



Research Article

Automatic Differentiation in Modern Machine Learning: Mathematical Foundations, Algorithmic Modes, Implementation Strategies, and Contemporary Challenges: A Comprehensive Review Article

Mohammad Qasam Adil, Mirwais Ghafare

Department of Mathematics, Faculty of Education, Helmand University, Lashkargah,
Helmand, Afghanistan

Corresponding Author, Email; university.info82@gmail.com

Copyright © 2026 by Authors, Published by **Interkoneksi: Journal of Computer Science and Digital Business**. This is an open access article under the CC BY License <https://creativecommons.org/licenses/by/4.0/>

Received : July 17, 2026

Revised : August 15, 2026

Accepted : September 13, 2026

Available online : October 25, 2026

How to Cite: Mohammad Qasam Adil, & Mirwais Ghafare. (2026). Automatic Differentiation in Modern Machine Learning: Mathematical Foundations, Algorithmic Modes, Implementation Strategies, and Contemporary Challenges: A Comprehensive Review Article. *Interkoneksi: Journal of Computer Science and Digital Business*, 4(2), 231–245. <https://doi.org/10.61166/interkoneksi.v4i2.99>

Abstract. Automatic differentiation (AD) is the algorithmic backbone of nearly every gradient-based method used in contemporary machine learning, ranging from stochastic gradient descent for deep neural networks and Adam-style adaptive optimizers, through gradient-based Markov chain Monte Carlo and variational inference in probabilistic programs, to end-to-end differentiable simulation in scientific machine learning. AD evaluates derivatives of numerical functions defined by computer programs to floating-point precision, and, in its reverse mode, at a computational cost that is bounded by a small constant multiple of the cost of evaluating the function itself, independently of the number of inputs. This review synthesizes the theoretical foundations, algorithmic modes, and implementation

strategies of AD; positions AD unambiguously against the two techniques with which it is most frequently confused — namely, numerical (finite-difference) differentiation, which is cheap but suffers from truncation and round-off error, and symbolic differentiation, which is exact but subject to expression swell for programs with control flow; and formalises the chain rule as the unifying mechanism that both the forward and reverse accumulation modes exploit. The complexity trade-offs between the two modes are analysed with reference to the shape of the Jacobian to be evaluated, and their combination for Hessian-vector products, mixed-mode, and higher-order derivatives is discussed. The historical convergence of reverse-mode AD with the backpropagation algorithm is traced from its independent origins in numerical rounding-error analysis (Linnainmaa, 1970), control theory (Werbos, 1974), general-purpose gradient compilation (Speelpenning, 1980), and connectionist learning (Rumelhart et al., 1986). The article further surveys the three principal implementation strategies — operator overloading, source-to-source transformation, and tracing with just-in-time compilation — as realised in the frameworks that dominate current practice, TensorFlow (Abadi et al., 2016), PyTorch (Paszke et al., 2019), and JAX (Bradbury et al., 2018). It closes with a discussion of open problems: correctness at points of non-differentiability, memory-efficient reverse mode through checkpointing, sound differentiation of stochastic and probabilistic programs, and formal verification of AD in compiler-embedded settings. The article integrates and updates prior treatments (Griewank & Walther, 2008; Baydin et al., 2018; Margossian, 2019) and reports on 2022–2024 developments in Enzyme, JAX, and differentiable programming for differential equations (Hückelheim et al., 2023; Sapienza et al., 2024).

Keywords: automatic differentiation; algorithmic differentiation; backpropagation; computational graph; forward mode; reverse mode; dual numbers; adjoint method; Jacobian-vector product; vector-Jacobian product; checkpointing; differentiable programming; probabilistic programming; JAX; PyTorch; TensorFlow; Enzyme.

INTRODUCTION

Every training loop in supervised deep learning, every step of a Hamiltonian Monte Carlo sampler, every iteration of a physics-informed neural solver, and every backward pass in a differentiable simulator relies on the same primitive operation: the evaluation of the derivative of a scalar- or vector-valued function with respect to a potentially very large collection of inputs. As model sizes have grown from thousands of parameters in early multilayer perceptrons to the hundreds of billions of parameters typical of modern transformer architectures, the machinery by which these derivatives are obtained has become as consequential to the discipline as the models themselves (Baydin et al., 2018; Griewank & Walther, 2008). Automatic differentiation (AD) is the family of techniques that has made this computation tractable, and its deep integration into the software stacks of TensorFlow (Abadi et al., 2016), PyTorch (Paszke et al., 2019), and JAX (Bradbury et al., 2018) has rendered it, in day-to-day practice, invisible to most users even as it underlies essentially everything they do.

The term automatic differentiation is used loosely in the machine learning literature and is frequently conflated either with the neural-network training procedure known as backpropagation, or with symbolic computer-algebra systems that manipulate mathematical expressions rather than execute programs. Baydin et al. (2018) argue that this terminological confusion has substantive consequences: a researcher who believes AD to be merely “backpropagation renamed” or a “variant of

symbolic differentiation” will systematically misjudge which operations are cheap to differentiate and which are prohibitively expensive, and will design algorithms and software accordingly. Clarifying these boundaries is therefore not a definitional exercise; it directly affects algorithm design, framework architecture, and the correctness of the derivatives ultimately used to update model parameters.

The present article has four aims. First (Section 2), it distinguishes AD from numerical and symbolic differentiation and identifies the chain-rule-based mechanism common to both of the accumulation modes AD comprises. Second (Sections 3–6), it develops the mathematical and computational foundations of AD, presents the forward and reverse accumulation modes in sufficient technical detail to explain why reverse mode dominates in machine learning, and analyses Jacobian, Hessian, and higher-order derivative computation strategies. Third (Sections 7–8), it traces the historical convergence of reverse-mode AD with the backpropagation algorithm from its independent origins in Linnainmaa (1970), Werbos (1974), Speelpenning (1980), and Rumelhart, Hinton, and Williams (1986), and reviews the three principal implementation strategies used in modern software: operator overloading, source-to-source transformation, and tracing with just-in-time compilation. Fourth (Sections 9–10), it surveys contemporary applications and open problems, including nested differentiation, memory-efficient checkpointing, sound differentiation of probabilistic programs (Lew et al., 2023), and the formal correctness of AD in compiler-embedded settings such as Enzyme (Hückelheim et al., 2023; Moses & Churavy, 2020).

Automatic Differentiation and Its Alternatives

1. Numerical (Finite-Difference) Differentiation:

The most direct approach to estimating the derivative of a function $f: \mathbb{R}^n \rightarrow \mathbb{R}^m$ is to perturb an input coordinate by a small step h and observe the resulting change in output, following the classical definition of the derivative as a limit. In finite-precision arithmetic, this limit is approximated with a nonzero h , typically through a forward difference or a central difference:

$$\begin{aligned}\partial f / \partial x_i &\approx [f(x + h \cdot e_i) - f(x)] / h && \text{(forward)} \\ \partial f / \partial x_i &\approx [f(x + h \cdot e_i) - f(x - h \cdot e_i)] / (2h) && \text{(central)}\end{aligned}$$

where e_i denotes the i -th standard basis vector. The central variant has truncation error of order $\mathcal{O}(h^2)$, whereas the forward variant is only $\mathcal{O}(h)$; however, both estimates degrade catastrophically as h approaches machine epsilon, because subtracting two nearly equal floating-point numbers amplifies round-off error (Griewank & Walther, 2008). Selecting h therefore involves an unavoidable trade-off between truncation error, which shrinks as h decreases, and cancellation error, which grows as h decreases; no single value of h is optimal across all functions or all regions of input space. Complex-step differentiation partially mitigates the cancellation problem for analytic functions (Martins et al., 2003), but it is not a general remedy.

A second, decisive limitation is computational cost. Estimating a full gradient of a function with n inputs by forward differences requires at least $n + 1$ function evaluations, and $2n$ evaluations for the central scheme, because each partial derivative demands its own perturbation. For the parameter counts typical of contemporary

neural networks — routinely 10^8 to 10^{11} — this linear scaling in n renders numerical differentiation unfeasible as a primary training mechanism (Margossian, 2019). It nonetheless remains valuable as an independent oracle for verifying the correctness of AD implementations, and gradient checking against finite differences is a standard diagnostic in production AD software (Baydin et al., 2018).

2. Symbolic Differentiation:

Symbolic differentiation, by contrast, manipulates an explicit mathematical expression representing f , applying the standard differentiation rules — the sum, product, quotient, and chain rules — to produce a new expression that represents the exact derivative up to the precision of any subsequent numerical evaluation. Computer-algebra systems such as Mathematica, Maple, and SymPy implement this approach, and it is highly effective for short, closed-form expressions free of iteration or branching.

Difficulties arise when the target of differentiation is not a compact expression but a program: one containing loops, conditionals, recursion, or the composition of many subroutines. Naive symbolic differentiation of such a program — for example, by first fully unrolling it into an expression tree — produces what is commonly termed expression swell: the size of the symbolic derivative grows exponentially with the depth of composition, because standard differentiation rules tend to duplicate subexpressions rather than share them (Baydin et al., 2018). A function composed of k nested operations can therefore yield a symbolic derivative whose evaluation cost is exponential in k , even though the original function itself may be evaluated in time linear in k . Symbolic systems mitigate this partially through common-subexpression elimination, but the fundamental mismatch between symbolic manipulation of static expressions and the dynamic, stateful semantics of imperative programs remains (Griewank & Walther, 2008).

3. Automatic Differentiation as a Distinct Third Approach:

Automatic differentiation is neither an approximation like numerical differentiation nor a manipulation of static expressions like symbolic differentiation. Rather, AD decomposes a program into the sequence of elementary operations it actually executes — additions, multiplications, exponentials, trigonometric functions, comparisons, and so on — each of which has a known, analytically simple local derivative, and combines these local derivatives via the chain rule as the program runs (Baydin et al., 2018; Griewank & Walther, 2008). Because AD operates on the sequence of primitives that a program actually performs, rather than on a symbolic expression standing for that program, it handles loops, conditionals, and recursion natively: each execution of a loop body simply contributes a new set of primitives to the chain-rule computation, and no expression swell occurs, because it is intermediate numerical values, not symbolic subexpressions, that get combined and discarded.

The resulting technique shares symbolic differentiation's exactness — the final answer is correct up to floating-point rounding, not a finite-difference estimate — while sharing numerical differentiation's applicability to arbitrary programs,

including those with data-dependent branching. This combination is precisely what has established AD as the default gradient mechanism in essentially every modern machine learning framework (Abadi et al., 2016; Paszke et al., 2019; Bradbury et al., 2018).

Mathematical and Computational Foundations

1. The Chain Rule and Evaluation Traces:

Every technique within the AD family rests on a single mathematical fact: the derivative of a composition of functions is the product of the derivatives of its constituents. For a composite scalar map $y = f(g(x))$, the chain rule states

$$dy/dx = (df/dg) \cdot (dg/dx),$$

and, more generally, for a composition of vector-valued mappings, the Jacobian of the composition is the product of the individual Jacobians. A computer program that computes a function of many variables can be viewed as a long composition of elementary operations applied in sequence to intermediate variables. Griewank and Walther (2008) formalise this sequence as an evaluation trace or Wengert list, in which each intermediate variable v_i is defined either as an input, or as the result of applying an elementary operation ϕ_i to variables computed earlier in the trace:

$$v_i = \phi_i(v_{\{j_1\}}, v_{\{j_2\}}, \dots, v_{\{j_k\}}), \text{ where } j_l < i.$$

Because the derivative of each elementary operation with respect to its own arguments is analytically known and cheap to evaluate — the derivative of a multiplication, for example, is simply its other operand — the derivative of the entire composed function can be assembled by repeatedly applying the chain rule along the trace, without ever needing to construct a closed-form symbolic expression for the composition as a whole (Baydin et al., 2018). This property is what allows AD to sidestep expression swell: at every step, only numerical values and their local partial derivatives are retained, not the full symbolic history of how those values were produced.

2. Computational Graphs:

The evaluation trace has a natural graphical representation as a directed acyclic graph (DAG), termed the computational graph, in which nodes correspond to variables — inputs, intermediate results, and outputs — and directed edges connect each variable to the operations that consume it (Baydin et al., 2018). The graph can be constructed statically, before execution, as in the original graph-mode implementation of TensorFlow (Abadi et al., 2016), or dynamically, one operation at a time, as the program actually runs, an approach termed define-by-run and popularised by PyTorch (Paszke et al., 2019). The dynamic approach trades some opportunities for ahead-of-time optimisation for substantially greater flexibility in expressing models whose structure depends on their input — recurrent networks over variable-length sequences, tree-structured neural architectures, or programs with data-dependent control flow.

Whether the graph is built statically or dynamically, it plays the same conceptual role: it records the dependency structure needed to propagate derivative information, either from inputs toward outputs (forward-mode accumulation) or

from outputs backward toward inputs (reverse-mode accumulation) (Griewank & Walther, 2008).

Forward-Mode Accumulation

1. Dual Numbers and Tangent Propagation:

Forward-mode AD computes derivatives in lockstep with the original function evaluation, propagating a tangent value alongside each intermediate variable in the evaluation trace. The elegant algebraic formalisation of this idea uses dual numbers, an extension of the reals of the form $a + b \cdot \epsilon$ in which ϵ is a nilpotent symbol satisfying $\epsilon^2 = 0$ (Baydin et al., 2018; Griewank & Walther, 2008). If a variable x is represented as the dual number $x + \dot{x} \cdot \epsilon$, where $\dot{x}$ is the seed derivative with respect to a chosen input direction, then applying any smooth elementary function f to this dual number yields, by Taylor expansion and the nilpotency of ϵ ,

$$f(x + \dot{x} \cdot \epsilon) = f(x) + f'(x) \cdot \dot{x} \cdot \epsilon.$$

The ϵ -component of the result is therefore exactly the directional derivative of f at x , computed automatically as a side effect of evaluating f on the dual-number input, with no separate differentiation step required.

In practice, dual numbers are typically implemented not through a literal algebraic extension but through operator overloading: each elementary arithmetic operation is redefined so that when it receives a (value, tangent) pair as an argument, it returns a new (value, tangent) pair whose tangent component has been updated according to the corresponding differentiation rule (Griewank & Walther, 2008). A single forward pass through the program then produces, alongside the ordinary function value, the directional derivative of the output with respect to whichever single input direction was seeded with a tangent of one — a Jacobian–vector product (JVP).

2. Complexity Characteristics of Forward Mode:

Forward mode has the property that a single execution of the augmented program yields the derivative of every output with respect to one input direction. To obtain the full Jacobian of a function with n inputs and m outputs, forward mode must therefore be run once per input, for a total of n passes, each costing a small constant multiple of the cost of a single ordinary function evaluation (Griewank & Walther, 2008). This makes forward mode efficient when n is small relative to m — few inputs and many outputs — because cost scales with the number of inputs. Conversely, when n is large, as in neural networks with millions or billions of parameters and a scalar loss output, forward mode becomes prohibitively expensive, since it would require one full pass per parameter to assemble the gradient (Baydin et al., 2018). This asymmetry is the primary reason that reverse mode, analysed next, dominates in machine learning.

Reverse-Mode Accumulation

1. Adjoint Variables and the Backward Pass:

Reverse-mode AD propagates derivative information in the opposite direction, from outputs back toward inputs, using adjoint variables. For each intermediate

variable v_i in the evaluation trace, its adjoint $\bar{v}_i$ is defined as the partial derivative of a chosen output y with respect to v_i :

$$\bar{v}_i \equiv \partial y / \partial v_i.$$

Reverse mode computes these adjoints by traversing the evaluation trace backwards, applying the chain rule at each step to accumulate the contribution of v_i to every variable that depends on it downstream (Griewank & Walther, 2008). For a variable v_i that feeds into operations producing $v_{\{j_1\}}$, ..., $v_{\{j_p\}}$, the adjoint update rule is

$$\bar{v}_i = \sum_k \bar{v}_{\{j_k\}} \cdot (\partial v_{\{j_k\}} / \partial v_i).$$

This traversal requires two passes rather than one. In the forward pass, the program is executed normally, and every intermediate value is recorded — either explicitly on a stack, as in tape-based implementations, or implicitly through the structure of a computational graph. In the backward pass, adjoints are initialised at the output (typically to unity for a scalar loss) and propagated backward through the recorded trace, with each elementary operation contributing its local derivative, multiplied by the adjoint of its output, to the adjoints of its inputs (Baydin et al., 2018). The requirement to retain intermediate forward-pass values for reuse in the backward pass gives reverse mode its characteristic memory cost, which scales with the length of the evaluation trace rather than with the number of inputs or outputs.

2. Complexity and the Memory–Computation Trade-off:

The decisive advantage of reverse mode is that a single forward–backward pass computes the derivative of one output with respect to every input, at a computational cost that is a small constant multiple of the cost of evaluating the function itself, independent of the number of inputs (Griewank & Walther, 2008; Baydin et al., 2018). For a scalar-valued loss function of n parameters, as is standard in neural network training, this means that the entire gradient — all n partial derivatives — is obtained in roughly the time of one or two forward evaluations of the loss, rather than in time proportional to n as forward mode would require. This asymptotic advantage — sometimes called the cheap-gradient principle — is precisely why reverse-mode AD, and not forward-mode AD, underlies gradient-based training in essentially every deep learning framework in current use (Paszke et al., 2019; Abadi et al., 2016).

The trade-off is memory. Because intermediate activations computed during the forward pass must be retained until they are consumed in the backward pass, the memory footprint of reverse-mode AD grows with the depth and width of the computation, which for deep networks processing large batches can become the binding constraint on trainable model size well before computational throughput does. This memory pressure motivates the checkpointing strategies analysed in Section 10.2, which trade recomputation for reduced storage.

Jacobian, Hessian, and Higher-Order Derivatives

When the object of interest is not a single gradient but the full Jacobian matrix $J \in \mathbb{R}^{m \times n}$, or a Hessian, or a higher-order derivative tensor, the choice between forward and reverse mode — or a combination of both, termed mixed mode —

depends on the relative dimensions of the input and output spaces (Griewank & Walther, 2008).

Forward mode computes a single column of J per pass, corresponding to a Jacobian-vector product Jv , and is therefore preferable when $m > n$. Reverse mode computes a single row of J per pass, corresponding to a vector-Jacobian product $u^T J$, and is preferable when $n > m$ — overwhelmingly the typical case in machine learning, where a high-dimensional parameter vector is mapped to a single scalar loss (Baydin et al., 2018). Both modes can be “seeded” with a canonical basis to accumulate the full Jacobian, but for non-square problems only one mode is efficient.

Hessian computation, required for second-order optimisation, natural-gradient methods, and curvature-aware regularisation, is generally approached by composing the two modes: reverse mode is used to obtain a function that computes the gradient, and forward mode is then applied to that gradient function to produce Hessian-vector products without ever materialising the full Hessian (Griewank & Walther, 2008; Pearlmutter, 1994). This forward-over-reverse composition scales linearly rather than quadratically in the parameter count and underlies practical Newton-Krylov and Gauss-Newton methods in machine learning. Contemporary systems such as JAX expose the two primitives `jvp` and `vjp` directly to users, so that such compositions can be constructed by nesting calls to the two modes (Bradbury et al., 2018).

HISTORICAL DEVELOPMENT

The mathematical core of reverse-mode differentiation was described well before its significance to machine learning was recognised. Linnainmaa's 1970 master's thesis at the University of Helsinki (Linnainmaa, 1970; 1976) presented what is now recognised as the first complete formulation of reverse-mode accumulation, framed as a method for propagating rounding-error estimates through a sequence of arithmetic operations. Independently, in the control-theory literature, Werbos's (1974) Harvard doctoral dissertation developed a closely related backward-propagation scheme for computing sensitivities in dynamical systems, several years before it was connected to neural network training (Werbos, 1990).

Speelpenning's (1980) dissertation at the University of Illinois provided one of the earliest treatments of reverse-mode AD as an explicit general-purpose algorithm for differentiating arbitrary programs, establishing the complexity bound — that the cost of computing a full gradient is a small constant multiple of the cost of evaluating the original function — that remains the central selling point of reverse mode (Griewank & Walther, 2008).

The connection to neural networks was made prominent by Rumelhart, Hinton, and Williams (1986), whose Nature paper popularised backpropagation as a procedure for training multilayer networks by propagating error signals backwards from the output layer. Baydin et al. (2018) observe that backpropagation, as described in the neural-network literature, is mathematically a special case of reverse-mode AD applied specifically to a layered composition of affine transformations and pointwise nonlinearities; the machine learning and numerical-computing communities developed and rediscovered closely related techniques largely without cross-

reference for roughly two decades. This history explains why the terms are still frequently used interchangeably, even though AD is the more general technique, applicable to arbitrary programs rather than to feedforward layered architectures alone.

Implementation Strategies in Modern Software

1. Operator Overloading:

Operator overloading implements AD by defining a new numeric type — for forward mode, a dual number carrying a value and a tangent; for reverse mode, a node carrying a value and a reference to the operation that produced it — and redefining the basic arithmetic operators and mathematical functions so that they operate on the new type while transparently recording or propagating derivative information (Griewank & Walther, 2008). This is the strategy underlying eager-execution frameworks such as PyTorch, in which ordinary Python arithmetic on tensor objects implicitly builds a dynamic computational graph that is later traversed backward via the autograd engine (Paszke et al., 2019). Its principal advantage is that it requires minimal restructuring of user code: differentiable programs are written with ordinary control flow and function calls. Its principal cost is the runtime overhead of graph construction and the additional memory required to store the graph itself.

2. Source-Code Transformation:

An alternative strategy analyses the source code or an intermediate representation of a program directly and mechanically generates new source code that computes both the original function and its derivative, without constructing a runtime graph object (Griewank & Walther, 2008). Because the transformation happens ahead of time, source-transformation tools can apply conventional compiler optimisations to the generated derivative code, and they avoid the per-operation overhead intrinsic to operator overloading. Van Merriënboer, Breuleux, Bergeron, and Lamblin (2018) describe an approach of this kind — Tangent — for dynamically typed, NumPy-style array programs, and Innes (2018) develops a source-transformation method that operates directly on the static single-assignment intermediate representation used by modern compilers, thereby implementing AD as a compiler pass rather than as a library feature. Innes et al. (2019) extend this line of work into a general differentiable programming system intended to serve scientific computing alongside machine learning workloads.

3. Tracing, Just-in-Time Compilation, and Hybrid Systems:

A third family of systems occupies an intermediate position between pure operator overloading and pure source transformation, tracing a concrete execution of a Python function to record the sequence of primitive operations invoked and then compiling that trace with a just-in-time compiler before differentiating it. JAX exemplifies this approach: it traces NumPy-style functions into an intermediate representation (jaxpr), applies forward- or reverse-mode transformation rules to that representation, and compiles the result with the XLA compiler for execution on CPU, GPU, or TPU hardware (Bradbury et al., 2018). This design allows JAX to expose

composable function transformations — `grad`, `jvp`, `vjp`, `vmap`, and `jit` — that can be nested arbitrarily, enabling constructs such as forward-over-reverse Hessian–vector products to be expressed directly in user code.

A recent complementary development is Enzyme (Moses & Churavy, 2020; Hückelheim et al., 2023), a reverse-mode AD tool operating at the LLVM compiler intermediate-representation level. By differentiating already-optimised low-level code, Enzyme can generate efficient derivatives of GPU kernels and heavily inlined numerical routines that earlier source-level AD tools could not handle at all, at the cost of requiring careful reasoning about low-level memory operations that source-level tools are able to avoid entirely.

4. Numerical Stability Considerations:

Because AD computes the exact derivative of the sequence of floating-point operations a program actually performs, rather than of the mathematical function that sequence is intended to approximate, it can propagate and amplify numerical instabilities already latent in the forward computation, and it can introduce new ones at points of non-smoothness — absolute value, minimum and maximum, ReLU, and other piecewise-defined activations — where the mathematical derivative is undefined or discontinuous even though a particular subgradient is returned in practice (Griewank & Walther, 2008). Margossian (2019) emphasises that correctness of an AD tool, in the sense of faithfully implementing the chain rule, does not guarantee numerical robustness of the derivatives it returns — particularly for iterative numerical solvers, root-finding routines, or functions involving cancellation-prone subtractions — and that gradient checking against finite differences remains a standard diagnostic despite AD's general superiority in cost and precision.

Programming-language research has begun to formalise these correctness concerns more rigorously. Laurel, Yang, Ugare, Nagel, Singh, and Misailovic (2022) develop an abstract-interpretation framework for reasoning about the correctness of higher-order AD at points of non-differentiability, and Hückelheim et al. (2023) catalogue a set of recurring pitfalls in applying AD to scientific-computing codes — incorrect handling of aliasing, in-place mutation, and control flow that depends on the very quantities being differentiated — underscoring that the theoretical simplicity of the chain rule does not eliminate the practical difficulty of implementing it correctly on real-world software.

Applications In Machine Learning And Scientific Computing

1. Gradient-Based Optimisation and Neural Network Training:

The dominant application of reverse-mode AD is the computation of loss gradients with respect to model parameters for use in first-order optimisation methods — stochastic gradient descent, momentum-based variants, and adaptive methods such as Adam and its descendants. Every major deep learning framework exposes reverse-mode AD as its central computational primitive precisely because the cost of the full gradient, independent of parameter count, is what makes training networks with billions of parameters computationally feasible at all (Abadi et al., 2016; Paszke et al., 2019; Bradbury et al., 2018).

2. Probabilistic Programming and Variational Inference:

Beyond supervised training, AD underlies gradient-based approaches to approximate Bayesian inference, including variational inference and gradient-based Markov chain Monte Carlo methods such as Hamiltonian Monte Carlo, both of which require derivatives of a log-density or evidence lower bound with respect to model or variational parameters (Margossian, 2019). Probabilistic programming languages such as Stan, Pyro, and NumPyro increasingly build on general-purpose AD systems to obtain these derivatives automatically from arbitrary generative-model code. Recent work by Lew, Huot, Staton, and Mansinghka (2023) extends AD itself to soundly differentiate expectations of probabilistic programs (ADEV), addressing cases in which naive application of AD to stochastic computations yields biased or undefined derivative estimates.

3. Scientific Machine Learning and Differentiable Simulation:

A growing body of work applies AD not only to neural networks but to numerical simulators themselves, enabling gradients to be propagated through solvers for differential equations, physical simulations, and engineering models — an approach often described as differentiable programming (Innes et al., 2019; Sapienza et al., 2024). This allows a simulator to be embedded as a differentiable layer within a larger learning system, so that model parameters or boundary conditions can be optimised using gradients obtained by differentiating the entire simulation pipeline end-to-end, rather than by treating the simulator as a black box queried through slower and less accurate finite-difference approximations. Neural ordinary differential equations (Chen et al., 2018) and physics-informed neural networks (Raissi et al., 2019) are emblematic recent examples that rely critically on AD, in the latter case to enforce differential constraints inside the loss function itself.

OPEN PROBLEMS AND FUTURE DIRECTIONS

1. Nested and Higher-Order Differentiation:

Meta-learning, hyperparameter optimisation, and Hessian-based methods all require differentiating through a computation that itself contains a differentiation step — for instance, differentiating the outcome of an inner optimisation loop with respect to hyperparameters that control that loop. Composing AD with itself under forward-over-reverse or reverse-over-reverse schemes is conceptually straightforward (Section 6), but Laurel et al. (2022) show that giving these compositions a fully general and provably correct semantics, particularly in the presence of control flow and non-smooth operations, remains an unresolved problem in the underlying programming-languages theory, and that naive nesting can silently produce incorrect higher-order derivatives in certain edge cases.

2. Checkpointing and Memory-Efficient Reverse Mode:

The memory cost of reverse mode identified in Section 5.2 becomes acute for very deep networks and for AD applied to long iterative solvers, since the number of intermediate values that must be retained scales with the length of the computation.

Checkpointing strategies address this by storing only a subset of intermediate values during the forward pass and recomputing the rest on demand during the backward pass, trading additional computation for reduced memory (Griewank & Walther, 2000, 2008). Determining the checkpointing schedule that optimally balances this trade-off for a given computational graph — particularly for graphs with irregular or dynamically determined structure — continues to motivate algorithmic work at the intersection of AD and compiler scheduling.

3. Correctness and Formal Verification of AD Systems:

As AD systems have become more deeply embedded within compiler toolchains rather than implemented as standalone libraries, verifying that these systems compute mathematically correct derivatives across the full range of language features they must support — aliasing, mutation, exceptions, and non-smooth primitives — has emerged as an active concern (Hückelheim et al., 2023; Laurel et al., 2022). Enzyme (Moses & Churavy, 2020) illustrates both the promise and the difficulty of this direction: it generates efficient derivatives of already-optimised low-level code, but doing so correctly requires reasoning about memory operations that source-level tools are able to avoid. Sound differentiation of probabilistic and stochastic programs (Lew et al., 2023) constitutes a parallel line of formalisation work.

4. Differentiable Programming for Scientific Computing:

A recent comprehensive review by Sapienza, Bolibar, Schäfer, Groenke, and collaborators (2024) surveys differentiable-programming approaches for differential equations, comparing continuous-adjoint, discrete-adjoint, and full-AD sensitivity methods across accuracy, stability, and memory profiles. Their analysis highlights that neither forward nor reverse mode is uniformly optimal for problems dominated by stiff differential equations, and that hybrid analytic-algorithmic approaches remain an active research frontier.

CONCLUSION

Automatic differentiation is neither an approximation, as numerical differentiation is, nor a symbolic manipulation, as computer algebra is. It is a distinct third method that decomposes a program into elementary operations with known local derivatives and combines them via the chain rule as the program executes, yielding exact derivatives at a cost that is bounded relative to the cost of the original computation. Reverse-mode accumulation — the case that computes a full gradient in a single backward pass at a cost independent of the number of inputs — is what has rendered gradient-based training of large-scale machine learning models computationally feasible, and its identity with the backpropagation algorithm long used in neural network research reflects a convergence of ideas first developed independently in numerical analysis, control theory, and connectionism.

Contemporary deep learning and scientific-computing frameworks have absorbed AD so thoroughly into their design that differentiability is now treated as a default property of numerical code rather than as a capability that must be separately engineered. The problems that remain open are correspondingly less about whether

derivatives can be computed than about whether they can be computed correctly, efficiently, and verifiably across the full breadth of modern software: through nested compositions of differentiation with itself, through memory-constrained checkpointing schedules, through low-level, heavily optimised code that earlier generations of AD tools were never designed to handle, and through stochastic and probabilistic programs in which the very notion of the derivative of an expectation must be redefined. These are the directions along which the theoretical core of AD, essentially unchanged since Linnainmaa's original 1970 formulation, continues to be extended.

Acknowledgements:

The authors thank the Department of Mathematics, Faculty of Education, Helmand University, for institutional support throughout the preparation of this manuscript. No external funding was received.

Conflict of Interest Statement:

The authors declare no competing or conflicting interests.

Data Availability Statement:

This review article does not report new empirical data. All sources cited are publicly available through the venues indicated in the reference list.

Author Contributions:

M. Q. Adil conceptualised the review, drafted Sections 1–6 and 11, and prepared the reference list. M. Ghafare drafted Sections 7–10 and contributed the analysis of implementation strategies and open problems. Both authors revised the manuscript, verified all citations, and approved the final version.

REFERENCES

- Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., ... Zheng, X. (2016). TensorFlow: A system for large-scale machine learning. In Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16) (pp. 265–283). USENIX Association.
- Baydin, A. G., Pearlmutter, B. A., Radul, A. A., & Siskind, J. M. (2018). Automatic differentiation in machine learning: A survey. *Journal of Machine Learning Research*, 18(153), 1–43. <https://www.jmlr.org/papers/volume18/17-468/17-468.pdf>
- Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., & Zhang, Q. (2018). JAX: Composable transformations of Python+NumPy programs [Computer software]. Google Research. <https://github.com/google/jax>
- Chen, R. T. Q., Rubanova, Y., Bettencourt, J., & Duvenaud, D. (2018). Neural ordinary differential equations. In Advances in Neural Information Processing Systems 31 (pp. 6571–6583). Curran Associates.

- Griewank, A., & Walther, A. (2000). Algorithm 799: Revolve — An implementation of checkpointing for the reverse or adjoint mode of computational differentiation. *ACM Transactions on Mathematical Software*, 26(1), 19–45. <https://doi.org/10.1145/347837.347846>
- Griewank, A., & Walther, A. (2008). *Evaluating derivatives: Principles and techniques of algorithmic differentiation* (2nd ed.). Society for Industrial and Applied Mathematics. <https://doi.org/10.1137/1.9780898717761>
- Hückelheim, J., Menon, H., Moses, W., Christianson, B., Hovland, P., & Hascoët, L. (2023). Understanding automatic differentiation pitfalls. *arXiv*. <https://arxiv.org/abs/2305.07546>
- Innes, M. (2018). Don't unroll adjoint: Differentiating SSA-form programs. *arXiv*. <https://arxiv.org/abs/1810.07951>
- Innes, M., Edelman, A., Fischer, K., Rackauckas, C., Saba, E., Shah, V. B., & Tebbutt, W. (2019). A differentiable programming system to bridge machine learning and scientific computing. *arXiv*. <https://arxiv.org/abs/1907.07587>
- Laurel, J., Yang, R., Ugare, S., Nagel, R., Singh, G., & Misailovic, S. (2022). A general construction for abstract interpretation of higher-order automatic differentiation. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2), Article 174. <https://doi.org/10.1145/3563324>
- Lew, A. K., Huot, M., Staton, S., & Mansinghka, V. K. (2023). ADEV: Sound automatic differentiation of expected values of probabilistic programs. *Proceedings of the ACM on Programming Languages*, 7(POPL), Article 121. <https://doi.org/10.1145/3571198>
- Linnainmaa, S. (1970). The representation of the cumulative rounding error of an algorithm as a Taylor expansion of the local rounding errors [Master's thesis, University of Helsinki].
- Linnainmaa, S. (1976). Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 16(2), 146–160. <https://doi.org/10.1007/BF01931367>
- Margossian, C. C. (2019). A review of automatic differentiation and its efficient implementation. *WIREs Data Mining and Knowledge Discovery*, 9(4), Article e1305. <https://doi.org/10.1002/widm.1305>
- Martins, J. R. R. A., Sturdza, P., & Alonso, J. J. (2003). The complex-step derivative approximation. *ACM Transactions on Mathematical Software*, 29(3), 245–262. <https://doi.org/10.1145/838250.838251>
- Moses, W., & Churavy, V. (2020). Instead of rewriting foreign code for machine learning, automatically synthesize fast gradients. In *Advances in Neural Information Processing Systems 33* (pp. 12472–12485). Curran Associates.
- Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Köpf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., ... Chintala, S. (2019). PyTorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32* (pp. 8024–8035). Curran Associates.
- Pearlmutter, B. A. (1994). Fast exact multiplication by the Hessian. *Neural Computation*, 6(1), 147–160. <https://doi.org/10.1162/neco.1994.6.1.147>

- Raissi, M., Perdikaris, P., & Karniadakis, G. E. (2019). Physics-informed neural networks: A deep learning framework for solving forward and inverse problems involving nonlinear partial differential equations. *Journal of Computational Physics*, 378, 686–707. <https://doi.org/10.1016/j.jcp.2018.10.045>
- Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533–536. <https://doi.org/10.1038/323533a0>
- Sapienza, F., Bolibar, J., Schäfer, F., Groenke, B., Pal, A., Boussange, V., Heimbach, P., & Pérez-Carrasco, R. (2024). Differentiable programming for differential equations: A review. *arXiv*. <https://arxiv.org/abs/2406.09699>
- Speelpenning, B. (1980). Compiling fast partial derivatives of functions given by algorithms [Doctoral dissertation, University of Illinois at Urbana-Champaign]. Department of Computer Science Technical Report UIUCDCS-R-80-1002.
- Van Merriënboer, B., Breuleux, O., Bergeron, A., & Lamblin, P. (2018). Tangent: Automatic differentiation using source-code transformation for dynamically typed array programming. In *Advances in Neural Information Processing Systems* 31 (pp. 6256–6265). Curran Associates.
- Werbos, P. J. (1974). Beyond regression: New tools for prediction and analysis in the behavioral sciences [Doctoral dissertation, Harvard University].
- Werbos, P. J. (1990). Backpropagation through time: What it does and how to do it. *Proceedings of the IEEE*, 78(10), 1550–1560. <https://doi.org/10.1109/5.58337>