A Review of Sensitivity Methods for Differential Equations

Facundo Sapienza fsapienza@berkeley.edu	*Department of Statistics, University of California, Berkeley (USA)*
Jordi Bolibar	*TU Delft, Department of Geosciences and Civil Engineering, Delft (Netherlands)* *Utrecht University, Institute for Marine and Atmospheric research, Utrecht (Netherlands)*
Fernando Pérez	*Department of Statistics, University of California, Berkeley (USA)*

To the community, by the community. This manuscript was conceived with the goal of shortening the gap between developers and practitioners of differential programming tools in modern scientific machine learning. With the advent of new tools and new software, we the authors believe it is important to create pedagogical contents that allow the broader community to understand these methods. Furthermore, we hope this incentives new users to be an active part of the ecosystem, by using and developing open-source tools. This work was done under the premise *open-science from scratch*, meaning all the contents of this work, both code and text, had been in the open from the start and that any interested person can jump in and contribute to the project.

Contents

1 Introduction

Evaluating how changes in the parameter of a certain function plays a central role in optimization, sensitivity analysis, Bayesian inference, uncertainty quantification, among others. Modern machine learning applications require the use of gradients to exploit more efficiently the space of parameters. When optimizing a loss function, gradient-based methods (for example, gradient descent and its many variants [Rud16]) are more efficient at finding minimal and converge faster to them than than gradient-free methods. When numerically computing the posterior of a probabilistic model, gradient-based sampling strategies converge faster to the posterior distribution than gradient-free methods. Second derivatives further help to improve convergence rates of these algorithms and allow uncertainty quantification around parameter values.

Dynamical systems where the goal is to model observations governed by differential equations is not an exception to the rule. In the field of statistics, the sensitivity equations provide a way to compute gradients of a given loss function with respect to the parameters of the dynamical system, which can be used for estimation of the parameters. In numerical analysis, the estimation of the gradient can be used to understand how sensitive is the solution of a differential equation to a parameter. In recent years, there has been an increasing interest in designing machine learning pipelines that include constraints to the physical system in the form of differential equations. Examples of this include physics-informed neural networks (PINNs) [RPK19] and universal differential equations (UDEs) [Rac+20].

However, when working with differential equations the computation of gradients is not an easy task, both regarding the mathematical framework and software implementation involved. Except for a small set of particular cases, most differential equations require numerical methods to estimate their solution. This means that solutions cannot be directly differentiated and require special treatment if, besides the numerical solution, we want to extract first or second order derivatives. Furthermore, numerical solutions introduce approximation errors and these errors can be propagated and even amplified during the computation of the gradient. On the other side, there is a broad literature in numerical methods for solving differential equations. If well each method provides different guarantees and advantages depending the use case, this means that the tools developed to compute gradients when using a solver need to be universal enough in order to apply to all or at least a large set of these. The first goal of this article is making a review of the different methods that exists to archive this goal.

> **Question 1.** *How can we compute the gradient of a function that depends on the numerical solution of a differential equation?*

Notice here the phrasing *the gradient of a function that depends*, emphasizing the fact that in many cases we may be interested in computing the gradient of a function that depends on the solution of the differential equation. This is certainly the case in machine learning and optimization where the goal is to minimize a loss function that depends of some predicted and target responses.

The broader set of tools known as Automatic Differentiation (AD) aims to compute derivatives by sequentially applying the chain rule the the sequence of unit operations that compose a computer program. The premise is simple: every program, including a numerical

solver, is ultimately described by a long chain of simple algebraic operations (addition, multiplication) that are i) easy to differentiate and ii) their combination is easy to differentiate by using the chain rule. If well many modern applications use AD at some extend, there is also a family of methods that compute the gradient by relying in a secondary set of differential equations. We are going to refer to this family of methods as *continuous*, and we will dedicate them a special treatment in future sections to distinguish them from the discrete algorithms that resemble more to pure AD.

The difference between the different methods encapsulates both their formulation and implementation in different software, but also their serve different objectives too. Different methods have different computational complexities depending the number of parameters and differential equations, and these complexities are also balanced between total execution time and required memory. The second goal of this work is then to illustrate the different advantages and limitations of this methods, and how to use them in modern scientific software.

> **Question 2.** *What are the pros and cons of these methods and how can I implement them in my research?*

If well these methods can be (in principle) implemented in different programming languages, here we decided to use the Julia programming language for the different examples. Julia is a recently new and mature programming language that has already a large tradition in implementing packages aiming to advance differential programming [Bez+17].

Without aiming at making an extensive and specialized review on the field, we find this resource useful for other researchers and students working on problems that combine optimization and sensitivity analysis with differential equations. Differential programming is opening new ways

> **Question 3.** *What are the new opportunities in embracing these tools?*

1.1 Use cases

1.2 Some basic jargon

2 General formulation

Consider a system of ordinary differential equations given by

$$\frac{du}{dt} = f(u, \theta, t), \tag{1}$$

where $u \in \mathbb{R}^n$, $\theta \in \mathbb{R}^p$, and initial condition $u(t_0) = u_0$. Here n denotes the total number of ordinary differential equations and p the size of a parameter embedded in the functional form of the differential equation. Although we consider here the case of ordinary differential equations, that is, when the derivatives are just with respect to the time variable t, we will later include the case of partial differential equations (PDE). We are interested in computing the gradient of a given loss function $L(u(\cdot, \theta))$ with respect to the parameter θ. Examples of loss functions include

$$L(u(\cdot, \theta)) = \|u(t_1, \theta) - u_1\|_2^2, \tag{2}$$

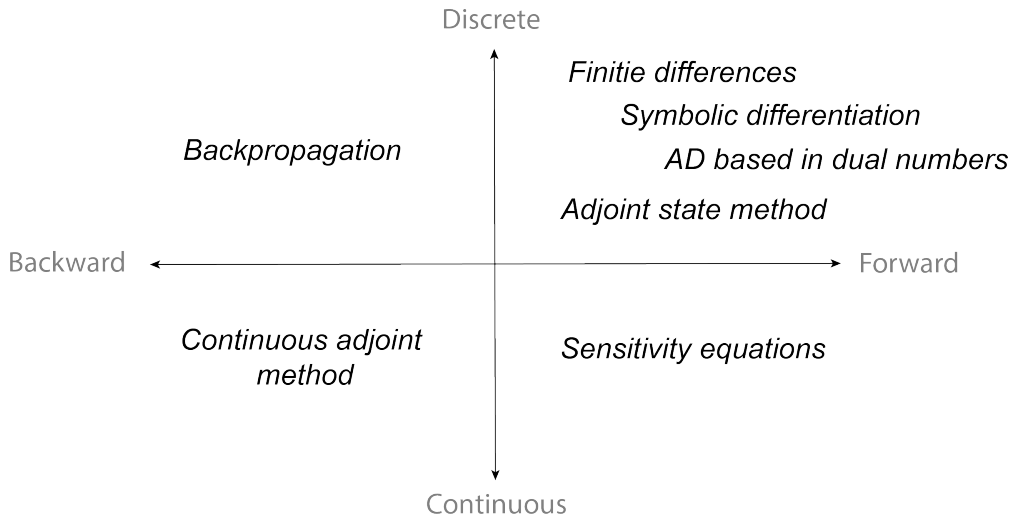

Figure 1: Schematic representation of the different methods available for differentiation involving differential equation solutions.

where u_1 is the desired target observation at some later time t_1; and

$$L(u(\cdot, \theta)) = \int_{t_0}^{t_1} h(u(t; \theta), \theta))dt, \tag{3}$$

with h some function that quantifies the contribution of the error term at time $t \in [t_0, t_1]$. We are interested in computing the gradient of the loss function with respect to the parameter θ, which can be written as

$$\frac{dL}{d\theta} = \frac{dL}{du}\frac{du}{d\theta}. \tag{4}$$

The first term is usually easy to evaluate, since it just involves the partial derivative of the scalar loss function to the solution. The second term on the right hand side is the actual bottleneck and it is usually referred to as the sensitivity,

$$s = \frac{\partial u}{\partial \theta} \in \mathbb{R}^{n \times p}. \tag{5}$$

Depending on the number of parameters and the complexity of the differential equation we are trying to solve, there are different methods to compute gradients with different numerical and computational advantages and that also scale differently depending of the number of differential equations n and number of parameters p. These methods can be roughly classified as:

- *Discrete* vs *continuous* methods.
- *Forward* vs *backwards* methods.

The first difference regards the fact that the method for computing the gradient can be either based on the manipulation of atomic operations that are easy to differentiate using the chain rule several times (discrete), in opposition to the approach of approximating the

gradient as the numerical solution of a new set of differential equations (continuous). The second distinction is related to the fact that some methods compute gradients by resolving a new sequential problem that may move in the same direction of the original numerical solver - i.e. moving forward in time - or, instead, they solve a new system that goes backwards in time. Figure 1 displays a classification of some methods under this two-fold classification. In the following section we are going to explore more in detail these methods.

3 Methods

3.1 Finite differences

The simplest way of evaluating a derivative is by computing the difference between the evaluation of the function at a given point and a small perturbation of the function. In the case of a loss function, we can approximate

$$\frac{dL}{d\theta_i}(\theta) \approx \frac{L(\theta + \varepsilon e_i) - L(\theta)}{\varepsilon}, \tag{6}$$

with e_i the i-th canonical vector and ε a small number. Even better, it is easy to see that the centered difference scheme

$$\frac{dL}{d\theta_i}(\theta) \approx \frac{L(\theta + \varepsilon e_i) - L(\theta - \varepsilon e_i)}{2\varepsilon}, \tag{7}$$

leads also to more precise estimation of the derivative.

However, there are a series of problems associated to this approach. The first one is due to how this scales with the number of parameters p. Each directional derivative requires the evaluation of the loss function L twice. For the centered differences approach in Equation (7), this requires a total of $2p$ function evaluations, which at the same time demands to solve the differential equation in forward mode each time for a new set of parameters. A second problem is due to truncation errors. Equation (6) involves the subtraction of two numbers that are very close to each other.

As ε gets smaller, this will lead to truncation of the subtraction, introducing numerical errors than then will be amplified by the division by ε. Due to this, some heuristics have been introduced in order to pick the value of ε that will minimize the error, specifically picking $\varepsilon^* = \sqrt{\varepsilon_{\text{machine}}}\|\theta\|$, with $\varepsilon_{\text{machine}}$ the machine precision (e.g. `Float64`).

Replacing derivatives by finite differences is also a common practice when solving partial differential equations (PDEs), a technique known as the *method of lines*. To illustrate this point, let's consider the case of the one-dimensional heat equation

$$\frac{\partial u}{\partial t} = D\,\frac{\partial^2 u}{\partial x^2}, \quad u(0,t) = \alpha(t), \quad u(1,t) = \beta(t) \tag{8}$$

that includes both spatial and temporal partial derivatives of the unknown function $u(x,t)$. In order to numerically solve this equation, we can define a spatial grid with coordinates $m\Delta x$, $m = 0, 1, 2, \ldots, N$ and $\Delta x = 1/N$. If we call $u_m(t) = u(m\Delta x, t)$ the value of the

solution evaluated in the fixed points in the grid, then we can replace the second order partial derivative in Equation (8) by the corresponding second order finite difference[1]

$$\frac{du_m}{dt} = D\frac{u_{m-1} - 2u_m + u_{m+1}}{\Delta x^2} \tag{9}$$

for $m = 1, 2, \ldots, N-1$ (in the extremes we simply have $u_0(t) = \alpha(t)$ and $u_N(t) = \beta(t)$). Now, equation (9) is a system of ordinary differential equations (just temporal derivatives) with a total of $N-1$ equations. This can be solved directly using an ODE solver. Further improvements can be made by exploiting the fact that the coupling between the different functions u_m is sparse, that is, the temporal derivative of u_m just depends of the values of the function in the neighbour points in the grid.

3.2 Automatic Differentiation

Automatic differentiation (AD) is a technology that allows computing gradients thought a computer program. The main idea is that every computer program manipulating numbers can be reduced to a sequence of simple algebraic operations that can be easily differentiable. The derivatives of the outputs of the computer program with respect to their inputs are then combined using the chain rule. One advantage of AD systems is that we can automatically differentiate programs that include control flow, such as branching, loops or recursions. This is because at the end of the day, any program can be reduced to a trace of input, intermediate and output variables [Bay+15].

Sometimes AD is compared against symbolic differentiation. According to [Lau19], these two are the same and the only difference is in the data structures used to implement them, while [Ell18] suggests that AD is symbolic differentiation performed by a compiler.

Depending if the concatenation of these gradients is done as we execute the program (from input to output) or in a later instance were we trace-back the calculation from the end (from output to input), we are going to talk about *forward* or *backward* AD, respectively.

Forward mode AD can be implemented in different ways depending on the data structures we use at the moment of representing a computer program. Examples of these data structures include dual numbers and Wengert lists (see [Bay+15] for a good review on these methods). Let's consider the case of dual numbers. We can define an abstract type, defined as a dual number, composed of two elements:

$$x_\epsilon = x_1 + \epsilon x_2, \tag{10}$$

where ϵ is an abstract number with the property $\epsilon^2 = 0$ and $\epsilon \neq 0$. Given two dual numbers $x_\epsilon = x_1 + \epsilon x_2$ and $y_\epsilon = y_1 + \epsilon y_2$, it is easy to derive using the fact $\epsilon^2 = 0$ that

$$x_\epsilon y_\epsilon = x_1 y_1 + \epsilon(x_1 y_2 + x_2 y_1) \qquad \frac{x_\epsilon}{y_\epsilon} = \frac{x_1}{y_1} + \epsilon\frac{x_2 y_1 - x_1 y_2}{y_1^2}. \tag{11}$$

From these last examples, we can see that the dual component of the dual number carries the information of the derivatives when combining operations. Intuitively, we can think about ϵ

[1] Since $u_m(t)$ is a function of one single variable, we write the total derivative $\frac{du_m}{dt}$ instead of the partial derivative symbol used before $\frac{\partial u}{\partial t}$, which it is usually used just for multivariable function.

as being a differential in the Taylor expansion:

$$f(x + \epsilon) = f(x) + \epsilon f'(x) + \mathcal{O}(\epsilon^2) \tag{12}$$

When computing first order derivatives, we can ignore everything of order ϵ^2 or larger, which is represented in the condition $\epsilon^2 = 0$. This implies that we can use dual numbers to implement forward AD through a numerical algorithm. Implementing such a type in a programming language implies defining what it means to perform basic operations and then combine them using the chain rule provided by the dual number properties.[2]

Backward mode AD is also known as the adjoint of cotangent linear mode, or backpropagation in the field of machine learning. Given a directional graph of operations defined by a Wengert list, we can compute gradients of any given function backwards as

$$\bar{v} = \frac{\partial \ell}{\partial v_i} = \sum_{w:v \to w \in G} \frac{\partial w}{\partial v} \bar{w}. \tag{13}$$

Another way of implementing backwards AD is by defining a *pullback* function [Inn18], a method also known as *continuation-passing style* [Wan+19]. In the backward step, this executes a series of functions calls, one for each elementary operation. If one of the nodes in the graph w is the output of an operation involving the nodes $v_1, \ldots, v_m$, where $v_i \to w$ are all nodes in the graph, then the pullback $\bar{v}_1, \ldots, \bar{v}_m = \mathcal{B}_w(\bar{w})$ is a function that accepts gradients with respect to w (defined as $\bar{w}$) and returns gradients with respect to each v_i ($\bar{v}_i$) by applying the chain rule. Consider the example of the multiplicative operation $w = v_1 \times v_2$. Then

$$\bar{v}_1, \bar{v}_2 = v_2 \times \bar{w}, \quad v_1 \times \bar{w} = \mathcal{B}_w(\bar{w}), \tag{14}$$

which is equivalent to using the chain rule as

$$\frac{\partial \ell}{\partial v_1} = \frac{\partial}{\partial v_1}(v_1 \times v_2)\frac{\partial \ell}{\partial w}. \tag{15}$$

In the Julia ecosystem, `ForwardDiff.jl` implements forward mode AD with multidimensional dual numbers [RLP16], while `ReverseDiff.jl` and `Zygote.jl` use callbacks to compute gradients. When gradients are being computed with less than ~ 100 parameters, the former is faster (see documentation).

3.2.1 AD connection with JVPs and VJPs

When working with unit operations that involve matrix operations dealing with vectors of different dimensions, the order in which we apply the chain rule matters. When computing a gradient using AD, we can encounter vector-Jacobian products (VJPs) or Jacobian-vector products (JVP). As their name indicate, the difference between them regards the fact if the quantity we are interested in computing is described by the product of a Jacobian (the two dimensional matrix with the gradients as rows) by a vector on the left side (VJP) or the right (JVP).

[2] See `https://alemorales.info/post/automatic-differentiation-with-dual-numbers/` for an example on how these operations can be implemented in Julia.

For the examples we care here, the Jacobian is described as the product of multiple Jacobian using the chain rule. In this case, the full gradient is computed as the chain product of vectors and Jacobians. Let's consider for example the case of a loss function $L : \mathbb{R}^n \mapsto \mathbb{R}$ that can be decomposed as $L(\theta) = \ell \circ g_k \circ \ldots \circ g_2 \circ g_1(\theta)$, with $\ell : \mathbb{R}^{d_k} \mapsto \mathbb{R}$ the final evaluation of the loss function after we apply in order a sequence of intermediate functions $g_i : \mathbb{R}^{d_{i-1}} \mapsto \mathbb{R}^{d_i}$, $d_0 = n$. Examples of this are neural networks and iterative differential equation solvers. Now, using the chain rule, we can calculate the gradient of the final loss funcition as

$$\nabla_\theta L = \nabla \ell \cdot Dg_k \cdot Dg_{k-1} \cdot \ldots \cdot Dg_2 \cdot Dg_1. \tag{16}$$

Notice that in the last equation, $\nabla \ell \in \mathbb{R}^{d_k}$ is a vector, while all the other term Dg_i are full matrices. In order to compute $\nabla_\theta L$, we can solve the multiplication starting from the right side, which will correspond to multiple the Jacobians forward in time from Dg_1 to Dg_k, or from the left side, moving backwards in time. The important aspect of this last case is that we will always been computing VJPs, since the product of a vector and a matrix is a vector. Since VJP are easier to evaluate than full Jacobians, the backward mode will be in general faster (see Figure 2). For general rectangular matrices $A \in \mathbb{R}^{d_1 \times d_2}$ and $B \in \mathbb{R}^{d_2 \times d_3}$, the cost of the matrix multiplication AB is $\mathcal{O}(d_1 d_2 d_3)$. This implies that forward AD requires a total of

$$d_2 d_1 n + d_3 d_2 n + \ldots + d_k d_{k-1} n + d_k n = \mathcal{O}(kn) \tag{17}$$

operations, while backwards mode AD requires

$$d_k d_{k-1} + d_{k-1} d_{k-2} + \ldots + d_2 d_1 + d_1 n = \mathcal{O}(k + n) \tag{18}$$

operations. In general, when the function we are trying to differentiate has a larger input space than output, which is usually the case when working with scalar loss functions, AD in backward mode is more efficient as it propagates the chain rule by computing VJPs. On the other side, when the output dimension is larger than the input space dimension, forwards AD is more efficient. This is the reason why in most machine learning application people use backwards AD. However, notice that backwards mode AD requires us to save the solution thought the forward run in order to run backwards afterwards, while in forward mode we can just evaluate the gradient as we iterate our sequence of functions. This means that for problems with a small number of parameters, forward mode can be faster and more memory-efficient that backwards AD.

3.3 Sensitivity Equations

An easy way to derive an expression for the sensitivity s is by deriving the sensitivity equations [RH17], a method also referred to as continuous local sensitivity analysis (CSA). If we consider the original system of ODEs and we differentiate with respect to θ, we then obtain

$$\frac{d}{d\theta}\frac{du}{dt} = \frac{d}{d\theta} f(u(\theta), \theta, t) = \frac{\partial f}{\partial \theta} + \frac{\partial f}{\partial u}\frac{\partial u}{\partial \theta}, \tag{19}$$

that is

$$\frac{ds}{dt} = \frac{\partial f}{\partial u} s + \frac{\partial f}{\partial \theta}. \tag{20}$$

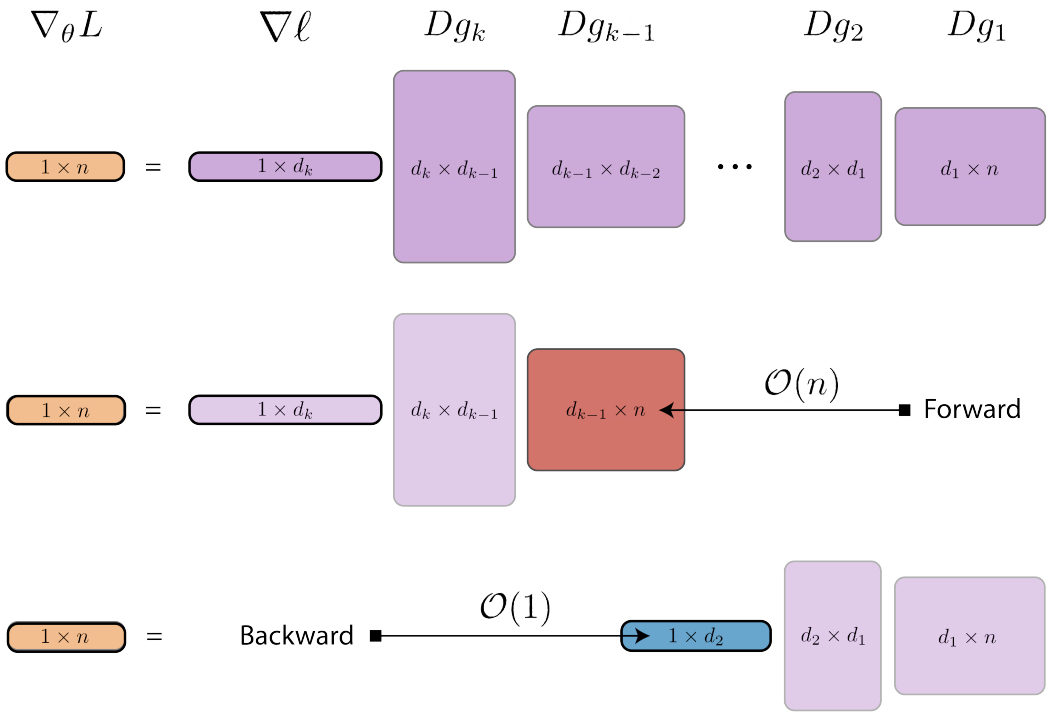

Figure 2: Comparison between forward and backward AD.

By solving the sensitivity equation at the same time we solve the original differential equation for $u(t)$, we ensure that by the end of the forward step we have calculated both $u(t)$ and $s(t)$. This also implies that as we solve the model forward, we can ensure the same level of numerical precision for the two of them.

In opposition to the methods previously introduced, the sensitivity equations find the gradient by solving a new set of continuous differential equations. Notice also that the obtained sensitivity $s(t)$ can be evaluated at any given time t. This method can be labeled as forward, since we solve both $u(t)$ and $s(t)$ as we solve the differential equation forward in time, without the need of backtracking any operation though the solver.

For systems of equations with few number of parameters, this method is useful since the system of equations composed by Equations (1) and (20) can be solved in $\mathcal{O}(np)$ using the same precision for both solution and sensitivity numerical evaluation. Furthermore, this method does not required saving the solution in memory, so it can be solved purely in forward mode without backtracking operations.

3.4 Adjoint state method

The adjoint state method is another example of a forward discrete method, but that computes exact gradient by computationally effectively solving a new set of equations (the adjoint equation) before computing the gradient. Just as for forward automatic differentiation, the adjoint state method evaluates the gradient by moving forward in time and applying the chain rule sequentially over a discrete set of operations that dictate the updates by

the numerical scheme for solving the differential equation. However, it does so by directly computing the gradient by solving a new system of equations. In order to apply the state adjoin method as we solve numerically a system of differential equations, we differentiate a given loss function with respect to one state or evaluation in time of the full solution $u(t;\theta)$. Let's denote $u = u(\tau,\theta)$ the state of the solution at some given time τ. We are interested in differentiating a function $h(u,\theta)$ constrained to satisfy the algebraic equation $g(u,\theta) = 0$.[3] This constraint may come form the numerical scheme we use to solve the differential equation. For an explicit scheme, this may look like $u = u(\tau;\theta) = (I + \Delta\tau\, K)\, u(t-\tau)$, with u being the updated solution when we move Δt forward in time following some numerical scheme defined by a matrix K. Now,

$$\frac{dh}{d\theta} = \frac{\partial h}{\partial \theta} + \frac{\partial h}{\partial u}\frac{\partial u}{\partial \theta}, \tag{21}$$

and also for the constraint $g(u,\theta) = 0$ we can derive

$$\frac{dg}{d\theta} = 0 = \frac{\partial g}{\partial \theta} + \frac{\partial g}{\partial u}\frac{\partial u}{\partial \theta} \quad \Rightarrow \quad \frac{\partial u}{\partial \theta} = -\left(\frac{\partial g}{\partial u}\right)^{-1}\frac{\partial g}{\partial \theta}. \tag{22}$$

If we replace this last expression into equation (21), we obtain

$$\frac{dh}{d\theta} = \frac{\partial h}{\partial \theta} - \frac{\partial h}{\partial u}\left(\frac{\partial g}{\partial u}\right)^{-1}\frac{\partial g}{\partial \theta}. \tag{23}$$

Now, let's define the adjoint $\lambda(t)$ as the solution of the linear system of equations

$$\left(\frac{\partial g}{\partial u}\right)^{T}\lambda = \left(\frac{\partial h}{\partial u}\right)^{T}, \tag{24}$$

that is,

$$\lambda^{T} = \frac{\partial h}{\partial u}\left(\frac{\partial g}{\partial u}\right)^{-1}. \tag{25}$$

Finally, if we replace Equation (25) into (23), we obtain

$$\frac{dh}{d\theta} = \frac{\partial h}{\partial \theta} - \lambda^{T}\frac{\partial g}{\partial \theta}. \tag{26}$$

The important trick to notice here is the rearrangement of the multiplicative terms involved in equation (23). Computing the full Jacobian/sensitivity $\partial u/\partial\theta$ will be computationally expensive and involves the product of two matrices. However, we are not interested in the calculation of the Jacobian, but instead in the VJP given by $\frac{\partial g}{\partial u}\frac{\partial u}{\partial \theta}$. By rearranging these terms, we can make the same computation more efficient. Let's see this by considering the simple example of a linear system of equations. Suppose that the constraint $g(u,\theta) = 0$ takes the form $A(\theta)u = b(\theta)$ [Joh12]. In that case, we have

$$\frac{\partial g}{\partial \theta} = \frac{\partial A}{\partial \theta}u - \frac{\partial b}{\partial \theta}, \tag{27}$$

[3] A useful tutorial on this is provided in `https://www.youtube.com/watch?v=wNiulgucIbc`.

so the gradient can be computed as

$$\frac{dh}{d\theta} = \frac{\partial h}{\partial \theta} - \lambda^T \left(\frac{\partial A}{\partial \theta} u - \frac{\partial b}{\partial \theta} \right) \tag{28}$$

with λ the solution of the linear system

$$A(\theta)^T \lambda = \frac{\partial h}{\partial u}^T. \tag{29}$$

This is a linear system of equations with the same size of the original $Au = b$, but involving the adjoint matrix A^T. Computationally this also means that if we can solve the original system then we can also solve the adjoint (for example, with LU factorization $A = LU$ we have $A^T = U^T L^T$). This example also allows us to see the improvement in efficiency achieved by first computing the adjoint and then the full gradient. For a linear constraint, equation (28) becomes

$$\frac{dh}{d\theta} = \frac{\partial h}{\partial \theta} - \underbrace{\frac{\partial h}{\partial u}}_{1\times n} \underbrace{A^{-1}}_{n\times n} \underbrace{\left(\frac{\partial A}{\partial \theta} u - \frac{\partial b}{\partial \theta} \right)}_{n\times p}. \tag{30}$$

Computing first the last product will cost $O(n^2 p)$ to generate a $n \times p$ matrix. On the other side, if first we solve the first two terms in the product, this will cost $O(n)$ and them the product will be just $O(np)$. In order words, it is easy to compute the full gradient if we treat the last term in equation (30) as a VJP.

In order to compute the gradient of the full solution of the differential equation, we apply this method sequentially using the chain rule. One single step of the state method can be understood as the chain of operations $\theta \mapsto g \mapsto u \mapsto L$. This allows us to create adjoints for any primitive function g (i.e. the numerical solver scheme) we want, and then incorporated it as a unit of any AD program.

3.5 Continous Adjoint Sensitivity Analysis (CASA)

This is a generalization of the adjoint state method to the continuous domain, where now instead we are going to define an adjoint $\lambda(t)$ that is a function of time and that solves its own set of differential equations [Ma+21]. Consider an integrated loss function of the form

$$L(u; \theta) = \int_{t_0}^{t_1} h(u(t; \theta), \theta) dt, \tag{31}$$

which includes the simple case of the loss function $\mathcal{L}_k(\theta)$ in Equation (**??**). Using the Lagrange multiplier trick, we can write a new loss function $I(\theta)$ identical to $L(\theta)$ as

$$I(\theta) = L(\theta) - \int_{t_0}^{t_1} \lambda(t)^T \left(\frac{du}{dt} - f(u, t, \theta) \right) dt \tag{32}$$

where $\lambda(t) \in \mathbb{R}^n$ is the Lagrange multiplier of the continuous constraint defined by the differential equation. Now,

$$\frac{dL}{d\theta} = \frac{dI}{d\theta} = \int_{t_0}^{t_1} \left(\frac{\partial h}{\partial \theta} + \frac{\partial h}{\partial u} \frac{\partial u}{\partial \theta} \right) dt - \int_{t_0}^{t_1} \lambda(t)^T \left(\frac{d}{dt} \frac{du}{d\theta} - \frac{\partial f}{\partial u} \frac{du}{d\theta} - \frac{\partial f}{\partial \theta} \right) dt. \tag{33}$$

Notice that the term involved in the second integral is the same we found when deriving the sensitivity equations. We can derive an easier expression for the last term using integration by parts. Using our usual definition of the sensitivity $s = \frac{du}{d\theta}$, and performing integration by parts in the term $\lambda^T \frac{d}{dt}\frac{du}{d\theta}$ we derive

$$\frac{dL}{d\theta} = \int_{t_0}^{t_1} \left(\frac{\partial h}{\partial \theta} + \lambda^T \frac{\partial f}{\partial \theta}\right) dt - \int_{t_0}^{t_1} \left(-\frac{d\lambda^T}{dt} - \lambda^T \frac{\partial f}{\partial u} - \frac{\partial h}{\partial u}\right) s(t)\, dt - \left(\lambda(t_1)^T s(t_1) - \lambda(t_0)^T s(t_0)\right). \tag{34}$$

Now, we can force some of the terms in the last equation to be zero by solving the following adjoint differential equation for $\lambda(t)^T$ in backwards mode

$$\frac{d\lambda}{d\theta} = -\left(\frac{\partial f}{\partial u}\right)^T \lambda - \left(\frac{\partial h}{\partial u}\right)^T, \tag{35}$$

with final condition $\lambda(t_1) = 0$. Then, in order to compute the full gradient $\frac{dL}{d\theta}$ we do

1. Solve the original differential equation $\frac{du}{dt} = f(u, t, \theta)$;
2. Solve the backwards adjoint differential equation (35);
3. Compute the simplified version of the full gradient in Equation (34) as

$$\frac{dL}{d\theta} = \lambda^T(t_0) s(t_0) + \int_{t_0}^{t_1} \left(\frac{\partial h}{\partial \theta} + \lambda^T \frac{\partial f}{\partial \theta}\right) dt. \tag{36}$$

In order to solve the adjoint equation, we need to know $u(t)$ at any given time. There are different ways in which we can accomplish this: i) we can solve for $u(t)$ again backwards; ii) we can store $u(t)$ in memory during the forward step; or iii) we can do checkpointing to save some reference values in memory and interpolate in between. Computing the ODE backwards can be unstable and lead to exponential errors, [Kim+21]. In [Che+19], the solution is recalculated backwards together with the adjoint simulating an augmented dynamics:

$$\frac{d}{dt}\begin{bmatrix} u \\ \lambda \\ \frac{dL}{d\theta} \end{bmatrix} = \begin{bmatrix} -f \\ -\lambda^T \frac{\partial f}{\partial u} \\ -\lambda^T \frac{\partial f}{\partial \theta} \end{bmatrix} = -[1, \lambda^T, \lambda^T] \begin{bmatrix} f & \frac{\partial f}{\partial u} & \frac{\partial f}{\partial \theta} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}, \tag{37}$$

with initial condition $[u(t_1), \frac{\partial L}{\partial u(t_1)}, 0]$. One way of solving this system of equations that ensures stability is by using implicit methods. However, this requires cubic time in the total number of ordinary differential equations, leading to a total complexity of $\mathcal{O}((n+p)^3)$ for the adjoint method. Two alternatives are proposed in [Kim+21], the first one called *Quadrature Adjoint* produces a high order interpolation of the solution $u(t)$ as we move forward, then solve for λ backwards using an implicit solver and finally integrating $\frac{dL}{d\theta}$ in a forward step. This reduces the complexity to $\mathcal{O}(n^3+p)$, where the cubic cost in the number of ODEs comes from the fact that we still need to solve the original stiff differential equation in the forward step. A second but similar approach is to use a implicit-explicit (IMEX) solver, where we use the implicit part for the original equation and the explicit for the adjoint. This method also will have complexity $\mathcal{O}(n^3 + p)$.

4 Do we need full gradients?

5 Open Science from Scratch

6 Glossary

References

[Joh12] Steven G. Johnson. "Notes on Adjoint Methods for 18.335". In: 2012.

[Bay+15] Atilim Gunes Baydin et al. "Automatic differentiation in machine learning: a survey". In: *arXiv* (2015). DOI: 10.48550/arxiv.1502.05767.

[RLP16] J. Revels, M. Lubin, and T. Papamarkou. "Forward-Mode Automatic Differentiation in Julia". In: *arXiv:1607.07892 [cs.MS]* (2016). URL: https://arxiv.org/abs/1607.07892.

[Rud16] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *arXiv preprint arXiv:1609.04747* (2016).

[Bez+17] Jeff Bezanson et al. "Julia: A Fresh Approach to Numerical Computing". In: *SIAM Review* 59.1 (2017), pp. 65–98. ISSN: 0036-1445. DOI: 10.1137/141000671.

[RH17] James Ramsay and Giles Hooker. *Dynamic data analysis*. Springer, 2017.

[Ell18] Conal Elliott. "The simple essence of automatic differentiation". In: *Proceedings of the ACM on Programming Languages* 2.ICFP (2018), p. 70. DOI: 10.1145/3236765.

[Inn18] Michael Innes. "Don't Unroll Adjoint: Differentiating SSA-Form Programs". In: *arXiv* (2018).

[Che+19] Ricky T. Q. Chen et al. "Neural Ordinary Differential Equations". In: *arXiv:1806.07366 [cs, stat]* (Dec. 2019). arXiv: 1806.07366. URL: http://arxiv.org/abs/1806.07366 (visited on 02/25/2022).

[Lau19] Soeren Laue. *On the Equivalence of Forward Mode Automatic Differentiation and Symbolic Differentiation*. 2019. DOI: 10.48550/ARXIV.1904.02990. URL: https://arxiv.org/abs/1904.02990.

[RPK19] M. Raissi, P. Perdikaris, and G.E. Karniadakis. "Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations". In: *Journal of Computational Physics* 378 (2019), pp. 686–707. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2018.10.045.

[Wan+19] Fei Wang et al. "Backpropagation with Continuation Callbacks:Foundations for Efficient and ExpressiveDifferentiable Programming". In: *Proceedings of the ACM on Programming Languages* 3.ICFP (2019), p. 96. DOI: 10.1145/3341700.

[Rac+20] Christopher Rackauckas et al. "Universal differential equations for scientific machine learning". In: *arXiv preprint arXiv:2001.04385* (2020).

[Kim+21] Suyong Kim et al. "Stiff neural ordinary differential equations". en. In: *Chaos: An Interdisciplinary Journal of Nonlinear Science* 31.9 (Sept. 2021), p. 093122. ISSN: 1054-1500, 1089-7682. DOI: 10.1063/5.0060697. URL: https://aip.scitation.org/doi/10.1063/5.0060697 (visited on 02/25/2022).

[Ma+21] Yingbo Ma et al. "A Comparison of Automatic Differentiation and Continuous Sensitivity Analysis for Derivatives of Differential Equation Solutions". In: *arXiv:1812.01892 [cs]* (July 2021). arXiv: 1812.01892. URL: http://arxiv.org/abs/1812.01892 (visited on 02/25/2022).