Don't Unroll Adjoint: Differentiating SSA-Form Programs

Michael J Innes
Julia Computing, Inc.
mike.j.innes@gmail.com

Abstract

This paper presents reverse-mode algorithmic differentiation (AD) based on source code transformation, in particular of the Static Single Assignment (SSA) form used by modern compilers. The approach can support control flow, nesting, mutation, recursion, data structures, higher-order functions, and other language constructs, and the output is given to an existing compiler to produce highly efficient differentiated code. Our implementation is a new AD tool for the Julia language, called Zygote, which presents high-level dynamic semantics while transparently compiling adjoint code under the hood. We discuss the benefits of this approach to both the usability and performance of AD tools.

1 Introduction

Reverse-mode algorithmic differentiation (AD) [19] is at the heart of recent developments in machine learning (ML) and deep learning [2]. ML systems place extreme demands on the tools used to build them; they typically require the highest performance, yet researchers increasingly need the flexibility of a fully differentiable programming language [11].

AD systems face a tradeoff between providing an expressive, full-featured programming model and producing optimised programs [14]. Current ML frameworks use *tracing* approaches to record the numerical operations in the program, which is simple to implement but has significant limitations. Preserving host language semantics (e.g. control flow) requires dynamically building the trace [13], which adds overhead and precludes many optimisations. Saving and compiling traces [3] helps performance at the cost of significantly reduced expressiveness.

We present AD over a Static Single Assignment (SSA) representation of programs in a way that supports control flow, higher-order functions and nested derivatives. The differentiated code can be further fed into a traditional compiler such as LLVM [12], which results in an extremely efficient derivative program. Further, it opens up the opportunity for robust traditional compiler techniques to be extended to machine learning, enabling kernel fusion or compilation for accelerators with no artificial limitations on the kinds of models that researchers can express.

We additionally introduce Zygote, a working implementation of this technique which augments the Julia compiler [4] and is designed for use with the Flux machine learning stack [10]. We discuss Zygote's interaction with Julia's programming model and compiler, and the performance characteristics that result from this combination.

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

2 Tapes & Wengert Lists

2.1 Notation & Background

Given a target program that outputs a scalar l (typically a loss or objective to be minimised), we write the gradient $\partial l/\partial x$ as $\bar{x}$. For uniformity we do not specify the derivatives of component functions like $\sin(x)$ or $a \times b$ directly in the rules of differentiation, but instead treat these as handled via a higher-order differentiation function $\mathcal{J}$. Given a function $y = f(x_1, x_2, ...)$, we write $y, \mathcal{B}_y = \mathcal{J}(f, x_1, x_2, ...)$; $\mathcal{J}$ returns the usual result y as well as a *pullback function* $\mathcal{B}_y$. Then $\bar{x}_1, \bar{x}_2, ... = \mathcal{B}_y(\bar{y})$; the pullback accepts the gradient with respect to y and returns gradients with respect to each input x_i. Pullbacks are linear functions which implement the chain rule for f, as in equation 1, and for mathematical primitives they are easily written down. Some examples are shown in Table 1.

$$\bar{x} = \frac{\partial l}{\partial x} = \frac{\partial l}{\partial y}\frac{\partial y}{\partial x} = \mathcal{B}_y(\bar{y}) \tag{1}$$

This notation has the benefit of treating program subroutines uniformly with mathematical primitives. In the vector case $\partial y/\partial x$ may be a large Jacobian which we wish to avoid instantiating explicitly. Calling $\mathcal{J}$ with a user-defined f can generate an appropriate pullback via some AD technique (such as the one we describe).

Table 1: Pullbacks for some simple mathematical functions.

FUNCTION	PULLBACK
$y = a + b$	$(\bar{y}, \bar{y})$
$y = a \times b$	$(\bar{y} \times b, \bar{y} \times a)$
$y = \sin(x)$	$\bar{y} \times \cos(x)$
$y = \exp(x)$	$\bar{y} \times y$
$y = \log(x)$	$\bar{y}/x$

2.2 Differentiating Wengert Lists

Consider the following mathematical function, which may be part of our target program. We assume that y is further used to calculate l, and that we know $\partial l/\partial y$.

$$y = f(a, b) = \frac{a}{a + b^2}$$

We can rewrite this equivalently by naming each intermediate result.

$$\begin{aligned} y_1 &= b^2 \\ y_2 &= a + y_1 \\ y_3 &= \frac{a}{y_2} \end{aligned}$$

This form can be viewed as a limited programming language; it is often referred to as a Wengert list, tape or graph [1]. The Wengert list is easy to differentiate. First wrap all function calls with J to create a *primal* version of f.

$$\begin{aligned} y_1, \mathcal{B}_1 &= \mathcal{J}(\hat{}, b, 2) \\ y_2, \mathcal{B}_2 &= \mathcal{J}(+, a, y_1) \\ y_3, \mathcal{B}_3 &= \mathcal{J}(/, a, y_2) \end{aligned}$$

Given the gradient $\bar{y}_i$, we can call the pullback $\mathcal{B}_i$ to get gradients for the inputs to y_i. Where a variable x is used multiple times, each corresponding pullback produces a contribution to the gradient (the $\bar{a}_i$ below) which must be summed, as per the multivariable chain rule.

By applying these steps we can begin with the gradient $\bar{y} = 1$ and proceed in reverse over the list to get $\partial y/\partial a$ and $\partial y/\partial b$. This can be realised either by interpreting the Wengert expression in reverse, or by explicitly creating an adjoint expression as follows.

$$\begin{aligned} \bar{y}_3 &= 1 \\ \bar{a}_1, \bar{y}_2 &= \mathcal{B}_3(\bar{y}_3) \\ \bar{a}_2, \bar{y}_1 &= \mathcal{B}_2(\bar{y}_2) \\ \bar{a} &= \bar{a}_1 + \bar{a}_2 \\ \bar{b}, \_ &= \mathcal{B}_1(\bar{y}_1) \end{aligned}$$

Realising this code as a function, with $\bar{y}_3$ as an argument, creates the pullback for f.

3 Static Single Assignment

3.1 Generalising the Wengert List

SSA form [6] generalises the Wengert list with `goto`-based control flow, while preserving the explicit data flow that makes analysis straightforward. The Wengert-list-like code between labels and `goto` instructions is referred to as a *basic block*. Our SSA notation uses numbered variables like $\%1$, $\%2$ etc. and for convenience uses multiple return values, which can be simulated with tuples. The adjoint code also makes use of underlined "alpha nodes" like $\underline{\%1}$ which refer to values from the primal computation – just like primitive pullbacks close over values – and will be generalised in the case of control flow.

Primal code is created much as with simpler Wengert lists. To construct the adjoint, observe that unrolling the adjoint must be equivalent to constructing the adjoint for an unrolled primal. Thus, all basic blocks must be run in reverse order; there is an (iteration of an) adjoint block for each primal one. To achieve this we invert the primal's control flow graph (CFG) and insert dummy ϕ nodes into the primal to record and replay control flow in reverse. After this the basic blocks themselves can be differentiated.

As with the Wengert list, data flow in the adjoint is reversed; a primal SSA definition $\%x$ corresponds to the single usage of the gradient $\overline{\%x}$ with a pullback, and uses of $\%x$ correspond to contributions to the gradient. As SSA definitions dominate their uses, so gradient uses post-dominate their contributions. The complication is that data flow crosses between basic blocks, and a usage of $\%x$ may not actually execute depending on control flow. Thus the adjoint must only take into account gradients that dynamically reach the current block; this can be achieved by propagating gradients in a reversed dataflow analysis of the primal, and inserting zeros and ϕ nodes into the adjoint where necessary. For the purpose of finding reaching gradients of $\%x$, primal ϕ nodes involving $\%x$ can be treated as equivalent to identity($\%x$).

SSA definitions may take on different values in each iteration of a primal block; alpha nodes refer to the value in the corresponding primal iteration. Given the reversed block order the right semantics can be achieved by storing values on a stack, and alpha nodes are then resolved by popping from the stack [8]. This is not the only possible approach; for example, the values could be recomputed (checkpointing), and mixed approaches are able to make time-space tradeoffs [9]. In a reversible neural network [5], the core adjoint transformation remains the same but alpha values will be re-calculated in reverse.

For an example of these rules in practice, consider a simple implementation of x^n for natural n.

```
function pow(x, n)
  r = 1
  while n > 0
    n -= 1
    r *= x
  end
  return r
end
```

The primal code illustrates how loops are represented in SSA form, via ϕ nodes. Both relevant variables, r and n, are explicitly carried between the two blocks comprising the loop.

```
block $\#1$:
        $\%1 \leftarrow \phi(\#0 \to false, \#2 \to true)$
        $\%2 \leftarrow \phi(\#0 \to 1, \#2 \to \%6)$
        $\%3 \leftarrow \phi(\#0 \to n, \#2 \to \%5)$
        $\%4 \leftarrow \%3 > 0$
          goto $\#3$ if not $\%4$
block $\#2$:
        $\%5 \leftarrow \%3 - 1$
    $\%6, \%7 \leftarrow \mathcal{J}(\times, \%2, x)$
          goto $\#1$
block $\#3$:
          return $\%2$
```

In the adjoint code, we again have two ϕ functions in the loop header, effectively tracking $\bar{x}$ ($\%1$) and $\bar{r}$ ($\%2$). Block 1 has two predecessors, block 2 and the implicit block 0 (which corresponds to the return block in the primal). Only r is used in that block (as a return value), so $\bar{x}$ has no gradient contribution and must be initialised to 0. x is used once in each iteration of the loop, so we accumulate $\bar{x}$ across all iterations.[1] Note also the alpha nodes referring to both the recorded branch conditions and the backpropagators for $\times$; these will be further processed into stack operations.

```
block $\#1$:
        $\%1 \leftarrow \phi(\#0 \to 0, \#2 \to \%5)$
        $\%2 \leftarrow \phi(\#0 \to \bar{y}, \#2 \to \%3)$
          goto $\#4$ if not $\underline{\%1}$
block $\#2$:
    $\%3, \%4 \leftarrow \underline{\%7}(\%2)$
        $\%5 \leftarrow \%1 + \%4$
          goto $\#2$
block $\#3$:
          return $\%2, 0$
```

3.2 Handling Language Features

SSA is a very general representation that does not detail much of a language's semantics (e.g. type system, data structures, memory model). Differentiation depends on these details, largely by way of the primitive definitions provided. For example, the IR may not only contain numerical operations, but also many supporting functions such as for modifying state or manipulating data structures, and we need pullbacks for these operations.

The most fundamental data structure is the *cons cell*, a tuple of two values like $C = (x_1, x_2)$. If we call `first(C)` to retrieve the first element we must then find the gradient with respect to C in the adjoint program. We create an *adjoint object* $\bar{C}$, which mirrors the structure of C while storing the gradient of each internal element $(\bar{x}_1, \bar{x}_2)$. Summing adjoint objects sums the elements. The pullbacks for operations on C are given in Table 3.2.

We can now differentiate any function of cons cells. Any other data structure differs only in number of fields or names of accessor functions.

To handle mutation, consider a one-element "box" structure B. We can get(B) to retrieve the current stored

Table 2: Pullbacks for cons cells

FUNCTION	PULLBACK
$C = \text{cons}(x_1, x_2)$	$(\text{first}(\bar{C}), \text{second}(\bar{C}))$
$y = \text{first}(C)$	$\text{cons}(\bar{y}, 0)$
$y = \text{second}(C)$	$\text{cons}(0, \bar{y})$

[1]Seemingly, so also is r. But note each loop iteration sees a *different* definition of r, so the gradients are independent. A benefit of SSA form is that this distinction becomes syntactically clear, and need not be handled specially.

Table 4: Benchmarks on some simple functions.

BENCHMARK	FORWARD	ZYGOTE	PYTORCH	REVERSEDIFF
SINCOS	15.9NS	20.7NS	69,900NS	670NS
LOOP	4.17μS	29.5μS	17,500μS	171μS
LOGSUMEXP	0.96μS	1.26μS	219μS	15.9μS
LOGISTIC REGRESSION	4.67μS	17.6μS	142μS	89.9μS
2-LAYER MNIST MLP	27.7μS	207μS	369μS	N/A

value, and set(B, x) to erase that value and replace it with x. The adjoint object $\bar{B}$ is also a box, which we retrieve via lookup rather than by pullback return values. The pullbacks are given in table 3.

A mutable cons can be seen as a boxed cons or a cons of boxes; in either case it generalises similarly to other mutable data structures. For example, a stack can be implemented as a box containing a cons-based linked list. One caveat: pullbacks frequently close over their inputs (for example, both input arrays in matrix multiplication), and if they are mutated the pullback will be incorrect. Arrays must therefore either be immutable, be copied on capture, or have mutations recorded and reversed during the adjoint program. This is generally *not* true for operations on data structures, so things like stacks need no special support.

Table 3: Pullbacks for boxes

FUNCTION	PULLBACK
$x = \text{get}(B)$	$\text{set}(\bar{B}, \text{get}(\bar{B}) + \bar{x})$
$\text{set}(B, x)$	$(\bar{x} = \text{get}(\bar{B}); \text{set}(\bar{B}, 0); \bar{x})$

Closures are just objects with a `call` method; the fields of the object represent the closure's environment. When calling closures we need to recognise a hidden zeroth argument, the closure itself, and produce an adjoint for that object. In our compiler all functions actually accept this hidden argument—which may be empty as a special case—so both closures and higher-order functions are supported with no extra effort.

Given that adjoint code makes use of both stacks and closures, the above ensures that the AD can consume its own output, thus allowing higher-order derivatives via nested application of $\mathcal{J}$ (as in $\mathcal{J}(\mathcal{J}, f, x)$).

These extensions are enough to support a very general subset of the Julia language, thanks to its simple and very uniform semantics. In other cases (such as when class-based objects or lower-level system routines are used), more may be needed. For example, a matrix multiplication might be expressed either by `A * B` or by `alloc`/`free` and passing of pointers, which is harder to differentiate efficiently. For this reason AD is more effective in high-level compiled languages (e.g. Julia, Swift, Rust, Nim) than traditional ones such as C/C++, Fortran and LLVM IR, even though these can all be expressed as SSA.

4 Results

Zygote is designed to generate adjoint code amenable to Julia's standard compiler analysis and optimisation passes, the most important of which is type inference. Julia's introspection tools can be used to check that generated output is reasonable. For example we show that Julia is able to fully infer the adjoint of a simple neural network, including proving non-differentiability (gradient of type `Nothing`). This works just as well on larger models such as VGG19, and this level of static analysis is what enables us to target TPUs without tracing [7].

After optimisation, the code for `gradient(pow, 2, 3)` is similar to the left below (converted to high-level Julia code for ease of reading). The stacks emitted in adjoint code have low overhead at less than 10 nanoseconds per operation on a typical CPU; this is noticeable compared to scalar numerical operations, but generally negligible in array code. It compares especially favourably to constructing and differentiating a program trace, as in other dynamic AD systems, which has typical overhead in the microseconds per operation [17].

```julia
loss(m, x) = sum(m(x))
m = Chain(Dense(10,5,relu),Dense(5,2))
x = rand(10)
@code_typed(gradient(loss, m, x))
# Tuple{NamedTuple{(:layers,),Tuple{
#   Tuple{NamedTuple{(:w, :b, :f),
#     Tuple{Array{Float64,2},
#           Array{Float64,1},
#           Nothing}},
#   NamedTuple{(:w, :b, :f),
#     Tuple{Array{Float64,2},
#           Array{Float64,1},
#           Nothing}}}
# }},Array{Float64,1}}
```

```julia
function grad_pow(x, n)
  r = 1
  Bs = Tuple{Int,Int}[]
  while n > 0
    push!(Bs, (r, x))
    r *= x
    n -= 1
  end
  dx = 0
  dr = 1
  for i = length(Bs):-1:1
    (r, x) = Bs[i]
    dx += dr*r
    dr = dr*x
  end
  return dx
end
```

To confirm this in more realistic cases, Table 4 provides a set of simple benchmarks between a plain Julia forward pass, Zygote, PyTorch [15] and ReverseDiff [18] (a tracing-based AD with optional compilation). These mix scalar (`sincos` and `loop`) and vector examples to both stress-test AD overhead and show more realistic speedups, respectively.

These results can be compared to the most notable existing source-to-source AD systems, Tapenade [9] and Stalin∇ [16]. Tapenade is capable of producing very fast code that is similarly amenable to the optimisations of existing Fortran and C compilers. However, it operates directly on source files (requiring “caller-derives" usage that prevents libraries from abstracting over differentiation), and lacks generality (its output often needs modification before it can be differentiated again). Meanwhile, Stalin∇ is mathematically general and provides a convenient higher-order-function interface, but only operates on a λ-calculus IR, a non-standard representation that eschews a large body of work on optimising compilers.

Our contribution is thus to provide a best of both worlds: a system that looks to the user like Stalin∇, but to the compiler like Tapenade. Our results confirm that we can reach the quality of hand-written derivatives without modifications to an existing optimising compiler.

5 Conclusion

This paper presents a system for differentiation via the $\mathcal{J}$ function and pullbacks, and uses these to build a system for differentiation via the $\mathcal{J}$ function and pullbacks. Current ML frameworks face a fundamental tradeoff between performance and flexibility, but we hope to have shown that this tradeoff is not fundamental. Our new AD, Zygote, supports a full range of language features—from control flow to macros—while producing highly optimised code.

By transforming SSA-form IR we can differentiate rich and expressive programs with extremely low run-time overhead, while opening up opportunities for even more optimisation in future. As SSA is used as an intermediate representation (IR) by many recent language compilers, differentiation could be added as a first-class language feature to many modern compiled languages, enabling truly differentiable programming.

Acknowledgements

Thanks to Avik Sengupta for many insightful conversations. The manuscript was also much improved through feedback from Simon Byrne, Elliot Saba and Deniz Yuret. The implementation was greatly aided by compiler work done for Julia 1.0 by Jarrett Revels, Keno Fischer, Jameson Nash and Jeff Bezanson. The work owes much to enlightening conversations and inspirational work done by people at Julia Computing and the wider Julia community: all of the above along with Viral Shah, James Bradbury, Tim Besard, Simon Danisch, Lyndon White, Shashi Gowda, and many more.

References

[1] M. Bartholomew-Biggs, S. Brown, B. Christianson, and L. Dixon. Automatic differentiation of algorithms. *Journal of Computational and Applied Mathematics*, 124(1-2):171–190, 2000.

[2] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind. Automatic differentiation in machine learning: a survey. *Journal of machine learning research*, 18(153):1–153, 2017.

[3] J. Bergstra, F. Bastien, O. Breuleux, P. Lamblin, R. Pascanu, O. Delalleau, G. Desjardins, D. Warde-Farley, I. Goodfellow, A. Bergeron, et al. Theano: Deep learning on gpus with python. In *NIPS 2011, BigLearning Workshop, Granada, Spain*, volume 3, pages 1–48. Citeseer, 2011.

[4] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[5] B. Chang, L. Meng, E. Haber, L. Ruthotto, D. Begert, and E. Holtham. Reversible architectures for arbitrarily deep residual neural networks. *arXiv preprint arXiv:1709.03698*, 2017.

[6] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(4):451–490, 1991.

[7] K. Fischer and E. Saba. Automatic full compilation of julia programs and ml models to cloud tpus, 2018.

[8] R. Giering and T. Kaminski. Recipes for adjoint code construction. *ACM Transactions on Mathematical Software (TOMS)*, 24(4):437–474, 1998.

[9] L. Hascoet and V. Pascual. The tapenade automatic differentiation tool: principles, model, and specification. *ACM Transactions on Mathematical Software (TOMS)*, 39(3):20, 2013.

[10] M. Innes. Flux: Elegant machine learning with julia. *Journal of Open Source Software*, 2018.

[11] M. Innes, S. Karpinski, V. Shah, D. Barber, P. Stenetorp, T. Besard, J. Bradbury, V. Churavy, S. Danisch, A. Edelman, et al. On machine learning and programming languages. 2018.

[12] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[13] D. Maclaurin, D. Duvenaud, and R. P. Adams. Autograd: Effortless gradients in numpy. In *ICML 2015 AutoML Workshop*, 2015.

[14] G. Neubig, C. Dyer, Y. Goldberg, A. Matthews, W. Ammar, A. Anastasopoulos, M. Ballesteros, D. Chiang, D. Clothiaux, T. Cohn, et al. Dynet: The dynamic neural network toolkit. *arXiv preprint arXiv:1701.03980*, 2017.

[15] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

[16] B. A. Pearlmutter and J. M. Siskind. Reverse-mode ad in a functional framework: Lambda the ultimate backpropagator. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(2):7, 2008.

[17] PyTorch Team. PyTorch, a, year in... `pytorch.org/blog/a-year-in`, 2018. Accessed: 2018-09-22.

[18] J. Revels. Reversediff.jl. `github.com/JuliaDiff/ReverseDiff.jl`, 2018. Accessed: 2018-09-22.

[19] B. Speelpenning. Compiling fast partial derivatives of functions given by algorithms. Technical report, Illinois Univ., Urbana (USA). Dept. of Computer Science, 1980.