aNNoTest: An Annotation-based Test Generation Tool for Neural Network Programs

Mohammad Rezaalipour and Carlo A. Furia

Software Institute – USI Università della Svizzera italiana
Lugano, Switzerland
{rezaam, furiac}@usi.ch

***Abstract*—Even though neural network (NN) programs are often written in Python, using general-purpose test-generation tools for Python to test them is likely to be ineffective, as these tools do not support the particular input constraints that NN programs often require. To address this challenge, we present aNNoTest: an automated unit-test generation tool for NN programs written in Python. aNNoTest offers a simple annotation language that is suitable to concisely express the usual input constraints of NN programs; it then uses these annotations to precisely generate valid inputs that are capable of revealing bugs. This short paper describes how aNNoTest works in practice, and reports some experiments that demonstrate its effectiveness as a bug-finding tool for NN programs. aNNoTest is available as open source.**

***Index Terms*—Test Generation, Neural Networks, Debugging, Python**

I. Introduction

As neural network (NN) programs are increasingly deployed in safety-critical systems, developing techniques to effectively test them becomes paramount. Most of the research on NN testing focused on exercising model-level properties like robustness and training performance [1], [2]. However, additional challenges come from the way in which NN programs are usually written.

NN programs are often developed by domain experts, written in dynamic programming languages such as Python, using libraries and frameworks such as Keras or TensorFlow. There is evidence that such NN programs are prone to suffer from various defects [3], [4], including low-level bugs—such as type errors and other kinds of runtime failures—that derive from the dynamically typed nature of Python but would be caught by the compiler in a statically-typed language. Python's dynamic type system is also a challenge for automated test-case generation tools, which tend to be more novel and less developed than for statically-typed languages like Java [5]–[7]. The few Python test-generation tools that exist—such as Pynguin [8]—are general-purpose, and hence may not be effective to generate tests for NN programs.

As we demonstrate in Sec. II, using a general-purpose test-case generation tool on a NN program is likely to generate many invalid inputs, which trigger spurious crashes without finding any actual bug. General-purpose tools implement several approaches to mitigate the challenges of targeting a dynamically typed language like Python—for instance, leveraging type hints [9]. However, NN programs manipulate complex, precisely-constrained objects such as tensors and vectors, which compounds the challenges of dealing with a dynamically typed language and makes general-purpose annotations such as type hints insufficient for precise test-input generation.

Work partially supported by SNF grant 200021-182060 (Hi-Fi).

To address these challenges, we present aNNoTest: a unit-test generation tool for NN programs written in Python. aNNoTest [10] relies on a simple, domain-specific annotation language called aN, which one can use to precisely and concisely express the complex input constraints of a NN program's functions to be tested. Given some annotations, aNNoTest automatically generates unit tests with high precision. As we discuss in Sec. IV, we were able to apply aNNoTest to several open-source NN programs, generating tests revealing numerous bugs (including several of those manually surveyed in [4]). A short demo of aNNoTest is available at https://youtu.be/3Y1sraVajIA.

II. Using aNNoTest

In this section, we briefly demonstrate how to use aNNoTest to generate unit tests that can expose bugs in neural-network programs written in Python. We also discuss how aNNoTest is more effective than other, general-purpose test-generation tools for Python when testing NN programs with their particular constraints on inputs. To this end, we discuss using aNNoTest in comparison with Pynguin [8] and Deal [11]—the only general-purpose state-of-the-art automated test generation tools for Python available at the time of writing. As we will demonstrate in the rest of the paper, aNNoTest is designed as *complementary* to these two tools: while its annotations are applicable, in principle, to any Python program, aNNoTest is primarily a specialized tool for NN programs; Pynguin and Deal are general-purpose tools suitable for all sorts of Python programs.

Consider function `model_discriminator` from project ADV (Keras Adversarial Models) [12] whose implementation is in Lst. 1. When Python 3's interpreter evaluates the expression on line 11, the execution *crashes*, because `hidden_dim / 2` returns a **`float`** but the constructor `Dense` only works correctly if its first argument is an **int**.[1] This bug was among those surveyed by Islam et al. [4], and was eventually fixed with an explicit **int** conversion in a later revision of the ADV project. We selected this example because it is simple, yet realistic; using it makes for a clearer presentation, but we stress that the challenges that we highlight become much more problematic as soon as one target larger and more complex examples (such as those that we discuss in Sec. IV).

On the face of it, generating valid tests that exercise function `model_discriminator`, and trigger the bug, should be straightforward: the function's implementation is short, consists of straight-line code (no branching), and only takes four arguments. Furthermore, *every valid* input would trigger the bug, and valid input values for three out of four arguments are provided as defaults. And yet, automatically generating several valid inputs for this function turns out to be surprisingly hard for general-purpose testing tools—as we now discuss in detail.

A. General-Purpose Testing Tools

1) Pynguin: Consider Pynguin [8], a general-purpose automated test generation tool for Python. Pynguin's test-generation strategy is based on a genetic algorithm that tries to maximize branch coverage. For a simple, straight-line function like `model_discriminator`, Pynguin only generates[2] one test input consisting of string `"!b)p"` as actual value for argument `input_shape`. This input is *invalid*, since `input_shape` must be a *shape*—basically, a tuple of nonnegative integers that denote the dimensions of a multi-dimensional array.

It should be no surprise that Pynguin is ineffective on this example: Python does not require function arguments to be annotated with their intended types, and this crucial piece of information is thus not available to the test-case generation tool. Unfortunately, even if `model_discriminator` were annotated using Python's *type hints*—which Pynguin partially supports—these are not expressive enough to precisely encode the range of `model_discriminator`'s valid inputs. The best we can do is annotating `input_shape` with type `Tuple[int, int]`, `hidden_dim` with type `int`, `reg` with type `Callable`, and `output_activation` with type `Literal`. These constraints are both too loose to identify valid inputs only, and not fully compatible with Pynguin's generation algorithm. As a result, Pynguin generates two test inputs, both invalid: in one, `input_shape` is the lone integer `-1262`; in another one, it is the tuple `(345, True)`; neither of them is a valid shape.

2) Deal: Encoding arbitrarily complex constraints is possible using a tool like Deal [11], which offers an expressive language to encode function *preconditions* (as well as other design-by-contract annotations).

Using Deal's annotation language, we can precisely express `model_discriminator`'s input constraints, which we can elicit

```
1  def model_discriminator(input_shape,
2                          hidden_dim=1024,
3                          reg=lambda: l1l2(1e-5, 1e-5),
4                          output_activation="sigmoid"):
5      return Sequential([
6          Flatten(name="discriminator_flatten",
7                  input_shape=input_shape),
8          Dense(hidden_dim, name="discriminator_h1",
9                W_regularizer=reg()),
10         LeakyReLU(0.2),
11         Dense(hidden_dim / 2, name="discriminator_h2",  # bug
12               W_regularizer=reg()),
13         LeakyReLU(0.2),
14         Dense(hidden_dim / 4, name="discriminator_h3",
15               W_regularizer=reg()),
16         LeakyReLU(0.2),
17         Dense(1, name="discriminator_y", W_regularizer=reg()),
18         Activation(output_activation)], name="discriminator")
```

Listing 1. Function `model_discriminator` from project ADV (Keras Adversarial Models).

```
@arg(input_shape): np_shapes(min_dims=2, max_dims=2, min=1, max=28)
@arg(hidden_dim): ints(min=1024, max=2048)
```

Listing 2. Annotations for function `model_discriminator` in Lst. 1. The annotations' syntax is slightly simplified for readability.

from examples of client code in other parts of project ADV (as well as from the default values for some of the arguments). In particular, `input_shape` should be a two-element tuple of integers in the range 1–28,[3] and `hidden_dim` should be a positive integer, typically in the range 1024–2048. For simplicity, we can tentatively ignore the more complex constraints on arguments `reg` and `output_activation`, and simply use the default values of those two arguments.

Even though we can precisely express the constraints on `input_shape` and `hidden_dim`, Deal still fails to generate any valid inputs for `model_discriminator`. This is because Deal's input generation algorithm produces *random* inputs, and then uses the constraints/preconditions to filter a posteriori the random inputs. Thus, it's exceedingly unlikely that a random value (among all possible Python types) happens to satisfy detailed constraints such as those required by `model_discriminator`.

In hindsight, it is unsurprising that Pynguin and Deal—two state-of-the-art *general-purpose* test-case generation tools for Python—are ineffective on this deceptively simple example. NN programs often consist of structurally simple, usually short, functions with complex constraints on their numerous input arguments [13]. In order to generate valid tests for these programs, we need a *specialized* approach which supports both 1) concisely *expressing* the complex input constraints; and 2) using those constraints to *drive the generation* of possible valid inputs. This is what aNNoTest is designed for.

B. aNNoTest

aNNoTest is a unit-test generation tool specifically designed to be effective on Python implementations of NN

[1]The function has a similar bug on line 14; we focus on line 11's bug, which masks the bug on line 14.

[2]We ran Pynguin 0.33.0 (the latest version at the time of writing) with its default configuration: DynaMOSA for test-suite generation and MUTATION_ANALYSIS for assertion generation.

[3]Precisely, the tuple's components would be valid even if they were larger than 28, but the 1–28 range is sufficient to generate a wide variety of valid inputs that reflect typical usage.

programs. To this end, it offers aN: a simple annotation language suitable to concisely express the typical constraints on function inputs in such programs.

1) Annotations: Lst. 2 shows some aN annotations for function `model_discriminator`, which precisely characterize valid input values for arguments `input_shape` and `hidden_dim`, as we discussed them above. Users of aNNoTest can decide how much annotation effort to spend, depending on which functions they want to focus on testing, and on how thorough the testing should be.

Concretely, aNNoTest's distribution includes a module `an_language.py` with concrete syntax for the aN annotation language. To annotate a function, users of aNNoTest import this module, and then use the imported decorators just before each function to be annotated, directly in the source code.

In this example, we did not annotate two arguments, and just relied on their default values. If we wanted a more extensive test generation process, we could provide sets of possible suitable functions (argument `reg`) and library function names (argument `output_activation`), and constrain aNNoTest to pick input values from these sets.

2) Test generation: Writing annotations is the only manual part of using aNNoTest. After adding function Lst. 2's annotations to the source code, we call aNNoTest with `annotest <program_root>`. The tool scans through the whole program and generates tests for all annotated functions.

When an argument is left without aN annotations, aNNoTest uses the argument's default value. Thus, aNNoTest can only test a function if all its arguments have some aN annotations or a default value. In the running example, arguments `reg` and `output_activation` are not annotated; thus, aNNoTest always sets `reg` to `l1l2(1e-05, 1e-05)` and `output_activation` to `"sigmoid"` (see Lst. 1).

aNNoTest generates Hypothesis test templates [14] using the format recognized by Python's `unittest` testing framework, which can be used to actually expand and run the tests.

In our running example, aNNoTest generates test templates nearly instantaneously; running them with `unittest` takes 0.3 seconds, and produces 14 distinct valid test inputs (one of which triggers the bug at Lst. 1's line 11).

III. Design and Implementation

aNNoTest inputs an annotated NN program and outputs Hypothesis test templates that encode all the information to generate unit tests that satisfy the annotated constraints. This section first describes the main features of the aN annotation language (Sec. III-A), and then how aNNoTest aggregates the annotation information and uses it to generate tests.

A. The aN Annotation Language

Fig. 1 shows the main kinds of annotations supported by aNNoTest's aN annotation language. The largest class of constraints are *type constraints*: **@arg**(var): $TypeConstr$ constrains argument var to values of a specific type and range. These can be subsets of booleans, integers, and floating points (**`bools`**, **`ints`**, **`floats`**), as well as compound types such

```
        An → @arg(var): TypeConstr | @require(BooleanExpr)
TypeConstr → froms(list) | bools() | ints(min=-Inf, max=+Inf)
           | floats ( min=-Inf, max=+Inf, exclude_min=False,
                      exclude_max=False, exclude_NaN=True,
                      exclude_Inf=True )
           | tuples(TypeConstr*)
           | np_shapes(min_dims=1, max_dims=1, min=1, max=1)
           | int_lists(min=-Inf, max=+Inf, min_len=0, max_len=10)
           | np_arrays(np_type, shape=TypeConstr)
           | dicts(keys=TypeConstr,
                   values=TypeConstr, min_size=0, max_size=+Inf)
           | anys(TypeConstr+) | objs(function)
```

Fig. 1. The main annotations supported by aNNoTest.

as tuples and dictionaries (**`tuples`**, **`dicts`**). Type constraints **`np_shapes`**, **`int_lists`**, and **`np_arrays`** specify lists and tuples of numeric values (including those supported by the NumPy library), which feature frequently in NN programs. Finally, **`froms`** supplies a list of concrete values to sample from; **`anys`** is the union of multiple type constraints; and **`objs`** introduces custom *generator functions* to generate arbitrarily complex objects.

Preconditions are constraints that involve multiple arguments at once or need to be conditional.[4] In aN, **@require**(p) specifies a precondition p, where p is an arbitrary Python Boolean expression involving the annotated function's arguments.

The aN language includes a few other annotations [10] (which we do not discuss here for brevity), such as to skip testing certain functions, to add a test timeout, and to use constructors in test code.

B. Testable Functions

By default, aNNoTest generates tests for all functions in a NN program about which it has sufficient information—either through user-provided annotations or through default values.

More precisely, an argument `a` of a function `f` is *testable* if at least one of the following conditions holds: 1) `f` includes an aN annotation that constrains `a`; 2) `a` has a default value (it is an optional argument); 3) `a` is `self`, and the enclosing class `C` has a constructor that is testable. 4) `a` is a non-keyword (`*args`) or keyword (`**kwargs`) variable-number argument. Thus, if `a` is testable, it means that aNNoTest knows how to generate at least one valid value for it: one that satisfies an annotation, a default, one that can be constructed, or that can simply be omitted. A function `f` is testable if all its arguments are testable.

C. Strategies

For each *testable* argument, aNNoTest generates a suitable Hypothesis strategy: a custom object-generating function.

[4]Even though aNNoTest's preconditions are similar to Deal's, the bulk of NN annotations can be expressed using aNNoTest's type constraints and other, simpler annotations that crucially support the efficient generation of input values.

Several of aN's type constraints match some of Hypothesis's built-in strategies. For instance, Hypothesis strategies `array_shapes` and `integers` are suitable to easily encode aN's constraints **`np_shapes`** and **`ints`** used in Lst. 2's running example. aNNoTest can also reuse the default values of arguments using Hypothesis's `just` strategy.

More complex type constraints do not have a one-to-one matching Hypothesis strategy. In these cases, aNNoTest automatically combines available strategies or even generates new strategies. For example, this is the case of constraint **`int_lists`**, which aNNoTest encodes into a generator function for integer lists with suitable characteristics.

Another interesting case are **`objs`**`(gen)` type annotations, where `gen` is a user-defined function that should be used to generate values. In this case, aNNoTest embeds and adapts `gen`'s implementation into Hypothesis code, so that it can be used as a generation strategy for the corresponding argument.

D. Templates

For each testable function `f`, aNNoTest combines the strategies for each of `f`'s arguments—built as discussed in the previous section—into a *test template*.

For a function `f` that is an instance method of some class `C`, aNNoTest generates a template that first instantiates object `o` of class `C` using `C`'s constructor and the strategies recursively assigned to the constructor's argument. Then, it calls `o.f(...)` passing the values obtained by the strategies of `f`'s other arguments.

To encode **`@require`** annotations (preconditions), aNNoTest uses Hypothesis's `assume` function.

E. Executing the Tests

The final output of a run of aNNoTest consists of several Hypothesis test templates—one for each testable function in the analyzed NN program. aNNoTest uses `unittest`'s format to encode the test templates. Thus, users invoke `unittest` to pass aNNoTest's output to Hypothesis, which actively generates concrete input values using the strategies, runs the tested functions, and reports any test failure.

F. Implementation Limitations

aNNoTest cannot test nested functions—functions that are defined within other functions. This limitation, which also applies to manually written tests, is simply due to the fact that a nested function is invisible outside its host function; hence, we cannot test the nested function unless we also test the host function.

aNNoTest's implementation relies on the Python `ast` library to extract information from a project's source code; thus, aNNoTest can only process code that is syntactically valid. If a module fails a syntax check, aNNoTest skips it and provides a warning message.

IV. Experiments

In order to evaluate aNNoTest's capabilities, effectiveness at finding NN bugs, and usability, we used it on real-world NN programs in two experimental evaluations, which we presented in detail in previous work [10], and summarize here.

TABLE I
A summary of experiments with aNNoTest on testing 19 open-source NN programs. Each experiment summarizes the results over two fully-annotated *complete projects* and 19 partially-annotated project where aNNoTest *reproduced* known bugs [4]. Each row reports the total size of the projects used in that experiment (in lines of code loc), and the number of unique crashing bugs found by tests generated by aNNoTest—split into confirmed true bugs, spurious bugs (triggered by invalid inputs), and the corresponding precision $=$ true$/($true $+$ spurious$)$.

experiment	loc	bugs		
		true	spurious	precision
Complete projects	3917	50	6	89%
Reproduced bugs	14219	93	0	100%

A. Bug Finding

In the first experiment (*Complete projects* in Tab. I), we took the latest revisions of projects ADV (the same mentioned in Sec. II) and GANS [15] and annotated with aN their functions and methods based on the information available in their project repositories, as well as on some trial and errors. These two projects are some of the largest and most popular among those in [4]'s survey of NN program bugs. Using these annotations, aNNoTest generated 5655 test inputs, which triggered 56 crashes. Only 6 of them correspond to invalid inputs (i.e., they are spurious crashes), whereas the remaining 50 crashes expose actual bugs in the project.[5] Overall, aNNoTest generated 5649 valid tests exposing 50 bugs and thoroughly exercising the programs under test. Thus, aNNoTest can generate tests with good precision, and find bugs that require very specific inputs.

In the second experiment (*Reproduced bugs* in Tab. I), we went through a broader selection of 19 open-source NN programs studied in [4]. We opportunistically annotated all buggy functions reported by the survey [4], in order to trigger aNNoTest to reproduce those known bugs. Indeed, all tests generated by aNNoTest using these annotations were valid, and exposed 93 unique bugs; out of them, 62 correspond to the 81 bugs reported in [4], and 31 are previously unknown bugs. Thus, aNNoTest can also be used selectively and nimbly on parts of a project, and can achieve a high recall ($77\% = 62/81$) using Islam et al. as ground truth.[6]

B. Annotation Effort

Since aNNoTest crucially relies on annotations, it's important to assess the annotation effort that it requires in practical scenarios.

For experiment *Complete projects* in Tab. I, we wrote 2 annotations per function on average, 80% of functions had

[5]Since the projects are no longer maintained, we could not report the bugs officially, but there is abundant circumstantial evidence to confirm that they are genuine flaws.

[6]The density of bugs in the NN programs used in our experiments is consistent with Islam et al. [4]'s manual analysis.

3 annotations or less, 96% of all annotations fit a single line, and only 10% of all functions have more than 5 lines of annotations. For experiment *Reproduced bugs* in Tab. I, we wrote 6 annotations per function on average. It may be counterintuitive that this second experiment required more annotations per function than the first experiment—where we deliberately tried to be as thorough as possible with the annotations. The explanation is that less annotation reuse was possible in experiment *Reproduced bugs*, which involved a small number of different functions in each of the 19 analyzed projects.

Qualitatively, writing aN annotations is usually not a complex task for a programmer who is familiar with the functionality of the program that is being annotated and tested. Roughly, the first author spent around 10–15 minutes to annotate each function on average, including the time to get familiar with the projects; this average varies significantly from function to function, and from project to project—some are trivially derivative, other involve subtle constraints that are not obvious at first glance. Nevertheless, after understanding a function's range of valid inputs, expressing it as aN annotations is usually straightforward.

As evidence of aN's conciseness, consider that the Hypothesis test templates generated automatically by aNNoTest are, on average, 5.5 times bigger than the corresponding aN annotations. Thus, even though one could directly write Hypothesis test templates, aNNoTest automates away a good part of the tedious effort and lets users focus on understanding the requirements. aNNoTest's technique paper [10] provides more data and a broader discussion about the cost-effectiveness of using aNNoTest through its annotations.

C. Test Coverage

aNNoTest's automatically generated tests can also achieve high coverage, which compares favorably to tests written manually by project maintainers. To demonstrate this, we considered project Vision, PyTorch's machine vision library [16]; we had to select a different project, since the 19 projects from the survey [4] that we used in the rest of the experiments practically include no tests (the only exception is project ADV, which includes a single integration test).

From project Vision, we selected three of its larger-than-average modules; the developer-written tests for these modules achieve an average coverage[7] of 55% (not uniformly: 96%, 79%, and 16% branch coverage in each of the modules [10, Sec. 4.1]). After annotating these three modules using the same approach as the other experiments, aNNoTest generated tests that achieve an average branch coverage of 80%. (The results are similar if we consider statement coverage instead.)

Regarding conciseness, the annotations we wrote for the three modules of project Vision consist of 88 lines; for comparison, the manually written tests for the same modules span 597 lines of code.

[7]Measured using Python's `Coverage.py`.

V. Related Work

Testing is an essential task to ensure software quality [17]. For this reason, automated test generation has long been a central topic of research in software engineering. Many of the mature tools that are available target statically typed languages. For instance, Randoop [5] and EvoSuite [6] are two popular test-case generation tools for Java programs, respectively based on random testing and evolutionary algorithms. The typing annotations that are required by statically typed languages such as Java provide valuable information for test-case generation and other kinds of dynamic analysis.

In contrast, dynamically typed programming languages such as Python do not require static typing annotations, which makes developing test generation tools for them a novel challenge. As a result, fewer test-case generation tools are available for Python than for statically typed languages. Pynguin [8] is based on evolutionary algorithms, and supports type hints to narrow down the space of test generation. As we discussed in Sec. II, type hints are often insufficiently expressive to constrain the complex argument types found in NN programs. Deal [11] is a design-by-contract Python library that leverages pre- and postconditions for test generation and static analysis. As outlined in Sec. II, Deal's expressive language can easily encode argument constraints in NN programs; however, Deal's test-case generation is not driven by these constraints, and hence it may be ineffective. Hypothesis [14] is a property-based testing framework for Python programs, which we use as the back-end of aNNoTest. As we mentioned in Sec. IV, using aNNoTest's domain-specific annotation language aN instead of directly writing Hypothesis templates and strategies has advantages in terms of conciseness, and better fits the characteristics of NN programs.

Testing NN programs compounds the challenges of dynamically-typed languages with the observation that these programs and the bugs they usually contain are quite different in nature compared to those found in "traditional" programs [3], [4]. aNNoTest focuses on unit testing of NN programs at the implementation level; the bulk of the recent research on testing NN has focused, [1], [2], [18], [19].

VI. aNNoTest: Tool Availability

The aNNoTest tool is available as open source. The simplest way to use it is by installing it from PyPI[8] with `pip install annotest`.

aNNoTest's main repository[9] includes the tool's source code and instructions to use it. aNNoTest's current release at the time of writing is version 0.1; this is a more recent version than the one used for the experiments in the paper that presented the aNNoTest technique [10], which fixes several minor bugs and introduces minor improvements to its functionality.[10] The present paper's description refers to aNNoTest 0.1—although we remark that the differences with

[8]https://pypi.org/project/annotest/

[9]https://github.com/atom-sw/annotest

[10]The older aNNoTest version used for [10]'s experiments is still available in that paper's replication package.

the previous versions are really minor, in particular with respect to its behavior on the experiments reported in Sec. IV.

A companion repository[11] includes the source code of 62 bugs in open-source NN projects surveyed by Islam et al. [4]—which we reproduced using aNNoTest as described in Sec. IV. We curated this repository to ensure that each bug is easily reproducible using aNNoTest—or with any other Python source-code tool. In particular, each bug comes with an installation script that creates a virtual environment with all dependencies, our aN annotations, the bug-triggering tests generated by aNNoTest, and scripts to rerun the tests or aNNoTest on the program. Besides documenting several realistic examples of using aNNoTest, this repository can support further work in this area, by providing a curated collection of real-world reproducible NN bugs.

VII. Conclusions and Future Work

In this paper, we described aNNoTest: an automated test generation tool for NN programs. We provided the motivation behind aNNoTest, outlined how to use it, and reported experimental results indicating its effectiveness at finding bugs in NN programs.

aNNoTest is fundamentally oracle agnostic: it can use any user-provided oracle to identify faults. The experiments we conducted so far mostly use implicit crash oracles, or library assertion failures; this is consistent with the focus on NN bugs, where low-level failures and crashes are widespread [13]. As future work, we may extend aNNoTest with oracle-generation capabilities. Given its current design, a natural choice would be extending aN with syntax for *postconditions*, which can be used as oracles of correctness. Another interesting direction would be generating regression tests, similarly to what tools such as Pynguin (or Java's Evosuite [6]) already do.

[11]https://github.com/atom-sw/annotest-subjects

References

[1] S. Lee, S. Cha, D. Lee, and H. Oh, "Effective white-box testing of deep neural networks with adaptive neuron-selection strategy," in *Proc. ISSTA*, 2020, pp. 165–176.

[2] V. Riccio, G. Jahangirova, A. Stocco, N. Humbatova, M. Weiss, and P. Tonella, "Testing machine learning based systems: a systematic mapping," *Empir. Softw. Eng.*, vol. 25, no. 6, pp. 5193–5254, 2020.

[3] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang, "An empirical study on tensorflow program bugs," in *Proc. ISSTA*, 2018, pp. 129–140.

[4] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan, "A comprehensive study on deep learning bug characteristics," in *Proc. ESEC/FSE*, 2019, pp. 510–520.

[5] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proc. ICSE*, 2007, pp. 75–84.

[6] G. Fraser and A. Arcuri, "EvoSuite: automatic test suite generation for object-oriented software," in *Proc. ESEC/FSE*, 2011, pp. 416–419.

[7] M. Kessel and C. Atkinson, "Diversity-driven unit test generation," *Journal of Systems and Software*, vol. 193, p. 111442, 2022.

[8] S. Lukasczyk, F. Kroiß, and G. Fraser, "Automated unit test generation for Python," in *Proc. SSBSE*, 2020, pp. 9–24.

[9] "Pep 484 – type hints," https://www.python.org/dev/peps/pep-0484/.

[10] M. Rezaalipour and C. A. Furia, "An annotation-based approach for finding bugs in neural network programs," *Journal of Systems and Software*, vol. 201, p. 111669, 2023.

[11] "Deal," https://github.com/life4/deal.

[12] "Keras adversarial models," https://github.com/bstriner/keras-adversarial/.

[13] M. J. Islam, R. Pan, G. Nguyen, and H. Rajan, "Repairing deep neural networks: Fix patterns and challenges," in *Proc. ICSE*, 2020, pp. 1135–1146.

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

[15] "Keras & tensorflow implementation of progressive growing of gans for improved quality, stability, and variation," https://github.com/naykun/TF_PG_GANS.

[16] "TorchVision: PyTorch's Computer Vision library," https://github.com/pytorch/vision.

[17] G. Candea and P. Godefroid, "Automated software test generation: Some challenges, solutions, and recent advances," in *Computing and Software Science - State of the Art and Perspectives*, ser. Lecture Notes in Computer Science, B. Steffen and G. J. Woeginger, Eds. Springer, 2019, vol. 10000, pp. 505–531.

[18] J. Chen, J. Wang, X. Ma, Y. Sun, J. Sun, P. Zhang, and P. Cheng, "QuoTe: Quality-oriented testing for deep learning systems," *ACM Trans. Softw. Eng. Methodol.*, 2023.

[19] S. Gu, J. Liu, Z. Hui, W. Liu, and Z. Chen, "MetaA: Multi-dimensional evaluation of testing ability via adversarial examples in deep learning," in *Proc. QRS*, 2022, pp. 1004–1013.