WASP CC&SE Module 2, assignment 2:

Rasmus Kjær Høier

My research focuses on using bilevel optimization techniques for deep learning. Many areas of deep learning can be rephrased as bilevel optimization problems where some constraints are placed on the network. I have used bilevel optimization for training quantized neural networks as well as for training energy based neural networks without the use of backpropagation. I like this research subject because it often can be used to find simple justifications and improvements for commonly used heuristics. My main passion is accelerating biologically motivated deep learning algorithms. Most such algorithms manage to remove some biologically implausible features of back propagation at the cost of greatly increased computational costs. I am trying to find algorithms that are both biologically plausible and practical.

The most software development adjacent part of my work is writing implementations of algorithms in order to experimentally verify their performance (mainly accuracy and runtime). It is important to make implementations publicly available when releasing a paper, as it allows others to easily test your claims as well as to build upon your work. I chose the subjects *Architecture and Design*, *Maintenance and Evolution* and *Automated Software Testing* and I mainly reflect on them from the point of view of a researcher wanting to share reproducible experiments (so it is not too specific to my specific research field). I have a background in natural science and my motivation for taking this course was to learn some software engineering basics, so that I can write more maintainable and shareable code.

Architecture and Design

I think the word bricolage describes my way of approaching software engineering during my undergraduate degree. Bricolage means constructing something out of whatever is at hand. In a programming context that could mean making do with what you already know, e.g. using for loops to modify rows of a Pandas dataframe rather than reading the documentation to learn how to efficiently vectorize operations. It could also mean relying too much on information that provide immediate solutions to a problem (e.g. stackoverflow), without providing deeper insight. I guess what I am saying is that there are often very many ways to solve problems, and choosing the best one requires deeper understanding, which comes faster when you don't take shortcuts but actually read the documentation.

During my PhD my approach has changed a lot and I now try to write my code in a more modular fashion, by dividing the code into smaller functions and using structs and classes (depending on language) to avoid passing too many arguments to my functions. This makes it easier for me to profile, debug and reuse code.

Most of the litterature is about people working in large teams (like the Microsoft papers [1] and [2]) and the development of systems for deployment so they deal with issues I don't have at the moment. For example they deal with the issue of version control for datasets, which I haven't had to worry about because I use toy benchmarking datasets. However issues like reuse of trained models and code is something I need to think about in my work. For my use cases a model is useless if I don't know what hyperparameters and what learning algorithm produced it, so I log all parameters as well as details regarding the training when running experiments. I save logs, configuration files and intermediate and final states of models to disk.

Maintenance and Evolution

When writing software for paper publications it is common for authors to just throw their code on github without thinking of maintenance and evolution.

Even if your paper is finished it is still important to make it easy for others to run it in the future and to be able to build upon it. In an academic setting I think it is actually more about evolution than maintenance. You don't want to have to update you papers code repository for every new version of JAX or Pytorch, but you should make it as easy as possible for others to install the correct versions of the software you used, e.g. by sharing instructions for creating the virtual environment you used to run the experiments. The evolution aspect consists in others being able to use your implementation in develop it further.

Automated Software Testing

I think automated testing can be great, but I personally don't use it for experiments for papers. I think it is more relevant in cases where your code develops into more than just an attachment to your paper. For example if you decide to publish it as a package/module that other people can use for different experiments or applications. In the past I have used github actions to run automated test after each push to a julia package I have (and github actions + CompatHelper.jl to automate pull requests when libraries my package depended on released new versions).

However, I do think it could be useful to always add one single test to your papers code. Typically you will include a file containing specifications for recreating the virtual environment (.toml for julia and .yml for conda). You could write a test that checks if the versions of packages used to run the code are the same as the versions specified in the virtual environment file. In the past I have issues reproducing a particular papers results because Pytorch had changed weight initialization scheme between two releases, which resulted in significantly different performance for the particular model. Testing that the correct packages are used and printing a warning could prevent such issues.

Future trends and directions for ML and software engineering with respect to my field

I don't see software engineering in an academic setting changing significantly in the future. There will probably be steady quality of life improvements, like better linters (like Luigis pynblint), and improved AI pair programming tools and version control systems. I like git a lot (much better than SVN, which I was taught in undergrad), but I still hope we will eventually get a version control system with a cleaner syntax and better warnings/error messages. Perhaps the next generation of version control system will also address the issue of data versioning mentioned by reference [2]. Perhaps linting and version control will eventually just be aspects of our AI copilots.

I would like for the ML community to move beyond Python in the future, but inspite of all Pythons flaws I don't think it will happen anytime soon. I am mainly referring to Pythons inherent slowness (which prompts you to write critical code in C++), the global interpreter lock, the poor and slow package management systems (conda and pip) and the fact that using something as fundamental as a for loop is detrimental to performance. Despite these flaws Python has a massive community and there is a sea of engineers with knowledge of the language. I believe this is a major reason why Google gave up on their *Swift for Tensorflow* project and are now developing the Python library JAX. In my opinion JAX is pretty great (It is basically a differentiable sublanguage within Python), but something as simple as a for loop has to be written as a very inelegant function call in order to avoid serious performance issues:

`jax.lax.fori_loop(lower, upper, body_fun, init_val)`. Although different from the example in reference [3] I think this is a good example of *technical debt*. Needing to write performance critical code in C++ and having to deal with peculiarities like jax.lax.fori_loop makes development slower and costlier long term, but in the short term this is outweighed by the benefits of Pythons existing ecosystem and community.

In the field of biologically inspired neural networks there is starting to be more interest in novel hardware accelerators. These include neuromorphic processing units based on analog electronics [4] (memristors in particular) and photonic processing units [5]. In the broader field of ML there are a variety of specialized CPUs and TPUs being developed and eventually quantum computing will also be viable for at least certain specialized problems. For the developers/academics I think having a diverse range of hardware accelerators to target might mean that you will need to use a different software framework/API depending on what hardware accelerator you are targeting. However, a benefit of the inertia of the community is that it will probably be sufficient to know Python.

References

[1] Kim et al, 2018, *Data Scientists in Software Teams: State of the Art and Challenges*

[2] Amershi et al, 2019, *Software Engineering for Machine Learning: A Case Study*

[3] Sculley et al, 2018, *Hidden Technical Debt in Machine Learning Systems*

[4] Kendall, 2020, *Training End-to-End Analog Neural Networks with Equilibrium Propagation*

[5] Launay et al, 2020, *Hardware Beyond Backpropagation: a Photonic Co-Processor for Direct Feedback Alignment*