Automatic Differentiation (or Differentiable Programming)

Atilim Gunes Baydin
National University of Ireland Maynooth
Joint work with Barak Pearlmutter

Alan Turing Institute, February 5, 2016
- A brief introduction to AD
- My ongoing work
Vision

Functional programming languages with
- deeply embedded,
- general-purpose

differentiation capability, i.e., *automatic differentiation* (AD) in a functional framework
Vision

Functional programming languages with
  ■ deeply embedded,
  ■ general-purpose
differentiation capability, i.e., **automatic differentiation** (AD) in a functional framework

We started calling this **differentiable programming**

Christopher Olah’s blog post (September 3, 2015)
http://colah.github.io/posts/2015-09-NN-Types-FP/
The AD field

AD is an active research area
http://www.autodiff.org/

Traditional application domains of AD in industry and academia (Corliss et al., 2002; Griewank & Walther, 2008) include

- Computational fluid dynamics
- Atmospheric chemistry
- Engineering design optimization
- Computational finance
AD in probabilistic programming

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

- Hamiltonian Monte Carlo (Neal, 1994)
- No-U-Turn sampler (Hoffman & Gelman, 2011)
- Riemannian manifold HMC (Girolami & Calderhead, 2011)
- Optimization-based inference

Stan (Carpenter et al., 2015)
http://mc-stan.org/
What is AD?

Many machine learning frameworks (Theano, Torch, Tensorflow, CNTK) handle derivatives for you

- You build models by defining **computational graphs**
  → (constrained) symbolic language
  → highly limited control-flow (e.g., Theano’s scan)

- 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”
What is AD?

Many machine learning frameworks (Theano, Torch, Tensorflow, CNTK) handle derivatives for you

- You build models by defining computational graphs
  → (constrained) symbolic language
  → highly limited control-flow (e.g., Theano’s scan)

- 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)
What is AD?

Many machine learning frameworks (Theano, Torch, Tensorflow, CNTK) handle derivatives for you

- You build models by defining **computational graphs**
  - (constrained) symbolic language
  - highly limited control-flow (e.g., Theano’s *scan*)

- 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
What is AD?

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

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

\[
\begin{align*}
f'(a, a', b, b'): & \quad (\text{c}, \text{c}') = (a \times b, a' \times b + a \times b') \\
& \quad (\text{d}, \text{d}') = (\sin \text{c}, \text{c}' \times \cos \text{c}) \\
& \quad \text{return } (\text{d}, \text{d}')
\end{align*}
\]

- 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
```

```text
f(2, 3)

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

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

i.e., a Jacobian-vector product

\[ J_f(a, b) | \begin{pmatrix} 2 \\ 3 \end{pmatrix} \]
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)
```

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

\[
\text{a} = 2 \\
\text{a}' = 1 \\
\text{b} = 3 \\
\text{b}' = 0 \\
\text{c} = \text{a} \times \text{b} = 6 \\
\text{c}' = \text{a}' \times \text{b} + \text{a} \times \text{b}' = 3 \\
\text{d} = \log \text{c} = 1.791 \\
\text{d}' = \text{c}' \times (1 / \text{c}) = 0.5 \\
\text{return } 1.791, 0.5
\] (tangent)

\[
\text{i.e., a Jacobian-vector product }
J_{f}(\text{a, b})|\begin{pmatrix} \text{two} \\ \text{three} \end{pmatrix} = \partial_{\text{a}} f(\text{a, b})|\begin{pmatrix} \text{two} \\ \text{three} \end{pmatrix} = \text{zero, five}.
\] 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)
    a = 2
    b = 3
    c = a * b = 6
    d = log c = 1.791
    return 1.791

    (primal)
```

i.e., a Jacobian-vector product

\[
J_f (a, b) \bigg|_{(2, 3)} = \frac{\partial f(a, b)}{\partial a} \bigg|_{(2, 3)} = \text{zero.pnum}. \text{five.pnum}
\]

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)

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

\[
f(2, 3) \quad \begin{align*}
    a &= 2 \\
    b &= 3 \\
    c &= a \times b = 6 \\
    d &= \log c = 1.791 \\
    \text{return } 1.791
    \end{align*}
\]

\[
\begin{align*}
    a' &= 1 \\
    b' &= 0 \\
    c' &= a' \times b + a \times b' = 3 \\
    d' &= c' \times (1 / c) = 0.5 \\
    \text{return } 1.791, 0.5
    \end{align*}
\]

(i.e., a Jacobian-vector product)

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)

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

\[ f(2, 3) \]
\[ \text{return } 1.791 \]

**(primal)**

\[ a = 2 \quad a' = 1 \]
\[ b = 3 \quad b' = 0 \]
\[ c = a \times b = 6 \]
\[ c' = a' \times b + a \times b' = 3 \]
\[ d = \log c = 1.791 \]
\[ d' = c' \times (1 / c) = 0.5 \]
\[ \text{return } 1.791, 0.5 \]

**(tangent)**

i.e., a Jacobian-vector product \[ J_f (1, 0)\big|_{(2,3)} = \frac{\partial}{\partial a} f(a, b)\big|_{(2,3)} = 0.5 \]

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

f(2, 3)
```

(a, b) = (2, 3)
c = a * b = 6
d = log c = 1.791

(d', c', a', b') = (1, 0.166, 0.5, 0.333)

\[
\begin{align*}
J^T f &= \nabla f |_{(two.pnum, three.pnum)} = (zero.pnum, five.pnum, zero.pnum, three.pnum(three.pnum))
\end{align*}
\]

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

\( f(a, b): \)

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

\( f(2, 3) \)

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

\((\text{primal})\)

\( i.e., \) a transposed Jacobian-vector product

\( J^T f(\text{/one.pnum}) \bigg| (\text{/two.pnum}, \text{/three.pnum}) = \nabla f \bigg| (\text{/two.pnum}, \text{/three.pnum}) = (\text{/zero.pnum}, \text{/five.pnum}, \text{/zero.pnum}, \text{/three.pnum/three.pnum/three.pnum}) \)

This is called the reverse (adjoint) mode of AD. Backpropagation is just a special case of the reverse mode: code a neural network objective computation, apply reverse AD.
Function evaluation traces

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

\( f(2, 3) \)

(\textbf{primal})

\begin{align*}
    a &= 2 \\
    b &= 3 \\
    c &= a \times b = 6 \\
    d &= \log c = 1.791 \\
    \text{return } 1.791
\end{align*}

(\textbf{adjoint})

\begin{align*}
    a &= 2 \\
    b &= 3 \\
    c &= a \times b = 6 \\
    d &= \log c = 1.791 \\
    \quad d' &= 1 \\
    \quad c' &= d' \times \left( \frac{1}{c} \right) = 0.166 \\
    \quad b' &= c' \times a = 0.333 \\
    \quad a' &= c' \times b = 0.5 \\
    \text{return } 1.791, 0.5, 0.333
\end{align*}

\( i.e., \) a transposed Jacobian-vector product

\( J^T f(\|one.pnum\|) = \nabla f\bigg|_{\|two.pnum\|, \|three.pnum\|} = (\|zero.pnum\|, \|five.pnum\|, \|zero.pnum\|.\|three.pnum\|/three.pnum\|/three.pnum\|) \)

This is called the \textbf{reverse (adjoint) mode} of \textbf{AD.}

\textbf{Backpropagation} is just a special case of the reverse mode:
Function evaluation traces

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

\( f(2, 3) \)

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

\( (\text{primal}) \)

\( a' = c' \times b = 0.333 \)
\( b' = c' \times a = 0.5 \)
\( \text{return } 1.791, 0.5, 0.333 \)

\( (\text{adjoint}) \)

i.e., a transposed Jacobian-vector product

\[ J_f^T (1) \big|_{(2, 3)} = \nabla f \big|_{(2, 3)} = (0.5, 0.333) \]

This is called the reverse (adjoint) mode of AD

Backpropagation is just a special case of the reverse mode:

code a neural network objective computation, apply reverse AD
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 & Pearlmutter, 2008; Pearlmutter & 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/dysvunctional-language
- Recently, my DiffSharp library
  http://diffshape.github.io/DiffSharp/
AD in a functional framework

“Generalized AD as a first-class function in an augmented λ-calculus” (Pearlmutter & 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)) \\
\text{(\(\min\): gradient descent)}
\]
AD in a functional framework

“Generalized AD as a first-class function in an augmented λ-calculus” (Pearlmutter & 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)

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

\[
\frac{d}{dx} \left( x \left( \frac{d}{dy} x + y \right) \right)_{y=1} \left. \right|_{x=1} \overset{?}{=} 1
\]
DiffSharp

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

- implemented in F#
- generalizes functional AD to high-performance 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
- F#’s “code quotations” (Syme, 2006) has great potential for deeply embedding transformation-based AD
### DiffSharp

#### Higher-order differentiation API

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$f : R \rightarrow R$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>diff $f'$</td>
<td>$(R \rightarrow R) \rightarrow R \rightarrow R$</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>diff' $(f, f')$</td>
<td>$(R \rightarrow R) \rightarrow R \rightarrow (R \times R)$</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>diff2 $f''$</td>
<td>$(R \rightarrow R) \rightarrow R \rightarrow R$</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>diff2' $(f, f'')$</td>
<td>$(R \rightarrow R) \rightarrow R \rightarrow (R \times R)$</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>diff2'' $(f, f', f'')$</td>
<td>$(R \rightarrow R) \rightarrow R \rightarrow (R \times R \times R)$</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>diffn $f^{(n)}$</td>
<td>$\rightarrow (R \rightarrow R) \rightarrow R \rightarrow R$</td>
<td>X, F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>diffn' $(f, f^{(n)})$</td>
<td>$\rightarrow (R \rightarrow R) \rightarrow R \rightarrow (R \times R)$</td>
<td>X, F</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$f : R^m \rightarrow R^m$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>grad $\nabla f$</td>
<td>$(R^m \rightarrow R) \rightarrow R^m \rightarrow \mathbb{R}^n$</td>
<td>X, R</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>grad' $(f, \nabla f)$</td>
<td>$(R^m \rightarrow R) \rightarrow R^m \rightarrow (R \times \mathbb{R}^m)$</td>
<td>X, R</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>gradv $\nabla f \cdot v$</td>
<td>$(R^m \rightarrow R) \rightarrow R^m \rightarrow R$</td>
<td>X, F</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>gradv' $(f, \nabla f \cdot v)$</td>
<td>$(R^m \rightarrow R) \rightarrow R^m \rightarrow (R \times R)$</td>
<td>X, F</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>hessian $H_f$</td>
<td>$(R^m \rightarrow R) \rightarrow R^m \rightarrow \mathbb{R}^{m \times m}$</td>
<td>X, R-F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>hessian' $(f, H_f)$</td>
<td>$(R^m \rightarrow R) \rightarrow R^m \rightarrow (R \times \mathbb{R}^{m \times m})$</td>
<td>X, R-F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>hessianv $H_f v$</td>
<td>$(R^m \rightarrow R) \rightarrow R^m \rightarrow R^m$</td>
<td>X, F-R</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>hessianv' $(f, H_f v)$</td>
<td>$(R^m \rightarrow R) \rightarrow R^m \rightarrow (R \times \mathbb{R}^{m \times m})$</td>
<td>X, F-R</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>gradhessian $(\nabla f, H_f)$</td>
<td>$(R^m \rightarrow R) \rightarrow R^m \rightarrow \mathbb{R}^{m \times m \times m}$</td>
<td>X, R-F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>gradhessian' $(f, \nabla f, H_f)$</td>
<td>$(R^m \rightarrow R) \rightarrow R^m \rightarrow (R \times \mathbb{R}^{m \times m \times m})$</td>
<td>X, R-F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>gradhessianv $(\nabla f \cdot v, H_f v)$</td>
<td>$(R^m \rightarrow R) \rightarrow R^m \rightarrow (R \times \mathbb{R}^{m \times m \times m})$</td>
<td>X, R-F</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>gradhessianv' $(f, \nabla f \cdot v, H_f v)$</td>
<td>$(R^m \rightarrow R) \rightarrow R^m \rightarrow (R \times \mathbb{R}^{m \times m \times m})$</td>
<td>X, R-F</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>laplacian $\text{tr}(H_f)$</td>
<td>$(R^m \rightarrow R) \rightarrow R^m \rightarrow R$</td>
<td>X, R-F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>laplacian' $(f, \text{tr}(H_f))$</td>
<td>$(R^m \rightarrow R) \rightarrow R^m \rightarrow (R \times R)$</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>
</tr>
</thead>
<tbody>
<tr>
<td>$f : R^m \rightarrow R^m$</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>jacobian $J_f$</td>
<td>$(R^m \rightarrow R^m) \rightarrow R^m \rightarrow \mathbb{R}^{m \times m}$</td>
<td>X, F/R</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>jacobian' $(f, J_f)$</td>
<td>$(R^m \rightarrow R^m) \rightarrow R^m \rightarrow (R^m \times R^m)$</td>
<td>X, F/R</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>jacobianv $J_f v$</td>
<td>$(R^m \rightarrow R^m) \rightarrow R^m \rightarrow R^m$</td>
<td>X, F</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>jacobianv' $(f, J_f v)$</td>
<td>$(R^m \rightarrow R^m) \rightarrow R^m \rightarrow (R^m \times R^m)$</td>
<td>X, F</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>jacobiant $J_f^T$</td>
<td>$(R^m \rightarrow R^m) \rightarrow R^m \rightarrow \mathbb{R}^{m \times m}$</td>
<td>X, F/R</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>jacobiant' $(f, J_f^T)$</td>
<td>$(R^m \rightarrow R^m) \rightarrow R^m \rightarrow (R^m \times R^m)$</td>
<td>X, F/R</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>jacobiantv $J_f^T v$</td>
<td>$(R^m \rightarrow R^m) \rightarrow R^m \rightarrow R^m$</td>
<td>X, R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jacobiantv' $(f, J_f^T v)$</td>
<td>$(R^m \rightarrow R^m) \rightarrow R^m \rightarrow (R^m \times R^m)$</td>
<td>X, R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jacobiantv'' $(f, J_f^T v')$</td>
<td>$(R^m \rightarrow R^m) \rightarrow R^m \rightarrow (R^m \times (R^m \rightarrow R^m))$</td>
<td>X, R</td>
<td></td>
<td></td>
</tr>
<tr>
<td>curl $\nabla \times f$</td>
<td>$(R^3 \rightarrow R^3) \rightarrow R^3 \rightarrow R^3$</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>curl' $(f, \nabla \times f)$</td>
<td>$(R^3 \rightarrow R^3) \rightarrow R^3 \rightarrow (R^3 \times R^3)$</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>div $\nabla \cdot f$</td>
<td>$(R^n \rightarrow R^n) \rightarrow R^n \rightarrow R$</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>div' $(f, \nabla \cdot f)$</td>
<td>$(R^n \rightarrow R^n) \rightarrow R^n \rightarrow (R^n \times R^n)$</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>curl div $(\nabla \times f, \nabla \cdot f)$</td>
<td>$(R^3 \rightarrow R^3) \rightarrow R^3 \rightarrow (R^3 \times R^3)$</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
<tr>
<td>curl div' $(f, \nabla \times f, \nabla \cdot f)$</td>
<td>$(R^3 \rightarrow R^3) \rightarrow R^3 \rightarrow (R^3 \times R^3 \times R^3)$</td>
<td>X, F</td>
<td>A</td>
<td>X</td>
</tr>
</tbody>
</table>
DiffSharp

Matrix operations

High-performance OpenBLAS backend by default, currently working on a CUDA-based GPU backend

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)
Hype
Extracts from Hype neural network code,
freely use F# and higher-order functions, don’t think about
gradients or backpropagation

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

```fsharp
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

Derivatives are instantiated within the optimization code

```ocaml
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'
```
Hamiltonian Monte Carlo with DiffSharp

Try it on your system: http://diffsharp.github.io/DiffSharp/examples-hamiltonianmontecarlo.html

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

let hmc n hdelta hsteps (x0:DV) (f:DV->D) =
  let u x = -log (f x) // potential energy
  let k p = (p * p) / D 2. // kinetic energy
  let hamilton x p = u x + k p
  let x = ref x0
  [for i in 1..n do
   let p = DV.init x0.Length (fun _ -> rndn())
   let x', p' = leapFrog u k hdelta hsteps (!x, p)
   if rnd() < float (exp ((hamilton !x p) - (hamilton x' p')))) then x := x'
   yield !x]
```
Thank You!

References

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