DeepMutation: Mutation Testing of Deep Learning Systems

Lei Ma[1,2], Fuyuan Zhang[2], Jiyuan Sun[3], Minhui Xue[4], Bo Li[5], Felix Juefei-Xu[6], Chao Xie[3], Li Li[7], Yang Liu[2], Jianjun Zhao[3], Yadong Wang[1]
[1]Harbin Institute of Technology, [2]Nanyang Technological University Singapore, [3]Kyushu University
[4]New York University Shanghai, [5]University of Illinois at Urbana–Champaign
[6]Carnegie Mellon University, [7]Monash University

arXiv:1805.05206v1 [cs.SE] 14 May 2018

***Abstract*—Deep learning (DL) defines a new data-driven programming paradigm where the internal system logic is largely shaped by the training data. The standard way of evaluating DL models is to examine their performance on a test dataset. The quality of the test dataset is of great importance to gain confidence of the trained models. Using inadequate test dataset, DL models that have achieved high test accuracy may still suffer from vulnerability against (adversarial) attacks.**

In software testing, mutation testing is a well-established technique to evaluate the quality of test suites. However, due to the fundamental difference of traditional software and deep learning-based software, traditional mutation testing techniques cannot be directly applied to DL systems. In this paper, we propose the mutation testing framework specialized for DL systems. We first propose a source-level mutation testing technique to slightly modify source (*i.e.*, training data and training programs) of DL software, which shares the same spirit of traditional mutation testing. Then we design a set of model-level mutation testing operators that directly mutate on DL models without a training process. The effectiveness of the proposed mutation techniques is demonstrated on two public datasets MNIST and CIFAR-10 with three DL models.

I. Introduction

In recent years, deep learning (DL) has shown great success in many areas, including safety-critical applications, such as autonomous driving [1], robotics [2], games [3], video surveillance [4], etc. However, with the witness of recent catastrophic accidents (*e.g.*, Tesla/Uber) relevant to DL, the robustness and safety of DL systems become a big concern. Currently, the performance of DL systems is mainly measured by the accuracy on a set of prepared test data. Without a systematic way to evaluate and understand the quality of the test data, it is difficult to conclude that good performance on the test data indicates the robustness and generality of a DL system. This problem is further exacerbated by many recently proposed adversarial attacks, which make minor changes (*e.g.*, invisible to human eyes [5]) on the input data to fool DL systems. Due to the unique characteristics of DL systems, new evaluation criteria on the quality of DL systems are highly desirable, and the quality evaluation of test data is of special importance.

For traditional software, mutation testing [6] has been established as one of the most important techniques to systematically evaluate the quality of tests. Mutation testing measures the quality of tests by examining whether a test set can reveal certain types of defects that are injected into the original software under test (SUT). A core component of mutation testing is to define mutation operators that perform fault injection into the SUT to simulate software defects introduced by human developers.

Unlike traditional software systems, whose decision logic is implemented by software developers in the form of code, the behavior of a DL system is mostly determined by the structure of Deep Neural Networks (DNNs) as well as the connection weights in the network. Specifically, the weights are obtained through a model training procedure on training data set, and the DNN structures are often defined by code fragments (*i.e.*, training program) in high level languages (*e.g.*, Python [7], [8] and Java [9]). Therefore, the training data set and the structure of DNNs are two major sources of defects of DL systems.

In the spirit of simulating real defects introduced by human developers, a reasonable approach to mutation testing for DL systems is to inject faults into the training data or the DNN training program. After the faults are injected, the training process could be re-executed, using the mutated training data or training program, to generate the corresponding *mutated* DL model.

In this paper, we propose a framework to perform mutation testing for DL systems. We first design eight source-level mutation testing operators that directly manipulate the training data and training programs. The design intention is to simulate possible faults and problems in the process of collecting training data and implementing training program. For source-level mutation testing, training DNN models can be computationally intensive: the training process can take minutes, hours, even longer [10]. Therefore, we further design eight mutation operators to directly mutate DL models. These model-level mutation operators not only enable more efficient generation of large sets of mutants, but also simulate potential model-level problems that are difficult to produce by mutating training data and/or programs (*e.g.*, incorrect configurations of training process). We have performed in-depth evaluation of the proposed mutation testing techniques on two widely used datasets, namely MNIST and CIFAR-10, and three popular DL models with different structures and complexity. The evaluation result demonstrates the usefulness of the proposed

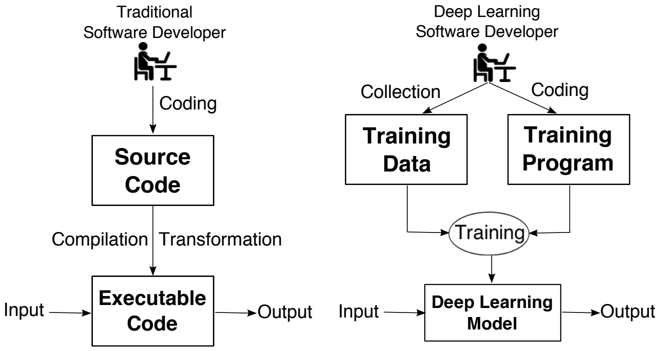

Fig. 1. A comparison of traditional software and DL software development.

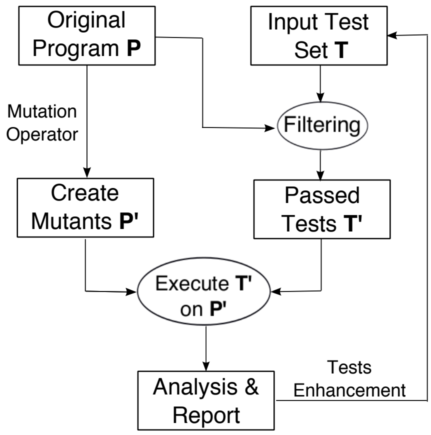

Fig. 2. Key process of general mutation testing.

techniques as a promising measurement towards constructing high quality test data sets, which would eventually facilitate the robustness enhancement of DL systems.

Currently, testing for DL software is still at an early stage, with some initial research work focused on accuracy and neuron coverage, such as DeepXplore [11], DeepGauge [12], and DeepCover [13]. To the best of our knowledge, our work is the first attempt to study mutation testing techniques specialized for DL systems. The main contributions of this paper are summarized as follows:

- We propose eight source-level (*i.e.*, on the training data and training program) mutation testing operators for DL systems.
- We propose eight mutation testing operators specifically for DL models.
- We propose mutation testing workflows to apply source-level and model-level mutation operators. We also define DL-specific mutation testing metrics to quantify the measurement.
- We evaluate the proposed source-level and model-level mutation testing operators on widely studied real-world data sets and DL models, to demonstrate the usefulness of the technique.

The rest of the paper is organized as follows. Section II introduces the background of DL programming paradigm and mutation testing on traditional software. Section III describes the source-level mutation operators as well as the metrics for mutation testing. Section IV describes the model-level mutation operators. Section V gives the details of the evaluation and discusses the results. After the discussion of related work in Section VI, Section VII concludes and highlights future work.

II. Background

A. Programming Paradigms

Building deep learning based systems is fundamentally different from that of traditional software systems. Traditional software is the implementation of logic flows crafted by developers in the form of source code (see Figure 1), which can be decomposed into units (*e.g.*, classes, methods, statements, branches). Each unit specifies some logic and allows to be tested as targets of software quality measurement (*e.g.*, statement coverage, branch coverage). After the source code is programmed, it is compiled into executable form, which will be running in respective runtime environments to fulfill the requirements of the system. For example, in object-oriented programming, developers analyze the requirements and design the corresponding software architecture. Each of the architectural units (*e.g.*, classes) represents specific functionality, and the overall goal is achieved through the collaborations and interactions of the units.

Deep learning, on the other hand, follows a data-driven programming paradigm, which programs the core logic through the model training process using a large amount of training data. The logic is represented by sets of weights fed into non-linear activation functions [14]. To obtain a DL software `F` for a specific task `M`, a DL developer (see Figure 1) needs to collect training data, which specifies the desired behavior of `F` on `M`, and prepare a training program, which describes the DL network structure and runtime training behaviors. The DL model is built by running the training program on the training data. We can see that the major effort for a DL developer is to prepare a set of training data and design DNN model structure, and DL logic is determined automatically through the training procedure. In contrast to traditional software, DL models are often difficult to be decomposed or interpret, making them unamenable to most existing software testing techniques. Moreover, it is challenging to find high quality training and test data that represent the problem space and have good coverage of the models to evaluate their generality.

B. Mutation Testing

The general process of mutation testing [15], [6] for traditional software is illustrated in Figure 2. Given an original program P, a set of faulty programs P' (mutants) are created based on predefined rules (mutation operators), each of which slightly modifies P. For example, a mutation operator can syntactically change '$+$' operator in the program to '$-$' operator [16], [17], [18]. A step of preprocessing, usually before the actual mutation testing procedure starts, is used to filter out irrelevant tests. Specifically, the complete test

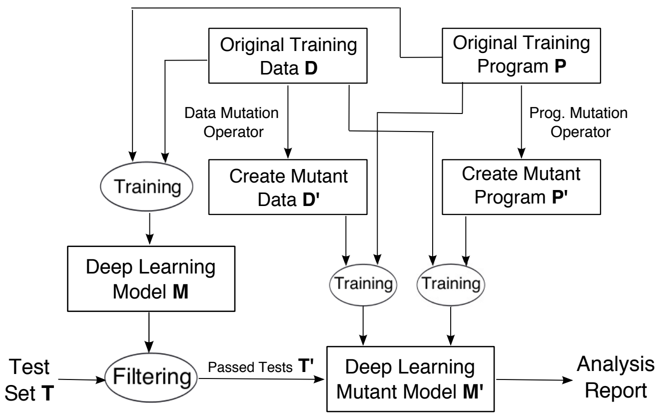

Fig. 3. Source-level mutation testing workflow of DL systems.

set T is executed against P and only the passed tests T' (a subset of T) are used for mutation testing. In the next step, each mutant of P' is executed on T'. If the test result for a mutant $p' \in P'$ is different from that of P, then p' is killed; otherwise, p' is survived. When all the mutants in P' have been tested with T', *mutation score* is calculated as the number of killed mutants over the total number of mutants (*i.e.*, $\#mutants_{killed}/\#mutants_{all}$), which indicates the quality of input test set. Conceptually, a test suite with a higher mutation score is more likely to capture real defects in the program, which has also been supported by empirical evidence [19]. After obtaining the mutation testing results, the developer needs to enhance the quality of test set (*e.g.*, by adding more tests). The general goal of mutation testing is to increase the mutation score to gain confidence on test set T in terms of detecting real faults.

III. Source-level Mutation Testing of DL Systems

Figure 1 illustrates that, in general, traditional software is mostly programmed by human developers in the form of source code, which introduces the major source of bugs for coding process. Mutation testing tries to imitate human-induced defects to some extent, through slightly changing the program code as if the modification was erroneously made by a human developer.

With the same spirit of mutation testing for traditional software, directly introducing defects into the programming sources of a DL system is a reasonable approach to creating its mutated DL models, which simulates the bugs introduced by human developers. In this section, we propose a source-level mutation testing technique for DL systems. We design a general mutation testing workflow for DL systems, and propose a set of mutation operators as the key components of the workflow. Furthermore, we define the mutation testing metrics for quantitative measurement and evaluation.

A. Source-level Mutation Testing Workflow for DL Systems

Figure 3 shows the key workflow of our source-level mutation testing technique. At the initialization phase, a DL developer prepares a training program P and a set of training data D. After the training process, which runs P with D, a DL model M is obtained. When the mutation testing starts, the original training data D and program P are slightly modified by applying mutation operators (defined in Table I), and the corresponding mutants D' and P' are generated. Since D' and P' simulate defects caused by developers, in the next step, either a training data mutant or training program mutant participates in the training process to generate a mutated DL model M'. When mutated DL models are obtained, they are executed and analyzed against the filtered test set T' for evaluating the quality of test data.[1]

TABLE I
Source-level mutation testing operators for DL systems.

Fault Type	Level	Target	Operation Description
Data Repetition (DR)	Global	Data	Duplicates training data
	Local		Duplicates specific type of data
Label Error (LE)	Global	Data	Falsify results (e.g., labels) of data
	Local		Falsify specific results of data
Data Missing (DM)	Global	Data	Remove selected data
	Local		Remove specific types of data
Data Shuffle (DF)	Global	Data	Shuffle selected training data
	Local		Shuffle specific types of data
Noise Perturb. (NP)	Global	Data	Add noise to training data
	Local		Add noise to specific type of data
Layer Removal (LR)	Global	Prog.	Remove a layer
Layer Addition (LA_s)	Global	Prog.	Add a layer
Act. Fun. Remov. (AFM_s)	Global	Prog.	Remove activation functions

B. Source-level Mutation Operators for DL Systems

We propose two groups of mutation operators, namely *data mutation operators* and *program mutation operators*, which simulate the corresponding sources of faults (see Table I).

1) Data Mutation Operators: Training data plays a vital role in building DL models. The training data is usually large in size and labeled manually [20], [21], [22]. Preparing training data is usually laborious and sometimes error-prone. Our data mutation operators try to imitate the various kinds of problems that are introduced during the data collection process. These operators can either be applied globally to all types of data, or locally only to specific types of data within the entire training data set.

- **Data Repetition (DR)**: The DR operator duplicates a small portion of training data. The training data is often collected from multiple sources, some of which are quite similar. In this case, the same data point can be collected more than once.
- **Label Error (LE)**: Each data point (d, l) in the training data set D is a in the form of a pair, where d represents the feature data and l is the label for d. As D is often quite large (*e.g.*, MNIST dataset [20] contains $60,000$ training data), it is not uncommon that some data points can be mislabeled. The LE operator mimics such kind of mistakes by changing the label for a data.

[1] T' consists of the test data points in T that are correctly processed by the original DL model M.

- **Data Missing (DM)**: The DM operator removes some of the training data. It mimics that the developer inadvertently or mistakenly deletes some data points.
- **Data Shuffle (DF)**: The DF operator shuffles the training data into different orders before the training process. Theoretically, the training program runs against the training data should obtain the same DL model. However, the implementation of training program is often sensitive to the order of training data. When preparing training data, developers often pay little attention to the order of data, and thus can easily overlook the problem in the training program. The DF operator tries to simulate such problems, which are related to data order.
- **Noise Perturbation (NP)**: The NP operator randomly adds noise to training data. A data point could carry noise from various sources. For example, an camera-captured image could include noise caused by different weather conditions (*i.e.*, rain, snow, dust, etc.). The NP operator tries to simulate issues related to noisy training data (*e.g.*, NP can add random perturbations to some pixels of an image).

2) Program Mutation Operators: Similar to traditional programs, a training program is commonly coded using high-level programming languages (*e.g.*, Python and Java) under specific DL framework. There are plenty of syntax-based mutation testing tools available for traditional software [23], [24], [25], [26], [27], and it seems straightforward to directly apply these tools to the training program. However, this approach often does not work, due to the fact that DL training programs are sensitive to code changes. Even a slight change can cause the training program to fail at runtime or to produce noticeable training process anomalies (*e.g.*, obvious low predication accuracy at the early iterations/epochs of the training). Considering the characteristics of DL training programs, we design the following operators to better simulate the faults, which are usually minor and unnoticeable.

- **Layer Removal (LR)**: The LR operator randomly deletes a layer of the DNNs on the condition that input and output structures of the deleted layer are the same. Although it is possible to delete any layer that satisfies this condition, arbitrarily deleting a layer (*e.g.*, `Softmax` layer) can generate DL models that are obviously different from the original DL model. Therefore, the LR operator mainly focuses on layers (*e.g.*, `Maxpooling`, `BatchNormalization`, `Dense` layer [28]), whose deletion does not make much difference on the mutated model. The LR operator mimics the case that a line of code representing a DNN layer is removed by the developer.
- **Layer Addition (LA_s)**: In contrast to the LR operator, the LA_s operator adds a layer to the DNNs structure. Similar to LR, LA_s also focuses on adding layers like `Maxpooling`, `BatchNormalization`, `Activation`, which mimics the faults caused by adding or duplicating a line of code representing a DNN layer.

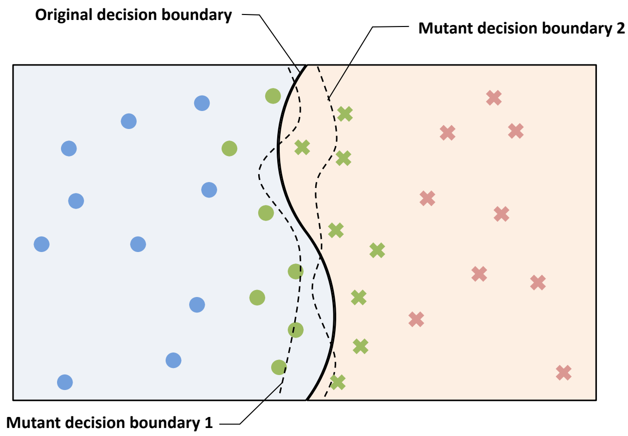

Fig. 4. Example of DL model and its two generated mutant model for binary classification with their decision boundaries. In the figure, some data scatter closer to the decision boundary (in green color). Our mutation testing metrics favor to identify the test data that locate in the sensitive region near the decision boundary.

- **Activation Function Missing (AFM_s)**: Activation function plays an important role of the non-linearity of DNNs for higher representativeness (*i.e.*, quantified as `VC dimension` [14]). The AFM_s operator randomly removes all the activation functions of a layer, to mimic the situation that the developer fails to add the activation layer or activation parameters.

C. Mutation Testing Metrics for DL Systems

After the training data and training program are mutated by the mutation operators, a set of mutant DL models M' can be obtained through training. Each test data point $t' \in T'$ that is correctly handled by the original DL model M, is evaluated on the set of mutant models M'. We say that test data T' kill mutant m' if there exists a test input $t' \in T'$ that is not correctly handled by m'. The mutation score of traditional mutation testing is calculated as the ratio of killed mutants to all mutants. However, it is inappropriate to use the same mutation score metrics of traditional software as the metrics for mutation testing of DL systems. In the mutation testing of DL systems, it is relatively easy for T' to kill a mutant m' when the size of T' is large, which is also convinced from our experiment in Section V. Therefore, if we were to use the mutation score for DL systems as the ratio of killed mutants to all mutants, our metric would lose the precision to evaluate the quality of test data for DL systems.

In this paper, we focus on DL systems for classification problems[2]. Suppose we have a k-classification problem and let $C = \{c_1, \ldots, c_k\}$ be all the k classes of input data. For a test data point $t' \in T'$, we say that t' *kills* $c_i \in C$ of mutant $m' \in M'$ if the following conditions are satisfied: (1) t' is

[2]Although, the mutation score metric defined in this paper mainly focuses on classification problems, the similar idea can be easily extended to handle numerical predication problem as well, with a user-defined threshold as the error allowance margin [29].

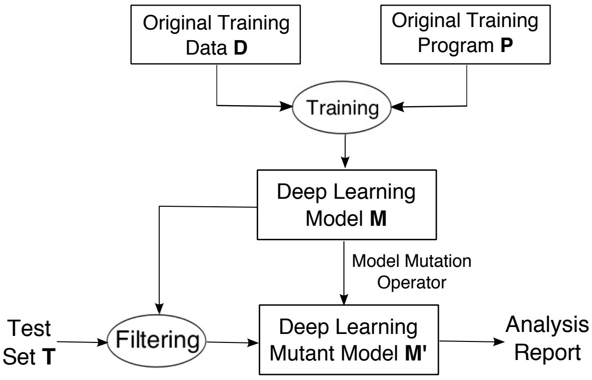

Fig. 5. The model level mutation testing workflow for DL systems.

classified as c_i by the original DL model M, and (2) t' is not classified as c_i by m'. We define the mutation score for DL systems as follows, where $\mathrm{KilledClasses}(T', m')$ is the set of classes of m' killed by test data in T':

$$\mathrm{MutationScore}(T', M') = \frac{\sum_{m' \in M'} |\mathrm{KilledClasses}(T', m')|}{|M'| \times |C|}$$

In general, it could be difficult to precisely predict the behavioural difference introduced by mutation operators. To avoid introducing too many behavioural differences for a DL mutant model from its original counterpart, we propose a DL mutant model quality control procedure. In particular, we measure error rate of each mutant m' on T'. If the error rate of m' is too high for T', we don't consider m' a good mutant candidate as it introduces large behavioral difference. We excluded such mutant models from M' for further analysis.

We define average error rate (AER) of T' on the each mutant model $m' \in M'$ to measure the overall behavior differential effects introduced by all mutation operators.

$$\mathrm{AveErrorRate}(T', M') = \frac{\sum_{m' \in M'} \mathrm{ErrorRate}(T', m')}{|M'|}$$

Figure 4 shows an example of DL model for binary classification, with the decision boundary of the original model and the decision boundaries of two mutant models. We can see that the mutant models are more easily to be killed by data in green, which lies *near* the decision boundary of the original DL model. The *closer* a data point is to the decision boundary, the higher chance it has to kill more mutant models, which is reflected as the increase of the mutation score and AER defined for DL systems. Mutation testing often generates a large number of similar mutants with decision boundaries close to that of the original DL model.

IV. Model-level Mutation Testing of DL Systems

In Section III, we define the source-level mutation testing procedure and workflow, which simulate the traditional mutation testing techniques designed to work on source code (see Figure 1). In general, to improve mutation testing efficient, many traditional mutation testing techniques are designed to work on a low-level software representation (*e.g.*, `Bytecode` [27], [17], `Binary Code` [30], [31]) instead of the source code, which avoid the program compilation and transformation effort. In this section, we propose the model-level mutation testing for DL system towards more efficient DL mutant model generation.

TABLE II
Model-level mutation testing operators for DL systems.

Mutation Operator	Level	Description
Gaussian Fuzzing (GF)	Weight	Fuzz weight by Gaussian Distribution
Weight Shuffling (WS)	Neuron	Shuffle selected weights
Neuron Effect Block. (NEB)	Neuron	Block a neuron effect on following layers
Neuron Activation Inverse (NAI)	Neuron	Change the activation status of a neuron
Neuron Switch (NS)	Neuron	Switch two neurons of the same layer
Layer Deactivation (LD)	Layer	Deactivate the effects of a layer
Layer Addition ($\mathrm{LA_m}$)	Layer	Add a layer in neuron network
Act. Func. Remov. ($\mathrm{AFR_m}$)	Layer	Remove neuron activation function

A. Model-Level Mutation Testing Workflow for DL Systems

Figure 5 shows the overall workflow of DL model level mutation testing workflow. In contrast to the source-level mutation testing that modifies the original training data D and training program P, model level mutation testing directly changes the DL model M obtained through training from D and P. For each generated DL mutant model $m' \in M'$ by our defined model-level mutation operators in Table II, input test dataset T is run on M to filter out all incorrect data and the passed data are sent to run each m'. The obtained execution results adopt the same mutation metrics defined in Section III-C for analysis and report.

Similar to source-level mutation testing, model-level mutation testing also tries to evaluate the effectiveness and locate the weakness of a test dataset, which helps a developer to further enhance the test data to exercise the fragile regions of a DL model under test. Since the direct modification of DL model avoids the training procedure, model-level mutation testing is expected to be more efficient for DL mutant model generation, which is similar to the low-level mutation testing techniques of traditional software.

B. Model-level Mutation Operators for DL Systems

Mutating training data and training program will eventually mutate the DL model. However, the training process can be complicated, being affected by various parameters, such as the number of iterations of running the training program, thresholds of training errors, etc. To simulate possible problems from DL model perspectives, we propose model-level mutation operators, which directly mutate the structure and parameters of DL models. Table II summarizes the proposed model-level mutation operators, which range from weight level to layer level in terms of application scopes of the operators.

- **Gaussian Fuzzing (GF)**: Weights are basic elements of DNNs, which describe the importance of connections between neurons. Weights greatly contribute to the decision logic of DNNs. A natural way to mutate the weight is to fuzz its value and change the connection importance it represents. The GF operator follows the Gaussian distribution $X \sim \mathcal{N}(w, \sigma^2)$ to mutate a given weight value w, where σ is a user-configurable standard deviation parameter. The GF operator mostly fuzzes a weight to its nearby value range (*i.e.*, the fuzzed value locates in $[w - 3\sigma, w + 3\sigma]$ with 99.7 % probability), but also allows a weight to be changed to a greater distance with smaller chance.

- **Weight Shuffling (WS)**: The output of a neuron is often determined by neurons from the previous layer, each of which has connections with weights. The WS operator randomly selects a neuron and shuffles the weights of its connections with previous layer.
- **Neuron Effect Blocking (NEB)**: When a test data point is read into a DNN, it is processed and propagated through connections with different weights and neuron layers until the final results are produced. Each neuron contributes to the DNN's final decision to some extent according to its connection strength. The NEB operator blocks a neuron effects from all of the connected neurons in next layers, which can be achieved by resetting its connection weights of the next layers to zero. The NEB removes the influence of a neuron to the final DNN's decision.
- **Neuron Activation Inverse (NAI)**: The activation function plays a key role in creating the non-linear behaviors of the DNNs. Many widely used activation functions (*e.g.*, ReLU [32], Leaky ReLU [33]) show quite different behaviors depending on their activation status. The NAI operator tries to invert the activation status of a neuron, which can be achieved by changing the sign of the output value of a neuron before applying its activation function. This facilitates to create more mutant neuron activation patterns, each of which can show new mathematical properties (*e.g.*, linear properties) of DNNs [13].
- **Neuron Switch (NS)**: The neurons of a DNN's layer often have different impact on the connected neurons in next layers. The NS operator switches two neurons within a layer to exchange their roles and influences for next layers.
- **Layer Deactivation (LD)**: Each layer of a DNN transforms the output of its previous layer and propagates its results to its following layers. The LD operator is a layer level mutation operator that removes a whole layer's transformation effects as if it is deleted from the DNNs. However, simply removing a layer from a trained DL model can break the model structure. We restrict the DL operator to those layers such as `Maxpooling` and `regularization`, that have consistent input and output shapes and have no parameters requiring training.
- **Layer Addition ($\mathbf{LA_m}$)**: The $\mathrm{LA_m}$ operator tries to make the opposite effects of the LD operator, by adding a layer to the DNNs. Similar to the LD operator, the $\mathrm{LA_m}$ operator works under the same conditions to avoid breaking original DNNs; besides, the $\mathrm{LA_m}$ operator also includes the duplication and insertion of copied layer after its original layers, which also requires the shape of layer input and output to be consistent.
- **Activation Function Removal ($\mathbf{AFR_m}$)**: $\mathrm{AFR_m}$ operator removes the effects of activation function of a whole layer. The $\mathrm{AFR_m}$ operator differs from the NAI operator in two perspectives: (1) $\mathrm{AFR_m}$ works on the layer level, (2) $\mathrm{AFR_m}$ removes the effects of activation function, while NAI operator keeps the activation function and tries to invert the activation status of a neuron.

V. Evaluation

We have implemented *DeepMutation*, a DL mutation testing framework including both proposed source-level and model-level mutation testing techniques baed on Keras (ver.2.1.3) [8] and Tensorflow (ver.1.5.0) [7]. The source-level mutation testing technique is implemented by Python and has two key components: *automated training data mutant generator* and *Python training program mutant generator* (see Figure 3 and Table I). The model-level mutation testing automatically analyzes a DNN's structure and uses our defined operators to mutate on a copy of the original DNN. Then the generated mutant models are serialized and stored as `.h5` file format. The weight-level and neuron-level mutation operators (see Table II) are implemented through mutating the randomly selected portion of the DNN's weight matrix elements. The implementation of layer-level mutation operators are more complex. We first analyze the whole DNN's structure to identify the candidate layers of the DNN that satisfy the layer-level mutation condition. Then, we construct a new DL mutant model based on the original DL model through the functional interface of Keras and Tensforflow [28].

In order to demonstrate the usefulness of our proposed mutation testing technique, we evaluated the implemented mutation testing framework on two practical datasets and three DL model architectures, which will be explained in the rest of this section.

A. Subject Dataset and DL Models

We selected two popular publicly available datasets MNIST [36] and CIFAR-10 [37] as the evaluation subjects. MNIST is for handwritten digit image recognition, containing 60,000 training data points (*i.e.*, image data) and 10,000 test data points, with a total number of 70,000 data points in 10 classes (digits from 0 to 9). CIFAR-10 dataset is a collection of images for general purpose image classification, including 50,000 training data and 10,000 test data in 10 different classes (*e.g.*, airplanes, cars, birds, and cats).

For each dataset, we study popular DL models [20], [34], [35] that are widely used in previous work. Table III summarizes the structures and complexity of the studied DNNs, as well as the prediction accuracy we obtained through the training of original dataset and training programs. The studied DL models A, B and C contain $\underline{107,786}$, $\underline{694,402}$, and $\underline{1,147,978}$

TABLE III
EVALUATION SUBJECT DATASET AND DL MODELS. OUR SELECTED SUBJECT DATASET MNIST AND CIFAR-10 ARE POPULAR AND WIDELY STUDIED IN PREVIOUS WORK. WE TRAIN THE DNNS MODEL WITH ITS CORRESPONDING ORIGINAL TRAINING DATA AND TRAINING PROGRAM. THE OBTAINED DL MODEL REFERS TO THE ORIGINAL DL (*i.e.*, THE DL MODEL M IN FIGURE 3 AND 5), WHICH WE USE AS THE BASELINE IN OUR EVALUATION. EACH STUDIED DL MODEL STRUCTURE AND THE OBTAINED ACCURACY ARE SUMMARIZED IN THE TABLE.

MNIST		CIFAR-10
A (LeNet5)[20]	B[34]	C[35]
Conv(6,5,5)+Relu	Conv(32,3,3)+Relu	Conv(64,3,3)+Relu
MaxPooling (2,2)	Conv(32,3,3)+Relu	Conv(64,3,3)+Relu
Conv(16,5,5)+Relu	MaxPooling(2,2)	MaxPooling(2,2)
MaxPooling(2,2)	Conv(64,3,3)+Relu	Conv(128,3,3)+Relu
Flatten()	Conv(64,3,3)+Relu	Conv(128,3,3)+Relu
FC(120)+Relu	MaxPooling(2,2)	MaxPooling(2,2)
FC(84)+Relu	Flatten()	Flatten()
FC(10)+Softmax	FC(200)+Relu	FC(256)+Relu
	FC(10)+Softmax	FC(256)+Relu
		FC(10)
#Train. Para. 107,786	694,402	1,147,978
Train. Acc. 97.4%	99.3%	97.1%
Test. Acc. 97.0%	98.7%	78.3%

trainable parameters, respectively. The trainable parameters of DNNs are those parameters that could be adjusted during the training process for higher learning performance. It is often the case that the more trainable parameters a DL model has, the more complex a model would be, which requires higher training and prediction effort. We follow the training instructions of the papers [20], [34], [35] to train the original DL models. Overall, on MNIST, model A achieves 97.4% training accuracy and 97.0% test accuracy; model B achieves 99.3% and 98.7%, comparable to the state of the art. On CIFAR-10, model C achieves 97.1% training accuracy and 78.3% test accuracy, similar to the accuracy given in [35].

Based on the selected datasets and models, we design experiments to investigate whether our mutation testing technique is helpful to evaluate the quality and provide feedback on the test data. To support large scale evaluation, we run the experiments on a high performance computer cluster. Each cluster node runs a GNU/Linux system with Linux kernel 3.10.0 on a 18-core 2.3GHz Xeon 64-bit CPU with 196 GB of RAM.

B. Controlled Dataset and DL Mutant Model Generation

1) Test Data: The first step of the mutation testing is to prepare the test data for evaluation. In general, a test dataset is often independent of the training dataset, but follows the same probability distribution as the training dataset [38], [39]. A good test dataset should be comprehensive and covers diverse functional aspects of DL software use-case, so as to assess performance (*i.e.*, generalization) and reveal the weakness of a fully trained DL model. For example, in the autonomous driving scenario, the captured road images and signals from camera, LIDAR and Infrared sensors are used as inputs for DL software to predict the steering angle and braking/acceleration control [40]. A good test dataset should contain a wide range of driving cases that could occur in practice, such as strait road, curve road, different road surface conditions and weather conditions. If a test dataset only covers limited testing scenarios, good performance on the test dataset does not mean that the DL software has been well tested.

TABLE IV
THE CONTROLLED EXPERIMENT DATA PREPARATION SETTINGS.

Controlled Data Set	MNIST/CIFAR-10			
	Setting 1		Setting 2	
	Group 1	Group 2	Group 1	Group 2
Source	Train. data	Train. data	Test data	Test data
Sampling	Uniform	Non-uniform	Uniform	Non-uniform
#Size	5000	5000	1000	1000

To demonstrate the usefulness of our mutation testing for the measurement of test data quality, we performed a controlled experiment on two data settings (see Table IV). Setting one samples $5,000$ data from original training data while setting two sampled $1,000$ from the accompanied test data, both of which take up approximately 10% of the corresponding dataset. Each setting has a pair of dataset (T_1, T_2), where T_1 is uniformly sampled from all classes and T_2 is non-uniformly sampled[3]. The first group of each setting covers more comprehensive use-case of the DL software of each class, while the second group of dataset mainly focuses on a single class. It is expected that T_1 should obtain a higher mutation score, and we check whether our mutation testing confirms this. We repeat the data sampling for each setting five times to counter randomness effects during sampling. This allows to obtain five pairs of data for each setting (*i.e.*, $(T_1, T_2)_1$, $(T_1, T_2)_2$, $\ldots$, $(T_1, T_2)_5$). Each pair of data will be evaluated on the generated DL mutant models, and we average the mutation testing analysis results.

After the candidate data are prepared for mutation testing, they are executed on each of corresponding original DL models to filter out those failed cases, and only the passed data are used for further mutation testing analysis. This procedure generates a total of 30 (=2 setting * 3 models * 5 repetition) pairs of candidate datasets, where each of the three DL models has 10 pairs (*i.e.*, 5 pairs for each setting) of dataset for analysis.

2) DL Mutant Model Generation: After preparing the controlled datasets, we start the mutation testing procedure. One key step is to generate the DL mutant models. For each studied DL model in Table III, we generate the DL mutant models using both the source-level and model-level DL mutant generators.

To generate source-level DL mutant models, we configure our data-level mutation operators to automatically mutate 1% of original training data and apply program-level mutation operator to the training program (see Table I). After the mutant dataset (program) are generated, they are trained on the original training program (training data) to obtain the mutant DL models. Considering the intensive training effort, we configure to generate 20 DL mutants for each data-level mutation operator (*i.e.*, 10 for global level and 10 for local

[3]To be specific, we prioritize to select one random class data with 80% probability, while data from other classes share the remaining 20% chance.

TABLE V

THE AVERAGE ERROR RATE OF CONTROLLED EXPERIMENT DATA ON THE DL MUTANT MODELS. WE CONTROL THE SAMPLING METHOD AND DATA SIZE TO BE THE SAME, AND LET THE DATA SELECTION SCOPE AS THE VARIABLE. THE FIRST GROUP SAMPLE DATA FROM ALL CLASSES OF ORIGINAL PASSED TEST DATA, WHILE THE SECOND GROUP SAMPLE DATA FROM A SINGLE CLASS.

Model	Source Level (%)				Model Level (%)			
	5000 train.		1000 test.		5000 train.		1000 test.	
Samp.	Uni.	Non.	Uni.	Non.	Uni.	Non.	Uni.	Non.
A	2.43	0.13	0.23	0.17	4.55	4.30	4.38	4.06
B	0.49	0.28	0.66	0.21	1.67	1.56	1.55	1.47
C	3.84	2.99	17.20	13.44	9.11	7.34	11.48	9.00

level). For program-level mutators, we try to perform mutation whenever the conditions are satisfied with a maximal 20 mutant models for each program-level operator.

To generate model-level mutants at the weight and neuron level, we configure to sample 1%, weights and neurons from the studied DNNs, and use the corresponding mutation operators to randomly mutate the selected targets (see Table II). On the layer level, our tool automatically analyzes the layers that satisfy the mutation conditions and applies the corresponding a mutation operator. The model-level mutant generation is rather efficient without the training effort. Therefore, for each weight- and neuron-level mutation operator we generate 50 mutant models. Similarly, our tool tries to generate layer-level mutant models when DNN's structure conditions are satisfied with a maximal 50 mutant models for each layer-level mutation operator.

C. Mutation Testing Evaluation and Results

After the controlled datasets and mutant models are generated, the mutation testing starts the execution phase by running candidate dataset on mutant models, after which we calculate the mutation score and average error rate (AER) for each dataset. Note that the dataset used for evaluation are those data that passed on original DL models. In addition, we also introduce a quality control procedure for generated mutant models. After we obtained the passed test data T' on the original model (see Figure 3), we run it against each of its corresponding generated mutant models, and we remove those models with high error rate[4], as such mutant model show big behavioral difference from original models.

Table V summarizes the AER obtained for each controlled dataset on all DL mutant models. We can see that most of the AER are relatively small. This confirms that the obtained DL mutant models indeed enable to simulate the minor faults of DL models. In all the experimentally controlled data settings, the uniformly sampled data group achieves higher average error rate on the mutant models, which indicates the uniformly sampled data has higher defect detection ability (better quality from testing perspective). For model C, when considering both source-level and model-level, a relatively low AER is

[4]In this study, we set the error rate bar to be 20%. It could be configured to be even smaller to only keep DL mutants that have even more similar behaviors with original models.

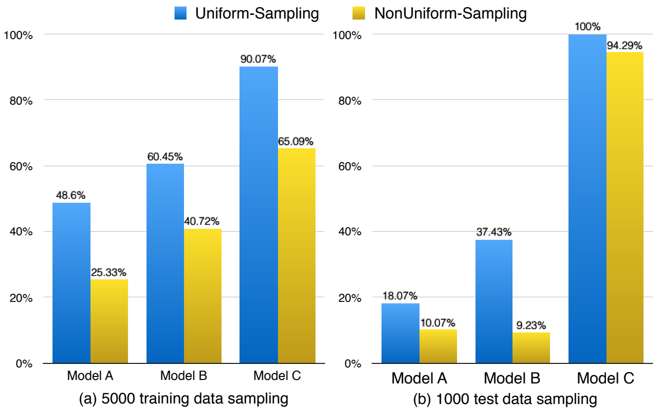

Fig. 6. The averaged mutation score results of source-level mutation testing.

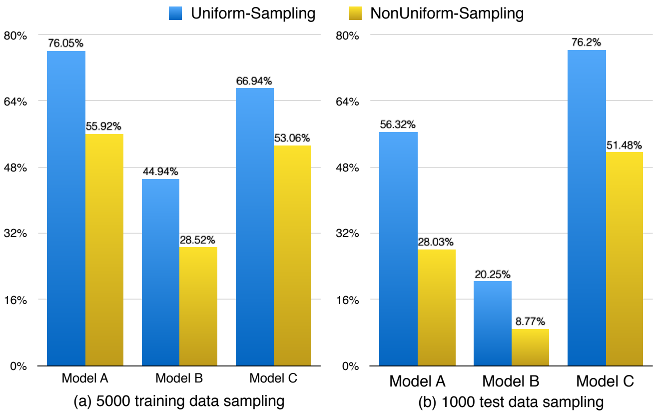

Fig. 7. The averaged mutation score results of model-level mutation testing.

obtained for the sampled training data sets from 2.99% up to 9.11%, but with a higher AER of sampled testing data from 9.00% to 17.20%. This means that the sampled test data quality of model C could be better compared with the sampled training data, although the sampled training data has larger data size (*i.e.*, 5,000).

In line with the AER, the averaged mutation score for each setting (see Table IV) is also calculated, as shown in Figure 6 and 7. Again, on all the controlled data pair settings, a higher mutation score is obtained by uniform sampling method, which also confirms our expectation on the test data quality. Besides the AER that measures the ratio of data that can detect the defects of mutant models, mutation score measures how well the test data cover mutation models from the testing use-case diversity perspective. The mutation score does necessarily has a positive correlation with the AER, which is demonstrated in the next section.

Intuitively, a test dataset with more data might uncover more defects and testing aspects. However, this is not generally correct as confirmed in our experiment. In Table V, for source-level mutation testing of model B, the obtained AER of $1,000$ uniformly sampled test data (*i.e.*, 0.66%) is higher than the one obtained from the uniformly sampled $5,000$ training data (*i.e.*, 0.49%). This is more obvious on model C. When the same sampling method are used, the AER obtained from the sampled 1000 test data is all higher than the sampled 5,000 training data. The same conclusion could also be reached by observing the mutation score (see Figure 6(a) and (b)). The mutation scores on model A and B are the cases where a larger data size obtains a higher mutation score, whereas the result on model C shows the opposite cases.

When performed on the same set of data, the source-level mutation testing and model-level mutation testing show some different behaviors. Note that, on source-level, we configure to mutate 1% of the training data; on the model-level, we use the same ratio (*i.e.*, 1%) for weight and neuron level mutators. Overall, the generated mutant models by source-level and model-level mutation testing also behave differently. For example, comparing the same data pair setting of Figures 6(a) and 7(a), the source-level mutation testing obtains lower mutation score on model A, but obtains higher mutation score on model B. This means that the same 1% mutation ratio results in different DL mutant model effects by source-level and model-level mutation testing procedure. For flexibility, in our tool, we provide the configurable option for both source-level and model-level mutant generation.

D. Mutation Testing of Original Test Data by Class

Given a DL classification task, the developers often prepare the test data with great care. On one hand, they try to collect data from diverse classes that cover more use-case scenarios. On the other hand, they also try to obtain more sensitive data for each class to cover its corner case. The same test dataset might show different testing performance on different DL models; the data from different classes of the same test data might contribute differently to testing performance as well. In this section we investigate how each class of the original test dataset behaves from the mutation testing perspective.

1) Test Data and Mutant Models: Similar to the experimental procedure in Section V-B ,we first prepare the test data of each class for mutation testing. For the accompanied original test data in MNIST (CIFAR-10), we separate them into the corresponding 10 test dataset by class (*i.e.*, $t_1, t_2, \ldots, t_{10}$). For each class of the test data t_i, we follow the same mutation testing procedure to perform data filtering procedure on model A, B and C, respectively. In the end, we obtain 30 test datasets, including 10 datasets by class (*i.e.*, $t'_1, t'_2, \ldots, t'_{10}$) for each studied DL model. We reuse the generated model-level DL mutant models of Section V-B and perform mutation testing on the prepared dataset.

2) Mutation Testing Results of Test Data by Class: Table VI summarizes the obtained mutation score and AER for each model. We can see that, in general, the test data of different classes obtain different mutation scores and AER. Consider the results of model A as an example, the test data of class 3 obtains the lowest mutation score and AER (*i.e.*, 6.25% and 1.48%). It indicates that, compared with the test data of other classes, the test data of class 3 could still be further enhanced. In addition, this experiment demonstrates that a higher AER

TABLE VI

THE MODEL-LEVEL MUTATION TESTING SCORE AND AVERAGE ERROR RATE OF TEST DATA BY CLASS. ACCORDING TO OUR MUTATION SCORE DEFINITION, THE MAXIMAL MUTATION SCORE THAT TEST DATA OF A SINGLE CLASS COULD ACHIEVE IS 10%.

M.	Eval.	Classification Class (%)									
		0	1	2	3	4	5	6	7	8	9
A	mu.	7.22	8.75	9.03	6.25	8.75	8.19	8.75	9.17	9.72	9.03
	avg.err	3.41	3.50	1.81	1.48	4.82	2.52	5.50	4.25	10.45	3.11
B	mu.	1.59	3.29	8.29	7.44	5.49	4.02	8.17	3.66	5.85	8.41
	avg.err.	0.41	1.42	1.12	1.55	1.07	2.92	2.95	1.21	1.24	2.11
C	mu.	8.33	7.95	8.97	9.74	9.74	9.62	9.62	8.97	9.74	7.56
	avg.err.	3.67	6.22	14.80	8.84	9.11	11.53	6.83	11.48	8.87	8.55

does not necessarily result in a higher mutation score. For model A, the AER obtained by class 1 is larger than class 2 while the mutation score of class 1 is smaller.

In summary, our mutation testing enables the quantitative analysis on test data quality of each class. Based on the mutation testing feedback, the DL developers could prioritize to augment and enhance the weak test data to cover more defect-sensitive cases.

E. Threats To Validity

The selection of the subject datasets and DL models could be a threat to validity. In this paper, we try to counter this issue by using two widely studied datasets (*i.e.*, MNIST and CIFAR-10), and DL models with different network structures, complexities, and have competitive prediction accuracy. Another threat to validity could be the randomness in the procedure of training source-level DL mutant models. The TensorFlow framework by default uses multiple threads for training procedure, which can cause the same training dataset to generate different DL models. To counter such effects, we tried our best to rule out non-deterministic factors in training process. We first fix all the random seeds for training programs, and use a single thread for training (by setting Tensorflow parameter `intra_op_parallelism_threads` and `inter_op_parallelism_threads`). We found that such a setting can make the training progress deterministic when running on CPU, which still has non-deterministic behavior when running on GPU. Therefore, for the controlled evaluation described in this paper, we performed all the source-level DL mutant model training by CPU to reduce the threat caused by randomness factor in training procedure. Another threat is the randomness during our data sampling. To counter this factor, we repeat the sampling and experiment procedure five times, and take the average of the corresponding evaluation results.

VI. RELATED WORK

A. Mutation Testing of Traditional Software

The history of mutation testing dated back to 1971 in Richard Liption's Paper [15], and the field started to grow with DeMillo *et al.* [41] and Hamlet [42] pioneering works in late 1970s. Afterwards, mutation testing has been extensively studied for traditional software, which has been proved to be a useful methodology to evaluate the effectiveness of test data. As a key component in mutation testing procedure, mutation operators are widely studied and designed for different programming languages. Budd *et al*. was the first to design mutation operators for Fortran [43], [44]. Arawal *et al*. later proposed a set of 77 mutation operators for ANSI C [45]. Due to the fast development of programming languages that incorporates many features (*e.g*., Object Oriented, Aspect-Oriented), mutation operators are further extended to cover more advanced features in popular programming languages, like Java [46], [47], C# [48], [49], SQL [50], and AspectJ [51]. Such work on mutation testing is most relevant to ours. Different from traditional software, DL defines a novel data-driven programming paradigm with different software representations, causing the mutation operators defined for traditional software unable to be directly applied to DL based software. To the best of our knowledge, our work *DeepMutation* is the first to propose mutation testing frameworks for DL systems, with the design of both source-level and model-level mutation operators.

Besides the design of mutation operators, great efforts have also been devoted to other key issues of mutation testing, such as theoretical aspects [52], [53], [54] of mutation testing, performance enhancement [55], [56], [57], [58], [15], platform and tool support [59], [60], [61], [27], as well as more general mutation testing applications for test generation [56], [62], [63], networks [64], [65]. We refer interesting readers to a recent comprehensive survey on mutation testing [6].

B. Testing and Verification of DL Systems

Testing. Testing machine learning systems mainly relies on probing the accuracy on test data which are randomly drawn from manually labeled datasets and *ad hoc* simulations [66]. DeepXplore [11] proposes a white-box differential testing algorithm to systematically generate adversarial examples that cover all neurons in the network. By introducing the definition of neuron coverage, they measure how much of the internal logic of a DNN has been tested. DeepCover [67] proposes the test criteria for DNNs, adapted from the MC/DC test criteria [68] of traditional software. Their test criteria have only been evaluated on small scale neuron networks (with only `Dense` layers, and at most 5 hidden layers, and no more than 400 neurons). The effectiveness of their test criteria remain unknown on real-world-sized DL systems with multiple types of layers. Most recently, DeepGauge [69] proposes multi-granularity testing coverage for DL systems, which is based on the observation of DNNs' internal state. Their testing criteria have been shown to be effective as the guidance criteria for test generation, which is also scalable to complex DNNs like ResNet-50 (with hundrunds of layers and near 100,000 neurons).

Verification. Another interesting avenue is to provide reliable guarantees on the security of deep learning systems by formal verification. The abstraction-refinement approach in [70] verifies safety properties of a neural network with 6 neurons. DLV [71] enables to verify local robustness of deep neural networks. Reluplex [72] adopts an SMT-based approach that verifies safety and robustness of deep neural networks with ReLu activation functions. Reluplex has demonstrated its usefulness on a network with 300 ReLu nodes in [72]. DeepSafe [73] uses Reluplex as its underlying verification component to identify safe regions in the input space. AI^2 [74] proposes the verification of DL systems based on abstract interpretation, and designs the specific abstract domains and transformation operators. VERIVIS [75] is able to verify safety properties of deep neural networks when inputs are modified through given transformation functions. The transformation functions given in [75] are much simpler than what real-world transformations can do to inputs.

The existing work of formal verification shows that formal technique for DNNs is promising [71], [70], [72], [73], [74], [75]. However, most of current formal verification techniques have been demonstrated only on simple DNNs network architectures. Enhancing the performance of the existing verification techniques and designing more scalable verification methods towards complex real-world DNNs would be important research directions.

DeepMutation originally proposes to use mutation testing to systematically evaluate the quality of tests of DNNs, which is orthogonal to these existing testing and verification techniques. We believe that DeepMutation enables to facilitate the design of more advanced testing and verification techniques, and we leave such studies in our future work.

VII. Conclusion and Future Work

In this paper, we have studied the usefulness of mutation testing techniques for DL systems. We first proposed a source-level mutation testing technique that works on training data and training programs. We designed a set of source-level mutation operators that mimic the faults introduced during the DL development process. In addition, we also proposed a model-level mutation testing technique and designed a set of mutation operators that directly inject faults into DL models. Furthermore, we proposed the mutation testing metrics to measure the quality of test data. We implemented the proposed mutation testing framework *DeepMutation* and demonstrated its usefulness on two popular datasets MNIST and CIFAR-10 with three DL models.

Mutation testing has gained great success in traditional software, and has also been widely applied to many application domains. We believe that mutation testing is a promising technique that could greatly facilitate the DL developer to create higher quality test data. The high quality test data would provide more comprehensive feedback and guidance for further in-depth understanding and constructing DL systems. In future work, we plan to propose more advanced mutation operators to cover more diverse aspects of DL systems, and also investigate new approaches to applying mutation testing to the automated testing, attack, and defense of DL systems.

REFERENCES

[1] P. Holley, "Texas becomes the latest state to get a self-driving car service," https://www.washingtonpost.com/news/innovations/wp/2018/05/07/texas-becomes-the-latest-state-to-get-a-self-driving-car-service/?noredirect=on&utm_term=.924daa775616, 2018.

[2] F. Zhang, J. Leitner, M. Milford, B. Upcroft, and P. Corke, "Towards vision-based deep reinforcement learning for robotic motion control," *arXiv:1511.03791*, 2015.

[3] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[4] A. Karpathy, G. Toderici, S. Shetty, T. Leung, R. Sukthankar, and L. Fei-Fei, "Large-scale video classification with convolutional neural networks," in *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, 2014, pp. 1725–1732.

[5] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *ICLR*, 2015.

[6] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Sep. 2011.

[7] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283.

[8] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2015.

[9] A. Gibson, C. Nicholson, J. Patterson, M. Warrick, A. D. Black, V. Kokorin, S. Audet, and S. Eraly, "Deeplearning4j: Distributed, open-source deep learning for Java and Scala on Hadoop and Spark," 5 2016. [Online]. Available: https://figshare.com/articles/deeplearning4j-deeplearning4j-parent-0_4-rc3_8_zip/3362644

[10] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le *et al.*, "Large scale distributed deep networks," in *Advances in neural information processing systems*, 2012, pp. 1223–1231.

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

[12] A. Authors, "DeepGauge," https://deepgauge.github.io/, 2018.

[13] Y. Sun, X. Huang, and D. Kroening, "Testing Deep Neural Networks," *ArXiv e-prints*, Mar. 2018.

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

[15] A. J. Offutt and R. H. Untch, *Mutation 2000: Uniting the Orthogonal*. Boston, MA: Springer US, 2001, pp. 34–44.

[16] K. N. King and A. J. Offutt, "A fortran language system for mutation-based software testing," *Softw. Pract. Exper.*, vol. 21, no. 7, pp. 685–718, Jun. 1991. [Online]. Available: http://dx.doi.org/10.1002/spe.4380210704

[17] Y.-S. Ma, J. Offutt, and Y. R. Kwon, "Mujava: An automated class mutation system: Research articles," *Softw. Test. Verif. Reliab.*, vol. 15, no. 2, pp. 97–133, Jun. 2005. [Online]. Available: http://dx.doi.org/10.1002/stvr.v15:2

[18] J. Offutt, P. Ammann, and L. L. Liu, "Mutation testing implements grammar-based testing," in *Proceedings of the Second Workshop on Mutation Analysis*, ser. MUTATION '06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 12–. [Online]. Available: https://doi.org/10.1109/MUTATION.2006.11

[19] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ser. ISSTA 2014. New York, NY, USA: ACM, 2014, pp. 437–440. [Online]. Available: http://doi.acm.org/10.1145/2610384.2628055

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

[21] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," *Master's thesis, Department of Computer Science, University of Toronto*, 2009.

[22] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "ImageNet Large Scale Visual Recognition Challenge," *IJCV*, vol. 115, no. 3, pp. 211–252, 2015.

[23] "Cosmic Ray: mutation testing for Python," https://github.com/sixty-north/cosmic-ray/.

[24] "MutPy is a mutation testing tool for Python," https://github.com/mutpy/mutpy/.

[25] A. Derezinska and K. Hałas, "Improving mutation testing process of python programs," in *Software Engineering in Intelligent Systems*, R. Silhavy, R. Senkerik, Z. K. Oplatkova, Z. Prokopova, and P. Silhavy, Eds., 2015, pp. 233–242.

[26] R. Just, "The Major mutation framework: Efficient and scalable mutation analysis for Java," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, San Jose, CA, USA, July 23–25 2014, pp. 433–436.

[27] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque, "Pit: A practical mutation testing tool for java (demo)," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ser. ISSTA 2016. New York, NY, USA: ACM, 2016, pp. 449–452. [Online]. Available: http://doi.acm.org/10.1145/2931037.2948707

[28] C. Francois, *Deep Learning with Python*. Manning Publications Company, 2017.

[29] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," *arXiv preprint arXiv:1708.08559*, 2017.

[30] M. Becker, C. Kuznik, M. M. Joy, T. Xie, and W. Mueller, "Binary mutation testing through dynamic translation," in *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2012)*, June 2012, pp. 1–12.

[31] M. Becker, D. Baldin, C. Kuznik, M. M. Joy, T. Xie, and W. Mueller, "Xemu: An efficient qemu based binary mutation testing framework for embedded software," in *Proceedings of the Tenth ACM International Conference on Embedded Software*, ser. EMSOFT '12. New York, NY, USA: ACM, 2012, pp. 33–42. [Online]. Available: http://doi.acm.org/10.1145/2380356.2380368

[32] V. Nair and G. E. Hinton, "Rectified linear units improve restricted boltzmann machines," in *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ser. ICML'10. USA: Omnipress, 2010, pp. 807–814. [Online]. Available: http://dl.acm.org/citation.cfm?id=3104322.3104425

[33] A. L. Maas, A. Y. Hannun, and A. Y. Ng, "Rectifier nonlinearities improve neural network acoustic models," in *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013.

[34] C. Xiao, B. Li, J.-Y. Zhu, W. He, M. Liu, and D. Song, "Generating Adversarial Examples with Adversarial Networks," *ArXiv e-prints*, Jan. 2018.

[35] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *Security and Privacy (SP), IEEE Symposium on*, 2017, pp. 39–57.

[36] Y. LeCun and C. Cortes, "The MNIST database of handwritten digits," 1998.

[37] N. Krizhevsky, H. Vinod, C. Geoffrey, M. Papadakis, and A. Ventresque, "The cifar-10 dataset," http://www.cs.toronto.edu/kriz/cifar.html, 2014.

[38] B. D. Ripley and N. L. Hjort, *Pattern Recognition and Neural Networks*, 1st ed. New York, NY, USA: Cambridge University Press, 1995.

[39] C. M. Bishop, *Neural Networks for Pattern Recognition*. New York, NY, USA: Oxford University Press, Inc., 1995.

[40] S. Thrun, "Toward robotic cars," *Commun. ACM*, vol. 53, no. 4, pp. 99–106, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1721654.1721679

[41] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, Apr. 1978. [Online]. Available: http://dx.doi.org/10.1109/C-M.1978.218136

[42] R. Hamlet, "Testing programs with the aid of a compiler," *IEEE Transactions on Software Engineering*, vol. 3, pp. 279–290, 07 1977. [Online]. Available: doi.ieeecomputersociety.org/10.1109/TSE.1977.231145

[43] T. A. Budd and F. G. Sayward, "Users guide to the pilot mutation system," Yale University, New Haven, Connecticut, techreport 114, 1977.

[44] T. A. Budd, R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "The design of a prototype mutation system for program testing," in *Proceedings of*

the AFIPS National Computer Conference, Anaheim, New Jersey, 5-8 June 1978, pp. 623–627.

[45] H. Agrawal, R. A. DeMillo, B. Hathaway, W. Hsu, W. Hsu, E. W. Krauser, R. J. Martin, A. P. Mathur, and E. Spafford, "Design of mutant operators for the c programming language," Purdue University, West Lafayette, Indiana, techreport SERC-TR-41-P, March 1989.

[46] S. Kim, J. A. Clark, and J. A. McDermid, "Investigating the effectiveness of object-oriented testing strategies using the mutation method," in *Proceedings of the 1st Workshop on Mutation Analysis (MUTATION'00)*, San Jose, California, 6-7 October 2001, pp. 207–225.

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

[48] A. Derezińska, "Advanced mutation operators applicable in c# programs," Warsaw University of Technology, Warszawa, Poland, techreport, 2005.

[49] ——, "Quality assessment of mutation operators dedicated for c# programs," in *Proceedings of the 6th International Conference on Quality Software (QSIC'06)*, Beijing, China, 27-28 October 2006.

[50] W. K. Chan, S. C. Cheung, and T. H. Tse, "Fault-based testing of database application programs with conceptual data model," in *Proceedings of the 5th International Conference on Quality Software (QSIC'05)*, Melbourne, Australia, 19 -20 September 2005, pp. 187–196.

[51] F. C. Ferrari, J. C. Maldonado, and A. Rashid, "Mutation testing for aspect-oriented programs," in *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation (ICST '08)*, Lillehammer, Norway, 9-11 April 2008, pp. 52–61.

[52] R. A. DeMillo, D. S. Guindi, K. N. King, W. M. McCracken, and A. J. Offutt, "An extended overview of the mothra software testing environment," in *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis (TVA'88)*, Banff Alberta,Canada, July 1988, pp. 142–151.

[53] A. J. Offutt, "The coupling effect: Fact or fiction," *ACM SIGSOFT Software Engineering Notes*, vol. 14, no. 8, pp. 131–140, December 1989.

[54] ——, "Investigations of the software testing coupling effect," *ACM Transactions on Software Engineering and Methodology*, vol. 1, no. 1, pp. 5–20, January 1992.

[55] T. A. Budd, "Mutation analysis of program test data," phdthesis, Yale University, New Haven, Connecticut, 1980.

[56] R. A. DeMillo and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Transactions on Software Engineering*, vol. 17, no. 9, pp. 900–910, September 1991.

[57] B. J. M. Grün, D. Schuler, and A. Zeller, "The impact of equivalent mutants," in *Proceedings of the 4th International Workshop on Mutation Analysis (MUTATION'09)*, Denver, Colorado, 1-4 April 2009, pp. 192–199.

[58] Y. Jia and M. Harman, "Constructing subtle faults using higher order mutation testing," in *Proceedings of the 8th International Working Conference on Source Code Analysis and Manipulation (SCAM'08)*, Beijing, China, 28-29 September 2008, pp. 249–258.

[59] E. W. Krauser, A. P. Mathur, and V. J. Rego, "High performance software testing on simd machines," *IEEE Transactions on Software Engineering*, vol. 17, no. 5, pp. 403–423, May 1991.

[60] A. P. Mathur and E. W. Krauser, "Mutant unification for improved

[64] D. P. Sidhu and T. K. Leung, "Fault coverage of protocol test methods," in *Proceedings of the 7th Annual Joint Conference of the IEEE Computer*

vectorization," Purdue University, West Lafayette, Indiana, techreport SERC-TR-14-P, 1988.

[61] A. J. Offutt, R. P. Pargas, S. V. Fichter, and P. K. Khambekar, "Mutation testing of software using a mimd computer," in *Proceedings of the International Conference on Parallel Processing*, Chicago, Illinois, August 1992, pp. 255–266.

[62] B. Baudry, F. Fleurey, J.-M. Jezequel, and Y. L. Traon, "Genes and bacteria for automatic test cases optimization in the .net environment," in *Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02)*, Annapolis, Maryland, 12-15 November 2002, pp. 195–206.

[63] G. Fraser and A. Arcuri, "Evosuite: Automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 416–419. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025179

and Communcations Societies (INFOCOM'88), New Orleans, Louisiana, 27-31 March 1988, pp. 80–85.

[65] C. Jing, Z. Wang, X. Shi, X. Yin, and J. Wu, "Mutation testing of protocol messages based on extended ttcn-3," in *Proceedings of the 22nd International Conference on Advanced Information Networking and Applications (AINA'08)*, Okinawa, Japan, 25-28 March 2008, pp. 667–674.

[66] I. H. Witten, E. Frank, M. A. Hall, and C. J. Pal, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2016.

[67] Y. Sun, X. Huang, and D. Kroening, "Testing Deep Neural Networks," *ArXiv e-prints*, Mar. 2018.

[68] H. Kelly J., V. Dan S., C. John J., and R. Leanna K., "A practical tutorial on modified condition/decision coverage," NASA, Tech. Rep., 2001.

[69] L. Ma, F. Juefei-Xu, J. Sun, C. Chen, T. Su, F. Zhang, M. Xue, B. Li, L. Li, Y. Liu *et al.*, "Deepgauge: Comprehensive and multi-granularity testing criteria for gauging the robustness of deep learning systems," *arXiv preprint arXiv:1803.07519*, 2018.

[70] L. Pulina and A. Tacchella, "An abstraction-refinement approach to verification of artificial neural networks," in *International Conference on Computer Aided Verification*. Springer, 2010, pp. 243–257.

[71] M. Wicker, X. Huang, and M. Kwiatkowska, "Feature-guided black-box safety testing of deep neural networks," *CoRR*, vol. abs/1710.07859, 2017. [Online]. Available: http://arxiv.org/abs/1710.07859

[72] G. Katz, C. W. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient SMT solver for verifying deep neural networks," *CoRR*, vol. abs/1702.01135, 2017. [Online]. Available: http://arxiv.org/abs/1702.01135

[73] D. Gopinath, G. Katz, C. S. Pasareanu, and C. Barrett, "Deepsafe: A data-driven approach for checking adversarial robustness in neural networks," *CoRR*, vol. abs/1710.00486, 2017. [Online]. Available: http://arxiv.org/abs/1710.00486

[74] D. D.-C. P. T. S. C. M. V. Timon Gehr, Matthew Mirman, "Ai2: Safety and robustness certification of neural networks with abstract interpretation," in *2018 IEEE Symposium on Security and Privacy (SP)*, 2018.

[75] K. Pei, Y. Cao, J. Yang, and S. Jana, "Towards practical verification of machine learning: The case of computer vision systems," *CoRR*, vol. abs/1712.01785, 2017. [Online]. Available: http://arxiv.org/abs/1712.01785