Empirical Software Engineering manuscript No.
(will be inserted by the editor)

Clones in Deep Learning Code: What, Where, and Why?

Hadhemi Jebnoun · Md Saidur Rahman · Foutse Khomh · Biruk Asmare Muse

Received: date / Accepted: date

Abstract Deep Learning applications are becoming increasingly popular worldwide. Developers of deep learning systems like in every other context of software development strive to write more efficient code in terms of performance, complexity, and maintenance. The continuous evolution of deep learning systems imposing tighter development timelines and their increasing complexity may result in bad design decisions by the developers. Besides, due to the use of common frameworks and repetitive implementation of similar tasks, deep learning developers are likely to use the copy-paste practice leading to clones in deep learning code. Code clone is considered to be a bad software development practice since developers can inadvertently fail to properly propagate changes to all clones fragments during a maintenance activity. However, to the best of our knowledge, no study has investigated code cloning practices in deep learning development. The majority of research on deep learning systems mostly focusing on improving the dependability of the models. Given the negative impacts of clones on software quality reported in the studies on traditional systems and the inherent complexity of maintaining deep learning systems (e.g., bug fixing), it is very important to understand the characteristics and potential impacts of code clones on deep learning systems. This paper examines the frequency, distribution, and impacts of code clones and the code cloning practices in deep learning systems. To accomplish this, we use the NiCad clone detection tool to detect clones from 59 Python, 14 C#, and 6 Java based deep learning systems and an equal number of 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 tax-

Hadhemi Jebnoun · Md Saidur Rahman · Foutse Khomh · Biruk Asmare Muse
E-mail: {hadhemi.jebnoun,saidur.rahman,foutse.khomh, biruk-asmare.muse}@polymtl.ca
DGIGL, Polytechnique Montreal, QC, Canada,

onomy. In addition, we study the correlation between bugs and code clones to assess the impacts of clones on the quality of the studied systems. Finally, we introduce a code clone taxonomy related to deep learning programs based on 6 DL software systems (from 59 DL systems) and identify the deep learning system development phases in which cloning has the highest risk of faults. Our results show that code cloning is a frequent practice in deep learning systems and that deep learning developers often clone code from files contain in distant repositories in the system. In addition, we found that code cloning occurs more frequently during DL model construction, model training, and data preprocessing. And that hyperparameters setting is the phase of deep learning model construction during which cloning is the riskiest, since it often leads to faults.

Keywords Code Clones · Deep Learning · Clone Taxonomy

1 Introduction

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. DL is part of a broader family of ML 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). DL networks are artificial neural networks with more than three layers. DL enables computers to build concepts from data based on simpler concepts [26]. The field of DL 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 [26].

Traditionally, software systems are constructed deductively by writing down the rules that govern the behavior of the system as program code. However, with DL, these rules are inferred from training data and they are generated inductively. This consequently reduces the considerable manual work necessary to handcrafting features as required for classical machine learning approaches. DL models also have more sophisticated architecture than traditional ML models, capturing long-range dependencies and modeling them with data.

The main purpose of deep learning is to construct models with high performance that is able to learn patterns from the input data in order to make predictions for new data. To find the optimal model, deep learning practitioners used to implement quickly prototypes 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 with or without some modifications, they often may end up writing poor code and duplicated functions or blocks known as clones. Also, code clones are often 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 deep learning development [57]. For example, deep learning developers can clone models' architectures and model (hyper)parameters settings or initialization for similar model implementations.

Code clones are a kind of code smell and code smells are violations of some fundamental design principles. The existence of code smells [23] may affect the maintenance and the evolution of the software systems. In traditional systems, they negatively impact software quality [93] and tend to increase technical debts, and consequently incur additional development and maintenance costs. Some types of code smells may also increase the consumption of certain resources (processor, memory, etc.) [27,97] and, consequently, hinder the deployment of efficient and sustainable solutions. Anawer et. al. [3] reported that duplicate code may be related to increased energy consumption by the software applications. For deep learning systems, concerns are growing regarding the high energy consumption by deep learning systems[1]. 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 is likely to pose challenges to the maintenance of deep learning projects. In fact, ML-based systems are expected to have the maintenance concerns of traditional systems as well as those specific to ML.

Earlier studies on traditional software systems show that about 7%-23% of code in the software repositories are cloned code [83]. The intuitive benefit of clones is a short term productivity gain by code reuse. Some earlier studies reported positive impacts of clones [44] and showed that clones are not necessarily harmful [34,50]. However, there are substantial empirical evidences showing that clones can negatively impact software quality by affecting the stability of the code [58,64,74] and making it bug-prone [6,7,42,56,54,75,98], 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 [84].

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

[1] `https://www.forbes.com/sites/robtoews/2020/06/17/deep-learnings-climate-change-problem/?sh=7de914936b43`

aim to understand the extent to which DL developers duplicate code, as well as the impact of code clones on the maintenance of DL systems. To the best of our knowledge, this paper presents the first empirical study on DL code clones, where one performs a comparative study of the distribution and bug-proneness of clones in deep learning and traditional software systems. In this paper, we analyze clones in 59 Python, 14 C#, and 6 Java based deep learning systems and an equal number of traditional software systems. We 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 projects to perform a manual analysis of their code clones. We build a taxonomy of DL code clones 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 phases of the development process of deep learning systems in which code cloning has the highest risk of bugs.
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 are 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. The majority of code clones in deep learning code are located in different files i.e, in the files in same or in different directories.

– Although the prevalence and distribution vary with the systems for different programming languages (Python, Java, and C#), the overall trend of differences between deep learning and traditional systems remains the same, with respect to both prevalence and distribution of clones. For all programming languages, DL clones are prevalent and dispersed compared to clones in traditional systems. These differences are observed to be statistically significant in most cases in Python, but in a few cases for Java and C# systems. So, the observed relationships might be programming language specific.

– Code clones in deep learning code are likely to be more defect-prone compared to non-cloned code. Type 3 clones (i.e., clones with differences because of added, deleted, or modified lines) are at higher risk of bugs compared to other clone types.

– Our results show that 75.85% of the bug-fix commits in Python DL systems are related to clones while that is 45.31% and 27.54% respectively for Java and C# systems. Similarly, Type 3 clones in Python are more buggy compared to the same type in Java and C#, although the extent of bug-pronenes varies with programming languages. We also observe that clones in DL systems tend to take more time to get fixed compared to traditional systems.

– We classify code clones by deep learning phases and found that three main DL 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 have the highest risk of bugs: model construction (50%), model training (20%), data collection (13.3%), data preprocessing (10%), data post-processing (3.3%), and hyperparameter tuning (3.3%). In brackets we provided the percentage of clones that experienced a bug fixing change.

The rest of the paper is organized as follows. In Section 2, we describe basic concepts of code clones, deep learning, and the bug-proneness of code clones. In Section 3, we present our study design by introducing the research questions and methodology. Our findings and discussions are presented in Section 4. Section 6 discusses the threats to validity. Section 7 presents the related work and Section 8 concludes the paper and introduces some future work.

2 Background

In this section, we briefly discuss the concepts and terminology related to our empirical study. We give an overview of the characteristics of the deep learning systems and trends in the deep learning application development. We also provide brief descriptions of the phases of deep learning application development. In addition, 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.

2.1 Deep Learning

In this section, we review the main concepts of DL and discuss the different phases of the life cycle of DL system development.

Deep Learning (DL) is a sub-domain of Machine Learning (ML) that involves stacking of multiple layers of neural networks to provide a powerful model with the ability to learn from data. Machine learning techniques other than deep learning rely 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 recently and has dealt with complex applications with increasing accuracy over time [26]. Deep learning has been applied in various fields and in several activities and services in daily life including transportation, health, and finance [32, 61, 69]. Deep learning has contributed to several research fields such as computer vision [22,96], speech recognition [33,85], machine translation [9,18], software engineering [28,55, 103], etc.

There is an abundant literature on the software quality assurance of traditional software systems [1,14,30,51,78]. However, very few studies have investigated quality issues in deep learning software systems. Deep learning software development may face all the development and maintenance challenges of a traditional software system, in addition to DL/ML specific challenges related to their dependence to data, inductive nature, and the complexity to understand their behavior [88]. 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 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. [31]. This workflow was inferred from analyzing various deep learning frameworks (e.g, Tensorflow, Pytorch, and Theano) Fig. 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 [2]. 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.

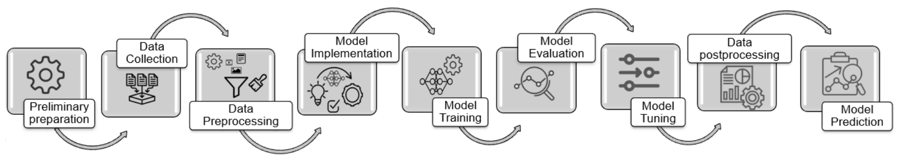

Fig. 1 Deep Learning Workflow

Preliminary preparation: It is a coding-based 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 to perform this DL phase.

Data collection: Deep learning practitioners start by gathering data required to model the target business problem. The collection could be from available open source or internal dataset. This step could be done by reading file(s) from the disk, or by calling a REST or Web service 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 possibly by modifying only the source paths. Thus, they may end up duplicating code for data acquisition or data streaming.

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 input data 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, they will be near-miss (closely similar) clones as they have the same or 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 pre-processing data tasks. Calling functions to collect data with or without modifications for each specific need is likely to introduce clones in deep learning code.

Model implementation: 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 [31]. 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. 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 [31]. Since many 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, the trained model is ready to be evaluated on its performance. Thus, deep learning practitioners need the validation dataset, 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 [31]. Koenzen et al. [48] 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's configuration. 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 model evaluations.

Model tuning: To optimize the performance of the model, hyper-parameters are tuned. Using an unsuitable loss function, incorrectly initializing 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 [8]). Hyper-parameter tuning is an essential phase to improve the model's performance. The implementation of this technique is either done 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 clones

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 fragments are usually identified by their file names, start line number, and end line number. Code clones could be detected by pair or by class.

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

Clone class: Code clones can also be presented as clone classes. Each clone class contains a set of fragments that are clones to each other.

2.2.1 Clone Taxonomies

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

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 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, the two fragments will be exact when we ignore the naming differences (function name, name of input variables). 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. Examples of two Type 3 code fragments are showed in Table 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.

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 4 shows an example of Type 4 clone. The two functions differ syntactically, but they

Table 1 Type 1 Clones

`def forward_activation(self, X):`
`    #compute post activation value of X`
`    if self.activation_fct == "sigmoid":`
`        return 1.0/(1.0 + np.exp(-X))`
`    elif self.activation_fct == "tanh":`
`        return np.tanh(X)`
	`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)`

Table 2 Type 2 Clones

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

Table 3 Type 3 Clones

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

Table 4 Type 4 Clones

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

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) [20], 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 [94].

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 [80]. In this paper, we follow the location-based taxonomy proposed by Kapser and Godfrey [43]. They 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.
- **Different directories** when clones are detected in different files and different directories.

They further sub-classified 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 using the location-based taxonomy. We use the different steps of the deep learning workflow presented above (Figure 1) to label the detected clones.

2.2.2 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 [58, 64, 74]. 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 [7, 5, 42, 25]. 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 [4, 6, 7].

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 [58, 59, 64, 74, 25]. The instability of clones, in turn, has also been found to be related to bugs [75]. The bug-proneness of clones may also vary based on the types of clones [65]. Several other prior research works have also investigated the bug-proneness of code clones [63, 56, 54]. 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 [10, 11]. 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 recent survey [57]. In this paper, 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.

3 Study Design

In this section, we first present our research questions by highlighting the motivation and research objectives. Then we describe our research methodologies.

3.1 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: Are code clones more prevalent 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. [42] Although code clones have been widely investigated for traditional software systems [84], the impact of code cloning in deep learning systems is still unknown. The widespread use of common open-source libraries and frameworks and the use of code examples from crowd-source question-answering web sites (e.g., Stack Overflow) may introduce duplicate code in DL systems. Moreover, given the code reuse from 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 prevalence and distribution of clones in deep learning systems. A comparative analysis of code cloning in traditional and deep learning based systems is important to understand if deep learning code is more prone to clones than traditional programs, and therefore deserve a special attention, from the research community and tools builders to help practitioners manage clones in deep learning systems efficiently.

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 use the taxonomy proposed by Kapser and Godfrey [43] 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 [6,7]. However, it is important but yet to know the impacts of cloning in deep learning systems especially on the bug-proneness. This research question investigates the relationship between bug-proneness and code clones occurrences in deep learning code by performing Mann Whitney statistical test and analyzing the effect size. We examine the impacts of different types of clones on the bug-proneness of deep learning code and assess 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 of 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 techniques. 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 may often duplicate the code of an existing tested 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 where code cloning occurs frequently.

RQ5: In which phases of deep learning development code cloning is more prone to faults?
Since previous works on traditional systems [5] 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 or in certain functions of the deep learning code are more bug-prone than others. Therefore, for this research question, we compare the risk of faults of clones found in the different functions of the deep learning code.

3.2 Study Overview

In this section, we present our study design as shown in Figure 2. We divide our methodology into five main steps.

Fig 2-A We first clone 79 DL and 79 traditional repositories from GitHub, the detailed methodology is described in the Subsection 3.2.1. We then detect code clones for both DL and traditional open-source projects using the NiCad clone detector (details are presented in section 3.2.3). 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 2-B We analyze the distribution of code clone by applying the location taxonomy. As shown in Fig 2-B, we have three locations where there might be code clones classes: same file, same directory, and different directories (details are presented in Section 3.2.4). An arrow towards a balance as shown in the figure 2-B represents the comparative analysis of the distribution of code clones between deep learning and traditional systems in terms of localization.

Fig 2-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 3.2.2). 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 3.2.6).

Fig 2-D We manually classify code clones that are DL-related to construct a taxonomy of code clones in DL code (see Section 3.2.5).

Fig 2-E Finally, we examine the risk of bugs of code clones occurring in specific phases of the DL development.

3.2.1 Subject Systems

We mined 138 open-source projects (59 Python, 6 Java and 14 C# deep learning projects with equal number of traditional projects in each programming language) from GitHub. We used the same Python projects investigated in Jebnoun et al. [40] study on detecting code smells in DL code. Our empirical

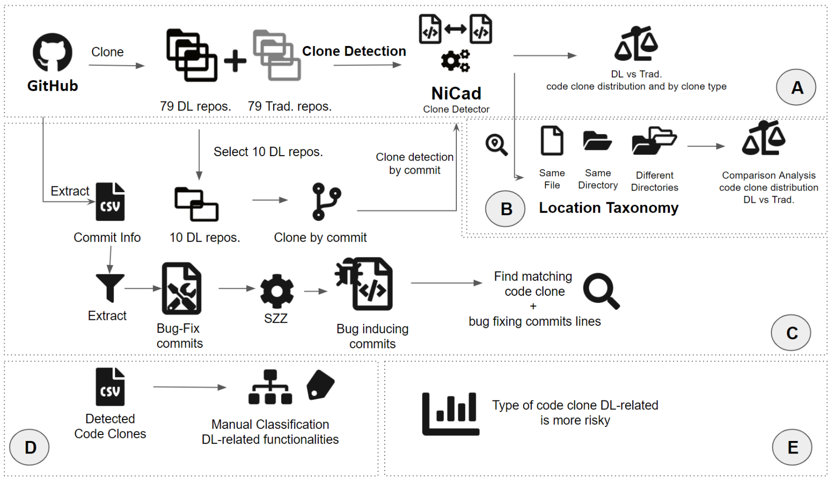

Fig. 2 Study Overview A- Detecting Code Clones in Deep Learning and Traditional Repositories, B- Applying Code Clone Location Taxonomy, C- Studying the Relationship between Bug-proneness and Code Clones, D-Classifying Code Clones Manually Based on DL-related Functionalities, E-Exploring riskier DL-related Code Clones

study is primarily focused on the set of projects [29] in Python, as it is the most widely used language in the machine learning field [12]. However, we also analyzed Java and C# projects for the generalizability of the findings. Deep learning projects were selected by first searching repositories using DL-related keywords (e.g., deep learning, deep neural network, convolutional neural network) [40] and manually filtering out tutorials and some projects with a low number of releases (to obtain our 59 Python, 6 Java and 14 C# 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. [17] (similarly to Jebnoun et al. [40] study). This benchmark consists of 106 GitHub repositories with at least 1K stars each from which we randomly selected a subset of 59 traditional Python projects. We use the same set of 59 traditional Python projects studied by Jebnoun et al. [40], for the comparative analysis in this study. We apply the same selection criteria for selecting 6 Java and 14 C# traditional projects.

3.2.2 Preprocessing of source code repositories

Selection of a Subset of Subject Systems: Our research questions (RQ1, RQ2) are based on the analysis of 59 Python, 6 Java, and 14 C# deep learning systems and an equal number of traditional systems. For RQ3, we analyzed 59 Python, 2 Java and 2 C# systems for each of the DL and traditional categories. However, we randomly select 6 Python 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 provide more details about the 6 DL repositories : name of the repository, number of commits, total lines of code, URL and size in appendix A. 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 [102] tool to compute the total source lines of code (SLOC) of Python, Java and C# code in each project from corresponding languages. SLOCCount is a software metrics open-source tool developed by David A. Wheeler supporting several programming languages. It is robust to handle variations 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. We consider only the size of the Python, Java and C00# code based on the type of the systems. 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. This gives a normalized (per LOC) representation of the frequency of clones for comparison.

3.2.3 Code Clone Detection

For detecting code clones, we use the *NiCad* tool [20]. NiCad [19] can detect both exact and near-miss clones with high precision and recall [82] with respect to blocks and functions granularity.

NiCad Settings: Table 5 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 of NiCad for the detection of Type 2 and Type 3 clones. Type 3 clones are detected with a dissimilarity threshold of 30%.

Clone Detection: We detect code clones for a particular snapshot and for each commit of each repository. To perform a comparative analysis of the

Table 5 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]

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 with settings detailed in Table 5. 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 [38]. 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: To perform clone-type based investigation, we need to separate each type of clones. This is because by definition (as in Section 2.2.1), Type 2 clones include Type 1 clones and Type 3 clones include both Type 1 and Type 2 clones. To separate Type 2 clones, we exclude matching clone classes between Type 1 and Type 2 from the outputs of Type 2 clone detection results from NiCad. As NiCad results for Type 2 contain clone classes from Type 1 since the set of Type 1 clones (exact copies) is a subset of the set of Type 2 clones (e.g. with differences in identifier names). 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.

3.2.4 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 [43]. 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 (in percentage) of both clones fragments and cloned LOCs distributed over the location types defined by the taxonomy. We perform Mann-Whitney statistical test to compare whether the distribution of code clones in DL and traditional systems are significantly different. We further extend our comparative analysis by considering individual clone types.

3.2.5 Labeling clones based on the taxonomy of deep learning tasks

Since common functional steps are followed to build deep learning models and the same deep learning libraries are used, deep learning practitioners may 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 all deep learning phases (i.e., building model and training) but also in 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 595 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 task or functionality. We further grouped the clone fragments in each subcategory and mapped them to the different phases of the development workflow of deep learning models, as discussed in Section 2.1. 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 at least two authors, all with academic and industry backgrounds. The resulting taxonomy was then cross-validated, and disagreements were resolved by group discussion.

Tables 6, 7, and 8 present 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 Tables 6, 7, and 8, we present the clone types of the example fragments, and the sub-category and category of DL-related phases to which the clone class or fragments belong to. 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 6 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) [77], 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.

Table 6 Example of Clone Fragment related to 'Model Training'

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

The second example contains two fragments of Type 3 clones as shown in Table 7. 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 8 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 [95], which is a Deep Convolutional Neural Network 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.

3.2.6 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 [6,7,42,56,54,58,64,74,75,98]. Therefore, it is important to study whether and to what extent this relationship holds in the context of deep learning systems.

Table 7 Example of Clone Fragment related to 'Data Collection'

Code Fragment	Clone Type	Sub-category	Category
`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)`			

Table 8 Example of Clone Fragment related to 'Data pre-processing'

Code Fragment	Clone Type	Sub-category	Category
`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

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. [79]. At least one of the keywords from this set should be in the commit message to consider it as a bug-fixing commit.

Bug Inducing commits: We use PyDriller [92] 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 modified by 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 are 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. 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, we obtain which tasks of DL code are more likely to introduce bugs than others when cloned. We perform MWW tests for the distributions of code clones and non-clone code in the DL bug-fix commits along with effect size analyses. To determine if there is a statistically significant difference between the number of commits that fix bugs on cloned lines and the number of commits that fix bugs on non-cloned lines.

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. We compute the difference in time between bug fixing commit and their related bug inducing commit as introduced by Kim and Whitehead [47]. To do so, 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 find if there is any significant difference in the time required to fix bugs related and not related to clones.

Testing statistical significance and the effect size: To answer the research questions, we computed different metrics based on the quantitative analysis of clones in deep learning and traditional systems, respectively. To verify whether the differences between the corresponding metrics for DL and traditional code are statistically significant, we performed Mann-Whiteny-Wilcoxon (MWW) [68] test (two-tailed, p-value significant at <0.5). We apply MWW test because this is a non-parametric test i.e., it does not require data to be normally distributed. Moreover, this test can be applied on small data set. However, MWW test only confirms whether the observed differences between two sets of values are by chance. So, the observed differences to be statistically significant, we also need to measure the effect size. We apply Cliff's delta [60] effect size along with the MWW test. This also a non-parametric measure of effect size. The range of the values for cliffs delta effect size is [-1, +1]. The value of effect size is generally interpreated as small (~ 0.20), medium (~ 0.5), and large (~ 0.8). Larger effect size refers to stronger relationships between the set of observations tested. Here, +1 and -1 values for the effect size indicate the absence of overlap while value 0 indicates a lot of overlap between two set of samples. We used `mannwhitneyu()` functions for the Python package `scipy.stats` for MWW test and calculated effect size by `cliffsDelat()` function as implemented in [21].

4 Study Findings and Discussions

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

4.1 **RQ1 : Are code clones more prevalent 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.

To answer this research question, we detect code clones in 59 deep learning systems and 59 traditional software systems developed in Python. 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. To compare we need normalized values of the clone densities in different systems as the systems varies in code size. For normalization, 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 3.2.2. We then perform the Mann-Whitney Wilcoxon (MWW) test [68] 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 [60], 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.

Clone Occurrences by Project Type: Fig. 3 shows the comparison of clone occurrences between DL-based and traditional systems considering all clone types 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. 3, 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, the difference is not by chance, we perform Mann-Whitney Wilcoxon (MWW) tests [68] (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 9. The column 'All' in Table 9 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 is statistically significant. Thus, DL systems have higher density of clones compared to traditional systems.

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 9, 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. 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.

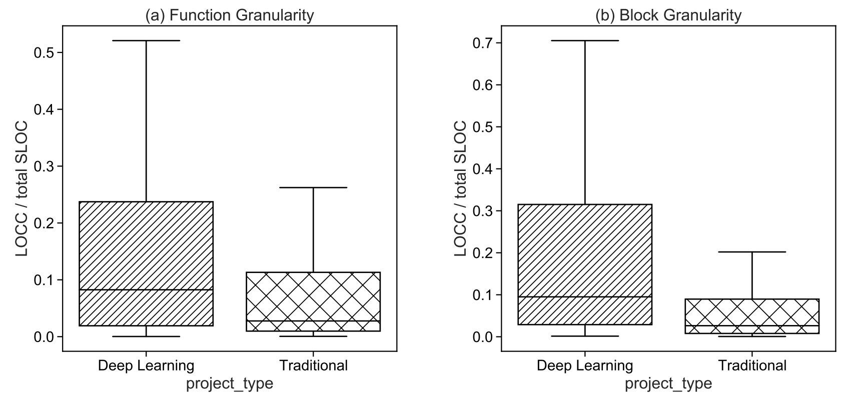

Fig. 3 Code Clones Occurrences in DL and Traditional Python Projects for Both Code Clones Granularities: (a) Function, (b) Block. ***LOCC***: Lines Of Code Clones, ***SLOC***: Source Lines Of Code.

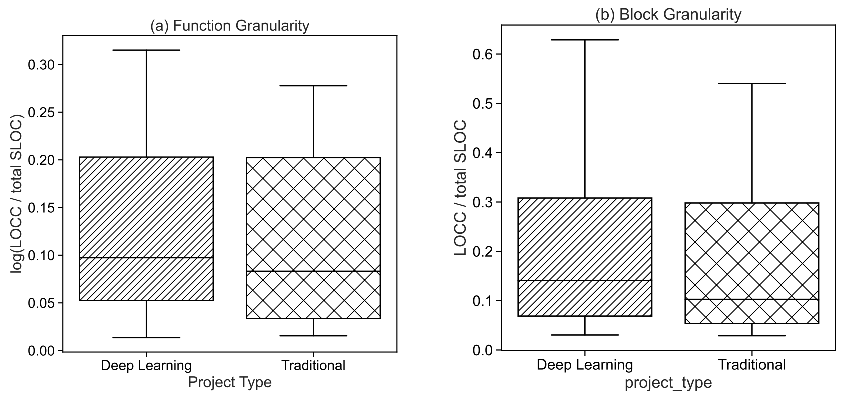

Fig. 4 Code Clones Occurrences in DL and Traditional Java Projects for Both Code Clones Granularities: (a) Function, (b) Block. ***LOCC***: Lines Of Code Clones, ***SLOC***: Source Lines Of Code.

We also studied 6 Java and 14 C# systems to investigate whether the clone distributions observed in Python-based systems generalize to other programming languages. Fig. 4 and Fig. 5 show the comparison of clone densities between DL-based and traditional Java and C# systems respectively, considering all clone types for both function and block granularity. In both Fig. 4 and Fig. 5, we observe that the median of the clone density for deep learning systems is comparatively higher than that of traditional systems. We observe similar differences for both function and block granularity. These observations for Java and C# systems are similar to the observation from Python based systems. This suggests that deep learning systems tend to have higher proportions of

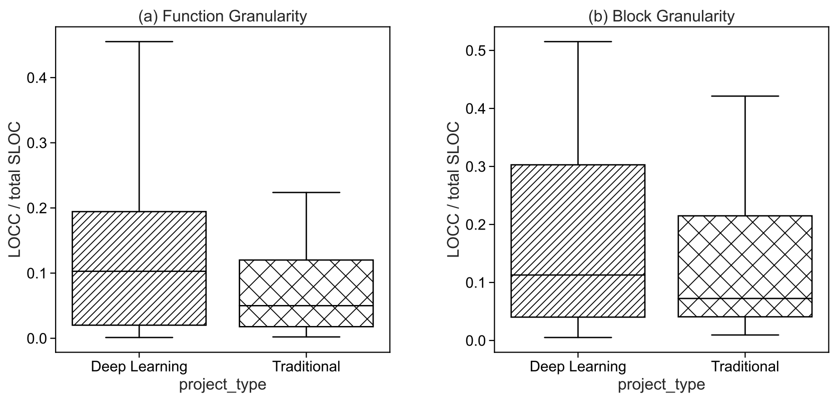

Fig. 5 Code Clones Occurrences in DL and Traditional C# Projects for Both Code Clones Granularities: (a) Function, (b) Block. ***LOCC***: Lines Of Code Clones, ***SLOC***: Source Lines Of Code.

cloned code compared to traditional systems. Table 10 and Table 11 show the results for statistical significance test (two-tailed MWW test, significant at <0.05) results. We do not observe statistically significant differences between the density of cloned code in Deep Learning and Traditional Java and C# systems costrasting our results based on Python based systems. These differences between clone densities in Python system from that of Java and C# systems could be explained by the differences in syntactic structures, compactness of code, availability and usage patterns of libraries, differences in object-oriented design, etc.

We therefore conclude 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, programming languages, 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 9 Mann-Whitney Test and Cliff's Delta Results between DL and Traditional Python Projects

Clone Types	**Type 1**		**Type 2**		**Type 3**		**All**	
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. 6, 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.

Table 10 Mann-Whitney Test and Cliff's Delta Results between DL and Traditional Java Projects

Clone Types	**Type 1**		**Type 2**		**Type 3**		**All**	
Granularity	**Function**	**Block**	**Function**	**Block**	**Function**	**Block**	**Function**	**Block**
p-value	0.93	0.92	0.57	0.64	0.81	0.64	0.73	0.60
Cliff's Delta	0.055	0.066	0.222	0.2	0.111	0.2	0.018	0.111

Table 11 Mann-Whitney Test and Cliff's Delta Results between DL and Traditional C# Projects

Clone Types	**Type 1**		**Type 2**		**Type 3**		**All**	
Granularity	**Function**	**Block**	**Function**	**Block**	**Function**	**Block**	**Function**	**Block**
p-value	0.64	0.39	0.62	0.42	0.05	0.22	0.22	0.31
Cliff's Delta	0.115	0.197	0.119	0.183	0.438	0.510	0.159	0.127

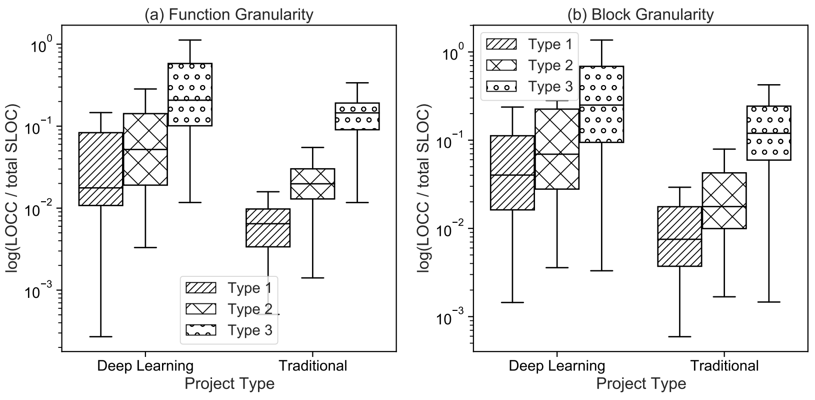

Fig. 6 Clone Density in DL and Traditional Python Projects for Clone Types and Granularity ***LOCC:*** *Lines Of Code Clones*

To calculate clone density for individual clone 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 system. 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. 6, 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. 6, 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

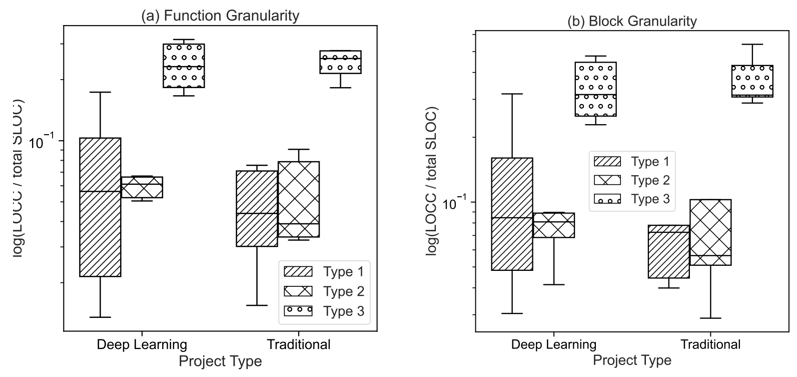

Fig. 7 Clone Density in DL and Traditional Java Projects for Clone Types and Granularity
LOCC: *Lines Of Code Clones*

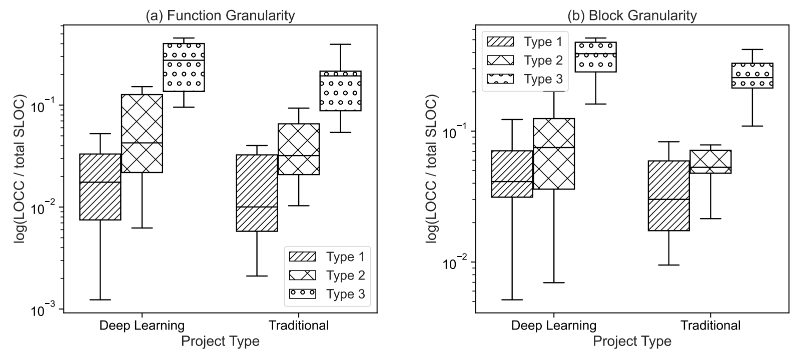

Fig. 8 Clone Density in DL and Traditional C# Projects for Clone Types and Granularity
LOCC: *Lines Of Code Clones*

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 9 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 clone densities in deep learning code and traditional code with respect to Type 1 clones 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 in clone densities 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 in clone densities 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 overall trend of deep learning systems having more clones than traditional systems, although the magnitude of the differences may not always be 'large'.

Fig. 7 and Fig. 8 show the distributions of clone densities in DL and traditional Java and C# systems respectively. We observe that the median density of clones in DL systems tend to be higher compared to traditional systems for clone types in Java and C# systems except for Type 3 clones in Java systems (Fig. 7). Our clone-type centric analysis show that densities of cloned code tend to be higher in DL systems compared to traditional systems. However, we do not observe these differences to be statistically significant.

As the dissimilarity threshold for the clone detection tool is known to be important confounding factor for the empirical analysis of clones, we also perform the analysis with 20% dissimilarity threshold for NiCad. We presented our results for 20% threshold in Appendix B (Fig. 29, Fig. 31, Fig. 30, and Fig. 32). Our results show that the overall trends in the prevalence of clones in deep learning and traditional systems for 20% threshold do not change from what we observed for 30% threshold.

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 address in the remaining research questions. However, further investigation is necessary to generalize the findings and to identify the factors that influence the prevalence of clones in DL and traditional software systems.

Summary of findings (RQ1): As shown by our experimental results, the density of code clones in DL-based systems tend to be higher than that of traditional systems, and the difference is statistically significant for Python systems. Regarding clone types, all three clone types (Type 1, Type 2, and Type 3) are more prevalent in DL-based systems than in traditional software systems. The comparative higher prevalence of clones in DL systems compared to traditional systems are likely to be programming language dependent.

4.2 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 tends to be 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 [43]. 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. Otherwise, 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 3.2.4). We then count the proportion of lines of code clones of different location categories for deep learning and traditional systems. 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 proportion (in percentage) of the lines of code clones, shows the distribution of clones from a relative code size point of view. A few larger clone fragments in closer proximity are likely to have less navigation overhead than a higher number of small clone fragments scattered in distant location, even if they have equal total LOC of the larger clone fragments. So, fragment-based analysis of the distribution of clones is also important to have insights on the potential impacts of clones on the maintenance of the systems. 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 to better understand the intents and potential impacts of the proximity of clones (in the code base) and their relative distribution, and impacts on software quality.

Overall location-based distribution of clones: Fig. 9 shows the distribution of code clones (for the function granularity) in DL and traditional Python 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' cate-

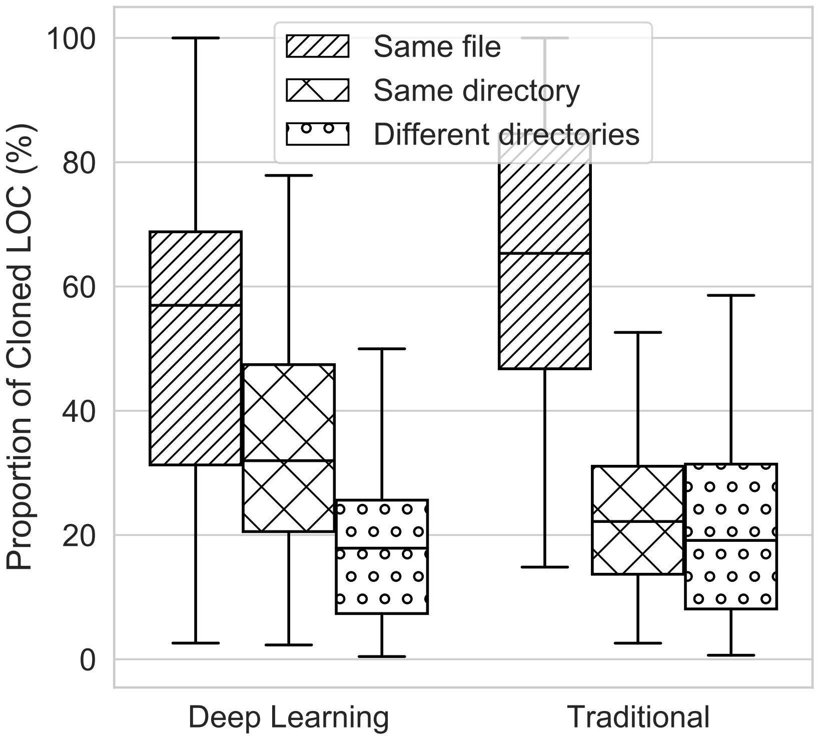

Fig. 9 Code Clones Distribution by Location in DL and Traditional Python Systems Regarding Percentage of Lines of Code Clones (LOCC). i.e, (LOCC/total LOCC)x 100

gory and finally the 'different directories' category. The distributions of 'same directory' and 'different directories' categories overlap significantly for clones in traditional code.

Table 12 Mann-Whitney Test and Cliff's Delta Results Regarding the Distributions of Clones in DL and Traditional (Trad) Python Projects.

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

SF: Same File, SD: Same Directory, DD: Different Directories, CD: Cliff's Delta

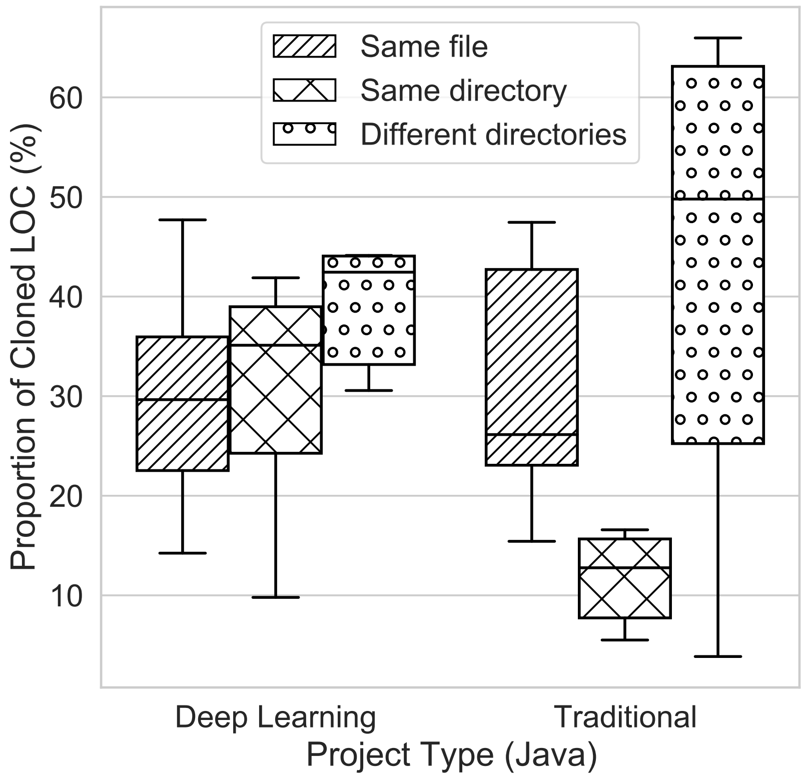

Fig. 10 Code Clones Distribution by Location in DL and Traditional Java Systems Regarding Percentage of Lines of Code Clones (LOCC). i.e, (LOCC/total LOCC)x 100

Table 12 shows the p-values from the Mann-Whitney test and Cliff's delta values for 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 12 designates the unfiltered Type 3 (i.e, include all fragments from Type 1, Type 2, and Type 3). We found statistically significant differences 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 reside in 'same file' compared to the percentages of cloned code that reside in the 'same directory' and in 'different directories'. Similarly, the percentage 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 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

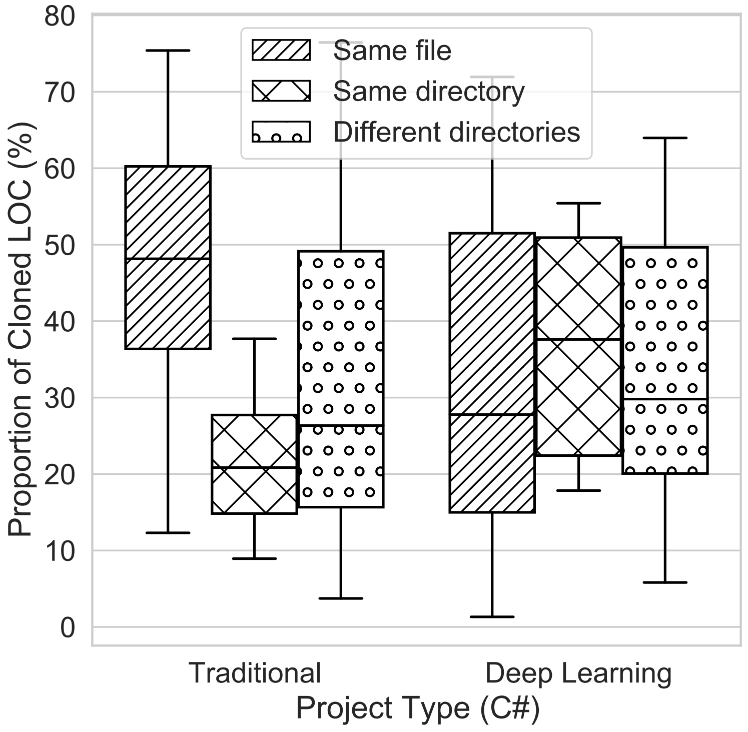

Fig. 11 Code Clones Distribution by Location in DL and Traditional C# Systems Regarding Percentage of Lines of Code Clones (LOCC). i.e, (LOCC/total LOCC)x 100

Table 13 Mann-Whitney Test and Cliff's Delta Results Regarding the Distributions of Clones in DL and Traditional (Trad) Java Projects.

		Location					
		SF-SD		**SF-DD**		**SD-DD**	
Clone Type	**Proj Type**	**p-value**	**CD**	**p-value**	**CD**	**p-value**	**CD**
ALL	**DL**	0.936	0.05	0.378	0.33	0.229	0.44
	Trad	0.092	0.61	0.810	0.11	0.297	0.38
Type 1	**DL**	0.155	0.66	0.155	0.66	0.810	0.11
	Trad	**0.022**	**0.86**	**0.012**	**1.0**	0.082	0.66
Type 2	**DL**	0.229	0.44	0.378	0.33	0.936	0.05
	Trad	0.065	0.66	0.784	0.13	0.315	0.4
Type 3	**DL**	0.810	0.11	0.297	0.38	0.297	0.38
	Trad	0.170	0.53	0.936	0.05	0.522	0.26

SF: Same File, SD: Same Directory, DD: Different Directories, CD: Cliff's Delta

< 0.05 (8.34e-13 and 2.05e-11 respectively) and with large effect sizes (0.83 and 0.81 respectively). We found no statistically significant difference between the percentage of clones contained in the 'same directory' category and the 'different directories' category, for traditional code.

Table 14 Mann-Whitney Test and Cliff's Delta Results Regarding the Distributions of Clones in DL and Traditional (Trad) C# Projects.

		Location					
		SF-SD		**SF-DD**		**SD-DD**	
Clone Type	**Proj Type**	**p-value**	**CD**	**p-value**	**CD**	**p-value**	**CD**
ALL	**DL**	0.368	0.21	0.752	0.07	0.56	0.14
	Trad	**0.001**	**0.71**	**0.044**	**0.46**	0.544	0.14
Type 1	**DL**	**0.039**	**1.0**	0.124	0.77	0.901	0.04
	Trad	0.087	0.51	**0.0169**	**0.72**	0.221	0.29
Type 2	**DL**	0.488	0.18	0.113	0.43	0.230	0.35
	Trad	**0.0001**	**0.86**	**0.004**	**0.66**	0.85	0.04
Type 3	**DL**	0.341	0.22	0.644	0.11	0.849	0.05
	Trad	**0.0003**	**0.80**	**0.009**	**0.59**	0.576	0.13

SF: Same File, SD: Same Directory, DD: Different Directories, CD: Cliff's Delta

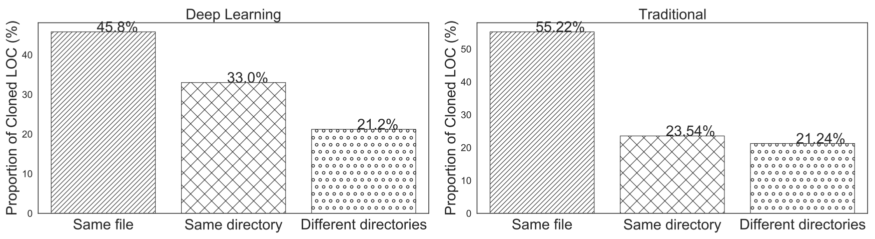

Fig. 12 Percentages of Lines of Code Clones by Location of Clones in both Deep Learning and Traditional Systems (Python)

Fig. 10 shows the proportion of cloned code (LOC %) in different location category for Java systems. Based on the median values of the proportion of cloned code, we observe that the highest proportions of cloned code for Java DL systems reside in 'different directory' followed by the proportion of cloned code in 'same directory' and in 'same file' categories. Similar trend in the distribution of cloned code is observed in Java traditional systems (except for 'same directory' category) with 'different directory' category containing the highest median percentage of cloned LOC. This is an indication that clones in Java DL and traditional systems tend to be dispersed. However, from the results of the MWW test and Cliff's delta effect size for 'All' types as shown in Table 13, we do not observe the differences to be statistically significant.

For C# systems as in Fig. 11, the median value for the distribution of clones in DL systems is the lowest for 'same file' location category while it is the opposite for traditional systems. This shows that clones in C# DL systems are likely to be dispersed compared to traditional C# systems. However, from our statistical test results for C# systems in Table 14, we do not observe statistically significant differences in distribution of cloned code in different location categories. For traditional C# systems, the differences in proportion of clones in 'same file' are significantly high (p-values < 0.05 with medium to large effect size) compared to 'same directory' and 'different directory' categories.

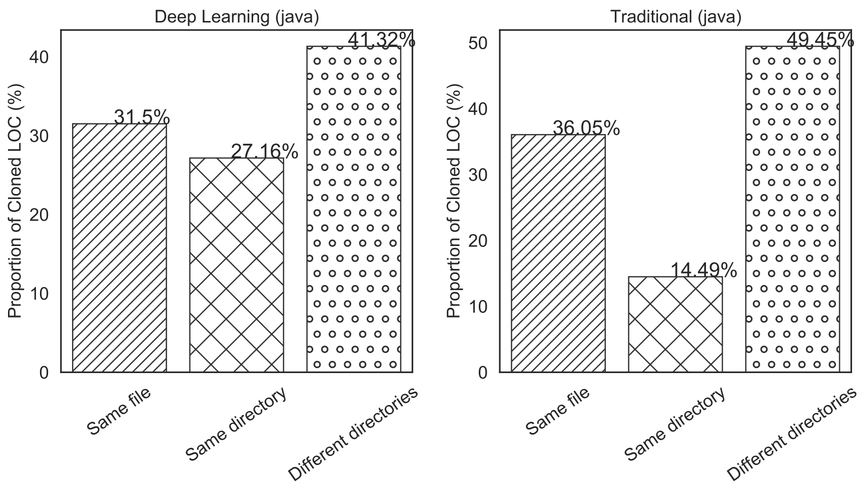

Fig. 13 Percentages of Lines of Code Clones by Location of Clones in both Deep Learning and Traditional Systems (Java)

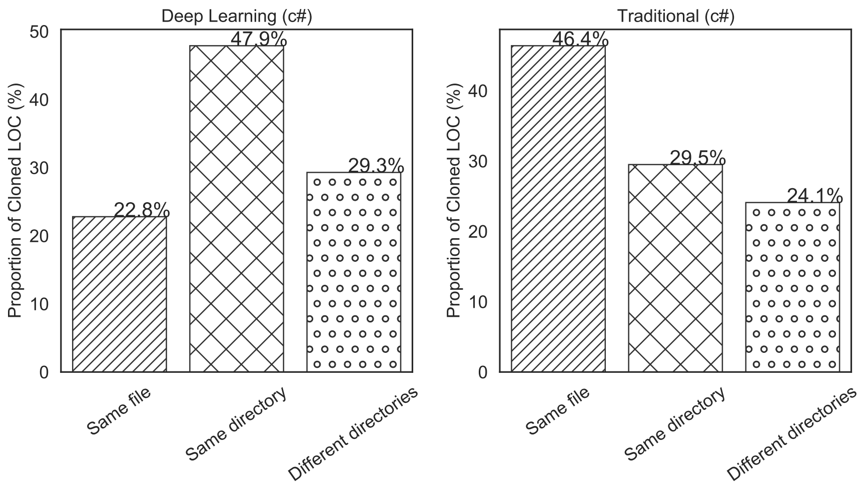

Fig. 14 Percentages of Lines of Code Clones by Location of Clones in both Deep Learning and Traditional Systems (C#)

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 Fig. 12, 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 Python deep learning code are more dispersed than code clones in non-DL systems. For Java as in Fig. 13, the highest proportion of clones for both DL (41.32%) and traditional (49.45%) systems are in 'different directories' contrasting to Python systems (Fig. 12). Although, traditional Java systems have higher proportion of clones in different directories compared to Java DL systems, DL systems have 68.48% (41.32 + 27.16) of clones in 'same directory' and 'different directories' combined, compared to that of traditional Java code (49.45+14.49= 63.94%). Thus, clones in Java DL systems are relatively more dispersed compared to clones in traditional systems. For C# systems as in Fig. 14, 77.2% (47.9 + 29.3) of cloned code is in 'same directory' and 'different directories' compared to 53.6% (29.5 + 24.1) for traditional. This shows that clones in C# DL systems are more dispersed compared to traditional cloned code.

For our location-based analysis of the distribution of clones, we observe that clones in DL systems are more dispersed compared to traditional clones. We also observe that clones in Java and C# DL systems are more dispersed compared to Python DL systems. This dispersion of cloned code in the deep learning systems may harm the maintenance of duplicated code due to potential navigation and comprehension overhead. The degree of dispersion of clones may also vary depending on programming languages.

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. 15-A, the median of the distribution of the percentage of Type 1 cloned lines in 'same file' in Python 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 Python 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 12, we highlight in bold the statistically significant differences for the distributions of cloned lines in different clone locations for both DL and traditional Python code with respect to clone types where p-values are < 0.05. Fig. 15 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 Python 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 proportion of Type 1 clones in non-DL code shows a statistically significant difference only between the 'different directories' location and the others locations. The distribution of lines of code clones located in 'different directories' is lower in comparison to the distribution of line of code clones in other locations (Same file (SF), and Same Directory (SD)).

The existence of exact clones (Type 1) in the same directory could shed lights on some implementation practices of deep learning developers in Python. 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. One example is shown in Table 15, where we found in the same directory 'inference', two models Movidius and Yolo that contains the exact function to calculate the accuracy of the model 'iou' which stands for Intersection Over Union for object detection. 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.

Type 2: As shown in Fig. 15-B, the distribution of the percentage of the lines of Type 2 cloned code in 'same file' in Python 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 Python DL code reside in closer proximity. For traditional systems, we see a similar distribution of the percentage of Type 2 cloned lines in different location categories. However, the percentages of Type 2 clones in 'different directories' in Python DL code tend to be slightly higher compared to that in traditional code.

We found these differences between the percentages of cloned lines in 'same file' and in both 'same directory' and 'different directories' categories for Type 2 clones in Python DL code statistically significant with p-values equal to 7.64e-09 and 5.98e-08, respectively (Table 12). The values of effect size of these differences are large between 'same file' and 'same directory' (0.76) and 'same file' and 'different directories' (0.77).

Hence, Type 2 clones are less dispersed in Python deep learning code than other types of clones. We notice a similar level of dispersion for Type 2 clones in non-DL Python code (as shown in Fig. 15-B).

Type 3: As shown in Fig. 15-C, Type 3 clones are less dispersed compared to Type 1 clones but comparatively more dispersed than Type 2 clones in Python deep learning code. We present the results of the MWW tests in Table 12 where the 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 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

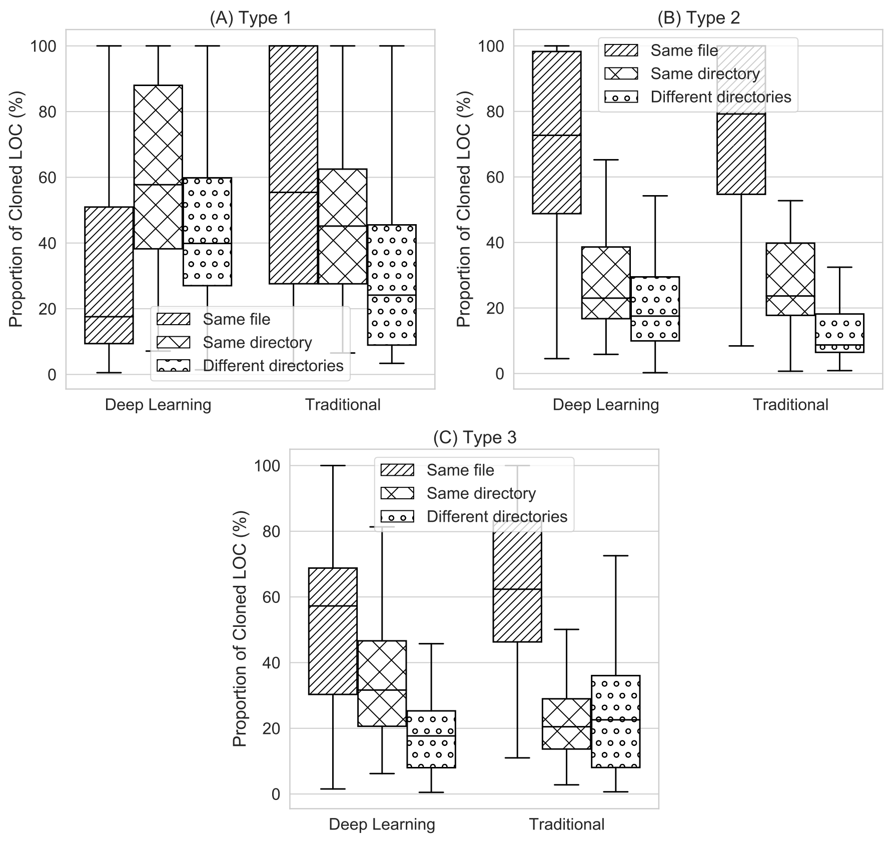

Fig. 15 Distribution of Different Types of Clones by Clone Location in DL and Traditional Code (Python)

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 12.

From our clone-type centric analysis of the distribution of cloned code in Java (Fig. 33) and C# (Fig. 34) systems in the Appendix C, Type 1 clones in DL systems are dispersed with major proportion of cloned code in 'same directory' and 'different directories' location categories than the proportion of cloned code in 'same file'. Type 2 clones on the other hand, tend to reside in 'same file' having higher median value of the percentages of cloned code in both DL and traditional systems. The median values for the percentage of Type 2 cloned code is slightly higher in DL systems compared to the traditional code. For Type 3 clones in Traditional Java and C# systems, the percentages of cloned code in 'same file' category is higher compared to the percentages of clones in 'same directory' and 'different directories'. Java and C# Traditional

Table 15 Clone Codes Example where the Location is in the Same Directory and Type 1 Clone

```
path: BerryNet/inference/movidius.py
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)
```

```
path: BerryNet/inference/yoloutils.py
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)
```

systems also have higher percentage of Type 3 clones in 'same file' compared to DL systems. This indicates that Type 3 clones in Java and C# DL systems are relatively more dispersed compared to traditional systems. Although we observe clones tend to be relatively dispersed in DL systems compared to traditional code, we do not observe statistically significant differences in the location based distribution of clones in most cases for Java systems. For traditional C# systems, we observe the differences between proportion of clones in 'same file' and in 'same directory' or 'different directories' to be statistically significant (except for Type 1).

From this analysis of the distribution of clones across the different files and directories of the studied projects, we can conclude that clones in deep learning code is more dispersed compared to clones in traditional code, although the differences in distribution may vary. 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 cloned 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 few 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 16 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 in Python systems. We observe that the proportion of clone fragments tend to be higher in the 'same file' location category. We found a statistically significant

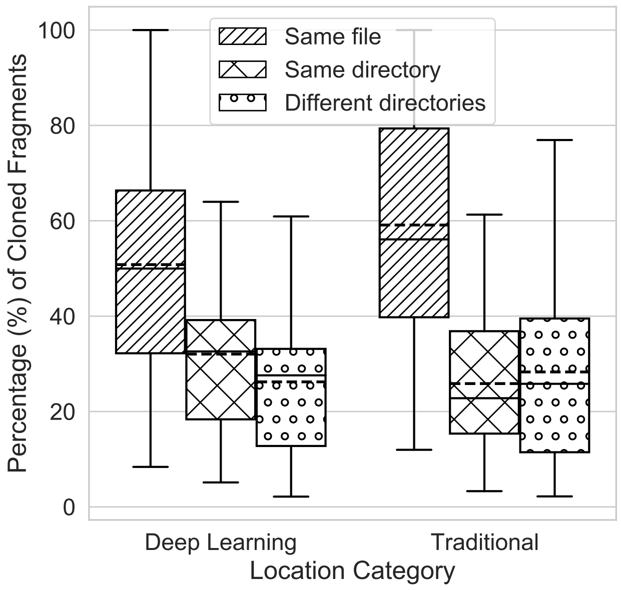

Fig. 16 Distribution of Percentage of Number of Fragments of Code Clones Classes per Clone Location in Python Systems.

difference between the distribution of the proportion of clone fragments located in the 'same file' and in both 'same directory' and 'different directories'. Their p-values are 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.

The mean value of the percentages of code fragments that belong to the same directory category in Python 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. 20-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. 20) is similar to the distributions of the percentages of lines of cloned code (Fig. 15).

For further insights, we analyze the proportion of the number of clones fragments in each clone class by their locations in deep learning and traditional code. As shown in Fig. 17 for Python systems, we identify that the distribution of the number of fragments in each clone class is of 46.7% 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 found to be located in the same file with a percentage

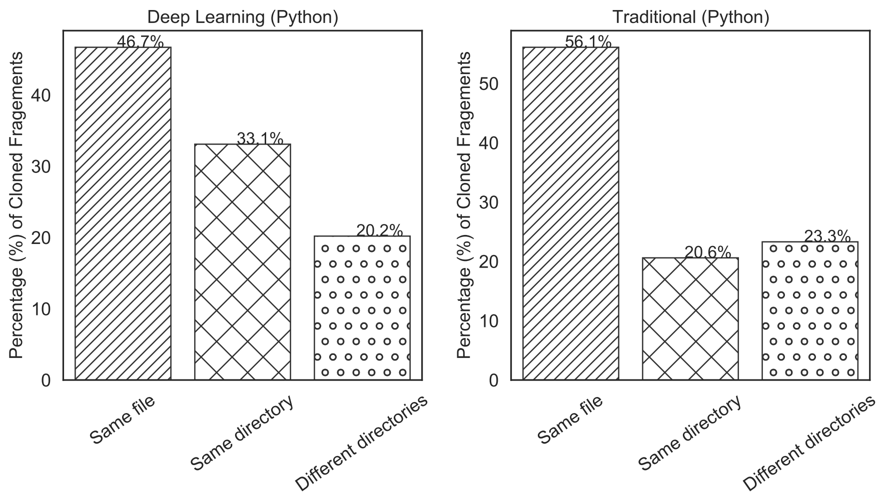

Fig. 17 Percentages of Average Number of Fragments of Code Clones by Location of Clones in both Deep Learning and Traditional Python Systems

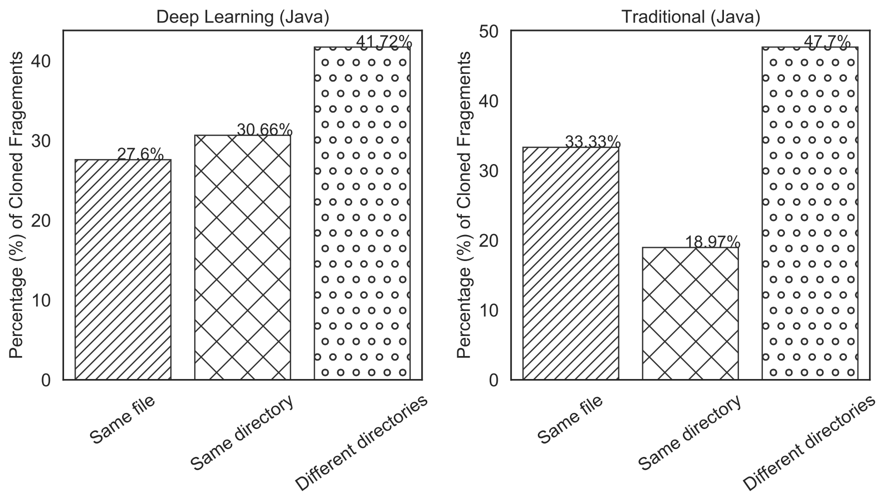

Fig. 18 Percentages of Average Number of Fragments of Code Clones by Location of Clones in both Deep Learning and Traditional Java Systems

equals to 46.7%, which is nearly the same proportion as it is located in different files (i.e, in the same directory or in different directories (33.1%+20.2%)). However, fragments of clones in Python deep learning code are relatively more dispersed compared to fragments of clones in traditional system as traditional systems have higher proportion of clones fragments in 'same file' compared to deep learning code as shown in Fig. 17. For Java systems (Fig. 19), the major proportion of clone fragments in both DL and traditional code are in 'same

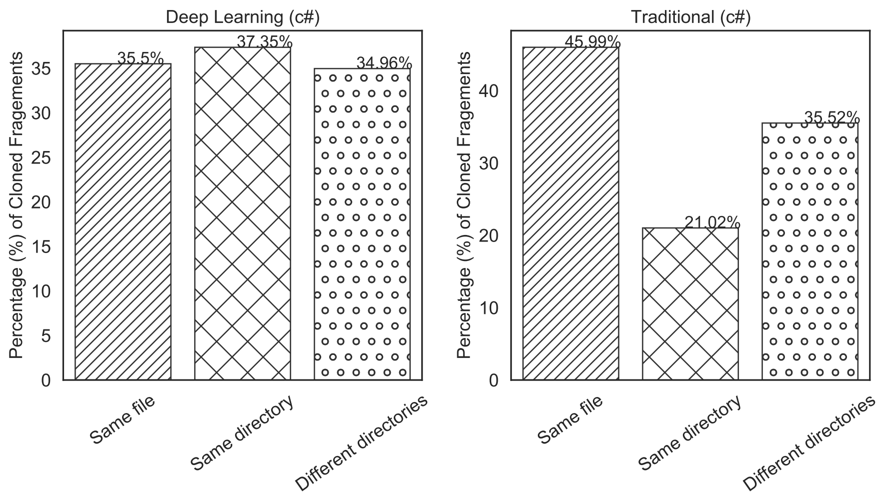

Fig. 19 Percentages of Average Number of Fragments of Code Clones by Location of Clones in both Deep Learning and Traditional C# Systems

directory' and 'different directories' containing 72.38% (30.66 + 41.72) and 66.67% (18.97 + 47.70) of cloned fragments respectively. So, clone fragments in Java DL systems are more dispersed compared to the clones in traditional Java systems. In Fig. 19 showing the distribution of cloned fragments in C# systems, the propertion of clones in DL systems and traditional systems are 72.31% (37.35+34.96) and 56.54% (21.02 + 35.52) respectively. This also indicates that for C#, clone fragments in DL code is more dispersed compared to traditional systems similar to Python and Java systems. This higher dispersion clone fragments in the deep learning code likely to have negative consequences on maintenance due navigation and comprehension overhead for distant code.

We also repeated the analysis for RQ2 with 20% dissimilarity threshold for NiCad clone detector. However, the overall trends in the distribution of clones in different location categories remain the same as shown by our results as shown in Appendix C.

Qualitative analysis of the clones in different locations category: We manually examined the clones contained 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 16. 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,

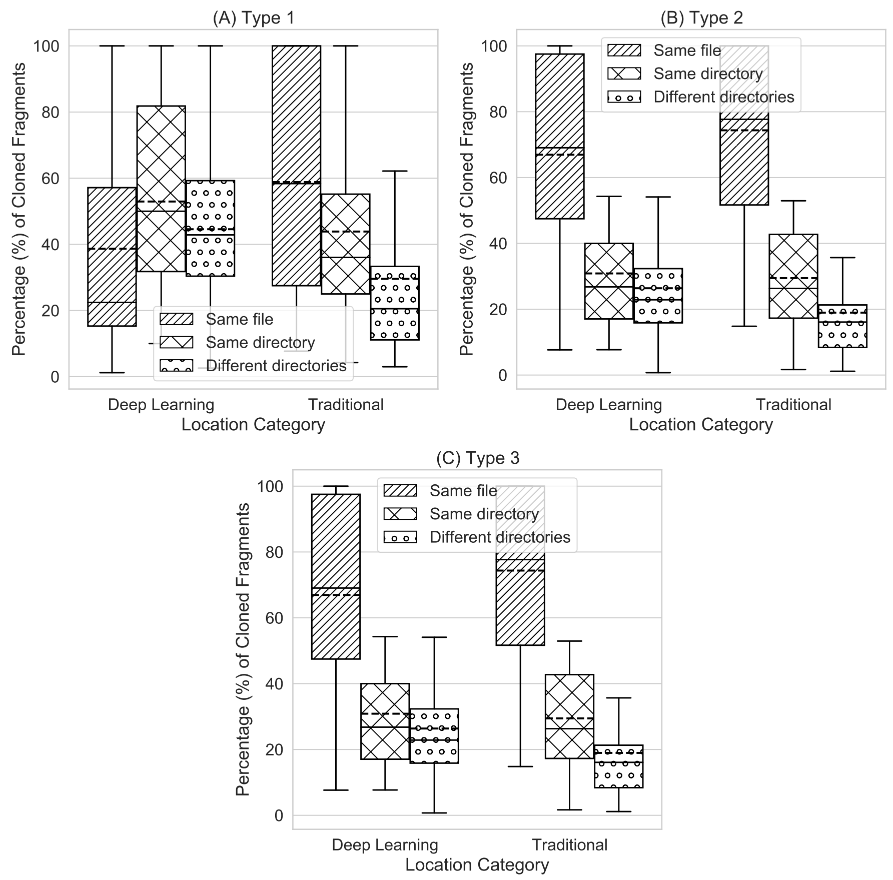

Fig. 20 Distribution of Percentage of Number of Fragments of Code Clones Classes per Clone Location.

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 [43] 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. 15-B.

Table 16 represents examples of cloned code where the location is in the same file. Through a manual analysis, we tried to understand where DL developers duplicate code regarding DL phases perspectives. It was found that 40 code clones classes were in the same file and were corresponding to the DL phase. 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

Table 16 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,
  )
```

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 16, 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. 15-B). Our findings also confirm the results of previous work [43].

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 [43]. In our case and as shown in Figure 15-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 a manual examination of all of the code clones (85) that exist in the same directory but in different files, 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 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 all the code clones fragments (102) that were located in different directories 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 in terms of the distribution of code clones. While for 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.

4.3 **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 [99]. Several previous studies in traditional code highlighted the negative impacts of code clones on the maintenance and comprehension of code. Barbour et al. [5] 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 the co-occurrences of clones and bugs in deep learning code, 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 3.2.6). 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 project. 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 Python 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. When we do similar analysis on bug-fix commits in C# and Java DL systems, we found that **27.54%** are related to clones in C# DL systems and **45.31%** are related to clones in Java DL systems. We perform MWW tests for the distributions of code clones and non-clone code in the Python DL 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. However, the degree of bug-proneness may vary with systems of different programming languages, as we observe in our results.

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 21, 22 and 23 show the percentages of clones from different types (Type 1, Type 2, and Type 3) that are related to bugs for Python, C# and Java DL systems respectively.

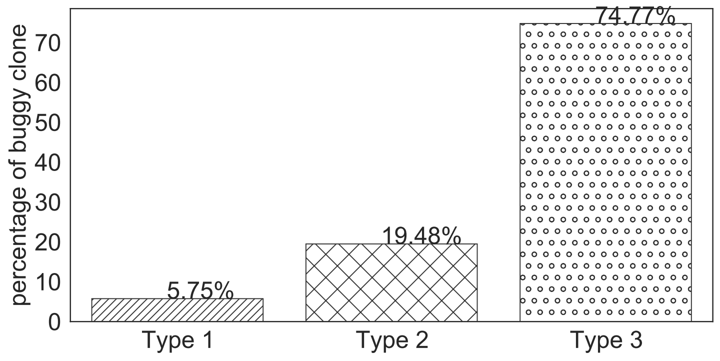

Fig. 21 Buggy Code Clones Occurrences by Clone Type for Python DL systems

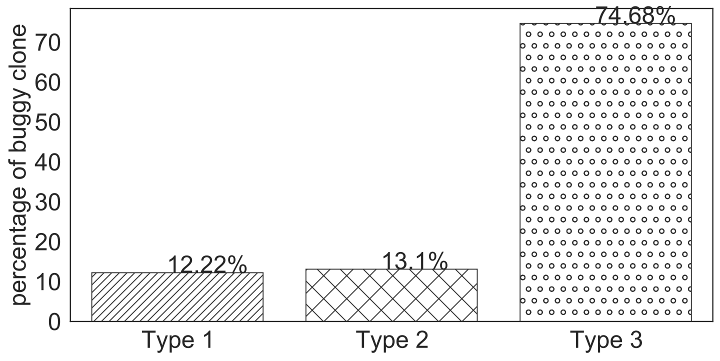

Fig. 22 Buggy Code Clones Occurrences by Clone Type for C# DL systems

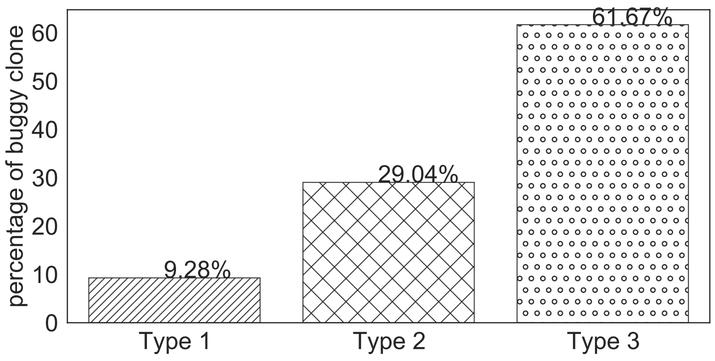

Fig. 23 Buggy Code Clones Occurrences by Clone Type for Java DL systems

We find that Type 3 clones are the most likely to be buggy in all types of DL systems as **74.77%, 74.68%** and **61.67%** of clone related bugs are Type 3 clones for Python, C# and Java DL projects. Then, we have Type 2 clones with a percentage of **19.48%, 13.1%** and **29.04%** for Python, C# and Java respectively, and finally Type 1 clones with a percentage of **5.75%, 12.22%** and **9.28%** clones related to bugs. These results obtained from 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 [65].

Since Type 3 clones are higher in density (Fig. 6) 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 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 24, 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. 6) 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. 21) despite the percentages of buggy clone fragments in Type 3 not being higher than that of Type 2 clones.

Figure 25 and 26 shows the trends for C# and Java DL systems. We find that 0.73% and 0.16% clone fragments are buggy in type 1 clones, 0.63% and 0.27% are buggy in Type 2 clones and 0.72% and 0.28% are buggy in Type 3 clones for C# and Java DL systems respectively. The highest percentage is from Type 3 for Java and Type 1 for C#.

Our results show that clones in deep learning systems are more related to bugs compared to non-cloned code. This observed bug-proneness of clones in DL code is likely to be related to different confounding factors such as relative

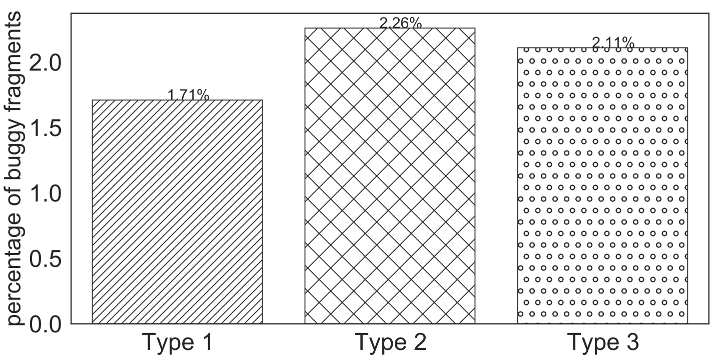

Fig. 24 Percentages of Buggy Code Fragments by Clone Type for python DL systems

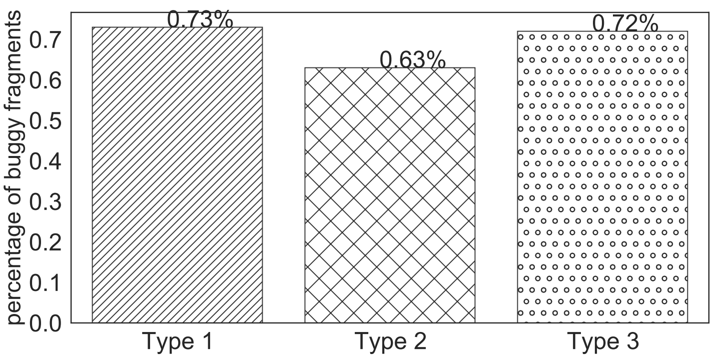

Fig. 25 Percentages of Buggy Code Fragments by Clone Type for C# DL systems

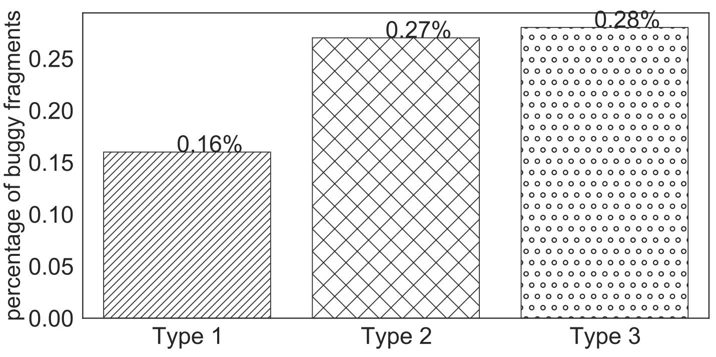

Fig. 26 Percentages of Buggy Code Fragments by Clone Type for Java DL systems

size and distribution of cloned and non-cloned code, clone types, programming language, clone detection tool and different configuration settings, etc. We carefully considered these confounding factors to avoid potential threats. As size of the clones and non-cloned is known to be an important confounding factor for empirical analysis of clones [86], we investigated the relative size of the cloned and non-cloned code. We observed that on an average 27.45% of code in the DL systems are cloned code. We also compared the relative size of the cloned and non-cloned functions. NiCad extracts all functions in the system as candidate clone list for clone detection. We used those extracted functions to separate cloned and non-cloned functions for our analysis by comparing to the clone detection results. We then analyzed the comparative size distribution of cloned and non-cloned functions. As shown in the Appendix D (Fig. 41, Fig. 42, Fig. 43), we do not observe any statistically significant difference between the size of the cloned and non-cloned functions. However, non-cloned functions tend to be larger in size compared to cloned functions as can be seen on the median values of their distributions. The differences between the sizes of cloned and non-cloned functions are not statistically significant in DL and traditional code, and we observe a similar trend for programming language variations (Fig. 43).

Now, we 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 in code clones is more than when bugs are not in code 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 clone related.

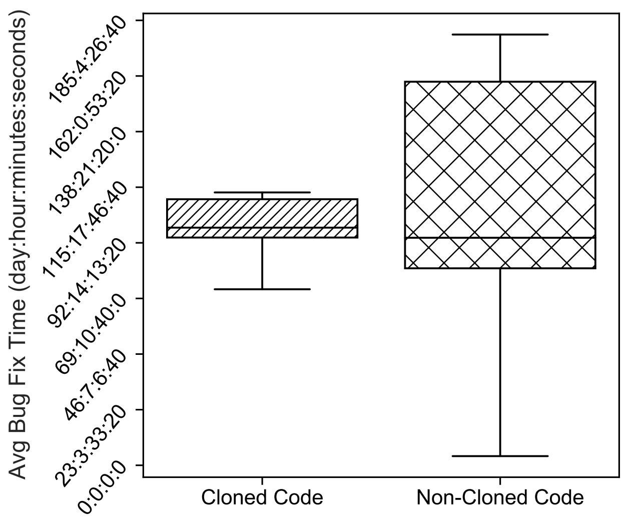

Fig. 27 Comparative Bug-Fix Times for Cloned and non-Cloned Code in Python DL Systems.

Figure 27 shows the distribution of the average times to fix bugs for python DL systems 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 27 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 134.0 days, 6.0 hours, 13.0 minutes and 20.0 seconds with a STD of 95.0 days, 22.0 hours, 46.0 minutes and 40.0 seconds, and when it comes to non cloned code, the mean time to fix a bug is 122.0 days, 16.0 hours, 26.0 minutes and 40.0 seconds with a STD of 96.0 days 15.0 hours 26.0 minutes and 40.0 seconds. Buggy cloned code seems to be taking comparatively more 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).

Similarly, the average bug fix time (mean=77 days) for cloned-codes in C# DL systems is higher than non-cloned code (mean=51 days). ConvNetSharp DL system has average bug fix time of 132 days for cloned code and 89 days for non cloned code while NeuralNetwork.NET DL system has average bug fix time of 22 days for cloned code and 13 days for non cloned code. On the other hand, the average bug fix time of cloned-codes in Java DL systems (mean=63 days) is slightly lower than that of non-cloned codes (mean=64 days). The knime-deeplearning Java DL system has average bug fix time of 105 days for clone-code and 102 days for non-clone code while Neuralnetworks DL system has average bug-fix time of 21 for cloned code and 26 for non-cloned code.

We further check what percentages of DL systems have higher bug-fix time for clones and what percentages of systems have the opposite. Among 10 DL subject systems, we observe in seven DL systems that the bug-fix time for clones is higher and for the rest (i.e., three systems), the bug-fix time for clones is lower. Overall, we can conclude that in a majority of cases (70%), 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): According to our results, we find that cloned code is likely to 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.

4.4 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 3.2.5).

Table 17 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 17. Also, the DL phase subcategories are not exclusive, since functions may be used in tasks related to different phases.

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 clones occurrences ratio 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 17 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 18. 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.

Table 17 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 subcat**	**% 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

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. Input processing includes all the transformation needed to apply on the input data to prepare data for model training (e.g., processing input of model inception v3 by normalizing each pixel of input image).

Table 18 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)
```

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 19 [2], 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.

Data post-processing: Data is often post-processed after an inductive process and converted into a format recommended by the stakeholders of the model and to satisfy the requirement of the application using 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. For example, in object detection the object localization is used to interpret the output by assigning each object to a class with a higher probability or by drawing bounding boxes on an image from inference results. 37.5% of

[2] https://github.com/tensorflow/ranking

Table 19 Example of Model Evaluation (Compute Metrics) Type 3 Clone

```python
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)
```

```python
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)
```

cloned functions are classified as processing output and the rest are found duplicated for computing 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 class labels data (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.

Model tuning: We found clones in the code used to tune different model configurations (e.g., best batch size to train the model) or hyperparameters. 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 development. Code related to the initialization of model hyperparameters are cloned most frequently, followed by code related to model training and data preprocessing.

4.5 **RQ5: In which phases of deep learning development code cloning is more prone to faults?**

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 has the highest risk of bugs. Our results of RQ3 show that code cloning can lead to bugs. A better understanding of the circumstances in which bugs frequently occur on cloned code will help raise the awareness of the developers to understand and mitigate the risks associated with their code cloning activities.

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 consider a clone fragment as 'buggy' when cloned lines intersect with lines modified by bug-fixing commit. We use the labelling of clones as in RQ4 to group the clones regarding the DL phases the clones are related. We computed the percentages of bugs 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 buggy as shown in Figure 28.

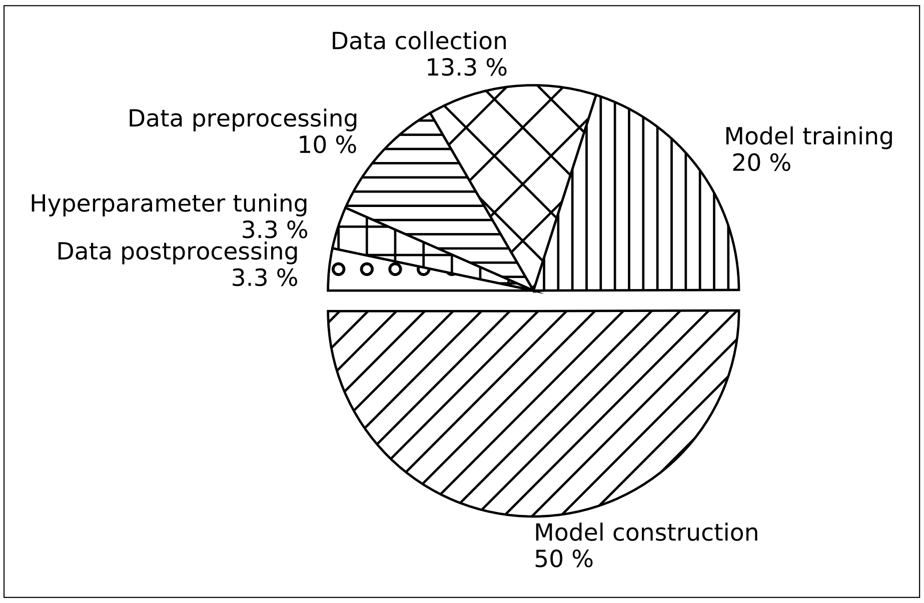

Fig. 28 Percentage of Bug-Fix Occurred with Cloned Functions with respect to Deep Learning Phases

Table 21 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 manually analyzed clone data). The majority of them are related to model (hyper)parameters initialization (46.66%). Table 20 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'[3]. The optimizers in deep learning are capable of reducing the losses by changing the

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

Table 20 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
```

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

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.

Table 21 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	

5 Research Implications

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

Code clones are prevalent 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 a code that works often seem 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 [52, 53] 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 [49] 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 [15, 70, 101].

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 [67] 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.

6 Threats to Validity

In this section, we discuss the potential threats to the validity of our research methodology and 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 two authors having academic and industry background. The results were then cross-validated, and disagreements were resolved by group discussion. We believe this 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. [79] 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-fix commit detection process, we reviewed a sample of the labeled commits (102) and confirmed that they corresponded to bug-fixes with high accuracy (100%). We applied SZZ algorithm to identify bug-inducing commits for corresponding bug-fix commits by locating the lines of code that induced the bug by tracing back from the lines changed by the bug-fixing commit. However, the accuracy of the SZZ is likely to be a threat. We randomly sampled 70 bug-inducing commits (from our systems) that were detected by the SZZ algorithm. Then we checked them manually and found only two (2.8%) false-positives. Thus, the impact of the SZZ algorithm on our findings might be insignificant. However, further study is needed to confirm the accuracy of the SZZ algorithm.

For detecting clones, we used the NiCad clone detector [19]. Since different settings can have different effects, that we call a confounding configuration problem [100], we have carefully set the parameters of NiCad by employing a standard configuration [81] and with these settings, NiCad is reported to be very accurate in clone detection [81,82]. Thus, we believe that our findings on code clones in deep learning code have relevance significance. We also repeated our analysis for varying dissimilarity thresholds (20% and 30%) for clone detection. Again, clone fragments can be of different sizes. The accuracy of the clone detection tool may vary across clones of different sizes. Although we are not aware of clone size related bias of NiCad, it might be a potential threat and warrants further empirical investigations.

With regard to **external validity**, our analysis is primarily focused on deep learning repositories that are written in Python with small number of Java and C# systems. As Python is the most popular 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 six DL repositories to be manually analyzed for the creation of the taxonomy. Therefore, this may threaten the generalizability of our results. We believe that our small scale but detail analysis 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 [39].

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.

7 Related Work

In this section, we discuss relevant studies from the literature 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. In addition, we examine the impact of code clones on traditional systems quality.

7.1 Software Engineering for AI-based system

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. [89] 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 [2] 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 [11]. This is a consequence of the heavy dependence of ML on data and models. Breck et al. [13] 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 focuses more on model quality rather than the infrastructure quality of machine learning systems. Zhang et al. [104] provide a comprehensive survey of ML testing covering 138 papers. The study of Zhang et al. [104] 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, test assessment criteria, the oracle problem, and testing cost reduction. Furthermore, Zhang et al. [104] 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 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. [2] performed a survey of Software Engineering practices for ML-based systems at Microsoft. They interviewed developers from Microsoft 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. [105]. Zhang et al. also surveyed DL practitioners about their software engineering practices. They proposed recommendations to improve the development process of DL applications. Wan et al.[99] 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. [16] 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. [37] analyzed stack overflow posts and 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. [106] studied deep learning applications built on top of TensorFlow [76] 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: Jebnoun et al. [40] studied the prevalence, evolution, and bug proneness of code smells in deep learning applications. They identify three types of smells that occur more frequently: long lambda functions, long ternary conditional expression, and complex container comprehension. They report that the number of code smells increases across releases. They focused on the relationship between code smells and bugs reported the overlap of nearly 63% of the changed file with files that contain code smells, and that frequent smells co-exist with software bugs more frequently than the others. Jiakun Liu et al [57] 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 [57] 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. [45] 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 [91]. They [45] 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. [71] 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. [72] 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. [48] 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.

7.2 Impacts of Code Clones

Since we investigate code clones in the deep learning code, we are interested in reviewing the existing literature on the impact of code clones in traditional software systems.

Roy and Cordy [83] 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. [87] showed that, contrary to intuition, the cloned code contains less problematic patterns than non-cloned code. Along the same line, Rahman et al. [73] 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. [35] 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. [36] 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. [4] 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. [24], 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 [50] 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. [41] 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 [6,7,42,56,54,75,98] report a similar conclusion about code clones, i.e., that they make the code bug-prone and increase maintenance costs. Juergens et al. [42] 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. [25] provide empirical evidences showing that unintentional inconsistencies of code clones leads to faults.

Barbour et al. [6] 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 riskiest pattern. Researchers have also examined the maintenance efforts that result from duplicating code. Hotta et al. [34] 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 [44] 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. [46] who suggest that refactoring techniques cannot tackle consistently changing code clones. Li et al [56] advise that maintaining duplicate code would be very beneficial for developers, as this would avoid introducing hard to detect bugs. Lozano and Wermelinger [59] 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 [58] also show that the existence of code clones may increase the change effort.

A recent study by Mondal et al. [62] shows that cloned code are more unstable than non-cloned code in general. However according to Selim et al. [90], the bug-proneness of code clones might be system dependent. Mondal et al. empirical study [66] 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. 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 [45,48] 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 paper, 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 has the highest risk of bugs.

8 Conclusion

This paper 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 (hyper)parameters 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.

As future work, 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.

Acknowledgements This work is supported by Fonds de Recherche du Quebec (FRQ) and the Natural Sciences and Engineering Research Council of Canada (NSERC). We would like to thank Dr. Amin Nikanjam for his valuable comments on the manuscript.

References

1. Al Dallal, J., Abdin, A.: Empirical evaluation of the impact of object-oriented code refactoring on quality attributes: A systematic literature review. IEEE Transactions on Software Engineering **44**(1), 44–69 (2017)
2. Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., Nagappan, N., Nushi, B., Zimmermann, T.: Software engineering for machine learning: A case study. In: 2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP), pp. 291–300. IEEE (2019)
3. Anwar, H., Pfahl, D., Srirama, S.N.: Evaluating the impact of code smell refactoring on the energy consumption of android applications. In: 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 82–86 (2019). DOI 10.1109/SEAA.2019.00021
4. Aversano, L., Cerulo, L., Di Penta, M.: How clones are maintained: An empirical study. In: 11th European Conference on Software Maintenance and Reengineering (CSMR'07), pp. 81–90. IEEE (2007)

5. Barbour, L., An, L., Khomh, F., Zou, Y., Wang, S.: An investigation of the fault-proneness of clone evolutionary patterns. Software Quality Journal **26**(4), 1187–1222 (2018)
6. Barbour, L., Khomh, F., Zou, Y.: Late propagation in software clones. In: 2011 27th IEEE International Conference on Software Maintenance (ICSM), pp. 273–282. IEEE (2011)
7. Barbour, L., Khomh, F., Zou, Y.: An empirical study of faults in late propagation clone genealogies. Journal of Soft. Evol. and Proc **25**, 1139–1165 (2013)
8. Bergstra, J., Bengio, Y.: Random search for hyper-parameter optimization. The Journal of Machine Learning Research **13**(1), 281–305 (2012)
9. Bordes, A., Chopra, S., Weston, J.: Question answering with subgraph embeddings. arXiv preprint arXiv:1406.3676 (2014)
10. Braiek, H.B., Khomh, F.: Deepevolution: A search-based testing approach for deep neural networks. 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME) pp. 454–458 (2019)
11. Braiek, H.B., Khomh, F.: On testing machine learning programs. Journal of Systems and Software **164**, 110542 (2020)
12. Braiek, H.B., Khomh, F., Adams, B.: The open-closed principle of modern machine learning frameworks. In: 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR), pp. 353–363. IEEE (2018)
13. Breck, E., Cai, S., Nielsen, E., Salib, M., Sculley, D.: The ml test score: A rubric for ml production readiness and technical debt reduction. In: 2017 IEEE International Conference on Big Data (Big Data), pp. 1123–1132. IEEE (2017)
14. Buckley, F.J., Poston, R.: Software quality assurance. IEEE Transactions on Software Engineering (1), 36–41 (1984)
15. Chen, B.: Berrynet deep learning gateway on raspberry pi and other edge devices. `https://github.com/DT42/BerryNet` (2019)
16. Chen, Z., Cao, Y., Liu, Y., Wang, H., Xie, T., Liu, X.: Understanding challenges in deploying deep learning based software: An empirical study. arXiv preprint arXiv:2005.00760 (2020)
17. Chen, Z., Chen, L., Ma, W., Zhou, X., Zhou, Y., Xu, B.: Understanding metric-based detectable smells in python software: A comparative study. Information and Software Technology **94**, 14–29 (2018)
18. Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., Kuksa, P.: Natural language processing (almost) from scratch. Journal of machine learning research **12**(ARTICLE), 2493–2537 (2011)
19. Cordy, J.R., Roy, C.K.: The nicad clone detector. In: 2011 IEEE 19th International Conference on Program Comprehension, pp. 219–220. IEEE (2011)
20. Cordy, J.R., Roy, C.K.: NiCad clone detector. `https://www.txl.ca/txl-nicaddownload.html` (2019). [Online; accessed 20-February-2020]
21. Ernst, N.: Cliff's delta implementation. `https://github.com/neilernst/cliffsDelta` (2019)
22. Farabet, C., Couprie, C., Najman, L., LeCun, Y.: Learning hierarchical features for scene labeling. IEEE transactions on pattern analysis and machine intelligence **35**(8), 1915–1929 (2012)
23. Fowler, M., Beck, K., Brant, J., Opdyke, W., Roberts, D.: Refactoring: Improving the design of existing code addison-wesley professional. Berkeley, CA, USA (1999)
24. Gode, N., Harder, J.: Clone stability. In: 2011 15th European Conference on Software Maintenance and Reengineering, pp. 65–74. IEEE (2011)
25. Göde, N., Koschke, R.: Frequency and risks of changes to clones. In: Proceedings of the 33rd International Conference on Software Engineering, pp. 311–320 (2011)
26. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016). `http://www.deeplearningbook.org`
27. Gottschalk, M., Josefiok, M., Jelschen, J., Winter, A.: Removing energy code smells with reengineering services. INFORMATIK 2012 (2012)
28. Gupta, R., Pal, S., Kanade, A., Shevade, S.: Deepfix: Fixing common c language errors by deep learning. In: Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence, pp. 1345–1351 (2017)

29. Hadhemii: DLCodeSmells. `https://github.com/Hadhemii/DLCodeSmells/blob/master/data/dlRepos.csv` (2019)
30. Hamdan, S., Alramouni, S.: A quality framework for software continuous integration. Procedia Manufacturing **3**, 2019–2025 (2015)
31. Han, J., Shihab, E., Wan, Z., Deng, S., Xia, X.: What do programmers discuss about deep learning frameworks. EMPIRICAL SOFTWARE ENGINEERING (2020)
32. Heaton, J.B., Polson, N.G., Witte, J.H.: Deep learning for finance: deep portfolios. Applied Stochastic Models in Business and Industry **33**(1), 3–12 (2017)
33. Hinton, G., Deng, L., Yu, D., Dahl, G.E., Mohamed, A.r., Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T.N., et al.: Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. IEEE Signal processing magazine **29**(6), 82–97 (2012)
34. Hotta, K., Sano, Y., Higo, Y., Kusumoto, S.: 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), pp. 73–82 (2010)
35. Islam, J.F., Mondal, M., Roy, C.K.: Bug replication in code clones: An empirical study. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 1, pp. 68–78. IEEE (2016)
36. Islam, J.F., Mondal, M., Roy, C.K., Schneider, K.A.: A comparative study of software bugs in clone and non-clone code. In: SEKE, pp. 436–443 (2017)
37. Islam, M.J., Nguyen, G., Pan, R., Rajan, H.: A comprehensive study on deep learning bug characteristics. arXiv preprint arXiv:1906.01388 (2019)
38. JEBNOUN, H.: 6dlreposdata. `https://github.com/Hadhemii/ClonesInDLCode/blob/master/data/6DLReposData.csv` (2020)
39. Jebnoun, H.: Clones in deep learning code (2020). URL `https://github.com/Hadhemii/ClonesInDLCode`
40. Jebnoun, H., Ben Braiek, H., Rahman, M.M., Khomh, F.: The scent of deep learning code: An empirical study. In: Proceedings of the 17th International Conference on Mining Software Repositories, pp. 1—-11 (2020)
41. Jiang, L., Su, Z., Chiu, E.: 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, pp. 55–64 (2007)
42. Juergens, E., Deissenboeck, F., Hummel, B., Wagner, S.: Do code clones matter? In: 2009 IEEE 31st International Conference on Software Engineering, pp. 485–495. IEEE (2009)
43. Kapser, C., Godfrey, M.W.: Toward a taxonomy of clones in source code: A case study. Evolution of large scale industrial software architectures **16**, 107–113 (2003)
44. Kapser, C.J., Godfrey, M.W.: "cloning considered harmful" considered harmful: patterns of cloning in software. Empirical Software Engineering **13**(6), 645 (2008)
45. Kery, M.B., Radensky, M., Arya, M., John, B.E., Myers, B.A.: 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, pp. 1–11 (2018)
46. Kim, M., Sazawal, V., Notkin, D., Murphy, G.: 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, pp. 187–196 (2005)
47. Kim, S., Whitehead Jr, E.J.: How long did it take to fix bugs? In: Proceedings of the 2006 international workshop on Mining software repositories, pp. 173–174 (2006)
48. Koenzen, A., Ernst, N., Storey, M.A.: Code duplication and reuse in jupyter notebooks. arXiv preprint arXiv:2005.13709 (2020)
49. Koschke, R.: Survey of research on software clones. In: Dagstuhl Seminar Proceedings. Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2007)
50. Krinke, J.: Is cloned code older than non-cloned code? In: Proceedings of the 5th International Workshop on Software Clones, pp. 28–33 (2011)

51. Kumlander, D.: Towards a new paradigm of software development: an ambassador driven process in distributed software companies. In: Advanced Techniques in Computing Sciences and Software Engineering, pp. 487–490. Springer (2010)
52. LeCun, Y., Boser, B.E., Denker, J.S., Henderson, D., Howard, R.E., Hubbard, W.E., Jackel, L.D.: Handwritten digit recognition with a back-propagation network. In: Advances in neural information processing systems, pp. 396–404 (1990)
53. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE **86**(11), 2278–2324 (1998)
54. Li, J., Ernst, M.D.: CBCD: Cloned buggy code detector. In: Proc. ICSE, pp. 310–320 (2012)
55. Li, X., Jiang, H., Ren, Z., Li, G., Zhang, J.: Deep learning in software engineering. arXiv preprint arXiv:1805.04825 (2018)
56. Li, Z., Lu, S., Myagmar, S., Zhou, Y.: CP-Miner: Finding copy-paste and related bugs in large-scale software code. IEEE TSE **32**, 176–192 (2006)
57. Liu, J., Huang, Q., Xia, X., Shihab, E., Lo, D., Li, S.: 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, pp. 1–10 (2020)
58. Lozano, A., Wermelinger, M.: Assessing the effect of clones on changeability. In: Proc. ICSM, pp. 227–236 (2008)
59. Lozano, A., Wermelinger, M.: Tracking clones' imprint. In: Proceedings of the 4th International Workshop on Software Clones, pp. 65–72 (2010)
60. Macbeth, G., Razumiejczyk, E., Ledesma, R.D.: Cliff's delta calculator: A non-parametric effect size program for two groups of observations. Universitas Psychologica **10**(2), 545–555 (2011)
61. Miotto, R., Wang, F., Wang, S., Jiang, X., Dudley, J.T.: Deep learning for healthcare: review, opportunities and challenges. Briefings in bioinformatics **19**(6), 1236–1246 (2018)
62. Mondal, M., Rahman, M.S., Roy, C.K., Schneider, K.A.: Is cloned code really stable? Empirical Softw. Engg. **23**(2), 693–770 (2018)
63. Mondal, M., Roy, B., Roy, C.K., Schneider, K.A.: Investigating context adaptation bugs in code clones. In: 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 157–168 (2019)
64. Mondal, M., Roy, C.K., Rahman, M.S., Saha, R.K., Krinke, J., Schneider, K.A.: Comparative stability of cloned and non-cloned code: An empirical study. In: Proc. ACM SAC, pp. 1227–1234 (2012)
65. Mondal, M., Roy, C.K., Schneider, K.A.: A comparative study on the bug-proneness of different types of code clones. In: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 91–100 (2015)
66. Mondal, M., Roy, C.K., Schneider, K.A.: Does cloned code increase maintenance effort? In: 2017 IEEE 11th International Workshop on Software Clones (IWSC), pp. 1–7. IEEE (2017)
67. Munappy, A., Bosch, J., Olsson, H.H., Arpteg, A., Brinne, B.: Data management challenges for deep learning. In: 2019 45th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), pp. 140–147. IEEE (2019)
68. Neuhäuser, M.: Wilcoxon–Mann–Whitney Test, pp. 1656–1658. Springer Berlin Heidelberg, Berlin, Heidelberg (2011). DOI 10.1007/978-3-642-04898-2_615. URL `https://doi.org/10.1007/978-3-642-04898-2_615`
69. Nguyen, H., Kieu, L.M., Wen, T., Cai, C.: Deep learning methods in transportation domain: a review. IET Intelligent Transport Systems **12**(9), 998–1004 (2018)
70. Pasumarthi, R.K., Bruch, S., Wang, X., Li, C., Bendersky, M., Najork, M., Pfeifer, J., Golbandi, N., Anil, R., Wolf, S.: Tensorflow ranking. `https://github.com/tensorflow/ranking` (2019)
71. Pimentel, J.F., Murta, L., Braganholo, V., Freire, J.: A large-scale study about quality and reproducibility of jupyter notebooks. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR), pp. 507–517. IEEE (2019)
72. Psallidas, F., Zhu, Y., Karlas, B., Interlandi, M., Floratou, A., Karanasos, K., Wu, W., Zhang, C., Krishnan, S., Curino, C., et al.: Data science through the looking glass and what we found there. arXiv preprint arXiv:1912.09536 (2019)

73. Rahman, F., Bird, C., Devanbu, P.: Clones: What is that smell? Empirical Software Engineering **17**(4-5), 503–530 (2012)
74. Rahman, M.S., Roy, C.K.: A change-type based empirical study on the stability of cloned code. In: Proc. SCAM, pp. 31–40 (2014)
75. Rahman, M.S., Roy, C.K.: On the relationships between stability and bug-proneness of code clones: An empirical study. In: Proc. SCAM, pp. 131–140 (2017)
76. Rampasek, L., Goldenberg, A.: Tensorflow: Biology's gateway to deep learning? Cell systems **2**(1), 12–14 (2016)
77. Redmon, J., Divvala, S., Girshick, R., Farhadi, A.: You only look once: Unified, real-time object detection. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 779–788 (2016)
78. Rochimah, S., Arifiani, S., Insanittaqwa, V.F.: Non-source code refactoring: a systematic literature review. International Journal of Software Engineering and Its Applications **9**(6), 197–214 (2015)
79. Rosen, C., Grawi, B., Shihab, E.: Commit guru: analytics and risk prediction of software commits. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp. 966–969. ACM (2015)
80. Roy, C.K., Cordy, J.R.: A survey on software clone detection research. Queen's School of Computing TR **541**(115), 64–68 (2007)
81. Roy, C.K., Cordy, J.R.: Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization. In: 2008 16th iEEE international conference on program comprehension, pp. 172–181. IEEE (2008)
82. Roy, C.K., Cordy, J.R.: A mutation/injection-based automatic framework for evaluating code clone detection tools. In: 2009 International Conference on Software Testing, Verification, and Validation Workshops, pp. 157–166. IEEE (2009)
83. Roy, C.K., Cordy, J.R.: Near-miss function clones in open source software: an empirical study. Journal of Software Maintenance and Evolution: Research and Practice **22**(3), 165–189 (2010)
84. Roy, C.K., Zibran, M.F., Koschke, R.: The vision of software clone management: Past, present, and future (keynote paper). In: proc. CSMR-WCRE, pp. 18–33 (2014)
85. Sainath, T.N., Mohamed, A.r., Kingsbury, B., Ramabhadran, B.: Deep convolutional neural networks for lvcsr. In: 2013 IEEE international conference on acoustics, speech and signal processing, pp. 8614–8618. IEEE (2013)
86. Saini, V., Sajnani, H., Lopes, C.: Cloned and non-cloned java methods: A comparative study. Empirical Softw. Engg. **23**(4), 2232–2278 (2018). DOI 10.1007/s10664-017-9572-7
87. Sajnani, H., Saini, V., Lopes, C.V.: 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, pp. 21–30. IEEE (2014)
88. Samek, W., Wiegand, T., Müller, K.R.: Explainable artificial intelligence: Understanding, visualizing and interpreting deep learning models. arXiv preprint arXiv:1708.08296 (2017)
89. Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.F., Dennison, D.: Hidden technical debt in machine learning systems. In: C. Cortes, N.D. Lawrence, D.D. Lee, M. Sugiyama, R. Garnett (eds.) Advances in Neural Information Processing Systems 28, pp. 2503–2511. Curran Associates, Inc. (2015). URL `http://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems.pdf`
90. Selim, G.M., Barbour, L., Shang, W., Adams, B., Hassan, A.E., Zou, Y.: Studying the impact of clones on software defects. In: 2010 17th Working Conference on Reverse Engineering, pp. 13–21. IEEE (2010)
91. Shen, H.: Interactive notebooks: Sharing the code. Nature **515**(7525), 151–152 (2014)
92. Spadini, D., Aniche, M., Bacchelli, A.: PyDriller: Python Framework for Mining Software Repositories (2018). DOI 10.1145/3236024.3264598
93. Suryanarayana, G., Samarthyam, G., Sharma, T.: Refactoring for software design smells: managing technical debt. Morgan Kaufmann (2014)
94. Svajlenko, J., Roy, C.K.: Evaluating modern clone detection tools. In: 2014 IEEE International Conference on Software Maintenance and Evolution, pp. 321–330. IEEE (2014)

95. Szegedy, C., Vanhoucke, V., Ioffe, S., Shlens, J., Wojna, Z.: Rethinking the inception architecture for computer vision. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp. 2818–2826 (2016)
96. Tompson, J.J., Jain, A., LeCun, Y., Bregler, C.: Joint training of a convolutional network and a graphical model for human pose estimation. In: Advances in neural information processing systems, pp. 1799–1807 (2014)
97. Vetro, A., Ardito, L., Morisio, M.: Definition, implementation and validation of energy code smells: an exploratory study on an embedded system (2013)
98. Wagner, S., Abdulkhaleq, A., Kaya, K., Paar, A.: On the relationship of inconsistent software clones and faults: An empirical study. In: Proc. SANER, pp. 79–89 (2016)
99. Wan, Z., Xia, X., Lo, D., Murphy, G.C.: How does machine learning change software development practices? IEEE Transactions on Software Engineering (2019)
100. Wang, T., Harman, M., Jia, Y., Krinke, J.: Searching for better configurations: a rigorous approach to clone evaluation. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering, pp. 455–465 (2013)
101. Weill, C., Gonzalvo, J., Kuznetsov, V., Yang, S., Yak, S., Mazzawi, H., Hotaj, E., Jerfel, G., Macko, V., Adlam, B., Mohri, M., Cortes, C.: Adanet. `https://github.com/tensorflow/adanet` (2019)
102. Wheeler, D.A.: SLOCCount. `https://dwheeler.com/sloccount/` (2004). [Online; accessed 19-May-2020]
103. White, M., Tufano, M., Vendome, C., Poshyvanyk, D.: Deep learning code fragments for code clone detection. In: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 87–98. IEEE (2016)
104. Zhang, J.M., Harman, M., Ma, L., Liu, Y.: Machine learning testing: Survey, landscapes and horizons. IEEE Transactions on Software Engineering (2020)
105. Zhang, X., Yang, Y., Feng, Y., Chen, Z.: Software engineering practice in the development of deep learning applications. arXiv preprint arXiv:1910.03156 (2019)
106. Zhang, Y., Chen, Y., Cheung, S.C., Xiong, Y., Zhang, L.: An empirical study on tensorflow program bugs. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 129–140. ACM (2018)

Appendix A Study Design

Table 22 shows the name, url, number of lines of code (SLOC), number of commits and the size of each selected 6 DL repository.

Table 22 The 6 analyzed DL repositories details

Repository	**URL**	**SLOC**	**#commits**	**Size**
keras-applications	https://github.com/keras-team/keras-applications.git	4263	89	small
DeepCTR	https://github.com/shenweichen/DeepCTR.git	4886	91	small
nn-wtf	https://github.com/lene/nn-wtf.git	2688	119	small
ranking	https://github.com/tensorflow/ranking.git	10778	145	medium
BerryNet	https://github.com/DT42/BerryNet.git	12963	396	medium
adanet	https://github.com/tensorflow/adanet.git	25165	432	large

Appendix B RQ1 Additional Results

B.1 Results of Clone Detection Using Threshold of 20%

In this section, we provide additional results for both programming languages (Java and C#) when using a dissimilarity threshold 20% in order to explore the impact of threshold on clone detection as we use in our analysis 30% as threshold. Figure 29 shows the code clones occurrences in DL and Traditional Java projects for both code clones granularities. Figure 30 shows the same analysis but for C# projects.

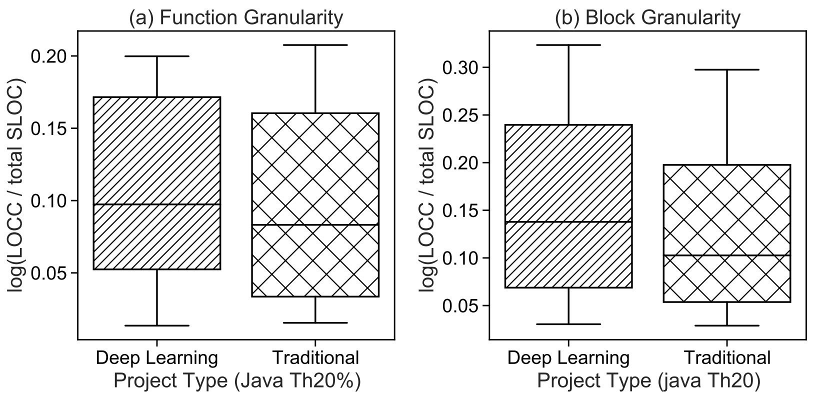

Fig. 29 Code Clones Occurrences in DL and Traditional Java Projects Using threshold as 20% for Both Code Clones Granularities: (a) Function, (b) Block. ***LOCC***: Lines Of Code Clones, ***SLOC***: Source Lines Of Code.

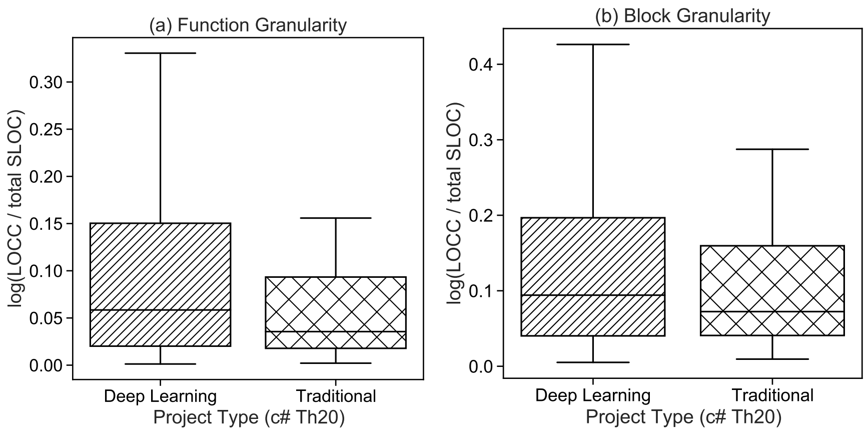

Fig. 30 Code Clones Occurrences in DL and Traditional C# Projects for Both Code Clones Granularities: (a) Function, (b) Block and using 20% as threshold. ***LOCC***: Lines Of Code Clones, ***SLOC***: Source Lines Of Code.

We further extend our analysis by comparing clones types. Figure 31 and Figure 32 illustrate the clone density in DL and traditional projects for clone types and granularity for the two programming languages (Java and C# respectively).

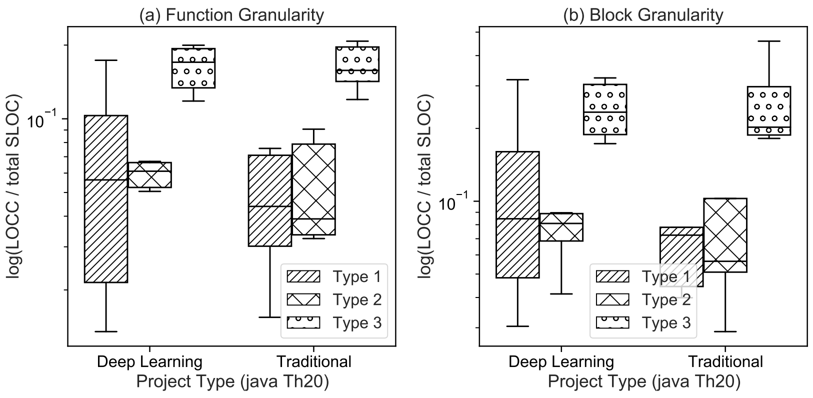

Fig. 31 Clone Density in DL and Traditional Java Projects for Clone Types and Granularity Using threshold as 20% ***LOCC:*** *Lines Of Code Clones*

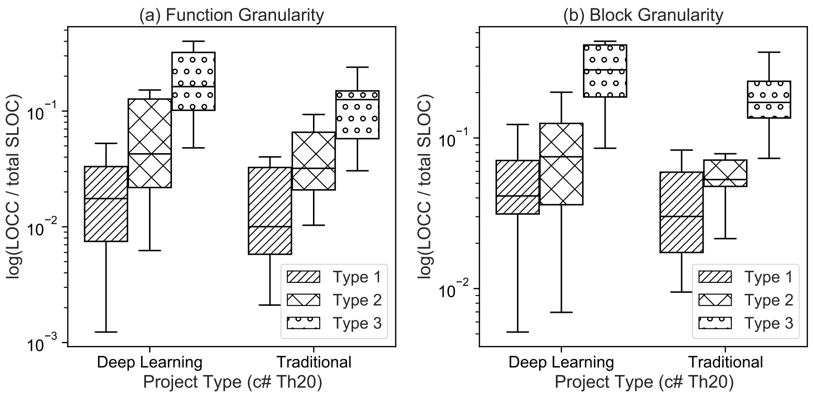

Fig. 32 Clone Density in DL and Traditional C# Projects for Clone Types and Granularity and using 20% as threshold. ***LOCC:*** *Lines Of Code Clones*

Appendix C RQ2 Additional Results

C.1 Other Programming Languages Analysis Results

We study the distribution of different clones types by clone location in DL and traditional code in java projects (Figure 33) and in C# projects (Figure 34)

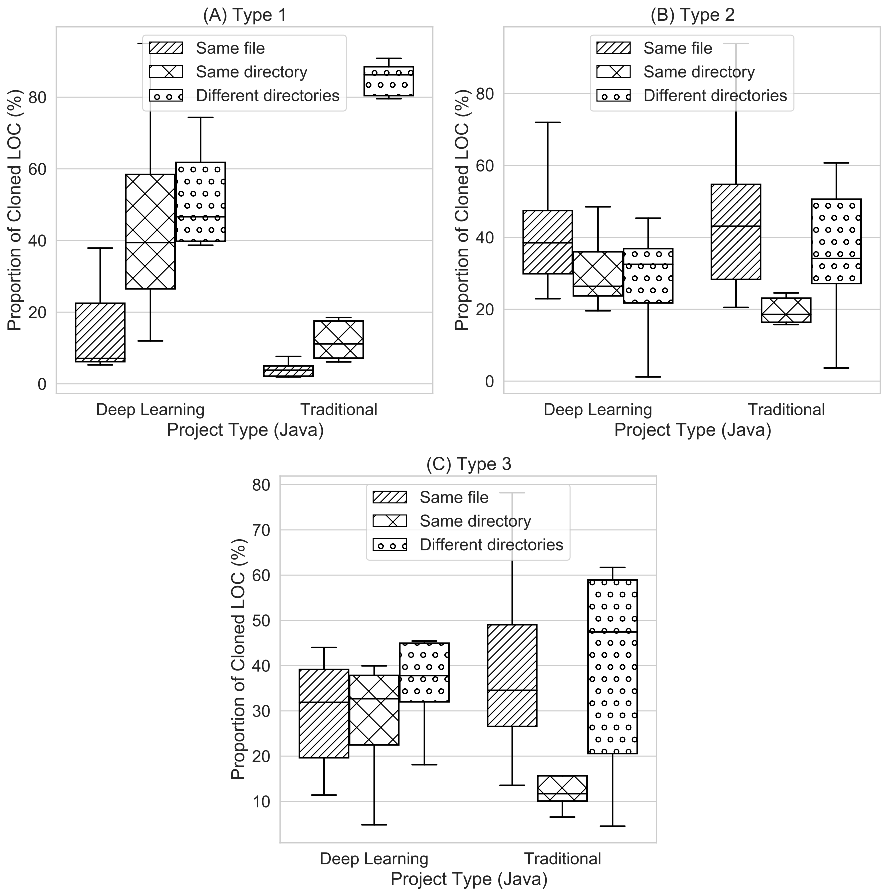

Fig. 33 Distribution of Different Types of Clones by Clone Location in DL and Traditional Code (Java)

C.2 Results of Clone Detection Using Threshold of 20%

In this section, we present the additional analysis we performed to address RQ2. We examine the code clone distribution by location in DL and traditional java (Figure 35) and C# (Figure 36) systems using 20% as dissimilarity threshold.

We further study the percentages of average number of fragments of code clones by location of clones in both deep learning and traditional using 20% dissimilarity threshold for the two programming language Java (Figure 37) and C# (Figure 38).

We then study the distribution of different types of clones in the different clones location (Same file, Same directory, and different directories) using 20%

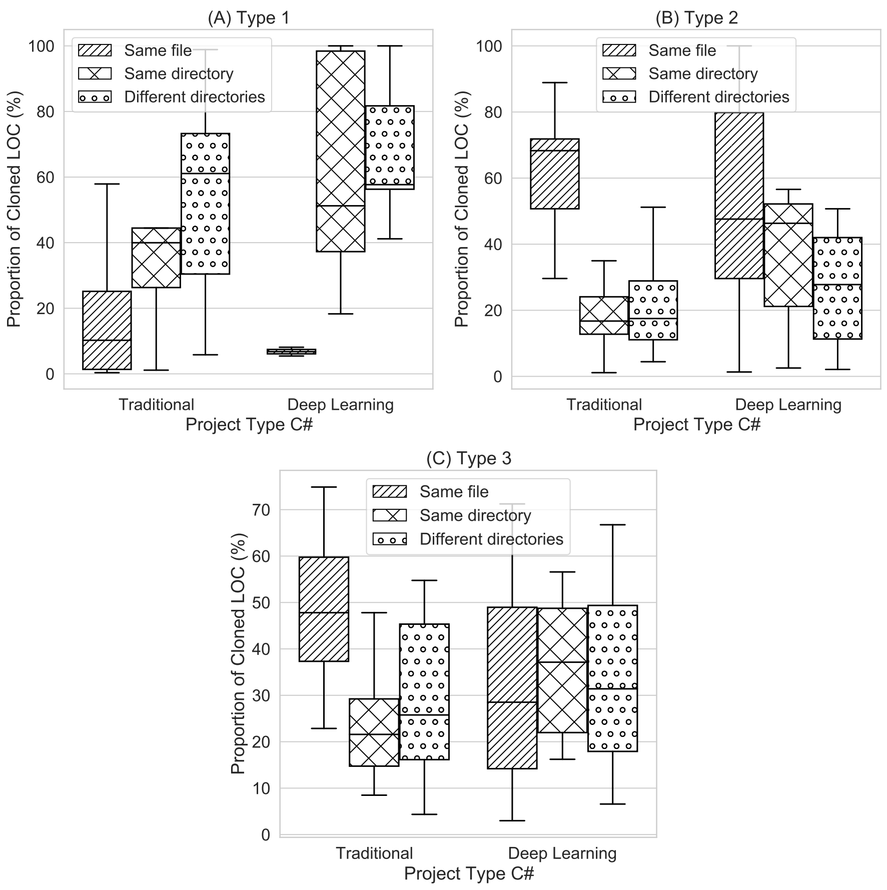

Fig. 34 Distribution of Different Types of Clones by Clone Location in DL and Traditional Code (C#)

dissimilarity threshold for the two programming languages Java (Figure 39) and C# (Figure 40).

Appendix D RQ3 Additional Results

In this section, we provide additional analysis on the distribution of the size of cloned and non-cloned functions in DL and traditional systems (Figure 41). This is done to understand if size is playing an important confounding role in identifying bug fixing commits that are related to clones. We study the distribution of the mean size of cloned and non-cloned functions per systems in DL and traditional systems in Python projects (Figure 42) and for Java and C# (Figure 43).

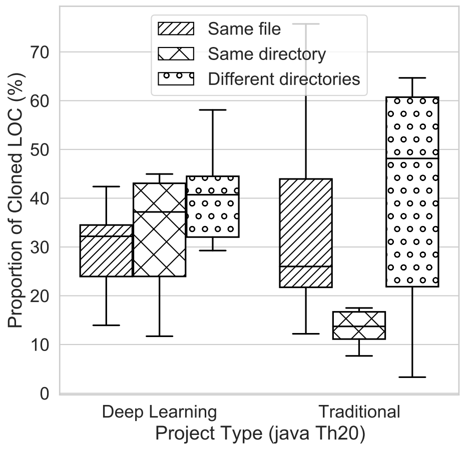

Fig. 35 Code Clones Distribution by Location in DL and Traditional java Systems using 20% as Threshold Regarding Percentage of Lines of Code Clones (LOCC). i.e, (LOCC/total LOCC)x 100

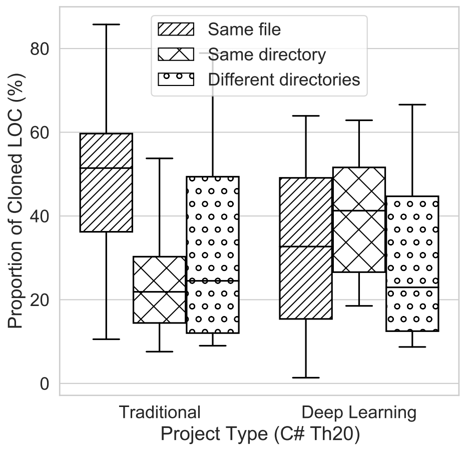

Fig. 36 Code Clones Distribution by Location in DL and Traditional C# Systems using 20% as Threshold Regarding Percentage of Lines of Code Clones (LOCC). i.e, (LOCC/total LOCC)x 100

Appendix E RQ4 Additional Results

As explained in RQ4 but in percentages, Table 23 shows the total number of code clones attributed to the DL phases. The total number of code clones manually analyzed is 595.

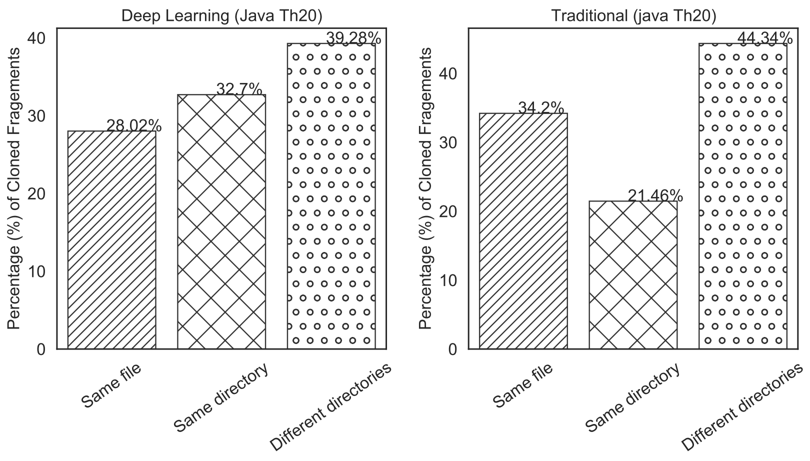

Fig. 37 Percentages of Average Number of Fragments of Code Clones by Location of Clones in both Deep Learning and Traditional Java Systems using 20% as threshold

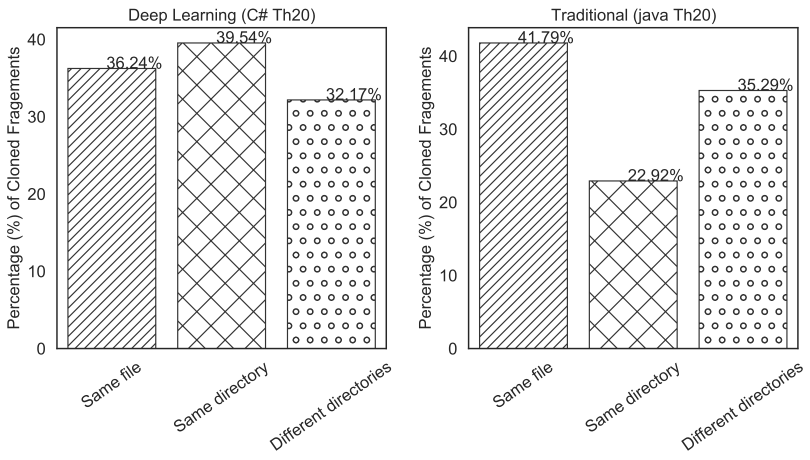

Fig. 38 Percentages of Average Number of Fragments of Code Clones by Location of Clones in both Deep Learning and Traditional C# Systems using 20% as threshold

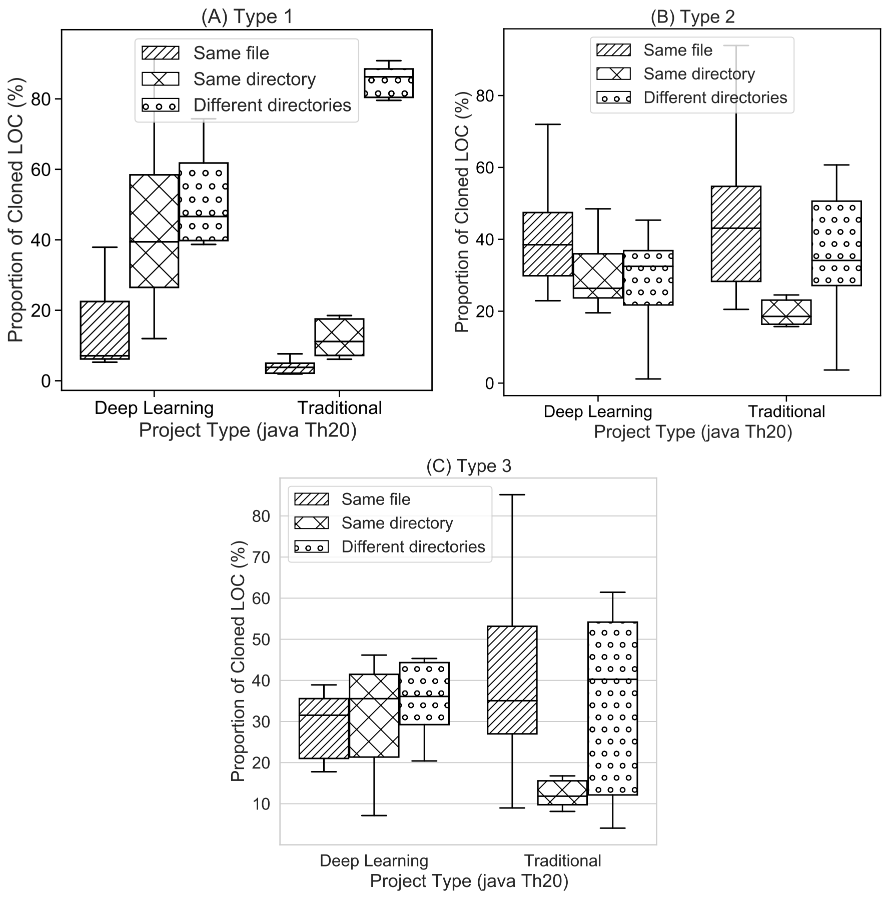

Fig. 39 Distribution of Different Types of Clones by Clone Location in DL and Traditional Code (Java) with 20% as threshold

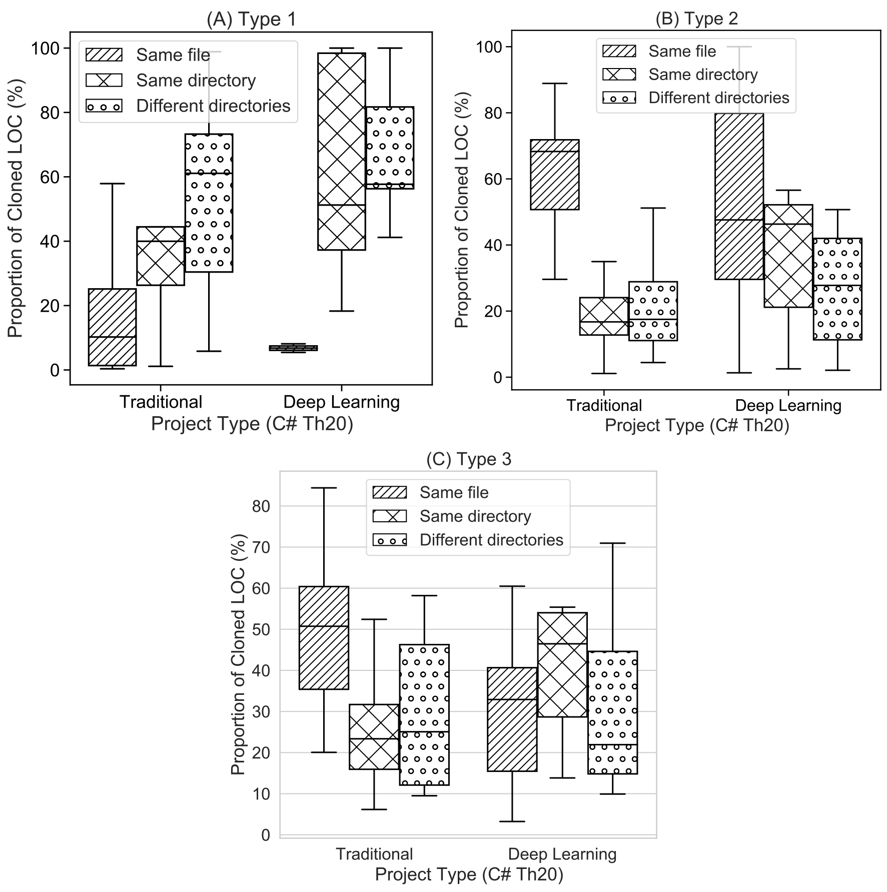

Fig. 40 Distribution of Different Types of Clones by Clone Location in DL and Traditional Code (C#) with 20% as threshold

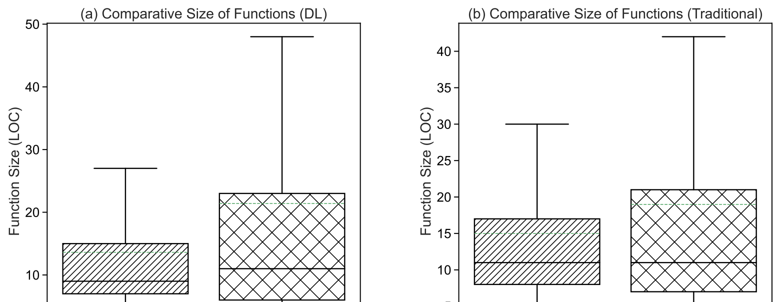

Fig. 41 Distribution of the Size of Cloned and Non-cloned Functions in DL and Traditional Systems

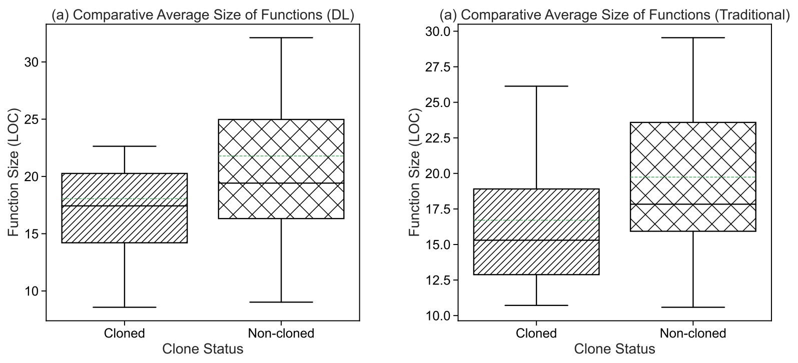

Fig. 42 Distribution of Mean Size of Cloned and Non-cloned Functions Per Systems in DL and Traditional Systems

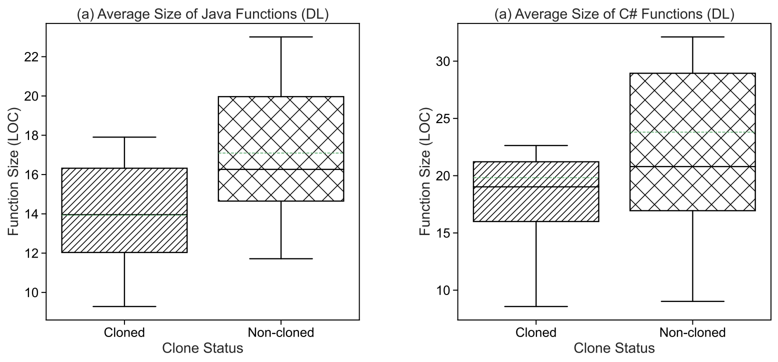

Fig. 43 Distribution of Mean Size of Cloned and Non-cloned Functions Per Systems in Java and C# DL Systems

Table 23 Total number 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 subcat**	**occs in total**
Preliminary preparation	hardware requirements	6	0	0	6	6
Data collection	load data	6	6	12	24	30
	load label	6	0	0	6	
Data postprocessing	compute output shape	0	0	6	6	50
	object localization	12	0	7	19	
	process output	12	0	7	19	
	set shape of output data	6	0	0	6	
Data preprocessing	apply data augmentation	6	0	0	6	110
	data normalization	0	0	12	12	
	get batches of data	0	6	0	6	
	get numerical feature columns	6	0	6	12	
	parse arguments	0	0	6	6	
	prepare tensor	12	0	0	12	
	process input	0	0	20	20	
	resize image	6	0	0	6	
	set shape of input data	0	0	12	12	
	set type of input data	0	0	6	6	
	setting format input data	0	0	6	6	
	split data	0	0	6	6	
Model prediction	inference	12	0	0	12	12
Model construction	model component format verif.	6	0	0	6	216
	activation function call	0	0	6	6	
	build model	6	0	0	6	
	build one subnetwork	0	0	6	6	
	compute model outputs	0	0	6	6	
	init evaluation metrics	0	0	6	6	
	initialize model graph	0	0	6	6	
	initialize model output	6	0	0	6	
	layer construction	0	6	6	12	
	model architecture instantiation	0	0	12	12	
	model (hyper)parameters init	33	46	65	144	
Model evaluation	performance metric computation	0	12	36	48	55
	test data prediction	0	0	7	7	
Model training	compute loss	30	0	0	30	110
	get pooling info	0	0	6	6	
	measure model accuracy	7	0	0	7	
	model training	6	0	12	18	
	one model step training	12	6	12	30	
	training procedure	0	0	7	7	
	weight normalization	0	0	12	12	
Model tuning	Minibatch size	0	0	6	6	6