Automatic Differentiation for Machine Learning

Niko Brümmer

Cyberupt B.V.

MLSS Stellenbosch
January 2019

Outline

Outline

Introduction

Derivatives are needed everywhere!

Derivatives of complicated, multivariate functions are an essential part of modern machine learning. It is used for:

- Backpropagation in almost any architecture where learning is accomplished via optimization.
- Variational inference (where numerical optimization is required)
- Some MCMC sampling algorithms, e.g. HMC.
- Computing gradient-based regularizers in GANs
- etc.

Modern machine learning tools (Tensorflow, MXNet, PyTorch, . . .) provide automatic support for computing derivatives.

- If you are interested in building a quick solution to your machine learning application, you can just use those tools and invest your energy elsewhere.

But if you want to create new tools, architectures, inference algorithms, optimization algorithms, . . . , a basic understanding of the principles of automatic differentiation could be very useful to you..

automatic differentiation
$\not\equiv$ symbolic differentiation
$\not\equiv$ numerical differentiation

$$\begin{aligned} &\text{automatic differentiation} \\ \not\equiv\ &\text{symbolic differentiation} \\ \not\equiv\ &\text{numerical differentiation} \end{aligned}$$

Symbolic differentiation

- $\frac{d}{dx}\sin(x) = ?$

$$\begin{aligned} &\text{automatic differentiation} \\ \not\equiv\ &\text{symbolic differentiation} \\ \not\equiv\ &\text{numerical differentiation} \end{aligned}$$

Symbolic differentiation

- $\frac{d}{dx}\sin(x) = \cos(x)$

automatic differentiation
$\not\equiv$ symbolic differentiation
$\not\equiv$ numerical differentiation

Symbolic differentiation

- $\frac{d}{dx}\sin(x) = \cos(x)$
- symbolic math tools (e.g. Mathematica) mimic manual differentiation
- not directly applicable for machine learning

$$\begin{aligned} &\text{automatic differentiation} \\ \not\equiv\ &\text{symbolic differentiation} \\ \not\equiv\ &\text{numerical differentiation} \end{aligned}$$

Numerical differentiation (finite differences)

- $\frac{d}{dx}\sin(x) \approx \frac{\sin(x+\delta)-\sin(x-\delta)}{2\delta} \approx \frac{\sin(x+\delta)-\sin(x)}{\delta}$
- requires tuning of δ, inaccurate at best
- intractably slow for large gradients—useless for optimization

automatic differentiation
$\not\equiv$ symbolic differentiation
$\not\equiv$ numerical differentiation

Automatic differentiation

- $\sin(x+\epsilon) = \sin(x) + \epsilon\cos(x)$, ϵ infinitessimal
- does exactly what we need for ML
- computes derivatives to machine precision
- often as fast as hand-optimized code

There are two main flavours of automatic differentiation:

- forward mode
- reverse mode

Forward mode

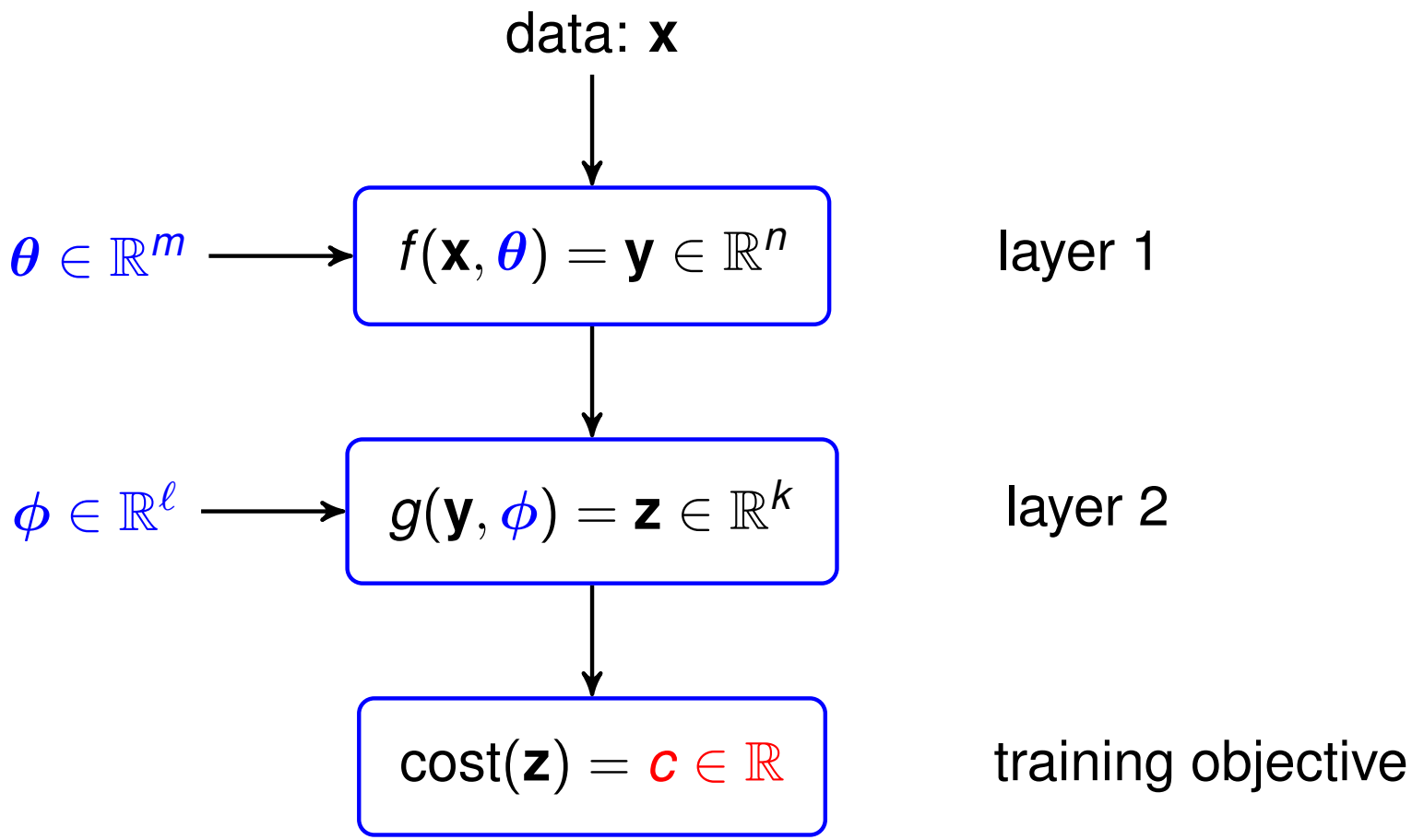
- Relatively easy to code using operator overloading.
  - Can often be done with almost no coding effort using the complex step trick.
- Mathematically ‘cute’.
- Too slow for gradients, but very useful elsewhere:
  - verifying correctness of derivatives provided by other methods,
  - computing Hessian-vector products,
  - etc.

Reverse mode

- Significant coding effort (mostly done already in your favourite ML toolkit).
- On-the-fly calculation / code generation.
- Fast gradient calculations—good for optimization.

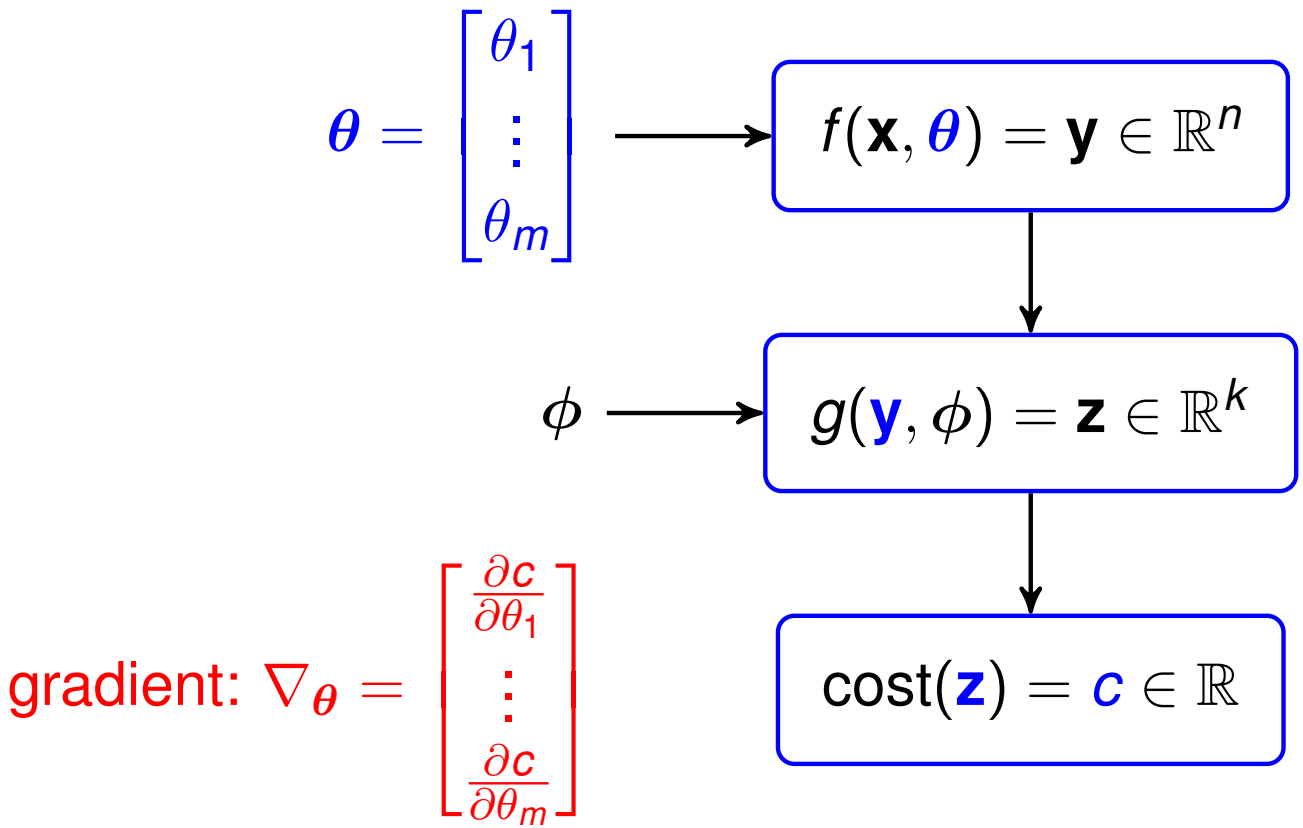
Outline

The typical ML learning problem

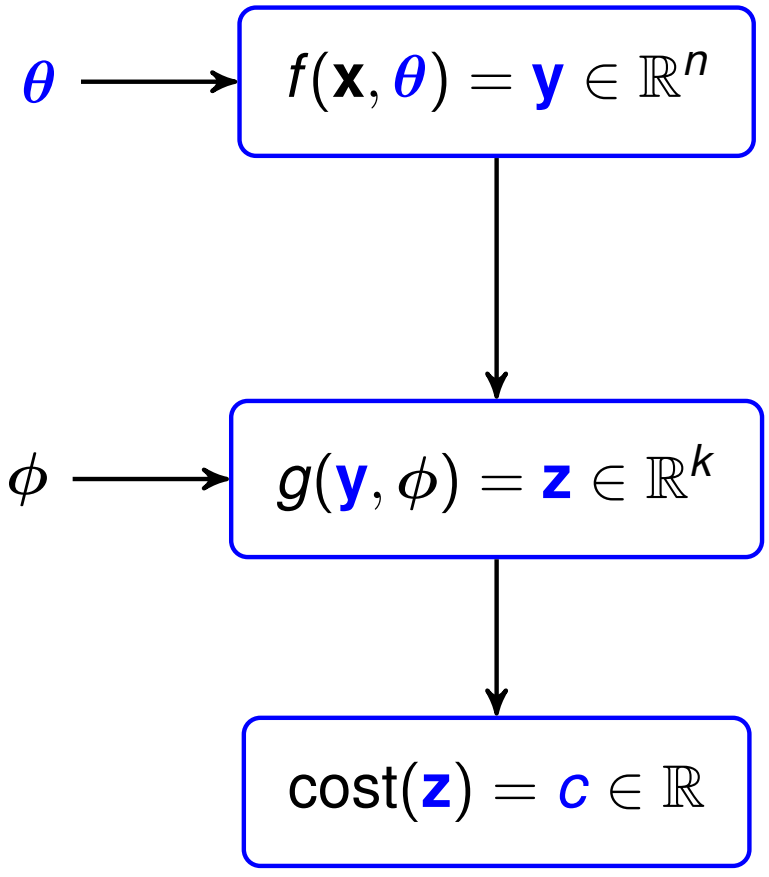


Minimize c by going down parameter gradients, $\nabla_{\boldsymbol{\theta}}$ and $\nabla_{\boldsymbol{\phi}}$.

The gradient



For small changes: $\delta c = \begin{bmatrix} \delta\theta_1 & \cdots & \delta\theta_m \end{bmatrix} \times \nabla_\theta$.



$$\mathbf{J}_1 = \begin{bmatrix} \frac{\partial y_1}{\partial \theta_1} & \cdots & \frac{\partial y_1}{\partial \theta_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_n}{\partial \theta_1} & \cdots & \frac{\partial y_n}{\partial \theta_m} \end{bmatrix}$$

$$\mathbf{J}_2 = \begin{bmatrix} \frac{\partial z_1}{\partial y_1} & \cdots & \frac{\partial z_1}{\partial y_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_k}{\partial y_1} & \cdots & \frac{\partial y_n}{\partial y_n} \end{bmatrix}$$

$$\mathbf{J}_3 = \begin{bmatrix} \frac{\partial c}{\partial z_1} & \cdots & \frac{\partial c}{\partial z_k} \end{bmatrix}$$

$$\nabla_\theta = \begin{bmatrix} \frac{\partial c}{\partial \theta_1} \\ \vdots \\ \frac{\partial c}{\partial \theta_2} \end{bmatrix} = \mathbf{J}_1^T \times \mathbf{J}_2^T \times \mathbf{J}_3^T$$

chain rule:

gradient $= \prod$ Jacobians

Comparison

$$
\begin{aligned}
\nabla_\theta &= \mathbf{J}_1^T \times \mathbf{J}_2^T \times \mathbf{J}_3^T \\
&= \begin{bmatrix} \frac{\partial y_1}{\partial \theta_1} & \cdots & \frac{\partial y_n}{\partial \theta_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial \theta_m} & \cdots & \frac{\partial y_n}{\partial \theta_m} \end{bmatrix} \times \begin{bmatrix} \frac{\partial z_1}{\partial y_1} & \cdots & \frac{\partial z_k}{\partial y_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial z_1}{\partial y_n} & \cdots & \frac{\partial y_n}{\partial y_n} \end{bmatrix} \times \begin{bmatrix} \frac{\partial c}{\partial z_1} \\ \cdots \\ \frac{\partial c}{\partial z_k} \end{bmatrix} \\
&= \left(\mathbf{J}_1^T \times \mathbf{J}_2^T\right) \times \mathbf{J}_3^T, \text{forward mode} \\
&= \mathbf{J}_1^T \times \left(\mathbf{J}_2^T \times \mathbf{J}_3^T\right), \text{reverse mode}
\end{aligned}
$$

Which mode is faster?

Comparison (MNIST example)

computational cost of ∇_θ

operation		cost	MNIST
evaluate	function value	$m + \ell$	10^5
$\mathbf{J}_1^T \times (\mathbf{J}_2^T \times \mathbf{J}_3^T)$	reverse AD	$mn + nk$	10^8
$(\mathbf{J}_1^T \times \mathbf{J}_2^T) \times \mathbf{J}_3^T$	memory!	$mnk + mk$	10^9
forward AD	m forward evals	$m(m + \ell)$	10^{10}
numerical diff.	accuracy!	$m(m + \ell)$	10^{10}

$\mathbf{x} \in \mathbb{R}^{100}$ (input dim. reduced by PCA)

$m = 10^5$ weights in layer 1

$\ell = 10^4$ weights in layer 2

$n = 10^3$ hidden units

$k = 10$ outputs

Outline

Forward AD using dual numbers

Recall complex numbers:

define: $i^2 = -1$

representation: $a + ib$,

$a \in \mathbb{R}$ is the real component
$b \in \mathbb{R}$ is the imaginary component

addition: $(a + ib) + (c + id) = (a + c) + i(b + d)$

multiplication: $(a + ib) \times (c + id) = (ac - bd) + i(ad + bc)$

Forward AD using dual numbers

Dual numbers are very similar:

define: $\epsilon^2 = 0$

representation: $a + \epsilon b$,

$a \in \mathbb{R}$ is the real component
$b \in \mathbb{R}$ is the infinitessimal component

addition: $(a + \epsilon b) + (c + id) = (a + c) + \epsilon(b + d)$

multiplication: $(a + \epsilon b) \times (c + \epsilon d) = ac + \epsilon(ad + bc)$

Notice the multiplicative cross-term, bd, does not appear as in the complex case.

Forward AD using dual numbers

Given some function, $f : \mathbb{R} \to \mathbb{R}$, its extension to the dual numbers can be defined via its Taylor series expansion:

$$
\begin{aligned}
f(a + \epsilon b) &= f(a) + \sum_{n=1}^{\infty} (\epsilon b)^n \frac{f^{(n)}(a)}{n!} \\
&= f(a) + \sum_{n=1}^{1} \epsilon^n b^n \frac{f^{(n)}(a)}{n!}, \qquad \epsilon^2 = \epsilon^3 = \cdots = 0 \\
&= f(a) + \epsilon b f'(a)
\end{aligned}
$$

When evaluating f(dual number), both the function value and the derivative can be recovered from the result!

Forward AD using dual numbers

Example

For $f(x) = x^2$:

$$\begin{aligned}(a+\epsilon)^2 &= a^2 + 2a\epsilon + \epsilon^2 \\ &= a^2 + \epsilon \times 2a \\ &= f(a) + \epsilon \times f'(a)\end{aligned}$$

Forward AD using dual numbers

Multivariate case

For $f : \mathbb{R}^m \to \mathbb{R}^n$, we have for $\mathbf{a}, \mathbf{b} \in \mathbb{R}^m$:

$$f(\mathbf{a} + \epsilon\mathbf{b}) = f(\mathbf{a}) + \epsilon\mathbf{J}(\mathbf{a}) \times \mathbf{b}$$

where $\mathbf{J}(\mathbf{a})$ is the Jacobian of f, evaluated at $\mathbf{a}$.

When evaluating f(dual argument), both the function value and the Jacobian-vector product can be recovered from the result.

- If we want the whole Jacobian, we need to do this m times, sending one-hot vectors into $\mathbf{b}$.
- For $n = 1$, $\nabla_{\mathbf{a}} = \mathbf{J}(\mathbf{a})^T$ is the gradient, and we still need to evaluate f at m different dual number arguments.

Forward AD using dual numbers

Operator overloading

Dual number AD can be implemented in software via operator overloading. Declare a new dual number data type and implement:

- All scalar arithmetic operators.
- All linear algebra operators and functions—and call down into BLAS/LAPACK.
- All other functions that you might need . . .

This is a fair amount of work, but there is a very accurate approximation, which you can do with almost no extra coding effort!

Forward AD using complex step trick

If complex arithmetic and function implementations are available, a very accurate practical approximation for ϵ is:

$$\tilde{\epsilon} = \sqrt{-10^{-40}} = 10^{-20}\sqrt{-1}$$

so that

$$\tilde{\epsilon}^2 = -10^{-40} \approx 0 = \epsilon^2$$

Forward AD using complex step trick

For $f : \mathbb{C} \to \mathbb{C}$ and $a \in \mathbb{R}$, one evaluation at a complex argument can recover both the function value:

$$f(a) \approx \text{real}\{f(a + \tilde{\epsilon})\}$$

and the derivative:

$$f'(a) \approx 10^{20} \, \text{imag}\{f(a + \tilde{\epsilon})\}$$

Forward AD using complex step trick

Multivariate case

For $f : \mathbb{C}^m \to \mathbb{C}^n$, we have for $\mathbf{a}, \mathbf{b} \in \mathbb{R}^m$:

$$f(\mathbf{a} + \tilde{\epsilon}\mathbf{b}) \approx f(\mathbf{a}) + \tilde{\epsilon}\mathbf{J}(\mathbf{a}) \times \mathbf{b}$$

where

$f(\mathbf{a})$ is recovered by taking the real part of the result, and

$\mathbf{J}(\mathbf{a}) \times \mathbf{b}$ is recovered by scaling the imaginary part of the result by 10^{20}.

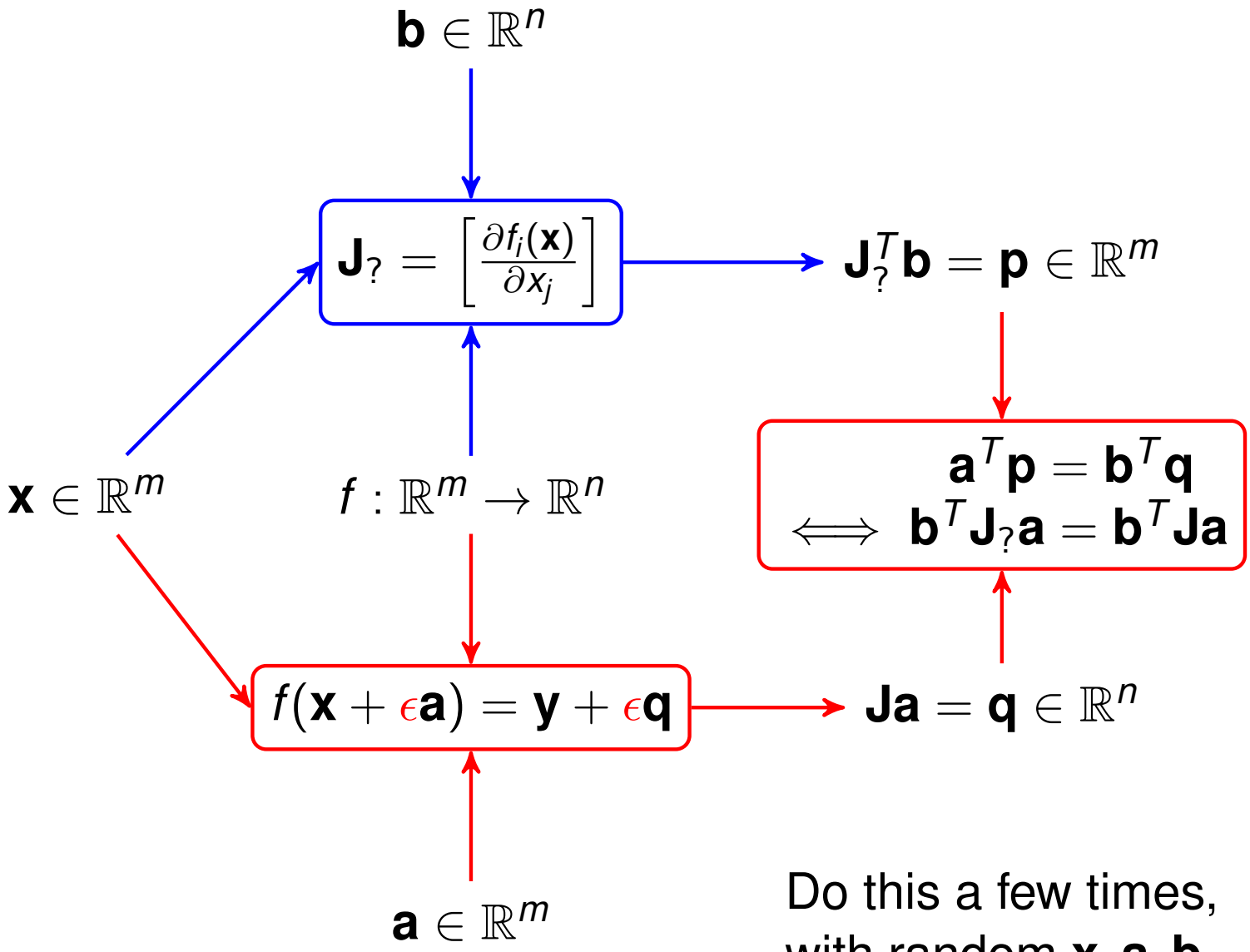
Forward AD using complex step trick

Caveats

The complex step trick is very useful in practice, but watch out for:

- Some functions may not allow complex arguments.
- You can't apply it twice to get second-order derivatives.
- You can't use it to differentiate operations which already use complex arithmetic.
- In matrix operations involving transpose, make sure you do pure transposes, not conjugate transposes.
- Square roots, Cholesky decompositions and other functions that expect positive, or positive definite arguments have to be handled with some extra care.

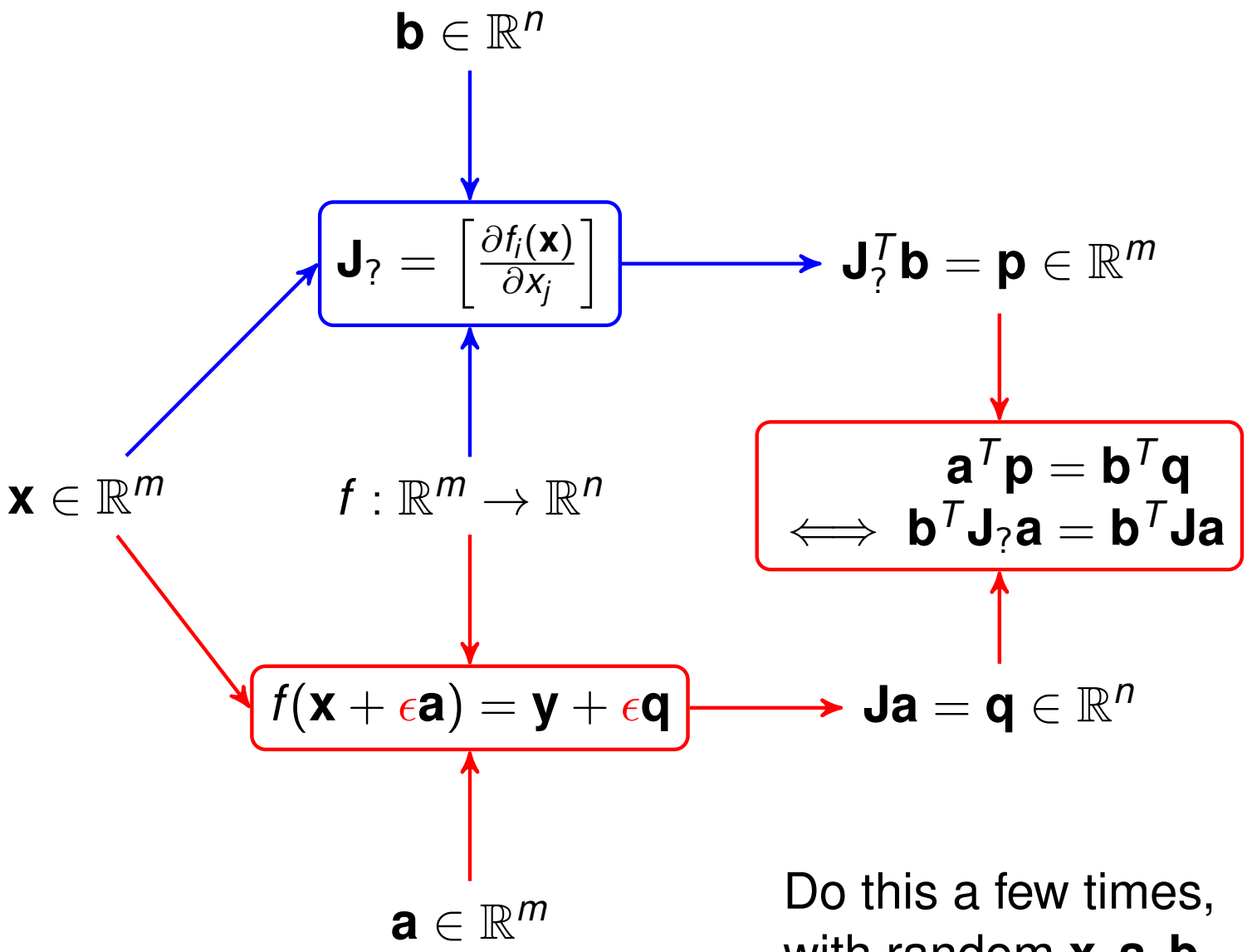
Stochastic derivative verification



Do this a few times,
with random $\mathbf{x}, \mathbf{a}, \mathbf{b}$.

Stochastic derivative verification

If you remember nothing else, remember this technique!



Do this a few times,
with random $\mathbf{x}, \mathbf{a}, \mathbf{b}$.

Hessian-vector products

What are Hessians?

For a function, $f : \mathbb{R}^m \to \mathbb{R}$, the Hessian, evaluated at $y = f(\mathbf{x})$ is the symmetric matrix of second-order partial derivatives:

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 y}{\partial x_1^2} & \cdots & \frac{\partial^2 y}{\partial x_1 \partial x_m} \\ \vdots & \ddots & \vdots \\ \frac{\partial^2 y}{\partial x_m \partial x_1} & \cdots & \frac{\partial^2 y}{\partial x_m^2} \end{bmatrix}$$

Applications of Hessians in machine learning

- Newton-Raphson minimization: $\mathbf{x}^{(i+1)} = \mathbf{x}^{(i)} - \mathbf{H}^{-1}\nabla_{\mathbf{x}}$
- Truncated Newton optimizers approximately solve the linear equation, $\mathbf{H}^{-1}\nabla_{\mathbf{x}}$, using conjugate-gradient iterations that make use of Hessian-vector products: $\mathbf{Hz}$.
- In Bayesian inference, the Laplace approximation requires $\log|\mathbf{H}|$ to approximate the evidence.
- Hyvärinen's score matching objective function for learning the parameters of continuous probability density models, with intractable normalization constants, requires evaluation of $\text{trace}(\mathbf{H})$.

Challenges when working with Hessians

- Machine learning toolkits do not necessarily supply Hessians.
- They are tedious to derive and hand-coding is error-prone.
- For m variables, the Hessian has m^2 components, often much too large to store or work with.
- Solving the linear equation, $\mathbf{H}^{-1}\nabla_{\mathbf{x}}$, has cost $\mathcal{O}(m^3)$.

Hessian-vector products

Solutions

If we can efficiently compute the Hessian-vector product, $\mathbf{Hy}$, for any $\mathbf{y} \in \mathbb{R}^m$, we can solve some of these challenges:

For small m, we can repeatedly send one-hot vectors into $\mathbf{Hy}$ to recover all m columns of $\mathbf{H}$.

- This is useful for fast, high-precision optimization w.r.t. a small number of variables, using Newton-Raphson;
- and for the Laplace approximation in low dimensions.

Hessian-vector products

Solutions

For large m, we can use a relatively small number of Hessian-vector products for:

- Approximate solution of $\mathbf{H}^{-1}\nabla_{\mathbf{x}}$, using the conjugate gradient algorithm. This is used in truncated Newton optimizers.
- If you need it, you can do a truncated eigenanalysis of $\mathbf{H}$ using the Lanczos algorithm.
- For randomly sampled $\mathbf{y} \in \mathbb{R}^m$, such that $\left\langle \mathbf{y}\mathbf{y}^T \right\rangle = \mathbf{I}$:

$$\text{trace}(\mathbf{H}) = \left\langle \mathbf{y}^T \mathbf{H} \mathbf{y} \right\rangle$$

This allows stochastic evaluation of the Hessian trace, using relatively few Hessian-vector products.

Hessian-vector product

Implementation using forward AD

For a function, $f : \mathbb{R}^m \to \mathbb{R}$, the Pearlmutter trick is that we can compute the Hessian vector product, at $f(\mathbf{x})$ as the directed derivative:

$$\mathbf{Hy} = \lim_{\alpha \to 0} \alpha^{-1} \begin{bmatrix} \frac{\partial f(\mathbf{x}+\alpha\mathbf{y})}{\partial x_1} \\ \vdots \\ \frac{\partial f(\mathbf{x}+\alpha\mathbf{y})}{\partial x_m} \end{bmatrix}$$

This can be computed efficiently, with almost no coding effort, using forward-mode AD!

Hessian-vector product

Implementation using forward AD

If our platform allows evaluation of the gradient of f at dual (in practice complex) arguments, the directed derivative can be recovered from the infinitessimal component of the RHS below:

$$\frac{\partial f(\mathbf{x} + \epsilon \mathbf{y})}{\partial x_i} = \frac{\partial f(\mathbf{x})}{\partial x_i} + \epsilon (\mathbf{H}\mathbf{y})_i$$

All of the components of the Hessian-vector product can be computed using a single gradient evaluation.

Summary

Forward mode AD

- Forward mode AD can be theoretically defined in terms of dual numbers.
- It can be implemented exactly (with some effort) using operator overloading,
- or very accurately approximated (with some caveats) using complex numbers.
- It is too slow to evaluate large gradients for optimization.
- It is very useful for verifying derivatives and for Hessian-vector products.

Outline

Reverse mode AD

Reverse mode AD:

- is accurate and fast for computing gradients, good for numerical optimization, and good for many other machine learning problems.
- can be implemented in a variety of ways, including DSLs, operator overloading and hand-coding, all of which we summarize below.
- is available in several modern machine learning toolkits (Tensorflow, MXNet, Pytorch, ...).

Reverse mode AD

Reverse AD computes gradients of function compositions by multiplying by Jacobian transposes in reverse order.

- This requires a mechanism to temporarily store Jacobians during forward evaluation of the function.
- Although typical Jacobians may be huge rectangular matrices, they can nevertheless usually be economically stored.
  - The big affine transforms found in NNs have constant Jacobians (just the weights themselves) and therefore require no extra storage.
  - The array of non-linear activation functions has a diagonal Jacobian.
  - Many Jacobians are rank-one matrices, etc.

Reverse AD implementation

Domain specific language

- The user defines differentiable functions by coding them using some domain specific language (DSL).
- A parser creates a computational graph, for example a directed acyclic graph (DAG).
- The DAG is converted to executable code, to implement both:
  - (forward) evaluation of the function—which effectively stores Jacobians,
  - backpropagation—which essentially multiplies Jacobian transposes in reverse order.
- On-the-fly evaluation of the DAG, rather than code generation is also possible.
- Optimization can be done on the DAG to make calculations faster.

Reverse AD implementation

Operator overloading

- The user defines differentiable functions by coding them in some existing language (C, FORTRAN, MATLAB, Python, Julia, . . . , but maybe not HTML).
- Define a new non-numeric data type, which can be sent into differentiable functions.
- Overload arithmetic and linear algebra operators.
- Overload all the functions you will ever need.
- When evaluated at arguments of this special type, operators and functions create a record of the computation flow, which forms the the computational graph (DAG).
- Once the DAG is available, proceed similarly to the DSL case.

Reverse AD implementation

Hand coding

If you are working on a platform where AD is not available, you can hand-code your own version of reverse AD.

- Acquire a basic understanding of matrix differential calculus. See for example Tom Minka's tutorial, "Old and new matrix algebra useful for statistics".
- Hand-code small functional blocks, each of which returns:
  - The function value(s).
  - A handle that can be used later for backpropagation—and which effectively stores the Jacobian.
- Test every block, using (complex step) forward mode AD—or where necessary fall back on finite differences.
- Also test derivatives of function compositions and also your whole architecture. The stochastic test described earlier is fast enough to test large architectures.

Summary

Reverse mode AD

- Fast, accurate gradients.
- Available in toolkits.
- Can be hand-implemented—from scratch, or to provide missing functions in existing toolkits.
- Use the stochastic forward AD test to check your own and also perhaps third-party reverse-mode implementations.

Summary

of the whole tutorial

- Automatic differentiation is not symbolic or numeric. It does exactly what machine learning requires. It is fast and accurate.
- Forward and reverse modes have complementary capabilities and can be used to verify each other.
- It is widely available in toolkits, but opportunities remain for some of your own hand-coding.

Questions?