The Differentiable Curry

Martin Abadi  Richard Wei  Gordon Plotkin  Dimitrios Vytiniotis  Dan Belov
Google Brain  DeepMind

1 Introduction

Differentiable programming allows programmers to calculate program gradients and unlocks experimentation with new optimizers and neural network architectures. This is why modern deep learning frameworks [1, 23, 20, 7] introduce derivative APIs (e.g. \texttt{tf.gradients} in TensorFlow). Programmers ask for the gradient of an objective function with respect to its parameters; which is then used to optimize these parameters, e.g. through stochastic gradient descent.

Recent projects, as Swift for TensorFlow (S4TF) (www.tensorflow.org/swift) and Julia Zygote [15], in the spirit of the seminal “Lambda the Ultimate Backpropagator” (LTUB) [21], advocate AD as a first-class construct in a general-purpose programming language, and aim to take advantage of traditional compiler optimizations for efficient code generation.

An important challenge in this setting is the differentiation of functions that accept or return other functions, perhaps capturing (differentiable or non-differentiable) variables. Partial applications must not “forget” to back-propagate to captured variables, and more generally we need AD that provably preserves equational reasoning – needed to justify inlining, common sub-expression elimination etc. As we will see (Section 2), higher-order functions are ubiquitous in modern statically-typed languages, even inside the implementation of end-to-end first-order programs. They have to be tackled heads-on to avoid additional complications in a compiler, such as extra inlining and loop unrolling or early defunctionalization, and to allow for separate compilation, to name a few.

This is the challenge we address. We focus on (i) statically-typed, (ii) compile-time, (ii) reverse-mode AD, a scenario exemplified by Swift AD. (http://bit.ly/swift-autodiff) Our contributions are:

- Following recent work [11] we introduce combinators for rep-functions that also return pullback linear maps (back-propagators), and show how they can be used for AD. Generalizing to higher-order functions boils down to introducing differentiable curry and eval combinators.

- Higher-order functions (like curry) accept or return closures. For a function from tensors to tensors, its pullback at some input is also a linear map from tensors to tensors. We use the term tangent space, denoted as \texttt{Tan t}, for the space of permutations of values of a type \( t \). Hence the tangent space of a tensor is just a tensor. But what should be the tangent space of a function type? Perhaps surprisingly, a function type itself is not the right answer. We provide two possible implementations for function tangents and differentiable currying, and explain the tradeoffs.

- The first implementation – novel to our knowledge – is typeable in a simply-typed discipline and bears similarities to tracing but may involve re-computation of the function we differentiate during the execution of its backwards derivative.

- The second is a dependently typed version of an idea behind “Lambda the Ultimate Backpropagator” [21], itself inspired by the view of functions as closures whose tangents are the tangents of their environments. We put that idea in the combinatory setting of Elliott. Our work implies that an “erased” version that relies on reinterpret casts in a weaker type system is in fact safe.
2 Higher-order functions and AD

Higher-order functions may not seem essential for differentiation but in a general-purpose programming language (e.g. Swift) are actually ubiquitous, even inside the implementation of a first-order program. Consider, for instance, the case of a recurrent neural network (RNN) model. An RNN, in its simplest form, folds a state transformer (called the RNN cell – e.g. a Long Short-Term Memory [14]) through a sequence of inputs and produces a final state. The state is often called the “hidden state” of the RNN:

\[
\text{rnnCell} :: (\text{Params}, \text{Tensor}, \text{Tensor}) \rightarrow \text{Tensor}
\]

\[
\text{rnnCell} \ (\text{params}, \text{hidden} \_ \text{state}, \text{input}) = \ldots \ \text{return} \ \text{new} \ \text{hidden} \_ \text{state}
\]

\[
\text{runRNN} :: \text{Params} \rightsquigarrow \text{Tensor} \rightsquigarrow [\text{Tensor}] \rightsquigarrow \text{Tensor}
\]

\[
\text{runRNN} \ \text{params} \ \text{init} \_ \text{state} \ \text{xs} =
\]

\[
\text{let} \ g :: \text{Tensor} \rightsquigarrow \text{Tensor} \rightsquigarrow \text{Tensor}
\]

\[
\text{in} \ \text{fold} \ g \ \text{init} \_ \text{state} \ \text{xs}
\]

Here function \( g \) is the partial application of \( \text{rnnCell} \) on \( \text{params} \), passed to the recursive function \( \text{fold} \). This example shows many features required for a general purpose language: (i) partial applications, (ii) recursive functions (such as \( \text{fold} \)), (iii) recursive datatypes (such as \([\text{Tensor}] \) above). In particular function \( g \), the partial application of \( \text{rnnCell} \) captures the parameters and – unlike our simple example above – needs to \textit{back-propagate} to these parameters. Moreover, we cannot eliminate higher-order functions by inlining, unless we unroll \( \text{fold} \). And even if we could unroll \( \text{fold} \), \( \text{rnnCell} \) might be imported from a different module whose source is not available for inlining.

In this extended abstract we will only focus on higher-order functions. We will show how we can express differentiation through higher-order programs following Elliott’s recipe by providing implementations of a small set of combinators, given below:

\[
id :: \tau \rightsquigarrow \tau
\]

\[
curry :: ((\tau_a \times \tau_b) \rightsquigarrow \tau_c) \rightarrow (\tau_a \rightsquigarrow (\tau_b \rightsquigarrow \tau_c))
\]

\[
eval :: (\tau \rightsquigarrow \sigma, \tau) \rightsquigarrow \sigma
\]

\[
prod :: (\tau_y \rightsquigarrow \tau_a) \rightarrow (\tau_y \rightsquigarrow \tau_b) \rightarrow (\tau_y \rightsquigarrow (\tau_a, \tau_b))
\]

\[
(\odot) :: (\tau_a \rightsquigarrow \tau_b) \rightarrow (\tau_b \rightsquigarrow \tau_c) \rightarrow (\tau_a \rightsquigarrow \tau_c)
\]

\[
\text{const}_\tau :: \sigma \rightarrow (\tau \rightsquigarrow \sigma)
\]

\[
\text{proj}_i :: (\tau_1 \ldots \tau_n) \rightsquigarrow \tau_i
\]

Unfolding types, we see that \textit{curry/eval} require a definition for function tangents, \( \text{Tang} (\tau \rightsquigarrow \sigma) \). What these should be (and why) is answered in this work.

\[
\frac{\mathcal{J}[\Delta \mapsto e : \tau] = b}{b = \mathcal{J}[\Delta, (x : \tau) \mapsto e : \sigma]} \quad \text{BDLAM}
\]

\[
\frac{\mathcal{J}[\Delta \mapsto \text{diff} x : e : \tau \rightsquigarrow \sigma] = \text{curry} \ b}{b_2 = \mathcal{J}[\Delta \mapsto e_2 : \tau]} \quad \text{BDAPP}
\]

\[
\frac{\mathcal{J}[\Delta \mapsto e_1 : \tau \rightsquigarrow \sigma]}{b_1 = \mathcal{J}[\Delta \mapsto e_1 : \tau_1]} \quad \frac{b_2 = \mathcal{J}[\Delta \mapsto e_2 : \tau_2]}{\mathcal{J}[\Delta \mapsto (e_1, e_2) : (\tau_1, \tau_2)] = \text{prod} \ b_1 b_2} \quad \text{BDPRD}
\]

Figure 1: Translating to combinators

3 Combinatory-style AD

We first illustrate how AD can be implemented using the aforementioned set of combinators.

3.1 Step 1: Translate to combinators

We first use the combinator language in the previous section as the target of conversion from a (conventional) call-by-value higher-order lambda calculus of differentiable functions \( \lambda_0 \). The translation, in Figure 1, has (yet) nothing to do with AD; rather it is reminiscent of the well-understood translation of \( \lambda \)-calculus into cartesian closed categories (CCCs) [8]. \( \mathcal{J}[\Delta \mapsto e : \tau] \) defines such a type-directed translation. The rules ensure that if \( \Delta \mapsto e : \tau \) then \( \mathcal{J}[\Delta \mapsto e : \tau] : \Delta \rightsquigarrow \tau \), where we abuse notation and refer to \( \Delta \) as the tuple of all types of environment variables. The conversion also assumes (not shown, for lack of space) that all differentiable primitives, such as \( (\odot) :: (\text{Float}, \text{Float}) \rightsquigarrow \text{Float} \) come with rep-function implementations.

3.2 Step 2: Implement combinators

Next, we implement our combinators, accepting and returning \( \tau \rightsquigarrow b \) values, i.e. functions of type \( \tau \rightarrow (b, \text{Tang} b \rightarrow \text{Tang} a) \). We also need to define a tangent space \( \text{Tang} \tau \) for every type \( \tau \). Figure 2 presents such a small library. Figure 6 in the Appendix gives the definition of \( \mathcal{T}[\tau] \) for all the types of \( \lambda_0 \). The cases for floats, products, and tensors of floats are standard; also the rules for “discrete” types all return Unit. Some combinators (e.g. \( \text{prod} \)) rely on having 0 and \( (+) \) defined for every tangent type \( \mathcal{T}[\tau] \), also found in Figure 6. In the next sections we proceed to
eval up all the resulting tangents from running our function, track all calls of a function and hence we have to sum these lists intuitively. Addition and zero are given by list concatenation and the empty list, as we see in the highlighted parts of Figure 6. These lists intuitively track all calls of a function and hence we have to sum up all the resulting tangents from running our function forwards and then backwards for every element, arriving at the implementation of curry in Figure 3. The eval combinator merely records the primal value (x) and the output tangent (g) in a singleton list.

4.1 Properties and metatheory

Is our construction correct? We answer by showing that it respects equational reasoning principles. For example, when given \( f : (\tau_1, \tau_2) \rightsquigarrow \tau_3 \) which we can curry and repeatedly evaluate with an argument of type \( \tau_1 \) and another of type \( \tau_2 \), we will get a function that is not only forward-equivalent, but also has an equivalent back-propagator. An example are the forward-equivalent functions foo1 and foo2 in the “Partial Application” column of Figure 4.

Consider foo1 and foo2 in the “Forgetting Results” column of Figure 4. The two functions should be equivalent in forward and reverse mode, but for foo1 we will back-propagate a tangent value of \([x, 0]\) for the use of g. In foo2, since g is not used at all, we will back-propagate \[]. We want to therefore treat \([x, 0]\) and \[] as equivalent, even if they are different lists.

Finally, foo1 and foo2 in the “Summing Results” column are forward-equivalent, hence we expect equivalent back-propagators. In the first case, we call f multiple times with the same argument and sum the results; in the second case we call it once. The tangents that are back-propagated to f in the first case will be \([x, g], (x, g)\) where g is the tangent corresponding to the result of foo1. In the second case we get \([x, g + g]\). We need these two tangents to be treated as equivalent, even if they are different lists.

We have formalized thus a notion of equivalence that goes beyond \(\beta\)-equivalence for back-propagators, and showed various CCC laws hold of our combinators wrt. that equivalence. These laws guarantee equivalences for the examples presented in this section.

5 Dependently-typed curry

Unfortunately the differentiable curry in Figure 3 has a back-propagator new_pb that involves a full forward computation of the original function f, at each of the recorded inputs it was applied to – hence suffers from redundant computation. We need something better.

A key insight from the LTUB work [21], leading to an efficient solution, has been this: a back-propagator for a function \( \tau \rightsquigarrow \sigma \) should take as argument a value of type \( \mathcal{T}[\sigma] \) but return not only tangent \( \mathcal{T}[\tau] \) but also the tangent of the environment \( \Delta \) over which the function closed when it was constructed; \( \mathcal{T}[\Delta] \). To understand the intuition, it’s helpful to think of
Forgetting Results

We define where we immediately hit a problem: it is no longer the case that composition (also in Figure 5), simply collects together environment tangents. These need not be maps from variables to tangents, rather just tuples.

In Figure 5 we give such a curry and eval, in a non-opinionated dependent-type notation (we also have produced in Agda and Coq). The reader is urged to examine the code, but ignore the type signatures, resting assured, it all type checks. Another remark is that composition (also in Figure 5), simply collects together environment tangents. These need not be maps from variables to tangents, rather just tuples.

We have showed that dependently typed currying satisfies similar laws as the simply-typed version. As a final remark, most production languages do not support dependent types. There our solution can be safely implemented using reinterpret casts. This is, in

<table>
<thead>
<tr>
<th></th>
<th>Partial Application</th>
<th>Forgetting Results</th>
<th>Summing Results</th>
</tr>
</thead>
<tbody>
<tr>
<td>f :: (Float, Float) ↠ Float</td>
<td>foo1, foo2 :: (Float ↠ Float, Float ↠ Float)</td>
<td>foo1 :: (Float ↠ Float) ↠ Float</td>
<td>foo1 f x = f x + f x</td>
</tr>
<tr>
<td>foo1 (a, b) = (λxb → f (a, xb)) b</td>
<td>foo1 (f, g) x = fst (f x, g x)</td>
<td>foo2 (a, b) = f (a, b)</td>
<td>foo2 f x = let y = f x in (y + y)</td>
</tr>
<tr>
<td>foo2 (a, b) = f (a, b)</td>
<td>foo2 (f, g) x = f x</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 4: Example equivalences (foo1 and foo2 in each column)

Figure 5: Dependently-typed differentiable curry/eval

curry :: ((τ₁, τ₂) ↠ τ₃) ↠ (τ₁ ↠ (τ₂ ↠ τ₃))
curry (exT τ₃ f) = exT () new_f

where
new_f :: Π(t:τ₁). Σ(g: τ₂ ↠ τ₃). T[g] → (τ₃, T[t])
new_f t = let gf :: Π(s:τ₁). Σ(r:τ₂). T[r] → ((τ₃, T[t]), T[s])
g f s = let (r, pullback) = f(t,s)
in (r, λgr →
in ((cte, ctt, cts))
g = exT (τ₃, T[t]) gb
new_pb :: T[g] → (τ₃, T[t])
new_pb env = env
in (g, new_pb)
eval :: (τ ↠ σ, τ ↠ σ)
eval = exT () new_f

where new_f (exT τ₆ f, x) = let (y, pb) = f x
in (y, λg → ((l), pb g))
comp :: (a ↠ b) → (b ↠ c) → (a ↠ c)
comp (exT τ₆ f) (exT τ₇ g) = exT (τ₇, τ₈) h
where h a = let (b, pb_f) = f a
(c, pb_gb) = g b
in (c, λgc → let (envg, gb) = pb_gb gc
(envf, ga) = pb_gb gb
in ((envf, envg), ga)
6 Discussion and extensions

6.1 Internalizing differentiation

We have not yet described how to calculate vector-Jacobian products. Indeed, rep-functions are ordinary functions returning linear maps, and our $\lambda_\theta$ of Section 3 does not include ordinary ($\rightarrow$), $\lambda$, or applications. As a result, it cannot type vjp (Figure 2)! We believe there is a principled way to integrate ordinary and differentiable arrows, while still preserving equational reasoning, out of scope for this abstract.

6.2 Further features

By taking the same approach of (i) compiling to combinators, and (ii) implementing these combinators in terms of a core language we can show how to introduce control flow and recursion into a differentiable programming language. Supporting richer algebraic types may also be possible, through user-specified definitions for the tangent spaces of these types.

7 Related work

We presented the marriage of ideas behind Elliot’s categorical presentation of AD [11] and the seminal LTUB work [21]. We extend Elliot’s presentation to differentiable currying and evaluation, while putting the ideas of Pearlmutter and Siskind in a typed setting.

The idea of using closures as back-propagators is receiving recent attention. For example Julia Zygote [15] and Swift AD adopts this design. Other recent work follows similar ideas [27, 26] but is using meta-programming as an implementation technique.

There exists work on differentiation semantics [24] and differentiable categories [5], usually interpreting types as vector spaces. Related ideas have appeared for higher-order lambda calculi [18, 9] but the underlying foundations only tackle forward mode.

AD has a long history in array programming and scientific computing [13]. Forward-mode AD has been presented before for (first-order) functional programs [16, 10], as libraries in general purpose languages [3], DSLs [6], and more. We urge the reader to consult the comprehensive survey [2]. Recent systems revisit efficient differentiable array programming [22, 25]. There exist also widely used AD libraries for Python [17], and new systems that target deep learning applications [12] – the latter supporting variable capture through early defunctionalization.

References


1 Using the erased AnyDerivative struct, see https://www.tensorflow.org/swift/api_docs/Structs/AnyDerivative.


A Appendix

\[
\begin{align*}
\mathcal{T}[\text{Float}] & = \text{Float} \\
\mathcal{T}[\text{Tensor<Float>}] & = \text{Tensor<Float>} \\
\mathcal{T}[\{\tau,\sigma\}] & = (\mathcal{T}[\tau], \mathcal{T}[\sigma]) \\
\mathcal{T}[\tau \rightarrow \sigma] & = [(\tau, \mathcal{T}[\sigma])] \\
\mathcal{T}[\text{Unit}] & = \text{Unit} \\
\mathcal{T}[\text{Tensor} \text{Int}] & = \text{Unit} \\
\mathcal{T}[\text{Int}] & = \text{Unit} \\
\mathcal{T}[\text{Bool}] & = \text{Unit} \\
0\mathcal{T}[\text{Float}] & = 0 \\
0\mathcal{T}[\text{Tensor<Float>}] & = 0 \\
0\mathcal{T}[\{\tau,\sigma\}] & = (0\mathcal{T}[\tau], 0\mathcal{T}[\sigma]) \\
0\mathcal{T}[\tau \rightarrow \sigma] & = [] \\
0\mathcal{T}[\tau] & = () \\
X_1 + \mathcal{T}[\text{Float}] X_2 & = X_1 + X_2 \\
X_1 + \mathcal{T}[\text{Tensor<Float>}] X_2 & = X_1 + X_2 \\
\{X_1, X_2\} + \mathcal{T}[\{\tau,\sigma\}] \{X_21, X_22\} & = \{X_1 + \mathcal{T}[\tau]X_21, X_21 + \mathcal{T}[\sigma]X_22\} \\
X_1 + \mathcal{T}[\tau \rightarrow \sigma] X_2 & = X_1 \leftarrow \leftarrow X_2
\end{align*}
\]

Figure 6: Tangent spaces for \( \lambda_0 \) (simply-typed)