MaXplore: Deep Learning Testing Inputs Generation Based on DeepXplore

Te Wei Lee
School of Information System,
Singapore Management University, Singapore
teweilee.2020@msc.smu.edu.sg

1. Introduction

Machine Learning Testing (ML testing) has become a new research area nowadays, people start to consider testing these "non-testable" systems. With different natures between Software testing and ML testing, it's inevitable for testers to resort to new set of testing strategies.

Among Machine Learning systems, computer vision systems achieve astounding performance. Successes in Deep Learning (DL) enables the model to transcend human performance in computer vision tasks. However, difficulties in testing and test inputs generation also deter users from trusting the systems. Meanwhile, various accidents have been reported from self-driving systems, lowering the system credibility. Motivated by previous research, we proposed ***MaXplore*** to address this issue. From the experiments, we empirically show the performance improvement on a new coverage, which also indicates its capability to find corner cases in Machine Learning systems.

The paper is organized as follows. we first discuss the background knowledge of DL (**Section 2**). After that, we have an overview on ML testing (**Section 3**). We also discuss some related work about ML testing inputs generation and their pros and cons (**Section 4**). Built on those concepts, we then introduce the main mechanism used in MaXplore (**Section 5**). Experiment results are shown with analytical explanations (**Section 6**). Finally, we talk about the limitations, challenges, and possible future research directions in this work (**Section 7**)

2. Background

2.1. Deep Neural Network

A Deep Neural Network (DNN) $M \in (\mathbf{L}, \mathbf{C}, \mathbf{A})$ consists of three components, which are layers (**L**), connections between layers (**C**), and activation functions (**A**). Each layer can have different neuron numbers setting, denoted by L_k with $k \in \{1, 2, ..., |\mathbf{L}|\}$. For any neuron n_{kj}, the subscript indicates its location in the DNN, where $j \in \{1, 2, ..., L_k\}$. Between layers, neurons are connected bipartedly, which means no neuron connections within a layer.

To introduce non-linearity in a DNN, we can simply add an activation function $\phi \in A$ after every hidden neuron. There are many function choices, as the final decision depends on the target distribution and empirical performance. The most popular one nowadays is called rectified linear units ($ReLU$). With an input greater than zero, a $ReLU$ is regarded as an activated neuron. Researchers have been studied about the activation patterns in a DNN, and found out that we can view them as the decision logic for prediction.

For each step in training, the DNN completes a forward pass and a back propagation. In the forward pass, each n_{kj} has an output h_{kj}, composed of the previous layer outputs.

$$a_{kj} = \sum_{l=1}^{L_{k-1}} w_{jl} \cdot h_l \tag{1}$$

$$h_{kj} = \phi(a_{kj}) \tag{2}$$

In the back propagation, we calculate errors from the final layer output and true label, using the chain rule to update the weights with gradients.

2.2. Convolutional Neural Network

Convolutional Neural Network (CNN) [1] is the state-of-the-art DNN structure that successfully learns features from image data. Each filter in a Convolutional layer convolves image data locally and represents a pattern in an image (Figure 1). With extracted features, we are able to capture image patterns in a smaller dimension.

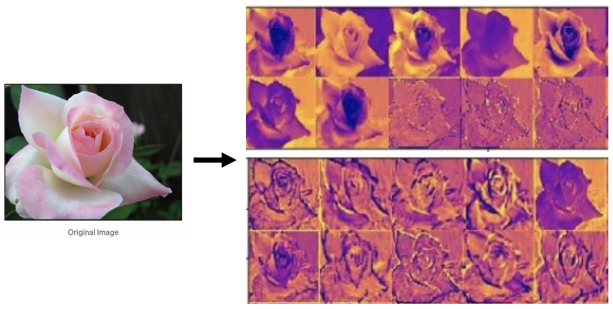


Figure 1: Convolutional layers.

3. Machine Learning Testing

Machine Learning (ML) systems have played an important role in the modern software engineering, as there are numerous applications on self-driving, computer vision, and speech processing etc. However, accidents have been reported due to ML system malfunctions, and the statistical nature makes them hard to test with traditional software testing techniques.

The ML testing research area has received attentions in recent years [2]. when building a system, we use training data with feature engineering to train the model. After model fitting and tuning, we then obtain a model with decision logic on the data. Typically, people separate a part of data for testing the model generalization. However, a mere score itself isn't enough since we often want to test on the decision logic in the model.

3.1. Differences from Software Testing

We can compare ML testing and Software testing in five aspects, which are testing components, oracles, inputs, output behaviors and criteria.

Testers set the target components and oracle before conducting a test. For testing components, we inspect primarily program codes in software testing, but in ML testing, a bug can lie in the data or learning programs. Meanwhile, in software testing, developers set expected values to verify program correctness. However, it's hard for ML tester to determine the test oracle, which is known as the *Oracle Problem*. The system correctness is usually verified by manually labeled data. Moreover, it's a domain specific issue, so testers need domain knowledge to confirm the correctness.

The inputs and outputs are different from traditional program testing. The inputs in ML testing are more diverse, as we can feed in learning programs or data to verify some desired properties. In terms of output behaviors, normally program codes return fixed outputs. Nonetheless, ML systems especially Deep Learning systems sometimes output stochastically even with the same random seed. Influenced by the framework and hardware computing interface (such as CUDA) compatibility, testers have to face with the reproduction issue.

Lastly, we can use branch coverage, line coverage for program testing. In ML testing, the testing criteria are different, and we cover some in **section 3.3**.

3.2. Testing Workflow

A typical testing workflow is consist of online testing and offline testing.

In offline testing, testers define the requirements for the ML system. Based on them, testers generate the test oracle and test inputs. The model outputs are well tracked and analyzed. If the testers discover any bug, the system need to be repaired. Otherwise, they repeat the evaluation process until a set of requirements satisfied. In the end, the system is deployed and the testers proceed to online testing.

After the model deployment, testers are able to utilize real-world inputs to verify the effectiveness, by making splits of data and getting the metric scores. if any required property isn't satisfied, they may choose to re-tune the model or construct a new model from scratch offline.

3.3. DeepGauge: ML Testing Criteria

Before getting into test inputs generation, we set a test criterion for tests evaluation and comparison. DeepGauge [3] is a multi-granularity methodology for DL testing evaluation. Due to the incompleteness of systematic criteria and the ineffectiveness with mere test scores, they proposed several metrics to compare different adversarial techniques which try to fool a DL system. If we can compare performances on adversarial or test inputs generation technique, the DL system become more robust with those inputs. Furthermore, those techniques can be guided by the metric to produce more useful inputs.

As there were multiple metrics proposed, we introduce four of them in this paper, which are *Neuron Coverage*, *k-Mulitsection Coverage* and *Strong Neuron Activation Coverage* as neuron level coverage. In layer level coverage, we discuss about the *Top-k Neuron Coverage*. In below equations, we use $\mathbf{T} = \{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_m\}$ to denote the test suite and N for total neurons. The $f(\mathbf{x}, n)$ gives us the output value of neuron n with input $\mathbf{x}$.

- *Neuron Coverage*
  Basically it's a coverage that gives use a ratio indicates how many neurons are activated for all test cases.

$$NCov = \frac{|\{n|\forall \mathbf{x} \in \mathbf{T}, f(\mathbf{x}, n) > 0\}|}{N} \tag{3}$$

- *k-Mulitsection Coverage*
  Instead of checking the neuron output greater than zero, we now cut neuron output range $[low_n, high_n]$ into k sections $S_i^n, i \in \{1, .., k\}$. We get a ratio about how many output sections are covered by the test suite.

$$kMultCov = \frac{\sum_n^N |\{S_i^n|\exists \mathbf{x} \in \mathbf{T}, f(\mathbf{x}, n) \in S_i^n\}|}{k \times N} \tag{4}$$

- *Strong Neuron Activation Coverage*
  For any neuron output within $[low_n^{train}, high_n^{train}]$ in training process, we see how many test cases explore the upper corner $U^{train} = [high_n^{train}, +\infty]$. *LayerSNACov* is the coverage in layer-level.

$$SNACov = \frac{|\{n|\exists \mathbf{x} \in \mathbf{T}, f(\mathbf{x}, n) \in U^{train}\}|}{N} \tag{5}$$

- *Top-k Neuron Coverage*
  We choose top k neurons by comparing their outputs in a layer for every test case. After combining the sets, we get the information how many neurons are once top k active neurons in a layer.

$$TopkCov = \frac{|\cup_{\mathbf{x} \in \mathbf{T}} (\cup_{1 \geq l \geq |\mathbf{L}|} top_k(\mathbf{x}, l)|}{N} \tag{6}$$

Approach Critique. DeepGauge provided criteria to evaluate testing techniques. The experiments also demonstrated how to do an evaluation analysis. However, the paper should conduct research on real bugs criteria analysis. In other words, how the neurons behave with a real buggy inputs? How it influences the criteria values? With these evidences, they could demonstrate the criteria usefulness or improve the original criteria based on the findings.

4. Related Work

We now review some research work about ML test inputs generation, and analyze their technique's strengths and drawbacks.

4.1. Concolic Testing on ML Systems

In this work [4], the authors tried to address the problem of individual discrimination in ML systems. For two individuals with all attributes the same except for the sensitive ones, such as races or genders etc, the model make different predictions, which indicates it's discriminating with bias. This is also known as a *Fairness* violation in ML testing.

The approach first came with LIME [5], which returned a linear model or a decision tree that explained the input model, by approximating outputs on localized sampled points (Figure 2). Thus, for any test input, the authors showed that we could get the model's decision logic. Similar to concolic execution, they toggled a path constraint with a heuristic, and checked whether the new input caused discrimination with different protected attribute's values.

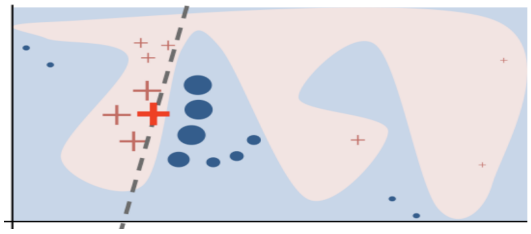


Figure 2: Locally explainable models.

Approach Critique. The approach was effective as they could find more test inputs than previous work. Also, by analyzing the new test inputs, reasoning the model discrimination became possible. However, for the drawbacks, they failed to show how to immune the model to fairness violations.

4.2. DeepConcolic

DeepConcolic [6] is another approach applying concolic execution, but in DL testing instead. A DL model contains enormous execution paths that it's infeasible to enumerate all of them. Inspired by the concolic testing success, the authors decided to adopt it in DL testing.

Their approach had three phases. First, they found the most promising test input able to improve the coverage. Let's assume we want to increase *Neuron Coverage* by activating a dormant neuron n_i. We then solve an optimization problem as below.

$$argmax_{\mathbf{x}} f(\mathbf{x}, n_i) \tag{7}$$

To understand it intuitively, although all $f(\mathbf{x}, n_i)$ are still lies below zero, we search for an input closer to zero.

After the input choice, the authors formulated a linear programming model with the activation and forward pass constraints. Here they used $a(\mathbf{x}, n_i) \in \{true, false\}$ to denote the neuron is activated or not.

$$\begin{cases} \forall n_j \in N, n_j \neq n_i & a(\mathbf{x}_{new}, n_j) = a(\mathbf{x}_{old}, n_j) \\ n_i & a(\mathbf{x}_{new}, n_i) = \neg a(\mathbf{x}_{old}, n_i) \end{cases} \tag{8}$$

Thus, after solving this problem, they obtained a new input that triggered the neuron. In the end, a test oracle was chose to verify the newly created input.

Approach Critique. The authors presented comprehensive details and flexibility for DeepConcolic, as it could optimize toward several coverage criteria. Also, they showed the DeepConcolic testing abstraction, which set a good standard for future research to follow after. Nonetheless, the approach suffers from inefficiency, since for criteria unmentioned in the paper, the testers have to redesign the heuristic. Most importantly, DL models are getting deeper and larger nowadays, so the linear program solving will become a bottleneck.

4.3. TensorFuzz

TensorFuzz [7] is a coverage-guided fuzzing technique. In coverage-guided fuzzing, when a new input improve the criterion, it's maintained in the final test suite. Since the *neuron coverage* is too easy to satisfy, they adopt the Nearest-Neighbor algorithm in their coverage analyzer component.

The approach is quite similar to DeepConcolic. First, they used a heuristic to find the most promising test input, which assigned more probabilities to recently added test inputs. After applying random changes to the input, the coverage analyzer searched for the nearest neighbor in terms of activation values from the maintained test suite. It checked whether the newly created input was further the nearest one than a given threshold, if so, it would be added. Simply put, the new input explores a new output region in the DNN.

Interestingly, they also showed that TensorFuzz was capable to capture numerical errors. In the experiment, the authors used poorly implemented loss function, which meant some inputs would cause exceptions. And TensorFuzz discovered a new input triggering the error. In addition, TensorFuzz can find many disagreements between the original 32-bit model and the quantized 16-bit version.

Approach Critique. TensorFuzz successfully showed its effectiveness with real world problems. The fast nearest neighbor implementation also resolved the possible bottleneck in the test inputs generation. Furthermore, with probability heuristic guidance, the input chooser would prefer to take the recently added input. In this way, the mutated input tended to explore more in the new output regions. Overall, this approach are easy to implement and effective.

4.4. DeepXplore

DeepXplore was a pioneer in the ML testing research area, as various work conducted experiments comparing with it. Being motivated by time-consuming test data labeling, the authors proposed an automated whitebox testing.

DeepXplore's novelty was that the authors used a gradient-based method to guide the test input generation, and trigger disagreements between several models (Figure 3). Normally, in the back propagation step, we update the DNN weight. For DeepXplore, they fixed the weight and mutated the input with gradients. Meanwhile, they also changed the objective function from the prediction loss into a new one. The F_i represents i-th model in DeepXplore.

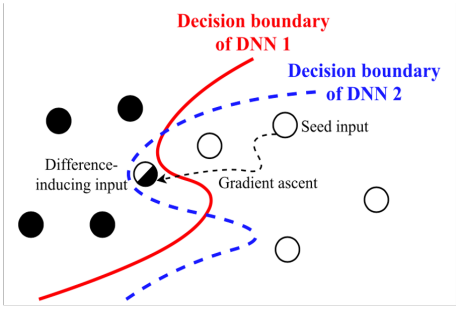


Figure 3: Disagreements between models.

$$obj = (\sum_{j \neq i} F_j(\mathbf{x})[class] - \lambda_1 \cdot F_i(\mathbf{x})[class]) + \lambda_2 \cdot f_i(\mathbf{x}, n_i) \tag{9}$$

For the left term, it tries to maximize the target model's and others' output, and the right term just maximizes the neuron output, indirectly increasing the *neuron coverage*. So, in the process, they kept tracking dormant neurons, and maximizing their outputs until the fixed point.

Besides, to make the new image input domain accepted, they modified the gradients to meet the lighting, big and small occlusions constraints. During the experiments, DeepXplore not only showed *Neuron Coverage* improvements, it also increased the model accuracy with augmented training data. As mentioned in previous section, if we treat one single neuron as a decision logic, then increasing *Neuron Coverage* will definitely explore more execution paths in the system.

Approach Critique. DeepXplore utilized the DNN gradient nature to produce the test input. They also made sure that newly created input wasn't randomized, with gradient constraints, those mutations were possible scenarios. However, it's inefficient to use multiple models to produce test cases for one model. Also, *Neuron Coverage* is easy to achieve, so a new coverage criteria should be adopted. Lastly, in the implementation, the authors averaged all neuron outputs in a layer level. They were trying to increase efficiency, but definitely, they lost precision on the objective optimization.

5. Methodology

Instead of using normalized *Neuron Coverage* as DeepXplore, the *Strong Neuron Activation Coverage* is a more suitable choice to explore unknown neuron output range. With neuron tracking for the training data, we get the $[low_{n_i}^{train}, high_{n_i}^{train}]$ range for a specific neuron n_i. Then we run gradient descent with hyper-parameters learning rate α and iteration e, by maximizing each single neuron's output. The above procedure gives us the first technique ***One-MaXplore***. However, this method is time-consuming.

In addition, when we take out the k neurons that yield top maximum output as in the *Top-k Neuron Coverage*, we are looking for the layer prominent decision pattern. Maximizing the outputs for those k neurons, the generated test input makes the pattern more obvious. For example, in Convolutional layer filters that capture vertical lines in images, maximizing the pattern yields an image with bolder vertical lines. Thus, this gives us another approach ***Topk-MaXplore***.

Despite of inefficiency with using multiple models, it's a necessary setting to produce buggy inputs, since disagreements among models imply sense of bugs in ML systems. Combined with ***Topk-MaXplore***, we get the third technique ***Multi-Topk-MaXplore***.

6. Experiments

The experiments are planned to answer questions listed as follows. Besides, the target model we used was LeNet-1. LeNet-4 and LeNet-5 were models for multiple models setting. And we only applied the top-k technique for the Convolutional layers. The dataset we used was MNIST for written digits classification.

RQ1 – How the output boundary in following layers behaves, when we apply ***One-MaXplore*** *on a shallower layer?*

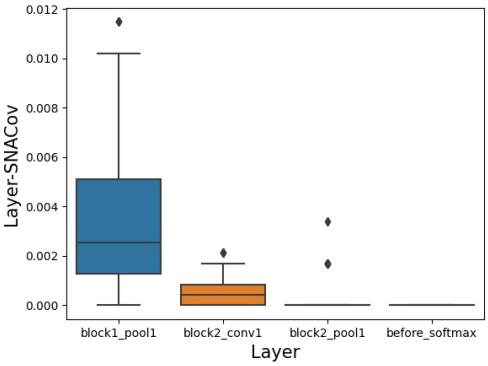
RQ2 – Comparing to ***One-MaXplore****, does* ***Topk-MaXplore*** *influence more neurons' output boundary?*

RQ3 – Comparing to ***DeepXplore****, does* ***Multi-Topk-MaXplore*** *have better Layer-SNACov metric performance?*

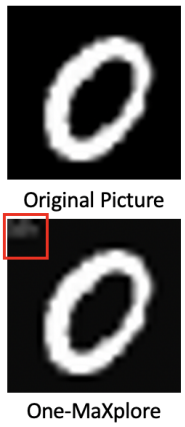
6.1. RQ1: Layer Output Boundary Behavior

In this experiment, we applied ***One-MaXplore*** to the first Convolutional layer, and recorded the following layers' SNACov. We used $\alpha = 0.05$ and $e = 10$. The result shows that 244 neurons change their maximum boundary, and more than 93% influence the following layers' SNACov (Figure 4). Thus, to mitigate the efficiency issue, it's possible to apply this technique to shallower layers only.

However, as the picture's red box showed, the change is localized as in DeepXplore. Furthermore, the boundary influence propagate not far, as the **block2_pool1** layer has pretty low SNACov.

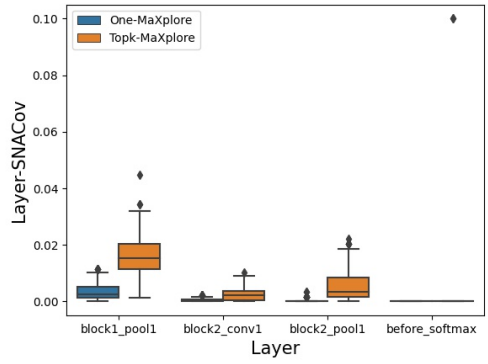


(a) Layer-SNACov in ***One-MaXplore***.

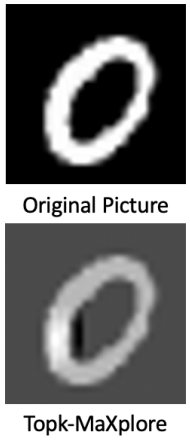


(b) Generated inputs.

Figure 4: Experiment 1 result.



(a) Two technique comparison.



(b) Generated inputs.

Figure 5: Experiment 2 result.

6.2. RQ2: *One-MaXplore* v.s. *Topk-MaXplore*

We used $k = 6$ for this experiment. From the result , we can infer that ***Topk-MaXplore*** has better SNACov statistics than ***One-MaXplore*** with the same hyper-parameter (Figure 5). Also, the change to input image is more globalized and emphasizes on certain patterns, which corresponds to our assumption about Convolutional layers.

6.3. RQ3: *Multi-Topk-MaXplore* v.s. *DeepXplore*

To make the results comparable, we fed multiple models with the same setting in the ***DeepXplore***. We used hyper-parameters $\alpha = 0.15, e = 25, k = 6, \lambda_1 = 1.5$ and $\lambda_2 = 1$. Also, we averaged the Layer-SNACov for 16 filters from the first and second convolutional layers.

The boxplot shows that in average ***Multi-Topk-MaXplore*** has better performance (Figure 6). However, it also suffers from large variance. The reason is that ***Topk-MaXplore*** only maximize k neurons' outputs, but for ***DeepXplore***, they average neurons in one filter and maximize it if it's dormant for a while. This makes their coverage metrics stable, but with worse performance in average.

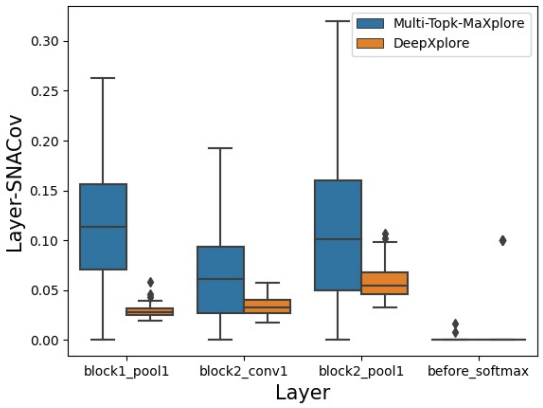


Figure 6: Experiment 3 result.

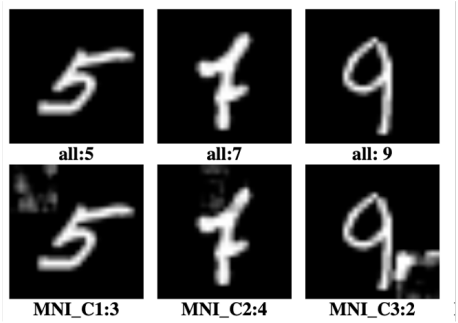


Figure 7: ***DeepXplore*** generated inputs.

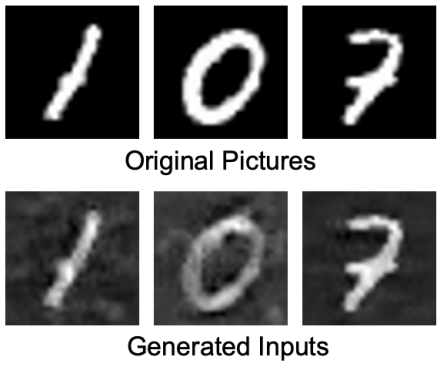


Figure 8: ***Multi-Topk-MaXplore*** generated inputs.

7. Challenges and Future Work

Apparently, ML testing can be applied in two scenarios, while the first is to explore unknown neuron output ranges, as most techniques in this paper addressing this problem, in-

cluding ***MaXplore***. Another one is to verify model stability, which is similar to TensorFuzz's applications.

Speaking of ***MaXplore***, we still want to know whether it improves the model accuracy with augmented training data. Also The setting of k is another interesting problem. It seems there's a trade-off between showing obvious patterns and maintaining image domain related constraints (for MNIST, it means showing the number clearly).

Besides, if we want to make sure the model is stable, the *k-Mulitsection Coverage* is a decent choice. With higher coverage, the test suite takes care of more neuron output ranges, which can be viewed as the model's execution paths. Nonetheless, it's mathematical intractable to optimize the coverage. Furthermore, the newly created inputs should not be a deviation from the training data distribution (for MNIST, it means adding white dirt to the black background). To address this, we plan to further study Generative Adversarial Network [8], because it has successfully showed effectiveness in learning the data distribution. Also, the k-Nearest-Neighbor method in TensorFuzz is also an alternative to optimize the *k-Mulitsection Coverage*. There's some potentials to combine them in future work.

References

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

[2] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[3] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu, J. Zhao, and Y. Wang, "Deepgauge: Multi-granularity testing criteria for deep learning systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 120–131. [Online]. Available: https://doi.org/10.1145/3238147.3238202

[4] A. Aggarwal, P. Lohia, S. Nagar, K. Dey, and D. Saha, "Black box fairness testing of machine learning models," ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 625–635. [Online]. Available: https://doi.org/10.1145/3338906.3338937

[5] M. T. Ribeiro, S. Singh, and C. Guestrin, ""why should i trust you?": Explaining the predictions of any classifier," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1135–1144. [Online]. Available: https://doi.org/10.1145/2939672.2939778

[6] Y. Sun, X. Huang, D. Kroening, J. Sharp, M. Hill, and R. Ashmore, "Deepconcolic: Testing and debugging deep neural networks," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2019, pp. 111–114.

[7] A. Odena, C. Olsson, D. Andersen, and I. Goodfellow, "TensorFuzz: Debugging neural networks with coverage-guided fuzzing," ser. Proceedings of Machine Learning Research, K. Chaudhuri and R. Salakhutdinov, Eds., vol. 97. Long Beach, California, USA: PMLR, 09–15 Jun 2019, pp. 4901–4911. [Online]. Available: http://proceedings.mlr.press/v97/odena19a.html

[8] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'14. Cambridge, MA, USA: MIT Press, 2014, p. 2672–2680.