Shifting Left for Early Detection of Machine-Learning Bugs

Ben Liblit[1], Linghui Luo[2], Alejandro Molina[3], Rajdeep Mukherjee[1], Zachary Patterson[4], Goran Piskachev[2], Martin Schäf[1], Omer Tripp[1], and Willem Visser[1]

[1] Amazon Web Services, USA
[2] Amazon Web Services, Germany
[3] Amazon, USA
[4] The University of Texas at Dallas, USA

Abstract. Computational notebooks are widely used for machine learning (ML). However, notebooks raise new correctness concerns beyond those found in traditional programming environments. ML library APIs are easy to misuse, and the notebook execution model raises entirely new problems concerning reproducibility. It is common to use static analyses to detect bugs and enforce best practices in software applications. However, when configured with new types of rules tailored to notebooks, these analyses can also detect notebook-specific problems.

We present our initial efforts in understanding how static analysis for notebooks differs from analysis of traditional application software. We created six new rules for the CodeGuru Reviewer based on discussions with ML practitioners. We ran the tool on close to 10,000 experimentation notebooks, resulting in an average of approximately 1 finding per 7 notebooks. Approximately 60% of the findings that we reviewed are real notebook defects.[5]

Keywords: Static analysis · Computational notebooks · Jupyter Notebook · Machine-learning bugs · Bug finding · Machine learning · PyTorch · CodeGuru Reviewer

1 Introduction

Static program analysis is *shifting left*: providing recommendations as early as possible in the software development life cycle. The earlier an issue is reported, the easier and less costly it is to fix. Many off-the-shelf analysis engines now integrate seamlessly into code reviews or builds, to good effect [3, 7].

Shifting left assumes a multi-stage process that culminates in deployed software. However, work in machine learning (ML) may not fit this model. Data scientists and ML experts often use computational notebooks for development, such as Jupyter notebooks [15]. Notebooks are iterative and interactive. A typical developer edits and evaluates a notebook locally, until it produces an acceptable model, and only then sends the notebook or model to the next stage of the development pipeline.

[5] Due to confidentiality limitations, we cannot disclose the exact number of notebook files and findings.

In traditional enterprise software development, code is developed in small increments, unit tested, and passed through code review, before running on real data. For a given programming language, notebook developers invest more time into notebooks between published revisions than traditional developers [11, 35]. Delayed feedback by human colleagues means that notebook developers stand to benefit even more from automated static analyses. However, false-positive rates must be low so as to not distract developers.

Notebooks also differ from enterprise software in that notebooks do not usually run in production. Thus, many of the issues typically covered by static analysis, such as security or resilience to untrusted inputs, are not interesting to notebook developers. Instead, reproducibility is a much bigger problem [36]. Notebooks have certain features, like out-of-order execution, which can harm reproducibility, and misuse of ML APIs, which can lead to accidental modifications of trained models. Accidental model modifications are a particular concern: such mistakes are difficult to notice, and may be cumbersome or impossible to revert.

This paper presents our initial efforts in understanding how static analysis for notebooks differs from analysis of traditional application software. To understand the problem space, we conducted a pilot study by interviewing a group of ML practitioners at Amazon. Based on discussions with them, we prioritized certain issues and implemented six rules using the Python static analysis engine in CodeGuru Reviewer [21]. We report on rule efficacy for a set of notebooks shared by Amazon developers. These six rules produced an average of 1 finding per 7 notebooks on a total of nearly 10,000 notebook files. We sampled a set of the findings to assess precision. Around 60% of these findings are true positives: real notebook bugs. Our results motivate future research on how to best integrate static analysis into the development workflow for computational notebooks, and what type of rules provide the best value for notebook developers.

2 Background

To understand what types of issues are worth catching in notebooks, we interviewed a group of five ML practitioners who come from different organizations in Amazon, and occupy different roles. As we already hit saturation [12] after the fifth interview, we did not interview more people. We asked about their habits when using notebooks and issues they often encounter. Many practitioners mentioned the challenge of reproducing results of notebooks when moving between different environments. Difficulties include failure to understand the execution order of notebook cells, non-determinism of some ML APIs, and losing track of dependencies. Practitioners told us that notebooks are far from intuitive, as cells can be executed in arbitrary order. Errors often occur across cells. Previous cells are often edited or even removed after execution. These changes may break the intended functionality of following cells. Because data exploration usually takes a long time, users often do not execute all cells after small changes. Thus, breaking changes might be unnoticed until another person tries to rerun the notebook. Section 4.1 of this paper discusses two concrete issues that fall into this category and introduces our approach to detect them with static analysis.

Misusing ML APIs can introduce other silent faults. Popular deep-learning libraries such as PyTorch [22], Keras [5], and TensorFlow [1] greatly simplify the development

of deep learning systems. However, due to high conceptual complexity of the field, unclear documentation, and unintuitive APIs, users commonly misuse these libraries and inadvertently inject faults during the development of deep-learning systems. Furthermore, ML libraries are moving targets: different versions may require to different method calls, produce different performances, or exhibit different functionality altogether.

To understand which misuses are prevalent across multiple versions of APIs and that are useful to catch, we collected a list of known misuses from both scientific literature [13, 23, 34, 37] and an internal survey. We asked several ML scientists (different than the five we interviewed) at our company to rate usefulness in this list and elaborate the reasons for their ratings. We prioritized certain issues from this list based on practitioner interest (how many votes for useful) and technical feasibility. These issues are silent at build and run time, of which a developer would not be aware, even after the code is deployed. Section 4.2 introduce four rules designed for catching these issues.

3 Static Analysis Framework

Our analyses are built on the framework we developed for CodeGuru Reviewer [21]. In this section, we briefly introduce this framework.

3.1 Code Representation

Our analysis represents each program as a collection of per-function graphs called *MU graphs*. A MU graph contains five kinds of nodes:

- Entry nodes represent the start of a function's execution: one per MU graph.
- Exit nodes represent the end of a function's execution: one per MU graph.
- Control nodes represent branched control flow, such as a conditional statement or loop.
- Action nodes represent individual execution steps, such as multiplying two values or calling a function.
- Data nodes represent local variables or synthetic temporary values within compound expressions.

There are also several types of edges in MU graphs, denoted by their label:

- Control edges order execution among entry, exit, control, and action nodes. No data node is ever the source or target of a control edge. Thus, discarding all data nodes and non-control edges would reduce a MU graph to a traditional control-flow graph (CFG).
- Data edges represent movement of data among control and action nodes, and are further categorized as follows:
  - Condition edges flow from a data node into a control node, representing the information used to decide how execution continues.
  - Definition edges flow from an action to a data node defined by that action.
  - Parameter edges flow from a data node into an action node.
  - Receiver edges flow from a data node into a method-calling action node. These highlight the special role of implicit self or this arguments.
  - Callee edges flow from a data node into a call action node, identifying the function to be called.

```
CustomRule rule = new CustomRule.Builder().withName("MathExp")
  .withComment("For small floats `x`, the subtraction in "
      + "`exp(x) - 1` can result in a loss of precision.")
  .withAllOf(
    b -> b.withMethodCallFilter(".*math\\.exp")
          .withDefinitionTransform()
          .as("MathExpResult"),
    b -> b.withConstantDataFilter("1").as("ConstantOne"))
  .check()
  .withActionFilter("-")
  .withDirectDataFromIdFilter("MathExpResult")
  .withDirectDataFromIdFilter("ConstantOne")
  .build();
```

Fig. 1. GQL rule for identifying suboptimal use of the math.exp function.

3.2 Query Language

Directly analyzing MU graphs can be cumbersome, and can miss important reuse opportunities. We therefore created an API, dubbed the Guru Query Language (GQL), to enable encapsulation, optimization, and reuse of a wide variety of analysis constructs. GQL is implemented as a Java library whose main interface with the analysis builder is the CustomRule class. CustomRule instances are created using the fluent builder pattern [9], where builder calls correspond to reasoning steps in the rule. A rule object can be evaluated at different scopes, from entire code bases to single functions. This is an important source of flexibility, enabled by MU graphs and their support for partial programs. Rule evaluation yields a RuleEvaluationResult for each function or method. If rule evaluation fails, the RuleEvaluationResult includes rich diagnostic information to support rule debugging.

To illustrate GQL syntax, Figure 1 shows a rule that identifies suboptimal use of the math.exp function. Here is an example of what the rule checks for:

```
def foo():
  import math
  return math.exp(1e-10) - 1
```

Rule definition begins by setting the rule's name and user-facing comment text. The following steps, up to the check statement, are preconditions that the rule checks for. Specifically, the withAllOf statement ensures that all the subrules nested within it evaluate successfully, where these check for math.exp calls as well as the presence of the constant value 1. The matches are stored into variables (or IDs), to enable downstream reuse thereof, using the as operation. The actual check, or postcondition, is the rule section after the check step. This rule's postcondition establishes whether there is a subtraction operation that the node defined by math.exp, along with the constant 1, flow into directly (that is, without the mediation of any other action).

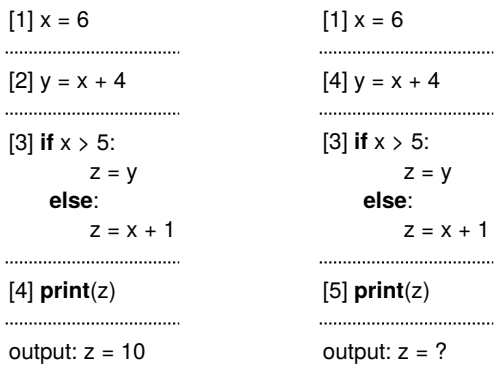

Fig. 2. Different execution orders result in different outputs.

4 Analysis Rules

In this section, we describe six analysis rules that we implemented using GQL.

4.1 Issues Specific to Computational Notebooks

Computational notebooks break some assumptions we may make when analyzing traditional code. We introduce two kinds of notebook-specific issues and the rules that we designed for catching them.

Invalid Execution Order: A notebook consists of a sequence of cells; most cells contain either Markdown documentation or code. Users can run individual cells as they wish. Thus, there is no guarantee that code cells in a notebook run in linear order, or even that linear order is intended. Cells with shared variables can produce different results when running in different order, as shown in Figure 2. Cell boundaries are marked with dotted lines. At the beginning of each cell, a number in square brackets [ ] shows the execution-order counter. These counters, stored in the metadata of a notebook file, indicate the execution order of the cells. On the left side of Figure 2, the cells were executed in linear order, causing the final value of z to be 10. On the right side of Figure 2, the execution order of the second and third cells are flipped. Furthermore, we do not know which code cell executed second in the right-side notebook, since no cell is marked "[2]". Perhaps the second-executed cell has already been deleted, or perhaps some other cell was executed second, then re-executed (and therefore renumbered) later. This uncertainty causes the final value of z to be under-determined. When y is assigned to z in the third cell, we can

```
__CELL_EDGE__(1)
x = 6
__CELL_EDGE__(3)
if x > 5:
    z = y
else:
    z = x + 1
__CELL_EDGE__(4)
y = x + 4
__CELL_EDGE__(5)
print(z)
```

Fig. 3. Converted Python code in execution order.

not assume that the definition of y is still x + 4. The recorded output for the right-side notebook would be hard or impossible to reproduce.

To address such threats to reproducibility, we designed a rule that detects cases where a used variable is not defined based on execution order. We leverage the execution counter metadata stored in notebook files to reconstruct cell code to be executed in the stored order. Specifically, we implemented a converter that converts notebook files into Python scripts that retain the execution metadata. For the example above, the converted Python script is presented in Figure 3. The __CELL_EDGE__ function is defined to do nothing, but represents a notebook cell edge. Our rule analyzes the converted Python scripts, which contain cell code in the execution order as Figure 3 shows. It starts from each variable use and searches backwards to determine whether that variable has been defined previously.

Variable Redefinition: Poor readability is another common issue in computational notebooks. During exploration, notebooks can easily get messy and difficult to read. One bad coding practice is to reuse the same variable name across multiple cells for different tasks. It is common for users to unknowingly overwrite data that is used across multiple cells. To address issues raised due to variables with unclear scopes, we designed a rule to detect a variable being defined with different types (variables whose type is unknown are excluded) in different cells, accompanied by usage in another cell that does not contain another definition. This rule analyzes our Python representation of notebooks by looking for calls to __CELL_EDGE__. From there it identifies the type of each variable in the cell and stores this information. If a variable is used in a cell that does not define the variable, but the variable is defined in at least two other cells with different types, that usage is marked as unclear. Figure 4 shows an example of this bad practice. The variable x is defined with two different types: **str** and **int**. For its usage in the third cell, it is not clear which type of data is expected to be passed to the call do_something. That depends on the execution order, where x can be either type in the third cell.

```
[?] x = "Hello World"
.....
[?] x = int(input('input:'))
.....
[?] do_something(x)
```

Fig. 4. Usage of variable with unclear scope.

4.2 Misuses of Deep Learning Libraries

As mentioned in Section 2, many issues are introduced by misusing the APIs of deep learning libraries. We introduce four representative misuses in PyTorch.

Missing zero_grad Call: Training of deep neural networks is based on iterative parameter updates [4] based on gradients that are computed via back-propagation [17]. These gradients are accumulated based on batches or mini-batches of stochastic samples of the training data-set [30]. In PyTorch, the gradients accumulate automatically in the back-propagation step of loss.backward, and developers must reset the gradient accumulation by calling zero_grad beforehand as shown in Figure 5. However, if the zero_grad step is omitted, then PyTorch would accumulate gradients indefinitely instead of updating them in batches. This default accumulating behavior is convenient as it simplifies the

```
# Case 1
model.load_state_dict(torch.load("model.pth"))
predicted = model.evaluate_on(test_data)

# Case 2
for batch_num in enumerate(dataloader):
    model.train()
    # forward, backwards and optimization steps
    if batch_num % 50 == 0:
        precision, recall, f1 = model.evaluate_on(data, batch_size)

# Case 3
model.train()
for ... # training loop
precision, recall, f1 = model.evaluate_on(data, batch_size)
```

Fig. 6. Three cases where eval should be called.

implementation of different batching approaches, but it is also easily forgotten. The impact of this type of error strongly depends on the task at hand, e.g., training a network from scratch would fail silently as the network would not learn properly and the developer would simply notice that the model is not improving, costing time and computational resources. A more severe case occurs when the task is refinement, i.e., optimizing a previously trained model on new data. In this case the first few iterations might achieve small improvements, but the network would simply not learn correctly. However, as the network was already trained, it could still perform well enough to potentially confuse the developer into thinking that things are in order, leading to invalid scientific results. Therefore, we designed a rule to detect missing zero_grad calls in training loops that invoke backward. The rule warns the users about the default accumulating behavior.

```
for train, test in loader:
    loss = metric(model(train), test)
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Fig. 5. Call zero_grad before backward.

Missing eval Call: During the optimization step of deep neural networks, developers often evaluate the predictive performance of the model on both training and test data. However, some layers in a neural network may behave differently depending on whether the network is trained or evaluated. A Dropout [31] layer disables different neurons during training to help the network learn better, but at evaluation time, the complete network is used to make predictions. Similarly, BatchNorm [14] changes internal parameters while training, but keeps parameters fixed during evaluation. To control this behavior, PyTorch mandates explicit train and eval calls to denote the start of the training and evaluation (also known as validation or testing) phases of a model, respectively. Using these calls incorrectly can lead to silent failures. Consider a version of the code where the developer forgets to call eval. In this case, the Dropout layers will indirectly change

```
x, y = torch.rand(4), torch.rand(4)
x.add_(y)
do_something(x)
```

```
x, y = torch.rand(4), torch.rand(4)
z = x.add_(y)
do_something(z)
```

Fig. 7. Left: compliant case. Right: non-compliant case.

the architecture of the network by activating and disabling different neurons. This would make all predictions unstable, i.e., for the same input data, the network would make different predictions when evaluated at different points in time. The BatchNorm layer would cause even more harm, as the parameters of the layer would adapt according to test data, leaking information from the test set into the model. This could mislead developers and scientists into thinking that the model behaves better than it actually does. We designed a rule to check whether eval is called (1) before testing a trained model loaded from disk, (2) before validating a model during the training phase, and (3) before testing a model directly after the training phase. Figure 6 gives examples of these common cases. Our rule searches both intra- and inter-procedurally, as calls to eval might be present inside the user-defined evaluate_on.

Use of Nondeterministic Algorithm: Reproducibility is a cornerstone of research in ML. Therefore, deterministic results are important to understand the impact of different configurations during the training and evaluation of neural networks [19]. Unfortunately, training and inference can be computationally expensive [28] and determinism is often abandoned in favor of approximate but faster results. The default configuration of PyTorch focuses on performance instead of determinism and provides some operations without deterministic implementations. Nevertheless, the official PyTorch documentation recommends limiting sources of nondeterministic behavior, and offers tips and APIs to control and warn about uses of non-reproducible code. To raise awareness among practitioners, we implemented a rule to check whether the non-deterministic version of an API is used instead of a deterministic alternative.

Unintended In-place Operation: The practical size of a neural network is limited by the available memory that stores the parameters and intermediate computation steps. To reduce memory consumption, PyTorch supports in-place operations over tensors, letting developers decide when to write results to existing memory instead of requiring extra space. However, as In-place operations change the content of a given torch.Tensor directly, they can cause loss of data if the operation is not intended. Figure 7 shows an example where in both variants, x.add_(y) will change the value of x in-place. In the right-side case, the return value of x.add_(y) is also explicitly assigned to a new variable z, making z a redundant alias for x. This is likely a mistake: x was probably not intended to be modified. Our rule catches torch.Tensor in-place operations that are then assigned to variables.

5 Experimental Evaluation

We evaluated our rules on several hundred code repositories containing a total of almost 10,000 experimentation notebook files (.ipynb) using PyTorch. The repositories

Table 1. Experimental Results

		Count of Findings				
Rule	Reported	Reviewed	TPs	FPs	Mixed	Precision
Invalid Execution Order	81.3%	20	11	8	1	58%
Variable Redefinition	14.2%	20	6	13	1	33%
Missing `eval` *Call*	3.0%	26	18	5	3	75%
Use of Nondeterministic Algorithm	0.9%	8	7	1	0	88%
Missing `zero_grad` *Call*	0.5%	4	1	3	0	25%
Unintended In-place Operation	0.1%	1	1	0	0	100%

were selected at random, without any bias, and cover a variety of ML application domains, including for example object recognition in images and videos, natural language processing, concept learning, healthcare, and speech recognition. We applied our notebook converter to these notebook files and analyzed the converted Python scripts. Since not all notebook files have metadata with the execution counter, our notebook converter supports two representations. One representation encodes the execution order as previously shown in Figure 3, while the other simply lists all cells in linear order, i.e., the argument passed to each `__CELL_EDGE__` call is simply the order of cell appearance in the notebook file. The linear representation is sufficient for all rules except *Invalid Execution Order*.

Table 1 shows the results of our experiment. We drew a random sample out of the overall findings pool to assess their correctness. The sample was drawn globally, and is thus uneven across the different rules yet roughly correlated with their frequency. The sampled findings were reviewed together with ML scientists. We use three ratings: "true positive" (TP) for findings judged to be real defects; "false positive" (FP) for findings judged to be harmless or correct code; and "mixed" for findings judged to be partially true. We compute precision as:

$$\text{Precision} = \frac{\text{TPs} + \text{Mixed}/2}{\text{TPs} + \text{FPs} + \text{Mixed}}$$

Invalid Execution Order produced over 80% of the overall findings, followed by *Variable Redefinition* with 14.2%. For both of these rules, we reviewed 20 of their findings. The *Invalid Execution Order* finding rated as "mixed" is due to a call of the form `foo(a, b, c)`, where all three arguments were stated to use undefined variables but in practice only some were undefined.

Unintended In-place Operation and *Use of Nondeterministic Algorithm* produced few findings, but achieved high precision of 100% and 88%, respectively. The only false positive is due to incomplete type information inferred by our Python front-end, Pyright [20]. This limitation also caused 2 of the 3 false positives for *Missing* `zero_grad` *Call*, as our rule uses type information to filter out training code using Apache MXNet [2]. MXNet automatically zeroes out gradients for users by default, so missing `zero_grad` is usually not a problem there. Another false positive for this rule is due to a third-party library API that calls `zero_grad`, but that was not available for analysis.

Missing `eval` *Call* achieves 78% precision. This rule produced 3% of the findings with 5 false positives out of 26 findings that were reviewed. We rated 3 as mixed due to

incomplete code, i.e., the eval call is missing but other functions are invoked, not visible to the analysis, that may perform this call. The most common finding pattern due to this rule is case 1 from our example in Figure 6: a trained model loaded from disk is directly applied to data without toggling the evaluation mode.

For *Invalid Execution Order*, all false positives are due to defective extraction of Python code from notebooks. Our prototype notebook converter sometimes fails to identify shell commands in notebook files, resulting in invalid lines of code in the converted Python script. Apart from this technical issue, the precision of this rule is actually quite high. We only have one finding where multiple variables at the same line are deemed undefined, one of which being a false positive. We tally this finding as mixed in Table 1.

Variable Redefinition suffers from a high rate of false positives, mostly because of special types in Python. One example is the Any type [25]. Pyright infers the return type of some library methods as Any, which is compatible with every other type. Our rule does not consider this case. Thus, if a variable is typed as Any in one notebook cell but has a concrete type in another cell, *Variable Redefinition* raises a warning. Union [24] is another special type. A variable with type Union[X, Y] can hold values of types X or Y. Consider x in the example code in Figure 8. For cell [1], Pyright infers that x has type **str**. However, for cell [2] Pyright infers that x has type Union[**int**, **str**]. Our rule considers **str** and Union[**int**, **str**] to be distinct types, thus raising a warning. However, in our review we rated such findings as false positives, as these mixtures of types appeared to be intentional in context. Lastly, we note the special Unbound type that Pyright infers for a variable that has never been initialized. We did not treat Unbound in any special way, which in turn caused some false positives. Future refinement of *Variable Redefinition* will add custom handling for these special types.

```
[1] x = "abc"
.....................................
[2] if flag:
        x = 1
    else:
        x = "def"
.....................................
[3] print(x)
```

Fig. 8. Code leading to Union type.

6 Related Work

In this section, we discuss the most relevant related work to our work.

Challenges in ML code. Many studies have discussed challenges in ML code [6, 13, 23, 26, 33]. A large-scale study [6] shows a rapid evolution of the use of ML libraries among GitHub projects. In this study, PyTorch is one of the most used libraries, which motivated us to focus on it here. Humbatova et al. [13] proposed a hierarchical taxonomy of faults in deep neural networks (DNN). Their list of faults is one of our sources for developing analysis rules. Our four rules for API misuses can be categorized into four of the five categories they identified: Model, Tenors, Training and API. Pimentel et al. [23] analyzed 1.4 million notebooks with reproducibility issues, e.g., most notebooks do not use any testing infrastructure and many notebooks have non-executed code cells, out-of-order cells, and skips in the execution count which is a challenge for reproducibility. The authors could execute only 24% of the notebooks and only 4% of them could reproduce

the expected results. Quaranta [26] identified this same problem. Quaranta also explored how notebooks are used among different users and found out that the notebooks are used in unstructured ways. These identified issues motivated us to develop rules targeting reproducibility of notebooks (e.g., *Invalid Execution Order*, *Use of Nondeterministic Algorithm*) and best practices (e.g., *Variable Redefinition*).

Static analyses for ML code. Some static analyses specifically target ML code [8, 10, 16, 38, 38]. Many of these deal with tensor shape in TensorFlow programs [8, 16, 18, 37]. Dolby et al. [8] introduced Ariadne as part of the WALA framework [29] to support static analysis of Python. As these analyses were targeting old versions of TensorFlow, they do not exist in code using more recent TensorFlow releases. Our early study on versions of ML libraries also shows these problematic TensorFlow versions are rarely used nowadays, whereas the misuses our rules address are prevalent across a wide range of PyTorch versions including the latest releases.

Another line of static analysis work focuses on providing best practices for ML practitioners. Wan et al. [34] studied 360 GitHub projects that use AWS AI or Google Cloud AI and identifier different types of API misuses generalized into eight anti-patterns. The authors implemented four different static checkers that can detect the anti-patterns. Quaranta et al. [27] proposed Pynblint, a static analyzer for Python notebooks. Pynblint performs a simple linter-based analysis to identify recommendations to the developer based on a list of 17 best practices based on code-quality or driving a more reproducible code. NBLyzer [32] is another static analyzer based on abstract interpretation for intra-cell analyses. NBLyzer supports two analyses, a code impact analysis and a data leakage analysis. Advanced by our analysis framework and the novel Python representation of notebooks with retaining cell information and execution order, our rules are not only inter-procedural but also inter-cell analyses.

7 Conclusion

This paper introduces our initial efforts to shift static analysis to the left for ML code. In support of this goal, we identified common defects that arise when developing ML models with computational notebooks. We showcased six analysis rules that catch both notebook-specific issues and misuses of deep-learning libraries. Finding real bugs with these rules on close to 10,000 experimentation notebooks demonstrates the value for ML practitioners in providing support for best practices, reproducibility, as well as assurance of scientific correctness. This motivates us to develop more rules in this space in the future.

Bibliography

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G.S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X.: TensorFlow: Large-scale machine learning on heterogeneous systems (2015), URL https://www.tensorflow.org/, software available from tensorflow.org

[2] Apache: Apache MXNet (2022), URL https://mxnet.apache.org/versions/1.9.1/

[3] Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., Engler, D.: A few billion lines of code later: Using static analysis to find bugs in the real world. Commun. ACM **53**(2), 66–75 (Feb 2010), ISSN 0001-0782, https://doi.org/10.1145/1646353.1646374, URL https://doi.org/10.1145/1646353.1646374

[4] Boyd, S., Boyd, S.P., Vandenberghe, L.: Convex optimization. Cambridge university press (2004)

[5] Chollet, F., et al.: Keras (2015), URL https://keras.io

[6] Dilhara, M., Ketkar, A., Dig, D.: Understanding software-2.0: A study of machine learning library usage and evolution. ACM Trans. Softw. Eng. Methodol. **30**(4) (jul 2021), ISSN 1049-331X, https://doi.org/10.1145/3453478, URL https://doi.org/10.1145/3453478

[7] Distefano, D., Fähndrich, M., Logozzo, F., O'Hearn, P.W.: Scaling static analyses at Facebook. Commun. ACM **62**(8), 62–70 (Jul 2019), ISSN 0001-0782, https://doi.org/10.1145/3338112, URL https://doi.org/10.1145/3338112

[8] Dolby, J., Shinnar, A., Allain, A., Reinen, J.: Ariadne: Analysis for machine learning programs. In: Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, p. 1–10, MAPL 2018, Association for Computing Machinery, New York, NY, USA (2018), ISBN 9781450358347, https://doi.org/10.1145/3211346.3211349, URL https://doi.org/10.1145/3211346.3211349

[9] Gamma, E., Helm, R., Johnson, R.E., Vlissides, J.M.: Design patterns: Abstraction and reuse of object-oriented design. In: Nierstrasz, O. (ed.) ECOOP'93 - Object-Oriented Programming, 7th European Conference, Kaiserslautern, Germany, July 26-30, 1993, Proceedings, Lecture Notes in Computer Science, vol. 707, pp. 406–431, Springer (1993), https://doi.org/10.1007/3-540-47910-4_21, URL https://doi.org/10.1007/3-540-47910-4_21

[10] Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., Chaudhuri, S., Vechev, M.: AI2: Safety and robustness certification of neural networks with abstract interpretation. In: 2018 IEEE Symposium on Security and Privacy (SP), pp. 3–18 (2018), https://doi.org/10.1109/SP.2018.00058

[11] Grotov, K., Titov, S., Sotnikov, V., Golubev, Y., Bryksin, T.: A large-scale comparison of Python code in Jupyter notebooks and scripts. In: Proceedings of the 19th International Conference on Mining Software Repositories, p. 353–364, MSR '22, Association for Computing Machinery, New York, NY, USA (2022), ISBN 9781450393034, https://doi.org/10.1145/3524842.3528447, URL https://doi.org/10.1145/3524842.3528447

[12] Guest, G., Bunce, A., Johnson, L.: How many interviews are enough? an experiment with data saturation and variability. Field methods **18**(1), 59–82 (2006)

[13] Humbatova, N., Jahangirova, G., Bavota, G., Riccio, V., Stocco, A., Tonella, P.: Taxonomy of real faults in deep learning systems. In: Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering, p. 1110–1121, ICSE '20, Association for Computing Machinery, New York, NY, USA (2020), ISBN 9781450371216, https://doi.org/10.1145/3377811.3380395, URL https://doi.org/10.1145/3377811.3380395

[14] Ioffe, S., Szegedy, C.: Batch normalization: Accelerating deep network training by reducing internal covariate shift. In: International conference on machine learning, pp. 448–456, PMLR (2015)

[15] Kluyver, T., Ragan-Kelley, B., Pérez, F., Granger, B., Bussonnier, M., Frederic, J., Kelley, K., Hamrick, J., Grout, J., Corlay, S., Ivanov, P., Avila, D., Abdalla, S., Willing, C., development team, J.: Jupyter notebooks – a publishing format for reproducible computational workflows. In: Loizides, F., Scmidt, B. (eds.) Positioning and Power in Academic Publishing: Players, Agents and Agendas, pp. 87–90, IOS Press (2016), URL https://eprints.soton.ac.uk/403913/

[16] Lagouvardos, S., Dolby, J., Grech, N., Antoniadis, A., Smaragdakis, Y.: Static Analysis of Shape in TensorFlow Programs. In: Hirschfeld, R., Pape, T. (eds.) 34th European Conference on Object-Oriented Programming (ECOOP 2020), Leibniz International Proceedings in Informatics (LIPIcs), vol. 166, pp. 15:1–15:29, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, Dagstuhl, Germany (2020), ISBN 978-3-95977-154-2, ISSN 1868-8969, https://doi.org/10.4230/LIPIcs.ECOOP.2020.15, URL https://drops.dagstuhl.de/opus/volltexte/2020/13172

[17] LeCun, Y., Touresky, D., Hinton, G., Sejnowski, T.: A theoretical framework for back-propagation. In: Proceedings of the 1988 connectionist models summer school, vol. 1, pp. 21–28 (1988)

[18] Liu, C., Lu, J., Li, G., Yuan, T., Li, L., Tan, F., Yang, J., You, L., Xue, J.: Detecting TensorFlow program bugs in real-world industrial environment. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 55–66 (2021), https://doi.org/10.1109/ASE51524.2021.9678891

[19] Madhyastha, P., Jain, R.: On model stability as a function of random seed. arXiv preprint arXiv:1909.10447 (2019)

[20] Microsoft: Pyright: Static type checker for Python (2022), URL https://github.com/microsoft/pyright

[21] Mukherjee, R., Tripp, O., Liblit, B., Wilson, M.: Static analysis for AWS best practices in Python code. In: Ali, K., Vitek, J. (eds.) 36th European Conference on Object-Oriented Programming, ECOOP 2022, June 6-10, 2022, Berlin, Germany, LIPIcs, vol. 222, pp. 14:1–14:28, Schloss Dagstuhl - Leibniz-Zentrum für Informatik

(2022), https://doi.org/10.4230/LIPIcs.ECOOP.2022.14, URL https://doi.org/10.4230/LIPIcs.ECOOP.2022.14

[22] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al.: PyTorch: An imperative style, high-performance deep learning library. Advances in neural information processing systems **32** (2019)

[23] Pimentel, J.a.F., Murta, L., Braganholo, V., Freire, J.: A large-scale study about quality and reproducibility of Jupyter notebooks. In: Proceedings of the 16th International Conference on Mining Software Repositories, p. 507–517, MSR '19, IEEE Press (2019), https://doi.org/10.1109/MSR.2019.00077, URL https://doi.org/10.1109/MSR.2019.00077

[24] Python Software Foundation: The Python standard library: typing — support for type hints: typing.Union (2022), URL https://docs.python.org/3/library/typing.html#typing.Union

[25] Python Software Foundation: The Python standard library: typing — support for type hints: The Any type (2022), URL https://docs.python.org/3/library/typing.html#the-any-type

[26] Quaranta, L.: Assessing the quality of computational notebooks for a frictionless transition from exploration to production. In: Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings, p. 256–260, ICSE '22, Association for Computing Machinery, New York, NY, USA (2022), ISBN 9781450392235, https://doi.org/10.1145/3510454.3517055, URL https://doi.org/10.1145/3510454.3517055

[27] Quaranta, L., Calefato, F., Lanubile, F.: Pynblint: a static analyzer for Python Jupyter notebooks. In: 2022 IEEE/ACM 1st International Conference on AI Engineering – Software Engineering for AI (CAIN), pp. 48–49 (2022), https://doi.org/10.1145/3522664.3528612

[28] Rasley, J., Rajbhandari, S., Ruwase, O., He, Y.: Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In: Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, pp. 3505–3506 (2020)

[29] Research, I.: WALA: The T. J. Watson libraries for analysis (2022), URL https://github.com/wala/WALA

[30] Ruder, S.: An overview of gradient descent optimization algorithms. arXiv preprint arXiv:1609.04747 (2016)

[31] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.: Dropout: a simple way to prevent neural networks from overfitting. The journal of machine learning research **15**(1), 1929–1958 (2014)

[32] Subotić, P., Milikić, L., Stojić, M.: A static analysis framework for data science notebooks. In: Proceedings of the 44th International Conference on Software Engineering: Software Engineering in Practice, p. 13–22, ICSE-SEIP '22, Association for Computing Machinery, New York, NY, USA (2022), ISBN 9781450392266, https://doi.org/10.1145/3510457.3513032, URL https://doi.org/10.1145/3510457.3513032

[33] Urban, C.: Static analysis of data science software. In: Chang, B.Y.E. (ed.) Static Analysis, pp. 17–23, Springer International Publishing, Cham (2019), ISBN 978-3-030-32304-2

[34] Wan, C., Liu, S., Hoffmann, H., Maire, M., Lu, S.: Are machine learning cloud APIs used correctly? In: 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pp. 125–137 (2021), https://doi.org/10.1109/ICSE43902.2021.00024

[35] Wan, Z., Xia, X., Lo, D., Murphy, G.C.: How does machine learning change software development practices? IEEE Transactions on Software Engineering **47**(9), 1857–1871 (2021), https://doi.org/10.1109/TSE.2019.2937083

[36] Wang, J., Kuo, T.y., Li, L., Zeller, A.: Restoring reproducibility of Jupyter notebooks. In: 2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), pp. 288–289 (2020)

[37] Wu, D., Shen, B., Chen, Y., Jiang, H., Qiao, L.: Tensfa: Detecting and repairing tensor shape faults in deep learning systems. In: 2021 IEEE 32nd International Symposium on Software Reliability Engineering (ISSRE), pp. 11–21 (2021), https://doi.org/10.1109/ISSRE52982.2021.00014

[38] Zhang, Y., Ren, L., Chen, L., Xiong, Y., Cheung, S.C., Xie, T.: Detecting numerical bugs in neural network architectures. In: Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, p. 826–837, ESEC/FSE 2020, Association for Computing Machinery, New York, NY, USA (2020), ISBN 9781450370431, https://doi.org/10.1145/3368089.3409720, URL https://doi.org/10.1145/3368089.3409720