ALAGAPPA CHETTIAR GOVERNMENT COLLEGE OF ENGINEERING AND TECHNOLOGY

(An Autonomous Institution Affiliated to Anna University, Chennai)

KARAIKUDI – 630003

DeepMutation: Mutation Testing of Deep Learning Systems

TECHNICAL SEMINAR-II REPORT

Submitted by

Dhurika V

BACHELOR OF ENGINEERING IN COMPUTER SCIENCE AND ENGINEERING

MAY 2023

TABLE OF CONTENT

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 an inadequate test dataset, DL models that have achieved high test accuracy may still lack generality and robustness. In traditional software testing, mutation testing is a well-established technique for quality evaluation of test suites, which analyses to what extent a test suite detects the injected faults. However, due to the fundamental difference between traditional software and deep learning-based software, traditional mutation testing techniques cannot be directly applied to DL systems. In this paper, we propose a mutation testing framework specialized for DL systems to measure the quality of test data. To do this, by sharing the same spirit of mutation testing in traditional software, we first define a set of source-level mutation operators to inject faults to the source of DL (i.e., training data and training programs). Then we design a set of model-level mutation operators that directly inject faults into DL models without a training process. Eventually, the quality of test data could be evaluated from the analysis on to what extent the injected faults could be detected. The usefulness of the proposed mutation testing techniques is demonstrated on two public datasets, namely MNIST and CIFAR-10, with three DL models.

Index Terms—Deep learning, Software testing, Deep neural networks, Mutation testing

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

ABBREVIATION	EXPANSION
DL	DEEP LEARNING
MT	MUTATION TESTING
SUT	SOFTWARE UNDER TEST
DNN	DEEP NEURAL NETWORK

CHAPTER 1

INTRODUCTION

1.1 INTRODUCTION TO MUTATION TESTING:

Mutation testing is a powerful technique used in software testing to assess the quality of test data and locate weaknesses in software systems. It provides a systematic approach to evaluating the effectiveness of test cases by introducing artificial faults, called mutations, into the code under test. By measuring the ability of test cases to detect these mutations, mutation testing helps identify areas where the test suite may be lacking in terms of its ability to detect real faults.

The main idea behind mutation testing is to simulate small changes or faults in the original code to create mutated versions of the software. These mutations are typically simple and realistic, such as changing a comparison operator or removing a line of code. Each mutant represents a potential fault that could exist in the software. The mutated versions are then executed using the existing test suite, and the behaviour of the mutated code is compared with the behaviour of the original code.

1.2 TRADITIONAL SOFTWARE MUTATION TESTING:

Mutation testing (MT) has been widely adopted in traditional software development as a systematic technique for evaluating test data quality and identifying weaknesses. MT involves the design and selection of mutation operators, which introduce potential faults into the software under test (SUT) by creating modified versions or mutants of the SUT. The quality of test sets is assessed by measuring the ability to detect behavioural differences between mutants and the original SUT.

1.3 DL SYSTEM MUTATION TESTING APPROACH:

Unlike traditional software, where decision logic is typically implemented through code, DL systems rely heavily on the structure of Deep Neural Networks (DNNs) and the connection weights within the network. Training data sets and training programs play crucial roles in determining the behaviour of DL systems. To perform mutation testing on DL systems, it is reasonable to design mutation operators that inject potential faults into either the training data or the DNN training program. By injecting faults and re-executing the training process using the mutated data or program, a set of mutated DL models is generated. Each mutant model is then evaluated against a test set to compare its behaviour with the original DL model. A test input is considered to detect a behavioural difference between the original DL model and a mutant if their outputs are inconsistent on that input.

CHAPTER 2

LITERATURE REVIEW

1. Ma, L., et al. (2020). DeepGauge: Comprehensive and Multi-Granularity Testing Criteria for Deep Learning Systems. IEEE Transactions on Software Engineering.

This paper introduces DeepGauge, a framework that includes deep mutation as one of its testing criteria for deep learning systems. It discusses the challenges of applying mutation testing to deep learning, presents mutation operators for training data and neural network structures, and demonstrates the effectiveness of deep mutation in identifying weaknesses in test data.

2. Guo, J., et al. (2020). DeepMutation: Mutation Testing of Deep Learning Systems. IEEE Transactions on Software Engineering.

This study proposes DeepMutation, a mutation testing approach tailored specifically for deep learning systems. It presents mutation operators for various components of deep learning, including activation functions, layers, and weights. The authors conduct experiments on several deep learning models and demonstrate the efficacy of DeepMutation in assessing the quality of test data and detecting vulnerabilities.

3. Ma, L., et al. (2021). DL-Mutation: A Mutation Testing Framework for Deep Learning Systems. IEEE Transactions on Reliability.

DL-Mutation is a comprehensive mutation testing framework for deep learning systems presented in this paper. It introduces mutation operators for both training data and DNN structures. The authors conduct experiments on real-world deep learning models and show the capability of DL-Mutation in identifying faults and evaluating the quality of test data.

4. Sun, H., et al. (2020). MutDL: Mutation Testing for Deep Learning Systems. In Proceedings of the 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering.

This paper presents MutDL, a mutation testing framework designed for deep learning systems. It proposes mutation operators for data augmentation techniques, network architectures, and hyperparameters. The authors validate MutDL on various deep learning models and demonstrate its effectiveness in assessing the quality of test data and identifying weak areas.

5. Tian, Y., et al. (2020). DeepFault: A Comprehensive Deep Learning Testing Framework. IEEE Transactions on Software Engineering.

DeepFault is a testing framework that incorporates deep mutation as one of its testing techniques. This paper presents mutation operators specifically designed for deep learning systems and evaluates their effectiveness on different models and datasets. The authors highlight the importance of deep mutation in uncovering vulnerabilities and improving the robustness of deep learning systems.

CHAPTER 3

BACKGROUND

3.1 PROGRAMMING PARADIGM

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, 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 fulfil the requirements of the system. For example, in object-oriented programming, developers analyse 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 encoded in a deep neural network, represented by sets of weights fed into non-linear activation functions [15]. To obtain a DL software F for a specific task M, a DL developer needs to collect training data, which specifies the desired behaviour of F on M, and prepare a training program, which describes the structure of DNN and runtime training behaviours. The DNN is built by running the training program on the training data. The major effort for a DL developer is to prepare a set of training data and design a 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 interpreted, 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.

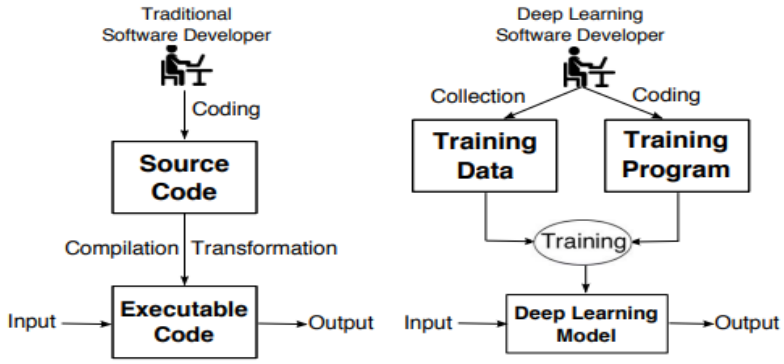

Fig. 3.1: A comparison of traditional and DL software development

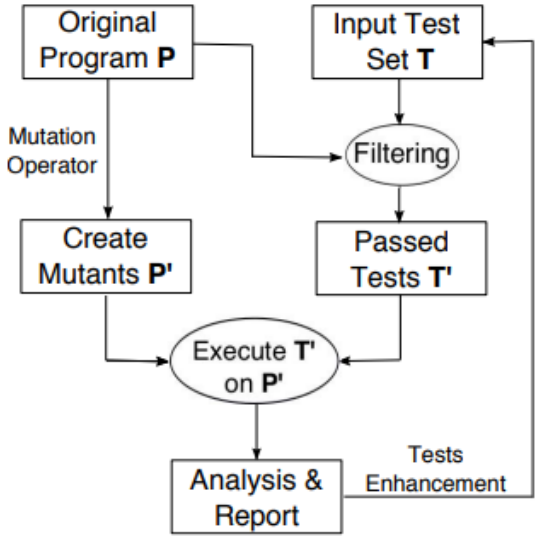

Fig. 3.2: Key process of general mutation testing.

3.2 Mutation Testing

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. A step of pre-processing, usually before the actual mutation testing procedure starts, is used to filter out irrelevant tests. Specifically, the complete test 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' ∈ P' is different from that of P, then p' is killed; otherwise, p' is survived. When all the mutants in P' have been tested against T', mutation score is calculated as the ratio of killed mutants to all the generated mutants (i.e., #mutantskilled /#mutantsall), which indicates the quality of test set. Conceptually, a test suite with a higher mutation score is more likely to capture real defects in the program. After obtaining the mutation testing results, the developer could further enhance the quality of test set (e.g., by adding/generating more tests) based on the feedback from mutation testing. The general goal of mutation testing is to evaluate the quality of test set T, and further provide feedback and guide the test enhancement.

CHAPTER 4

SOURCE-LEVEL MUTATION TESTING OF DL SYSTEMS

4.1 Source-level Mutation Testing Workflow for DL Systems

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, and the corresponding mutants D' and P' are generated. 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 analysed against the filtered test set T' for evaluating the quality of test data. We emphasize that, the proposed mutation operators in this paper are not intended to directly simulate human faults; instead, they aim to provide ways for quantitative measurement on the quality of test data set. In particular, the more behaviour differences between the original DL model and the mutant models (generated by mutation operators) T' could detect, the higher quality of T' is indicated.

4.2 Source-level Mutation Operators for DL Systems

We propose two groups of mutation operators, namely *data mutation operators* and *program mutation operators*, which perform the corresponding modification on sources to introduce potential faults.

TABLE 4.1: Source-level mutation testing operators for DL systems.

Fault Type	Level	Target	Operation Description
Data Repetition (DR)	Global Local	Data	Duplicates training data Duplicates specific type of data
Label Error (LE)	Global Local	Data	Falsify results (e.g., labels) of data Falsify specific results of data
Data Missing (DM)	Global Local	Data	Remove selected data Remove specific types of data
Data Shuffle (DF)	Global Local	Data	Shuffle selected training data Shuffle specific types of data
Noise Perturb. (NP)	Global Local	Data	Add noise to training data 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. (AFR_s)	Global	Prog.	Remove activation functions

4.3 Data Mutation Operators:

Training data plays a vital role in building DL models. The training data is usually large in size and labelled manually. Preparing training data is usually laborious and sometimes error-prone. Our data mutation operators are designed based on the observation of potential problems

that could occur 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, and the same data point can be collected more than once.

• **Label Error (LE):** Each data point (d, l) in the training dataset D, where d represents the feature data and l is the label for d. As D is often quite large (e.g., MNIST dataset [23] contains 60, 000 training data), it is not uncommon that some data points can be mislabelled. The LE operator injects such kind of faults by changing the label for a data.

• **Data Missing (DM):** The DM operator removes some of the training data. It could potentially happen by inadvertent or mistaken deletion of 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 same set of training data should obtain the same DL model. However, the implementation of training procedure 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 such problems during training.

• **Noise Perturbation (NP):** The NP operator randomly adds noise to training data. A data point could carry noise from various sources. For example, a camera-captured image could include noise caused by different weather conditions (i.e., rain, snow, dust, etc.). The NP operator tries to simulate potential issues relevant to noisy training data (e.g., NP adds random perturbations to some pixels of an image).

4.4 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, 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 prediction accuracy at the early iterations/epochs of the training). Considering the characteristics of DL training programs, we design the following operators to inject potential faults.

• **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 can generate DL models that are obviously different from the original DL model. Therefore, the LR operator mainly focuses on layers (e.g., Dense, BatchNormalization layer), whose deletion does not make too

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.

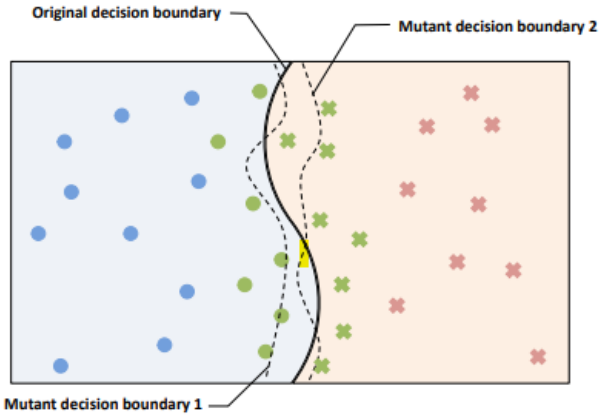

Fig. 4.2: Example of DL model and its two generated mutant models for binary classification with their decision boundaries. In the figure, some data scatter closer to the decision boundary (in green colour). Our mutation testing metrics favour to identify the test data that locate in the sensitive region near the decision boundary.

• **Layer Addition (LAs):** In contrast to the LR operator, the LAs operator adds a layer to the DNNs structure. LAs focuses on adding layers like Activation, BatchNormalization, which introduces possible faults caused by adding or duplicating a line of code representing a DNN layer

• **Activation Function Removal (AFRs):** Activation function plays an important role of the non-linearity of DNNs for higher representativeness (i.e., quantified as VC dimension [15]). The AFRs operator randomly removes all the activation functions of a layer, to mimic the situation that the developer forgets to add the activation layers.

CHAPTER 5

MODEL-LEVEL MUTATION TESTING OF DL SYSTEMS

5.1 Model-Level Mutation Testing Workflow for DL Systems

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' ∈ M' ,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 (e.g., intermediate code representation such as Java Bytecode) mutation testing techniques of traditional software.

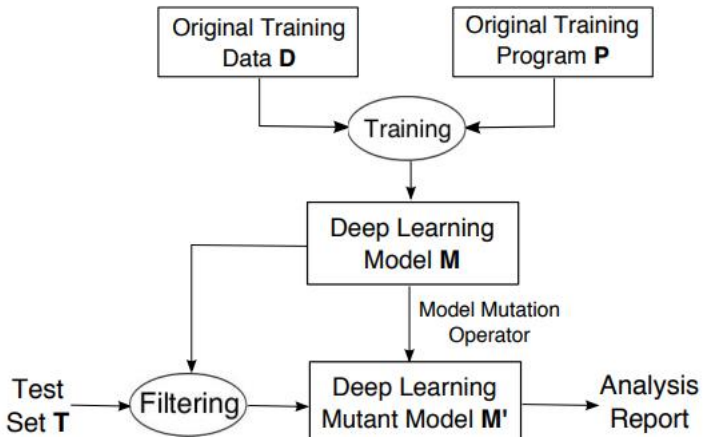

Fig. 5.1: The model level mutation testing workflow for DL systems

5.2 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 (e.g., the number of training epochs). To efficiently introduce possible faults, we further propose model-level mutation operators, which directly mutate the structure and parameters of DL models.

TABLE 5.1: 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	Invert 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 (LA_m)	Layer	Add a layer in neuron network
Act. Fun. Remov. (AFR_m)	Layer	Remove activation functions

• **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 to change the connection to fuzz its value to change the connection importance it represents. The GF operator follows the Gaussian distribution N (w, σ2) 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 − 3σ, w + 3σ] with 99.7 % probability), but also allows a weight to be changed to a greater distance with a 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 to the 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 neuron effects to all of the connected neurons in the 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 behaviours of the DNNs. Many widely used activation functions (e.g., ReLU, Leaky ReLU) show quite different behaviours 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.

• **Neuron Switch (NS):** The neurons of a DNN's layer often have different impacts on the connected neurons in the next layers. The NS operator switches two neurons within a layer to exchange their roles and influences for the 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 LD operator to layers whose the input and output shapes are consistent.

• **Layer Addition (LAm):** The LAm operator tries to make the opposite effects of the LD operator, by adding a layer to the DNNs. Similar to the LD operator, the LAm operator works under the same conditions to avoid breaking original DNNs; besides, the LAm 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 (AFRm):** AFRm operator removes the effects of activation function of a whole layer. The AFRm operator differs from the NAI operator in two perspectives: (1) AFRm works on the layer level, (2) AFRm removes the effects of activation function, while NAI operator keeps the activation function and tries to invert the activation status of a neuron.

CHAPTER 6

EVALUATION

6.1 Subject Dataset and DL Models

We selected two popular publicly available datasets MNIST and CIFAR-10 as the evaluation subjects. 105 MNIST is for handwritten digit image recognition, containing 60, 000 training data and 10, 000 test data, with a total number of 70, 000 data in 10 classes (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 that are widely used in previous work. Table III summarizes the structures and complexity of the studied DNNs, as well as the prediction accuracy obtained on our trained DNNs. The studied DL models A, B, and C contain 107, 786, 694, 402, and 1, 147, 978 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 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.

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 an 18- core 2.3GHz Xeon 64-bit CPU with 196 GB of RAM and also an NVIDIA Tesla M40 GPU with 24G.

6.2 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 a similar probability distribution as the training dataset. A good test data set 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. 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 conclude that the DL software has been well tested.

To demonstrate the usefulness of our mutation testing for the measurement of test data quality, we performed a controlled experiment on two data settings. 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 datasets (T1, T2), where T1 is uniformly sampled from all classes and T2 is non-uniformly sampled.5 The first group of each setting covers more diverse use-case of the DL software of each class, while the second group of datasets mainly focuses on a single class. It is expected that T1 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., (T1, T2)1, (T1, T2)2, ..., (T1, T2)5). Each pair of data is 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 analysis. This procedure generates a total of 30 (=2 settings * 3 models * 5 repetition) pairs of candidate datasets, where each of the three DL models has 10 pairs (i.e., 5 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. To generate source-level DL mutant models, we configure our data-level mutation operators to automatically mutate 1% of original training data and apply each of the program-level mutation operators to the training program. 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 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. On the layer level, our tool automatically analyses the layers that satisfy the mutation conditions and randomly applies the corresponding 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 maximal 50 mutant models for each layer-level mutation operator.

6.3 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 CIFAR10), 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 parameters. Such a setting enables the training progress deterministic when running on CPU, which still has non-deterministic behaviour when running on GPU. Therefore, for the controlled evaluation described in this paper, we performed 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 data sampling. To counter this, we repeat the sampling procedure five times and average the results.

CHAPTER 7

CONCLUSION AND FUTURE WORK

7.1 CONCLUSION

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 then designed a set of source-level mutation operators to inject faults that could be potentially 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.

7.2 FUTURE SCOPE

Mutation testing is a well-established technique for the test data quality evaluation in traditional software and has also been widely applied to many application domains. We believe that mutation testing is a promising technique that could facilitate DL developers to generate 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. This paper performs an initial exploratory attempt to demonstrate the usefulness of mutation testing for deep learning systems. In future work, we will perform a more comprehensive study to propose advanced mutation operators to cover more diverse aspects of DL systems and investigate the relations of the mutation operators, as well as how well such mutation operators introduce faults comparable to human faults. Furthermore, we will also investigate novel mutation testing guided automated testing, attack and defence, as well as repair techniques for DL systems.

CHAPTER 8

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.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, Sep. 1991.

8.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.

9.F. Chollet et al., “Keras,” https://github.com/fchollet/keras, 2015.

10.A. Gibson, C. Nicholson, J. Patterson, M. Warrick, A. D. Black, V. Kokorin, S. Audet, and S. Eraly, “Deeplearning4j: Distributed, opensource deep learning for Java and Scala on Hadoop and Spark,” 2016.

11.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.

12.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.

13.L. Ma, F. Juefei-Xu, J. Sun, C. Chen, T. Su, F. Zhang, M. Xue, B. Li, L. Li, Y. Liu et al., “Deepgauge: Multi-granularity testing criteria for deep learning systems,” The 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE’18), 2018.

14.Y. Sun, X. Huang, and D. Kroening, “Testing Deep Neural Networks,” ArXiv e-prints, Mar. 2018.

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