An Introduction to Differentiable Programming

Damiano Mazza
CNRS, LIPN, Université Sorbonne Paris Nord

FoPSS, Bertinoro, 13–14 February 2023
Remember derivatives?

\[ f : \mathbb{R} \rightarrow \mathbb{R} \]

\[ f'(x) := \lim_{h \to 0} \frac{f(x + h) - f(x)}{h} \]

Thought of as an operator \((-)': (\mathbb{R} \rightarrow \mathbb{R}) \rightarrow \mathbb{R} \rightarrow \mathbb{R}\)

\[
(\alpha f + \beta g)'(x) = \alpha f'(x) + \beta g'(x) \\
(fg)'(x) = f(x)g'(x) + g(x)f'(x) \\
(f \circ g)'(x) = f'(g(x))g'(x)
\]
Remember gradients?

\[ f : \mathbb{R}^n \to \mathbb{R} \]

Gradient:

\[ \nabla f : \mathbb{R}^n \to \mathbb{R}^n \]

\[ \partial_i f(x_1, \ldots, x_n) := \lim_{h \to 0} \frac{f(x_1, \ldots, x_i + h, \ldots, x_n) - f(x_1, \ldots, x_n)}{h} \]

\[ \nabla f = (\partial_1 f, \ldots, \partial_n f) \]
Why gradients?

Gradient descent!
Target function \( f : \mathbb{R}^m \to \mathbb{R}^p \), training set \( X \subseteq_{\text{fin}} \mathbb{R}^m \).
Parametric approximation \( g : \mathbb{R}^{n+m} \to \mathbb{R}^p \).
Want to find \( \vec{w}_{\text{opt}} \in \mathbb{R}^n \) such that \( g(\vec{w}_{\text{opt}}) \approx f \).
Define
\[
e(\vec{w}) := \frac{1}{|X|} \sum_{\vec{x} \in X} \| g(\vec{w}, \vec{x}) - f(\vec{x}) \|^2 : \mathbb{R}^n \to \mathbb{R}
\]
For initial \( \vec{w}_0 \in \mathbb{R}^n \), step \( \eta < 0 \), and \( i \in \mathbb{N} \), set
\[
\vec{w}_{i+1} := \vec{w}_i + \eta \nabla e(\vec{w}_i)
\]
When \( e(\vec{w}_i) \) is close to zero, set \( \vec{w}_{\text{opt}} := \vec{w}_i \).
Guaranteed to happen under convexity assumptions on \( e \).
Automatic Differentiation (AD)

- In machine learning (ML), $g$ is computed by a neural network (NN).

- In their simplest form, these are layers of neurons

$$
\theta(x_1, \ldots, x_k) := \sigma \left( \sum_{i=1}^{k} w_i x_i \right) : \mathbb{R}^k \rightarrow \mathbb{R}
$$

where $\sigma$ is some activation function.

- $n =$ number of weights in the net. (These days it can easily be $10^8$ or $10^9$).

AD = methods for automatically computing gradients of functions specified by a computer program (e.g. the loss function of a neural network).
Differentiable Programming

- In recent years (from 2015-2016), NN architectures used in ML started becoming more and more complex.

- I.e., $g$ is computed by more and more sophisticated programs.

- Need programming languages with a built-in engine for efficiently computing derivatives/gradients/Jacobians.

differentiable programming = programming languages + AD
Two wrong ideas

- Approximate the definition:

\[ f'(x) := \frac{f(x + h) - f(x)}{h} \]

with \( h \) very small.

- How do we choose \( h \)?
- Contains two “deadly sins” of numerical computation.

- Symbolic computation (like in Mathematica).

- Good idea, but needs to be extended to programs.
- Inefficient, needs sharing.
Dual numbers

• The commutative ring of dual numbers is defined as

\[ \hat{\mathbb{R}} := \mathbb{R}[\varepsilon]/\langle \varepsilon^2 \rangle \]

• Its elements are pairs \((a, a') \in \mathbb{R}^2\), with

\[
\begin{align*}
0 &= (0, 0) \\
(a, a') + (b, b') &= (a + b, a' + b') \\
1 &= (1, 0) \\
(a, a')(b, b') &= (ab, a'b + ab')
\end{align*}
\]
Dual numbers and derivatives

• If $f : \mathbb{R} \to \mathbb{R}$ is differentiable, we extend it to $\hat{f} : \hat{\mathbb{R}} \to \hat{\mathbb{R}}$ as follows:

$$\hat{f}(a, a') := (f(a), f'(a)a')$$

• Using the chain rule, we have

$$\hat{f} \circ g(a, a') = (f(g(a)), a'(f \circ g)'(a)) = (f(g(a)), f'(g(a))g'(a)a')$$

$$= \hat{f}(g(a), g'(a)a') = \hat{f}(\hat{g}(a, a'))$$

$$\hat{id}(a, a') = (a, \hat{id}'(a)a') = (a, 1 \cdot a') = (a, a')$$

• Furthermore, notice that

$$\hat{f}(a, 1) = (f(a), f'(a))$$
Dual numbers and AD for unary straight-line programs

- The above suggests the following program transformation:

```python
def f(x):
def df(x):
    dx = 1
    dz1 = dx * df1(x)
    z1 = f1(x)
    z2 = f2(z1)  #
    dz2 = dz1 * df2(z1)
    ...
    return g(zn)
    return dzn * dg(zn)
```

- Exact computation.

- Preserves the complexity (modulo a factor of 3).
Dual numbers and partial derivatives

• If \( f : \mathbb{R}^n \to \mathbb{R} \) is differentiable, define \( \hat{f} : \hat{\mathbb{R}}^n \to \hat{\mathbb{R}} \) as

\[
\hat{f}(a_1, a_1', \ldots, a_n, a_n') := (f(\bar{a}), \nabla f(\bar{a}) \cdot \bar{a}')
\]

\[
= \left( f(a_1, \ldots, a_n), \sum_{i=1}^{n} a_i' \partial_i f(a_1, \ldots, a_n) \right)
\]

• We still have

\[
f \circ (\hat{g}_1, \ldots, \hat{g}_n) = \hat{f} \circ (\hat{g}_1, \ldots, \hat{g}_n)
\]

• Furthermore, we have

\[
\hat{f}(a_1, 0, \ldots, a_i, 1, \ldots, a_n, 0) = (f(\bar{a}), \partial_i f(\bar{a}))
\]
Dual numbers and AD for straight-line programs

The above transformation generalizes to

```python
def f(x1,...,xn):
    ...
    z = g(w1,...,wk)
    ...
    return y
```

```python
def df(i,x1,...,xn):
    dx1,...,dxi,...,dxn = 0,...,1,...0
    ...
    z = g(w1,...,wk)
    dz = dw1 * dg(1,w1,...,wk) + ... + dwk * dg(k,w1,...,wk)
    ...
    return dy
```

- Still exact computation, still complexity-preserving.
- Covers the case of loss functions of NNs.
- However, inefficient for gradients: requires $n$ passes.
Dual numbers and AD for arbitrary programs

\[
\{ z = g(w_1, \ldots, w_k) \} := \begin{cases} 
  dz = dw_1 \cdot dg(1, w_1, \ldots, w_k) + \ldots + dw_k \cdot dg(k, w_1, \ldots, w_k) 
\end{cases}
\]

\[
\begin{align*}
  \text{if } x \leq 0: & \quad \text{if } x \leq 0: \\
  \quad \text{else:} & \quad \quad \text{else:} \\
  \quad \quad \quad \text{<code1>} & \quad \quad \quad \text{<code1>}
\end{align*}
\]

\[
\{ \text{while } x \leq 0: \} := \begin{cases} 
  \text{<code>} 
\end{cases}
\]

\[
\text{def } f(x_1, \ldots, x_n): \\
  \text{<code>} \\
  \text{return } y
\]

\[
\text{def } df(i, x_1, \ldots, x_n): \\
  \text{<code>} \\
  \text{return } y
\]

\[
\text{def } df(i, x_1, \ldots, x_n): \\
  dx_1, \ldots, dx_i, \ldots, dx_n = 0, \ldots, 1, \ldots, 0 \\
  \text{<code>} \\
  \text{return } dy
\]

\textbf{Theorem (Joss 1976).} \textit{Taking a set of basic arithmetic functions as primitive, } \\
[\text{df}(i)] = \partial_i[f] \textit{ almost everywhere.}
PCF with real numbers

Types: \[ A, B ::= R \mid A \times B \mid A \to B \]

Programs: \[ M, N, P ::= x \mid \lambda f(x).M \mid MN \mid \langle M, N \rangle \mid \pi_i M \]
\[ \mid \text{if}(P \leq 0; M, N) \mid r : R \mid f : \mathbb{R}^k \to \mathbb{R} \]

Evaluation:
\[ (\lambda f(x).M)N \to M[N/x][\lambda f(x).M/f] \]
\[ \pi_i \langle M_1, M_2 \rangle \to M_i \]
\[ \text{if}(r \leq 0; M, N) \to \begin{cases} M & \text{if } r \leq 0 \\ N & \text{if } r > 0 \end{cases} \]
\[ f\langle r_1, \ldots, r_k \rangle \to \llbracket f \rrbracket (r_1, \ldots, r_n) \]

Every program \( x_1 : R, \ldots, x_n : R \vdash M : R \) has a natural semantics \[ \llbracket M \rrbracket : \mathbb{R}^n \to \mathbb{R}. \]

We write \( d(M) \) for the open subset of \( \mathbb{R}^n \) where \( \llbracket M \rrbracket \) is differentiable.
AD in PCF

\[ \overrightarrow{D}(\mathbb{R}) := \mathbb{R} \times \mathbb{R} \]
\[ \overrightarrow{D}(A \times B) := \overrightarrow{D}(A) \times \overrightarrow{D}(B) \]
\[ \overrightarrow{D}(A \rightarrow B) := \overrightarrow{D}(A) \rightarrow \overrightarrow{D}(B) \]

\[ \overrightarrow{D}(x : A) := x : \overrightarrow{D}(A) \]
\[ \overrightarrow{D}(\lambda f(x).M) := \lambda f(x).\overrightarrow{D}(M) \]
\[ \overrightarrow{D}(\langle M, N \rangle) := \langle \overrightarrow{D}(M), \overrightarrow{D}(N) \rangle \]
\[ \overrightarrow{D}(\pi_i M) := \pi_i \overrightarrow{D}(M) \]
\[ \overrightarrow{D}(f) := \lambda z. \left( f(\pi_1 z), \sum_{i=1}^{k} (\pi_2 z_i) \cdot \partial_i f(\pi_1 z) \right) \]

Lemma. \( M \rightarrow N \) implies \( \overrightarrow{D}(M) \rightarrow^* \overrightarrow{D}(N) \)
Soundness for simple programs

**Basic assumption:** for every primitive \( f : \mathbb{R}^k \rightarrow \mathbb{R} \) and for all \( 1 \leq i \leq k \), we have \( \llbracket \partial_i f \rrbracket = \partial_i \llbracket f \rrbracket \) on \( \mathcal{d}(f) \).

Let \( x_1 : \mathbb{R}, \ldots, x_n : \mathbb{R} \vdash M : \mathbb{R} \) and let
\[
\overrightarrow{D}_i(M) := \pi_2 \overrightarrow{D}(M)[\langle x_1, 0 \rangle / x_1] \cdots [\langle x_i, 1 \rangle / x_i] \cdots [\langle x_n, 0 \rangle / x_n]
\]
We still have \( x_1 : \mathbb{R}, \ldots, x_n : \mathbb{R} \vdash \overrightarrow{D}_i(M) : \mathbb{R} \).

**Definition.** \( \overrightarrow{D} \) is sound on \( S \subseteq \mathcal{d}(M) \) if \( \llbracket \overrightarrow{D}_i(M) \rrbracket = \partial_i \llbracket M \rrbracket \) in \( S \).

Ideally, we would like \( \overrightarrow{D} \) to be sound on \( \mathcal{d}(M) \) for every \( M \)!

**Definition.** A PCF program is *simple* if it contains no if and no recursive def.

**Theorem.** For every simple program \( t \), \( \overrightarrow{D} \) is sound on \( \mathcal{d}(t) \).
Soundness for simple programs: proof idea

Two possibilities:

- Reduce to correctness for straight-line progs (direct):

  \[
  \begin{align*}
  \text{simple progs} & \xrightarrow{D} \text{simple progs} \\
  & \downarrow \text{nf} \quad \uparrow \text{nf} \\
  \text{straight-line progs} & \xrightarrow{\{-\}} \text{straight-line progs}
  \end{align*}
  \]

- Logical relations.
**Unsoundness**

Let

\[ \text{sillyId} := \text{if}(x \leq 0; \text{if}(-x \leq 0; 0, x), x). \]

We obviously have \( [\text{sillyId}] = \text{id} \). However

\[ \overrightarrow{D}_1(\text{sillyId}) = \pi_2(\text{if}(\pi_1 \langle x, 1 \rangle \leq 0; \text{if}(\pi_1 \langle -x, -1 \rangle \leq 0; \langle 0, 0 \rangle, \langle x, 1 \rangle), \langle x, 1 \rangle)) \]

\[ \sim \text{if}(x \leq 0; \text{if}(-x \leq 0; 0, 1), 1) \]

hence

\[ [\overrightarrow{D}_1(\text{sillyId})](0) \neq \partial_1[\text{sillyId}](0) \]

NB: may happen in practice! If \( \text{ReLU}(x) := \text{if}(x \leq 0; 0, x), \) then

\[ [\text{ReLU}(x) - \text{ReLU}(-x)] = \text{id} \]

has the same behavior as above.
Approximations and traces

On types:

\[
\begin{align*}
R \sqsubseteq R & \\
A' \sqsubseteq A & \\
A' \times B' \sqsubseteq A \times B & \\
A_1' \sqsubseteq A & \ldots \ A_n' \sqsubseteq A & \quad B' \sqsubseteq B \\
A_1' \times \cdots \times A_n' \rightarrow B' \sqsubseteq A \rightarrow B
\end{align*}
\]

On terms:

\[
\begin{align*}
A_1 \sqsubseteq A, \ldots, A_n \sqsubseteq A & \\
p : A_1 \times \cdots \times A_n & \\
x : A & \\
\Xi, p \sqsubseteq x & \vdash \pi_i p \sqsubseteq x & \\
\Xi \vdash \lambda p.t \sqsubseteq \lambda x.M
\end{align*}
\]

\[
\begin{align*}
\Xi, p \sqsubseteq x & \vdash t \sqsubseteq M & \\
\Xi \vdash t \sqsubseteq M & \Xi \vdash u_1 \sqsubseteq N & \ldots & \Xi \vdash u_n \sqsubseteq N
\end{align*}
\]

\[
\Xi \vdash t \langle u_1, \ldots, u_n \rangle \sqsubseteq MN
\]

\[
\Xi \vdash t_i \sqsubseteq M_i
\]

\[
\Xi \vdash \pi_i \langle t_i, t_i \rangle \sqsubseteq \text{if}(P \leq 0; M_1, M_2) \quad \text{if } \{1, 2\}
\]

\[
\Xi \vdash t \sqsubseteq \lambda_n f(x).M
\]

where \( \lambda_0 f(x).M := \lambda f(x).M \) and \( \lambda_{n+1} f(x).M := (\lambda f.\lambda x.M)(\lambda_n f(x).M) \)

On reductions: \( (t \rightarrow^* u) \sqsubseteq (M \rightarrow N) \) if \( t \sqsubseteq M, u \sqsubseteq N \) and \( t \) "simulates" \( M \).

\( t \sqsubseteq M \) if (reduction of \( t \)) \( \sqsubseteq \) (reduction of \( M \) to normal form).
Soundness on stable points

Let $x_1 : \mathbb{R}, \ldots, x_n : \mathbb{R} \vdash M : \mathbb{R}$.

**Definition.** A point $r \in \mathbb{R}^n$ is **stable** for $M$ if there exist $t \sqsubset M$ and $\varepsilon > 0$ such that
$$\forall r' \in \mathbb{R}^n, \|r' - r\| < \varepsilon \text{ implies } t[r'/x] \sqsubseteq M[r'/x].$$

**Theorem.** For every $M$, $\overrightarrow{D}$ is sound on the stable points of $d(M)$.

The proof is based on

**Lemma.** $r$ stable for $M$ implies that there exists $t \sqsubset M$ such that $\llbracket M \rrbracket = \llbracket t \rrbracket$ on a neighborhood of $r$.

**Lemma.** If $t[r/x] \sqsubseteq M[r/x]$ and $u$ is the normal form of $\overrightarrow{D}_i(t)[r/x]$, then $\overrightarrow{D}_i(M)[r/x]$ has a normal form $N$ and $u \sqsubseteq N$. 
Quasivarieties and unsoundness

$h : \mathbb{R}^k \to \mathbb{R}$ is basic if it is in the clone generated by $\{\llbracket f \rrbracket\}_f$ primitive.

**Additional assumption:** for every basic function $h$:

1. $h$ is continuous on its domain;
2. if $h \neq 0$, then $h^{-1}(0)$ is of Lebesgue measure zero in $\mathbb{R}^k$.

For example, me may restrict to $f$’s such that $\llbracket f \rrbracket$ is analytic on its domain.

**Definition.** Quasivariety $Z \subseteq \mathbb{R}^k$ if $\exists \{h_i : \mathbb{R}^k \to \mathbb{R}\}_{i<\omega}$ basic non-zero

$$Z \subseteq \bigcup_{i<\omega} h_i^{-1}(0)$$

Quasivarieties are negligible: they are of measure zero and are stable under subsets and countable unions.

**Theorem.** The unstable points of a program form a quasivariety.

**Corollary.** For every $M$, the set $\{r \in d(M) \mid \llbracket \overrightarrow{D}_i(M) \rrbracket (r) \neq \partial_i [M](r)\}$ is a quasivariety. In particular, $\overrightarrow{D}$ is sound on almost all of $d(M)$. 
Proof: logical predicates

\[ U(M) := \text{unstable converging points of } M. \quad \Gamma := x_1 : R, \ldots, x_n : R. \]

\[ P_\Gamma(R) := \{ \Gamma \vdash M : R \mid U(M) \text{ is a quasivariety} \} \]

\[ P_\Gamma(A \rightarrow B) := \{ \Gamma \vdash M : A \rightarrow B \mid \forall N \in P_\Gamma(A), MN \in P_\Gamma(B) \} \]

\[ P_\Gamma(A_1 \times A_2) := \{ \Gamma \vdash M : A \times B \mid \forall i \in \{1, 2\}, \pi_iM \in P_\Gamma(A_i) \} \]
Proof: logical predicates with quasicontinuity

\( U(M) := \) unstable converging points of \( M \). \( \Gamma := x_1 : R, \ldots, x_n : R \).

\[ P_\Gamma(R) := \{ \Gamma \vdash M : R \mid U(M) \text{ is a quasivariety and } [M] \text{ is cqc} \} \]

\[ P_\Gamma(A \to B) := \{ \Gamma \vdash M : A \to B \mid \forall N \in P_\Gamma(A), MN \in P_\Gamma(B) \} \]

\[ P_\Gamma(A_1 \times A_2) := \{ \Gamma \vdash M : A \times B \mid \forall i \in \{1, 2\}, \pi_i M \in P_\Gamma(A_i) \} \]

**Definition.** Quasiopen set of \( \mathbb{R}^n \) (\( U \) open and \( h \) basic):

\[ Q, Q' ::= U \mid h^{-1}(0) \mid \bigcup_{i<\omega} Q_i \mid Q \cap Q' \]

**Definition.** \( f : \mathbb{R}^n \to \mathbb{R}^m \) quasicontinuous if \( Q \subseteq \mathbb{R}^m \) quasiopen implies \( f^{-1}(Q) \) quasiopen. It is completely quasicontinuous \( \text{(cqc)} \) if \( \text{id}_{\mathbb{R}^k \times f} \) is quasicontinuous for all \( k \in \mathbb{N} \).

**Lemma.** \( \Gamma, y_1 : A_1, \ldots, y_m : A_m \vdash M : A \) and \( N_i \in P_\Gamma(A_i) \) for all \( 1 \leq i \leq m \) implies \( M[N_1/y_1] \cdots [N_m/y_m] \in P_\Gamma(A) \).
Back to derivatives

If $f : A \to B$, its derivative (if it exists), is a function

$$Df : A \to (A \to B)$$

where $A \to B$ is the space of linear functions from $A$ to $B$. Given $x \in A$, one often writes $D_x f$ for the function $Df(x) : A \to B$. With this notation, the chain rule becomes

$$D_x (f \circ g) = D_{g(x)} f \circ D_x g$$

If $A = \mathbb{R}^n$, $B = \mathbb{R}^m$ and $x \in \mathbb{R}^n$, $J_x f := D_x f$ is the Jacobian matrix $(m \times n)$, or gradient $\nabla_x f$ if $m = 1$. Composition is matrix product:

$$\mathbb{R}^{n_0} \xrightarrow{f_1} \mathbb{R}^{n_2} \xrightarrow{f_2} \cdots \xrightarrow{f_{p-1}} \mathbb{R}^{n_{p-1}} \xrightarrow{f_p} \mathbb{R}^{n_p}$$

$$J_x (f_p \circ \cdots \circ f_1) = J_{f_{p-1}(\cdots f_1(x))} f_p \cdots J_x f_1$$

NB: when $A = B = \mathbb{R}$, the Jacobian matrix is just a scalar, hence the high school definition.
Computing gradients: from forward to reverse mode

Consider a straight-line programs $P$ with $p$ lines. The $i$-th line  
$z_i = g_i(y_1, \ldots, y_k)$  
induces a function $f_i : \mathbb{R}^{n_{i-1}} \rightarrow \mathbb{R}^{n_i}$, with $n_{i-1} \geq k$ and $n_i$ equal to the number of variables (including $z_i$) used by the lines $> i$.  
Hence, $[P] = f_p \circ \cdots \circ f_1$ as above, and computing $\nabla [P]$ means:

• computing each matrix $J_{f_{i-1}(\ldots f_1(x))} f_i$
• multiply them together.

$\nabla$ may be adapted to compute $\nabla [P]$, starting from the right.  
But matrix product is associative, so we may also start from the left!

• Say that $n_0 \approx n_1 \approx \cdots \approx n_{p-1} \approx n$, whereas $n_p = 1$.
• $J_{x f_1}, J_{f_1(x)} f_2, \ldots, J_{f_{p-2}(\ldots f_1(x))} f_{p-1}$ are $n \times n$.
• $J_{f_{p-1}(\ldots f_1(x))} f_{p}$ is a row vector of size $n$!

We go from $O(n^2)$ scalar products to $O(n)$!
**Reverse mode AD as transposition**

Remember: if \( f : \mathbb{R}^n \rightarrow \mathbb{R}^m \) and \( x \in \mathbb{R}^n \)

\[
J_x f : \mathbb{R}^n \rightarrow \mathbb{R}^m
\]

Linear maps may be transposed:

\[
J^t_x f : \mathbb{R}^m \rightarrow \mathbb{R}^n
\]

Technically, this uses \((-)^\perp\), but \(\mathbb{R}^\perp \cong \mathbb{R}\).

The chain rule becomes

\[
\begin{align*}
\mathbb{R}^{n_0} & \xrightarrow{f_1} \mathbb{R}^{n_2} & \xrightarrow{f_2} & \ldots & \xrightarrow{f_{p-1}} & \mathbb{R}^{n_{p-1}} & \xrightarrow{f_p} & \mathbb{R}^{n_p} \\
J^t_x (f_p \circ \cdots \circ f_1) & = J^t_x f_1 \circ \cdots \circ J^t_{f_{p-1}(f_1(x))} f_p
\end{align*}
\]
Reverse mode AD for straight-line programs

def f(x1,...,xn):
    z1 = g1(v1,...,vk)
    ...
    zp = gp(w1,...,wh)
    return zp

~~>
def grad_f(x1,...,xn):
    z1 = g1(v1,...,vk)
    ...
    zp = gp(w1,...,wh)
    # reverse pass starts here
    dx1,...,dxn,dz1,...,dz{p-1},dzp = 0,...,0,0,...,0,1
    dw1  += dgp(1,w1,...,wh) * dzp
    ...
    dw1  += dgp(h,w1,...,wh) * dzp
    ...
    dv1  += dg1(1,v1,...,vk) * dz1
    ...
    dvk  += dg1(k,v1,...,vk) * dz1
    return dx1,...,dxn
Backpropagators and derivatives

- For an arbitrary space $E$, let $\mathbb{R}^\perp := \mathbb{R} \to E$ and $\mathbb{R}^\bullet := \mathbb{R} \times \mathbb{R}^\perp$.
- An element of $\mathbb{R}^\perp$ is called backpropagator.
- If $f : \mathbb{R} \to \mathbb{R}$ is differentiable, we define $f^\bullet : \mathbb{R}^\bullet \to \mathbb{R}^\bullet$ as follows:

$$f^\bullet(x, x^*) := (f(x), \lambda a. x^*(a f'(x)))$$

- We have

$$(f \circ g)^\bullet = f^\bullet \circ g^\bullet \quad \text{id}^\bullet = \text{id}$$

- Furthermore, notice that, taking $E = \mathbb{R}$,

$$f^\bullet(x, \lambda a. a) = (f(x), \lambda a. a f'(x))$$
Backpropagators and gradients

- If $f : \mathbb{R}^n \rightarrow \mathbb{R}$ is differentiable, we define $f^\bullet : \mathbb{R}^n \rightarrow \mathbb{R}$ as follows:

$$f^\bullet(x_1, x_1^*, \ldots, x_n, x_n^*) := \left( f(\vec{x}), \lambda a. \sum_{i=1}^{n} x_i^*(a \partial_i f(\vec{x})) \right)$$

- We still have

$$(f \circ (g_1, \ldots, g_n))^\bullet = f^\bullet \circ (g_1^\bullet, \ldots, g_n^\bullet)$$

- Furthermore, notice that, taking $E = \mathbb{R}^n$,

$$f^\bullet(x_1, \iota_1, \ldots, x_n, \iota_n) = (f(\vec{x}), \lambda a. a \nabla f(\vec{x}))$$

where $\iota_i : \mathbb{R} \rightarrow \mathbb{R}^n$ is the $i$-th injection.
Reverse mode AD in PCF with linear negation

- Types: \( A, B ::= R \mid A \times B \mid A \rightarrow B \mid R \multimap A \)
- Same programs. Typing judgments \( \Gamma_{\text{non-lin}}; \Delta_{\text{lin}} \vdash M : A \) to track linearity.
- **Linear factoring rule**: if \( x^*: R \multimap A \),
  \[
  x^* M + x^* N \rightarrow x^* (M + N)
  \]
- Reverse mode AD has source PCF and target PCF with linear negation.
- For any type \( E \), we let \( \vec{D}_E(R) := R \times (R \multimap E) \). Homomorphic on the rest.
- On programs, homomorphic everywhere except

  \[
  \vec{D}_E(r) := \langle r, \lambda a.0 \rangle \quad \vec{D}_E(f) := \lambda z. \left< f(\pi_1 z), \lambda a. \sum_{i=1}^k (\pi_2 z_i)(a \partial_i f(\pi_1 z)) \right>
  \]

**Lemma.** \( M \rightarrow N \) implies \( \vec{D}(M) \rightarrow^* \vec{D}(N) \)
Soundness for reverse mode AD

We work under the same assumptions about the f’s as above.

Let $x_1 : \mathbb{R}, \ldots, x_n : \mathbb{R} \vdash M : \mathbb{R}$ and let (using $\mathbb{R}^\perp = \mathbb{R} \rightarrow \mathbb{R}^n$)

$$\text{grad}(M) := (\pi_2 \overrightarrow{D}(M)[\langle x_1, \iota_1 \rangle / x_1] \cdots [\langle x_n, \iota_n \rangle / x_n])1$$

We have $x_1 : \mathbb{R}, \ldots, x_n : \mathbb{R} \vdash \text{grad}(M) : \mathbb{R}^n$.

**Definition.** $\overrightarrow{D}$ is sound on $S \subseteq d(M)$ if $[[\text{grad}(M)]] = \nabla [[M]]$ in $S$.

The soundness proof may be adapted to reverse mode:

**Theorem.** For every $M$, $\overrightarrow{D}$ is sound on the stable points of $d(M)$.

**Corollary.** For all $M$, the set $\{r \in d(M) \mid [[\text{grad}(M)]](r) \neq \nabla [[M]](r)\}$ is a quasivariety. In particular, $\overrightarrow{D}$ is sound on almost all of $d(M)$. 
Soundness and efficiency for simple programs

\[
\begin{array}{ccc}
\text{simple progs} & \xrightarrow{\mathcal{D}} & \text{simple progs} \\
\downarrow m & & \downarrow O(m) \text{ (by 2-functoriality)} \\
\text{straight-line progs} & \xrightarrow{(-)^*} & \text{straight-line progs} \\
& \sim & \text{straight-line progs}
\end{array}
\]

Without linear factoring, execution is inefficient. Consider

\[
M := (\lambda z. z \sin z) N
\]

\[
\mathcal{D}(M) \rightarrow^* (\lambda\langle z, z^* \rangle. \langle z \sin z, \lambda a.z^*(a \sin z) + z^*(a z \cos z) \rangle) \langle r, \lambda b.B \rangle
\]

Duplicating \(\lambda b.B\) is inefficient, need to apply factoring before. This brings up the question of how to implement all this.
A personal, partial bibliography

1964  Wengert: reverse mode AD
1976  Joss (PhD Thesis): forward mode AD as a transformation on (Turing-complete) straight-line programs
2008  Pearlmutter and Siskind: backprop is higher order! Differentiable programming ante litteram
2016  Abadi et al.: TensorFlow
2017  Paszke et al.: PyTorch
2018  Elliott (ICFP): AD is functorial!
2019  Wang, Zheng, Decker, Wu, Essertel, Rompf (ICFP): backprop as typed transformation, fully general, HO
2020  Abadi and Plotkin (POPL): first-order, “internal” AD
  Barthe, Crubillé, Dal Lago, Gavazzo (ESOP): correctness by logical relations
  Brunel, Mazza, Pagani (POPL): reverse mode AD with linear negation, simply-typed $\lambda$-calculus
  Huot, Staton, Vákár (FoSSaCS): correctness proofs by logical relations with diffeologies
  Mak, Ong (Arxiv): reverse mode AD base on differential forms
2021  Kerjean and Pédrot (unpublished): AD and Dialectica (related to Pearlmutter and Siskind?)
  Mazza and Pagani (POPL): (un)soundness of AD in PCF
  Sherman, Michel, Carbin (POPL): semantics for AD
  Vákár (ESOP): homomorphic AD
2022  Krawiec, Jones, Krishnaswami, Ellis, Eisenberg, Fitzgibbon (POPL): reverse mode AD in Haskell
  Vákár, Smeding (ToPLAS): categorically-grounded AD (related to Pearlmutter and Siskind?)
2023  Alvarez-Picallo, Ghica, Sprunger, Zanasi (CSL): reverse mode AD in string diagrams
  Lew, Huot, Mansinghka, ??? (unpublished): semantic proof of our POPL 2021 results, via $\omega$PAP functions
  Radul, Paszke, Frostig, Johnson, Maclaurin (POPL): how JAX works
  Smeding, Vákár (POPL): implementation of our POPL 2020 paper
Challenge: “internal” AD

Differentiation as a programming primitive, not a transformation (like [Pearlmutter and Siskind 2008], [Abadi and Plotkin 2020], or the differential λ-calculus):

\[ M, N ::= x | \lambda x. M | MN | \ldots | \mathcal{D}_\Gamma M \]

\[
\frac{x_1 : C_1, \ldots, x_n : C_n \vdash M : A}{x_1 : \mathcal{D}_\Gamma(C_1), \ldots, x_n : \mathcal{D}_\Gamma(C_n) \vdash \mathcal{D}_\Gamma M : \mathcal{D}_\Gamma(A)}
\]

- “True” differentiable programming (with higher-order derivatives).
- Naive idea: turn the transformation defn into rewriting rules.
- But the target language must be the same as the source.
- NB: with if-then-else, internal AD breaks the std semantics:

\[
[\lambda x.x] = [\lambda x.\text{ReLU}(x) - \text{ReLU}(-x)]
\]

\[
[\mathcal{D}_\Gamma(\lambda x.x)] \neq [\mathcal{D}_\Gamma(\lambda x.\text{ReLU}(x) - \text{ReLU}(-x))]
\]
Question: the benefit of compositionality?

Remember the two routes:

\[
\begin{align*}
[R^n \rightarrow R] & \quad \text{arbitrary progs} & [D(R)^n \rightarrow D(R)] \\
\text{exec} & \quad \text{compositional backprop} & \text{exec}
\end{align*}
\]

\[
\begin{align*}
[R^n \rightarrow R] & \quad \text{straight-line progs} & [R^{n+1} \rightarrow R^{1+n}] \sim [D(R)^n \rightarrow D(R)] \\
\text{backprop} & \quad \text{} & \text{}
\end{align*}
\]

Question:

are there examples (NN architectures...) where the HO route is \textit{substantially better} (faster, more convenient...) than the FO route? Current implementations do not seem to provide an answer.
Challenge: almost-everywhere correctness?

- The set of inputs on which AD is incorrect has measure zero.
- The set of representable reals has measure zero (it’s actually finite).
- Smartass. Ok, look, in PCF\(+,\times\) it’s actually of this form:

\[
\text{Fail} \subseteq \bigcup_{i<\omega} P_i^{-1}(0)
\]

where the \(P_i\) are polynomials (not identically zero, not necessarily distinct).

- In fact, the \(P_i\) come from “cusps” of if-then-else statements.
- Is it possible to automatically infer an upper bound on \text{Fail}?
**Question: AD in the differential \( \lambda \)-calculus?**

The diff \( \lambda \)-calculus computes derivatives with respect to numbers which are *not* the ones that programs have direct access to.

- In the differential \( \lambda \)-calculus:
  - type = topological \( \mathbb{R} \)-vector space
  - program \( A \rightarrow B \) = smooth function \( A \rightarrow B \)
  - derivative = smooth function of type \( A \rightarrow (A \rightarrow B) \)
  - unit type = \( \mathbb{R} \), Booleans = \( \mathbb{R}^2 \), reals = \( \mathbb{R}(\text{uncountable basis}) \).
  - \( 0.5 \cdot 2 + 0.5 \cdot 4 = 3 \neq 0.5 \cdot 2 + 0.5 \cdot 4 \).
- Different behavior at higher types. Below, \( f : \mathbb{R} \rightarrow \mathbb{R} \):
  \[
  D(\lambda x^R. f(fx)) = \lambda x^R. \alpha(fx) + f'(fx) \cdot (\alpha x) \quad \text{with} \quad \alpha : \mathbb{R} \rightarrow \mathbb{R}
  \]
  \[
  \overrightarrow{D}(\lambda x^R. f(fx)) = \lambda X^R^2. F(FX) \quad \text{with} \quad F : \mathbb{R}^2 \rightarrow \mathbb{R}^2
  \]
- There is no differential PCF! (Recently fixed by Ehrhard’s coherent differentiation).