# Automatic Differentiation heart of recent developments in Machine Learning

Piotr Mackiewicz
*Faculty of Electrical Engineering*
*Warsaw University of Technology*
Warsaw, Poland
piotr.mackiewicz.stud@pw.edu.pl

***Abstract*—This paper details the design and implementation of a Automatic Differentiation (AD) framework in the Julia programming language. The framework utilizes a dynamic computation graph and reverse-mode differentiation to provide the core functionality required for training deep learning models. To demonstrate its capabilities and validate its correctness, a Convolutional Neural Network (CNN) was built from scratch using the framework and trained on the IMDB sentiment analysis dataset. Results of the implementation proved functionally correct, achieving a competitive test accuracy with PyTorch and Flux.jl and comparable execution time for Flux.jl. However memory usage are not yet competitive with these mature libraries. This work highlights Julia's capabilities for AD development and provides an efficient framework for deep learning exploration and education.**

***Index Terms*—Automatic Differentiation, Julia Computing, Machine Learning**

## I. Differentiation in computer calculation

Derivatives are one of the fundamental tools of modern science and engineering, they make it possible to study the local structure of the solution space of differential equations, guide optimization processes and estimate the sensitivity of models to parameter changes. However, the increasing scale of computational problems from the simulation of turbulent flows to the learning of deep neural networks has forced the development of specialized differentiation techniques.

## II. Literature

### A. Automatic Differentiation

Automatic (or *algorithmic*) differentiation (AD) is currently the most developed method of differentiation for practical applications next to the Numerical Differentiation (often abbreviated FD for *finite differences*) [2] and Symbolic differentiation (SD). AD interprets a program that evaluates a scalar- or vector-valued function $f(x)$ as a sequence of *elementary operations* whose derivatives are known exactly. By representing the program as a directed acyclic graph (DAG) [1], AD propagates derivative information alongside the primal values by applying the chain rule to every elementary operation, carrying tangents in forward mode or accumulating adjoints (gradients) in reverse mode (backward).

- **Forward mode.** Each intermediate variable $v_k$ is paired with a *tangent* $\dot{v}_k$ that satisfies A single forward sweep yields a directional derivative $\nabla f(x)\, v$ at a cost of roughly one–two primal evaluations, independent of the output dimension.
- **Reverse mode.** Sweeping from the outputs back to the inputs, AD accumulates *adjoints* returning the full gradient $\nabla f(x)$ for a cost of four–six primal evaluations, regardless of input dimension. This procedure is the algorithmic core of back-propagation in deep learning.

AD supplies derivatives that are **accurate to machine precision**—limited only by floating-point round-off— [3] [4] and its cost scales linearly with the number of operations in the original code. Current research addresses reverse-mode memory pressure through checkpointing and selective recomputation, and extends AD to dynamic control flow such as data-dependent loops and branches [9]

In machine learning, AD enables the efficient training of deep neural networks; frameworks including PyTorch, TensorFlow, and Flux rely on it to compute gradients for models with millions of parameters in real time on GPUs.

Beyond ML, AD drives shape optimisation in aerodynamics (e.g. CFD) [5], inverse problems in geophysics [6], sensitivity analysis in quantitative finance [7], and robotics control [8].

Due to the absolutely crucial role of Automatic Differentiation in current research and engineering practice, it is worthwhile to undertake research on the use of a new programming language Julia and harness it to create a library for Automatic Differentiation and test it in a real-world problem of back-propagation in a Convolutional Neural Network (CNN) showing its capabilities and limitations of this programming language.

### B. Julia Programming Language

The Julia programming language is a high-level, high-performance, open-source language designed for numerical and scientific computing. The language's authors set themselves the goal of creating a language that would combine the best features of all major technical computing languages in one. In this way, over the years, the community has created a language in which we can find features of the following programming languages [10]:

1) Python - readability and ease of use.

2) MATLAB - mathematical syntax and numerical computing.
3) R - statistical computing.
4) C/C++ - performance.
5) Lisp - metaprogramming capabilities.

This approach has created a unique language with significant capabilities and distinctive features such as:

1) Multiple dispatch - A powerful feature for defining function behavior across different combinations of argument types.
2) Built-in parallelism: Native support for parallel and distributed computing.
3) Mathematical syntax: Clean, expressive syntax that's particularly friendly to scientists and mathematicians.
4) JIT compilation: Uses LLVM to compile code just-in-time for execution, enabling speed optimizations.

In view of the above, it can be said that Julia has excelled in bridging the gap between ease of use and execution speed as well as solving technical computing problems efficiently, without trading off performance for productivity, empowering researchers to write fast, clean, and reproducible code in one language. This is why this language is used by the world's leading institutions such as NASA [11], CERN [12], BlackRock [13] for Modeling Spacecraft Separation Dynamics, High Energy Physics Analysis, financial time-series analysis, respectively.

## III. FRAMEWORK DESIGN AND IMPLEMENTATION

The framework facilitates the construction of dynamic computation graphs, enabling automatic differentiation (AD) through reverse-mode propagation. It supports a variety of common neural network operations and layers, including dense layers, 1D convolutions, max pooling, embeddings, and activation functions. The design emphasizes modularity, extensibility, and a user-friendly interface leveraging Julia's multiple dispatch and operator overloading capabilities.

### A. Computation Graph

The cornerstone of the framework is a dynamically constructed computation graph. Each node in this graph represents either data (constants, variables) or an operation.

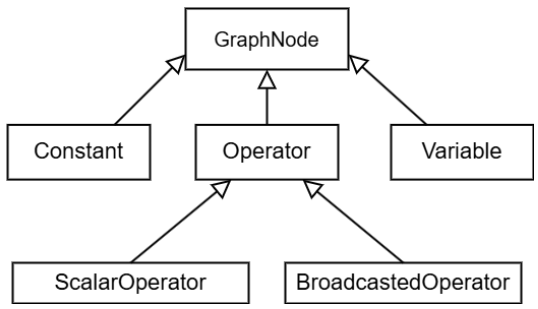


Fig. 1. Hierarchy of Computation Graph Nodes.

### B. Node Abstractions

The hierarchy of node types is defined as follows:

- `GraphNode`: An abstract base type for all elements within the computation graph.
- `Operator`: An abstract type representing computational operations. It serves as a parent for specific operation nodes.

### C. Graph Traversal and Topological Sort

To ensure computations are performed in the correct order, a topological sort is implemented:

- `visit(node, visited, order)`: A recursive helper function for depth-first traversal. For `Operator` nodes, it visits input nodes before adding the operator itself to the `order`.
- `topological_sort(head::GraphNode)`: Initializes an empty `visited` set and `order` vector, then calls `visit` on the head (output) node of the graph to produce a topologically sorted list of nodes.

## IV. AUTOMATIC DIFFERENTIATION IMPLEMENTATION

The framework implements reverse-mode automatic differentiation.

### A. Forward Pass

The forward pass computes the output value of each node in the graph.

- `reset!`: Resets the gradient of a `Variable` or `Operator` to `nothing` before a new forward pass.
- `compute!`: Computes the output for a given node. For `Operator` nodes, it calls a generic `forward(operator, input_outputs...)` method, which is dispatched to a specific implementation based on the operator type. The result is stored in `node.output`.
- `forward!`: Iterates through the topologically sorted nodes. For each node, it calls `reset!` and then `compute!`. Returns the output of the final node in the order.

### B. Backward Pass

The backward pass implements reverse-mode automatic differentiation to compute gradients of a scalar loss function with respect to all participating nodes in the graph.

- **Initialization**: The process begins by seeding the gradient of the graph's final output node (typically with 1.0). This represents $\partial L/\partial L$.
- **Reverse Traversal**: The framework iterates through the topologically sorted graph nodes in reverse order.
- **Operator Gradient Propagation:** For each `Operator`, its accumulated output gradient ($\partial L/\partial \text{output}_{\text{op}}$) is used. The operator-specific `backward` method then applies the chain rule locally to calculate the gradients for its inputs ($\partial L/\partial \text{input}_i$).
- **Gradient Accumulation** `update!`: Gradients are accumulated in `Variable` and `Operator` nodes, allowing

nodes to receive gradients from multiple downstream paths.

This systematic application of the chain rule efficiently calculates all required gradients for parameter updates.

## V. Key Neural Network Layer and Operations

Each specialized operation, or layer, is defined by a pair of methods: a `forward` method to compute its output and a `backward` method to calculate how gradients should flow back to its inputs.

- `EmbeddingLayer` Maps integer indices (e.g., words in a vocabulary) to dense, learnable vectors. It functions as a learnable lookup table or dictionary. Caches the input indices from the forward pass to efficiently accumulate gradients only for the embeddings that were actually used, avoiding a full matrix gradient computation.
- `permute_dims` Reorders the dimensions of a tensor. It's a utility for aligning data shapes between different layer types (e.g., from '(W,C,N)' to '(C,W,N)'). The inverse permutation is pre-calculated and cached when the node is created, saving computation during every backward pass.
- `Conv1DLayer` Detects local patterns in sequential data by sliding learnable filters (kernels) across the input. This implementation avoids explicit sliding-window loops by using the **`im2col`** (image-to-column) technique. This unnrolls all local input patches into columns of a large matrix, transforming the entire convolution operation into a single, matrix multiplication. This leverages fast BLAS libraries for a significant speedup. The backward pass uses the inverse operation, **`col2im`**.
- `Maxpool1d` Downsamples the input by selecting only the maximum value from each local window. This reduces dimensionality and creates a more robust, translation-invariant representation. It caches the index ('argmax') of the maximum value from each window during the forward pass. This allows the backward pass to instantly route the gradient to the correct input neuron without needing to re-calculate which one was the maximum.
- `Flatten` Reshapes a multi-dimensional feature tensor into a 2D matrix. It acts as a bridge between feature-extracting layers and classification layers. Caches the original input shape during the forward pass so that it can correctly "un-flatten" the gradient back to that shape during the backward pass.
- `DenseLayer` This is the fundamental building block of most neural networks fully connected layer and is used for classification and regression.
- Activation Functions Implementation includes sigmoid and ReLU function.

## VI. Evaluation and Comparison

### A. Model and Dataset Structre

The project utilizes IMDB Movie Review Dataset to perform a binary sentiment classification task. The objective is to train a neural network to predict whether a given movie review is positive or negative. Data has been prepared and stored in a Julia data format `JLD2`. The loaded dataset is structured as a dictionary containing six key components. These components include training and testing sets, a pre-trained word embedding matrix and vocabulary list.

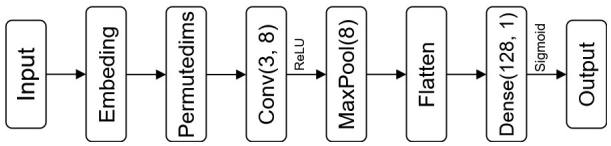


Fig. 2. Model Structure

### B. Training Process and Comparative Parameters

Test procedure involved running five training epochs with a batch size of 64 across all implementations[1], using the ADAM optimizer with a learning rate of $1e-3$ and the Binary Cross-Entropy (BCE) loss function as the criterion.

As part of the tests, the following parameters were compared:

- Training time (for initial start-up)[2]:
  - Per-epoch
  - Total Training time
- Final model accuracy and set loss test
- Complete memory allocations

### C. Experimental Results

TABLE I
Computational Time Comparison.

| Framework | Avg. Time/Epoch [s] | Total Time (5 epochs) [s] |
|---|---|---|
| PyTorch | 4.31 | 21.54 |
| Flux.jl | 15.49 | 77.43 |
| $\mu Flux.jl$ | 16.38 | 81.89 |

TABLE II
Accuracy Comparison - after 5 epoch.

| Framework | Test Loss | Test Accuracy |
|---|---|---|
| PyTorch | 0.3371 | 87.49% |
| Flux.jl | 0.3403 | 86.57% |
| $\mu Flux.jl$ | 0.3345 | 86.98% |

TABLE III
Memory Allocations Comparison - after 5 epoch.

| Framework | Allocations | Memory |
|---|---|---|
| Flux.jl | 64.30 M | 72.969 GiB |
| $\mu Flux.jl$ | 197.91 M | 110.216 GiB |

[1]Julia 1.11.3 with Flux 0.16.3, Python 3.11.9 with PyTorch 2.7.1

[2]Platform that was used for test has following specification: Operating System: Windows 11 Home, CPU: 16 × AMD Ryzen 7 4800H, RAM: 31.4 GB

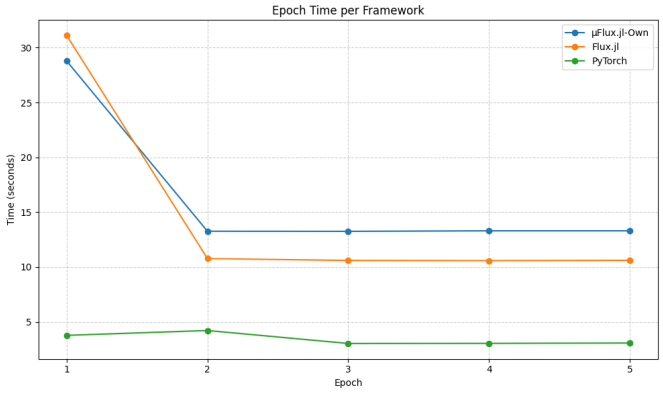


Fig. 3. Comparison of time per epoch.

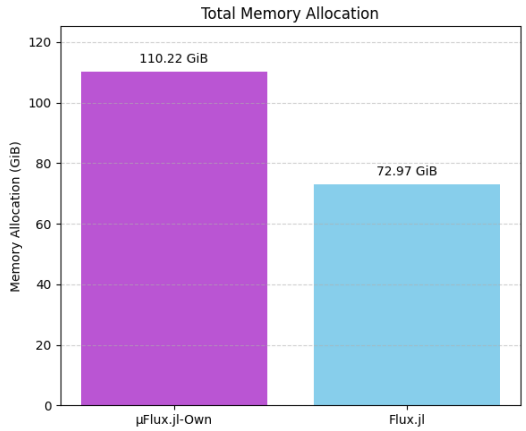


Fig. 4. Total Memory Allocation After 5 Epoch.

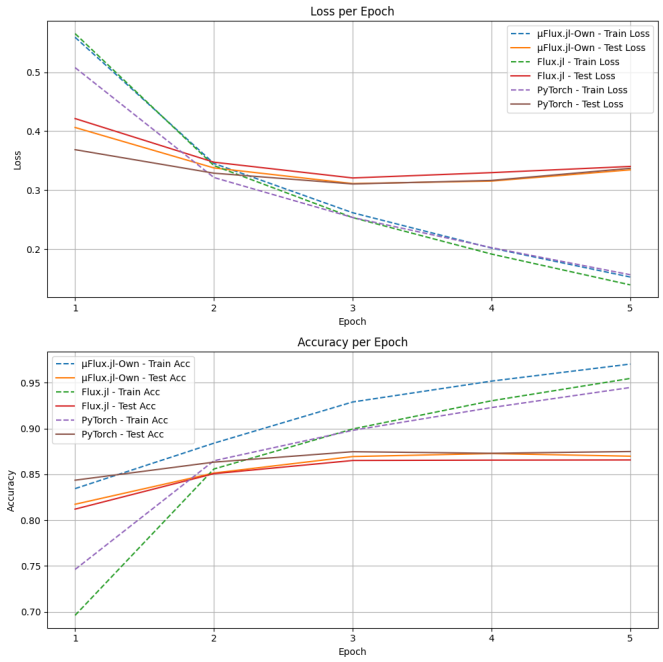


Fig. 5. Summary comparison of implementations results.

### D. Analysis of Results

**Accuracy:** The results from Table II and Figure 5 are encouraging. `μFlux.jl` achieved a test accuracy (86.98%) that was highly competitive, slightly surpassing `Flux.jl` (86.57%) and closely trailing `PyTorch` (87.49%). The test loss for `μFlux.jl` (0.3345) was also the lowest. This indicates that the core AD mechanism with in `μFlux.jl` are correctly implemented and effective for training the defined CNN model.

**Computational Time:** In terms of speed (Table I, Figure 3), `PyTorch`, was significantly faster (average 4.31s/epoch). `μFlux.jl` (average 16.38s/epoch) was slightly slower than `Flux.jl` (average 15.49s/epoch). The initial epoch for both Julia frameworks showed a higher execution time, due to attributable from JIT compilation overhead, which is expected.

**Memory Usage:** A significant observation from Table III is the memory consumption. `μFlux.jl` exhibited considerably higher memory allocations (197.91 M allocations leading to 110.216 GiB total memory) compared to `Flux.jl` (64.30 M allocations leading to 72.969 GiB). This proves that the current implementation of `μFlux.jl` is creating more intermediate objects than `Flux.jl`. This is a critical area for future optimization and development, as high memory usage can be a limiting factor for larger models and datasets.

## REFERENCES


[1] Soeren Laue, "On the Equivalence of Automatic and Symbolic Differentiation" arXiv preprint 2019.

[2] K. Webb, Section 1: Roundoff and Truncation Errors, ESC 440 – Numerical Methods for Engineers, Oregon State University, 2024.

[3] Andreas Griewank, Andrea Walther, "Evaluating derivatives: principles and techniques of algorithmic differentiation," Society for industrial and applied mathematics, 2008.

[4] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, Jeffrey Mark Siskind, "Automatic differentiation in machine learning: a survey" CoRR, 2015.

[5] Joaquim R.R.A. Martins, "Aerodynamic design optimization: Challenges and perspectives," Computers & Fluids, 2022 .

[6] Liu, Feng and Li, Haipeng and Zou, Guangyuan and Li, Junlun, "Automatic Differentiation–Based Full Waveform Inversion with Flexible Workflows" arXiv, 2024.

[7] Geeraert, Sbastien and Lehalle, Charles-Albert and Pearlmutter, Barak and Pironneau, Olivier and Reghai, Adil, Mini-Symposium on Automatic Differentiation and Its Applications in the Financial Industry. Proceedings of the 8th International Conference on Algorithmic Differentiation, 2017.

[8] Hu, Yuanming and Liu, Jiancheng and Spielberg, Andrew and Tenenbaum, Joshua B. and Freeman, William T. and Wu, Jiajun and Rus, Daniela and Matusik, Wojciech, ChainQueen: A Real-Time Differentiable Physical Simulator for Soft Robotics. IEEE International Conference on Robotics and Automation (ICRA), 2019.

[9] Laurent Hascoët and Jean-Luc Bouchot and Shreyas Sunil Gaikwad and Sri Hari Krishna Narayanan and Jan Hückelheim, "Profiling checkpointing schedules in adjoint ST-AD", arXiv, 2024

[10] Jeff Bezanson, Stefan Karpinski, Viral B. Shah, Alan Edelman, "Why We Created Julia", https://julialang.org, 2012

[11] Jonathan Diegelman, NASA Launch Services Program and A.I. Solutions, "Modeling Spacecraft Separation Dynamics in Julia", https://www.youtube.com/watch?v=tQpqsmwlfY0, 2021

[12] Benjamin Krikler, Eduardo Rodrigues, Philippe Gras, "Julia for HEP Mini-workshop", https://indico.cern.ch/event/1074269/, 2021

[13] Suparna Dutt D'Cunha, "How A New Programming Language Created By Four Scientists Now Used By The World's Biggest Companies", https://www.forbes.com/sites/suparnadutt/2017/09/20/this-startup-created-a-new-programming-language-now-used-by-the-worlds-biggest-companies/, 2017