Fast implementation of an automatic backward differentiation in the Julia language

Jakub Maciejewski Michał Ziober
Faculty of Electrical Engineering
Warsaw University of Technology
Warsaw, Poland

***Abstract*—The work focuses on the creation of an efficient implementation of automatic backward differentiation in the Julia language, which is adapted to create highly computationally efficient code. Titled library was tested on the MNIST image classification problem using CNN architecture. Neural network was created on the basis of computional graph which is convenient solution in automatic differentiation. The main part of the article was devoted to the problem of the neural netowrk optimization. A solution was undergone a series of tests where accuracy, time and memory allocation performance were checked, where the results obtained were compared with reference solutions (Flux, Keras with TensorFlow).**

***Index Terms*—Automatic differentiation , Julia, CNN, MNIST, Keras, Flux.**

I. Introduction

THERE are four types of derivative counting techniques. These are: manual, numerical, symbolic and automatic. Each of them has its own advantages and disadvantages. The numerical technique is able to calculate the approximate value of the derivative quite quickly while it will not be an exact value. The symbolic technique always produces an exact value while it is very slow and memory-intensive. Automatic differentiation, on the other hand, obtains the exact value in a relatively short time and can be widely applicable. It is used in neural networks or numerical optimization, among other applications.

Automatic differentiation is based on counting very well known simple differentials, and then thanks to the graph of simple expressions built, the chain method is used, which guarantees us the correctness of the whole process. In automatic differentiation itself, we are able to distinguish two main modes: backward and forward. The backward mode calculates derivatives based on the result of a function, and then proceeds sequentially from the result to the input of the function over successive elements of the expression graph. The forward mode does it the other way around, respectively, that is, it walks through the elements of the graph from the function's input to its result.

The Julia language allows for a very fast and efficient implementation due to its extensive capabilities. In most of the tests carried out (simple micro benchmarks) across several different languages (including Fortran, Go, Python, R and Matlab) Julia performed the best, and its score itself was very close to that of an efficient C implementation.

Appropriate implementations and even libraries for calculating automatic differentiation in both forward and backward modes began to emerge. An example is the FLUX library. In addition, projects are being developed to combine an efficient implementation of automatic differentiation in Julia language with one of the most popular machine learning platforms i.e. PyTorch, which would allow PyTorch users to take advantage of a faster implementation, and Julia developers would increase the audience.

II. The problem

The problem which we chose to test our implementation of the library to automatic backward differentiation is image classification of MNIST dataset which contains hand-written digits. There are 60 000 train and 10 000 test grayscale low resolution(28x28 pixels) images.

III. Architecture of the created neural network

Created neural network is an CNN (Convolutional Neural Network) type, widely used in image classification problem. The most important part of the CNN is the convolution layer, which is based on the mathematical operation of convolution. It uses small filters (also known as kernels) that are independent of each other, to produce new result.

Our neural network contains 7 layers that are connected sequentially (as shown in figure 1). It is:

1) **Convolution layer** – input layer to the network.
   Contains 6 filters (of size 3x3x1), each producing new channel of layer output. Also bias is used and ReLU activation function.
   Dimensions of layer: 28x28x1 $\rightarrow$ 26x26x6
2) **Max pool layer** – Pool size 2x2. Max value of each pool is used in creating output of layer.
   Dimensions of layer: 26x26x6 $\rightarrow$ 13x13x6
3) **Convolution layer** – second convolution layer. Contains 6 filters (of size 3x3x6). Uses bias and ReLU activation function.
   Dimensions of layer: 13x13x6 $\rightarrow$ 11x11x16
4) **Max pool** – Pool size 2x2. Due to odd width and height size of input values at the extreme positions (right and down) are not taken into account.
   Dimensions of layer: 11x11x16 $\rightarrow$ 5x5x16

5) **Flatten layer** – Transforms input (three-dimensional array) into a vector.
   Dimensions of layer: 5x5x16 → 400x1
6) **Dense layer** – Connects every input neuron with every output neuron, which produces dense mesh of links. Uses bias and ReLU as activation function.
   Dimensions of layer: 400x1 → 84x1
7) **Dense layer** – Output network layer. Uses bias and Softmax as activation function
   Dimensions of layer: 84x1 → 10x1

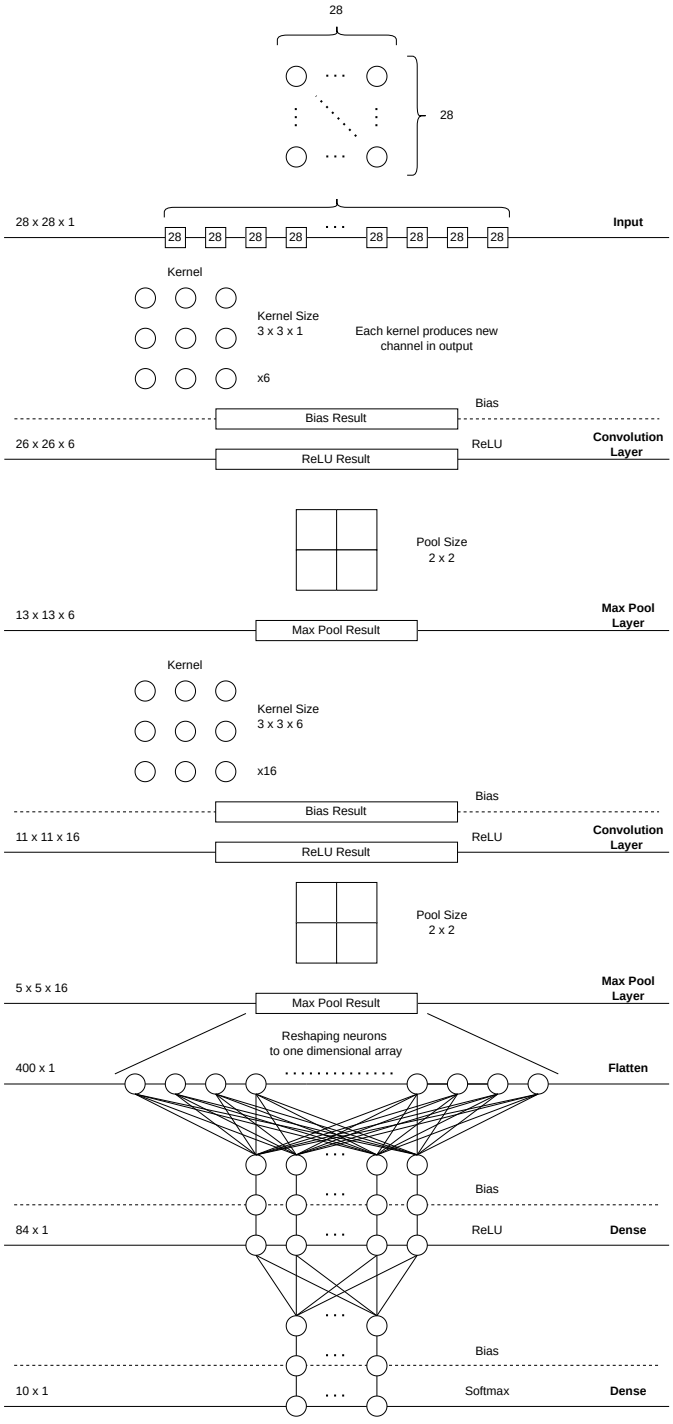

Fig. 1. Diagram of Neural Network Architecture

Output layer corresponds to solution of MNIST classification problem. Each output neuron refers to probability of classifying one of the ten classes.

A Mini-Batch Gradient Descent method was used to teach the neural network with specific parameters.

- **Weight update method** – Descent with 0.01 multiplier
- **Batch size** – 100
- **Epochs** – 5
- **Loss function** – cross entropy

IV. Implementation of a graph

The neural network was build on the basis of computational graph. Every element included in the neural network is of the GraphNode type. We have decided to define four types of nodes used in graph. It is:

- **Constant** ← GraphNode,
- **Variable** ← GraphNode,
- **ScalarOperator** ← Operator ← GraphNode,
- **BroadcastedOperator** ← Operator ← GraphNode,

where GraphNode and Operator are an abstract types.

Constant is the simplest node type and it is used as storage to constant values in graph. Variable is responsible for storing values, and their gradients during batch processing. Mainly all weights are keeped in Variable node. ScalarOperator and BroadcastedOperator are very similar and their common characteristic feature is the array of input nodes which are used to perform an operation. Operators have to implement both forward and backward method, which are needed respectively in forward and backward passes of the graph. Only difference between these two operators is returned type. The result of ScalarOperator is a scalar value, where for BroadcastedOperator it is an n-dimensional array. Operators are the core part, allowing to combine the graph into a whole.

The graph can be traversed forward and backward. Forward pass is an operation of classifying digit for given input image. Every GraphNode stores information about calculated result. Backward pass operation is used for calculating a gradient of each element in graph using chain rule. Gradient is later used to update weight values in the network after every batch.

V. Optimization

The first prototype of the solution was a non-optimal implementation. This section outlines the steps take to optimize it.

A. Smaller data types

Initially, all of the calculations were performed using Float64 data type. The decision has been made to change all of them to smaller type - Float32. It gave a big progress in reducing duration and memory allocation.

B. Strong typing

In the basic version the dynamic types were used in almost every fragment of code. In the optimized version, it was decided to move to strong typing. Dynamic one is more convenient for human use, but for the computer it is a problem that reflects on performance and memory allocation. This is because the compiler does not know what type variable is until the function is called. The situation is entirely different when the compiler knows type during compilation. Only needed amount of a memory is allocated.

In the project we provided variable types in almost every method definition to solve this problem. Some of them had to be universal and should take different input types. On a program-wide basis it is not big problem, because there are not a lot of such places.

It turned out that the biggest memory allocation gain was achieved when strong types were applied for graph's struct elements. Initially they were Any, but now they are:

- Variable:
  - `output::Array{Float32}`,
  - `gradient::Array{Float32}`.
- ScalarOperator:
  - `inputs::Vector{GraphNode}`,
  - `output::Float32`,
  - `gradient::Float32`.
- BroadcastedOperator:
  - `inputs::Vector{GraphNode}`,
  - `output::Array{Float32}`,
  - `gradient::Array{Float32}`,

C. Columns based iterations in multidimensional arrays

In Julia, arrays are stored by columns. Example:
array is defined like `[row_num,column_num]=elem: [1,1]=3, [1,2]=5; [2,1]=4, [2,2]=9.`
in the memory it is allocated like `[1,1][2,1][1,2][2,2] (3,4,5,9).`

If loops go by rows, there are huge amount of "memory jumps" (processor can't preload needed later data into cachememory) and it is not effective. We implemented going by columns in convolution and in a few smaller calculations. Initially, this optimization was not working (duration was 3 times longer). It turned out that only with strong typing optimization performance drastically increased.

D. Outputs pre-allocation

There are several layers and operators used in calculating both forward and backward (more expensive). Initially, in operators were allocationg output arrays inside forward/backward methods, filled with zeros and then values were assigned(or added) to them. That was incorrect approach because every graph pass operators allocated fresh memory for the array. Arrays used in convolutions were huge (among other in the project), so they were "hot places" where optimization needed to be done.

First approach was to define these arrays with "undef" types - following Julia tips for performance it is not recommended to define array with zeros when it is known this value will be reassign. Not every array could be changed with undef, but that was huge performance achievement.

It was noticed that even when above changes were performed, there was still a lot of allocations and program was slow when doing those calculations. That make sense, because there were still lots of allocations(in every forward/backward) in these places - now with "undef" values. Throughout life of the neural network, dimensions of forward results and backward results on layers are known(mentioned in section about architecture). Nothing stands in the way to allocate arrays for every step only once(whilst constructing network) and just pass them into calculations properly. It is one of the biggest improvement in the project and allowed to achieve even better performance.

Discussed approach was implemented not only in the biggest calculations(e.g. convolution, max pool), but in these smaller(e.g. math operations) too.

E. Array Slices

Initially, operations were mostly achived using broadcasted operators with arrays slices. Default Julia behavior is to create copy of that slice and then use it. This issue was at first resolved using @view macro, which allowed to use slices without creating copy, but it turned out later, that loops with proritization on columns were faster, and implementations with @view macro were replaced.

VI. Accuracy tests

On the Fig. 2. accuracy charts for tested implementations are presented. Average of 5 samples for each epoch.

As it could be seen, the best is Keras - its chart is always above the remaining two. Its accuracy in epoch 5 was 96.55%. Second is our implementation - 96.202%, and the third is Flux - 95.36%.

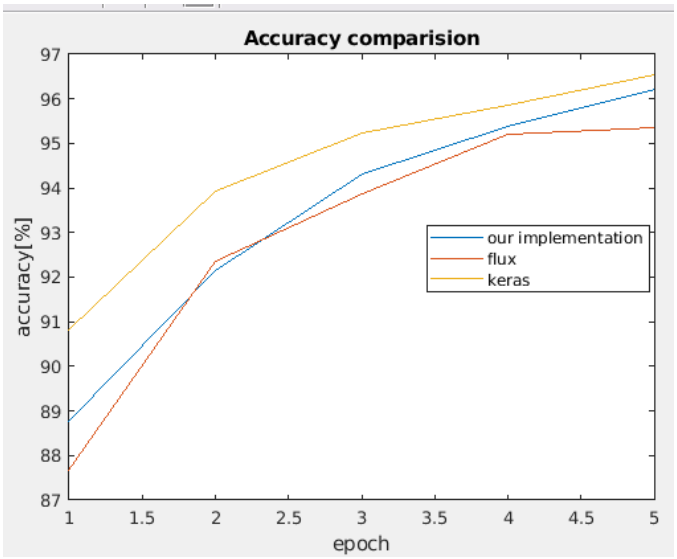

Fig. 2. Accuracy comparision per epoch for different implementations

We have also tried to teach create network with different but similar dataset. It was FashionMNIST, which contains 60000

low resolution grayscale images of clothes, divided into 10 categories. Given images were more complicated than number ones, so the neural network needed more time to learn, but eventually and accuracy 85% was achieved.

VII. Efficiency of the library

All the tests were performed on the same machine with operating system Ubuntu 22.04.3 LTS, processor AMD Ryzen 7 5800H with Radeon Graphics and also every implementation was provided with same parameters.

The project was compared to two reference implementations - one in Flux(Julia) and second in Keras with TensorFlow backend(Python). As for Flux, it was possible to compare both time and memory allocation, but with Keras there were some complications with memory. We didn't find a technique to follow memory allocation per epoch effectively in Python(where Keras was used). In python implementations we were able to obtain only current memory allocation for a process in opposite to Julia, where it is possible to track every allocation(sum of every allocated memory for operations performed) during lifetime of a program.

We decide not to attach Keras memory allocation results on "Allocation comparision"(Fig. 3.) chart because it would be unreliable. We attached Keras solution in time comparision chart(Fig. 4.).

We tested solutions with five epochs. Total allocation memory for flux is 41 576 GiB, our implementation: 7 442 GiB. We achieved almost six times better memory handling than reference solution developed in Flux. Unfortunately, our implementation is not as fast as reference implementation. Despite that, there is very huge progress when looking at version before optimalization - one epoch was processed about 40 minutes, where now it's only about 35 seconds.

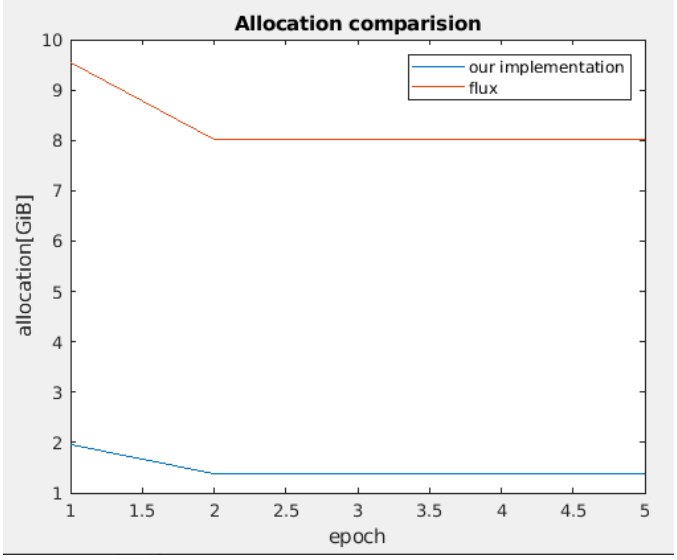

Fig. 3. Allocations comparison per epoch for Flux and our implementation

VIII. Conclusion

The goals of the project were achieved:

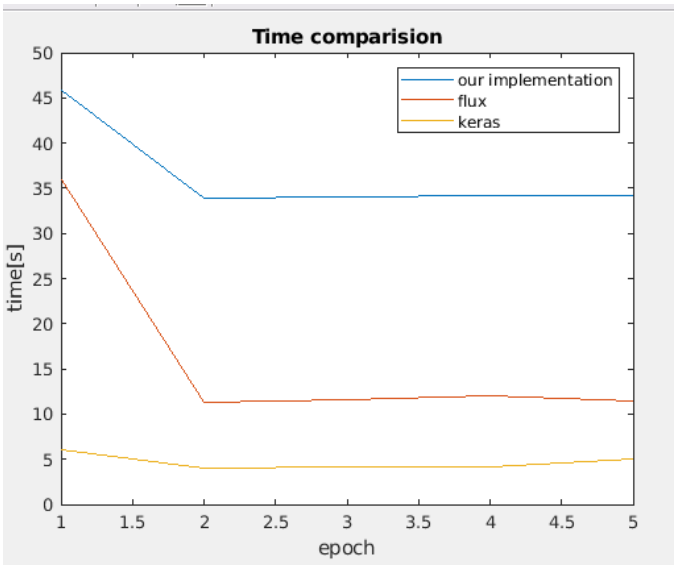

Fig. 4. Time comparison per epoch for Flux and our implementation

1) Titled library was developed.
2) Code was optimized.
3) Library is able to train network above 85% accuracy.

The result performance of the project is satisfactory. We believe there are still places in our code where performance could be improved. One of the things to improve time performance is parallel computing - there are some places in code, when it could be changed.

During optimization of the project, we have also tested computing convolution using Fast Fourier Transform using DSP.jl package. It turned out that this solution is not giving performance improvement, and even more memory was allocated, so it was decided to stop using it.

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.
[5] M. Innes, 'Flux: Elegant machine learning with Julia', JOSS, vol. 3, no. 25, p. 602, May 2018, doi: 10.21105/joss.00602.
[6] 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.
[7] J. Revels, M. Lubin, and T. Papamarkou, 'Forward-Mode Automatic Differentiation in Julia'. arXiv, Jul. 26, 2016. Accessed: Mar. 05, 2024.
[8] O'Malley, Daniel & E. Santos, Javier & Lubbers, Nicholas. (2022). Interlingual Automatic Differentiation: Software 2.0 between PyTorch and Julia.