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 Pearl-mutter 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
program transformation which can be realized entirely using operator overloading in a language that supports shift/reset [18] or equivalent delimited control operators. 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.

\[
 f : A \to B \quad \hat{f}_w : A \to B \quad w \in P
\]

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 \( L(w) = \|f(a) - \hat{f}_w(a)\| \) 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 \( w^1 \), both the loss \( L(w^t) \) and the gradient\(^\dagger\) \( \nabla L(w^t) \) can be computed: The gradient marks the direction which increases the loss \( L(w^t) \) 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 \( r \).

\[
 w^{t+1} = w^t - r \nabla L(w^t)
\]

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 \( f_{\hat{w}} = \hat{f}_{n, w_n} \circ \ldots \circ \hat{f}_{1, w_1} \) 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 \( \lambda \)-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).
Demystifying Differentiable Programming

Preprint, November 2018

Syntax:
\[ e ::= c \mid x \mid e + e \mid e * e \mid \text{let } x = e \text{ in } e \]

Symbolic differentiation rules:
\[
\begin{align*}
\frac{d}{dx}[c] &= 0 \\
\frac{d}{dx}[x] &= 1 \\
\frac{d}{dx}[e_1 + e_2] &= \frac{d}{dx}[e_1] + \frac{d}{dx}[e_2] \\
\frac{d}{dx}[e_1 * e_2] &= \left( e_1 * \frac{d}{dx}[e_2] + e_2 * \frac{d}{dx}[e_1] \right) \\
\frac{d}{dx}[\text{let } y = e_1 \text{ in } e_2] &= \text{let } y = e_1 \text{ in } \frac{d}{dx}[e_1] + \frac{d}{dx}[e_2] \\
\frac{d}{dx}[y] &= y' (y \neq x)
\end{align*}
\]

**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:
\[
\frac{d}{dx} \left[ e_1 * e_2 * ... * e_n \right] = \frac{d}{dx} \left[ e_1 \right] * e_2 * ... * e_n + e_1 * \frac{d}{dx} \left[ e_2 \right] * ... * e_n + ... + e_1 * e_2 * ... * \frac{d}{dx} \left[ e_n \right]
\]

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:
\[
\begin{align*}
\frac{d}{dx} \left[ \text{let } y_1, y_2, ..., y_n = e_1, e_2, ..., e_n \text{ in } \right. \\
\text{let } z_1 = y_1 + y_2 \text{ in } \\
\text{let } z_2 = z_1 + y_3 \text{ in } \\
... \\
\text{let } z_{n-1} = z_{n-2} + y_n \text{ in } \\
\frac{d}{dx} \left[ e_1 \right] \right]
\end{align*}
\]

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 \(y = ...\) into let \(y = ...\) and let \(y' = ...\). 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 \(x = ...\) expression. All expressions are of type \(\mathbb{R}\), so a derivative can be computed for any expression. We write \(\frac{d}{dx}[e]\) 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 \(\overrightarrow{D}_x[e^\tau]\), where the arrow indicates forward-mode and where \(\tau\) is the type associated with the expression \(e\). We omit the \(r\) for a cleaner presentation if \(r\) 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 \((\overrightarrow{D}_x[\mathbb{R}] = \mathbb{R} \times \mathbb{R})\), 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
The \( \text{NumF} \) 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 \( \text{grad} \) to compute the derivative of any function \( \text{NumF} \Rightarrow \text{NumF} \) (Figure 3, middle). Internally, \( \text{grad} \) 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 \( \text{grad} \) operator is shown in Figure 3, bottom. Note that the constant 2 is implicitly converted to new \( \text{NumF}(2,0) \) (tangents of constants are 0.0 because constants do not change). The use of \( \text{Double} \) instead of a generic number type is simply for clarity of presentation.

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

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

The \( \text{NumF} \) 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 \( \text{grad} \) to compute the derivative of any function \( \text{NumF} \Rightarrow \text{NumF} \) (Figure 3, middle). Internally, \( \text{grad} \) 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 \( \text{grad} \) operator is shown in Figure 3, bottom. Note that the constant 2 is implicitly converted to new \( \text{NumF}(2,0) \) (tangents of constants are 0.0 because constants do not change). The use of \( \text{Double} \) 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 \( \text{NumF} \Rightarrow \text{NumF} \) 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 \( \text{grad}(\text{grad}(f)) \). 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 \( \text{grad} \) should return 1, meaning that the outer grad should also return 1.

\[
\text{grad} \{ x: \text{NumF} = \}
\]

// Evaluates to 2 instead of 1! Unexpected.

\[
\text{val shouldBeOne = } \text{grad}(y = x + y)(1)
\]

\[
(1)
\]

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 \( \text{NumF} \) 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 \( \text{NumF} \) and \( \text{Double} \) in the same abstract class \( \text{Num} \), 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.

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 \( f: \mathbb{R}^n \rightarrow \mathbb{R} \), 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 \( f: \mathbb{R}^n \rightarrow \mathbb{R} \) represented as a straight-line program in ANF, i.e., as a sequence of \( \text{let} \ y_j = e_j \) expressions, with inputs \( x_i \) and output \( y_m \). The basic intuition is: instead of computing all \( n \times m \) internal derivatives \( \frac{\partial f}{\partial x_i} \), \( y_j \) as in forward-mode, we would rather only compute the \( m + n \) derivatives \( \frac{\partial f}{\partial y}_n, \frac{\partial f}{\partial y}_m \). For this, we need a way to compute derivatives starting with \( \frac{\partial f}{\partial y}_n = 1 \), 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 \( \mathbb{R}^n \rightarrow \mathbb{R}^m \) 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 \( y = x \times x, z = y \times y \) below:

**Forward pass:**
1. \( \text{let } y = x \times x \)
2. \( \text{let } z = y \times y \)

**Backward pass:**
1. \( \text{let } z' = \frac{\partial f}{\partial z} = 2 \times y \)
2. \( \text{let } y' = \frac{\partial f}{\partial y} \) for each \( y \)
3. \( \text{let } x' = \frac{\partial f}{\partial x} = \frac{\partial f}{\partial y} \times \frac{\partial f}{\partial y} \) for each \( x \)

The backward propagation depends crucially on the *chain rule of differentiation* (below), which is used in the last line to compose \( \frac{\partial f}{\partial x} \) with \( \frac{\partial f}{\partial y} \) for \( \frac{\partial f}{\partial z} \).

\[
\frac{\partial f}{\partial x} \left( f(x) \right) = \begin{cases} \frac{\partial f}{\partial y} \left( g(x) \right) & \text{in} \\ \frac{\partial f}{\partial x} \left( g(x) \right) & \text{in} \end{cases}
\]

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.

**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 \( p_{ik} \) indicates the \( i \)th parameter in \( r \)th 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 $\lambda$-calculus extended with reals, tuples, and references. As in forward AD, we assume typed expressions $e^\prime$, but we often drop $\tau$ 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 $\overline{D_x} e k$ 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 $\text{let } y = e_1 \text{ in } e_2$, 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.

### 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

Standard CPS transformation (excerpt):

$$\begin{align*}
\overline{[x]}k & = k(x) \\
\overline{[e_1 * e_2]}k & = \overline{[\lambda y. \overline{[e_1]}(\overline{[e_2]}(\lambda y. k(y_1 * y_2)))]} \\
\overline{[\lambda x. e]}k & = \overline{[\lambda x. k]}(\overline{[e]}k) \\
\overline{[e_1 \cdot e_2]}k & = \overline{[\lambda y. \overline{[e_1]}(\overline{[e_2]}(\lambda y. (y_1 * y_2)k))]
\end{align*}$$

Reverse AD with CPS transformation:

**Syntax:**

$$e ::= \text{c | x | e + e | e * e | let } x = e \text{ in } e | \lambda x. e | e | \text{fst } e | \text{snd } e | \text{ref } e | !e$$

**Type:**

$$\tau ::= \text{Unit | } \mathbb{R} | \text{Ref } \tau | \tau \star \tau | \tau \rightarrow \tau$$

**Variable sugaring:**

$$y = (y, y' : \mathbb{R}^{|\text{Ref } \tau|})$$

**CPS Transformation:**

$$\begin{align*}
\overline{D_x} e^\tau \overline{\epsilon k} & = k(c) \\
\overline{D_x} e^\tau \overline{\epsilon \text{ref } 0)} & = k(c, \text{ref } 0)) \\
\overline{D_x} [x]k & = k(x, x')
\end{align*}$$

$\overline{D_x} [\lambda y. e]k = \overline{D_x} [\lambda y. \overline{D_x} [\lambda y. e]k]$

$\text{let } y = (y_1 + y_2, \text{ref } 0) \text{ in } k(y)$

$\overline{D_x} [\lambda y. e]k = \overline{D_x} [\lambda y. \overline{D_x} [\lambda y. e]k]$

$\text{let } y = (y_1 + y_2, \text{ref } 0) \text{ in } k(y)$

$\overline{D_x} e_1 + e_2 k = \overline{D_x} e_1 k + \overline{D_x} e_2 k$

$\text{let } y = y_1 + y_2, \text{ref } 0 \text{ in } k(y)$

$\overline{D_x} e_1 * e_2 k = \overline{D_x} e_1 k * \overline{D_x} e_2 k$

$\text{let } y = y_1 + y_2, \text{ref } 0 \text{ in } k(y)$

$\overline{D_x} \text{let } y = e_1 \text{ in } e_2 k = \overline{D_x} e_1 k (\overline{D_x} e_2 k)$

$\overline{D_x} \text{let } y = e_1 \text{ in } e_2 k = \overline{D_x} e_1 k (\overline{D_x} e_2 k)$

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.

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 $A @\text{cps}[B, C]$ (or $A @\text{cps}[B]$ if $B =$ C). Semantically, this means that the shift construct can be used anywhere A-typed values are needed, but it must
be within a reset context of type B, which the occurrence 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 ↦ (v, dv/dy) ↦ (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 respectively in Scala. Constants are implicitly lifted to NumRs. Code first appeared in [71].

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 respectively in Scala. Constants are implicitly lifted to NumRs. Code first appeared in [71].

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 NumRs 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 trans-
form 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 per-
formance. For a typical $\mathbb{R}^n \rightarrow \mathbb{R}$ function, it is more effi-
cient 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 com-
pute 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 $\text{Num}$ under the same
abstract $\text{Num}$ from Section 2.4, and use the aforementioned
dynamic tagging method to address perturbation confusion.

```scala
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 ba-
sis, rooted in programming language concepts. Extending
the $\text{Num}$ type to tensors and relaying tensor operations to
high performance libraries provides all the necessary ma-
achinery 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 frame-
works 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 pro-
gramming (staging). Modern tools such as LMS (Lightweight Mod-
ular 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 recur-
sion. LMS also supports runtime specialization for further
code generation optimizations. The flavor of LMS is shown
in the following example:

```scala
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.
$\text{Rep}$ is used in LMS to mark staged expressions. The types
$\text{Rep[Vector[String]]}$ and $\text{Rep[String]}$ indicate staged IR con-
struction. The implementation crucially relies on type infer-
ence 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

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

becomes a series of method calls with closure arguments:

```scala
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
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 gen-
eration 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 $\text{Num}$ as follows:

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

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

Given this basic setting of $\text{Rep}$ 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 $\text{Rep[Boolean]}$ condition and two ($\Rightarrow \text{Rep[A]}
@cps[Rep[B]]$) typed parameters for the then- and else-bran-
ces. 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:

```cpp
def fun(f: Rep[A] => Rep[B] @cps[Rep[C]]) = {
    // Stack f onto k (continuation k is extended).
    val f1 = fun(x, k) => reset(k1(x))
    // If tree is empty, return the initial values.
    if (tree.isEmpty) init
    // Otherwise, recurse on subtrees and compose results.
    else h(t(tree.left), f(t(tree.right)))
}
```

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.

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

Below is an example using the TREE construct:

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

Here is the generated C++ code:

```cpp
def TREE(Tree tree, function<double, double>d) = {
    // Otherwise, recurse on subtrees and compose results.
    if (tree.isEmpty) init
    // Otherwise, recurse on subtrees and compose results.
    else h(t(tree.left), d(t(tree.right)))
}
```

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 Script4). 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 s gemm. 32x32x32. N N vec and RNN. elementW ise. fp are 20% slower for Lantern than PyTorch. After eliminating obvious possibilities 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.

---

4https://pytorch.org/docs/master/jit.html
5https://www.cs.toronto.edu/~kriz/cifar.html
6All 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. Dynnet 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  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]...
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 backpropagation 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.

---

9https://github.com/tensorflow/swift

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


