Proceedings of Science and Mathematics
Faculty of Science, Universiti Teknologi Malaysia

Vol. 15, 2023, page 34-44

Study on Application of Automatic Differentiation

Desmond Lee Ching Yiing, Yeak Su Hoe*
Department of Mathematics, Faculty of Science, UTM, Skudai, Johor Bahru, Malaysia
*Corresponding author: s.h.yeak@utm.my

Abstract
This study examines the use of Automatic Differentiation (AD) in various applications, such as machine learning and numerical methods. It explores problems and applications related to AD, methods for enhancing differentiation, and mathematical implementations. PyCharm, a Python programming tool, is utilized to facilitate the mathematical implementation process. Dual numbers, implemented through classes and methods, are used as a method of AD, allowing for simultaneous evaluation of functions and their derivatives. Forward mode differentiation is employed to construct the Jacobian matrix of functions with inputs and outputs, while reverse mode differentiation, also known as backpropagation, is used for computing gradients in deep learning problems with numerous input variables. The study showcases the effectiveness, efficiency, and simplicity of AD, including the representation of higher order derivatives through recursive computations. It discusses implicit deep-learning models constructed using iterative techniques like fixed-point iterations and compares them to Newton's method. Other topics covered include AD functions using numerical fixed points, computing derivatives using the Implicit Function Theorem, and the transformation of AD using Jacobian-vector products (JVPs) and vector-Jacobian products (VJPs). The study also explores Neural Ordinary Differential Equations (ODEs) and reverse-mode differentiation across ODE solvers. Additionally, it provides an overview of the complexity of AD operations and compares the runtime of symbolic differentiation with AD. Overall, AD proves to be a powerful tool for computing gradients in parameter optimization and sensitivity analysis across a wide range of applications. Future research should focus on exploring new applications using existing libraries to fully harness the potential of AD in deep learning and other fields.
Keywords: Automatic Differentiation; numerical techniques; Python; sensitivity analysis; parameter optimization

Introduction

This study focuses on the practical application of AD in various domains, specifically in deep learning and optimization. AD is a method widely used in machine learning frameworks like TensorFlow [1] and PyTorch [15] to compute derivatives of numerical functions. However, despite its name, AD is not completely automated and can lead to inefficient code if not used carefully. To address this, some refer to it as algorithmic differentiation. This study aims to highlight the principles, techniques, and mathematical implementation of AD, discussing its importance and efficiency in deep learning tasks.

The study covers a range of topics, including dual numbers, forward mode differentiation, and reverse mode differentiation, with a focus on numerical approaches and solutions. It provides a Python program that demonstrates the functionality of these algorithms. Additionally, the study explores the relevance of AD in machine learning and identifies applicable scenarios.

The significance of this study lies in its exploration of accurate AD implementation strategies that maintain mathematical precision. It also emphasizes the importance of readable and simple code implementation, crucial for practical applications. The study acknowledges that compiling time is not a major concern since AD programs are typically constructed once and used repeatedly for various computations in statistical applications.

Dual Numbers

One of AD approaches involves the use of dual numbers [2]. Assume the function g be sufficiently

smooth that it is differentiable sufficient number of times, and ϵ be a tiny infinitesimal parameter. ϵ^2 becomes negligible, resulting in $g(x+\epsilon) = g(x) + \epsilon g'(x) + \epsilon^2 g''(x) + \cdots$. Assume an augmented object $x+\epsilon$ be in the calculation of g, the gradients can be extracted from the coefficient of the ϵ term. The operators for dual numbers are defined as:

$$(x + a\epsilon) + (y + b\epsilon) = (x + y) + (a + b)\epsilon \text{ (addition)},$$

$$(x + a\epsilon)(y + b\epsilon) = xy + (xb + ya)\epsilon \text{ (multiplication)},$$

$$\frac{x + a\epsilon}{y + b\epsilon} = \frac{x}{y} + \epsilon\left(\frac{a}{y} - \frac{bx}{y^2}\right) \text{ (division)},$$

$$e^{x+a\epsilon} = e^x + a\epsilon e^x \text{ (exponential Taylor expansions)},$$

$$\sin(x + a\epsilon) = \sin(x) + a\epsilon\cos(x) \text{ (trigonometric functions)},$$

$$(x + a\epsilon)^{\frac{1}{2}} = \sqrt{x} + \frac{a\epsilon}{2\sqrt{x}} \text{ (square root)}.$$

where $\epsilon^2 = 0$ is the fundamental property guiding these arithmetic operations. The square root can be calculated using the Newton formula rather than a polynomial approximation. Using the Newton formula, the fixed-point iteration of the function for finding the square root can be expressed as $x_{i+1} = \frac{1}{2}\left(x_i + \frac{y}{x_i}\right),\ i = 0,1,2,\cdots,n$.

Forward Mode Differentiation

Each node, denoted as v, corresponds to an intermediate computation in this function [2]. Figure 1 illustrates the computational graph of the function $g(x_1, x_2) = x_1x_2 - \cos(x_2)$ via forward mode.

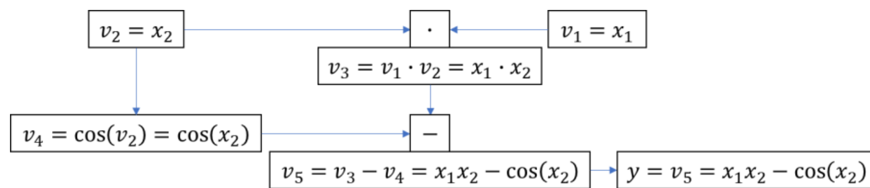

Figure 1 Computational graph of function $g(x_1, x_2) = x_1x_2 - \cos(x_2)$ via forward mode

In the view of Figure 1, assume the function g be given by $y = g(x_1, x_2) = x_1x_2 - \cos(x_2)$, the derivative g' is evaluated at $x_1 = 2$ and $x_2 = 3$. Forward mode AD only requires tracking of the intermediary computations at each node. The following calculations are $v_1 = 2$, $v_2 = 3$, $v_3 = 6$, $v_4 = \cos(3)$, and $v_5 \approx 6.99$. Using the dual trace and differentiating each equation, v with respect to x_2, the expressions are: $\dot{v_1} = 0$, $\dot{v_2} = 1$ $\dot{v_3} = 2$, $\dot{v_4} = -\sin(3)$, and $\dot{v_5} \approx 2.14$. Thus, a derivative is assigned to each intermediate variable v_i to determine $\dot{v_i} = \frac{\partial v_i}{\partial x_2}$.

It is highlighted that the vector-valued functions g, represented as $g: \mathbb{R}^n \to \mathbb{R}^m$ can have multiple inputs m and outputs n [2]. The Jacobian matrix, denoted as J_g, captures the gradient relationships between the outputs m and inputs n as follows:

$$J_g = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix},$$

where y_m represents the jth output and x_n represents the ith input. In addition, forward mode AD offers a matrix-free approach for calculating Jacobian-vector products (JVPs) as follows:

$$J_g \cdot v = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_n \end{bmatrix}.$$

By initializing $\dot{x} = v$, JVPs can be directly computed in a single forward pass. If the function g maps from $\mathbb{R}^n$ to $\mathbb{R}$, the directional derivative along a given vector v can be obtained by taking a linear combination of the partial derivatives $\nabla g.v$, where the gradient $\nabla g = \left[\frac{\partial y}{\partial x_1}, \cdots, \frac{\partial y}{\partial x_n}\right]$, corresponding to a $1 \times n$ Jacobian matrix that is constructed column by column using forward mode in n evaluations.

Reverse Mode Differentiation

The reverse mode differentiation is needed in cases where the number of input variables n is much larger than the number of outputs m, which is often observed in deep learning tasks [2]. Reverse mode differentiation, also known as backpropagation, is the preferred approach for computing gradients. Assume the same function g be $y = g(x_1, x_2) = x_1 x_2 - \cos(x_2)$. In primal trace, the values of the variable are unchanged.

The dual trace in reverse mode AD is more complex compared to the forward mode. This complexity can be confusing initially because we are more familiar with the step-by-step application of the chain rule to propagate derivatives forward [9]. This confusion is suggested to arises partly due to the familiarity with the chain rule as a step-by-step process for propagating derivatives forward [12]. The chain rule can be expressed as $\frac{\partial y_j}{\partial v_i} = \frac{\partial y_j}{\partial v_k} \cdot \frac{\partial v_k}{\partial v_i}$. However, the choice of v_k is not arbitrary. v_k would be the parent of v_i. If v_k has multiple parents, the chain rule is summed up for each parent using multivariable chain rule, yielding $\frac{\partial y_j}{\partial v_i} = \sum_{p \epsilon parents(i)} \frac{\partial y_j}{\partial v_p} \cdot \frac{\partial v_p}{\partial v_i}$. In this approach, each intermediate variable v_i is accompanied by an adjoint value, denoted as $\bar{v}_i = \frac{\partial y_j}{\partial v_i}$, representing the sensitively of a specific output y_j with respect to the changes in the corresponding variables v_i, and it can be expressed in terms of the adjoint of parents as $\bar{v}_i = \sum_p \bar{v}_p \cdot \frac{\partial y_j}{\partial v_i}$.

Figure 2 illustrates the computational graph of the function $g(x_1, x_2) = x_1 x_2 - \cos(x_2)$ via reverse mode.

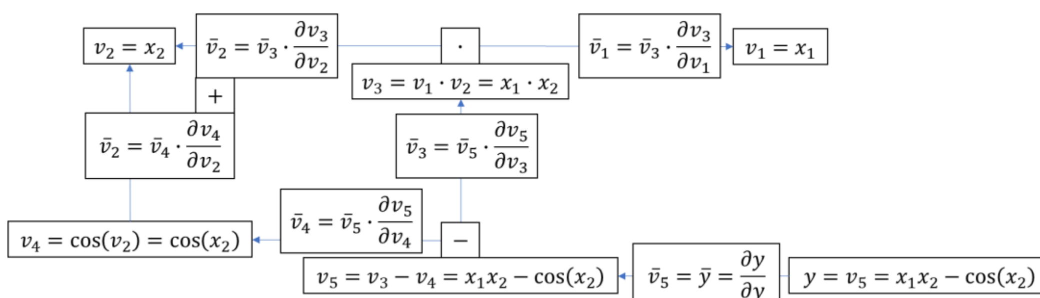

Figure 2 Computational graph of function $g(x_1, x_2) = x_1 x_2 - \cos(x_2)$ via reverse mode

In the view of Figure 2, this recursive method begins at the output node y and traverse adjoints back to the input nodes. In addition, v_1 and v_2 have v_3 as a parent, the following calculations are $\bar{v}_5 = 1$, $\bar{v}_4 = -1$, $\bar{v}_3 = 1$, $\bar{v}_2 = 2 - (-\cos(3)) \approx 2.14$, and $\bar{v}_1 = 3$. The reverse mode AD computes both $\frac{\partial y}{\partial x_1}$ and $\frac{\partial y}{\partial x_2}$ in the complete iteration.

In the extreme case of a function $g: \mathbb{R}^n \rightarrow \mathbb{R}$, reverse mode only needs to be applied once to compute the entire gradient $\nabla g = \left[\frac{\partial y}{\partial x_1}, \cdots, \frac{\partial y}{\partial x_n}\right]$, while forward mode would require n passes to achieve the same result. Similar to the matrix-free computation of JVPs using forward mode, the transposed JVPs can be evaluated by initializing the reverse pass with $\bar{y} = v$ as follows:

$$J_g^T \cdot v = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \begin{bmatrix} v_1 \\ \vdots \\ v_m \end{bmatrix}.$$

Higher Order Derivatives

In this study, the notation used to represent higher order derivatives often employs a more compact in Automatic Differentiation (AD) system supported by the previous study [3][4]. Generally, the nth order derivative is formed by composing of the derivative $n - 1$ times with itself, evaluated at $g(x)$. In order to highlight the nth order derivative, parentheses are used to indicate that it is composition of the functions $\frac{d^n}{dx^n} g(x)$.

Computation of Higher Order Derivatives via Forward Mode

Forward mode AD can be used to compute higher order derivatives. Figure 3 illustrates the computational graph of the function $g(x) = \sin(x^3)$ using higher order derivatives via forward mode.

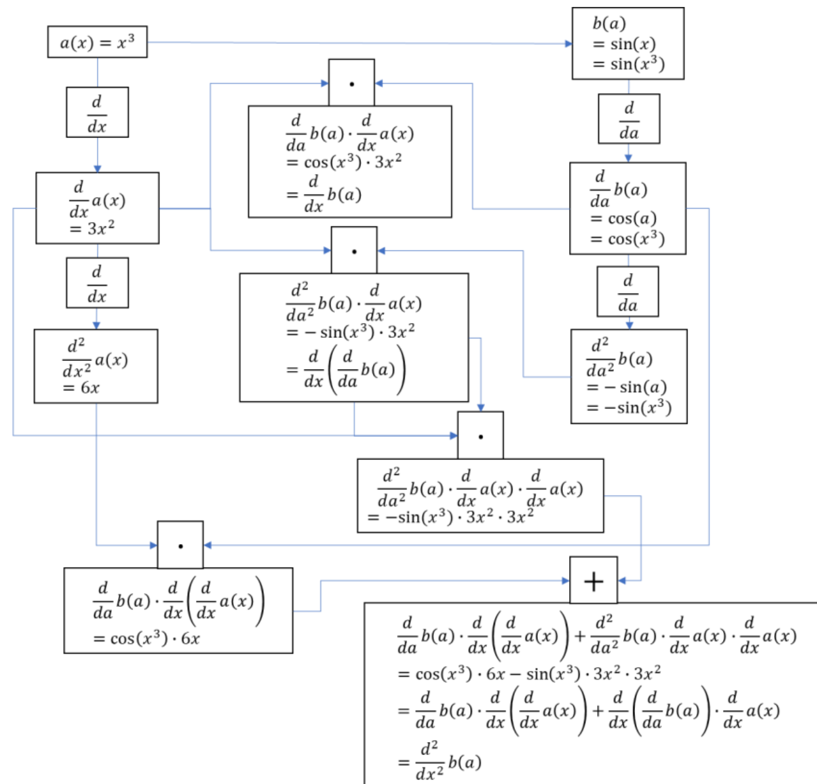

Figure 3 Computational graph of function $g(x) = \sin(x^3)$ using higher order derivatives via forward mode

In the view of Figure 3, assume the computation flows on the graph of the function $g(x) = \sin(x^3)$ be calculated using higher order derivatives. x is denoted as input variable, $a(x) = x^3$ is the intermediate value or node, and $b(a)$ is another intermediate value or node obtained from $\sin(a)$. At node a, the first derivative of function $a(x)$ with respect to x is calculated using power rule, yielding $\frac{d}{dx}a(x) = 3x^2$. The derivative of first derivative is respect to x is multiplied by the first derivative itself, resulting in $\frac{d^2}{dx^2}a(x) = 6x$. Moving to node b, the derivatives with respect to x is computed again. The derivative function for $b(a)$ with respect x is expressed as $\frac{d}{da}b(a) \cdot \frac{d}{dx}a(x)$ by applying the chain rule. The derivative of $b(a)$ with respect to a is calculated, yielding $\frac{d}{da}b(a) = \cos(x^3)$. Substituting the expressions for $\frac{d}{da}b(a)$ and $\frac{d}{dx}a(x)$ in $\frac{d}{dx}b(a)$, we obtain $\frac{d}{dx}b(a) = \cos(x^3) \cdot 3x^2$.

The second derivative of $b(a)$ with respect to x is expressed as $\frac{d}{da}b(a) \cdot \frac{d}{dx}\left(\frac{d}{dx}a(x)\right) + \frac{d}{dx}\left(\frac{d}{da}b(a)\right) \cdot \frac{d}{dx}a(x)$ by applying the product rule. The derivative $\frac{d}{dx}\left(\frac{d}{dx}a(x)\right)$ is calculated as $\frac{d^2}{da^2}b(a) \cdot \frac{d}{dx}a(x)$ using chain rule, where we obtain $-\sin(x^2)\, 3x^2$. By substituting the calculated expressions for $\frac{d}{da}b(a)$, $\frac{d}{dx}a(x)$, $\frac{d}{dx}\left(\frac{d}{da}b(a)\right)$ and $\frac{d^2}{da^2}b(a)$ into $\frac{d^2}{dx^2}b(a)$, we obtain the final expression for the second derivative as $\frac{d^2}{dx^2}b(a) = \cos(x^3) \cdot 6x - \sin(x^3) \cdot 3x^2 \cdot 3x^2$.

Implicit Models

Lower-precision math and iterative methods like fixed-point iterations can increase the efficiency of deep neural networks. Deep Equilibrium Models, which rely on repetitive processes to arrive at an equilibrium state, allow for more intricate representations. Implicit models provide benefits including innovative designs, robustness analysis, and interpretability while also making deep learning notation simpler [10].

Fixed-Point Iteration

The output of a neural network layer is determined using the fixed-point iteration approach, where the output y is computed repeatedly as $y = \tanh(\theta y + x)$, starting with an initial guess of $y = 0$. Until convergence, this iterative method updates y_{i+1} depending on y_i and the network parameters θ. This layer may be thought of as a straightforward recurrent network, with the hidden layer y operating on the input x. Despite having only one layer parameterized by θ, it benefits from the recurrent application of the nonlinearity offered by the tanh activation function while having just one layer parameterized by θ. This layer can be mathematically represented as an implicit layer, where y_{i+1} stands for the solution to a root-finding equation. A fixed-point equivalent form of the equation is created, with $y = g(x, y) = \tanh(\theta y + x)$. Starting with an initial guess y_0, the iteration process builds a sequence, updating y_{i+1} depending on y_i using the function g. The iteration keeps on going until the condition $|y_{i+1} - y_i| < \epsilon$, which ϵ is a tiny convergence tolerance, is met.

In some cases, the values of y might fluctuate without reaching a stable solution, and the behavior relies on the value of θ. However, the loop comes to an end at $y_{i+1} = \tanh(x)$, reaching a fixed point when θ is zero. The use of fixed-point iteration and implicit models is supported by the previous study in deep learning [13] to emphasize the importance of regularization techniques in solving inverse problems and the integration of data-driven regularization and convex feasibility.

Alternative Methods for Finding Root Using Newton's Method

The fixed-point iteration approach is contrasted with Newton's method, highlighting the latter's faster convergence in a recent study [7]. Newton's method is a significantly faster approach for finding the solution to the function $g(y) = \tanh(\theta y + x)$. Newton's method continues the update until it finds a root $g(y) = 0$ for the function $g: \mathbb{R}^n \to \mathbb{R}^n$. The updated equation for Newton's Method's is expressed as $y_{i+1} = y_i - \left(\frac{\partial g_i}{\partial y_i}\right)^{-1} g(y_i),\ i = 0,1,2,\cdots,n,$ where $\frac{\partial g_i}{\partial y_i} = I - diag\left(\operatorname{sech}^2\left(\theta_{r_i} y_i + x\right)\right)\theta_i$, which is Jacobian of g with respect to y.

However, it is much complex compared to the earlier fixed-point iteration technique. Although fewer iterations are required than in fixed-point iteration, each iteration takes considerably longer because a unique Jacobian matrix must be formed and inverted for each sample in the minibatch. It quickly becomes impractical to invert or even store these matrices as the hidden unit sizes grow, especially in convolutional networks. The recent study supports this study by providing an overview of implicit models, explaining the fixed-point iteration method, introducing an alternative method such as Newton's method, and discussing the challenges [6].

Efficient Differentiation of Fixed Points

As shown in Deep Equilibrium models, numerical fixed points provide a technique to automate the differentiation of functions. Differentiation may be mechanized by solving fixed-point equations, which equal a function and its derivative. Deep equilibrium models iteratively update the input until convergence and use fixed points to determine the equilibrium state of a system. This method is beneficial for automating differentiation in numerous contexts, notably in the context of Deep Equilibrium models, since it does away with the necessity for human derivative computations.

Implicit Function Theorem

The derivative of a function $g: \mathbb{R}^n \to \mathbb{R}^m$ at a point $x \in \mathbb{R}^n$ is denoted as $\frac{\partial g}{\partial x}: \mathbb{R}^n \to \mathbb{R}^m$, where $\frac{\partial g}{\partial x}$ is a linear function. This derivative function maps perturbations in the input space $\mathbb{R}^n$ to perturbations in the output space $\mathbb{R}^m$. The first-order Talyor series expansion of g at point x can be expressed as $g(x + v) = g(x) + \frac{\partial g}{\partial x} \cdot v + O(\|v\|^2)$, where v is perturbation vector in $\mathbb{R}^n$ and $O(\|v\|^2)$ represents the asymptotic upper bound on the error or approximation.

When a system of nonlinear equations in terms of y is parameterized by x, denoted as $g(x,y)=0$, the nominal solution is represented by the point (x_0,y_0). Jacobian of the solution mapping can be evaluated at the point (x_0,y_0) using the derivative of g. By differentiating both sides of the equation with respect to x and rearranging the equation, the Jacobian of the solution mapping can be expressed as $\frac{\partial y_0}{\partial x_0} = -\left(\frac{\partial g_0}{\partial y_0}\right)^{-1}\frac{\partial g_0}{\partial x_0}$.

The proposed study addresses this issue by introducing automatic implicit differentiation, an efficient and modular approach that leverages AD and the implicit function theorem to differentiate optimization problems [5]. By allowing the user to define an optimality function directly in Python, the study enables the seamless integration of implicit differentiation with state-of-the-art solvers.

Transformation Used in Automatic Differentiation

Jacobian-vector products (JVPs) and vector-Jacobian products (VJPs) are the transformations used to facilitate AD of fixed-point solvers and implicit functions. In JVPs, the Jacobian matrix $\frac{\partial g}{\partial x}$ of function $g(x)\colon \mathbb{R}^n \to \mathbb{R}^m$ can be computed. A mapping is defined as $(x,v) \to \left(g(x), \frac{\partial g}{\partial x}\cdot v\right)$, where parameters $x \in \mathbb{R}^n$ and $v \in \mathbb{R}^n$ is the right multiplication vector in JVPs. Similar to VJPs, it is defined as the mapping $(x,w) \to \left(g(x), w^T\frac{\partial g}{\partial x}\right)$, where $w \in \mathbb{R}^m$ is a dummy vector, such as a collection of the vector. Backpropagation is frequently used in reverse mode AD packages to compute the Jacobian matrix's left multiplication in VJPs.

Since the relationship between VJPs and the gradient of a scalar-valued function and the significance of gradient-based optimization, JVPs and VJPs facilitate function composition, which is a critical component of AD. The JVP of g can be expressed by composing the JVPs of f and h when two functions, f and h, are composed to form $g = f \circ h$. Specifically, JVPs of f can be defined as the mapping $(x,v) \to (f(x), \frac{\partial f}{\partial x}\cdot v)$, and the JVPs of h can be defined as the mapping $(y,u) \to (h(y), \frac{\partial h}{\partial y}\cdot u)$, then the JVPs of g can be derived as $g(x) + \left(\frac{\partial f}{\partial h}\cdot\frac{\partial h}{\partial y}\right)u$. Similar to how the VJPs of f and h for the same composition $g = f \circ h$ can be composed to get VJPs of g, indicated a ∇g. If the VJPs of f can be defined as the mapping of $(x,w) \to \left(f(x), w^T\frac{\partial f}{\partial x}\right)$, and the VJPs of h can be defined as the mapping $(y,w) \to \left(h(y), w^T\frac{\partial h}{\partial y}\right)$, then the VJP of g can be determined as $w^T\frac{\partial f}{\partial h} + \left(w^T\frac{\partial f}{\partial h}\right)\frac{\partial h}{\partial y}$.

A past study proposed a decomposition of reverse-mode AD into linearization followed by transposition [11]. This approach highlights the key distinction between forward-mode and reverse-mode AD and simplifies their joint implementation.

Fixed Points of Transformation Used in Automatic Differentiation

Both JVPs and VJPs play crucial roles in AD, enabling the computation of gradients and the linkage between Jacobians of functions and solution mappings. It involves finding a fixed-point solution mapping y that satisfies the fixed-point equation for any given parameter $x \in \mathbb{R}^n$, represented as $y(x) = g\big(x, y(x)\big)$. At a certain point x_0 with $y_0 = y(x_0)$, the derivative can be expressed as $\frac{\partial y_0}{\partial x_0} = \left(I - \frac{\partial g_0}{\partial y_0}\right)^{-1}\frac{\partial g_0}{\partial x_0}$, where $\frac{\partial g_0}{\partial y_0}$ and $\frac{\partial g_0}{\partial x_0}$ represent the Jacobians of the function g evaluated at that point.

This relationship connects the Jacobians of the function g with the Jacobians of the solution mapping y at that particular point. For the JVPs, the mapping $(x_0,v) \to \left(g_0, \frac{\partial g_0}{\partial x_0}\cdot v\right)$ can be computed, yielding $\frac{\partial y_0}{\partial x_0}\cdot v = \left(I - \frac{\partial g_0}{\partial y_0}\right)^{-1}\frac{\partial g_0}{\partial x_0}\cdot v$. For VJPs, the mapping $(x,w) \to \left(g(x), w^T\frac{\partial g}{\partial x}\right)$ is computed, yielding $w^T\frac{\partial y_0}{\partial x_0} = w^T\left(I - \frac{\partial g_0}{\partial y_0}\right)^{-1}\frac{\partial g_0}{\partial x_0}$.

Neural Ordinary Differential Equations

The primary challenge in training continuous-depth networks lies in conducting reverse-mode differentiation, also called backpropagation, across the Ordinary Differential Equation (ODE) solver. It scales linearly with the problem size, requires minimal memory, and handles numerical inaccuracies efficiently [8].

The optimization of a loss function, which is a scalar value is the input to the ODE solver, denoted as:

$$L(y(t)) = L\left(y(t_0) + \int_{t_0}^{t} g(y, t, \theta)dt\right),$$

where y is a vector and θ is the parameter for the function. In order to compute the gradients with respect to θ to optimize L, The adjoint is denoted as $a(t) = \frac{\partial L}{\partial y(t)}$ is to find the connection between each moment of the hidden state $y(t)$ and the gradient of the loss. Instead of directly calculating the partial derivatives, we determine the change in the partial derivative $\frac{\partial a(t)}{\partial t}$ by transitioning from the discrete case to the continuous case to calculate that $\frac{\partial a(t)}{\partial t} = -a(t)\frac{\partial g(y(t),t,\theta)}{\partial y(t)}$.

An augmented ODE is derived to calculate the gradients of loss function with respect to t and θ as follow:

$$\frac{d}{dt}\begin{bmatrix} y \\ \theta \\ t \end{bmatrix}(t) = g_{aug}([y, \theta, t]) = \begin{bmatrix} g([y, \theta, t]) \\ 0 \\ 1 \end{bmatrix}.$$

This augmented state's adjoint state associated is described as:

$$a_{aug} = \begin{bmatrix} a_y \\ a_\theta \\ a_t \end{bmatrix}, \quad a_y = \frac{\partial L}{\partial y(t)}, \quad a_\theta = \frac{\partial L}{\partial \theta(t)}, \quad a_t = \frac{\partial L}{\partial t(t)}.$$

The gradients of the augmented state are as follows:

$$\frac{\partial g_{aug}([y, \theta, t])}{\partial [y(t) \quad \theta(t) \quad t(t)]} = \begin{bmatrix} \frac{\partial g}{\partial y(t)} & \frac{\partial g}{\partial \theta(t)} & \frac{\partial g}{\partial t(t)} \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.$$

The adjoint state that holds to all variables in the differential equations is expressed as follows:

$$\frac{da(t)}{dt} = \frac{d}{dt}[a_y(t) \quad a_\theta(t) \quad a_t(t)] = -\left[a_y(t)\frac{\partial g}{\partial y(t)} \quad a_\theta(t)\frac{\partial g}{\partial \theta(t)} \quad a_t(t)\frac{\partial g}{\partial t(t)}\right].$$

Solving initial value problem of this adjoint augmented ODE, the gradients of the loss function with respect to y, θ, and t can be obtained as follows:

$$\frac{\partial L}{\partial y(t_0)} = a_y(t_1) + \int_{t_1}^{t_0} a_y(t)\frac{\partial g}{\partial y(t)}dt,$$

$$\frac{\partial L}{\partial \theta(t_0)} = \int_{t_1}^{t_0} a_\theta(t)\frac{\partial g}{\partial \theta(t)}dt,$$

$$\frac{\partial L}{\partial t(t_0)} = a_t(t_1) + \int_{t_1}^{t_0} a_t(t) \frac{\partial g}{\partial t(t)} dt.$$

Runtime of Automatic Differentiation

The mathematical difficulty, network size, and implementation efficiency all affect how quickly AD runs while computing numerical derivatives. There are graph computations, forward and reverse mode differentiation, and optimization techniques used, with the length of passes depending on the complexity of the operation and the size of the graph.

Runtime of Operations and their Complexity

TensorFlow was used to examine the runtime of addition and square root operations in AD. By taking into account the difficulty of the operations and the quantity of arithmetic operations involved, it is possible to mathematically explain the observed disparity, where the square root operation looked quicker than addition. When compared to addition, the square root operation often has a lower complexity, which may be represented as $O(g(x))$ since it just calls for simple mathematical operations and is unaffected by the size of the integers involved. Contrarily, addition is written as $O(1)$ because, regardless of the numbers being added, it just calls for a simple arithmetic operation. As a result, calculating the square root operation within a loop frequently requires less time than adding, resulting in a reduced runtime.

Derivatives in computational statistics play a key part in this investigation of the runtime differences between addition and square root operations in AD [14]. It highlights the simplicity of mathematical procedures and their independence from the magnitude of numbers in square root computations, providing a mathematical justification for why the square root operation is typically computationally cheaper than addition.

Comparison of Runtime Between Symbolic Differentiation and Automatic Differentiation

The derivative of the function $y = x_1^2 + x_1x_2$ is calculated in order to compare the runtimes of symbolic differentiation with AD. In symbolic differentiation, the derivative is obtained by differentiating each term independently and then using the product rule. The derivative is calculated as 11 when the values of final statement is $x_1 = 4$ and $x_2 = 3$.

On the other hand, the derivative is computed numerically using AD, more especially reverse mode AD. The derivative is determined by evaluating the derivatives at particular nodes in the computational graph of the function, yielding $\frac{\partial y}{\partial x_1} = \bar{a} = \bar{c}\frac{\partial c}{\partial a} + \bar{d}\frac{\partial d}{\partial a}$ in Figure 4. When the given values are substituted, the result, which was achieved without symbolic manipulation, similarly produces a derivative of solution, which is 11. Figure 4 illustrates the computational graph of function $y = x_1^2 + x_1x_2$ via reverse mode differentiation at $x_1 = 4$ and $x_2 = 3$.

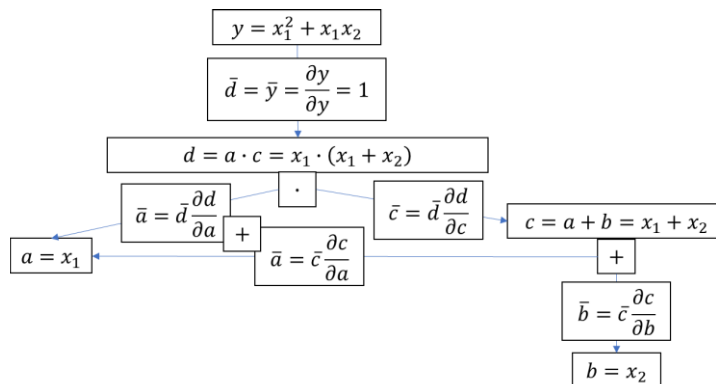

Figure 4 Computational graph of function $y = x_1^2 + x_1x_2$ via reverse mode

For computing derivatives, AD is quicker than symbolic differentiation in the code implementation. It can be computationally expensive to manipulate and simplify algebraic equations symbolically for symbolic differentiation, especially for complicated expressions. On the other hand, AD does not require symbolic manipulation because it directly evaluates derivatives at certain nodes in the computation network.

These findings are supported by the previous study, which highlights the advantages and disadvantages of symbolic differentiation and AD [14]. It clarifies runtime variations and emphasizes the value of thorough implementation and optimization in AD. These studies address the difficulties and trade-offs associated with AD while advancing our understanding of efficient differentiation techniques.

Conclusion

This study on AD contributes significantly to the knowledge in this field by exploring various aspects of AD and its techniques. It highlights the use of dual numbers for simultaneous computation of functions and their derivatives, provides implementation guidance for incorporating dual numbers into AD frameworks, and discusses the forward mode differentiation for calculation. The study extensively examines reverse mode differentiation, particularly in deep learning, and offers insights into its computational benefits. It also addresses the computation of higher order derivatives, the use of implicit models and efficient differentiation of fixed points, ODE in optimization, and the runtime characteristics of AD. Future work can focus on improving AD efficiency, extending AD to higher order derivatives, exploring the applications in deep learning, investigating alternative differentiation techniques, and conducting real-world experiments to validate the proposed approaches.

Acknowledgement

I want to sincerely appreciate those who assisted me throughout the course of the project.

References

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., ... & Zheng, X. (2016). TensorFlow: a system for large-scale machine learning. In 12th USENIX symposium on operating systems design and implementation (OSDI 16) (pp. 265-283).

[2] Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2018). Automatic differentiation in machine learning: a survey. Journal of Marchine Learning Research, 18, 1-43.

[3] Bettencourt, J., Johnson, M. J., & Duvenaud, D. (2019). Taylor-mode automatic differentiation for higher-order derivatives in JAX. In Program Transformations for ML Workshop at NeurIPS 2019.

[4] Bischof, C., Corliss, G., & Griewank, A. (1993). Structured second-and higher-order derivatives through univariate Taylor series. Optimization Methods and Software, 2(3-4), 211-232.

[5] Blondel, M., Berthet, Q., Cuturi, M., Frostig, R., Hoyer, S., Llinares-López, F., ... & Vert, J. P. (2022). Efficient and modular implicit differentiation. Advances in neural information processing systems, 35, 5230-5242.

[6] Bogaers, A. E., Kok, S., Reddy, B. D., & Franz, T. (2014). Quasi-Newton methods for implicit black-box FSI coupling. Computer Methods in Applied Mechanics and Engineering, 279, 113-132.

[7] Campbell, S. L., & Hollenbeck, R. (1996). Automatic differentiation and implicit differential equations. Computational Differentiation: Techniques, Applications, and Tools, 215-227.

[8] Chen, R. T., Rubanova, Y., Bettencourt, J., & Duvenaud, D. K. (2018). Neural ordinary differential equations. Advances in neural information processing systems, 31.

[9] Dennis Jr, J. E., & Schnabel, R. B. (1996). Numerical methods for unconstrained optimization and nonlinear equations. Society for Industrial and Applied Mathematics.

[10] El Ghaoui, L., Gu, F., Travacca, B., Askari, A., & Tsai, A. (2021). Implicit deep learning. SIAM Journal on

Mathematics of Data Science, 3(3), 930-958.

[11] Frostig, R., Johnson, M. J., Maclaurin, D., Paszke, A., & Radul, A. (2021). Decomposing reverse-mode automatic differentiation. arXiv preprint arXiv:2105.09469.

[12] Griewank, A., & Walther, A. (2008). Evaluating Derivatives. SIAM.

[13] Heaton, H., Wu Fung, S., Gibali, A., & Yin, W. (2021). Feasibility-based fixed point networks. Fixed Point Theory and Algorithms for Sciences and Engineering, 2021(1), 1-19.

[14] Margossian, C. C. (2019). A review of automatic differentiation and its efficient implementation. Wiley interdisciplinary reviews: data mining and knowledge discovery, 9(4), e1305.

[15] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L. and Lerer, A. (2017). Automatic differentiation in pytorch.