria; so it communicates with both previous components. It uses a population-based metaheuristic optimizer to generate testing inputs maximizing the coverage-based fitness function. In practice, a population-based metaheuristic algorithm is composed of four sub-components as follow:

1. *Population Initializer:* It initializes randomly a set of feasible candidate solutions.
2. *Fitness Evaluator:* It computes the fitness of a given candidate solution with respect to the established function.
3. *Population Updater:* It encapsulates the selected metaheuristic strategy to infer the next population in a way that increases the chance of finding more optimal candidate solutions.
4. *Feasibility Checker:* It ensures that the elements of the individuals belong to the valid range of values after updating them.

In our case, the exploration is performed on the metamorphic transformations' space. The generation process starts by running the population initializer once at the beginning to construct the initial random set of valid metamorphic transformations (the size of the population is the first common parameter). After that, it iteratively executes the three steps 2-3-4 until reaching the fixed maximum number of iterations (the second common parameter) in order to update the transformations based on their fitness, while keeping them within the valid boundaries, which guarantee obtaining valid and more prominent metamorphic transformations. The production of the next generation of individuals differs depending on the selected metaheuristic strategy. This step defines the inferring logic from one population to another, performing both the intensification (*i.e.*, exploitation of the results to concentrate the search on the regions around the found effective solutions) and diversification (*i.e.*, the exploration of non-visited regions to avoid missing potential interesting solutions).

Tester: It is the main component that manages the DNN testing workflow as presented in the pseudo code 5.1. First, the initial test data is loaded and filtered to keep only the test inputs that are correctly classified by the DNN (because the misclassified original inputs represent failed tests and therefore cannot be used as base for test cases generation). Second, it iterates over the remaining sample data, takes one atomic input during each iteration and starts generating synthetic inputs by mutating the original inputs. Then, it performs the follow-up tests for the valid synthetic testing inputs that are semantically equivalent to the original inputs with respect to a predefined similarity measure. Indeed, reaching high neuronal coverage and exploring the maximum of potential DNN's states provide meaningful guidance to enhance the diversity and the effectiveness of testing cases. However, the goal of testing is to find potential defects. Thus, the follow-up test objective could be detecting erroneous behaviors and evaluating the robustness of the DNN by checking if its prediction outcome differs from the original predicted value or accurately capture potential defects during DNN quantization for platform migration, by comparing the outputs of the original model and its quantized version. Last, it stores each created valid input that was capable of spawning a failed test, with respect to the pre-defined test objective for further analyses.

```
import Generator, CoverageAnalyzer, Transformer, DNN
#The initial test data is loaded
test_data = load_data(test_file)
#We set up the baseline global coverage as the neurons covered by the initial
    test data set
CoverageAnalyzer.init_global_coverage(DNN, test_data)
#We execute the DNN on them to obtain the neurons' activations for each test
    input and its predict outcome
predictions = DNN(test_data)
#We keep only those which are correctly classified by the DNN
sample_test_data = get_correctly_classified(predictions, test_data)
for orig_elt, label in sample_test_data:
    #We set up the baseline local coverage
    CoverageAnalyzer.init_local_coverage(DNN, orig_elt)
    #We initialize a random first population given its size.
    transformations = Generator.init(orig_elt, pop_size)
    n_iter = 0
    #We repeat the following tests until reaching the fixed maximum number of
        iterations
    while n_iter < Max_iterations:
        for tr in transformations:
            #We apply the transformation on the original input
            gen_inputs = Transformer.transform(orig_elt, tr)
```

```
21            for gen_elt in gen_inputs:
22                #We verify that the similarity between the transformed input
                      and its genuine one is higher than a pre-fixed threshold
23                sim_elt = compute_similarity(gen_elt, orig_elt)
24                if sim_elt > threshold:
25                    #We run the follow-up test.
26                    predict_label = DNN(gen_elt)
27                    is_failed = follow_up_test(label,predict_label)
28                    #We store each transformed input that was capable of
                          spawning a failed test for further analyses.
29                    if is_failed:
30                        store_data(gen_elt)
31        #We compute the fitness relying on the evaluation of DNN coverage
              exhibited by the resulting one or multiple transformed inputs.
32        gc = CoverageAnalyzer.get_global_coverage(DNN, gen_inputs)
33        lc = CoverageAnalyzer.get_local_coverage(DNN, gen_inputs)
34        fitness = Generator.eval_fitness(gc, lc)
35        #We infer the next population based on the previous population
              candidates and their fitness.
36        transformations = Generator.update(transformations, fitness)
37        n_iter += 1
```

Listing 5.1 DeepEvolution High-level Pseudo code

Our approach can be instantiated to test DNN-based applications. The concrete instance of DeepEvolution should define the parametric metamorphic transformations, the data similarity measurement, coverage-based fitness function, and the population-based metaheuristic algorithm. Then, DeepEvolution runs its testing workflow (see Figure 5.2), which explores the metamorphic transformations' space to find interesting elements that allow the creation of effective test cases, covering the major and minor behaviors of DNN under test.

5.2 DeepEvolution: Implementation

DNN–based models have dominated computer vision over the past few years; enabling rapid progress and achieving high performance that almost reach or surpasses humans on several visual recognition tasks such as object detection, face recognition, action and activity recognition [83]. In this section, we propose an implementation of DeepEvolution components to build a novel search-based approach for testing computer-vision DNN models. Briefly, the implementation is based on eight semantically preserved metamorphic transformations, neuron-level coverage criteria, and nature-inspired population-based metaheuristics. In the following, we elaborate on each of these aspects in more details.

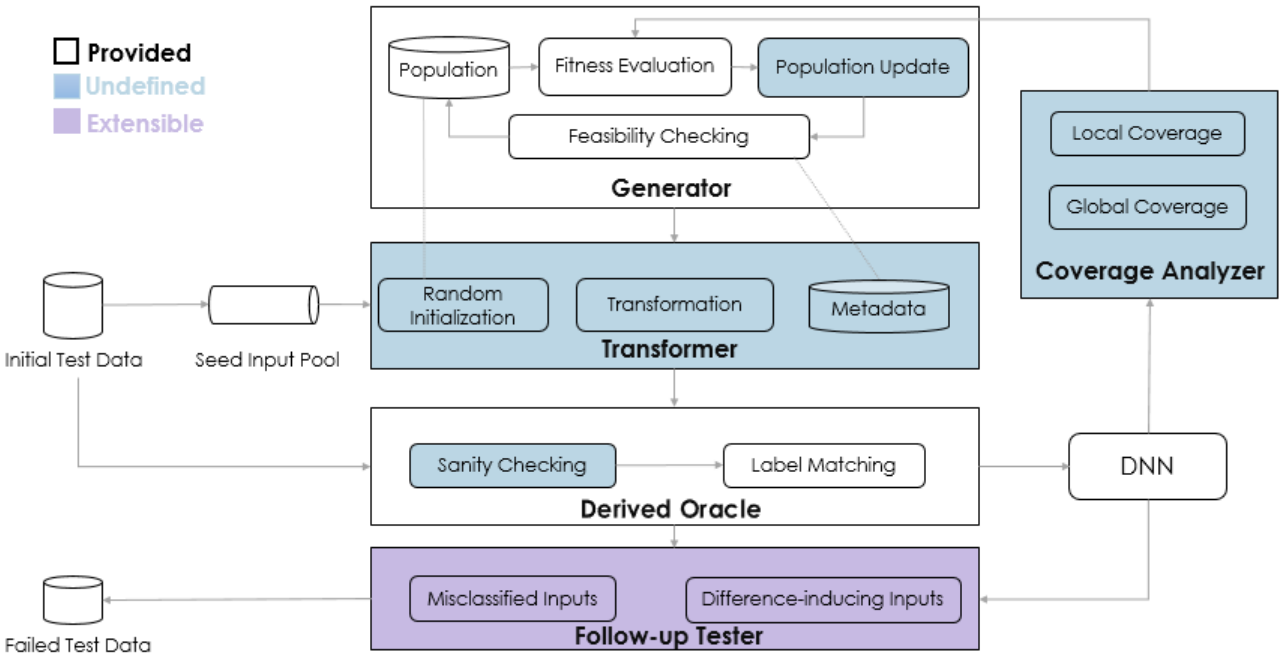

Figure 5.2 Overview of DeepEvolution Testing Workflow

5.2.1 Metamorphic Transformation

First, we gather a list of parametric image-based transformations that can be organised in two groups:

1. **Pixel-value transformations:** change image contrast, image brightness, image blur, image sharpness and random perturbations within a limited interval. Each one of these pixels' mutations takes one or multiple parameters, such as a floating-point factor controlling its effect, lower values mean less brightness, contrast, etc.., and higher values mean more.

2. **Affine transformations:** image translation, image scaling, image shearing, and image rotation. Each one of these affine transformations accepts geometric parameters, such as the components with respect to spatial axis (x and y) or an angle θ.

Because each image-based transformation has a theoretical domain, which defines the interval of possible values of its parameters such as brightness factor or rotation angle θ, when applying transformations, we need to take into account these domain boundaries to ensure that transformed inputs are semantically equivalent to the original ones, *i.e.*, given an image I, the application of one transformation on I generates another new image $I\prime$ such that the

semantics of I and $I\prime$ are the same from the perspective of humans.

To infer the valid domain interval of each transformation's parameters, we manually tune them to set up the appropriate range of values, *i.e.*, high and low boundaries, that preserves the input semantics before and after each transformation, with respect to the data distribution. The tuning process consists of the following steps: (1) we select a transformation and a sample data inputs; (2) starting with the theoretical range of values, we apply, repeatedly, the transformation on each input with several random parameters' values adopting an increasing strategy, *i.e.*, from small values to big ones; (3) we manually check the transformed inputs to identify the threshold values above which the resulting images loose their meaning or deviate significantly from the original images; (4) we update the range of parameters' values and the used increase step to re-iterate the process with the aim of refining the obtained range. The tuning process is terminated when we correctly define two bounds that represent high and low accepted values for each transformation.

To enable a large-scale generation of synthetic inputs from existing labeled testing data, we build a compound metamorphic transformation that assembles all the image-based transformations described above, in order to enhance the changeability of mutations and the diversity of generated inputs. Its application on a given image consists of applying the supported pixel-value transformations in sequence, and then, performing each single affine transformation once, on the resulting mutated input. We opted for this conservative strategy that consists of applying only one affine transformation following the pixel-value mutations because applying multiple affine transformations at once would increase the chances of generating meaningless images, *i.e.*, images that don't occur in real-world situations, *e.g.*, camera deficiencies or abnormalities in input preprocessing.
Our defined metamorphic transformation produces the following mutated inputs : inputs resulting from only pixel mutations and inputs that are the results of applying, separately, each one of the affine transformations. To verify that generated inputs remain semantically equivalent to the original inputs, we compute a Structural Similarity Index (SSIM) [84] which assesses the similarity between two images based on the visual impact of three characteristics: luminance, contrast, and structure. We expect that pixel-based mutated inputs differs from their originals with respect to these characteristics, but a very low SSIM indicates that the new image looses mostly all the information inherited from its parent. Therefore, we reject mutated synthetic inputs for which SSIM values are lower than a pre-defined threshold.

Since DeepEvolution relies on a search-based approach to explore the set of metamorphic transformations, we encode the parameters required (*i.e.*, the parameters of the supported transformations) for our compound metamorphic transformations as a vector 5.3, so the

search space of transformations is a multi-dimensional space where each component represents one parameter that may be related to either a pixel-value or affine transformation.

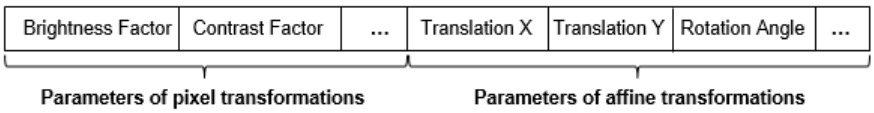

Figure 5.3 An illustration of the vector encoding the metamorphic transformation

5.2.2 DNN Coverage

As described in test adequacy evaluation, we need to define a function that includes two coverage levels (*i.e.*, local and global) in order to be efficient in the search process. Therefore, we adapt the Neuron Coverage (NC) metric proposed by Pei et al. [44] to capture two levels of coverage (*i.e.*, local and global) for each test input.

Local neurons coverage (NLNC): this represents the new neurons covered by the mutated test input that have not been covered by its corresponding original test input.

Global neurons coverage (NGNC): this consists of the new neurons covered by the mutated test input that have not been covered by all the previous test inputs, including both genuine and synthetic test inputs.

We define the following fitness function:

$$Fitness = \alpha \times NLNC + \beta \times NGNC \tag{5.1}$$

α and β are weights assigned to each coverage measure.

5.2.3 Nature-inspired Metaheuristics

We encode our compound metamorphic transformations (MT) as a vector, where each component represents one parameter that may be related to either a pixel-value or an affine transformation. To ensure semantically preserving transformations, we use the valid domain intervals of transformations that we have already tuned manually to create the high and low boundaries vectors, defining the sub-space of exploration. This encoding makes it easy and more natural the application of metaheuristic optimizers for the exploration of transformations to generate synthetic testing inputs. We use the valid domain interval of the transformations to implement the population initializer (which creates random semantically preserving metamorphic transformations) and the feasibility checker (which clips the

generated transformations' parameters within the range of acceptable values).

To evolve the population of transformations towards more prominent regions of the space, we implement nature-inspired metaheuristics as different instances of the population updater of the Generator component of DeepEvolution.
Nature-inspired metaheuristics are a special kind of population-based metaheuristic algorithms that performs subtle steps, including stochastic, diversity and selection, mimicking the behavior of biological organisms; where the fittest survive and reproduce. In this thesis, we instantiate DeepEvolution using 1 evolution-based algorithm and 9 swarm-based algorithms. We opted for investigating several metaheuristic algorithms because according to the No Free Lunch Theorem (NFL) [85], there is no algorithm that can outperform all other algorithms with regard to all possible classes of optimization problems. Thus, it is important to assess the performance of different metaheuristic searching strategies, when we deal with a new type of application.
In the following, we present our implementation of the selected algorithms.

Genetic Algorithm

Genetic Algorithm GA [86] is the most popular evolution-based method that mimics the behaviour of biological evolution including concepts from the Darwinian theory about reproduction and natural selection. This method involves basic operations of selection, crossover and mutation to generate, potentially, better individuals in every generation. Below, we detail our definition of these three main operations.

1. *Selection:* This operation ensures that "fitter" individuals have high chances to be selected for breeding the next population. To do that, we rank the individuals with respect to their fitness value. Then, we assign a probability of selection to each candidate by computing the softmax of the fitness values to turn them into a probability distribution. Candidates with high probability values have higher chances to be selected. Last, we stochastically and independently select random parent pairs among the population with respect to their estimated probability.

2. *Crossover:* Inspired from biological crossover, it represents the process of reproduction by taking a pair of individuals as parents to produce one child solution from them. The breeding operation consists of selecting an element from either the first or the second parent with respect to the following selection probabilities based on fitness evaluation.

$$P(parent_i) = \frac{fitness(parent_i)}{fitness(parent_1) + fitness(parent_2)}, \forall i \in \{1, 2\}$$

3. *Mutation:* To preserve and introduce diversity during the exploration of the search space, the generated individuals can be subject to random mutation, according to a pre-defined probability p. To enhance the reuse of implemented modules and ensure that the mutated elements stay within the permissible range of values. We generate a random candidate and we compute a weighted sum between the generated individual and the random one, according to a small mutation weight α.

$$mut\_ind = \alpha \times rand\_ind + (1 - \alpha) \times gen\_ind$$

Swarm-based Algorithms

Swarm-based methods mimic the behaviour of natural swarms, (*i.e.*, group of animals such as flocks of birds or ant colonies) interacting locally with one another and with their environment. For our testing data generator, we implement the following 9 well-regarded and recent swarm intelligence algorithms: (1) Particle Swarm Optimization (PSO) [87], (2) Cuckoo Search Algorithm (CSA) [88], (3)Firefly Algorithm (FFA) [89], (4) Bat Algorithm (BAT) [90], (5) Gray Wolf Optimizer (GWO) [91], (6) Moth Flame Optimizer (MFO) [92], (7) Whale Optimization Algorithm (WOA) [93], (8) Multi-Verse Optimizer (MVO) [94], (9) Salp Swarm Algorithm (SSA) [95]. These selected metaheuristics algorithms are flexible enough to be easily applicable to a broad class of constrained optimization problems involving high dimensional bounded real-valued vectors without any prior search space discretization. The space of transformation vectors that we have set up can be explored by their standard implementations and there is no specific operations to develop. During their implementation, we kept default configurations parameters (which is a conservative approach) and followed state-of-the-art solutions, regarding the conventions of multidimensional arrays for data representation and algebra routines for update computations.

5.3 Empirical Evaluation

We assess the effectiveness of DeepEvolution through the following three research questions:
RQ1: How much can DeepEvolution increase the coverage of generated test cases?
RQ2: Can DeepEvolution detect diverse erroneous behaviors in DNN models?
RQ3: Can DeepEvolution detect divergences induced by DNN quantization?

5.3.1 Experiment Setup

Hardware. The experiments are performed on a high performance computer that runs with Linux CentOS 7 system on Intel(R) Xeon(R) 3104 Bronze with 64 GB of RAM equipped with a NVIDIA GeForce RTX 2080 Ti GPU.

Software. We implemented DeepEvolution to test TensorFlow-based programs, using Python. We chose TensorFlow because of its popularity in the DL community. However, the DeepEvolution approach proposed in this thesis can be adapted for other DL libraries such as PyTorch. As for the metaheuristic algorithms, we implemented them from scratch using the Numerical computation Python library (NumPy).

Datasets. We selected the two popular publicly available datasets, MNIST [96] and CIFAR-10 [97], as our evaluation subjects.
MNIST [96] is a dataset of handwritten digit image recognition, containing $60,000$ training data and $10,000$ test data, with a total number of $70,000$ data in 10 classes (*i.e.*, handwritten digits from 0 to 9). Each MNIST image is a single-channel of size $28 \times 28 \times 1$.
CIFAR-10 [97] is a collection of images for general-purpose image classification, including $50,000$ training data and $10,000$ test data in 10 different classes (*e.g.*, airplanes, cars, birds, and cats). Each CIFAR-10 image is three-channel of size $32 \times 32 \times 3$.
Since neuronal coverage estimations and models post-execution analysis are computation-intensive tasks, we decided to take random samples from each of our studied test datasets as initial testing data. More specifically, we randomly selected two sample instances D_1 and D_2 from each dataset; with increasing size (*i.e.*, respectively 50 and 100).
Sampling with increasing size allows us to assess the effect that adding more data points in the initial test data, has on the effectiveness of our search-based approach.

DNNs For each dataset (*i.e.*, MNIST and CIFAR-10), we took, respectively, the official open-source implementation of TensorFlow models, LeNet [98] and CifarNet [99] to allow the reproducibility of our results and comparisons with our approach.
We trained LeNet [98] using the same hyperparameters' configuration mentioned in the script from [100]. In the case of CIFAR-10, we selected a CifarNet [99] DNN that allowed resolving the CIFAR-10 problem with an acceptable accuracy in a reasonable amount of time. In fact, the classification task of CIFAR-10 is generally harder than that of MNIST because of data size and complexity. When training the DNN, we used the hyperparameters introduced in

the configuration script provided in [101].
Selecting open-source models allows us to test DeepEvoluton on models implementations that have been carefully reviewed and tested by several developers over a long period.

Settings of DeepEvolution We adopt the default implementation of each metaheuristic algorithm, *i.e.*, we keep their internal parameters values to defaults values (which is a conservative approach) . Concerning the fitness function, we choose $\alpha_1 = 0.1$ and $\beta_1 = 1.0$, which is consistent with their corresponding measure magnitude and our intention of increasing the neuron coverage as our primary objective, but, we need to add the local-coverage measurement in order to refine the fitness evaluation between candidate solutions, especially when the testing process achieves higher coverage. Considering the compute-intensive post-execution tasks, we select equal values for the two common hyper-parameters of all population-based metaheuristics, $populationsize = 10$ and $maxiterations = 10$ because some metaheuristic algorithms rely on the iterative evolution of population and others rely on the availability of multiple candidates. We choose same values for them to make the evaluation as fair as possible. To avoid the effects of randomness used in our gradient-free optimizers, all results are averaged over 3 runs or more.

5.3.2 RQ1: DNN Coverage Increase

Motivation. The neuronal coverage indicates the DNN's internal states explored by the test data. We aim to evaluate if the generated test data can help increase the neuronal coverage, *i.e.*, triggering neurons, which are not covered by the original test data.

Answer. DeepEvolution significantly boosts the neuronal coverage. Table 5.1 shows the final neuronal coverage ratio achieved by each implemented nature-inspired metaheuristic. The results show that the synthetic test data generated by all the studied metaheuristic algorithms significantly enhance the two coverage measures, as confirmed by the Wilcoxon Signed Rank tests. Overall, we can see that the neuronal coverage ratios obtained are generally higher than 90%. This result confirms previous works' claims that neuron coverage computed by considering a neuron to be active or not (like a Boolean condition) is relatively easy to satisfy. Advanced neuronal coverage criteria should take into account the continuous value of neurons' activations and define multiple possible states for a neuron. We go into more details on the neuronal coverage improvements in the discussion from Section 5.4. However, reaching higher global coverage would not hinder the effectiveness of our exploration, as verified by the next investigated RQs, because we include the local

Table 5.1 The neuron coverage values obtained by each testing approach

Meta heuristic	MNIST		CIFAR-10	
	D_1	D_2	D_1	D_2
Traditional	44.77	50.89	48.03	53.16
BAT	94.85	96.35	96.02	97.99
CS	94.74	96.12	96.78	98.30
FFA	97.64	98.18	96.46	98.50
GA	83.96	88.28	92.37	95.81
GWO	92.77	94.55	96.32	97.83
MFO	93.11	95.10	95.64	97.52
MVO	86.57	90.06	93.76	96.54
PSO	91.11	94.04	95.50	97.49
SSA	98.22	98.66	96.69	98.53
WOA	94.55	95.91	95.85	97.68

coverage, which is able to capture the differences between DNN's behaviors in response to the generated inputs and their original parent.

Although the obtained neuronal coverage measures were generally high, the searching process reaches almost a stationary value when it could no longer improve the global coverage induced by the generated inputs, and as a consequence, its role becomes limited to only finding transformed inputs pushing the DNN to behave differently. Nevertheless, the augmentation of the original test data lend a refreshing boost that enabled the enhancement of neuronal coverage, which shows that adding more original instances enlarges the inputs search space to cover more possible test cases. This suggests that the quality of the initial input data is important for successfully covering the major patterns learned by the DNN model under test and increasing the chances of producing rare test inputs.

We do not intend to compare the performance of the different nature-inspired metaheuristics, since we kept their default parameters (*i.e.*, without any tuning). However, we noticed that *GA* and *MVO* perform slightly worse than the others. This can be explained by their tendency to spend more time in regions around the best found candidates; not going further to explore solutions that are far from the best recognized regions.

This characteristic is however helpful when a metaheuristic optimizer is used to find an optimal global solution at the end, but since our objective is to explore the maximum of relevant regions in the space, we need to increase the exploration ability of these metaheuristic. This can be done by enhancing the probability and the weight of mutations in *GA*; so that the child could be more different from its parents. For *MVO*, we can fix the starting parameters that emphasize the exploration such as higher TDR (*i.e.*, the distance of maximum variation around the best solution) and lower WEP (*i.e.*, probability of generating new candidates

around the best solution).

Furthermore, similarly to the usage of test coverage in traditional software testing, increasing the neuronal coverage has been shown to be an effective way to enhance the diversity of the generated inputs; allowing uncovering rare corner-case behaviors, and potentially, intensifying their defect-revealing ability. The effectiveness of our search-based approach in detecting defects is the main purpose of the next two research questions.

5.3.3 RQ2: DNN Erroneours Behaviors

Motivation. The DNN-based classifier should identify and learn high-level patterns that are able to differentiate between the inputs from different classes with high confidence. We aim to assess the effectiveness of our approach in testing the robustness of the trained DNN; by finding misclassified synthetic inputs.

Answer. DeepEvolution generates, continuously, valid synthetic data resulting from applying constrained-based metamorphic transformation controlled by a similarity verification that filters the transformed inputs, which do not preserve the semantic of its original version. Then, the objective is to ensure that DNN is capable of detecting the corresponding label for each mutated input; so the follow-up test consists of comparing the predicted outcome for the mutated input with the label of the genuine input, if they are not equal, we consider it as a failed test caused by an erroneous DNN's behavior. The detected erroneous behaviors by each implemented metaheuristic algorithm is shown in Table 5.2. For all the 10 metaheuristic algorithms, DeepEvolution successfully generates test data that expose erroneous behaviors in the DNNs under test. This finding is not very surprising since recent works [44] [56] have shown that there is a positive correlation between the neuronal coverage and erroneous behaviors' triggering tests.

The fact that all the 10 metaheuristic algorithms succeeded in revealing defects in the studied DNNs indicates that generating synthetic test inputs towards improving the neuronal coverage is an effective approach to trigger more states of a DNN. These optimized test data could increase the chances of detecting defects. This finding is consistent with the practical purpose of testing criteria widely adopted in traditional software testing. Also, the augmentation of sample data size, from D_1 to D_2 has significantly increased the number of erroneous behaviors detected. We obtain almost the double by doubling the input data size. This result suggests that DeepEvolution is capable of obtaining adversarial inputs for each original input and that the local coverage level integrated in the fitness function plays an important role in

Table 5.2 The number of erroneous behaviors detected by each metaheuristic algorithm

Meta heuristic	MNIST		CIFAR-10	
	D_1	D_2	D_1	D_2
BAT	488	963	317	642
CS	1567	3499	1533	3001
FFA	79	202	1142	2235
GA	168	434	203	422
GWO	1298	2411	1046	1929
MFO	1343	2955	1098	2387
MVO	378	774	370	764
PSO	1116	2913	1108	2403
SSA	112	262	1285	2738
WOA	1702	3601	1122	2335

assessing how much the DNN's state of the transformed input is different from the state that resulted from the original input. Thus, it is capable of finding corner-cases testing inputs even if the global neuronal coverage reaches higher values; as evidenced by the increase in the triggered erroneous behaviors when augmenting the initial test data despite no significant improvement in the coverage between the two dataset samples.

The results of *GA* and *MVO* reinforce the previous observation about their lack of exploration capability.

The default implementation of *BAT* also exhibits a similar insufficiency of diversification that could be remedied using adaptive rates of pulse emission r (*i.e.*, the probability of adjusting the found solutions) and loudness A (*i.e.*, the probability of generating a new candidate randomly). Specifically, we can configure adaptive rates of pulse emission and loudness that it favors; diversifying solutions from different regions of the space and then moving towards an intensive local search close to the best solution found. Additionally, we notice that *FFA* and *SSA* found significantly lower number of erroneous behaviors from the LeNet DNN on MNIST data. An in-depth investigation revealed that the valid testing inputs generated by *FFA* and *SSA* represent, respectively, 1.8% and 2.5% of the total synthetic data created, meaning that the majority of generated inputs are rejected by the similarity verification. These two metaheuristics explore well the search space and push the transformations' parameters to their limit bounds, but the grey-scale images of MNIST data lost their meaning quickly when we apply several transformations close to the acceptable bounds. We did not observe this limitation when dealing with CIFAR-10 because the transformations are more adequate for colored RGB images and even assembling them with higher intensity parameters does not result directly in meaningless data.

These results show that DeepEvolution can effectively generate tests to trigger

erroneous behaviors in the DNNs; providing feedback on the reliability and the robustness of the models.

5.3.4 RQ3: DNN Quantization Defects

Motivation. Due to the limited computation and storage resources on diverse deployment platforms, the trained parameters of DNNs are often quantized from high precision floating to a lower precision format, in order to fit their footprint to the targeted platform. However, this operation (*i.e.*, quantization) which results in information loss, is not without impact on the quality of the DNN models. It is therefore important to assess the impact of quantization on the performance of resulting DNNs. Obviously, it is not very useful to quantize a model if the performance of the resulting model is significantly reduced. The traditional testing approach which is based on random unseen data often fails to detect divergences between the original model and the quantized model. Our goal in this research question is to assess the usefulness of DeepEvolution in finding difference-inducing inputs that expose potential defects resulting from quantization.
TensorFuzz [67] performs a coverage-guided fuzzing process to generate mutated inputs that are able to expose disagreements between a DNN trained on MNIST (that is 32-bit floating point precision) and its quantized versions where all weights are truncated to 16-bit floating points. The tool and the DNNs used for its evaluation are released under an open source license on GitHub by the Google Brain research team[1].
In this thesis, we use it as baseline to assess DeepEvolution.
To ensure a fair comparison, we fix the configuration of TensorFuzz, including the data corpus size and number of mutations per element, in a way that the two approaches (TensorFuzz and DeepEvolution) produces the same number of test cases from each original test input.

Answer. Table 5.3 presents the number of synthetic test data that were able to induce a difference between the DNN's outcomes (difference-inducing inputs); exposing quantization defects.

As can be seen, all the implemented metaheuristics succeeded in exposing quantization defects and most of them outperformed TensorFuzz in terms of number of difference-inducing inputs found. However, the nature-inspired metaheuristics, *FFA* and *SSA*, only generated few difference-inducing inputs. As explained in the previous RQ, since the data is grey-scale, these two metaheuristics have the same limitation; they produce a high number of invalid in-

[1] `https://github.com/brain-research/tensorfuzz`

Table 5.3 The number of sensitive defects detected by each metaheuristic algorithm during DNN model quantization

Test Method	D_1	D_2
TensorFuzz	8	17
BAT	32	70
CS	71	136
FFA	3	8
GA	30	74
GWO	26	103
MFO	39	78
MVO	29	66
PSO	42	86
SSA	3	9
WOA	24	69

puts that are rejected by the similarity verification step. Adequate configuration and tuning is required to overcome this issue. Results of this evaluation are consistent with our fundamental motivation for leveraging metaheuristic-based searching algorithms, which is that by enabling the optimisation of coverage criteria, metaheuristic-based searching techniques can help increase diversity in generated test cases and hence improve their efficiency. DeepEvolution not only produces higher number of difference-inducing inputs than TensorFuzz. It also finds more subtle and interesting quantization defects resulting from different input mutations that can be studied furthermore, to understand the types of situations that quantized DNNs are not able to handle properly in comparison to their corresponding full-precision versions.

DeepEvolution can effectively detect potential defects introduced during DNN quantization from 32-bit floating point to 16-bit, outperforming coverage-guided fuzzing method.

5.4 Discussion

In this section, we discuss furthermore the advanced neuronal coverage criteria that defines multiple possible states of the neuron, deriving from its activation' continuous value because we found that our global neuron coverage is relatively easy to satisfy. We found DNNs models with high coverage levels that still exhibited erroneous behaviors. In the following, we propose a fine-grained fitness 5.2 that is based on the refined neuron coverage criteria proposed by Ma et al. [46] including :

1. **K-multisec. Neu. Cov. (KMNC):** the ratio of covered k-multisections of neurons

that is calculated by dividing the range of activations observed during training sessions for each neuron into k equal sections. Afterward, it watches how much the testing inputs are able to trigger the pre-defined sections for the model's neurons. A high MKNC value could indicate the quality of test data (whether it being genuine or synthetic) in terms of prompting the major patterns regions that the neural network learned from the training data.

2. **Neuron Bound. Cov. (NBC):** the ratio of covered boundary regions of neurons that consists of measuring how well the test datasets can push activation values to go above and below a pre-defined bound (*i.e.*, covering the upper boundary and the lower boundary values). Thus, a high NBC value could illustrate the capacity of test cases in reaching corner-case patterns regions to examine the model's behavior beyond its "normal" range of values.

These two complementary coverage metrics could be adopted to capture the global level of coverage in our fitness function. They help quantify the amount of new covered regions (whether they belong to major pattern regions or fall into corner-case ones). For the local coverage level, we draw inspiration from the coverage criteria proposed by Odena and Goodfellow [67] which consists of estimating the novelty through computing the distance between the activations' state vectors (*i.e.*, the activations of layers encoded as a real-valued vector) given by the new test input and its nearest neighbor from previously obtained activations' states, using the Euclidean distance metric. Thus, we store the activations' state resulting from running the original test input as a base state, Then, we compute the distance between the activations' state exhibited by the mutated test input and this base state, to estimate how much behavioral differences are induced by the performed transformation. This distance-based measure is more accurate than calculating the major and boundary regions covered by the mutated input, but uncovered by its original, because it takes full advantage of the continuous nature of activations' values and captures precisely the deviation in DNN's behavior succeeding the input transformation. We name these new coverage measures: local activations distance (LAD), novel global major regions coverage (NGMRC) and novel global boundary regions coverage (NGBRC) with their weight parameters, respectively, α_2, β_2 and γ_2.

$$Fitness_2 = \alpha_2 \times LAD + \beta_2 \times NGMRC + \gamma_2 \times NGBRC \tag{5.2}$$

Leveraging these newly defined fitness functions (*i.e.*, fine-grained and coarse-grained) on both DNNs and their corresponding data sample D_1, we obtain the following results. The Table 5.4 shows a comparison of coverage measure values obtained by each metaheuristics.

This comparison result reinforces our findings and validates the performance of DeepEvolution in enhancing the coverage based on the formulated fitness. As can be seen, the traditional test has very low *KMNC* and *NBC* values, while DeepEvolution was able to increase these values using a few original data instances. Nevertheless, the *NBC* for CIFAR-10 represent an exception since its corresponding values stay low after the testing process. We perform an in-depth investigation on this observation and we found that the official implementation of CIFAR-10 released by the TensorFlow team performs a data augmentation of CIFAR-10 training data, using multiple semantically preserving transformations[2] that are similar to our metamorphic relations. This helps improving the training process. However, since DeepEvolution seeks synthetic inputs in similar search space to the one of augmented data, it fails to push activations' values over the boundaries estimated during the training process. In our experimentation, we use $k = 100$ for *KMNC*. Achieving such high coverage value would have been more difficult if we adopted bigger k values such as 100, 1000 and 10000 to capture minor differences between neurons' outputs through more fine-grained sections. Therefore, the fine-grained neuronal coverage can be a good criteria for testing the adequacy of generated test cases. However, we need to assess their effectiveness in revealing latent defects in DNN-based models.

Table 5.4 Comparison of coverage measures' values obtained by each metaheuristic

Meta heuristic	**MNIST**			**CIFAR-10**		
	NC(%)	**KMNC(%)**	**NBC(%)**	**NC(%)**	**KMNC(%)**	**NBC(%)**
Traditional	44.77	8.48	0.06	48.03	11.48	0.45
BAT	94.85	66.65	32.87	96.02	36.42	0.53
CS	94.74	82.81	53.80	96.78	44.36	0.77
FFA	97.64	89.72	75.32	96.46	41.82	0.93
GA	83.96	66.70	21.22	92.37	38.43	0.54
GWO	92.77	80.72	49.97	96.32	45.40	0.93
MFO	93.11	75.54	43.67	95.64	42.93	0.80
MVO	86.57	66.25	26.13	93.76	39.74	0.59
PSO	91.12	78.41	41.45	95.50	45.34	0.80
SSA	98.22	89.80	75.21	96.70	42.93	0.89
WOA	94.55	81.66	53.04	95.85	44.52	0.90

Moreover, we complete the comparison between the two proposed coverage-based fitness by counting the number of erroneous behaviors that have been triggered during the test. The Table 5.5 summarizes the results of the comparison and shows that there is no significant improvement by the new fitness function, on the contrary, it often performs worse than the

[2] `https://github.com/tensorflow/models/blob/master/research/slim/preprocessing/cifarnet_preprocessing.py`

first fitness.

Table 5.5 Comparison of number of erroneous behaviors detected by both fitness functions

Meta heuristic	**MNIST**		**CIFAR-10**	
	$Fitness_1$	$Fitness_2$	$Fitness_1$	$Fitness_2$
BAT	488	362	317	315
CS	1567	1609	1533	1421
FFA	79	62	1142	1148
GA	168	142	203	215
GWO	1298	1081	1046	1093
MFO	1343	1240	1098	1119
MVO	378	340	370	387
PSO	1116	617	1108	908
SSA	112	120	1285	1343
WOA	1702	1742	1122	1281

Equally important, we also compare the number of difference-inducing inputs that have been generated by DeepEvolution using one of the two proposed fitness. The experimentation was done on the DNN with two versions, half-precision (16 bits) and full-precision (32 bits), provided by TensorFuzz. The results, presented in Table 5.6, show that the fine-grained fitness performs worse than the coarse-grained fitness, in the detection of quantization defects.

Table 5.6 Comparison of the number of difference-inducing inputs found by both fitness functions

Metaheuristic	$Fitness_1$		$Fitness_2$	
Algo.	D_1	D_2	D_1	D_2
BAT	32	70	13	22
CS	71	136	42	46
FFA	3	8	4	3
GA	30	74	15	17
GWO	26	103	22	36
MFO	39	78	26	31
MVO	29	66	17	24
PSO	42	86	20	19
SSA	3	9	1	2
WOA	24	69	10	23

The contribution of fine-grained neuronal coverage in the search-based approach that have been proposed needs an in-depth investigation, because the preliminary results suggest that adopting a more precise fitness function does not improve the effectiveness of testing. This can be explained by the fact that the fitness used by the search-based approach is aimed at estimating the quality of the evaluated solution with respect to the target optimization

problem. Thus, the fitness function should enable a comparison between the candidates and a selection based on their qualities. However, our fine-grained fitness makes comparison and selection more difficult by yielding higher and sometimes very close fitness values; because it always finds differences in the continuous activations' values between the original input and the transformed input. At this point, the coarse-grained fitness is more effective in the sense that it takes into account the differences in terms of activated neurons and newly covered ones. Therefore, it allows distinguishing at a high level, the candidates who have succeeded in generating a quite different DNN's behaviors or triggering uncovered neurons. We believe that there is a trade-off between the accuracy of coverage-based fitness functions in estimating novel coverage and behavioral differences and their ability to extract relevant information that can help differentiate the candidates.

5.5 Threats to Validity

In this section, we discuss potential threats to the validity of our work and highlight the measures taken to circumvent them.

Applicability of SBST to DNN Assurance Quality. In this thesis, we have demonstrated the potential of search-based techniques in improving DNN testing. More specifically, we have shown how the exploration of metamorphic transformations guided by increasing neuronal coverage criteria can help test the robustness of DNNS, and, consequently, contribute to the quality assurance of DNNs. However, the selection of experimental subjects (*i.e.*, dataset and DNN models) could be a threat to validity. We mitigated this threat by using practical model sizes and commonly-studied MNIST and CIFAR-10 datasets. For each studied dataset, we choose to use official TF implementation with their corresponding configuration in order to avoid possible implementation bugs or misunderstanding issues that could hinder our evaluation process.

Diverse Metaheuristic Algorithms. One of the most important components of our search-based approach is the gradient-free optimizer used to find the candidates maximizing the fitness function. We choose to implement nature-inspired population-based metaheuristic because of their randomness and non-deterministic nature that allowed them to be effective in resolving huge space problems. We have evaluated DeepEvolution with several metaheuristics algorithms. Nevertheless, the configuration of these metaheuristics could be a threat to validity. We have selected equal values for the two hyper-parameters, $populationsize = 10$ and $maxiterations = 10$ because some metaheuristic algorithms rely on the iterative

evolution of population and others rely on the availability of multiple candidates, so we choose the same value for them to make the evaluation as fair as possible.

Regarding the specific hyperparameters of each metaheuristic, we used the default configuration provided in their corresponding white-papers, without any prior tuning, because we do not intend to compare their relative performances in solving our testing problem. In fact, we expect their performance to increase if they are tuned to fit our optimisation problem. However, we have examined trade-offs between diversification and intensification, and emphasized the need to select appropriate hyperparameters that ensure the exploration of all the prominent regions that contain sub-optimal solutions with the objective to cover all the major and minor DNN's behavior, and potentially, discover most of the latent defects.

Without Manual Labeling Effort. The strength of metamorphic relations is that they do not require human intervention and they help automating large-scale data generation from partial oracle. However, it is necessary to validate that a metamorphic transformation preserves the semantic of inputs since we assume that the predicted outcomes are consistent. In this work, we tuned the parameters of each constrained image-based transformation and the threshold of the structural similarity index metric, manually. However, this could be a threat to validity. To mitigate this threat, we selected a sample of our generated images using a confidence level of 95% and an error margin of 5%, and verified them manually. We found them to be correct, *i.e.*, we didn't find any transformed input for which the genuine identity was changed.

5.6 Chapter Summary

This chapter proposed DeepEvolution, a search-based approach for testing DNNs, that leverages nature-inspired metaheuristics algorithms and metamorphic relations. It enables generating semantically preserved synthetic inputs that increase the internal coverage of DNNs in order to guide the testing process in finding corner case behaviors and exposing hidden issues and inconsistencies. Through case studies with popular image recognition models trained on both MNIST and CIFAR-10 datasets, we show that DeepEvolution can improve the coverage of DNNs and successfully expose corner-cases behaviors. We also demonstrate that DeepEvolution can outperform the coverage-guided fuzzing approach Tensorfuzz in detecting latent defects introduced during the quantization of models.

CHAPTER 6 CONCLUSION

In this chapter, we summarise our findings and conclude the thesis. In addition, we discuss the limitations of our proposed approaches and outline some directions for future work.

6.1 Summary

Testing has been well-established for traditional software, with many methods and practices that have demonstrated their effectiveness in exposing latent bugs and consequently improve the trustworthiness of software systems. However, the data-driven paradigm of DL software, where the decision logic is inferred automatically via a statistical learning algorithm from training data, makes existing testing approaches inadequate to be directly applied, pushing researchers to adapt different concepts and techniques by deriving their correspondents for DL testing. Although the renovation of software testing methods for DL systems have shown prominent results and provided meaningful feedbacks on their software quality, the domain of quality assurance for DNN-based software is still immature.

In this thesis, we aim to assist DL engineers in testing their DNN-based applications throughout different phases of the development cycle. Therefore, we follow a systematic approach that consists of: (1) investigating DL software issues and testing challenges; (2) outlining the strengths and weaknesses of the software-based testing techniques adapted for DL systems; and (3) proposing novel testing solutions to fill some of the identified literature gaps. This thesis makes the following contributions: (1) we provide a practical guide and its related TF-based toolkit for detecting issues in DL training programs, and (2) we propose a search-based approach for DNN-based model testing that allows assessing the DNN's performance and robustness during the engineering phase and finding latent quantization defects during the deployment phase. In the following, we summarize the contributions of both solutions.

Property-based testing for DNN training programs. DL training program is the implementation of the iterative, computational training algorithm that improves, gradually, the DNN's parameters through multiple passes on data. Like any computer software, DL training program may contain coding bugs or configuration inconsistencies. Thus, we gather and formulate training issues that were identified by experienced researchers and practitioners. Then, we propose their corresponding verification mechanisms based on invariants and heuristics that can be automatically verified throughout the training execution to detect anomalies; alerting developers about the presence of deficiencies in their training program.

To assess the effectiveness of our guide for DL training verification, we incorporate these verification routines in a TF-based library, TFCheck, which have been evaluated on real world, mutants, and synthetic TF training programs. Obtained results show that TFCheck can significantly assist DL engineers in testing and debugging their DNN code implementation through its monitored training executions.

Search-based approach for testing DNN-based models. Model testing techniques assess the quality of the prediction of a model in terms of performance and robustness. Traditionally, testing data are collected from real-world applications and manually labeled by human experts. Recently, researchers have been proposing different testing approaches that are able to extend the human-crafted test oracles, by applying mutations and transformations on existing data instances to generate new synthetic ones. To infer the labels associated to the generated inputs, they leverage pseudo-oracle techniques, including differential testing and metamorphic testing. Equally important, they define test adequacy evaluation metrics, such as neuronal coverage criteria, to provide meaningful guidance for the testing data generator with the aim to optimize both testing coverage and effort. The main limitations of these existing approaches are their reliance on random fuzzing or transformations that do not always produce test cases with a good diversity. To improve over these limitations, we propose DeepEvolution, a search-based approach for testing DNN-based models. Search-based approaches formulate the coverage criteria as an objective function to optimize through inputs generation, but it relies on metaheuristic-based optimizers that are gradient-free and can be applied to generate testing inputs, maximizing a coverage-based fitness function and using a wide variety of transformations thanks to their flexibility (as it does not required prior assumptions). Given the impressive performance and widespread adoption of computer-vision DNNs, we propose an instantiation of DeepEvolution for testing image recognition DNN models, we the aim to illustrate its concrete applicability and assess its effectiveness on popular image recognition models trained on both MNIST and CIFAR-10 datasets. Results show that DeepEvolution can improve the coverage of DNNs and successfully expose corner-cases behaviors. It also outperformed the coverage-guided fuzzing approach Tensorfuzz in detecting latent defects introduced during the quantization of models.

6.2 Limitations of the proposed approaches

- Although TFCheck's verification routines have shown their effectiveness in detecting training issues and consequently improve the trustworthiness of DNN programs, these verifications are still coarse-grained. In fact, there is a many-to-many relation between

the fired checks and the possible issues; so DL engineers would encounter difficulties to identify the real cause behind the detected problems. As a consequence, fixing the identified issues may require multiple trials and errors steps.

- DeepEvoution evolves actively the population of candidate solutions based on up-to-date feedbacks on their test adequacy. This process requires estimating for each generated input the increment in neuronal coverage and the divergences in neurons' activations comparing to its original parent. This online testing process includes post-analysis tasks that are compute-intensive and time-consuming, which could hinder the applicability of proposed approach in real-world settings with large-scale datasets.

6.3 Future work

- We plan to develop more advanced verification routines that are able to not only alert about the presence of issues, but also to localize the buggy component or identify the inappropriate configuration element. We will also explore the possibility of incorporating decision branches in the verification logic, to predict the most probable root cause of an issue, given the DNN program state. Besides, since the training process is iterative, the verification mechanism could be performed during successive iterations by changing some program variables and evaluating the program response to this change; in order to narrow down the source of errors.

- We are encouraged by the impressive potential of DeepEvolution on image classification tasks; so we aim to extend the evaluation by adding more efficient mutation operations that mimic real-world camera defects having effects on the produced images. Additionally, we also plan to extend its applicability on other popular tasks in the deep learning domain, such as speech recognition and bioinformatics by studying some domain knowledge to define specific constraints for their particular inputs mutation.

- In future instances of DeepEvolution, we plan to leverage parallel and distributed development techniques to improve the efficiency of the post-analysis tasks and accelerate the framework processing by using GPU-based implementations of the metaheuristic algorithms and proposing a parallel version of our coverage analyzer component to avoid this switch to CPU sequential processing after computing the prediction by the DNN on GPU. We need to take advantage of the high-performance and high compute capability of the GPU processor by having an end-to-end GPU-based testing process for DNNs.

- Last but not least, we have presented our tools to some organisations, having high SQA requirements for DNN-based models. In our future research, we plan to validate our results on real-world DL software applications with the objective of improving the effectiveness and the usability of our tools.

REFERENCES

[1] A. Karpathy, "Software 2.0," *https://medium.com/@karpathy/software-2-0-a64152b37c35*, 2018.

[2] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. J. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Józefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. G. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. A. Tucker, V. Vanhoucke, V. Vasudevan, F. B. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *CoRR*, vol. abs/1603.04467, 2016.

[3] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized neural networks: Training neural networks with low precision weights and activations," *The Journal of Machine Learning Research*, vol. 18, no. 1, pp. 6869–6898, 2017.

[4] "Tflite," https://www.tensorflow.org/lite, accessed: 2019-04-02.

[5] E. J. Weyuker, "On testing non-testable programs," *The Computer Journal*, vol. 25, no. 4, pp. 465–470, 1982.

[6] C. Ziegler, "A google self-driving car caused a crash for the first time," *https://www.theverge.com/2016/2/29/11134344/google-self-driving-car-crash-report*, 2016.

[7] C. Metz, "After fatal uber crash, a self-driving start-up moves forward," *https://www.nytimes.com/2018/05/07/technology/uber-crash-autonomous-driveai.html*, 2018.

[8] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," 2018.

[9] T. T. D. Team, R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov *et al.*, "Theano: A python framework for fast computation of mathematical expressions," *arXiv preprint arXiv:1605.02688*, 2016.

[10] R. Collobert, S. Bengio, and J. Mariéthoz, "Torch: a modular machine learning software library," Idiap, Tech. Rep., 2002.

[11] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*. ACM, 2014, pp. 675–678.

[12] T. Y. Chen, S. C. Cheung, and S. M. Yiu, "Metamorphic testing: a new approach for generating next test cases," Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong, Tech. Rep., 1998.

[13] W. M. McKeeman, "Differential testing for software," *Digital Technical Journal*, vol. 10, no. 1, pp. 100–107, 1998.

[14] M. D. Davis and E. J. Weyuker, "Pseudo-oracles for non-testable programs," in *Proceedings of the ACM'81 Conference*. ACM, 1981, pp. 254–257.

[15] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, "Dynamically discovering likely program invariants to support program evolution," *IEEE Transactions on Software Engineering*, vol. 27, no. 2, pp. 99–123, 2001.

[16] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2011.

[17] J. Edvardsson, "A survey on automatic test data generation," in *Proceedings of the 2nd Conference on Computer Science and Engineering*, 1999, pp. 21–28.

[18] T. J. Ostrand and M. J. Balcer, "The category-partition method for specifying and generating fuctional tests," *Communications of the ACM*, vol. 31, no. 6, pp. 676–686, 1988.

[19] M. Ramachandran, "Testing software components using boundary value analysis," in *2003 Proceedings 29th Euromicro Conference*. IEEE, 2003, pp. 94–98.

[20] T. Chen, X.-s. Zhang, S.-z. Guo, H.-y. Li, and Y. Wu, "State of the art: Dynamic symbolic execution for automated test generation," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1758–1773, 2013.

[21] P. McMinn, "Search-based software testing: Past, present and future," in *Proceedings of the 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, ser. ICSTW '11. Washington,

DC, USA: IEEE Computer Society, 2011, pp. 153–163. [Online]. Available: http://dx.doi.org/10.1109/ICSTW.2011.100

[22] L. Bianchi, M. Dorigo, L. M. Gambardella, and W. J. Gutjahr, "A survey on metaheuristics for stochastic combinatorial optimization," *Natural Computing*, vol. 8, no. 2, pp. 239–287, 2009.

[23] S. Krishnan, J. Wang, E. Wu, M. J. Franklin, and K. Goldberg, "Activeclean: Interactive data cleaning while learning convex loss models," *arXiv preprint arXiv:1601.03797*, 2016.

[24] Z. Qi, H. Wang, J. Li, and H. Gao, "Impacts of dirty data: and experimental evaluation," *arXiv preprint arXiv:1803.06071*, 2018.

[25] P. McDaniel, N. Papernot, and Z. B. Celik, "Machine learning in adversarial settings," *IEEE Security & Privacy*, vol. 14, no. 3, pp. 68–72, 2016.

[26] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in machine learning systems," in *Advances in neural information processing systems*, 2015, pp. 2503–2511.

[27] J. Jo and Y. Bengio, "Measuring the tendency of cnns to learn surface statistical regularities," *arXiv preprint arXiv:1711.11561*, 2017.

[28] D. Sculley, T. Phillips, D. Ebner, V. Chaudhary, and M. Young, "Machine learning: The high-interest credit card of technical debt," 2014.

[29] S. Krishnan, J. Wang, M. J. Franklin, K. Goldberg, T. Kraska, T. Milo, and E. Wu, "Sampleclean: Fast and reliable analytics on dirty data." *IEEE Data Eng. Bull.*, vol. 38, no. 3, pp. 59–75, 2015.

[30] S. Krishnan, M. J. Franklin, K. Goldberg, and E. Wu, "Boostclean: Automated error detection and repair for machine learning," *arXiv preprint arXiv:1711.01299*, 2017.

[31] N. Hynes, D. Sculley, and M. Terry, "The data linter: Lightweight, automated sanity checking for ml data sets."

[32] S. Nidhra and J. Dondeti, "Black box and white box testing techniques-a literature review," *International Journal of Embedded Systems and Applications (IJESA)*, vol. 2, no. 2, pp. 29–50, 2012.

[33] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in neural information processing systems*, 2014, pp. 2672–2680.

[34] C. Doersch, "Tutorial on variational autoencoders," *arXiv preprint arXiv:1606.05908*, 2016.

[35] A. Fischer and C. Igel, "An introduction to restricted boltzmann machines," in *iberoamerican congress on pattern recognition*. Springer, 2012, pp. 14–36.

[36] R. D. Hjelm, A. P. Jacob, T. Che, A. Trischler, K. Cho, and Y. Bengio, "Boundary-seeking generative adversarial networks," *arXiv preprint arXiv:1702.08431*, 2017.

[37] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid, "Deeproad: Gan-based metamorphic autonomous driving system testing," *arXiv preprint arXiv:1802.02295*, 2018.

[38] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples (2014)," *arXiv preprint arXiv:1412.6572*.

[39] L. Engstrom, D. Tsipras, L. Schmidt, and A. Madry, "A rotation and a translation suffice: Fooling cnns with simple transformations," *arXiv preprint arXiv:1712.02779*, 2017.

[40] J. Gilmer, R. P. Adams, I. Goodfellow, D. Andersen, and G. E. Dahl, "Motivating the rules of the game for adversarial example research," *arXiv preprint arXiv:1807.06732*, 2018.

[41] S. Gu and L. Rigazio, "Towards deep neural network architectures robust to adversarial examples," *arXiv preprint arXiv:1412.5068*, 2014.

[42] S.-M. Moosavi-Dezfooli, A. Fawzi, and P. Frossard, "Deepfool: a simple and accurate method to fool deep neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 2574–2582.

[43] J. Wang, J. Sun, P. Zhang, and X. Wang, "Detecting adversarial samples for deep neural networks through mutation testing," *arXiv preprint arXiv:1805.05010*, 2018.

[44] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 1–18.

[45] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 303–314.

[46] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu *et al.*, "Deepgauge: multi-granularity testing criteria for deep learning systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 120–131.

[47] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial examples in the physical world," *arXiv preprint arXiv:1607.02533*, 2016.

[48] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*. IEEE, 2016, pp. 372–387.

[49] L. Ma, F. Zhang, M. Xue, B. Li, Y. Liu, J. Zhao, and Y. Wang, "Combinatorial testing for deep learning systems," *arXiv preprint arXiv:1806.07723*, 2018.

[50] C. Nie and H. Leung, "A survey of combinatorial testing," *ACM Computing Surveys (CSUR)*, vol. 43, no. 2, p. 11, 2011.

[51] IBM. Ibm cplex mathematical programming solver for linear programming. [Online]. Available: https://www.ibm.com/analytics/cplex-optimizer

[52] Y. Sun, X. Huang, and D. Kroening, "Testing deep neural networks," *arXiv preprint arXiv:1803.04792*, 2018.

[53] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, "A practical tutorial on modified condition/decision coverage," 2001.

[54] Y. Sun, M. Wu, W. Ruan, X. Huang, M. Kwiatkowska, and D. Kroening, "Concolic testing for deep neural networks," *arXiv preprint arXiv:1805.00089*, 2018.

[55] K. Sen, D. Marinov, and G. Agha, "Cute: a concolic unit testing engine for c," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 263–272.

[56] J. Guo, Y. Jiang, Y. Zhao, Q. Chen, and J. Sun, "Dlfuzz: differential fuzzing testing of deep learning systems," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 739–743.

[57] J. Kim, R. Feldt, and S. Yoo, "Guiding deep learning system testing using surprise adequacy," *arXiv preprint arXiv:1808.08444*, 2018.

[58] A. Karpathy. (2018) Cs231n: Convolutional neural networks for visual recognition. [Online]. Available: http://cs231n.github.io/neural-networks-3/

[59] C. Murphy, G. E. Kaiser, and L. Hu, "Properties of machine learning applications for use in metamorphic testing," 2008.

[60] C. Murphy, G. E. Kaiser, and M. Arias, "An approach to software testing of machine learning applications." in *SEKE*, vol. 167, 2007.

[61] X. Xie, J. W. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *Journal of Systems and Software*, vol. 84, no. 4, pp. 544–558, 2011.

[62] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The weka data mining software: an update," *ACM SIGKDD explorations newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[63] X. Xie, J. W. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *Journal of Systems and Software*, vol. 84, no. 4, pp. 544–558, 2011.

[64] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: an automated class mutation system," *Software Testing, Verification and Reliability*, vol. 15, no. 2, pp. 97–133, 2005.

[65] A. Dwarakanath, M. Ahuja, S. Sikand, R. M. Rao, R. Bose, N. Dubash, and S. Podder, "Identifying implementation bugs in machine learning based image classifiers using metamorphic testing," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2018, pp. 118–128.

[66] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao *et al.*, "Deepmutation: Mutation testing of deep learning systems," *arXiv preprint arXiv:1805.05206*, 2018.

[67] A. Odena and I. Goodfellow, "Tensorfuzz: Debugging neural networks with coverage-guided fuzzing," *arXiv preprint arXiv:1807.10875*, 2018.

[68] D. Selsam, P. Liang, and D. L. Dill, "Developing bug-free machine learning systems with formal mathematics," *arXiv preprint arXiv:1706.08605*, 2017.

[69] X. Xie, L. Ma, F. Juefei-Xu, H. Chen, M. Xue, B. Li, Y. Liu, J. Zhao, J. Yin, and S. See, "Coverage-guided fuzzing for deep neural networks," *arXiv preprint arXiv:1809.01266*, 2018.

[70] M. Sharif, S. Bhagavatula, L. Bauer, and M. K. Reiter, "Accessorize to a crime: Real and stealthy attacks on state-of-the-art face recognition," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2016, pp. 1528–1540.

[71] M. Alzantot, Y. Sharma, S. Chakraborty, and M. Srivastava, "Genattack: Practical black-box attacks with gradient-free optimization," *arXiv preprint arXiv:1805.11090*, 2018.

[72] R. B. Grosse and D. K. Duvenaud, "Testing mcmc code," *arXiv preprint arXiv:1412.5218*, 2014.

[73] S. Cai, E. Breck, E. Nielsen, M. Salib, and D. Sculley, "Tensorflow debugger: Debugging dataflow graphs for machine learning," 2016.

[74] L. Bottou. (2015) Multilayer neural networks. [Online]. Available: http://videolectures.net/site/normal_dl/tag=983658/deeplearning2015_bottou_neural_networks_01.pdf

[75] A. Rakitianskaia and A. Engelbrecht, "Measuring saturation in neural networks," in *2015 IEEE Symposium Series on Computational Intelligence*, 2015, pp. 1423–1430.

[76] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[77] S. Cai, E. Breck, E. Nielsen, M. Salib, and D. Sculley, "Tensorflow debugger: Debugging dataflow graphs for machine learning," in *Proceedings of the Reliable Machine Learning in the Wild-NIPS 2016 Workshop*, 2016.

[78] K. Wongsuphasawat, D. Smilkov, J. Wexler, J. Wilson, D. Mané, D. Fritz, D. Krishnan, F. B. Viégas, and M. Wattenberg, "Visualizing dataflow graphs of deep learning models in tensorflow," *IEEE transactions on visualization and computer graphics*, vol. 24, no. 1, pp. 1–12, 2018.

[79] P. S. Foundation. (2013) Mutpy 0.4.0. [Online]. Available: https://pypi.python.org/pypi/

[80] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010, pp. 249–256.

[81] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep learning with limited numerical precision," in *International Conference on Machine Learning*, 2015, pp. 1737–1746.

[82] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplications," *arXiv preprint arXiv:1412.7024*, 2014.

[83] A. Voulodimos, N. Doulamis, A. Doulamis, and E. Protopapadakis, "Deep learning for computer vision: A brief review," *Computational intelligence and neuroscience*, vol. 2018, 2018.

[84] Z. Wang, A. C. Bovik, H. R. Sheikh, and E. P. Simoncelli, "Image quality assessment: from error visibility to structural similarity," *IEEE transactions on image processing*, vol. 13, no. 4, pp. 600–612, 2004.

[85] Y.-C. Ho and D. L. Pepyne, "Simple explanation of the no-free-lunch theorem and its implications," *Journal of optimization theory and applications*, vol. 115, no. 3, pp. 549–570, 2002.

[86] J. H. Holland, "Genetic algorithms," *Scientific american*, vol. 267, no. 1, pp. 66–73, 1992.

[87] R. Eberhart and J. Kennedy, "A new optimizer using particle swarm theory," in *Micro Machine and Human Science, 1995. MHS'95., Proceedings of the Sixth International Symposium on.* IEEE, 1995, pp. 39–43.

[88] X.-S. Yang and S. Deb, "Cuckoo search via lévy flights," in *Nature & Biologically Inspired Computing, 2009. NaBIC 2009. World Congress on.* IEEE, 2009, pp. 210–214.

[89] X.-S. Yang, "Firefly algorithm, stochastic test functions and design optimisation," *arXiv preprint arXiv:1003.1409*, 2010.

[90] ——, "A new metaheuristic bat-inspired algorithm," in *Nature inspired cooperative strategies for optimization (NICSO 2010).* Springer, 2010, pp. 65–74.

[91] S. Mirjalili, S. M. Mirjalili, and A. Lewis, "Grey wolf optimizer," *Advances in engineering software*, vol. 69, pp. 46–61, 2014.

[92] S. Mirjalili, "Moth-flame optimization algorithm: A novel nature-inspired heuristic paradigm," *Knowledge-Based Systems*, vol. 89, pp. 228–249, 2015.

[93] S. Mirjalili and A. Lewis, "The whale optimization algorithm," *Advances in Engineering Software*, vol. 95, pp. 51–67, 2016.

[94] S. Mirjalili, S. M. Mirjalili, and A. Hatamlou, "Multi-verse optimizer: a nature-inspired algorithm for global optimization," *Neural Computing and Applications*, vol. 27, no. 2, pp. 495–513, 2016.

[95] S. Mirjalili, A. H. Gandomi, S. Z. Mirjalili, S. Saremi, H. Faris, and S. M. Mirjalili, "Salp swarm algorithm: A bio-inspired optimizer for engineering design problems," *Advances in Engineering Software*, vol. 114, pp. 163–191, 2017.

[96] Y. LeCun, "The mnist database of handwritten digits," *http://yann. lecun. com/exdb/mnist/*, 1998.

[97] A. Krizhevsky, V. Nair, and G. Hinton, "The cifar-10 dataset," *http://www.cs.toronto.edu/kriz/cifar.html*, 2014.

[98] "Variant of lenet," https://github.com/tensorflow/models/blob/master/research/slim/nets/lenet.py, accessed: 2019-04-02.

[99] "Cifar10," https://github.com/tensorflow/models/blob/master/research/slim/nets/cifarnet.py, accessed: 2019-04-02.

[100] "Configuration of lenet," https://github.com/tensorflow/models/blob/master/research/slim/scripts/train_lenet_on_mnist.sh, accessed: 2019-04-02.

[101] "Configuration of cifar10," https://github.com/tensorflow/models/blob/master/research/slim/scripts/train_cifarnet_on_cifar10.sh, accessed: 2019-04-02.