POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Understanding the Impact of Poor Coding Practices on the Quality of Deep Learning Systems

HADHEMI JEBNOUN
Département de génie informatique et génie logiciel

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

Décembre 2020

POLYTECHNIQUE MONTRÉAL
affiliée à l'Université de Montréal

Ce mémoire intitulé :

Understanding the Impact of Poor Coding Practices on the Quality of Deep Learning Systems

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

DEDICATION

To my family who always supported me ...

ACKNOWLEDGEMENTS

This thesis could not be realized without the assistance of several individuals that I would like to thank.

First, I would first like to express my gratitude to my adviser, Dr. Foutse Khomh, for everything he has done for me throughout my master's studies.

Secondly, I also like to thank Masud Rahman and Md Saidur Rahman who help me with writing the conference and journal papers in this thesis.

Special thanks to Houssem Ben Braiek for his help and suggestions during my master's studies. We, also, have been co-authoring the first paper integrated into this thesis.

I would like to thank all of my friends and members of the computer engineering and software engineering department (GIGL).

I express my gratefulness to my family members and relatives for their support and inspiration.

Finally, I would like to thank my committee members, Dr. Michel C. Desmarais, Dr. Heng Li, and Dr. Foutse Khomh for evaluating my master's thesis.

RÉSUMÉ

Les applications basées sur l'apprentissage profond, en anglais Deep Learning (DL), sont de plus en plus utilisées pour résoudre diverses tâches de notre quotidien grâce aux récentes prouesses des modèles d'apprentissage profond qui surpassent déjà les compétences humaines dans un large éventail de tâches, de la classification des images à la reconnaissance de la parole et au traitement du langage naturel. Ces progrès tendent à élargir l'application de l'apprentissage profond dans des domaines aussi critiques en termes de sécurité comme les voitures autonomes et la santé. Les spécialistes de l'apprentissage profond partagent les mêmes préoccupations que les ingénieurs logiciels d' autres domaines en ce qui concerne l'efficacité, la complexité et la maintenabilité des systèmes logiciels. En revanche, le processus de développement continu des systèmes d'apprentissage profond, caractérisé par un rythme rapide et une complexité accrue, pourrait conduire à de mauvais choix de conception par le développeur. De plus, en raison de l'utilisation prédominante de Frameworks similaires et du codage répétitif de tâches similaires, les développeurs de systèmes d'apprentissage profond ont donc tendance à recourir à la pratique du copier-coller, générant ainsi des clones dans le code d'apprentissage profond.

La plupart des travaux de recherche sur l'apprentissage profond se sont axés plus particulièrement sur l'amélioration de la précision du modèle, mais à notre connaissance, il n'existe pas de travaux de recherche qui étudient les odeurs de code et en particulier les pratiques de duplication de code dans le cadre du développement de l'apprentissage profond. Compte tenu de l'impact négatif des odeurs de code et des clones de code sur la qualité des logiciels, tel que signalé dans de nombreuses études réalisées sur les systèmes traditionnels, et de la complexité inhérente à la maintenance des systèmes d'apprentissage profond, comme le test et la correction des bugs, il est indispensable de mener des études sur les odeurs de code dans les systèmes d'apprentissage profond. Dans ce mémoire, nous étudions les odeurs de code et les clones de code, qui sont également une sorte d'odeur de code dans les applications d'apprentissage profond.

Premièrement, nous menons une étude empirique pour investiguer la distribution, l'évolution et la propension aux bugs des odeurs de code dans les applications d'apprentissage profond. À cette fin, nous effectuons une analyse comparative entre les applications d'apprentissage profond et les applications open-source traditionnelles collectées à partir de GitHub. Deuxièmement, nous analysons la fréquence, la distribution et l'impact des clones de code et des pratiques de duplication de code dans les systèmes d'apprentissage profond. Pour ce faire, nous utilisons l'outil de détection de clones NiCad pour détecter les clones de 59 systèmes d'apprentissage profond et de 59 systèmes logiciels traditionnels. Nous analysons ensuite la fréquence et la distribution des clones de code dans les systèmes d'apprentissage profond en les comparant à ceux des systèmes traditionnels. De plus, nous étudions la distribution des clones de code détectés en appliquant une taxonomie basée sur la localisation. En outre, nous étudions la corrélation entre les bugs et les clones de code pour évaluer le risque potentiel que pose ces clones. Enfin, nous introduisons une taxonomie des clones de code liée au code d'apprentissage profond et nous découvrons les phases les plus risquées du développement des systèmes d'apprentissage profond. Nous avons établi plusieurs résultats intéressants. Premièrement, les odeurs de code *longue expression lambda*, *longue expression conditionnelle ternaire*, et *compréhension complexe des conteneurs* sont fréquemment observées dans les projets d'apprentissage profond. Autrement dit, le code d'apprentissage profond contient des expressions plus complexes ou plus longues que le code traditionnel. Deuxièmement, le nombre d'odeurs de code augmente au cours du temps dans les applications d'apprentissage profond. Troisièmement, nous avons constaté une coexistence entre les odeurs de code et les bugs logiciels dans le code d'apprentissage profond examiné, ce qui confirme notre hypothèse sur la dégradation de la qualité du code des applications d'apprentissage profond. En ce qui concerne l'étude des clones de code, nos résultats montrent que la duplication de code est une pratique fréquente dans les systèmes d'apprentissage profond et que les développeurs de systèmes logiciels utilisant l'apprentissage profond ont l'habitude de dupliquer le code dans des endroits plus éloignés (exemple, dans des dossiers différents).

Nous présentons également une taxonomie des clones de code d'apprentissage profond afin de mieux comprendre quand un code d'apprentissage profond est dupliqué. Enfin, nous montrons que la définition des hyperparamètres du modèle d'apprentissage profond est la tâche la plus risquée lors de la construction du modèle, car conduisant fréquemment à des erreurs.

ABSTRACT

Deep Learning (DL) based applications are increasingly being used in our society to solve a variety of tasks, thanks to the recent progress of deep learning models, which are now outperforming humans on a wide range of tasks, from image classification to speech recognition and natural language processing. This progress is being made towards the widespread application of DL in safety-critical applications such as autonomous cars and healthcare.

Deep learning practitioners share similar concerns as software engineers in other domains with regards to efficiency, complexity, and maintainability. On the other hand, the continuous development of deep learning systems which takes place at a rapid pace as well as their increasing complexity could lead to bad design choices on the part of the developers. Furthermore, due to the prevalent use of similar frameworks and repeated coding of similar tasks, deep learning developers, therefore, tend to use copy-paste practice, creating clones in deep learning code. The majority of research in deep learning has focused on improving the accuracy of the model, and to the best of our knowledge, there hardly exists any research that studies code smells and in particular code cloning practices in deep learning development. Given the negative impacts of code smells as well as code clones on software quality reported in the studies on traditional systems and the inherent complexity of maintaining deep learning systems (testing and fixing bugs is challenging), it is very important to study code smells in deep learning systems.

In this thesis, we study code smells and code clones (which is also a kind of code smell) in deep learning applications. Firstly, we conduct an empirical study to understand the distribution, evolution, and bug-proneness of code smells in deep learning applications. To this end, we perform a comparative analysis between deep learning and traditional open-source applications collected from GitHub. Secondly, we analyze the frequency, distribution, and impacts of code clones and the code cloning practices in deep learning systems. We achieve this by using the NiCad clone detection tool to detect code clones in 59 deep learning and 59 traditional software systems. We then analyze the comparative frequency and distribution of code clones in deep learning systems and the traditional ones. Further, we study the distribution of the detected code clones by applying a location-based taxonomy. Besides, we study the correlation between bugs and code clones to assess the risk of code cloning. Lastly, we introduce a code clone taxonomy related to deep learning code and identify the most risky phases in the development process of deep learning systems.

We have several major findings in this thesis. First, *long lambda expression*, *long ternary* *conditional expression*, and *complex container comprehension* smells are frequently found in deep learning software systems. That is, the deep learning code involves more complex or longer expressions than the traditional code does. Second, the number of code smells increases across the releases of deep learning applications. Third, we found that there is a co-existence between code smells and software bugs in the studied deep learning applications, which confirms our conjecture on the degraded code quality of deep learning applications. In terms of code clones, our results show that code cloning is a frequent practice in deep learning systems and that deep learning developers often clone code from files contained in distant repositories in the system. We further present a taxonomy of clones in deep learning code, to provide a deeper understanding of when deep learning code is cloned. Finally, we show that setting hyperparameters of the deep learning model is the riskiest task during the model construction phase, since it often leads to faults.

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
Trad	Traditional
SE	Software Engineering
SQA	Software Quality Assurance
LPL	Long Parameter List
LM	Long Method
LSC	Long Scope Chaining
LC	Large Class
LMC	Long Message Chain
LBCL	Long Base Class List
LLF	Long Lambda Function
LTCE	Long Ternary Conditional Expression
CCC	Complex Container Comprehension
MNC	Multiply-Nested Container
SF	Same File
SD	Same Directory
DD	Different Directories
STD	Standard Deviation

LIST OF APPENDICES

CHAPTER 1 INTRODUCTION

In recent years, the development of Deep Learning (DL) systems has shown great progress. This success is mainly due to the availability of vast amounts of training data as well as to the advances in hardware and software for computation. However, since DL systems have significant differences from traditional software systems, it is necessary to take into consideration their distinctive characteristics when developing them. For instance, DL systems rely heavily on DL libraries/frameworks and code reuse (due to common development logic) for the majority of tasks. Dedicated software quality approaches and benchmarking efforts should be devoted to DL systems, considering their specificities of use, the constraints of the resources that they consume, and the capabilities that they enable. Furthermore, because of the popularity of deep learning applications, practitioners are often pressured to rapidly develop, maintain, and improve these applications. Software quality concerns are unfortunately not prioritized by today's DL practitioners. Rather, they focus on addressing a given problem and mostly on improving the accuracy of the model. As a result, DL developers may adopt poor design and implementation practices, known as code smells. The presence of code smells may negatively influence the evolution of DL systems and degrade their software quality. This thesis examines the code quality of deep learning systems in-depth, by detecting code smells and code clones and investigating the distribution, evolution, and bug-proneness of code smells and the distribution (in terms of occurrences) and locations of code clones in DL code. We also examine the reasons behind code duplicating and identify the risk of bug introduction in duplicated code through out the different development phases. The main goal of this thesis is to improve our understanding of coding practices in deep learning systems and their impact on quality.

1.1 Deep Learning based Software

Deep learning (DL) is a subset of machine learning (ML), which in turn is a subset of artificial intelligence (AI), the science of mimicking human capabilities by machines. Deep learning is part of a broader family of machine learning methods inspired by the mechanism and structure of the human brain, a network of billions of neurons. This structure is simulated by networks of artificial neurons known as artificial neural networks (ANN). Deep learning networks are artificial neural networks with more than three layers. Deep learning enables computers to build concepts from data based on simpler concepts [2]. The field of Deep Learning is currently revolutionizing almost every industry in countless ways and is having a gigantic impact on domains like healthcare, communication, transport, finance, etc.

DL models have high learning capacity that allows them to capture increasingly complex patterns directly from data without the need of handcrafted feature engineering [3]. Traditionally, software systems are constructed deductively by writing down the rules that govern the behavior of the system as program code. However, with deep learning, these rules are inferred from training data and they are generated inductively. This consequently reduces the considerable manual work required to craft hand-made features that are required for classical machine learning approaches. DL models also feature more sophisticated architecture than other models, capturing long-range dependencies and modeling them with data.

1.2 Code smells and Clones in DL systems

The main purpose of deep learning is to construct models with high performance that is able to learn knowledge from the input data in order to make predictions for new data. Producing quickly powerful deep learning models may lead to undermining the quality of the system. Based on the experience of Google, Sculley et al [4] highlighted the technical debt of developing quickly machine learning systems which consequently leads to difficult and expensive maintenance. Actually, ML-based systems hold the maintenance concerns of traditional systems as well as those specific to ML. Sculley et al [4] also mentioned that building such incredible deep learning algorithms requires sound software engineering practices.

To find the optimal model, deep learning practitioners usually implement multiple prototypes quickly, by experimenting with different configurations. They then compare the performance of the different models to identify the best configuration leading to the most efficient model. And as DL developers may have to follow the same or similar steps to build models without or with some modifications, they often end up writing poor code and duplicated functions or blocks known as clones. The difference between the cloned fragments derives from the dissimilarity between models' architectures and also from model (hyper)parameters settings or initialization. Code clones also stem from code reuse. Indeed, reusing code with or without modification by copying and pasting fragments from one location to another is a common practice during software development and maintenance activities, including for deep learning code [5].

The existence of code smells [6] in deep learning systems may hamper their maintenance and the evolution. Code smells are violations of fundamental design principles. In traditional systems, they negatively impact software quality [7] and tend to increase technical debts, and consequently incur additional development and maintenance costs. They may also increase the consumption of certain resources (processor, memory, etc.) [8, 9] and, consequently, hinder the deployment of efficient and sustainable solutions.

Earlier studies on traditional software systems show that about 7%-23% of code in the software repositories are cloned code [10]. The intuitive benefit of clones is the productivity gain by code reuse. Some earlier studies reported positive impacts of clones [11] and showed that clones are not harmful [12, 13]. However, there are substantial empirical evidences showing that clones can negatively impact software quality by affecting the stability of the code [**? ? ?**] and making it bug-prone [14, 15, 16, 17, 18, 19, 20], and consequently adding complexities and costs to software maintenance. To leverage the benefits of code reuse while consciously avoiding possible issues, developers should be aware of clones and manage clones properly to ensure their consistent evolution [21].

1.3 Thesis Statement

Although code smells and code cloning practices, as well as their impacts, have been widely investigated for traditional software systems, we know little about the coding practices in machine learning based systems despite the recent upsurge in the development of machine learning ; in particular deep learning based systems. No study to date has examined the code quality of DL software systems by studying the distribution and impacts of code smells, and by investigating the risks associated with their existence in the DL systems. We aim to fill in this gap by empirically investigating code smells and cloning practices in deep learning systems.

It is important to understand the extent to which DL practitioners are following good coding practices (avoiding code smells) and the impact that poor coding practices has on the quality of DL systems. To the best of our knowledge, this thesis presents the first empirical study on DL code smells.

1.4 Thesis Overview

In this thesis, we provide two empirical studies. We, first, investigate the prevalence, evolution, and bug-proneness of code smells in deep learning systems. Second, we study the distribution (in terms of occurrences), location, and bug-proneness of code clones. In addition, we propose a DL-related taxonomy of code clones and identify the phase in the development process of DL code in which cloning is the most risky. In the following, we present the details of each study :

1. *An empirical study of the scent of deep learning code.* We conduct an empirical study on the quality of deep learning code using 118 open-source software systems. By collecting a total of 59 deep learning systems and 59 traditional software systems from GitHub. First, we determine the distribution of 10 code smells in the deep learning systems,

and then compare them with the smells from the traditional systems. Second, we investigate the trend of code smells found in the deep learning projects over time. We detect the code smells from each of their releases and then analyse the changes in smell instances across the releases. Third, we analyse the commits of deep learning projects to determine the co-existence between code smells and software bugs within DL code. We first separate the bug-fixing commits using their labels and extract the files that were changed to fix the bugs. Then we determine whether these changed files contain code smells in our deep learning applications. We also analysed the time spent for fixing the buggy code in DL systems.

2. *Clones in deep learning code : what, Where, and Why ?* We introduce the first empirical study on DL code clones, where we perform a comparative study of the distribution and bug-proneness of clones in deep learning and traditional systems. We analyze clones in 59 deep learning systems and 59 traditional systems and compare the distribution of clones from the perspectives of clone types and their locations. To gain further insights into the reasons behind developers' code cloning practices in the studied deep learning systems, we randomly select six deep learning project and perform a manual analysis of their code clones. We build a deep learning code clones taxonomy in which we assign each detected code clones to the corresponding deep learning phase. We further study the relation between bug-proneness and code cloning in the context of deep learning systems. Finally, we identify the riskier activities in the development process of deep learning systems, by analyzing the distribution of DL-related buggy clones across the development phases.

1.5 Thesis Contribution

In this thesis, we conduct an empirical investigation of the coding practices of DL developers, with the aim of understanding to what extent deep learning practitioners use best practices while implementing their solutions. Our analysis led to the following observations :

— We found no statistically significant difference between the code smell occurrences in deep learning projects and that in the traditional ones. When types of smells were considered, we found that *Long Lambda Function*, *Long Ternary Conditional Expression* and *Complex Container Comprehension* are more frequent within the deep learning code than within the traditional source code.

— Our analysis suggests that the number of code smells increases across the releases in the deep learning applications.

— We found that 62.84% of the changed files in DL systems overlap with the smelly code. Furthermore, frequent smells (e.g., *Long Ternary Conditional Expression*, *Long*

Parameter List) co-exist with software bugs more often than the others. We found that the buggy code from the DL applications needs more time to fix when it is smelly than when it is not smelly.

— Cloning is frequent in deep learning code, we found that code clones occurrences in deep learning code is higher than in traditional code. All three clone types (Type 1, Type 2, and Type 3) are more prevalent in deep learning systems than in traditional systems.
— Fragments of code clones in deep learning systems are found dispersed. We found that the majority of code clones in deep learning code are located in different files (in the same directory or in different directories).
— Code clones in deep learning systems are likely to be more bug-prone compared to non-cloned code and Type 3 clones are at higher risk of bugs compared to other clone types.
— We classify code clones by deep learning phases and found that three main deep learning phases are more prone to code cloning : model construction (36.08%), model training (18.56%), and data preprocessing (18.56%).
— The following six deep learning phases contain the majority of buggy clones : model construction (50%), model training (20%), data collection (13.3%), data preprocessing (10%), data post-processing (3.3%), and hyper parameter tuning (3.3%). This makes these deep learning phases riskier from a code cloning point of view.

1.6 Organization of the Thesis

The remainder of this thesis is organized as follows :

— Chapter 2 introduces the fundamental concepts and methods, that help to understand our research work, and which are related to deep learning development, code smells, and code clones.
— Chapter 3 outlines a comprehensive review of software quality assurance in deep learning systems, and of the impact of code clones and code smells on software quality.
— Chapter 4 presents our first empirical study, in which we investigate the distribution, evolution, and bug-proneness of code smells in deep learning systems.
— Chapter 5 presents our second empirical study on code clones in deep learning systems.
— Chapter 6 provides a summary and conclusion of the thesis and discusses future work.

CHAPTER 2 BACKGROUND

In this chapter, we introduce the necessary background knowledge on deep learning development, code smells, and code clones. This will be useful to follow the rest of this thesis.

Chapter Overview : Section 2.1 introduces the concepts and the development workflow of Deep learning systems. Section 2.2 describes the 10 code smells analyzed in this thesis. Section 2.3 defines the terminologies of code clones. Finally, Section 2.4 concludes the chapter.

2.1 Deep Learning

In this section, we define deep learning and discuss the different phases of DL systems development life cycle.

Deep Learning (DL) is a sub-domain of Machine Learning (ML) involving multiple layers of neural networks to provide a powerful model with the ability to learn from data. Machine learning techniques other than deep learning relied heavily on feature engineering. Whereas deep learning is capable to learn representation from raw data which makes deep learning a powerful ML technique. Thanks to the increase in the amount of data available and the advancement in computer infrastructure both hardware and software, deep learning has earned growing popularity these days and has dealt with complex applications with increasing accuracy over time [2]. Deep learning has been applied in various fields and in several activities and services in daily life including transportation, health, and finance. Deep learning has contributed to several other domains such as computer vision, speech recognition, machine translation, software engineering, etc.

There is an abundant literature on the software quality assurance of traditional software systems [22, 23, 24, 25, 26]. However, very few studies have investigated quality issues in deep learning software systems. Yet, deep learning software development faces all the development and maintenance challenges of traditional software systems, in addition to specific challenges related to their dependence to data, their inductive nature, and the difficulty to understand their behavior. To help developers create reliable deep learning systems efficiently, it is important to understand their design and implementation practices and how these practices impact the quality of DL software systems. In this thesis, we examine the prevalence and impacts of code smells as well as code clones in deep learning code. In the following, we describe the steps followed to construct a deep learning model. We adopted this development phases from the workflow described by Han et al. [27]. This workflow was inferred from an analysis of various deep learning frameworks (e.g, Tensorflow, Pytorch, and Theano).

We add the data post-processing and the data collection phases to this workflow to get a more accurate classification of code clones regarding development activities. We provide our conjectures of why deep learning practitioners may duplicate code to perform each step of this workflow. Regarding other code smells, we assume that they may exist in every step of the deep learning workflow, since they correspond to poor coding practices on lines and functions or classes, regarding their complexity and length.

Figure 2.1 shows the steps of this workflow that we detail in the following. We present 9 steps represented in a linear diagram, however, deep learning workflows are non-linear and may contain several feedback loops [28].

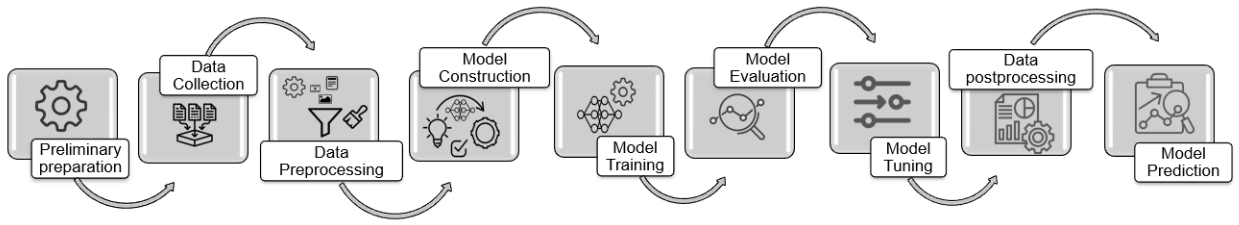

Figure 2.1 Deep Learning Workflow

Preliminary preparation : It is the phase in which developers prepare the environment, resolve installation issues, frameworks/libraries versions, configuring hardware requirements (CPU, GPU management), etc. This initial step may be prone to code clones since if deep learning developers use different models, each model requires a specific configuration. These configurations are likely to be same or similar for closely similar models. Consequently, we may find exact duplication and near-miss clones in the code used to perform this task. In addition, to parallelize data preparation, setting the number of cores, and the number of threads per core, developers may duplicate code with different values. Hence, the spread of code duplication.

Data collection : Deep learning practitioners start by gathering data required to meet the business problem. The collection could be from available datasets whether open source or internal. This step could be done by reading file(s) from the disk, or by calling a rest API, or by using data collector functionalities provided by the deep learning frameworks. Whenever deep learning practitioners need to collect data, they will use the same or similar call or logic by modifying only the source paths when it comes to different data locations. They may end up duplicating code to get the data.

Data preprocessing : Once the dataset is selected and collected, data should be prepared as input to the chosen architecture of the models. Each model requires an input of specific characteristics regarding size, shape, format, and data type. This phase is done before model training and it depends on the required input for each model. It is common for each model and the developers may opt for code reuse by copy-paste to implement common functions leading to duplicate code. Even if those are not exact clones, those will be near-miss clones as they have the same or closely similar algorithmic logic. After preprocessing the data to be suitable to model learning, the dataset is split into three different subsets as follows : training data, validation data, and test data. As for splitting data, it is a common practice used by DL practitioners. The majority of deep learning frameworks provide ready to use functions to perform those tasks. Calling those functions to collect data with or without modifications for each specific need is likely to introduce clones in deep learning code.

Model Construction : In this step, deep learning developers construct and configure the DL model with the chosen architecture. Then comes the hyperparameters set up and the selection of activation functions, loss functions, and model optimizers. Another option is to use pretrained models that are available from online sources or load them from disk storage. This practice is used to speed up model construction and training steps. This phase is considered as the most crucial phase in deep learning development since it is dedicated to the issues related to the model itself : the choice of the model [27]. The setup of deep learning models has common steps. These steps are blocks of code that are common between models, and each performs a sequence of calls to DL routines. And due to the use of the same frameworks and libraries, the DL code can have duplicated blocks of code between functions or cloned functions.

Model training : Once the model implementation and data preparation are complete, the model is ready to be trained. The training process updates the parameters iteratively to minimize the loss, i.e., the prediction error. At the end of the training, the model is generated with better accuracy and performance [27]. Since most of DL models may share the same algorithmic logic and share some computational functions (e.g., loss function, activation function), the deep learning code may have duplicated code fragments that are either exact or near-miss clones.

Model evaluation : At this level, we have a trained model that is ready to be evaluated on its performance. Thus, deep learning practitioners need the validation data set, that was hidden from the model during training, for evaluation. The evaluation of the model is frequently done by visualizing the performance metrics of the trained model ; assessing the changes to the loss function, model accuracy, etc [27]. Koenzen et al. [29] have shown that code snippets related to visualization tend to have a duplication rate of up to 21% in Jupyter notebook. Evaluating deep learning models is an integral part of the DL development process. It allows developers to find the best model. This step gives a better idea of how well the model may perform on unseen data. This is an essential phase in the model construction process, hence one can find the code for model evaluation in each deep learning project. Their logic is similar if not exact. Hence, deep learning code may contain duplicated functions or blocks of code used to perform models evaluations.

Model tuning : To optimize the performance of the model, hyper-parameters are tuned. Using an unsuitable loss function, initializing wrongly the weights, or choosing an inappropriate learning rate will negatively affect the model's performance. This step is empirical, it is usually done through a trial and error process that aims to compute the optimum values of models hyper-parameters (e.g, Grid Search technique [30]). Hyper-parameter tuning is an essential phase to improve the model's performance. The implementation of this technique is either developed by deep learning practitioners or by invoking ready to use optimization functions from a framework. Because these functions have the same logic, calling the same or similar set of framework routines to perform this task will likely result in duplicated code blocks or functions in the deep leaning code.

Data postprocessing : After an inductive process of learning from raw data, the output of the model may not be well-suited to represent the prediction results in an application specific and user-interpretable from. Hence, prediction results should be post-processed to be more meaningful and informative to end-users. This phase is frequently used in object detection, where the code interprets output by assigning the class with a higher probability to each object or by drawing the resulted bounding boxes on an image. If models have a common objective, such as detecting objects, they may induce code duplication, during data post-processing.

Model prediction : After training the model or using a pretrained model, the model is ready to make a prediction for new given data. The model prediction is implemented frequently as a function named *predict* and a call to this function. When using the same logic, steps of implementation, and renaming strategy, the deep learning model prediction code may have duplicated code blocks or functions.

2.2 Code Smells

In software engineering, the term code smell was first coined by Kent Beck [31], to describe symptoms in the source code of an application that indicate poor design or implementation choices [32]. These smells do not prevent the program from working. However, they are a violation of the best practices that may increase the risk of software bugs or failures in the future.

Our study is based on an existing Python-based tool named Pysmell [33]. We consider 10 code smells for our study, as were considered by Chen et al. [1]. To determine thresholds for code smells, we use an experience-based strategy where the thresholds are defined by 101 experienced developers with 4-10 years of experience. They are also active contributors on popular Python projects from GitHub. We select five code smells that are related to object-oriented programming and another five code smells that were defined by Chen et al. [1] from their analysis of bad coding patterns in real world python systems. Below is the list of the 10 selected code smells.

Long Parameter List (LPL) [34] : A method or a function that has a large number of parameters.

Long Method (LM) [34] : A method or a function that is extremely long.

Long Scope Chaining (LSC) [34] : A method or a function that has a deep nested closure.

Large Class (LC) [34] : A class that has a large number of source code lines.

Long Message Chain (LMC) [35] : An expression for accessing an object using the dot operators through a long sequence of attributes or method calls.

Long Base Class List (LBCL) [1] : When a class extends too many base classes due to the multiple inheritances that Python language supports, it makes code hard to understand.

Long Lambda Function (LLF) [1] : An anonymous function that is extremely long and complex in terms of conditions and parameters.

Long Ternary Conditional Expression (LTCE) [1] : A ternary conditional expression that is extremely long.

Complex Container Comprehension (CCC) [1] : One-line comprehension list, set or dictionary that contains a large number of clauses and filter expressions.

Multiply-Nested Container (MNC) [1] a container (including set, list, tuple, dict) that is deeply nested.

2.3 Code Clones

In this section, we briefly discuss the concepts and terminology of code clones. Next, we give an overview of the taxonomy of code clones, common causes behind code cloning, and the impacts of clones on software systems based on existing literature.

2.3.1 Code clones terminologies

Code clones are exact or similar copies of code fragments usually created by copying and pasting code fragments for code reuse. It can be similar code fragments, with renamed or added lines. The code fragment is identified by its file name, start line number, and end line number. Code clones could be detected by pair or by class.

Clone pair : Clone detection result is represented by pairs of fragments. Two fragments that are clones to each other form a clone pair.

Clone class : Code clones are detected by classes. Each class contains a set of fragments that are clones to each other.

In our study, we detect code clones by class.

2.3.2 Clone Taxonomies

In our study, we are interested in exploring two kinds of clone taxonomies, similarity-based clone taxonomy and location-based clone taxonomy. An explanation of both of them will be provided in the next two subsections.

2.3.2.1 Similarity-based Clone Taxonomy

Basically, there are two kinds of similarities between the two code fragments : functional (semantic) and textual (syntactic).

Textual similarity is when a copied fragment is used with or without minor modification. There are three types of syntactically similar clones :

— ***Type 1 :*** identical code clones except for differences in white-spaces, layouts and comments. It is known as exact clones. Table 2.1 presents an example of two fragments of code clones where the difference between them is the comment highlighted in grey. The pair of code fragments are exact copies of each other. Hence, they are clones of Type 1.

— ***Type 2 :*** syntactically identical code clones except for differences in identifiers name, data types, whitespace, layouts, and comments are Type 2 clones. As shown in Table 2.2, we ignore the renaming differences (function name, name of input variable). These two code fragments are Type 2 clones of each other.

— ***Type 3 :*** Code clones with some modification, addition or deletion of lines in addition to a difference in identifiers, data types, whitespaces, and comments. An example of two code fragments that are Type 3 clones of each other is displayed in Table 2.3. These two code fragments are different in the function name and the addition of 2 lines for another condition in the second code fragment.

Table 2.1 Type 1 Clones

```
def forward_activation(self, X):
    #compute post activation value of X
    if self.activation_function == "sigmoid":
        return 1.0/(1.0 + np.exp(-X))
    elif self.activation_function == "tanh":
        return np.tanh(X)
    elif self.activation_function == "relu":
        return np.maximum(0,X)
    elif self.activation_function == "leaky_relu":
        return np.maximum(self.leaky_slope*X,X)
```

```
def forward_activation(self, X):
    if self.activation_function == "sigmoid":
        return 1.0/(1.0 + np.exp(-X))
    elif self.activation_function == "tanh":
        return np.tanh(X)
    elif self.activation_function == "relu":
        return np.maximum(0,X)
    elif self.activation_function == "leaky_relu":
        return np.maximum(self.leaky_slope*X,X)
```

Table 2.2 Type 2 Clones

```
def forward_activation_fct(self, X):
    if self.activation_function == "sigmoid":
        return 1.0/(1.0 + np.exp(-X))
    elif self.activation_function == "tanh":
        return np.tanh(X)
    elif self.activation_function == "relu":
        return np.maximum(0,X)
```

```
def forward_activation(self, input):
    if self.activation_function == "sigmoid":
        return 1.0/(1.0 + np.exp(-input))
    elif self.activation_function == "tanh":
        return np.tanh(input)
    elif self.activation_function == "relu":
        return np.maximum(0,input)
```

Functional similarity is when two pieces of code are similar in functionality without being written in a textually identical/similar way. This kind of similarity is called semantic clones and referred to as ***Type 4***. Table 2.4 shows an example of Type 4 clone. The two functions differ syntactically, but they achieve the same result, which is to compute the post-activation of X with respect to the activation function.

In our study, we are interested in detecting both exact (Type 1) and near-miss clones (Type 2 and Type 3). Thus, we use the most recent version of Nicad (NiCad-5.2), at the date of launching clone detection on our subject systems. We use NiCad because it was found to achieve higher precision and recall in near-miss clone detection [36].

2.3.2.2 Location-based Clone Taxonomy

Clone taxonomies are categorized based on three attributes : similarities, location, and refactoring opportunities as introduced in the survey by Roy and Cordy [37]. In this thesis, we follow the location-based taxonomy proposed by Kapser and Godfrey [38]. Kapser and Godfrey [38] introduced a categorization scheme for code clones, and applied their taxonomy in a case study performed on the file system of the Linux operating system. They provide a hierarchical classification of clones using attributes such as locations and functionality. Their taxonomy mainly consists of three partitions of the physical locations of clones in the source code as follows :

— **Same file** when clones reside in different locations of the same file.

— **Same directory** when clones belong to different files but within the same directory.

Table 2.3 Type 3 Clones

```python
def forward_activation_fct(self, X):
    if self.activation_function == "sigmoid":
        return 1.0/(1.0 + np.exp(-X))
    elif self.activation_function == "tanh":
        return np.tanh(X)
    elif self.activation_function == "relu":
        return np.maximum(0,X)
```

```python
def forward_activation(self, x):
    if self.activation_function == "sigmoid":
        return 1.0/(1.0 + np.exp(-x))
    elif self.activation_function == "tanh":
        return np.tanh(x)
    elif self.activation_function == "relu":
        return np.maximum(0,x)
    elif self.activation_function == "leaky_relu":
        return np.maximum(self.leaky_slope*X,X)
```

Table 2.4 Type 4 Clones

```python
def forward_activation(self, X):
    if self.activation_fct == "sigmoid":
        return 1.0/(1.0 + np.exp(-X))
    elif self.activation_fct == "tanh":
        return np.tanh(X)
```

```python
def forward_activation(self, x):
    vals = { "sigmoid" : 1.0/(1.0+np.exp(-x)),
             "tanh" : np.tanh(x) }
    return vals[self.activation_fct]
```

— **Different directories** when clones are detected in different files and different directories.

They further sub-classify the clones by the type of the region in which they are located (i.e., function, loop, function ending, etc). In our study, we apply the same location-based classification and propose a sub-classification of clones based on the functionalities related to deep learning. We perform this classification by manual analysis of the clones and the resulted location based classification. We use the different steps of the deep learning workflow presented above (Figure 2.1) to label the detected clones.

2.3.3 Bug-proneness of Code Clones

Code cloning facilitates code reuse and thus intuitively increases productivity. However, this productivity gain may be outweighed by the negative impacts of clones on software maintenance as suggested by empirical evidences from different studies [**?** **?** **?**]. For example, code-duplication increases both size (code bloating) and complexity of the software system. Because of these confounding factors, software maintainability may become increasingly complicated. One of the key challenges posed by code clones is ensuring the consistent evolution of clones during software maintenance ; meaning that all cloned copies should be updated with necessary changes. This is because inconsistent changes to clones or missing change propagation are likely to introduce bugs [69, 88**?** **?**]. As consistent changes to clones is important, missing changes, once identified, should also be propagated (late propagation) accordingly. However, late propagation of changes to clones have also been found to be prone to bugs introduction [14, 39**?**]. Again, as the cloned copies of code fragments are expected to evolve consistently, cloned code are likely to experience frequent changes, and thus negatively affect the stability of the software systems [69, 72? ? ?]. The instability of clones, in turn, has also been found to be related to bugs [?]. The bug-proneness of clones may also vary based on the types of clones [40]. Several other prior research works have also investigated the bug-proneness of code clones [17, 41?]. These multiple studies on the impacts of clones, from different perspectives, show that the bug-proneness of clones is an important concern. Given the complexity and lack of explainability or the 'black-box' nature of the deep learning models, testing deep learning based systems and thus fixing bugs are quite challenging [42, 43]. Thus, it will be of interest to study the relationship between software bug-proneness and code clones in the deep learning applications context. Since, duplicating code is a common practice in the deep learning development process, as reported by deep learning practitioners in a previous survey [5]. In this thesis, we aim to empirically analyze deep learning systems to understand the extent and impacts of clones. We aim to raise the awareness of deep learning developers on the impact of code cloning, since it is likely to add more complexity and cost to the development and maintenance of deep learning systems.

2.4 Chapter Summary

In this chapter, we provided an overview of the key concepts and methodologies that are related to deep learning software systems development. We also introduced the concepts of code smells and code clones. This background information is offered to provide a clear understanding of the different techniques used in this thesis.

The following chapter provides a review of the literature on the software quality assurance of deep learning software systems.

CHAPTER 3 A COMPREHENSIVE REVIEW OF SOFTWARE ENGINEERING STUDIES ON DEEP LEARNING SYSTEMS

Deep learning is one of the most significant areas of contemporary AI research. It has been shown to be very beneficial in finding complex structures in high-dimensional data and it, therefore, impacts a wide, diverse variety of application areas [44]. Work done by deep learning in the recent years has also shown promising results within natural language understanding and related fields, notably in the areas of sentiment analysis and question answering, while demonstrating excellent record in speech and image recognition. Deep Learning is currently playing a vital role in virtually all kinds of information processing systems. Therefore, the quality assurance of deep learning software systems is becoming a compelling issue. Several studies have found that it is relatively fast and cheap to develop and deploy deep learning systems. It therefore often happens that the quality of their software is affected. This introduces risks for code smells. The prevalence of frameworks and standardized approaches for building deep learning models makes it likely to see developers opting for copy-paste when writing some redundant parts of deep learning code, which is likely to lead to code clones, which is a type of code smell. Since code smells have been shown to negatively impact the quality of traditional systems, and since deep learning systems are especially complex, it is crucial that a serious study be executed for deep learning code smells. While there exist several studies around solving software engineering problems for traditional systems, there are fewer studies that tackle software engineering challenges in AI/ML/DL-based applications.

The purpose of this chapter is to review existing software engineering techniques that can be used to build better deep learning systems. First, we identify and explain the challenges that should be addressed when developing DL programs, as well as the DL software quality assurance techniques found in the literature. Next, we report about the impact of code smells and code clones on the software quality of traditional systems. At the end, we discuss the gaps in the literature regarding the quality assurance of deep learning systems, which we aim to address in this thesis.

Chapter Overview : Section 3.1 reviews existing studies that tackle the challenges faced by developers when building AI/ML/DL systems. We also review previous studies that examine the quality assurance of deep learning systems. Section 3.2 examines the impact of code smells and code clones on traditional systems quality. Section 3.3 provides an overview of the gaps in the literature that our research work aims to fill. The conclusion of this chapter can be found in Section 3.4.

3.1 Software Engineering for AI/ML/DL systems

Thanks to the democratization of powerful open-source AI/ML/DL libraries/frameworks, complex prediction systems are built quickly. Due to this rapid release of this type of system, the software quality is often sacrificed. Thus, it becomes challenging and expensive to maintain them over time because of technical debt. Sculley et al. [45] discuss the challenges in designing ML systems and explain how poor engineering choices can be very expensive. The challenges discussed include : hidden feedback loops, data dependencies, configuration debt, common ML code smells, etc. Amershi et al [28] reported the best practices used by Microsoft software engineers while developing projects that are related to Artificial Intelligence and Machine learning. They mainly focused on the differences between ML-based software projects and traditional projects, and the challenges of adapting Agile principles to ML-based systems. Their study was conducted via interviews with selected Microsoft developers and a large-scale survey within the company. They report that maintaining and versioning data is a crucial task for ML-based systems. They also remark that data is harder to version than code. And that in addition to being a software engineer, ML skills are needed to build ML-based systems. Furthermore, it is more challenging to handle distant modules in ML-based systems.

Testing and Monitoring : One of the important strategies to reduce technical debt and lower long-term maintenance costs is testing and monitoring. ML-based systems are more difficult to test than traditional software systems [43]. This is a consequence of the heavy dependence of ML on data and models. Breck et al. [46] have outlined specific testing and monitoring needs based on practical experience at Google. They provide 28 actionable tests that can be used to measure the production readiness of a ML-based system and reduce technical debt. Breck et al.'s study, as most existing studies from the literature, focuses more on model quality rather than the infrastructure quality of machine learning systems. Zhang et al. [47] provide a comprehensive survey of ML testing covering 138 papers. The study of Zhang et al. [47] presents definitions and research status of many testing properties such as correctness, robustness, and fairness. In addition, they discuss the need to test the different components involved in the ML model building (data, learning program, and framework). Since ML testing remains at an early stage in its development, they present many challenges. Among these challenges, they found challenges in test input generation, challenges on test assessment criteria, challenges relating to the oracle problem, and challenges in testing cost reduction. Furthermore, Zhang et al. [47] analyze some research directions to benefit ML developers and the research community. They suggest testing more application scenarios since most of previous studies focus on image classification. It will be worth investigating testing many other areas such as speech recognition or natural language processing. They also mentioned uncovered testing opportunities like testing unsupervised and reinforcement learning systems.

Software Engineering Practices and Challenges : Amershi et al. [28] performed a survey of Software Engineering practices for ML-based systems at Microsoft. They interviewed Microsoft developers to understand their development practices and the benefits of these practices. Another study related to software engineering practices for DL applications was conduced by Zhang et al. [48]. Zhang et al. also surveyed DL practitioners about their software engineering practices. They formulate recommendations to improve the development process of DL applications. Wan et al.[49] studied the features and impacts of machine learning on software development. They compare various aspects of software engineering and work characteristics in both the machine learning systems and non-machine learning software systems. A recent study by Chen et al. [50] examined challenges in deploying DL software by analyzing Stack Overflow posts and posts from other popular Q&A website for developers. They proposed a taxonomy of the challenges faced by developers when deploying DL software.

Bugs in Deep Learning Code : Islam et al. [51] analyzed stack overflow posts, as well as, bug-fix commits from popular deep learning frameworks to understand the characteristics of DL systems' bugs (their types, root causes, and effects). Zhang et al. [52] studied deep learning applications built on top of TensorFlow [53] by collecting their program bugs from GitHub and Stack Overflow. They identified the root causes and symptoms of the collected bugs. They also studied the detection and localization challenges of these bugs.

Code Smells in Deep Learning Applications : Jiakun Liu et al [5] investigated technical debt in deep learning frameworks by mining the self-admitted technical debt comments provided by developers. Among the types of design debt, Jiakun Liu et al [5] report that DL developers consider code duplication to be a contributing factor to technical debt and increased maintenance costs.

Computational Notebooks : Nowadays, we are witnessing a proliferation of computational notebooks in data science studies, thanks to their strengths in presenting data stories and their flexibility. However, using these notebooks in a real project may induce technical debt, because of their lack of abstraction, modularisation, and automated tests. Recently, a fair amount of research works has been conducted on computational notebooks, mostly focusing on the challenges that they pose to data scientists and the poor software engineering practices observed in these notebooks.

One common bad practice that is frequently observed in computational notebooks is the copying and pasting of code, by data scientists in order to save time and effort. Kery et al.

[54] conducted two case studies where they interviewed 21 data scientists and surveyed 45 data scientists to understand the use of literate programming tools. They studied Jupyter Notebook as it is the most popular literate tool [55]. They [54] identified the good and bad practices employed by data scientists. One practical limit of Jupyter Notebook is its size and performance. The limited size often leads data scientists to copy-paste code into a new Notebook when the maximum size is reached. In addition, they copy-paste code to ensure that the code dependencies are properly located next to the new code, instead of extracting the code to a function. Pimentel et al. [56] conducted an empirical study of 1.4 million notebooks from GitHub ; examining reproducibility issues, and challenges related to the implementation of projects within Jupyter notebook.

They also provide a set of best practices to improve reproducibility. Additionally, they identified and reported good and bad practices followed by developers of computational notebooks. One best practice that is reported is the use of markdown and visualization, which are two key features of literate notebooks. The use of a convenient and comprehensive filename is also reported to be a frequent good practice in computational notebooks. The bad practices identified include the lack of testing code, as well as poor programming practices that make reasoning and reproducing results more difficult, such as non-executed code cells and hidden states. Psallidas et al. [57] also examined the quality of notebooks (through an analysis of 6 million python notebooks from GitHub and 2 million enterprise DS pipelines developed within COMPANYX). They also performed an analysis of 12 popular deep learning libraries over 900 releases. They report that the majority of notebooks use only a few libraries and that commonly used tools are mature and popular.

Koenzen et al. [29] examined how code is cloned in Jupyter notebooks and found that 7.6% of code clones are self-duplication. They also performed an observational lab study and found that frequently reuse code is copied from web tutorial sites, API documentation, and Stack Overflow.

3.2 Impact of Code Smells on Software Quality

Since we investigate code smells in the deep learning code, we are interested in reviewing the existing literature on the impact of code smells in traditional software systems. In the following subsection, we discuss the impact of code smells and code clones (which is a type of code smell) separately since the literature has mostly treated them separately.

3.2.1 Impact of Code Smells

Code smells frequently arise from several factors. The major causes are quick bug fixes, inexperienced developers, non-awareness of the best practices and of the peril of code smells on code quality and the evolutivity of software. Given the rapid pace of development and release of deep learning applications, we expect code smells to be prevalent in these systems. Code smells are violation of best practices that make software hard to understand and maintain. Numerous studies have provided evidences of the negative impact of code smells on software quality.

Yamashita [58] examined the capability of twelve code smells to explain maintenance problems through qualitative and quantitative analyses. The results of their work provide empirical evidences of the impact of design aspects on maintenance problems. Soh et al. [59] investigate the effects of code smells on development and maintenance activities. They found that code smells impact code editing and navigating activities and that file sizes impact the reading and searching efforts. Hence, code smells and file sizes hinder maintenance activities. Perpletchikov and Ryan [60] investigate the impact of design level coupling on software maintainability. Their findings provide empirical evidences of the negative impact of highly-coupled elements on the analyzability and changeability of software.

MacCormack et al. [61] evaluate the impact of architectural design choices on maintenance costs and refactoring efforts. They examine the nature of dependencies of each component in a system, i.e, direct, and indirect, and their effects on maintenance costs. Their results suggest that architectural debt can be assessed by understanding patterns of coupling among components in a system.

Khomh et al. [62] investigate the effect of antipatterns on classes in object-oriented systems with respect to change and fault-proneness. They show that classes with the presence of antipatterns are more prone to change and faults in almost all releases. In addition, structural changes affect more classes with antipatterns than others. They also highlight the importance of detecting antipatterns to improve quality and testing activities.

3.2.2 Impact of Code Clones

Roy and Cordy [10] report that code clones represents between 7%-23% of the code of traditional software systems. Multiple studies from the literature have examined the impacts of clones on traditional software systems from different software quality perspectives, e.g., change-proneness, bug-proneness, challenges in consistent update, and overall maintenance efforts and costs. Sajnani et al. [63] showed that, contrary to intuition, the cloned code contains less problematic patterns than non-cloned code. Along the same line, Rahman et al.

[64] reports no correlation between bug-proneness and code clones. However, these conclusions about the lack of harmfulness of clones are contradicted by Islam et al. [65] who found that code cloning activities contribute to replicating bugs. Islam et al. suggest to prioritize refactoring and tracking for clone fragments containing method calls and/or if-conditions, to prevent bugs being replicated. Another study by Islam et al. [66] examined bugs that were reported during the evolution of a software system for two different programming languages (Java and C) and found that clone code tends to be more bug-prone than non-clone code.

Aversano et al. [39] investigated how clones are maintained considering the inconsistency that may induce code clones when fixing a bug in just one fragment or when evolving code fragments. They found that the majority of clone classes are always maintained consistently. Similar work was also performed by Göde et al. [67], confirming the findings of Aversano et al. Göde et al. also found cloned code to be even more stable than non cloned code. They also report that near-miss clones (Type 2 and Type 3) are more stable than exact clones (Type 1). Krinke [13] conducted a comparative study (in terms of the average age) between cloned code and non-cloned code. They observed that cloned code is usually older than non-cloned code and that cloned code in a file is usually older than the non-cloned code in the same file. This confirms previous observations that code clones are more stable than non-cloned code. Therefore, maintaining code clones is not necessarily more expensive than maintaining a non-cloned code.

Jiang et al. [68] examined the bug-proneness of cloned code and confirmed that code cloning can be error-prone and directly related to inconsistencies in the code. They proposed an algorithm able to locate clone related bugs by detecting such inconsistencies. When a code fragment contains bugs and is reused by duplicating it with some adjustments w.r.t to the need, it may increase the spread of bugs in the system. Several other previous studies from the literature [14, 15, 16, 17, 18, 19, 20] report a similar conclusion about code clones, i.e., that they make the code bug-prone and increase maintenance costs. Juergens et al. [16] examined the root cause of faults in cloned code and report that one of the major sources of faults is inconsistent code clones. They provided an open-source algorithm for the detection of inconsistent clones. Göde and Koschke. [69] provide empirical evidences showing that unintentional inconsistencies of code clones leads to faults. Barbour et al. [14] examined late propagation evolutionary patterns of clones and identified 8 types of late propagation. They further examined the risk of faults in these evolutionary patterns and found that late propagation in which a clone is modified and then re-synchronized without any modification to the other clone in the clone pair is the most risky pattern.

Researchers have also examined the maintenance efforts that result from duplicating code.

Hotta et al. [12] conducted an empirical study on 15 open-source systems, comparing the modification frequency of code clones and non-clones code. They concluded that the existence of clones does not impact software evolution negatively. Kapser and Godfrey [11] performed an empirical study of code cloning patterns, reporting the reasons behind the different patterns. They also report that the majority of clones have a positive impact on software maintainability. However, their claim is contradicted by Kim et al. [70] who suggest that refactoring techniques cannot tackle consistently changing code clones. Li et al [71] advise that maintaining duplicate code would be very beneficial for developers, as this would avoid introducing hard to detect bugs. Lozano and Wermelinger [72] have also shown the negative impacts of code clones in terms of maintenance cost and system stability. They found that code clones have a higher density of changes than non-cloned code. Lozano and Wermelinger [**?**] also show that the existence of code clones may increase the change effort. A recent study by Mondal et al. [73] shows that cloned code are more unstable than non-cloned code in general. However according to Selim et al. [74], the bug-proneness of code clones might be system dependent. Mondal et al. empirical study [75] shows that cloned code tends to require more effort in maintenance than non-cloned code and that Type 2 and Type 3 clones often need a special attention when making management decisions since they require more effort.

While previous works examined the prevalence and impacts of code clones in traditional software systems, we investigate the distribution and impacts (bug-proneness) of code clones in the deep learning code. We manually investigate clones in deep learning systems with the aim to derive insights on 'what' functions deep learning practitioners clone and 'why' they clone.

3.3 Discussion

Here we detail the identified gaps in the literature as our motivation for conducting this analysis of deep learning systems. In addition, we discuss why we focus on DL systems instead of broader AI/ML systems.

3.3.1 Code Smells in Deep Learning Code

The majority of work on deep learning systems has the main objective to provide a model with better performance. While there exist several works tackling the problem of software engineering practices and challenges for AI-based systems, the research on code inspection in this area remains limited. To fill this literature gap, this thesis studies the distribution of code smells in deep learning systems and compares them to traditional ones. We also discuss these distributions from different perspectives, varying from high-level to low-level insights (e.g., the entire dataset first, later per smell, or by groups (size of projects), etc.). Furthermore, we study the trend of the evolution of code smells over time and the relation between smelly files and bug-fixes.

3.3.2 Code Clones in Deep Learning Code

Previous studies on duplicated codes in data scientists' projects have almost exclusively focused on analyzing computational notebooks. A number of studies' results have shown that copying and pasting of cells within the same notebook is a widespread practice. There are some works discussing the common code duplication practices of deep learning practitioners. However, these works were limited to interviews with practitioners and focuses only on computational notebooks. In this thesis, we examine the distribution of code clones deep learning systems, in terms of occurrences and location, and propose a taxonomy of code clones in deep learning systems. We also study the relationship between code clones and bug fixes, and examine the model construction phases in which cloning is the most risky.

3.3.3 Why Deep Learning Systems ?

The choice of Deep learning was based on the two following facts : First, the popularity of DL technology as a high capacity model that enables the creation of software 2.0, and that learns from the data to perform a human task. However, machine learning was widely applied to build explanatory and statistical models with lower capacity but higher explainability to understand relationships between variables. Second, the DL is the same algorithm (backpropagation) and different architectures using common components. Thus, we have open-source frameworks (Tensorflow, PyTorch, Mxnet) that give similar features and there is now a relatively common way to construct the DL code. This motivates us, as DL users too, to detect the python code smells and code clones in this emergent type of projects and communicate the reasons behind, which allows DL practitioners to avoid them in the future. However, the ML solutions have different algorithms and multiple possible implementations, which makes it difficult to infer conclusions about coding practices in these projects that are different from any python code project.

3.4 Chapter Summary

Recently, researchers have started to investigate the code quality of deep learning systems to understand the main challenges of deep learning systems development. In this chapter, we explained the need for code quality assurance in deep learning systems. Next, we reviewed previous studies on the quality assurance of AI/ML/DL systems. We also reviewed existing studies on traditional software with respect to the impact of code smells and clones. We finally highlighted some gaps in the literature related to the code smell detection in DL programs. In the following chapters, we attempt to fill the identified gaps in the literature by studying code smells and code clones in deep learning applications, to help DL practitioners write code with good quality and increase their awareness about the peril of poor coding practices.

CHAPTER 4 THE SCENT OF DEEP LEARNING CODE : AN EMPIRICAL STUDY

4.1 Introduction

The rise of open source DL frameworks has contributed to the democratization of deep learning technology. Thanks to scripting-based libraries such as Tensorflow and Pytorch, DL practitioners are now capable of developing functional prototypes quickly and experimenting with them. However, such prototypes mostly consist of glue code that patches together the program identifiers, external libraries, data processing functions and training algorithms [4]. That is, due to heterogeneous components and dependencies, the correctness of deep learning code might often be traded with its code quality. Such a choice might turn the deep learning code into complex software applications that are hard to comprehend, debug or even to enhance in the long run.

ML-based systems encounter all the maintenance issues of traditional software systems. However, they suffer from an additional set of issues that arise from their statistical and data-driven nature [4]. There have been a few earlier works that examined software bugs in deep learning frameworks [51] and analysed the software engineering practices followed by DL practitioners [28]. They suggest that poor coding practices and quick solutions often result in low-quality code containing various code smells. The presence of code smells within the software systems might incidentally degrade their quality and performance, and thus hinder their maintenance and evolution. While there have been a number of studies on the code quality of traditional software systems and a few on ML-based systems, to date, no investigation has been performed on the code quality of DL-based software systems.

In this chapter, we introduce our empirical study on the quality of deep learning code using 118 open-source software systems. We first determine the prevalence and trends of code smells in the DL code, and then contrast them with the code smells from traditional software code. We also show that the amount of code smells increases in the DL applications across releases, and that code smells and software bugs often co-exist within the deep learning code. To the best of our knowledge, this is the first study that investigates the code quality of DL-based software systems. Our study answers three research questions as follows.

RQ$_1$: Does deep learning code smell like the traditional software code ?

We collect a total of 59 deep learning systems and 59 traditional software systems from GitHub. We determine the distribution of 10 code smells in the deep learning systems, and then compare them with the smells from the traditional systems. We found no statistically significant difference between the code smell occurrences in deep learning projects and that in the traditional ones. When types of smells were considered, we found that *Long Lambda Function*, *Long Ternary Conditional Expression* and *Complex Container Comprehension* are more frequent within the deep learning code than within the traditional source code.

RQ$_2$: What is the global trend of code smells in deep learning projects over multiple releases ?

We investigate the trend of code smells found in the deep learning projects over time. We detect the code smells from each of their releases and then analyse the changes in smell instances across the releases. Our analysis suggests that the number of code smells increases across the releases in the deep learning applications.

RQ$_3$: Is there a co-existence between code smells and software bugs in deep learning applications ? We analyse 37,951 commits from 59 deep learning projects to determine the co-existence between code smells and software bugs within DL code. We first separate the bug-fixing commits using their labels and extract the files that were changed to fix the bugs. Then we determine whether these changed files contain code smells in our deep learning applications. We found that 62.84% of the changed files overlap with the smelly code. Furthermore, frequent smells (e.g., Long Ternary Conditional Expression, Long Parameter List) co-exist with the software bugs more often than the others. We also analysed the time spent for fixing the buggy code. We found that the buggy code from the DL applications needs more time to fix when it is smelly than when it is odorless.

Chapter Overview Section 4.2 presents an overview of the design of our empirical study. Section 4.3 reports our findings and discussions with respect to the three research questions. Section 4.4 discusses research implications. Section 4.5 addresses the threats to validity of our empirical study on code smells in DL code. Finally, we conclude the chapter in Section 4.6.

4.2 Study Design

Fig. 4.1 shows the schematic diagram of our empirical study. It has three major steps. First, deep learning-based and traditional software systems are carefully selected from GitHub for the study (Fig. 4.1-(a)). Each of the software systems (a.k.a., repositories) is pre-processed and prepared for code smell detection. Second, we detect code smells using PySmell tool from each of the releases of DL-based and traditional systems (Fig. 4.1-(b)). Third, we collect bug-fixing commits and their changed source files to determine the co-existence between code smells and software bugs (Fig. 4.1-(c)). The following subsections discuss these steps in details.

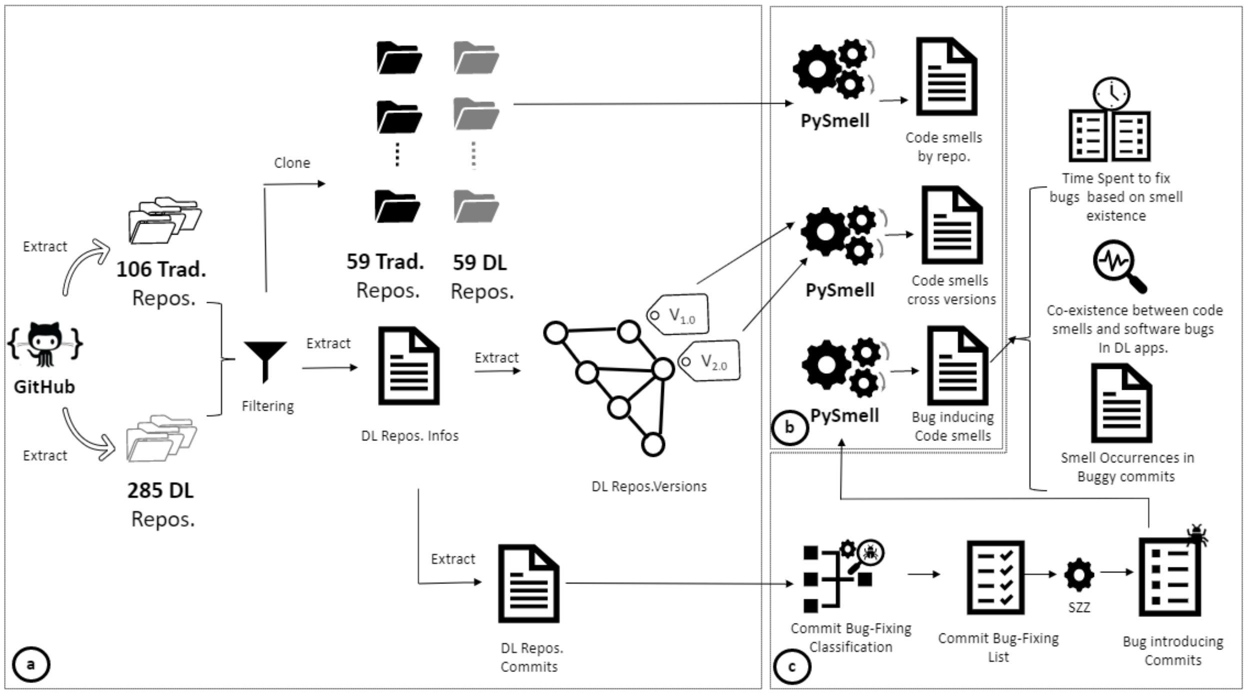

Figure 4.1 Schematic diagram of the empirical study -(a) Subject system collection and filtration, (b) Code smell detection, and (c) Code smell-bug co-existence analysis

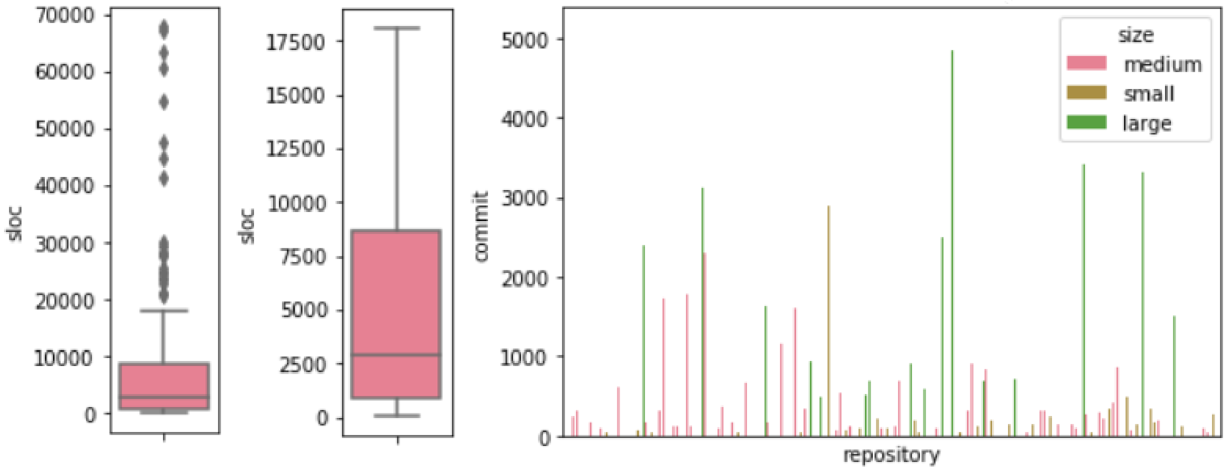

Figure 4.2 (a) Distribution of SLOC, and (b) Commits in the selected DL-based subject systems

4.2.1 Subject System Collection & Filtration

System Collection : We attempt to contrast between DL-based and traditional software systems in terms of their code quality (e.g., presence of code smells). Thus, we need to collect both types of systems for our study. In order to collect deep learning systems, we perform keyword search with GitHub Search API [76]. In particular, we choose a set of popular keywords related to various deep learning technology and frameworks as follows. **{***Deep-learning, deep-neural-network, neural-network, CNN, RNN, convolutional neural network, recurrent neural network, Caffe, Keras, Tensorflow, Theano, tflearn, Paddle incubator-mxnet and Torch***}**

We also limit our search to Python-based systems since Python is the most widely used programming language in the DL-based applications to date [77]. The search retrieves a total of 285 DL-based repositories with at least 57 commits each. In order to select the traditional software systems, we reuse the benchmark of Chen et al. [1]. The benchmark provides a total of 106 popular repositories with at least 1000 stars each.

System Filtration : The previous step provides a total of 391 (285 DL-based + 106 traditional) software systems. However, all of them might not be appropriate for our study. We thus manually check each of these systems and discard the inappropriate ones such as tutorials and non-popular projects. Out of 285 repositories, 139 repositories were tutorials, which left us with 146 real deep learning repositories. We also carefully select popular and mature DL projects from them by employing maturity and popularity metrics (e.g., issue count, commit count, contributor count, fork count, stars). We retain only such repositories that have at least four releases each, and discard the rest. Thus, we ended up with a total of 59 DL-based software systems. Out of 106 traditional software systems, we found that 25 systems do not exist any more, which leaves us with 81 traditional systems. However, we randomly choose 59 of them for the study, which ensures a parity in size with our DL-based systems. Thus, we ended up with a total of 118 DL-based and traditional software systems for our empirical study.

System Clustering : While 118 subject systems were selected above, we attempt to better understand them by analysing their SLOC (Source Line of Code) and number of commits. We first calculate the SLOC of each system using Radon [78], a Python-based tool, where only Python files are considered. Then we categorize the subject systems into three clusters – *small, medium* and *large* – using KBinsDiscretizer [79] from Scikit-Learn library [80]. KBinsDiscretizer accepts number of clusters as a parameter and provides balanced clusters using quantile strategy. According to Brown [81], a project is considered *small* if it has SLOC$<= 10K$, *medium* if it has SLOC$<= 100K$ and *large* if it has SLOC$> 10M$. However, since our DL-based systems were not big enough, we adapt these thresholds for our study. In particular, we consider a system small when SLOC$<= 4K$, medium when SLOC$<= 15K$ and large when SLOC$> 15K$. Fig. 4.2 shows the (a) distribution of SLOC and (b) number of commits/system in 59 DL-based subject systems. We see that DL-based systems are mostly small or medium. The largest DL-based system has a total SLOC of ≈90K. These clusters are later used for answering RQ_1 (Section 4.3.1).

We also analyze the releases of our deep learning subject systems. We found that the median number of release is 10, i.e., 50% of the DL-based systems have more than 10 releases. For the sake of our analysis, we thus divide the release history of each project into 10 major releases, which involves the merging of actual releases. The systems having less than 10 releases are kept as is. Table 4.2 shows the number of deep learning systems from three different clusters across 10 releases.

4.2.2 Code Smell Detection

Our smell detection strategy is based on PySmell [33] as discussed in section 2. To perform our study, we adapted the PySmell code available on GitHub to meet the needs of our project. The code takes as input a folder that contains all the projects under study, and then detects different types of smells from each of the source files. PySmell is a metric-based tool that detects code smells using rules and thresholds. It marks an entity (e.g., class, function) smelly whenever the entity activates any of the predefined rules designed for the smell types. Finally, we capture the code smell occurrences for each smell type against different granularity of program entities – class, function, line. We employ several code level metrics [1] for smell detection as follows :

— **PAR** : number of parameters ;
— **MLOC** : method/function lines of code ;
— **DOC** : depth of closure ;
— **CLOC** : class lines of code ;
— **LMC** : length of message chain ;
— **NBC** : number of base classes ;
— **NOO** : number of operators and operands ;
— **NOC** : number of characters ;
— **NOL** : number of lines ;
— **NOFF** : number of for clauses and filter expressions ;
— **LEC** : length of element chain ;
— **DNC** : depth of nested container ;
— **NCT** : number of container types.

Since PySmell's rules are configurable, we use appropriate thresholds to configure these rules that are derived from the work experience of expert Python developers [1]. Such a strategy has been adopted by earlier studies for smell detection [82].

Table 4.1 presents the strategies and the metrics' thresholds that were used to detect the code smells. All metrics and thresholds used in our empirical study were taken from an earlier work [1].

Table 4.1 Experience-based thresholds and strategies used by smell type via Chen et al. [1] study

Code Smell	Metric	Thresh.	Strategy
LPL	PAR	5	(PAR, HigherThan)
LM	MLOC	38	(MLOC, HigherThan)
LSC	DOC	3	(DOC, HigherThan)
LC	CLOC	29	(CLOC, HigherThan)
LMC	LMC	5	(LMC, HigherThan)
LBCL	NBC	3	(NBC, HigherThan)
LLF	NOC PAR NOO	48 3 7	(NOC, HigherThan) and ((PAR, HigherThan) or (NOO, HigherThan))
LTCE	NOC NOL	54 3	(NOC, HigherThan) or (NOL, HigherThan)
CCC	NOC NOFF NOO	62 3 8	((NOC, HigherThan) and (NOO, HigherThan)) or (NOFF, HigherThan)
MNC	LEC DNC NCT	3 3 2	(LEC, HigherThan) or ((DNC, HigherThan) and (NCT, HigherThan))

As shown in Fig. 4.1-(b), there are three steps in our smell detection process. First, we clone the repositories of both traditional and DL projects' GitHub repositories. Then, we run the Pysmell tool on them to detect the code smells that occurred in the files of the cloned software projects. This allows us to compare the distribution of Python code smells between traditional and DL projects. In a second step, we restore all the versions of each of the projects collected in the previous step (4.1-(b)). Then, we run the Pysmell tool on each release version of each project to analyse the trend of code smells over releases. In the third step, we restore the code to its status before applying a bug fixing commit. Then, we execute the Pysmell tool on each file that has been changed by an identified bug fixing commit in order to study the relationship between bugs and code smells in DL-based software projects.

4.2.3 Experimental Data Analysis

In this section, we describe how to analyze the detected code smells with respect to bugs' existence in the system and the time needed to fix them.

Detecting Bug-Fixing and Bug-Inducing Commits : Since we attempt to determine the potential correlation between code smells and software bugs, we need to detect the bug-fixing and bug-inducing commits from version control history. In order to detect the bug-fixing commits, we employ a keyword search-based approach. In particular, we use a list of keywords for the search – {bug, fix, wrong, error, fail, problem, patch}, as were used by Rosen et al. [83]. If a commit log contains one of these keywords, we consider it as a bug-fixing commit. Once a bug-fixing commit is detected, we use the SZZ algorithm [84] to identify the bug-inducing commits from the version history that introduced the bug.

Co-existence between Code Smells and Bugs : We determine the co-existence between code smells and software bugs by investigating how the bug-inducing code overlaps with the smelly code. First, we identify the files that were changed later to fix the bugs (i.e., bug-inducing files) and that also contain one or more code smells. Second, we calculate the percentage of occurrences for each smell type in these smelly, bug-inducing files. Our goal was to identify the most frequent code smells that co-exist with the bugs in the deep learning applications. Third, we analyze the distribution of the number of bug fixing commits with respect to smelly files and smell-free files.

Time spent to fix a bug that co-exists with code smells : To evaluate the cost of code smells in productivity, we compare the time taken to fix bugs when the files contain code smells and when they do not. We compute bug-fix time by measuring the time interval between the bug-introducing changes and their corresponding fixes [85]. We use Mann-Whitney Wilcoxon test to compare bug-fix time and examine the negative impact of the code smells on bug-fixing and developer productivity.

Table 4.2 Number of small, medium and large DL-based projects across 10 releases

	v1	**v2**	**v3**	**v4**	**v5**	**v6**	**v7**	**v8**	**v9**	**v10**
Small Projects	18	18	18	18	18	14	12	11	11	9
Medium Projects	19	19	19	19	19	17	16	15	14	14
Large Projects	22	22	22	22	22	18	16	14	12	11

4.3 Study Findings and Discussions

In this section, we present the results of our study in details, and answer three research questions as follows.

4.3.1 RQ_1 : Does Deep Learning Code smell like the Traditional Software Code ?

In this section, we determine the distribution of code smells in both deep learning and traditional software systems, and compare their distributions. We calculate the occurrences of code smells across different dimensions (e.g., project type, smell type) and then compare the DL-based systems with the traditional systems.

Smell Occurrences by Project Type : We compare DL-based systems with traditional systems in terms of their code smell density. First, we calculate the number of code smells in each project, and then divide them with their number of source code lines (a.k.a., SLOC). Fig. 4.3 shows the comparison between DL-based and traditional systems using box plots where normalized smell occurrences per source code line are considered. We perform non-parametric Mann-Whitney Wilcoxon tests, and found that there is no significant difference (i.e., p-value $0.117 > 0.05$) between DL-based code and traditional code in terms of their code smell density.

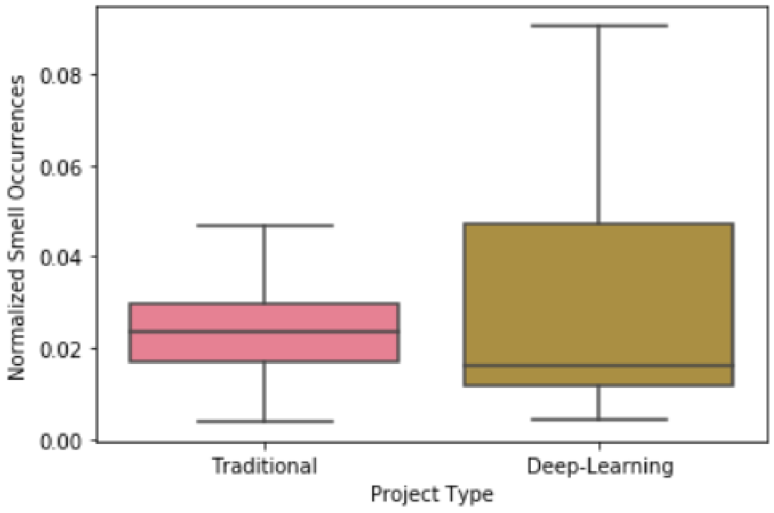

Figure 4.3 Smell occurrences in DL and traditional projects

Finding 1 : There is no statistically significant difference between deep learning code and traditional code in terms of their code smell occurrences.

Smell Occurrences by Project Size : Although the above analysis shows no difference between DL-based and traditional systems in code smell density, we further extend our comparative analysis by considering both type and size of the systems. We calculate the occurrence of code smells in each of the projects from small, medium and large clusters, and compare the smells/project between DL-based and traditional projects. Fig. 4.4 shows our comparative analysis across three sizes (or clusters). We see that the number of code smells per project increases when the size of the project increases for both DL-based and traditional projects. Such a trend is consistent with the earlier findings with traditional systems [86]. However, we found two noticeable differences in the density of code smells between deep learning and traditional software projects when we consider the size. First, the number of code smells in the *small* DL projects is significantly higher than that of *small* traditional projects. Second, the number of code smells in the *large* DL projects is significantly lower than that of the large traditional systems. We perform Mann-Whitney Wilcoxon tests in both cases, and found p-values 0.0002 and 1.19e-05 respectively that are less than 0.05 and thus indicate the statistical significance.

Finding 2 : The number of code smells per project increases with the increase in project size for both deep learning and traditional software projects. However, small DL-based projects contain more code smells than that of small traditional projects. On the contrary, large DL-projects contain comparatively less smells than that of large traditional software projects.

Complementary Manual Analysis : We manually analyse the source code of 30 deep learning projects and 30 traditional projects, and derive several meaningful insights. First, the core deep learning code that involves data pre-processing and model training tends to be smelly. Once the code is built, DL systems tend to get larger and mature by including more features for various application domains. In other words, the large DL projects tend to have a mix of core deep learning code and traditional source code. Such a mixture of heterogeneous code is often encapsulated and its functionality is exposed through a simple endpoint. Such a workaround might explain the comparatively less number of code smells in the large DL-based software systems (Fig. 4.4).

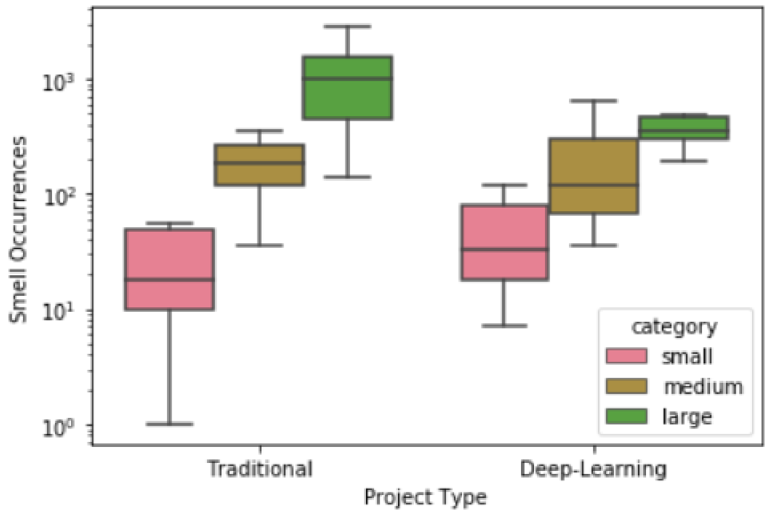

Figure 4.4 Smell occurrences by project type and project size (small, medium and large)

Smell Occurrences by Smell Type : While our above analyses show interesting findings, we further contrast the code smell distributions between DL-based and traditional subject systems in terms of code smell types. In particular, we perform statistical significance tests on these distributions across 10 different code smell types, and then collect the p-values from Mann-Whitney Wilcoxon tests. Table 4.3 shows the p-values from our tests across project size and code smell type.

Given the comparison and p-values from the Table 4.3, we also divide the code smells into three groups as follows – **Group-I :** smells that occur both in DL-based and in traditional software systems with no statistically significant difference, **Group-II :** smells that occur more in the DL-based systems than in the traditional systems, and **Group-III :** smells that occur more in the traditional systems than in the DL-based systems. Then we further investigate the prevalence and distribution of code smells across these groups.

Group-I : Large Class, Long Method and Long Scope Chaining : Based on a significance threshold of 0.05, we found three python code smells – Large Class (LC), Long Method (LM) and Long Scope Chaining (LSC) – from Table 4.3 for which DL systems and traditional systems have no statistically significant difference in their smell densities. From Fig. 4.5-(a), we see that these code smells have similar variances across both project types. These smells are often a result of poor coding practices by the developers. Thus, they might be invariant of the type of the subject systems.

Group-II : Complex Container Comprehension, Long Ternary Conditional Expression, and Long Lambda Function : From Table 4.3, we found three other code smells that occur more frequently within the DL code than in the traditional code. From Fig. 4.5-(b), we also see that their code smell distributions are significantly different in terms of median and variance. We thus manually analyse these three code smells for further insights as follows.

Complex Container Comprehension (CCC) : Container comprehension is a quick solu-

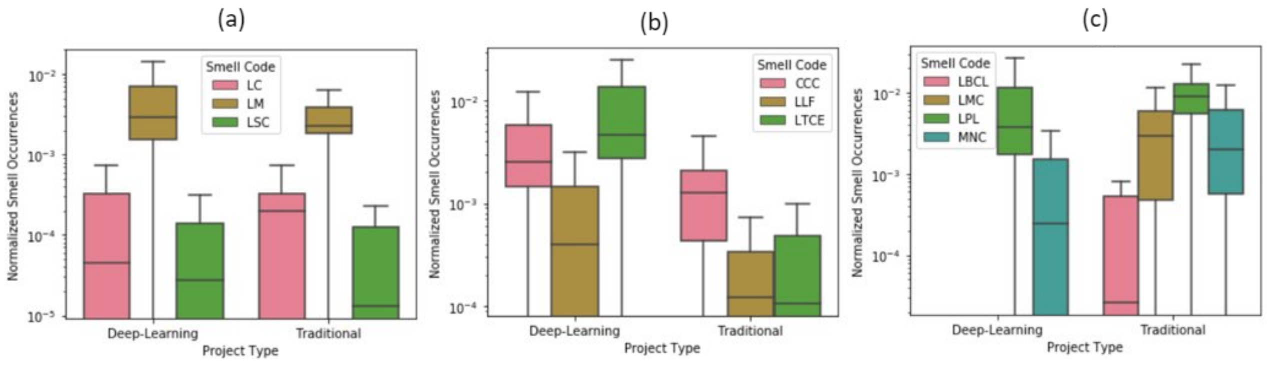

Figure 4.5 Smell Occurrences by Smell Type and by Project Type

Table 4.3 Mann-Whitney test and Cliff's Delta results between DL and Traditional Projects for each Smell Type in Total without splitting projects by size and for each size of projects.

	LC	LM	LSC	MNC	LTCE	LPL	LMC	LLF	LBCL	CCC
p-value (Total)	0.06	0.43	0.36	**5.44e-05**	**2.94e-12**	**1.22e-20**	**0.002**	**0.04**	**8.42e-08**	**0.02**
p-value (Small)	0.26	0.05	**0.02**	0.33	**1.73e-05**	0.3	**0.0005**	**0.005**	0.425	**0.035**
p-value (Medium)	0.403	0.18	0.22	**0.004**	**1.22e-06**	**0.01**	**6.4e-08**	0.37	**0.006**	**0.006**
p-value (Large)	**0.001**	0.06	0.21	**7.75e-06**	**5.40e-08**	**0.0001**	**9.29e-09**	**0.026**	**4.01e-07**	**0.02**
Cliff's Delta (Total)	0.17	0.02	0.04	0.41	0.73	0.31	0.82	0.18	0.45	0.23
Cliff's Delta (Small)	0.08	0.32	0.27	0.07	0.76	0.1	0.47	0.43	0.02	0.34
Cliff's Delta (Medium)	0.048	0.17	0.13	0.49	0.88	0.44	0.94	0.06	0.38	0.47
Cliff's Delta (Large)	0.54	0.28	0.14	0.8	0.98	0.66	0.99	0.35	0.84	0.36

tion for constructing one-line Python objects (e.g., list, set, dict). One-line statements often become complex and hard to comprehend when more and more clauses and filter expressions are added. DL developers often choose a one-line statement to build a sequence of arguments, which results in a long-expression. They use the hard-coded sequences to initialize a list of possible hyper-parameters since these values are often found in the DL white papers and books. Besides, comprehensions are constructs that allow the containers to be built from other containers. Thus, the developer often use CC to reformat the data structure of a set of values, which makes it suitable as arguments for configuring other DL routines. Hence, the longer the comprehension, the higher the risk of turning into CCC smell.

Long Ternary Conditional Expression (LTCE) : Ternary conditional expression is a conditional variable assignment in one line that makes the code compact. It allows conditional flows into the code and replaces multiple if-else blocks with a single line of code. However, excessive use of this expression could hurt the readability of code. Besides, combination of multiple terms and expressions (e.g., lambda expression) could make this conditional expression unnecessarily complex. DL developers often use ternary conditional operators either to execute a particular routine or to conditionally assign a particular value to a configuration setting. This helps them switch between DNN design and hyper-parameter tuning and determine the impact of their different choices. Our study also reports the high occurrences of LTCE in the DL projects.

Long Lambda Function (LLF) : Lambda function is a single-line function. While it is easy to define or use, the function could be hard to manage and maintain when many complex operations are involved. DL developers often choose lambda functions to carry out data processing by creating anonymous function at run-time and then by sending them to appropriate DL routines as parameters. However, the function becomes long and complex when multiple data elements are handled simultaneously by these routines.

Group-III : Long Base Class List, Long Message Chain, Long Parameter List and Multiply

Nested Container : Using a significance threshold of 0.05, we found four code smells from Table 4.3 that occur more frequently in the traditional code than in the DL code. From Fig. 4.5-(c), we see that their distribution is comparatively lower for the DL-based code. We thus analyse these four types of code smells, and attempt to explain the findings as follows. First, two smells – Long Base Class List (LBCL) and Long Message Chain (LMC) – are mostly related to object-oriented programming (OOP), which might explain their low occurrences in the Python-based deep learning code. Due to the scripting nature of Python-code, DL practitioners might not be interested to use the OOP paradigm of Python language. Second, we also observe low occurrences of Long Parameter List (LPL) and Multiply Nested Container (MNC) smells in the deep learning code. These smells arise from the complexity of the code that involves long parameter list and nested logic. Since DL practitioners implement their applications using a data-driven training process with ready-to-use routines from the libraries, the odds of long parameter list and deep nested logic occuring is lower.

4.3.2 $\mathbf{RQ_2}$: What is the global trend of code smells in deep learning projects over multiple releases ?

Although we have studied the distribution of code smells in DL applications and compared it with such distribution in the traditional applications, we also want to further analyse their prevalence over time. We thus investigate the global trend of code smells exhibited by deep learning projects over time. To perform this analysis, we compute the density of code smells per version for each of the DL projects. Then, we categorize each project into our pre-defined size-based categories (i.e., small, medium and large). Then we plot the smell occurrences over versions for each size-based category. Fig. 4.6 shows the global trends of code smell occurrence in our subject systems.

From Fig. 4.6, we see that the smell occurrences have an increasing trend. This can be explained intuitively by the increase in source code size (SLOC) and complexity over the life cycle of the DL projects. Our result is also consistent with previous findings on the traditional projects [87]. However, most of the DL projects are new projects that are yet to mature. Thus, the comparison with traditional systems in terms of evolution of code smells could be more effective and more fair when these DL projects would reach similar level of maturity. However, the increasing trend of DL code smells suggest that DL practitioners might not have paid enough attention to the quality of their code even after releasing several versions. Thus, our findings confirm the need for early refactoring of code smells by deep learning practitioners to avoid a costly maintenance in the later releases.

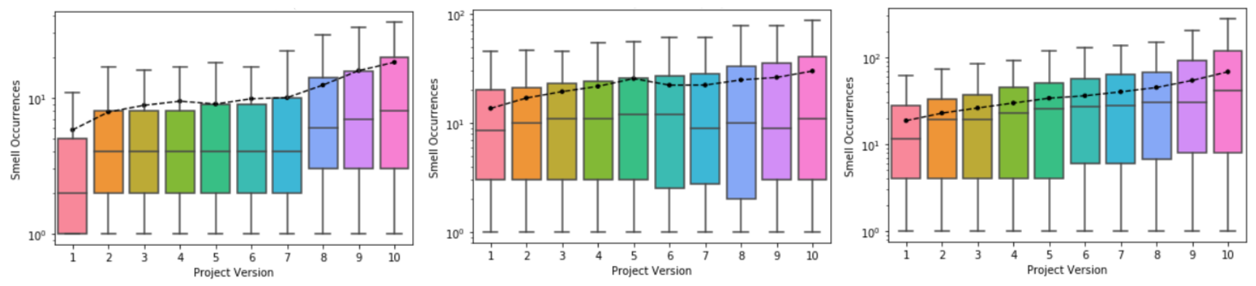

Figure 4.6 Trend of code smells in DL projects over time

Finding 4 : The amount of code smells contained in deep learning applications increases gradually over the subsequent releases of the applications.

4.3.3 RQ$_3$: Is there a co-existence between code smells and software bugs in deep learning applications ?

We consider that code smells and software bugs co-exist if the bug-fixing code overlaps with the code containing smells. In particular, we detect the files that were changed to fix the bugs and that also contain one or more code smells at the same time. We present our analysis using several dimensions (e.g., overlap ratio, smell type) as follows.

Table 4.4 shows the likelihood of co-existence with software bugs for different types of code smells. Based on our analysis, we found that, in DL-based software projects, 62.48% of the buggy files (i.e. changed by a bug-fixing commit) contain at least one occurrence of code smell.

Finding 5 : About 62.48% of the bug-fixing files overlap with the smelly code. Thus, code smells and software bugs are likely to co-exist in the deep learning applications.

We also present the overlap ratio in more granular level by computing the percentage of the smelly, buggy files per type of smell (Table 4.4). We provide the percentage of smelly, buggy files containing at least one occurrence of a particular smell. We found that at least 40% of the smelly, buggy files contain LTCE, CCC , LM, or LPL smells. On the other hand, the remaining smells (e.g., MNC, LLF, LSC and LC) could be found in less than 10% of smelly and buggy files.

Table 4.4 The Overlap Ratio Percentage between Buggy Files and Smelly Files by Code Smell Type

Code Smell	**LM**	**LTCE**	**LLF**	**LSC**	**LPL**	**CCC**	**MNC**	**LC**	**LMC**	**LBCL**
Overlap Ratio %	47.57	48.85	7.546	3.5144	47.5	43.14	6.25	7.27	0.094	0.23

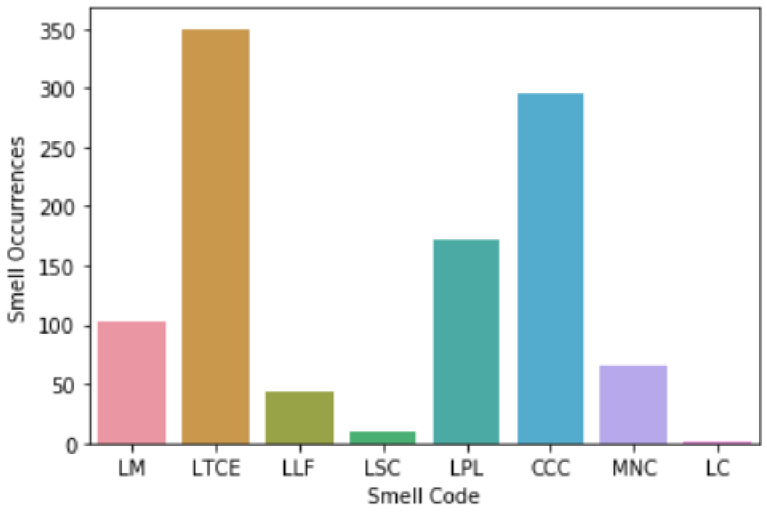

Figure 4.7 Smell Occurrences in Buggy Commits

Moreover, we present Figure 4.7 that shows the number of code smell occurrences for each of the eight smell types that partially overlap with the bug-fixing code within the deep learning systems. These substantial differences of total number of instances between smell types reinforce our previous finding and confirm that four code smells (LTCE, LM, LPL, CCC) frequently overlap with the bug-fixing code, whereas the other code smells (LLF, LSC) do not overlap much with the bug-fixing code.

Given our findings and analysis above, Long Ternary Conditional Expression (LTCE), Complex Container Comprehension (CCC), and Long Parameter List (LPL) are the types of code smells that could more likely lead to software bugs or failures. In particular, two Python-based smells –LTCE and CCC – bear more risks than other Python-related code smells given our findings in the Fig. 4.4. Deep learning developers often rely on advanced functionalities and the grammatical flexibility of Python language for rapid development. Unfortunately, such development practices turn the deep learning code into complex structures (e.g., LTCE), which ultimately can lead to software bugs and failures.

Our results heavily rely on the density distribution of various code smells (e.g., Fig. 4.4) found in our subject systems. Thus, they also reinforce the fact that the presence of code smells within the systems might increase the chance of software bugs.

Finding 6 : Long Ternary Conditional Expression, Complex Container Comprehension and Long Parameter List are more associated with the software bugs than the other smells in deep learning applications.

We also compare the number of bugs corrections that are performed to either smelly and smell-free source code files. We analyze the distribution of the number of bug fixing commits with respect to smelly files and not-smelly ones (Figure 4.8). We found that files containing at least one code smell have significantly higher number of bug corrections throughout the project than the files that do not contain any code smell. The Mann-Whitney Wilcoxon test yields a significant p-value of $3.6e - 150 < 0.05$ with a medium Cliff's Delta effect size (i.e., 0.28).

Finding 7 : Smelly files tend to have more related bug fixing commits than non-smelly files, which indicates that they might be more prone to faults. It might also be the sign that bugs occurring on smelly files require multiple fixes. In our future work, we plan to investigate this hypothesis in more details.

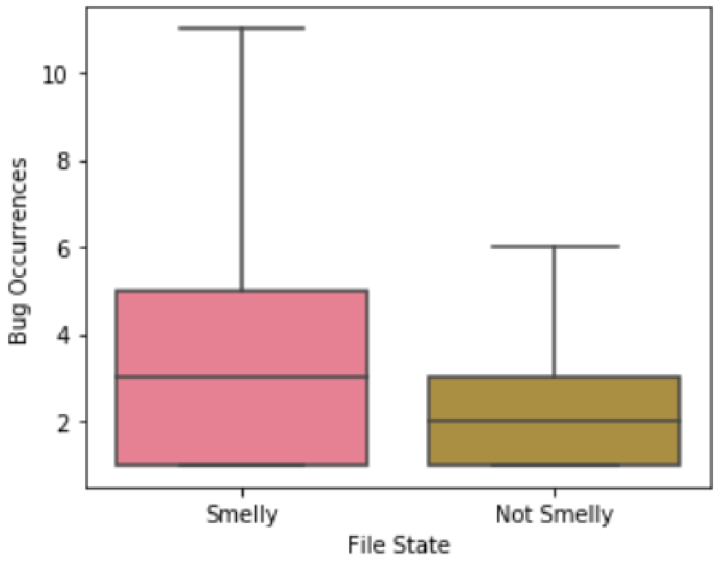

Figure 4.8 Number of Bugs Correction per Buggy File and per State of File (Smelly or not)

To gain further insights on the effort needed to fix the bugs in both smelly and smell-free code, we compare the time spent to fix buggy code containing code smells with the time spent to fix buggy code without code smells. We found that fixing the code with code smells takes significantly longer time than fixing the code without any code smells. We perform Mann-

Whitney Wilcoxon test and obtained a significant p-value of $2.02e-10 < 0.05$. Furthermore, Fig. 4.9 shows the distribution of time spent for fixing the buggy code. We see that median bug-fix time for smelly code is significantly higher than that of smell-free code. Based on the above analysis, we find significant evidence that code smells might have a significant impact on bug-fixing time in deep learning based systems.

Finding 8 : The presence of code smells has a significant impact on the bug-fixing time of deep learning applications. Time to fix the bugs within smelly code is significantly higher than the bug-fixing time of smell-free deep learning code.

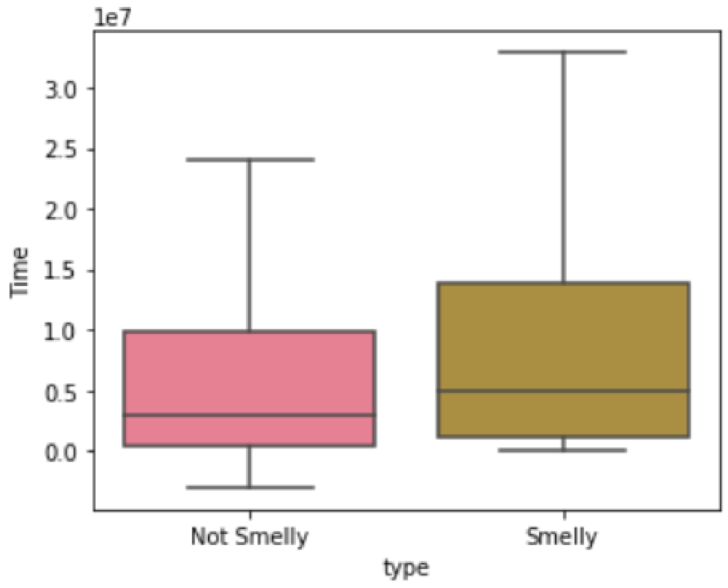

Figure 4.9 Time to Fix Bugs Distribution by Buggy Commit when it is Smelly and when Not

4.4 Research implications

In this section, we discuss several implications of our findings.

Smelly one-line long and complex statements : In the deep learning code, decision logics are learned statistically from the data rather than from complex control flows, nested loops and branches. The DL practitioners might abuse the feature of one-line statements including container comprehensions, ternary conditional operators, and lambda functions with the intention of compacting the code. Our analysis on the prevalence of code smells in the DL projects shows that these one-line statements tend to be longer and involve both complex conditions and sophisticated operations. These two code smells are the most frequent in the DL code. To improve their code quality, DL practitioners should avoid these quick solutions and refactor these one-line statements by decomposing them into manageable separate functions. Besides, the hard-coded long containers like lists or sets are also considered to be CCC code smells since they hinder the code readability and comprehension. Thus, the DL practitioners should also consider the separation of concerns in their application by cutting off the initial configuration options from the code and adopt maintainable structured files like JSON or XML that can be easily loaded as Python objects.

An increasing trend of code smells over versions : The increasing trend of code smells in DL applications call for the development of techniques and tools that can help DL practitioners refactor their code and improve the code quality.

Co-existence between software bugs and code smells To boost productivity, deep learning developers should become aware of the costs of poor coding practices. It is important to be more conscious about code smells and their peril rather than reaching an non-manageable code state, where the only solution is to write the code again from the scratch.

4.5 Threats to validity

We now describe the threats to the validity of our study.

Construct Validity threats concern the relation between theory and observation. In our study, threats to the construct validity are mainly due to measurement errors. We use the PySmell tool to detect smells in both types of project (e.g., traditional and DL software project). Relying on the outcome of PySmell tool may pose a threat to validity. To mitigate the risk, we use the experience-based strategy that relies on the thresholds pre-defined by active, experienced python developers. Thus, our thresholds are possibly more appropriate than the thresholds that are defined statistically by analysing the traditional python projects. We are aware that our results can be affected by the presence of false positives and false negatives. However, Chen et al.[1] reported a precision above 82% and a recall of 98% for the experience-based strategy, which are acceptable performance, especially for the recall.

Internal Validity threats concern our selection of projects which could have influenced the results. In our investigation, the searching requests based on GitHub topics and keywords may pose an internal threats to validity. However, we filter the projects using maturity and popularity metrics such as issues count, commits count, contributors count, forks count and stargazers count in order to eliminate all the tutorials and the shared code snippets, and keep only real engineered DL projects.

External Validity threats concern the possibility to generalize our results. In our work, we focused only on python projects, which may reduce the generalizability to all types of DL projects. However, it is important to mention that Python is the most popular and most used programming language [77] in the DL community ; so our empirical study on the code quality of DL software projects written in python can spawn useful insights on the software quality of existing DL software systems.

4.6 Chapter Summary

In this chapter, we focus on inspecting DL code by performing a comparison of smells occurrences between traditional and deep learning applications. We analyze a total of 118 repositories (59 deep learning + 59 traditional). We make the following observations :
1) No significant difference : There is no statistically significant difference in the code smell occurrences between deep learning and traditional software systems.
2) Prevalence of code smells in deep learning projects : The most frequent smell types found are *Long Ternary Conditional Expression*, *Complex Container Comprehension*, and *Long Lambda Function*.
3) Violations of the best practices : DL practitioners might not be aware of the code smells in their code, which possibly explains the increasing trend of smell occurrences across the software releases.
4) Code smells lead to bugs : Our findings confirm that the presence of code smells may increase the chances of bugs occurrence.
Ours is the first work that extensively investigates the code quality of 59 open-source, deep learning applications. It can help the researchers better understand the code quality and maintainability of the deep learning applications that are likely to grow in the coming years. Similarly, it can help the practitioners to calibrate their development practices by detecting and refactoring their code smells that are also likely to grow. Our replication package [1] can also be used for replication and third-party reuse.

1. `https://github.com/Hadhemii/DLCodeSmells`

CHAPTER 5 CLONES IN DEEP LEARNING CODE : WHAT, WHERE, AND WHY ?

5.1 Introduction

The main purpose of deep learning is to construct models with high performance that are able to learn knowledge from the input data in order to make predictions for new data. To find the optimal model, deep learning developers experiment with multiple prototypes using different configurations. They then compare the performance of the different models to identify the best configuration leading to the most efficient model. And as DL developers may have to follow the same or similar steps to build models without or with some modifications, they often end up writing poor code ; duplicating functions or blocks of code. Therefore, creating clones. Code clones are often also created through code reuse. Indeed, reusing code with or without modification by copying and pasting fragments from one location to another is a common practice during software development and maintenance activities, including for deep learning code [5]. For example, deep learning developers can clone models' architectures and model (hyper)parameters settings or initialization. Given the known negative impacts of code clones on traditional software systems and the complexity of deep learning systems, it is reasonable to assume that code duplication (whether within the same project or between projects) is likely to pose challenges to the maintenance of deep learning projects. Code clones constitutes technical debt and Sculley et al. [4] observed that technical debt can cripple machine learning systems when developers fail to follow good software development practices.

In fact, ML-based systems hold the maintenance concerns of traditional systems as well as those specific to ML.

Although the impact of code cloning practices on the quality of traditional software systems is relatively well understood, since it has been the subject of multiple investigations (e.g., [14, 15, 16, 17, 18, 19, 20]), we still know little about the impact of code cloning practices in machine learning based systems, despite the recent upsurge in the development of machine learning or in particular deep learning based systems. To date, no study has examined the code quality of DL software systems with respect to code cloning. We currently don't know the potential risks associated with the practice of duplicating code in deep learning systems. In this chapter, We aim to fill this gap by empirically investigating clones in deep learning systems. We perform a comparative study of the distribution and bug-proneness of clones in deep learning and traditional systems. We analyze clones in 59 deep learning systems and 59 traditional systems and compare their distribution from the perspectives of clone types and their locations. To gain further insights into the reasons behind code cloning and developers' code cloning practices in deep learning systems, we randomly select six deep learning projects and perform a manual analysis. We build a deep learning code clones taxonomy, where clones are associated to their corresponding deep learning phase. We further study the relation between bug-proneness and code clones in the context of deep learning systems. Finally, we identify the phases of the development process of deep learning systems in which code cloning is the most risky.

Our empirical study resulted in the following key findings :

— Cloning is frequent in deep learning code, we found that code clones occurrences in deep learning code is higher than in traditional code. All three clone types (Type 1, Type 2, and Type 3) are more prevalent in deep learning code than in traditional systems.
— Fragments of code clones in deep learning systems are dispersed. We found that the majority of code clones in deep learning code are located in different files. i.e, in the same directory or in different directories.
— Code clones in deep learning code are likely to be more defect-prone compared to non-cloned code and Type 3 clones are at higher risk of bugs compared to other clone types.
— We classify code clones by deep learning phases and found that three main deep learning phases are more prone to code cloning : model construction (36.08%), model training (18.56%), and data preprocessing (18.56%).
— We found clones from the following phases to be the most risky. We display the percentage of co-occurrence between bug-fix and code clones with respect to these DL phases : model construction (50%), model training (20%), data collection (13.3%), data preprocessing (10%), data post-processing (3.3%), and hyper parameter tuning (3.3%).

Chapter Overview Section 5.2 presents the design of our empirical study. Section 5.3 describes our findings and discusses the results of each research question. Section 5.4 discusses the implications of our findings. Threats to validity are presented in Section 5.5. Finally, Section 5.6 concludes the chapter.

5.2 Study Design

In this section, we first present our research questions by highlighting the motivation and research objectives. Then, we outline how we answered them by detailing the research methodologies.

5.2.1 Study Objectives

In our empirical study, we first examine the distribution of code clones in deep learning code in terms of clone type and clone location. Then, we compare them to the distribution of code clones in traditional code. Second, we investigate the relationship between code clones and bug-proneness in deep learning code. Third, we examine the reasons behind code duplication in deep learning code and build a taxonomy of code clones occurring in different phases of the development process of deep learning systems. Finally, we determine the riskiest phases or activities of deep learning application development by analyzing the bug-proneness of clones in each phase. To achieve these above-mentioned research objectives, we empirically investigate the following five research questions :

RQ1 : Do code clones occur more frequently in deep learning code than traditional source code ?

Code reuse by code cloning is a common practice in software development. Despite the intuitive productivity gain that one can expect from reusing code through code cloning, there are evidences showing that clones can negatively impact software quality ; increasing complexity and maintenance costs. Although code clones have been widely investigated for traditional software systems [21], the impact of code cloning in deep learning systems is still unknown. Moreover, given the widespread use of common open-source libraries and frameworks, the use of code examples from crowd-source question-answering web sites (e.g., Stack Overflow) and open-source repositories, and the repetitive use of similar development phases or tasks (e.g., data preprocessing, model training) during deep learning system development, it is reasonable to expect that code clones would exist in deep learning systems. Since we know from the studies on traditional software systems, that the impacts of clones vary based on the types and frequency of clone occurrences, it is therefore important to examine the frequency and distribution of clones in deep learning systems. A comparative analysis of code cloning in traditional and deep learning based systems could help leverage knowledge of clones in traditional systems to improve the maintenance and management of clones in deep learning systems.

RQ2 : How are code clones distributed in deep learning code in comparison to traditional source code ?

Code clones location impacts the refactoring cost. Navigating into distant duplicated code fragments adds comprehension overhead. Respectively, dispersed code clones can be hard to manage and may incur an increased cost of maintenance. To understand where deep learning practitioners duplicate code, we study the distribution of code clones in deep learning and traditional codes. We employ the taxonomy used by Kapser and Godfrey [38] to categorize the detected code clones by their locations (i.e, same file, same directory, and different directories).

RQ3 : Do cloned and non-cloned code suffer similarly from bug-proneness in deep learning projects ?
Studies of code clones in traditional software systems suggest that clones can have an adverse impact on the maintainability of the system ; increasing the risk of fault [14, 15]. However, it is unclear if cloning in deep learning systems is equally risky. This research question investigates the relationship between bug-proneness and code clones occurrences in deep learning code. We examine the effect of different types of clones on the bug-proneness of deep learning code and assesses the effort required to fix bugs in cloned and non-cloned code (by computing the time to fix of each bug).

RQ4 : Why do deep learning developers clone code ?
Given the complexity in deep learning code, constructing an efficient DL model can be a tedious job. DL developers should be experienced in the problem domains and should also have a sufficient understanding of deep learning tricks. They also need to have coding skills with deep learning frameworks as well as the ability to manage the computing resources. When faced with a new task, to mitigate the risk of writing erroneous code, DL developers often duplicate the code of an existing model with the same or similar logic with or without modifications, depending on the requirements of their task. To understand activities in the development of deep learning applications that are more prone to code duplication, we conduct a manual analysis of code clone classes. We categorize clones based on the development phases in which they occurred. This analysis allows us to identify activities (i.e., phases) in the deep learning development process were code cloning occurs frequently.

RQ5 : Where in the deep learning code is cloning the most risky ?
Since previous works on traditional systems [88] have shown that the risk of faults in cloned code vary depending on the types of code that is cloned, we are interested in investigating whether clones occurring at certain stages of the development process of deep learning systems and–or in certain functions of the deep learning code are more bug-prone than others. Therefore, in this research question, we compare the risk of faults of clones found in the different functions of the deep learning code.

5.2.2 Study Overview

In this section, we present our study design as shown in Figure 5.1. We can divide our methodology into five main parts.

Fig 5.1-A We first clone 59 DL and 59 traditional repositories from GitHub, and the

detailed methodology is described in the Subsection 5.2.2.1. We then detect code clones for both DL and traditional open-source projects using the NiCad clone detector (details are presented in section 5.2.2.3). We finally compare the clone distribution between both type of systems (i.e., DL and traditional systems) in terms of lines of code and clone types.

Fig 5.1-B We analyze the distribution of code clone by applying the location taxonomy. As shown in Fig 5.1-B, we have three locations where there might be code clones classes : same file, same directory, and different directories (details are presented in Section 5.2.2.4). An arrow towards a Balance as shown in the figure 5.1-B represents the comparative analysis of the distribution of code clones between deep learning and traditional systems in terms of localization.

Fig 5.1-C The third part involves studying the relationship between code clones occurrences and bug-proneness. We detect code clones for each commit of a set of six deep learning repositories (discussed in Section 5.2.2.2). Then, we identify bug-fixing commits relying on the commit history. Next, we extract bug-inducing commits by applying the SZZ algorithm. Once we have bug-inducing commits with their corresponding changed lines, we match these buggy lines with lines that are cloned to find the relationship between bug and clones (details are shown in Section 5.2.2.6).

Fig 5.1-D We manually classify code clones that are DL-related to construct a taxonomy of code clones in DL code (see Section 5.2.2.5).

Fig 5.1-E Finally, we examine the risk of bugs of code clones occurring in specific functions of the DL code.

5.2.2.1 Subject Systems

We mined 112 open-source projects (59 deep learning + 59 traditional projects) from GitHub. We used the same projects investigated in our first study (Chapter 4) on detecting code smells in DL code. Our empirical study is based on this set of projects (Chapter 4) including only projects where the main language is Python as it is the most widely used language in the machine learning field [77]. Deep learning projects were selected by first searching repositories using DL-related keywords (e.g., deep learning, deep neural network, convolutional neural network) and manually filtering out tutorials and some projects with a low number of releases (to obtain our 59 DL repositories). To obtain our dataset of traditional systems (i.e., non deep-learning code), we used a benchmark from an existing study by Chen et al. [1] (similarly to Chapter 4). This benchmark consists of 106 repositories with at least 1k stars each from which randomly selected a subset of 59 traditional projects. We use the same set of 59 traditional projects, selected in Chapter 4, for the comparative analysis in this chapter.

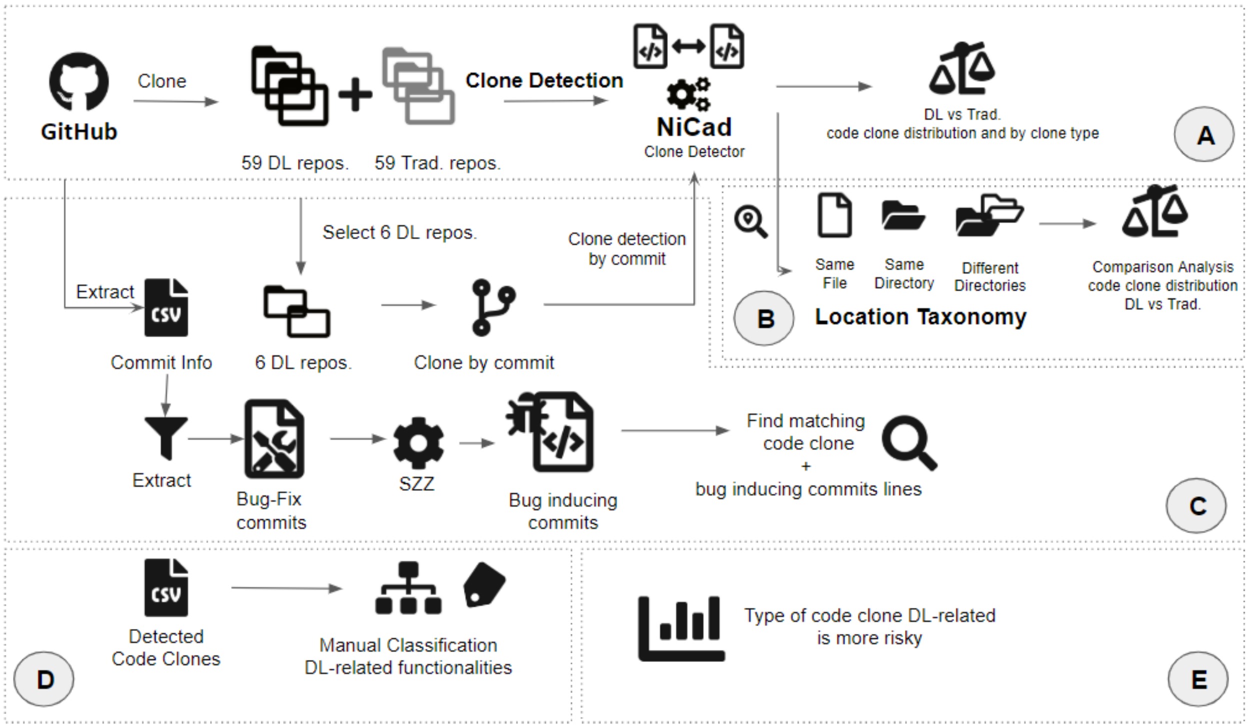

Figure 5.1 Study Overview A- Detecting code clones Deep Learning and Traditional repositories, B- Applying Code Clone Location Taxonomy, C- Studying the relationship between bug-proneness and code clones, D-Classifying code clones manually based on DL-related functionalities, E-Exploring riskier DL-related code clones

5.2.2.2 Preprocessing of source code repositories

Selection of a Subset of Subject Systems : The majority of our research questions (RQ1, RQ2, and RQ3) are based on the analysis of 59 traditional and 59 deep learning systems. However, we randomly select 6 deep learning projects out of the 59 projects to analyze the distribution and impacts of clones associated with different phases or activities in deep learning system development by manual investigation (RQ4 and RQ5). We select a subset of systems to keep the cost and effort of manual analysis in a feasible range. We use this subset of systems to study the relationship between bug-proneness and code clones in deep learning code because detection of code clones by NiCad can be very time-consuming especially for the detection of clones at every commit as we did for RQ4 and RQ5. So, we use the six selected systems to perform manual analysis for extracting the taxonomy of code clones in deep learning code regarding development steps. Our selected systems are from diverse application domains with varying (small to medium) size (SLOC) and lengths (in number of commits) of evolution history.

Computing Source Code Lines : We use the SLOCCount [89] tool to compute the total source lines of code (SLOC) of python code in each project. SLOCCount is a software metrics open-source tool developed by David A. Wheeler supporting several programming languages. It can handle awkward situations in many languages such as the use of string constants as comments in Python code. Since one project could contain different programming languages, SLOCCount results list the available languages in the system. In front of each language name, we have a total SLOC. We select the total source lines of code number shown next to Python. We execute the SLOCCount tool for both DL and traditional repositories. We also measure the SLOC for each commit of each DL repository from the selected set of six deep learning repositories for further analysis. We normalize the detected lines of code clones by the size of the project (SLOC) to have a more accurate comparison of deep learning and traditional systems regarding the frequency and distribution of clones. We divide the total lines of cloned code for each project by the total lines of code of the corresponding project.

5.2.2.3 Code Clone Detection

For detecting code clones, we use the *NiCad* tool [90]. NiCad [91] can detect both exact and near-miss clones with high precision and recall [92] with respect to blocks and functions granularity.

NiCad Settings : Table 5.1 shows the setup for detecting the three types of clones. We detect code clones with a minimum size of five lines of code. We detect both exact clones (Type 1) and near-miss (Type 2 and Type 3) clones. We use the blind renaming option for NiCad for the detection of Type 2 and Type 3 clones. Type 3 clones are detected with a dissimilarity threshold of 30%.

Table 5.1 NiCad Settings

Clone Type	Identifier Renaming	Dissimilarity Threshold	Size (LOC)
Type 1	none	0%	[5-2500]
Type 2	blind	0%	[5-2500]
Type 3	blind	30%	[5-2500]

Clone Detection : We detect code clones for a particular snapshot and for each commit of each repository. To perform a comparative analysis of the frequency and distribution of clones in deep learning and traditional code, we detect three types of code clones (Type 1, Type 2, and Type 3) in both deep learning and traditional code using the NiCad settings detailed in Table 5.1. We detect clones on a particular snapshot, which in our case is the last available version of the project on GitHub at the time of cloning the repository. For RQ1, we use both granularities (function and block) to detect code clones. For the rest of the research questions (RQ2-RQ5), we analyze code clones at function granularity. To study the relation between the bug-proneness and code clones, we use the commit history to extract commits information of each repository from the set of six deep learning repositories and detect code clones for every commit of the repositories.

Results Cleaning : We exclude matching clone classes between Type 1 and Type 2 from the outputs of Type 2 clone detection results. As NiCad results for Type 2 contain clone classes from Type 1 since the set of Type 1 clones is a subset of the set of Type 2 clones. Hence, we remove matching clone classes based on the clone fragment specifications (file path, start and end line number, etc). Thus, if the same clone class (i.e., containing the same set of clone fragments specifications) exists in both Type 1 and Type 2 clone detection results, we remove such clone classes from Type 2 results as they are Type 1 clones. So, the filtered Type 2 results contain exclusively Type 2 clones without any Type 1 clone fragment in it. Similarly, we exclude the matching Type 1 and Type 2 classes from Type 3 clone detection results to have Type 3 clones exclusively. This is because Type 3 clone results by definition contain Type 1 and Type 2 clone fragments which need to be removed to perform the clone-type centric analysis correctly.

5.2.2.4 Location taxonomy based labeling of clones

To answer RQ2 that investigates the distribution of code clones in deep learning and traditional codes in terms of location, we apply the taxonomy used by Kapser and Godfrey [38]. It categorizes the detected clones based on their relative locations in the file system structure for both types of projects. We label a clone class by ***'Same file'*** when all the fragments in the detected clone class are from the same file. We label a clone class by ***'Same directory'*** when all the files associated with the detected clone class belong to the same directory. And finally, we assign a clone class to the ***'Different directories'*** label, when clone fragments are from different files located in different directories.

We then calculate the proportion (percentage) of clones distributed over the location types defined by the taxonomy. We perform Mann-Whitney statistical test to compare the distribution of code clones in DL and traditional systems. We further extend our comparative analysis by considering individual clone types.

5.2.2.5 Labeling clones based on the taxonomy of deep learning tasks

Since there are common functionalities followed to build deep learning models and because of the use of the same deep learning libraries, deep learning practitioners often copy-paste ready to use functionalities with or without modification. This aims to gain productivity and to reduce the risk of writing erroneous new code when tested implementations are available. We expect code duplication not only in the code related to core functionalities of the deep learning phases but also in code related to model testing.

Therefore, we categorize different types of cloning practices by DL phases and label them via a manual inspection of the code clones found in the selected six DL repositories. We found 577 clone fragments in these six deep learning repositories. We manually analyzed each of these clone fragments. We used a bottom-up approach, where we first assigned each clone fragment with a label corresponding to the DL tasks or functionality to which it is related. We further grouped the clone fragments in each subcategory and mapped them to the different phases of the development process of deep learning models, as discussed in Chapter 2. We ensure that the relation between a subcategory and a category is : “to perform". For example, *initialize weights to perform* a *model construction*. Here, the sub-category is ‘initializing weights’ and category is ‘model construction’. We also categorize and label functions that are not related to deep learning such as logging, test, etc, as ***’others’***. The manual classification for generating this DL taxonomy was done by multiple persons, all with academic and industry backgrounds. The resulting taxonomy was then cross-validated, and disagreements were resolved by group discussion.

Table 5.2 presents some real-world examples from the detected clones. In case of exact (Type 1) clones, we show only one code fragment from the clone class since all the fragments are identical except for formatting differences. For near-miss clones we show both fragments in a clone pair. In Table 5.2, we present the clone types of the example fragments, and the sub-category and category of DL-related phases to witch the clone class or fragments belongs. Our objective is to first assign a sub-category (DL sub-task) for each clone class and then group them into top-level categories (DL phases).

The first example in Table 5.2 represents a Type 1 clone fragment. Here, the function `iou(box1, box2)` computes Intersection Over Union (IOU) between the predicted box and any ground truth box. It is a metric that computes the accuracy of YOLO (You Only Look Once) [93], a real-time object detection system based on Convolutional Neural Network (CNN). Therefore, we designate this clone class to the sub-category ‘Measure model accuracy’. As this computation function is performed to train the model, we label it as the **‘Model training’** category.

The second example contains two fragments of Type 3 clones. The purpose of the first function (`read_images_from_file`) is to read multiple images from the given file. Whereas the second function (`read_images_from_url`) reads input images from a given URL. Thus, the function of this clone class is to load data. Since load data is performed to collect data, we assign this clones class to the **'Data collection'** phase of deep learning.

The last example in Table 5.2 is a Type 1 clone fragment. This code fragment (*process_inceptionv3_input*) prepares a given image for the model input requirements. In this case, the model is Inception v3 [94], which is a Deep Convolutional Neural Network (CNN) with 48 layers. We assign this clone class to the 'Resize image' sub-category and we label it as belonging to the **'Data preprocessing'** category.

5.2.2.6 Code Clone and Bugs

Several studies have been focused on investigating the relationship between clones and bug-proneness. Some studies based on traditional software systems have shown that code clones may introduce bugs and negatively impact software maintainability. Therefore, it is important to study whether and to what extent this relationship holds in the context of deep learning systems.

Detecting bug fixing commits : We extract all the commits information from each of the six selected repositories. We leverage a keyword-based approach to classify commits relying on keywords occurrence such as *'bug', 'fix', 'solve', 'problem'* in the commit messages. We use the set of keywords used by Rosen et al. [83]. At least one of the keywords from this set should be in the commit message to consider this commit as a bug-fixing commit.

Bug Inducing commits : We use PyDriller [95] to extract bug-inducing commits for each bug-fix commit. PyDriller is a python-based framework that supports mining information from Git repositories. We used PyDriller to extract information such as commits and diffs from each of the selected repositories. We mainly use this framework to get bug-inducing commits given a bug-fix commit. It returns the set of commits that last modified the lines that are changed in the files included in the bug-fix commit by applying the SZZ algorithm.

Bug Proneness of code clone : We define a bug to be related to code clone if the lines changed in bug-fix commits intersect with lines in between the start line and end line of the detected code clones. We further analyze to identify riskier DL-related functionalities that are likely to introduce bugs. Hence, we similarly match lines changed in bug-fix commits with the corresponding lines of the cloned DL-related functions. We then extract the most frequently occurring cloned DL-related functions related to bugs in DL projects. Therefore,

Table 5.2 Categorization of Clone codes examples

Code Fragment	Clone Type	Sub-category	Category
`def iou(box1, box2): tb = (min(box1[0] + 0.5 * box1[2], box2[0] + 0.5 * box2[2]) - max(box1[0] - 0.5 * box1[2], box2[0] - 0.5 * box2[2])) lr = (min(box1[1] + 0.5 * box1[3], box2[1] + 0.5 * box2[3]) - max(box1[1] - 0.5 * box1[3], box2[1] - 0.5 * box2[3])) if tb < 0 or lr < 0: intersection = 0 else: intersection = tb*lr return intersection / (box1[2] * box1[3] + box2[2] * box2[3] - intersection)`	Type 1	Measure model accuracy	Model training
`def read_images_from_file(filename, rows, cols, num_images, depth=1): """Reads multiple images from a single file.""" ... _check_describes_image_geometry (rows, cols, depth) with open(filename, 'rb') as bytestream: return images_from_bytestream (bytestream, rows, cols, num_images, depth)`	Type 3	Load data	Data collection
`def read_images_from_url(url, rows, cols, num_images, depth=1): """Reads multiple images from a single URL.""" ... _check_describes_image_geometry (rows,cols, depth) with urllib.request.urlopen(url) as bytestream: return images_from_bytestream (bytestream, rows, cols, num_images, depth)`			
`def process_inceptionv3_input(img): image_size = 299 mean = 128 std = 1.0/128 dx, dy, dz = img.shape delta = float(abs(dy - dx)) if dx > dy: #crop the x dimension img = img[int(0.5*delta):dx -int(0.5*delta), 0:dy] else: img = img[0:dx, int(0.5*delta):dy -int(0.5*delta)] img = cv2.resize(img, (image_size, image_size)) img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB) for i in range(3): img[:, :, i] = (img[:, :, i] - mean) * std return img`	Type 1	Resize image	Data preprocessing

we obtain which functions of DL code are more likely to introduce bugs than others when cloned.

Time to fix bugs when it is related to code clones : We investigate whether or not clones have impacts on the time required to fix a bug. The objective is to know whether clones hinder bug fixing ; making bugs long-lived in the deep learning systems. We thus study the comparative time it takes to fix bugs when it is related and not related to clones respectively. Thus, we compute the difference in time between bug fixing commit and their related bug inducing commit as introduced by Kim and Whitehead [85]. Therefore, we extract from the commit history the time of each bug-introducing commit as well as the time of their corresponding bug-fix commit. We calculate the bug-fix time by taking the difference between bug-fix commit and bug inducing commit. Once we have the time difference (in seconds) we carry out a non-parametric Man-Whitney test to compare if there is any significant difference in the time required to fix bugs related and not related to clones.

5.3 Study Findings and Discussions

In this section, we present the results of our study in details, and answer our five research questions as follows.

5.3.1 RQ1 : Do code clones occur more frequently in deep learning code than traditional source code ?

Due to the complexity, the lack of explainability of deep learning code, and the excessive use of ready to use routines from popular deep learning frameworks and libraries, deep learning code is likely to have duplicated code fragments i.e., code clones. Although many studies investigated the distribution, evolution, and impacts of clones regarding traditional software, none of the existing studies, has investigated the code cloning practices in deep learning code. Thus, in this research question, we study the distribution of different types of code clones in both deep learning and traditional systems, to understand and compare the prevalence of clones in these two types of software systems (i.e., deep learning and traditional software software systems).

To answer this research question, we detect code clones in 59 deep learning systems and 59 traditional software systems. We calculate the number of occurrences of code clones across different dimensions (e.g., project type, clone type, clone granularity). Then we compare the density of clones in DL-based systems with that of the traditional systems. First, we calculate the total number of lines of code clones in each project and then divide that by the total number of source code lines (i.e., SLOC) for normalized representation of the clone density. We count SLOC using the tool SLOCCount as discussed in section 5.2.2.2. We then perform the Mann-Whitney Wilcoxon (MWW) test [96] to compare the distribution of clones in deep learning and traditional systems by testing if there exists any statistically significant differences in clone densities in these two types of systems. To have deeper insights, we also compare the clone densities with respect to the three clone types (Type 1, Type 2, and Type 3). However, MWW test only verifies whether the difference between two set of observations is by chance and does not express the effect or magnitude of the differences. Thus, we calculate the effect size to determine the magnitude of the differences between each two distributions. We measure Cliff's Delta [97], which is a non-parametric estimate of effect size and does not require data to be normally distributed. When Cliff's Delta is beyond of 0.2 and below 0.5, then the effect size is low. When it is beyond 0.5 and below 0.8, the effect size is medium. Beyond 0.8, the effect size is large. In this research question, we present the findings for both clone granularities : function and block.

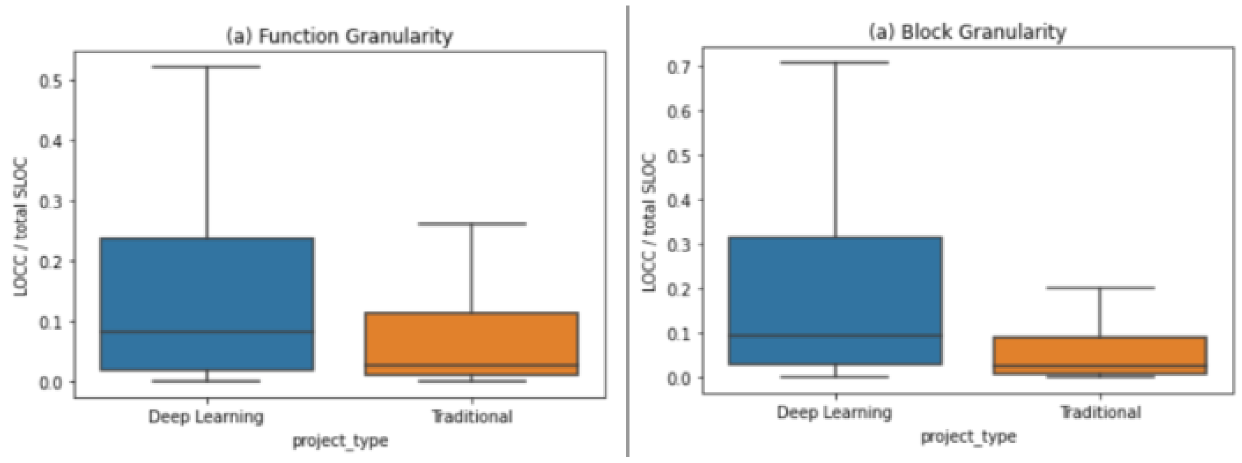

Figure 5.2 Code clones occurrences in DL and traditional projects for both code clones granularities : (a) Function, (b) Block. ***LOCC*** : Lines Of Code Clones, ***SLOC*** : Source Lines Of Code

Clone Occurrences by Project Type : Fig. 5.2 shows the comparison of clone occurrences between DL-based and traditional systems considering all clone types and for both function and block granularity. The box plots represent *clone density* defined by the normalized lines of code clones per lines of source code for both deep learning and traditional software systems. This metric computes the density of clones as a ratio of the total lines of cloned code and the total lines of source code in the corresponding systems. In Fig. 5.2, we observe that the median of the clone density for deep learning systems is comparatively higher than that of traditional systems. We observe the similar differences for both function and block granularity. This suggests that deep learning systems tend to have higher proportions of cloned code compared to traditional systems.

To investigate whether the observed differences that DL systems having higher density of clones compared to traditional systems are statistically significant i.e, that the difference is not by chance, we perform Mann-Whitney Wilcoxon (MWW) tests [96] (two-tailed, significance at < 0.05). We chose the MWW test because it is a non-parametric test and thus does not assume data to be normally distributed. Also, it can be applied on small sample sizes. We present our statistical test results in Table 5.3. The column 'Total' in Table 5.3 shows the p-values for MWW test for all clone types. The p-values for function and block granularity are 1.78e-07 and 3.01e-10 respectively which are < 0.05 indicating that our observation of DL systems having higher density of clones compared to traditional systems is statistically significant.

Now to observe the magnitude of the differences in clone density of DL and traditional systems, we analyze Cliff's delta effect sizes. As shown in Table 5.3, the effect size in column 'Total' (which includes all clone types) for function granularity belongs to the large category as it is equal to 0.8. Thus, we have an 80% chance that deep learning code will have higher density of function clones than traditional code. Whereas for block granularity, we have an equal likelihood ($0.53 \sim 0.5$) of having higher density of block clones in deep learning code in comparison to traditional code. So, when we consider all clone types, the observed higher clone density in deep learning systems in comparison to traditional systems is statistically significant with medium to large effect size. We therefore concludes that deep learning systems tend to have higher proportions of cloned code compared to traditional systems. However, this may depend on confounding factors, i.e, using the same libraries/frameworks, having the same decision logic across deep learning models construction, as explained by the difference in effect size for function (high) and block (medium) granularity, despite the differences in clone density being statistically significant for both granularities.

Table 5.3 Mann-Whitney test and Cliff's Delta Results between DL and Traditional Projects

Clone Types	**Type 1**		**Type 2**		**Type 3**		**Total**	
Granularity	**Function**	**Block**	**Function**	**Block**	**Function**	**Block**	**Function**	**Block**
p-value	3.13e-06	7.61e-08	3.38e-05	4.17e-06	1.77e-03	2.87e-04	1.78e-07	3.01e-10
Cliff's Delta	0.58	0.62	0.46	0.60	0.32	0.37	0.8	0.53

Code Clone occurrences by Clone Type : In Fig. 5.3, we compare clone densities for deep learning and traditional systems regarding individual clone types (Type 1, Type 2, and Type 3) for both function and block granularity. To calculate clone density for individual clone

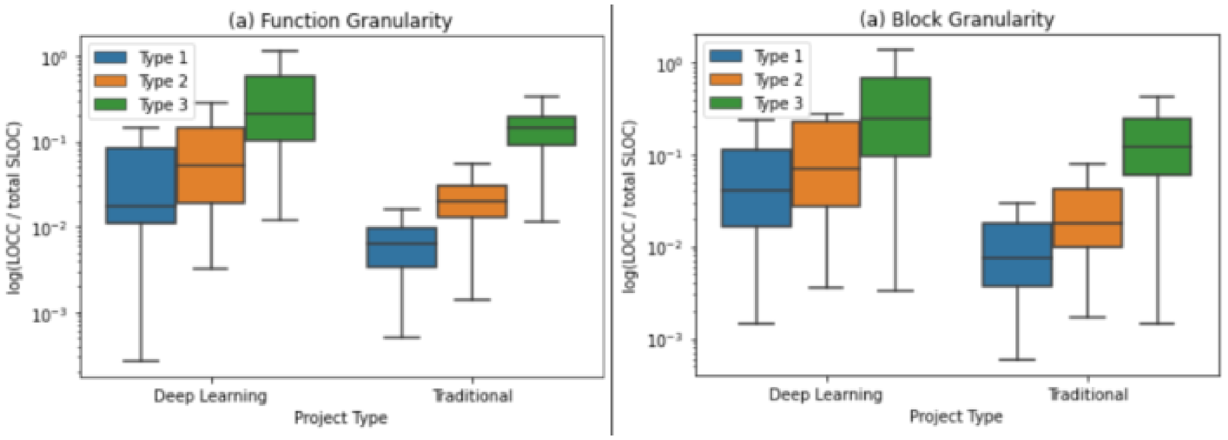

Figure 5.3 Code Clones Occurrences by Project Type and by Clone Type and by Granularity ***LOCC :*** *Lines Of Code Clones*

types, we divide the total lines of cloned code for a particular clone type by the total number of source code lines in a given systems. We calculate clone density for both function and block granularity. We use a log scale for the y-axis to make the results more clear. As shown in Fig. 5.3, for deep learning systems, the density of Type 3 clones is the highest followed by Type 2 and Type 1 clones respectively. We observe the same trends in comparative densities for all types of clones in traditional systems, and for both function and block granularity in both types of systems. Now, when we compare the clone densities in deep learning and traditional systems based on the box-plots in Fig. 5.3, we observe that for all types of clones, deep learning systems have higher median for clone densities compared to that of traditional systems. The same overall trend holds for both function and block granularity. This suggests that deep learning systems tend to have higher density of cloned code compared to traditional systems regarding all three clone types.

To verify whether these differences of deep learning systems having higher densities of clones compared to traditional systems are statistically significant, we perform MWW tests (two-tailed, significance at 0.05) on the corresponding clone densities for all individual clone types, for both function and block granularity. We also compute the Cliff's delta effect size to determine the magnitude of the differences in clone densities for DL and traditional systems. Table 5.3 shows the p-values of the MWW tests along with the values for the corresponding effect size. We observe that there is a statistically significant difference between deep learning code and traditional code with respect to Type 1 with p-values of 3.13e-06 and 7.61e-08, which are < 0.05 for both function and block granularities, respectively. The values of Cliff's Delta are 0.58 and 0.62 which belong to the 'medium' category. We also found a statistically significant difference for Type 2 clones (p-values of 3.38e-05, 4.17e-06). The values of effect size for Type 2 for both function and block granularities are medium with values of 0.46 and 0.60. Similarly, we also obtained a statistically significant difference between deep learning code and traditional code regarding Type 3 clones. The effect size for Type 3 clones is small with values of 0.32 for function granularity and 0.37 for the block granularity. Overall, the results of our clone-type based analysis show that Type 3 clones comprise the highest proportion of clones in deep learning code. Moreover, for all clone types, deep learning code has higher density of clones than traditional systems, and these differences are statistically significant. This indicates an over all trend of deep learning systems having more clones than traditional systems, although the magnitude of the differences may not always be 'large'.

These results support our hypothesis about the existence of a higher proportion of code clones in deep learning code compared to traditional systems. The observed prevalence of code clones in deep learning systems underscores the importance of investigating the reasons for such cloning practices and their impacts on the quality of deep learning systems, which we do in the remaining research questions.

Summary of findings (RQ1) : As shown by our experimental results, the density of code clones in deep learning systems tend to be higher than that of traditional systems, and the difference is statistically significant. Regarding clone types, all three clone types (Type 1, Type 2, and Type 3) are more prevalent in deep learning-based systems than in traditional software systems.

5.3.2 RQ2 : How are code clones distributed in deep learning code in comparison to traditional source code ?

As we observed in RQ1, the density of code clones in deep learning code is higher compared to traditional systems. We aim to further explore the distribution of code clones in terms of their locations. Since, distant code clones may hinder the maintenance process by adding navigation and code comprehension overhead, it is of interest to study how the code clones are distributed in the deep learning system in terms of their locations.

We categorize the detected code clones classes by their location based on the taxonomy proposed by Kapser and Godfrey [38]. If a clone class contains code fragments that are all from the same file, we label them as belonging to the 'Same File' category. When the clone fragments are from the same directory but from different files, we assign them to the 'Same directory' category. Else, if the clone fragments of a clone class are from files from different directories, we categorize them as belonging to the 'Different directories' category (further details about this classification are provided in section 5.2.2.4). We then count the percentage of lines of code clones of different location categories for deep learning and traditional systems for comparison. We also perform the same analysis for individual types of clones (Type 1, Type 2, and Type 3) to have insights on their comparative location-based distributions in DL and traditional systems. The analysis based on the percentages of the lines of code clones, shows the distribution of clones from a relative volumetric point of view. However, how the clone fragments in the clone classes are disperse in the system is likely to have impacts on their maintenance. So, we also analyze the location-based distribution of the percentages of clone fragments for different types of clones. In addition, we manually investigate samples of cloned fragments from each location category to gain insights about the characteristics of the clones and better understand the intents and potential impacts of the proximity of clones (in the code base) and their relative distribution, on software quality.

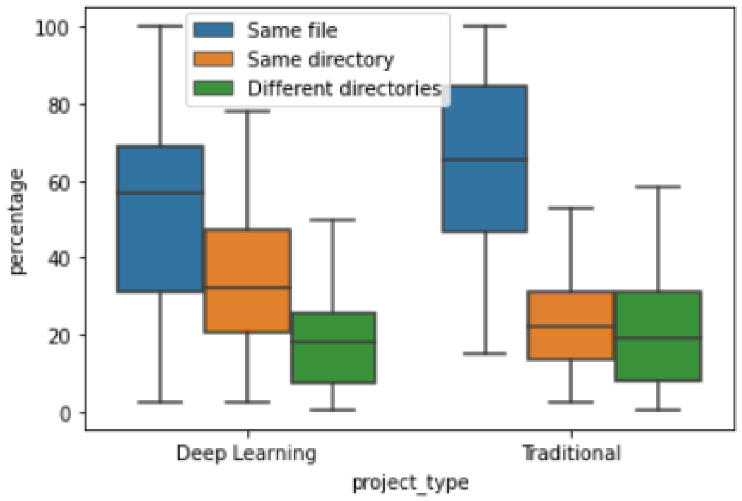

Figure 5.4 Code Clones Distribution by Location in DL and Traditional code regarding percentage of lines of code clones (LOCC). i.e, (LOCC/total LOCC)x 100

Overall location-based distribution of clones : Fig. 5.4 shows the distribution of code clones (for the function granularity) in DL and traditional source code, regarding the locations of the clones. Based on the median values of the percentages of cloned lines of code for each location type in deep learning systems, we observe the following location-based distribution of clones : the percentages of cloned lines of code in 'same file' are higher than that in 'same directory' which in turn are higher than the percentages of cloned lines of code in 'different directories'. In traditional systems, we observe a location-based distribution similar to that of DL systems with the highest percentage of cloned lines being in the 'same file' followed by the 'same directory' category and finally the 'different directories' category. With the distributions of 'same directory' and 'different directories' categories overlapping significantly.

Table 5.4 Mann-Whitney test and Cliff's Delta results regarding the distributions of clones in DL and Traditional (Trad) Projects.

Clone Type	**ALL**				**Type 1**				**Type 2**				**Type 3**			
Proj Type	**DL**		**Trad**		**DL**		**Trad**		**DL**		**Trad**		**DL**		**Trad**	
Location	**p**	**CD**	**p**	**CD**	**p**	**CD**	**p**	**CD**	**p**	**CD**	**p**	**CD**	**p**	**CD**	**p**	**CD**
SF - SD	**2.911e-04**	**0.4**	**8.34e-13**	**0.83**	**1.94e-3**	**0.41**	0.16	0.15	**7.64e-09**	**0.76**	**3.4e-10**	**0.86**	**6.98e-4**	**0.36**	**1.42e-12**	**0.83**
SF - DD	**2.9e-10**	**0.7**	**2.05e-11**	**0.81**	**0.018**	**0.32**	**0.01**	**0.42**	**5.98e-08**	**0.77**	**2.82e-10**	**0.91**	**3.66e-10**	**0.7**	**1.59e-10**	**0.77**
SD - DD	**4.79e-05**	**0.46**	0.15	0.13	**0.03**	**0.28**	**0.02**	**0.35**	0.09	0.21	**6.36e-03**	**0.41**	**2.57e-05**	**0.48**	0.29	0.07

SF : Same File, SD : Same Directory, DD : Different Directories, P : p-value, CD : Cliff's Delta

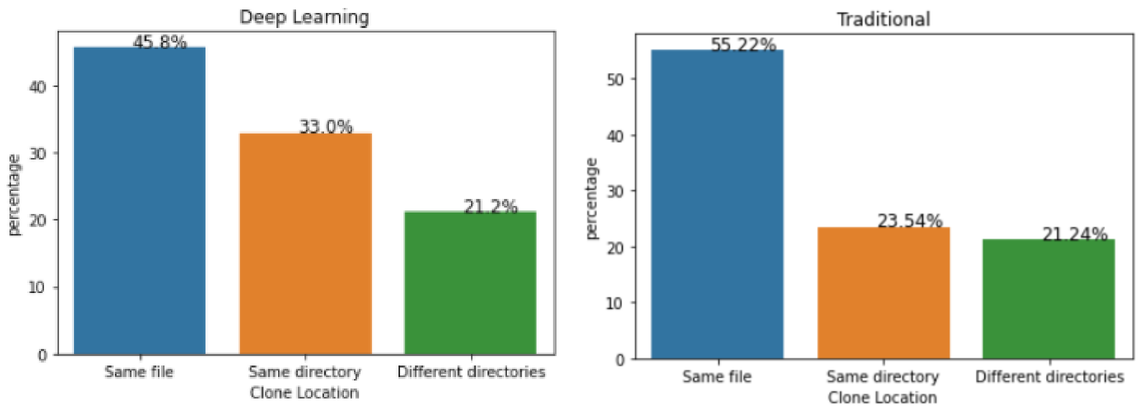

Figure 5.5 Percentages of Lines of Code Clones by Location of Clones in both Deep Learning and Traditional Systems

Table 5.4 shows the p-values from the Mann-Whitney test and Cliff's delta values of different code clones location between the same type of systems (DL and traditional code) with respect to the relation between code clones locations and clone types. ALL in the Table 5.4 designates the unfiltered Type 3 (i.e, include all fragments from Type 1, Type 2, and Type 3). We found a statistically significant difference between the 'same file' category and both the 'same directory' and the 'different directories' categories in the DL code with p-values equal to 2.91e-04 and 2.9e-10 (which are < 0.05) respectively and with medium effect size of 0.4 and 0.7, respectively. Thus, in DL systems, a significantly higher percentages of the cloned code resides in 'same file' compared to the percentages of cloned code that resides in the 'same directory' and in 'different directories'. Similarly, the percentages of cloned lines in the 'same directory' is significantly higher compared to the percentage of clones contained in 'different directories' with p-value equals to 4.79e-05 (< 0.05) and with a medium effect size. For traditional code, we observe that a statistically higher percentage of cloned lines of code is located in the 'same file' compared to the percentage of clones that are located in the 'same directory' and in 'different directories', with p-values < 0.05 (8.34e-13 and 2.05e-11 respectively) and with large effect sizes (0.83 and 0.81 respectively). We found no significant difference between the percentage of clones contained in the 'same directory' category and the 'different directories' category, for traditional code.

For further insights, we analyzed the average percentages of lines of cloned code by their locations in deep learning and traditional code. As shown in Figure 5.5, we identify that 45.8% of the DL-related clones are in the 'same file', 33% are in the 'same directory' and 21.2% are in 'different directories'. Hence, DL clones are more dispersed having fewer percentages of clones in the same file and more than 54% (33% + 21.2%) in different files and directories. Code clones in traditional code, on the other hand, are more localized. More than the half of the code clones in non-DL code (55.22%) are in the same file, 23.54% are in the same directory and 21.24% are in different directories. Therefore, according to our results, code clones in deep learning code are more dispersed than code clones in non-DL systems. This dispersion in the deep learning code may harm the maintenance of duplicated code due to potential navigation and comprehension overhead.

Location-based distribution of different types of clones : We analyze the location-based distribution of different types of clones as follows :

Type 1 : As shown in Fig. 5.6-A, the median of the distribution of the percentage of Type 1 cloned lines in 'same file' in DL code is the lowest compared to the percentages of cloned lines in 'same directory' and in 'different directory'. This shows that Type 1 clones in DL code are dispersed in different files and directories. However, for traditional systems, we observe that majority of the Type 1 clones are in 'same file' compared to 'same directory' and 'different directories'. This suggests that Type 1 clones in traditional systems reside in closer proximity unlike the Type 1 clones in DL systems.

To investigate whether the observed differences are statistically significant, we perform MWW tests (two-tailed, significance at 0.05) and measure Cliff's delta effect size. In table 5.4, we highlight in bold the statistically significant differences for the distributions of cloned lines in different clone locations for both DL and traditional code with respect to clone types where p-values are < 0.05. Fig. 5.6 represents these differences by showing the distribution of percentages of lines of code clones for each clone type and for both types of systems. Type 1 clones in deep learning code is less localized with a statistically significant difference between the same file location category and the others categories and with a small effect size. Whereas Type 1 clones in non-DL code shows a statistically significant difference only between the 'different directories' location and the others locations with lower distribution of lines of code clones in comparison to the distribution of line of code clones in other locations (Same file (SF), and Same Directory (SD)).

The analysis of exact clones (Type 1) in the same directory could shed lights on some of the implementation practices of deep learning developers. For example, the occurrence of exact functions in the same directory may suggest that when DL developers have a working code that builds a model properly, they are inclined to copy-paste this same code in another file in the same directory to construct a similar model to try another configuration. Building models may have the same common functions like computing accuracy or implementing the activation function. These functions could be exact for each model, which may explain the high occurrence of Type 1 clones in the same directory in deep learning projects, compared to traditional projects.

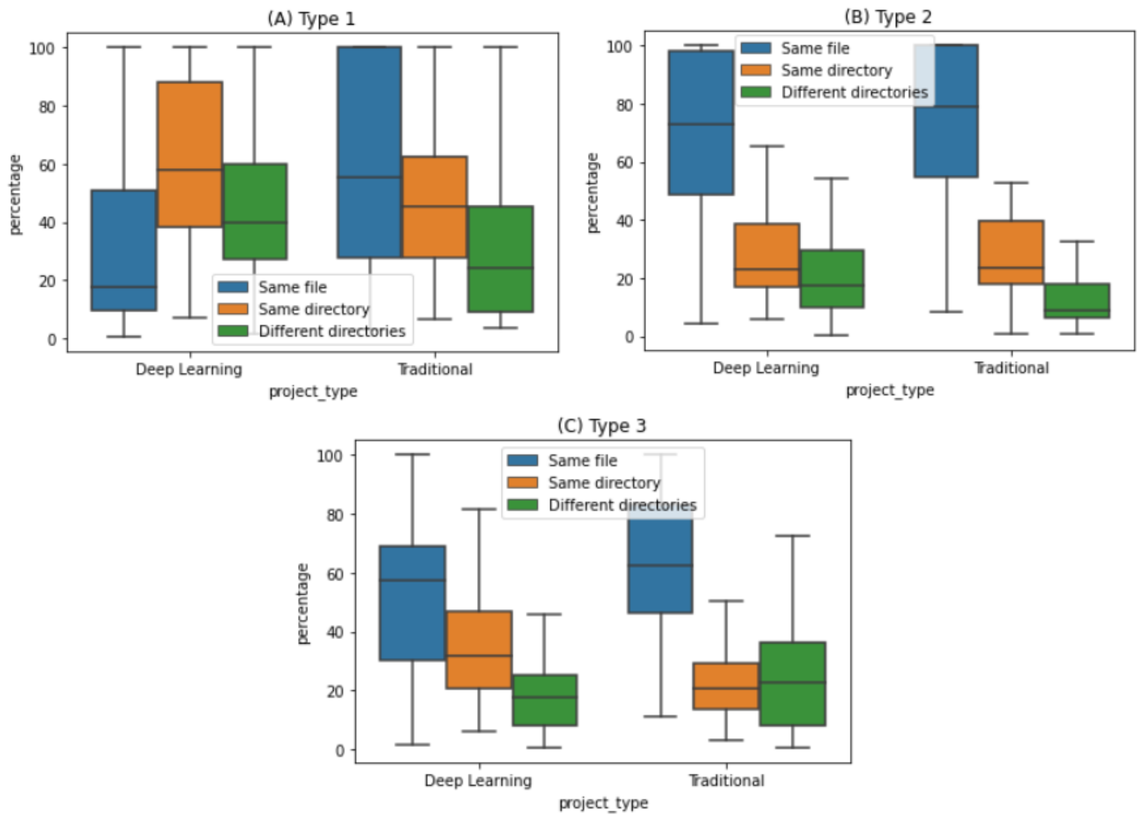

Figure 5.6 Clone type by clone location in DL and Traditional Code

Type 2 : As shown in Figure 5.6-B, the distribution of the percentage of the lines of Type 2 cloned code in 'same file' in DL code is higher compared to the distributions of type 2 clones in 'same directory' and 'different directories' categories. This implies that Type 2 clones in DL code reside in closer proximity. For traditional systems, we see a similar distribution of the percentage of Type 2 cloned lines in different locations category. However, the percentages of Type 2 clones in 'different directories' in DL code tend to be slightly higher compared to the same distribution (i.e., percentages of Type 2 clones in 'different directories') in traditional

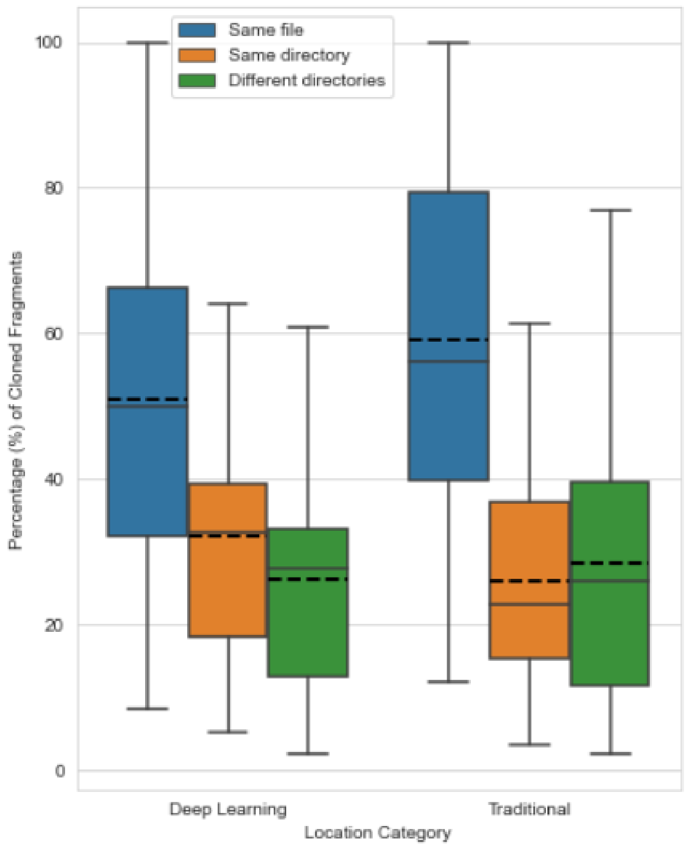

Figure 5.7 Distribution of percentage of number of fragments of code clones classes per clone location.

code.

We found a statistically significant difference between the percentages of cloned lines in 'same file' and in both 'same directory' and 'different directories' categories for Type 2 clones in DL code with p-values equal to 7.64e-09 and 5.98e-08, respectively (Table 5.4). The values of effect size of these differences are large between 'same file' and 'same directory' and 'same file' and 'different directories' (0.76 and 0.77, respectively). Hence, Type 2 clones are less dispersed in deep learning code than other types of clones. We notice a similar level of dispersion for Type 2 clones in non-DL code (as shown in Fig. 5.6-B).

Type 3 : As shown in Fig. 5.6-C, Type 3 clones are less dispersed compared to Type 1 clones but comparatively more dispersed than Type 2 clones in deep learning code. We present the results of the MWW tests in Table 5.4 where the values p-values are < 0.05 for deep learning code. The percentage of Type 3 clones located in the same file is the highest, in comparison to the percentages of Type 3 clones located in the 'same directory' and 'different directories'. In

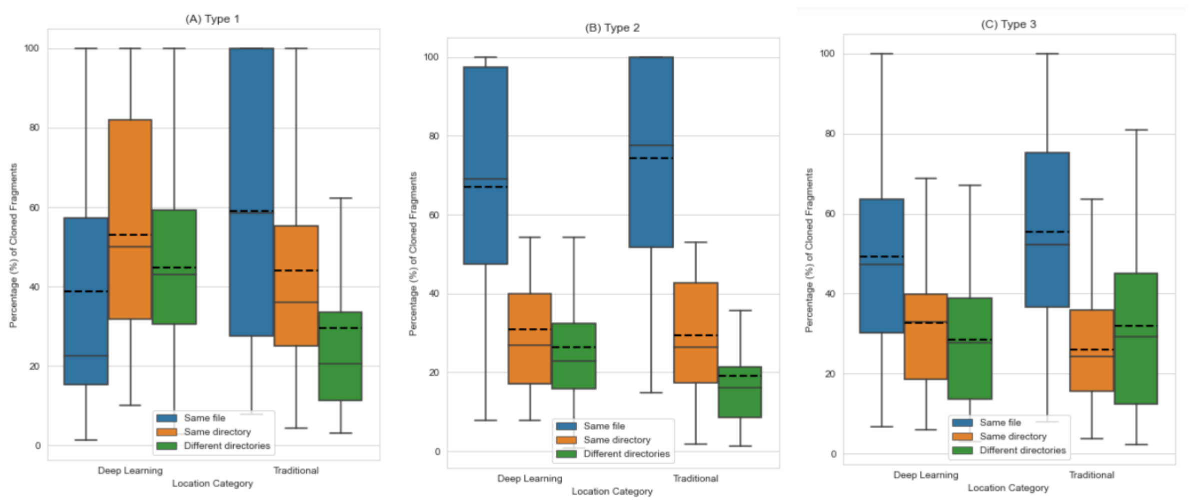

Figure 5.8 Distribution of percentage of number of fragments of code clones classes per clone location.

non-DL code, Type 3 clones are also located in the ‘same file’ in high numbers. For traditional code, we found a higher percentage of lines of code clones located in the ‘same file’ compared to the ‘same directory’ and ‘different directories’ categories. However, the difference is not statistically significant between the ‘same directory’ and ‘different directories’ categories, in traditional code as shown in Table 5.4.

From this analysis of the distribution of clones across the different files and repositories of the studied projects, we can conclude that clones in deep learning code is more dispersed compared to clones in traditional code. Type 1 and Type 3 clones have relatively higher trends of being dispersed while type 2 clones tend to be more localized (in the same file). However, the percentages of lines of code may not always fully reflect their relative impacts on the systems. For example, a higher number of small sized cloned fragments scattered in distant locations may pose higher challenges in change propagation than a small number of larger cloned fragments located not too far apart. Therefore, we further analyze the distribution of the number of cloned fragments in different location categories.

Location-based distribution of clone fragments : Figure 5.7 shows the percentages of number of fragments ((number of clone fragments in a location category/total number of clone fragments)x100) for each location category. We observe that the proportion of clone fragments tend to be higher in the ‘same file’ location category. We found a statistically significant difference between the distribution of the proportion of clone fragments located in the same file and in both ‘same directory’ and ‘different directories’ with p-values equal to 4.46e-05 and 1.94e-07 respectively with values of effect size of 0.44 (small) and 0.57 (medium), respectively in deep learning code.

As shown in Fig. 5.7, the mean value of the percentages of code fragments that belong to the same directory category in deep learning code is 32.04% with a standard deviation (STD) of 18.48. The number of fragments influences the degree to which the identified code clones from the same directory are difficult to maintain, the higher the number of fragments, the more troublesome their maintenance is likely to be. Hence, DL code may become more problematic with the spread of many duplicated code fragments that tend to be exact (Fig. 5.8-A) and in different files, but in the same directory. We observe that the distributions of the percentages of clone fragments of different clone types in different locations (Fig. 5.8) is similar to the distributions of the percentages of lines of cloned code (Fig. 5.6).

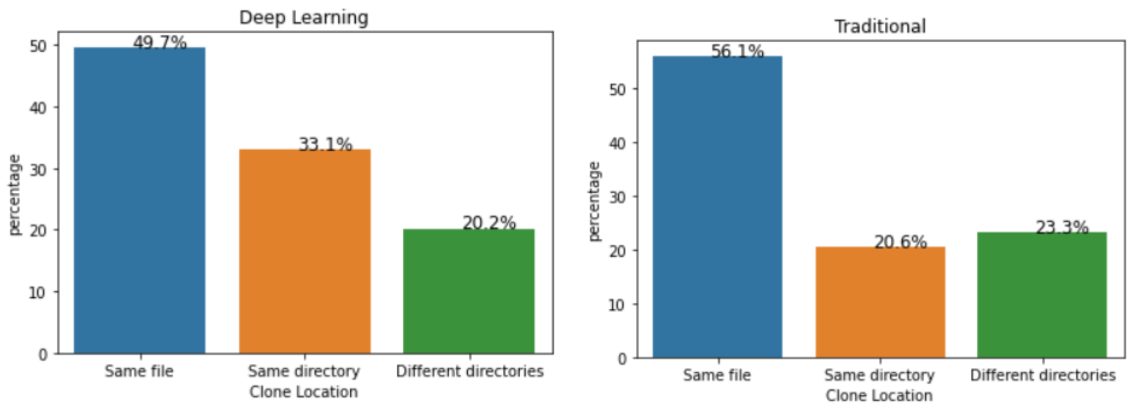

Figure 5.9 Percentages of average number of fragments of Code Clones by Location of Clones in both Deep Learning and Traditional Systems

For further insights, we analyze the average percentages of the number of clones fragments in each clone class by their locations in deep learning and traditional code. As shown in Figure 5.9, we identify that an average distribution of the number of fragments in each clone class is of 49.7% when clones are DL-related clones and are in the 'same file', 33.1% are in the 'same directory' and 20.2% are in 'different directories'. Hence, based on the distribution of clone fragments, DL clones are roughly equally distributed in same file (49.7%) and in different files and directories (33.1%+20.2%). However, fragments of clones in deep learning code are relatively more dispersed compared to fragments of clones in traditional system as traditional systems have higher percentage of clones fragments in 'same file' compared to deep learning code as shown in Fig. 5.9. This dispersion in the deep learning code may have negative consequences on maintenance.

Qualitative analysis of the clones in different locations category : We manually examined the clones contains in the different locations categories and observed that :

Same file : DL practitioners often duplicate functions in the same file when configuring different models. We found cloned functions in the same file with names representing the name of the model with slight modification in initializing (hyper)parameters of each model as shown in Table 5.5. This type of code clones located in close proximity may be relatively less problematic. This is mainly due to the ease of navigation between code clones during maintenance as their locations are not too distant from each other. Consequently, they may be less prone to inconsistent updates, which is a key reason behind the introduction of faults in cloned code. Duplicating code in close proximity is widely used to simplify the conception of the system [38] by renaming functions to facilitate code reuse and to make the cloned functions' name more related to its purpose, which improves program comprehension. These type of cloned functions with structural similarity but with identifier naming and data type differences are Type 2 clones. The trends of such clones to be in closer proximity is also reflected in our results from Fig. 5.6-B.

Table 5.5 Clone codes example where the location is in the same file (the differences are highlighted in gray)

```
def mobile_imagenet_config():
  return tf.contrib.training.HParams(
      stem_multiplier=1.0,
      dense_dropout_keep_prob=0.5,
      num_cells=12,
      filter_scaling_rate=2.0,
      drop_path_keep_prob=1.0,
      num_conv_filters=44,
      use_aux_head=1,
      num_reduction_layers=2,
      data_format='NHWC',
      skip_reduction_layer_input=0,
      total_training_steps=250000,
      use_bounded_activation=False,
  )
```

```
def large_imagenet_config():
  return tf.contrib.training.HParams(
      stem_multiplier=3.0,
      dense_dropout_keep_prob=0.5,
      num_cells=18,
      filter_scaling_rate=2.0,
      num_conv_filters=168,
      drop_path_keep_prob=0.7,
      use_aux_head=1,
      num_reduction_layers=2,
      data_format='NHWC',
      skip_reduction_layer_input=1,
      total_training_steps=250000,
      use_bounded_activation=False,
  )
```

Table 5.5 represents an example of a cloned code where their location is in the same file. We manually analyzed 40 code clones classes that are in the same file from the deep learning code, we found 27 of them to be similar functions with a common purpose. These functions were cloned to perform the same or closely similar tasks (to build the model with some modifications). The modifications were achieved by renaming functions (giving them names that are more meaningful and relevant to the task context) and parameterizing the code with different values that are specific to each model. As shown in Table 5.5, we have two functions that parameterize two different models. This is performed by calling a library routine capable of setting the hyperparameters of the model. It is configured as key-value pairs to build the model. Each function is renamed to be relevant to the model and we see slight differences between the values of each hyperparameter. An important number of this type of duplication is cloning similar functions with different names and parameter types, leading to Type 2 clones. This type of code duplication may explain the high percentage of Type 2 clones that are found to reside in the same file (Fig. 5.6-B). Our findings also confirms the results of previous work [38].

Same directory : Regarding the second category where clones exist in different files but in the same directory, it is common to find duplicated functions without or with minor changes [38]. In our case and as shown in Figure 5.6-A, for deep learning code, Type 1 clones are the type of clones that are frequently located in the same directory but in different files. From 85 code clones classes examined manually, we found in 49 code clones classes (57%) a file named utility containing useful functions needed to perform the construction of deep learning models. This suggests that DL developers either refactor their code without deleting the old functions, or that different developers working on the same project are not conscious about the existence of such files.

Different directories : Regarding clones located in different directories, we manually analyzed 102 code clones fragments that were located in different directories and found that 63% of them are not related to deep learning code. We often detect this type of duplication when it comes to verifying libraries' versions to choose the right routine call, deallocate memory, or getting model metadata (logging).

Summary of findings (RQ2) : According to our results, code clones in deep learning code are more dispersed than in traditional code. Regarding clone types, Type 1 clones are more dispersed in DL systems while Type 2 clones tend to be localized in the same file. Type 3 clones are spread in different locations but with a high percentage of lines of code clones residing in the same file. Regarding the distribution of the number of clone fragments, clones in deep learning code tend be more dispersed compared to clones in traditional software systems.

5.3.3 RQ3 : Do cloned and non-cloned code suffer similarly from bug-proneness in deep learning projects ?

Since deep learning systems are relatively fast to develop and deploy, code quality is often overlooked and it is frequently the case that the code is re-used and rarely refactored [49]. Several previous studies in traditional code highlighted the negative impacts of code clones on the maintenance and comprehension of code. Barbour et al. [88] found clones to be related to a high risk of bugs. Given the complexity of deep learning systems, it is likely that clones can have a similar averse effect on maintenance and bug-proneness. Hence, in this work, we analyze the bug-proneness of clones in deep learning code from two perspectives : (1) we examine correlations between clones occurrences in deep learning code and bugs occurrences, and (2) examine whether clones affect the time required to fix bugs in deep learning systems. We perform these investigations first on all clones and then for individual clone types (Type 1, Type 2, Type 3). We analyze all the commit history to identify all buggy commits (details are presented in Section 5.2.2.6). In order to determine the co-existence between bugs and code clones in deep learning code, we match code changes in bug-fixing commits with code clones by finding the intersection between the lines changed to fix bugs and the cloned lines of code.

We consider that a bug fix commit is related to code clones when the buggy lines belong to any duplicated code, otherwise it is considered as related to non-cloned code. Then we calculate the percentage of bug-fix commits related to cloned and non-cloned code for each system. Finally, we compute the average percentage of bug-fix commits related to cloned and non-cloned code, to comparatively evaluate their bug-proneness in the context of deep learning code.

Our results show that **75.85%** of bug-fix commits in DL systems are related to clones, i.e., in other words, more than three-quarters of the bugs in deep learning code are related to clones. We perform MWW tests for the distributions of code clones and non-clone code in the bug-fix commits. We found a statistically significant difference between the distribution of the number of commits that fix bugs on cloned lines and the distribution of bug-fix commits on non-cloned code, with a p-value equals to 0.026 and an effect size of 0.55 (medium). Thus, the bug-proneness of cloned code in deep learning code is higher compared to that of non-clone code.

Now, we further investigate the bug-proneness of different types of clones in DL code to gain deeper insights on the types of clones that are likely to be more risky (in terms of bug occurrence). This information would help deep learning developers to carefully prioritize clones for refactoring and tracking. Figure 5.10 shows the percentages of clones from different types (Type 1, Type 2, and Type 3) that are related to bugs. We find that Type 3 clones are the most likely to be related to bugs as **74.77%** of clone related bugs are Type 3 clones. In second position we have Type 2 clones with a percentage of **19.48%**, and finally Type 1 clones with a percentage of **5.75%**. These results obtained on deep learning code are similar to the findings of previous works comparing the bug-proneness of Type 1, Type 2, and Type 3 clones in traditional software systems [98].

Since Type 3 clones are higher in density (Fig. 5.3) and prevalent according to the distribution of the percentages of lines of code in different types of clones, they are possibly being associated with higher percentages of bugs too. Thus, we investigate further their bug-proneness

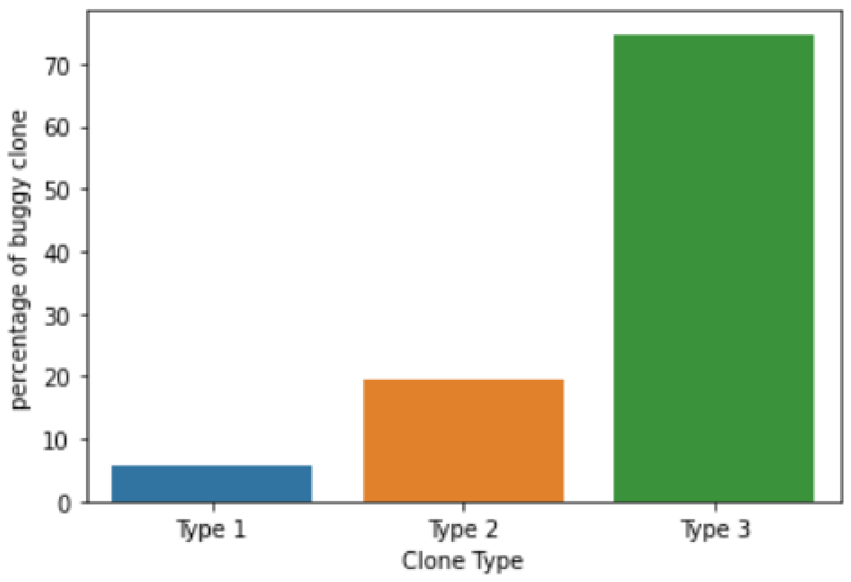

Figure 5.10 Buggy code clones occurrences by clone type

by studying the percentage of clone fragments in each clone types (Type 1, Type 2, Type 3) that are related to bugs. As shown in Figure 5.11, we find that 1.71% of clone fragments are buggy in Type 1 clones, 2.26% of clone fragments are buggy in Type 2 clones and 2.11% of clone fragments are buggy in Type 3 clones. This shows that a higher percentage of Type 2 clone fragments are related to bugs, followed by Type 3 clones and then Type 1 clones. However, there are more Type 3 clones in the deep learning code (Fig. 5.3) compared to Type 1 and Type 2 clones. This explains the observation that Type 3 clones contains the highest fractions of clone related bugs (Fig. 5.10) despite the percentages of buggy clone fragments in Type 3 not being higher than that of Type 2 clones.

Now, to study how clones in deep learning code affect the time to fix bugs in deep learning code. We investigate whether the time required to fix bugs when the bug is related to code clones is more expensive than when it is not related to clones. Thus, we calculated the time between bug-fixing commit and the corresponding bug-inducing commit. Then, we compare the average time spent to fix bugs when bugs are related to clones (bug-fix lines intersect with code clones) and when the bug is not related to a clone.

Figure 5.12 shows the distribution of the average times to fix bugs when the bug is located in a clone and when it is not. Comparing the median between the two distributions of time, we can see on Figure 5.12 that the time required to fix bugs when there is cloned code is slightly higher than the time required to fix a bug when the bug is not related to a code clone. The mean value of the average time to fix bugs in clones in DL code is 1.16e+07

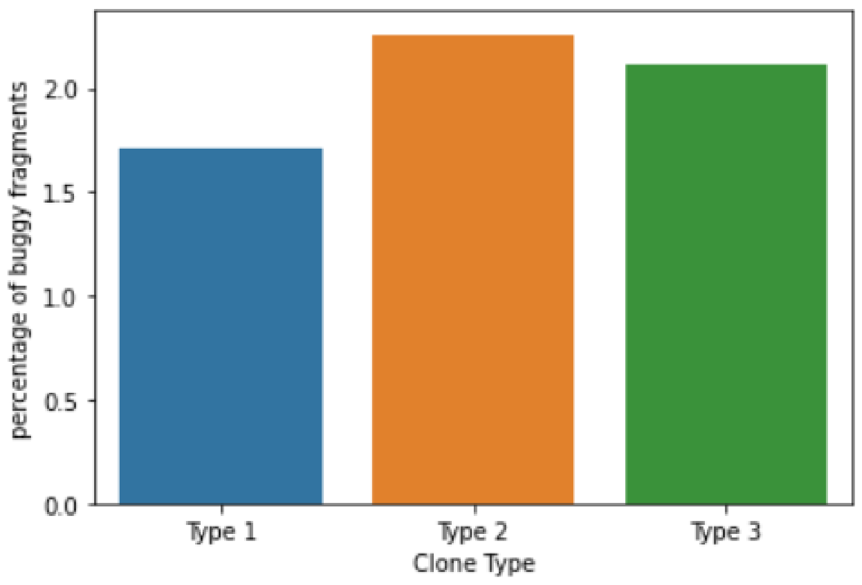

Figure 5.11 Percentages of buggy code fragments by clone type

seconds with a STD of 8.29e+06, and when it comes to non cloned code, the mean time to fix a bug is 1.06e+07 seconds with a STD of 8.35e+06. Buggy cloned code seems to be taking comparatively higher time to get fixed in deep learning code. We perform a Man-Whitney test comparing the distribution of times. However, we found no statistically significant difference between the time to fix bugs in cloned and non-cloned code (p-value =0.34, effect size 0.2).

We further check what percentages of DL systems have higher bug-fix time for clones and what percentages of systems have the opposite. Among 6 DL repositories, we observe in four (4) DL systems that the bug-fix time for clones is higher and for the rest (i.e., 2 systems), the bug-fix time for clones is lower. Overall, we can conclude that in a majority of cases (66.66%), bugs related to clones take a longer time to get fixed, suggesting that bugs occurring in cloned code may be more challenging to fix.

The observation that bugs in cloned code take comparatively longer time to fix means that the existence of code clones in deep learning code may hinder the maintenance of this type of system.

Summary of findings (RQ3) : We find that cloned code may be more bug-prone than non-cloned code in deep learning systems. In addition, Type 3 clones have a relatively higher odd to be involved in bugs in the deep learning code, followed by Type 2 and Type 1 clones respectively. Also, bugs related to clones in DL code tend to take more time to get fixed compared to other bugs.

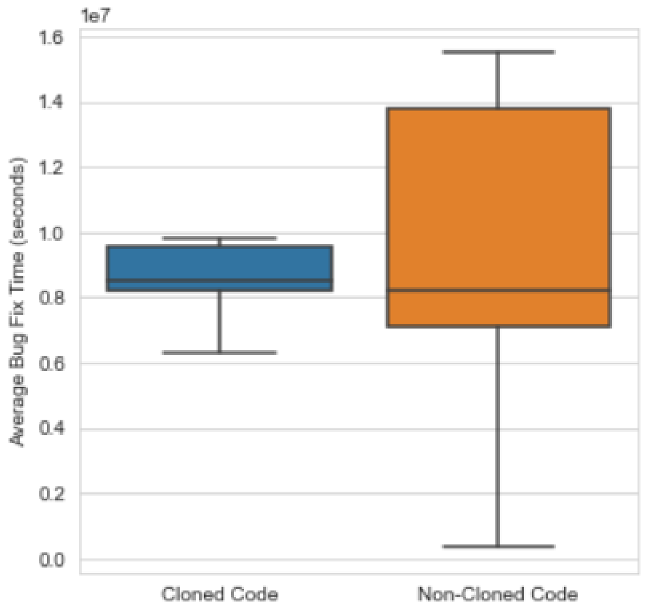

Figure 5.12 Comparative bug-fix times for cloned and non-cloned code in DL systems.

5.3.4 RQ4 : Why do deep learning developers clone code ?

In this research question, we examine the reasons behind the practice of code cloning in deep learning systems. We manually analyzed the detected code clones for a selected subset of six deep learning projects. We labeled each detected code clone class by the functionality it serves (task). Then, we assign each labeled clone class to its corresponding DL phase making sure that the relation between the labeled task and the DL phase is 'to perform' (more details in section 5.2.2.5).

Table 5.6 shows the taxonomy of code clones that resulted from our manual analysis of the selected six DL projects. We show only the related DL phases that co-occur with the detected code clones. Therefore, we have neither all the DL phases nor all of its subcategories presented in Table 5.6. Also, the DL phase subcategories are not exclusive, since functions may be used in tasks related to different phases.

Table 5.6 Percentages of Occurrence of Code Clones in DL Phases

dl_phase category	**dl_phase subcategory**	**Type 1 %occs**	**Type 2 % occs**	**Type 3 % occs**	**% occs in sub category**	**% occs in total**
Preliminary preparation	hardware requirements	100	0	0	100.0	1.03
Data collection	load data	20	20	40	80.0	5.15
	load label	20	0	0	20.0	
Data postprocessing	compute output shape	0	0	12.5	12.5	8.25
	object localization	25	0	12.5	37.5	
	process output	25	0	12.5	37.5	
	set shape of output data	12.5	0	0	12.5	
Data preprocessing	apply data augmentation	5.55	0	0	5.55	18.56
	data normalization	0	0	11.11	11.11	
	get batches of data	0	5.55	0	5.55	
	get numerical feature columns	5.55	0	5.55	11.11	
	parse arguments	0	0	5.55	5.55	
	prepare tensor	11.11	0	0	11.11	
	process input	0	0	16.66	16.66	
	resize image	5.55	0	0	5.55	
	set shape of input data	0	0	11.11	11.11	
	set type of input data	0	0	5.55	5.55	
	setting format input data	0	0	5.55	5.55	
	split data	0	0	5.55	5.55	
Model prediction	inference	100	0	0	100.0	2.06
Model construction	model component format verif.	2.86	0	0	2.86	36.08
	activation function call	0	0	2..86	2.86	
	build model	2.86	0	0	2.86	
	build one subnetwork	0	0	2.86	2.86	
	compute model outputs	0	0	2.86	2.86	
	init evaluation metrics	0	0	2.86	2.86	
	initialize model graph	0	0	2.86	2.86	
	initialize model output	2.86	0	0	2.86	
	layer construction	0	2.86	2.86	5.71	
	model architecture instantiation	0	0	5.71	5.71	
	model (hyper)parameters init	14.29	20	28.75	62.86	
Model evaluation	performance metric computation	0	22.22	66.66	88.89	9.28
	test data prediction	0	0	11.11	11.11	
Model training	compute loss	27.77	0	0	27.78	18.56
	get pooling info	0	0	5.56	5.56	
	measure model accuracy	5.56	0	0	5.56	
	model training	5.56	0	11.11	16.67	
	one model step training	11.11	5.56	11.11	27.78	
	training procedure	0	0	5.56	5.56	
	weight normalization	0	0	11.11	11.11	
Model tuning	Minibatch size	0	0	100	100	1.03

In this research question, we study clones at the granularity of the function. A function can implement one or more tasks that are involved in a DL phase. In the following, we discuss the characteristics of clones in deep learning systems that are associated with different functionalities and development phases of deep learning systems. The following phases are listed based on the percentage of occurrences with clones from highest to lowest.

Model construction : Our manual analysis shows that the most frequent DL-phase category that co-exists with code clones is the model construction phase with 36.08% of DL-related code clones. This is an indication that DL practitioners duplicate code frequently when building the model and specifically when initializing hyperparameters/parameters with 62.86% of code clones classes being related to the DL-phase subcategory of the model construction. Table 5.6 shows that the majority of clones created by developers when initializing (hyper)parameters during the construction of the model are Type 3 clones (they represent 28.75% overall). The *data preprocessing* and *model training* phases contained 18.56% of all the clones that we manually analyzed.

Model training : Computing loss and training in each step of the model are the most frequent activities performed during model training. These activities are associated with 27.78% of clones from the model training subcategory. According to our manual analysis, computing loss functions are often copied/pasted from other functions located in the same location as the model implementation, or written from scratch. i.e., by calling DL libraries routines to perform loss computation. Some developers may also reuse the corresponding code from the online sources. The duplication of code for loss function is illustrated in the example in the table 5.7. The first line in the table corresponds to calculating the loss of RankingLoss. The second line corresponds to computing the loss of Softmax. The two functions are Type 3 clones to each other. Ranking and Softmax are two types of loss functions in deep learning. In fact, the loss computation is often common between deep learning models. Even when they are different, some of them have similar implementation logic. Hence, the prevalence of duplicated code that computes loss functions.

Data preprocessing Processing input is related to 16.66% of clones associated with the 'data preprocessing' phase of the DL development workflow. Process input includes all the transformation needed on the input data to perform model training (e.g., process input of model inception v3 by normalizing each pixel of image input).

Model evaluation : Each DL model is evaluated to improve its performance. Overall, 9.28% of the DL-related cloned code corresponds to model evaluation of which 89% of clones are related to performance metric computation and 11.11% to test data prediction. Measurement metrics used to evaluate the models tend to be duplicated frequently for each model and for each metric. One example of cloning metric computation code can be seen in Table 5.8 [1], where we have the implementation of two measures : Mean Reciprocal Rank and Mean Average Precision. The two functions are clones of each other. The clone is of Type 3. The differences are in the renaming and function calls that corresponds to each metric computation.

1. `https://github.com/tensorflow/ranking`

Table 5.7 Example of Model Training (Compute Loss) Type 3 Clone

```
    def compute(self, labels, logits, weights, reduction):
    """Computes the reduced loss for tf.estimator (not tf.keras).

    Note that this function is not compatible with keras.

    Args:
      labels: A `Tensor` of the same shape as `logits` representing graded
        relevance.
      logits: A `Tensor` with shape [batch_size, list_size]. Each value is the
        ranking score of the corresponding item.
      weights: A scalar, a `Tensor` with shape [batch_size, 1] for list-wise
        weights, or a `Tensor` with shape [batch_size, list_size] for item-wise
        weights.
      reduction: One of `tf.losses.Reduction` except `NONE`. Describes how to
        reduce training loss over batch.

    Returns:
      Reduced loss for training and eval.
    """
    losses, loss_weights = self.compute_unreduced_loss(labels, logits)
    weights = tf.multiply(self.normalize_weights(labels, weights), loss_weights)
    return tf.compat.v1.losses.compute_weighted_loss(
        losses, weights, reduction=reduction)
```

```
def compute(self, labels, logits, weights, reduction):
    """See `_RankingLoss`."""
    labels, logits = self.precompute(labels, logits, weights)
    losses, weights = self.compute_unreduced_loss(labels, logits)
    return tf.compat.v1.losses.compute_weighted_loss(
        losses, weights, reduction=reduction)
```

Data post-processing : Data is often post-processed after an inductive process and converted into a format recommended by the stakeholders of the model. Our findings show that 8.25% of cloned code are related to DL functions used in the data post-processing phase. There are various techniques to perform this phase. We found that object localization functions are duplicated functions with 37.5% from the total cloned functions to perform data post-processing phase. Object localization is used to interpret output assigning each object to a class with a higher probability or by drawing bounding boxes on an image from inference results. 37.5% of cloned functions are classified as process output. And the rest are found duplicated for computing output shape and set shape of output data.

Data collection : Data collection operations are frequently cloned. 5.15% of our manually analyzed clones were related to data collection. Among them, we found 80% of clones to be related to loading data either from files or from an URL or using a library to get data. The rest are derived from load label of classes of data functions (20%).

Model prediction : 2.06% of our manual analysis code clones are related to inference. All of them are Type 1 clones. Subsequently, according to our manual analysis, we can say that DL developers often duplicate the same code to create an inference process from a trained model.

Table 5.8 Example of Model Evaluation (Compute Metrics) Type 3 Clone

```
def mean_reciprocal_rank(labels,
                         predictions,
                         weights=None,
                         topn=None,
                         name=None):
  """Computes mean reciprocal rank (MRR).
  Args:
    labels: A `Tensor` of the same shape as `predictions`. A value >= 1 means a
      relevant example.
    predictions: A `Tensor` with shape [batch_size, list_size]. Each value is
      the ranking score of the corresponding example.
    weights: A `Tensor` of the same shape of predictions or [batch_size, 1]. The
      former case is per-example and the latter case is per-list.
    topn: An integer cutoff specifying how many examples to consider for this
      metric. If None, the whole list is considered.
    name: A string used as the name for this metric.

  Returns:
    A metric for the weighted mean reciprocal rank of the batch.
  """
  metric = metrics_impl.MRRMetric(name, topn)
  with tf.compat.v1.name_scope(metric.name, 'mean_reciprocal_rank',
                               (labels, predictions, weights)):
    mrr, per_list_weights = metric.compute(labels, predictions, weights)
    return tf.compat.v1.metrics.mean(mrr, per_list_weights)
```

```
def mean_average_precision(labels,
                           predictions,
                           weights=None,
                           topn=None,
                           name=None):
  """Computes mean average precision (MAP).
  Args:
    labels: A `Tensor` of the same shape as `predictions`. A value >= 1 means a
      relevant example.
    predictions: A `Tensor` with shape [batch_size, list_size]. Each value is
      the ranking score of the corresponding example.
    weights: A `Tensor` of the same shape of predictions or [batch_size, 1]. The
      former case is per-example and the latter case is per-list.
    topn: A cutoff for how many examples to consider for this metric.
    name: A string used as the name for this metric.

  Returns:
    A metric for the mean average precision.
  """
  metric = metrics_impl.MeanAveragePrecisionMetric(name, topn)
  with tf.compat.v1.name_scope(metric.name, 'mean_average_precision',
                               (labels, predictions, weights)):
    per_list_map, per_list_weights = metric.compute(labels, predictions,
                                                    weights)
  return tf.compat.v1.metrics.mean(per_list_map, per_list_weights)
```

Model tuning : We found clones in the code used to find the best batch size to train the model. Hyperparameter tuning operations are also often cloned. All the clones found to be related to this category were type 3 clones. Meaning that DL developers often duplicate the hyperparameter tuning code of other models and apply some modifications to it (adding few extra lines), for example to adjust the batch size. Clones in hyperparameter tuning code represents 1.03% of the analyzed clones.

Preliminary preparation : The code used to prepare the environment for model training appears to also contain clones. To optimise the model training time, developers write code to manage the hardware, e.g., CPU and GPU management. Among the manually analyzed clones, 1.03% of them belong to the environment configuration category. All these clones were Type 1 clones, suggesting that developers often duplicate these configuration codes without modifications.

Our analysis show that code duplication is a common practice among DL developers. They duplicate code during almost all the phases of the development process of deep learning models, in addition to duplicating traditional methods like test and logging. Since duplicating code may lead to bug propagation and inconsistency in the program. We recommend that DL developers pay a close attention to these clones during the maintenance and evolution of their systems.

Summary of findings (RQ4) : According to our findings, code duplication is more prevalent during the model construction phase of deep learning systems. Code related to the initialization of model hyperparameters are the most cloned, followed by code related to model training and data preprocessing.

5.3.5 RQ5 : Where in the deep learning code is cloning the most risky ?

After applying our taxonomy to the selected code clones from the analyzed subset of six systems, it is of interest to identify deep learning activities during which cloning is the most risky. By risky here we refer to the risk of bug introduction. Our results of RQ3 show that code cloning can lead to bug. A better understanding of the circumstances in which bugs frequently occur on cloned code will help raise the awareness of DL developers about the potential risks of their cloning actions.

To carry out this investigation, we consider the relation between bugs and code clones and determine which part of the DL code is more prone to bugs when it is duplicated. We identify labeled code clones lines (the labels are from RQ4) that intersect with lines that fix bugs in the system. We computed the percentages of bug related cloned functions for each DL phase. Our result shows that code cloned during the model construction phase are related to bugs in higher numbers ; 50% of them are related to bugs as shown in Figure 5.13.

Table 5.10 shows which DL-related cloned functions (tasks) are the most involved with bugs. The corresponding DL phases are also provided in the Table. We display only DL-related tasks and phases where clones are involved in bugs (i.e., other phases of the DL workflow were the phenomenon is not observed are omitted). As mentioned earlier, the model construction phase contains the highest proportion of buggy clones (i.e., 50% of all buggy clones in our

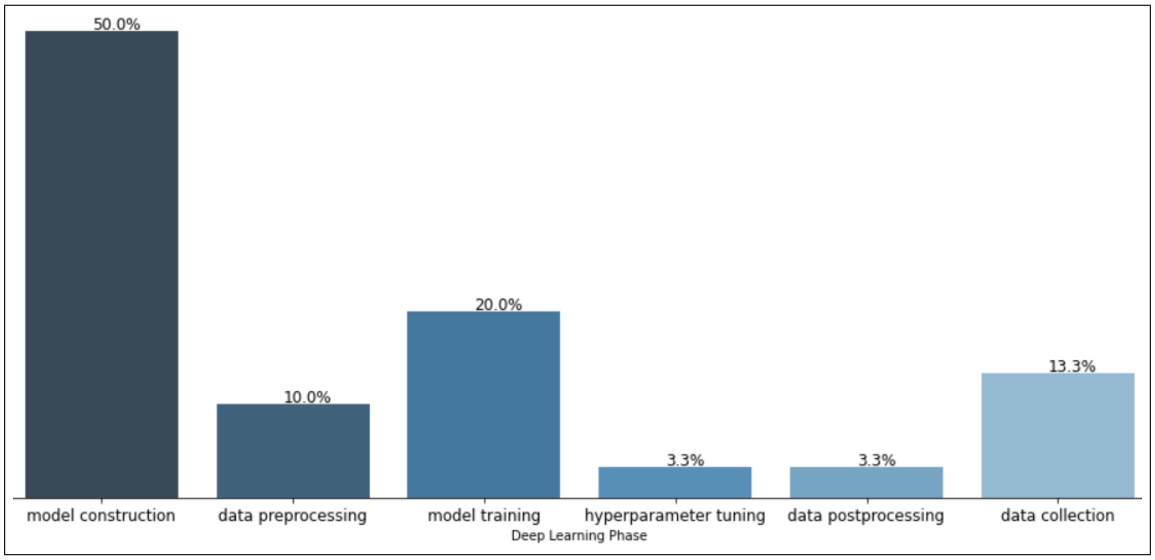

Figure 5.13 Percentage of Bug-fix occurred with cloned functions with respect to deep learning phases

manually analyzed clone data). The majority of them are related to model (hyper)parameters initialization (46.66%). Table 5.9 shows an example of bug fix in a buggy clone. The presented cloned code fragments are from a bug-fix commit with the message 'Minor optimizer consistency fixes'[2]. The optimizers in deep learning are capable of reducing the losses by changing the attributes of the neural network (i.e., learning rate). Those optimizers have a common implementation of the hyperparameter initialization function. In the example of commit bug-fix from Table 5.9, to fix the instantiation of the number of iteration by adding a data type, the deep learning developer had to propagate the same change to several optimizer initializations. In this example, we have seven optimizers, i.e, SGD, RMSProp, Adagrad, Adadelta, Adam, Adamax, and Nadam that share the same initialization implementation and the developer needed to propagate the fixing change seven times.

The DL phase with the second highest proportion of buggy clones is 'Model training' with a proportion of 20%. The phase with the third highest proportion of buggy clones is the data collection phase with 13.3%. 10% of buggy clones are related to data pre-processing; the majority of them are in code related to tensor operations (66.66%) and code for setting the shape of input data. Only a small amount of the analyzed buggy clones were found to be related to data post-processing (3.3%) and tuning of the hyperparameters of the model (3.3%). In light of these results, we recommend that DL developers pay particular attention when duplicating code during the model construction phase. Although it may seems like a

2. `https://github.com/keras-team/keras/commit/2d8739d`

Table 5.9 Example of Bug Fix Commit Code Change

```
def __init__(self, lr=0.01, epsilon=None, decay=0., **kwargs):
        super(Adagrad, self).__init__(**kwargs)
        with K.name_scope(self.__class__.__name__):
            self.lr = K.variable(lr, name='lr')
            self.decay = K.variable(decay, name='decay')
            self.iterations = K.variable(0, name='iterations')
        if epsilon is None:
            epsilon = K.epsilon()
        self.epsilon = epsilon
        self.initial_decay = decay
```

```
def __init__(self, lr=0.01, epsilon=None, decay=0., **kwargs):
        super(Adagrad, self).__init__(**kwargs)
        with K.name_scope(self.__class__.__name__):
            self.lr = K.variable(lr, name='lr')
            self.decay = K.variable(decay, name='decay')
            self.iterations = K.variable(0, dtype='int64', name='iterations')
        if epsilon is None:
            epsilon = K.epsilon()
        self.epsilon = epsilon
        self.initial_decay = decay
```

good idea to copy the code of an existing model, to speed up the model construction phase, there are perils to this practice.

Table 5.10 Percentage of DL-related Cloned Functions with Bugs

Taxonomy	**Task**	**% occs in DL step**	**% occs from total**
Data collection	Load data	100	13.3
Data post-processing	Set shape of output data	100	3.3
Data pre-processing	Prepare tensor	66.66	10
	Set shape of input data	33.33	
Hyperparameter Tuning	Hyperparameter tuning	100	3.3
Model Construction	Model component format verification	6.66	50
	Initialize model graph	6.66	
	Initialize model output	6.66	
	Layer construction	13.33	
	Model architecture instantiation	20	
	Model (hyper)parameters initialization	46.66	
Model training	Model training	33.33	20
	One model step training	66.66	

Summary of findings (RQ5) : Code clones that are related to model construction are the most bug-prone among deep learning clones followed by the clones related to model training and data collection.

5.4 Research Implications

In this section, we discuss the implications of our findings with regard to cloning activity in the DL code.

Cloning is frequent in deep learning code : In light of the higher density of code-clones identified in deep learning code, it appears that DL developers prefer to reuse existing solutions instead of creating new ones from scratch. As they need to experiment with different configurations to find the best DL model, duplicating code that works often seem like a good idea to save time and effort. Our results show that developers often copy-paste exact code (frequently for loss computation) in the same location as the calling statement. We assume that this proximity aims to ease the maintenance of the resulting code. However, maintaining multiple clone copies always increase the risk of failing to propagate changes consistently ; leading to bugs. Our results show that clones are more prevalent in deep learning code in comparison to traditional code. We attribute this phenomenon to the fact that a same decision logic can be used several times in a deep learning model and also across models. For example, when creating a convolutional neural network [99, 100] model that consists of a set of layers. Each layer is initialized according to its type and its parameter values needed, and blocks of codes are stacked to create the architecture. These blocks are exact or similar copies of each other. Hence, the prevalence of code clones in the code of this model.

Deep learning code clones are dispersed : Considering the code clones being distant as found in deep learning code, such dispersion of code clones is problematic [101] from a maintenance point of view. When changing code, it is most likely easier to change the code in the same file than in different files or folders. Fragments of related code in distant locations may add navigation and comprehension overhead during code change. Thus, the maintenance will be difficult to handle. Due to the high percentages of distant code clones (same directory and different directories), deep learning practitioners should be aware of the potential negative impacts of such cloning practices.

In addition, our analysis of the percentages of clone fragments in different location categories show that clones in deep learning code are more dispersed in the code, which is also problematic. Because of the negative impact of Code clones on maintenance, developers should consider refactoring them. We noticed some signs of refactoring in some of the studied deep learning systems. Specifically, we observed the use of files with a name ending with '_utility' that contains all the useful functions and functions likely to be used in different parts of the system.

Code clones in DL code are related to bugs : Our results show that cloned code may be more bug-prone than non-cloned code in deep learning systems. In addition, Type 3 clones have a relatively higher odd to be involved in bugs in the deep learning code than Type 2 and Type1 clones. Also, code clones that are related to model construction phase are the most bug-prone. In particular those related to model (hyper)parameters initialization. Since the main challenge of DL developers is to provide a model with high accuracy, setting model (hyper)parameters is an important step to implement an efficient model. Therefore bugs occurring in the code responsible for this critical task is likely to have a severe impact on the quality of the deep learning system.

Due to the data-driven nature of deep learning, data collection is a crucial task [102] and any bug occurring in the code responsible for this phase is also likely to significantly impact the quality of the deep learning system.

Our findings regarding the prevalence and distribution of clones in deep learning code, their bug-proneness and insights on the characteristics and impacts of the clones related to different DL phases are thus important for deep learning practitioners. These findings can help researchers to further investigate the characteristics and evolution of clones in deep learning code and also guide practitioners to adapt the best software development practices to the maintenance and evolution of the deep learning systems.

5.5 Threats to validity

In this section, we discuss the potential threats to the validity of our research findings.

In terms of **Internal validity**, we manually labeled each detected code clones class to its corresponding DL phase. Then, we also manually assigned them to one of the steps of the DL code process. However, this relies on the subjective judgment of the persons who performed the manual classification. This is a threat to the internal validity of our experiment. To mitigate this threat, the manual classification for creating the taxonomy was done by multiple authors having academic and industry background. The results were then cross-validated, and disagreements were resolved by group discussion. We believe that this validation process decreased the chances of incorrect tagging. However, future research may further improve our approach and provide additional perspectives about our results by surveying deep learning practitioners.

In terms of **construct validity** threats, which concern the relationship between theory and observation. We followed the approach proposed by Rosen et al. [83] to detect bug-fix commits by employing a set of keywords that are bug fixing related. If the commit message contains one of the keywords, it will be labeled as a bug-fix commit. To reduce the imprecision in the bug-detection process, we reviewed a sample of the labeled commits and confirmed that they corresponded to bug-fixes.

For detecting clones, we used the NiCad clone detector [91]. Since different settings can have different effects, that we call a confounding configuration problem [103], we have carefully set the parameters of NiCad by employing a standard configuration [104] and with these settings, NiCad is reported to be very accurate in clone detection [92, 104]. Thus, we believe that our findings on code clones in deep learning code have relevance significance.

With regard to **external validity**, we only cover deep learning repositories that are written in python. As it is the most used programming language in the machine learning field, we can assume that the small data used brings a lot of knowledge. In addition, we selected only 6 DL repositories to be manually analyzed for the creation of the taxonomy. Therefore, this may threaten the generalizability of our results. We believe that even a small number of deep learning repositories will provide a comprehensive overview of how and why deep learning practitioners had to duplicate code. We assume that, for each deep learning project, we can have different percentages of code clones occurrences alternating between the different phases of deep learning workflow. The fact that they exist may challenge the development of this type of system. Future studies should validate the generalizability of our findings with other DL systems that are written in other programming languages.

In terms of **threats to reliability**, we investigate in our study open-source deep learning and traditional projects that are available on GitHub. And we provide a replication package that contains needed data and scripts to replicate our study [105].

And with respect to **threats to conclusion validity**, we use non-parametric statistical tests to analyze the difference between distributions. Non-parametric tests are adequate because they make no assumption on the nature of the data distribution.

5.6 Chapter Summary

This chapter presents an empirical study of code clones in deep learning systems. Through quantitative and qualitative analyses, we have examined the characteristics, distribution, and impacts of clones in deep learning code. We have shown that code clones are prevalent and dispersed in deep learning systems (which may add navigation and comprehension overhead). In addition, our results show a higher association between code clones and bug occurrences. Furthermore, cloning code responsible for model hyperparameters initialisation appeared to be a very risky activity, since a large proportion of clones in this part of the code were found to be buggy. Although duplicating code may lead to short term productivity gains, deep learning practitioners should be aware of the perils of such practice.

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

With the increasing popularity of deep learning and its rapid development in recent years, it has become a game changer in many fields. Some of these fields requiring high safety critical standards. It is therefore necessary to ensure the good quality of deep learning systems. In this thesis, we explore the code quality and coding practices of deep learning practitioners. We examine the impact of code smells and clones on the quality of deep learning systems and formulate recommendations for both researchers and practitioners. In Chapter 3, we review the existing literature discussing the challenges and code quality assurance of AI/ML/DL systems, the impact of code smells and code clones on software quality. In Chapter 2, we provide an overview of the concepts and terminology that are useful to better follow our empirical studies. In Chapter 4, we perform a comparison of smells occurrences between traditional and deep learning applications. We analyze a total of 118 repositories (59 deep learning and 59 traditional). We make the following observations : First, we found no statistically significant difference in the code smell occurrences between deep learning and traditional software systems. Second, we found that the most frequent smell types occurring in deep learning code are : *Long Ternary Conditional Expression*, *Complex Container Comprehension*, and *Long Lambda Function*. We also observed an increasing trend of smell occurrences across the software releases of deep learning systems. And finally, our findings confirm that the presence of code smells may increase the chances of bugs occurrence in deep learning systems. In Chapter 5, we analyze clones in deep learning code. We use the same dataset used in Chapter 4 (59 deep learning open software systems and 59 traditional open source software systems). We show that cloning is frequent in deep learning code. We also found that deep learning systems tend to have higher density of clones. This prevalence of clones in deep learning code might be due to the complexity and the black-box nature of deep learning systems. In terms of location, we observed that code clones are dispersed in deep learning code compared to traditional systems. This dispersion may lead to higher maintenance costs since it may hinder code comprehension. Furthermore, we found that code clones in deep learning code are bug-prone even with higher trends of being associated with bugs compared to non-cloned code. We have also shown that some phases and activities (e.g., model construction) related to deep learning system development are comparatively more susceptible to buggy clones, warranting caution from deep learning developers when duplicating code. Although code duplication may lead to short term productivity gains, deep learning practitioners should be aware of the perils of such practice.

Ours is the first work that extensively investigates the code quality of 59 open-source, deep learning applications. It can help the researchers better understand the code quality and maintainability of the deep learning applications that are likely to grow in the coming years. We hope that our work will raise the awareness of DL developers on code quality issues and prompt them to adopt the best software engineering practices.

6.2 Limitations of the proposed approaches

— Our works rely on the results of two tools for detecting code smells and code clones. However, Pysmell has proved its ability in detecting code smells with 97.7% of average precision [106]. Furthermore, NiCad has shown its effectiveness in detecting code clones. We have also used a common configuration for NiCad making our results more accurate [92, 104].

— In our thesis, we analyze only 59 deep learning systems and that are written in Python. Despite the wide use of Python language in the AI field, our findings remain not quite generalizable.

6.3 Future work

There are many potential future research directions that stem from the contributions made in this thesis. These possible directions are outlined below :

— We have explored only 10 types of code smells that are python related as well as the code cloning practices in DL systems written in Python. Therefore, it is a new research challenge for the researchers to identify code smells specific to deep learning code and thus providing design patterns and practices to follow to the AI community.

— We may integrate more deep learning systems with different programming languages to further abstract DL source code so that only meaningful results and trends are recognized.

— The quantitative and qualitative analysis in the code clones study provide us with a picture of the characteristics, distribution and impacts of clones in deep learning code. However, we need further studies on the evolution patterns and the impacts of clones on different aspects of the quality of deep learning code, to guide the practitioners to better manage clones in deep learning systems. Thus, we plan our future research towards the investigation of the clone genealogy in deep learning code, to have deeper insights into how clones in deep learning code evolve, which in turn can help practitioners adopt safer code reuse practices, leverage existing libraries and open-source resources in the rapidly growing domain of deep learning and other machine learning based system development.

— As extensive future work, we want to focus on limitations of the approach used in our code clones study. The first is to address a different level of granularity as opposed to the function level. Currently, our approach only analyzes code clones with function granularity. We would like to investigate more in the deep learning code and change the granularity to the block level. This change in granularity would increase the context for certain practices of code duplication and should make the approach more meaningful and insightful to developers.

  Our second future work is similar to the first, but we want to focus on learning smaller duplicates called micro-clones. Since Python is the most used programming language in ML code and it is less verbose than other programming languages, we assume that micro-clones will be even more prevalent in the code of DL systems. We want to detect code clones between 1-4 lines, which would allow us to deeply understand the spread of code clones in DL code.

REFEFENCES

[1] Z. Chen, L. Chen, W. Ma, X. Zhou, Y. Zhou, and B. Xu, “Understanding metric-based detectable smells in python software : A comparative study,” *Information and Software Technology*, vol. 94, pp. 14–29, 2018.

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

[3] ——, *Deep Learning.* MIT Press, 2016, http://www.deeplearningbook.org.

[4] 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 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds. Curran Associates, Inc., 2015, pp. 2503–2511. [Online]. Available : http://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems.pdf

[5] J. Liu, Q. Huang, X. Xia, E. Shihab, D. Lo, and S. Li, “Is using deep learning frameworks free ? characterizing technical debt in deep learning frameworks,” in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering : Software Engineering in Society*, 2020, pp. 1–10.

[6] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, “Refactoring : Improving the design of existing code addison-wesley professional,” *Berkeley, CA, USA*, 1999.

[7] G. Suryanarayana, G. Samarthyam, and T. Sharma, *Refactoring for software design smells : managing technical debt.* Morgan Kaufmann, 2014.

[8] M. Gottschalk, M. Josefiok, J. Jelschen, and A. Winter, “Removing energy code smells with reengineering services,” *INFORMATIK 2012*, 2012.

[9] A. Vetro, L. Ardito, and M. Morisio, “Definition, implementation and validation of energy code smells : an exploratory study on an embedded system,” 2013.

[10] C. K. Roy and J. R. Cordy, “Near-miss function clones in open source software : an empirical study,” *Journal of Software Maintenance and Evolution : Research and Practice*, vol. 22, no. 3, pp. 165–189, 2010.

[11] C. J. Kapser and M. W. Godfrey, ““cloning considered harmful” considered harmful : patterns of cloning in software,” *Empirical Software Engineering*, vol. 13, no. 6, p. 645, 2008.

[12] K. Hotta, Y. Sano, Y. Higo, and S. Kusumoto, “Is duplicate code more frequently modified than non-duplicate code in software evolution ? an empirical study on open source software,” in *Proceedings of the Joint ERCIM Workshop on Software Evolution (EVOL) and International Workshop on Principles of Software Evolution (IWPSE)*, 2010, pp. 73–82.

[13] J. Krinke, “Is cloned code older than non-cloned code ?” in *Proceedings of the 5th International Workshop on Software Clones*, 2011, pp. 28–33.

[14] L. Barbour, F. Khomh, and Y. Zou, “Late propagation in software clones,” in *2011 27th IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 2011, pp. 273–282.

[15] ——, “An empirical study of faults in late propagation clone genealogies,” *Journal of Software : Evolution and Process*, vol. 25, no. 11, pp. 1139–1165, 2013.

[16] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, “Do code clones matter ?” in *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009, pp. 485–495.

[17] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, “CP-Miner : Finding copy-paste and related bugs in large-scale software code,” *IEEE TSE*, vol. 32, pp. 176–192, 2006.

[18] J. Li and M. D. Ernst, “Cbcd : Cloned buggy code detector,” in *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, 2012, pp. 310–320.

[19] M. S. Rahman and C. K. Roy, “On the relationships between stability and bug-proneness of code clones : An empirical study,” in *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. IEEE, 2017, pp. 131–140.

[20] S. Wagner, A. Abdulkhaleq, K. Kaya, and A. Paar, “On the relationship of inconsistent software clones and faults : an empirical study,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 79–89.

[21] C. K. Roy, M. F. Zibran, and R. Koschke, “The vision of software clone management : Past, present, and future (keynote paper),” in *proc. CSMR-WCRE*, 2014, pp. 18–33.

[22] J. Al Dallal and A. Abdin, “Empirical evaluation of the impact of object-oriented code refactoring on quality attributes : A systematic literature review,” *IEEE Transactions on Software Engineering*, vol. 44, no. 1, pp. 44–69, 2017.

[23] F. J. Buckley and R. Poston, “Software quality assurance,” *IEEE Transactions on Software Engineering*, no. 1, pp. 36–41, 1984.

[24] S. Hamdan and S. Alramouni, "A quality framework for software continuous integration," *Procedia Manufacturing*, vol. 3, pp. 2019–2025, 2015.

[25] D. Kumlander, "Towards a new paradigm of software development : an ambassador driven process in distributed software companies," in *Advanced Techniques in Computing Sciences and Software Engineering.* Springer, 2010, pp. 487–490.

[26] S. Rochimah, S. Arifiani, and V. F. Insanittaqwa, "Non-source code refactoring : a systematic literature review," *International Journal of Software Engineering and Its Applications*, vol. 9, no. 6, pp. 197–214, 2015.

[27] J. Han, E. Shihab, Z. Wan, S. Deng, and X. Xia, "What do programmers discuss about deep learning frameworks," *EMPIRICAL SOFTWARE ENGINEERING*, 2020.

[28] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann, "Software engineering for machine learning : A case study," in *2019 IEEE/ACM 41st International Conference on Software Engineering : Software Engineering in Practice (ICSE-SEIP).* IEEE, 2019, pp. 291–300.

[29] A. Koenzen, N. Ernst, and M.-A. Storey, "Code duplication and reuse in jupyter notebooks," *arXiv preprint arXiv :2005.13709*, 2020.

[30] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *The Journal of Machine Learning Research*, vol. 13, no. 1, pp. 281–305, 2012.

[31] M. Fowler, K. Beck, and W. R. Opdyke, "Refactoring : Improving the design of existing code," in *11th European Conference. Jyväskylä, Finland*, 1997.

[32] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1.* IEEE Press, 2015, pp. 403–414.

[33] C. Zhifei, "Pysmell a tool for detecting code smells in python code," Nov. 2018. [Online]. Available : https://github.com/chenzhifei731/Pysmell

[34] M. Fowler, *Refactoring : improving the design of existing code.* Addison-Wesley Professional, 2018.

[35] W. H. Brown, R. C. Malveau, H. W. McCormick, and T. J. Mowbray, *AntiPatterns : refactoring software, architectures, and projects in crisis.* John Wiley & Sons, Inc., 1998.

[36] J. Svajlenko and C. K. Roy, "Evaluating modern clone detection tools," in *2014 IEEE International Conference on Software Maintenance and Evolution.* IEEE, 2014, pp. 321–330.

[37] C. K. Roy and J. R. Cordy, “A survey on software clone detection research,” *Queen’s School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[38] C. Kapser and M. W. Godfrey, “Toward a taxonomy of clones in source code : A case study,” *Evolution of large scale industrial software architectures*, vol. 16, pp. 107–113, 2003.

[39] L. Aversano, L. Cerulo, and M. Di Penta, “How clones are maintained : An empirical study,” in *11th European Conference on Software Maintenance and Reengineering (CSMR’07)*. IEEE, 2007, pp. 81–90.

[40] M. Mondal, C. K. Roy, and K. A. Schneider, “A comparative study on the bug-proneness of different types of code clones,” in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 91–100.

[41] M. Mondal, B. Roy, C. K. Roy, and K. A. Schneider, “Investigating context adaptation bugs in code clones,” in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2019, pp. 157–168.

[42] H. B. Braiek and F. Khomh, “Deepevolution : A search-based testing approach for deep neural networks,” *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pp. 454–458, 2019.

[43] ——, “On testing machine learning programs,” *Journal of Systems and Software*, vol. 164, p. 110542, 2020.

[44] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *nature*, vol. 521, no. 7553, pp. 436–444, 2015.

[45] 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.

[46] E. Breck, S. Cai, E. Nielsen, M. Salib, and D. Sculley, “The ml test score : A rubric for ml production readiness and technical debt reduction,” in *2017 IEEE International Conference on Big Data (Big Data)*. IEEE, 2017, pp. 1123–1132.

[47] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, “Machine learning testing : Survey, landscapes and horizons,” *IEEE Transactions on Software Engineering*, 2020.

[48] X. Zhang, Y. Yang, Y. Feng, and Z. Chen, “Software engineering practice in the development of deep learning applications,” *arXiv preprint arXiv :1910.03156*, 2019.

[49] Z. Wan, X. Xia, D. Lo, and G. C. Murphy, “How does machine learning change software development practices ?” *IEEE Transactions on Software Engineering*, 2019.

[50] Z. Chen, Y. Cao, Y. Liu, H. Wang, T. Xie, and X. Liu, "Understanding challenges in deploying deep learning based software : An empirical study," *arXiv preprint arXiv :2005.00760*, 2020.

[51] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," *arXiv preprint arXiv :1906.01388*, 2019.

[52] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis.* ACM, 2018, pp. 129–140.

[53] L. Rampasek and A. Goldenberg, "Tensorflow : Biology's gateway to deep learning ?" *Cell systems*, vol. 2, no. 1, pp. 12–14, 2016.

[54] M. B. Kery, M. Radensky, M. Arya, B. E. John, and B. A. Myers, "The story in the notebook : Exploratory data science using a literate programming tool," in *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*, 2018, pp. 1–11.

[55] H. Shen, "Interactive notebooks : Sharing the code," *Nature*, vol. 515, no. 7525, pp. 151–152, 2014.

[56] J. F. Pimentel, L. Murta, V. Braganholo, and J. Freire, "A large-scale study about quality and reproducibility of jupyter notebooks," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR).* IEEE, 2019, pp. 507–517.

[57] F. Psallidas, Y. Zhu, B. Karlas, M. Interlandi, A. Floratou, K. Karanasos, W. Wu, C. Zhang, S. Krishnan, C. Curino *et al.*, "Data science through the looking glass and what we found there," *arXiv preprint arXiv :1912.09536*, 2019.

[58] A. Yamashita, "Assessing the capability of code smells to explain maintenance problems : an empirical study combining quantitative and qualitative data," *Empirical Software Engineering*, vol. 19, no. 4, pp. 1111–1143, 2014.

[59] Z. Soh, A. Yamashita, F. Khomh, and Y.-G. Guéhéneuc, "Do code smells impact the effort of different maintenance programming activities ?" in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 393–402.

[60] M. Perepletchikov and C. Ryan, "A controlled experiment for evaluating the impact of coupling on the maintainability of service-oriented software," *IEEE Transactions on software engineering*, vol. 37, no. 4, pp. 449–465, 2010.

[61] A. MacCormack and D. J. Sturtevant, "Technical debt and system architecture : The impact of coupling on defect-related activity," *Journal of Systems and Software*, vol. 120, pp. 170–182, 2016.

[62] F. Khomh, M. Di Penta, Y.-G. Guéhéneuc, and G. Antoniol, “An exploratory study of the impact of antipatterns on class change-and fault-proneness,” *Empirical Software Engineering*, vol. 17, no. 3, pp. 243–275, 2012.

[63] H. Sajnani, V. Saini, and C. V. Lopes, “A comparative study of bug patterns in java cloned and non-cloned code,” in *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation.* IEEE, 2014, pp. 21–30.

[64] F. Rahman, C. Bird, and P. Devanbu, “Clones : What is that smell ?” *Empirical Software Engineering*, vol. 17, no. 4-5, pp. 503–530, 2012.

[65] J. F. Islam, M. Mondal, and C. K. Roy, “Bug replication in code clones : An empirical study,” in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1. IEEE, 2016, pp. 68–78.

[66] J. F. Islam, M. Mondal, C. K. Roy, and K. A. Schneider, “A comparative study of software bugs in clone and non-clone code.” in *SEKE*, 2017, pp. 436–443.

[67] N. Gode and J. Harder, “Clone stability,” in *2011 15th European Conference on Software Maintenance and Reengineering.* IEEE, 2011, pp. 65–74.

[68] L. Jiang, Z. Su, and E. Chiu, “Context-based detection of clone-related bugs,” in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, 2007, pp. 55–64.

[69] N. Göde and R. Koschke, “Frequency and risks of changes to clones,” in *Proceedings of the 33rd International Conference on Software Engineering*, 2011, pp. 311–320.

[70] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, “An empirical study of code clone genealogies,” in *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, 2005, pp. 187–196.

[71] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, “Cp-miner : Finding copy-paste and related bugs in large-scale software code,” *IEEE Transactions on software Engineering*, vol. 32, no. 3, pp. 176–192, 2006.

[72] A. Lozano and M. Wermelinger, “Tracking clones’ imprint,” in *Proceedings of the 4th International Workshop on Software Clones*, 2010, pp. 65–72.

[73] M. Mondal, M. S. Rahman, C. K. Roy, and K. A. Schneider, “Is cloned code really stable ?” *Empirical Softw. Engg.*, vol. 23, no. 2, p. 693–770, 2018.

[74] G. M. Selim, L. Barbour, W. Shang, B. Adams, A. E. Hassan, and Y. Zou, “Studying the impact of clones on software defects,” in *2010 17th Working Conference on Reverse Engineering.* IEEE, 2010, pp. 13–21.

[75] M. Mondal, C. K. Roy, and K. A. Schneider, "Does cloned code increase maintenance effort ?" in *2017 IEEE 11th International Workshop on Software Clones (IWSC)*. IEEE, 2017, pp. 1–7.

[76] G. developers, "Github rest api search topics," Dec. 2019. [Online]. Available : https://developer.github.com/v3/search/#search-topics

[77] H. B. Braiek, F. Khomh, and B. Adams, "The open-closed principle of modern machine learning frameworks," in *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. IEEE, 2018, pp. 353–363.

[78] P. community, "Radon," dec 2019. [Online]. Available : https://pypi.org/project/radon/

[79] scikit-learn developers (BSD License), "sklearn.preprocessing.kbinsdiscretizer," Dec. 2019. [Online]. Available : https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.KBinsDiscretizer.html

[80] ——, "scikit-learn machine learning in python," Nov. 2019. [Online]. Available : https://scikit-learn.org/stable/

[81] P. C. Brown, *TIBCO Architecture Fundamentals*. Addison-Wesley, 2011.

[82] T. Paiva, A. Damasceno, E. Figueiredo, and C. Sant'Anna, "On the evaluation of code smells and detection tools," *Journal of Software Engineering Research and Development*, vol. 5, no. 1, p. 7, 2017.

[83] C. Rosen, B. Grawi, and E. Shihab, "Commit guru : analytics and risk prediction of software commits," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. ACM, 2015, pp. 966–969.

[84] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes ?" in *ACM sigsoft software engineering notes*, vol. 30, no. 4. ACM, 2005, pp. 1–5.

[85] S. Kim and E. J. Whitehead Jr, "How long did it take to fix bugs ?" in *Proceedings of the 2006 international workshop on Mining software repositories*, 2006, pp. 173–174.

[86] A. G. Koru, D. Zhang, and H. Liu, "Modeling the effect of size on defect proneness for open-source software," in *Third International Workshop on Predictor Models in Software Engineering (PROMISE'07 : ICSE Workshops 2007)*. IEEE, 2007, pp. 10–10.

[87] A. Chatzigeorgiou and A. Manakos, "Investigating the evolution of bad smells in object-oriented code," in *2010 Seventh International Conference on the Quality of Information and Communications Technology*. IEEE, 2010, pp. 106–115.

[88] L. Barbour, L. An, F. Khomh, Y. Zou, and S. Wang, "An investigation of the fault-proneness of clone evolutionary patterns," *Software Quality Journal*, vol. 26, no. 4, pp. 1187–1222, 2018.

[89] D. A. Wheeler, "SLOCCount," https://dwheeler.com/sloccount/, 2004, [Online ; accessed 19-May-2020].

[90] J. R. Cordy and C. K. Roy, "NiCad clone detector," https://www.txl.ca/txl-nicaddownload.html, 2019, [Online ; accessed 20-February-2020].

[91] ——, "The nicad clone detector," in *2011 IEEE 19th International Conference on Program Comprehension.* IEEE, 2011, pp. 219–220.

[92] C. K. Roy and J. R. Cordy, "A mutation/injection-based automatic framework for evaluating code clone detection tools," in *2009 International Conference on Software Testing, Verification, and Validation Workshops.* IEEE, 2009, pp. 157–166.

[93] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once : Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[94] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 2818–2826.

[95] D. Spadini, M. Aniche, and A. Bacchelli, *PyDriller : Python Framework for Mining Software Repositories*, 2018.

[96] M. Neuhäuser, *Wilcoxon–Mann–Whitney Test.* Berlin, Heidelberg : Springer Berlin Heidelberg, 2011, pp. 1656–1658. [Online]. Available : https://doi.org/10.1007/978-3-642-04898-2_615

[97] G. Macbeth, E. Razumiejczyk, and R. D. Ledesma, "Cliff's delta calculator : A non-parametric effect size program for two groups of observations," *Universitas Psychologica*, vol. 10, no. 2, pp. 545–555, 2011.

[98] M. Mondal, C. K. Roy, and K. A. Schneider, "A comparative study on the bug-proneness of different types of code clones," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 91–100.

[99] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. E. Hubbard, and L. D. Jackel, "Handwritten digit recognition with a back-propagation network," in *Advances in neural information processing systems*, 1990, pp. 396–404.

[100] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[101] R. Koschke, "Survey of research on software clones," in *Dagstuhl Seminar Proceedings.* Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[102] A. Munappy, J. Bosch, H. H. Olsson, A. Arpteg, and B. Brinne, "Data management challenges for deep learning," in *2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA).* IEEE, 2019, pp. 140–147.

[103] T. Wang, M. Harman, Y. Jia, and J. Krinke, "Searching for better configurations : a rigorous approach to clone evaluation," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013, pp. 455–465.

[104] C. K. Roy and J. R. Cordy, "Nicad : Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *2008 16th iEEE international conference on program comprehension.* IEEE, 2008, pp. 172–181.

[105] H. Jebnoun, "Clones in deep learning code," Oct. 2020. [Online]. Available : https://github.com/Hadhemii/ClonesInDLCode

[106] Z. Chen, L. Chen, W. Ma, and B. Xu, "Detecting code smells in python programs," in *2016 International Conference on Software Analysis, Testing and Evolution (SATE).* IEEE, 2016, pp. 18–23.

APPENDIX A ACHIEVEMENTS

Parts of the content presented in this thesis is published or submitted for publication as follows :

- The scent of Deep Learning Code : An Empirical Study
  Hadhemi Jebnoun, Houssem Ben Braiek, M. Masudur Rahman, and Foutse Khomh, published in the *Proceeding of The 17th International Conference on Mining Software Repositories (MSR)*, Seoul, South Korea, May, 2020.
- Clones in Deep Learning Code : What, Where, and Why ?
  Hadhemi Jebnoun, Md Saidur Rahman, and Foutse Khomh, submitted to the *Empirical Software Engineering (EMSE) journal, Novembre, 2020.*