On Machine Learning and Programming Languages

Mike Innes
Stefan Karpinski
Viral Shah
Julia Computing, Inc.

David Barber
Pontus Stenetorp
University College London
London, UK

Tim Besard
Ghent University
Ghent, Belgium

James Bradbury
Salesforce Research

Valentin Churavy
Simon Danisch
Alan Edelman
Jon Malmaud
Jarrett Revels
Massachusetts Institute of Technology
Cambridge, MA

Deniz Yuret
Koç University
Istanbul, Turkey

ABSTRACT

The complexity of Machine Learning (ML) models and the frameworks people are using to build them has exploded along with ML itself. State-of-the-art models are increasingly programs, with support for programming constructs like loops and recursion, and this brings out many interesting issues in the tools we use to create them — that is, programming languages (PL). This paper[1], discusses the necessity for a first class language for machine learning, and what such a language might look like.

ACM Reference Format:
Mike Innes, Stefan Karpinski, Viral Shah, David Barber, Pontus Stenetorp, Tim Besard, James Bradbury, Valentin Churavy, Simon Danisch, Alan Edelman, Jon Malmaud, Jarrett Revels, and Deniz Yuret. 2018. On Machine Learning and Programming Languages. In *Proceedings of SysML conference (SysML 2018)*. ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 PIG LATIN, AND OTHER HIDDEN LANGUAGES

TensorFlow (TF) and its ilk[2] are already programming languages [1], albeit limited ones. Though nominally written using another language, typically Python, this is only to build an expression tree in TF's internal language, which it then evaluates.

TF's lazy style is in effect meta-programming: writing code that writes code. In TF, Python serves as a meta-language for writing programs in TF's graph-based language.[3] TF's graph supports constructs like variable scoping and control flow, but instead of using Python syntax, you manipulate these constructs through an API.

TF and similar tools present themselves as "just libraries", but they are extremely unusual ones. Most libraries provide a simple set of functions and data structures, not an entirely new programming system and runtime. Why is such a complex approach necessary?

2 WHY CREATE A NEW LANGUAGE?

ML research has extremely high computational demands, and simplifying the modelling language makes it easier to add domain-specific optimisations and features. Modern ML requires excellent hardware support, good numerics, low interpreter overhead and multiple kinds of parallelism. Where general-purpose languages struggle to provide these features, TF can handle them seamlessly.

These impressive optimisations rely on simplifying assumptions (e.g. no recursion or custom gradients), which make it easier to apply optimisations or deploy to small devices. Unfortunately, ML researchers thoroughly enjoy violating these assumptions. Models now demand conditional branching (easy to add), loops for recurrence (less easy), and even recursion over trees [12] (virtually impossible). In many areas of ML, models are becoming increasingly like programs, including ones that reason about other programs (e.g. program generators [8] and interpreters [5]), and with non-differentiable components like Monte Carlo Tree Search. It's enormously challenging to build runtimes that provide complete flexibility while achieving top performance, but the most powerful models and groundbreaking results need both.

Another practical downside of this approach, at least in its current incarnations, is the need for meta-programming of the kind discussed above. Building and evaluating expression trees imposes significant additional burdens on both the programmer and the compiler. It becomes tricky to reason about because the code now has two execution times, each with different language semantics, and things like step-through debugging are much harder. This could be resolved by creating a syntactic language for the new runtime, but this requires creating a full new programming language. Is this worthwhile when we have popular numerical languages already?

[1] Based on a longer blog post at https://julialang.org/blog/2017/12/ml&pl
[2] We use TensorFlow for example, but could substitute other "define-before-run" frameworks like CNTK or MXNet.
[3] TF's graph is effectively a dataflow-based AST (Abstract Syntax Tree).

SysML 2018, February 2018, Stanford CA, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

3 CAN WE JUST USE PYTHON?

As ML models began to need the full power of a programming language, Chainer and others pioneered a "define-by-run" [9] approach wherein a Python program is itself the model, using runtime automatic differentiation (AD) for derivatives. This is fantastic from a usability standpoint: if you want a recursive model that operates over trees, simply write that down, and let the AD do its magic!

However Python scales poorly to ML's heavy computational demands. A huge amount of work goes into replicating optimisations that fast languages get for free, and the PL boneyard is full of failed efforts to make Python faster. Python's semantics also make it fundamentally difficult to provide model-level parallelism or compile models for small devices.

Efforts like Gluon for MXNet are attempting to get the best of both approaches, combining basic dynamic AD with code-tracing to produce "static sub-graphs" which can then be optimised. Such hybrids are a mashing together of disparate implementations and APIs, and it is unclear how to support both kernel-level optimisations and high-level graph scheduling.

4 WHAT MIGHT A TAILOR-MADE ML LANGUAGE LOOK LIKE?

There are few domains as demanding about language-level design issues as machine learning. But it's not unprecedented: in areas like formal reasoning and verification or cluster computing, new, tailor-made languages have proved an effective solution.

An obvious current challenge for ML languages is achieving generality alongside performance, and the early hybrid approaches will need much more development. Future ML runtimes will need to support arbitrary mixing of approaches (e.g. static within dynamic within static) and compiling of dynamic code for deployment. Ideally, there will only be single, flexible "graph format" (or AST). The AST should have a syntax and statically describe dynamic behaviour (e.g. with a written for loop); i.e. it should look a lot more like a standard programming language.

Programmable semantics would open up new levels of flexibility, and could be provided by a feature similar to macros. This would allow functionality like multi-GPU training to be built on top of the core system, by specifying where the code should have pure dataflow semantics (vs imperative semantics, which are more flexible but may include side-effects that are unsafe to optimise). It could also allow the kinds of program manipulation needed by probabilistic programming languages, or the vectorisation (batching) passes usually implemented by hand in NLP models [4].

As well as the PL community, ML engineers should pay close attention to the Automatic Differentiation (AD) community. ML languages can take inspiration from languages designed for truly first-class derivative support [11]. Such languages can mix symbolic with runtime techniques (helping with the above tradeoffs), combine forward and reverse mode AD (for improved performance and memory usage), and differentiate GPU kernels.

ML languages increasingly need more means for extension. Gone are the days when it was enough to hard-code support for strided arrays on NVIDIA GPUs; cutting-edge techniques like sparse machine learning [7], new hardware like TPUs, Nervana, FPGAs, and CoreML chips all call for greater levels of flexibility. Large-scale refactoring for each new development will not scale. Here we expect ML systems to take inspiration from numerical computing, where adding new hardware support or data representations can be accomplished by a user in high-level code [2].

Type systems can provide safety benefits, but there is room for more array-aware type systems where dimensions are meaningful (e.g. spatial vs channel vs batch dimensions in images), to help protect hairy dimension-permuting code. We expect the trend towards dynamic typing to continue,[4] mainly due to practitioners' preference for interactivity and scripting, but hope to see further innovations like CNTK's optionally dynamic dimensions.

ML engineers are increasingly interested in traditional software engineering problems [10] like maintenance and integration with production systems. The ML programming model makes it harder to create abstraction barriers and interfaces between components, and re-training of a model can easily break backwards compatibility. ML languages will likely be able to incorporate solutions to these problems, but this remains an open design problem.

A downside to any new language is that it requires a new library ecosystem. TensorFlow requires it's own libraries for tasks like image processing and file IO, instead of reusing vast effort behind the Python ecosystem. ML practitioners should not split from the wider numerical and HPC community. An ideal ML ecosystem is an ideal numerical one, and collaboration between these communities will multiply everyone's efforts.

We expect to see these developments coming from several angles. Graph IRs and formats like XLA, ONNX and NNVM are becoming ever more sophisticated and will likely take more inspiration from traditional language design, perhaps even becoming fully-fledged programming languages. TensorFlow's XLA has started a push towards special-purpose compiler stacks that now includes TVM, DLVM, and myelin. Meanwhile, projects like the PyTorch JIT, Gluon and Tangent are efforts to make Python itself a better modelling language. Having just argued that ML is a numerical programming languages problem, the authors feel that the Julia language [3] is an excellent substrate for experimenting with these kinds of language-level issues, and will continue to push the boundaries with projects such as Knet, Flux, Cassette, CUDAnative, DataFlow.jl.

5 CONCLUSION

ML models have become extremely general information-processing systems that build ever higher-level and more complex abstractions; recurrence, recursion, higher-order models, even stack machines [6] and language interpreters [5]. ML is a new programming paradigm, albeit one that's heavily numerical, differentiable and parallel. And as in any engineering field, the tooling available will have a profound impact on the scope and quality of future work.

All this suggests that designers of ML systems have a momentous challenge ahead of them. The ML and PL communities need to combine forces, and a core challenge is integrating the disparate expertise, in order to build systems that treat numerics, derivatives and parallelism as first-class features, without sacrificing traditional programming ideas and wisdom.

[4] Internally, current systems span the gamut from fully dynamic (PyTorch and its ATen backend) to unusually static (TensorFlow's XLA and MXNet, where all dimensions are known before the graph is run).

REFERENCES

[1] Martín Abadi, Michael Isard, and Derek G. Murray. 2017. A Computational Model for TensorFlow: An Introduction. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL 2017)*. https://doi.org/10.1145/3088525.3088527

[2] Tim Besard, Pieter Verstraete, and Bjorn De Sutter. 2016. High-level GPU programming in Julia. (2016). arXiv:1604.03410

[3] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. 2017. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59 (2017), 65–98. https://doi.org/10.1137/141000671

[4] Guy E. Blelloch. 1990. *Vector Models for Data-parallel Computing*. MIT Press.

[5] Matko Bošnjak, Tim Rocktäschel, Jason Naradowsky, and Sebastian Riedel. 2017. Programming with a Differentiable Forth Interpreter. In *Proceedings of the 34th International Conference on Machine Learning (Proceedings of Machine Learning Research)*, Doina Precup and Yee Whye Teh (Eds.), Vol. 70. PMLR, International Convention Centre, Sydney, Australia, 547–556. http://proceedings.mlr.press/v70/bosnjak17a.html

[6] Samuel R. Bowman, Jon Gauthier, Abhinav Rastogi, Raghav Gupta, Christopher D. Manning, and Christopher Potts. 2016. A Fast Unified Model for Parsing and Sentence Understanding. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Association for Computational Linguistics, Berlin, Germany, 1466–1477. arXiv:1603.06021 http://www.aclweb.org/anthology/P16-1139

[7] Laurent El Ghaoui, Guan-Cheng Li, Viet-An Duonga, Vu Pham, Ashok Srivastava, and Kanishka Bhaduri. 2011. Sparse Machine Learning Methods for Understanding Large Text Corpora. In *Proc. Conference on Intelligent Data Understanding*. https://people.eecs.berkeley.edu/~elghaoui/pubs_cidu2011.html

[8] Justin Johnson, Bharath Hariharan, Laurens van der Maaten, Judy Hoffman, Li Fei-Fei, C. Lawrence Zitnick, and Ross Girshick. 2017. Inferring and Executing Programs for Visual Reasoning. In *The IEEE International Conference on Computer Vision (ICCV)*. arXiv:1705.03633

[9] Graham Neubig, Chris Dyer, Yoav Goldberg, Austin Matthews, Waleed Ammar, Antonios Anastasopoulos, Miguel Ballesteros, David Chiang, Daniel Clothiaux, Trevor Cohn, Kevin Duh, Manaal Faruqui, Cynthia Gan, Dan Garrette, Yangfeng Ji, Lingpeng Kong, Adhiguna Kuncoro, Gaurav Kumar, Chaitanya Malaviya, Paul Michel, Yusuke Oda, Matthew Richardson, Naomi Saphra, Swabha Swayamdipta, and Pengcheng Yin. 2017. DyNet: The Dynamic Neural Network Toolkit. (January 2017). arXiv:1701.03980

[10] D. Sculley et al. 2015. Hidden Technical Debt in Machine Learning Systems. In *Advances in Neural Information Processing Systems 28 (NIPS 2015)*. https://papers.nips.cc/paper/5656-hidden-technical-debt-in-machine-learning-systems

[11] Jeffrey Mark Siskind and Barak A. Pearlmutter. 2016. Efficient Implementation of a Higher-Order Language with Built-In AD. (2016). arXiv:1611.03416

[12] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing*. Association for Computational Linguistics, Beijing, China, 1556–1566. http://www.aclweb.org/anthology/P15-1150