Efficient Automatic Differentiation Library for Convolutional Neural Networks in Julia Language

Mikołaj Guryn
Faculty of Electrical Engineering
Warsaw University of Technology
Warsaw, Poland
mikolaj.guryn@gmail.com

***Abstract*—This paper explores the integration of automatic differentiation (AD) algorithms within convolutional neural networks (CNNs), leveraging the computational advantages of the Julia programming language to address implementation challenges. By examining the current state of AD in machine learning, particularly in the efficient training of CNNs, alongside the numerical computing capabilities of Julia, we identify opportunities for optimization and enhanced performance. The literature review underscores the significance of AD in machine learning, the computational flexibility of Julia, and the potential of these technologies to revolutionize neural network training. This study aims to provide a comprehensive understanding of how Julia's features can be harnessed to improve AD's implementation in CNNs, contributing to the advancement of machine learning methodologies.**

***Index Terms*—CNN, Julia, Efficiency**

I. Introduction

Automatic differentiation (AD) has emerged as a cornerstone technique in the realm of machine learning, offering a systematic approach to calculating gradients, which are essential for the training of models such as convolutional neural networks (CNNs). The efficiency of AD, particularly in the context of complex neural architectures, is paramount for the rapid development and training of models. Concurrently, the Julia programming language has gained recognition for its high-performance capabilities in numerical and computational science, including machine learning applications. This paper delves into the intersection of AD and CNNs, exploring how Julia's unique features can address the computational challenges inherent in the effective implementation of AD algorithms. Through a literature review of seminal and current works, we chart the evolution of AD and its applications in machine learning, underscoring the role of Julia in fostering computational advancements.

II. Literature Review

In this article there were reviewed a variety of papers regarding automatic differentiation (AD), numerical computing and Julia programming language.

The first paper by Baydin et al. [1] highlights the critical role of AD in machine learning, offering insights into its application across a variety of algorithms and models. AD's ability to automate derivative calculations underpins the efficiency of neural network training, a fundamental process in the development of AI applications.

Bezanson et al. [2] introduced Julia as a strong programming language for mathematical computation. They showed Julia's features that are advantageous for the implementation of AD algorithms, emphasizing Julia superiority in computational field over MATLAB, Python and R.

Bücker et al. [3] delve into the applications, theory, and efficient implementation of AD. These works collectively underscore the importance of optimizing AD to overcome computational efficiency challenges.

The integration of AD within Julia, as discussed by Grøstad and Revels et al. [4], showcases the language's capability to enhance the implementation of AD in computational tasks. Specifically, Grøstad's application of AD in Julia for solving partial differential equations (PDEs) parallels the challenges encountered in CNN training, while Revels et al. focus on the advantages of forward-mode AD in Julia, underscoring its utility in optimizing machine learning workflows.

Finaly Schäfer er al. [5] introduces a Julia package that streamlines the integration of defferent automatic differentiation (AD) systems, reducing the complexity of implementing AD in computational projects. This feature is pivotal for developing an efficient automatic differentiation library for convolutional neural networks in Julia, as it allows for more robust and flexible handling of derivatives, essential for optimizing neural network training. Along with its appendices it provides robust benchmarks on presented AD packages.

III. State Of The Art

Automatic differentiation (AD) being a pivotal technology in the optimization of convolutional neural networks (CNNs), enabled the efficient calculation of gradients that are fundamental to training deep learning models. The state-of-the-art in AD for CNNs typically involves leveraging reverse-mode differentiation, commonly known as backpropagation, which is particularly efficient for functions with high-dimensional outputs and scalar loss functions. Recent advancements have focused on optimizing these computational pathways through memory reduction techniques, such as checkpointing, and exploiting parallelism in modern hardware architectures like GPUs and TPUs. Additionally, the development of software frameworks that support dynamic computation graphs, such as PyTorch and TensorFlow, has facilitated more flexible design and experimentation with novel CNN architectures.

These improvements not only enhance the performance but also reduce the computational cost associated with training complex models, thereby broadening the scope of problems that can be tackled using deep learning.

IV. Implementation

In the implementation at hand, automatic differentiation (AD) is utilized to effectively train convolutional neural networks (CNNs) on handwritten numbers dataset (MNIST Dataset) by calculating gradients necessary for the optimization of the network parameters during training. This specific implementation leverages reverse-mode AD, commonly known as backpropagation, across multiple layers of the network including convolutional, pooling, and dense layers.

A. Convolutional layer

The convolutional layers use spatial filters to capture features from input data, and gradients are calculated through a custom backpropagation routine that considers the effects of both weights and biases. This is particularly evident in how the gradients for convolutional layers are accumulated over each minibatch and then applied to update the weights and biases.

B. Maxpool layer

The pooling layers, specifically max pooling, are used to reduce spatial dimensions and control overfitting but do not involve learnable parameters; thus, they only propagate gradients without updating any weights.

C. Dense layer

The dense layers, which follow the flattening of pooled outputs, involve both forward propagation using matrix multiplication and a backward pass that computes gradients with respect to both weights and biases. These gradients are then used to update the parameters using a straightforward stochastic gradient descent method.

D. Gradients

Notably, the architecture ensures that gradient updates are normalized by the batch size, maintaining consistency in learning steps regardless of batch configuration. This normalization helps in stabilizing the learning process across different training configurations. The system is designed to handle updates at the end of processing each batch, which allows for the use of minibatch training methods effectively to optimize network training speed and memory usage.

E. Activations and evaluation

Additionally, the implementation supports modular activation functions, enabling easy experimentation with different types of nonlinearities in dense layers. The inclusion of testing against a separate data set provides a robust mechanism for evaluating model performance and guarding against overfitting, by periodically checking the accuracy and loss on data not seen during the training process.

F. Network handler

Getting it all together, in order to being able to tell that the presented implementation of a convolutional neural network (CNN) is a proper library and to facilitate a cohesive workflow between the layers, there was introduced a module called *NetworkHandler* which promotes modularity and maintainability. Each layer is equipped with its own forward and backward pass functions, following with mutable structs storing intermediate values and gradients. Each layer can be initialized according to the specific needs, then they should be organized into a tuple data structure to form the overall network architecture. Then the *NetworkHandler* can take this tuple and invoke sequentialy the forward and backward passes for each layer included, ensuring smooth propagation of data and gradients. At the end of each batch, the *NetworkHandler* calls and update method to adjust the weights stored in the mutable structs for all learnable layers, thus optimizing the network.

Additionaly there is provided an evaluation function, which follows similar procedure to assess the network's performance on a test set. This comprehensive approach ensures that the CNN implementation is not only flexible and user-friendly but also robust and scalable for various applications.

V. Optimization

Optimization not only refers to adjusting model parameters to minimize a loss function but also involves optimizing the computational efficiency, both in terms of time and memory allocation. This dual focus ensures that models are not only accurate but also run efficiently on available hardware, crucial for training large networks and managing resources effectively. After completing solution results were as follows in Fig. 1 and Table I.

TABLE I
Mean results of the given solution after 3 epochs

Parameter	Values
Time	263.4s
Allocated Memory	371.161 GiB
Accuracy Train	89.82%
Accuracy Test	90.78%
Test Loss	0.30766

The provided table highlights significant opportunities for optimization, particularly in time and memory usage, as evidenced by the 263.4 seconds of runtime and 371.161 GiB of allocated memory. These figures suggest that while achieving commendable train and test accuracies of *89.82%* and *90.78%* respectively, there is substantial scope to enhance the efficiency of the process. Optimizing these aspects could lead to faster training times and more economical use of memory without compromising on the quality of outcomes.

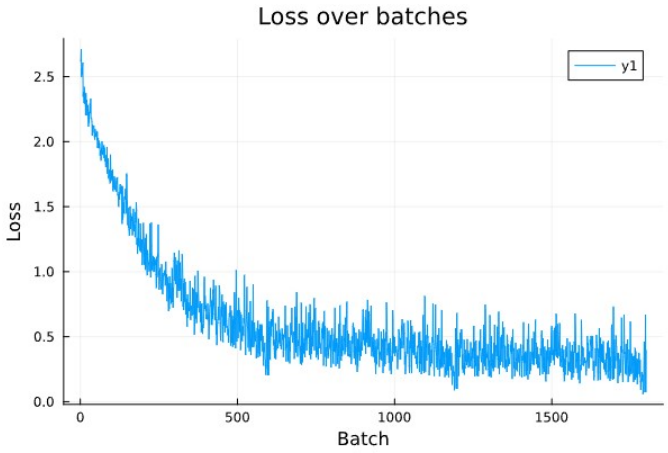

Fig. 1. Training loss over batches, step = 0.01

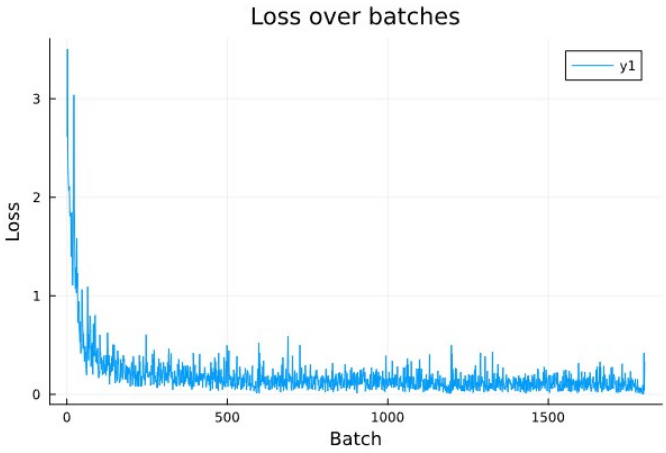

Fig. 2. Training loss over batches, step = 0.5

In the process of optimizing the convolutional neural network, various step sizes were tested to fine-tune the training performance, with 0.5 emerging as the optimal value, leading to significant improvements in learning speed and convergence rates. Additionally, a major enhancement involved transitioning from dynamic to static memory allocations, slicing matrices without copying them using Julia macro *@views* and using a tool to check methods allocation usage *@profview*. The last tool helped in identifying code lines that were the most burdensome for the program. It mostly highlighted the code with broadcast operations that should be changed to manual *for* loops which resulted in about *5%* memory allocation loss, places when multiplying matrices by default in Julia were not effective (about *10%*) time reduction) and the biggest change of all that took about *70%* of the memory allocation usage was localized in backward pass of the convolutional layer, where were a lot of unnecessary multiplications across whole matricex and not only on the certain parts of it.

These steps helped in reducing computational overhead and memory usage, also mentioned changes collectively streamlined the execution flow, decreased the runtime by *82.38%* and minimized memory footprint by *97.06%*, thereby enhancing the overall efficiency of the model training process. This approach not only optimized the training cycle but also improved the scalability and stability of the network during extensive training sessions. Optimized results are shown in Fig. 2 and Table II.

TABLE II
MEAN RESULTS OF THE OPTIMIZED SOLUTION AFTER 3 EPOCHS

Parameter	Values
Time	46.42s
Allocated Memory	10.917 GiB
Accuracy Train	96.87%
Accuracy Test	96.46%
Test Loss	0.11469

VI. REFERENCE SOLUTIONS

In the following section, we conduct a comparative analysis between the reference implementations and our custom solution for convolutional neural networks (CNNs). This comparison focuses on assessing the computational efficiency, memory utilization, and accuracy performance metrics across multiple test scenarios. By evaluating these key aspects, we aim to highlight the distinctive advantages and potential limitations of our approach relative to established benchmarks in the field.

A. TensorFlow Keras with Python

The first reference solution utilizes TensorFlow, a popular deep learning framework, to construct and train a convolutional neural network (CNN) on the MNIST dataset. This implementation leverages TensorFlow's high-level Keras API to streamline model construction, including layers such as Conv2D for convolutional operations and MaxPooling2D for downsampling. Furthermore, it employs a MemoryAndTimeCallback to monitor the memory usage and computational time per epoch, providing insights into the runtime efficiency and resource utilization of the model during training. Results for this implementation are shown in Table III.

TABLE III
MEAN RESULTS OF THE TENSORFLOW KERAS SOLUTION AFTER 3 EPOCHS

Parameter	Values
Time	1.70s
Allocated Memory	0.756 GiB
Accuracy Train	97.75%
Accuracy Test	98.19%
Test Loss	0.0723

B. Flux with Julia

The second reference solution utilizes the Flux.jl framework, a flexible deep learning library for Julia, to train a convolutional neural network on the MNIST dataset. The implementation leverages the built-in functionality of Flux to define a sequential model with convolutional, max pooling,

and dense layers. It employs a custom data loader to handle batch processing and utilizes Flux's gradient descent optimizer to update model weights efficiently. Accuracy and loss metrics are calculated directly on the full dataset post-training, providing a straightforward evaluation of model performance. Its results are shown in Table IV.

TABLE IV
MEAN RESULTS OF THE FLUX SOLUTION AFTER 3 EPOCHS

Parameter	Values
Time	11.73s
Allocated Memory	8.512 GiB
Accuracy Train	94.41%
Accuracy Test	94.79%
Test Loss	0.18542

VII. COMPARISON

This section will be dedicated to comparisons of the presented results.

A. Julia (optimized solution) versus Tensorflow Keras

Comparing presented optimized solution to Tensorflow Keras solution we can clearly observe the drastical difference in training time, where Python solution was approximately *27* times faster than our optimized solution and required about *10 GiB* less used memory allocations. Accuracy however is not much better than in out optimized solution, because it differs only by approximately *2%* so we can consider this a small success. Naturaly, the test loss in Python solution would be smaller than in our optimized one in Julia.

B. Julia (optimized solution) versus Julia Flux

When it comes to Julia (our) versus Julia (Flux) showdown we can observe that Flux implementation works about *4* times faster than our optimized implementation and uses about *2.4 GiB* memory allocations less. However the training and test accuracy along with test loss are better in out optimized solution than in this presented in Flux.

VIII. CONCLUSIONS

In this study, we integrated automatic differentiation (AD) algorithms within convolutional neural networks (CNNs), leveraging the computational capabilities of the Julia programming language. We implemented each layer of the CNN in custom module files, equipped with their own forward and backward pass functions, encapsulated in mutable structs for efficient data handling. The modular design allowed layers to be initialized according to specific requirements and organized into a tuple for seamless integration. To facilitate the workflow, a *NetworkHandler* module was introduced to manage the forward and backward passes and update the weights at the end of each batch, ensuring optimal training performance.

Our results highlighted the flexibility and potential of Julia for implementing AD in CNNs. However, when comparing our optimized solution to established frameworks like TensorFlow with Python and Flux with Julia, we observed significant disparities in performance. The TensorFlow Keras implementation demonstrated drastically faster training times and lower memory usage compared to our Julia-based solution, albeit with only a marginal improvement in accuracy. Similarly, the Flux implementation in Julia showed better efficiency than our custom solution, though it lagged behind TensorFlow in both speed and memory consumption.

Therefore, while Julia has shown promise for integrating AD in CNNs, our study reveals that it has not yet surpassed Python in terms of performance in this context. The current implementation might not represent the best possible use of Julia, and optimizing various aspects like memory allocation and computation could lead to better performance. This highlights the necessity for continued exploration and optimization of Julia-based implementations to fully exploit its computational strengths. Future work should focus on deeper optimization techniques and more extensive testing to determine if Julia can indeed outperform Python in machine learning tasks and validate Julia's purported advantages in machine learning applications.

All in all, our optimized implementation is subjectively satisfactory for detecting numbers on the MNIST dataset, achieving high accuracy with acceptable training time and memory consumption. Despite the noted performance disparities, our approach demonstrates that Julia can be effectively used for CNN training, providing a robust foundation for further enhancements and applications in machine learning.

REFERENCES

[1] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, 'Automatic Differentiation in Machine Learning: a Survey'.

[2] 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, Jan. 2017, doi: 10.1137/141000671.

[3] M. Bücker, G. Corliss, U. Naumann, P. Hovland, and B. Norris, Eds., Automatic Differentiation: Applications, Theory, and Implementations, vol. 50. in Lecture Notes in Computational Science and Engineering, vol. 50. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006. doi: 10.1007/3-540-28438-9.

[4] S. Grøstad, 'Automatic Differentiation in Julia with Applications to Numerical Solution of PDEs', Master's thesis in Applied Physics and Mathematics, Norwegian University of Science and Technology Faculty of Information Technology and Electrical Engineering Department of Mathematical Sciences, 2019. [Online].

[5] Frank Schäfer, Mohamed Tarek, Lyndon White, Chris Rackauckas, 'AbstractDifferentiation.jl: Backend-Agnostic Differentiable Programming in Julia' arXiv:2109.12449 https://ar5iv.labs.arxiv.org/html/2109.12449