The Journal of Systems & Software 201 (2023) 111669

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

An annotation-based approach for finding bugs in neural network programs ☆

Mohammad Rezaalipour *, Carlo A. Furia

Software Institute, USI Università della Svizzera italiana, Lugano, Switzerland

ARTICLE INFO

Article history:
Received 27 October 2022
Received in revised form 24 January 2023
Accepted 27 February 2023
Available online 8 March 2023

Keywords:
Test generation
Neural networks
Debugging
Python

ABSTRACT

As neural networks are increasingly included as core components of safety–critical systems, developing effective testing techniques specialized for them becomes crucial. The bulk of the research has focused on testing neural-network *models*; but these models are defined by writing programs, and there is growing evidence that these *neural-network programs* often have bugs too.

This paper presents ANNoTest: an approach to generating test inputs for neural-network programs. A fundamental challenge is that the dynamically-typed languages (e.g., Python) commonly used to program neural networks cannot express detailed constraints about valid function inputs (e.g., matrices with certain dimensions). Without knowing these constraints, automated test-case generation is prone to producing invalid inputs, which trigger spurious failures and are useless for identifying real bugs. To address this problem, we introduce a simple annotation language tailored for concisely expressing valid function inputs in neural-network programs. ANNoTest takes as input an annotated program, and uses property-based testing to generate random inputs that satisfy the validity constraints. In the paper, we also outline guidelines that simplify writing ANNoTest annotations.

We evaluated ANNoTest on 19 neural-network programs from Islam et al's survey. Islam et al. (2019), which we manually annotated following our guidelines — producing 6 annotations per tested function on average. ANNoTest automatically generated test inputs that revealed 94 bugs, including 63 bugs that the survey reported for these projects. These results suggest that ANNoTest can be a valuable approach to finding widespread bugs in real-world neural-network programs.

1. Introduction

Neural networks have taken the (programming) world by storm. With their capabilities of solving tasks that remain challenging for traditional software, they have become central components of software systems implementing complex functionality such as image processing, speech recognition, and natural language processing, where they can reach performance at or near human level. These tasks are widely applicable to domains such as automotive and healthcare, where safety, reliability, and correctness are critical. Therefore, the software engineering (research) community has been hard at work designing techniques to assess and ensure the dependability of software with neural network (NN) components.

Testing techniques, in particular, are being extensively developed to cater to the specific requirements of NN (and, more generally, machine learning) systems (Riccio et al., 2020). Most of this research focuses on testing NN *models*: instances of a specific NN architecture, trained on some data and then used to classify or transform new data. Testing a NN model entails assessing qualities such as its robustness and performance as a classifier. However, neural networks are programs too: a NN model is usually implemented in a programming language like Python, using frameworks such as Keras or TensorFlow. There is clear evidence that these *neural network programs* tend to be buggy (Islam et al., 2019; Humbatova et al., 2020); therefore, a technique for finding these bugs would be practically very useful and complement the extensive work on NN model testing (Zhang et al., 2022). This paper presents a novel contribution in this direction.

NN programs may seem simple by traditional metrics of complexity: for example, the average project size of the NN projects surveyed by Islam et al. (2019) is just 2165 lines of code; and the majority of the bugs they found are relatively simple ones such as crashes and API misuses. Nevertheless, other characteristics make traditional test-case generation techniques ineffective to test such programs. NN programs are written in dynamically typed languages like Python, where the type of variables is unknown statically. Without this information, generating valid inputs is challenging for generic techniques such as random testing and

☆ Editor: Aldeida Aleti.

* Corresponding author.
E-mail address: rezaam@usi.ch (M. Rezaalipour).

https://doi.org/10.1016/j.jss.2023.111669

genetic algorithms (Lukasczyk et al., 2020). Even if type annotations were available, NN programs routinely manipulate complex data structures – such as vectors, tensors, and other objects – whose precise "shape" is not expressible with the standard types (integers, strings, and so on). As we demonstrate in Section 2 and 4.3.4, without such precise information automated test case generation tends to generate many invalid inputs that trigger spurious failures.

This paper presents ANNoTest: an approach to automatically generating bug-finding inputs for NN program testing. A key component of ANNoTest (described in Section 3) is AN: a simple annotation language to concisely and precisely express the valid inputs of functions in NN programs. The AN language supports expressing the kinds of constraints that are needed in NN programs (for example: a variable should be a vector of size from 2 to 5 with components that are positive integers). AN is also easily extensible to accommodate other constraints that a specific NN program may need to encode.

Given an annotated NN program, ANNoTest automatically generates unit tests for the program that span the range of valid inputs. To this end, the current implementation of ANNoTest uses property-based testing (more precisely, the Hypothesis (MacIver et al., 2019) test-case generator). Using the AN language decouples specifying the constraints from the back-end used to generate the actual tests; therefore, different back-end tools could also be used that better suite the kinds of constraints used in a project's annotations.

Section 4 describes an extensive experimental evaluation of ANNoTest, targeting 19 open-source NN programs, manually analyzed by Islam et al. (2019), using some of the most widely used NN frameworks (Keras, TensorFlow, and PyTorch). After we manually annotated 24 functions, ANNoTest generated tests triggering 63 known bugs reported by Islam et al. (2019) for these functions, as well 31 previously unknown bugs. To experiment with ANNoTest's capabilities when used extensively, we also annotated all functions in two larger NN projects; the total of 330 annotations that we wrote enabled ANNoTest to discover 50 bugs with only 6 false positives. These experiments demonstrate that ANNoTest can be used both extensively on a whole project, and opportunistically on only a few selected functions that are critical. Since our evaluation is based on Islam et al.'s (2019) extensive survey, it can assess ANNoTest's capabilities of finding relevant bugs in real-world NN programs. In other experiments, we quantify the amount of annotations needed by ANNoTest, compare it to generic (non NN-specific) test-case generators for Python, as well as to developer-written tests, so as to better understand the trade-off between programmer effort and quality assurance benefits it offers.

In summary, this paper makes the following contributions:

- ANNoTest: an approach for test-case generation geared to the characteristics of NN programs.
- AN: a simple annotation language capable of concisely expressing precise constraints on the valid inputs of functions in NN programs, with basic guidelines to use it.
- An experimental evaluation of ANNoTest's bug-finding capabilities on 19 open-source NN projects surveyed by Islam et al. (2019).
- For reproducibility, the implementation of ANNoTest and all experimental artifacts are publicly available: https://figshare.com/s/00ef658a6a51cccbaed6

Scope. While ANNoTest is applicable, in principle, to any Python programs – not just NN programs – it was designed to primarily cater to the characteristics of NN programs. As we will see concretely with Section 2's example, NN programs often involve complex constraints on their inputs, which are impossible or highly impractical to express using Python's type hints annotations. ANNoTest provides annotations that go beyond type hints, and hence are especially useful for the kinds of constraints that we commonly find in NN programs. On the other hand, being able to *express* complex constraints is not sufficient to build tests automatically; as we will see in Section 4.3.4, *generating* inputs that satisfy the constraints is challenging; simple strategies such as generating input at random and then filtering them using the constraints are mostly ineffective. ANNoTest defines suitable generators for each of its constraints, so that valid inputs can be generated efficiently and automatically even for the complex combinations of input constraints that are common in NN programs.

2. An example of using ANNoTest

DenseNet (DenseNetprojectpage:) is a small Python library that implements densely connected convolutional networks (Huang et al., 2019) (a NN architecture where each layer is directly connected to every other layer) on top of the Keras framework. Listing 1 shows a slightly simplified excerpt of function `DenseNet` – the main entry point to the library – in an earlier version of the project. (https://github.com/cmasch/densenet/blob/70ee31d0f6f800324fbe98ea687122395248d39e/densenet.py)

The complete implementation of function `DenseNet` comprises 34 lines of code (excluding comments and empty lines), and follows a straightforward logic: after checking the input arguments (code in Listing 1), it combines suitable instances of Keras classes to model a densely connected network, and finally returns a model object to the caller. Listing 1's code, however, has a bug at line 11 — one of the bugs collected in Islam et al.'s (2019) survey. The expression assigned to `dense_layers` is a floating point number because the division operator `/` always returns a float in Python 3; however, if `dense_layer` is not an integer, a later call in `DenseNet`'s code to the Keras library fails. DenseNet's developers discovered the bug and fixed it (by adding an `int` conversion at line 11) in a later project revision. (https://github.com/cmasch/densenet/commit/693d772ae9dcdb4d524b25d7d2f6428de4a524ff#diff-813086a9be01b05b352f0111384c48e74735b009e22f4bab1f3dcaa06e2303c2R68)

`DenseNet`'s implementation is deceptively simple: despite its small size and linear structure, it only accepts input arguments in very specific ranges. Argument `input_shape`, for example, corresponds to a so-called *shape tuple* of integers; in `DenseNet`, it should be a triple of integers with first element at least 20. If the first element is less than 20, `DenseNet` eventually fails while trying to create a layer with a negative dimension — which violates an assertion of the Keras library. Another example is argument `dense_layers`, which can be an integer or an integer list; if it is the latter, its length must be equal to argument `dense_block`, or `DenseNet` terminates at line 9 with an assertion violation.

Without knowing all these details about valid inputs, testing `DenseNet` using a general-purpose automated test-case generator would trigger lots of spurious failures[1] when executing tests that call `DenseNet` with invalid inputs. The few failing but valid tests that trigger bugs such as that in Listing 1 would be a needle in the haystack of all invalid tests, thus essentially making

[1] For example, Pynguin Lukasczyk et al. (2020) generates 8 tests, all invalid and none triggering the failure at line 11. With type hints (supported by Pynguin), it generates 5 tests, 4 invalid and none triggering (any) failure. Section 4.3.4 describes more experiments with Pynguin. (As we discuss in Section 4.1, Pynguin does not work with the version of TensorFlow used by Listing 1's code; thus, we mocked the relevant library calls in this example.)

```
1  def DenseNet(input_shape=None, dense_blocks=3, dense_layers=-1, growth_rate=12, nb_classes=None,
2          dropout_rate=None, bottleneck=False, compression=1.0, weight_decay=1e-4, depth=40):
3    if nb_classes == None:
4      raise Exception('Please define number of classes')
5    if compression <= 0.0 or compression > 1.0:
6      raise Exception('Compression must be between 0.0 and 1.0.')
7    if type(dense_layers) is list:
8      if len(dense_layers) != dense_blocks:
9        raise AssertionError('Dense blocks must be the same as layers')
10   elif dense_layers == -1:
11     dense_layers = (depth - 4) / 3   # Bug: division / returns a float
12   # ... 23 more lines of code ...
```

Listing 1: An excerpt of function `DenseNet` from project DenseNet. The code has a bug on line 11.

```
13 @arg(input_shape): tuples(ints(min=20, max=70), ints(min=20, max=70), ints(min=1, max=3))
14 @arg(dense_blocks): ints(min=2, max=5)
15 @arg(dense_layers): anys(-1, ints(min=1, max=5), int_lists(min_len=2, max_len=5, min=2, max=5))
16 @arg(growth_rate): ints(min=1, max=20)
17 @arg(nb_classes): ints(min=2, max=22)
18 @arg(dropout_rate): floats(min=0, max=1, exclude_min=True, exclude_max=True)
19 @arg(bottleneck): bools()
20 @arg(compression): floats(min=0, max=1, exclude_min=True)
21 @arg(weight_decay): floats(min=1e-4, max=1e-2)
22 @arg(depth): ints(min=10, max=100)
23 @require(type(dense_layers) is not list or len(dense_layers)==dense_blocks)
```

Listing 2: AN annotations for function `DenseNet` in Listing 1.

automated test-case generation of little help to speed up the search for bugs.

To precisely and concisely express the complex constraints on valid inputs that often arise in NN programs, we designed the AN annotation language — which is a central component of the ANNOTEST approach. Listing 2 shows annotations written in AN[2] that characterize `DenseNet`'s valid inputs. Whereas Section 3 will present AN's features in greater detail, it should not be hard to glean the meaning of the annotations in Listing 2. For example, the first annotation encodes the aforementioned constraint on `input_shape`, and the last one expresses the relation between `dense_layers` and `dense_blocks` when the former is a list. It should also be clear that AN's expressiveness is much greater than what is allowed by the standard programming-language types — such as Python's type hints. (Typehints:)

Equipped with the annotations in Listing 2, ANNOTEST generates and runs 36 unit tests for `DenseNet` in 53 s. All the tests are valid, and only one is failing, reaching Listing 1's line 11 and then ending with a failure due to `dense_layers` being a float that we described above — precisely revealing the bug.

The experimental evaluation of ANNOTEST– described in Section 4 – will analyze many more NN programs whose characteristics, input constraints, and faulty behavior are along the same lines as the example discussed in this section. This will demonstrate ANNOTEST's capabilities of precisely testing and finding bugs in NN programs.

3. How ANNOTEST works

Fig. 1 overviews the overall process followed by the ANNOTEST approach. To test a NN program with ANNOTEST, we first have to annotate its functions (including member functions, that is methods) using the AN annotation language (Section 3.1). This is the only step that is manual, since the annotations have to encode valid inputs of the tested functions — the same kind of information that is needed to write unit tests. Section 3.2 provides guidelines that help structure the manual annotation process so that it only requires a reasonable amount of effort; furthermore, users do not need to annotate a whole program but only those functions that they want to test with ANNOTEST. Then, the ANNOTEST tool takes as input an annotated program and generates unit tests for it. To this end, it encodes the constraints expressed by the AN annotations in the form of test templates for the property-based test-case generator Hypothesis (Section 3.4); then, it runs Hypothesis which takes care of generating suitable tests. Finally, the generated unit tests can be run as usual to find which are passing and which are failing — and thus expose some bugs in the NN program (Section 3.5).

3.1. The AN annotation language

By writing annotations in the AN language, developers can precisely express the valid inputs of a function in a NN program.[3] To this end, AN provides *type* annotations (Section 3.1.1) and *preconditions* (Section 3.1.3), as well as an extension mechanism to define arbitrarily complex constraints (Section 3.1.2). In addition, AN offers a few *auxiliary* annotations (Section 3.1.4), which encode other kinds of information that is practically useful for test-case generation.

3.1.1. Type annotations

A *type* annotation follows the syntax `@arg(v):T`, where `v` is a function argument (parameter), and T is a *type constraint* that

[2] The AN annotations in the paper use a pretty-printed and slightly simplified syntax.

[3] Directly annotating the source code, rather than having a separate generator used only when testing, also helps keep the annotations consistent with the implementation.

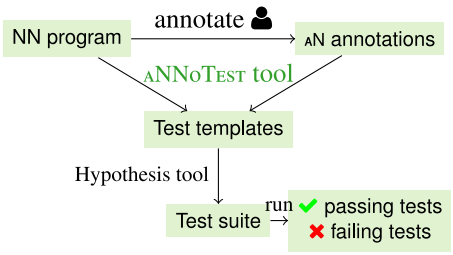

Fig. 1. An overview of how the ANNOTEST approach works.

specifies a set of possible values for `v`. A type annotation refers to the function that immediately follows it in the source code. A function can have up to as many type annotations as it has arguments.

AN supports several different *type constraints*, which can express a broad range of constraints — from simple ones, such as those that are also expressible using Python's type hints, up to complex instances of special-purpose classes. The simplest, and most specific, type constraint uses keyword **`froms`**[4] to enumerate a list of valid values. For example, constraint **`froms`**`([0, 0.0, None, zero()])` corresponds to any of the four values: integer zero, floating-point zero, `None`, and what is returned by the call `zero()`.

Constraints for **atomic types** specify that an argument is a Boolean (**`bools`**), an integer number (**`ints`**), or a floating-point number (**`floats`**). Integer arguments can be restricted to a range between `min` and `max` values; for example, Listing 2's line 17 constrains `nb_classes` to be an integer between 2 and 22. Floating-point arguments can also be restricted to ranges, and the ranges can be open, closed, or half-open; for example, Listing 2's line 20 constrains `compression` to be a number in the half-open interval $(0, 1]$ which includes 1 but excludes 0. Floating-point constraints also support including or excluding the special values `NaN` and `Inf`, as well as the precision (in bits) of the generated floating point values.

Constraints for **sequences** specify that arguments are Python **`lists`**, **`tuples`**, or an array in the NumPy (NumPy:) library (which is widely used in NN programs, as well as other data-intensive applications). Lists and tuples can have any number of elements, whose possible values are also constrained using AN's type constraints. For example, Listing 2's line 13 specifies a tuple with 3 integer elements: the first and second one between 20 and 70, and the third one between 1 and 3. AN also includes shorthands for lists with homogeneous elements: Listing 2's line 15 uses shorthand **`int_lists`** to specify lists of length between 2 and 5, whose elements are integers between 2 and 5.

The *shape* of a NumPy array is a tuple of positive integers that characterize its size. For example, the tuple `(256, 256, 3)` is the shape of a 3-dimensional array whose first two dimensions have size 256 and whose last dimension has size 3; arrays with this shape can represent 256×256 pixel color pictures. Type constraint **`np_shapes`** specifies arguments that represent shapes with a certain range of possible dimensions and sizes. For example, **`np_shapes`**`(min_dims=3, max_dims=3)` are the shapes of all 3-dimensional arrays whose dimensions can have any size.

Type constraint **`np_arrays`** specifies NumPy array arguments with any shape and whose elements have any of the valid NumPy types. The shape can be constrained by an **`np_shapes`** annotation or given directly as a tuple. For example, using the shape mentioned in the previous paragraph, **`np_arrays`**`(np_type=dtype("uint32"), shape=(256,256,3))` specifies $256 \times 256 \times 3$ arrays whose components are unsigned 32-bit integers (one of NumPy's data-types), which could represent random color pictures.

Type constraints for **maps** specify Python's widely used associative dictionaries: **`dicts`**$(K, V,$ `min_size, max_size`$)$ corresponds to all subsets of the Cartesian product $K \times V$ with between `min_size` and `max_size` elements, where K and V are type constraints that apply to the keys and values respectively. A typical usage of this is to constraint Python's optional keyword argument `**kwargs`. For example, Listing 3 shows how we used **`dicts`** to constrain the `**kwargs` argument of function `dim_ordering_reshape` (https://github.com/bstriner/keras-adversarial/blob/master/examples/image_utils.py#L34) (from a project using NN models to simulate multi-player games), so that it simply consists of all mappings from string ``` ``input_shape'' ``` to singletons representing the shapes of monodimensional arrays.

To express the **unions** of several type constraints, AN includes the **`anys`** type constraint, which specifies the union of its arguments. For example, Listing 2's line 15 says that `dense_layers` can be any of: (i) the number -1, (ii) an integer between 1 and 5, or (iii) an integer list with between 2 and 5 elements that are between 2 and 5.

3.1.2. Custom generators

While AN's type annotations can define a broad range of frequently used constraints, they cannot cover all cases that one may encounter in practice. To support *arbitrary* type constraints, AN includes the **`objs`**`(gen)` annotation. This is used as a type constraint, and identifies all values that are produced by the user-provided *generator* function `gen`. Function `gen` must be visible at the entry of the functions whose annotations refer to it; `gen` itself is marked with the annotation `@generator`.

For instance, Listing 4 shows the annotations we wrote for function `build_gan` (https://github.com/bstriner/keras-adversarial/blob/master/keras_adversarial/adversarial_utils.py#L10) (from the same project as Listing 3). The function combines two Keras model instances, `generator`[5] and `discriminator`, to build GANs (Generative Adversarial Networks (Goodfellow et al., 2014)). These instances are complex objects that are built by calls to the Keras library; therefore, we introduced two custom generators, `gan_gens` and `gan_discs`, that construct such instances for testing `build_gan`. Listing 4 shows `gan_gens`'s implementation: the generator's input are constrained by using AN's type annotations as usual; ANNOTEST will use `gan_gens`'s output as input for `build_gan`.

Whereas generators such as `gan_gens` may look daunting to write at first, we found that they simply encapsulate existing snippets of the project that call the function under test (`build_gan` in Listing 4). Based on this observation, Section 3.3 presents a simple process to build generators by combining common refactoring steps; this drastically alleviates the effort to write generators, reducing it to just selecting the right snippets of client code in the project.

3.1.3. Preconditions

Argument annotations constrain each function argument individually. *Preconditions* may express constraints that affect multiple arguments simultaneously: `@require`(P), where P is a Python Boolean expression, specifies that a function's arguments

[4] AN type constraints use names that are "pseudo-plurals" (by adding a trailing `s`) of the corresponding Python types. This avoids using reserved keywords and also conveys the idea that a type constraint identifies a set of values. This convention is also customary in property-based testing (Claessen and Hughes, 2000).

[5] It is just a coincidence that one argument is also named "generator".

```
24 @arg(k): ints(min=1, max=1000)
25 @arg(w): ints(min=1, max=1000)
26 @arg(kwargs): dicts(keys=froms(["input_shape"]), values=np_shapes(min_dims=1, max_dims=1))
27 def dim_ordering_reshape(k, w, **kwargs):
```

Listing 3: An example of AN annotations for a function with keyword arguments.

```
28 @arg(generator): objs(gan_gens)
29 @arg(discriminator): objs(gan_discs)
30 @arg(name): froms(["gan1", "gan2", "gan3",
31            "gan4", "gan5"])
32 def build_gan(generator, discriminator,
33         name="gan"):
34   # ...
35
36 @generator
37 @exclude
38 @arg(latent_dim): ints(min_value=1,
39              max_value=1000)
40 @arg(input_shape): np_shapes(min_dims=2)
41 def gan_gens(latent_dim, input_shape):
42   from examples.example_gan import
       model_generator
43   generator = model_generator(latent_dim,
44                     input_shape)
45   return generator
```

Listing 4: An example of using type constraint `objs` and a custom generator function.

must be such that P evaluates to true. A precondition refers to the function that immediately follows it in the source code. Expression P may refer to any arguments of the specified function, as well as to any other program element that is visible at the function's entry (such as other class members). A function can have any number of preconditions, all of which constraint the function's argument. For example, Listing 2's line 23 requires that, whenever argument `dense_layer` is a list, it should have as many elements as the value of integer argument `dense_blocks`.

3.1.4. Auxiliary annotations

The AN language includes a few more features to control the test-generation process. Functions marked with `@exclude` are not tested (such as generator `gan_gens` in Listing 4). Annotation `@timeout` introduces a timeout to the unit tests generated for the function it refers to.

Python modules may include snippets of code that is not inside any functions or methods but belongs to an implicit "main" environment. ANNOTEST will generate tests for this environment for any module that is annotated with `@module_test`. Since modules do not have arguments, these tests simply import and execute the main environment. This is a simple feature, but practically useful since some of the NN program bugs that were surveyed (Islam et al., 2019) are located in the main environment.

To test an instance method `m`, one needs to generate an instance `o` of `m`'s class `C` to use as target of the call to `m`. To this end, `C`'s constructor is called. The constructor may also be equipped with AN annotations; as a result, testing `m` entails also testing `C`'s constructor. This can be a problem if the constructor has bugs that prevent a correct execution of `m`. To handle this scenario, AN includes the annotation `@cc_example`, which supplies a constructor with a list of concrete inputs for it. If `C`'s constructor is equipped with this annotation, ANNOTEST will only call it using

```
46 @arg(image_path):
47   froms(["image1.png", "image2.png",
48        "image3.png", "image4.png"])
49 @arg(generator): objs(grids)
50 @arg(cmap):
51  froms(['gray', 'bone', 'pink',
52      'spring', 'summer', 'cool'])
53 @cc_example(["image1.png",
54        grids(3, 6, 6, 3), 'gray'])
55 def __init__(self, image_path,
56        generator, cmap='gray'):
57  # ...
```

Listing 5: An example of using the `cc_example` auxiliary annotation on the constructor of class `ImageGridCallback`.

the inputs given by the `@cc_example` annotation when it needs to create instances to test any methods of `C`. This way, one can effectively decouple testing a class's constructor from testing the class's (regular) methods, so that any bugs in the former do not prevent testing of the latter. For example, the constructor of class `ImageGridCallback` (https://github.com/bstriner/keras-adversarial/blob/master/keras_adversarial/image_grid_callback.py#L7) shown in Listing 5 is regularly tested through its type annotations; however, when it is used to construct instances of the class to test other methods, it is only called with the more restricted set of inputs specified by the `@cc_example` annotation. The example also demonstrates that a generator function (`grids` in this case) can also be used as a regular function (second component of `@cc_example`).

3.2. Annotation guidelines

To test a NN program using ANNOTEST, one must first annotate the functions to be tested using the language described in Section 3.1. Ultimately, writing suitable annotations requires knowledge about the program's specification — that is, its intended behavior. The very same knowledge is necessary to write *unit tests* for the programs; the only difference is that a test supplies individual (valid) inputs, whereas an annotation can capture a range of possible (valid) inputs.

This entails that the effort of writing annotations (or tests) for a project depends on whether the programmer already has this knowledge – typically, because they are developers of the project under test – or is trying to test a project they are not familiar with. In this section, we focus on the latter, more challenging scenario. To help such a process of "discovery" – figuring out suitable annotations for NN programs written by others – and to make it cost-effective, we present some simple guidelines that suggest which artifacts to inspect and in which order. In the experiments described in Section 4, we followed these guidelines to annotate NN projects systematically and with reasonable effort — despite our previous lack of familiarity with those codebases.

Consider a Python function `f` in some NN project `P` that we would like to test. If `f`'s behavior (and, in particular, the

Table 1
Guidelines to inspect the implementation of a NN function `f` to suggest how to annotate it using AN's annotation language. Each SOURCE of information in `f` or elsewhere in `f`'s project `P` suggests matching AN ANNOTATIONS.

	SOURCE	ANNOTATIONS
1	calls of `f` in its project `P`	basic type annotations `@arg`
2	assertions and exceptions raised by `f`'s implementation	refined type annotations `@arg`, preconditions `@require`
3	calls of NN framework functions in `f`'s implementation	refined type annotations `@arg`, preconditions `@require`, custom generators
4	calls of other functions `g` in `P`	annotations of `g`

constraints on its inputs) is documented in the project, this documentation should be the first source of information to write AN annotations. However, if `f` lacks any (precise) documentation,[6] we will have to inspect its implementation. Table 1 lists four sources of information about `f`'s valid inputs in increasing level of detail.

To bootstrap the process, we inspect any usage of `f` within the NN program `P`. Since we focus on testing *programs*, not libraries, it is likely that every major function is called somewhere in `P`. These calls of `f` provide basic examples of valid inputs, which we loosely encode using AN's type annotations of Section 3.1.1. In Listing 1's example, looking at usages of `DenseNet` indicates that `input_shape` should be a triple of `int`, `compression` should be a `float`, and so on.[7]

Next, we look into `f`'s implementation for any (implicit or explicit) *input validation*. Often, a function uses exceptions or assertions to signal invalid input arguments. This information is useful to *refine* the basic type annotations, and may also suggest constraint that involve multiple arguments — which we can encode using AN's preconditions of Section 3.1.3. In Listing 1's example, `DenseNet`'s initial validation clearly indicates, among other things, `compression`'s precise interval of validity, and the precondition on line 23 in Listing 2.

The library functions from some NN framework used in `f`'s implementation may also (indirectly) introduce requirements on `f`'s inputs or otherwise suggest plausible ranges of variability. Indirect constraints may be more complex, and may even require custom generators (Section 3.3). In the running example, a call to Keras's `Convolution2D` constructor in `DenseNet` (not in Listing 1) suggests the range for argument `weight_decay` at line 21 in Listing 2.

Whenever `f`'s implementation calls other functions in the same project, this process can be repeated for these other functions, thus ensuring the consistency of the other functions' and `f`'s annotations. In the running example, `DenseNet` calls in a loop another function `dense_block` in the same project, passing `growth_rate` as argument and then incrementing it in each iteration. The input constraints of `dense_block`, once figured out, indirectly suggest the validity range for `DenseNet`'s `growth_rate` at line 16 in Listing 2.

The guidelines we described are flexible and remain useful even if they are not followed in full. For example, sometimes we found it useful to start from very narrow annotations (merely encoding the available examples of usages of `f` in `P`) and relax them as we discovered more information – rather than going from basic to specific as we did in most examples – since this allowed us to generate some sample tests early on. The guidelines are also applicable with different levels of exhaustiveness, regardless of whether your goal is to annotate as much as possible in a project, or just test a few selected functions. In the former case, it is advisable to start annotating the simplest, shortest functions, so that their annotations can then suggest how to annotate the more complex, longer ones.

3.3. Building custom generators by refactoring

As presented in Section 3.1.2, annotation `@arg(a): @objs(f)` tells ANNOTEST to use a *custom generator* function `f` in order to build suitable inputs for some argument `a`. In principle, `f` may be an arbitrarily complex piece of code; in practice, we found that the very projects we are annotating already include snippets of code that can be reused as generators of complex objects. In this section, we demonstrate, on an example, how to build such generators by applying a few refactoring operations to the relevant snippets of code. Modern IDEs such as PyCharm(ThePyCharmPythonIDE) can automate such refactoring steps. This drastically reduces the effort of building custom generators to just selecting the right snippets of code and doing some copy-pasting in the IDE.

Listing 6 shows the signature of function `G_convblock` (https://github.com/naykun/TF_PG_GANS/blob/master/Tensorflow-progressive_growing_of_gans/model.py#L21) in project GANS (described in Section 4.1); the first function argument `net` expects objects encoding Keras network architectures. This complex type is not directly supported by AN's built-in annotations; thus, we should define a custom generator function `generator_G_convblock` that builds valid instances of the type.

To this end, we first look for any *client code* of `G_convblock`. Another function `Generator` in project GANS, shown in Listing 7, calls `G_convblock` (line 85) after building a suitable network architecture object (line 83). Thus we can use parts of `Generator` to build `generator_G_convblock`: the "extract function" refactoring (Extractfunctionrefactoring) applied to lines 75–83 in Listing 7 outputs Listing 8's generator function. Now, `generator_G_convblock` is a new function, which we can annotate like any other functions that is processed by ANNOTEST.

In this example it was easy to identify a contiguous sequence of statements and extract it into a generator function. In other cases, the relevant client code may mix statements useful for the generator with others that pertain to a different functionality. In these cases, we can simply extract a larger snippets of code, and then refactor it to remove unused statements. In Listing 7's example, we could extract all lines 66–83 into a new function; then, all statements before line 75 are not used by the final line 83, and thus can be removed from the generator (leading to the same generator as in Listing 8). In all the experiments of this paper, these simple refactoring steps were sufficient to build all necessary custom generator functions.

3.4. Test generation

The annotations written in the AN language supply all the information that is needed to generate unit tests for every annotated function. In principle, we could use any technique for test-case generation and then filter any generated tests, keeping only those that comply with the annotations. However, the experiments reported in Section 4.3.4 indicate that such an aimless strategy would be inefficient, especially given the dynamically typed nature of Python.

Instead, ANNOTEST uses *property-based test-case generation* to actively match the constraints introduced by AN annotations.

[6] Many of the NN programs we used in Section 4's experiments are sparsely documented.

[7] For example, the `README.md` file in DenseNet's repository presents an example of using function `DenseNet` where argument `input_shape` is set to the triple `(28, 28, 1)`.

```
@arg(net): objs(generator_G_convblock)
def G_convblock(net, num_filter, filter_size, actv, init,
          pad='same', use_wscale=True, use_pixelnorm=True, use_batchnorm=False, name=None):
  # ... 24 lines of body code ...
```

Listing 6: Signature of project GANS's function `G_convblock`, whose first argument `net` requires a custom generator.

```
def Generator(num_channels=1, resolution=32, label_size=0, fmap_base=4096, fmap_decay=1.0,
        fmap_max=256, latent_size=None, normalize_latents=True, use_wscale=True,
        use_pixelnorm=True, use_leakyrelu=True, use_batchnorm=False, tanh_at_end=None,
        **kwargs):
  R = int(np.log2(resolution))
  assert resolution == 2 ** R and resolution >= 4
  cur_lod = K.variable(np.float32(0.0), dtype='float32', name='cur_lod')

  def numf(stage): return min(int(fmap_base / (2.0 ** (stage * fmap_decay))), fmap_max)
  if latent_size is None:
    latent_size = numf(0)
  (act, act_init) = (lrelu, lrelu_init) if use_leakyrelu else (relu, relu_init)

  inputs = [Input(shape=[latent_size], name='Glatents')]
  net = inputs[-1]

  if normalize_latents:
    net = PixelNormLayer(name='Gnorm')(net)
  if label_size:
    inputs += [Input(shape=[label_size], name='Glabels')]
    net = Concatenate(name='G1na')([net, inputs[-1]])
  net = Reshape((1, 1,K.int_shape(net)[1]), name='G1nb')(net)

  net = G_convblock(net, numf(1), 4, act, act_init, pad='full', use_wscale=use_wscale,
            use_batchnorm=use_batchnorm, use_pixelnorm=use_pixelnorm, name='G1a')
  # ... 20 more lines of code ...
```

Listing 7: An excerpt of project GANS's function `Generator`, a client of Listing 6's function `G_convblock`.

```
@generator
@exclude
@arg(latent_size): ints(min=1)
@arg(normalize_latents): bools()
@arg(label_size): ints()
def generator_G_convblock(label_size, latent_size, normalize_latents):
  inputs = [Input(shape=[latent_size], name='Glatents')]
  net = inputs[-1]
  if normalize_latents:
    net = PixelNormLayer(name='Gnorm')(net)
  if label_size:
    inputs += [Input(shape=[label_size], name='Glabels')]
    net = Concatenate(name='G1na')([net, inputs[-1]])
  net = Reshape((1, 1, K.int_shape(net)[1]), name='G1nb')(net)
  return net
```

Listing 8: The custom generator for argument `net` of Listing 6's function `G_convblock`, built by factoring out lines 75–83 in Listing 7.

More precisely, the current implementation of ANNOTEST uses the Hypothesis property-based test-case generator (Hypothesis:) through its API. To test a Python function using Hypothesis, we have to write a *test template*, which consists of a *parametric* unit test method that calls a collection of *strategies*. A strategy is a sort of generator function, which outputs values of a certain kind. A parametric test method calls some of the strategies, combines their outputs, and uses them to call the function under test.

ANNOTEST automatically builds a suitable Hypothesis strategy for each `@arg` annotation. Hypothesis provides built-in strategies that cover basic type annotations, such as Python's atomic types and tuples. ANNOTEST reuses the built-in strategies whenever possible, and combines them to generate values for more complex or specialized constraints (such as `int_lists`). For instance, Listing 9 shows parts of the parametric tests generated by ANNOTEST to encode the annotations in Listing 2's running example.

```
103 @given(input_shape=tuples(integers(min_value=20, max_value=70),
104                integers(min_value=20, max_value=70),
105                integers(min_value=1, max_value=3)),
106     dense_blocks=integers(min_value=2, max_value=5),
107     dense_layers=one_of(st.just(-1),
108                integers(min_value=1, max_value=5),
109                int_lists_an(min_len=2, max_len=5, min=2, max=5)),
110     growth_rate=integers(min_value=1, max_value=20),
111     nb_classes=integers(min_value=2, max_value=22),
112     dropout_rate=floats(min_value=0, max_value=1,
113                 exclude_min=True, exclude_max=True),
114     bottleneck=booleans(),
115     compression=floats(min_value=0, max_value=1, exclude_min=True),
116     weight_decay=floats(min_value=0.0001, max_value=0.01),
117     depth=integers(min_value=10, max_value=100))
118 @settings(deadline=None, suppress_health_check=[HealthCheck.filter_too_much,
119                             HealthCheck.too_slow])
120 def test_DenseNet(input_shape, dense_blocks, dense_layers, growth_rate,
121           nb_classes, dropout_rate, bottleneck, compression,
122           weight_decay, depth):
123   assume(type(dense_layers) is not list or
124       len(dense_layers) == dense_blocks)
125   DenseNet(input_shape, dense_blocks, dense_layers, growth_rate,
126         nb_classes, dropout_rate, bottleneck, compression,
127         weight_decay, depth)
128
129 @defines_strategy()
130 def int_lists_an(min_len=1, max_len=None, min=1, max=None):
131   if max_len is None:
132     max_len = min_len + 2
133   if max is None:
134     max = min + 5
135   return lists(integers(min, max),
136           min_size=min_len, max_size=max_len)
```

Listing 9: Hypothesis test template built by ANNoTest for `DenseNet`'s annotations in Listing 2.

ANNoTest reuses Hypothesis's built-in strategies `integers` (line 103) and `floats` (line 112); and combines Hypothesis strategies `lists` and `integers` (lines 129–136) to render AN's **`int_lists`** type constraint.

To encode arbitrary **`objs`** annotations (Section 3.1.2), ANNoTest first builds strategies for the annotations of each user-written custom generator function, as if it was testing the generator; then, it combines them to build a new strategy that follows the generator's implementation to output the actual generated objects — used as inputs for the function under test.

To encode `@require` annotations (preconditions), ANNoTest uses Hypothesis's `assume` function. When test-case generation reaches an `assume`, it checks whether its Boolean argument evaluates to true: if it does, generation continues as usual; if it does not, the current test input is discarded, and the process restarts with a new test. Thus, `assumes` can effectively act as filters to further discriminate between test inputs — a feature that ANNoTest leverages to enforce precondition constraints where appropriate in a parametric test. Line 123 in Listing 9 shows an example of using `assume` to encode the running example's precondition (line 23 in Listing 2).

After translating the annotations into suitable test templates, ANNoTest simply runs Hypothesis on those templates. The property-based test-case generator "runs" the templates to build unit tests that satisfy the encoded properties; it also runs these unit tests, and reports any failure to the user. Hypothesis's output is also ANNoTest's final output to the user.

Alternative back-ends. ANNoTest's current implementation uses Hypothesis as back-end, since property-based testing is a framework for defining testing properties in a naturally *generative* way. However, using other test-input generation engines as back-end is possible in principle. Automatically translating all AN annotations to preconditions (Boolean predicates) is straightforward, which could be passed to a tool like Deal (Anon, 2018). As we demonstrate in Section 4.3.4, Deal is not very effective at *generating* inputs that satisfy the preconditions, when these encode the complex combinations of constraints that are common in NN programs; however, Deal can also use preconditions for *static checking*, which would provide a complementary usage of ANNoTest's annotations. Pynguin Lukasczyk et al. (2020) is a general-purpose test-case generator for Python. In order to use it as a back-end for ANNoTest, we could leverage its genetic algorithm, which tries to maximize the *branch coverage* of the tests it generates. As done with EvoSuite (a test-case generation tool for Java that is also based on genetic algorithms) in related work (Fraser and Arcuri, 2015; Galeotti et al., 2015), one could express the input constraints as a series of branches in the instrumented program, so that Pynguin would be driven to find inputs that "pass" all the constraints — the valid inputs that we are looking for.

3.5. Failing tests and oracles

The ANNoTest approach, and the AN annotation language on which it is based, works independent of how a test is classified as failing or passing. In other words, ANNoTest generates test inputs that are consistent with the annotations; determining whether

the resulting program behavior is correct requires an *oracle* (Barr et al., 2015). In this paper, we only ran the tests generated by aNNoTest with *crashing* oracles: an execution is *failing* when it cannot terminate normally, that is it leads to an assertion violation, an unhandled exception, or some other low-level abrupt termination.

While crashing bugs are the most frequent ones, NN programs also exhibit other kinds of bugs such as performance loss, data corruption, and incorrect output (Islam et al., 2019). In principle, if we equipped the NN programs with oracles suitable to detect such kinds of bugs, aNNoTest could still be used to generate test inputs. However, some of these bug categories may be easier to identify by testing a NN at a different level than the bare program code. For example, bugs that lead to poor robustness of a NN classifier involve testing a fitted model rather than the model's implementation (Sun et al., 2019; Hu et al., 2019; Shen et al., 2018). Revisiting the aNNoTest approach to make it applicable to different kinds of oracles belongs to future work.

4. Experimental evaluation

The experimental evaluation aims at determining whether the aNNoTest approach is effective at detecting real bugs in NN programs, and whether it requires a reasonable annotation effort. Precisely, we address the following research questions:

RQ1. Does aNNoTest generate tests that expose bugs with few false positives (invalid tests)?

RQ2. Can aNNoTest reproduce known, relevant bugs (that were discovered and confirmed by expert manual analysis)?

RQ3. How many annotations does aNNoTest need to be effective?

RQ4. How does aNNoTest compare to other generic (non-NN specific) test-case generation techniques?

RQ5. How does aNNoTest compare to manual-written tests in terms of coverage?

4.1. Experimental subjects

To include a broad variety of real-world NN projects, we selected our experimental subjects following Islam et al.'s (2019) extensive survey of bugs and their replication package, (Islam\etal'sNNbugsdataset:) which collects hundreds of NN program bugs from Stack Overflow posts and public GitHub projects. The former are unsuitable to evaluate aNNoTest, since they usually consist of short, often incomplete, snippets of code that punctuate a natural-language text. In contrast, the GitHub projects provide useful subjects for our evaluation.

The survey (Islam et al., 2019) lists 557 bugs in 127 GitHub projects using the NN frameworks Keras, TensorFlow, PyTorch, Theano, and Caffe. With 350 bugs in 42 projects, Keras is the most popular project in this list; we target it for the bulk of our evaluation. Starting from all 42 Keras projects, we excluded: (i) 3 projects that were no longer publicly available; (ii) 7 projects with no bugs classified as "crashing" (see Section 3.5); (iii) and 5 projects that still use Python 2. While it could be modified to run with Python 2, we developed aNNoTest primarily for Python 3, which is the only supported major version of the language at the time of writing. We excluded another 4 projects whose repositories were missing some components necessary to execute them (such as data necessary to train or test the NN model, or to otherwise run the NN program). Finally, 7 projects did not include any reproducible crashing bugs (see Section 4.2 for how we determined these). This left 16 projects using Keras, which we selected for our evaluation. To demonstrate that aNNoTest is applicable also to other NN frameworks, we also selected 2 projects based on TensorFlow and 1 project based on PyTorch; these are among the largest projects using those frameworks analyzed by Islam et al. (2019). The leftmost columns of Table 3 list all selected 19 projects used in our evaluation, and their size in lines of code and number of functions. These projects (and their known bugs) are based on Islam et al.'s (2019) detailed survey of real-world NN bugs; this ensures that our subjects are representative of realistic NN programs and of the bugs that commonly affect them.

Comparison with Pynguin. To answer RQ4, we want to compare aNNoTest to Pynguin (a general-purpose test-case generator for Python programs) on generating tests for realistic NN programs. Unfortunately, all the NN projects that we use for aNNoTest's evaluation are incompatible with Python 3.8 (mainly because they require TensorFlow 1.x), whereas Pynguin only runs with Python 3.8 (or later versions). Therefore, we considered PyTorch's machine vision project Vision: (Vision(0.11.2):) an actively maintained open-source NN program that is compatible with Python 3.8 and includes type hints (used by Pynguin). Pynguin can only generate tests for 40 of Vision's 104 modules; current limitations[8] of its implementation prevent it from running correctly on the other 64 modules. For our experiments, we selected module `mnist` (https://github.com/pytorch/vision/blob/v0.11.2/torchvision/datasets/mnist.py) in package `torchvision.dataset` — one of the largest among those that Pynguin can analyze.

Comparison with manual tests. Manually writing aN annotations, and then letting aNNoTest generate tests automatically, is an alternative to the usual approach of writing unit tests manually. Thus, RQ5 compares manually-written tests to those generated by aNNoTest in terms of coverage. Unfortunately, none of the 19 projects selected by Islam et al. (2019) contains any unit tests.[9] Therefore, we resorted to project Vision again, as it contains an extensive manually-written test suite. For our experiments, we selected three Vision modules of substantial size that are tested in different ways: module `backbone_utils` is among the most thoroughly tested (the project's test suite reaches 96% branch coverage); module `image` is fairly well tested (79% branch coverage, which is an average coverage figure among the project's modules); and module `_video_opt` is scarcely tested (16% branch coverage, and is only tested indirectly by the unit tests of other client modules).

4.2. Experimental setup

This section describes how we setup each project before applying aNNoTest; and the experiments we conducted to answer the RQs.

4.2.1. Project setup

As first step, we created an Anaconda (Anaconda:) environment for each project to configure and run it independent of the others. Every project has *dependencies* that involve specific libraries. Collecting all required dependencies can be tricky: a project may work only with certain library versions, older versions of a library may no longer be available, and newer backward-compatible versions may conflict with other dependencies. A handful of projects detail the specific versions of

[8] Including bugs, one of which we reported to Pynguin's maintainers who fixed it. (https://github.com/se2p/pynguin/issues/20).

[9] Project ADV includes a single integration test; the other projects include no tests at all.

the libraries they need in a `setup.py`, `requirements.txt`, or Jupiter Notebook file — or at least in a human-readable `readme`. In many cases, none of these were available, so we had to follow a trial-and-error process: (i) search the source code for `import L` statements; (ii) retrieve the version of library `L` that was up-to-date around the time of the project's analyzed commit; (iii) in case that version is no longer available or conflicts with other libraries, try a slightly more recent or slightly older version of `L`.

NN programs usually need *datasets* to run. When a suitable dataset was not available in a project's repository, we inspected the source code and its comments to find references to public datasets that could be used, fetched them, and added them to the project's environment. In a few cases, the project included functions to generate a sample dataset, which was usually suitable to be able to at least test the project. For a few projects using very large datasets, we shrank them by removing some data points so that certain parts of the project's code ran more efficiently. Whenever we did this, we ascertained that using the modified dataset did not affect general program behavior in terms of *reachability* — which is what matters for detecting the crashing bugs that we target in our evaluation.

Properly setting up all NN programs so that they can be automatically run and tested was quite time-consuming at times, since several of the projects' repositories are incomplete, outdated, and poorly documented. Our replication package includes all required dependencies, which can help support future work in this area.

4.2.2. Experimental process

To address RQ1, we selected the latest versions of two projects among the largest and most popular ones (ADV and GANS in Table 2) and followed the guidelines described in Section 3.2 to fully annotate them with AN. "Fully annotate" means that we tried to annotate every function of the project's source code, and to write annotations that are as accurate as possible: neither unnecessarily constraining (skipping some valid inputs) nor too weak (allowing invalid inputs).

To address RQ2, we tried to use ANNOTEST to reproduce the bugs reported by Islam et al. (2019) for the selected projects. More precisely, Islam et al.'s (2019) companion dataset identifies each bug b by a triple (ℓ, b^-, b^+): line ℓ in commit b^- is the faulty statement, which is fixed by the (later) commit b^+. As we mentioned above, Islam et al.'s (2019) dataset was collected by manual analysis, and thus some of the bugs are not (no longer) reproducible, are duplicate, or are otherwise outside ANNOTEST's scope. For our evaluation, we selected only *unique reproducible crashing bugs*: (i) "crashing" means that the fault triggers a runtime program failure, which we use as oracle[10]; the crashing location c may be different from the bug location ℓ; (ii) "reproducible" means that we could manually run the program to trigger the failure; (iii) "unique" means that we merged bugs that are indistinguishable by a crashing oracle (for example, they crash at the same program point, or they fail the same assertion) or that refer to the very same triple in Islam et al.'s (2019) dataset.

Out of all 213 bugs in Islam et al. (2019) for the 19 selected projects, we identified 81 unique reproducible crashing bugs. For each such bug $b = (\ell, c, b^-, b^+)$ we annotated the project's commit b^- starting from the function (or method) `f` where location ℓ is, and continuing with the other functions that depend on `f`. We stopped annotating as soon as the annotations where sufficient to exercise function `f` (including, in particular, reaching ℓ and/or crash location c). Then, we ran ANNOTEST to generate tests for `f` and any other functions that we annotated. We count bug b as *reproduced* if some of the generated tests fails at crashing location c, and does not fail if run on the patched version b^+.

To address RQ3, we measured the annotations we wrote for RQ1's and RQ2's experiments; and we compared the size (in lines of code) of these manually-written annotations to the Hypothesis code generated automatically by ANNOTEST from the annotations.

To address RQ4, we compared ANNOTEST to Pynguin and Deal. As we discuss in Section 5, Pynguin Lukasczyk et al. (2020) is a state-of-the-art unit-test generator for Python that uses type hints to improve its effectiveness (although it also works without type hints); Deal (Anon, 2018) is a Python library for Design by Contract, supporting annotations such as preconditions, as well as test-case generation and static analysis based on them. For the comparison with Pynguin, we annotated the functions in Vision's module `mnist` (see Section 4.1) using AN similarly to what done for RQ1, writing 21 regular annotations and 1 generator for 23 functions under test; then, we compared Pynguin's generated tests to ANNOTEST's. For the comparison with Deal, we took all functions in our running examples Listing 1–Listing 5 and added preconditions in Deal's syntax that express the same input constraints as our annotations in AN's syntax; then, we compared Deal's generated tests to ANNOTEST's.

To address RQ5, we annotated the functions in Vision's modules `backbone_utils`, `image`, and `_video_opt` (see Section 4.1) using AN similarly to what done for RQ1. Since the goal is comparing to manually written tests, we ignored the tests when writing AN annotations, and only considered examples of function usages in the library implementation or comments. Using tool *Coverage.py* (Coverage.pyv.~6.5.0) we measured the branch coverage achieved on each module by: (i) the manually-written unit tests in Vision's test suite; (ii) the tests generated by ANNOTEST from the annotations. We used branch coverage but note that, on these subjects, this metric correlates very strongly (Pearson correlation coefficient: 0.94) with statement coverage; thus, using either coverage metric would lead to the same findings.

Annotation effort. As we mentioned in Section 3.2, gaining an accurate understanding of a program's behavior is necessary regardless of the approach one follows to build tests. In our experiments, we found that finding plausible ranges for a function's inputs requires only modest effort in the majority of cases. This is in accordance with the so-called *locality principle* (Denning, 2005), which implies that a significant part of a program's behavior often can be understood by observing only a small number of program inputs (Ding and Zhong, 2003). Regardless of whether one is targeting a program that is easy or hard to test, ANNOTEST can support the tester's job by providing a means of expressing the input constraints, of exercising them with automatic test generation.

4.3. Experimental results

4.3.1. RQ1: Precision

Table 2 shows the results of applying ANNOTEST to the latest commits[11] of projects ADV and GANS. With the goal of annotating the projects as thoroughly as possible, we ended up writing some AN annotations for 42% of their 249 functions. Most of the functions that we left without annotations do not need any special constraints to be tested — usually because they either are simple utility functions that are only called in specific ways by the rest of the project or have no arguments. There are a few additional cases of functions that are not used anywhere in the project and whose intended usage we could not figure out in

[10] While Islam et al. (2019) classify some bugs as "crashing", we also included bugs in other categories provided they can eventually generate a crash.

[11] The projects are however no longer maintained; therefore, we did not submit any of the found bugs to the projects' repositories.

Table 2
Two projects fully annotated with ANNoTest and the found bugs. Each row shows data about a PROJECT (identified by an acronym; see Table 3 for the URL of their GitHub repositories): its size in lines of code LOC and number of FUNCTIONS (including methods); the average (per function) number #A of annotations we added to the project, the percentage %F of functions with at least one annotation, and the percentage %G of annotations that use custom generators; and the number of unique crashing BUGS found by generating tests based on the templates — split into confirmed TRUE bugs, SPURIOUS bugs (triggered by invalid inputs), and the corresponding PRECISION = TRUE/(TRUE + SPURIOUS).

PROJECT	LOC	FUNCTIONS	ANNOTATIONS			BUGS		
			#A	%F	%G	TRUE	SPURIOUS	PRECISION
ADV	1421	100	1.58	49%	7%	33	5	87%
GANS	2496	149	1.15	37%	6%	17	1	94%
Overall	3917	249	1.33	42%	7%	50	6	89%

any other way; in these cases, we did not annotate them (and excluded them from testing). With these annotations, ANNoTest reported 56 crashes, 50 of which we confirmed as genuine unique crashing bugs; this corresponds to a precision of 89%.

As previously reported (Sun et al., 2017), bugs due to project dependency conflicts are quite common in NN programs. An interesting example is a crash that occurs in ADV when it accesses attribute `W` (https://github.com/bstriner/keras-adversarial/blob/master/examples/example_rock_paper_scissors.py#L62) in Keras's class `Dense`. (https://github.com/keras-team/keras/blob/keras-1/keras/layers/core.py#L588) This attribute was renamed to `kernel` (https://github.com/keras-team/keras/blob/keras-2/keras/layers/core.py#L823) in Keras version 2.0. Since ADV explicitly supports this major version of Keras, this crash is a true positive. Another confirmed bug we found was due to a function in ADV still using tuple parameter unpacking (https://github.com/bstriner/keras-adversarial/blob/master/examples/example_aae_cifar10.py#L69-L70) — a Python 2 feature removed in Python 3. The ADV project developers probably forgot to update this one instance consistently with how they updated the rest of the project, (https://github.com/bstriner/keras-adversarial/blob/master/examples/example_aae.py#L46-L47) which is indeed designed to work with Python 3.

A tricky example of false positive occurred in project GANS's function `create_celeba_channel_last`, (https://github.com/naykun/TF_PG_GANS/blob/master/Tensorflow-progressive_growing_of_gans/h5tool3.py#L500) which creates an HDF5 ([HDF5 (Hierarchical Data Format 5) for Python:]https://www.h5py.org/) file for the CelebA dataset (Liu et al., 2015). One of the tests generated by ANNoTest crashes (https://github.com/naykun/TF_PG_GANS/blob/master/Tensorflow-progressive_growing_of_gans/h5tool3.py#L520) as it is unable to create a file. However, the failure does not happen if we run the function manually using the very same inputs; thus, the testing environment is responsible for the spurious failure.

These experiments suggest that ANNoTest can be quite effective to pin down bugs, problems, and inconsistencies in NN programs, thus helping systematically improve their quality.

> *Applied to two fully-annotated open-source NN programs,* ANNoTest *generated tests revealing 50 bugs with 89% precision.*

4.3.2. RQ2: Recall

Table 3 shows the results of applying ANNoTest to detect 81 unique reproducible crashing bugs in 19 projects surveyed by Islam et al. (2019) and selected as explained in Section 4.1. Using the annotations we provided, ANNoTest reproduced 63 of these bugs without generating any spurious failing tests. This corresponds to a 100% precision and 78% recall relative to the unique reproducible known bugs from Islam et al. (2019). With the same annotations, ANNoTest also revealed another 31 failures that we confirmed as additional crashing bugs in the same projects.[12]

While ANNoTest was quite effective at reproducing the known bugs in these projects, it is interesting to discuss the issues that prevented it from achieving 100% recall. We identified several scenarios: (i) masking; (ii) scripting code; (iii) nested functions; (iv) lazy features; (v) and inaccessible code.

Masking occurs when an earlier crash prevents program execution from reaching the location of another bug b'. Masking is usually not a problem when the earlier crash is determined by a known bug b: in this case, we can just run tests on the project commit b^+ where b has been fixed, so that execution can reach the other bug b'. However, if a bug b' is masked by an unknown bug (column OTHER in Table 3), and we do not know how to fix the unknown bug to allow the program to continue, b' is effectively unreachable. We could not reproduce 4 known bugs because of masking. One of them occurrs (https://github.com/naykun/TF_PG_GANS/commit/efc6c3681587319c72e0e867b2b0e673aa018c17#diff-2add825310f36eb8852870389321d3e6a7416fed8f9aacd3e0b29fd0a2336b1dL196-L197) in project GANS, and is masked by an unexpected crash (https://github.com/naykun/TF_PG_GANS/commit/efc6c3681587319c72e0e867b2b0e673aa018c17#diff-2add825310f36eb8852870389321d3e6a7416fed8f9aacd3e0b29fd0a2336b1dL187) occurring in the same function `Discriminator`. In project UN, some missing statements make it impossible to distinguish three known bugs, (https://github.com/taashi-s/UNet_Keras/commit/fd81da67bfcf173331e036874250040138e76bc8f#diff-e1afe2b6eb4252b0f813153018d4e40a721ed0bac509ce0a3f75d14c046fc800R51), (https://github.com/taashi-s/UNet_Keras/commit/fd81da67bfcf173331e03687425040138e76bc8f#diff-e1afe2b6eb4252b0f813153018d4e40a721ed0bac509ce0a3f75d14c046fc800R52), (https://github.com/taashi-s/UNet_Keras/commit/fd81da67bfcf173331e03687425040138e76bc8f#diff-e1afe2b6eb4252b0f813153018d4e40a721ed0bac509ce0a3f75d14c046fc800R53) since they all crash the same test. Therefore, we consider 1 of them reproduced and 2 not reproduced due to masking. One of the tests produced for project GANS stopped before finding a known bug, (https://github.com/naykun/TF_PG_GANS/commit/efc6c3681587319c72e0e867b2b0e673aa018c17#diff-2add825310f36eb8852870389321d3e6a7416fed8f9aacd3e0b29fd0a2336b1dL35) with a `SIGKILL` (triggered by memory-related issues).

ANNoTest generates unit tests, which target specific functions in a program's source code. This excludes any code snippets in the "main" section of a Python file (under `if __name__=='__main__'`), which executes when the file is run as a script from the command line. Therefore, ANNoTest could not reproduce 6 bugs affecting this *scripting code*, such as one known bug in project CONV. (https://github.com/heuritech/convnets-keras/commit/b1b472ccf59bfc3edb7ad033299875c905b

[12] Islam et al.'s (2019) survey is not meant to be an exhaustive catalog of all bugs in these projects.

Table 3
Bugs from Islam et al. (2019) that ANNOTEST could reproduce. Each row shows data about a PROJECT (identified by an acronym and the URL of its GitHub repository): its DNN framework (Keras, TensorFlow, Torch), its size in lines of code LOC and the number of TOTAL and TESTED functions (including methods); the number of its different REVISIONS that we analyzed, the average (per tested function) number #A of annotations we added, the percentage %F of functions with at least one annotation, and the percentage %G of annotations that use custom generators; and the number of crashing BUGS found by generating tests based on the templates — the number of *reproducible* KNOWN bugs reported by Islam et al. (2019), how many of these the tests REPRODUCED, how many OTHER confirmed true bugs and SPURIOUS bugs (triggered by invalid inputs) the tests also reported in the same experiments, and the corresponding PRECISION = (REP + OTHERS)/(REP + OTHERS + SPURIOUS) and RECALL = REP/KNOWN.

	PROJECT	LOC	FUNCTIONS		REV	ANNOTATIONS			BUGS					
			TOTAL	TESTED		#A	%F	%G	KNOWN	REP	OTHER	SPURIOUS	PRECISION	RECALL
K	NAAS[a]	140	7	0	2	–	0%	0%	2	2	1	0	100%	100%
K	ADV[b]	1421	100	4	2	1.5	4%	0%	8	6	3	0	100%	75%
K	DN[c]	82	5	2	1	14.0	40%	0%	2	2	2	0	100%	100%
K	DCF[d]	748	35	1	1	4.0	3%	0%	1	0	0	0	–	0%
K	KIS[e]	2050	92	2	1	1.5	2%	0%	6	5	0	0	100%	83%
K	FRCNN[f]	1643	55	3	1	1.7	5%	0%	6	3	0	0	100%	50%
K	CONV[g]	350	20	0	1	–	0%	–	1	0	0	0	–	0%
K	mCRNN[h]	225	1	0	1	–	0%	0%	1	1	5	0	100%	100%
K	IR[i]	306	38	0	1	–	0%	–	2	0	0	0	–	0%
K	RE[j]	966	25	1	1	15.0	4%	0%	1	1	5	0	100%	100%
K	CAR[k]	353	21	1	1	7.0	5%	0%	1	1	1	0	100%	100%
K	GANS[l]	2496	149	2	1	12.5	1%	4%	6	4	5	0	100%	67%
K	KAX[m]	227	15	0	1	–	0%	–	1	0	0	0	–	0%
K	VSA[n]	630	38	2	1	6.0	5%	0%	2	2	4	0	100%	100%
K	UN[o]	440	28	3	2	3.3	11%	30%	6	2	1	0	100%	33%
K	LSTM[p]	477	27	0	1	–	0%	–	1	0	0	0	–	0%
F	TC[q]	285	7	0	2	–	0%	0%	9	9	2	0	100%	100%
F	TPS[r]	286	2	2	1	4.0	100%	87%	24	24	0	0	100%	100%
T	DAF[s]	1094	70	1	1	9.0	1%	67%	1	1	2	0	100%	100%
Overall		14219	735	24	23	6.0	3%	12%	81	63	31	0	100%	78%

[a] https://github.com/anastassia-b/neural-algorithm-artistic-style
[b] https://github.com/bstriner/keras-adversarial
[c] https://github.com/cmasch/densenet
[d] https://github.com/csvance/deep-connect-four
[e] https://github.com/dhkim0225/keras-image-segmentation
[f] https://github.com/dishen12/keras_frcnn
[g] https://github.com/heuritech/convnets-keras
[h] https://github.com/jamesmf/mnistCRNN
[i] https://github.com/javiermzll/Image-Recognition
[j] https://github.com/katyprogrammer/regularization-experiment
[k] https://github.com/michalgdak/car-recognition
[l] https://github.com/naykun/TF_PG_GANS
[m] https://github.com/notem/keras-alexnet
[n] https://github.com/Spider101/Visual-Semantic-Alignments
[o] https://github.com/taashi-s/UNet_Keras
[p] https://github.com/yagotome/lstm-ner
[q] https://github.com/dennybritz/cnn-text-classification-tf
[r] https://github.com/iwyoo/tf_ThinPlateSpline
[s] https://github.com/zzsdsgdtc/BiDAF_PyTorch

f8e37#diff-4a9f068fbd6ab76d347ca7772f3da3f100db338cd6c8fb3900adef38ab9dff20L325) Another example is the only known bug (https://github.com/notem/keras-alexnet/commit/94638c596ca6f3f474241e8a058fd893e1f5ffaa#diff-23de837fc8b40e270ddb47d0ae913f55e8d31635b80daa5618273535b9d3cd28L198) in project KAX, which occurs in a function that depends on command line arguments.

ANNOTEST can test *nested functions* only indirectly, that is when they are called by a top-level function as part of testing the latter. It does not support annotating nested functions and generating unit tests for them since they are not accessible outside their parent functions. We could not reproduce 3 known bugs because they affected nested functions. An example is in project FRCNN's function `rpn_loss_regr_fixed_num`, (https://github.com/dishen12/keras_frcnn/commit/d91c0adc5ccd34f6e346fdeddc0a2ce7085a4ffb#diff-a3429d56d560ec95c6b119754a121d183b32f8a4b73786f8760d083353914efbL18) which is defined inside top-level function `rpn_loss_regr`.

Functions using Python's `yield` statement are *lazy*, that is their evaluation is delayed. This means that they may not be executed by ANNOTEST's unit testing environment (or rather its Hypothesis back-end's). We could not reproduce 1 known bug (https://github.com/dhkim0225/keras-image-segmentation/commit/992685cde39c3d53ea881d22b9cb26e84963d4bb#diff-d0ff8417443a18c35cc6c3183197d82f48cee72d735133ff901da033d0e32242L89) in project KIS because it uses `yield` to build a lazy iterator.

As we remarked above, a bug's crashing location c may differ from the actual error location ℓ in commit b^-. If c is in a portion of the code that is *not accessible* to the testing environment, ANNOTEST cannot reproduce the bug even if it is reproducible in principle. This scenario occurred for 3 known bugs that ANNOTEST did not reproduce. Two of them are in project UN (https://github.com/taashi-s/UNet_Keras/commit/b1b6d938bdd7a3e30f3d1fa58009f4850cbc2958#diff-e1afe2b6eb4252b0f813153018d4e40a721ed0bac509ce0a3f75d14c046fc800L31), (https://github.com/taashi-s/UNet_Keras/commit/b1b6d938bdd7a3e30f3d1fa58009f4850cbc2958#diff-e1afe2b6eb4252b0f813153018d4e40a721ed0ba

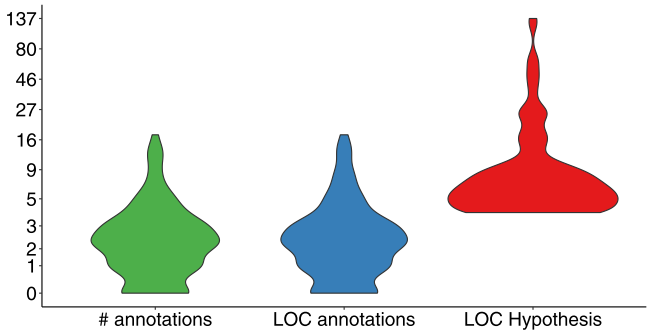

Fig. 2. Distribution of the number of AN annotations, lines of code (LOC) of AN annotations, and LOC of generated Hypothesis templates for RQ1's experiments.

c509ce0a3f75d14c046fc800L35) and only crash in a module whose implementation is incomplete in that program revision. Another one (https://github.com/javiermzll/CCN-Whale-Recognition/commit/e2d3ff925460060f0127c894368147b54b5f03c0#diff-1b740140b6c82aacc5a6f6b319be9cf103ee72b424ad475f795ea72d4b267849L46) occurs in project IR: we tried to no avail to reproduce it at a different, accessible location.

Finally, we could not reproduce 1 bug (https://github.com/javiermzll/CCN-Whale-Recognition/commit/e2d3ff925460060f0127c894368147b54b5f03c0#diff-1b740140b6c82aacc5a6f6b319be9cf103ee72b424ad475f795ea72d4b267849L46) in project IR simply because we could not figure out suitable type constraints to properly exercise the corresponding function.

> ANNoTest generated tests revealing 63 known NN bugs in 19 NN programs, with a recall of 78%.

4.3.3. RQ3: Amount of annotations

For the ANNoTest approach to be practical, it is important that it requires a reasonable amount of manual annotations. We leave to future work a detailed empirical evaluation of the time and expertise that is needed to write AN annotations. Here, we discuss quantitative measures of ANNoTest's annotation overhead. We focus on RQ1's experiments (Section 4.3.1), which analyzed projects ADV and GANS in full, as they give a better idea of the effort needed to use ANNoTest systematically on whole projects.[13]

Annotation amount. The amount of annotations that we wrote was usually limited. In RQ1's experiments, we wrote 2 annotations[14] per project function on average (median); 80% of functions have 3 annotations or less. Annotations are mostly concise: 96% of them fit a single line, and only 10% (12) of all functions have annotations that span more than 5 lines (usually decorating functions with several complex arguments). Fig. 2 pictures the distributions, overall functions, of number of annotations (left) and lines of code (LOC, middle) of annotations; since most annotations are a single line, these two distributions are nearly identical.

The average number of annotations per tested function is higher (6.0) in Table 3 since in each of those experiment we annotated a limited portion of a project focusing on a specific function that had a known bug; therefore, several of the annotations are duplicated or only slightly modified from one experiment to the other. If we had fully annotated the projects, we would have likely amortized some of this annotation effort.

In terms of time, we spent, on average, 10–15 min to write the annotations of each function. This time includes inspecting the project's source code to become familiar with how it works. As pointed out in Section 4.2, this effort is amortized over various related functions, and is unevenly distributed, with a few "complex" functions taking considerably more time to understand than most "simple" functions. As mentioned in Section 3.2, we consider the overall effort comparable to the time to manually write unit tests for the same functions.

Another way of quantifying the effort-benefit trade-off is measuring the amount of annotations per detected bugs: this ratio is $6.6 = 330/50$ for the fully-annotated projects in Table 2 and $1.5 = 145/94$ for the experiments in Table 3. These are encouraging figures, if we think of the amount of manually-written tests that may have been necessary to discover the same bugs (see also Section 4.3.5).

The percentage of annotations using generators is higher (12%) for the projects in Table 3. More precisely, the two projects in Table 2 use 15 generators, 73% (11/15) of which generate NN models. Among the 16 generators built for the projects in Table 3, 31% (5/16) generate NN models, 37% (6/16) provide TensorFlow's tensor objects, and 25% (4/16) load datasets from disk. The one remaining generator function loads an image from hard disk, turns it into a NumPy array and passes it to a function. As we explained in Section 3.3, we built all generators by applying light refactoring operations to suitable portions of existing client code within the same project.

Hypothesis overhead. Since ANNoTest translates AN annotations to Hypothesis templates, we can quantify how concise AN is compared to directly encoding constraints in Hypothesis. The rightmost plot in Fig. 2 pictures the distribution of LOC of generated Hypothesis code. Clearly, Hypothesis code is considerably more verbose than AN annotations: Hypothesis templates are 5.5 (median overhead) times longer – 11.6 times longer in terms of mean overhead – than the AN annotations they encode, which points to the benefits of using AN's concise language.

> In our experiments, ANNoTest used 2 annotations per function on average; 96% of all annotations fit a single line.

4.3.4. RQ4: Comparison to generic test-case generators

We designed ANNoTest not as a general-purpose testing tool but as one specifically geared towards NN programs. Therefore, we expect ANNoTest to outperform generic test-case generators for Python when generating tests for these programs.

As we discussed in Section 4.1, we ran **Pynguin** on module `mnist` in project Vision; the module includes type hints annotations, which Pynguin uses to improve the accuracy of its generated tests. Pynguin[15] generated 19 tests, reporting 6 tests as passing (they terminate without errors), and 13 tests as failing (they throw an exception). By manual inspection, we determined that: (i) 2 of the 6 passing tests and 10 of the 13 failing tests are actually *invalid*, since they call functions with input values that are not valid according to the functions' docstring, type hints,[16] or other available documentation; (ii) the other 3 failing tests should be classified as *passing*, since throwing an exception is the functions' expected behavior in those cases. In all, 63% $((2 + 10)/19)$ of the tests generated by Pynguin are invalid, and 79% $((3 + 2 + 10)/19)$ are misclassified. We cannot expect Pynguin to perform better, since it simply lacks the information

[13] The figures for RQ2's experiments are, however, generally similar.

[14] An annotation is any instance of the kinds presented in Section 3.1.

[15] We report experiments that used Pynguin's default configuration; however, using other generation strategies did not significantly change the outcome.

[16] Pynguin may violate type hints whose format it does not support.

Table 4
A comparison (part of) Vision's programmer-written test suite and ANNOTEST's generated tests in terms of coverage. For each MODULE, the table reports the branch COVERAGE of the programmer-written PROJECT TEST SUITE on the module (split between UNIT tests directly targeting the module's functions, and coverage achieved INDIRECTly by other modules' tests calling the module); the number of unit test FUNCTIONS directly exercising the module; and the size of these tests in lines of code LOC and number of characters CHARS; the number of functions we ANNOTATED; the COVERAGE achieved by ANNOTEST on these functions; and the size of these annotations in lines of code LOC and number of characters CHARS.

MODULE	PROJECT TEST SUITE					ANNOTEST			
	COVERAGE		#FUNCTIONS	TEST SIZE		#TESTS	COVERAGE	ANNOTATION SIZE	
	UNIT	INDIRECT		LOC	CHARS	ANNOTATED		LOC	CHARS
`_video_opt`	0%	16%	0	0	0	8	76%	25	1566
`image`	79%	0%	15	359	13381	11	84%	40	1858
`backbone_utils`	82%	14%	9	238	8478	3	82%	23	1044
Overall	50%	5%	24	597	21859	22	80%	88	4468

to precisely characterize valid inputs; in contrast, leveraging the AN annotations' information, ANNOTEST generated 11 tests for module `mnist`: all of them are valid and passing.[17]

Deal's expressive annotation language is capable of concisely encoding most of the AN annotations as preconditions (`@deal.pre`). Then, Deal's test-case generation engine draws inputs randomly and uses preconditions to filter them; therefore, the stronger a precondition is, the more it will struggle to find any valid inputs. In all our examples (Listing 1–Listing 5), Deal could not generate a single valid input that satisfies all constraints. Even after removing some of the most complex constraints (for example, the first one in Listing 2), Deal's built-in test-case generator could not generate valid inputs. Here too, we cannot expect Deal to perform better, since, unlike ANNOTEST, its test-case generation process is not built around the kinds of complex constraints that arise in NN programs.[18]

> ANNOTEST outperforms other test-case generation techniques that are not designed specifically for NN programs.

These results are another manifestation of the trade-off between specification accuracy and test effectiveness: precise tests require precise knowledge of the expected program constraints (and behavior), regardless of whether this knowledge is formalized as annotations, as executable code, or is applied directly by the programmer.

4.3.5. RQ5: Code coverage

Table 4 compares the manually-written tests in three of project Vision's modules (see Section 4.1) to those generated by ANNOTEST after annotating the functions in these modules.

Module `_video_opt` is scarcely tested in Vision: there are no unit tests for this module (column UNIT in Table 4), but tests in other modules still indirectly exercise 16% of its branches (column INDIRECT). In contrast, ANNOTEST reaches a 76% coverage after annotating 8 functions in this module. Vision's unit tests for module `image` achieve a 79% coverage; ANNOTEST reaches a higher 84% coverage. Finally, Vision's unit tests for module `image` achieve a 82% coverage, the same as ANNOTEST. The whole test suite in Vision actually further exercises module `backbone_utils`, as tests in other modules indirectly add an additional 14% of coverage. Overall, ANNOTEST-generated tests achieve a high coverage — comparable to or often higher than that of the programmer-written test suite.[19]

In order to achieve this coverage, what is the amount of code (manual tests) or annotations (ANNOTEST) that is required? Vision's unit tests for Table 4's three modules consist of 24 tests, spanning 597 lines of code or 21859 characters; ANNOTEST's annotations are only needed for 22 functions, and span 88 lines or 4468 characters. This confirms that ANNOTEST's annotations are concise — considerably more concise than unit tests achieving a lower coverage. Naturally, the sheer size of a piece of code is an imperfect measure of the effort needed to produce it; however, AN annotations encode essentially the same information as parametric tests, and their succinctness is an advantage.

100% coverage? Neither ANNOTEST nor the programmer-written test suite managed to cover 100% of the branches in the three modules. In a few cases, increasing the coverage would be possible by simply writing more unit tests or more general annotations. For instance, none of the manual tests for module `backbone_utils` instantiates class `BackboneWithFPN` by passing argument `None` to its constructor's parameter `extra_blocks`; the corresponding branch (https://github.com/pytorch/vision/blob/b4686f2b7409d1783dfbb951492cd59bfed08bce/torchvision/models/detection/backbone_utils.py#L44) is thus never covered (but it is by ANNOTEST). Conversely, ANNOTEST does not test function `_read_video_from_memory` (https://github.com/pytorch/vision/blob/b4686f2b7409d1783dfbb951492cd59bfed08bce/torchvision/io/_video_opt.py#L265) in module `_video_opt` because we could not find meaningful examples of its usage.[20] In other cases, however, achieving a 100% coverage is impractical due to constraints in the test execution environment. For instance, a branch (https://github.com/pytorch/vision/blob/b4686f2b7409d1783dfbb951492cd59bfed08bce/torchvision/io/image.py#L160) in function `decode_jpeg` of module `image` requires running the module on a machine with a GPU supporting the CUDA API. (CUDA) There is actually a manual test (https://github.com/pytorch/vision/blob/b4686f2b7409d1783dfbb951492cd59bfed08bce/test/test_image.py#L382) covering this branch, but it was not activated in our experiments since we did not run them with CUDA. We found a few other examples of this scenario (https://github.com/pytorch/vision/blob/b4686f2b7409d1783dfbb951492cd59bfed08bce/torchvision/io/image.py#L12), (https://github.com/pytorch/vision/blob/b4686f2b7409d1783dfbb951492cd59bfed08bce/torc

[17] While experimenting with testing the Vision project using ANNOTEST, we found a bug in a module that Pynguin cannot test. We submitted a fix as a pull request (https://github.com/pytorch/vision/pull/5238) that was promptly accepted. Interestingly, the affected function (https://github.com/pytorch/vision/blob/v0.11.2/torchvision/models/detection/backbone_utils.py#L49) already included a developer-written parameterized test, (https://github.com/pytorch/vision/blob/v0.11.2/test/test_backbone_utils.py#L25) which nonetheless did not detect "our" bug; this further demonstrates ANNOTEST's practical effectiveness.

[18] In addition, Deal focuses on using annotations for static analysis.

[19] While testing module `image` in these experiments, ANNOTEST detected a failure in function `decode_jpeg` (https://github.com/pytorch/vision/blob/main/torchvision/io/image.py#L127) (which is already thoroughly tested in the project's test suite). Reporting this failure (https://github.com/pytorch/vision/issues/6607) to the project maintainers prompted them to modify the function's documentation so as to more accurately reflect its intended, implemented behavior.

[20] As discussed in Section 4.2, we did not consider the manually-written tests when writing AN annotations, so that the comparison in terms of coverage is fair and meaningful.

hvision/io/_video_opt.py#L14) where increasing the test coverage requires specific hardware or system libraries.

Bug density. Users of aNNoTest write annotations to then generate unit tests automatically. In RQ1's experiments, aNNoTest generated 5649 (valid) tests overall; only 1% of them fail and expose a bug. Thus, bugs in NN programs are *rare* (Roy et al., 2018). This suggests that directly writing tests that selectively expose these bugs may be challenging even for programmers knowledgeable of the program under test. The same knowledge is sufficient to write aN annotations and generate tests from them.

> aNNoTest achieves high code coverage, comparable to that of manually-written test cases.

4.4. Threats to validity

Identifying valid test inputs, and distinguishing between spurious and authentic bugs, is crucial to ensure *construct validity* (i.e., the experimental measures are adequate). Unfortunately, a reliable and complete ground truth is not available: the documentation of NN programs is often incomplete (when it exists), so we had to manually discover the intended behavior of NN programs from examples, manual code analysis, and background knowledge. Our reference – Islam et al.'s (2019) survey – was also compiled by purely manual analysis; therefore, it does not aim at completeness, and includes bugs that are not reproducible (see Section 4.2). These limitations imply that we cannot make claims of completeness ("we found all bugs"); nevertheless, we still have a good confidence in the correctness of our results ("we found real bugs"): since we focused on bugs detected by crashing oracles, most bugs we found with aNNoTest are clear violations of the program's requirements.

Since aNNoTest uses manually-written annotations, quantifying the annotation effort is needed for *internal validity* (i.e., the experimental results are suitable to support the findings). We mostly reported simple measures (number of annotations, number of functions that require annotations, etc.) which are unambiguous. In contrast, we do not make any strong claims about the time and relative effort needed by programmers to annotate: these heavily depend on a programmer's knowledge of the NN program and of the domain; precisely assessing them would require controlled experiments and user studies, which are outside this paper's scope. However, we remark that expressing aN annotations requires a knowledge of the program under test of the same kind that is needed to write effective unit tests.

Picking experimental subjects from Islam et al.'s (2019) extensive survey of real-world NN bugs helps *external validity* (i.e., the findings generalize). As we discussed in Section 4.1, we excluded some projects for practical reasons (e.g., no longer available or incomplete) and we focused on those using the Keras NN framework. While this focus does not seem especially restrictive (the majority of projects in the survey uses Keras, and we also analyzed projects using other frameworks), applying aNNoTest to very different kinds of NN programs may require different kinds of annotations or other changes in the approach. The aN annotation language is extensible with generators (Section 3.1.2), which can further help generalizability. Furthermore, in addition to Islam et al.'s (2019) subjects, we also extensively analyzed the *latest* versions of projects ADV and GANS (Section 4.3.1), so that our evaluation did not only include projects with known bugs.

5. Related work

Automated test-case generation. Since testing is a fundamental activity to ensure software quality (Candea and Godefroid, 2019), software engineering research has devised several different techniques to *automate* the generation of test inputs (Anand et al., 2013). Randoop (Pacheco et al., 2007) (based on random testing) and EvoSuite (Fraser and Arcuri, 2011) (based on genetic algorithms (Ali et al., 2010)) are two of the most popular tools for Java implementing automated test-case generation. Techniques such as those implemented by Randoop and Evosuite usually depend on the typing information about a method's input that is provided statically in languages such as Java.

Test-case generation for Python. In contrast, programs written in dynamically typed languages like Python do not include such information, which complicates test-case generation. In fact, despite Python's popularity (Cass, 2019), the first widely available tools for automated test-case generation in Python appeared only in recent years (MacIver et al., 2019; Lukasczyk et al., 2020). Pynguin Lukasczyk et al. (2020) is based on genetic algorithms like EvoSuite, and relies on Python's type hints. Hypothesis (MacIver et al., 2019) implements property-based testing, which generates random inputs trying to satisfy some programmer-written properties. Deal (Anon, 2018) is a Python library for design by contact that provides decorations to express pre- and postconditions; based on them, it supports both static and dynamic (i.e., test-case generation) analysis. aNNoTest is also an automated test-case generator for Python, but it provides a specialized set of expressive annotations useful to precisely express the valid inputs of NN programs. Then, it defers the actual test-input generation to Hypothesis, which it uses as back-end. As we demonstrated in Section 4.3.4, directly using Pynguin, Hypothesis, or Deal to generate tests for NN programs might be possible in principle, but it would involve plenty of additional manual work to express the necessary constraints indirectly through a combination of type hints (Pynguin) and testing strategies (Hypothesis), and to program test-case generation strategies that match them (Deal).

Bugs in NN programs. Following the increasing in popularity of NN and other forms of machine learning (ML), some recent research has looked into the nature of bugs that occur in NN and ML programs to understand how they differ compared to "traditional" software. Thung et al. (2012) studied bugs and human-written patches in 3 ML projects (Apache Mahout, Lucene, and OpenNLP) and classified them according to criteria such as bug severity and fixing effort. A similar study (Sun et al., 2017) of three other ML projects (Scikit-learn, Paddle, and Caffe) revealed that compatibility bugs due to conflicts between project dependencies are quite common in these programs — as they were in the subjects we used in Section 4's experiments.

Zhang et al. (2018)'s analysis of TensorFlow-based NN projects found that modeling mistakes, incorrect shape of input tensors, and unfamiliarity of users with TensorFlow's computation model were among the most frequent origins of bugs. Once again, these findings set NN programs apart from traditional software. Recent studies by Islam et al. (2019, 2020) on 5 NN frameworks confirmed some of Zhang et al. (2018)'s findings and further found that bug fix patterns in NN programs are often different compared to traditional programs. In the same line of research, Humbatova et al. (2020)'s extensive taxonomy of bugs in deep learning systems identified several causes of bugs that are specific to NN program, including incorrect/incomplete models, wrong input data types, and training process issues.

Bugs in NN models. As we recalled in Section 1, a NN program implements in code a NN *model* that is trained on some *data*, both of which can also be plagued by mistakes. Hence, traditional software engineering approaches to test generation (Sun et al., 2019), mutation testing (Hu et al., 2019; Shen et al., 2018), fault localization (Eniser et al., 2019), and even automated program repair (Sohn et al., 2019) have been applied to NN models and training data to assess and improve their quality, robustness, and

correctness. Under this paradigm, bugs are revealed by *adversarial examples*, e.g., two slightly different inputs that appear identical to the human eye but result in widely different classification by a trained model (Sun et al., 2019). Adversarial examples correspond to failing tests; and fault localization and fixing correspond to finding (Eniser et al., 2019) and changing (Sohn et al., 2019) neuron weights in a model, This kind of research is complementary to our work on aNNoTest, which is specific to NN programs but focuses on testing and finding faults in their code implementations.

Test oracles. Testing a program comprises three main steps (Ammann and Offutt, 2017). First, selecting concrete *inputs* (arguments and pre-state); second, *executing* the program under test on those inputs; and third, checking whether the program behaved as expected while executing — in particular, whether its *output* (return values and post-state) is as expected. The present paper's contribution, as well as the related work we discussed in the rest of this section, concerns the first step: test-input generation. In contrast, addressing the third step requires an *oracle*: a mechanism to check the outcome of test execution; thus, the problem of designing such mechanisms is known as the test oracle problem (Shahamiri et al., 2009; Barr et al., 2015; Oliveira et al., 2014; Pezzè and Zhang, 2014).

Similarly as for test-input generation, a key research challenge is *automating* the generation of suitable oracles, so as to reduce the required developer effort. The simplest kind of test oracle are *implicit oracles*, such as the crashing oracles we used for our experiments with aNNoTest. More expressive automated test oracles may be derived from some kind of formal specification (Aichernig, 1999), such as assertions (Coppit and Haddox-Schatz, 2005) and contracts (Araujo et al., 2011), as well from informal or semi-formal documentation written in natural language (Schwitter, 2002; Blasi et al., 2018). In absence of specifications, a practical option is building *regression oracles* (Yoo and Harman, 2012), which check whether a new version of a program retains the same input/output behavior on the test inputs as a previous version (Xie, 2006); test-input generators – like the aforementioned Randoop, EvoSuite, and Pynguin – are usually capable of building some kind of regression oracles automatically.

6. Conclusions and future work

The paper presented the aNNoTest approach to generate inputs that test NN programs written in Python. aNNoTest relies on code annotations that precisely and succinctly describe the range of valid inputs for the functions under test. Using this information, aNNoTest can generate tests that avoid spurious failures, and thus have a good chance of exposing actual bugs. In an experimental evaluation targeting 19 open-source NN programs, aNNoTest was able to reveal 94 bugs (including 63 previously known ones) with an overhead of 6 annotations per tested function on average.

Future work. A natural continuation of the work on aNNoTest is extending aN to support more kinds of constraints. As discussed in Section 4.3.3, most of the generator functions we wrote for our experiments generate complex NN model objects such as tensors; being able to specify such objects concisely would further increase the applicability and convenience of using aNNoTest.

This paper's contributions address the test-input generation problem, which is largely independent of the test-oracle problem (see Sections 5 and 3.5). In future work, we may extend aNNoTest to add support for other kinds of oracles. Since aNNoTest is based on annotations – a form of lightweight formal specification – adding *postconditions* would be a natural way to do so. Unlike the annotations currently supported by aNNoTest, which act as constraints on the pre-state and hence require a matching generation mechanism, postconditions are evaluated on a test's post-state, and hence can simply be evaluated to determine whether the test is passing or failing. Regression oracles are another kind of oracles that are commonly supported by test generation tools such as Pynguin Lukasczyk et al. (2020); aNNoTest could add support for a similar mechanism to generate *regression tests*, whose assertion capture the post-state of the program under test, and can be re-run on future versions of the program to determine whether its expected behavior has changed. Given aNNoTest's focus, it could target regression oracles that capture NN-specific properties (Zhang et al., 2014; Ding et al., 2017; Nejadgholi and Yang, 2019).

CRediT authorship contribution statement

Mohammad Rezaalipour: Conceptualization, Methodology, Software, Validation, Investigation, Data curation, Writing – original draft, Visualization. **Carlo A. Furia:** Conceptualization, Methodology, Writing – review & editing, Supervision.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

The manuscript includes a link to my data/code.

Acknowledgments

Work partially supported by Schweizerischer Nationalfonds (SNF), Switzerland grant 200021-182060 (Hi-Fi).

References

Aichernig, B.K., 1999. Automated black-box testing with abstract VDM oracle. In: Felici, M., Kanoun, K. (Eds.), Computer Safety, Reliability and Security. pp. 250–259.

Ali, S., Briand, L.C., Hemmati, H., Panesar-Walawege, R.K., 2010. A systematic review of the application and empirical investigation of search-based test case generation. IEEE Trans. Softw. Eng. 36 (6), 742–762.

Ammann, P., Offutt, J., 2017. Introduction to Software Testing Edition 2. Cambridge University Press, New York, NY.

Anand, S., Burke, E.K., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., McMinn, P., Bertolino, A., Jenny Li, J., Zhu, H., 2013. An orchestrated survey of methodologies for automated software test case generation. J. Syst. Softw. 86 (8), 1978–2001. http://dx.doi.org/10.1016/j.jss.2013.02.061.

Anon, 2018. Deal: A python library for design by contract. https://github.com/life4/deal.

Araujo, W., Briand, L.C., Labiche, Y., 2011. On the effectiveness of contracts as test oracles in the detection and diagnosis of race conditions and deadlocks in concurrent object-oriented software. In: International Symposium on Empirical Software Engineering and Measurement. pp. 10–19. http://dx.doi.org/10.1109/ESEM.2011.9.

Barr, E.T., Harman, M., McMinn, P., Shahbaz, M., Yoo, S., 2015. The oracle problem in software testing: A survey. IEEE Trans. Softw. Eng. 41 (5), 507–525.

Blasi, A., Goffi, A., Kuznetsov, K., Gorla, A., Ernst, M.D., Pezzè, M., Castellanos, S.D., 2018. Translating code comments to procedure specifications. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. Association for Computing Machinery, New York, NY, USA, pp. 242–253. http://dx.doi.org/10.1145/3213846.3213872.

Candea, G., Godefroid, P., 2019. Automated software test generation: Some challenges, solutions, and recent advances. In: Steffen, B., Woeginger, G.J. (Eds.), Computing and Software Science - State of the Art and Perspectives. In: Lecture Notes in Computer Science, vol. 10000, Springer, pp. 505–531. http://dx.doi.org/10.1007/978-3-319-91908-9_24.

Cass, S., 2019. The top programming languages 2019. https://spectrum.ieee.org/the-top-programming-languages-2019.

Claessen, K., Hughes, J., 2000. QuickCheck: A lightweight tool for random testing of haskell programs. In: Odersky, M., Wadler, P. (Eds.), Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming. ICFP '00, Montreal, Canada, September 18-21, 2000, ACM, pp. 268–279. http://dx.doi.org/10.1145/351240.351266.

Coppit, D., Haddox-Schatz, J., 2005. On the use of specification-based assertions as test oracles. In: 29th Annual IEEE/NASA Software Engineering Workshop. pp. 305–314. http://dx.doi.org/10.1109/SEW.2005.33.

Denning, P.J., 2005. The locality principle. Commun. ACM 48 (7), 19–24. http://dx.doi.org/10.1145/1070838.1070856.

Ding, J., Kang, X., Hu, X.-H., 2017. Validating a deep learning framework by metamorphic testing. In: IEEE/ACM 2nd International Workshop on Metamorphic Testing. MET, pp. 28–34. http://dx.doi.org/10.1109/MET.2017.2.

Ding, C., Zhong, Y., 2003. Predicting whole-program locality through reuse distance analysis. In: Cytron, R., Gupta, R. (Eds.), Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation 2003. San Diego, California, USA, June 9-11, 2003, ACM, pp. 245–257. http://dx.doi.org/10.1145/781131.781159.

Eniser, H.F., Gerasimou, S., Sen, A., 2019. DeepFault: Fault localization for deep neural networks. In: Hähnle, R., van der Aalst, W. (Eds.), Fundamental Approaches to Software Engineering. Springer International Publishing, Cham, pp. 171–191.

Fraser, G., Arcuri, A., 2011. EvoSuite: Automatic test suite generation for object-oriented software. In: SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference. ESEC-13, Szeged, Hungary, September 5-9, 2011, ACM, pp. 416–419. http://dx.doi.org/10.1145/2025113.2025179.

Fraser, G., Arcuri, A., 2015. 1600 Faults in 100 projects: Automatically finding faults while achieving high coverage with EvoSuite. Empir. Softw. Eng. 20 (3), 611–639. http://dx.doi.org/10.1007/s10664-013-9288-2.

Galeotti, J.P., Furia, C.A., May, E., Fraser, G., Zeller, A., 2015. Inferring loop invariants by mutation, dynamic analysis, and static checking. IEEE Trans. Softw. Eng. 41 (10), 1019–1037.

Goodfellow, I.J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., Bengio, Y., 2014. Generative adversarial nets. In: Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2. pp. 2672–2680.

Hu, Q., Ma, L., Xie, X., Yu, B., Liu, Y., Zhao, J., 2019. DeepMutation++: A mutation testing framework for deep learning systems. In: 34th IEEE/ACM International Conference on Automated Software Engineering. ASE, pp. 1158–1161. http://dx.doi.org/10.1109/ASE.2019.00126.

Huang, G., Liu, Z., Pleiss, G., Van Der Maaten, L., Weinberger, K., 2019. Convolutional networks with dense connectivity. IEEE Trans. Pattern Anal. Mach. Intell..

Humbatova, N., Jahangirova, G., Bavota, G., Riccio, V., Stocco, A., Tonella, P., 2020. Taxonomy of real faults in deep learning systems. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 1110–1121. http://dx.doi.org/10.1145/3377811.3380395.

Islam, M.J., Nguyen, G., Pan, R., Rajan, H., 2019. A comprehensive study on deep learning bug characteristics. In: Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. pp. 510–520. http://dx.doi.org/10.1145/3338906.3338955.

Islam, M.J., Pan, R., Nguyen, G., Rajan, H., 2020. Repairing deep neural networks: Fix patterns and challenges. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering. pp. 1135–1146. http://dx.doi.org/10.1145/3377811.3380378.

Liu, Z., Luo, P., Wang, X., Tang, X., 2015. Deep learning face attributes in the wild. In: Proceedings of International Conference on Computer Vision. ICCV.

Lukasczyk, S., Kroiß, F., Fraser, G., 2020. Automated unit test generation for python. In: Proceedings of the 12th Symposium on Search-Based Software Engineering. pp. 9–24.

MacIver, D., Hatfield-Dodds, Z., Contributors, M., 2019. Hypothesis: A new approach to property-based testing. J. Open Source Softw. 4 (43), 1891. http://dx.doi.org/10.21105/joss.01891.

Nejadgholi, M., Yang, J., 2019. A study of oracle approximations in testing deep learning libraries. In: 34th IEEE/ACM International Conference on Automated Software Engineering. ASE, pp. 785–796. http://dx.doi.org/10.1109/ASE.2019.00078.

Oliveira, R.A., Kanewala, U., Nardi, P.A., 2014. Chapter three - automated test oracles: State of the art, taxonomies, and trends. In: Memon, A. (Ed.), In: Advances in Computers, vol. 95, Elsevier, pp. 113–199. http://dx.doi.org/10.1016/B978-0-12-800160-8.00003-6.

Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T., 2007. Feedback-directed random test generation. In: 29th International Conference on Software Engineering. pp. 75–84.

Pezzè, M., Zhang, C., 2014. Chapter one - automated test oracles: A survey. In: Memon, A. (Ed.), In: Advances in Computers, vol. 95, Elsevier, pp. 1–48. http://dx.doi.org/10.1016/B978-0-12-800160-8.00001-2.

Riccio, V., Jahangirova, G., Stocco, A., Humbatova, N., Weiss, M., Tonella, P., 2020. Testing machine learning based systems: A systematic mapping. Empir. Softw. Eng. 25 (6), 5193–5254. http://dx.doi.org/10.1007/s10664-020-09881-0.

Roy, S., Pandey, A., Dolan-Gavitt, B., Hu, Y., 2018. Bug synthesis: Challenging bug-finding tools with deep faults. In: Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018, ACM, pp. 224–234. http://dx.doi.org/10.1145/3236024.3236084.

Schwitter, R., 2002. English as a formal specification language. In: Proceedings. 13th International Workshop on Database and Expert Systems Applications. pp. 228–232. http://dx.doi.org/10.1109/DEXA.2002.1045903.

Shahamiri, S.R., Kadir, W.M.N.W., Mohd-Hashim, S.Z., 2009. A comparative study on automated software test oracle methods. In: Fourth International Conference on Software Engineering Advances. pp. 140–145. http://dx.doi.org/10.1109/ICSEA.2009.29.

Shen, W., Wan, J., Chen, Z., 2018. MuNN: Mutation analysis of neural networks. In: 2018 IEEE International Conference on Software Quality, Reliability and Security Companion. QRS-C, pp. 108–115. http://dx.doi.org/10.1109/QRS-C.2018.00032.

Sohn, J., Kang, S., Yoo, S., 2019. Search based repair of deep neural networks. arXiv:1912.12463v1.

Sun, Y., Huang, X., Kroening, D., Sharp, J., Hill, M., Ashmore, R., 2019. Structural test coverage criteria for deep neural networks. ACM Trans. Embed. Comput. Syst. 18 (5s), http://dx.doi.org/10.1145/3358233.

Sun, X., Zhou, T., Li, G., Hu, J., Yang, H., Li, B., 2017. An empirical study on real bugs for machine learning programs. In: 24th Asia-Pacific Software Engineering Conference. pp. 348–357.

Thung, F., Wang, S., Lo, D., Jiang, L., 2012. An empirical study of bugs in machine learning systems. In: IEEE 23rd International Symposium on Software Reliability Engineering. pp. 271–280.

Xie, T., 2006. Augmenting automatically generated unit-test suites with regression oracle checking. In: Thomas, D. (Ed.), ECOOP 2006 – Object-Oriented Programming. pp. 380–403.

Yoo, S., Harman, M., 2012. Regression testing minimization, selection and prioritization: A survey. Softw. Test. Verif. Reliab. 22 (2), 67–120.

Zhang, Y., Chen, Y., Cheung, S.-C., Xiong, Y., Zhang, L., 2018. An empirical study on TensorFlow program bugs. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis. pp. 129–140. http://dx.doi.org/10.1145/3213846.3213866.

Zhang, J., Chen, J., Hao, D., Xiong, Y., Xie, B., Zhang, L., Mei, H., 2014. Search-based inference of polynomial metamorphic relations. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. pp. 701–712. http://dx.doi.org/10.1145/2642937.2642994.

Zhang, J.M., Harman, M., Ma, L., Liu, Y., 2022. Machine learning testing: Survey, landscapes and horizons. IEEE Trans. Softw. Eng. 48 (1), 1–36. http://dx.doi.org/10.1109/TSE.2019.2962027.

Mohammad Rezaalipour is a Ph.D. student in the Software Institute, part of the Faculty of Informatics of the Università della Svizzera Italiana (USI).

Carlo A. Furia is an associate professor in the Software Institute, part of the Faculty of Informatics of the Università della Svizzera Italiana (USI).