Automatic Differentiation in Myia

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

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

Abstract

Automatic differentiation is an essential feature of machine learning frameworks. However, its implementation in existing frameworks often has 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. Given a function $f : \mathbb{R}^n \to \mathbb{R}^m$, *forward-mode* AD applies the chain rule starting from the inputs and requires n evaluations of the transformed program to compute the derivative. *Reverse-mode* AD starts from the output and requires m evaluations.

Reverse mode is usually preferred in machine learning since often $m = 1$ with n large, potentially in the order of millions. Note that implementing reverse mode is more challenging than forward mode. Firstly, it requires reversing the program's control flow. Secondly, the backward computation reuses the intermediate values of the forward computation. In the presence of scoping and/or loops, this requires the use of runtime data structures to keep values alive. The implementation of automatic differentiation is further complicated by the fact that it must often be implemented for existing languages which were designed without support for AD in mind.

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 if the execution path diverges it requires repeated tracing and application of the chain rule. Moreover, the forward pass is executed regardless of whether the intermediate values are required for the backward pass or not.

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.

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

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 [11] pioneered the use of automatic differentiation in machine learning research, at a time when the gradient computations for new models were still derived manually. Theano 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 such as common subexpression elimination (CSE) and dead code elimination (DCE). 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. Furthermore, the explicit construction of the graph is a time-consuming and error-prone process for the user compared to a more implicit approach such as operator overloading.

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.

Many other machine learning frameworks with support for AD have been developed over the years such as MXNet, CNTK, Caffe, Caffe 2, 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'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.

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.

While the representation that Myia uses remains flexible at this stage of development, it is essentially a minimalistic Lisp, close to pure lambda calculus: Functions and closures are first class objects, branches in conditionals are represented as thunks (functions with no arguments, to be called conditionally), and loops are implemented using recursion.

2.1 Python parsing

While the aforementioned functional representation is different from Python, there is a straightforward translation. Figure 1 shows an example translation using our IR as it currently stands (the thunks for the branches of the conditional are folded in for readability).

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. We also disallow dynamic execution and introspection through functions such as `eval`. The most significant difference, however, is that Myia

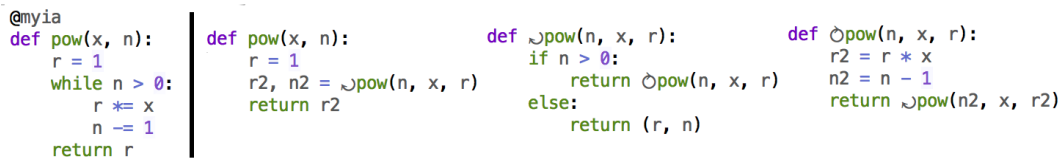

Figure 1: Transform of a simple Python program into Myia's functional representation. The original function (left) is split into three functions or basic blocks (right): the main body, the condition and the loop body. The latter two take as parameters the free variables used and set by the loop, thus the ostensibly imperative code on the left is successfully converted to a functional representation. The Python syntax used on the right does not correspond to Myia's actual representation and is only illustrative.

does not support destructive assignment i.e. statements of the form `x[i] = y` or `x += y` which assume mutability.[1]

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

Given we have parsed Python code into a functional representation, we need to transform it so that it can compute the gradients of the functions the user asks for. We have the following requirements for this tranformation:

1. It must be able to differentiate *any* valid program.
2. Nested application of the transformation 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 explicit runtime data structures, all regular optimization methods will remain valid and effective.

In [9] a method for implementing reverse mode AD is introduced on a functional representation very close to ours. It does so by defining 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. An illustration of the transform is shown in figure 2.

[1] 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: $\texttt{x}_2$ `= x + y`, with substitution of $\texttt{x}_2$ for every occurrence of `x` following the assignment.

```
1  def f(x, fv):
2      def clos(y):
3          a = g(y, fv)
4          b = h(a, y)
5          return b
6      c = clos(x)
7      return c
```

```
1  def ↑f(↑x, ↑fv):
2      def ↑clos(↑y):
3          ↑a, ◊a = ↑g(↑y, ↑fv)
4          ↑b, ◊b = ↑h(↑a, ↑y)
5          def ◊clos(∇b):
6              ∇y, ∇g, ∇fv, ∇h, ∇a = 0...
7              ∇h, ∇a, ∇y += ◊b(∇b)
8              ∇g, ∇y, ∇fv += ◊a(∇a)
9              return ((∇fv,), ∇y)
10         return ↑b, ◊clos
11     ↑c, ◊c = ↑clos(↑x)
12     def ◊f(∇c):
13         ∇x, ∇fv, ∇clos = 0...
14         ∇clos, ∇x += ◊c(∇c)
15         ∇fv, += ∇clos
16         return ((), ∇x, ∇fv)
17     return ↑c, ◊f
```

Figure 2: How a Python function (left) is transformed into a function that can backpropagate a gradient (right). Note that the transform, in Myia, operates on the functional intermediate representation, not on Python code directly. The code on the right is therefore illustrative only. $\uparrow x$ represents the result of transforming x with $J(x)$. ∇x is the gradient with respect to x. $\diamondsuit x$ is a backpropagator function that maps ∇x to the gradients with respect to the function and arguments used to compute x. To gain an intuition about the transform, notice how each line in the original function is mirrored in the transformed version: Each statement in f now returns an additional value (a backpropagator for that statement) in $\uparrow f$. In $\diamondsuit f$, where backpropagation happens, the order of the statements is reversed and they look as if they were turned "inside out" with function and arguments on the left hand side. That illustrates how the backpropagation process starts from the output gradients in order to calculate input gradients. Line 15 shows how to retrieve the gradients with respect to a closure's free variables.

This particular approach satisfies the desire to be able to differentiate any valid program while supporting higher-order differentiation. However, there are still potential issues regarding optimization in a naive implementation:

- The $\overleftarrow{\mathcal{J}}$ operator is dynamic and may require runtime introspection. For example, it is possible to apply $\overleftarrow{\mathcal{J}}$ on itself arbitrarily many times, including a number of times computed by the program itself. However, we do not want to encumber our runtime with a compiler. We must thus limit valid programs to those for which we can identify all possible arguments to $\overleftarrow{\mathcal{J}}$, so that we can compile their transform ahead of time. Fortunately, unbounded self-application of $\overleftarrow{\mathcal{J}}$ is not typical, so we believe this restriction is acceptable.
- $\overleftarrow{\mathcal{J}}$ modifies functions to return two values: the original return value and a backpropagator closure. Likewise, the backpropagator returns multiple values corresponding to gradients with respect to each of its inputs. This is represented as tuples of return values, thus many calls now involve the allocation of data structures. However, as one may observe in figure 2, these tuples are deconstructed as soon as they are returned, so we can solve the issue with a form of copy elision/return value optimization.
- Backpropagation is performed by calling closures that encapsulate the necessary information, but calling a closure is more costly than calling a known function. This being said, in any situation where the target of a call is known in the original program, the targets of the calls produced by the transform are also known: the transform does not add uncertainty in the program, therefore optimizations such as inlining can still be applied.
- The transform produces code to compute a lot of gradients that we have no use for, for example gradients with respect to constant arguments. This is something that inlining and dead code elimination can deal with.

4 Implementation

Myia currently supports a subset of Python that includes the `if`, `while` and `for` statements, nested `def` and `lambda`, simple and mutual recursion, and most arithmetic and logical operators. These can be evaluated by a stack-based virtual machine that supports debugging and tail call elimination. $\overleftarrow{\mathcal{J}}$ has been implemented and tested, including second-order and third-order derivatives on non-trivial loops. A prototype of abstract interpreter can currently infer types and shapes in most situations.

Myia functions can be written alongside general purpose Python code. A `@myia` decorator signifies that the abstract syntax tree (AST) of a function should be parsed and converted into our functional representation. Any auxiliary global function used by a `@myia`-decorated function is automatically compiled alongside it. A high-level interface to $\overleftarrow{\mathcal{J}}$ called `grad` is provided, such that `grad(f)(x, y, ...)` returns the derivative of `f(x, y, ...)` (currently limited to be a scalar) with respect to the first argument `x`.

5 Future work

Myia is currently a functioning prototype, but we see many avenues to improvement and new features.

One approach we are considering is changing our intermediate representation into a graph-based one (cf. sea-of-nodes [4] and Thorin [7]) in order to simplify algebraic optimizations and parallel scheduling. For this purpose, we are currently experimenting with a graph-based representation closely related to A-normal form [5]. We implemented crude inlining, dead code elimination, common subexpression elimination, constant propagation, and a few other simple optimizations on this representation. Preliminary experiments show that on simple programs which involve no conditionals or recursion, we can aggressively apply these optimizations and eliminate all unnecessary tuples and closures. The end result is an efficient program which is close to what a framework such as Theano or TensorFlow would generate. This is an expected but reassuring result. It means that the high flexibility of our framework comes at no cost when considering simple models that are already handled well by existing frameworks.

In future work we also intend on writing a GPU-enabled backend for Myia, and replacing the virtual machine in favor of compiling down to a lower-level language.

Lastly, we are considering approaches to dealing with the overhead of the "copy on write" implementation of `x[i] = y`, which guarantees the immutability of arrays. A simple but efficient optimization is to mutate arrays in place when we can guarantee that the variable will not be read anymore. This technique is used by Theano. We are also considering the use of persistent data structures [8], which is a more general approach in functional programming to efficiently implement immutable data structures.

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. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

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

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

[4] Cliff Click and Keith D. Cooper. Combining analyses, combining optimizations. *ACM Trans. Program. Lang. Syst.*, 17(2):181–196, March 1995.

[5] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, PLDI '93, pages 237–247, New York, NY, USA, 1993. ACM.

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

[7] Roland Leissa, Marcel Koster, and Sebastian Hack. A graph-based higher-order intermediate representation. In *Proceedings of the 13th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '15, pages 202–212, Washington, DC, USA, 2015. IEEE Computer Society.

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

[9] Barak A. Pearlmutter and Jeffrey Mark Siskind. Reverse-mode AD in a functional framework: Lambda the ultimate back propagator. *ACM Trans. Program. Lang. Syst.*, 30:7:1–7:36, March 2008.

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

[11] Theano Development Team. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints*, abs/1605.02688, May 2016.