SciML - Automatic Differentiation

Mark Asch - IMU/VLP/CSU

2023

Program

1. Automatic differentiation for scientific machine learning:
   (a) Differentiable programming with autograd and PyTorch.
   (b) Gradients, adjoints, backpropagation and inverse problems.
   (c) Neural networks for scientific machine learning.
   (d) Physics-informed neural networks.
   (e) The use of automatic differentiation in scientific machine learning.
   (f) The challenges of applying automatic differentiation to scientific applications.

Differentiable programming

- Differential programming is a technique for automatically computing the derivatives of functions.

- This can be done using a variety of techniques, including:

  ⇒ **Symbolic differentiation**: This involves using symbolic mathematics to represent the function and its derivatives. This can be a powerful technique, but it can be difficult to use for complex functions.

  ⇒ **Numerical differentiation**: This involves using numerical methods to approximate the derivatives of the function. This is a simpler technique than symbolic differentiation, but it is less accurate.

  ⇒ **Automatic differentiation**: This is a technique that combines symbolic and numerical differentiation to automatically compute the derivatives of functions. This is the most powerful technique for differential programming, and it is the most commonly used technique in scientific machine learning.

- The mathematical theory of differential programming is based on the concept of gradients.

  ⇒ The gradient of a function is a vector that tells you how the function changes as its input changes. In other words, the gradient of a function tells you the direction of steepest ascent or descent.
  ⇒ The gradient of a function can be calculated using the gradient descent algorithm. The gradient descent algorithm works by starting at a point and then moving in the direction of the gradient until it reaches a minimum or maximum.
  ⇒ In ML, we use stochastic gradient optimization methods

- Differential programming can be used to solve a variety of problems in scientific machine learning, including:

  ⇒ Calculating the gradients of loss functions for machine learning models—his is important for training machine learning models.
  ⇒ Solving differential equations—this can be used to model the behavior of physical systems.
  ⇒ Performing optimization—this can be used to find

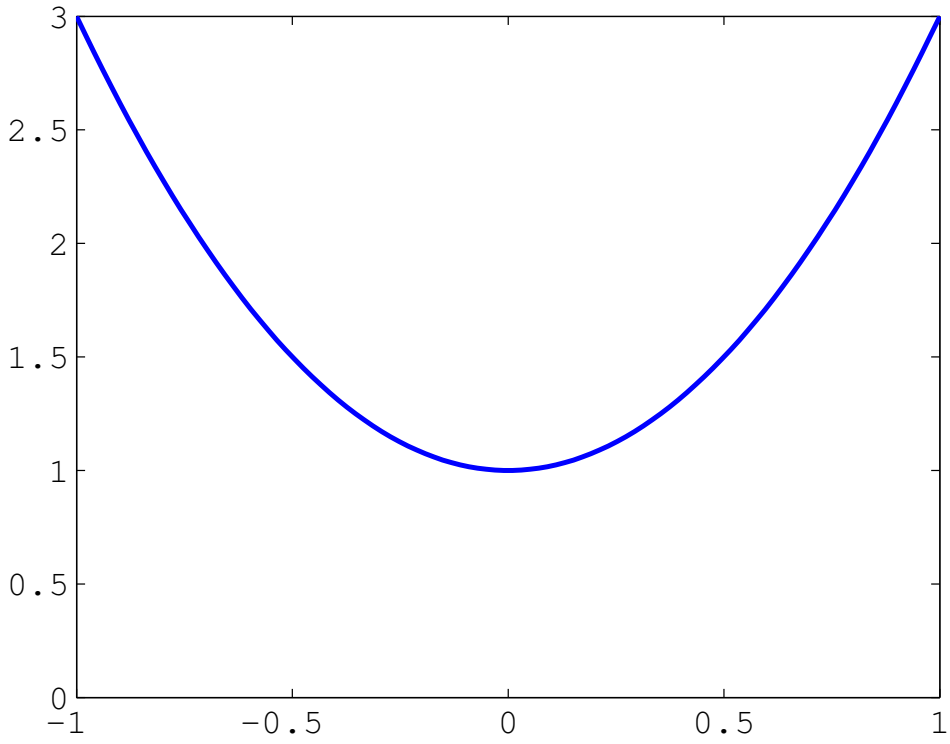
the optimal solution to a problem.

$\Rightarrow$ Solving **inverse and data assimilation problems**—this is none other than a special case of optimization.

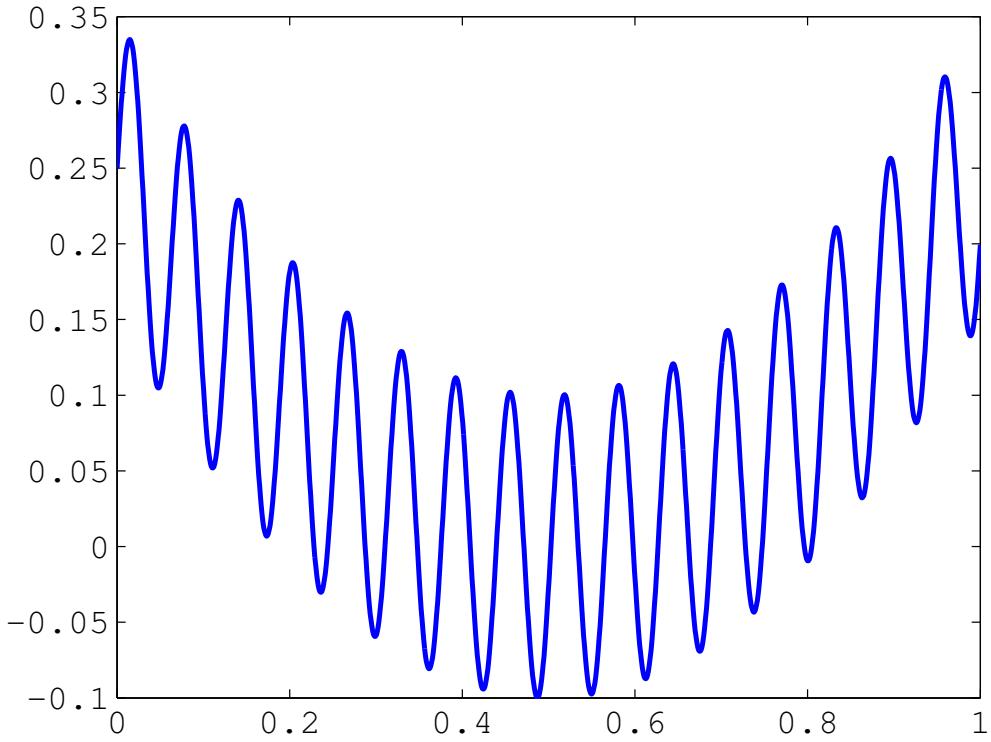
OPTIMIZATION

Recall: Optimization

- Optimization routines typically use local information about a function to iteratively approach a local minimum.

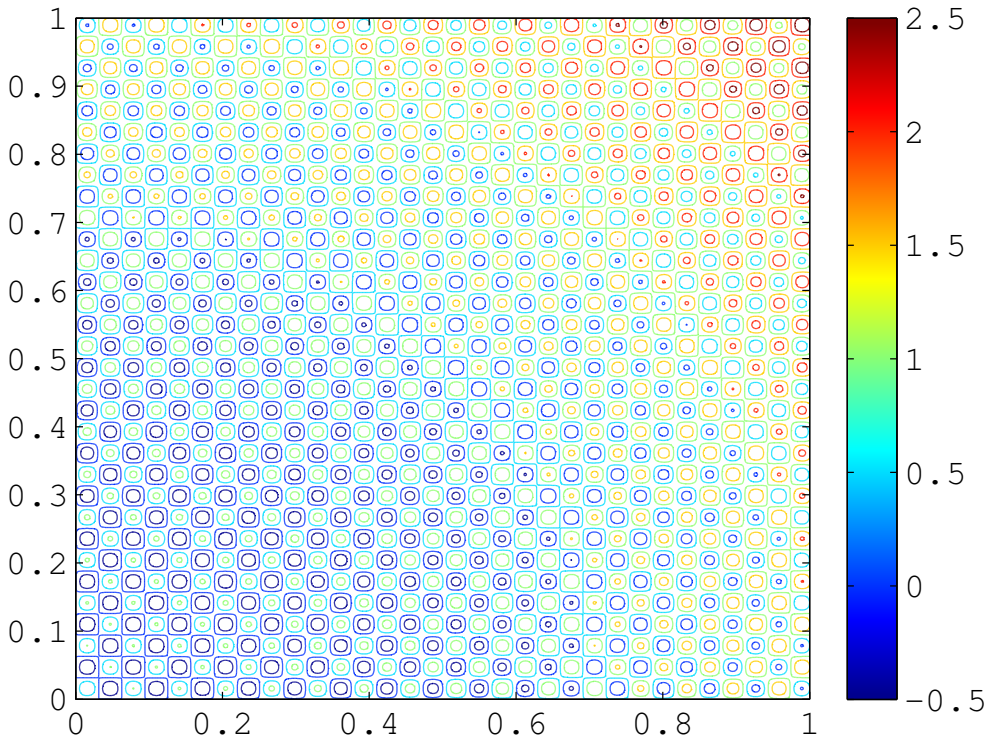


- In this (rare) case, where we have a convex function, we easily find a global minimum.



- But in general, global optimization can be very difficult

$\Rightarrow$ we usually get stuck in local minima!



- Things get MUCH harder in higher spatial dimensions...

  $\Rightarrow$ and much worse when there is some noise in the system.

DIFFERENTIAL PROGRAMMING

Recall: Differentiable Programming

There are 3 ways to compute derivatives of functions:

1. Symbolic differentiation.
2. Numerical differentiation.
3. Automatic differentiation.

- See `04_diff_prog` for simple examples of each of these.
- See `04_diff_Autograd` for use of `Autograd`
- See PyTorch tutorials (next lecture topic) for much more extensive use of AD in numerous optimization, ML and SciML cases.

Symbolic Differentiation

- Computes exact, analytical derivatives, in the form of a mathematical expression.

  ⇒ There is no approximation error.
  ⇒ Operates recursively by applying simple rules to symbols.

✘ There may be no analytical expression for gradients of some functions.

✘ Can lead to redundant and overly complex expressions.

✘ Based on the `sympy` package of Python.

Numerical Differentiation

Definition 1. If f is a differentiable function, then

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

- Using Taylor expansions, and the definition of the derivative, we can obtain finite-difference, numerical approximations to the derivatives of f, such as

$$f'(x) = \frac{f(x+h) - f(x)}{h} + \mathcal{O}(h),$$

$$f'(x) = \frac{f(x+h) - f(x-h)}{2h} + \mathcal{O}(h^2)$$

$\Rightarrow$ conceptually simple and very easy to code

$\Rightarrow$ curse of dimensionality: to compute gradients of $f : \mathbb{R}^m \to \mathbb{R}$, requires at least $\mathcal{O}(m)$ function evaluations

$\Rightarrow$ big numerical errors due to truncation and roundoff.

AUTOMATIC DIFFERENTIATION

Automatic Differentiation

- Automatic differentiation is an umbrella term for a variety of techniques for efficiently computing accurate derivatives of more or less general programs.

  ⇒ It is employed by all major neural network frameworks, where a single reverse-mode AD backpass (also known as "backpropagation") can compute a full gradient.
  ⇒ Numerical differentiation would either require many forward passes or symbolic differentiation that is simply untenable due to expression explosion.
  ⇒ The survey paper [1] provides an excellent review of all the methods and tools available.

- Many algorithms in machine learning, computer vision, physical simulation, and other fields require the calculation of gradients and other derivatives.

  ⇒ Manual derivation of gradients can be both time-consuming and error-prone.

⇒ Automatic differentiation comprises a set of techniques to calculate the derivative of a numerical computation expressed as a computer code.

⇒ These techniques of AD, commonly used for data assimilation in atmospheric sciences and optimal design in computational fluid dynamics, have more recently also been adopted by machine learning researchers.

⇒ The backpropagation algorithm, used for optimally computing the weights of a neural network, is just a special case of general AD.

⇒ AD can be found in all the major software libraries for ML/DL, such as TensorFlow, PyTorch, JaX.

⇒ It is also found in the most recent implementations of MCMC that are based on Hamiltonian Monte Carlo (HMC), such as Stan, PyMC3, Pyro and others—see lecture on Probabilistic Programming.

- Practitioners across many fields have built a wide set of automatic differentiation tools, using different programming languages, computational primitives, and intermediate compiler representations.

  ⇒ Each of these choices comes with positive and negative trade-offs, in terms of their usability, flexibility,

and performance in specific domains.

⇒ Nevertheless, the availability of such tools should not be neglected, since the potential gain from their use is very large.

⇒ Moreover, the fact that they are already built-in to a large number of ML methods, makes their use quite straightforward.

- AD can be readily and extensively used and is thus applicable to many industrial and practical Digital Twin contexts [2].

AD for SciML

- Recent progress in machine learning (ML) technology has been spectacular.

- At the heart of these advances is the ability to obtain high-quality solutions to non-convex optimization problems for functions with billions—or even hundreds of billions—of parameters.

- Incredible opportunity for progress in classical applied mathematics problems.

  ⇒ In particular, the increased proficiency for systematically handling large, non-convex optimization scenarios may help solve some classical problems that have long been a challenge.

  ⇒ We now have the chance to make substantial headway on questions that have not yet been formulated or studied because we lacked the tools to solve them.

  ⇒ To be clear, we do not wish to oversell the state of the art, however:

→ Algorithms that identify the global optimum for non-convex optimization problems do not yet exist.
→ The ML community has instead developed efficient, open source software tools that find candidate solutions.
→ They have created benchmarks to measure solution quality.
→ They have cultivated a culture of competition against these benchmarks.

Automatic Differentiation—backprop, autograd, etc.

- Backprop is a special case of autodiff.
- Autograd is a particular autodiff package.
- In practice, we will pricipally use PyTorch's autodiff functions.

Remark 1. Autodiff is NOT finite differences, nor symbolic differentiation. Finite differences are too expensive (one forward pass for each discrete point). They induce huge numerical errors (truncation/approximation and roundoff) and are very unstable in the presence of noise.

Remark 2. Autodiff is both efficient—linear in the cost of computing the value—and numerically stable.

Remark 3. The goal of autodiff is not a formula, but a procedure for computing derivatives.

Tools for AD

- New opportunities that exist because of the widespread, open-source deployment of effective software tools for automatic differentiation.

- While the mathematical framework for automatic differentiation was established long ago—dating back at least to the evolution of adjoint-based optimization in optimal control[3, 2]—ML researchers have recently designed efficient software frameworks that natively run on hardware accelerators (GPUs).

- These frameworks have served as a core technology for the ML revolution over the last decade and inspired high-quality software libraries such as

  ⇒ JAX,
  ⇒ PyTorch,
  ⇒ TensorFlow.

- The technology's key feature is: **the computational cost of computing derivatives of a target**

loss function is independent of the number of parameters;

$\Rightarrow$ this trait makes it possible for users to implement gradient-based optimization algorithms for functions with staggering numbers of parameters.

AD Sayings

"Gradient descent can write code better than you, I'm sorry."

"Yes, you should understand backprop."

"I've been using PyTorch a few months now and I've never felt better. I have more energy. My skin is clearer. My eye sight has improved."

- Andrej Karpathy [~2017] (Tesla AI, OpenAI)

BACKPROPAGATION

Backpropagation: optimization problem

- We want to solve a (nonlinear, non-convex) optimization problem (in very high dimensions), either

  $\Rightarrow$ for a dynamic system,

  $$\frac{d\mathbf{x}}{dt} = f(\mathbf{x}; \theta),$$

  where $\mathbf{x} \in \mathbb{R}^n$ and $\theta \in \mathbb{R}^p$ with $n, p \gg 1$.

  $\Rightarrow$ or for a machine learning model

  $$\mathbf{y} = f(\mathbf{x}; \mathbf{w}),$$

  where $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{w} \in \mathbb{R}^p$ with $n, p \gg 1$.

- To find the minimum/optimum, we want to minimize an appropriate cost/loss function

  $$J(\mathbf{x}, \theta), \quad \mathcal{L}(\mathbf{w}, \theta)$$

usually some error norm, and then (usually) compute its average

- The best/fastest way to solve this optimization problem, is to use gradients and gradient-based methods.

Backpropagation: definition

- Backpropagation is an algorithm for computing gradients.

- Backpropagation is an instance of reverse mode automatic differentiation

  $\Rightarrow$ very broadly applicable to machine learning, data assimilation and inverse problems in general

  $\Rightarrow$ it is "just" a clever and efficient use of the Chain Rule for derivatives

Backpropagation: theory

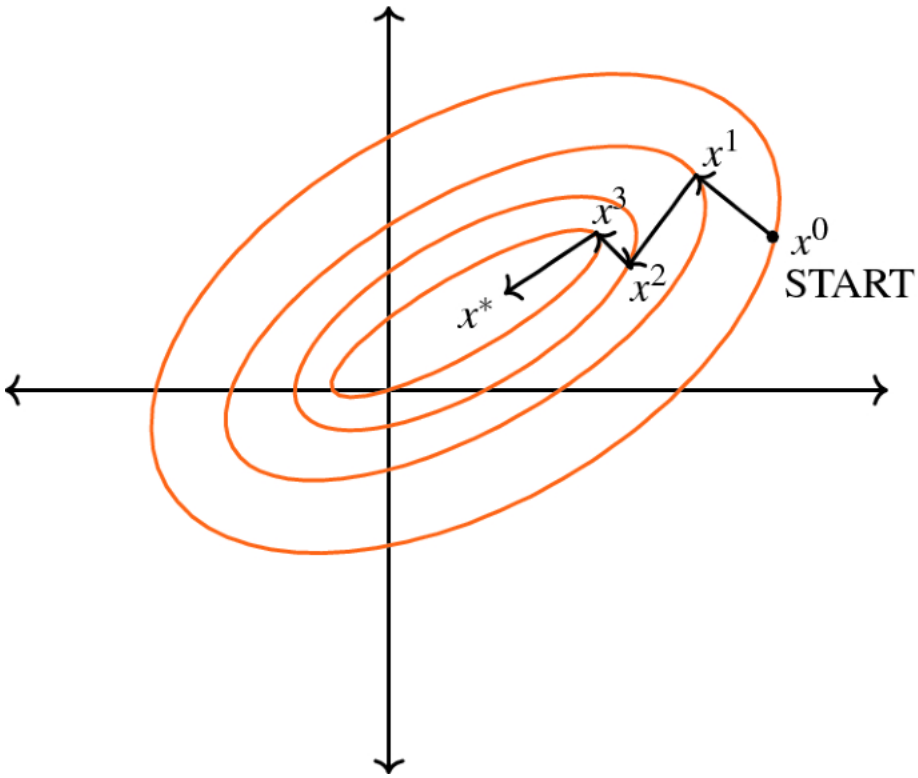
- We can prove mathematically the following equivalences:

  $\Rightarrow$ Backpropagation
  $\Rightarrow$ $\Updownarrow$
  $\Rightarrow$ Reverse-mode automatic differentiation
  $\Rightarrow$ $\Updownarrow$
  $\Rightarrow$ Discrete adjoint-state method

- Recall: the adjoint-state method is the theoretical basis for Data Assimilation, as well as many other inverse problems—see Basic Course, Lecture on Ajjoint Methods (11_DA_adj.pdf).

Recap: Gradient Descent

- Recall: gradient descent moves opposite the gradient, thus in the direction of steepest descent



- What is the optimization parameter space?

  $\Rightarrow$ one coordinate for each weight or bias of the network, in all the layers, for a multilayer neural net

$\Rightarrow$ parameter values at all discrete points, for a (discretized) differential equation

$\rightarrow$ very high dimensiional

$\rightarrow$ very hard to visualize

- we want to compute the cost/loss function gradient, which is usually the average over the training samples of the loss gradient,

$$\nabla_w \mathcal{L} = \frac{\partial \mathcal{L}}{\partial w}, \quad \nabla_\theta \mathcal{L} = \frac{\partial \mathcal{L}}{\partial \theta},$$

or, in general

$$\nabla_z \mathcal{L} = \frac{\partial \mathcal{L}}{\partial z},$$

where $z = w$ or $z = \theta$, etc.

Recap: Chain Rule

- Recall: if $f(x)$ and $x(t)$ are univariate (differentiable) functions, then

$$\frac{\mathrm{d}}{\mathrm{d}t} f(x(t)) = \frac{\mathrm{d}f}{\mathrm{d}x}\frac{\mathrm{d}x}{\mathrm{d}t}$$

- and this can be easily generalized to the multivariate case, such as

$$\frac{\mathrm{d}}{\mathrm{d}t} f(x(t), y(t)) = \frac{\mathrm{d}f}{\mathrm{d}x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\mathrm{d}f}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}t}$$

Chain Rule - a simple example

- Consider
$$f(x, y, z) = (x + y)z$$
- Decompose f into simple differentiable elements
$$q(x, y) = x + y,$$
then
$$f = qz$$
- **Note**: each element has an analytical (exact/known) derivative—eg. sums, products, sines, cosines, min, max, exp, log, etc.
- Now we can compute the gradient of f with respect to its three variables, using the chain rule

$\Rightarrow$ we begin with
$$\frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q$$

and

$$\frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1$$

$\Rightarrow$ then the chain rule gives the terms of the gradient,

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial x} = z \cdot 1$$

$$\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q}\frac{\partial q}{\partial y} = z \cdot 1$$

$$\frac{\partial f}{\partial z} = q$$

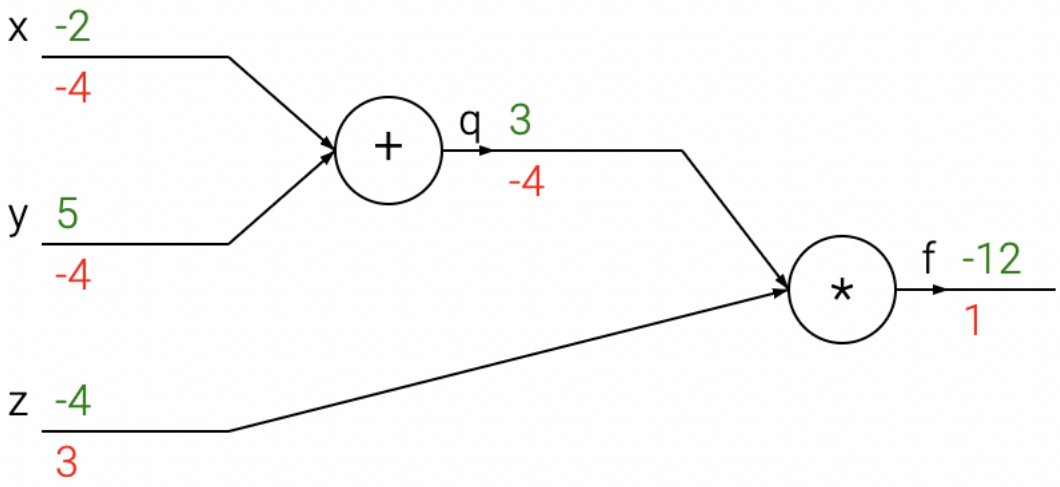
Here is a Python code snippet

```
# set some inputs
x = -2; y = 5; z = -4
# perform the forward pass
q = x + y # q becomes 3
f = q * z # f becomes -12
# perform the backward pass (backpropagation)
# in reverse order:
# first backprop through f = q * z
dfdz = q # df/dz = q, so gradient on z becomes 3
dfdq = z # df/dq = z, so gradient on q becomes -4
dqdx = 1.0
dqdy = 1.0
# now backprop through q = x + y
dfdx = dfdq * dqdx  # The * here is the chain rule
dfdy = dfdq * dqdy
```

- We obtain the gradient in the variables `[dfdx, dfdy, dfdz]` that give us the sensitivity of the function `f` to the variables `x, y` and `z`.

Computational Graphs

- It's all done with graphs... DAGs, in fact
- The above computation can be visualized with a circuit diagram:



- the forward pass, computes values from inputs to outputs
- the backward pass then performs backpropagation, starting at the end and recursively applying the chain rule to

compute the gradients all the way to the inputs of the circuit.

$\Rightarrow$ The gradients can be thought of as flowing backwards through the circuit.

Forward vs Reverse Mode

- Forward mode is used for

  $\Rightarrow$ solving nonlinear equations
  $\Rightarrow$ sensitivity analysis
  $\Rightarrow$ uncertainty propagation/quantification

$$f(x + \Delta x) \approx f(x) + f'(x)\Delta x$$

- Reverse mode is used for

  $\Rightarrow$ machine/deep learning
  $\Rightarrow$ optimization

Backprop - ML example

- For a univariate, logistic least-squares problem, we have:

  $\Rightarrow$ linear model/function of x: $z = wx + b$
  $\Rightarrow$ nonlinear activation: $y = \sigma(x)$
  $\Rightarrow$ quadratic loss: $\mathcal{L} = (1/2)(y - t)^2$, where t is the target/observed value

- **Objective**: find the values of the parameters/weights, w and b, that minimize the loss $\mathcal{L}$

  $\Rightarrow$ to do this, we will use the gradient of $\mathcal{L}$ with respect to the parameters/weights, w and b,

$$\nabla_w \mathcal{L} = \frac{\partial \mathcal{L}}{\partial w}, \quad \nabla_b \mathcal{L} = \frac{\partial \mathcal{L}}{\partial b}$$

Calculus Approach

- Calculus approach:

$$
\begin{aligned}
\mathcal{L} &= \frac{1}{2}(\sigma(wx+b)-t)^2 \\
\frac{\partial \mathcal{L}}{\partial w} &= \frac{\partial}{\partial w}\left[\frac{1}{2}(\sigma(wx+b)-t)^2\right] \\
&= \frac{1}{2}\frac{\partial}{\partial w}(\sigma(wx+b)-t)^2 \\
&= (\sigma(wx+b)-t)\frac{\partial}{\partial w}(\sigma(wx+b)-t) \\
&= (\sigma(wx+b)-t)\sigma'(wx+b)\frac{\partial}{\partial w}(wx+b) \\
&= (\sigma(wx+b)-t)\sigma'(wx+b)x
\end{aligned}
$$

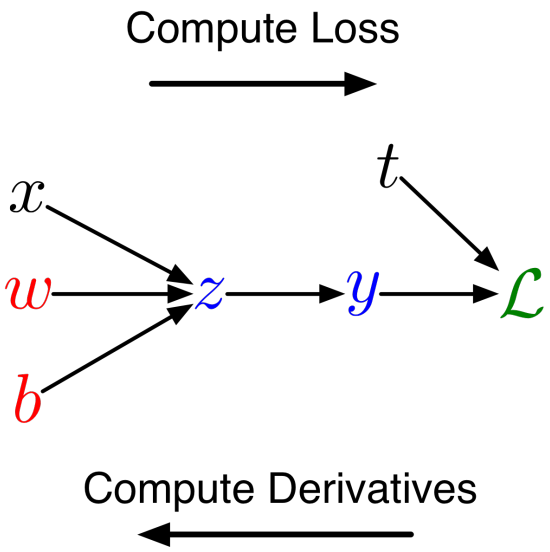
$$
\begin{aligned}
\frac{\partial \mathcal{L}}{\partial b} &= \frac{\partial}{\partial b}\left[\frac{1}{2}(\sigma(wx+b)-t)^2\right] \\
&= \frac{1}{2}\frac{\partial}{\partial b}(\sigma(wx+b)-t)^2 \\
&= (\sigma(wx+b)-t)\frac{\partial}{\partial b}(\sigma(wx+b)-t) \\
&= (\sigma(wx+b)-t)\sigma'(wx+b)\frac{\partial}{\partial b}(wx+b) \\
&= (\sigma(wx+b)-t)\sigma'(wx+b)
\end{aligned}
$$

- It's a mess... too many computations, too complex to program!

- Structured approach:

compute loss	compute derivatives
forwards	backwards
$z = wx + b$	$\frac{\partial \mathcal{L}}{\partial y} = y - t$
$y = \sigma(z)$	$\frac{\partial \mathcal{L}}{\partial z} = \frac{\partial \mathcal{L}}{\partial y}\frac{\partial y}{\partial z} = \frac{\partial \mathcal{L}}{\partial y}\sigma'(z)$
$\mathcal{L} = \frac{1}{2}(y - t)^2$	$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial w} = \frac{\partial \mathcal{L}}{\partial z}x$
	$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial z}\frac{\partial z}{\partial b} = \frac{\partial \mathcal{L}}{\partial z} \cdot 1$

$\Rightarrow$ can easily be written as a computational graph with
- $\rightarrow$ nodes = inputs and computed quantities
- $\rightarrow$ edges = nodes computed directly as functions of other nodes



- Loss is computed in the forward pass
- Gradient is computed in the backward pass
  - $\Rightarrow$ the derivatives of y and z are exact/known
  - $\Rightarrow$ the derivatives of $\mathcal{L}$ are computed, starting from the end
  - $\Rightarrow$ the gradients wrt to the parameters are readily obtained by backpropagation using the chain rule!

Backprop in Practice

- Consider the function

$$f(x,y) = \frac{x + \sigma(y)}{\sigma(x) + (x+y)^2}, \quad \sigma(x) = \frac{1}{1+e^{-x}}$$

- TBC... (exercise)

Full Backprop Algorithm

Let $v_1, \ldots, v_N$ be a topological ordering of the computation graph (i.e. parents come before children.)

v_N denotes the variable we're trying to compute derivatives of (e.g. loss).

forward pass

$$\text{For } i = 1, \ldots, N$$

$$\text{Compute } v_i \text{ as a function of } \mathrm{Pa}(v_i)$$

backward pass

$$\overline{v_N} = 1$$

$$\text{For } i = N-1, \ldots, 1$$

$$\overline{v_i} = \sum_{j \in \mathrm{Ch}(v_i)} \overline{v_j} \frac{\partial v_j}{\partial v_i}$$

where $\bar{v}_i$ denotes the derivatives of the loss function with respect to v_i,

$$\frac{\partial \mathcal{L}}{\partial v_i}$$

- Computational cost of backprop: approximately two forward passes, and hence linear in the number of unknowns

  $\Rightarrow$ Backprop is used to train the overwhelming majority of neural nets today.

⇒ Optimization algorithms, in addition to gradient descent (e.g. second-order methods) use backprop to compute the gradients.

⇒ Backprop can thus be used in SciML, and in particular for Digital Twins (direct and inverse problems), wherever derivatives and/or gradients need to be computed.

AUTOGRAD

Autograd

- `Autograd` can automatically differentiate native Python and Numpy code.

  ⇒ It can handle a large subset of Python's features, including loops, ifs, recursion and closures.

  ⇒ It can even take derivatives of derivatives of derivatives, etc.

  ⇒ It supports reverse-mode differentiation (a.k.a. back-propagation), which means it can efficiently take gradients of scalar-valued functions with respect to array-valued arguments, as well as forward-mode differentiation (to compute sensitivities), and the two can be composed arbitrarily.

  ⇒ The main intended application of Autograd is gradient-based optimization.

- After a function is evaluated, `Autograd` has a graph specifying all operations that were performed on the inputs with respect to which we want to differentiate.

⇒ This is the computational graph of the function evaluation.

⇒ To compute the derivative, we simply apply the basic rules of (analytical) differentiation to each node in the graph.

- Reverse mode differentiation

⇒ Given a function made up of several nested function calls, there are several ways to compute its derivative.

⇒ For example, given

$$L(x) = F(G(H(x))),$$

the chain rule says that its gradient is

$$dL/dx = dF/dG * dG/dH * dH/dx.$$

⇒ If we evaluate this product from right-to-left:

$$(dF/dG * (dG/dH * dH/dx)),$$

the same order as the computations themselves were performed, this is called forward-mode differentiation.

$\Rightarrow$ If we evaluate this product from left-to-right:

$$((dF/dG * dG/dH) * dH/dx),$$

the reverse order as the computations themselves were performed, this is called reverse-mode differentiation.

- Compared to finite differences or forward-mode, reverse-mode differentiation is by far the more practical method for differentiating functions that take in a (very) large vector and output a single number.

- In the machine learning community, reverse-mode differentiation is known as 'backpropagation', since the gradients propagate backwards through the function (as seen above).

- It's particularly nice since you don't need to instantiate the intermediate Jacobian matrices explicitly, and instead only rely on applying a sequence of matrix-free vector-Jacobian product functions (VJPs).

- Because Autograd supports higher derivatives as well,

Hessian-vector products (a form of second-derivative) are also available and efficient to compute.

Remark. Autograd is now being superseded by JAX.

EXAMPLES

Bibliography

References

[1] A. Baydin, B. Pearlmutter, A. Radul, J. Siskind. Automatic differentitation in machine learning: a survey. *Journal of Machine Learning Research,* 18 (2017), article 153.

[2] M. Asch. *Digital Twins: from Model-Based to Data-Driven.* SIAM, 2022.

[3] M. Asch, M. Bocquet, M. Nodet. *Data Assimilation: Theory, Algorithms and Applications.* SIAM, 2016.