Automatic Differentiation in Myia

Extended Abstract

Olivier Breuleux
University of Montreal
breuleuo@iro.umontreal.ca

Bart van Merriënboer
University of Montreal
bart.van.merrienboer@umontreal.ca

ABSTRACT

Automatic differentiation is an essential feature of machine learning frameworks. However, existing implementations often have limitations. In dataflow programming frameworks such as Theano or TensorFlow the representation used makes supporting higher-order gradients difficult. On the other hand, operator overloading frameworks such as PyTorch are flexible, but do not lend themselves well to optimization. With Myia, we attempt to have the best of both worlds: Building on the work by Pearlmutter and Siskind we implement a first-order gradient operator for a subset of the Python programming language.

1 INTRODUCTION

1.1 Automatic differentiation

Gradient descent is the bread and butter of optimization in machine learning, and it requires easy and efficient access to first and second order derivatives. *Automatic differentiation* [6] (AD) provides this by transforming a numerical program through applying the chain rule to a set of primitive operators for which the derivatives are known. This approach ensures a constant overhead per operation while calculating derivatives accurately up to working precision. There are two main ways to apply the chain rule: starting from the inputs (*forward-mode* AD), or starting from the outputs (*reverse-mode* AD).

Reverse mode is usually preferred in machine learning because it is more efficient than forward mode when the number of parameters is larger than the number of outputs. Implementing reverse mode is more challenging than forward mode, since the control flow must be reversed and the backward phase reuses the intermediate values of the forward phase. In the presence of scoping and/or loops, this requires the use of runtime data structures to keep values alive. Further complications arise when AD must be implemented for existing languages which did not account for it in their design.

Traditionally, two different implementation approaches are distinguished in the AD literature [3]: *Operator overloading* (OO) involves tracing the program at runtime. Tracing follows function calls and control flow, so the final trace is a linear series of elementary operations (the *tape*) to which we can apply the chain rule. This approach is easy to implement and flexible, but may calculate unneeded intermediate values, and if the execution path diverges it requires repeated tracing and application of the chain rule.

Source code transformation (SCT) consists of applying the chain rule directly to an intermediate representation of the program, producing a new program that computes the derivative. This requires a transformation that reverses the execution path of the program, which means we need to explicitly consider control flow operators and function calls. Note that the resulting program must still use a runtime data structure to store intermediate values.

1.2 Automatic differentiation in ML

Many implementations of automatic differentiation exist [2]. However, machine learning frameworks with support for automatic differentiation failed to build on existing software used in other fields, and were developed almost entirely in parallel.

Theano [13] pioneered the use of automatic differentiation in machine learning research. It requires the user to explicitly construct a dataflow graph (computation graph) using Python. The chain rule is then applied to this graph followed by a series of optimizations. The computation graphs on which Theano operates, however, have little expressive power: there are no function calls and there is only limited support for loops through the monolithic `scan` construct. The explicit construction of the graph is also a tedious and error-prone process for the user.

TensorFlow [1] built on the dataflow programming paradigm, but without addressing all the fundamental flaws. TensorFlow's graphs must also be manually built. They also suffer from limited expressive power, and a fortiori, so do their derivatives.

PyTorch [10] uses operator overloading, and hence benefits from the full expressivity of the Python language. That said, it inherits the shortcomings inherent to the operator overloading approach.

Other machine learning frameworks with support for AD have been developed over the years such as MXNet, CNTK, Caffe, Chainer, Autograd, torch-autograd, etc. In general though, the same two techniques are employed: Either the user is required to explicitly construct a dataflow graph, or operator overloading is used.

2 MYIA

Myia[1]'s objective is to combine the usability of frameworks such as PyTorch, which allow users to write code directly in a dynamic language, with the performance benefits of source code transformation. Myia strives to support the following:

- Like Theano or TensorFlow, it should be amenable to static analysis and global optimization.
- Like PyTorch, the automatic differentiation should be fully general and support all major programming language constructs (function calls, conditionals, loops, recursion, etc.)
- Tight integration with the Python ecosystem, which is still the language of choice of most deep learning researchers.

2.1 Intermediate representation

Myia's approach is to parse the user's Python code into a functional representation which is amenable to reverse mode automatic differentiation. The resulting code is not executed by the Python interpreter, but by a custom runtime.

[1]Code available at `https://github.com/mila-udem/myia`

Myia's representation is graph-based and partly inspired from sea-of-nodes [4] and Thorin [7]. Like these representations, but unlike Theano or TensorFlow, it readily supports function abstraction, closures and recursion. In order to simplify algebraic optimizations and parallel scheduling, we use graph-based A-normal form [5] rather than CPS, the latter being too rigid about evaluation order. We have implemented crude inlining, dead code elimination, common subexpression elimination, constant propagation, and a few other simple optimizations.

2.2 Python parsing

While Myia's IR is different from Python, there is a straightforward translation. Figure 1 shows how a `while` loop is translated. Myia also supports `if`, `for`, nested `def` and `lambda`, simple and mutual recursion, and most arithmetic and logical operators.

Myia does not aim to support all of Python's features. In order to enable type inference and static optimization, we require objects to have static types and we disallow dynamic features such as `eval`. The most significant difference, however, is that Myia does not support destructive assignment i.e. statements of the form `x[i] = y` or `x += y` which assume mutability.[2]

There are two issues with destructive assignment. The first is that it is a side effect that creates potentially complex data dependencies, impeding code analysis, optimization and parallelism. The second is particular to the problem of reverse-mode automatic differentiation: In order to perform the backpropagation pass, we need to be able to trace back the computation, and we will likely need the old value of `x`. Hence we deviate from the standard Python semantics in Myia, and interpret `x[i] = y` similarly to e.g., OCaml's functional update syntax. One can read it as `x = x with x[i] = y`, which only shadows the `x` variable without modifying the original data.

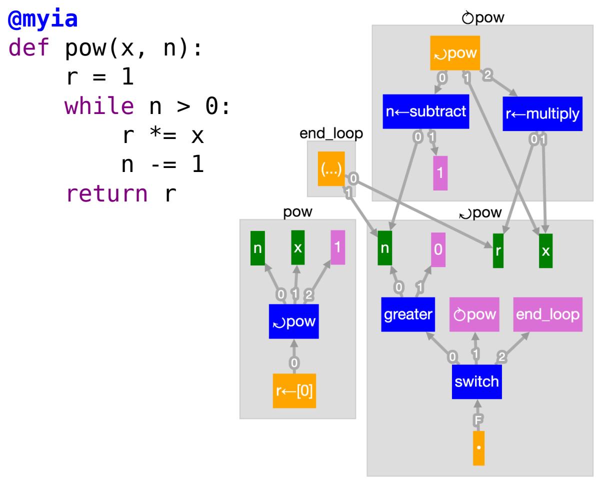
3 THE GRADIENT TRANSFORM

Myia presently offers a first-class higher order method, `grad` (more may be added in the future, e.g. for forward AD). `grad` performs reverse-mode AD on its argument and returns a function such that `grad(f)(x, y) == df(x, y)/dx`. We have the following requirements for this transform:

(1) It must be able to differentiate *any* valid program.
(2) Nested application of `grad` must be supported, allowing for higher-order derivatives to be taken.
(3) It must be amenable to type inference and optimization.

As discussed earlier, operator overloading does not lend itself well to optimization. On the other hand, most traditional source code transformation methods use a stack (*tape*) to store intermediate values on at runtime. This introduces a mutable runtime structure into the program, which complicates type inference and optimization. Higher-order gradients are complicated by the fact that the gradient transform must explicitly handle read and write operations on this tape. If, on the other hand, the transform produces a program without side-effects and valid in the original formulation, then it should be possible to get higher order gradients simply by applying the transform repeatedly. Since we do not introduce

[2]Note that the statement `x = x + y` is not considered to be a destructive assignment, because it can be safely rewritten as an assignment to a new variable: x_2 `= x + y`, with substitution of x_2 for every occurrence of `x` following the assignment.



```
@myia
def pow(x, n):
    r = 1
    while n > 0:
        r *= x
        n -= 1
    return r
```

Figure 1: Transform of a simple Python program into Myia's representation. Grey boxes are functions, green nodes are inputs, orange nodes are return nodes, blue nodes are applications, pink nodes are constants. Pointers from a function to another means the former is a closure nested in the latter. `(...)` builds a tuple of its inputs and `[0]` takes the first element of a tuple.

explicit runtime data structures, all regular optimization methods will remain valid and effective.

In [9] a method for implementing reverse mode AD in functional languages is introduced. It defines a source code transformation (denoted $\overleftarrow{\mathcal{J}}$) on functions such that as the forward computation progresses, the backpropagation computation is constructed as a chain of closures. That chain packages together the intermediate computations with the code necessary to compute and combine gradients with respect to them, eliminating the need for a tape. The difficulty of taking derivatives in the presence of closures is solved by tracking the gradients with respect to their free variables. This approach thus satisfies the desire to be able to differentiate any valid program while supporting higher-order differentiation.

Optimization of functional programs, including the optimization of closures, is a well-studied field [11][12]. Even using a limited set of local optimizations and inlining, we were able to simplify the gradient of straightforward programs to resemble what one would write by hand.

4 FUTURE WORK

Myia is currently a functioning prototype, but we see many avenues to improvement and new features. We plan to implement advanced control flow analysis in order to help optimizing the output of the gradient transform in the general case. We also intend on writing a GPU-enabled backend for Myia, and compiling down to a lower-level language. We are also considering approaches to dealing with the overhead of the "copy on write" implementation of `x[i] = y`, which guarantees the immutability of arrays, for example by using persistent data structures [8].

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. (2015). https://www.tensorflow.org/ Software available from tensorflow.org.

[2] Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2015. Automatic differentiation in machine learning: a survey. *arXiv preprint arXiv:1502.05767* (2015).

[3] Christian H Bischof and H Martin Bücker. 2000. *Computing derivatives of computer programs.* Technical Report. Argonne National Lab., IL (US).

[4] Cliff Click and Keith D. Cooper. 1995. Combining Analyses, Combining Optimizations. *ACM Trans. Program. Lang. Syst.* 17, 2 (March 1995), 181–196. https://doi.org/10.1145/201059.201061

[5] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation (PLDI '93).* ACM, New York, NY, USA, 237–247. https://doi.org/10.1145/155090.155113

[6] Andreas Griewank and Andrea Walther. 2008. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation* (second ed.). Society for Industrial and Applied Mathematics, Philadelphia, PA, USA.

[7] Roland Leissa, Marcel Koster, and Sebastian Hack. 2015. A Graph-based Higher-order Intermediate Representation. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '15).* IEEE Computer Society, Washington, DC, USA, 202–212. http://dl.acm.org/citation.cfm?id=2738600.2738626

[8] Chris Okasaki. 1998. *Purely Functional Data Structures.* Cambridge University Press, New York, NY, USA.

[9] Barak A. Pearlmutter and Jeffrey Mark Siskind. 2008. Reverse-mode AD in a functional framework: Lambda the ultimate back propagator. *ACM Trans. Program. Lang. Syst.* 30, Article 7 (March 2008), 36 pages. Issue 2.

[10] PyTorch Development Team. 2017. PyTorch: Tensors and Dynamic neural networks in Python with strong GPU acceleration. (2017). http://pytorch.org/ http://pytorch.org/.

[11] Olin Grigsby Shivers. 1991. *Control-flow Analysis of Higher-order Languages of Taming Lambda.* Ph.D. Dissertation. Pittsburgh, PA, USA. UMI Order No. GAX91-26964.

[12] Jeffrey Siskind and Barak Pearlmutter. 2008. *Using Polyvariant Union-Free Flow Analysis to Compile a Higher-Order Functional-Programming Language with a First-Class Derivative Operator to Efficient Fortran-like Code.* Technical Report.

[13] Theano Development Team. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints* abs/1605.02688 (May 2016). http://arxiv.org/abs/1605.02688