

Differentiable Functional Programming

Atılım Güneş Baydin

University of Oxford

<http://www.robots.ox.ac.uk/~gunes/>

F#unctional Londoners Meetup, April 28, 2016



About me

- Current (from 11 April 2016):
Postdoctoral researcher,
Machine Learning Research Group, University of Oxford
<http://www.robots.ox.ac.uk/~parg/>
- Previously:
Brain and Computation Lab, National University of Ireland Maynooth: <http://www.bcl.hamilton.ie/>
- Working primarily with **F#**, on **algorithmic differentiation, functional programming, machine learning**

Today's talk

- Derivatives in computer programs
- Differentiable functional programming
- DiffSharp + Hype libraries
- Two demos

Derivatives in computer programs
How do we compute them?

Manual differentiation

$$f(x) = \sin(\exp x)$$

```
let f x = sin (exp x)
```

Manual differentiation

$$f(x) = \sin(\exp x)$$

```
let f x = sin (exp x)
```

Calculus 101: differentiation rules

$$\begin{aligned}\frac{d(fg)}{dx} &= \frac{df}{dx}g + f\frac{dg}{dx} \\ \frac{d(af + bg)}{dx} &= a\frac{df}{dx} + b\frac{dg}{dx} \\ &\dots\end{aligned}$$

Manual differentiation

$$f(x) = \sin(\exp x)$$

```
let f x = sin (exp x)
```

Calculus 101: differentiation rules

$$\begin{aligned}\frac{d(fg)}{dx} &= \frac{df}{dx}g + f\frac{dg}{dx} \\ \frac{d(af + bg)}{dx} &= a\frac{df}{dx} + b\frac{dg}{dx} \\ &\dots\end{aligned}$$

$$f'(x) = \cos(\exp x) \times \exp x$$

```
let f' x = (cos (exp x)) * (exp x)
```

Manual differentiation

It can get complicated

$$f(x) = 64x(1-x)(1-2x)^2(1-8x+8x^2)^2$$

(4th iteration of the logistic map $l_{n+1} = 4l_n(1-l_n)$, $l_1 = x$)

```
let f x =  
    64*x * (1-x) * ((1 - 2*x) ** 2) * ((1 - 8*x + 8*x*x) ** 2)
```


Manual differentiation

It can get complicated

$$f(x) = 64x(1-x)(1-2x)^2(1-8x+8x^2)^2$$

(4th iteration of the logistic map $l_{n+1} = 4l_n(1-l_n)$, $l_1 = x$)

```
let f x =  
    64*x * (1-x) * ((1 - 2*x) ** 2) * ((1 - 8*x + 8*x*x) ** 2)
```

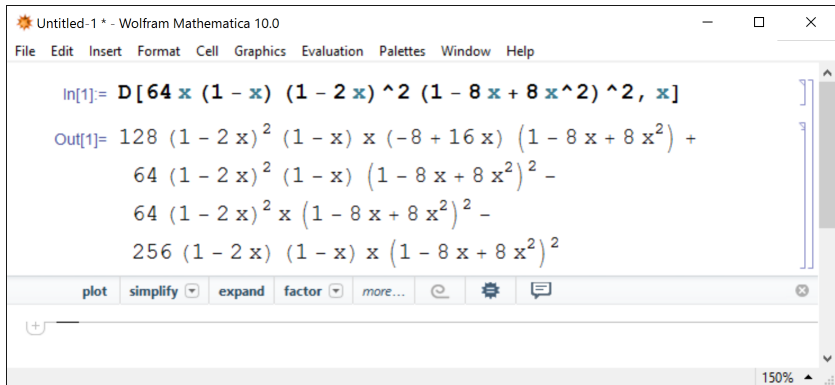
$$f'(x) =$$

$$128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2) + 64(1-x)(1-2x)^2(1-8x+8x^2)^2 - 64x(1-2x)^2(1-8x+8x^2)^2 - 256x(1-x)(1-2x)(1-8x+8x^2)^2$$

```
let f' x = 128*x * (1-x) * (-8+16*x) * (1-2*x)**2 * (1-8*x+8*x*x  
    x) + 64 * (1-x) * (1-2*x)**2 * (1-8*x+8*x*x)**2 - 64*x*(1-2*  
    x)**2 * (1-8*x+8*x*x)**2 - 256*x*(1-x) * (1-2*x) * (1-8*x  
    +8*x*x)**2
```

Symbolic differentiation

Computer algebra packages help: Mathematica, Maple, Maxima



The screenshot shows the Wolfram Mathematica 10.0 interface. The title bar reads "Untitled-1 * - Wolfram Mathematica 10.0". The menu bar includes "File", "Edit", "Insert", "Format", "Cell", "Graphics", "Evaluation", "Palettes", "Window", and "Help". The input area contains the command:

```
In[1]:= D[64 x (1 - x) (1 - 2 x)^2 (1 - 8 x + 8 x^2)^2, x]
```

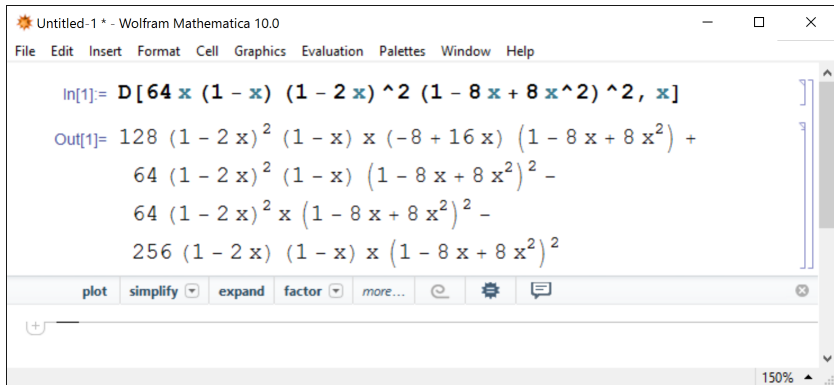
The output area displays the result of the differentiation:

```
Out[1]= 128 (1 - 2 x)^2 (1 - x) x (-8 + 16 x) (1 - 8 x + 8 x^2) +  
        64 (1 - 2 x)^2 (1 - x) (1 - 8 x + 8 x^2)^2 -  
        64 (1 - 2 x)^2 x (1 - 8 x + 8 x^2)^2 -  
        256 (1 - 2 x) (1 - x) x (1 - 8 x + 8 x^2)^2
```

Below the output, there is a toolbar with buttons for "plot", "simplify", "expand", "factor", "more...", and icons for undo, redo, and help. The status bar at the bottom right shows "150%" zoom level.

Symbolic differentiation

Computer algebra packages help: Mathematica, Maple, Maxima



The screenshot shows the Wolfram Mathematica 10.0 interface. The title bar reads "Untitled-1 * - Wolfram Mathematica 10.0". The menu bar includes "File", "Edit", "Insert", "Format", "Cell", "Graphics", "Evaluation", "Palettes", "Window", and "Help". The input field contains the command: `In[1]:= D[64 x (1 - x) (1 - 2 x)^2 (1 - 8 x + 8 x^2)^2, x]`. The output field displays the result: `Out[1]= 128 (1 - 2 x)^2 (1 - x) x (-8 + 16 x) (1 - 8 x + 8 x^2) + 64 (1 - 2 x)^2 (1 - x) (1 - 8 x + 8 x^2)^2 - 64 (1 - 2 x)^2 x (1 - 8 x + 8 x^2)^2 - 256 (1 - 2 x) (1 - x) x (1 - 8 x + 8 x^2)^2`. Below the output is a toolbar with buttons for "plot", "simplify", "expand", "factor", "more...", and icons for undo, redo, and help. The status bar at the bottom right shows "150%".

```
In[1]:= D[64 x (1 - x) (1 - 2 x)^2 (1 - 8 x + 8 x^2)^2, x]
```

```
Out[1]= 128 (1 - 2 x)^2 (1 - x) x (-8 + 16 x) (1 - 8 x + 8 x^2) +  
        64 (1 - 2 x)^2 (1 - x) (1 - 8 x + 8 x^2)^2 -  
        64 (1 - 2 x)^2 x (1 - 8 x + 8 x^2)^2 -  
        256 (1 - 2 x) (1 - x) x (1 - 8 x + 8 x^2)^2
```

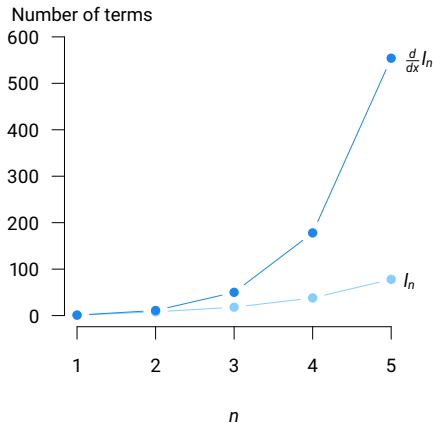
But, it has some **serious drawbacks**

Symbolic differentiation

We get “**expression swell**”

Logistic map $l_{n+1} = 4l_n(1 - l_n)$, $l_1 = x$

n	l_n	$\frac{d}{dx} l_n$
1	x	1
2	$4x(1 - x)$	$4(1 - x) - 4x$
3	$16x(1 - x)(1 - 2x)^2$	$16(1 - x)(1 - 2x)^2 - 16x(1 - 2x)^2 - 64x(1 - x)(1 - 2x)$
4	$64x(1 - x)(1 - 2x)^2(1 - 8x + 8x^2)^2$	$128x(1 - x)(-8 + 16x)(1 - 2x)^2(1 - 8x + 8x^2) + 64(1 - x)(1 - 2x)^2(1 - 8x + 8x^2)^2 - 64x(1 - 2x)^2(1 - 8x + 8x^2)^2 - 256x(1 - x)(1 - 2x)(1 - 8x + 8x^2)^2$



Symbolic differentiation

We are **limited to closed-form** formulae

Symbolic differentiation

We are **limited to closed-form** formulae

You can find the derivative of math expressions:

$$f(x) = 64x(1 - x)(1 - 2x)^2(1 - 8x + 8x^2)^2$$

Symbolic differentiation

We are **limited to closed-form** formulae

You can find the derivative of math expressions:

$$f(x) = 64x(1-x)(1-2x)^2(1-8x+8x^2)^2$$

But not of algorithms, branching, control flow:

```
let f x n =  
  if n = 1 then  
    x  
  else  
    let mutable v = x  
    for i = 1 to n  
      v <- 4 * v * (1 - v)  
    v  
let a = f x 4
```

Numerical differentiation

A very common hack:

Use the limit definition of the derivative

$$\frac{df}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

Numerical differentiation

A very common hack:

Use the limit definition of the derivative

$$\frac{df}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

to approximate the numerical value of the derivative

```
let diff f x =  
  let h = 0.00001  
  (f (x + h) - f (x)) / h
```

Numerical differentiation

A very common hack:

Use the limit definition of the derivative

$$\frac{df}{dx} = \lim_{h \rightarrow 0} \frac{f(x+h) - f(x)}{h}$$

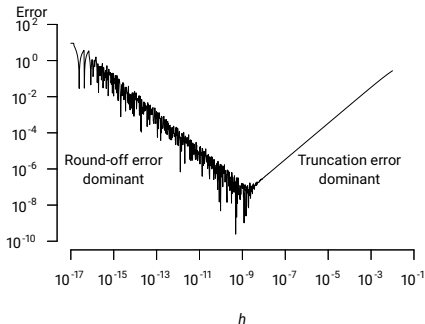
to approximate the numerical value of the derivative

```
let diff f x =  
  let h = 0.00001  
  (f (x + h) - f (x)) / h
```

Again, some **serious drawbacks**

Numerical differentiation

We must select a proper value of h
and we face **approximation errors**



Computed using

$$E(h, x^*) = \left| \frac{f(x^* + h) - f(x^*)}{h} - \frac{d}{dx}f(x)|_{x^*} \right|$$
$$f(x) = 64x(1-x)(1-2x)^2(1-8x+8x^2)^2$$
$$x^* = 0.2$$

Numerical differentiation

Better approximations exist

- Higher-order finite differences

E.g.

$$\frac{\partial f(\mathbf{x})}{\partial x_i} = \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x} - h\mathbf{e}_i)}{2h} + O(h^2) ,$$

- Richardson extrapolation
- Differential quadrature

but they increase rapidly in complexity and never completely eliminate the error

Numerical differentiation

Poor performance:

$f : \mathbb{R}^n \rightarrow \mathbb{R}$, approximate the gradient $\nabla f = \left(\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n} \right)$ using

$$\frac{\partial f(\mathbf{x})}{\partial x_i} \approx \frac{f(\mathbf{x} + h\mathbf{e}_i) - f(\mathbf{x})}{h}, \quad 0 < h \ll 1$$

We must repeat the function evaluation n times for getting ∇f

Algorithmic differentiation (AD)

Algorithmic differentiation

- Also known as *automatic differentiation* (Griewank & Walther, 2008)
- Gives numeric code that **computes the function AND its derivatives** at a given point

<pre>f(a, b): c = a * b d = sin c return d</pre>	→	<pre>f'(a, a', b, b'): (c, c') = (a*b, a'*b + a*b') (d, d') = (sin c, c' * cos c) return (d, d')</pre>
--	---	--

- Derivatives propagated at the elementary operation level, as a side effect, at the same time when the function itself is computed
→ Prevents the “expression swell” of symbolic derivatives
- Full expressive capability of the host language
→ **Including conditionals, looping, branching**

Function evaluation traces

All **numeric evaluations** are sequences of elementary operations:
a **“trace,”** also called a **“Wengert list”** (Wengert, 1964)

```
f(a, b):  
  c = a * b  
  if c > 0  
    d = log c  
  else  
    d = sin c  
  return d
```


Function evaluation traces

All **numeric evaluations** are sequences of elementary operations:
a **“trace,”** also called a **“Wengert list”** (Wengert, 1964)

```
f(a, b):  
  c = a * b  
  if c > 0  
    d = log c  
  else  
    d = sin c  
  return d
```

f(2, 3)

Function evaluation traces

All **numeric evaluations** are sequences of elementary operations:
a “**trace**,” also called a “**Wengert list**” (Wengert, 1964)

f(a, b):	a = 2
c = a * b	
if c > 0	b = 3
d = log c	
else	c = a * b = 6
d = sin c	
return d	d = log c = 1.791
f(2, 3)	return d
	(primal)

Function evaluation traces

All **numeric evaluations** are sequences of elementary operations:
a “**trace**,” also called a “**Wengert list**” (Wengert, 1964)

$f(a, b):$	$a = 2$	$a = 2$
$c = a * b$		$a' = 1$
if $c > 0$	$b = 3$	$b = 3$
$d = \log c$		$b' = 0$
else	$c = a * b = 6$	$c = a * b = 6$
$d = \sin c$		$c' = a' * b + a * b' = 3$
return d	$d = \log c = 1.791$	$d = \log c = 1.791$
$f(2, 3)$	return d	$d' = c' * (1 / c) = 0.5$
		return d, d'
	(primal)	(tangent)

Function evaluation traces

All **numeric evaluations** are sequences of elementary operations:
a “**trace**,” also called a “**Wengert list**” (Wengert, 1964)

f(a, b):	a = 2	a = 2
c = a * b		a' = 1
if c > 0	b = 3	b = 3
d = log c		b' = 0
else	c = a * b = 6	c = a * b = 6
d = sin c		c' = a' * b + a * b' = 3
return d	d = log c = 1.791	d = log c = 1.791
		d' = c' * (1 / c) = 0.5
f(2, 3)	return d	return d, d'
	(primal)	(tangent)

i.e., a Jacobian-vector product $\mathbf{J}_f(1, 0)|_{(2,3)} = \frac{\partial}{\partial a} f(a, b)|_{(2,3)} = 0.5$

This is called the **forward (tangent) mode** of AD

Function evaluation traces

f(a, b):

 c = a * b

 if c > 0

 d = log c

 else

 d = sin c

 return d

f(2, 3)

Function evaluation traces

f(a, b):	a = 2
c = a * b	b = 3
if c > 0	c = a * b = 6
d = log c	d = log c = 1.791
else	return d
d = sin c	
return d	(primal)

f(2, 3)

Function evaluation traces

```
f(a, b):  
    c = a * b  
    if c > 0  
        d = log c  
    else  
        d = sin c  
    return d
```

(primal)

f(2, 3)

```
a = 2  
b = 3  
c = a * b = 6  
d = log c = 1.791  
d' = 1  
c' = d' * (1 / c) = 0.166  
b' = c' * a = 0.333  
a' = c' * b = 0.5  
return d, a', b'
```

(adjoint)

Function evaluation traces

f(a, b):	a = 2	a = 2
c = a * b	b = 3	b = 3
if c > 0	c = a * b = 6	c = a * b = 6
d = log c	d = log c = 1.791	d = log c = 1.791
else	return d	d' = 1
d = sin c		c' = d' * (1 / c) = 0.166
return d	(primal)	b' = c' * a = 0.333
		a' = c' * b = 0.5
f(2, 3)		return d, a', b'
		(adjoint)

i.e., a transposed Jacobian-vector product

$$\mathbf{J}_f^T(1)|_{(2,3)} = \nabla f|_{(2,3)} = (0.5, 0.333)$$

This is called the **reverse (adjoint) mode** of AD

Backpropagation is just a special case of the reverse mode:
code your neural network objective computation, apply reverse AD

How is this useful?

Forward vs reverse

In the extreme cases,

for $F : \mathbb{R} \rightarrow \mathbb{R}^m$, forward AD can compute all $\left(\frac{\partial F_1}{\partial x}, \dots, \frac{\partial F_m}{\partial x} \right)$

for $f : \mathbb{R}^n \rightarrow \mathbb{R}$, reverse AD can compute $\nabla f = \left(\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n} \right)$

in **just one evaluation**

Forward vs reverse

In the extreme cases,

for $F : \mathbb{R} \rightarrow \mathbb{R}^m$, forward AD can compute all $\left(\frac{\partial F_1}{\partial x}, \dots, \frac{\partial F_m}{\partial x} \right)$

for $f : \mathbb{R}^n \rightarrow \mathbb{R}$, reverse AD can compute $\nabla f = \left(\frac{\partial f}{\partial x_1}, \dots, \frac{\partial f}{\partial x_n} \right)$

in **just one evaluation**

In general, for $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$, the Jacobian $\mathbf{J} \in \mathbb{R}^{m \times n}$ takes

- $O(n \times \text{time}(f))$ with forward AD

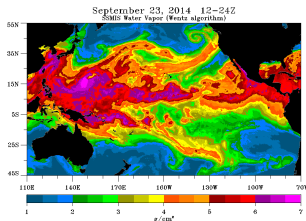
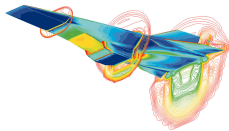
- $O(m \times \text{time}(f))$ with reverse AD

Reverse mode performs better when $n \gg m$

How is this useful?

Traditional application domains of AD in industry and academia (Corliss et al., 2002)

- Computational fluid dynamics
- Atmospheric chemistry
- Engineering design optimization
- Computational finance



Functional AD
or
"Differentiable functional programming"

AD and functional programming

AD has been around since the 1960s

(Wengert, 1964; Speelpenning, 1980; Griewank, 1989)

The foundations for AD in a functional framework

(Siskind and Pearlmutter, 2008; Pearlmutter and Siskind, 2008)

With research implementations

- R6RS-AD

<https://github.com/qobi/R6RS-AD>

- Stalingrad

<http://www.bcl.hamilton.ie/~qobi/stalingrad/>

- Alexey Radul's DVL

<https://github.com/axch/dysvfunctional-language>

- Recently, my DiffSharp library

<http://diffsharp.github.io/DiffSharp/>

Differentiable functional programming

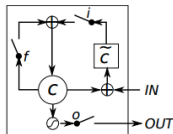
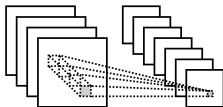
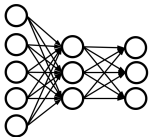
Deep learning: neural network models are assembled from **building blocks** and trained with **backpropagation**

Differentiable functional programming

Deep learning: neural network models are assembled from **building blocks** and trained with **backpropagation**

Traditional:

- Feedforward
- Convolutional
- Recurrent

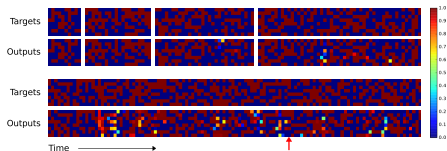


Differentiable functional programming

Newer additions:

Make **algorithmic** elements **continuous and differentiable**

→ enables use in deep learning



NTM on copy task
(Graves et al. 2014)

- Neural Turing Machine (Graves et al., 2014)
 - can infer algorithms: copy, sort, recall
- Stack-augmented RNN (Joulin & Mikolov, 2015)
- End-to-end memory network (Sukhbaatar et al., 2015)
- Stack, queue, deque (Grefenstette et al., 2015)
- Discrete interfaces (Zaremba & Sutskever, 2015)

Differentiable functional programming

Stacking of many layers, trained through backpropagation

AlexNet, 8 layers (ILSVRC 2012)



VGG, 19 layers (ILSVRC 2014)



ResNet, 152 layers (deep residual learning) (ILSVRC 2015)



(He, Zhang, Ren, Sun. "Deep Residual Learning for Image Recognition." 2015. arXiv:1512.03385)

Differentiable functional programming

One way of viewing deep learning systems is
“**differentiable functional programming**”

Two main characteristics:

- **Differentiability**

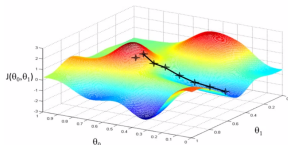
- optimization

- Chained function **composition**

- successive
transformations

- successive levels of
distributed representations
(Bengio 2013)

- the chain rule of calculus
propagates derivatives



$$g : A \rightarrow B$$

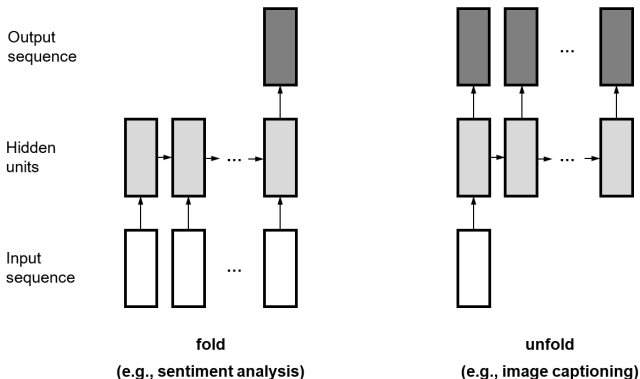
$$f : B \rightarrow C$$

$$f \circ g : A \rightarrow C$$

The bigger picture

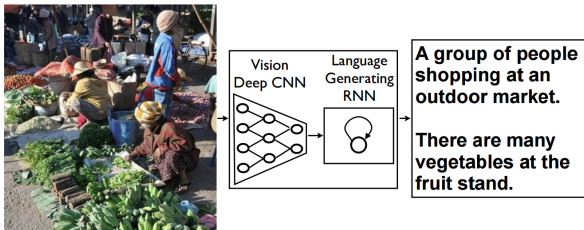
In a functional interpretation

- **Weight-tying** or multiple applications of the same neuron (e.g., ConvNets and RNNs) resemble **function abstraction**
- **Structural patterns** of composition resemble **higher-order functions** (e.g., map, fold, unfold, zip)



The bigger picture

Even when you have **complex compositions**, differentiability ensures that they can be trained end-to-end with backpropagation



(Vinyals, Toshev, Bengio, Erhan. "Show and tell: a neural image caption generator." 2014. arXiv:1411.4555)

The bigger picture

Christopher Olah's blog post (September 3, 2015)

<http://colah.github.io/posts/2015-09-NN-Types-FP/>

"The field does not (yet) have a unifying insight or narrative"

David Dalrymple's essay (January 2016)

<http://edge.org/response-detail/26794>

*"The most natural playground ... would be a new language that can **run back-propagation directly on functional programs.**"*

The bigger picture

Christopher Olah's blog post (September 3, 2015)

<http://colah.github.io/posts/2015-09-NN-Types-FP/>

"The field does not (yet) have a unifying insight or narrative"

David Dalrymple's essay (January 2016)

<http://edge.org/response-detail/26794>

*"The most natural playground ... would be a new language that can **run back-propagation directly on functional programs.**"*

AD in a functional framework is a manifestation of this vision.

DiffSharp

The ambition

- Deeply embedded AD (forward and/or reverse) as part of the language infrastructure
- Rich API of differentiation operations as higher-order functions
- High-performance matrix operations for deep learning (GPU support, model and data parallelism), gradients, Hessians, Jacobians, directional derivatives, matrix-free Hessian- and Jacobian-vector products

The ambition

- Deeply embedded AD (forward and/or reverse) as part of the language infrastructure
- Rich API of differentiation operations as higher-order functions
- High-performance matrix operations for deep learning (GPU support, model and data parallelism), gradients, Hessians, Jacobians, directional derivatives, matrix-free Hessian- and Jacobian-vector products

I have been working on these issues with Barak Pearlmutter and created **DiffSharp**:

<http://diffsharp.github.io/DiffSharp/>

DiffSharp

“Generalized AD as a first-class function in an augmented λ -calculus” (Pearlmutter and Siskind, 2008)

Forward, reverse, and **any nested combination** thereof,
instantiated according to usage scenario

Nested lambda expressions with free-variable references

$$\min(\lambda x . (f\ x) + \min(\lambda y . g\ x\ y))$$

```
let m = min (fun x -> (f x) + min (fun y -> g (x y)))
```

DiffSharp

“Generalized AD as a first-class function in an augmented λ -calculus” (Pearlmutter and Siskind, 2008)

Forward, reverse, and **any nested combination** thereof,
instantiated according to usage scenario

Nested lambda expressions with free-variable references

$$\min (\lambda x . (f x) + \min (\lambda y . g x y))$$

```
let m = min (fun x -> (f x) + min (fun y -> g (x y)))
```

Must handle “perturbation confusion” (Manzyuk et al., 2012)

$$\left. \frac{d}{dx} \left(x \left(\frac{d}{dy} x + y \right) \right) \right|_{y=1} \bigg|_{x=1} \stackrel{?}{=} 1$$

```
let d = diff (fun x -> x * (diff (fun y -> x + y) 1.)) 1.
```

DiffSharp

Higher-order differentiation API

	Op.	Value	Type signature	AD	Num.	Sym.
$f : \mathbb{R} \rightarrow \mathbb{R}$	diff	f'	$(\mathbb{R} \rightarrow \mathbb{R}) \rightarrow \mathbb{R} \rightarrow \mathbb{R}$	X, F	A	X
	diff'	(f, f')	$(\mathbb{R} \rightarrow \mathbb{R}) \rightarrow \mathbb{R} \rightarrow (\mathbb{R} \times \mathbb{R})$	X, F	A	X
	diff2	f''	$(\mathbb{R} \rightarrow \mathbb{R}) \rightarrow \mathbb{R} \rightarrow \mathbb{R}$	X, F	A	X
	diff2'	(f, f'')	$(\mathbb{R} \rightarrow \mathbb{R}) \rightarrow \mathbb{R} \rightarrow (\mathbb{R} \times \mathbb{R})$	X, F	A	X
	diff2'',	(f, f', f'')	$(\mathbb{R} \rightarrow \mathbb{R}) \rightarrow \mathbb{R} \rightarrow (\mathbb{R} \times \mathbb{R} \times \mathbb{R})$	X, F	A	X
	diffn	$f^{(n)}$	$\mathbb{N} \rightarrow (\mathbb{R} \rightarrow \mathbb{R}) \rightarrow \mathbb{R} \rightarrow \mathbb{R}$	X, F		X
	diffn'	$(f, f^{(n)})$	$\mathbb{N} \rightarrow (\mathbb{R} \rightarrow \mathbb{R}) \rightarrow \mathbb{R} \rightarrow (\mathbb{R} \times \mathbb{R})$	X, F		X
$f : \mathbb{R}^n \rightarrow \mathbb{R}$	grad	∇f	$(\mathbb{R}^n \rightarrow \mathbb{R}) \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}^n$	X, R	A	X
	grad'	$(f, \nabla f)$	$(\mathbb{R}^n \rightarrow \mathbb{R}) \rightarrow \mathbb{R}^n \rightarrow (\mathbb{R} \times \mathbb{R}^n)$	X, R	A	X
	gradv	$\nabla f \cdot \mathbf{v}$	$(\mathbb{R}^n \rightarrow \mathbb{R}) \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}$	X, F	A	
	gradv'	$(f, \nabla f \cdot \mathbf{v})$	$(\mathbb{R}^n \rightarrow \mathbb{R}) \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}^n \rightarrow (\mathbb{R} \times \mathbb{R})$	X, F	A	
	hessian	\mathbf{H}_f	$(\mathbb{R}^n \rightarrow \mathbb{R}) \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}^{n \times n}$	X, R-F	A	X
	hessian'	(f, \mathbf{H}_f)	$(\mathbb{R}^n \rightarrow \mathbb{R}) \rightarrow \mathbb{R}^n \rightarrow (\mathbb{R} \times \mathbb{R}^{n \times n})$	X, R-F	A	X
	hessianv	$\mathbf{H}_f \mathbf{v}$	$(\mathbb{R}^n \rightarrow \mathbb{R}) \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}^n$	X, F-R	A	
	hessianv'	$(f, \mathbf{H}_f \mathbf{v})$	$(\mathbb{R}^n \rightarrow \mathbb{R}) \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}^n \rightarrow (\mathbb{R} \times \mathbb{R}^n)$	X, F-R	A	
	gradhessian	$(\nabla f, \mathbf{H}_f)$	$(\mathbb{R}^n \rightarrow \mathbb{R}) \rightarrow \mathbb{R}^n \rightarrow (\mathbb{R}^n \times \mathbb{R}^{n \times n})$	X, R-F	A	X
	gradhessian'	$(f, \nabla f, \mathbf{H}_f)$	$(\mathbb{R}^n \rightarrow \mathbb{R}) \rightarrow \mathbb{R}^n \rightarrow (\mathbb{R} \times \mathbb{R}^n \times \mathbb{R}^{n \times n})$	X, R-F	A	X
	gradhessianv	$(\nabla f \cdot \mathbf{v}, \mathbf{H}_f \mathbf{v})$	$(\mathbb{R}^n \rightarrow \mathbb{R}) \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}^n \rightarrow (\mathbb{R} \times \mathbb{R}^n)$	X, F-R	A	
	gradhessianv'	$(f, \nabla f \cdot \mathbf{v}, \mathbf{H}_f \mathbf{v})$	$(\mathbb{R}^n \rightarrow \mathbb{R}) \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}^n \rightarrow (\mathbb{R} \times \mathbb{R} \times \mathbb{R}^n)$	X, F-R	A	
	laplacian	$\text{tr}(\mathbf{H}_f)$	$(\mathbb{R}^n \rightarrow \mathbb{R}) \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}$	X, R-F	A	X
	laplacian'	$(f, \text{tr}(\mathbf{H}_f))$	$(\mathbb{R}^n \rightarrow \mathbb{R}) \rightarrow \mathbb{R}^n \rightarrow (\mathbb{R} \times \mathbb{R})$	X, R-F	A	X
$\mathbf{f} : \mathbb{R}^n \rightarrow \mathbb{R}^m$	jacobian	\mathbf{J}_f	$(\mathbb{R}^n \rightarrow \mathbb{R}^m) \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}^{m \times n}$	X, F/R	A	X
	jacobian'	(f, \mathbf{J}_f)	$(\mathbb{R}^n \rightarrow \mathbb{R}^m) \rightarrow \mathbb{R}^n \rightarrow (\mathbb{R}^m \times \mathbb{R}^{m \times n})$	X, F/R	A	X
	jacobianv	$\mathbf{J}_f \mathbf{v}$	$(\mathbb{R}^n \rightarrow \mathbb{R}^m) \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}^m$	X, F	A	
	jacobianv'	$(f, \mathbf{J}_f \mathbf{v})$	$(\mathbb{R}^n \rightarrow \mathbb{R}^m) \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}^n \rightarrow (\mathbb{R}^m \times \mathbb{R}^m)$	X, F	A	
	jacobianT	\mathbf{J}_f^T	$(\mathbb{R}^n \rightarrow \mathbb{R}^m) \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}^{n \times m}$	X, F/R	A	X
	jacobianT'	(f, \mathbf{J}_f^T)	$(\mathbb{R}^n \rightarrow \mathbb{R}^m) \rightarrow \mathbb{R}^n \rightarrow (\mathbb{R}^m \times \mathbb{R}^{n \times m})$	X, F/R	A	X
	jacobianTv	$\mathbf{J}_f^T \mathbf{v}$	$(\mathbb{R}^n \rightarrow \mathbb{R}^m) \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}^m \rightarrow \mathbb{R}^n$	X, R		
	jacobianTv'	$(f, \mathbf{J}_f^T \mathbf{v})$	$(\mathbb{R}^n \rightarrow \mathbb{R}^m) \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}^m \rightarrow (\mathbb{R}^m \times \mathbb{R}^n)$	X, R		
	jacobianTv'',	$(f, \mathbf{J}_f^T(\cdot))$	$(\mathbb{R}^n \rightarrow \mathbb{R}^m) \rightarrow \mathbb{R}^n \rightarrow (\mathbb{R}^m \times (\mathbb{R}^m \rightarrow \mathbb{R}^n))$	X, R		
	curl	$\nabla \times \mathbf{f}$	$(\mathbb{R}^3 \rightarrow \mathbb{R}^3) \rightarrow \mathbb{R}^3 \rightarrow \mathbb{R}^3$	X, F	A	X
	curl'	$(\mathbf{f}, \nabla \times \mathbf{f})$	$(\mathbb{R}^3 \rightarrow \mathbb{R}^3) \rightarrow \mathbb{R}^3 \rightarrow (\mathbb{R}^3 \times \mathbb{R}^3)$	X, F	A	X
	div	$\nabla \cdot \mathbf{f}$	$(\mathbb{R}^n \rightarrow \mathbb{R}^n) \rightarrow \mathbb{R}^n \rightarrow \mathbb{R}$	X, F	A	X
	div'	$(\mathbf{f}, \nabla \cdot \mathbf{f})$	$(\mathbb{R}^n \rightarrow \mathbb{R}^n) \rightarrow \mathbb{R}^n \rightarrow (\mathbb{R}^n \times \mathbb{R})$	X, F	A	X
	curldiv	$(\nabla \times \mathbf{f}, \nabla \cdot \mathbf{f})$	$(\mathbb{R}^3 \rightarrow \mathbb{R}^3) \rightarrow \mathbb{R}^3 \rightarrow (\mathbb{R}^3 \times \mathbb{R})$	X, F	A	X
	curldiv'	$(\mathbf{f}, \nabla \times \mathbf{f}, \nabla \cdot \mathbf{f})$	$(\mathbb{R}^3 \rightarrow \mathbb{R}^3) \rightarrow \mathbb{R}^3 \rightarrow (\mathbb{R}^3 \times \mathbb{R}^3 \times \mathbb{R})$	X, F	A	X

DiffSharp

Matrix operations

<http://diffsharp.github.io/DiffSharp/api-overview.html>

High-performance OpenBLAS backend by default, work on a CUDA-based GPU backend underway

Support for 64- and 32-bit floats (faster on many systems)

Benchmarking tool

<http://diffsharp.github.io/DiffSharp/benchmarks.html>

A growing collection of tutorials: gradient-based optimization algorithms, clustering, Hamiltonian Monte Carlo, neural networks, inverse kinematics

Hype

Hype

<http://hypelib.github.io/Hype/>

An experimental library for “compositional machine learning and **hyper**parameter optimization”, built on DiffSharp

A robust optimization core

- highly configurable functional modules
- SGD, conjugate gradient, Nesterov, AdaGrad, RMSProp, Newton's method
- Use nested AD for gradient-based hyperparameter optimization (Maclaurin et al., 2015)

Researching the differentiable functional programming paradigm for machine learning

Hype

Extracts from Hype neural network code,
use higher-order functions, don't think about gradients or
backpropagation

<https://github.com/hypelib/Hype/blob/master/src/Hype/Neural.fs>

```
1: // Use mixed mode nested AD
2: open DiffSharp.AD.Float32
3:
4: type FeedForward() =
5:     inherit Layer()
6:     // Feedforward layers executed as "fold", DM -> DM
7:     override n.Run(x:DM) = Array.fold Layer.run x layers
8:
9: type GRU(inputs:int, memcells:int) =
10:     inherit Layer()
11:     // RNN many-to-many execution as "map", DM -> DM
12:     override l.Run (x:DM) =
13:         x |> DM.mapCols
14:             (fun x ->
15:                 let z = sigmoid(l.Wxz * x + l.Whz * l.h + l.bz)
16:                 let r = sigmoid(l.Wxr * x + l.Whr * l.h + l.br)
17:                 let h' = tanh(l.Wxh * x + l.Whh * (l.h .* r))
18:                 l.h <- (1.f - z) .* h' + z .* l.h
19:                 l.h)
```

Hype

Extracts from Hype optimization code

<https://github.com/hypelib/Hype/blob/master/src/Hype/Optimize.fs>

Optimization and training as higher-order functions

→ can be composed, nested

```
1: // Minimize function `f`
2: static member Minimize (f:DV->D, w0:DV) =
3:     Optimize.Minimize (f, w0, Params.Default)
4:
5: // Train model function `f`
6: static member Train (f:DV->DV->D, w0:DV, d:Dataset) =
7:     Optimize.Train ((fun w v -> toDV [f w v]), w0, d)
```

Hype

User doesn't need to think about derivatives

They are instantiated within the optimization code

```
1: type Method
2:   | CG -> // Conjugate gradient
3:     fun w f g p gradclip ->
4:       let v', g' = grad' f w // gradient
5:       let g' = gradclip g'
6:       let y = g' - g
7:       let b = (g' * y) / (p * y)
8:       let p' = -g' + b * p
9:       v', g', p'
10:  | NewtonCG -> // Newton conjugate gradient
11:    fun w f _ p gradclip ->
12:      let v', g' = grad' f w // gradient
13:      let g' = gradclip g'
14:      let hv = hessianv f w p // Hessian-vector product
15:      let b = (g' * hv) / (p * hv)
16:      let p' = -g' + b * p
17:      v', g', p'
18:  | Newton -> // Newton's method
19:    fun w f _ _ gradclip ->
20:      let v', g', h' = gradhessian' f w // gradient, Hessian
21:      let g' = gradclip g'
22:      let p' = -DM.solveSymmetric h' g'
23:      v', g', p'
```

Hype

But they can use derivatives within their models, if needed

- input sensitivities
- complex objective functions
- adaptive PID controllers
- integrating differential equations

```
1: // Leapfrog integrator, Hamiltonian
2: let leapFrog (u:DV->D) (k:DV->D) (d:D) steps (x0, p0) =
3:   let hd = d / 2.
4:   [1..steps]
5:   |> List.fold (fun (x, p) _ ->
6:     let p' = p - hd * grad u x
7:     let x' = x + d * grad k p'
8:     x', p' - hd * grad u x')
```

Hype

But they can use derivatives within their models, if needed

- input sensitivities
- complex objective functions
- adaptive PID controllers
- integrating differential equations

```
1: // Leapfrog integrator, Hamiltonian
2: let leapFrog (u:DV->D) (k:DV->D) (d:D) steps (x0, p0) =
3:   let hd = d / 2.
4:   [1..steps]
5:   |> List.fold (fun (x, p) _ ->
6:     let p' = p - hd * grad u x
7:     let x' = x + d * grad k p'
8:     x', p' - hd * grad u x') (x0, p0)
```

Thanks to nested generalized AD

- you can optimize components that are internally using differentiation
- resulting higher-order derivatives propagate via forward/reverse AD as needed

Hype

We also provide a Torch-like API for neural networks

```
1: let n = FeedForward()  
2: n.Add(Linear(dim, 100))  
3: n.Add(LSTM(100, 400))  
4: n.Add(LSTM(400, 100))  
5: n.Add(Linear(100, dim))  
6: n.Add(reLU)
```

Hype

We also provide a Torch-like API for neural networks

```
1: let n = FeedForward()
2: n.Add(Linear(dim, 100))
3: n.Add(LSTM(100, 400))
4: n.Add(LSTM(400, 100))
5: n.Add(Linear(100, dim))
6: n.Add(reLU)
```

A cool thing: thanks to AD, we can freely code
any F# function as a layer, it just works

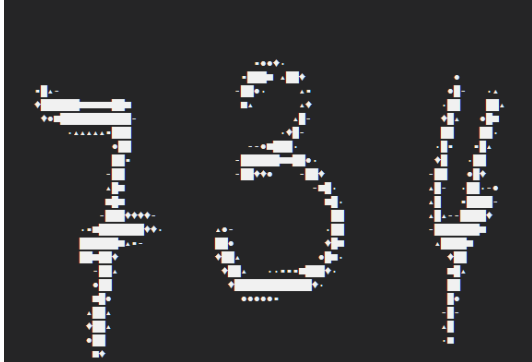
```
1: n.Add(fun m -> m |> DM.mapCols softmax) // A "map" of softmax
2:
3: let dropout (x:DM) = // Implement a new layer (dropout)
4:     x .* (Rnd.UniformDM(x.Cols, x.Rows) |> DM.Round) * 2.f
5:
6: n.Add(dropout) // Add any function as a layer
```

Hype

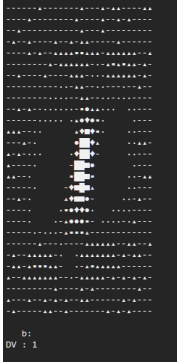
<http://hypelib.github.io/Hype/feedforwardnets.html>

We also have some nice additions for F# interactive

```
Hype.Dataset
X: 784 x 3
Y: 10 x 3
X's columns reshaped to (28 x 28), presented in a (1 x 3) grid:
DM : 28 x 84
```



```
Hype.Neural.Linear
784 -> 1
Learnable parameters: 785
Init: Standard
W's rows reshaped to (28 x 28)
DM : 28 x 28
```



Roadmap

- Transformation-based, context-aware AD
F# quotations (Syme, 2006) give us a direct path for deeply embedding AD
- Currently experimenting with GPU backends (CUDA, ArrayFire, Magma)
- Generalizing to tensors
(for elegant implementations of, e.g., ConvNets)

Demos

Thank You!

References

- Baydin AG, Pearlmutter BA, Radul AA, Siskind JM (Submitted) Automatic differentiation in machine learning: a survey [arXiv:1502.05767]
- Baydin AG, Pearlmutter BA, Siskind JM (Submitted) DiffSharp: automatic differentiation library [arXiv:1511.07727]
- Bengio Y (2013) Deep learning of representations: looking forward. Statistical Language and Speech Processing. LNCS 7978:1–37 [arXiv:1404.7456]
- Graves A, Wayne G, Danihelka I (2014) Neural Turing machines. [arXiv:1410.5401]
- Grefenstette E, Hermann KM, Suleyman M, Blunsom, P (2015) Learning to transduce with unbounded memory. [arXiv:1506.02516]
- Griewank A, Walther A (2008) Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. Society for Industrial and Applied Mathematics, Philadelphia [DOI 10.1137/1.9780898717761]
- He K, Zhang X, Ren S, Sun J (2015) Deep residual learning for image recognition. [arXiv:1512.03385]
- Joulin A, Mikolov T (2015) Inferring algorithmic patterns with stack-augmented recurrent nets. [arXiv:1503.01007]
- Maclaurin D, David D, Adams RP (2015) Gradient-based Hyperparameter Optimization through Reversible Learning [arXiv:1502.03492]
- Manzyuk O, Pearlmutter BA, Radul AA, Rush DR, Siskind JM (2012) Confusion of tagged perturbations in forward automatic differentiation of higher-order functions [arXiv:1211.4892]
- Pearlmutter BA, Siskind JM (2008) Reverse-mode AD in a functional framework: Lambda the ultimate backpropagator. ACM TOPLAS 30(2):7 [DOI 10.1145/1330017.1330018]
- Siskind JM, Pearlmutter BA (2008) Nesting forward-mode AD in a functional framework. Higher Order and Symbolic Computation 21(4):361–76 [DOI 10.1007/s10990-008-9037-1]
- Sukhbaatar S, Szlam A, Weston J, Fergus R (2015) Weakly supervised memory networks. [arXiv:1503.08895]
- Syme D (2006) Leveraging .NET meta-programming components from F#: integrated queries and interoperable heterogeneous execution. 2006 Workshop on ML. ACM.
- Vinyals O, Toshev A, Bengio S, Erhan D (2014) Show and tell: a neural image caption generator. [arXiv:1411.4555]
- Wengert R (1964) A simple automatic derivative evaluation program. Communications of the ACM 7:463–4
- Zaremba W, Sutskever I (2015) Reinforcement learning neural Turing machines. [arXiv:1505.00521]