The simple essence of automatic differentiation

Differentiable programming made easy

Conal Elliott

Target

November 2018

Differentiable programming made easy

Current AI revolution runs on large data, speed, and AD.

Differentiable programming made easy

Current AI revolution runs on large data, speed, and AD, but

- AD algorithm (backprop) is complex and stateful.
- Complex graph APIs.

Differentiable programming made easy

Current AI revolution runs on large data, speed, and AD, but

- AD algorithm (backprop) is complex and stateful.
- Complex graph APIs.

Solutions:

- AD: Simple, calculated, efficient, parallel-friendly, generalized.
- API: *derivative*.

What's a derivative?

What's a derivative?

- Number
- Vector
- Covector
- Matrix
- Higher derivatives

What's a derivative?

- Number
- Vector
- Covector
- Matrix
- Higher derivatives

Chain rule for each.

Derivatives as linear maps (Fréchet)

$$\mathcal{D} :: (a \to b) \to (a \to (a \multimap b))$$

$\mathcal{D}\ f\ \ a$ is a local *linear* approximation to f at a.

Derivatives as linear maps (Fréchet)

$$\mathcal{D} :: (a \to b) \to (a \to (a \multimap b))$$

$\mathcal{D}\ f\ a$ is a local *linear* approximation to f at a:

$$\lim_{\varepsilon \to 0} \frac{\| f\ (a + \varepsilon) - (f\ a + \mathcal{D}\ f\ a\ \varepsilon) \|}{\| \varepsilon \|} = 0$$

Composition

Sequential:

$$(\circ) :: (b \rightarrow c) \rightarrow (a \rightarrow b) \rightarrow (a \rightarrow c)$$
$$(g \circ f)\ a = g\ (f\ a)$$

$$\mathcal{D}\ (g \circ f)\ a = \mathcal{D}\ g\ (f\ a) \circ \mathcal{D}\ f\ a \quad \text{-- chain rule}$$

Composition

Sequential:

$$(\circ) :: (b \to c) \to (a \to b) \to (a \to c)$$
$$(g \circ f)\ a = g\ (f\ a)$$

$$\mathcal{D}\ (g \circ f)\ a = \mathcal{D}\ g\ (f\ a) \circ \mathcal{D}\ f\ a \quad \text{-- chain rule}$$

Parallel:

$$(\vartriangle) :: (a \to c) \to (a \to d) \to (a \to c \times d)$$
$$(f \vartriangle g)\ a = (f\ a, g\ a)$$

$$\mathcal{D}\ (f \vartriangle g)\ a = \mathcal{D}\ f\ a \vartriangle \mathcal{D}\ g\ a$$

Linear functions

Linear functions

Linear functions are their own derivatives everywhere.

$$
\begin{aligned}
\mathcal{D}\ \mathit{id}\ \ \ a &= \mathit{id} \\
\mathcal{D}\ \mathit{fst}\ \ a &= \mathit{fst} \\
\mathcal{D}\ \mathit{snd}\ a &= \mathit{snd} \\
&\ldots
\end{aligned}
$$

Compositionality

Chain rule:

$$\mathcal{D}\ (g \circ f)\ a = \mathcal{D}\ g\ (f\ a) \circ \mathcal{D}\ f\ a \quad \text{-- non-compositional}$$

Compositionality

Chain rule:

$$\mathcal{D}\ (g \circ f)\ a = \mathcal{D}\ g\ (f\ a) \circ \mathcal{D}\ f\ a \quad \text{-- non-compositional}$$

To fix, combine regular result with derivative:

$$\hat{\mathcal{D}} :: (a \to b) \to (a \to (b \times (a \multimap b)))$$
$$\hat{\mathcal{D}}\ f = f \vartriangle \mathcal{D}\ f \quad \text{-- specification}$$

Often much work in common to f and $\mathcal{D}\ f$.

Abstract algebra for functions

$$
\begin{array}{l}
\textbf{class } \mathit{Category}\ (\leadsto) \textbf{ where} \\
\quad \mathit{id}\ \ :: a \leadsto a \\
\quad (\circ) :: (b \leadsto c) \to (a \leadsto b) \to (a \leadsto c)
\end{array}
$$

$$
\begin{array}{l}
\textbf{class } \mathit{Category}\ (\leadsto) \Rightarrow \mathit{Cartesian}\ (\leadsto) \textbf{ where} \\
\quad \mathit{exl}\ :: (a \times b) \leadsto a \\
\quad \mathit{exr} :: (a \times b) \leadsto b \\
\quad (\vartriangle)\ :: (a \leadsto c) \to (a \leadsto d) \to (a \leadsto (c \times d))
\end{array}
$$

Plus laws and classes for arithmetic etc.

Compiling to categories

$$sqr\ a = a * a$$

$$magSqr\ (a, b) = sqr\ a + sqr\ b$$

$$cosSinProd\ (x, y) = (cos\ z, sin\ z)\ \mathbf{where}\ z = x * y$$

Compiling to categories

$$sqr\ a = a * a$$

$$magSqr\ (a, b) = sqr\ a + sqr\ b$$

$$cosSinProd\ (x, y) = (cos\ z, sin\ z)\ \mathbf{where}\ z = x * y$$

In categorical vocabulary:

$$sqr = mul \circ (id \vartriangle id)$$

$$magSqr = add \circ ((sqr \circ exl) \vartriangle (sqr \circ exr))$$

$$cosSinProd = (cos \vartriangle sin) \circ mul$$

Automated translation & generalization. See ICFP 2017 paper.

Automatic differentiation (specification)

$$\textbf{newtype}\ D\ a\ b = D\ (a \to b \times (a \multimap b))$$

$$\hat{\mathcal{D}} :: (a \to b) \to D\ a\ b$$
$$\hat{\mathcal{D}}\ f = D\ (f \vartriangle \mathcal{D}\ f) \quad \text{-- not computable}$$

Automatic differentiation (specification)

$$\textbf{newtype}\ D\ a\ b = D\ (a \to b \times (a \multimap b))$$

$$\hat{\mathcal{D}} :: (a \to b) \to D\ a\ b$$
$$\hat{\mathcal{D}}\ f = D\ (f \vartriangle \mathcal{D}\ f) \quad \text{-- not computable}$$

Specification: $\hat{\mathcal{D}}$ preserves *Category* and *Cartesian* structure:

$$\hat{\mathcal{D}}\ id = id$$
$$\hat{\mathcal{D}}\ (g \circ f) = \hat{\mathcal{D}}\ g \circ \hat{\mathcal{D}}\ f$$

$$\hat{\mathcal{D}}\ exl = exl$$
$$\hat{\mathcal{D}}\ exr = exr$$
$$\hat{\mathcal{D}}\ (f \vartriangle g) = \hat{\mathcal{D}}\ f \vartriangle \hat{\mathcal{D}}\ g$$

The game: solve these equations for the RHS operations.

Automatic differentiation (solution)

$$\textbf{newtype}\ D\ a\ b = D\ (a \to b \times (a \multimap b))$$

$$linearD\ f = D\ (\lambda a \to (f\ a, f))$$

$\textbf{instance}\ Category\ D\ \textbf{where}$

$\quad id = linearD\ id$

$\quad D\ g \circ D\ f = D\ (\lambda a \to \textbf{let}\ \{(b, f') = f\ a; (c, g') = g\ b\}\ \textbf{in}\ (c, g' \circ f'))$

$\textbf{instance}\ Cartesian\ D\ \textbf{where}$

$\quad exl\ = linearD\ exl$

$\quad exr = linearD\ exr$

$\quad D\ f \vartriangle D\ g = D\ (\lambda a \to \textbf{let}\ \{(b, f') = f\ a; (c, g') = g\ a\}\ \textbf{in}\ ((b, c), f' \vartriangle g'))$

$\textbf{instance}\ NumCat\ D\ \textbf{where}$

$\quad negate = linearD\ negate$

$\quad add = linearD\ add$

$\quad mul = D\ (mul \vartriangle (\lambda(a, b) \to \lambda(da, db) \to b * da + a * db))$

Generalizing AD

$$\textbf{newtype}\ D\ a\ b = D\ (a \to b \times (a \multimap b))$$

$$linearD\ f = D\ (\lambda a \to (f\ a, f))$$

$$
\begin{aligned}
&\textbf{instance}\ Category\ D\ \textbf{where} \\
&\quad id = linearD\ id \\
&\quad D\ g \circ D\ f = D\ (\lambda a \to \textbf{let}\ \{(b, f') = f\ a; (c, g') = g\ b\}\ \textbf{in}\ (c, g' \circ f'))
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance}\ Cartesian\ D\ \textbf{where} \\
&\quad exl\ = linearD\ exl \\
&\quad exr = linearD\ exr \\
&\quad D\ f \vartriangle D\ g = D\ (\lambda a \to \textbf{let}\ \{(b, f') = f\ a; (c, g') = g\ a\}\ \textbf{in}\ ((b, c), f' \vartriangle g'))
\end{aligned}
$$

Each D operation just uses corresponding $(\multimap)$ operation.

Generalize from $(\multimap)$ to other cartesian categories.

Generalized AD

$$\mathbf{newtype}\ D_{(\leadsto)}\ a\ b = D\ (a \to b \times (a \leadsto b))$$

$$linearD\ f\ f' = D\ (\lambda a \to (f\ a, f'))$$

$$\begin{array}{l}\mathbf{instance}\ Category\ (\leadsto) \Rightarrow Category\ D_{(\leadsto)}\ \mathbf{where} \\ \quad id = linearD\ id\ id \\ \quad D\ g \circ D\ f = D\ (\lambda a \to \mathbf{let}\ \{(b, f') = f\ a; (c, g') = g\ b\}\ \mathbf{in}\ (c, g' \circ f'))\end{array}$$

$$\begin{array}{l}\mathbf{instance}\ Cartesian\ (\leadsto) \Rightarrow Cartesian\ D_{(\leadsto)}\ \mathbf{where} \\ \quad exl\ = linearD\ exl\ exl \\ \quad exr = linearD\ exr\ exr \\ \quad D\ f \vartriangle D\ g = D\ (\lambda a \to \mathbf{let}\ \{(b, f') = f\ a; (c, g') = g\ a\}\ \mathbf{in}\ ((b, c), f' \vartriangle g'))\end{array}$$

$$\begin{array}{l}\mathbf{instance}\ \ldots \Rightarrow NumCat\ D\ \mathbf{where} \\ \quad negate = linearD\ negate\ negate \\ \quad add\ = linearD\ add\ add \\ \quad mul = ??\end{array}$$

Numeric operations

Specific to (linear) *functions*:

$$mul = D\ (mul \vartriangle (\lambda(a, b) \rightarrow \lambda(da, db) \rightarrow b * da + a * db))$$

Numeric operations

Specific to (linear) *functions*:

$$mul = D\ (mul \vartriangle (\lambda(a, b) \rightarrow \lambda(da, db) \rightarrow b * da + a * db))$$

Rephrase:

$$\begin{aligned}
&scale :: Multiplicative\ a \Rightarrow a \rightarrow (a \rightarrow a)\\
&scale\ u = \lambda v \rightarrow u * v
\end{aligned}$$

$$\begin{aligned}
&(\triangledown) :: (a \rightarrow c) \rightarrow (b \rightarrow c) \rightarrow ((a \times b) \rightarrow c)\\
&(f \triangledown g)\ (a, b) = f\ a + g\ b
\end{aligned}$$

Now

$$mul = D\ (mul \vartriangle (\lambda(a, b) \rightarrow scale\ b \triangledown scale\ a))$$

New generalized vocabulary

$$
\begin{array}{l}
\textbf{class } \mathit{Category}\ (\leadsto) \Rightarrow \mathit{Cocartesian}\ (\leadsto)\ \textbf{where} \\
\quad \mathit{inl}\ :: a \leadsto (a \times b) \\
\quad \mathit{inr} :: b \leadsto (a \times b) \\
\quad (\triangledown) :: (a \leadsto c) \to (b \leadsto c) \to ((a \times b) \leadsto c)
\end{array}
$$

$$
\begin{array}{l}
\textbf{class } \mathit{ScalarCat}\ (\leadsto)\ a\ \textbf{where} \\
\quad \mathit{scale} :: a \to (a \leadsto a)
\end{array}
$$

Differentiation:

$$\mathcal{D}\ (f \triangledown g)\ (a, b) = \mathcal{D}\ f\ a \triangledown \mathcal{D}\ g\ b$$

The rest are linear.

Linear transformations as functions

$$
\begin{array}{l}
\textbf{newtype}\ a \overset{+}{\rightarrow} b = \mathit{AddFun}\ (a \rightarrow b)
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Category}\ (\overset{+}{\rightarrow})\ \textbf{where} \\
\quad \mathit{id}\ \ = \mathit{AddFun}\ \mathit{id} \\
\quad (\circ) = \mathit{inNew}_2\ (\circ)
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Cartesian}\ (\overset{+}{\rightarrow})\ \textbf{where} \\
\quad \mathit{exl}\ = \mathit{AddFun}\ \mathit{exl} \\
\quad \mathit{exr} = \mathit{AddFun}\ \mathit{exr} \\
\quad (\vartriangle)\ = \mathit{inNew}_2\ (\vartriangle)
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Cocartesian}\ (\overset{+}{\rightarrow})\ \textbf{where} \\
\quad \mathit{inl}\ = \mathit{AddFun}\ (\lambda a \rightarrow (a, 0)) \\
\quad \mathit{inr} = \mathit{AddFun}\ (\lambda b \rightarrow (0, b)) \\
\quad (\triangledown) = \mathit{inNew}_2\ (\lambda f\ g\ (a, b) \rightarrow f\ a + g\ b)
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Multiplicative}\ s \Rightarrow \mathit{ScalarCat}\ (\overset{+}{\rightarrow})\ s\ \textbf{where} \\
\quad \mathit{scale}\ s = \mathit{AddFun}\ (s\ *)
\end{array}
$$

Extracting a data representation

- Finally, extract a matrix or gradient vector.
- Very inefficient for gradient-based optimization!
- Alternatively, represent as "generalized matrices" ($M_s\ a\ b$). Then solve more homomorphisms.

Efficiency of composition

- Composition is associative.
- Some associations are more efficient than others, so
  - Associate optimally.
  - Equivalent to *matrix chain multiplication* — $O(n \log n)$.
  - Choice determined by *types*, i.e., compile-time information.

Efficiency of composition

- Composition is associative.
- Some associations are more efficient than others, so
  - Associate optimally.
  - Equivalent to *matrix chain multiplication* — $O(n \log n)$.
  - Choice determined by *types*, i.e., compile-time information.
- All right: "forward mode AD" (FAD).
- All left: "reverse mode AD" (RAD).
- RAD is *much* better for gradient-based optimization.

Left-associating composition (RAD)

CPS-like category:

- Represent $a \leadsto b$ by $(b \leadsto r) \to (a \leadsto r)$.
- Meaning: $f' \mapsto (\lambda h \to h \circ f')$.
- Construct $h \circ \mathcal{D}\, f\ \ a$ directly, without $\mathcal{D}\, f\ \ a$.

Old technique (Cayley 1854), vastly generalized by Yoneda.

Continuation category (specification)

$$\textbf{newtype}\ \mathit{Cont}^{r}_{(\leadsto)}\ a\ b = \mathit{Cont}\ ((b \leadsto r) \rightarrow (a \leadsto r))$$

$$\begin{aligned}&\mathit{cont} :: \mathit{Category}\ (\leadsto) \Rightarrow (a \leadsto b) \rightarrow \mathit{Cont}^{r}_{(\leadsto)}\ a\ b\\&\mathit{cont}\ f = \mathit{Cont}\ (\circ\, f)\end{aligned}$$

Require *cont* to preserve structure. Solve for methods.

Continuation category (solution)

instance $\mathit{Category}\ (\leadsto) \Rightarrow \mathit{Category}\ \mathit{Cont}^{r}_{(\leadsto)}$ **where**

$\mathit{id} = \mathit{Cont}\ \mathit{id}$

$\mathit{Cont}\ g \circ \mathit{Cont}\ f = \mathit{Cont}\ (f \circ g)$

instance $\mathit{Cartesian}\ (\leadsto) \Rightarrow \mathit{Cartesian}\ \mathit{Cont}^{r}_{(\leadsto)}$ **where**

$\mathit{exl} = \mathit{Cont}\ (\mathit{join} \circ \mathit{inl})$

$\mathit{exr} = \mathit{Cont}\ (\mathit{join} \circ \mathit{inr})$

$(\vartriangle) = \mathit{inNew}_2\ (\lambda f\ g \rightarrow (f \triangledown g) \circ \mathit{unjoin})$

instance $\mathit{Cocartesian}\ (\leadsto) \Rightarrow \mathit{Cocartesian}\ \mathit{Cont}^{r}_{(\leadsto)}$ **where**

$\mathit{inl} = \mathit{Cont}\ (\mathit{exl} \circ \mathit{unjoin})$

$\mathit{inr} = \mathit{Cont}\ (\mathit{exr} \circ \mathit{unjoin})$

$(\triangledown) = \mathit{inNew}_2\ (\lambda f\ g \rightarrow \mathit{join} \circ (f \vartriangle g))$

instance $\mathit{ScalarCat}\ (\leadsto)\ a \Rightarrow \mathit{ScalarCat}\ \mathit{Cont}^{r}_{(\leadsto)}\ a$ **where**

$\mathit{scale}\ s = \mathit{Cont}\ (\mathit{scale}\ s)$

Reverse-mode AD without tears

$$D_{Cont^{r}_{M_s}}$$

Duality

- Vector space dual: $u \multimap s$, with u a vector space over s.
- If u has finite dimension, then $u \multimap s \cong u$.
- Represent $a \multimap b$ by $(b \multimap s) \to (a \multimap s)$ by $b \to a$.
- *Ideal* for extracting gradient vector. Just apply to 1 (id).

Duality (specification)

$$\textbf{newtype}\ \mathit{Dual}_{(\leadsto)}\ a\ b = \mathit{Dual}\ (b \leadsto a)$$

$$\begin{aligned}
&\mathit{asDual} :: \mathit{Cont}^{s}_{(\leadsto)}\ a\ b \to \mathit{Dual}_{(\leadsto)}\ a\ b \\
&\mathit{asDual}\ (\mathit{Cont}\ f) = \mathit{Dual}\ (\mathit{dot}^{-1} \circ f \circ \mathit{dot})
\end{aligned}$$

where

$$\begin{aligned}
&\mathit{dot} && :: u \to (u \multimap s) \\
&\mathit{dot}^{-1} && :: (u \multimap s) \to u
\end{aligned}$$

Require *asDual* to preserve structure. Solve for methods.

Duality (solution)

$\textbf{newtype}\ \mathit{Dual}_{(\leadsto)}\ a\ b = \mathit{Dual}\ (b \leadsto a)$

$\textbf{instance}\ \mathit{Category}\ (\leadsto) \Rightarrow \mathit{Category}\ \mathit{Dual}_{(\leadsto)}\ \textbf{where}$

$\quad \mathit{id} \ = \mathit{Dual}\ \mathit{id}$

$\quad (\circ) = \mathit{inNew}_2\ (\mathit{flip}\ (\circ))$

$\textbf{instance}\ \mathit{Cocartesian}\ (\leadsto) \Rightarrow \mathit{Cartesian}\ \mathit{Dual}_{(\leadsto)}\ \textbf{where}$

$\quad \mathit{exl} \ = \mathit{Dual}\ \mathit{inl}$

$\quad \mathit{exr} = \mathit{Dual}\ \mathit{inr}$

$\quad (\vartriangle) \ = \mathit{inNew}_2\ (\triangledown)$

$\textbf{instance}\ \mathit{Cartesian}\ (\leadsto) \Rightarrow \mathit{Cocartesian}\ \mathit{Dual}_{(\leadsto)}\ \textbf{where}$

$\quad \mathit{inl} \ = \mathit{Dual}\ \mathit{exl}$

$\quad \mathit{inr} = \mathit{Dual}\ \mathit{exr}$

$\quad (\triangledown) \ = \mathit{inNew}_2\ (\vartriangle)$

$\textbf{instance}\ \mathit{ScalarCat}\ (\leadsto)\ s \Rightarrow \mathit{ScalarCat}\ \mathit{Dual}_{(\leadsto)}\ s\ \textbf{where}$

$\quad \mathit{scale}\ s = \mathit{Dual}\ (\mathit{scale}\ s)$

Backpropagation

Backpropagation

$$D_{Dual_{\multimap^{+}}}$$

Conclusions

- Simple AD algorithm, specializing to forward, reverse, mixed.
- No graphs, tapes, tags, partial derivatives, or mutation.
- Parallel-friendly and low memory use.
- Calculated from simple, regular algebra problems.
- Generalizes to derivative categories other than linear maps.
- Differentiate regular Haskell code (via plugin).
- ICFP 2018 paper: pictures, proofs, incremental computation.

Reflections: recipe for success

Reflections: recipe for success

Key principles:

- Capture main concepts as first-class values.
- Focus on abstract notions, not specific representations.
- Calculate efficient implementation from simple specification.

Not previously applied to AD (afaik).

Reflections: recipe for success

Key principles:

- Capture main concepts as first-class values.
- Focus on abstract notions, not specific representations.
- Calculate efficient implementation from simple specification.

Not previously applied to AD (afaik).

Quandary: Most programming languages poor for function-like things.

Reflections: recipe for success

Key principles:

- Capture main concepts as first-class values.
- Focus on abstract notions, not specific representations.
- Calculate efficient implementation from simple specification.

Not previously applied to AD (afaik).

Quandary: Most programming languages poor for function-like things.

Solution: Compiling to categories.

Symbolic vs automatic differentiation

Often described as opposing techniques:

- *Symbolic*:
  - Apply differentiation rules symbolically.
  - Can duplicate much work.
  - Needs algebraic manipulation.
- *Automatic*:
  - FAD: easy to implement but often inefficient.
  - RAD: efficient but tricky to implement.

Symbolic vs automatic differentiation

Often described as opposing techniques:

- *Symbolic*:
  - Apply differentiation rules symbolically.
  - Can duplicate much work.
  - Needs algebraic manipulation.
- *Automatic*:
  - FAD: easy to implement but often inefficient.
  - RAD: efficient but tricky to implement.

My view: *AD is SD done by a compiler.*

Compilers already work symbolically and preserve sharing.