DeepMutation++: a Mutation Testing Framework for Deep Learning Systems

Qiang Hu[1], Lei Ma[1], Xiaofei Xie[2] Bing Yu[1], Yang Liu[2] and Jianjun Zhao[1,3]
[1]Kyushu University, Japan [2]Nanyang Technological University, Singapore [3]Shanghai Jiao Tong University, China

***Abstract*—Deep neural networks (DNNs) are increasingly expanding their real-world applications across domains, e.g., image processing, speech recognition and natural language processing. However, there is still limited tool support for DNN testing in terms of test data quality and model robustness. In this paper, we introduce a mutation testing-based tool for DNNs, DeepMutation++, which facilitates the DNN quality evaluation, supporting both feed-forward neural networks (FNNs) and stateful recurrent neural networks (RNNs). It not only enables static analysis of the robustness of a DNN model against the input as a whole, but also allows the identification of the vulnerable segments of a sequential input (e.g. audio input) by runtime analysis. It is worth noting that DeepMutation++ specially features the support of RNNs mutation testing. The tool demo video can be found on the project website https://sites.google.com/view/deepmutationpp.**

I. Introduction

Deep neural networks (DNNs) have achieved tremendous success in many application domains such as autonomous driving, machine translation, healthcare, and robotics. In general, the current state-of-the-art DNNs can be roughly categorized into Feed-Forward Neural Networks (FNNs) and Recurrent Neural Networks (RNNs). Both FNNs and RNNs play important roles in handling specific types of applications.

A typical FNN (e.g. fully connected neural network, convolutional neural network) processes the input information layer by layer as a forward pass until an output decision is made. FNNs are demonstrated to be effective in extracting and handling inputs with region-local features, and play as key roles in many state-of-the-art image processing applications.

an RNN (e.g., long short-term memory (LSTM), gated recurrent unit (GRU)), on the other hand, often leverages memory cells, control units, or partially propagates information backward to *capture* the temporal information of a sequential input. In particular, under the current time frame, an output of an RNN is not only determined by the given input, but also by its internal states. Such characteristics contribute to the success of RNNs in handling sequential data such as speech audio, natural language text, and financial stock prediction.

Although the quality assurance, especially the testing of DNNs is highly demanding with some initial studies available, the current testing of DNN is still at an early stage with many challenging issues ahead. Testing tool support could be an essential element to facilitate the solution exploration towards addressing such challenges.

In traditional software, mutation testing is an important technique to analyze the test data quality. A key idea is to generate a set of mutant program by injecting faults into the original program. The quality of the test data is then indicated by its ability to differentiate the behaviors of the original program and the mutants.

Following the similar spirit of mutation testing of traditional software, in this paper, we introduce DeepMutation++, a mutation testing framework for both FNNs and RNNs. DeepMutation++ incorporates eight model-level operators for FNN models introduced in DeepMutation [1] and further proposes nine new operators specialized for RNN models. In particular, to cater for the characteristics of RNNs, DeepMutation++ supports both static mutant generation to analyze the test data as a whole and dynamic mutant generation to detect the vulnerable segments of a test input at runtime. Note that a segment is a chunk of input data that is processed in an iteration of the RNN. Different from the structural coverage criteria of DNN [2], [3], DeepMutation++ enables directly provided feedback on the robustness of a DNN against an input. Intuitively, an input that is close to the decision boundary of a neural network is relatively difficult to be handled by a DNN robustly. Therefore, the analysis of the test data in differentiating the behaviors of a DNN and the generated mutants enables evaluation of both 1) the robustness of a DNN, and 2) the quality of the test data in potentially triggering the vulnerable decision behaviors of a DNN.

We demonstrate the usefulness of DeepMutation++ on two typical scenarios for DNN robustness analysis and test data vulnerable segment detection: 1) the FNN (i.e. LetNet-5) based image processing on MNIST dataset, and 2) the RNN (i.e., LSTM and GRU) based text-based sentiment analysis on IMDB dataset. We report the efficiency of our framework in DNN mutant generation as well as the robustness analysis results. We also find that the metrics defined based on mutation testing could be an important indicator of DNN robustness, with a strong correlation with the robustness of a DNN against adversarial attacks.

II. The Framework DeepMutation++

Figure 1 shows a simplified workflow overview of DeepMutation++ framework. Overall, DeepMutation++ supports both typical types of neural networks, i.e., FNNs and RNNs. Given a DNN model and a set of test data under analysis, DeepMutation++ first leverages the provided mutation operators to generate a set of high quality DNN mutants with a user specified quality threshold. After a certain number of mutants are generated, DeepMutation++ analyzes the behavior differences of the original DNN and the generated DNN mutants against

978-1-7281-2508-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ASE.2019.00126

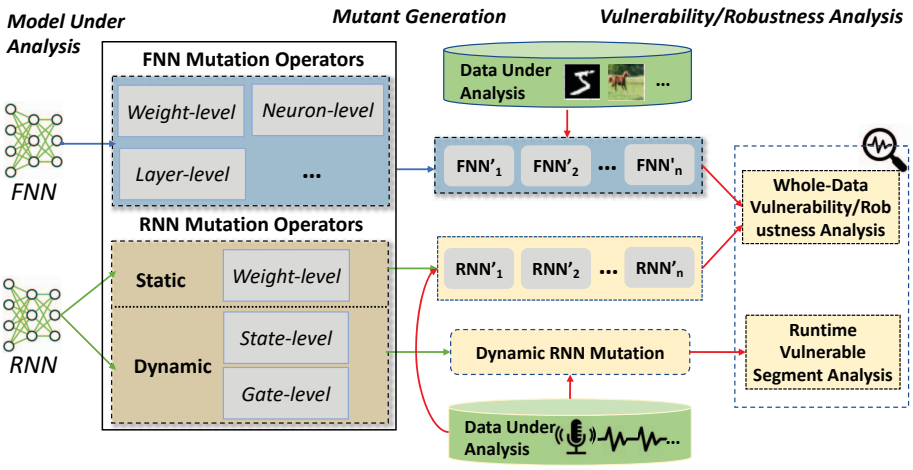

Fig. 1: The workflow overview of DeepMutation++ framework.

TABLE I: FNN Mutation operators

Level	Operator	Description
Weight	Gaussian fuzzing (GF)	fuzz weight value
	Weight Shuffle (WS)	shuffle weights
Neuron	Neuron Effect Block (NEB)	block a neuron effect *0*
	Neuron Activation Inverse (NAI)	invert the activation status
	Neuron Switch (NS)	switch two neurons of the same layer
Layer	Layer Remove (LR)	remove one layer
	Layer Addition (LA)	add a activation layer
	Layer Duplication (LD)	duplicate one layer

the provided test inputs for robustness analysis. Moreover, DeepMutation++ specially features a dynamic mode to analyze the vulnerable input segments for an RNN model based on the mutant generated on-the-fly. Finally, DeepMutation++ outputs the analysis reports that indicate test data quality and DNN robustness at the same time.

A. Mutant Generation

Mutation operator is a key component in mutation testing. DeepMutation++ supports eight FNN mutation operators and nine RNN mutation operators.

FNN mutation operators: For FNNs, DeepMutation++ incorporates the 8 proposed model-level mutation operators of DeepMutaion (see Table I). During the runtime mutation operator selection of DeepMutation++, we favor more in selecting weight-level and neuron-level operators, since we find that the three layer-level operators could often generate models below the quality threshold, increasing the time cost in mutant generation. More details for each of these operators are available in [1].

RNN mutation operators: Besides supporting FNNs, DeepMutation+ also features in supporting mutation testing of RNNs. In this paper, we focus on two typical types of RNNs, i.e., LSTM and GRU. Except for the basic weights, RNN often consists of gate unit and memory cell unit, where different gates depart model weights into multiple parts. For example, LSTM often consists of four types of units: input gate, forget gate, output gate, and memory cell unit while a typical GRU contains three types of units: reset gate, update gate, and the memory cell unit. Consider the unique characteristics and usage case of RNNs in mutation testing, DeepMutation++ supports RNN mutation operators from two levels, i.e., static level and dynamic level.

Table II summarizes the proposed RNN mutation operators. In particular, the static-level operators are used to perform offline mutant model generation. The generated mutants are stored in a user-specified location, after which the post-phase mutation analysis could be performed. However, the generated offline mutants only allow analyzing the quality of a sequential input data as a whole. To enable the input segment-level analysis, the dynamic operators mutate the internal runtime status (i.e., memory state, gate control state) of an RNN on-the-fly segment by segment. Note that the dynamic-level operators don't store mutants and they just modify the calculation way of the units in the original model to obtain the corresponding new prediction result. The descriptions of the proposed RNN mutation operators are as follows.

- **Weight Gaussian Fuzzing:** Randomly sample a user-specified ratio among all the weights of an RNN, and perform Gaussian distribution based fuzzing on the chosen weights.
- **Weight Precision Reduction:** Randomly sample a user-specified ratio of weights and reduce their precision.
- **State Clear:** Clear the run-time state value to zero to make an RNN forget its previous memory.
- **State Reset:** Reset the current run-time state to a previous state. Instead of forgetting all the memory, this operator restores the RNN to a previous state.
- **State Gaussian Fuzzing:** Follow the Gaussian distribution to mutate current run-time state.
- **State Precision Reduction:** Reduce the precision of the run-time state, i.e., reducing the floating bit size or taking the round-off results.
- **Gate Clear:** Choose a gate type and clear a run-time gate value to zero, blocking the gate at the current iteration.
- **Gate Gaussian Fuzzing:** Randomly select a gate and follow the Gaussian distribution to mutate a run-time gate value.
- **Gate Precision Reduction:** Randomly select a gate and reduce the precision of the run-time gate value.

TABLE II: RNN Mutation operators

Level		Operator	Description
Static	Weight	Weight Gaussian Fuzzing (WGF)	fuzz weights
		Weight Precision Reduction (WPR)	reduce weights' precision
Dynamic	State	State Clear (SC)	clear the state to *0*
		State Reset (SR)	reset state to previous state
		State Gaussian Fuzzing (SGF)	fuzz state value
		State Precision Reduction (SPR)	reduce state value's precision
	Gate	Gate Clear (SC)	clear the gate value to *0*
		Gate Gaussian fuzzing (GGF)	fuzz gate value
		Gate Precision Reduction (GPR)	reduce gate value's precision

B. Mutation Testing Analysis

Since the cost of mutation testing for traditional software is a major concern, we explain the time cost of DeepMutation++ in terms of mutant generation phase and post-analysis phase. The static mutant generation follows four key steps, 1) load the model, 2) parse the parameters, 3) select and mutate the parameters, and 4) store the generated mutant models. Among these steps, parameters parsing, selection and mutation are often very efficient. The time overhead is mainly due to the model loading and storing. The dynamic mutant generation perturbates runtime internal states of an RNN to analyze the potential impacts on the DNN decision.

To analyze the robustness of a model against the inputs, DeepMutation++ puts all test data which are correctly classified by the original model to each mutant to make the prediction. Different from mutation testing of traditional software, the test data prediction could be performed in batches. With GPU support, prediction on a large number of data could be completed efficiently. In addition, the prediction on different mutant could also be easily conducted as the parallel processes. For runtime analysis, since all the outputs difference are analyzed online, DeepMutation++ only needs to calculate average values as the segments' vulnerability results, therefore, the overhead of the segment level analysis is also very efficient. The detailed results of time cost analysis are on our website[1].

C. Metrics

DeepMutation++ currently supports two killing score metrics to approximate the vulnerability of an input or a segment.

Metric 1 ($KScore_1$): Given an input t, a DNN m and its mutant m', we define t is killed by m' if the outputs are inconsistent at t, i.e., $m(t) \neq m'(t)$. Given a set of mutant DNNs M, we define the killing score as: $KS_1(t, m, M) = \frac{|\{m' | m' \in M \wedge m(t) \neq m'(t)\}|}{|M|}$

Metric 2 ($KScore_2$): Given the i-th segment t_i of an input t, an RNN model m and its mutant m' that is generated by mutating t_i with dynamic-level operators. Given a set of mutant RNNs M, we define segment-level killing score as: $KS_2(t_i, m, M) = \frac{\sum_{m' \in M} ||prob(t_i, m) - prob(t_i, m')\}||_p}{|M|}$, which indicates the prediction probability divergence on the output.

$KScore_1$ is used to calculate the killing score of a whole-data while $KScore_2$ is used to calculate the killing score of a segment. We define that, for an input, the larger the value of $KScore_1$, the less robust the model against the input. For a segment of an input, the larger the value of $KScore_2$, the less robust the model against the segment.

D. Post-phase Robustness Analysis and Sorting

DeepMutation++ is able to analyze the model robustness and detect the vulnerable data or the vulnerable segments of a test data. It provides two interfaces *sort_data* and *sort_segment* to do the post-phase robustness analysis.

Given the mutants resulting path and the test data, the interface *sort_data* sorts the test data according to the mutants and metric $KScore_1$. The result calculated using *sort_data* indicates the sorting of the inputs' vulnerability from light to severe. In the end, *sort_data* outputs the analysis results as a *Numpy* file and a histogram report file. The *Numpy* file saves the $KScore_1$ results and the histogram depicts the mutants killing distribution.

For dynamic analysis, given a *CSV* file which records the dynamic mutation results, *sort_segment* sorts the segments of data according to the results and metric $KScore_2$. The result of *sort_segment* also represents the segments' vulnerability from light to severe. Finally, *sort_data* will output a *Numpy* file and a scatter report file. The *Numpy* file keeps the $KScore_2$ results and the scatter plot depicts the behavior difference at each segment before and after mutation. Note that, when using dynamic-level operators to mutate state or gate values at a segment, DeepMutation++ records the original output and the new output after mutation, based on which $KScore_2$ is calculated.

E. Implementation

DeepMutation++ is implemented in Python based on Keras2.2.4 with TensorFlow1.13 as a backend. The source code and the detailed usage description of the tool are available on our website[1]. The implementation of static mutation operators is directly on the top of Keras' APIs, e.g., $model.layer()$, $model.get\_weights()$. The RNN mutator implementation is a bit difficult since it is unable to easily control the RNN' internal state via Keras API, forcing us to have to bridge the implementation partially inside the Keras library level, e.g., *keras.layers.recurrent.py*.

III. Usage Example: Mutant DNN Generation

DeepMutation++ is current provided as a command line tool. We give one basic usage example on the usage of DeepMutaion++ to generate mutants. We put more detailed tool usage, e.g., robustness analysis, on the website.

```
python generator.py -model_path mnist.h5
-data_type mnsit -threshold 0.9 -operator 0
-ratio 0.01 -save_path mutants -num 200
-standard_deviation 0.01
```

In this case, DeepMutation++ loads the model from the path *mnist.h5* (*-model_path*) and follows the Gaussian Fuzzing operator *0* (*-operator*) and mutation ratio *0.01* (*-ratio*) to generate mutants. In order to manage the quality of the mutants, DeepMutation++ loads all the *mnits* (*-data_type*) test data to calculate mutants' accuracy, and only those mutants with accuracy higher than the threshold *0.9* (*-threshold*) is stored in the folder *mutants* (*-save_path*).

[1] https://sites.google.com/view/deepmutationpp

TABLE III: Number of LeNet-5 mutants in each $KScore_1$ range under different mutation ratio

Ratio	(0, 0.25]	(0.25, 0.5]	(0.5, 0.75]	(0.75, 1]
0.01	90	8	3	0
0.03	192	23	2	0
0.05	261	32	3	0

TABLE IV: Robustness of models at segments of inputs

RNN	Random		Spearman Correlation	
	KS_2	#E.	r_s	p
LSTM	0.09	45.3	-0.508	7.83e-18
GRU	0.23	57.18	-0.362	3.53e-09

IV. Experiments

We demonstrate two use cases of DeepMutation++ on DNN robustness analysis on whole input level and segement level. Based on the LeNet-5 model, we first use Gaussian Fuzzing operator under different mutation ratio to generate 200 mutants. Then, we perform the analysis using these mutants to calculate $KScore_1$. Table III shows the results, where each row gives the number of mutants whose score lie in the corresponding range (as in each column). We can see that, for each score range, the number of mutants tends to increase as we increase the mutation ratio. In this case, the data whose $KScore_1$ is larger than 0.5 kill the most mutants, and these data needs special attention for further analysis.

We further study the robustness of RNN models (i.e., LSTM and GRU) on IMDB sentiment analysis. We only explain the segment-level analysis and put more complete information on our site. To be specific, we randomly select 50 test inputs that are correctly handled by the original models. Suppose the RNN processes it with n iterations (i.e., n segments), for each of the n segments, we mutate the state value 100 times and obtain 100 prediction results. Then, the $KScore_2$ is computed for each segment to measure robustness. For each segment, we use FGSM adversarial attack difficulty (i.e., the number of epochs for successful attacks) as a direct indicator of model robustness against that segment. We analyze the correlation between the $KScore_2$ and the attacking difficulty (e.g., number of attacking steps). Table IV summarizes the averaged results. Column $\#KS_2$ shows the $KScore_2$ of the all input segments. Column $\#E.$ shows the average epochs used to generate an adversarial example. The statistical analysis confirms the significantly negative correlation. The results indicate that if the $KScore_2$ is small (resp. large), the model tends to be more robust (resp. vulnerable) at the segment. The results also confirm the usefulness of our mutation analysis tool for measuring the RNN model robustness at the segment level.

V. Related Work

DNN testing has been studied over the last several years [4]–[8]. However, there are limited tools combine the FNN and RNN mutation testing features as we know.

DeepXplore [2] proposes the neuron coverage criteria to measure the percentage of the activated neurons. DeepGaugen [3] further proposes multi-granularity testing coverage for deep learning systems based on the observation of DNNs' internal state. A set of combinatorial testing criteria proposed by DeepCT [9] balances the defect detection ability and a reasonable number of tests. Different from the coverage guided testing criteria, Kim et al. introduces a test adequacy criterion which measures the surprise of an input as the difference in DL system's behaviour between the input and the training data [10]. The biggest difference between DeepMutation++ and these existing works is that DeepmMutation++ is a mutation testing technique based tool, all these exiting methods can be further performed on the test data after mutation testing.

As the first mutation testing techniques for DL systems, DeepMutation [1] proposes both source-level and model-level operators to verify the quality of test data. Wang et al. [11] use the model-level mutation operators in DeepMutaion to generate mutants and detect adversary sample accordingly, this work proves the usefulness of mutation testing from the side. By contrast, DeepMutation++ not only contains all the model-level mutation operators in DeepMutation, it also supports RNN models.

VI. Conclusion

We present DeepMutation++, a tool for mutation testing of deep learning systems. DeepMutation++ supports to statically generate high-quality mutants for both FNNs and RNNs, and dynamically mutate run-time states of an RNN. We demonstrated the usage and usefulness of DeepMutation++ in analyzing DNN robustness, the vulnerable inputs, and the corresponding vulnerable segments.

Acknowledgment

This work was partially supported by 973 Program in China (No.2015CB352203), JSPS KAKENHI Grant 19K24348, 19H04086, 18H04097 and Qdaijump Research Program No.01277.

References

[1] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao *et al.*, "Deepmutation: Mutation testing of deep learning systems," in *ISSRE*, 2018, pp. 100–111.

[2] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *SOSP*. ACM, 2017, pp. 1–18.

[3] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu *et al.*, "Deepgauge: Multi-granularity testing criteria for deep learning systems," in *ASE*. ACM, 2018, pp. 120–131.

[4] A. Odena and I. Goodfellow, "Tensorfuzz: Debugging neural networks with coverage-guided fuzzing," *ICML*, 2019.

[5] X. Xie, L. Ma, F. Juefei-Xu, H. Chen, M. Xue, B. Li, Y. Liu, J. Zhao, J. Yin, and S. See, "Coverage-guided fuzzing for deep neural networks," *ISSTA*, 2019.

[6] Y. Tian, K. Pei, S. Jana, and B. Ray, "Deeptest: Automated testing of deep-neural-network-driven autonomous cars," in *ICSE*. ACM, 2018, pp. 303–314.

[7] X. Du, X. Xie, Y. Li, L. Ma, Y. Liu, and J. Zhao, "Deepstellar: model-based quantitative analysis of stateful deep learning systems," in *FSE/ESEC*. ACM, 2019, pp. 477–487.

[8] X. Xie, L. Ma, H. Wang, Y. Li, Y. Liu, and X. Li, "Diffchaser: Detecting disagreements for deep neural networks," in *IJCAI*, 2019.

[9] L. Ma, F. Juefei-Xu, M. Xue, B. Li, L. Li, Y. Liu, and J. Zhao, "Deepct: Tomographic combinatorial testing for deep learning systems," in *SANER*. IEEE, 2019, pp. 614–618.

[10] J. Kim, R. Feldt, and S. Yoo, "Guiding deep learning system testing using surprise adequacy," in *ICSE*, 2019, pp. 1039–1049.

[11] J. Wang, G. Dong, J. Sun, X. Wang, and P. Zhang, "Adversarial sample detection for deep neural network through model mutation testing," in *ICSE*, 2019, pp. 1245–1256.