Automatic Differentiation in Convolutional neural network.

Piotr Klepacki
Faculty of Electrical Engineering
Warsaw University of Technology
Warsaw, Polska
01159314@pw.edu.pl

I. Introduction

Derivatives are fundamental to the field of machine learning. Its forms, especially gradients and Hessians, are a key element in a learning process of neural networks. Automatic Differentiation (AD) is a family of techniques used for efficient and accurate computation of function derivatives. AD, in contrast to traditional numerical differentiation, provides the value of a derivative through the use of symbolic rules and tracking of variables. The backpropagation algorithm, a key element in training of neural networks, is based on a specific form of AD - Reverse-Mode Automatic Differentiation. It involves propagating derivatives backward from a given output, in this case the value of an output of the prior layer in a neural network. Reverse-Mode AD is noted as being particularly advantageous for modeling tasks characterized by high dimensionality, like Convolutional neural networks (CNNs), which helps in numerical optimization. While AD can be succesfully implemented in any programming language used for machine learning, for this task we choose Julia for our library, since it was designed with technical computing and high performance in mind from its inception. It combines the productivity of dynamic languages with the speed typically associated with static languages. Julia's generic functions and a flexible type system allows for creation of successful type inference, even without mandatory explicit type declarations, allowing the compiler to automatically specialize methods and generate efficient code. Julia's primary abstraction mechanism is dynamic multiple dispatch, which selects the most appropriate code method based on the types of all arguments, making it well-suited for mathematical and scientific programming. This design philosophy enables users to implement core functionality and libraries directly in Julia while maintaining high performance, effectively addressing the "two language problem" often found in technical computing. Julia's transparent data and performance models, aided by its sophisticated type system and inference, contribute to its suitability for demanding computational tasks. Given the critical role of Reverse-Mode AD in modern machine learning, and Julia's design for high-performance technical computing, the implementation of AD libraries in Julia seems like an obvious direction of ML research. However, scholarly works focused on the implementation of AD algorithms in Julia for machine learning applications is fairly limited, mainly to the forward-mode AD. Our work aims to fill in the gap that remains regarding the implementation Reverse-Mode AD in the language. We present the application of Automatic Differentiation within the context of Convolutional neural networks (CNNs), with a particular emphasis on optimized implementation using the Julia programming language.

II. Literature Review

Given the relative novelty of Julia as a programming language, coupled with its limited usage in machine learning applications, the available research pertaining to the focus of this paper is limited. In order to effectively address this gap, the literature connected to this topic has been divided into two categories: scholarly works describing automatic differentiation algorithms and works describing performance and usage of Julia.

A. Automatic Differentiation Algorithms

A comprehensive overview of Automatic Differentiation Algorithms employed in machine learning is presented in *Automatic Differentiation in Machine Learning: a Survey* [1]. The authors combine insights gained from the utilization of AD in other fields, such as atmospheric science or fluid dynamics, with its application in machine learning. Consequently, the paper encompasses both theoretical knowledge of these algorithms and their practical implementations.

Expanding upon this groundwork is *A Review of Automatic Differentiation and its Efficient Implementation* [2], which delves deeper into the performance of AD algorithms. Although the author primarily uses C++ for implementation purposes, the provided code snippets and recommended optimization strategies serve as valuable resources for implementation of AD algorithms in alternative programming languages. Furthermore, the paper points out the advantages of Reverse-Mode AD for modeling tasks characterized by high dimensionality, which is the case for this project.

A comprehensive resource on Reverse-Mode Automatic Differentiation is the publication *'The Stan Math Library: Reverse-Mode Automatic Differentiation in C++'* [3]. Although the library discussed in this work is also implemented in C++, it provides tips applicable to other languages such as Julia. Notably, the paper offers practical advice for memory

management during high-dimensional data operations and outlines various strategies for optimizing the AD algorithms.

B. Julia

The primary source of information regarding Julia programming language and its potential applications comes from its creators. In their paper, *Julia: A Fast Dynamic Language for Technical Computing* [4] , the authors delve into fundamental aspects of the language, including basics such as methods, functions and types. Subsequently, their later publication, *Julia: A Fresh Approach to Numerical Computing* [5] , contains more advanced topics. The authors provide insights into Julia's performance and offer various strategies for optimizing code, in domains such as linear algebra or parallel computing.

While there is no literature specifically adressing Reverse-Mode Automatic Differentiation in Julia, the authors of ForwardDiff package for Julia in *Forward-Mode Automatic Differentiation in Julia* [6] describe their implementation. Although this work primarily focuses on Forward-Mode AD, it offers practical guidance for optimizing Julia code performance and avoiding slowdowns related to memory management. While the algorithms differ, the paper's recommendations are applicable to both Forward and Reverse AD. Given the lack of scholarly articles strictly on the topic of Julia implementation of Reverse-Mode Automatic Differentiation, this paper could help to address this knowledge gap.

III. Library implementation

For the purposes of our library, we chose to implement Reverse-Mode AD using computational graphs. This approach allows us to take advantage of Julia‘s architecture, mainly multiple-dispatching, which is similar to overloading in object oriented programming languages. Since AD views any numeric computation as a composition of a finite set of elementary operations [4], our implementation had to be able to break down the target function into a set of elementary operations. The resulting graph has to directly reflect the composition of the function and be traversable to allow for computation of the target gradient.

A. Graph

Our computational graphs is composed of three fundamental node types:

- Constant contains predefined value.
- Variable contains value and gradient of the next equation.
- Operator contains a representation of the function operating on supplied argument values.

Operator node type is composed of two additional subtypes, ScalarOperator for scalar arguments and BroadcastedOperator for vector arguments. In addition to being suitable for Reverse-Mode AD, this structure also helps to visualize the dependency relations between intermediate variables. To enable the computation of gradient values using our graph, we used multiple-dispatching with basic arithmetic operations. Building on that, we were able to use the same mechanism for more complex functions like **diff()** which is used for calculating the gradient. By using the type of function stored in the operator node as an argument, we easily differentiated the implementation for multiple functions, from elementary operations to the computation of the entire neural network layer.

Before we walk trough the resulting graph, the nodes are sorted in a topological order, providing the framework for applying the chain rule.

B. Library structure

At the time of writing, the library is made up of 9 component files.

1) activation_functions.jl: Calculations of different activation functions.

2) computational-graph.jl: Basic structs and functions used for building the computational graph. Additionally all methods for calculation of layer gradients are declared there and new ones can be easily added.

3) data_loader.jl: Helper functions for loading the data.

4) gradient.jl: Functions for sorting and traversing the graph in both directions.

5) loss_functions.jl: Calculations of different loss functions.

6) network.jl: Structs and functions used for creation of various neural network layers.

7) optimisers.jl: Framework for addition of optimisations algorithms. Implementation of ADAM.

8) rng.jl: Helper functions for various randomization methods.

9) utils.jl: Various functions used for creation of neural network architecture.

IV. Difficulties and optimisation

The main problem we encuntered during the implementation of our library is paradoxically connected to the key strength of Julia - its optimisation potential. Julia offers the users greater control of the written program, which can both slow down and speed up the calculations, depending on the quality of the implementation. In the following subsections, we present what optimisation methods were used during the implementation of our library, as well as what obstacles were they intended to surpass.

A. Type declaration

Julia‘s type system is dynamic, which while being convinient to use, can also lead to drop in program performance. Fortunetly it also allows indicating that certain values are of specific types. This helps with efficent memory allocation but is a lot less flexible, requiring to handle each type independedly. To combine both the efficency of static typing with univesality of dynamic typing Julia introduced parametric abstract types. The structure defined with parametric abstract type can be seen as declaration of an unlimited number of types. During the initialization of the structure, defined type is then used for structure's remaining lifespan. In our implementation we use parametric abstract typing thought the whole library but mainly in computational-graph.jl to allow for creation of computational graph containing various data types.

B. Vectorized operations

Most of the calculations needed for computations of neural networks are performed on vectors and matrices, not scalar values. We designed our library aiming to avoid using costly and convoluted nested loops. To achive this goal all of the necessary Operators have two versions where possible.

- ScalarOperator for scalar values.
- BroadcastedOperator for vector values.

Additionally, we used another of Julia's features, vectorized operations. Adding a dot to the operations allows Julia to recognize the vectorized nature of the operation on a synthetic level. This helped us to save time and memory by combining multiple operations into a single loop.

C. Matrices splicing

By default Julia returns a copy of the spliced matrix, which can lead to unnecessary memory allocation, which is especially noticeable when operating on huge sets of data used for neural network training. To remedy this issue we used views wherever applicable. Views act like an array while actually being a reference to the fragment of the original array. This helps to avoid unnecessary memory allocation but as per Julia's documentation is not necessarily faster in all applications.

D. Indexing

To

V. Testing

To check the performance of our library, we tested the accuracy of the trained network, time required for all computations connected to model training and memory allocated during this time. To compare our implementations with state-of-the-art algorithms, we also performed the experiments for identical neural network using reference implementation in Julia's ML library Flux and Python's PyTorch.

A. Accuracy

The training accuracy was calculated as the mean accuracy for each epoch. The test results were computed after completion of each epoch.

1) Training: Both of the reference implementations achived similar training results, with final mean accuracy above 0.94. Our implementation converged a lot faster with the same accuracy in the first epoch and scoring almost 1.0 for the final one. While this could point to overfitting, test results does not show significant decrease in performance compared to the reference libraries.

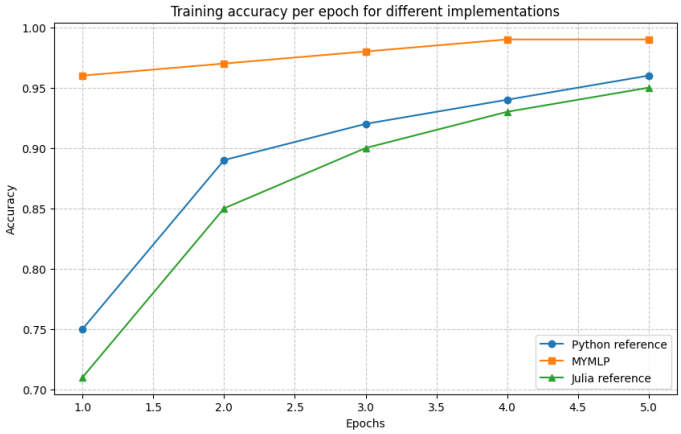

Fig. 1. Training accuracy for conv network

2) Testing: All three neural networks scored similarly in test accuracy. Python reference and MYMLP achieved their respective maximum performance before the final epoch which again points to slight overfitting which fortunately does not result in a significant decrease in their abilities.

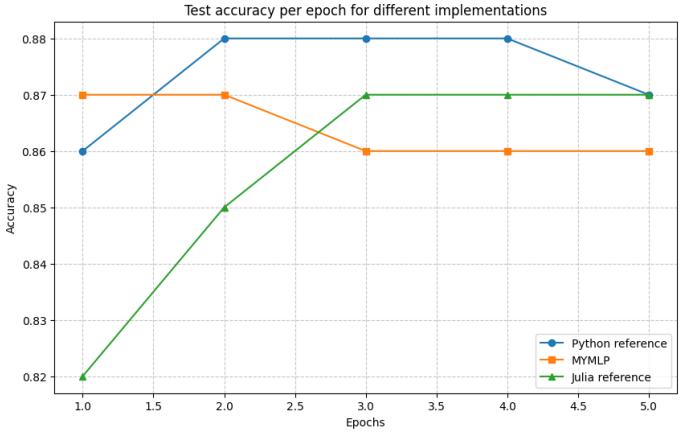

Fig. 2. Test accuracy for conv network

B. Computation time

The efficiency of our implementation was tested by measuring the total time needed for computation of 5 training epochs and calculations necessary for obtaining the results from earlier graphs.

1) Convolutional network: Our implementations proved to be the slowest of the three. The fastest was the PyTorch implementations, whose native functions written in C++ outperformed the Flux.jl package. To try and find the couse of the discrepancies between ours and reference computation times, we performed several tests. The most important clues came from tests of a neural network without convolutional layers.

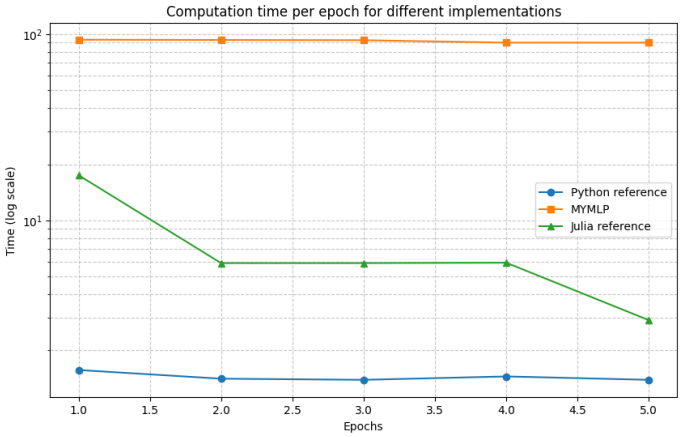

Fig. 3. Computation time for conv network

2) Dense network: The neural network with only dense layers achieved similar results to Flux.jl. This points to problems with our implementation of convolutional layer being the root issue for high computational time.

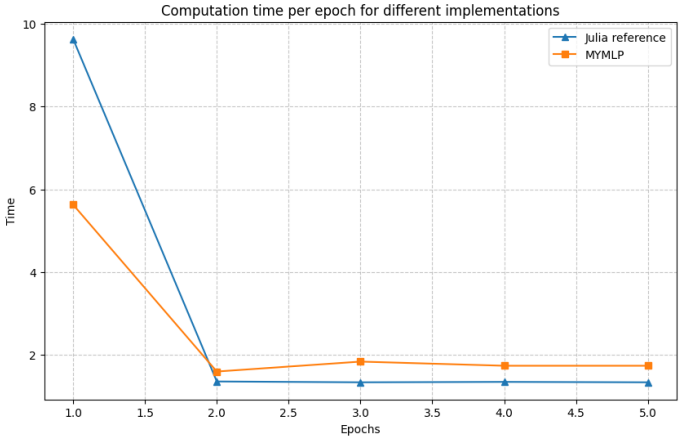

Fig. 4. Computation time for dense network

C. Memory allocation

We can draw similar conclusions from our testing of memory allocation. While both dense and convolutional network requires more allocated memory for our implementation, this is far more noticeable for the latter one.

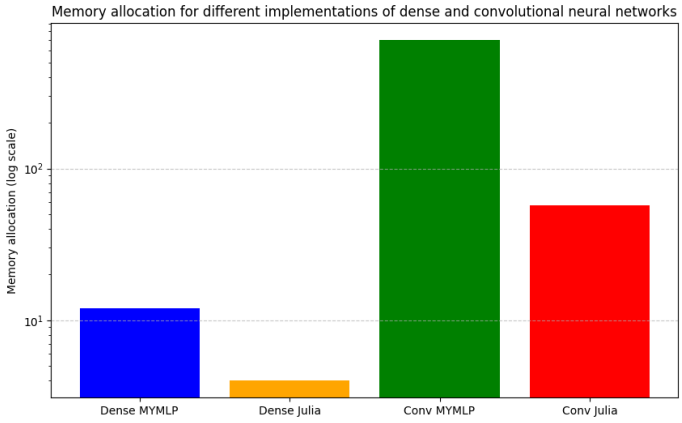

Fig. 5. Memory allocation comparison

VI. Conclusion

We present a library for Automatic Differentiation in neural networks The implementation was written in Julia and was based on computational graphs. Additionally the library was expanded to accommodate its usage in convolutional neural networks. Experiments were performed to measure the accuracy and effiency of resulting neural network and to compare it to state-of-the-art ML libraries - Flux for Julia nad PyTorch for Python. Our implementation achieved accuracy comparable to the reference networks but was outclassed in terms of memory and time efficency. Additional tests pointed to issues with our implementation of convolutional layers which could be the main culprit. Optimization of functions related to this part of our library is an obvious area for further work and should be the first step in further research. Additionally more work could be done to accomodate various other network layers as well as more optimisation and activation funcions.

References

[1] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, 'Automatic Differentiation in Machine Learning: a Surveyc'.
[2] C. C. Margossian, 'A review of automatic differentiation and its efficient implementation', WIREs Data Min & Knowl, vol. 9, no. 4, p. e1305, Jul. 2019, doi: 10.1002/widm.1305.
[3] B. Carpenter, M. D. Hoffman, M. A. Brubaker, D. D. Lee, P. Li, M. Betancourt, 'The Stan Math Library: Reverse-Mode Automatic Differentiation in C++' Apr 2022
[4] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, 'Julia: A Fast Dynamic Language for Technical Computing'
[5] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, 'Julia: A Fresh Approach to Numerical Computing'
[6] J. Revels, M. Lubin, and T. Papamarkou, 'Forward-Mode Automatic Differentiation in Julia'. arXiv, Jul. 26, 2016.