Py-holmes: Causal Testing for Deep Neural Networks in Python

Wren McQueary
George Mason University
Fairfax, USA
wmcquear@gmu.edu

Sadia Afrin Mim
George Mason University
Fairfax, USA
safrinmi@gmu.edu

Md Nishat Raihan
George Mason University
Fairfax, USA
mraihan2@gmu.edu

Justin Smith
Lafayette College
Easton, USA
smithjus@lafayette.edu

Brittany Johnson
George Mason University
Fairfax, USA
johnsonb@gmu.edu

ABSTRACT

Deep learning has become a go-to solution for many problems. This increases the importance of our ability to understand and improve these technologies. While many tools exist to support debugging deep learning models (e.g., DNNs), few attempt to provide support for understanding the root cause of unexpected behavior. Causal testing is a technique that has been shown to help developers understand and fix the root causes of defects. Causal testing may be particularly valuable in DNNs, where causality is often hard to understand due to the abstractions DNNs create to represent data. In theory, causal testing is capable of supporting root cause debugging in various types of programs and software systems. However, the only implementation that exists is in Java and was not implemented as an end-to-end tool or for use on DNNs, making validation of this theory difficult. In this paper, we introduce py-holmes, a prototype tool that supports causal testing on Python programs, for both DNNs and shallow programs. For more information about py-holmes' internal process, see our GitHub repository: https://go.gmu.edu/pyHolmes_Public_Repo. Our demo video can be found here: https://go.gmu.edu/pyholmes_demo_2024.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**; • **Computing methodologies** → **Neural networks**; • **Applied computing** → Optical character recognition.

KEYWORDS

Deep Learning, Counterfactual Explanation, Causal Testing, Adversarial Sample Generation

ACM Reference Format:
Wren McQueary, Sadia Afrin Mim, Md Nishat Raihan, Justin Smith, and Brittany Johnson. 2024. Py-holmes: Causal Testing for Deep Neural Networks in Python. In *Proceedings of the 32nd ACM Symposium on the Foundations of Software Engineering (FSE '24), November 15–19, 2024, Porto de Galinhas, Brazil.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3663529.3663807

1 INTRODUCTION AND MOTIVATION

As deep learning (DL) has become more ubiquitous, there has been an increased interest in promoting software engineering practices, such as testing and debugging, in the context of deep learning model development [1]. Researchers have proposed a variety of solutions to support debugging deep neural networks (DNNs) [2–11], including attempts at supporting root-cause debugging. However, to our knowledge and unlike Causal Testing [12], these interventions have not been evaluated regarding their ability to help developers understand the causes of their code's behavior. Furthermore, existing tools for debugging DNNs lack support for verifiable explanations (such as counterfactual ones) which would aid reasoning about the root causes of defects [1, 5, 6, 11, 13].

Causal testing is a technique that has been shown to help developers understand the root causes of unexpected behavior [12]. Unlike existing debugging techniques that help programmers understand *what* the defect was and and *where* it occurred, causal testing helps programmers answer the *why*, such that they can produce fixes that address the root cause of the defect.

Causal testing has the potential to support root cause debugging for both DNNs and traditional (shallow) software. However, the current implementation (Holmes[1]) is only designed for shallow software in Java, whereas Python is much more prevalent for DL software [14]. Furthermore, the current version of Holmes is a proof of concept that requires additional effort to make it fully functional. To this end, we present *py-holmes*, a prototype tool that provides Python causal testing support for both DNNs and shallow programs.

In deep learning, py-holmes can offer two primary benefits. **First**, py-holmes explains failed tests by generating counterfactual input samples, which are similar to the original test input, but on which the model exhibits less loss, or perhaps passes the test. The differences between these new samples and the original reveal which features originally confounded the model, and these explanations can be verified by running these samples through the user's model and confirming the model's loss is as py-holmes reported. This verifiability could be useful in fairness testing, where surrogate models are commonly trained to explain mistakes made by a base model, but often fail to accurately represent its decision-making [15]. **Second**, py-holmes could help developers identify underrepresented regions of the feature space, and point the way toward new natural samples that could improve the model's performance in those regions.

[1] https://holmes.cs.umass.edu

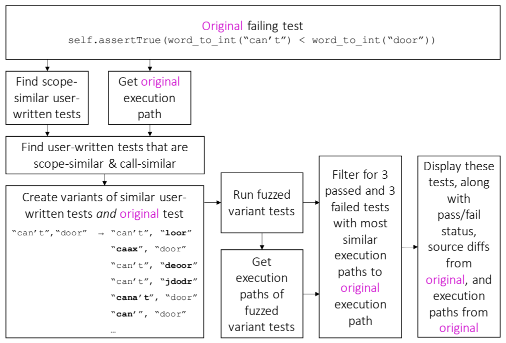

Original failing test
self.assertTrue(word_to_int("can't") < word_to_int("door"))
Find scope-similar user-written tests
Get original execution path
Find user-written tests that are scope-similar & call-similar
Create variants of similar user-written tests *and* original test
"can't","door" → "can't", "loor"
"caax", "door"
"can't", "deoor"
"can't", "jdodr"
"cana't", "door"
"can'", "door"
…
Run fuzzed variant tests
Get execution paths of fuzzed variant tests
Filter for 3 passed and 3 failed tests with most similar execution paths to original execution path
Display these tests, along with pass/fail status, source diffs from original, and execution paths from original

Figure 1: Overview of py-holmes' shallow mode.

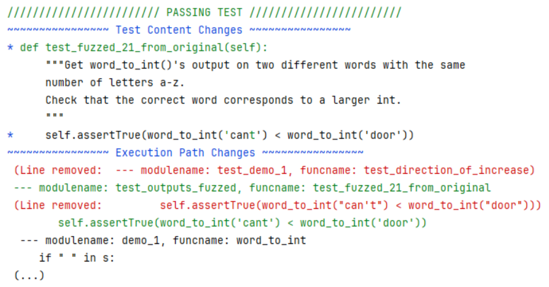
2 PY-HOLMES: PYTHON CAUSAL TESTING

Py-holmes shows the developer a set of variant test inputs, where each is minimally different from the original failing input in terms of both **value difference** and **execution difference**, but elicits program behavior that passes, or comes closer to passing, the original unit test. As a step forward from Holmes, py-holmes includes a novel DL mode alongside the existing shallow mode. In shallow mode, py-holmes generates counterfactual inputs for unit tests of shallow software, similarly to Holmes. In DL mode, py-holmes generates counterfactual inputs for DNNs implemented with PyTorch. Along with these inputs, py-holmes also presents the developer with a summary of code coverage differences (for tests of shallow software) or neuron activation differences (for tests of deep neural networks) versus the original input. These inputs and execution difference summaries are intended to aid the developer in understanding why the original failure occurred. Figure 2 shows a sample report for a shallow test, where the removal of an apostrophe from an input string caused a previously failing test to pass. Figure 5 shows a sample report for a DL test.

Py-holmes' two modes use similar principles, with different implementations. Table 1 analogizes the DL and shallow modes.

Table 1: Comparison of py-holmes' deep and shallow modes.

Principle	**Shallow**	**DL**
Input	Literals	Tensor
Similar **value** generation	Random character edits (for strings), adding noise (for numbers)	Assistive sample generation
Execution similarity metric	$d_{line} + 10d_{call}$	Cosine distance of final embedding layer activations
Test case filtering	Find 3 passing and 3 failing tests that elicit the most similar execution trace to the original	Find 3 passing and 3 failing inputs that elicit the most similar activation pattern to the original



```
//////////////////////// PASSING TEST ////////////////////////
~~~~~~~~~~~~~~~~ Test Content Changes ~~~~~~~~~~~~~~~~
* def test_fuzzed_21_from_original(self):
      """Get word_to_int()'s output on two different words with the same
      number of letters a-z.
      Check that the correct word corresponds to a larger int.
      """
*     self.assertTrue(word_to_int('cant') < word_to_int('door'))
~~~~~~~~~~~~~~~~ Execution Path Changes ~~~~~~~~~~~~~~~~
 (Line removed:  --- modulename: test_demo_1, funcname: test_direction_of_increase)
 --- modulename: test_outputs_fuzzed, funcname: test_fuzzed_21_from_original
 (Line removed:         self.assertTrue(word_to_int("can't") < word_to_int("door")))
        self.assertTrue(word_to_int('cant') < word_to_int('door'))
  --- modulename: demo_1, funcname: word_to_int
     if " " in s:
 (...)
```

Figure 2: Portion of a py-holmes shallow report

2.1 Shallow Mode Process

Figure 1 depicts py-holmes' internal process in shallow mode.

2.1.1 Generating Variant Tests. Py-holmes produces variant tests by perturbing ("fuzzing") one or more literals contained in a failing test. To avoid unintentionally fuzzing oracle-defining literals, py-holmes constructs a directed graph of the unit test method in which each literal and variable name is represented as a node. For each "=" assignment that appears in the method, directed edges are drawn from all literals and variable names on the right side of the "=" to all variable names on the left side. Any literal node from which an oracle node can be reached by following this graph is considered to be involved in defining an oracle, and is therefore protected from fuzzing. When an input value has been designated for fuzzing, the nature of the fuzzing depends on its type. A string will undergo random character edits (eg `"door"` to `"dmor"`), whereas a number will undergo slight perturbations (eg 3 to 2, or `10.5` to `10.641`).

If the user's repository contains any other test methods that are both *scope-similar* (importing a nonempty subset of the same user-written files) and *call-similar* (calling the same functions in the same order) to the original test method, fuzzed variants are produced from these tests as well. This allows py-holmes to create valid variants of string enums without needing to chance upon them via random character edits.

2.1.2 Filtering Variant Tests for Similar Execution to the Original. Py-holmes runs the original test, as well as all variant tests produced in the previous step, and records their full execution traces. Next py-holmes filters the variant tests for the 3 passing and 3 failing tests that had the most similar execution traces to the original test's execution trace. The distance between two execution traces is $d_{line} + 10d_{call}$, where d_{line} is the number of single-line changes to the sequence of source line executions, and d_{call} is the number of changes to what functions are called, and at what points in the trace. Py-holmes concludes by showing the user a report containing these 6 tests, along with diffs for their source code and execution traces when compared to the original test.

2.2 DL Mode Process

Figure 3 depicts py-holmes' internal process in DL mode.

2.2.1 Generating Variant Inputs. Py-holmes runs the original failing input (a PyTorch tensor) through the model under test, and obtains the loss gradient with respect to that input. This gradient is then multiplied by an adjustment rate set by the user and subtracted

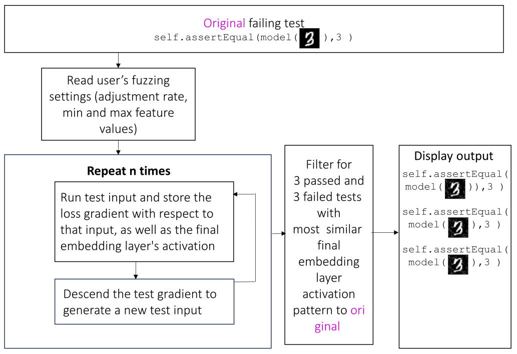

Original failing test
self.assertEqual(model( ),3)
Read user's fuzzing settings (adjustment rate, min and max feature values)
Repeat n times
Run test input and store the loss gradient with respect to that input, as well as the final embedding layer's activation
Descend the test gradient to generate a new test input
Filter for 3 passed and 3 failed tests with most similar final embedding layer activation pattern to original
Display output
self.assertEqual(model( )),3)
self.assertEqual(model( ),3)
self.assertEqual(model( ),3)

Figure 3: Overview of py-holmes' DL mode.

from the input to create a new input that will elicit a lower loss. This process is identical to adversarial sample generation, except it descends the loss gradient instead of ascending it [16]. Inspired by this name, we call our approach *assistive sample generation.* Py-holmes iterates this process, generating a sequence of inputs, each one incrementally farther from the original but eliciting smaller losses from the model as long as the adjustment rate is low enough.

2.2.2 Filtering Variant Inputs for Similar Execution to the Original. Next py-holmes filters the variant inputs produced in the previous step for the 3 passing and 3 failing inputs that caused the most similar neuron activation pattern to the original in the final embedding layer of the model. Activation similarity is measured using cosine distance (defined as $1 - \frac{s}{2}$, where s is the cosine similarity). Py-holmes concludes by showing the user a report containing these 6 inputs, along with the model's loss on each input, the neuron activation distance of the final embedding versus the original, and the value of the new input tensor.

3 PY-HOLMES WITH NEURAL NETWORKS

Suppose a user is building a fully connected DNN to classify digits from the MNIST database. The MNIST database contains labeled, grayscale images of handwritten digits 0-9, with a resolution of 28x28 pixels [17]. Given an image from the MNIST database, the user's model guesses which digit is written in the image.

We used the MNIST database to informally evaluate py-holmes during its development, because the database's pictoral nature made it easy to visualize py-holmes' changes to inputs. However, py-holmes' DL mode can be used on any tensor input, including non-pictoral inputs such as vectors. We intend to perform a user study of py-holmes on a greater diversity of datasets, along with a comparitive analysis between py-holmes and other tools.

After training their model, the user evaluates it on a validation dataset. They notice that their model misclassifies a particularly obvious handwritten 5, despite otherwise performing well. They want more insight into why the failure occurred, in order to make a more informed decision about how to adjust the way they train their model.

The user decides to use py-holmes to find nearby inputs that would have passed, thereby explaining the failure. They write a

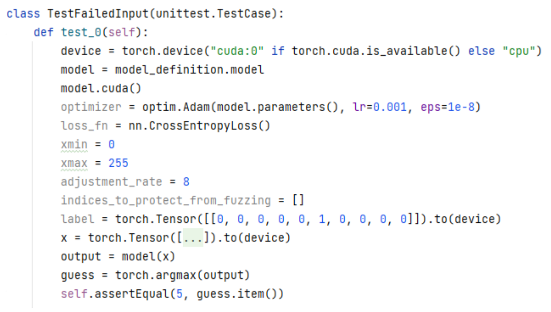

```python
class TestFailedInput(unittest.TestCase):
    def test_0(self):
        device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
        model = model_definition.model
        model.cuda()
        optimizer = optim.Adam(model.parameters(), lr=0.001, eps=1e-8)
        loss_fn = nn.CrossEntropyLoss()
        xmin = 0
        xmax = 255
        adjustment_rate = 8
        indices_to_protect_from_fuzzing = []
        label = torch.Tensor([[0, 0, 0, 0, 0, 1, 0, 0, 0, 0]]).to(device)
        x = torch.Tensor([...]).to(device)
        output = model(x)
        guess = torch.argmax(output)
        self.assertEqual(5, guess.item())
```

Figure 4: The user-written unit test. The line `x = torch.Tensor([...]).to(device)` is shortened for brevity. At its full length, it defines each value in the input tensor.

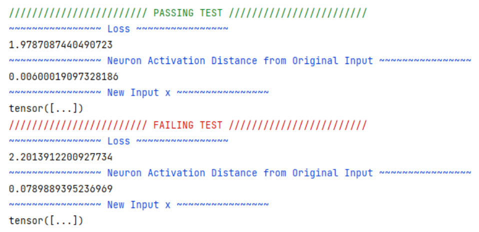

```
/////////////////////// PASSING TEST ///////////////////////
~~~~~~~~~~~~~~~~ Loss ~~~~~~~~~~~~~~~~
1.9787087440490723
~~~~~~~~~~~~~~~~ Neuron Activation Distance from Original Input ~~~~~~~~~~~~~~~~
0.00600019097328186
~~~~~~~~~~~~~~~~ New Input x ~~~~~~~~~~~~~~~~
tensor([...])
/////////////////////// FAILING TEST ///////////////////////
~~~~~~~~~~~~~~~~ Loss ~~~~~~~~~~~~~~~~
2.2013912200927734
~~~~~~~~~~~~~~~~ Neuron Activation Distance from Original Input ~~~~~~~~~~~~~~~~
0.0789889395236969
~~~~~~~~~~~~~~~~ New Input x ~~~~~~~~~~~~~~~~
tensor([...])
```

Figure 5: py-holmes' final report in DL mode. For brevity, this figure shows only one passing test and one failing test, and the input x tensors are shortened to `tensor([...])`.

`unittest`-style unit test of their model on that input, shown in Figure 4, inside of a new file, which they choose to name `my_test.py`. This unit test instantiates all of the required variables for py-holmes to run in its deep learning mode. The purpose of each variable in this unit test is discussed further in our artifact's readme file.

Now the user calls py-holmes from the command line with `python py_holmes.py -dl -f my_test.py -l all`. Py-holmes finds variants of the user's input and prints the readout shown in Figure 5, which gives the three passing and three failing inputs which elicited the smallest changes in the final embedding layer.

Although the tensors contain visual information, py-holmes outputs its tensors as text. The user decides to use matplotlib to visualize the tensors and heatmaps of the pixel value changes. This visualization is shown in Figure 6 and can be reproduced by running the Jupyter Notebook `mnist_demo.ipynb` in our artifact. From the visualization, the user concludes that their model failed because it over-relies on 5s that cross into the red areas of the plots.

Prior research has demonstrated that adversarial sample generation can be used as an effective data augmentation approach to improve robustness [18], but that introducing non-natural adversarial samples into a training dataset may reduce overall accuracy [19]. Because py-holmes' variant inputs are unnatural and assistive rather than adversarial, they should not be added directly to the training dataset, but they can guide a developer's search for new natural data to add to the training dataset. In the case of MNIST, a developer can hand-draw new natural samples informed by py-holmes' variant inputs, as shown in Figure 7. We hypothesize that new natural samples avoiding densely red areas would

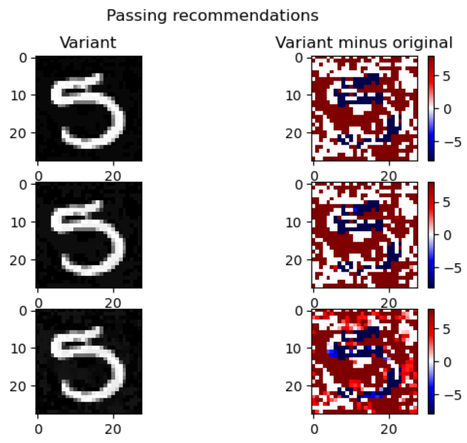


Figure 6: Variant inputs found by py-holmes.

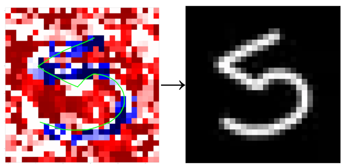


Figure 7: Users can search for new samples (right) using a variant input's difference from the original (left).

especially improve the model's accuracy on inputs near the original failing input. However, this would require the inputs to be close enough to uphold local monotonicity in the loss gradient, because the new user-drawn sample would be an approximate reflection of the sample generated by py-holmes across the original sample in the feature space. We hypothesize that a version of py-holmes which instead ascends the loss gradient could produce useful samples for augmenting the training dataset even outside the zone of local monotonicity, because no acts of reflection would be involved in generating the samples. We intend to test these hypotheses soon in our user study. We also hope to further integrate py-holmes with PyTorch, to eliminate the user's overhead work of writing unit tests for the inputs they wish to pass to py-holmes.

4 RELATED WORK

Most relevant to this work is the development of the original Causal Testing tool, Holmes [12], which is tailored for JUnit tests. This tool exhibits methodological parallels with py-holmes, especially in the generation of slightly varied inputs and runtimes.

Pynguin is an automated regression testing tool, but it is only compatible with Python versions 3.10 and above, whereas common Python versions for DNN development are 3.7 and 3.8 [20].

Hypothesis is a prominent property-based testing tool for Python. It identifies flaws in users' code by searching for test inputs that falsify an intended invariant for that code [21]. Hypothesis and py-holmes could be used in tandem when debugging shallow software, with Hypothesis identifying faults, and py-holmes explaining them.

Tools also exist for debugging DNNs. TensorFuzz mirrors the functionality of Hypothesis, except for DNNs rather than shallow software [5]. DeepConcolic aims to maximize coverage by generating a comprehensive test suite [6]. DeepHunter is a coverage-guided fuzz testing framework for DNNs, which uses metamorphic mutation guided by an array of coverage criteria, and can outperform existing methods in terms of coverage and defect detection, especially during DNN quantization for platform migration [7].

LIME generates similar inputs by deleting portions of the original input, such as segments from images and words from sentences. By running these modified inputs through the model under test, it can obtain enough behavioral information to train a simple, interpretable linear model in a small region of the input space. This linear model can then be used to generate explanations [8]. Whereas LIME's explanations are obtained by observing the linear model (which is only accurate if local linearity holds), py-holmes' explanations are the variant inputs themselves, paired with the model's loss on each one. Unlike LIME's explanations, py-holmes' can be verified by running the inputs back through the model.

Grad-CAM is a tool for convolutional neural networks (CNNs), which overlays a heatmap onto an input image to highlight the regions that were most influential in determining the model's output on a task. Like py-holmes on the MNIST task, Grad-CAM identifies changes in the input image which would have reduced the model's loss on a multiclass classification task, producing a verifiable counterfactual explanation which can also guide a developer's search for more training inputs. In addition to classification tasks, Grad-CAM supports captioning and visual question-answering tasks. However, it is limited to CNNs with image-based tasks [9].

DeepDiagnosis debugs DNNs by observing models during training, scrutinizing a range of error scenarios, and delivering practical solutions for correcting errors. In contrast to prevailing techniques, DeepDiagnosis demonstrates enhanced precision and efficiency in identifying faults, locating bugs, and recognizing symptoms [10].

MODE and the solution proposed by Cadamuro et. al are also tools for finding training data biases in DNNs [2, 11]. These tools focus on identifying and repairing model flaws, while py-holmes could complement them by offering insight into the flaws' causes.

5 CONTRIBUTIONS & FUTURE WORK

We introduced py-holmes, a tool for helping programmers understand the root causes of Python unit test failures in traditional and deep learning programs. We characterized how a software developer can use py-holmes to find such a root cause and work to adjust their software accordingly. We are currently designing a user study to evaluate py-holmes' ability to support root cause understanding and debugging in practical scenarios.

ACKNOWLEDGEMENTS

Many thanks to Ebtesam Al Haque, Antonis Anastasopoulos, and Carlotta Domeniconi for their guidance on this research.

DATA-AVAILABILITY STATEMENT

Release v1.0.1 of our GitHub repository is available as an artifact on Zenodo [22]. This artifact's readme file contains instructions for installing py-holmes and reproducing our results.

REFERENCES

[1] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Transactions on Software Engineering*, vol. 48, no. 1, pp. 1–36, 2020.

[2] G. Cadamuro, R. Gilad-Bachrach, and X. Zhu, "Debugging machine learning models," in *ICML Workshop on Reliable Machine Learning in the Wild*, vol. 103, 2016.

[3] D. Kang, D. Raghavan, P. Bailis, and M. Zaharia, "Model assertions for debugging machine learning," in *NeurIPS MLSys Workshop*, vol. 3, 2018, p. 10.

[4] N. Rauschmayr, V. Kumar, R. Huilgol, A. Olgiati, S. Bhattacharjee, N. Harish, V. Kannan, A. Lele, A. Acharya, J. Nielsen *et al.*, "Amazon sagemaker debugger: a system for real-time insights into machine learning model training," *Proceedings of Machine Learning and Systems*, vol. 3, pp. 770–782, 2021.

[5] A. Odena, C. Olsson, D. Andersen, and I. Goodfellow, "Tensorfuzz: Debugging neural networks with coverage-guided fuzzing," in *International Conference on Machine Learning*. PMLR, 2019, pp. 4901–4911.

[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)*. IEEE, 2019, pp. 111–114.

[7] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See, "Deephunter: a coverage-guided fuzz testing framework for deep neural networks," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 146–157.

[8] 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*, 2016, pp. 1135–1144.

[9] R. R. Selvaraju, M. Cogswell, A. Das, R. Vedantam, D. Parikh, and D. Batra, "Grad-cam: Visual explanations from deep networks via gradient-based localization," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 618–626.

[10] M. Wardat, B. D. Cruz, W. Le, and H. Rajan, "Deepdiagnosis: automatically diagnosing faults and recommending actionable fixes in deep learning programs," in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 561–572.

[11] S. Ma, Y. Liu, W.-C. Lee, X. Zhang, and A. Grama, "Mode: automated neural network model debugging via state differential analysis and input selection," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2018, pp. 175–186.

[12] B. Johnson, Y. Brun, and A. Meliou, "Causal testing: understanding defects' root causes," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 87–99.

[13] F. Schneider, F. Dangel, and P. Hennig, "Cockpit: A practical debugging tool for the training of deep neural networks," *Advances in Neural Information Processing Systems*, vol. 34, pp. 20 825–20 837, 2021.

[14] A. McFarland, "5 best machine learning (ai) programming languages," https://www.unite.ai/5-best-machine-learning-ai-programming-languages/, accessed: 2022-11-09.

[15] U. Aïvodji, H. Arai, O. Fortineau, S. Gambs, S. Hara, and A. Tapp, "Fairwashing: the risk of rationalization," in *International Conference on Machine Learning*. PMLR, 2019, pp. 161–170.

[16] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[17] 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.

[18] T. Liu and Y. Sun, "End-to-end adversarial sample generation for data augmentation," in *Findings of the Association for Computational Linguistics: EMNLP 2023*, 2023, pp. 11 359–11 368.

[19] D. Tsipras, S. Santurkar, L. Engstrom, A. Turner, and A. Madry, "Robustness may be at odds with accuracy," *arXiv preprint arXiv:1805.12152*, 2018.

[20] S. Lukasczyk and G. Fraser, "Pynguin: Automated unit test generation for python," *arXiv preprint arXiv:2202.05218*, 2022.

[21] D. R. MacIver, Z. Hatfield-Dodds *et al.*, "Hypothesis: A new approach to property-based testing," *Journal of Open Source Software*, vol. 4, no. 43, p. 1891, 2019.

[22] W. McQueary, S. A. Mim, M. N. Raihan, J. Smith, and B. Johnson, "Py-holmes fse 2024 artifact version 1." [Online]. Available: https://doi.org/10.5281/zenodo.11168191

Received 2024-01-29; accepted 2024-04-15