Efficient automatic differentiation library in Julia for use in recurrent neural networks

Sebastian Górka
Student
Warsaw University of Technology
Warsaw, Poland
01159247@pw.edu.pl

***Abstract*— This article is discussion on high performance neural network training framework based on computational graph. To measure performance and accuracy capabilities of training neural networks with created library, it is compared with corresponding solutions implemented in Flux and TensorFlow. The MNIST problem with recurrent neural network is used in every solution as benchmark.**

***Index terms*—Automatic differentiation, Recurrent neural networks, Julia, High performance**

I. Introduction

There are numerous approaches to derivative calculation in computer programs, such as hard-coding derivative function, numerical methods, symbolic derivative calculation and finally – automatic (also algorithmic) differentiation (AD). First three approaches have significant drawbacks [1] – manually calculated derivatives are error prone, they are calculated slowly and are highly inflexible; numerical derivative calculation on the other hand could be inaccurate due to propagation of numerical errors; symbolic calculation is not able to calculate derivatives of some of computer functions (e.g. those including loops or conditional expressions). On contrary there is AD, which if carefully implemented, could solve major part of issues with the first three methods.

We can distinguish two types of AD: forward (FAD) and reverse mode (RAD). The latter will be discussed and used in this paper. RAD is based on using chain rule to calculate derivative of a function with respect to its output. It can be implemented by defining a computational graph [2], [3] – a directed graph, which nodes represent variables, constants, elementary arithmetic operations or at least operations, which derivative equations are known to the programmer. Edges between nodes are representation of data flow in the graph. For each operation there should be defined two functions: forward sweep function – how to calculate actual output of the operation and reverse sweep (derivative calculation) – function used to calculate derivative of operation implemented in particular node. Combining various nodes in proper sequence in the graph any complex function can be implemented, calculated in forward sweep and differentiated in reverse sweep.

Gradients calculated by the AD library will be used to train recurrent neural networks, specific instance of deep neural networks, commonly used for sequential data generation and for classification tasks.

This article is a discussion of high performance implementation of AD library to be used in training of RNN. Due to the fact that many neural networks (especially deep, as RNN) consist of millions of parameters to be optimized with methods such as gradient descent, an efficient implementation is needed as much as high performance programming language. Good choice would be Julia with its extensive support of linear algebra, operator overloading and focus on high performance vector/matrix operations. She is compiled to LLVM IR [4] and takes an advantage of its powerful middle-end and back-end optimization features [5].

There are numerous libraries out there implementing operations on neural networks, such as PyTorch [6], TensorFlow [7], or even in Julia environment: Flux [8]. The goal of this paper is to compare created solution with the solutions available on the market and possibly find some space for performance-related improvements. Benchmarks will be performed on the MNIST dataset [9], which consists of pictures of hand-written digits and is one of the most famous benchmark datasets in machine learning community.

II. The problem and the architecture

As mentioned above this paper will focus on solving MNIST problem. This dataset consists of 60 000 of training and 10 000 of test images of hand-written digits. The goal is to train the neural network that is able to tell, which digit is presented on the image.

The proposed architecture of neural network is following. Raw image of the size of 28x28 pixels is converted to vector with 784 elements. Then this vector serves as an input to the recurrent layer which consists of five vanilla RNN cells.

Each cell consumes 196-element vector and the state of recurrent layer and produces 64-element vector. Output of the last, fifth cell of recurrent layer is an input to fully connected layer that maps 64-dimensional vectors to 10-dimensional points. After processing those points by softmax layer, the index of the biggest value in vector is the predicted digit. Schematic graph of this architecture is visible on Figure 1

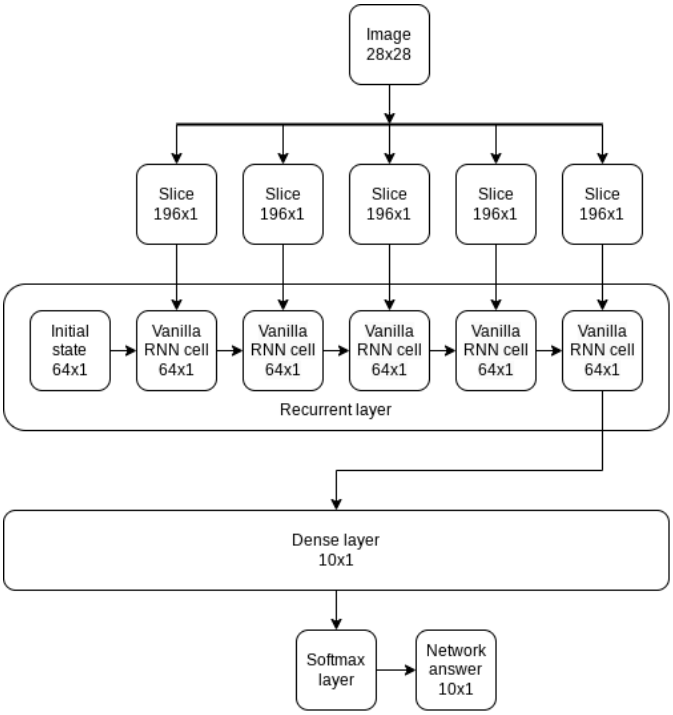

Figure 1: Proposed architecture of the neural network

In order to be able to measure the quality and performance of this library, learning parameters were imposed in advance. The optimization algorithm was established as simple gradient descent. Learning rate is set to $1.5e^{-2}$. Network has to train for five epochs in mini-batches of 100 data points.

III. The library

Whole library implementation is based on computational graph, which consists of parametrized nodes – structures deriving from abstract type `Node{T}`, where `T` is type of the numbers used in the graph. There are many various node sub-types defined in this library. Main purpose of this diversity was to distinguish kinds of nodes, such as constant (non-differentiable), operator (accepts inputs and transforms them into output; calculates partial derivatives with respect to these inputs) and variable.

It is possible to not care about the exact type of value in the node in Julia, but to avoid boxing of values, node subtypes are even more specific. They are either representing scalar or multi-dimensional constants, variables or operators.

This library is aimed to be used in context of training neural networks, so the most common layers, loss and activation functions were defined by the author. Layers are implemented as operator graph nodes. Operators have knowledge on how to calculate node output in forward sweep and how to calculate parial derivatives with respect to each input in order to calculate gradients in backward sweep. Thanks to that graph that represents complex function (such as artificial neural network) is able to calculate its result that depends on the input to the graph. Furthermore, graph is able to calculate derivatives of this complex function with respect to the nodes of interest. It enables this library to be used in neural networks training – partial derivatives can be used in various gradient descent optimization methods applied in the process of training.

To hide the process of orchestration of deep neural networks from graph primitives, this library introduces structures that represent layers of the network and the network that is composed of them.

IV. Performance

Performance of training neural networks is very important aspect of library created for that purpose. Training datasets very often are composed of millions of records and complex network architectures imply optimizing thousands or millions of parameters. There are already about 17 000 parameters in such a simple architecture as proposed in this paper.

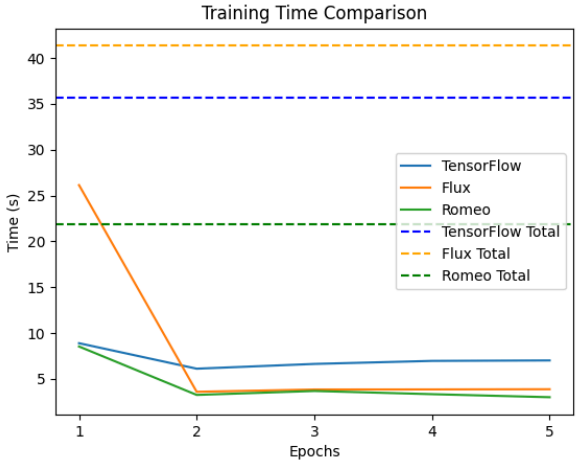

Figure 2: Comparison of execution time between reference solutions and Romeo library

To compare performance of this library two reference solutions were provided. One in Flux.jl and another one in TensorFlow. It is hard to compare memory related metrics between program created in Julia and program created in Python. Creators of these languages provided users with dif-

ferent profiling tools which take into account different metrics while calculating amount of allocated memory. Execution time is much easier to measure, so let's start with that. Comparison of time spent on each epoch by each solution is presented on Figure 2.

As we can see, Romeo is significantly faster than both Flux and TensorFlow. About two times faster than Flux in this case due to the fact that in the first epoch Flux compilation time is greater because Flux is generating gradient-calculating code.

Memory allocation information plot is visible on Figure 3. TensorFlow metrics are provided, but they should not be compared with solutions created in Julia, because their meaning is slightly different. To calculate allocated memory size in Flux and Romeo `@time` macro was used. It acquires its data from Julia garbage collector, so it provides information about each allocation. On the other hand `Process().memory_full_info().pss` used to calculate allocations in TensorFlow returns number of bytes assigned to the process, so it does not show total size of all allocations.

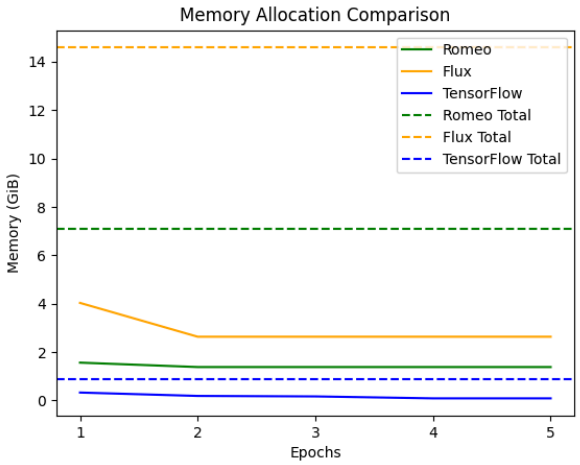

Figure 3: Comparison of memory allocation size between reference solutions and Romeo library

In both training time and allocation size comparison aspects Romeo is more efficient than both solutions (ignoring allocation info from TensorFlow). Romeo is designed to have minimal overhead. It avoids unnecessary computations and data transformations, which allows it to run faster and use less memory than other solutions.

Another aspect that contributes to the performance of Romeo is the use of strong typing in the computational graph. This allows the Julia compiler to generate highly efficient machine code for the operations in the graph. Each node in the computational graph has a specific type, which is determined at compile time. This means that the operations can be optimized based on the types of their inputs, leading to faster execution times.

Moreover, Romeo leverages Julia's broadcast function calls, which are a powerful tool for writing high-performance code. Broadcasting in Julia allows for efficient and concise element-wise operations on arrays. In the context of neural networks, this means that operations on large datasets can be performed quickly and efficiently, without the need for explicit loops. This is particularly beneficial when working with large training datasets, as it allows for efficient parallelization of computations.

V. Training results

Even more important aspect, in process of training neural networks, than peformance is accuracy of the model and its ability to minimize loss function as soon as possible. Comparison of reviewed solutions is presented on Figure 4 and Figure 5. In terms of accuracy, Romeo capabilities are very similar to reference solutions, although Romeo's loss descent is not satifying.

The slow convergence in minimizing the loss function suggests potential issues in the training process or the architecture of the Romeo model. It might be incorrectly selected gradient clipping theshold or some numerical instability in loss function calculation.

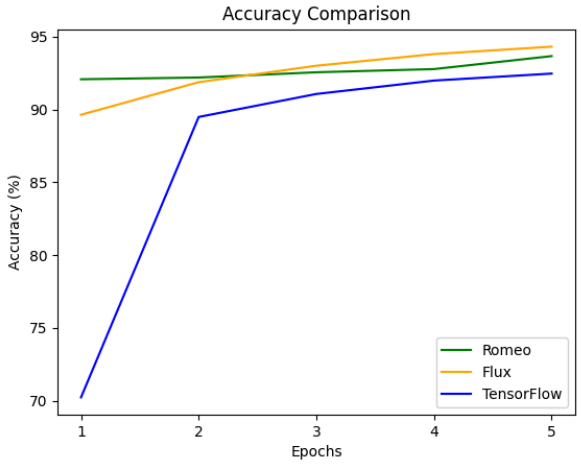

Figure 4: Comparison of accuracy between reference solutions and Romeo library

In contrast, the reference solutions exhibit a more rapid decrease in the loss function, indicating better optimization during training. This highlights the need for continuous evaluation and iterative improvements in the Romeo model to ensure it can compete effectively with these established solutions. Future work should focus on identifying and addressing the factors contributing to Romeo's slower loss descent.

By addressing these challenges, it is possible to improve Romeo's overall performance, ensuring it achieves higher accuracy.

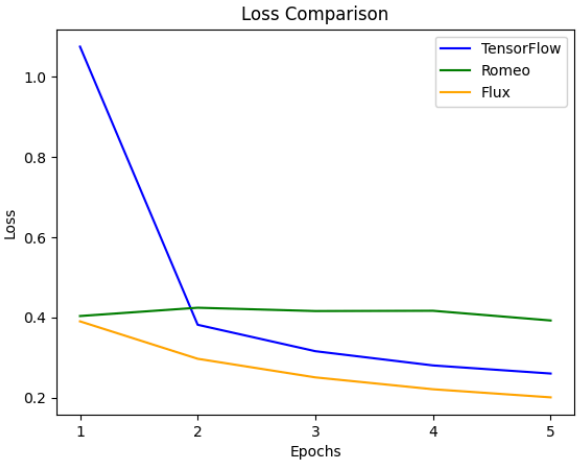

Figure 5: Comparison of loss function between reference solutions and Romeo library

VI. Conclusions

In this paper, we introduced a high-performance automatic differentiation (AD) library tailored for training recurrent neural networks (RNNs) in Julia. This library, referred to as Romeo, is built on a computational graph framework, enabling efficient reverse mode automatic differentiation. By leveraging Julia's strengths in linear algebra, operator overloading, and its powerful LLVM-based optimization features, Romeo aims to offer a performant alternative to established neural network frameworks like Flux and TensorFlow.

Our experimental results on the MNIST dataset demonstrate that Romeo can achieve significant performance gains in terms of execution time and memory efficiency. Specifically, Romeo is approximately twice as fast as Flux, primarily due to its reduced compilation overhead and optimized gradient calculation mechanisms. The library's use of strong typing and efficient broadcasting further enhances its performance by enabling the Julia compiler to generate highly efficient machine code.

However, while Romeo shows promising performance metrics, its training results indicate areas for improvement. The accuracy of the models trained with Romeo is comparable to those trained with Flux and TensorFlow. Yet, the convergence of the loss function is slower, suggesting potential issues in the training process or model architecture. This could be attributed to factors such as suboptimal gradient clipping thresholds or numerical instabilities in the loss function calculation.

Addressing these issues is crucial for ensuring Romeo's competitiveness with established solutions. Future work should focus on refining the training process, optimizing hyperparameters, and ensuring numerical stability. Additionally, continuous evaluation and iterative improvements are necessary to enhance Romeo's overall performance and achieve higher accuracy in model training.

References

[1] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic differentiation in machine learning: a survey." 2018.

[2] A. Savine, "Computation Graphs for AAD and Machine Learning Part I: Introduction to Computation Graphs and Automatic Differentiation," *Wilmott*, vol. 2019, no. 104, pp. 36–61, 2019, doi: https://doi.org/10.1002/wilm.10804.

[3] M. Abadi *et al.*, "TensorFlow: A System for Large-Scale Machine Learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA: USENIX Association, Nov. 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

[4] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A Fresh Approach to Numerical Computing," *SIAM Review*, vol. 59, no. 1, pp. 65–98, 2017, doi: 10.1137/141000671.

[5] C. Lattner, "Introduction to the LLVM Compiler System," in *ACAT 2008: Advanced Computing and Analysis Techniques in Physics Research*, Nov. 2008.

[6] A. Paszke *et al.*, "Automatic differentiation in PyTorch," 2017.

[7] Martín~Abadi *et al.*, " TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems." [Online]. Available: https://www.tensorflow.org/

[8] M. Innes, "Flux: Elegant Machine Learning with Julia," *Journal of Open Source Software*, 2018, doi: 10.21105/joss.00602.

[9] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.