Library for automatic differentiation, adapted for use in training convolutional neural networks

Jakub Jasiński
Faculty of Electrical Engineering
Warsaw University of Technology
Warsaw, Poland
jakub.jasinski2.stud@pw.edu.pl

***Abstract*—This article will present the implementation of a library for training a convolutional neural network in the Julia language with the use of a method called automatic differentiation. In the beginning, this method's theoretical background will be presented with an explanation of its benefits. Secondly, information about efficient implementation and optimising performance and memory management solutions will be provided. At last, there will be a test of how this solution compares to other already existing ones.**

***Index Terms*—Julia, reverse-mode automatic differentiation, convolutional neural networks, MNIST**

I. Introduction

The term neural network is nowadays widely known all around the world. With the ultra-fast pace of AI and machine learning development, we need more advanced and fast methods to train deep learning models efficiently. One of the challenges of AI may be image processing and classification. Convolutional neural networks that preserve relationships between image regions seem to be a good choice for this type of task. Accurate processing of many images may seem time-consuming and complex, so you need code that will make the work easier. The article will describe how these networks can learn and then present the results concerning existing solutions.

II. Related work

A. Julia programming language

Julia is a high-level, high-performance programming language specifically designed for technical computing, including mathematical and scientific computations [1]. Julia is fast because of careful language design and the combination of carefully chosen technologies that work very well together. It allows users to write clear, generic, abstract code that closely reassembles mathematical formulas yet produces fast, low-level machine code that has traditionally only been generated by static languages. Julia's carefully made libraries capitalize on the language's design principles, effectively leveraging features such as its expressive type system, multiple dispatch, metaprogramming for code generation and JIT compilation using the LLVM compiler framework to deliver high-performance implementations for various computational tasks.

B. Automatic differentiation algorithms

There are 4 main methods for calculating derivatives by computer systems: manual deviration, numerical differentiation, symbolic differentiation and automatic differentiation. As it turns out, the first three have many drawbacks, including too much complexity or not being resistant to rounding errors. On the other hand, automatic differentiation (AD) performs a non-standard interpretation of a given computer program by replacing the domain of the variables to incorporate derivative values and redefining the semantics of the operators to propagate derivatives per the chain rule of differential calculus [2]. AD employs two main algorithms for computing derivatives efficiently: forward mode and reverse mode. AD in the reverse accumulation mode corresponds to a generalized backpropagation algorithm, which propagates derivatives backwards from a given output. In the first phase, the original function code is run forward, intermediate variables are calculated, and the computational graph's dependencies are recorded through a bookkeeping procedure. In the second phase, derivatives are calculated by propagating adjoints in reverse, from the outputs to the inputs.

C. Effective implementation in convolutional neural networks

Surveys have shown that backpropagation is used in a majority of the real-world applications of artificial neural networks (ANNs) [3]. The Convolutional neural network (CNN) is a kind of feedforward neural network that can extract features from data with convolution structures. In contrast to conventional feature extraction techniques, CNNs eliminate the need for manual feature extraction. The architecture of CNN is inspired by visual perception [4]. Modern differentiation packages utilize various computational methodologies to enhance adaptability, execution speed, and memory utilization. Most important ways of obtaining that are operation overloading, region-based memory, and expression templates. Various mathematical techniques can yield high-performance gains when applied to complex algorithms. For instance, semi-analytical derivatives can reduce significantly the runtime required to solve and differentiate an algebraic equation numerically. [5]

III. Solution description

A. Operators overloading

We can distinguish two standards for implementing automatic differentiation, automatic code generation [6] and operator overloading [7], used in this solution. An important part of this implementation is the generation of the computational graph. There are 3 types of nodes in it:

1) Constant - has a defined value but does not store gradient information (input image, expected label)
2) Variable - stores information about its value and gradient result after using it (weights of kernels or dense layers)
3) Operator - represents a function operating on its arguments, resulting from its function and a gradient (dense operator, maxpool operator, etc.)

After defining the order of operations in the code, you must generate a computational graph. By defining the operators first, the code should recognize each node's appropriate inputs and outputs. Then, we perform topological sorting of the graph to obtain the target graph on which the calculations will be performed. Moving a graph forward involves iteratively traversing each vertex, starting from the first one. For each operator node, the value of the function obtained from the input is calculated and passed on. Moving the graph backwards is more complicated and time-consuming. Starting from the last vertex in the graph, we initialize its gradient with seed = 1.0. Then we iterate through the graph backwards, passing on the gradient. When we encounter an operator node, we calculate the gradient of a given function using the previously accumulated gradient. In the case of a variable node, we add this stored gradient to the gradient value that this node stores.

B. Network structure

The network consists of layers listed below:

1) Convolutional layer - With a 3x3x6 kernel, stride of 1, and no padding, a convolutional layer applied to a 28x28x1 input will produce a 26x26x6 feature map. Each channel represents different learned features from the input image. After convolution, the ReLU activation function will be applied.
2) Maxpool layer - The max-pooling layer will reduce the spatial dimensions by selecting the maximum value in each 2x2 region, resulting in output dimensions 13x13x6.
3) Flattening layer - This process reshapes the data from 13x13x6 into a single vector of length 1014.
4) Fully connected layer - It will compute a weighted sum of the inputs for each of its neurons, applying an activation function ReLU to generate the final outputs. The result will be a vector of length 84.
5) Fully connected layer - It will compute a weighted sum of the inputs for each of its neurons, applying an activation function Identity to generate the final outputs. The result will be a vector of length 10.
6) Loss counting - To compute the loss, typically, the logits are passed through a softmax activation function to obtain class probabilities. Then, the cross-entropy loss is calculated between these predicted probabilities and the actual labels.

C. Training process

During each epoch, the training process is repeated for each data sample. Training data is grouped into batches; one batch is processed during one training iteration, and the graph is traversed forward and backward for each input. After processing one batch, the weights are updated using the stochastic gradient descent algorithm [8]. The weights that are updated in this network are the weights of the kernel and fully connected layers. Before the learning process, the weights are initialized using a method called Xavier Glorot's initialization [9].

D. Dataset

To test the solution, the MNIST [10] dataset was used. It is a large collection of 70,000 28x28 pixel grayscale images of handwritten digits, used extensively in machine learning for training and testing models. It has become a standard benchmark for image processing systems. Its simplicity and accessibility make it an ideal dataset for researchers and educators to test and demonstrate machine learning algorithms. On Figure 1 are presented samples of data.

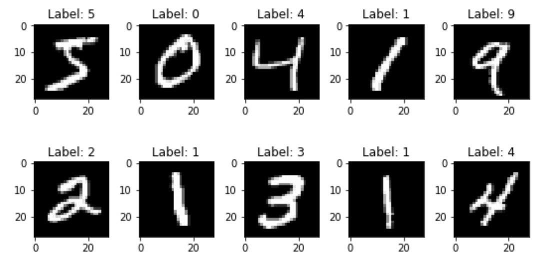

Fig. 1. MNIST dataset examples

IV. Optimatization

This section will describe the optimization steps to train and test the model faster, requiring less memory consumption.

A. Using Tullio macros

Counting value and gradient of the convolutional layer was performed with the help of Tullio [11] library. Using the @tullio macro in Julia can significantly improve speed and memory efficiency. Automatically fusing multiple element-wise operations into a single loop reduces the overhead of nested loops, leading to faster execution times, especially on large arrays. Additionally, @tullio minimizes the need for temporary memory allocations by operating directly on array elements, resulting in lower memory usage and more efficient memory access patterns.

B. *Using cache memory*

When calculating values in the maxpool layer, the coordinates where the maximum elements are located are calculated. When calculating the gradient, these coordinates would have to be recalculated. However, you can use the cache memory to save the values obtained during the forward pass and use them during the backward pass.

C. *Static typing*

Where it was possible, static typing was applied. Using static typing in Julia can improve performance by providing hints to the compiler for optimization and memory allocation [12]. It enhances code robustness by catching type-related errors at compile-time, leading to more predictable behaviour and easier debugging.

D. *Omitting unused values*

In the previously shown network architecture, you can omit the gradient calculation for the input image and expected label values. These values are constants, so calculating their gradient does not help train the network.

V. Accuracy and Performance Tests

The testing process consisted of running the entire training and testing process 10 times and measuring the correctness, loss value, running time, amount of allocated memory for each epoch and the entire process. The tests were performed for the previously presented solution and reference solutions in Flux [13] and Pytorch [14]. The experiments were processed on CPU 13th Gen Intel(R) Core(TM) i7-13700H.

We can distinguish the following parameters involved in learning:

- Number of epochs = 3
- Batchsize = 100
- Learing rate = 0.01

A. *Accuracy*

The accuracy results of individual solutions are presented in the Figure 2

Accuracy \ Solution	Julia	Flux	PyTorch
Epoch 1	88.01	89.58	87,08
Epoch 2	89.85	91.48	89.69
Epoch 3	90.81	92.92	90.43
Test data	91.79	93.21	91.20

Fig. 2. Accuracy results

The network training accuracy results for our library demonstrate the success of its implementation. The result above 90% itself can be considered a big step towards perfection. Overtaking PyTorch and almost catching up with Flux shows that the library works as intended when it comes to correctness.

B. *Execution time and memory*

The time-execution results of individual solutions are presented in Figure 3

Time [seconds] \ Solution	Julia	Flux	PyTorch
Epoch 1	43.05	22.17	9.36
Epoch 2	38.09	5.32	10.75
Epoch 3	36.03	5.50	9.40
Overall	117.17	32.99	29.55

Fig. 3. Time execution results

It looks worse when it comes to the speed of the program. Our solution works much slower compared to the reference solutions. The optimizations used were probably not enough to match existing methods of training the network. This is probably also related to less effective memory management. The memory allocation results of individual solutions are presented in Figure 4

Memory \ Solution	Julia	Flux	PyTorch
Epoch 1	61.99 GiB	6.66 GiB	27,20 MiB
Epoch 2	61.58 GiB	4.79 GiB	0,07 MiB
Epoch 3	61.58 GiB	4.79 GiB	0,04 MiB
Overall	185.15 GiB	16.24 GiB	27.31 MiB

Fig. 4. Memory allocation results

Our library significantly loses in the context of memory allocation. It can be noted that in each subsequent epoch it allocates almost the same amount of memory and the remaining solutions allocate less. Please note that the allocated memory in PyTorch is extremely small, this may be related to the difference between programming languages.

C. *Loss function minimalization*

It is also important to see how the loss function is minimized throughout the training phase. Comparison between these three methods can be seen in Figures 5 and 6

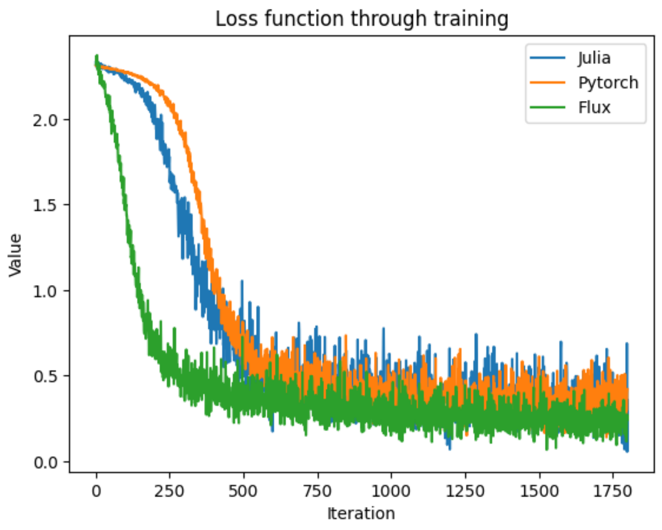

Fig. 5. Loss function through training

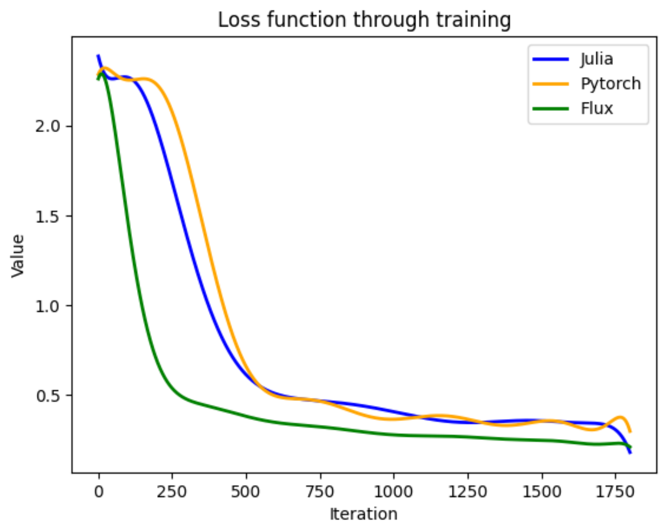

Fig. 6. Loss function through training, smoothed

As you can see, the library written in Julia does not stand out when it comes to minimizing the loss function. After all training iterations, the averaged results are comparable to the reference solutions. What is different is the larger differences between individual subsequent loss values compared to other solutions.

VI. Summary

The implementation of the library for automatic differentiation was successful. The achieved network learning coefficient is satisfactory and does not differ significantly from existing, recognized solutions in this area. There is a lot of room for improvement when it comes to runtime and memory allocations. In the future, we should take a closer look at the implementation of reference solutions and become more familiar with the optimizations offered by the Julia language.

References

[1] 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.
[2] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, ‘Automatic Differentiation in Machine Learning: a Survey’.
[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] Z. Li, F. Liu, W. Yang, S. Peng and J. Zhou, "A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects," in IEEE Transactions on Neural Networks and Learning Systems, vol. 33, no. 12, pp. 6999-7019, Dec. 2022, doi: 10.1109/TNNLS.2021.3084827.
[5] 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.
[6] Bart van Merriënboer, Alexander B Wiltschko, Dan Moldovan, Tangent: Automatic Differentiation Using Source Code Transformation in Python, 2017, url: https://arxiv.org/pdf/1711.02712. pdf
[7] Corliss, G F, and Griewank, A. Operator overloading as an enabling technology for automatic differentiation. United States: N. p., 1993. Web.
[8] Shun-ichi Amari, Backpropagation and stochastic gradient descent method, Neurocomputing, Volume 5, Issues 4–5, 1993, Pages 185-196, ISSN 0925-2312, https://doi.org/10.1016/0925-2312(93)90006-O.
[9] International Journal Of Engineering And Computer Science Volume 13 Issue 04 April 2024, Page No. 26115-26120 ISSN: 2319-7242 DOI: 10.18535/ijecs/v13i04.4809
[10] Li Deng The mnist database of handwritten digit images for machine learning research, IEEE Signal Processing Magazine, 29(6):141–142, 2012.
[11] Michael Abbott, Dilum Aluthge, N3N5, Vedant Puri, Chris Elrod, Simeon Schaub, Carlo Lucibello, Jishnu Bhattacharya, Johnny Chen, Kristoffer Carlsson, Maximilian Gelbrecht. (2023). mcabbott/Tullio.jl: v0.3.7 (v0.3.7). Zenodo. https://doi.org/10.5281/zenodo.10035615
[12] https://docs.julialang.org/en/v1/manual/performance-tips/, Online access 21/05/24
[13] https://fluxml.ai/Flux.jl/stable/, Online access 21/05/24
[14] https://pytorch.org/docs/stable/index.html, Online access 21/05/24