Refactoring Machine Learning

Andrew Slavin Ross
Paulson School of Engineering and Applied Sciences
Harvard University
andrew_ross@g.harvard.edu

Jessica Zosa Forde
Project Jupyter
jzf2101@columbia.edu

Abstract

Results in machine learning scholarship are sometimes based on untested, difficult-to-read code that has only been seen by a single researcher. We argue that this is bad, and that machine learning scholarship could be improved by adhering to software development best practices. We identify several practices which are not widely followed within the research community but we believe would help improve the reliability of results. We describe how to apply these best practices in a machine learning research setting. Finally, we suggest several modifications to the publication review process to encourage adherence.

1 Introduction: A Tale of Two Rigors

Researching machine learning often requires skill in both mathematics and software development. These two fields, however, have very different ideas about what constitutes rigor or clarity.

As new ML researchers have flooded into the field [38], some have lamented a slackening of mathematical rigor [34, 22]. Even techniques that work well empirically have sometimes turned out to be based on faulty proofs [19] or imprecise explanations [18]. A major focus of criticism, perhaps more from a general scientific perspective than a mathematical one, has been on reproducibility [17].

One emerging school of thought in how to address some of these challenges is that we need to bring back the mathematical "rigor police" [34]. We need to clearly separate which of our conclusions are empirical and which are theoretically backed [22]; and the ones that are theoretically backed need to be based on more solid analysis. One embodiment of this zeitgeist may be seen in the shifting of the subfield of adversarial robustness towards provable, certified defenses [33].

Another common refrain is that we need to adhere to a stricter notion of scientific rigor in our empirical studies [36]. If our methods are stochastic, we need to always rerun them with multiple random seeds [17]; if our baselines have hyperparameters, we need to tune them just as aggressively as those of our main contribution [28]; and if we introduce techniques with multiple components, we must always run thorough ablation studies [40]. Furthermore, we need to publish the full code and data pipelines behind our papers [12, 3] and even systematically recruit independent researchers to reproduce our results [2].

While these steps are all laudable, most of them tend to be couched in a mathematical or scientific notion of rigor. Even more engineering-minded critiques tend to be focused on technical debt in real-world applications [35] rather than "research debt" incurred in academic settings [32].

The argument in this paper is that there are a wealth of techniques and best practices from the world of professional software development that have been insufficiently embraced or even understood by the machine learning community. These practices are generally focused on enhancing code clarity and comprehensibility while reducing the incidence of bugs.

The issue of bugs in machine learning research code is also an important one, which has gone largely unaddressed by the critiques cited above. Researchers may encounter bugs during the course of

32nd Conference on Neural Information Processing Systems (NeurIPS 2018), Montréal, Canada.

implementing research code that make certain techniques seem more effective than they actually are [23]. Many of these problems are fixed before publication, but it is possible that some errors may remain [25, 24, 6], especially if researchers are engaged in active experimentation in the final hours before a deadline. In professional and open source software development, projects of sufficient size are rarely entirely bug-free [41, 11, 9]. However, bugs in software ultimately need to be fixed; the existence of real people continually using a software application exerts an external pressure to identify and resolve issues. Research pipelines, generally run a limited number of times before publication, face no such pressure. Although we do not want to increase the workload of overburdened machine learning researchers, we do feel that in the publication review process, there should be *some* way to reward researchers for basing their conclusions on code that is clear and validated by many tests and sets of eyes—as opposed to code which is opaque, untested, and only seen by a single person.

For the rest of this paper, we will describe practices commonly used for quality assurance in real-world software development. We will make suggestions for how they might be applied to machine learning research. Finally, we will propose an alteration to the review process for machine learning conferences that will reward research based on code that is not just reproducible, but clear and well-tested.

2 Software Best Practices + ML

2.1 Test-Driven Development

Test-driven development [7, 39] is a method of developing software in which application code is developed in conjunction with (or even after) automated tests for how that code should behave. For any function or class, one generally writes simple "sanity-check" tests as well as more complex tests that span a space of "edge cases" describing how the function should behave under different conditions. Automated tests of a single function or class are called unit tests; longer, more involved tests which span multiple modules are usually referred to as integration or acceptance tests. Writing tests provides an extra layer of certainty that code is correct, helps prevent bugs before they happen by forcing developers to consider failure modes, and encourages modularity in software design (since modular components are easier to test).

Testing does not appear to be a common practice in machine learning research labs. One seemingly-persuasive objection is that many machine learning techniques involve stochasticity, which can be difficult to deterministically test. However, this is not an insurmountable barrier. First of all, stochasticity is rarely present in all parts of a method. Novel penalty terms with deterministic inputs and outputs can and should always be tested. The same goes for preprocessing transformations. Finally, irreducible stochasticity can be dealt with in a manner similar to the reparameterization trick [20]; functions with stochastic outputs can be rewritten deterministically to accept random variables as additional inputs, which can enable exact testing. For an excellent tutorial on how to test Markov-Chain Monte Carlo code (inspired by a minor bug that resulted in the authors' retraction of their paper [23, 25]), see [16].

2.2 Code Review

Code review is another important software development practice that often goes hand-in-hand with automated testing. Many companies have strict rules that code changes cannot be released until several developers besides the author(s) have closely read and reviewed them. Often such changes go through multiple rounds of revisions and refactoring before the process completes. While there is some disagreement about the frequency with which bugs are actually caught during code review [10], there is evidence to support the claim that code which is heavily reviewed tends to contain fewer bugs [27]. This may be because the knowledge that others will be reviewing code encourages developers to ensure that it is well-tested and clearly written beforehand, which prevents bugs from being introduced in the first place.

To our knowledge, there have not been any studies examining the practice and frequency of code review in machine learning groups. Anecdotally, code review does not appear to be standard practice in the field, though this may differ slightly in large engineering organizations. Either way, the possibility that many widely-cited research results are based on code that has never been tested or even seen by anyone other than the first author should be of concern to the research community.

2.3 Object-Oriented Design, Meaningful Names, and Refactoring

Testing and peer review are important, but apart from the minor tweaks that testing stochastic code encourages, they do not encode any strong opinions on the structure and organization of research code—or how we think about research more generally. However, in this section, we would like to make a slightly broader set of arguments about how we organize both code and *concepts* in machine learning research.

Many experts in software design [30] and human-centered design more generally [31] have pointed out the dangers of using the wrong abstractions.[1] The difficulty and importance of choosing good names can be summarized by the popular adage that "there are only two hard things in Computer Science: cache invalidation and naming things" [14].

One example of this issue in machine learning is pointed out by [22], who argue that the field is guilty of using misleading colloquialisms to describe techniques. However, while choosing misleading names is definitely counterproductive, negative issues can also result from choosing meaningless names, as software developers have long argued [37]. Although rigorous adherence to formal mathematics and statistics is often seen as a corrective to machine learning's failings, those fields have their own troubling tendencies to name concepts after people or even Greek letters [1, 8].

We have two categories of suggested changes. The first are stylistic and the second are structural.

Stylistically, we would like to suggest that machine learning move towards meaningful, intention-revealing names for techniques and variables, in both code and proofs. We suggest discouraging the use of Greek letters, single Roman letters, or researchers' last names except when absolutely necessary. For example, we might prefer `params` over `theta` in code, or even in a proof. In both cases, one could consider rewriting complicated expressions using the "extract variable" pattern [13], where intermediate quantities are given short but meaningful names. Using intention-revealing names instead of Greek letters in derivations may seem anathema to mathematicians, but we believe that the goal of typesetting should be clarity, not concision.

Structurally, we believe it would be helpful to organize both machine learning code and concepts in a more object-oriented fashion. In particular, we recommend studying the object-oriented best practices, such as SOLID [5, 26]. Specific examples of how this could affect machine learning include treating models, datasets, and predictions as classes, and using compositions of objects with dependency injection to avoid "god functions" [4] and "parameter hell" [21].[2] Thinking in terms of inheritance patterns may also be helpful for organizing and describing different abstract machine learning methods as well. Many techniques can be defined in a modular fashion, or considered as different special cases of the same abstract formulation. Object-oriented design provides an intuitive framework for representing these kinds of relationships.

Overall, what we would love to see is a motion towards *refactoring* [15], both within specific research projects' codebases and across the field. Knowledge—not just code—can and should be frequently renamed and reorganized. In doing so, we echo the calls of platforms such as `distill.pub`, whose stated mission is to help pay down the "research debt" machine learning has incurred in its rapid expansion [32]. We believe that software design has many helpful tools to offer in this effort.

3 Incentivizing Meaningful Change

In the previous section, we argued that software development best practices offer many tools for improving the quality of machine learning research. However, from the authors' personal experience, familiarity with these best practices is not always sufficient to guarantee adherence. Instead, we suspect that it could be more effective to provide incentives. To that end, we propose the following set of changes to the machine learning conference review process:

- Create a platform for anonymized code submission to conferences.
- Require that computational papers provide reference code with their initial submission replicating at least toy experiments.

[1] For a strange but excellent recorded talk about this problem, see [5].

[2] [29] recommends limiting methods to just four parameters, *including* keyword arguments.

- Require that reference code contain a suite of automated tests and sanity checks suitable for external review.
- Encourage reviewers to audit code for both correctness and clarity and factor it into their acceptance decision.

While these requirements may seem stringent, we think they could have many positive effects. In particular, we hypothesize that:

- Requiring submission of well-structured code alongside a paper strongly will discourage last-minute experimentation and tweaking, which is a potential source of error.
- Requiring testing will help catch additional bugs and encourage research software to be cleaner and more reusable.
- If papers are always accompanied by well-structured code, they may be easier to both replicate and extend, which could spur innovation.
- Even if doing a full code review is not always feasible for overburdened paper reviewers, the simple fact that a reviewer *could* potentially examine a paper's code in a way that affects the acceptance decision may make more authors follow better practices.

Of course, there are also negative potential consequences, including difficulty recruiting reviewers, selection bias towards papers in areas that require more (or less) code, and inconsistencies in the quality of code review that lead to unfair outcomes. However, these consequences seem possible to avoid if the review process is well designed and implemented, and the potential benefits to machine learning scholarship remain significant. As such, we recommend that a major machine learning publication venue decide to experiment with this process, gather data, and recommend improvements.

4 Conclusion

In this paper, we argued that machine learning scholarship could be improved by adherence to software development best practices. We described several practices that we believe could be particularly helpful and described how they could be applied to machine learning research code. Finally, we suggested a modification to the machine learning publication review process that we believe could improve the reliability and reproducibility of results.

Overall, we hope the field of machine learning will encourage more research into critiquing and correcting its own methodology. One area for future work that we believe would be fruitful (if controversial) would be an empirical investigation into the prevalence of bugs in machine learning research pipelines. We strongly suspect the issue is more prevalent and problematic than commonly assumed, but have been unable to find any comprehensive studies that explore it. Given the frequency with which bugs occur in commercial software, we believe it is important for the field's credibility to take steps to measure and mitigate its mistakes.

References

[1] `https://en.wikipedia.org/wiki/List_of_probability_distributions`.

[2] 2019 ICLR reproducibility challenge. `https://reproducibility-challenge.github.io/iclr_2019`.

[3] CodaLab. `http://codalab.org`.

[4] SourceMaking Anti-Patterns. The blob. `https://sourcemaking.com/antipatterns/the-blob`.

[5] Kane Baccigalupi. Going evergreen. `http://confreaks.tv/videos/rubyconf2014-going-evergreen`, 2014.

[6] Virginia Barbour, Theodora Bloom, Jennifer Lin, and Elizabeth Moylan. Amending published articles: time to rethink retractions and corrections? *F1000Res.*, 6, November 2017.

[7] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[8] Sung-Hyuk Cha. Comprehensive survey on distance/similarity measures between probability density functions. *International Journal of Mathematical Models and Methods in Applied Sciences*, 1(4):300–307, 2007.

[9] Alex Cooper, Jamie Townsend, and Michael Hughes. autograd issue 412. `https://github.com/HIPS/autograd/pull/412`, June 2018. Accessed: 2018-10-23.

[10] Jacek Czerwonka, Michaela Greiler, and Jack Tilford. Code reviews do not find bugs: how the current code review best practice slows us down. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 27–28. IEEE Press, 2015.

[11] Andrii Degeler. Microsoft's latest windows 10 update is reportedly wiping user data. `https://www.engadget.com/2018/10/05/windows-10-october-update-1809-delete-data-wipe-user-profile/`, October 2018. Accessed: 2018-11-26.

[12] Jessica Forde, Matthias Bussonnier, Félix-Antoine Fortin, Brian Granger, Tim Head, Chris Holdgraf, Paul Ivanov, Kyle Kelley, M Pacer, Yuvi Panda, et al. Reproducing machine learning research on Binder. In *Machine Learning Open Source Software 2018: Sustainable communities (NIPS 2018 Workshop)*, 2018.

[13] Martin Fowler. Extract variable. `https://refactoring.com/catalog/extractVariable.html`, 1999.

[14] Martin Fowler. TwoHardThings. `https://martinfowler.com/bliki/TwoHardThings.html`, 2009.

[15] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.

[16] Roger B Grosse and David K Duvenaud. Testing MCMC code. *2014 NIPS workshop on Software Engineering for Machine Learning*, 2014.

[17] Peter Henderson, Riashat Islam, Philip Bachman, Joelle Pineau, Doina Precup, and David Meger. Deep reinforcement learning that matters. *AAAI Conference on Artificial Intelligence*, 2018.

[18] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, pages 448–456, 2015.

[19] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *International Conference on Learning Representations*, 2015.

[20] Diederik P Kingma, Tim Salimans, and Max Welling. Variational dropout and the local reparameterization trick. In *Advances in Neural Information Processing Systems*, pages 2575–2583, 2015.

[21] Jon Kuperman. Parameter hell. `https://codeplanet.io/parameter-hell`, 2014.

[22] Zachary C Lipton and Jacob Steinhardt. Troubling trends in machine learning scholarship. *ICML 2018: The Debates*, 2018.

[23] Richard Mann. Rethinking retractions. `http://prawnsandprobability.blogspot.com/2013/03/rethinking-retractions.html`, 2013.

[24] Richard Mann. Rethinking retractions: Rethought. `http://prawnsandprobability.blogspot.com/2017/06/rethinking-retractions-rethought.html`, 2016.

[25] Richard P Mann, Andrea Perna, Daniel Strömbom, Roman Garnett, James E Herbert-Read, David J T Sumpter, and Ashley J W Ward. Multi-scale inference of interaction rules in animal groups using bayesian model selection. *PLoS Comput. Biol.*, 9(3):e1002961, March 2013.

[26] Robert C Martin. *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.

[27] Shane McIntosh, Yasutaka Kamei, Bram Adams, and Ahmed E Hassan. The impact of code review coverage and code review participation on software quality: A case study of the qt, vtk, and itk projects. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 192–201. ACM, 2014.

[28] Gábor Melis, Chris Dyer, and Phil Blunsom. On the state of the art of evaluation in neural language models. *International Conference on Learning Representations*, 2018.

[29] Sandi Metz and Katrina Owen. 99 bottles of OOP. `https://www.sandimetz.com/99bottles`, 2017.

[30] Sandy Metz. The wrong abstraction. `https://www.sandimetz.com/blog/2016/1/20/the-wrong-abstraction`, 2016.

[31] Don Norman. *The design of everyday things: Revised and expanded edition*. Constellation, 2013.

[32] Chris Olah and Shan Carter. Research debt. *Distill*, 2(3):e5, 2017.

[33] Aditi Raghunathan, Jacob Steinhardt, and Percy Liang. Certified defenses against adversarial examples. *arXiv preprint arXiv:1801.09344*, 2018.

[34] Ali Rahimi and Ben Recht. Back when we were kids. *NIPS Test-of-Time Award Talk*, 2017.

[35] D. Sculley, Gary Holt, Daniel Golovin, Eugene Davydov, Todd Phillips, Dietmar Ebner, Vinay Chaudhary, and Michael Young. Machine learning: The high interest credit card of technical debt. In *SE4ML: Software Engineering for Machine Learning (NIPS 2014 Workshop)*, 2014.

[36] D. Sculley, Jasper Snoek, Alex Wiltschko, and Ali Rahimi. Winner's curse? on pace, progress, and empirical rigor. *International Conference on Learning Representations, Workshop Track*, 2018.

[37] Ben Shneiderman and Richard Mayer. Syntactic/semantic interactions in programmer behavior: A model and experimental results. *International Journal of Computer & Information Sciences*, 8(3):219–238, 1979.

[38] Yoav Shoham, Raymond Perrault, Erik Brynjolfsson, and Jack Clark. Artificial intelligence index, 2017 annual report. `http://cdn.aiindex.org/2017-report.pdf`, 2017.

[39] James Shore. The art of agile development: Test-driven development. `https://www.jamesshore.com/Agile-Book/test_driven_development.html`, 2010.

[40] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in Neural Information Processing Systems*, pages 5998–6008, 2017.

[41] Tom Warren. iOS 11 bugs are so common they now appear in apple ads. `https://www.theverge.com/2018/3/16/17131148/apple-ios-11-bug-face-id-ad`, March 2018. Accessed: 2018-11-26.