SENSE & SENSITIVITIES: THE PATH TO GENERAL-PURPOSE ALGORITHMIC DIFFERENTIATION

Michael J Innes [1]

ABSTRACT

We present Zygote, an algorithmic differentiation (AD) system for the Julia language. Zygote is designed to address the needs of both the machine learning and scientific computing communities, who have historically been siloed by their very different tools. As well as fostering increased collaboration between these communities, we wish to enable *differentiable programming* (∂P), in which arbitrary numerical programs can make use of gradient-based optimisation. We present and evaluate our proposed solutions to the performance/expressiveness tradeoffs in current systems, as well as our work applying AD to many common programming language features, which is applicable to work in other languages and systems.

1 INTRODUCTION

Algorithmic Differentiation (AD)[1] has a split personality. Its forward (Wengert, 1964) and later reverse (Speelpenning, 1980) modes were first developed for scientific computing, in languages like Fortran. Since then it has proven a sharp tool in the numerical computing toolbox, finding applications to the valuation of contracts in finance, inference in statistical models, fine-tuning of systems in engineering, process optimisation in operations research, state estimation in quantum mechanics, and much more. Supporting this, diverse implementations have flourished in many high-performance languages (Hascoet & Pascual, 2013; Utke et al., 2008; Hogan, 2014; Shiriaev & Griewank, 1996; Griewank et al., 1996).

Alongside this, much recent innovation has come from the machine learning (ML) community, who independently rediscovered the reverse mode as 'backpropagation of errors' (Rumelhart et al., 1988) and built AD systems tailored to their use cases (Bergstra et al., 2010; Maclaurin et al., 2015; Tokui et al., 2015; Chen et al., 2015; Abadi et al., 2016; Neubig et al., 2017; Paszke et al., 2017). The difficulty of writing Fortran and C++ has led ML practitioners to use higher-level languages; they generally write relatively simple programs (network architectures), though are typically more demanding about expressiveness and support for higher order differentiation. Where HPC developers are intolerant of AD overhead, ML practitioners typically vectorise (batch) their programs manually to amortise the high cost of the runtime, which is practical for simple matrix-multiply-based architectures.

Zygote is designed to bring these parallel universes together. It supports high-level and expressive semantics with full support for the control flow and data model of its host language, Julia (Bezanson et al., 2017). Yet a low-overhead source code transform (SCT) based implementation makes it applicable to scalar code. A major consequence is that existing Julia packages now form part of a differentiable library ecosystem, without needing to be rewritten for a given framework. Recent work in ML increasingly incorporates advanced numerical programs, such as physics engines (Degrave et al., 2019), ray tracers (Li et al., 2018) and scientific models (Innes et al., 2019), and we believe there is enormous value in domain experts and ML practitioners sharing code. We refer to the building of these complex end-to-end differentiable systems as Differentiable Programming, or ∂P (Wikipedia contributors, 2019).

1.1 Convergence in AD Design

Operator-overloading and source-to-source approaches to AD have traditionally been distinct, but recent work in the ML community blurs this line. Graph building by operator overloading can be thought of as *partial evaluation*, eliding indirection in the host language (function calls, control flow) and applying a transformation to the resulting trace of numerical operations. But the trace need not be a pure Wengert list, and 'staging' more of the host language's constructs (for example, control flow) into the trace makes this look increasingly like source-to-source AD. Many state-of-the-art ML systems, which originally took very different approaches to AD, are converging towards these *staged pro-*

[1]Julia Computing, Inc., Edinburgh, United Kingdom. Correspondence to: Michael J Innes <mike.j.innes@gmail.com>.

Proceedings of the 3^{rd} MLSys Conference, Austin, TX, USA, 2020.

[1]Also expanded as *automatic* or *analytic differentiation*.

gramming approaches (Agrawal et al., 2019; PyTorch Team, 2018; Frostig et al., 2018).

In a recent system like JAX (Frostig et al., 2018), the Python interpreter can be viewed as taking on the role of compiler. Like Julia's abstract interpreter, it evaluates code with partial information (types) in order to statically resolve polymorphic methods, and then takes a back seat for program runtime. There is an important difference, however. The tracing approach is inherently *lossy* with respect to program semantics (a non-lossy system would simply be a Python compiler). In particular, data dependent control flow, I/O, mutable data structures, and global variables can all lead to errors or surprising semantics. Users must be careful to respect referential transparency, avoiding many useful features of the host language.

Zygote extends the staged approach with support for a broad range of language features. We take a fully-staged compiler IR that preserves all semantics, carry out the derivative transformation (§2.3), add support for Julia's features one by one (§3), and finally apply Julia's optimisation procedures. Being semantically lossless, and avoiding the need for user annotations, is essential for our goal of building a differentiable library ecosystem, since otherwise differentiation would have to be manually enabled and verified correct for each new library.

Our approach and philosophy is closest to that of Tapenade, but with the twist that our source transformation operates on Julia IR "in flight" in the compiler rather than textual source code (§2) (though it is nevertheless purely syntactic, and does not rely on non-local static information such as inferred types). Zygote is also comparable to the Swift for TensorFlow project, but does not require differentiable functions and types to be explicitly annotated, and operates on (lowered) surface syntax rather than typed compiler IR. The elegant recursive formulation of AD was introduced by Stalin∇ (Pearlmutter & Siskind, 2008) and also used in Myia (van Merriënboer et al., 2018), but not generalised beyond a simple λ-calculus language; our work shows that this approach can be married with efficient Tapenade-like handling of control flow, as well as extending its semantics to handle practical language features like mutable data structures.

2 Transforming SSA-form IR

Historically, source code transform (SCT)-based AD systems have worked by parsing source code from files, transforming the abstract syntax tree (AST) and emitting a new source code file. The user can then inspect, modify and compile the derivative code at will. This is both a useful feature and a drawback, given that this *caller-derives* interface requires manual intervention and does not allow libraries to abstract over differentiation.

A compiler-integrated approach instead hides this process, generating and compiling necessary derivative code alongside the original. This has several benefits: gradient code will never be out of sync with the original program, and can be automatically updated even when functions are dynamically re-defined; a *callee-derives* interface means libraries can differentiate user-provided functions without the user being aware of it; and since differentiated code need not be represented textually, it frees us to use a more convenient representation.

Static Single Assignment (SSA) form (Cytron et al., 1991) turns out to be convenient for differentiation, being as expressive as an AST but with a greatly reduced number of features. This section outlines our proposed derivative transform for SSA form code, building on the Wengert list transformation that all reverse-mode ADs use at a minimum.

2.1 Notation & Background

The (partial) derivative of a function $y = f(x)$ is typically written $\frac{\partial y}{\partial x}$. An important special case is when the function output is a scalar l, typically a loss to be minimized. We write this as $\frac{\partial l}{\partial x} = \bar{x}$, known as a *sensitivity* (or, equivalent for our purposes, a *gradient*). Reverse mode AD propagates sensitivities without caring about the details of l, so this notation usefully abstracts over it. In forward mode the equivalent *perturbation* of an intermediate x by some scalar input m is written $\frac{\partial x}{\partial m} = \dot{x}$. This discussion focuses on reverse mode as the technically more difficult case.

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 making no distinction between program subroutines and basic mathematical functions. Indeed, the goal of our AD will be to produce a pullback for the whole program as if it had been built-in to begin with. We can then define differentiation recursively: If during differentiation of f we call $\mathcal{J}(g, ...)$, we can ei-

Table 1. Pullbacks for some simple mathematical functions.

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

ther look up a built-in gradient or generate an appropriate pullback for g via some AD technique (such as Innes 2020).

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, using the arrow $y \leftarrow f(x)$ to assign the name y to the value of $f(x)$ (i.e. a `let` binding).

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

This form can be viewed as a simple programming language; it is often referred to as a Wengert list, tape or graph (Bartholomew-Biggs et al., 2000). The Wengert list is easy to differentiate. First wrap all function calls with J to create a *primal* version of f.

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

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. This is motivated by the multivariable chain rule given in equation 2.

$$\bar{x} = \frac{\partial l}{\partial x} = \frac{\partial l}{\partial y_1}\frac{\partial y_1}{\partial x} + \frac{\partial l}{\partial y_2}\frac{\partial y_2}{\partial x} \tag{2}$$

$$= \mathcal{B}_{y_1}(\bar{y}_1) + \mathcal{B}_{y_2}(\bar{y}_2) \tag{3}$$

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. (The underlined variables were defined in the primal, and the adjoint closes over them.)

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

Realising this code as a function, with $\bar{y}_3$ as an argument, creates the pullback for f. Inlining all function calls yields an efficient symbolic derivative; the $\mathcal{J}$ notation really is just notation.

$$\begin{aligned} y_2 &\leftarrow a + b^2 \\ \bar{y}_2 &\leftarrow -\frac{a}{y_2^2} \\ y &\leftarrow \frac{a}{y_2} \\ \bar{a} &\leftarrow \frac{1}{y_2} + \bar{y}_2 \\ \bar{b} &\leftarrow 2b\bar{y}_2 \end{aligned}$$

This is enough to implement most common AD systems, which use operator overloading to build a Wengert list during program execution (known as a "dynamic graph" in the ML world). If numerical evaluation is interleaved with forward execution and reverse transformation, the adjoint list need not be explicitly realised. However, the reflection and dynamism required is still expensive; to avoid this we must begin to generalise the Wengert list to express more powerful programs.

2.3 Static Single Assignment

SSA form (Cytron et al., 1991) generalises the Wengert list with `goto`-like control flow, while preserving the explicit data flow that makes analysis straightforward. For example, consider a simple branching function:

$$f(x) = \begin{cases} x & x > 0 \\ 0.01x & \text{otherwise} \end{cases}$$

Which we can represent as:

block #1: (x)
  br #2 unless $x > 0$
  br #3 (x)
block #2:
  $y_1 \leftarrow 0.01 \times x$
  br #3 (y_1)
block #3: (y)
  return y

In SSA form the function is split into a series of *basic blocks*, delimited by **block** labels. Each block has a (possibly empty) Wengert list of operations to execute, and ends with a branch to another block (**br**) which may depend on a condition, or else **return**s a value to be used as the output of the function. Blocks are equivalent to a set of mutually-recursive closures (Steele Jr & Sussman, 1976; Appel, 1998; MLIR Contributors, 2019).

Primal code is created much as before, with the addition that dummy arguments are added to blocks to record control flow. In this case block 3 has two predecessors, and b tells us which of the two to branch to in the adjoint.

block #1: (x)
  br #2 unless $x > 0$
  br #3 $(x, 1)$
block #2:
  $y_1, \mathcal{B}_{y_1} \leftarrow \mathcal{J}(\times, 0.01, x)$
  br #3 $(y_1, 2)$
block #3: (y, b)
  return y

An important concept is the control flow graph (CFG). Each block is a vertex, with directed edges for each possible branch between blocks. To generate an adjoint for our IR, we begin with the CFG. The adjoint CFG is the transpose of the primal CFG, having all edges reversed. To make this intuitive, consider unrolling a given execution of the function into a Wengert list; the reversed Wengert list must effectively run each (instruction of every) block in reverse order. Thus each time block A branches to block B in the primal, block $\bar{B}$ much branch to $\bar{A}$ in the adjoint.

Creating the adjoint follows the rules for Wengert lists as above, with the addition of differentiating branches. If A passes arguments (x, y) to B, then $\bar{B}$ should pass $(\bar{x}, \bar{y})$ to $\bar{A}$, just as if the branch were a function call. However, since A may branch to multiple different blocks, passing different arguments, $\bar{A}$'s argument list must include all gradients it may need. For example, if A also branches to C with arguments (y, z), A's argument list must be $(\bar{x}, \bar{y}, \bar{z})$. $\bar{B}$ will pass $(\bar{x}, \bar{y}, 0)$ and $\bar{C}$ will pass $(0, \bar{y}, \bar{z})$.[2]

block #1: $(\bar{y})$
  br #3 $(\bar{y})$ unless $\underline{b} \neq 1$
  br #2
block #2:
  $,\bar{x}_1 \leftarrow \underline{\mathcal{B}_{y_1}}(\bar{y})$
  br #3 $(\bar{x}_1)$
block #3: $(\bar{x})$
  return $\bar{x}$

As in primitive pullbacks, the adjoint code closes over variables from the primal (the underlined variables). Note, however, that since blocks can execute multiple times, these variables actually refer to a set of values, one for each execution of the given block. The primal can be augmented to record these values on stacks, which the adjoint then pops when the variable is used, so that each run of an adjoint block sees the value of the corresponding primal definition. 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 (Hascoet & Pascual, 2013). In a reversible programming model, such as reversible neural networks (Chang et al., 2017), the core adjoint transformation remains the same but primal values can be re-calculated in reverse, and a combination of approximate reversal and checkpointing is used for differentiation of ODEs (Rackauckas et al., 2018).

For a more complex example of these rules in practice we take a simple calculation of x^n, represented in Julia code as:

```
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

[2] In SSA it is valid to use a variable defined by a previous block (so long as the definition dominates the usage). We avoid special handling for these variables turning them into explicit block arguments, so that all variables in a block are locally defined.

form, with mutable variables like r split into immutable r_i and explicitly carried across loop iterations.

block #1: (x, n)
  br #2 $(n, 1, 1)$
block #2: (n_1, r_1, b)
  br #4 unless $n_1 > 0$
block #3:
  $n_2 \leftarrow n_1 - 1$
  $r_2, \mathcal{B}_{r_2} \leftarrow \mathcal{J}(\times, r_1, x)$
  br #2 $(n_2, r_2, 2)$
block #4:
  return r_1

The adjoint code is similarly a loop that computes $\bar{r}$ and $\bar{x}$ (note that $\bar{x}_i$ is the i^{th} variable representing $\bar{x}$, not the gradient of x_i). x is used once in each iteration of the loop, so we accumulate $\bar{x}$ across all iterations.[3]

block #1: $(\bar{y})$
  br #2 $(\bar{y}, 0)$
block #2: $(\bar{r}_1, \bar{x}_1)$
  br #4 unless $\underline{b} \neq 1$
  br #3
block #3:
  $\bar{r}_2, \bar{x}_2 \leftarrow \underline{\mathcal{B}_{r_2}}(\bar{r}_1)$
  $\bar{x}_3 \leftarrow \bar{x}_1 + \bar{x}_2$
  br #2 $(\bar{r}_2, \bar{x}_3)$
block #4:
  return $\bar{x}_1$

Storing pullbacks (rather than raw primal values) allows us to handle dynamic code where the definition of the function f is not known until runtime. We need not sacrifice performance for this; where f can be statically resolved, the closure type tags can be elided and the pullback definition inlined so that only values are stored contiguously on the stack, behaving at runtime similarly to Tapenade.

3 Differentiation Semantics

Zygote's core transform is simple and mechanical, and only around 200 lines of code. It is worth emphasising that almost all of Zygote's semantics and functionality are provided via its library of custom adjoints (that is, manual overrides of the $\mathcal{J}$ function), which encompass both core mathematical definitions and support for data structures and mutation, new numerical and mathematical types, hardware specialisations, and mixed-mode AD, typically expressed in only a few lines of code.

Where practical this section shows Julia code for the implementation, rather than a mathematical abstraction, to demonstrate how these features look in practice. All of them work currently; most are supported out of the box in the core Zygote library, while fixed-point iterative computations and support for cross-language AD have working prototypes outside of the package.

3.1 Custom Adjoints

Manually defining gradients is a crucial part of Zygote's interface, and not just for supplying primitives: users are encouraged to use custom adjoints to build entirely new features, and we show some examples of their somewhat surprising expressive power.

Gradient hooks allow an arbitrary function to be applied to the gradient, for example `hook(-, x)` to reverse the sign of $\bar{x}$. There are many uses for this, including gradient clipping and debugging.

```
hook(f, x) = x
@adjoint hook(f, x) =
  (x, dx -> (nothing, f(dx)))
```

The function `nestlevel` is able to do reflection on the gradient process itself; if called within a differentiated function it will return the order of differentiation being performed.

```
nestlevel() = 0
@adjoint nestlevel() =
  (nestlevel()+1, _ -> ())
```

A simple implementation of checkpointing is similarly straightforward.

```
checkpoint(f, x) = f(x)
@adjoint checkpoint(f, x) =
  (f(x), dy -> J(f, x)[2](dy))
```

The remaining functionality in this section is similarly provided by custom adjoints.

3.2 Data Structures & Mutation

Although Julia's data model is fairly complex, we can define differentiation of data structures by starting with a simple tuple like $C = (x_1, x_2)$. If we call $\text{first}(C)$ to retrieve the

[3] 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.

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 as follows.

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

Any other (immutable) struct differs only in number of fields or names of accessor functions, making it straightforward to generalise this.

To handle mutation, consider a one-element "box" structure B. We can $\text{get}(B)$ to retrieve the current stored value, and $\text{set}(B, x)$ 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; a global lookup is necessary to handle the non-local dataflow that mutation introduces. The pullbacks are as follows.

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

A mutable struct can be seen as a boxed tuple or a tuple 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 tuple-based linked list. In general we will want to use more efficient data structures (e.g. stacks in contiguous memory or hash maps), but this formalism allows us to easily derive appropriate specialised pullbacks for them.

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 other data structures (which do not get captured), so things like stacks need no special support.

Closures are just structs with a `call` method (c2 Wiki Contributors, 2018); the parts of the struct represent the closure's environment. When calling closures we need to recognise a hidden zeroth argument, the closure environment, 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 $\mathcal{J}(\mathcal{J}, f, x)$).

3.3 Concurrency and Parallelism

Julia supports a concurrency model based on communicating sequential processes (CSP, Hoare 1978). A zero-argument function or closure (a thunk) can be scheduled as a *task* (or *coroutine*), and executed independently of the main thread. Tasks communicate with each other through shared queues called *channels*. Typically, the main thread will create a series of tasks and wait for them all to finish before continuing.

Zygote makes CSP differentiable by the following transformation. Firstly, when a task is scheduled, its thunk f is replaced by $\mathcal{J}(f)$, producing a pullback. Once the task is complete, we associate it with an adjoint task which will run the pullback. During the reverse pass, we reach the point where the original task was awaited in the primal code, and schedule the adjoint task. The adjoint task executes and communicates with other adjoint tasks as needed, finally producing a gradient of the thunk $\bar{f}$.

Channels can be differentiated as in §3.2; for each channel c we create an empty adjoint channel $\bar{c}$. Sending a value to c becomes receiving a sensitivity from $\bar{c}$ and vice versa.

Julia supports shared-memory parallelism by multiplexing tasks onto OS threads, so support for tasks means that multithreaded code is also differentiable. Julia uses the same concepts, though a slightly different API, for distributed / multi-node parellelism, so the same techniques can be straightforwardly transferred to differentiation of distributed code. In an experimental setting we were able to achieve a $1.5\times$ speedup when using two cores to get the gradient of a simple function using map-reduce parallelism.

Care must be taken that accumulate/reset operations in the adjoint are atomic, since there may otherwise be a race condition due to multiple reads from the same array location in the primal, or due to tasks sharing mutable state. Differentiation of parallel code at other levels of abstraction, such as the level of parallel `for` loops or map-reduce, presents different challenges and opportunities (Hückelheim et al., 2019; Hovland, 1997; Naumann et al., 2008; Bücker et al., 2001).

3.4 Mixed-Mode AD

While reverse mode AD is a powerful tool, especially in optimisation problems, there are many alternative ways to calculate derivatives, and specialised approaches can have advantages in many situations. For example, Julia's forward-mode AD (Revels et al., 2016) has constant memory overhead (compared to reverse mode's tape, linear in the number of instructions executed) and has minimal time overhead,

making it ideal for long-running computations with a small number of inputs. Similarly, TaylorSeries.jl (Benet et al., 2018) can calculate arbitrary-order forward-mode derivatives in one shot.

Mixed mode is exposed by writing `forwarddiff(f, x)`. This calculates the same result as $f(x)$, but additionally calculates the Jacobian via forward mode, stores it, and applies it during the backwards pass using a custom adjoint. Similarly, checkpointed AD is exposed via `checkpoint(f, x)` (§3.1). Zygote can be instructed to always use forward mode (or another AD technique) on a given function, or even to have heuristics for the best method, so that for users of a library, differentiation is efficient by default.

There are many more ways to exploit problem structure in AD. In nested optimisation problems, for example, assuming convergence of the inner solver makes some of its gradients analytical zeros, and avoids differentiation of the solver operations. As another example, consider evaluating an infinite Taylor series; in practice only a finite number of terms are considered, up to numerical precision. By default, each differentiation of this finite series will drop a term, reducing precision. However, one can define a custom adjoint that derives the Taylor series itself analytically, which is then evaluated to the same precision as the original function. This trick can be generalised to a "fixed-point iteration" operator which has an appropriate adjoint defined (Schlenkrich et al., 2008). Similar concerns come up when differentiating domain specific languages (DSLs), such as Halide (Li et al., 2018); even if the DSL compiles to differentiable Julia code, it is easier to exploit performance and numerical optimisations by differentiating at the highest level of abstraction available.

A similar problem is differentiating code in other languages, for example Python code invoked via PyCall.jl (Johnson et al., 2018). In this case, we can write an adjoint for the low-level `pycall` function which invokes a Python AD, capturing its tape in a pullback. To a user, calling imported Python functions inside a call to `gradient` then works transparently.

3.5 Complex Differentiation

Complex numbers are not a special case at the AD level, but instead are treated as any other user-defined type. Zygote's pre-defined rules for numerical operations (e.g. multiplication and addition) immediately generalize to the complex numbers, and only rules for the `real` and `imag` functions are needed in addition for full complex support.

Zygote defines the sensitivity of a complex number $z = x + yi$ by $\bar{z} = \bar{x} + \bar{y}i$. This definition is useful for gradient descent since for small, real η, $f(z + \eta\bar{z}) \approx f(z) + \eta\bar{z}z^*$, and thus the usual gradient update $z := z - \eta\bar{z}$ lowers the loss. (This is equivalent to differentiating a pair of two reals (x, y).)

This sensitivity is not the true complex derivative $\frac{\partial}{\partial z} = \frac{\partial}{\partial x} + \frac{\partial}{\partial iy} = \frac{\partial}{\partial x} - i\frac{\partial}{\partial y}$, which (for holomorphic functions) will satisfy $f(z+\epsilon) \approx f(z)+\frac{\partial f}{\partial z}\epsilon$. By the Cauchy-Riemann equations, $\frac{\partial f}{\partial z}$ is conjugate to the sensitivity $\bar{z}$ of $\Re f(z)$ making it straightforward and efficient to calculate. In the more general non-holomorphic case one needs either the equivalent 2×2 real Jacobian or the two Wirtinger derivatives $(\frac{\partial f}{\partial z}, \frac{\partial f}{\partial z^*})$, both of which are readily derived from the sensitivities of $\Re f(z)$ and $\Im f(z)$.

This generalises straightforwardly to $\mathbb{C}^m \to \mathbb{C}^n$ functions, meaning Zygote can be used for general complex differentiation.

3.6 Staged Programming

Zygote does not require users to manually specify which code should be staged and optimised – that is the job of a compiler – but nor does it prevent users from explicitly staging computation. In fact, Julia is an excellent tool for staged- and meta-programming techniques, and Zygote simply works with this as any other language feature.

For example, many numerical libraries provide an `einsum` interface, allowing tensor operations to be expressed with a syntax based on Einstein notation. The syntax is usually expressed as a string and, in dynamic interfaces like PyTorch, parsing the string incurs an overhead each time the expression is run. While Julia provides a dynamic interface, we don't have to pay this cost: Einsum can be implemented as a *macro*, explicitly parsing the notation at compile time and leaving only raw tensor operations behind. Zygote sees only the final matrix multiply and sum operations, so this has no overhead compared to writing them manually. The same is true of Julia's other powerful metaprogramming and staging tools, such as generated functions.

3.7 Hardware Backends

Zygote transforms generic programs and mathematical expressions – written in terms of mathematical operators like $\times$, $+$ etc. – into new generic programs that calculate a gradient. Thus Zygote is completely agnostic to the data types running through the program and how they are implemented or represented in memory. A Zygote program written for floating point numbers therefore works equally well with rational numbers, arbitrary-precision floats and integers, measurements, hardware-specific types like `BFloat16`, and combinations of these.

```
julia> gradient(x -> x^2 + 3x + 1, 1/3)
(3.6666666666666665,)
```

```
julia> gradient(x -> x^2 + 3x + 1, 1//3)
(11//3,)

julia> gradient(x -> x^2 + 3x + 1,
                1/3 ± 0.01)
(3.6666666666666665 ± 0.02,)
```

The same is true for arrays; the program `gradient(x -> σ.(W*x .+ b), x)` works equally well whether W, b and x are dense arrays, sparse arrays, arrays backed by GPU memory, or distributed arrays stored over a cluster of hundreds of nodes. The operations $\times$, broadcasting and so on are called on the adjoint arrays and thus launched on the GPU or cluster as appropriate.

Julia's TPU support, in XLA.jl (Fischer & Saba, 2018), takes advantage of this composability. Rather than being tied to a particular AD implementation, as in current TPU frontends, XLA.jl compiles general Julia code to the TPU. When the program being compiled happens to call `gradient`, ML happens.

3.8 External Libraries

Support for types and libraries distinguishes frameworks from programming languages, and we support these in differentiable programming too. For example, the Colors.jl package (Holy et al., 2018) provides representations of RGB colours (among many other colour spaces), and functions over these colour spaces can be differentiated. By default, each field of a structure is differentiated independently, as in §3.2.

```
julia> a = RGB(1, 0, 0);

julia> gradient(a -> a.r^2, a)
((r = 2.0f0, g = nothing, b = nothing),)
```

Colors.jl also provides many useful procedures, such as for computing perceptual colour differences (Luo et al., 2001). These can also be differentiated and even used as loss functions in machine learning models.

```
julia> b = RGB(0, 1, 0);

julia> colordiff(a, b)
86.60823557376344

julia> gradient(b -> colordiff(a, b), b)
((r = -1.77, g = 28.88, b = -0.04),)
```

We emphasise that `colordiff` comprises hundreds of lines of code including types, dispatch, control flow, table lookups, and other language features. It was written before Julia had any AD support, but nevertheless has not needed modifications in order to be safely differentiable. This is only possible with Zygote's semantics-preserving approach to AD.

Aside from the correctness benefits of working with types, it is increasingly recognised that incorporating existing knowledge and code into machine learning leads to richer and more powerful models; this is particularly valuable in scientific computing, where powerful explicit models exist for many systems that need not be learned from scratch (Innes et al., 2019).

3.9 Deep Learning

Zygote is not, in itself, a deep learning library. Deep learning tools and interfaces – such as for common architectures like `LSTM` (Gers et al., 1999) and gradient descent rules like `ADAM` (Kingma & Ba, 2014) – are provided by higher-level libraries like Flux (Innes, 2018; Innes et al., 2018). Nevertheless, many standard things are simple with Zygote alone. Getting the gradients $(\bar{W}, \bar{b})$ for a logistic regression is a one-liner:

```
gradient(
  (W, b) -> loss(σ.(W*x .+ b), y), W, b)
```

More complex architectures differ only in the details of the forward pass, which can include loops and recursion, and reuse common patterns and layers from libraries. A hand-written LSTM looks as follows:

```
for (x, y) in (xs, ys)
  forget = σ.(Wf*x + Uf*h + bf)
  input  = σ.(Wi*x + Ui*h + bi)
  output = σ.(Wo*x + Uo*h + bo)
  cell′ = tanh.(Wc*x + Uc*h + bc)
  cell = forget .* cell + input .* cell′
  h = o .* tanh.(cell)
  loss += distance(h, y)
end
```

More complex models with many parameters can be handled by bundling the weights into structures (as in autograd (Maclaurin et al., 2015), §3.2). We can then get the gradient of a model m (which is made callable with an input x to invoke the forward pass) as follows.

```
gradient(m -> distance(m(x), y), m)
```

The model m is equivalent to a *closure*, where closed-over variables are trainable weights. In Flux we refer to this kind of closure as a "layer", since they can be composed together just as in a high level library like Keras (Chollet et al., 2015), which effectively implements function combinators. However, they can also be freely mixed with more "imperative" code, as in this homebrew maxout layer (Goodfellow et al., 2013).

```
m1 = Chain(Dense(10, 5, relu), Dense(5, 2))
m2 = Chain(Dense(10, 5, relu), Dense(5, 2))
model = x -> softmax(max.(m1(x), m2(x)))
```

We suggest that this kind of layer is a fundamental unit of abstraction in differentiable programming (much as procedures, objects and functions are in their respective paradigms) and thus has relevance well beyond neural networks.

3.10 Higher-Order Derivatives

Zygote naturally and intuitively supports higher-order derivatives, by differentiating a program that calls `gradient`.

```
gradient(x -> gradient(sin, x)[1], π/2)
(-1.0,)
```

However, we note that alongside the first-order performance benefits of SCT ADs like Tapenade, Zygote also inherits some of their pitfalls with respect to higher-order differentiation. In particular, the differentiation transform tends to double the size of the original code, leading to exponential code in the order of differentiation, and correspondingly large compile times (tens of seconds for third-order derivatives); this is mitigated by the interpreted approach taken by many tracing ADs. We believe the most promising overall solution will be to interleave SCT differentiation with optimisation (particularly dead code elimination, algebraic simplification and common subexpression elimination), as implemented by Myia (van Merriënboer et al., 2018).

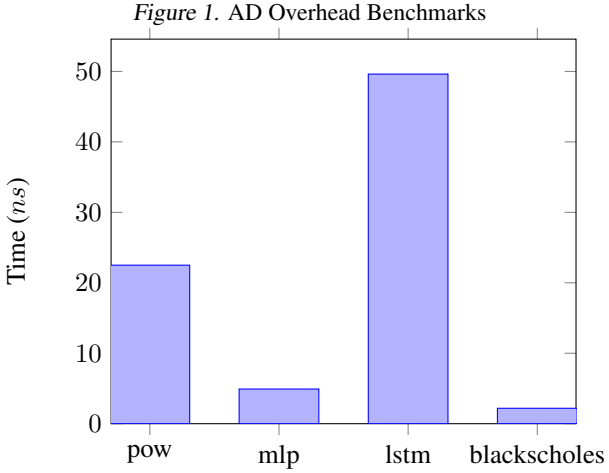
Zygote can also differentiate, or be differentiated by, other ADs in Julia, similar to §3.4. In many cases this is preferable to nested reverse mode; for example, for Hessians, forward-over-reverse better exploits the numerical and runtime properties of forward and reverse mode AD.

4 Performance Characteristics

To evaluate performance we measure AD overhead; that is, the average time spent manipulating AD data structures rather than doing essential numerical work when evaluating a primitive operation (such as addition of two tensors). This metric gives an approximate threshold at which AD becomes the bottleneck in execution time, rather than the numerical workload. For example, overhead of $1\mu s$ is acceptable when running a large ResNet model on a GPU, where kernel launch times are in the microsecond range and large convolution operations take far longer. Conversely, scalar operations lie in the nanosecond range and $1\mu s$ overhead would contribute orders of magnitude to overall execution time.

We measure overhead by replacing all array operations with

Figure 1. AD Overhead Benchmarks

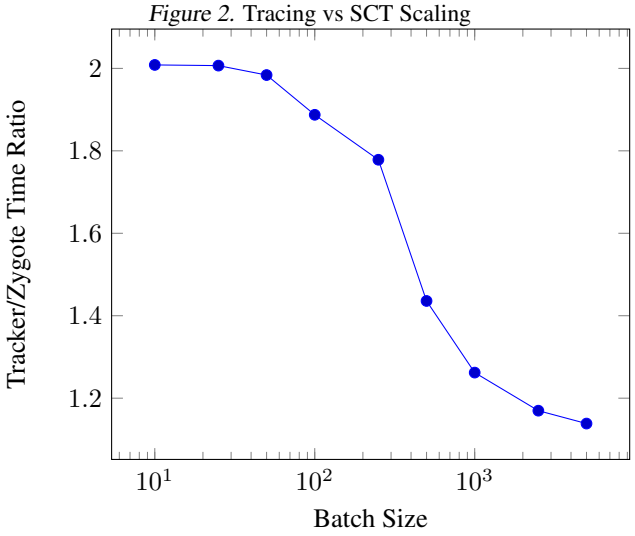


altered versions that do no work, other than computing shapes. All computation time then comes from overhead, either from the Julia runtime or Zygote, and this is averaged over the number of primitive operations (custom adjoints) in the computation. (In the case of the `pow` benchmark, we assume that the cost of the scalar numerical workload is negligible, making the overhead estimate conservative.) The results are shown in figure 1, and found to be on the order of $50ns$ or less. Overhead is primarily caused by (a) differentiation producing code that is harder for the Julia compiler to infer, resulting in worse optimisation and (b) the management of Zygote's heap-allocated stacks. Compiler improvements and stack pre-allocation strategies are planned to reduce this overhead even further, as well as to make optimisations more reliable in the presence of Julia's heuristic-based compiler.

To demonstrate the impact of AD overhead, we calculate gradients for a multi-layer perceptron using both a tracing AD, Tracker.jl, and Zygote, measuring the ratio of wall clock time taken over a range of batch sizes. Because Tracker's overhead is comparable to GPU kernel launch overhead, we expect it to take about twice as long at small batch sizes, which is indeed what we find to be the case (figure 2). As batch size grows, AD overhead is amortised, resulting in similar performance for both systems. Benchmarks were conducted on a 3.6GHz Intel i7-7700 with an NVIDIA GTX 1080 GPU.

While Julia's overall suitability for any given AD use case (deep learning, probabilistic programming, computational fluid dynamics, finance ...) depends on domain-specific factors such as implementation quality of numerical kernels, these benchmarks show Zygote's suitability to build such a system in combination with other tools, and we note that in many domains common kernels are shared between systems (e.g. BLAS, or CUDNN in deep learning), making AD a bottleneck by default.

Figure 2. Tracing vs SCT Scaling



5 Conclusion

We have presented Zygote, a tool for analytic differentiation of code in the Julia language. Zygote consolidates the best ideas from the many AD tools that came before it, aiming to create a unified interface that meets the needs of as many applications as possible. We have shown how this synthesis of flexibility and high performance can be achieved, especially the simplicity that can be found when using powerful tools from the programming language and compilers communities. Building on work by the AD community, we have also shown how SCT AD can be applied to a wide range of language features, from closures to concurrency.

We believe that there is huge potential at the intersection of machine learning and other fields, but current ML frameworks require that domain specific code be rewritten, greatly slowing experimentation. Julia is the first platform to make existing numerical libraries differentiable by default, enabling code sharing and reuse between domain experts and ML researchers in a growing differentiable library ecosystem.

References

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. Tensorflow: a system for large-scale machine learning. In *OSDI*, volume 16, pp. 265–283, 2016.

Agrawal, A., Modi, A. N., Passos, A., Lavoie, A., Agarwal, A., Shankar, A., Ganichev, I., Levenberg, J., Hong, M., Monga, R., et al. Tensorflow eager: A multi-stage, python-embedded dsl for machine learning. *arXiv preprint arXiv:1903.01855*, 2019.

Appel, A. W. SSA is functional programming. *ACM SIGPLAN Notices*, 33(4):17–20, 1998.

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

Benet, L., Sanders, D., et al. TaylorSeries.jl. https://github.com/JuliaDiff/TaylorSeries.jl, 2018. Accessed: 2018-09-22.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. Theano: a cpu and gpu math expression compiler. In *Proceedings of the Python for scientific computing conference (SciPy)*, volume 4. Austin, TX, 2010.

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

Bücker, H. M., Lang, B., Bischof, C. H., et al. Bringing together automatic differentiation and openmp. In *Proceedings of the 15th international conference on Supercomputing*, pp. 246–251. ACM, 2001.

c2 Wiki Contributors. Closures and objects are equivalent. wiki.c2.com/?ClosuresAndObjectsAreEquivalent, 2018. Accessed: 2018-09-22.

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

Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274*, 2015.

Chollet, F. et al. Keras, 2015.

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

Degrave, J., Hermans, M., Dambre, J., et al. A differentiable physics engine for deep learning in robotics. *Frontiers in neurorobotics*, 13, 2019.

Fischer, K. and Saba, E. Automatic full compilation of julia programs and ml models to cloud tpus. *arXiv preprint arXiv:1810.09868*, 2018.

Frostig, R., Johnson, M. J., and Leary, C. Compiling machine learning programs via high-level tracing, 2018.

Gers, F. A., Schmidhuber, J., and Cummins, F. Learning to forget: Continual prediction with lstm. 1999.

Goodfellow, I. J., Warde-Farley, D., Mirza, M., Courville, A., and Bengio, Y. Maxout networks. *arXiv preprint arXiv:1302.4389*, 2013.

Griewank, A., Juedes, D., and Utke, J. Algorithm 755: Adol-c: a package for the automatic differentiation of algorithms written in c/c++. *ACM Transactions on Mathematical Software (TOMS)*, 22(2):131–167, 1996.

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

Hoare, C. A. R. Communicating sequential processes. In *The origin of concurrent programming*, pp. 413–443. Springer, 1978.

Hogan, R. J. Fast reverse-mode automatic differentiation using expression templates in c++. *ACM Transactions on Mathematical Software (TOMS)*, 40(4):26, 2014.

Holy, T., Jones, D. C., and contributors. Colors.jl. github.com/JuliaGraphics/Colors.jl, 2018. Accessed: 2018-09-22.

Hovland, P. D. *Automatic differentiation of parallel programs*. Number 2003. University of Illinois at Urbana-Champaign Champaign, IL, USA, 1997.

Hückelheim, J., Hovland, P., Strout, M. M., and Müller, J.-D. Reverse-mode algorithmic differentiation of an openmp-parallel compressible flow solver. *The International Journal of High Performance Computing Applications*, 33(1): 140–154, 2019.

Innes, M. Flux: Elegant machine learning with julia. *J. Open Source Software*, 3(25):602, 2018.

Innes, M., Saba, E., Fischer, K., Gandhi, D., Rudilosso, M. C., Joy, N. M., Karmali, T., Singh, A. P., and Shah, V. Fashionable modelling with flux. *arXiv preprint arXiv:1811.01457*, 2018.

Innes, M., Edelman, A., Fischer, K., Rackauckas, C., Saba, E., Shah, V. B., and Tebbutt, W. A differentiable programming system to bridge machine learning and scientific computing. *CoRR*, abs/1907.07587, 2019. URL http://arxiv.org/abs/1907.07587.

Innes, M. J. Sense sensitivities: The path to general-purpose algorithmic differentiation, 2020.

Johnson, S. G. et al. PyCall.jl. https://github.com/JuliaPy/PyCall.jl, 2018. Accessed: 2018-09-22.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Li, T.-M., Gharbi, M., Adams, A., Durand, F., and Ragan-Kelley, J. Differentiable programming for image processing and deep learning in halide. *ACM Transactions on Graphics (TOG)*, 37(4):139, 2018.

Luo, M. R., Cui, G., and Rigg, B. The development of the cie 2000 colour-difference formula: Ciede2000. *Color Research & Application: Endorsed by Inter-Society Color Council, The Colour Group (Great Britain), Canadian Society for Color, Color Science Association of Japan, Dutch Society for the Study of Color, The Swedish Colour Centre Foundation, Colour Society of Australia, Centre Français de la Couleur*, 26(5):340–350, 2001.

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

MLIR Contributors. Block arguments vs phi nodes, 2019. URL https://github.com/tensorflow/mlir/blob/master/g3doc/Rationale.md#block-arguments-vs-phi-nodes. Accessed 15-August-2019.

Naumann, U., Hascoët, L., Hill, C., Hovland, P., Riehme, J., and Utke, J. A framework for proving correctness of adjoint message-passing programs. In *European Parallel Virtual Machine/Message Passing Interface Users' Group Meeting*, pp. 316–321. Springer, 2008.

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

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

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

PyTorch Team. TorchScript. pytorch.org/docs/stable/jit.html, 2018. Accessed: 2018-09-22.

Rackauckas, C., Ma, Y., Dixit, V., Guo, X., Innes, M., Revels, J., Nyberg, J., and Ivaturi, V. A comparison of automatic differentiation and continuous sensitivity analysis for derivatives of differential equation solutions. *arXiv preprint arXiv:1812.01892*, 2018.

Revels, J., Lubin, M., and Papamarkou, T. Forward-mode automatic differentiation in julia. *arXiv preprint arXiv:1607.07892*, 2016.

Rumelhart, D. E., Hinton, G. E., Williams, R. J., et al. Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1, 1988.

Schlenkrich, S., Walther, A., Gauger, N. R., and Heinrich, R. Differentiating fixed point iterations with adol-c: Gradient calculation for fluid dynamics. In *Modeling, Simulation and Optimization of Complex Processes*, pp. 499–508. Springer, 2008.

Shiriaev, D. and Griewank, A. Adol-f: Automatic differentiation of fortran codes. *Computational Differentiation: Techniques, Applications, and Tools*, pp. 375–384, 1996.

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

Steele Jr, G. L. and Sussman, G. J. Lambda: The ultimate imperative. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE ARTIFICIAL INTELLIGENCE LAB, 1976.

Tokui, S., Oono, K., Hido, S., and Clayton, J. Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, volume 5, pp. 1–6, 2015.

Utke, J., Naumann, U., Fagan, M., Tallent, N., Strout, M., Heimbach, P., Hill, C., and Wunsch, C. Openad/f: A modular open-source tool for automatic differentiation of fortran codes. *ACM Transactions on Mathematical Software (TOMS)*, 34(4):18, 2008.

van Merriënboer, B., Breuleux, O., Bergeron, A., and Lamblin, P. Automatic differentiation in ml: Where we are and where we should be going. In *Advances in Neural Information Processing Systems*, pp. 8770–8780, 2018.

Wengert, R. E. A simple automatic derivative evaluation program. *Communications of the ACM*, 7(8):463–464, 1964.

Wikipedia contributors. Differentiable programming, 2019. URL https://en.wikipedia.org/wiki/Differentiable_programming. Accessed 15-August-2019.