Demystifying Differentiable Programming: Shift/Reset the Penultimate Backpropagator

Fei Wang, Daniel Zheng, James M. Decker, Xilun Wu, Grégory Essertel, Tiark Rompf
Purdue University

Abstract

Deep learning has seen tremendous success over the past decade in computer vision, machine translation, and gameplay. This success rests crucially on *gradient-descent optimization* and the ability to "learn" parameters of a neural network by backpropagating observed errors. However, neural network architectures are growing increasingly sophisticated and diverse, which motivates an emerging quest for even more general forms of *differentiable programming*, where arbitrary parameterized computations can be trained by gradient descent. In this paper, we take a fresh look at automatic differentiation (AD) techniques, and especially aim to demystify the *reverse-mode* form of AD that generalizes backpropagation in neural networks.

We uncover a tight connection between reverse-mode AD and delimited continuations, which permits implementing reverse-mode AD purely via operator overloading and without managing any auxiliary data structures. We further show how this formulation of AD can be fruitfully combined with multi-stage programming (staging), leading to an efficient implementation that combines the performance benefits of deep learning frameworks based on explicit reified computation graphs (e.g., TensorFlow) with the expressiveness of pure library approaches (e.g., PyTorch).

1 Introduction

Under the label *deep learning*, artificial neural networks have seen a remarkable renaissance over the last decade. After a series of rapid advances, they now match or surpass human performance in computer vision, machine translation, and gameplay. Common to all these breakthroughs is the underlying dependency on optimization by gradient descent: a neural network "learns" by adjusting its parameters in a direction that minimizes the observed error on a task. Hence, a crucial ability is that of backpropagating errors through the network to compute the gradient of a loss function [60]. Beyond this commonality, however, deep learning architectures vary widely. In fact, many of the practical successes are fueled by increasingly sophisticated and diverse network architectures that in many cases depart from the traditional organization into layers of artificial neurons. For this reason, prominent deep learning researchers have called for a paradigm shift from deep learning towards *differentiable programming* [37, 46] — essentially, functional programming with a first-class gradient operator — based on the expectation that further advances in artificial intelligence will be enabled by the ability to "train" arbitrary parameterized computations by gradient descent.

Programming language designers, key players in this vision, are faced with the challenge of adding efficient and expressive program differentiation capabilities. Forms of automatic gradient computation that generalize the classic backpropagation algorithm are provided by all contemporary deep learning frameworks, including TensorFlow and PyTorch. These implementations, however, are ad-hoc, and each framework comes with its own set of trade-offs and restrictions. In the academic world, automatic differentiation (AD) [64, 74] is the subject of study of an entire community. Unfortunately, results disseminate only slowly between communities, and while the forward-mode flavor of AD is easy to grasp, descriptions of the reverse-mode flavor that generalizes backpropagation often appear mysterious to PL researchers. A notable exception is the seminal work of Pearlmutter and Siskind [53], which has cast AD in a functional programming framework and laid the groundwork for first-class, unrestricted, gradient operators in a functional language. Recent work by Elliott [25] presented a unification of forward and reverse AD based on the "compiling to categories" approach [24], translating Haskell code to parameterized cartesian closed categories. However, a key limitation of this approach is that the extracted computations are straight dependency graphs that do not model control flow. Hence, loops, functions, etc. need to be partially evaluated as part of the translation, potentially leading to code explosion or nontermination. It also takes a certain mindset to appreciate a category-heavy explanation as "the simple essence", as Elliott [25] aspires to deliver.

The goal of the present work is to further demystify differentiable programming and reverse-mode AD for a PL audience, and to reconstruct the forward- and reverse-mode AD approaches based on well-understood program transformation techniques and without appealing to category theory. We describe forward-mode AD as symbolic differentiation of ANF-transformed programs, and reverse-mode AD as a specific form of symbolic differentiation of CPS-transformed programs. In doing so, we uncover a deep connection between reverse-mode AD and delimited continuations.

In contrast to previous descriptions, this formulation suggests a novel view of reverse-mode AD as a purely local

Preprint, November 2018.

program transformation which can be realized entirely using operator overloading in a language that supports `shift`/`reset` [18] or equivalent delimited control operators[1]. By contrast, previous descriptions require non-local program transformations to carefully manage auxiliary data structures (often called a *tape*, *trace*, or *Wengert-list* [74]), either represented explicitly, or in a refunctionalized form as in Pearlmutter and Siskind [53].

Delimited control operators lead to an expressive implementation in the style of PyTorch (define-by-run). We further show how to combine this approach with *multi-stage programming* to derive a framework in the style of TensorFlow (define-then-run). The result is a highly efficient and expressive DSL, dubbed Lantern, that reifies computation graphs at runtime in the style of TensorFlow [1], but also supports unrestricted control flow in the style of PyTorch [51]. Thus, our approach combines the strengths of these systems without their respective weaknesses, and explains the essence of deep learning frameworks as the combination of two well-understood and orthogonal ideas: staging and delimited continuations.

The rest of this paper is organized around our contributions as follows:

- We derive forward-mode AD from high-school symbolic differentiation rules in an effort to ground the discussion and provide accessibility to PL researchers at large (Section 2).
- We then present our reverse-mode AD transformation based on delimited continuations and contrast it with existing methods. This results in a PyTorch-style define-by-run framework (Section 3).
- We combine our reverse-mode AD implementation in an orthogonal way with staging, removing interpretive overhead from differentiation. This results in a TensorFlow-style define-then-run framework, but one more expressive than TensorFlow (Section 4).
- We present Lantern, a deep learning DSL implemented using these techniques, and evaluate it on real-world models, including convolutional and recurrent neural networks, and tree-recursive networks (Section 5).

Finally, Section 6 discusses related work, and Section 7 offers concluding thoughts.

2 Differentiable Programming Basics

Broadly speaking, a neural network is a special kind of parameterized function approximator $\hat{f}_w$. The training process optimizes the parameters w to improve the approximation of an uncomputable *ground truth* function f based on training data.

$$f : A \to B \qquad \hat{f}_w : A \to B \qquad w \in P$$

For training, we take input/output samples $(a, f(a)) \in A \times B$ and update w according to a *learning rule*. In typical cases where the functions f and $\hat{f}_w$ are maps $\mathbb{R}^n \to \mathbb{R}^m$ and w is in the form of $\mathbb{R}^k$, we want to find the weights w that achieve the smallest error or loss $L(w) = \left\| f(a) - \hat{f}_w(a) \right\|$ on a given training set, in the hope that the training set is representative enough that the quality of the approximation of $\hat{f}_w$ will generalize to other inputs of f.

While there exist many ways to update w, the most popular method is gradient descent. This is largely due to the fact that gradients can be computed efficiently even for extremely large numbers of parameters. We briefly describe gradient descent, as follows:

Given a training sample $(a, f(a)) \in A \times B$ and some initialization of w at w^i, both the loss $L(w^i)$ and the gradient[2] $\nabla L(w^i)$ can be computed: The gradient marks the direction which increases the loss $L(w^i)$ most rapidly, and the gradient descent algorithm dictates that w should be updated in the direction of the negative gradient by a small step defined by the *learning rate* r.

$$w^{i+1} = w^i - r * \nabla L(w^i)$$

This update step is performed many times. In practice, however, gradient descent is almost never used in this pure form. Most commonly used are *stochastic gradient descent* (SGD) flavors that operate on batches of training samples at a time. Popular variants are SGD with momentum [54], Adagrad [22], and Adam [35].

An important property is that differentiability is compositional. Traditional neural networks (i.e., those organized into layers) have a simple function composition $\hat{f}_w = \hat{f}_{n,w_n} \circ \ldots \circ \hat{f}_{1,w_1}$ where each $\hat{f}_{i,w_i}$ represents a layer. Other architectures similarly compose and enable end-to-end training. A popular example is image captioning, which composes convolutional neural networks (CNN) [39] and recurrent neural networks (RNN) [26].

Imagine, however, that $\hat{f}_w$ and by extension $L(w)$ is not just a simple sequence of function composition, but is instead defined by a *program*, e.g., a λ-term with complex control flow. How, then, should $\nabla L(w)$ be computed?

2.1 From Symbolic Differentiation to Forward-Mode AD

Symbolic differentiation techniques to obtain the derivative of an expression are taught in high schools around the world. Some of the well-known rules are shown in Figure 1 (we will explain the one dealing with let expressions shortly).

[1]Our description reinforces the functional “Lambda, the ultimate backpropagator” view of Pearlmutter and Siskind [53] with an alternative encoding based on delimited continuations, where control operators like `shift`/`reset` act as a powerful front-end over λ-terms in CPS — hence, as the “penultimate backpropagator”.

[2]The gradient ∇f of a function $f : \mathbb{R}^n \to \mathbb{R}$ is defined as the vector of partial derivatives of f with respect to each of its parameters: $\nabla f(u) = \left(\frac{\partial f(u)}{\partial u_1}, \frac{\partial f(u)}{\partial u_2}, \ldots, \frac{\partial f(u)}{\partial u_n}\right)$

Syntax:

$$e ::= c \mid x \mid e + e \mid e * e \mid \text{let } x = e \text{ in } e$$

Symbolic differentiation rules:

$$\begin{array}{rcl}
{}^{d}\!/_{dx}\,[\![c]\!] & = & 0 \\
{}^{d}\!/_{dx}\,[\![x]\!] & = & 1 \\
{}^{d}\!/_{dx}\,[\![e_1 + e_2]\!] & = & {}^{d}\!/_{dx}\,[\![e_1]\!] + {}^{d}\!/_{dx}\,[\![e_2]\!] \\
{}^{d}\!/_{dx}\,[\![e_1 * e_2]\!] & = & {}^{d}\!/_{dx}\,[\![e_1]\!] * e_2 + e_1 * {}^{d}\!/_{dx}\,[\![e_2]\!] \\
{}^{d}\!/_{dx}\,[\![\text{let } y = e_1 \text{ in } e_2]\!] & = & \text{let } y\ = e_1 \text{ in} \\
 & & \text{let } y' = {}^{d}\!/_{dx}\,[\![e_1]\!] \text{ in} \\
 & & {}^{d}\!/_{dx}\,[\![e_2]\!] \\
{}^{d}\!/_{dx}\,[\![y]\!] & = & y' \quad (y \neq x)
\end{array}$$

Figure 1. Symbolic differentiation for a simple expression language, extended with let expressions.

As such, symbolic differentiation is a candidate for a first approach to compute derivatives of program expressions. However, it has problems: some differentiation rules may cause code explosion, not only in size, but also in terms of computation cost. Consider the following example:

$$\begin{array}{rcll}
{}^{d}\!/_{dx}\,[\![e_1 * e_2 * \ldots * e_n]\!] & = & {}^{d}\!/_{dx}\,[\![e_1]\!] * e_2 * \ldots * e_n & + \\
 & & e_1 * {}^{d}\!/_{dx}\,[\![e_2]\!] * \ldots * e_n & + \\
 & & \ldots & + \\
 & & e_1 * e_2 * \ldots * {}^{d}\!/_{dx}\,[\![e_n]\!] &
\end{array}$$

The size-n term on the left-hand side is transformed into n size-n terms, which is a non-linear increase. Worse, each e_i is now evaluated n times.

This problem is well recognized in the AD space and often cited as a major motivation for more efficient approaches. In fact, many AD papers go to great lengths to explain that "AD is not symbolic differentiation" [6, 53]. However, let us consider what happens if we convert the program to administrative normal form (ANF) [27] first, binding each intermediate result in a let expression:

$$\begin{array}{ll}
{}^{d}\!/_{dx}\,[\![& \text{let } y_1,\ y_2,\ \ldots,\ y_n = e_1,\ e_2,\ \ldots,\ e_n \text{ in} \\
 & \text{let } z_1 = y_1 * y_2 \text{ in} \\
 & \text{let } z_2 = z_1 * y_3 \text{ in} \\
 & \ldots \\
 & \text{let } z_{n-1} = z_{n-2} * y_n \text{ in} \\
 & z_{n-1}\ ]\!]
\end{array}$$

$$\begin{array}{lll}
= & \text{let } y_1 = e_1 & \text{in let } y_1' = {}^{d}\!/_{dx}\,[\![e_1]\!] \text{ in} \\
 & \ldots & \\
 & \text{let } y_n = e_n & \text{in let } y_n' = {}^{d}\!/_{dx}\,[\![e_n]\!] \text{ in} \\
 & \text{let } z_1 = y_1 * y_2 & \text{in let } z_1' = y_1' * y_2 + y_1 * y_2' \text{ in} \\
 & \text{let } z_2 = z_1 * y_3 & \text{in let } z_2' = z_1' * y_3 + z_1 * y_3' \text{ in} \\
 & \ldots & \\
 & \text{let } z_{n-1} = z_{n-2} * y_n & \text{in let } z_{n-1}' = z_{n-2}' * y_n + z_{n-2} * y_n' \\
 & \text{in } z_{n-1}' &
\end{array}$$

After ANF-conversion, the expression size increases only by a constant factor. The program structure remains intact, and just acquires an additional let binding for each existing binding. No expression is evaluated more often than in the original computation.

This example uses the standard symbolic differentiation rules for addition and multiplication, but also makes key use of the let rule in Figure 1, which splits a binding let $y = \ldots$ into let $y = \ldots$ and let $y' = \ldots$. Using terminology from the AD community, we call y the *primal* and y' the *tangent*. The rules in Figure 1 work with respect to a fixed x, which we assume by convention does not occur bound in any let $x = \ldots$ expression. All expressions are of type $\mathbb{R}$, so a derivative can be computed for any expression. We write ${}^{d}\!/_{dx}\,[\![e]\!]$ using bracket syntax to emphasize that symbolic differentiation is a syntactic transformation.

For straight-line programs, applying ANF conversion followed by symbolic differentiation achieves exactly the same result as standard presentations of forward-mode AD. Hence, it seems to us that the AD community has taken a too narrow view of symbolic differentiation, excluding the possibility of let-bindings, and we believe that repeating the mantra "AD is not symbolic differentiation" is ultimately harmful and contributes to the mystical appearance of the field. We believe that understanding sophisticated AD algorithms as *specific forms* of symbolic differentiation will overall lead to a better understanding of these techniques.

2.2 Forward-Mode AD for Lambda Terms

We now proceed beyond straight-line programs and enrich our grammar with lambdas and applications in Figure 2. A consequence of this is the need to distinguish number-typed expressions from function-typed expressions, since it is only possible to differentiate with respect to numeric expressions. We define a new differentiation operator $\overrightarrow{\mathcal{D}_x}[\![e^\tau]\!]$, where the arrow indicates forward-mode and where τ is the type associated with the expression e. We omit the τ for a cleaner presentation if τ is not explicitly used. We use the same notation to transform variables in argument positions, and to explain how types are transformed. The key strategy for numeric values is to always pair the primal value with its tangent ($\overrightarrow{\mathcal{D}_x}[\![\mathbb{R}]\!] = \mathbb{R} \times \mathbb{R}$), even for function arguments and results. This generalizes the paired let-bindings from Figure 1. Note that differentiation is still with respect to a fixed x. Compared to the previous section, we no longer rely on an ANF-pre-transform pass. Instead, the rules for addition and multiplication insert let-bindings directly. It is important to note that the resulting program may not be in ANF due to nested let-bindings, but code duplication is still eliminated thanks to the strict pairing of primals and tangents. Readers acquainted with forward-mode AD will note that this methodology is standard [6], though the presentation is not.

2.3 Implementation using Operator Overloading

Pairing the primal and tangent values for numeric expressions is quite convenient, because when dealing with function application, the let-insertion needs both the primal and tangent of the parameter to perform the tangent computation. Since the transformation is purely local, working with pairs of numeric expressions makes it immediately clear that

$$e ::= \ldots \mid \lambda x.\, e \mid e\ e$$
$$\tau ::= \mathbb{R} \mid \tau \to \tau$$

$$\overrightarrow{\mathcal{D}_x}[\![\mathbb{R}]\!] = \mathbb{R} \times \mathbb{R}$$
$$\overrightarrow{\mathcal{D}_x}[\![\tau_1 \to \tau_2]\!] = \overrightarrow{\mathcal{D}_x}[\![\tau_1]\!] \to \overrightarrow{\mathcal{D}_x}[\![\tau_2]\!]$$

$$\overrightarrow{\mathcal{D}_x}[\![c^{\mathbb{R}}]\!] = (c, 0)$$
$$\overrightarrow{\mathcal{D}_x}[\![x^{\mathbb{R}}]\!] = (x, 1)$$
$$\overrightarrow{\mathcal{D}_x}[\![y^{\mathbb{R}}]\!] = (y, y')$$
$$\overrightarrow{\mathcal{D}_x}[\![y^{\tau \neq \mathbb{R}}]\!] = y$$
$$\begin{aligned}\overrightarrow{\mathcal{D}_x}[\![e_1 + e_2]\!] = \\ &\text{let } (a, a') = \overrightarrow{\mathcal{D}_x}[\![e_1]\!] \text{ in} \\ &\text{let } (b, b') = \overrightarrow{\mathcal{D}_x}[\![e_2]\!] \text{ in} \\ &(a + b, a' + b')\end{aligned}$$

$$\begin{aligned}\overrightarrow{\mathcal{D}_x}[\![e_1 * e_2]\!] = \\ &\text{let } (a, a') = \overrightarrow{\mathcal{D}_x}[\![e_1]\!] \text{ in} \\ &\text{let } (b, b') = \overrightarrow{\mathcal{D}_x}[\![e_2]\!] \text{ in} \\ &(a * b, a * b' + a' * b)\end{aligned}$$
$$\overrightarrow{\mathcal{D}_x}[\![\lambda y.\, e]\!] = \lambda \overrightarrow{\mathcal{D}_x}[\![y]\!].\ \overrightarrow{\mathcal{D}_x}[\![e]\!]$$
$$\overrightarrow{\mathcal{D}_x}[\![e_1\ e_2]\!] = \overrightarrow{\mathcal{D}_x}[\![e_1]\!]\ \overrightarrow{\mathcal{D}_x}[\![e_2]\!]$$
$$\begin{aligned}\overrightarrow{\mathcal{D}_x}[\![\text{let } y = e_1 \text{ in } e_2]\!] = \\ &\text{let } \overrightarrow{\mathcal{D}_x}[\![y]\!] = \overrightarrow{\mathcal{D}_x}[\![e_1]\!] \text{ in } \overrightarrow{\mathcal{D}_x}[\![e_2]\!]\end{aligned}$$

Figure 2. Formal grammar of enriched expression language (top), forward-mode AD transform of the expressions in the enriched formal grammar (bottom).

this strategy can be implemented easily in standard programming languages by overloading operators. This is standard practice, which we illustrate through our implementation in Scala (Figure 3).

```
// Differentiable number type.
class NumF(val x: Double, val d: Double) {
  def +(that: NumF) = new NumF(this.x + that.x, this.d + that.d)
  def *(that: NumF) =
    new NumF(this.x * that.x, this.d * that.x + that.d * this.x)
  ... }
// Differentiation operator.
def grad(f: NumF => NumF)(x: Double) = {
  val y = f(new NumF(x, 1.0)); y.d }
// Example and test.
val df = grad(x => 2*x + x*x*x)
forAll { x => df(x) == 2 + 3*x*x }
```

Figure 3. Forward-mode AD in Scala (operator overloading)

The `NumF` class encapsulates the primal as `x` and tangent as `d`, with math operators overloaded to compute primal and tangent values at the same time. To use the forward-mode AD implementation, we still need to define an operator `grad` to compute the derivative of any function `NumF => NumF` (Figure 3, middle). Internally, `grad` invokes its argument function with a tangent value of 1 and returns the tangent field of the function result. In line with the previous sections, we only handle scalar functions, but the approach generalizes to multidimensional functions as well. An example using the `grad` operator is shown in Figure 3, bottom. Note that the constant 2 is implicitly converted to `new NumF(2.0, 0.0)` (tangents of constants are 0.0 because constants do not change). The use of `Double` instead of a generic number type is simply for clarity of presentation.

2.4 Nested Gradient Invocation and Perturbation Confusion

In the current implementation, we can compute the gradient of any function of type `NumF => NumF` with respect to any given value using forward-mode AD. However, our `grad` function is not truly first-class, since we cannot apply it in a nested fashion, as in `grad(grad(f))`. This prevents us from computing higher order derivatives, and from solving nested min/max problems in the form of:

$$\min_x \max_y f(x, y)$$

Yet, even this somewhat restricted operator has a few subtleties. There is a common issue with functional implementations of AD that, like ours, expose a gradient operator within the language. In the simple example shown below, the inner call to `grad` should return 1, meaning that the outer `grad` should also return 1.

```
grad { x: NumF =>
  // Evaluates to 2 instead of 1! Unexpected.
  val shouldBeOne = grad(y => x + y)(1)
  x * NumF(shouldBeOne, 0)
}(1)
```

However, this is not what happens. The inner `grad` function will also collect the tangent from `x`, thus returning 2 as the gradient of `y`. The outer `grad` will then give a result of 2 as gradient of `x`. This issue is called *perturbation confusion* because the `grad` function is confusing the perturbation (i.e. derivative) of a free variable used within the closure with the perturbation of its own parameter.

The root of this problem is that the two `grad` invocations differentiate with respect to different variables (outer `grad` wrt. `x`, inner `grad` wrt. `y`); their gradient updates should not be mixed. We do not provide any new solution for perturbation confusion, but our implementation can be easily extended to support known solutions, either based on *dynamic tagging*, or based on types as realized in Haskell[3], which lifts tags into the type system using rank-2 polymorphism, just like the `ST` monad [36].

2.5 First-Class Gradient Operator

While not the main focus of our work, we outline one way in which our `NumF` definition can be changed to support first-class gradient computation, while preventing perturbation confusion. Inspired by DiffSharp [7], we change the class signatures as shown below. We unify `NumF` and `Double` in the same abstract class `Num`, and add a dynamic tag value `tag`. The `grad` operator needs to assign a new tag for each invocation, and overloaded operators need to take tags into account to avoid confusing different ongoing invocations of `grad`.

```
abstract class Num
class NumV(val x: Double) extends Num
class NumF(val x: Num, val d: Num, val tag: Int) extends Num {...}
def grad(f: Num => Num)(x: Num): Num = {...}
```

Alternative implementations that use parametric types and type classes instead of OO-style inheritance are also possible.

This concludes the core ideas of forward-mode AD. Implementations based on operator overloading are simple and direct, and exist in many languages. As noted earlier, we propose that forward-mode AD be viewed as a specific kind of symbolic differentiation, either using standard differentiation rules after ANF-conversion, or using transformation rules that insert let-bindings on the fly, operating on value-derivative pairs (i.e. primals and tangents).

[3]http://conway.rutgers.edu/~ccshan/wiki/blog/posts/Differentiation

3 Differentiable Programming with Reverse-Mode AD

Forward-mode AD is straightforward to implement and generalizes to functions with multiple inputs and outputs. However, it is inefficient for functions with many inputs, and neural networks generally have many inputs and few outputs. To compute the gradient of a function $f : \mathbb{R}^n \to \mathbb{R}$, we have to compute n forward derivatives either sequentially or simultaneously, but this leads to $O(n)$ more operations than the original function. Is there a better approach?

We consider again $f : \mathbb{R}^n \to \mathbb{R}$ represented as a straight-line program in ANF, i.e., as a sequence of $\texttt{let}\ y_j = e_j$ expressions, with inputs x_i and output y_m. The basic intuition is: instead of computing all $n * m$ internal derivatives ${}^{d}\!/_{dx_i}\ y_j$ as in forward-mode, we would rather only compute the $m + n$ derivatives ${}^{d}\!/_{dy_j}\ y_m$ and ${}^{d}\!/_{dx_i}\ y_m$. For this, we need a way to compute derivatives starting with ${}^{d}\!/_{dy_m}\ y_m = 1$, and accumulate derivatives backwards through the program until we reach the inputs x_i. This form of AD is called reverse-mode AD, and is the basis for backpropagation for neural networks. The approach generalizes to functions $\mathbb{R}^n \to \mathbb{R}^m$ with multiple outputs, and is generally more efficient than forward-mode AD when $n >> m$.

The distinct feature of reverse-mode AD is that it has two phases: the forward pass computes all primal values, and the backward pass computes all gradients (also called *adjoints* or *sensitivities* in the AD community). This idea is illustrated by the example of computing x^4 via $y = x * x, z = y * y$ below.

Forward pass:	Backward pass:
$\texttt{let}\ y = x * x$	$\texttt{let}\ z' = {}^{d}\!/_{dz}\ z = 1.0$
$\texttt{let}\ z = y * y$	$\texttt{let}\ y' = {}^{d}\!/_{dy}\ z = 2 * y$
	$\texttt{let}\ x' = {}^{d}\!/_{dx}\ z = {}^{d}\!/_{dx}\ y * {}^{d}\!/_{dy}\ z$
	$= 2 * x * y' = 4 * x^3$

The backward propagation depends crucially on the *chain rule of differentiation* (below), which is used in the last line to compose ${}^{d}\!/_{dx}\ y$ with ${}^{d}\!/_{dy}\ z$ for ${}^{d}\!/_{dx}\ z$.

$$
\begin{aligned}
{}^{d}\!/_{dx}\ f(g(x)) \quad = \quad & \text{let } y\ = g(x) \text{ in} \\
& \text{let } y' = {}^{d}\!/_{dx}\ g(x) \text{ in} \\
& {}^{d}\!/_{dx}\ y * {}^{d}\!/_{dy}\ f(y)
\end{aligned}
$$

A more general presentation of reverse-mode AD is given in Figure 4. There are several differences compared with forward-mode AD. For one, since an intermediate result may be used in multiple calculation steps, and one backward-pass step can only capture the partial gradient from the associated calculation, the gradients are typically stored in mutable variables and modified by relevant steps in the backward pass. We call this *destination-passing style*, where the reference cells accumulating the gradients are passed to the operations in the backward pass. (An alternative pure functional implementation for straight-line programs is discussed in Section 3.4.) In addition to this, reverse-mode AD also must remember values from the forward pass – whether by additional data structures or other means – to support partial derivative computation in the backward pass.

Forward pass:	Backward pass:
$\texttt{let}\ x_i\ \texttt{be inputs}$	$y'_n\ \mathrel{+}= 1$
$\texttt{let}\ x'_i = \text{ref } 0$	$p'_{n2} \mathrel{+}= {}^{d}\!/_{dp_{n2}}\ [\![p_{n1} \oplus p_{n2}]\!] * !y'_n$
$\texttt{let}\ p_{tk} \in \{x_i\} \cup \{y_j \mid j < t\}$	$p'_{n1} \mathrel{+}= {}^{d}\!/_{dp_{n1}}\ [\![p_{n1} \oplus p_{n2}]\!] * !y'_n$
$\texttt{let}\ y_1 = p_{11} \oplus p_{12}$	$\dots$
$\texttt{let}\ y'_1 = \text{ref } 0$	$\dots$
$\texttt{let}\ y_2 = p_{21} \oplus p_{22}$	
$\texttt{let}\ y'_2 = \text{ref } 0$	$p'_{22} \mathrel{+}= {}^{d}\!/_{dp_{22}}\ [\![p_{21} \oplus p_{22}]\!] * !y'_2$
$\dots$	$p'_{21} \mathrel{+}= {}^{d}\!/_{dp_{21}}\ [\![p_{21} \oplus p_{22}]\!] * !y'_2$
$\texttt{let}\ y_n = p_{n1} \oplus p_{n2}$	$p'_{12} \mathrel{+}= {}^{d}\!/_{dp_{12}}\ [\![p_{11} \oplus p_{12}]\!] * !y'_1$
$\texttt{let}\ y'_n = \text{ref } 0$	$p'_{11} \mathrel{+}= {}^{d}\!/_{dp_{11}}\ [\![p_{11} \oplus p_{12}]\!] * !y'_1$

Figure 4. Reverse-mode AD: general pattern for a straight-line program in ANF. The left column is the forward pass extended with allocating mutable variables to hold the adjoints. The double indices of p_{tk} indicates the kth parameter in tth equation. The adjoints are successively updated by the backward pass on the right. Note that "ref" and "!" are mutable reference allocation and dereference operators.

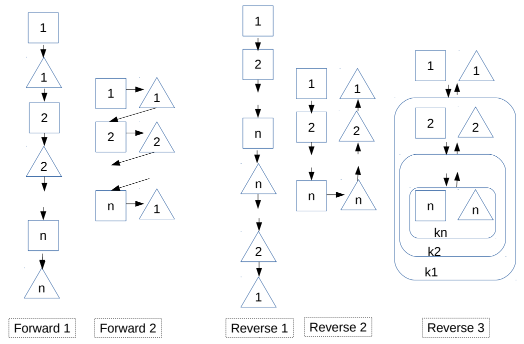

Figure 5. The flow of computation for forward-mode and reverse-mode AD. Forward 1 and Reverse 1 show the basic computation flow, where squares represent value computations and triangles represent gradient computations. Forward 2 handles value/gradient computations in pairs via operator overloading (as in Figure 3). The intermediate idea of Reverse 2 is harder to implement directly, but inspires the form of Reverse 3, where overloaded operators handle value/gradient computations, and relay following computations to continuations. k_1, k_2, k_n represent nested invocations of *delimited continuations* in reverse-mode AD.

3.1 Reverse-Mode AD using Continuations

The computation flows of forward-mode and reverse-mode are illustrated in Figure 5. In forward-mode, the AD algorithm interleaves value calculations *in squares* and gradient calculations *in triangles* (Forward 1). It is easy to pair the neighboring value calculations and gradient calculations (Forward 2) so that the forward-mode can be achieved via operator overloading. In reverse-mode, the computation has to finish value calculations first, then carry out gradient calculations in reverse order. It is tempting to simply "fold" the gradient calculations up in parallel with the value calculations, to achieve a similar looking presentation (Reverse 2). Though we cannot directly implement the Reverse 2 mode since the computation flows both down and up, we take inspiration from "There and Back again" [20] and look for ways to model the computation as a sequence of function calls, where the call path implements the forward pass and the return path implements the backward path.

With this intuition, it is not hard to see that a transformation to continuation-passing style (CPS) provides exactly the right structure, i.e., for each computation step, the following forward pass and the leading backward pass are contained in a continuation (as k_1, k_2, k_n in Reverse 3, Figure 5). We show the central rules for a continuation-based transformation in Figure 6, based on a λ-calculus extended with reals, tuples, and references. As in forward AD, we assume typed expressions e^τ, but we often drop τ for convenience. Our reverse AD transformation pairs each numeric value with a mutable reference cell that holds its adjoint, which is updated by the respective numeric operators. In contrast to the standard CPS rules (Figure 6 above), the continuation parameter k in the reverse AD rule $\overleftarrow{\mathcal{D}_x}\llbracket e \rrbracket k$ is a *delimited continuation*, since the continuation returns to the caller *before* the update of adjoints. The Reverse 3 mode in Figure 5 depicts this scoped model.

3.2 Implementation Using Operator Overloading

Our first implementation in Scala is mechanical, directly following the rules in Figure 6. It is shown in Figure 7. Just like in forward-mode AD, we associate values and their gradients closely as two fields of a class, here `NumR`. Every operator takes a delimited continuation `k`, which is expected to take the intermediate variable `y`, handle the rest of the forward pass after this computation step, as well as the leading part of the backward pass before this step. Once the continuation returns, the gradients (`y.d` and possibly other gradients in the closure) should have been correctly updated, and the operator then updates the gradients of the dependent variables using side-effects.

However, it is still cumbersome to use this implementation. For a simple function like $y = 2 * x + x * x * x$, we have to explicitly construct delimited continuations for each step (bottom shaded box in Figure 7). Fortunately, there exist *delimited control operators* that enable programming with delimited continuations in a direct style, without making continuations explicit.

Standard CPS transformation (excerpt):

$$
\begin{array}{rcl}
\llbracket x \rrbracket k & = & k(x) \\
\llbracket e_1 * e_2 \rrbracket k & = & \llbracket e_1 \rrbracket(\lambda y_1 . \llbracket e_2 \rrbracket(\lambda y_2 . k(y_1 * y_2))) \\
\llbracket \lambda x . e \rrbracket k & = & k(\lambda x . \lambda \kappa \llbracket e \rrbracket \kappa) \\
\llbracket e_1\ e_2 \rrbracket k & = & \llbracket e_1 \rrbracket(\lambda y_1 . \llbracket e_2 \rrbracket(\lambda y_2 . (y_1\ y_2) k))
\end{array}
$$

Reverse AD with CPS transformation:

Syntax:

$$
\begin{array}{rcl}
e & ::= & c \mid x \mid e + e \mid e * e \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
 & \mid & \lambda x .\ e \mid e\ e \mid \texttt{fst}\ e \mid \texttt{snd}\ e \mid \texttt{ref}\ e \mid !e \\
 & \mid & e := e \mid (e,\ e)
\end{array}
$$

Type:

$$
\tau \ ::= \ \texttt{Unit} \mid \mathbb{R} \mid \texttt{Ref}\ \tau \mid \tau * \tau \mid \tau \to \tau
$$

Variable sugaring:

$$
\begin{array}{rcl}
\ddot{y} & = & (y,\ y')^{\mathbb{R} * (\texttt{Ref}\ \mathbb{R})} \\
 & \mid & y^{\tau \neq \mathbb{R}}
\end{array}
$$

CPS Transformation:

$$
\begin{array}{rcl}
\overleftarrow{\mathcal{D}_x}\llbracket c^{\tau \neq \mathbb{R}} \rrbracket k & = & k(c) \\
\overleftarrow{\mathcal{D}_x}\llbracket c^{\mathbb{R}} \rrbracket k & = & k((c,\ \texttt{ref}\ 0)) \\
\overleftarrow{\mathcal{D}_x}\llbracket x \rrbracket k & = & k((x,\ x')) \\
\overleftarrow{\mathcal{D}_x}\llbracket y \rrbracket k & = & k(\ddot{y}) \\
\overleftarrow{\mathcal{D}_x}\llbracket e_1 + e_2 \rrbracket k & = & \overleftarrow{\mathcal{D}_x}\llbracket e_1 \rrbracket(\lambda(\ddot{y}_1) . \overleftarrow{\mathcal{D}_x}\llbracket e_2 \rrbracket(\lambda(\ddot{y}_2) . \\
 & & \texttt{let}\ \ddot{y} = (y_1 + y_2,\ \texttt{ref}\ 0)\ \texttt{in} \\
 & & k(\ddot{y}) \\
 & & y_1' \mathrel{+}= !y' \\
 & & y_2' \mathrel{+}= !y')) \\
\overleftarrow{\mathcal{D}_x}\llbracket e_1 * e_2 \rrbracket k & = & \overleftarrow{\mathcal{D}_x}\llbracket e_1 \rrbracket(\lambda(\ddot{y}_1) . \overleftarrow{\mathcal{D}_x}\llbracket e_2 \rrbracket(\lambda(\ddot{y}_2) . \\
 & & \texttt{let}\ \ddot{y} = (y_1 * y_2,\ \texttt{ref}\ 0)\ \texttt{in} \\
 & & k(\ddot{y}) \\
 & & y_1' \mathrel{+}= y_2 * !y' \\
 & & y_2' \mathrel{+}= y_1 * !y')) \\
\overleftarrow{\mathcal{D}_x}\llbracket \lambda y . e \rrbracket k & = & k(\lambda \ddot{y} . \lambda \kappa . \overleftarrow{\mathcal{D}_x}\llbracket e \rrbracket \kappa) \\
\overleftarrow{\mathcal{D}_x}\llbracket e_1\ e_2 \rrbracket k & = & \overleftarrow{\mathcal{D}_x}\llbracket e_1 \rrbracket(\lambda \ddot{y}_1 . \overleftarrow{\mathcal{D}_x}\llbracket e_2 \rrbracket(\lambda \ddot{y}_2 . (\ddot{y}_1\ \ddot{y}_2) k)) \\
\overleftarrow{\mathcal{D}_x}\llbracket \text{let}\ y = e_1\ \text{in}\ e_2 \rrbracket k & = & \overleftarrow{\mathcal{D}_x}\llbracket e_1 \rrbracket(\lambda \ddot{y} . \overleftarrow{\mathcal{D}_x}\llbracket e_2 \rrbracket k) \\
\overleftarrow{\mathcal{D}_x}\llbracket \text{fst}\ e \rrbracket k & = & \overleftarrow{\mathcal{D}_x}\llbracket e \rrbracket(\lambda \ddot{y} . k(\text{fst}\ \ddot{y})) \\
\overleftarrow{\mathcal{D}_x}\llbracket \text{snd}\ e \rrbracket k & = & \overleftarrow{\mathcal{D}_x}\llbracket e \rrbracket(\lambda \ddot{y} . k(\text{snd}\ \ddot{y})) \\
\overleftarrow{\mathcal{D}_x}\llbracket \text{ref}\ e \rrbracket k & = & \overleftarrow{\mathcal{D}_x}\llbracket e \rrbracket(\lambda \ddot{y} . k(\text{ref}\ \ddot{y})) \\
\overleftarrow{\mathcal{D}_x}\llbracket !e \rrbracket k & = & \overleftarrow{\mathcal{D}_x}\llbracket e \rrbracket(\lambda \ddot{y} . k(!\ddot{y})) \\
\overleftarrow{\mathcal{D}_x}\llbracket e_1 := e_2) \rrbracket k & = & \overleftarrow{\mathcal{D}_x}\llbracket e_1 \rrbracket(\lambda \ddot{y}_1 . \overleftarrow{\mathcal{D}_x}\llbracket e_2 \rrbracket(\lambda \ddot{y}_2 . k(\ddot{y}_1 := \ddot{y}_2))) \\
\overleftarrow{\mathcal{D}_x}\llbracket (e_1,\ e_2) \rrbracket k & = & \overleftarrow{\mathcal{D}_x}\llbracket e_1 \rrbracket(\lambda \ddot{y}_1 . \overleftarrow{\mathcal{D}_x}\llbracket e_2 \rrbracket(\lambda \ddot{y}_2 . k((\ddot{y}_1,\ \ddot{y}_2))))
\end{array}
$$

Figure 6. Reverse-mode AD: CPS transformation rules to implement the transform in Figure 4. Note that the continuation k is considered part of the target language. We use e_1 `+=` e_2 and `let` (y_1, y_2) `=` e_1 `in` e_2 as syntactic sugar in the standard way.

3.3 Implementation using Control Operators

The `shift` and `reset` operators [18] are well-defined control operators for delimited continuations. They work together to capture a partial return path up to a programmer-defined bound: in our case the remainder of the forward pass. In Figure 8, the keyword `shift` provides access to a delimited continuation that reaches up the call chain to the nearest enclosing `reset`. The Scala compiler transforms all the intermediate code into a continuation, and passes it to the `shift` construct as the parameter `k` [57]. The implementation of `NumR` with `shift/reset` operators is almost identical to the CPS `NumR` implementation in Figure 7 (modulo the added `shift`). Note that the `shift/reset` operators in Scala involve type annotation in the form of `A @cps[B, C]` (or `A @cps[B]` if `B == C`). Semantically, this means that the `shift` construct can be used anywhere `A`-typed values are needed, but it must

```
// Differentiable real number type.
class NumR(val x: Double, var d: Double) {
  def +(that: NumR) = { (k: NumR=>Unit) =>
    val y = new NumR(x + that.x, 0.0); k(y)
    this.d += y.d; that.d += y.d
  }
  def *(that: NumR) = { (k: NumR=>Unit) =>
    val y = new NumR(x * that.x, 0.0); k(y)
    this.d += that.x * y.d
    that.d += this.x * y.d
  } ...
}
// Differentiation operator.
def grad(f: NumR => (NumR=>Unit) => Unit )(x: Double) = {
  val z = new NumR(x, 0.0)
  f(z)(r => r.d = 1.0)
  z.d
}
// Example: 2*x + x*x*x.
val df = grad { x => k =>
  (2*x) (y1=>( x*x )(y2=>(y2 *x )(y3=>(y1 + y3)k)))
}
forAll { x => df(x) = 2 + 3*x*x }
```

Figure 7. Automatic Differentiation in Scala: reverse-mode AD in continuation-passing style (top), `grad` function definition and use case (bottom). Handling of continuations is highlighted. Note that *val* and *var* mean immutable and mutable variables respectively in Scala. Constants are implicitly lifted to `NumR`s. Code first appeared in [71].

```
// Differentiable number type.
class NumR(val x: Double, var d: Double) {
  def +(that: NumR) = shift { (k: NumR=>Unit) =>
    val y = new NumR(x + that.x, 0.0); k(y)
    this.d += y.d; that.d += y.d
  }
  def *(that: NumR) = shift {( k: NumR=>Unit) =>
    val y = new NumR(x * that.x, 0.0); k(y)
    this.d += that.x * y.d
    that.d += this.x * y.d
  } ...
}
// Differentiation operator.
def grad(f: NumR => NumR @cps[Unit] )(x: Double) = {
  val z = new NumR(x, 0.0)
  reset { f(z).d = 1.0 }
  z.d
}
// Example: 2*x + x*x*x.
val df = grad {
  x => 2*x + x*x*x
}
forAll { x => df(x) = 2 + 3*x*x }
```

Figure 8. Automatic Differentiation in Scala: reverse-mode using delimited continuations, with `shift`/`reset` operators (top), `grad` function definition, and use case (bottom). Handling of continuations is confined to implementation logic and does not leak into user code. Constants are implicitly lifted to `NumR`s. Code first appeared in [71].

be within a `reset` context of type `B`, which the occurence of `shift` changes to type `C` [5]. For reverse-mode AD, it requires the continuation `k` to be of type `NumR => Unit`, and the body of `shift` to be of type `Unit`.

It is important to note at this point that this method may appear similar to Pearlmutter and Siskind [53]. However, there are substantial differences despite the similarities and shared goals. The implementation proposed by Pearlmutter and Siskind returns a pair of a value and a backpropagator: $x \mapsto (v, dv/dy \mapsto dx/dy)$ for backward propagation. Doing this correctly requires a non-local program transformation, as noted in that paper. Further tweaks are required if a lambda uses variables from an outer scope: there must be some mechanism that allows backpropagation for captured variables, not just the function inputs.

Using delimited continuations with `shift`/`reset` operators, by contrast, enables reverse-mode AD with only local transformations. Any underlying non-local transformations are implicitly resolved by `shift` and `reset`. Beyond this, it is also worth noting that our method can allocate all closures and mutable variables on the stack, i.e, we never need to return closures that escape their allocation scope. Indeed, our computation graph is never reified, but instead remains implicit in the function call stack. A consequence of this is that tail calls become proper function calls. The proposed implementation is also extremely concise, to the point that it can serve as a *specification* of reverse-mode AD and can be used to teach AD to students.

3.4 Purely Functional Implementation

Since our presentation makes central use of mutable state, an interesting question is whether a purely functional formulation is also possible. For example, since the continuation `k` takes a new `NumR`, updates its gradient, and returns `Unit`, why not simply let `k` return the new gradient and avoid mutation? The type of `k` would change to `Double => Double` accordingly. Unfortunately, this simple change is not enough, because the continuation `k` may update the gradients of *more than one* `NumR`. If earlier `NumR`s are also involved in the computation in `k`, then `k` needs to update their gradients too, but returning just a `Double` without side-effects cannot achieve that. Thus, a pure functional implementation is easy to achieve for straightline programs [25], but not for ones with complex control flow and especially nested lambdas.

Based on this observation, we can build a purely functional implementation by adding a layer of indirection. Each `NumR` is assigned a unique id, and we change the type of continuations to `NumR => Map[Id, Double]`, returning an immutable map from `NumR` ids to their calculated gradient updates. In essence, this model uses a reified functional store for gradient updates instead of storing the gradients directly in the Scala heap. Since there is no conceptual simplification, we prefer the model based on direct mutation for our presentation.

3.5 Nested Invocations For Higher-Order Gradients

Just like with forward-mode AD in Section 2.4, we are interested in extending the current reverse-mode implementation to support nested invocations of the `grad` operator.

While it would be possible to nest reverse-mode AD within reverse-mode AD (i.e., reverse-of-reverse) based on iterated CPS conversion and higher-order control operators such as `shift2` [18], we cannot implement reverse-of-reverse in Scala since the compiler only provides a single CPS transform layer. However, it is possible to nest forward-mode AD with reverse-mode AD for higher-order derivatives: this "mixed differentiation" may be practically relevant for performance. For a typical $\mathbb{R}^n \rightarrow \mathbb{R}$ function, it is more efficient to compute first-order gradient with reverse-mode, and second-order gradient with forward-mode, thus achieving a "forward-of-reverse" combination. In particular, we can compute Hessians as the Jacobian of gradients [6], and Hessian vector products in a single forward-of-reverse pass [14].

To achieve this, we need to unify `NumR` under the same abstract `Num` from Section 2.4, and use the aforementioned dynamic tagging method to address perturbation confusion.

```
class NumR(val x: Num, var d: Num, tag: Int) extends Num {...}
```

4 Reifying Computation Graphs via Multi-Stage Programming

CPS conversion puts our reverse-mode AD on a firm basis, rooted in programming language concepts. Extending the `Num` type to tensors and relaying tensor operations to high performance libraries provides all the necessary machinery for a deep learning framework in the expressive PyTorch-style that performs gradient computation as part of the normal program execution ("define-by-run").

From PyTorch-style to TensorFlow-style However, TensorFlow-style frameworks have traditionally been more performant than define-by-run ones. TensorFlow-style frameworks construct a restricted dataflow model *before* executing gradient computation, which offers a larger optimization surface on the tensor IR level ("define-then-run"). Can we also realize a TensorFlow-style framework, but with a richer and more standard IR language, better supporting native control flow and recursion?

TensorFlow-style via Multi-Stage Programming This question can be naturally addressed by *multi-stage programming* (staging). Modern tools such as LMS (Lightweight Modular Staging) [58] blend normal program execution with IR construction. A LMS-based IR can represent not only tensor operations, but also native control flow, closures, and recursion. LMS also supports runtime specialization for further code generation optimizations. The flavor of LMS is shown in the following example:

```
def nameScore(name: Rep[String]) = name.map(c => c - 64).sum
def totalScore(names: Rep[Vector[String]]) = {
  val scores = for ((a,i) <- names.zipWithIndex) yield (i * nameScore(a))
  scores.sum
}
```

Here, a simple score is computed for a vector of strings. `Rep` is used in LMS to mark staged expressions. The types `Rep[Vector[String]]` and `Rep[String]` indicate staged IR construction. The implementation crucially relies on type inference and advanced operator overloading capabilities, which extend to built-in control flow constructs like `if`, `for`, and `while`, so that normal syntax can be used. As shown in the example, an expression

```
for ((a,i) <- names) yield ...
```

becomes a series of method calls with closure arguments:

```
names.map((a,i) => ...)
```

There are no fundamental challenges with staging our reverse-mode AD in CPS using LMS, as it is a well-known insight that multi-stage programs that use continuations at generation time can generate code in CPS [9]. LMS can also be set up to generate low-level, efficient code in C++ and CUDA. This enables a TensorFlow-style framework with rich analysis and optimization opportunities, much like an aggressive whole-program compiler.

The apparent downsides of TensorFlow-style systems, however, are the rather clunky user programming model offered by current frameworks, the absence of sophisticated control flow constructs, and the inability to use standard debugging facilities. However, our system largely avoids the downsides of current static frameworks thanks to staging (in particular, the LMS framework). Of course, TensorFlow can also be viewed as a staging programming model, but the staged language is a restricted dataflow language. On the other hand, LMS provides a rich staged language that includes subroutines, recursion, and more.

We show below how LMS is added to our CPS-based AD system, and demonstrate how to support CPS code generation and hence gradient computation for branches, loops, and recursion in a natural form.

4.1 Staging Reverse-Mode AD: Straight-Line Code

We begin by investigating how to stage and perform AD on straight-line programs (i.e., those without loops, branches, or recursion). In line with the requirement of destination-passing style, we present a staged version of `Num` as follows:

```
class Num(val x: Rep[Double], val d: Rep[Var[Double]]) {...}
```

Here, the `Rep[Var[Double]]` maps to type `double&` in C++, which allows us to accumulate gradients `d` by reference. Note that our presentation is isomorphic to staging `Num` as in `Rep[Num]`, since both fields of `Num` are already staged.

Given this basic setting of `Rep` types, we refer to generic types (`A`, `B`, `C`) in the following part of this section to illustrate the abstraction of branches, loops, and recursion with CPS.

4.2 Staging Reverse-Mode AD: Conditionals

We cannot simply use the virtualized `if` operator in LMS for branches because we need to manage continuations using `shift` operators. Thus, we define a standalone `IF` function, which takes a `Rep[Boolean]` condition and two `(=> Rep[A] @cps[Rep[B]])` typed parameters for the `then`- and `else`-branches. In Scala, `=> T` typed parameters are *passed by name*, so that the parameters are evaluated each time they are used. The `IF` function accesses the delimited continuation `k` via `shift`, and invokes the continuation either with the `then`-branch parameter or the `else`-branch parameter, based on the value of the condition. Since the continuation `k` is repeated in both branches, simply passing `k` to both will result in code duplication (exponential for consecutive `IF` invocations). Rather, we stage `k` as a C++ lambda using the `fun` operator provided by LMS:

```
def fun(f: Rep[A] => Rep[B]): Rep[A => B]
```

Internally, `IF` invokes `k` in each branch within a `reset`:

```
def IF(c: Rep[Boolean])(a: => Rep[A] @cps[Rep[B]])
    (b: => Rep[A] @cps[Rep[B]]): Rep[A] @cps[Rep[B]] =
  shift { k: (Rep[A] => Rep[B]) =>
    // Emit k as a proper function to avoid code duplication.
    val k1 = fun(k)
    // Emit conditional, with each branch enclosed by a reset.
    if (c) reset(k1(a)) else reset(k1(b))
}
```

Below is an example using the `IF` construct:

```
def snippet(in: Rep[Double]): Rep[Double] =
  gradR(x => IF(x.x > 0.0){ -1.0*x*x }{ x*x })(in)
```

Here is the generated C++ code:

```
double Snippet(double in) {
  auto k = [&](double x, double& d) { d = 1.0; };
  double d = 0.0;
  if (in > 0.0) { k(-in * in, d); return -2.0 * in * d; }
  else { k(in * in, d); return 2.0 * in * d; }
}
```

4.3 Staging Reverse-Mode AD: Loops

Differentiable loop constructs are important for deep learning, for example in recurrent neural networks. By the rules of CPS transformation, loops need to be transformed into recursive functions. A loop construct consists of an initial value `Rep[A]`, a loop guard, and a loop body of type `Rep[A] => Rep[A] @cps[Rep[B]]` as parameters. The loop guard can be either `Rep[A] => Rep[Boolean]`, like a `while` construct, or simply a `Rep[Int]`, like a `for` construct. The actual loop logic can be described as follows: if the loop guard is true, recursively call the loop after invoking the loop body; else call the continuation. The `WHILE` construct is defined below, mimicking the standard `while` loop.

```
def WHILE(init: Rep[A])(c: Rep[A] => Rep[Boolean])
    (b: Rep[A] => Rep[A] @cps[Rep[B]]): Rep[A] @cps[Rep[B]] =
  shift { k: (Rep[A] => Rep[B]) =>
    // Recursive function implementing loop semantics.
    lazy val loop: Rep[A] => Rep[B] = fun { (x: Rep[A]) =>
      if (c(x)) reset(loop(b(x))) else reset(k(x))
    }
    loop(init)
}
```

Below is an example using the `WHILE` construct:

```
def snippet(in: Rep[Double]): Rep[Double] =
  gradR(x => WHILE(x)(t => t.x > 1.0)(t => t * 0.5))(in)
```

Here is the generated C++ code:

```
double Snippet(double in) {
  auto k = [&](double x, double& d) { d = 1.0; };
  auto loop = [&](double x, double& d) {
    if (x > 1.0) { double d1 = 0.0; loop(0.5 * x, d1); d += 0.5 * d1; }
    else k(x, d); };
  double d = 0.0;
  loop(in, d);
  return d;
}
```

4.4 Staging Reverse-Mode AD: Functions & Recursion

As a true differentiable programming framework, we aim to handle general forms of recursion. This is useful in deep learning: one application is processing tree-structured data, such as sentence parse trees (see Section 5).

We need the ability to "stack" continuations to transform recursive functions into CPS. For that, we first introduce a `FUN` subroutine that works like `fun` (provided by LMS), but produces a staged function in CPS. `FUN` "inserts" its parameter `f` before its delimited continuation `k`, captured by `shift`.

```
def FUN(f: Rep[A] => Rep[B] @cps[Rep[C]]) = {
  // Stack f onto k (continuation k is extended).
  val f1 = fun((x, k) => reset(k(f(x))))
  {(x: Rep[A]) => shift {k: (Rep[B] => Rep[C]) => f1((x, fun(k)))}}
}
```

With this `FUN` subroutine, implementing a tree traversal is straightforward. We can define a `TREE` abstraction to recursively traverse a `Rep[Tree]` data structure. For empty trees, the `init` value is passed along directly. For non-empty trees, the result of the left child and right child are composed by the `b` function supplied by the caller.

```
def TREE(init: Rep[B])(t: Rep[Tree])
    (b: (Rep[B], Rep[B]) => Rep[B] @cps[Rep[C]]): Rep[B] @cps[Rep[C]] = {
  def f = FUN { tree: Rep[Tree] =>
    // If tree is empty, return the initial values.
    if (tree.isEmpty) init
    // Otherwise, recurse on subtrees and compose results.
    else b(f(tree.left), f(tree.right))
  }
  f(t)
}
```

Below is an example using the `TREE` construct:

```
def snippet(tree: Rep[Tree], in: Rep[Double]): Rep[Double] =
  gradR(x => TREE(x)(tree){(l, r) => l * r * tree.value})(in)
```

Here is the generated C++ code:

```
double Snippet(Tree tree, double in) {
  auto k = [&](double x, double& d) { d = 1.0; };
  auto rec = [&](Tree tree, function<void(double, double&)> k,
                 double x, double& d) {
    if (!tree.isEmpty) {
      auto k_l = [&](double x_l, double& d_l) {
        auto k_r = [&](double x_r, double& d_r) {
          double x_t = tree.value;
          double dt = 0.0;
          k(x_l * x_r * x_t, dt);
          d_l += x_r * x_t * dt;
          d_r += x_l * x_t * dt;
        };
        rec(tree.right, k_r, x, d);
      };
      rec(tree.left, k_l, x, d);
    } else k(x, d);
  };
  double d = 0.0;
  rec(tree, k, in, d);
  return d;
}
```

With the above implementations, we have established a staged reverse-mode AD framework that supports branches, loops, and recursion. Though implementing these control-flow operators requires some engineering, they simply combine CPS transformation with staging and follow a straightforward, principled design.

The resulting framework provides a programming interface that is similar in style and expressiveness to PyTorch. It also generates an intermediate representation with inlined AD logic (pure manipulation of `Doubles` or `Tensors`) which allows extensive optimization similar in style to TensorFlow.

We note in passing that while it is, naturally, an option to implement CPS at the LMS IR level, we choose to forgo this route in favor of the presented implementation for accessibility and simplicity. A good, *selective*, CPS transform (that transforms only the minimum necessary code to CPS) is nontrivial to implement [57].

5 Evaluation

So far, we have shown both how to implement PyTorch-style reverse-mode AD using delimited continuations, and how to mix in multi-stage programming for TensorFlow-style graph reification. Now, we extend our implementation to tensor operations, and present *Lantern*, a system that scales our described approach to real-world deep learning workloads.

Tensor operations are implemented via BLAS library functions (for CPU), or via custom kernels and cuBLAS/cuDNN library functions (for GPU). We would like to stress that the efficiency of Lantern can be further improved by more sophisticated backend engineering, which is not the focus of this paper. One direction is tensor IR level optimization similar to TVM [13] and Glow [59], including constant folding, operator fusion, and systematic operation scheduling. Tensor IR level optimization is naturally supported by define-then-run systems (e.g. Lantern and TensorFlow), but not by define-by-run systems (e.g. PyTorch, though recently PyTorch 1.0 moves towards this direction by extracting computation graphs using Torch Script[4]). Another important direction is advanced batching support, either in the form of *autobatching* à la Dynet [44] or *dynamic batching* à la TensorFlow Fold [41]. Advanced batching support is particularly useful in dynamic models where manual batching is challenging. Another use is suggesting optimal batch sizes based on model and hardware details (e.g. GPU memory size).

Even with the current level of backend engineering, our evaluation shows that Lantern is competitive on contemporary machine learning models, thus pushing the boundaries of existing frameworks in various dimensions.

5.1 CNN: SqueezeNet and ResNet50

SqueezeNet [32] and ResNet50 [30] are contemporary convolutional neural network models for image classification. PyTorch and TensorFlow implementations of these models exist on GitHub. We handwrote identical Lantern models and also imported ONNX models into Lantern. We evaluate these implementations on the CIFAR-10 [5] dataset and plot the median runtime for training of one epoch. [6] As shown in Figure 9, all three models have similar runtime performance. This result is expected, since SqueezeNet and ResNet50 are mostly composed of convolutional layers, which are bottlenecked by the performance of the same set of cuDNN API functions used by all frameworks. Getting Lantern on par with PyTorch and TensorFlow took non-trivial effort: memory management techniques were crucial.

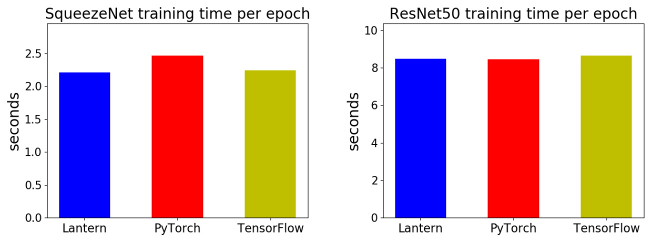

Figure 9. Running time of SqueezeNet and ResNet50 for different frameworks.

5.2 RNN: DeepSpeech2

DeepSpeech2 [4] is a representative deep neural network for automatic speech recognition (ASR), which reaches state-of-the-art performance on real-world datasets. DeepSpeech2 is the most complex model in our evaluation: it is a real production model with convolutional, batch norm, and RNN layers, and is trained with the CTC loss function. Its predecessor DeepSpeech is included in the MLPerf benchmark suite.

We evaluated DeepSpeech2 models on the Librispeech [49] dataset, but skipped TensorFlow because it uses a custom CPU implementation of `CTCLoss`, making a fair comparison impossible. Lantern and PyTorch models both use bidirectional RNNs with ReLU activation and SGD with momentum.

We observed that Lantern is 35% slower than PyTorch (Figure 10 left). Detailed profiling shows that invocations (by RNN layers) of `sgemm_32x32x32_NN_vec` and `RNN_elementWise_fp` are 20% slower for Lantern than PyTorch. After eliminating obvious possibilites such as code duplication, different kernel algorithms, and performance-related flags, we were still unable to close the speed gap. We suspect performance differences may be due to memory alignment/coalesing issues or accidental synchronization overhead. We expect that, with enough tuning effort, we can match PyTorch's performance: it comes down to careful flag tuning, memory management, and other low level details.

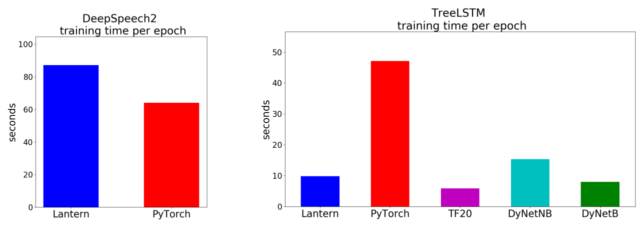

Figure 10. Running time of DeepSpeech2 and TreeLSTM for different frameworks.

[4]https://pytorch.org/docs/master/jit.html
[5]https://www.cs.toronto.edu/~kriz/cifar.html
[6]All evaluations are run on a single GeForce GTX 1080 Ti with PyTorch 1.0rc, TensorFlow 1.12.0-rc0, and Lantern. All use CUDA 10.

5.3 TreeLSTM: Sentiment Analysis

TreeLSTM is a state-of-the-art *recursive* machine learning model that heavily depends on dynamic control flow guided by structural training data. These models are useful for handling natural language parse trees and embedding abstract syntax trees of programming languages.

We showcase TreeLSTM on the Sentiment Classification task [67] using the Stanford Sentiment Treebank dataset [15], which contains parse trees of movie reviews and sentiment labels for every tree node. TreeLSTM learns hidden vectors embeddings for each node i in this format, based on the vector embeddings of each word from the `Embedding` look-up table:

$$h_i = \text{Bi-LSTM}(\text{Embedding}(i.\text{word}), h_{i.\text{left}}, h_{i.\text{right}})$$

by minimizing softmax-cross-entropy loss with regard to the true labels.

We evaluated Lantern along with PyTorch, TensorFlow Fold [41], and Dynet [44]. Lantern, PyTorch, and Dynet directly support recursive models implemented as recursive functions. Lantern's code is even more concise given the `TREE` abstraction (Section 4.4). Both Lantern and PyTorch implementations used batch size 1, because batching recursive models is challenging given the irregular structure of the training data. Dynet supports heuristic on-the-fly autobatching, which allows automated batching regardless of model structure. TensorFlow Fold supports dynamic batching, which augments the computation graph with regard to the training data, so that the reified static graph (with extra `concat`, `while_loop`, and `gather` nodes) can handle the training data in batches.

The evaluation of TreeLSTM is shown in Figure 10 (right). Due to the challenge in batching, tensor computations are essentially matrix-vector multiplications, which doesn't benefit much from GPU acceleration. The PyTorch model is the slowest, since interpreting recursive functions in Python is not efficient. Lantern generates C++/CUDA code with recursive closures. Lantern's better performance is most likely due to compilation optimization, and a unified C++/CUDA platform that handles both data-loading and tensor computations. We run TensorFlow Fold at batch size 20 (TF20). Due to dynamic batching, the performance benefits greatly from a GPU backend, making TF20 the fastest in our comparison. Dynet without autobatching (DyNetNB) has a lightweight graph representation and optimized C++ backend [43], which easily outperforms PyTorch, but not Lantern. Enabling autobatching showed another 50% improvement on GPU, which outperforms the current Lantern backend.

6 Related Work

Automated Differentiation: History Gradient-based optimization lies at the heart of machine learning, with backpropagation [60], a special case of reverse-mode AD, as a key ingredient for training neural networks. The fundamental idea of automatic differentiation (AD) emerged in the 1950s as programs that automatically calculate derivatives [8, 45]. A formal introduction to forward-mode AD appeared in the 1960s [74]. The application of gradient descent first arose in control theory [10, 11]. In the 1970s, Linnainmaa [40] introduced the concept of reverse-mode AD and the related idea of computation graphs, which are now widely used by modern machine learning frameworks. Speelpenning [64] implemented reverse-mode AD in a general-purpose programming language, which is considered the first implementation of reverse-mode AD that performed gradient computations automatically. At the same time, backpropagation was invented and reinvented within the machine learning community [50, 60, 75]. This divergence continued until Hecht-Nielsen [31] cited the work from both communities.

Automated Differentiation: A PL View AD has also received attention from the programming language community, with recent proposals to generalize neural network models to differentiable functional programs [28, 47]. This development is also fueled by modern deep learning frameworks, which define neural networks "very much like a regular program" [2, 38]. Some recent research demonstrates this direct correspondence between the two fields by implementing differentiable analogues of traditional data structures [17] and machine models [29]. Another line of work has aimed to formalize AD, both forward-mode [62] and reverse-mode [53]. There exist high-level languages with first-class AD operators [63], as well as flexible AD library implementations, e.g., DiffSharp [7]. A Haskell implementation of forward-mode AD was proposed by Elliott [23]. Swift for TensorFlow [7] integrates AD as a first-class feature in a general purpose language. For a thorough view of AD and deep learning from a functional programming perspective, we advise readers to refer to Baydin et al. [6].

A Tale of Two Styles Most modern deep learning frameworks compute gradients of training loss with respect to neural network parameters to perform parameter update (backpropagation). This is generally done in one of two ways [6]. The first is to let users define computation graphs using a domain-specific language (DSL) and to interpret graph operations at runtime. Computation graphs represent entire programs and are more amenable to global analysis and optimizations like operator fusion. However, graph-building DSLs are limited in expressivity, contain unintuitive control structures, and are difficult to debug. Frameworks such as Theano [3] and TensorFlow [1] belong to this category. The other way is to integrate general-purpose programming languages with reverse-mode AD as a library, of which Torch [16], PyTorch [51, 52], Autograd [42], and Chainer [68]

[7] https://github.com/tensorflow/swift/blob/master/docs/AutomaticDifferentiation.md

are well-known representatives. Caffe [33], MXNet [12], and CNTK [61] are somewhere in the middle. The tight integration between host languages and AD frameworks of the pure-library category has certain usability benefits, such as natural control flow and easy debugging, but at the expense of efficiency. A recent effort to bridge this gap is ONNX [48], a neural network exchange format which enables easy conversion between frameworks.

Staging: A Unification of the Two Styles The present work aims to reap the benefits of both models by using a computation graph DSL almost as expressive as a general programming language. Previous attempts at building source-to-source deep learning compilers mostly focus on either the define-by-run or define-then-run approach, as noted by Baydin et al. [6]. Tangent [69, 76] implements a source-to-source compiler in Python which supports automatic differentiation, but this framework constrains the host language to a limited subset of Python. DLVM [72, 73] compiles deep learning programs written in Swift into a domain-specific SSA IR, performs analyses and transformations (including source code transformation AD), and generates code via LLVM. Swift for TensorFlow [8] mixes the two approaches: it enables imperative-style programs but uses a "graph program extraction" compiler transform to automatically extract tensor code and build computation graphs.

Our transformation of high-level neural networks to low-level code is fueled by the idea of multi-stage programming (staging). More than 30 years ago, Jørring and Scherlis [34] observed that many computations can be naturally separated into stages distinguished by frequency of execution or availability of data. The idea to treat staging as an explicit *programming model* was popularized, among others, by Taha and Sheard [66]. Since then, there are modern staging approaches which blend normal program execution with the delayed construction of an *intermediate program representation* (IR), which may be a computation graph, or more customarily, an abstract syntax tree (AST). An example is the Lightweight Modular Staging (LMS) framework [58], which provides a rather seamless implementation of staging in the Scala language and has been utilized in a range of existing applications [55, 56, 65]. Lantern, our deep learning framework built on LMS, achieves a balance between source code expressivity and target code performance via this compiler building technique.

Delimited Continuations: A Simpler Essence Lantern relies on delimited continuations [18, 19, 21], as implemented in Scala [57]. In parallel to our work,[9] Elliott [25] proposed a generalized view of AD based on the paradigm of "compiling to categories" [24]. The paper echoes our view of AD as a specific form of symbolic differentiation and also mentions continuations for reverse AD, but overall it approaches the problem from a categorial perspective that is quite different from ours. Although the paper mentions performance and parallelization as potential benefits, it presents no experimental evaluation. A key limitation of the approach appears to be that extracted computations are straight-line dependency graphs that do not model control flow. Hence, loops, functions, etc. need to be partially evaluated as part of the translation, potentially leading to code blow-up or nontermination. Another claimed benefit is the purely functional implementation. However, we believe this is largely due to the restricted scope. As we discuss in Section 3.4, some notion of gradient updates is likely required in the presence of in-graph functions. In comparison, our work proposes what we think is an "even simpler essence" of automatic differentiation. In particular, we show that continuations are central to reverse-mode AD, but that category theory is optional. Focusing on continuations as the key enabler makes reverse-mode AD (and hence gradient-descent optimization) immediately applicable to basically any program, including in-graph recursion and higher-order functions. Our presentation is closely in line with Wang and Rompf [71] and Wang et al. [70], who introduced a PL-centric view of deep learning techniques to the machine learning community, including backpropation with continuations and experiments with much earlier prototypes of the Lantern system that did not support GPU execution. The present paper is the first to present related ideas to a PL audience and to evaluate Lantern on realistic deep learning models and environments (e.g., ResNet, DeepSpeech2 on GPUs).

7 Conclusions

With this paper, we set out to demystify reverse-mode AD by examining it through the lens of program transformation. In doing so, we uncovered a tight connection between reverse-mode AD and delimited continuations. With the help of control operators, we provided an implementation of reverse-mode AD using operator overloading that is no more complex than forward-mode implementations.

We further combined this formulation of AD with multi-stage programming (staging), which leads to a highly efficient implementation that combines the performance benefits of deep learning frameworks based on explicit reified computation graphs (e.g., TensorFlow) with the expressivity of pure library approaches (e.g., PyTorch).

Based on these two ideas, we have built a deep learning framework named Lantern. With native C++/CUDA backends, Lantern attains competitive performance for a variety of state-of-the-art deep learning models, such as SqueezeNet, ResNet, DeepSpeech2, and TreeLSTM.

[8]https://github.com/tensorflow/swift

[9]Drafts of both papers independently appeared on arXiv within days of each other in spring 2018.

References

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Gregory S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian J. Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Józefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Gordon Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul A. Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda B. Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. *CoRR* abs/1603.04467 (2016). arXiv:1603.04467 http://arxiv.org/abs/1603.04467

[2] Martin Abadi, Michael Isard, and Derek G. Murray. 2017. A Computational Model for TensorFlow (An Introduction). http://dl.acm.org/citation.cfm?doid=3088525.3088527

[3] Rami Al-Rfou, Guillaume Alain, Amjad Almahairi, Christof Angermüller, Dzmitry Bahdanau, Nicolas Ballas, Frédéric Bastien, Justin Bayer, Anatoly Belikov, Alexander Belopolsky, Yoshua Bengio, Arnaud Bergeron, James Bergstra, Valentin Bisson, Josh Bleecher Snyder, Nicolas Bouchard, Nicolas Boulanger-Lewandowski, Xavier Bouthillier, Alexandre de Brébisson, Olivier Breuleux, Pierre Luc Carrier, Kyunghyun Cho, Jan Chorowski, Paul F. Christiano, Tim Cooijmans, Marc-Alexandre Côté, Myriam Côté, Aaron C. Courville, Yann N. Dauphin, Olivier Delalleau, Julien Demouth, Guillaume Desjardins, Sander Dieleman, Laurent Dinh, Melanie Ducoffe, Vincent Dumoulin, Samira Ebrahimi Kahou, Dumitru Erhan, Ziye Fan, Orhan Firat, Mathieu Germain, Xavier Glorot, Ian J. Goodfellow, Matthew Graham, Çaglar Gülçehre, Philippe Hamel, Iban Harlouchet, Jean-Philippe Heng, Balázs Hidasi, Sina Honari, Arjun Jain, Sébastien Jean, Kai Jia, Mikhail Korobov, Vivek Kulkarni, Alex Lamb, Pascal Lamblin, Eric Larsen, César Laurent, Sean Lee, Simon Lefrançois, Simon Lemieux, Nicholas Léonard, Zhouhan Lin, Jesse A. Livezey, Cory Lorenz, Jeremiah Lowin, Qianli Ma, Pierre-Antoine Manzagol, Olivier Mastropietro, Robert McGibbon, Roland Memisevic, Bart van Merriënboer, Vincent Michalski, Mehdi Mirza, Alberto Orlandi, Christopher Joseph Pal, Razvan Pascanu, Mohammad Pezeshki, Colin Raffel, Daniel Renshaw, Matthew Rocklin, Adriana Romero, Markus Roth, Peter Sadowski, John Salvatier, François Savard, Jan Schlüter, John Schulman, Gabriel Schwartz, Iulian Vlad Serban, Dmitriy Serdyuk, Samira Shabanian, Étienne Simon, Sigurd Spieckermann, S. Ramana Subramanyam, Jakub Sygnowski, Jérémie Tanguay, Gijs van Tulder, Joseph P. Turian, Sebastian Urban, Pascal Vincent, Francesco Visin, Harm de Vries, David Warde-Farley, Dustin J. Webb, Matthew Willson, Kelvin Xu, Lijun Xue, Li Yao, Saizheng Zhang, and Ying Zhang. 2016. Theano: A Python framework for fast computation of mathematical expressions. *CoRR* abs/1605.02688 (2016). arXiv:1605.02688 http://arxiv.org/abs/1605.02688

[4] Dario Amodei, Rishita Anubhai, Eric Battenberg, Carl Case, Jared Casper, Bryan Catanzaro, Jingdong Chen, Mike Chrzanowski, Adam Coates, Greg Diamos, Erich Elsen, Jesse Engel, Linxi Fan, Christopher Fougner, Tony Han, Awni Y. Hannun, Billy Jun, Patrick LeGresley, Libby Lin, Sharan Narang, Andrew Y. Ng, Sherjil Ozair, Ryan Prenger, Jonathan Raiman, Sanjeev Satheesh, David Seetapun, Shubho Sengupta, Yi Wang, Zhiqian Wang, Chong Wang, Bo Xiao, Dani Yogatama, Jun Zhan, and Zhenyao Zhu. 2015. Deep Speech 2: End-to-End Speech Recognition in English and Mandarin. *CoRR* abs/1512.02595 (2015).

[5] Kenichi Asai and Yukiyoshi Kameyama. 2007. Polymorphic Delimited Continuations. In *APLAS (Lecture Notes in Computer Science)*, Vol. 4807. Springer, 239–254.

[6] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. 2018. Automatic differentiation in machine learning: a survey. *CoRR* abs/1502.05767 (2018).

[7] Atilim Günes Baydin, Barak A. Pearlmutter, and Jeffrey Mark Siskind. 2016. DiffSharp: An AD Library for .NET Languages. *CoRR* abs/1611.03423 (2016).

[8] L. M. Beda, L. N. Korolev, N. V. Sukkikh, and T. S. Frolova. 1959. *Programs for automatic differentiation for the machine BESM*. Technical Report. Institute for Precise Mechanics and Computation Techniques, Academy of Science, Moscow, USSR. (In Russian).

[9] Malgorzata Biernacka, Dariusz Biernacki, and Olivier Danvy. 2005. An Operational Foundation for Delimited Continuations in the CPS Hierarchy. *Logical Methods in Computer Science* 1, 2 (2005). https://doi.org/10.2168/LMCS-1(2:5)2005

[10] A Bryson and Yu-Chi Ho. 1975. Applied optimal control: Optimization, estimation, and control (revised edition). *Levittown, Pennsylvania: Taylor & Francis* (1975).

[11] Arthur E Bryson and Walter F Denham. 1962. A steepest-ascent method for solving optimum programming problems. *Journal of Applied Mechanics* 29, 2 (1962), 247–257.

[12] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. 2015. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *arXiv preprint arXiv:1512.01274* (2015).

[13] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *OSDI*. USENIX Association, 578–594.

[14] Bruce Christianson. 1992. Automatic Hessians by reverse accumulation. *IMA J. Numer. Anal.* 12, 2 (1992), 135–150.

[15] Jason Chuang. 2013. Stanford Sentiment Treebank. https://nlp.stanford.edu/sentiment/treebank.html

[16] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. 2011. Torch7: A Matlab-like Environment for Machine Learning. In *BigLearn, NIPS Workshop*.

[17] Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett (Eds.). 2015. *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada.* http://papers.nips.cc/book/advances-in-neural-information-processing-systems-28-2015

[18] Olivier Danvy and Andrzej Filinski. 1990. Abstracting Control. In *LISP and Functional Programming*. 151–160.

[19] Olivier Danvy and Andrzej Filinski. 1992. Representing Control: A Study of the CPS Transformation. *Mathematical Structures in Computer Science* 2, 4 (1992), 361–391.

[20] Olivier Danvy and Mayer Goldberg. 2005. There and back again. *Fundamenta Informaticae* 66, 4 (2005), 397–413.

[21] Olivier Danvy and Lasse R. Nielsen. 2003. A first-order one-pass CPS transformation. *Theor. Comput. Sci.* 308, 1-3 (2003), 239–257.

[22] John C. Duchi, Elad Hazan, and Yoram Singer. 2011. Adaptive Subgradient Methods for Online Learning and Stochastic Optimization. *Journal of Machine Learning Research* 12 (2011), 2121–2159. http://dl.acm.org/citation.cfm?id=2021068

[23] Conal Elliott. 2009. Beautiful differentiation. In *International Conference on Functional Programming (ICFP)*. http://conal.net/papers/beautiful-differentiation

[24] Conal Elliott. 2017. Compiling to categories. *PACMPL* 1, ICFP (2017), 27:1–27:27.

[25] Conal Elliott. 2018. The simple essence of automatic differentiation. *PACMPL* 2, ICFP (2018), 70:1–70:29.

[26] Jeffrey L Elman. 1990. Finding structure in time. *Cognitive science* 14, 2 (1990), 179–211.

[27] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. 1993. The Essence of Compiling with Continuations. In *PLDI*. ACM, 237–247.

[28] Brendan Fong, David I Spivak, and Rémy Tuyéras. 2017. Backprop as Functor: A compositional perspective on supervised learning. *arXiv preprint arXiv:1711.10455* (2017).

[29] Alex Graves, Greg Wayne, and Ivo Danihelka. 2014. Neural Turing Machines. *CoRR* abs/1410.5401 (2014). arXiv:1410.5401 http://arxiv.org/abs/1410.5401

[30] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In *CVPR*. IEEE Computer Society, 770–778.

[31] Robert Hecht-Nielsen. 1988. Theory of the backpropagation neural network. *Neural Networks* 1, Supplement-1 (1988), 445–448. https://doi.org/10.1016/0893-6080(88)90469-8

[32] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR* abs/1602.07360 (2016).

[33] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross B. Girshick, Sergio Guadarrama, and Trevor Darrell. 2014. Caffe: Convolutional Architecture for Fast Feature Embedding. *CoRR* abs/1408.5093 (2014). arXiv:1408.5093 http://arxiv.org/abs/1408.5093

[34] Ulrik Jørring and William L. Scherlis. 1986. Compilers and Staging Transformations. In *POPL*. ACM Press, 86–96.

[35] Diederik P. Kingma and Jimmy Ba. 2014. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2014). arXiv:1412.6980 http://arxiv.org/abs/1412.6980

[36] John Launchbury and Simon L. Peyton Jones. 1994. Lazy Functional State Threads. In *PLDI*. ACM, 24–35.

[37] Yann LeCun. 2018. Deep Learning est mort. Vive Differentiable Programming! https://www.facebook.com/yann.lecun/posts/10155003011462143.

[38] Yann LeCun. 2018. LcCun's facebook pose on Differentiable programming. https://www.facebook.com/yann.lecun/posts/10155003011462143

[39] Yann LeCun, Bernhard E Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne E Hubbard, and Lawrence D Jackel. 1990. Handwritten digit recognition with a back-propagation network. In *Advances in neural information processing systems*. 396–404.

[40] Seppo Linnainmaa. 1976. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics* 16, 2 (1976), 146–160.

[41] Moshe Looks, Marcello Herreshoff, DeLesley Hutchins, and Peter Norvig. 2017. Deep Learning with Dynamic Computation Graphs. *ICLR* (2017).

[42] Dougal Maclaurin. 2016. *Modeling, Inference and Optimization with Composable Differentiable Procedures*. Ph.D. Dissertation.

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

[44] Graham Neubig, Yoav Goldberg, and Chris Dyer. 2017. On-the-fly Operation Batching in Dynamic Computation Graphs. In *NIPS*. 3974–3984.

[45] John F Nolan. 1953. Analytical differentiation on a digital computer.

[46] Christopher Olah. 2015. Neural Networks, Types, and Functional Programming. http://colah.github.io/posts/2015-09-NN-Types-FP/.

[47] Christopher Olah. 2015. Neural Networks, Types, and Functional Programming. http://colah.github.io/posts/2015-09-NN-Types-FP/

[48] ONNX working groups. 2017. ONNX: Open Neural Network Exchange format. https://onnx.ai/

[49] Vassil Panayotov, Guoguo Chen, Daniel Povey, and Sanjeev Khudanpur. 2015. Librispeech: An ASR corpus based on public domain audio books. *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2015), 5206–5210.

[50] D.B. Parker, Massachusetts Institute of Technology, and Sloan School of Management. 1985. *Learning Logic: Casting the Cortex of the Human Brain in Silicon*. Massachusetts Institute of Technology, Center for Computational Research in Economics and Management Science. https://books.google.com/books?id=2kS9GwAACAAJ

[51] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. 2017. PyTorch: Tensors and dynamic neural networks in Python with strong GPU acceleration. www.pytorch.org

[52] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. (2017).

[53] Barak A. Pearlmutter and Jeffrey Mark Siskind. 2008. Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. *ACM Trans. Program. Lang. Syst.* 30, 2 (2008), 7:1–7:36.

[54] Ning Qian. 1999. On the momentum term in gradient descent learning algorithms. *Neural Networks* 12, 1 (1999), 145–151. https://doi.org/10.1016/S0893-6080(98)00116-6

[55] Tiark Rompf and Nada Amin. 2015. Functional pearl: a SQL to C compiler in 500 lines of code. In *ICFP*. ACM, 2–9.

[56] Tiark Rompf, Kevin J. Brown, HyoukJoong Lee, Arvind K. Sujeeth, Manohar Jonnalagedda, Nada Amin, Georg Ofenbeck, Alen Stojanov, Yannis Klonatos, Mohammad Dashti, Christoph Koch, Markus Püschel, and Kunle Olukotun. 2015. Go Meta! A Case for Generative Programming and DSLs in Performance Critical Systems. In *SNAPL (LIPIcs)*, Vol. 32. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 238–261.

[57] Tiark Rompf, Ingo Maier, and Martin Odersky. 2009. Implementing first-class polymorphic delimited continuations by a type-directed selective CPS-transform. In *ICFP*. ACM, 317–328.

[58] Tiark Rompf and Martin Odersky. 2010. Lightweight modular staging: a pragmatic approach to runtime code generation and compiled DSLs. In *GPCE*. ACM, 127–136.

[59] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Summer Deng, Roman Dzhabarov, James Hegeman, Roman Levenstein, Bert Maher, Nadathur Satish, Jakob Olesen, Jongsoo Park, Artem Rakhov, and Misha Smelyanskiy. 2018. Glow: Graph Lowering Compiler Techniques for Neural Networks. *CoRR* abs/1805.00907 (2018).

[60] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. 1986. Learning representations by back-propagating errors. *nature* 323, 6088 (1986), 533.

[61] Frank Seide and Amit Agarwal. 2016. Cntk: Microsoft's open-source deep-learning toolkit. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. ACM, 2135–2135.

[62] Jeffrey Mark Siskind and Barak A. Pearlmutter. 2008. Nesting forward-mode AD in a functional framework. *Higher-Order and Symbolic Computation* 21, 4 (2008), 361–376.

[63] Jeffrey Mark Siskind and Barak A. Pearlmutter. 2016. Efficient Implementation of a Higher-Order Language with Built-In AD. *CoRR* abs/1611.03416 (2016).

[64] Bert Speelpenning. 1980. *Compiling fast partial derivatives of functions given by algorithms*. Ph.D. Dissertation.

[65] Arvind K. Sujeeth, HyoukJoong Lee, Kevin J. Brown, Tiark Rompf, Hassan Chafi, Michael Wu, Anand R. Atreya, Martin Odersky, and Kunle Olukotun. 2011. OptiML: An Implicitly Parallel Domain-Specific Language for Machine Learning. In *ICML*. Omnipress, 609–616.

[66] Walid Taha and Tim Sheard. 2000. MetaML and multi-stage programming with explicit annotations. *Theor. Comput. Sci.* 248, 1-2 (2000), 211–242.

[67] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. 2015. Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks. *CoRR* abs/1503.00075 (2015). arXiv:1503.00075 http://arxiv.org/abs/1503.00075

[68] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. 2015. 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)*, Vol. 5.

[69] B. van Merriënboer, A. B. Wiltschko, and D. Moldovan. 2017. Tangent: Automatic Differentiation Using Source Code Transformation in Python. *ArXiv e-prints* (Nov. 2017). arXiv:cs.MS/1711.02712

[70] Fei Wang, James Decker, Xilun Wu, Gregory Essertel, and Tiark Rompf. 2018. Backpropagation with Callbacks: Foundations for Efficient and Expressive Differentiable Programming. In *NIPS*.

[71] Fei Wang and Tiark Rompf. 2018. A Language and Compiler View on Differentiable Programming. *ICLR Workshop Track* (2018). https://openreview.net/forum?id=SJxJtYkPG

[72] Richard Wei, Vikram S. Adve, and Lane Schwartz. 2017. DLVM: A modern compiler infrastructure for deep learning systems. *CoRR* abs/1711.03016 (2017).

[73] Richard Wei, Lane Schwartz, and Vikram Adve. 2017. A modern compiler infrastructure for deep learning systems with adjoint code generation in a domain-specific IR. In *NIPS AutoDiff Workshop*.

[74] R. E. Wengert. 1964. A simple automatic derivative evaluation program. *Commun. ACM* 7, 8 (1964), 463–464. https://doi.org/10.1145/355586.364791

[75] Paul Werbos. 1974. Beyond regression: New tools for prediction and analysis in the behavior science. *Unpublished Doctoral Dissertation, Harvard University* (1974).

[76] Alex Wiltschko. 2017. Tangent: Source-to-Source Debuggable Derivatives. https://research.googleblog.com/2017/11/tangent-source-to-source-debuggable.html