Differentiable Programming

Atılım Güneş Baydin

National University of Ireland Maynooth

(Based on joint work with Barak Pearlmutter)

Microsoft Research Cambridge, February 1, 2016

Hamilton Institute

Maynooth University
National University of Ireland Maynooth

Deep learning layouts

Neural network models are assembled from **building blocks**

and trained with **backpropagation**

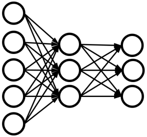
Deep learning layouts

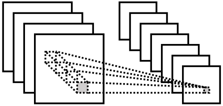
Neural network models are assembled from **building blocks**

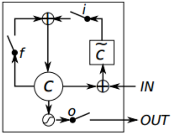
and trained with **backpropagation**

Traditional:

- Feedforward
- Convolutional
- Recurrent



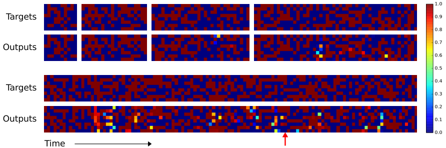




Deep learning layouts

Newer additions:

Make **algorithmic** elements **continuous and differentiable**
$\rightarrow$ enables use in deep learning



NTM on copy task
(Graves et al. 2014)

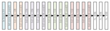
- Neural Turing Machine (Graves et al., 2014)
  $\rightarrow$ can infer algorithms: copy, sort, recall
- Stack-augmented RNN (Joulin & Mikolov, 2015)
- End-to-end memory network (Sukhbaatar et al., 2015)
- Stack, queue, deque (Grefenstette et al., 2015)
- Discrete interfaces (Zaremba & Sutskever, 2015)

Deep learning layouts

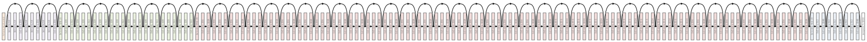
Stacking of many layers, trained through backpropagation

AlexNet, 8 layers (ILSVRC 2012)

VGG, 19 layers (ILSVRC 2014)



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



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

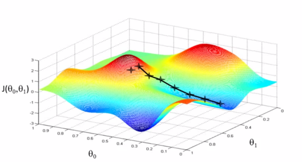
The bigger picture

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

Two main characteristics:

- **Differentiability**

  $\rightarrow$ optimization



- Chained function **composition**

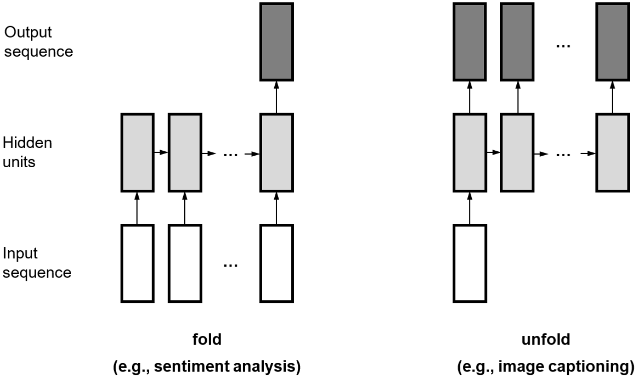
  $\rightarrow$ successive transformations
  $\rightarrow$ successive levels of distributed representations (Bengio 2013)
  $\rightarrow$ the chain rule of calculus propagates derivatives

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

The bigger picture

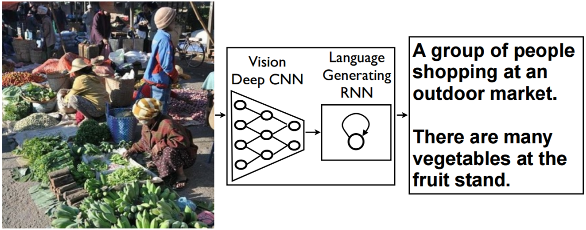
In a functional interpretation

- **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

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

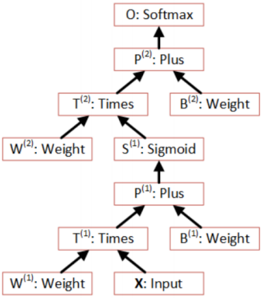
Frameworks

"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)

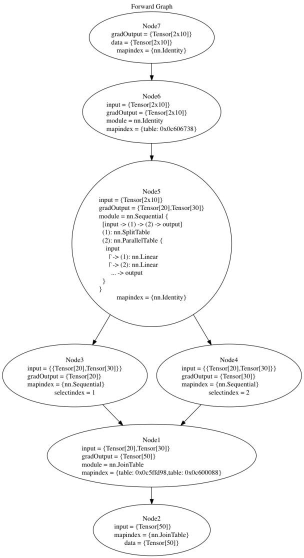
Frameworks

"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
  $\rightarrow$ 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
  $\rightarrow$ 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
  $\rightarrow$ 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)

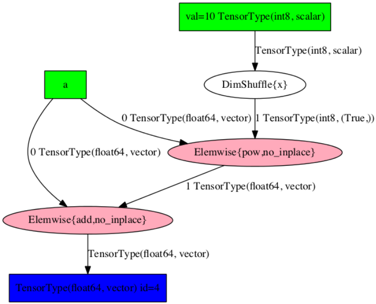
Because "automatic" is a generic (and bad) term, **algorithmic differentiation** is a better name

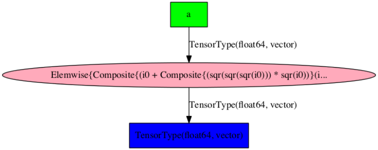
“But, how is AD different from Theano?”

“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):

```
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):

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

You build this symbolic graph:

```
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:

```
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:

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

For Python, autograd

`https://github.com/HIPS/autograd`

Harvard Intelligent Probabilistic Systems Group
(Dougal Maclaurin, David Duvenaud, Ryan P Adams. “Autograd: effortless gradients in Numpy.” 2015)

Here is the difference

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

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

- Derivatives propagated at the elementary operation level, as a side effect, at the same time when the function itself is computed
  $\rightarrow$ Prevents the "expression swell" of symbolic derivatives
- Full expressive capability of the host language
  $\rightarrow$ **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):
   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)

```
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)

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

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)

```
a  = 2
a’ = 1
b  = 3
b’ = 0
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 $\mathbf{J}_f\,(1,0)\big|_{(2,3)} = \frac{\partial}{\partial a} f(a,b)\big|_{(2,3)} = 0.5$

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

Function evaluation traces

```
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):
   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

```
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)

```
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 1.791, 0.5, 0.333
```

(adjoint)

Function evaluation traces

```
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**)

```
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 1.791, 0.5, 0.333
```

(**adjoint**)

i.e., a transposed Jacobian-vector product
$\mathbf{J}_f^T(1)\big|_{(2,3)} = \nabla f\big|_{(2,3)} = (0.5, 0.333)$

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

Backpropagation is just a special case of the reverse mode:
code 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)

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/dysvunctional-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

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

($\min$: gradient descent)

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

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

($\min$: gradient descent)

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

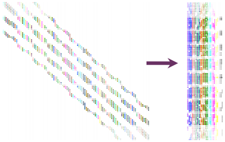
$$D\left(\lambda x\, .\, x \times \left(D\left(\lambda y\, .\, x + y\right) 1\right)\right) 1$$

$$\frac{\mathrm{d}}{\mathrm{d}x}\left(x\left(\frac{\mathrm{d}}{\mathrm{d}y}x + y\right)\Bigg|_{y=1}\right)\Bigg|_{x=1} \stackrel{?}{=} 1$$

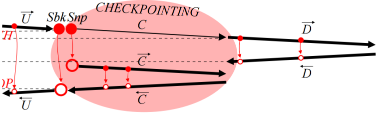
Tricks of the trade

Many methods from AD research

- Hessian-vector products (Pearlmutter, 1994)
- Tape reduction and elimination (Naumann, 2004)
- Context-aware source-to-source transformation (Utke, 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#
$\rightarrow$ the best tool for this job
$\rightarrow$ cross-platform (Linux, Mac OS, Windows)
$\rightarrow$ easy deployment with nuget
$\rightarrow$ the immense .NET user base of C# and F# users
$\rightarrow$ implicit quotations in F# 4.0 is a “killer feature” for deeply embedding transformation-based AD

DiffSharp

Higher-order differentiation API

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

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

An experimental library for “compositional machine learning and **hype**rparameter 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

```
// Use mixed mode nested AD
open DiffSharp.AD.Float32

type FeedForward() =
    inherit Layer()
    // Feedforward layers executed as "fold", DM -> DM
    override n.Run(x:DM) = Array.fold Layer.run x layers

type GRU(inputs:int, memcells:int) =
    inherit Layer()
    // RNN many-to-many execution as "map", DM -> DM
    override l.Run (x:DM) =
        x |> DM.mapCols
            (fun x ->
                let z = sigmoid(l.Wxz * x + l.Whz * l.h + l.bz)
                let r = sigmoid(l.Wxr * x + l.Whr * l.h + l.br)
                let h' = tanh(l.Wxh * x + l.Whh * (l.h .* r))
                l.h <- (1.f - z) .* h' + z .* l.h
                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
→ works with any function that you want to describe your data
→ can be composed, curried, nested

```
// Minimize function `f`
static member Minimize (f:DV->D, w0:DV) =
    Optimize.Minimize (f, w0, Params.Default)

// Train model function `f`
static member Train (f:DV->DV->D, w0:DV, d:Dataset) =
    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

```
type Method
    | CG -> // Conjugate gradient
        fun w f g p gradclip ->
            let v', g' = grad' f w // gradient
            let g' = gradclip g'
            let y = g' - g
            let b = (g' * y) / (p * y)
            let p' = -g' + b * p
            v', g', p'
    | NewtonCG -> // Newton conjugate gradient
        fun w f _ p gradclip ->
            let v', g' = grad' f w  // gradient
            let g' = gradclip g'
            let hv = hessianv f w p // Hessian-vector product
            let b = (g' * hv) / (p * hv)
            let p' = -g' + b * p
            v', g', p'
    | Newton -> // Newton's method
        fun w f _ _ gradclip ->
            let v', g', h' = gradhessian' f w // gradient, Hessian
            let g' = gradclip g'
            let p' = -DM.solveSymmetric h' g'
            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

```
// Leapfrog integrator, Hamiltonian
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)
```

Hype

But they can use derivatives within their models, if needed

→ input sensitivities

→ complex objective functions

→ adaptive PID controllers

→ integrating differential equations

```
// Leapfrog integrator, Hamiltonian
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)
```

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

```
let n = FeedForward()
n.Add(Linear(dim, 100))
n.Add(LSTM(100, 400))
n.Add(LSTM(400, 100))
n.Add(Linear(100, dim))
n.Add(reLU)
```

Hype

We also provide a Torch-like API for neural networks

```
let n = FeedForward()
n.Add(Linear(dim, 100))
n.Add(LSTM(100, 400))
n.Add(LSTM(400, 100))
n.Add(Linear(100, dim))
n.Add(reLU)
```

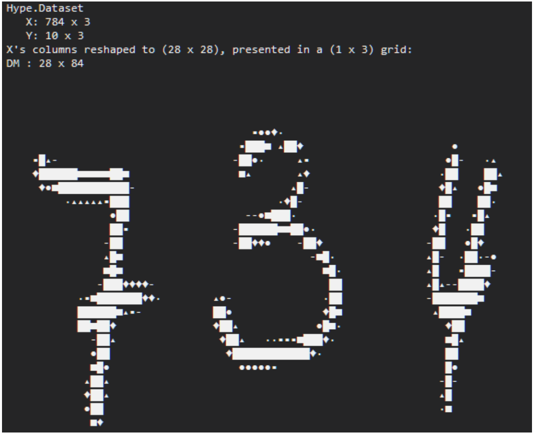
A cool thing: thanks to AD, we can freely code
any F# function as a layer, it just works

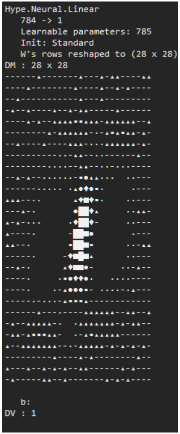
```
n.Add(fun m -> m |> DM.mapCols softmax) // A "map" of softmax

let dropout (x:DM) = // Implement a new layer (dropout)
    x .* (Rnd.UniformDM(x.Cols, x.Rows) |> DM.Round) * 2.f

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)

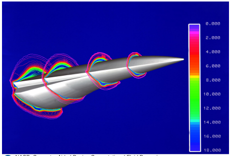
- Hamiltonian Monte Carlo
  `http://diffsharp.github.io/DiffSharp/examples-hamiltonianmontecarlo.html`
- No-U-Turn sampler
- Gradient-based maximum a posteriori estimates

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

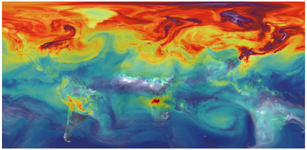
Other areas

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



NASP–Computer Aided Design Computational Fluid Dynamics
NASA Langley Research Center 6/6/1988 Image # EL–1996–00191



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]