Implementation of Convolutional Neural Network with Automatic Differentiation library in Julia

1st Jan Walczak
Faculty of Electrical Engineering
Warsaw University of Technology
Warsaw, Poland
01168967@pw.edu.pl

2nd Patryk Jankowicz
Faculty of Electrical Engineering
Warsaw University of Technology
Warsaw, Poland
01168933@pw.edu.pl

***Abstract*—This paper presents a pure-Julia implementation of Convolution Neural Network built on top of a custom Automatic Differentiation library. The first part of the article provides a theoretical introduction to Automatic Differentiation topic and its modes, strategies for efficient implementation and finally motivations for choosing Julia as the implementation language. Second part details practical side of the project: the chosen approach – reverse mode of AD and structure of the developed solution. It is followed by challenges related to coding part and an outline of undertaken steps for improving memory footprint and runtime performance. In the final section the custom CNN solution is compared against established libraries, in Julia as well as Python, like Flux, PyTorch and TensorFlow on accuracy, time and memory efficiency.**

***Index Terms*—automatic differentiation, Julia, machine learning, convolutional neural networks (CNN), backpropagation, deep learning, neural networks**

I. Introduction and related work

A. Automatic Differentiation

Automatic differentiation (AD) is a technique for evaluating derivatives of functions implemented as computer programs. It works by decomposing a computation into elementary operations and systematically applying the chain rule to propagate derivatives [1]. Unlike symbolic differentiation, AD avoids generating large intermediate expressions, and unlike numerical methods, it produces exact results without relying on finite differences.

The main objective of this paper is to discuss and present an approach for implementing a Convolutional Neural Network (CNN) in Julia, with a focus on optimization and comparison to existing libraries such as PyTorch, TensorFlow, and Flux. The primary aspect emphasized in terms of resemblance is code optimization – specifically, time efficiency and memory allocation.

B. Modes of Automatic Differentiation

AD is typically performed in one of two modes: forward or reverse . Forward mode propagates derivatives from inputs to outputs and computes one directional derivative per pass. It is most efficient when the number of inputs is small [2]. Reverse mode works in the opposite direction, propagating gradients from outputs to inputs. This enables computation of full gradients for scalar-valued functions with respect to many inputs in a single backward pass [3], [4]. Reverse mode is especially suitable for gradient-based optimization in machine learning models with many parameters.

C. Efficient Implementation Strategies

The effectiveness of AD depends not only on the differentiation mode but also on how the system is implemented. Two common implementation strategies are operator overloading and source transformation [1]. Operator overloading tracks derivatives by redefining basic arithmetic operations on custom data types, allowing AD logic to run alongside normal program execution. This method integrates well with high-level languages and is relatively simple to implement. However, it may introduce runtime overhead and can be less efficient for deeply nested or large computational graphs [2]. Source transformation, in contrast, analyzes and rewrites program code before execution to generate derivative code directly. This approach can be more complex to implement, but it enables greater optimization opportunities, especially when combined with compiler infrastructure. It is particularly effective in managing control flow, recursion, and complex data structures, where performance matters [4], [7]. General strategies for improving AD performance include checkpointing to reduce memory usage in reverse mode, simplifying computation graphs, and reorganizing computations to minimize redundant evaluation [3], [5]. An example of this is the generalized approach to fast automatic differentiation, which introduces optimizations in memory layout and operation sequencing to speed up derivative calculations, particularly in large-scale problems [6]. Alternative formulations of AD that emphasize composability and algebraic structure have also been proposed to clarify the underlying principles and make AD systems more modular [7].

D. AD in Julia

Julia offers a flexible environment for implementing AD systems efficiently. Its support for multiple dispatch, metaprogramming, and just-in-time (JIT) compilation allows for high-performance differentiation without sacrificing code expressiveness. Forward-mode AD in Julia is often implemented using dual numbers and operator overloading, where each numeric type is extended to carry its derivative alongside the

value [2]. This enables efficient derivative computation with minimal changes to user code and integrates well with Julia's performance model. More advanced AD systems in Julia rely on source transformation techniques. These methods analyze function definitions and generate corresponding derivative code at compile-time or through runtime introspection [4]. Julia's metaprogramming capabilities and its intermediate representation (IR) make this type of transformation practical and performant.

II. Methodology and implementation

A. Description of the chosen approach

From the two approaches described in I-B the reverse one was chosen in the considered implementation in Julia. The decision was driven by its computational efficiency when the number of network parameters vastly exceeds the number of outputs — a hallmark of deep learning architectures like the implemented CNN. So the use of reverse-mode AD is instrumental in training of deep learning models, where number of trainable weights may reach millions.
To implement and execute core functionality of AD library the Computational Graphs were chosen. This approach not only allows to build such graph before training loop and then reuse it across minibatches (that way enhancing efficiency and reducing memory allocations), but also it is widely adopted architecture in this domain.

B. Structure of the solution

The architecture of the solution was divided into two main files `AD.jl` that is a core for the solution as it encapsulates Computational Graphs responsible for both the forward-pass as well as backpropagation part. Additionally, it overloads base mathematical operations for the use of graph, where each element is treated as a Node structure. For the high-level operations the `NETWORK.jl` was created, where all of the layer types (embedding, conv1d, maxpool...), chain, loss functions, optimizers (SGD, Adam), metrics are defined. The latter's operations and data handling are based on the functionality provided by the AD library – as functions referenced to it – that is why Automatic Differentiation is a core part of the implementation.
It is worth mentioning that every part of the solution was designed with a modularity in mind, so that entire architecture can be seen as a useful and compatible library.

C. Challenges related to the implementation

- The initial goal and main challenge determining the further approach to architecture and modularity of the solution was creating a library that would out-perform other established frameworks available in the market. Preparing a fully-fledged framework, like Flux, along with optimization and improvements were out of the physical scope of this project. This lead to focusing on single-type test scenario, CNN, with tailored optimization attitude.
- Choosing the right attitude and characteristic, involving completely different from basic loops solution, and applying it to optimize the functions responsible for convolutional layer operations was crucial. Without this step, the network either wouldn't learn effectively or each epoch would take significantly longer to complete.
- During work, the demanding part of the project turned out to be defining workarounds for mathematical operations used in CNN structures, like computing gradients for 4-dimension data.
- Focusing on minimizing memory allocations as well as incrementing new data dimensions by implementing mini-batching to ensure stability of training along with evaluation process.
- Aside from performance issues, at some point the network stopped learning after 2 or 3 epochs that was caused by static embeddings. The solution was imitating Flux's structure by treating this layer as a separate one, being updated by chosen optimizer within backpropagation pass, so that it could learn during the training process.
- Due to complex, multi-node structure it was a laborious task to debug any problems (like optimization bottlenecks) in final-connected network.

To summarize this chapter, many implementation challenges were addressed by adopting and replicating design patterns inspired by the Flux library. Aligning with conventions such as layer chaining (e.g., the `Chain` struct), modular gradient definitions, and optimizer abstractions ensured a consistent and extensible codebase.

D. Undertaken steps for optimization

The primary objective of this project was to develop a model that was, firstly – accurate, secondly – efficient. To achieve efficiency, a crucial step involved optimizing the code through several methods outlined below:

- Matrix operation vectorization for convolution using im2row1d transformations. This involves transforming each sliding window of the input sequence into rows of a 2D matrix so that the convolution can be performed as a single large matrix–matrix multiplication. Described approach not only leverages highly optimized linear-algebra routines (including BLAS) for both forward and backward passes but also improves cache locality and reduces interpreter overhead, leading to substantial speedups on longer sequences. Moreover, experiments with PyTorch framework showed, in case of dataset in text format conv1d operations outperformed conv2. This is due to fact that text data naturally lives in one dimension, as a sequence of token embeddings, so a 1D convolution aligns exactly with the structure of the inputs. The Conv2D operation inflates both the number of parameters and the amount of computation for no representational gain (because it would treat sequence as a height-weight grid).
- Thread-parallel execution for pooling operations.

- Across all layers and operations—dense weight tensors, convolution filters, activations, and gradients, the use of `Float32` precision was implemented in order to reduce memory footprint.
- Instead of processing each sample individually, multiple samples are grouped together into mini-batches that are subsequently fetched into training and evaluation operations.
- Across entire solution concrete types are used rather than undefined (::Any, abstract or union-typed fields). This allows compiler to generate highly optimized, inlined machine code without runtime type checks or dynamic dispatch, resulting in significant speedups.
- In-place operations – wherever possible, broadcasting update of existing variables rather than allocating new tensors. The `forward!` and `backward!` routines leverage in-place writes to reduce peak memory usage, e.g. accumulating gradients into pre-allocated `.grad` field.
- Memory reuse – instead of allocating new arrays, the `@views` macro were obtained allowing to avoid copying data by pointing into the existing array's memory.
- The `Trainer` constructor builds and topologically sorts just once the full computation graph using a dummy input of fixed batch size. During each mini-batch, this sorted graph is reused to avoid reconstructing the graph every iteration

III. Solution comparison to reference models

Important part of this project was the comparison of own implementation of CNN network based on custom Automatic Differentiation library to established frameworks. Tests were conducted in Flux (Julia), PyTorch and TensorFlow (both Python) – with each library the the identical network (in terms of architecture) was created. In every experiment the same dataset for training and evaluation was used and the measurements taken under consideration are:

- average time taken per epoch (first epoch with allocations separately and average time of epochs 2-5),
- average of total sum of memory allocation per epoch,
- accuracy for training and testing set reached after 5 epochs of training.

At this point it should be noted that due to the architecture of the TensorFlow framework, it was impossible to measure total allocations, so this value will be omitted in the final table. This is due to the way how TensorFlow works – code is executed in C++ language, so beyond the Python scope. A similar situation takes place with PyTorch where, due to call of `torch.profiler.profile` RAM usage immediately spikes so the results collected that way were not credible. In Julia allocations were check with `BenchmarkTools` module's macro `@allocated`.
Platform used for tests (using CPU without any GPU acceleration):

- AMD Ryzen 7 5800H @ 3.2GHz, 8 cores, 16 threads
- 32 GB DDR4 @ 3200MHz

All networks were built with nearly identical architecture consisting of:

Fig. 1. Architecture of CNN used for tests across all frameworks

Where $\|V\|$ – vocabulary size, d - dimensional vector space; in permuteDims (embedding_dim, sequence_length, batch).

TABLE I
Models comparison - Time performance

Model	First epoch time [s]	Average time [s]
Own	42,85	13,5525
Flux	43,79	14,82
PyTorch	3,47	3,8325
TensorFlow	4	2,67

*Time presented in the table above was measured in a separate run than memory computation.

TABLE II
Models comparison - Memory allocations

Model	First epoch allocations [MB]	Average allocations [MB]
Own	12102.34	10610.65
Flux	14683.87	11906.98

TABLE III
Models comparison - Reached accuracy

Model	Train accuracy	Test accuracy
Own	0,9517	0,869
Flux	0,9507	0,8676
PyTorch	0,9444	0,8641
TensorFlow	0,9482	0,8757

Table 1 shows that the pure-Julia implementation and Flux exhibit nearly identical first-epoch runtimes ($\approx$ 43 s) where all of the pre-allocations take place. In contrast, PyTorch (3.5 s) and TensorFlow (4 s) are an order of magnitude faster. Averaged over multiple epochs, custom model trains in 13.55 s/epoch, Flux in 14.82 s, PyTorch in 3.83 s, and TensorFlow in 2.67 s. This performance gap largely stems from the highly-tuned C/C++ and Fortran back-ends used by Python frameworks, whereas Julia relies on higher-level code. Nevertheless, it's worth noting that the custom implementation maintains a small but consistent advantage over Flux in average-epoch time.
In terms of memory allocations, the implementation allocates about 12 GB in the first epoch and 10.6 GB on average, versus Flux's 14.7 GB and 11.9 GB. This shows that in disputed implementation better memory allocation optimization steps were undertaken.
Despite these runtime and memory differences, all frameworks reach comparable accuracy levels (Table 3). Custom model achieves 95.17 % training and 86.9 % test accuracy, matching Flux (95.07 %/86.76 %), PyTorch (94.44 %/86.41 %), and

TensorFlow (94.82 %/87.57 %). The slight differences in results might be caused by pseudo-random weights initialization at the beginning of the training process.

IV. Summary

In this paper, is has been demonstrated that a pure-Julia implementation of a convolutional neural network, built from scratch on top of a custom automatic-differentiation (AD) library can achieve comparable accuracy, superior runtime results and memory efficiency compared to established frameworks.

As experiments conducted in III showed, created model reached almost identical training and test accuracy as Julia's Flux or Python's PyTorch and TensorFlow. All of the results are comparable and the slight differences fall within the expected variation due to random weight initialization. Testing with CPU, presented implementation and Flux both required $\approx$ 3x the time for the first epoch (due to pre-allocations) compared to the average for the rest of epochs. Both significantly slower than Python-based covered libraries. However, owing to laborious optimization process described in II-D, custom implementation allocates less memory than Flux library in first epoch as well as across entire training and evaluating process. Results like these are the best proof that there is still high ceiling for possible optimizations in Julia's implementation

In spite of the presented results the developed library has several limitations. Firstly, it does not leverage GPU acceleration, all of the operations are based only on CPU. Additionally, due to time constraints project work was focused only on necessary options, so despite modular and instructive approach, result cannot be called lawful library like already mentioned Flux.

Looking ahead, first steps taken would be the ones described in the previous paragraph. Apart from the optimization and acceleration field, the library should be developed in more universal direction, so that it could support wider range of neural networks and algorithms rather than just CNN and MLP. Despite significant efforts made to optimize the soulution there are still many already known possible improvements such as transformation of all `struct` constructs from mutable to static one which could yield notable results.

In summary, the outcome of the project is a successfully built custom CNN implementation on top of the Automatic Differentiation library. The final product match state-of-the-art libraries in terms of accuracy, with even better time results and more optimized memory allocations than direct rival in Julia – Flux.

References

[1] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: a survey," *Journal of Machine Learning Research*, vol. 18, no. 153, pp. 1–43, 2018.

[2] J. Revels, M. Lubin, and T. Papamarkou, "Forward-mode automatic differentiation in Julia," *arXiv preprint arXiv:1607.07892*, 2016.

[3] C. C. Margossian, "A review of automatic differentiation and its efficient implementation," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 9, no. 4, p. e1305, 2019.

[4] B. Van Merriënboer, O. Breuleux, A. Bergeron, and P. Lamblin, "Automatic differentiation in ML: Where we are and where we should be going," *Advances in Neural Information Processing Systems*, vol. 31, 2018.

[5] M. Bartholomew-Biggs, S. Brown, B. Christianson, and L. Dixon, "Automatic differentiation of algorithms," *Journal of Computational and Applied Mathematics*, vol. 124, no. 1–2, pp. 171–190, 2000.

[6] Y. G. Evtushenko and V. I. Zubov, "Generalized fast automatic differentiation technique," *Computational Mathematics and Mathematical Physics*, vol. 56, pp. 1819–1833, 2016.

[7] C. Elliott, "The simple essence of automatic differentiation," *Proceedings of the ACM on Programming Languages*, vol. 2, no. ICFP, pp. 1–29, 2018.

[8] Arun Verma, "An introduction to automatic differentiation", CURRENT SCIENCE, VOL. 78, NO. 7, 10 APRIL 2000

[9] Mario Lezcano-Casado, "Automatic Differentiation: Theory and Practice", arXiv 2207.06114, 2022

[10] Richard D. Neidinger, "Deep Learning", IT Press, chapter 6.5, 2016

[11] Ian Goodfellow, Yoshua Bengio, Aaron Courville, "A review of automatic differentiation and its efficient implementation", doi 10.1002/widm.1305, ISSN 1942-4795, 2019

[12] Justin Domke, Statistical Machine Learning – Automatic Differentiation and Neural Networks