Automatic Differentiation in AI

Why Julia & Enzyme?

Salim Alkhaddoor
Friedrich Schiller University Jena

FRIEDRICH-SCHILLER-UNIVERSITÄT JENA

Motivation

- Gradients power modern optimization (SGD, quasi-Newton) in ML and scientific computing.
- Three ways to get derivatives:
  - **Numerical (finite differences)**: trivial to implement but needs $O(d)$ function calls for d inputs; sensitive to step size; truncation & roundoff errors accumulate [1, 2].
  - **Symbolic**: algebraically exact but brittle on real programs (control flow, mutability) and risks expression swell [3, 1].
  - **Automatic Differentiation (AD)**: applies the chain rule to the *executed* program; derivatives are accurate to machine precision with cost within a small constant of the primal evaluation [2, 1].
- **Rule of thumb** reverse-mode for scalar losses with many inputs; forward-mode when outputs dominate; mix modes when dimensions are comparable [2].

Fundamentals of AD

- Program-level chain rule without forming Jacobians explicitly. For $y = f(x)$:

$$\text{JVP: } J_f(x)\, v \quad \text{(forward mode)}, \qquad \text{VJP: } J_f(x)^\top w \quad \text{(reverse mode)}[2,\ 1].$$

- **Forward mode** (dual numbers): one sweep per seed v; overhead scales with $\#$inputs; integrates naturally with control flow via overloaded primitives [1].
- **Reverse mode** (adjoints/tape): for a scalar loss, only a single backward sweep is needed; for an m-dimensional output, m sweeps are required (or batched vector–Jacobian products). Overhead therefore scales with the number of outputs; it requires saving or recomputing intermediates, and checkpointing trades memory for time [4, 2].
- **Mixed & higher-order** compose JVPs and VJPs for Hessian-vector and Jacobian-vector products at near-primal cost [5].

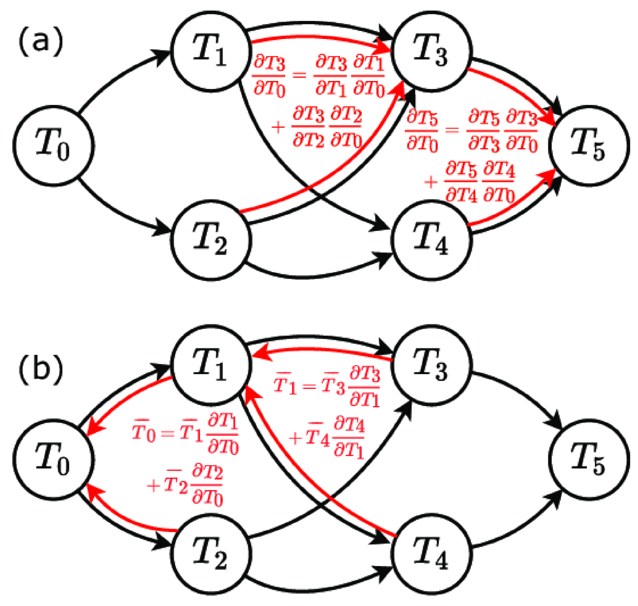


Figure: FIG. 1. (a) Forward-mode and (b) reverse-mode automatic differentiation on computational graphs. Black arrows denote the forward pass from inputs to outputs. Red arrows show the forward chain rule in (a) and adjoint back-propagation in (b).

Julia & Its AD Ecosystem

Why Julia for AD?

- **LLVM JIT** compiles Julia code to optimized native machine code at runtime; inspect with `@code_llvm f(1.0)` [6].
- **Multiple Dispatch** selects methods based on argument types for specialization and speed:

```
f(x::Int)      = x + 1
f(x::Float64)  = 2x
```

- **Parametric Types** generic, type-safe data structures for reusable algorithms: `struct Pair{T}; x::T; y::T; end` [6].
- **Compiler Introspection** direct access to Julia's AST and IR for metaprogramming.

Core AD Packages

- `ForwardDiff.jl` forward-mode AD via dual numbers. [1, 7].
- `ReverseDiff.jl` reverse-mode AD with runtime tapes. [1, 8].
- `Zygote.jl` source-to-source AD on Julia IR. [9].
- `ChainRulesCore.jl` infrastructure for defining custom forward/reverse rules. [10].
- `Enzyme.jl` AD as an LLVM IR pass (forward and reverse). [11].

AD Implementation Paradigms

- **Operator Overloading (Dual Numbers)** seamlessly overload arithmetic to carry derivatives; trivial in Julia but can allocate memory per operation—mitigated by pooling or static arrays [1].
- **Source/IR Transformation** perform AD at compile time by rewriting AST or LLVM IR; Zygote (SSA-based) inlines and optimizes gradients [9], while Enzyme integrates as an LLVM pass for deep optimization [11].
- **Tape-Based (Wengert Lists)** record ops at runtime, then run a backward sweep. Handles dynamic control flow; large tapes require checkpointing to save memory [12] [4].
- **Custom Gradients & Hybrid Modes** define bespoke derivative rules with ChainRulesCore.jl for non-standard code paths [10]; combine forward and reverse sweeps for efficient Jacobian-/Hessian–vector products.
- **Emerging Paradigms** incremental AD for streaming data, event-driven AD in reactive systems, and probabilistic AD via Monte Carlo estimators [13].

Deep Dive: Enzyme

Architecture & Phases

- **Frontend → IR** capture Julia/C/C++/Fortran function as LLVM-IR or MLIR, preserving control flow and type metadata.
- **Activity Analysis** lightweight pass identifies "active" (differentiable) values and instructions [11].
- **Adjoint & Shadow Buffers** allocate dual-value buffers for primal and adjoint data, enabling in-place accumulation.
- **Gradient Codegen** the intrinsic `__enzyme_autodiff` emits optimized derivative IR for forward, reverse or mixed modes.
- **Re-Optimization** rerun LLVM passes (inlining, GVN[i], loop vectorization) to fuse primal and adjoint code and remove dead branches.

Advanced Capabilities

- *Higher-Order Derivatives*: nest `autodiff` calls for Hessian-vector products or full Hessians.
- *Custom Rules*: define low-level derivatives for intrinsics, memory-side effects or GPU kernels.
- *Mixed-Precision*: supports FP16<–>FP32 for performance and numerical stability.
- *Checkpointing Integration*: use runtime checkpoints to trade memory for recomputation.

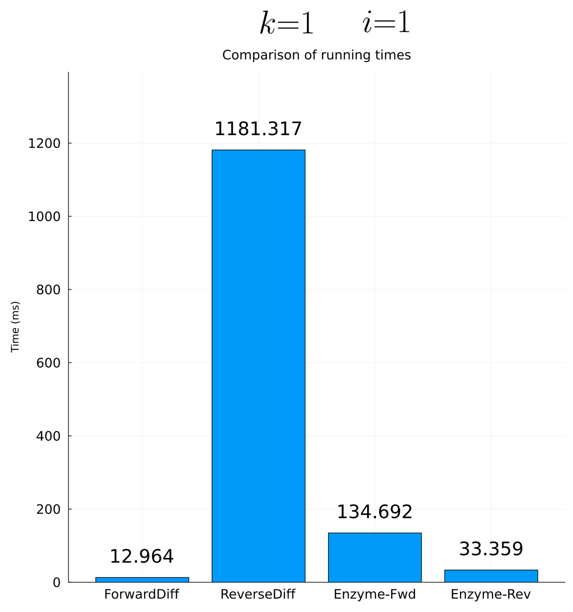
Example Workflow

```
using Enzyme

function loss(x)
        return sum(tanh.(x).^3) + dot(x, x)
end

# 1st-order reverse-mode gradient d(loss)/d x at x
x  = randn(1000)
dx = zeros(length(x))
Enzyme.autodiff(Enzyme.Reverse, loss, Enzyme.Duplicated(x, dx))
# gradient now stored in dx

# Hessian-vector product via forward-over-reverse (FoR)
function hvp(x, v)
        function g(u)
                du = zeros(length(u))
                Enzyme.autodiff(Enzyme.Reverse, loss, Enzyme.Duplicated(u, du))
                return dot(du, v)  # scalar: <grad(loss(u)), v>
        end
        # JVP of g at x in direction v equals H(x)*v
        Enzyme.autodiff(Enzyme.Forward, g, Enzyme.Duplicated(x, v))
end
```

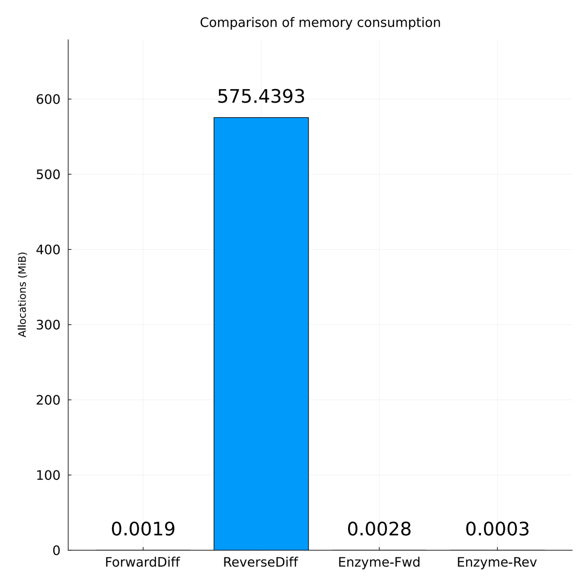
Benchmarks: Enzyme vs. JuliaDiff

Forward function

$$\operatorname{loss}(p) = \sum_{k=1}^{100000} \sum_{i=1}^{5} \sin\big(t_k\, p_i\big)\, e^{-\,p_{5+i}\, t_k}, \qquad t_k = k, \quad p \in \mathbb{R}^{10}.$$



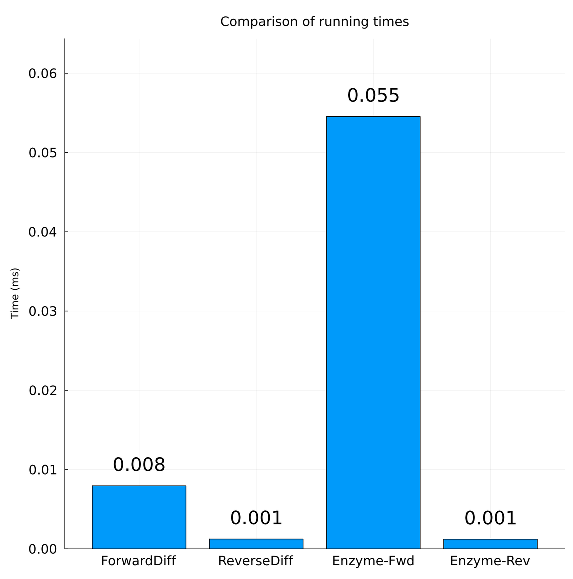
(a) Forward: Time



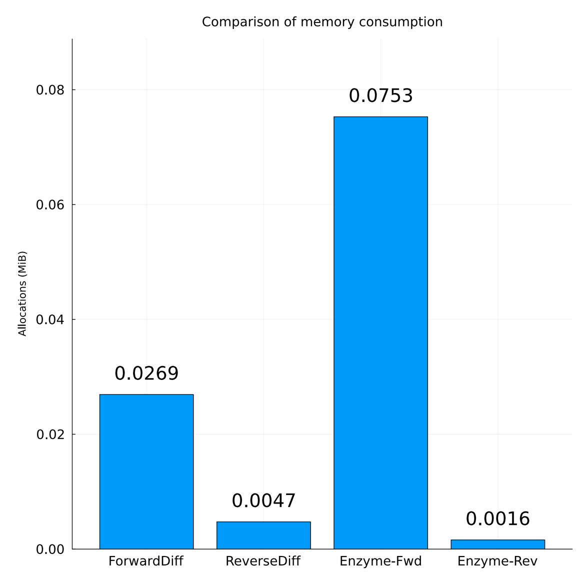
(b) Forward: Memory

Reverse function

$$\operatorname{loss}(x) = \sum_{i=1}^{n} \tanh\big(x_i\big)^3 + x^\top x = \sum_{i=1}^{n} \Big[\tanh\big(x_i\big)^3 + x_i^2\Big], \quad x \in \mathbb{R}^n, \quad n = 50.$$



(c) Reverse: Time



(d) Reverse: Memory

Practical Considerations & Impact

- **Maximal Performance** exploits post-optimization LLVM IR for near-native speed, minimizing overhead in adjoint generation.
- **Language-Agnostic** operates on LLVM IR, allowing differentiation across many LLVM-based languages (e.g., C, C++, Fortran, Rust, Julia, Swift) as long as the code is statically analyzable. Not every language or FFI boundary[ii] is automatically differentiable.
- **HPC Scalability** designed for large-scale CPU clusters; GPU and distributed-memory backends under active development [14].
- **Requirements** code must be analyzable at the LLVM IR level; manual annotations may be needed for side-effects, aliasing, or non-standard memory layouts.
- **Use Cases** PDE solvers, scientific sensitivity analysis, differentiating legacy C/Fortran HPC codes, batched Jacobian computations for machine learning or UQ.

References

[1] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic Differentiation in Machine Learning: A Survey," *J. Mach. Learn. Res.*, vol. 18, no. 153, pp. 1–43, 2018.

[2] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, 2nd ed., SIAM, 2008. doi:10.1137/1.9780898717761.

[3] U. Naumann, *The Art of Differentiating Computer Programs*, SIAM, 2012. doi:10.1137/1.9781611972078.

[4] A. Griewank and A. Walther, "Algorithm 799: REVOLVE: An Implementation of Checkpointing for the Reverse or Adjoint Mode of Computational Differentiation," *ACM Trans. Math. Softw.*, vol. 26, no. 1, pp. 19–45, 2000. doi:10.1145/347837.347846.

[5] B. A. Pearlmutter, "Fast Exact Multiplication by the Hessian," *Neural Computation*, vol. 6, no. 1, pp. 147–160, 1994. doi:10.1162/neco.1994.6.1.147.

[6] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A Fresh Approach to Numerical Computing," *SIAM Rev.*, vol. 59, no. 1, pp. 65–98, 2017. doi:10.1137/141000671.

[7] J. Revels, M. Lubin, et al., "ForwardDiff.jl Documentation," JuliaDiff, 2021–. Available: `https://juliadiff.org/ForwardDiff.jl/stable/`

[8] M. Innes, et al., "ReverseDiff.jl," JuliaDiff, 2021–. Available: `https://github.com/JuliaDiff/ReverseDiff.jl`

[9] M. Innes, "Don't Unroll Adjoint: Differentiating SSA-Form Programs," arXiv:1810.07951, 2018. Available: `https://arxiv.org/abs/1810.07951`

[10] L. White, J. R. Zammit, F. Hagen, et al., "ChainRulesCore.jl," JuliaDiff Project Documentation, 2021–. Available: `https://juliadiff.org/ChainRulesCore.jl/stable/`

[11] W. S. Moses, V. Churavy, L. Hannel, J. Paulo, A. Shafran, and T. Schulthess, "Enzyme: High-Performance Automatic Differentiation for LLVM," in *Advances in Neural Information Processing Systems 33 (NeurIPS)*, 2020. Available: `https://enzyme.mit.edu`

[12] A. Walther and A. Griewank, "Getting Started with ADOL-C," in *Dagstuhl Seminar Proceedings 09061: Automatic Differentiation: Applications, Theory, and Tools*, 2009. Available: `https://drops.dagstuhl.de/entities/document/10.4230/DagSemProc.09061.10`

[13] S.-X. Zhang, Z.-Q. Wan, and H. Yao, "Automatic differentiable Monte Carlo: Theory and application," *Phys. Rev. Research*, vol. 5, 033041, 2023. doi:10.1103/PhysRevResearch.5.033041.

[14] W. S. Moses, J. Hückelheim, L. Hannel, T. Besard, et al., "Reverse-Mode Automatic Differentiation and Optimization of GPU Kernels via Enzyme," in *Proc. SC '21*, ACM, 2021. doi:10.1145/3458817.3476165.

[i] **GVN (Global Value Numbering):** LLVM optimization that removes fully or partially redundant computations and redundant loads, improving code generated by AD passes. [ii] **FFI boundary:** Interface where code calls into a foreign language/runtime (e.g., `ccall` from Julia to C/Fortran). Across an FFI boundary, differentiation is not automatic unless custom rules or wrappers are provided.