How are Deep Learning Models Similar? An Empirical Study on Clone Analysis of Deep Learning Software

Xiongfei Wu
University of Science and Technology of China
sa517406@mail.ustc.edu.cn

Liangyu Qin
University of Science and Technology of China
sa517274@mail.ustc.edu.cn

Bing Yu
Kyushu University
3ie17301w@s.kyushu-u.ac.jp

Xiaofei Xie
Nanyang Technological University
xfxie@ntu.edu.sg

Lei Ma
Kyushu University
malei@ait.kyushu-u.ac.jp

Yinxing Xue
University of Science and Technology of China
yxxue@ustc.edu.cn

Yang Liu
Nanyang Technological University
yangliu@ntu.edu.sg

Jianjun Zhao
Kyushu University
zhao@ait.kyushu-u.ac.jp

ABSTRACT

Deep learning (DL) has been successfully applied to many cutting-edge applications, e.g., image processing, speech recognition, and natural language processing. As more and more DL software is made open-sourced, publicly available, and organized in model repositories and stores (Model Zoo, ModelDepot), there comes a need to understand the relationships of these DL models regarding their maintenance and evolution tasks. Although clone analysis has been extensively studied for traditional software, up to the present, clone analysis has not been investigated for DL software. Since DL software adopts the data-driven development paradigm, it is still not clear whether and to what extent the clone analysis techniques of traditional software could be adapted to DL software.

In this paper, we initiate the first step on the clone analysis of DL software at three different levels, i.e., source code level, model structural level, and input/output (I/O)-semantic level, which would be a key in DL software management, maintenance and evolution. We intend to investigate the similarity between these DL models from clone analysis perspective. Several tools and metrics are selected to conduct clone analysis of DL software at three different levels. Our study on two popular datasets (i.e., MNIST and CIFAR-10) and eight DL models of five architectural families (i.e., LeNet, ResNet, DenseNet, AlexNet, and VGG) shows that: 1). the three levels of similarity analysis are generally adequate to find clones between DL models ranging from structural to semantic; 2). different measures for clone analysis used at each level yield similar results; 3) clone analysis of one single level may not render a complete picture of the similarity of DL models. Our findings open up several research opportunities worth further exploration towards better understanding and more effective clone analysis of DL software.

ICPC '20, October 5–6, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7958-8/20/05…$15.00
https://doi.org/10.1145/3387904.3389254

KEYWORDS

Code clone detection, deep learning, model similarity

ACM Reference Format:
Xiongfei Wu, Liangyu Qin, Bing Yu, Xiaofei Xie, Lei Ma, Yinxing Xue, Yang Liu, and Jianjun Zhao. 2020. How are Deep Learning Models Similar? An Empirical Study on Clone Analysis of Deep Learning Software. In *28th International Conference on Program Comprehension (ICPC '20), October 5–6, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3387904.3389254

1 INTRODUCTION

Deep learning (DL) has been widely applied to many cutting-edge applications across domains, e.g., autonomous driving, image recognition, speech recognition, machine translation, etc. We are witnessing a trend that more and more DL software is made open-sourced, publicly available, and organized in model repositories and stores (Model Zoo [1], ModelDepot [2]). In some cases, the training program source code, especially the code fragments relevant to deep neural network (DNN) structure implementation, is copy-pasted and modified for similar purposes. A challenge that naturally arises is to analyze the relation of different DL software for better management, searching, recommendation, etc.

In traditional software, *software clone analysis* [21, 53] is among one of the most important techniques in software management, evolution, and maintenance. It analyzes the similarity extent of software artifacts, on which activities in software life-cycle (e.g., code recommendation, refactoring, code smell analysis, fault localization) could be better performed. Clone analysis of traditional software has been extensively studied at different levels: from textual based to semantic based [11]. *However, for DL software developed by the new data-driven programming paradigm, little progress of clone analysis has been made so far, and it is not clear what would be a suitable way to perform clone analysis for DL software either.*

Before answering how the DL software is similar, it is particularly important to investigate the following research problems: **RP1**. On what representations or levels, should we compare their similarity (or conduct clone analysis)? **RP2**. On each representation or level, what metrics could we adopt? **RP3**. How is the similarity of one representation (or at one level) relevant to those of other representations? *To the best of our knowledge, this paper is the first study that aims to address these problems in the context of DL software.*

Towards approaching answers to these questions, this paper performs an empirical study of clone analysis for the commonly-used DL models, including LeNet, ResNet, DenseNet, VGG, and AlexNet (see Fig. 1). In particular, to investigate the **RP1**, we conduct the clone analysis at three different levels, including *the source code level* at the bottom, *the DNN structure level* at the middle, and *the input/output semantics level* at the top. For **RP2**, we adopt *textual similarity* at the static source code level, the *static weight distance-based similarity* analysis at the DNN structure level, and the DL *runtime similarity analysis based on the pairs of input-output (I/O) relations* at the semantic level. Intuitively, when two DL models share the same structure under the white-box context, the distance of weights can be directly calculated by cosine similarity or Euclidean distance. In the case of black-box or the models with different structures, we perform model distillation [25] on the two models to extract their decision logic into the same DNN structure, based on which the weight distance-based method is used for similarity estimation. Finally, to investigate the **RP3**, we check to what extent traditional textual-based similarity analysis [21] could be useful for DNN structure similarity estimation, and what is the relevance between the DNN structure and the DL runtime I/O semantics.

Our empirical study reveals that, for RP1, the three-level analysis from bottom to top of the DL model is generally adequate to represent a DL model — two DL models can be considered similar, if they are similar at least at one level. For RP2, we find that at each level, various similarity measures (or clone detectors) of the same category do not affect the analysis results — e.g., PyCode_Similar [3] and MOSS [5] yield similar results for textual clone analysis at the source code level, and cosine similarity or Euclidean distance also yield similar results for weight distance-based similarity at DNN structure level. For RP3, results show that clone analysis of one single level may not reach the complete picture of the similarity between two DL models — code textual or DNN architectural similarity between two DL models cannot determine their semantic similarity (i.e., the same functionality), and vice versa.

To summarize, this paper makes the following contributions:

- We initiate the first step to study the usefulness of the existing clone analysis for DL software. In particular, MOSS and PyCode_Similar are used to perform textual level clone analysis.
- We investigate the structure level and I/O level clone analysis on DL models by comparing weight distance-based similarity (e.g., cosine similarity and Euclidean distance), model distillation and Jensen–Shannon divergence (JSD).
- We find that textual clone analysis is not effective to capture the structure similarity of DL models. The weight difference is useful when the models have the same structure. I/O based methods can represent the similarity but it highly depends on the selected inputs.
- Based on the results, we discuss and pinpoint some promising directions towards the more effective clone analysis of DL software. More detailed results are publicly available at our website [4].

2 BACKGROUND AND MOTIVATION

2.1 Clone Detection for Traditional Software

In general, code clone refers to identical or similar pieces of code fragments recurring in software systems. In general, code clone can be categorized into four types [53], where the first three types (i.e., Type I, II, and III) are *textual* and the last Type (i.e., IV) is *functional* (i.e., *semantic* in this paper).

Type I Clone: In Type I clone, two code segments are identical to each other. However, there might be some variations in white space, comments, and layouts.

Type II Clone: Syntactically equivalent fragments with some variations in identifiers, literals, types, whitespace, layout and comments.

Type III Clone: In Type III clone, the copied code segment is further modified with inserted, deleted, or updated statements.

Type IV Clone: In Type IV clone, the code fragments are semantically equivalent for performing the same computation, but are syntactically different.

In traditional software, code cloning usually occurs due to code reuse or plagiarism. Only in a few cases, it happens because of the coincidence of having the same/identical implementation. In general, textual similarity analysis (e.g., *n*-gram based [5] or token-based [33]) is mostly applied for detecting Type I and II clones. Structural similarity analysis (e.g., PDG-based [21] or 3D-CFG based [13]) is mainly used for detecting Type III, and some part of Type IV clones (e.g., reorder clone [53]). Last, I/O semantics analysis [30] is for Type IV clone. As an active research area in recent decades, the detection techniques on each type of clone have been extensively studied for traditional software [11].

2.2 Deep Neural Networks

However, it is not obvious how existing clone analysis techniques can be used for DL software (i.e., training program and model). Deep neural network (DNN) is an artificial neural network with multiple layers between the input and output layers [55], which approximates the correct mathematical relationship between the input to the output. DNN often has multiple layers of computing units (i.e., neurons): a layer of input neurons (i.e., input layer), followed by one or more layers of hidden units, and a layer of output neurons (i.e., output layer). These layers are densely or sparsely connected to process and propagate the information throughout the whole DNN to make the decision.

While the decision logic of traditional software is directly encoded in the software code, the DL training program only defines the DNN structure and runtime learning behavior. The final decision logic of DL model is automatically learned on the training dataset, and is encoded as a DNN in the format of a network structure and connection strength (weights) between neurons and layers.

In this paper, we intend to investigate whether similarity at one particular level can indicate the similarity at other levels (i.e., source code, weight distance level of the output layers, and trained model level with stable accuracy.) For example, we study if two DL models have similar code (source code level), how about their structural level or I/O level similarity?

2.3 Motivating Example

Fig. 1 summarizes our clone analysis on DL software at different levels. Like traditional software, clone analysis of DL software can be conducted on the source code level, the structure level and the runtime execution level. On the source code perspective, textual clone (Type I, II, III) analysis could be performed on the training

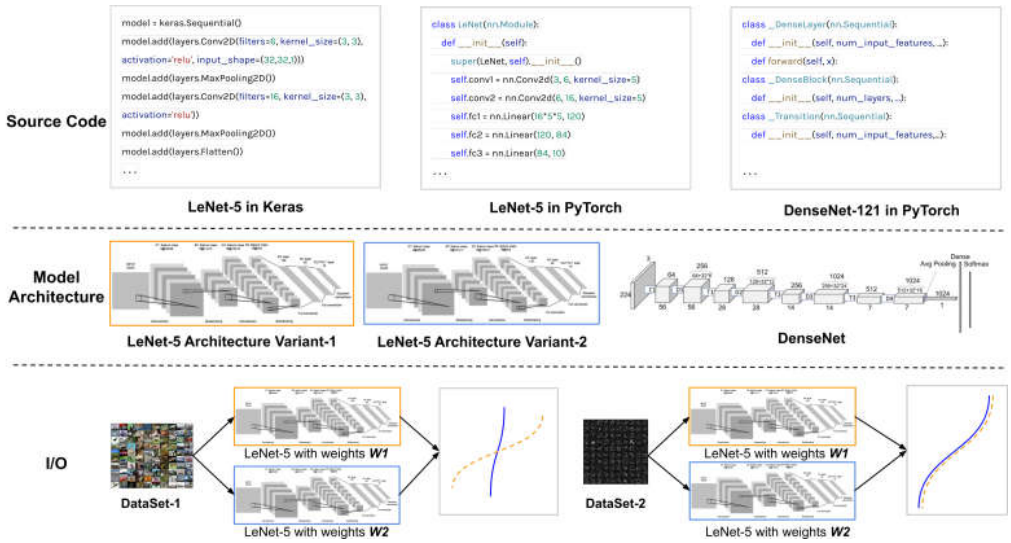


Figure 1: Motivating example of deep learning software clone at different levels.

program code, which defines the DNN structure and runtime training behaviors. For example, it is interesting to investigate how VGG-16 in PyTorch and ResNet-18 in PyTorch are textually similar, as they follow some similar API invocations (e.g., ReLU, Conv2D, etc.). Similarly, we also intend to investigate how the same DNN of different implementations (LeNet-5 in Keras or in PyTorch) are textually similar.

On the structure level, we obtain a DL model when training completes. If two DL models share the same structure, it is possible to directly compare the weight distance for the similarity estimation of model runtime behavior. However, it becomes quite challenging if models under analysis adopt different structures or the target model is a black-box. Hence, at this level, we calculate the weight distance-based similarity regardless of their architectural differences.

At the I/O semantics level, regardless of model transparency and structure, we analyze the I/O relations of two models on a dataset. In particular, after feeding the same dataset as input to different models, their corresponding output relation could reflect the similarity. However, as various inputs of a DL model may yield different results, we may need to use several training datasets to measure their I/O semantic similarity.

This paper performs an exploratory study and quantitative analysis on how these DL models are similar based on the existing clone analysis techniques at three different levels.

3 STUDY OVERVIEW

3.1 Overview of the Study Design

Fig. 2 sketches the overview of our study. In traditional software, clone analysis has been extensively studied based on static analysis and dynamic analysis [11]. In particular, the textual analysis is used to compare the similarity between the source code. Structural analysis is used to compare the similarity between the structure of the source code such as abstract syntax tree (AST), control flow graph (CFG), etc. Dynamic analysis is used to measure the similarity by comparing the input and output values.

Similar to traditional software, the clone analysis of DL software could also be performed at different levels during development phases, i.e., training code, deep neural networks and dynamic prediction. Inspired by the clone analysis techniques on the traditional software, in this paper, we perform an empirical study on how effective the existing clone analysis techniques are on DL software. In particular, we adopt the clone analysis techniques on the training program (textual analysis), DNN (structure-level analysis) and dynamic prediction of multiple models (data-driven I/O semantic analysis). By applying these techniques on DL software, we aim to answer the research questions (RQs), which facilitate better understanding on differences and challenges of clone analysis between traditional software and DL software.

In the paper, we aim to investigate the following research questions (RQs):

RQ1. How effective is the textual similarity analysis on the source code of training programs in evaluating the similarity of the structures of DL models?

RQ2. Given DL models with different structures, how do the distillation techniques help on similarity analysis for estimating the similarity of DL run-time semantics, and how effective is the weight distance-based similarity especially when DL models have the same structure?

RQ3. Given different inputs, how does I/O-semantics similarity analysis perform in capturing the functional similarity of DL models?

The Intention of RQ Design. RQ1-RQ3 are designed to cover the scope of our study for RP1 (on what representations or levels for clone analysis). Specifically, RQ1 is at the source code level, RQ2 is at the DNN structure level, and RQ3 is at the I/O semantics level. For RP3 (the similarity at one level relevance to those of other levels), RQ1 is to check whether the traditional textual analysis technique works for textual clones and indicates higher-level similarity (i.e., structure level); RQ2 studies whether the DNN structural-based analysis is helpful for the similarity estimation of runtime semantics, if the model structures are identical (or different) in RQ2; RQ3

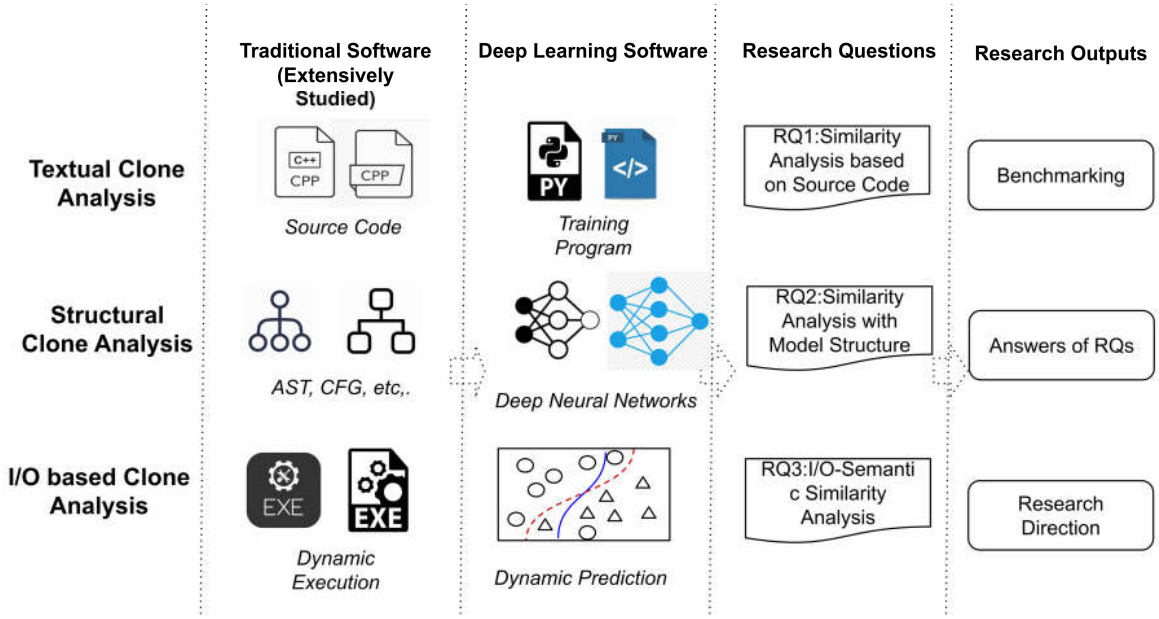


Figure 2: Overview of our study design of clone analysis for deep learning software at different levels.

intends to study whether the traditional black-box testing method (i.e., based on I/O relations) can effectively detect functional clones among DL models, and whether textual similarity at the source code level is relevant to functional clones.

3.2 Datasets and Models

In this paper, we select two publicly available datasets (i.e., MNIST [40], CIFAR-10 [38]) as the subject datasets for training and prediction, both of which are widely studied in the machine learning and software engineering community.

MNIST is a collection of single-channelled image data of size $28 \times 28 \times 1$ for 10 hand-written digit recognition. MNIST contains 70, 000 data in total, consisting of 60, 000 training data and 10, 000 test data.

CIFAR-10 is a collection of images for general-purpose image classification (e.g., airplane, automobile, bird, cat), including 60, 000 color images in total with 50, 000 training images and 10, 000 test images, respectively. Each CIFAR-10 image is three-channel of size $32 \times 32 \times 3$, which is about 4 times dimensionality of MNIST image.

Based on the datasets above, we follow the best deep learning practice and select different DNNs to perform the study, i.e., LeNet-5 [39], AlexNet [37], ResNet-18 [23], ResNet-34 [23], DenseNet-121 [28], DenseNet-161 [28] and VGG-16 [56]. To be representative, these seven popular DL models from the previous study follow different complexity and structures. For each dataset, we trained 7 DL models, where the accuracy of each model is summarized in Table 1. Note that, the aims of this paper is to study the similarity of different models (e.g., with different accuracy). Hence, high accuracy is not a must for a model. Conversely, the more differences the accuracy of models have, the lower the similarity could be.

Table 1: Accuracy, developers, structures of subject models

Family	family-1	family-2		family-3		family-4	family-5
Developer	Author-1	Author-2					Author-3
Model	LeNet-5	ResNet-18	ResNet-34	DenseNet-121	DenseNet-161	VGG-16	AlexNet
MNIST	96.8%	98.93%	99.12%	99.5%	99.2%	98.28%	98.71%
CIFAR-10	64.60%	92.80%	93.20%	94.20%	94.50%	87.7%	81.70%

3.3 Evaluation Metrics of Similarity

Ground truth of similarity. So far, there is no perfect ground truth for representing the similarity between two DNNs. In this paper, we use two metrics from the DL best practice to represent the similarity. We conduct clone analysis based on the assumption that the two DL models are trained on the same training dataset, and it could require future analysis in measuring two models' similarity when the training dataset is unknown.

(1) *Accuracy*. Accuracy is a simple and direct way to approximate the performance of the model. If two models share similar high accuracy on the same test data, they tend to be similar.
(2) *Output distribution similarity*. A more fine-grained way is to evaluate the similarity of the output distribution of two models [67]. In this paper, we use Jensen-Shannon divergence (JSD), calculate the similarity of the softmax outputs (i.e., the probability outputs), which is a symmetric version of Kullback-Leibler divergence for distribution similarity measurement.

Similarity measure at different levels. To address the RP2 (what metrics we should adopt for clone analysis), we calculate the similarity in each level (i.e., source code, structure and I/O semantics) by using the following metrics to measure the similarity at each level:

(1) **Source code similarity** Unlike traditional software similarity matrices which are symmetrical, the PYCODE_SIMILAR uses the

similarity in [63] which is asymmetric. The equation of the algorithm is: $Similarity\ Score = \frac{|common\ value\ seq|}{|signature\ seq|}$. Specifically, we adopt PyCode_Similar [3] or MOSS [5] for textual similarity measuring.

(2) **Structural similarity** Cosine similarity is a measure to compute cos similarity between two given non-zero vectors a and b, their cosine similarity is $cos(\theta) = \frac{a \cdot b}{||a|| \times ||b||}$. Besides, we also use Euclidean distance that is $d(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$. We adopt cosine similarity and Euclidean distance at DNN structure level.

(3) **I/O similarity** In statistics, a commonly used method to measure the similarity of two probability distributions is the Jensen-Shannon divergence (JSD). It is also known as information radius (IRad) or total divergence to the average, which is defined as: $JSD(P_1|P_2) = \frac{1}{2}KL(P_1|\frac{P_1+P_2}{2}) + \frac{1}{2}KL(P_2|\frac{P_1+P_2}{2})$. P_1 and P_2 in the equation are probability distributions and KL stands for Kullback-Leibler divergence, which is defined as: $KL(P_1|P_2) = \sum_{i=1}^{n} P_1(x_i) log\left(\frac{P_1(x_i)}{P_2(x_i)}\right)$. JSD is a symmetric variant of Kullback-Leibler divergence (KLD). Furthermore, the Jaccard index is used to check the prediction distribution. Given two sets A and B, their Jaccard index is $J(A, B) = \frac{|A \cap B|}{|A \cup B|}$. Hence, we adopt JSD and Jaccard index at I/O semantics level.

Notably, to better address RP2, at each level, we adopt two similarity measures and check whether their results are consistent.

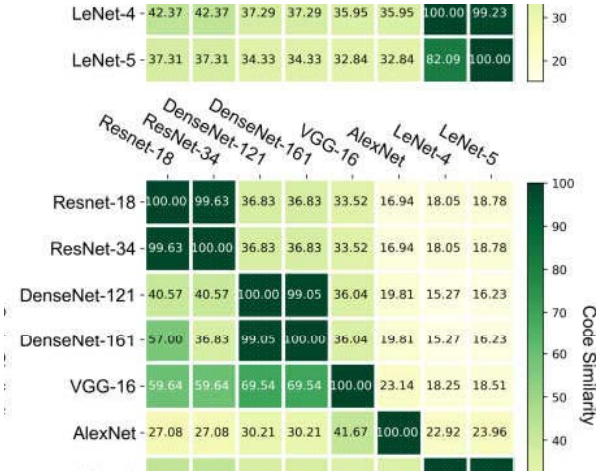
4 EMPIRICAL STUDY

In this section, we discuss the detailed evaluation results for answering our research questions. All the experiments were conducted on a workstation with E5-8160 * 2, four GTX 2080Ti, and 384 GB RAM, running on Ubuntu 16.04 LTS. All the models used in the experiment were implemented using PyTorch except a TensorFlow implementation of LeNet-5 was used (see Table 2). Due to the page limit, for RQ2 and RQ3, we only show the summarized results on CIFAR-10 dataset in the paper and put more complete results (including results on MNIST) on our website [4].

4.1 RQ1: Results of Code Similarity Analysis

4.1.1 Experimental Setup. In general, there are several types of techniques that could be applied for source code clone detection: token-based [22, 33, 52, 58, 60], tree-based [7, 9, 20, 31, 41], graph-based [16, 36, 44], metric-based [18, 49], n-gram based [5, 10, 24, 54] and deep learning based [42, 59, 64]. In this paper, we adopt a tree-based tool, namely pycode-similar [3], to conduct the fast textual clone detection among DL models. Among plenty of existing clone detection techniques, the rationale to adopt the tree-based approach is that this algorithm is easy to understand and apply. Compared with the token-based approach [22, 33, 52, 58, 60], the tree-based approach has the advantage in terms of detection precision and efficiency, as it conducts tree matching with the aid of AST structure and information. In practice, we employ pycode-similar [3] and MOSS [5], these two tools that support Python languages to analyze

Figure 3: Code similarity score using pycode_similar. Note that the row represents the signature code and the column represents the target code for comparison. In the color indicator, darker implies higher similarity.



the code similarity among DL models[1]. Notice that MOSS is a n-gram based tool, while PYCODE-SIMILAR is tree-based.

As we mainly focus on evaluating the DL models, we only take into account the training source code of the DL models for a fair comparison. The code regarding input preprocessing is not included for clone analysis in this study. As mentioned in Table 1, we choose 7 models from 5 structural families as the subjects, on which we evaluate whether models of different structures are similar. In addition, to evaluate the impacts of different implementations on even the same model structures, we collected two versions of LeNet-5 using different frameworks for analysis, i.e., PyTorch and TensorFlow.

4.1.2 Results. Fig. 3 shows the results of the code similarity between different models that are implemented on the same framework, i.e., PyTorch. Code analysis using pycode-similar indicates three interesting findings: 1) models from the same family exhibit very high code similarity. For DenseNet and ResNet, the similarity is more than 99%. For the LeNet family, the similarity decreases but is still more than 80%. 2) models from different families seem to be quite different, exhibiting significantly lower similarity, e.g., all results are less than 60%, except comparing VGG-16 and others. The reason is that the VGG-16 is written by the same author of the ResNet family and DenseNet family, their code style is quite similar. 3) AlexNet seems to be least textually similar compared to other models. The rationale is that the source code of AlexNet is written by different developers. To sum up, results show that code analysis on training programs could be effective in distinguishing the structures of different models *when they are developed using the same framework.*

We further study the effects of different developers and frameworks for code similarity analysis. Table 2 shows two different versions of LeNet-5 developed by the same developer using two different frameworks. Unsurprisingly, the results show that their

[1]In fact, we also tried Nicad [15]. It supports Python language but encounters some errors when analyzing the DL libraries (See details at our website [4])

Table 2: Code similarity between different implementation of LeNet-5 using PyTorch and TensorFlow.

	LeNet-5-PyTorch	LeNet-5-TensorFlow
LeNet-5-PyTorch	100%	11.71%
LeNet-5-Tensorflow	5.29%	100%

code similarity is quite low (11.71% or 5.29%) even if they have the same structure but are with different implementations because of using the APIs from different frameworks.

Hence, the results indicate that different developers and frameworks may result in low code similarity for DL modes even for the same DNN structure.

Besides PYCODE-SIMILAR, we also apply MOSS [5] for the above models. The results are consistent with those in Fig. 3—only models from the same family are with code similarity above 90%, and models from different families exhibit even lower code similarity (usually lower than 10%) due to the fact that MOSS adopts a more strict rule in detecting code clones than PYCODE-SIMILAR. Interested readers can find more detailed results in applying MOSS at our website [4].

Answer to RQ1: Regardless of whether using PYCODE-SIMILAR or MOSS, they yield consistent analysis results. Models from the same DL family (of the same DNN structure) using the same framework have high code similarity (above 99% in many cases). Models from different families or implemented by different frameworks exhibit a similarity of less than 50% in most cases.

4.2 RQ2: Results of Structural Similarity Analysis

After the models are trained, we study the clone analysis on model structures via the weights-based similar analysis. If the models have the same structure, their functional difference is determined only by the weights. Otherwise, for models with different structures, we adopt knowledge distillation [26] to *mutually* convert them for ease of weights-based analysis, in both directions from one model to the other. Notably, as described in §3.3, after knowledge distillation, we adopt the accuracy of the model to approximately represent the semantics, i.e., if their accuracy is similar and high enough on the same training dataset, the semantics should also be similar.

4.2.1 Effectiveness of Weights-based Distance Measure for Structural Similarity. For each model in Table 1, we conduct the study on both training accuracy differences and weight differences. During the training process, we record 10 versions at each 10 epochs, i.e., the model at epoch 0, 10, ..., 100. Note that, we got 10 different versions for each model. The assumption is that the 10 versions may have different semantics but their structures are the same. For each two out of the 10 versions, we calculate the cosine similarity/Euclidean distance between the weights as well as the accuracy distance. Then the relationship between the cosine similarity/Euclidean distance and the accuracy distance shows how these versions are similar in semantics.

Table 3: Comparison between weights difference (WD) and accuracy difference (AD) on DenseNet-121.

	0		20		40		60		80	
	WD	AD	WD	AD	WD	AD	WD	AD	WD	AD
0	0	0	9.28	0.35	9.67	0.32	9.67	0.37	9.66	0.38
20	9.28	0.35	0	0	6.85	0.04	7.04	0.01	7	0.03
40	9.67	0.32	6.85	0.04	0	0	3.24	0.05	3.58	0.07
60	9.67	0.37	7.04	0.01	3.24	0.05	0	0	2.74	0.02
80	9.66	0.38	7	0.03	3.58	0.07	2.74	0.02	0	0

Table 3 shows the difference measurement results for DenseNet-121. The first column shows the models generated at different epoch settings compared with models generated on the settings of the first and second row. Columns *WD* and *AD* refer to settings of the weights difference (i.e., Euclidean distance between weights of models) and the accuracy difference, respectively. From the results, we observe that there is a relationship between WD and AD, i.e., when the weights difference is high, the function of models tends to be different (i.e., high accuracy difference). For example, consider the models at epoch 0 and 80, which have very different accuracy (0.38), we could find that the weights difference (9.66) is also very high. However, consider epoch 60 and 80, their accuracy is close (0.02) and the weights difference also becomes similar (2.74). The results indicate that when using weights to perform similarity analysis, the accuracy of two DL models needs to be stable. *In other words, for two DL models that have the same structure, if they have the similar accuracy on the same dataset, their weights difference should be small.*

4.2.2 Model Distillation. When the structure of two DL models is different (e.g., m_1 and m_2), the direct model comparison by weights and layers becomes difficult.For DNN structural similarity comparison, we cannot simply apply those code-structure (PDG-based [21]) or model similarity (e.g., [14, 51]) analysis techniques. In this study, we employ a state-of-the-art DNN conversion technique Knowledge Distillation, which is a method of network approximation [26]. By introducing a soft-target associated with the teacher network (complex but superior inferential performance) as part of the total loss, to induce student network (student network: streamlined, low complexity) training to achieve knowledge transfer. By using knowledge distillation, we get two converted models (m_1' distilled from m_1 and m_2' from m_2) with the same structure from two given DL models (m_1 and m_2), and then compare their similarity through calculating the similarity based on the weights (parameters) of two student models m_1' and m_2'. We use the weights-based similarity between two student models to represent the similarity of two original DL models.

4.2.3 Experimental Setup. We select seven models, i.e., DenseNet-121, DenseNet-161, ResNet-18, ResNet-34, LeNet-5, AlexNet and VGG-16 and compare each two of them. For each two models that have different structures, we distill them to models with the same structure. One important assumption of weights similarity comparison is to assure that one model (e.g., m_1') after knowledge distillation is functionally comparable to the original model (e.g., m_1) — we try to achieve this by preserving the accuracy after knowledge distillation. In Table 4, we show the results of pair-wise application of knowledge distillation on the seven models. For example, in the first row of Table 4, the original model ResNet-18 has the accuracy of 0.928, which achieves an accuracy of 0.899 after converted into

Table 4: Accuracy of DL models before and after distillation.

Original Model	Accuracy	Target Model	Accuracy
ResNet-18	0.928	ResNet-34	0.899
		LeNet-5	0.712
		DenseNet-121	0.917
		DenseNet-161	0.918
		VGG-16	0.934
		AlexNet	0.853
ResNet-34	0.932	ResNet-18	0.907
		LeNet-5	0.715
		DenseNet-121	0.912
		DenseNet-161	0.914
		AlexNet	0.857
		VGG-16	0.936
LeNet-5	0.646	ResNet-18	0.896
		ResNet-34	0.920
		DenseNet-121	0.914
		DenseNet-161	0.921
		AlexNet	0.855
		VGG-16	0.934
DenseNet-121	0.942	LeNet-5	0.723
		ResNet-18	0.899
		ResNet-34	0.906
		DenseNet-161	0.923
		AlexNet	0.851
		VGG-16	0.934
DenseNet-161	0.945	LeNet-5	0.708
		ResNet-18	0.902
		ResNet-34	0.911
		DenseNet-121	0.914
		AlexNet	0.855
		VGG-16	0.932
AlexNet	0.817	LeNet-5	0.722
		ResNet-18	0.917
		ResNet-34	0.920
		DenseNet-121	0.927
		DenseNet-161	0.929
		VGG-16	0.900
VGG-16	0.877	LeNet-5	0.712
		ResNet-18	0.917
		ResNet-34	0.922
		DenseNet-121	0.930
		DenseNet-161	0.930
		AlexNet	0.855

Figure 4: Cosine similarity of distilled models. In the color indicator, darker implies more similar.

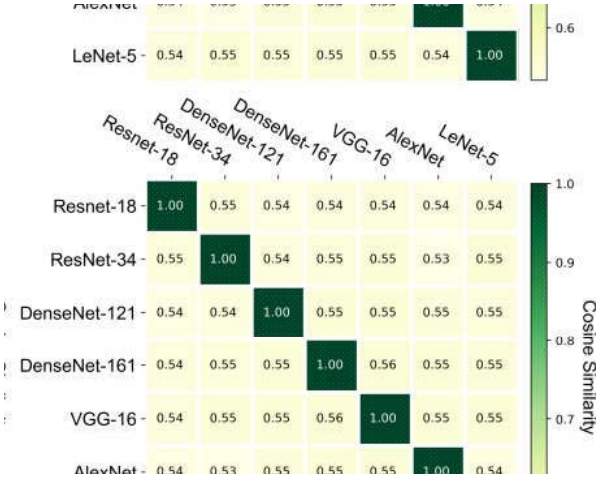
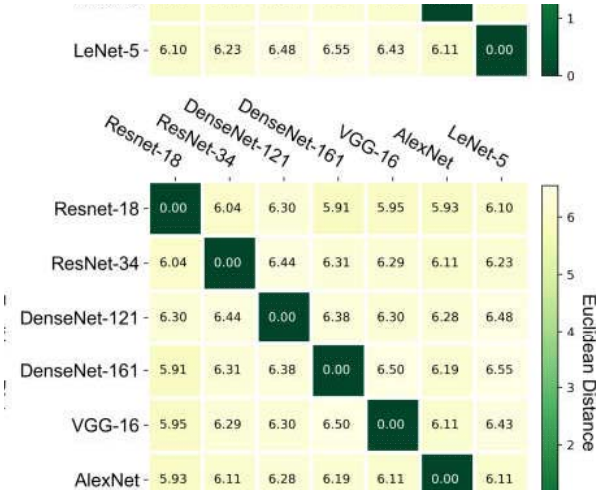


Figure 5: Euclidean distance (normalized) of distilled models. In the color indicator, darker implies more similar.

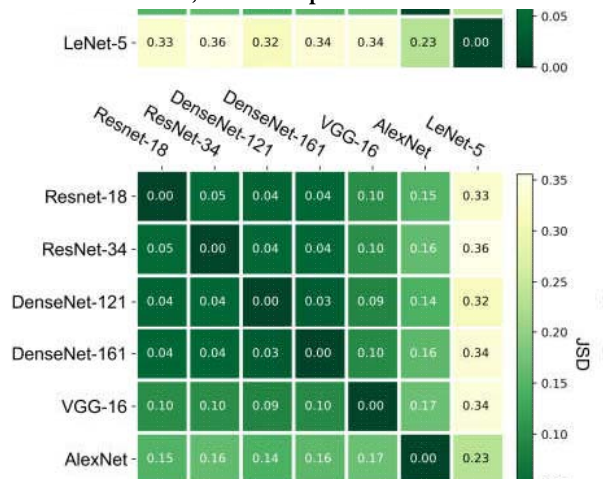


the structure of ResNet-34. The that LeNet-5 seems to be difficult to achieve as high accuracy as other models, mostly due to its simple DNN structure and model capacity.

4.2.4 Results Analysis. In Fig. 4, we show the cosine similarity between the weights of all the models that were converted into the structure of ResNet-18. Due to the page limit, we put more complete results in our website [4]. Interestingly, the results imply three facts. 1) Their weights are neither too similar nor different—except the similarity with the model itself (i.e., 1.0), any other two models have a cosine similarity between 0.5 and 0.6. The results could be explained by the accuracy difference in Table 4. Except for LeNet-5, the original accuracy of other models is very close. Hence, all of the weight similarity are also close. 2) The models from the same family fail to exhibit higher similarity—the ResNet-18 distilled from ResNet-34 has a cosine similarity of 0.55, while the ResNet-18 distilled from DenseNet-121 also has that of 0.54. 3) For LeNet-5, the original accuracy is 0.646 which is much lower than other models, i.e., LeNet-5 should have different semantics than other models. However, after distillation, the similarity results show that the difference becomes similar to any other two models (i.e., between 0.5 and 0.6).

In addition to cosine similarity, we also employ normalized Euclidean distance for weight-based analysis. In Fig. 5, any two distilled models of the same structure have a normalized Euclidean distance between 5 and 6, except that the distance is 0.0 when compared with itself. Hence, using Euclidean distance (normalized) has a similar trend as using cosine similarity in Fig. 4.

Figure 6: JSD between models using CIFAR-10 as input. In the color indicator, darker implies more similar.



Answer to RQ2: For the DL models, after being converted into the DNN of the exact same structure via knowledge distillation, these models show a moderate similarity (cosine similarity 55%) pair-wisely. In other words, weight-based similarity analysis (cosine similarity or Euclidean distance) struggles to distinguish the functionality among these models, and all the models seem to be identical, as the unique information of each model may be lost during the knowledge distillation.

4.3 I/O Semantic Similarity among DL Models

In this section, we first briefly introduce the basic idea of how to detect functional clone based on I/O behaviors (or semantics) analysis. Then, we treat the trained DL models as traditional software and measure their output similarity on the basis of the JSD (see §3.3), given the same testing data as input.

4.3.1 I/O Semantics Similarity. In traditional software, when the black-box testing is conducted for two executables, functional equivalence is a particular case of semantic equivalence that concerns the input/output behavior of a piece of code, regardless of the intermediate program states [30]. Given two program executables p_1 and p_2, the functional similarity can be measured by their output o_1 and o_2 for the input i. If o_1 and o_2 are always identical for any tried input i, p_1 and p_2 are *functionally* equivalent. Otherwise, their similarity can be measured by the probability that p_1 and p_2 yield the same output [12]. Hence, when it comes to the trained DL models, a program is analogical to a DL model with the training data, the input analogical to the testing data to predict, and the output analogical to the output distribution of the DL model. If two trained DL models are *functionally* equivalent (or similar), given the same test data, the output decision boundaries should be equivalent (or similar). Notably, in the scenario of RQ3, the training data is considered as one part of the model itself — if the model is not well trained, we cannot fairly compare the I/O semantics due to the noise brought by the affected accuracy.

4.3.2 Experimental Setup. To obtain the I/O semantic similarity between two DL models (say m_1 and m_2), we train these two models on the same dataset, and then use the randomly generated input (or something they have never seen) to obtain the output distribution (say $\tilde{o}_1$ and $\tilde{o}_2$). After that, a metric is needed to measure the similarity between $\tilde{o}_1$ and $\tilde{o}_2$. Since The KLD is not symmetric, we adopt the JSD. In this paper, all seven models are trained with the training dataset of CIFAR-10. To compare the I/O semantics, the testing datasets of CIFAR-10, MNIST, and randomly generated data (with the same size of CIFAR-10) are separately used as test input, and the output (i.e., predicted label distribution) of the seven trained models are pair-wisely compared based on JSD.

Figure 7: Jaccard similarity between models using CIFAR-10 as input. In the color indicator, darker implies more similar.

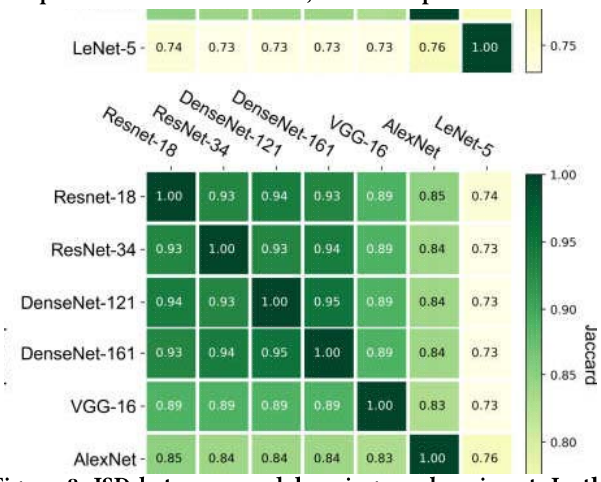
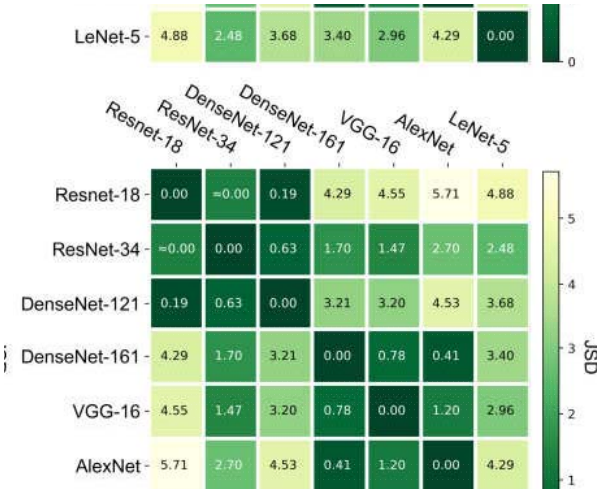


Figure 8: JSD between models using random input. In the color indicator, darker implies more similar.



4.3.3 Results Analysis. In Fig.6, we train the seven models with the CIFAR-10 training dataset and compare their output distributions using CIFAR-10 also as the testing dataset. Results of Fig.6 show that for the same or similar tasks, the six models (except LeNet-5) have very small JSD (< 0.17) in all cases — JSD shows that these six models (except LeNet-5) have very close I/O semantics, being functionally similar. We further inspect this case and check the prediction distribution via the Jaccard similarity in Fig.7. In Fig.6, we can observe that LeNet-5 has a significantly large JSD (above 0.33)

with other models. Similarly, in Fig. 7, we also observe that LeNet-5 has a significantly small Jaccard similarity with other models. Hence, using CIFAR-10 as the testing dataset for the prediction input, these models expect LeNet-5 to exhibit a high functional similarity via I/O semantics analysis, and the analysis on JSD or Jaccard similarity shows similar trend in Fig.6 and Fig. 7.

To confirm the results, more testing datasets should be fed as the prediction input. Hence, we also conduct the experiments on randomly generated data for 20 times and on MNIST. Surprisingly, using different testing datasets can lead to different conclusions. In Fig. 8, with randomly generated data as the testing input, we observe two set of similar models: ResNet-18, ResNet-34 and DenseNet-121 are functionally similar (using random input and yielding the almost identical output distribution); besides, DenseNet-161, VGG-16, AlexNet are functionally similar. Still, LeNet-5 is quite different from other models. On MNIST as the testing input, we observe that all the models have different output distribution—these models are functionally different on predicting MNIST. Notably, the Jaccard similarity analysis produces the same results as that of using JSD.

Answer to RQ3: For the trained models, we find that these models (except LeNet-5) are functionally similar when the testing dataset is close to the training dataset. However, when using randomly generated dataset or MNIST as test input, these tools do not exhibit high functionally similarity. The results are consistent when using JSD or Jaccard similarity for I/O semantics analysis.

5 THREATS TO VALIDITY AND DISCUSSION

5.1 Threats to Validity

The selection of the subject datasets and DNN models could be a threat to validity. Our results may not be general for all models and datasets. In this paper, we selected two popular datasets and the widely-used seven DNN models to reduce the threat. The evaluation metrics may also be a threat to the validity. To reduce the threats, at each level, we adopt two metrics to calculate the similarity. In addition, we use the best practice to select the metrics to calculate the code textual similarity, structural similarity and I/O similarity. Another possible threat is whether the model is properly trained. As using different weights (e.g., epochs) for training may yield different accuracy, the results may not be generalized. To mitigate the problem, we select the models trained from 10 epochs with best tuned weight so that the results are representative.

5.2 Discussion

Based on the findings of this empirical study, we discuss the following open research problems (RPs) and our potential insights as the first step.

5.2.1 RP1. On what representations or levels, should we conduct clone analysis? In this paper, we choose the representations from three levels from bottom to top, namely training code, DNN structure and I/O semantics. They can basically represent the behaviors of DL models at static, structural and dynamic (runtime) levels. If two models are from the same family developed with the same frameworks, they usually have a high textual code similarity. If the structures of the two models are exactly identical (even the same DNN depth), they usually have high weights-based similarity. However, when the structures of these models are different, after we convert them mutually via knowledge distillation, all models seem to be moderately similar on the basis of weight similarity analysis. Furthermore, via I/O semantic analysis, we find that all models except LeNet-5 are functionally similar if using CIFAR-10 as the training and testing dataset. To sum up, the clone analysis on the three levels is capable of telling the differences between DL models to some extent. However, it still leaves two issues to address: *1) How to calculate the structural differences in a more accurate way in alternative ways without knowledge distillation.* 2) How to calculate the I/O semantics analysis when the proper training date is unavailable.

5.2.2 RP2. On each representation or level, what metrics should we adopt? In this study, at each level of clone analysis, we adopt two tools or measures for evaluating the similarity between the two models. Specifically, for code similarity analysis, we apply pycode-similar and MOSS. These two tools are token- or AST-based clone detectors, and they yield quite consistent analysis results. After looking into the source code of these models, we confirm that these two detectors' results are valid, and even CFG- or PDG-based tools (Nicad) do not perform better. For the structural analysis, we currently employ the weights (a.k.a., parameters) based similarity analysis (cosine similarity or Euclidean distance), and these two metrics produce similar results. However, these two metrics are not incapable of distinguishing the models after knowledge distillation. We hold the position that it is not the issue of the metrics, but the problem of knowledge distillation, which makes all the distilled models different from the original ones. Last, the two metrics for I/O semantics (JSD or Jaccard index) yield similar results. However, the results of the current I/O semantics analysis may vary with different testing input. Besides, we now make use of CIFAR-10 as the training dataset, and it is desired to use various training datasets in the future.

5.2.3 RP3. How is the similarity at one level relevant to that at other levels? In traditional software, textual or structural similarity usually indicates or highly correlates functional similarity — *programs of the similar implementations usually perform the same task*, and hence code recommendation can be facilitated by the clone analysis. However, in DL software, we cannot reach the same conclusion. In fact, the textual or structural similarity detected among DL models fails to indicate their functional similarity—in general, the similarity at one level fails to highly correlate to that of another level. Via our preliminary study, we observe that DL software is essentially data-driven program, and the training dataset shapes the functionality of the DL software. *An more complete and accurate comparison of two DL models should take into account the similarity and the distribution of the training dataset. In addition, it is an interesting research direction about how to combine the code, parameters and I/O semantics together.* .

5.2.4 What are research opportunities regarding clones in DL Models? Traditional software clones have many applications such as code recommendation, refactoring, code smell analysis and fault localization. However, whether we can do the similar tasks in DL

models with the clone detected? For example, could we detect the adversarial attacks with clone analysis instead of the ineffective transfer attack? How to use the similarity to interpret the deep learning models? Can we re-use the clones in the models to improve the maintainability?

6 RELATED WORK

Our study is related to two lines of studies, namely clone detection and analysis, and deep learning testing and analysis.

6.1 Clone Detection and Analysis

We briefly introduce clone detection and analysis techniques from the following three perspectives.

6.1.1 Textual Similarity Analysis. There are three techniques commonly used in textual similarity analysis, namely longest common subsequence (LCS) [8], token-based method, and n-gram technique. The representative tools based on LCS include CloneDetective [32] and NiCad [15]. CCFinder [33] and CP-Miner [43] belongs to the token-based tool family. Besides, MOSS [5] relies on the concept of n-gram. However, most of the abovementioned detectors cannot be directly applied to Python program due to the lack of support of syntax front. Hence, we adopt Pycode_Similar and MOSS for textual analysis on Type-I and Type-II clone.

6.1.2 Structural Similarity Analysis. To address the issues of textual analysis due to no consideration of program structures, code clone detectors or analyzers gradually employ different representations for comparing the code of programs [53]. Early in 1998, CloneDR [6] is proposed to use Abstract Syntax Tree (AST) to represent the code structure and locate clones, which has a time complexity of $O(n^3)$. To lower the complexity of tree-comparison, Deckard [29] employs a characteristic vector to approximate an AST. Deckard and 3D-CFG [13] further utilize program dependency graphs (PDGs) as well as control flow graphs (CFGs) to improve the detection accuracy. There are also some tools (ModelCD [51], SIMONE [14] and ConQAT [17]) on model-based clone detection, but they only work for MATLAB or Simulink models, not designed for DL models. In this paper, we propose to measure the structural similarity via the weight similarity analysis after knowledge distillation.

6.1.3 Semantic Similarity Analysis. To realize the detection of cloned code performing the same computation, MeCC [34] presents a memory comparison-based clone detector, which is an I/O semantics analysis based on white-box testing technique. Besides, Zhang et al. [63] propose the concept of value dependence graph (VDG), and then apply LCS to assess the similarity between core values extracted from two implementations of one algorithm. Differently, in DL models, neither the input nor the output is a single value or data list. Hence, the distribution divergence used in this paper is required for comparing I/O semantics between DL models.

Compared to the existing clone detection techniques which focus on the clone analysis for traditional software, we initiate the first step towards the code clone analysis for DL programs. Except the source code level analysis, considering the difference between traditional software and DL software, we also investigate the structure level and I/O level clone analysis on DL models.

6.2 Deep Learning Testing and Analysis

Recent progress starts to be made on testing and analysis of DL software [65, 66]. DeepXplore [50] proposed the first differential testing frameworks to capture the differences of two DNNs. TensorFuzz [48] proposed the coverage-guided testing technique to detect the disagreements of different DNNs. Intuitively, TensorFuzz tends to generate new input that is far from the existing inputs. DiffChaser [62] transforms the problem of detecting disagreements in detecting samples that are near the decision boundary. The genetic algorithm is further used to generate such samples by minimizing the output difference. In addition, DiffChaser could generate targeted disagreements. The techniques above mainly focus on generating disagreements but are difficult to quantify the similarity directly. Instead, we perform the study on quantifying the similarity of DL software at different levels.

In addition, some testing criteria are also proposed for measuring the internal behaviors of DNNs. Followed by DeepXplolre that propsoed neuron coverage, DeepGauge [46] proposed a set of testing criteria such as k-multisection neural coverage, neural boundary coverage and etc. DeepCT [45] proposes a set of combinatorial testing criteria for testing DNNs. The surprise adequacy test criteria are proposed in [35]. DeepStellar [19] proposes a set of testing criteria for recurrent neural networks. With these metrics, some coverage-guided testing tools [57, 61] are developed to test deep neural networks. DeepMutation [27, 47] utilizes the idea of mutation testing to build testing frameworks for deep learning software. We believe a possible research direction is to develop clone analysis techniques based on the existing and further proposed new testing criteria. Different from the above studies trying to propose effective testing criteria for DNNs, our work mainly focuses on analyzing the similarity and clones between DL models.

7 CONCLUSION

In this paper, we performed the first step in studying how the existing clone analysis techniques perform in the deep learning software. The study shows that the textual clone analysis is not effective to evaluate the structure similarity of models, especially when they are written by different developers or on different frameworks. For models with the same structure, the weights difference is useful to show the similarity. For models that have different structures, knowledge distillation is not useful to capture the differences. The I/O based technique can represent the similarity but the results are very dependent on the selected inputs. We further discussed the challenge and the potential research direction about clone analysis in DL software. We believe similarity analysis is very important for model analysis, selection and explanation. This study makes the first step towards this direction on developing techniques to evaluate the similarity of models based on clone analysis techniques.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comprehensive feedback. This research was supported (in part) by the National Natural Science Foundation of China (General Program) Grant No.61972373, CAS Pioneer Hundred Talents Program, JSPS KAKENHI Grant No.20H04168, 19K24348, 19H04086, 18H04097, and Qdai-jump Research Program No.01277.

REFERENCES

[1] [n.d.]. Deep Learning code and pretrained models for transfer learning, educational purpose, and more. https://modelzoo.co/. Accessed October, 2019.

[2] [n.d.]. ModelDepot-Open, Transparent Machine Learning for Engineers. https://modeldepot.io. Accessed October, 2019.

[3] [n.d.]. A simple plagiarism detection tool for python code. https://github.com/fyrestone/pycode_similar. Accessed October, 2019.

[4] 2019. An Emperical Study on Clone Analysis of Deep Learning Software. https://sites.google.com/view/dl-model-similarity Last accessed 25 October 2019.

[5] Alex Aiken. 2017. *MOSS, A System for Detecting Software Plagiarism.* https://theory.stanford.edu/~aiken/moss/

[6] Ira D. Baxter, Andrew Yahin, Leonardo Mendonça de Moura, Marcelo Sant'Anna, and Lorraine Bier. 1998. Clone Detection Using Abstract Syntax Trees. In *1998 International Conference on Software Maintenance, ICSM 1998, Bethesda, Maryland, USA, November 16-19, 1998.* 368–377.

[7] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272).* 368–377. https://doi.org/10.1109/ICSM.1998.738528

[8] Lasse Bergroth, Harri Hakonen, and Timo Raita. 2000. A Survey of Longest Common Subsequence Algorithms. In *Seventh International Symposium on String Processing and Information Retrieval, SPIRE 2000, A Coruña, Spain, September 27-29, 2000.* 39–48.

[9] Christopher Brown and Simon Thompson. 2010. Clone Detection and Elimination for Haskell *(PEPM '10).* ACM, New York, NY, USA, 111–120. https://doi.org/10.1145/1706356.1706378

[10] Steven Burrows, S. M. M. Tahaghoghi, and Justin Zobel. 2007. Efficient Plagiarism Detection for Large Code Repositories. *Softw. Pract. Exper.* 37, 2 (Feb. 2007), 151–175. https://doi.org/10.1002/spe.v37:2

[11] Ragkhitwetsagul Chaiyong. 2018. *Code similarity and clone search in large-scale source code data.* Ph.D. Dissertation. University College London.

[12] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. BinGo: cross-architecture cross-OS binary search. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016.* 678–689.

[13] Kai Chen, Peng Liu, and Yingjun Zhang. 2014. Achieving accuracy and scalability simultaneously in detecting application clones on Android markets. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014.* 175–186.

[14] J. R. Cordy. 2015. SIMONE: architecture-sensitive near-miss clone detection for Simulink models. In *2015 First International Workshop on Automotive Software Architecture (WASA).* 1–2.

[15] J. R. Cordy and C. K. Roy. 2011. The NiCad Clone Detector. In *2011 IEEE 19th International Conference on Program Comprehension.* 219–220. https://doi.org/10.1109/ICPC.2011.26

[16] Jonathan Crussell, Clint Gibler, and Hao Chen. 2012. Attack of the Clones: Detecting Cloned Applications on Android Markets. In *ESORICS.*

[17] Florian Deissenboeck, Benjamin Hummel, Elmar Jürgens, Bernhard Schätz, Stefan Wagner, Jean-François Girard, and Stefan Teuchert. 2008. Clone Detection in Automotive Model-Based Development *(ICSE '08).* Association for Computing Machinery, New York, NY, USA, 603–612. https://doi.org/10.1145/1368088.1368172

[18] John L. Donaldson, Ann-Marie Lancaster, and Paula H. Sposato. 1981. A Plagiarism Detection System *(SIGCSE '81).* ACM, New York, NY, USA, 21–25. https://doi.org/10.1145/800037.800955

[19] Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Yang Liu, and Jianjun Zhao. 2019. DeepStellar: Model-Based Quantitative Analysis of Stateful Deep Learning Systems. In *The 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.*

[20] Raimar Falke, Pierre Frenzel, and Rainer Koschke. 2008. Empirical evaluation of clone detection using syntax suffix trees. *Empirical Software Engineering* 13 (2008), 601–643.

[21] Mark Gabel, Lingxiao Jiang, and Zhendong Su. 2008. Scalable detection of semantic clones. In *30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, May 10-18, 2008.* 321–330.

[22] David Gitchell and Nicholas Tran. 1999. Sim: A Utility for Detecting Similarity in Computer Programs *(SIGCSE '99).* ACM, New York, NY, USA, 266–270. https://doi.org/10.1145/299649.299783

[23] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep residual learning for image recognition. In *CVPR.* 770–778.

[24] Abram Hindle, Earl T. Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. 2012. On the Naturalness of Software *(ICSE '12).* IEEE Press, Piscataway, NJ, USA, 837–847. http://dl.acm.org/citation.cfm?id=2337223.2337322

[25] Geoffrey Hinton, Oriol Vinyals, and Jeffrey Dean. 2015. Distilling the Knowledge in a Neural Network. In *NIPS Deep Learning and Representation Learning Workshop.* http://arxiv.org/abs/1503.02531

[26] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. 2015. Distilling the Knowledge in a Neural Network. arXiv:stat.ML/1503.02531

[27] Q. Hu, L. Ma, X. Xie, B. Yu, Y. Liu, and J. Zhao. 2019. DeepMutation++: A Mutation Testing Framework for Deep Learning Systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE).* 1158–1161.

[28] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q Weinberger. 2017. Densely connected convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition.*

[29] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stéphane Glondu. 2007. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007.* 96–105.

[30] Lingxiao Jiang and Zhendong Su. 2009. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis, ISSTA 2009, Chicago, IL, USA, July 19-23, 2009.* 81–92.

[31] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. 2007. Context-based Detection of Clone-related Bugs *(ESEC-FSE '07).* ACM, New York, NY, USA, 55–64. https://doi.org/10.1145/1287624.1287634

[32] Elmar Jürgens, Florian Deissenboeck, and Benjamin Hummel. 2009. CloneDetective - A workbench for clone detection research. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings.* 603–606.

[33] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code. *IEEE Trans. Software Eng.* 28, 7 (2002), 654–670.

[34] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. 2011. MeCC: memory comparison-based clone detector *(ICSE '11).* 301–310.

[35] J. Kim, R. Feldt, and S. Yoo. 2019. Guiding Deep Learning System Testing Using Surprise Adequacy. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE).* 1039–1049. https://doi.org/10.1109/ICSE.2019.00108

[36] Jens Krinke. 2001. Identifying similar code with program dependence graphs. *Proceedings Eighth Working Conference on Reverse Engineering* (2001), 301–309.

[37] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. 2017. ImageNet Classification with Deep Convolutional Neural Networks. *Commun. ACM* 60, 6 (May 2017), 84–90. https://doi.org/10.1145/3065386

[38] Nair Krizhevsky, Hinton Vinod, Christopher Geoffrey, Mike Papadakis, and Anthony Ventresque. 2014. The cifar-10 dataset. http://www.cs.toronto.edu/kriz/cifar.html.

[39] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. 1998. Gradient-based learning applied to document recognition. *Proc. of the IEEE* 86, 11 (1998), 2278–2324.

[40] Yann LeCun and Corrina Cortes. 1998. The MNIST database of handwritten digits.

[41] Huiqing Li and Simon Thompson. 2009. Clone Detection and Removal for Erlang/OTP Within a Refactoring Environment *(PEPM '09).* ACM, New York, NY, USA, 169–178. https://doi.org/10.1145/1480945.1480971

[42] Liuqing Li, He Feng, Wenjie Zhuang, Na Meng, and Barbara G. Ryder. 2017. CCLearner: A Deep Learning-Based Clone Detection Approach. *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (2017), 249–260.

[43] Zhenmin Li, Shan Lu, Suvda Myagmar, and Yuanyuan Zhou. 2004. CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code. In *6th Symposium on Operating System Design and Implementation (OSDI 2004), San Francisco, California, USA, December 6-8, 2004.* 289–302.

[44] Chao Liu, Chen Chen, Jiawei Han, and Philip S. Yu. 2006. GPLAG: detection of software plagiarism by program dependence graph analysis. In *KDD.*

[45] Lei Ma, Felix Juefei-Xu, Minhui Xue, Bo Li, Li Li, Yang Liu, and Jianjun Zhao. 2019. DeepCT: Tomographic Combinatorial Testing for Deep Learning Systems. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER).* IEEE, 614–618.

[46] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, et al. 2018. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering.* ACM, 120–131.

[47] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao, and Y. Wang. 2018. DeepMutation: Mutation Testing of Deep Learning Systems. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE).* 100–111.

[48] Augustus Odena, Catherine Olsson, David Andersen, and Ian Goodfellow. 2019. TensorFuzz: Debugging Neural Networks with Coverage-Guided Fuzzing. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research),* Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.), Vol. 97. PMLR, Long Beach, California, USA, 4901–4911.

[49] K. J. Ottenstein. 1976. An Algorithmic Approach to the Detection and Prevention of Plagiarism. *SIGCSE Bull.* 8, 4 (Dec. 1976), 30–41. https://doi.org/10.1145/382222.382462

[50] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 1–18.

[51] Nam H. Pham, Hoan Anh Nguyen, Tung Thanh Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2009. Complete and accurate clone detection in graph-based models. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*. 276–286.

[52] Lutz Prechelt, Guido Malpohl, and Michael Philippsen. 2002. Finding Plagiarisms among a Set of Programs with JPlag. *J. UCS* 8 (2002), 1016–.

[53] Chanchal Kumar Roy and James R. Cordy. 2007. A Survey on Software Clone Detection Research. *School of Computing TR 2007-541, Queen's University* 115 (2007).

[54] Saul Schleimer, Daniel S. Wilkerson, and Alex Aiken. 2003. Winnowing: Local Algorithms for Document Fingerprinting *(SIGMOD '03)*. ACM, New York, NY, USA, 76–85. https://doi.org/10.1145/872757.872770

[55] Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural Networks* 61 (Jan 2015), 85–117. https://doi.org/10.1016/j.neunet.2014.09.003

[56] Karen Simonyan and Andrew Zisserman. 2014. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:cs.CV/1409.1556

[57] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*. ACM, 303–314.

[58] G. Whale. 1990. Identification of Program Similarity in Large Populations. *Comput. J.* 33, 2 (April 1990), 140–146. https://doi.org/10.1093/comjnl/33.2.140

[59] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)* (2016), 87–98.

[60] Michael J. Wise. 1996. YAP3: Improved Detection of Similarities in Computer Program and Other Texts. *SIGCSE Bull.* 28, 1 (March 1996), 130–134. https://doi.org/10.1145/236462.236525

[61] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 2019. DeepHunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 146–157.

[62] Xiaofei Xie, Lei Ma, Haijun Wang, Yuekang Li, Yang Liu, and Xiaohong Li. 2019. DiffChaser: Detecting Disagreements for Deep Neural Networks. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, 5772–5778. https://doi.org/10.24963/ijcai.2019/800

[63] Fangfang Zhang, Yoon-chan Jhi, Dinghao Wu, Peng Liu, and Sencun Zhu. 2012. A first step towards algorithm plagiarism detection. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*. 111–121.

[64] Huangzhao Zhang, Zhuo Li, Ge Li, Lei Ma, Yang Liu, and Zhi Jin. 2020. Generating Adversarial Examples for Holding Robustness of Source Code Processing Models. In *The 34th AAAI Conference on Artificial Intelligence (AAAI 2020)*.

[65] Jie M Zhang, Mark Harman, Lei Ma, and Yang Liu. 2020. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering* (2020).

[66] Tianyi Zhang, Cuiyun Gao, Lei Ma, Michael Lyu, and Miryung Kim. 2019. An empirical study of common challenges in developing deep learning applications. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 104–115.

[67] X Zhang, X Xie, L Ma, X Du, Q Hu, Y Liu, J Zhao, and M Sun. 2020. Towards characterizing adversarial defects of deep learning software from the lens of uncertainty. In *The 42nd International Conference on Software Engineering (ICSE 2020)*.