Implementation of an automatic differentiation library for recurrent neural networks in Julia

1st Karol Kociołek
Faculty of Electrical Engineering
Warsaw University of Technology
Warsaw, Poland
karol.kociolek.stud@pw.edu.pl

***Abstract*—This project develops an automatic differentiation (AD) library for training recurrent neural networks (RNNs) using the Julia programming language, targeted at classification problem of popular database of handwritten digits - MNIST. The library utilizes reverse-mode AD with gradient accumulation to enhance computational efficiency. The implementation is compared with reference solutions in frameworks like Flux and PyTorch to evaluate its performance and effectiveness.**

I. Introduction

The implementation of automatic differentiation (AD) libraries is essential for machine learning. AD enables efficient computation of derivatives, which are crucial for optimizing neural networks. Recurrent Neural Networks (RNNs) benefit significantly from AD. This paper focuses on the development and implementation of an AD library specifically for RNNs using Julia, a high-performance programming language. Various papers cover subjects needed for implementation.

Automatic differentiation has been extensively reviewed in the context of machine learning by Baydin et al. [1], providing an in-depth survey of its theoretical foundations and practical implementations. AD is a set of techniques to evaluate the derivative of a function specified by a computer program. Furthermore, Margossian [5] offers a comprehensive review of AD and its efficient implementation, emphasizing techniques that enhance performance and accuracy.

Julia's relevance in numerical computing is highlighted by Bezanson et al. [2] showcasing its superiority over other languages like Python in terms of speed. Julia is particularly well-suited for AD due to its performance and flexibility. Grøstad [3] explores the practical application of AD in Julia. Additionally, Revels et al. [6] proposes AD method in julia, showcasing its characteristics as ideal for such tasks.

Training deep neural networks poses several challenges, including difficulty in initializing the network weights and managing the gradients. Glorot and Bengio [7] analyze these challenges, for example vanishing or exploding gradients.

RNNs are designed to handle sequential data by maintaining a hidden state that captures information from previous time steps. Schmidt [9] introduces RNNs and explains the adaptation of the backpropagation algorithm for RNNs, known as Backpropagation Through Time (BPTT). BPTT is crucial for training RNNs as it unfolds the network over time and propagates the error backward. However, RNNs suffer from the vanishing and exploding gradient problem, where gradients can become too small or too large, hindering effective training. Various approaches to mitigate this issue include using architectures like Long Short-Term Memory (LSTM) networks, which are discussed in depth by Hochreiter and Schmidhuber [11]. For a deeper dive into BPTT, see Agrawal [14], and for approaches to address the vanishing or exploding gradient problem and BPTT, refer to the tutorial by Britz [12]. Additionally, some papers propose alternative methods for RNNs and compare their performance on tasks such as the MNIST problem. For example, Shuai Li et al. [10] proposed the Independently Recurrent Neural Network.

Similar to implemented problem was discussed in master thesis by Nicolaisen [13] where he implements Automatic differentiation to find gradients for recurrent neural networks in Futhark.

For benchmarking purposes, implementation is compared against two established libraries: Flux in Julia, renowned for its elegant and efficient models as discussed by Innes [4]. PyTorch in Python, discussed by Paszke et al. [8] , is a widely-used library in the Python.

II. Classification dataset

The implementation centers around classifying the MNIST dataset, one of the most widely used datasets for training and testing image processing systems. MNIST comprises a large collection of handwritten digits, with 60,000 training images and 10,000 testing images, each of size 28x28 pixels. Examples from the dataset are shown in Figure 1.

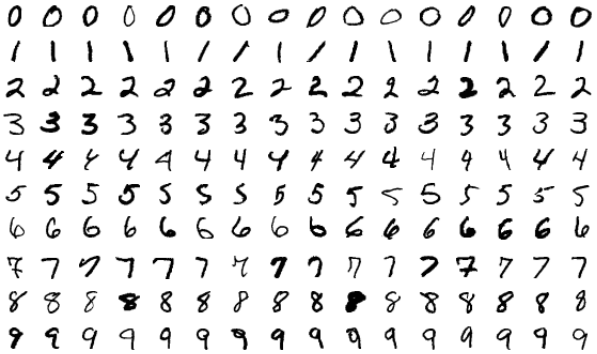

Fig. 1. MNIST dataset

III. Reference implementations

To achieve an optimal implementation and provide a necessary reference for working optimization, this implementation is based on an example executed in Julia using the Flux library. The results of this implementation are presented in the tables I and II.

For comparative benchmarking purposes, tests were conducted on a network with four vanilla recurrent cells with tanh activation fuction, featuring 196 inputs and 64 outputs, and one fully connected (dense) layer with identity activation with 64 inputs and 10 outputs (corresponding to the 10 classified digits). The settings included a batch size of 100, a learning rate (eta) of 15e-3, and 5 epochs. The primary metrics evaluated were:

- time
- memory allocations
- accuracy

Epoch	Accuracy
1	0.891
2	0.919
3	0.930
4	0.938
5	0.944

TABLE I
Training accuracy - Flux

Time	Allocations	Test accuracy
51,2 sec	14,534 GiB	0.944

TABLE II
Performance metrics - Flux

Additionally, an example was created in Python using the PyTorch library for a more detailed analysis of solution (without memory allocation analysys). The results are presented in the tables III and IV.

Epoch	Accuracy
1	0.639
2	0.829
3	0.884
4	0.906
5	0.918

TABLE III
Training accuracy - Pytorch

Time	Test accuracy
94 sec	0.92

TABLE IV
Performance metrics - Pytorch

IV. Implementation

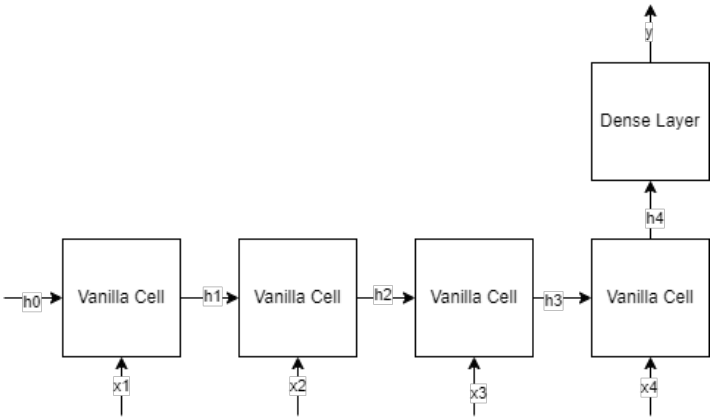

Fig. 2. RNN architecture

The implementation relies heavily on Julia's standard library. The network offers customization options for various parameters, including the input layer size (Wxh), the number of recurrent network cells (dependent on the input and input layer sizes), the hidden layer size (Whh), the output size (y), the number of epochs, the learning rate, the batch size and choice of activation functions (supporting tanh, identity and relu activations). The tested architecture, depicted in Figure 2, mirrors the one described in the "Reference Implementations" section. Additionally, a vanilla cell implementation, as illustrated in Figure 3, is provided. The architecture follows a many-to-one configuration.

Furthermore, the program incorporates operations for creating computational graphs and automatic differentiation, both in the forward and backward passes.

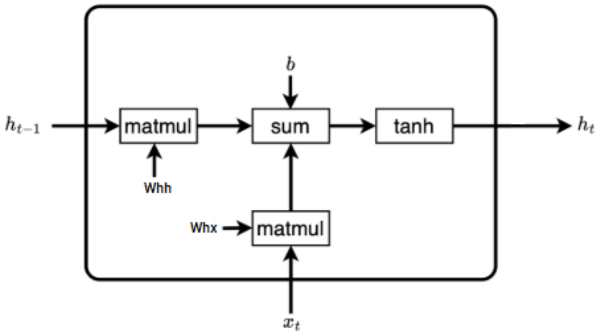

Fig. 3. Implemented vanilla cell

V. Optimization

The implementation also aimed to leverage Julia as an optimal language and to create an equally optimal library. Here are some examples of solutions implemented to ensure rapid execution and minimal memory usage:

- By explicitly declaring data types as for example float32 whenever feasible, rather than relying on float64 or Julia's dynamic type selection, memory allocation was nearly halved without sacrificing precision.
- In order to improve accuracy results, initialization weight solutions were analyzed. Xavier initialization allowed for a significant improvement in results.
- The @views macro was applied to minimize memory allocation by creating views of data instead of full copies.
- Using @time macro to investigate time and memory allocations.
- Preallocating memory for variables like xtrain, ytrain, xtest, and ytest instead of allocating memory in each iteration.
- Updating weights every few samples (batch size) instead of after each sample reduces the number of weight update operations.
- Utilizing topological sort to determine computation order in the computational graph.
- Use of Julia's broadcasting like for example .*, .+ etc. ensures that operations are applied element-wise across arrays efficiently.
- Enhancing the cross-entropy loss function by normalizing prediction results with softmax and using stable numerical computations, avoiding overflow issues in exponential calculations by subtracting the maximum beforehand.
- Resetting gradients and node computation results after each iteration allows reusing the same data structures without needing to reallocate them.
- Sparse matrices, such as onehotbatch, were employed to optimize memory usage and computation efficiency.

A. Computing environment

All tests were performed on the same computing environment with the following characteristics:

- Operating system: Microsoft Windows 10 Pro
- CPU:Intel(R) Core(TM) i7-6600U
- RAM: 16GB
- GPU: Intel(R) HD Graphics 520

VI. Results

After conducting a comprehensive analysis of the network's performance, it can be confidently stated that the implementation was successful. Although the achieved results may fall slightly short when compared to the reference solutions, it was still managed to attain a comparable order of magnitude regarding both time and memory allocation, alongside achieving similar levels of accuracy. For an in-depth examination of the results, please refer to tables V and VI. Additionally, the learning accuracy for each epoch is visually represented in figure 4, showcasing significant progress in the network's learning process over time.

Epoch	Accuracy
1	0.700
2	0.858
3	0.888
4	0.901
5	0.910

TABLE V
Training accuracy - Implementation

Time	Allocations	Test accuracy
135 sec	85 Gib	0.917

TABLE VI
Performance metrics - Implementation

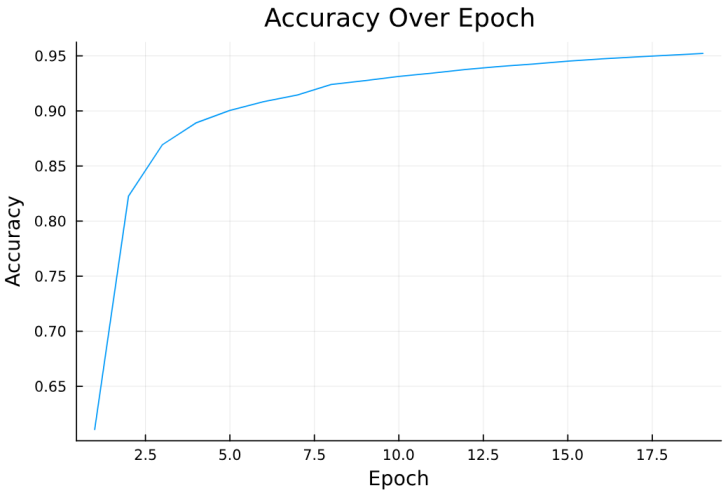

Fig. 4. Accuracy over epoch

VII. Conclusions

In this study, an automatic differentiation (AD) library tailored for training recurrent neural networks (RNNs) was developed using the Julia programming language. The primary focus was on addressing the classification problem posed by the MNIST dataset, a standard benchmark in image processing. Approach leveraged reverse-mode AD with gradient accumulation to enhance computational efficiency.

Comparative analysis was conducted with established frameworks such as Flux in Julia and PyTorch in Python to assess the performance and effectiveness of implementation. Despite achieving slightly worse results compared to the reference solutions, implementation demonstrated commendable performance, showcasing comparable levels of accuracy and efficiency in terms of time and memory allocation.

Implementation provided extensive customization options for various parameters, including input layer size, the number of recurrent network cells, hidden layer size, output size,

epochs, learning rate, batch size and activation functions. The architecture, designed in a many-to-one configuration, proved suitable.

Furthermore, optimization techniques were applied to ensure rapid implementation and minimal memory allocation. These techniques included among others declaring data types utilizing the float32 data type, implementing efficient weight initialization strategies use of stable numerical computaions, optimal gradient update using batches and optimizing memory allocation processes.

The results of analysis, presented in tables and visualized in figures, underscored the success of the implementation.

In summary, the developed library offers a significant opportunity to delve deeper into the intricacies of automatic differentiation and recurrent neural networks while showcasing the capabilities of the Julia programming language in efficiently implementing such solutions, enriching understanding of these fundamental concepts in machine learning and neural networks.

VIII. Ideas for futher studies

Further research related to this library could include:

- Creation of a more versatile library, including features such as the ability to add or subtract layers and support for architectures like many-to-many and others.
- Further optimization efforts aimed at achieving results surpassing those of the reference solutions.

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 Review*, vol. 59, no. 1, pp. 65-98, Jan. 2017, doi: 10.1137/141000671.

[3] 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]. Available: https://ntnuopen.ntnu.no/ntnu-xmlui/bitstream/handle/11250/2624618/no.ntnu:inspera:2497321.pdf

[4] M. Innes, "Flux: Elegant machine learning with Julia" *Journal of Open Source Software*, vol. 3, no. 25, p. 602, May 2018, doi: 10.21105/joss.00602.

[5] C. C. Margossian, "A review of automatic differentiation and its efficient implementation" *WIREs Data Mining and Knowledge Discovery*, vol. 9, no. 4, p. e1305, Jul. 2019, doi: 10.1002/widm.1305.

[6] J. Revels, M. Lubin, and T. Papamarkou, "Forward-Mode Automatic Differentiation in Julia" *arXiv*, Jul. 26, 2016. Accessed: Mar. 05, 2024. [Online]. Available: http://arxiv.org/abs/1607.07892

[7] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks" in *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, 2010.

[8] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library" in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds. Curran Associates, Inc., 2019, pp. 8024–8035.

[9] R. M. Schmidt, "Recurrent Neural Networks (RNNs): A Gentle Introduction and Overview," arXiv:1912.05911 [cs.LG], 2019.

[10] Shuai Li, Wanqing Li , Chris Cook , Ce Zhu , Yanbo Gao "Independently Recurrent Neural Network (IndRNN): Building A Longer and Deeper RNN"

[11] Sepp Hochreiter, Juurgen Schmidhuber, "LONG SHORT-TERM MEMORY"

[12] Denny Britz, "Recurrent Neural Networks Tutorial, Part 3 – Backpropagation Through Time and Vanishing Gradients", 2015, URL, https://dennybritz.com/posts/wildml/recurrent-neural-networkstutorial-art-3/

[13] Andreas Nicolaisen, "Using Automatic differentiation to find gradients for recurrent neural networks in Futhark"

[14] Aditya Agrawal, "Back Propagation in Recurrent Neural Networks", 2018, URL: https://www.adityaagrawal.net/blog/deep$_l earning/bprop_r nn$