Differentiable Programming

Atılım Güneş Baydin
National University of Ireland Maynooth
(Based on joint work with Barak Pearlmutter)

Microsoft Research Cambridge, February 1, 2016
Deep learning layouts

Neural network models are assembled from **building blocks** and trained with **backpropagation**
Deep learning layouts

Neural network models are assembled from **building blocks** and trained with **backpropagation**

Traditional:
- Feedforward
- Convolutional
- Recurrent
Deep learning layouts

Newer additions:

Make **algorithmic** elements **continuous and differentiable** → enables use in deep learning

- 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)
Deep learning layouts

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)

The bigger picture

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
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.

The bigger picture

These insights clearly put into words in 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”

and reiterated in 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.”
In this talk

Vision:

Functional languages with
- deeply embedded,
- general-purpose

differentiation capability, i.e., differentiable programming
In this talk

Vision:
Functional languages with
- deeply embedded,
- general-purpose
differentiation capability, i.e., differentiable programming

Automatic (algorithmic) differentiation (AD) in a functional framework is a manifestation of this vision.
In this talk

I will talk about:
- Mainstream frameworks
- What AD research can contribute
- My ongoing work
Mainstream Frameworks
Frameworks

“Theano-like”

- Fine-grained
- Define **computational graphs** in a **symbolic** way
- Graph analysis and optimizations

Examples:

- Theano
- Computation Graph Toolkit (CGT)
- TensorFlow
- Computational Network Toolkit (CNTK)

(Kenneth Tran. "Evaluation of Deep Learning Toolkits". 
https://github.com/zer0n/deepframeworks)
Frameworks

“Torch-like”

- Coarse-grained
- Build models by combining pre-specified modules
- Each module is manually implemented, hand-tuned

Examples:
- Torch7
- Caffe
Frameworks

Common in both:

- Define models using the framework’s (constrained) symbolic language
- The framework handles backpropagation → you don’t have to code derivatives (unless adding new modules)
- Because derivatives are “automatic”, some call it “autodiff” or “automatic differentiation”
Frameworks

Common in both:

- Define models using the framework’s (constrained) symbolic language
- The framework handles backpropagation → you don’t have to code derivatives (unless adding new modules)
- Because derivatives are “automatic”, some call it “autodiff” or “automatic differentiation”

This is NOT the traditional meaning of automatic differentiation (AD) (Griewank & Walther, 2008)
Frameworks

Common in both:

- Define models using the framework’s (constrained) symbolic language
- The framework handles backpropagation → you don’t have to code derivatives (unless adding new modules)
- Because derivatives are “automatic”, some call it “autodiff” or “automatic differentiation”

This is NOT the traditional meaning of automatic differentiation (AD) (Griewank & Walther, 2008)

Because “automatic” is a generic (and bad) term, algorithmic differentiation is a better name
“But, how is AD different from Theano?”
“But, how is AD different from Theano?”

In Theano

- express all math relations using symbolic placeholders
- use a mini-language with very limited control flow (e.g. `scan`)
- end up designing a symbolic graph for your algorithm
- Theano optimizes it
“But, how is AD different from Theano?”

Theano gives you automatic derivatives

- Transforms your graph into a derivative graph
- Applies optimizations
  - Identical subgraph elimination
  - Simplifications
  - Stability improvements
    (http://deeplearning.net/software/theano/optimizations.html)
- Compiles to a highly optimized form
“But, how is AD different from Theano?”
You are limited to symbolic graph building, with the mini-language
“But, how is AD different from Theano?”
You are limited to symbolic graph building, with the mini-language
For example, instead of this in pure Python (for $A^k$):

```python
result = 1
for i in xrange(k):
    result = result * A
```
"But, how is AD different from Theano?"

You are limited to symbolic graph building, with the mini-language

For example, instead of this in pure Python (for $A^k$):

```python
result = 1
for i in xrange(k):
    result = result * A
```

You build this symbolic graph:

```python
import theano
import theano.tensor as T

k = T.iscalar("k")
A = T.vector("A")

# Symbolic description of a loop
result, updates = theano.scan(fn=lambda prior_result, A: prior_result * A,
                               outputs_info=T.ones_like(A),
                               non_sequences=A,
                               n_steps=k)

final_result = result[-1]

# compiled function that returns $A^{**k}$
power = theano.function(inputs=[A,k], outputs=final_result, updates=updates)
```
“But, how is AD different from Theano?”

AD allows you to **just fully use your host language** and gives you **exact and efficient derivatives**
“But, how is AD different from Theano?”

AD allows you to **just fully use your host language** and gives you **exact and efficient derivatives**

So, you just do this:

```python
result = 1
for i in xrange(k):
    result = result * A
```
“But, how is AD different from Theano?”

AD allows you to **just fully use your host language** and gives you **exact and efficient derivatives**

So, you just do this:

```python
code
result = 1
for i in xrange(k):
    result = result * A
```

For Python, autograd

https://github.com/HIPS/autograd

Harvard Intelligent Probabilistic Systems Group
Here is the difference

- AD does not use symbolic graphs
- Gives numeric code that **computes the function AND its derivatives** at a given point

```python
f(a, b):
c = a * b
d = sin c
return d
f'(a, a', b, b'):
(c, c') = (a*b, a'*b + a*b')
(d, d') = (sin c, c' * cos c)
return (d, d')
```

- 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)

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

```plaintext
f(2, 3)

a = 2
b = 3

<table>
<thead>
<tr>
<th>c = a * b = 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>d = log c = 1.791</td>
</tr>
</tbody>
</table>

```

```plaintext
a = 2
a' = 1
b = 3
b' = 0

<table>
<thead>
<tr>
<th>c = a * b = 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>c' = a' * b + a * b' = 3</td>
</tr>
<tr>
<td>d = log c = 1.791</td>
</tr>
</tbody>
</table>

| d' = c' * (1 / c) = 0.5 |
```

This is called the forward (tangent) mode of AD.
Function evaluation traces

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

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

f(2, 3)
```

```text
a = 2
b = 3
c = a * b = 6
d = log c = 1.791
return 1.791
```
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)
```

```
a = 2
b = 3
c = a * b = 6
d = log c = 1.791
```

```
return 1.791
```

*(primal)*
Function evaluation traces

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

\[
\begin{align*}
f(a, b): & \\
  c &= a \times b \\
  \text{if } c > 0 & \\
    d &= \log c \\
  \text{else} & \\
    d &= \sin c \\
  \text{return } d
\end{align*}
\]

\[
\begin{align*}
  f(2, 3): & \\
  a &= 2 & a &= 2 \\
  b &= 3 & a' &= 1 \\
  c &= a \times b = 6 & b' &= 0 \\
  d &= \log c = 1.791 & c' &= a' \times b + a \times b' = 3 \\
  \text{return } 1.791 & \\
  \quad (\text{primal}) & \\
  \text{return } 1.791, 0.5 & \\
  \quad (\text{tangent})
\end{align*}
\]

\[
\text{i.e., a Jacobian-vector product }
\]

\[
\begin{align*}
  J_f(a, b) = & \\
  \frac{\partial f(a, b)}{\partial a} & = f'(a, b) |_{(a, b)} = c' = a' \times b + a \times b' = 3 \\
  \frac{\partial f(a, b)}{\partial b} & = f'(a, b) |_{(a, b)} = d' = c' \times (1 / c) = 0.5 \\
  \text{This is called the } \text{forward (tangent) mode of AD}
\end{align*}
\]
Function evaluation traces

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

\[ f(a, b): \]
\[
  c = a \times b \\
  \text{if } c > 0 \\
  \quad d = \log c \\
  \text{else} \\
  \quad d = \sin c \\
  \text{return } d
\]

\[ f(2, 3) \]
\[
  a = 2 \\
  b = 3 \\
  c = a \times b = 6 \\
  d = \log c = 1.791 \\
  \text{return } 1.791
\]

**(primal)**

\[ a’ = 1 \]
\[
  b = 3 \\
  b’ = 0 \\
  c = a \times b = 6 \\
  c’ = a’ \times b + a \times b’ = 3 \\
  d = \log c = 1.791 \\
  d’ = c’ \times (1 / c) = 0.5 \\
  \text{return } 1.791, 0.5
\]

**(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

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

```python
f(2, 3)
```

\[
\begin{align*}
a &= 2 \\
b &= 3 \\
c &= a * b = 6 \\
d &= \log c = 1.791 \\
d' &= 1 \\
c' &= d' * (1 / c) = 0.166 \\
b' &= c' * a = 0.333 \\
a' &= c' * b = 0.5 \\
\end{align*}
\]

This is called the reverse (adjoint) mode of AD. Backpropagation is just a special case of the reverse mode:
Function evaluation traces

\[ f(a, b) : \]
\[
\begin{align*}
    c &= a \times b \\
    \text{if } c > 0 &\quad d = \log c \\
    \text{else} &\quad d = \sin c \\
    \text{return } d
\end{align*}
\]

\( f(2, 3) \)

\( a = 2 \)
\( b = 3 \)
\( c = a \times b = 6 \)
\( d = \log c = 1.791 \)
\( \text{return } 1.791 \)

(\textit{primal})

\[
\begin{align*}
    c' &= \frac{d'}{c} = 0.166 \\
    b' &= c' \times a = 0.333 \\
    a' &= c' \times b = 0.5 \\
    \text{return } 1.791, 0.5, 0.333
\end{align*}
\]

(\textit{adjoint})

i.e., a transposed Jacobian-vector product

\[
\begin{align*}
    J^T f(\mathbf{u}) &= \nabla f(\mathbf{u}) = (0, 0.5, 0.333)
\end{align*}
\]

This is called the reverse (adjoint) mode of AD. Backpropagation is just a special case of the reverse mode:
Function evaluation traces

\[
f(a, b) = \begin{cases} 
  c = a \times b \\
  d = \log c \\
  \text{return } 1.791 \\
\end{cases} \\
\text{if } c > 0 \\
\text{else} \\
\text{return } d \\
\]

\[
f(2, 3) = \\
a = 2 \\
b = 3 \\
c = a \times b = 6 \\
d = \log c = 1.791 \\
\text{return } 1.791 \\
\]

\[
\text{(primal)} \\
\]

\[
\]

\[
\]

\[
\]

\[
\text{(adjoint)} \\
\]

\[
\text{i.e., a transposed Jacobian-vector product} \\
J^T f (\mid one.pnum) \mid \mid (\text{two.pnum, three.pnum}) = \nabla f | (\text{two.pnum, three.pnum}) = \begin{pmatrix} \text{zero.pnum, five.pnum} \\
\text{zero.pnum, three.pnum/three.pnum/three.pnum} \end{pmatrix} \\
\]

\[
\text{This is called the reverse (adjoint) mode of AD} \\
\text{Backpropagation is just a special case of the reverse mode:} \\
\text{code your neural network objective computation, apply reverse AD} \\
\]

\[
\]

\[
\]

\[
\]

\[
\]
Function evaluation traces

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

f(2, 3)
```

```
a = 2
b = 3
c = a * b = 6
d = log c = 1.791
return 1.791
```

```
(primal)
```

```
(primal)
```

```
(adjoint)
```

i.e., a transposed Jacobian-vector product

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

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
Torch-autograd

There are signs that this type of generalized AD will become mainstream in machine learning
Torch-autograd

There are signs that this type of generalized AD will become mainstream in machine learning

A very recent development (November 2015)

**Torch-autograd** by Twitter Cortex (inspired by Python autograd)
https://blog.twitter.com/2015/autograd-for-torch

“autograd has dramatically sped up our model building ... extremely easy to try and test out new ideas”
A cool functional DSL for Torch and Caffe

A side note about the functional interpretation deep learning:

dnngraph by Andrew Tulloch
http://ajtulloch.github.io/dnngraph/

Specify neural network layouts in Haskell,
it gives you Torch and Caffe scripts
What Can AD Research Contribute?
The ambition

- Deeply embedded AD
- Derivatives (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)

I have been working on these issues with Barak Pearlmutter and created DiffSharp (later in the talk)
The ambition

- Deeply embedded AD
- Derivatives (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)

The embodiment of the “differentiable programming” paradigm
The ambition

- Deeply embedded AD
- Derivatives (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)

The embodiment of the “differentiable programming” paradigm

I have been working on these issues with Barak Pearlmutter and created DiffSharp (later in the talk)
AD in a functional framework

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/dysvunctonal-language

- Recently, my DiffSharp library
  http://diffsharp.github.io/DiffSharp/
AD in a functional framework

“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

\[
\min (\lambda x . (f \ x) + \min (\lambda y . g \ x \ y))
\]

(min: gradient descent)
AD in a functional framework

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

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

Nested lambda expressions with free-variable references

\[
\min (\lambda x \cdot (f x) + \min (\lambda y \cdot g x y))
\]

(min: gradient descent)

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

\[
D (\lambda x . x \times (D (\lambda y . x + y) 1)) 1
\]

\[
\frac{d}{d x} \left( x \left( \frac{d}{d y} x + y \right) \bigg|_{y=1} \right) \bigg|_{x=1} \equiv 1
\]
Tricks of the trade
Many methods from AD research

- Hessian-vector products (Pearlmutter, 1994)
- Tape reduction and elimination (Naumann, 2004)
- Utilizing sparsity by matrix coloring (Gebremedhin et al., 2013)

- Reverse AD checkpointing (Dauvergne & Hascoët, 2006)
My Ongoing Work
DiffSharp

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

- AD with linear algebra primitives
- arbitrary nesting of forward/reverse AD
- a comprehensive higher-order API
- gradients, Hessians, Jacobians, directional derivatives, matrix-free Hessian- and Jacobian-vector products
DiffSharp

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

- AD with linear algebra primitives
- arbitrary nesting of forward/reverse AD
- a comprehensive higher-order API
- gradients, Hessians, Jacobians, directional derivatives, matrix-free Hessian- and Jacobian-vector products

Implemented in F#
→ the best tool for this job
→ cross-platform (Linux, Mac OS, Windows)
→ easy deployment with nuget
→ the immense .NET user base of C# and F# users
→ implicit quotations in F# 4.0 is a “killer feature” for deeply embedding transformation-based AD
# DiffSharp

Higher-order differentiation API

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( f : \mathbb{R} \to \mathbb{R} )</td>
<td>diff</td>
<td>( f' )</td>
<td>( \mathbb{R} \to \mathbb{R} \to \mathbb{R} )</td>
<td>X, F</td>
<td>A</td>
</tr>
<tr>
<td>diff'</td>
<td>( f, f' )</td>
<td>( \mathbb{R} \to \mathbb{R} \to \mathbb{R} \times \mathbb{R} )</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>diff2</td>
<td>( f'' )</td>
<td>( \mathbb{R} \to \mathbb{R} )</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>diff2'</td>
<td>( f, f'' )</td>
<td>( \mathbb{R} \to \mathbb{R} \to \mathbb{R} \times \mathbb{R} )</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>diff2'</td>
<td>( f, f', f'' )</td>
<td>( \mathbb{R} \to \mathbb{R} \to \mathbb{R} \times \mathbb{R} \times \mathbb{R} )</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>dffn</td>
<td>( f^{(n)} )</td>
<td>( \mathbb{N} \to (\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to \mathbb{R} )</td>
<td>X, F</td>
<td>X</td>
<td></td>
</tr>
<tr>
<td>dffn'</td>
<td>( f, f^{(n)} )</td>
<td>( \mathbb{N} \to (\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to \mathbb{R} \times \mathbb{R} )</td>
<td>X, F</td>
<td>X</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( f : \mathbb{R}^n \to \mathbb{R} )</td>
<td>grad</td>
<td>( \nabla f )</td>
<td>( \mathbb{R}^n \to \mathbb{R} \to \mathbb{R}^n \to \mathbb{R}^n )</td>
<td>X, R</td>
<td>A</td>
</tr>
<tr>
<td>grad'</td>
<td>( (f, \nabla f) )</td>
<td>( \mathbb{R}^n \to \mathbb{R} \to \mathbb{R}^n \to \mathbb{R}^n )</td>
<td>X, R</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>gradv</td>
<td>( \nabla f \cdot v )</td>
<td>( \mathbb{R}^n \to \mathbb{R} \to \mathbb{R}^n \to \mathbb{R} )</td>
<td>X, F</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>gradv'</td>
<td>( (f, \nabla f \cdot v) )</td>
<td>( \mathbb{R}^n \to \mathbb{R} \to \mathbb{R}^n \to (\mathbb{R} \times \mathbb{R}) )</td>
<td>X, F</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>hessian</td>
<td>( \mathbf{H}_f )</td>
<td>( \mathbb{R}^n \to \mathbb{R} \to \mathbb{R}^n \times \mathbb{R}^n )</td>
<td>X, R-F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>hessian'</td>
<td>( (f, \mathbf{H}_f) )</td>
<td>( \mathbb{R}^n \to \mathbb{R} \to \mathbb{R}^n \to \mathbb{R}^n \times \mathbb{R}^n )</td>
<td>X, R-F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>hessianv</td>
<td>( \mathbf{H}_f v )</td>
<td>( \mathbb{R}^n \to \mathbb{R} \to \mathbb{R}^n \to \mathbb{R}^n )</td>
<td>X, F-R</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>hessianv'</td>
<td>( (f, \mathbf{H}_f v) )</td>
<td>( \mathbb{R}^n \to \mathbb{R} \to \mathbb{R}^n \to \mathbb{R}^n \times \mathbb{R}^n )</td>
<td>X, F-R</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>gradhessian</td>
<td>( \nabla (\nabla f, \mathbf{H}_f) )</td>
<td>( \mathbb{R}^n \to \mathbb{R} \to \mathbb{R}^n \to \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n )</td>
<td>X, R-F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>gradhessian'</td>
<td>( (f, \nabla (\nabla f, \mathbf{H}_f)) )</td>
<td>( \mathbb{R}^n \to \mathbb{R} \to \mathbb{R}^n \to \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n )</td>
<td>X, R-F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>gradhessianv</td>
<td>( \nabla (\nabla f \cdot v, \mathbf{H}_f v) )</td>
<td>( \mathbb{R}^n \to \mathbb{R} \to \mathbb{R}^n \to \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n )</td>
<td>X, F-R</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>gradhessianv'</td>
<td>( (f, \nabla (\nabla f \cdot v, \mathbf{H}_f v)) )</td>
<td>( \mathbb{R}^n \to \mathbb{R} \to \mathbb{R}^n \to \mathbb{R}^n \times \mathbb{R}^n \times \mathbb{R}^n )</td>
<td>X, F-R</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>laplacian</td>
<td>( \nabla^2 f )</td>
<td>( \mathbb{R}^n \to \mathbb{R} \to \mathbb{R}^n \to \mathbb{R} )</td>
<td>X, R-F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>laplacian'</td>
<td>( f, \nabla^2 f )</td>
<td>( \mathbb{R}^n \to \mathbb{R} \to \mathbb{R}^n \times \mathbb{R} )</td>
<td>X, R-F</td>
<td>A</td>
<td>X</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( f : \mathbb{R}^n \to \mathbb{R}^m )</td>
<td>jacobian</td>
<td>( \mathbf{J}_f )</td>
<td>( \mathbb{R}^n \to \mathbb{R}^m \to \mathbb{R}^n \to \mathbb{R}^{m \times n} )</td>
<td>X, F/R</td>
<td>A</td>
</tr>
<tr>
<td>jacobian'</td>
<td>( (f, \mathbf{J}_f) )</td>
<td>( \mathbb{R}^n \to \mathbb{R}^m \to \mathbb{R}^n \to (\mathbb{R}^m \times \mathbb{R}^{m \times n}) )</td>
<td>X, F/R</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>jacobianv</td>
<td>( \mathbf{J}_f v )</td>
<td>( \mathbb{R}^n \to \mathbb{R}^m \to \mathbb{R}^n \to \mathbb{R}^m )</td>
<td>X, F</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>jacobianv'</td>
<td>( (f, \mathbf{J}_f v) )</td>
<td>( \mathbb{R}^n \to \mathbb{R}^m \to \mathbb{R}^n \to (\mathbb{R}^m \times \mathbb{R}^m) )</td>
<td>X, F</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>jacobianT</td>
<td>( \mathbf{J}_f^T )</td>
<td>( \mathbb{R}^n \to \mathbb{R}^m \to \mathbb{R}^n \to \mathbb{R}^{n \times m} )</td>
<td>X, F/R</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>jacobianT'</td>
<td>( (f, \mathbf{J}_f^T) )</td>
<td>( \mathbb{R}^n \to \mathbb{R}^m \to \mathbb{R}^n \to (\mathbb{R}^{m \times n} \times \mathbb{R}^n) )</td>
<td>X, F/R</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>jacobianTv</td>
<td>( \mathbf{J}_f^Tv )</td>
<td>( \mathbb{R}^n \to \mathbb{R}^m \to \mathbb{R}^n \to \mathbb{R}^m \to \mathbb{R}^n )</td>
<td>X, R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jacobianTv'</td>
<td>( (f, \mathbf{J}_f^Tv) )</td>
<td>( \mathbb{R}^n \to \mathbb{R}^m \to \mathbb{R}^n \to (\mathbb{R}^m \times \mathbb{R}^n) )</td>
<td>X, R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jacobianTv''</td>
<td>( (f, \mathbf{J}_f^Tv)(\cdot) )</td>
<td>( \mathbb{R}^n \to \mathbb{R}^m \to \mathbb{R}^n \to (\mathbb{R}^{m \times (\mathbb{R} \to \mathbb{R})}) )</td>
<td>X, R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>curl</td>
<td>( \nabla \times f )</td>
<td>( \mathbb{R}^3 \to \mathbb{R}^3 \to \mathbb{R}^3 )</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>curl'</td>
<td>( (f, \nabla \times f) )</td>
<td>( \mathbb{R}^3 \to \mathbb{R}^3 \to \mathbb{R}^3 \times \mathbb{R}^3 )</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>div</td>
<td>( \nabla \cdot f )</td>
<td>( \mathbb{R}^n \to \mathbb{R} \to \mathbb{R} )</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>div'</td>
<td>( (f, \nabla \cdot f) )</td>
<td>( \mathbb{R}^n \to \mathbb{R} \to \mathbb{R}^n )</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>curldiv</td>
<td>( (\nabla \times f, \nabla \cdot f) )</td>
<td>( \mathbb{R}^3 \to \mathbb{R}^3 \times \mathbb{R}^3 )</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>curldiv'</td>
<td>( (f, \nabla \times f, \nabla \cdot f) )</td>
<td>( \mathbb{R}^3 \to \mathbb{R}^3 \times \mathbb{R}^3 \times \mathbb{R}^3 )</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
</tbody>
</table>
DiffSharp

Matrix operations

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)

Benchmarks tool

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

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

An experimental library for “compositional machine learning and hyperparameter 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
Excerpts 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
Hype

Extracts from Hype optimization code
https://github.com/hypelib/Hype/blob/master/src/Hype/Optimize.fs

Optimization and training as higher-order functions
→ works with any function that you want to describe your data
→ can be composed, curried, nested

```plaintext
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

```haskell
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')) (x0, p0)
```
Hype
But they can use derivatives within their models, if needed
→ input sensitivities
→ complex objective functions
→ adaptive PID controllers
→ integrating differential equations

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

```plaintext
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)
```
Hype

We also provide a Torch-like API for neural networks

```fsharp
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

```fsharp
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

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

I would like to see this work integrated with tools in other languages (C++, Python) and frameworks (Torch, CNTK)
Conclusion
Conclusion

An exciting research area at the intersection of

- programming languages
- functional programming
- machine learning
Beyond deep learning

Applications in probabilistic programming
(Wingate, Goodman, Stuhlmüller, Siskind. “Nonstandard interpretations of probabilistic programs for efficient inference.” 2011)

- Hamiltonian Monte Carlo
- No-U-Turn sampler
- Gradient-based maximum a posteriori estimates

For example, Stan is built on AD
http://mc-stan.org/
(Carpenter et al., 2015)
Other areas

Any work in AD remains applicable to the **traditional application domains of AD** in industry and academia (Corliss et al., 2002)

- Computational fluid dynamics
- Atmospheric chemistry
- Engineering design optimization
- Computational finance
Thank You!

References

- Baydin AG, Pearlmutter BA, Siskind JM (Submitted) DiffSharp: automatic differentiation library [arXiv:1511.07727]