# Performant and Accurate Automatic Differentiation In Julia for Recurrent Neural Networks

1st Michał Ankiersztajn
*Faculty Of Electrical Engineering*
*Warsaw University Of Technology*
Warsaw, Poland
michalankiersztajn@gmail.com

2nd Paweł Tęcza
*Faculty Of Electrical Engineering*
*Warsaw University Of Technology*
Warsaw, Poland
01159535@pw.edu.pl

***Abstract*—This paper explores ways of efficient automatic differentiation implementations integrated with recurrent neural networks in Julia. By researching the current state of automatic differentiation in neural networks and the potential of Julia in machine learning problems, especially in recurrent neural networks, improvement opportunities regarding efficiency and accuracy in automatic differentiation models were identified. This study aims to show how to improve automatic differentiation with Julia to advance recurrent neural network usage.**

***Index Terms*—Recurrent Neural Networks, Julia, Automatic Differentiation**

## I. Introduction

Automatic Differentiation (AD) is the bottom-line technique in the modern world of machine learning. It offers a flexible and benchmarkable way of calculating gradients, an essential part of recurrence neural networks. Julia has gained recognition as a programming language that can efficiently solve numerical and neural network problems. This paper dives into efficient Automatic Differentiation with the Julia programming language.

## II. Literature Overview

This article is based on various papers on automatic differentiation, numerical computing, and Julia programming language.

Book by Grøstad and Revels [1] has the required automatic differentiation theory. Moreover, in the book, an introduction to Julia programming language can be found, including its characteristics and more advanced concepts such as metaprogramming. Grøstad and Revels showcase examples of automatic differentiation within Julia with ways to optimise the code.

A paper by B. van den Berg, T. Schrijvers, J. McKinna and A. Vandenbroucke [2] showcases reverse-mode automatic differentiation along with how it differs from forward-mode automatic differentiation. An implementation of complexity $O(N \cdot \log V)$ for reverse-mode automatic differentiation can be found. On top of all that, the paper explains why reverse mode works better when there are many variables, and forward mode is preferable for a few variables.

A paper by H. Salehinejad, S. Sankar, J. Barfett, E. Colak, and S. Valaee [3] presents recurrent neural network architecture along with the history of the most significant advances within them, such as architecture, activation, and loss functions and much more. The paper showcases how the recurrent neural network has improved over the years.

A paper by R. Pascanu, C. Gulcehre, K. Cho, and Y. Bengio [4] explores different ways of extending and creating recurrent neural networks to achieve deeper learning. It highly focuses on architecture and improvements to it. Furthermore, it reveals and shows how to outperform conventional recurrent neural networks.

## III. Automatic Differentiation

Automatic differentiation utilises calculus's chain rule to effectively and precisely determine derivatives. It achieves this by decomposing functions into a sequence of simple operations and fundamental functions, each of which has a known derivative [6]. There are two main concepts of implementing considered technique. Forward automatic differentiation (forward mode) and backward automatic differentiation (reverse mode).

Both mentioned variations of AD use a computational graph [5]. The computational graph is a directed acyclic graph representing the sequence of operations and intermediate variables in evaluating a function. Each node in the graph corresponds to an operation or a variable, and the edges indicate the flow of data between these operations, allowing the efficient calculation of derivatives through forward and backward passes.

## IV. Recurrent Neural Networks

Recurrent Neural Networks (RNNs) represent a class of artificial neural networks adept at processing data sequences by maintaining a form of memory that captures information from previous inputs. This unique characteristic enables RNNs to excel in tasks where context and temporal dynamics are crucial, such as natural language processing, time-series forecasting, and speech recognition.

RNNs require a specific approach to calculate derivatives automatically in reverse mode. This approach is called Backpropagation Through Time (BPTT) [7]. It is an extension of the backpropagation algorithm used to train recurrent neural networks, which helps to handle the temporal nature of RNNs, enabling them to learn from sequential data by unrolling

the network across time. It is strongly related to one of the most significant issues with training simple recurrent neural networks.

The exploding and vanishing gradient problems are critical challenges in learning recurrent neural networks. These issues arise due to the nature of backpropagation through time, significantly impacting the network's ability to learn from long-term dependencies in sequential data.

## V. Implementation

Considered RNN implementation uses computational graph for the sake of automatic differentiation. This approach is a flexible solution that simplifies the process of orchestrating the model. The main goal of implementation is to achieve an efficient and generic recurrent neural network acting as a classifier.

Network architecture is relatively simple. It consists of one unrollable recurrent layer and one dense layer.

### A. Recurrent layer

The reference layer provides the ability to include backpropagation through time. It concerns two main arguments: current (t) data input and previous (t-1) hidden state. It provides an output state, which is used in the next iteration. Unrolling this layer results in the structure shown in Fig. 1.

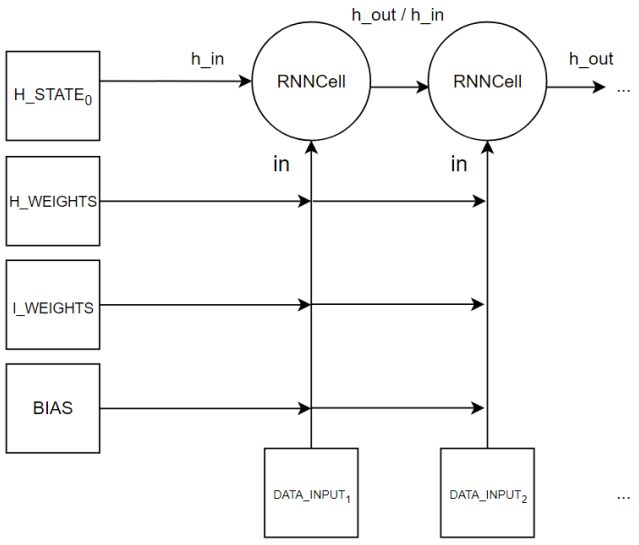


Fig. 1. Unrolled recurrent layer.

Every recurrent cell (received via the unrolling layer) is added to the computational graph, tracking every intermediate result and corresponding gradients. The number of timestamps considered is an end-user decision strictly correlated with the number of recurrent cells. Three optimisable parameters are related to this layer – input weights, hidden weights, and recurrent biases.

### B. Dense layer

The dense layer acts as a proxy, applying a linear transformation to achieve the required features. Layer and associated computations are inserted into the computational graph. Two optimisable parameters are related to this layer – input weights and biases.

### C. Loss function

The computational graph ends with a loss function node. This node describes the correctness of network prediction in the forward pass and the gradient optimisation during backpropagation.

The described solution calculates loss based on probabilities vector – dense layer's output is passed into the cross entropy loss function using numerically stable Softmax as due to the explosive nature of the RNN gradients, the model will likely end up with matrices of NaNs.

### D. Weights initialization

Finding proper initial weights is crucial in avoiding the exploding gradient problem. The chosen approach is Xavier's weight initialisation, also known as Glorot initialization [9].

Without proper initialisation, the variance of activations can grow or shrink exponentially with each layer, leading to the exploding or vanishing gradient problem. By maintaining a balanced variance, Xavier's initialisation helps to stabilise the training process.

### E. Learning algorithm

Implementation's network learning algorithm is stochastic gradient descent [10], which is described as simple, performant and effective. The speed of gradient descension can be modified through the alfa parameter.

## VI. Optimization

This paper's optimisation refers to the allocated memory and time required to train the model. It ensures that the model is flexible, can work with larger datasets with more parameters and is still completed in an acceptable time on most computers.

### A. Stable types

To limit memory allocations and increase calculation speed, the model should operate on primitive types. This AD library thoroughly worked on Float32s, Julia's lowest possible float type, which is enough to complete the task without meaningful loss in accuracy. However, depending on needs, the type can be changed to other types, such as Float64, Int and custom abstract types. Primitive types are crucial for performance and memory reasons as otherwise, Julia uses Boxing, which creates additional pointers that we need to allocate and access.

### B. Caching

A technique called 'caching' has been used to create everything only once, if possible. This way, garbage collecting is limited, time spent creating, and the amount of memory used is reduced. It is being used all over the program; such an example is resetting the state; instead of creating an extensive matrix of zeros a couple of thousands of times, a single matrix of zeros is created and used to reset the state in every batch.

### C. Dot notation

In Julia, it is possible to use 'dot notation', which tells Julia to complete vector and matrix calculations in the same loop at the parser level. It has been used extensively in forward and backward functions in the graph. It not only speeds up our program but also reduces memory allocations as everything is done in the same loop; thus, there is no need to allocate memory for new vectors or matrices.

Moreover, it simplifies managing the size of input data. With 'dot notation', the function can be written as if it were for a single value x, but it works for matrixes and vectors.

### D. Static Arrays and Matrices

Both static arrays and matrices were not used in the project's final version because of their numerous limitations in the current state. It's impossible to perform some of the most basic operations between a matrix and a static matrix, thus making this feature unusable and irrelevant.

### E. Mutability

The use of mutable data structures is often questioned as they lead to unreadable and harder-to-understand code. However, in the case of MLP optimisation, it is a tradeoff worth it. The graph is built once and used throughout the program multiple times. This optimisation alone speeds up the program by around 10x and reduces memory usage by around 20 GiB.

## VII. Results

The considered solution was tested in a classification problem. A well-known MNIST image dataset was used to train and test the network.

For reference and comparison, other solution results are provided as well. Three main metrics considered are memory allocation during training, time of training and accuracy of test data predictions.

Network and learning parameters (such as learning rate or batch size) are equal across all solutions.

### A. Implementation

The implementation results are pretty promising in terms of speed and memory allocation. All mentioned optimisations saved much of the time needed for learning. Fig. 2. visualises the learning process from a loss value perspective, whereas Fig. 3. shows related accuracy.

The training consisted of 5 epochs, each containing 600 batches of 100 pictures. Each epoch works on randomly picked batch data permutations, which helps the network maintain its generalisation ability. Mean loss value gradually decreases and – as anticipated – decrease slows down with several processed batches.

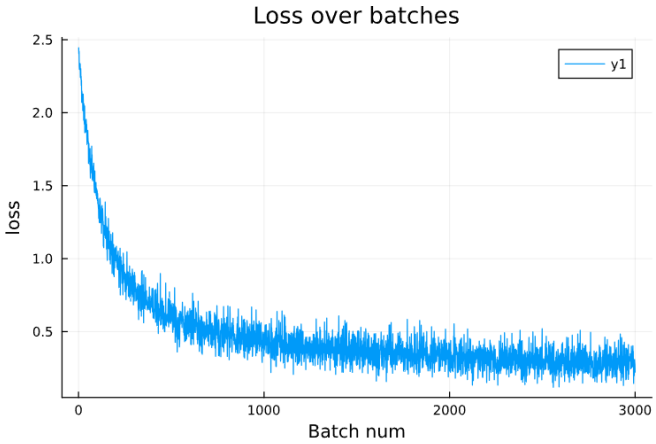


Fig. 2. Loss values over processed consecutive batches during training.

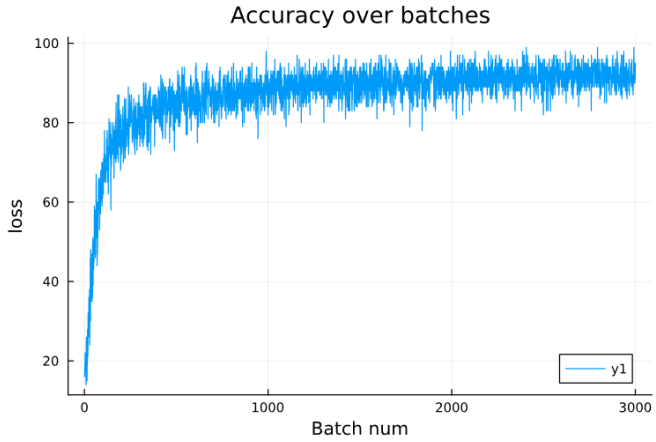


Fig. 3. accuracy over processed consecutive batches during training.

As in the case of a decrease in loss value, the most significant accuracy increase can be seen mainly at the beginning of the training process. Nonetheless, more prolonged network learning results in better predictions. However, it is essential to remember that prolonged training can cause the network to overfit the training data and perform exceptionally well on training data but poorly on unseen test data.

TABLE I
Result metrics of described implementation (5 epochs)

| **Metric** | ***Values*** |
|---|---|
| Test accuracy | 92.24 % |
| Memory allocation | 4.59 GiB |
| Training time | 11.53 s |

After 5 epochs of training, our solution achieves 92.24% accuracy. The training takes around 11.53 seconds and allocates approximately 4.59 GiB of memory.

### B. Flux reference solution

The first reference solution was based on the Flux library. Flux is a powerful and flexible machine learning tool for the Julia programming language, designed to provide a straightforward interface for building and training neural networks. The presented results are crucial for the comparison with our library because of the usage of the same environment.

TABLE II
RESULT METRICS OF FLUX SOLUTION (5 EPOCHS)

| **Metric** | ***Values*** |
|---|---|
| Test accuracy | 94.4 % |
| Memory allocation | 13.16 GiB |
| Training time | 12.78 s |

Flux scores better accuracy than presented implementation. The difference in percentage points is around 2, which is significant. Nonetheless, the solution described in this article works faster and consumes less memory. The fact of achieving better performance than Flux is very satisfactory for us.

### C. PyTorch reference solution

PyTorch is a widely used machine learning library incorporated as an addition to the Python environment. It is highly regarded for its flexibility and ease of use, making it a preferred choice in deep learning. In-depth memory allocation is measured with the help of PyTorch Profiler.

TABLE III
RESULT METRICS OF PYTORCH SOLUTION (5 EPOCHS)

| **Metric** | ***Values*** |
|---|---|
| Test accuracy | 94.55 % |
| Memory allocation | 31.57 GiB |
| Training time | 11.88 s |

PyTorch came out on top in the aspect of achieving highest accuracy, but allocated significantly more memory than Julia's solutions. In case of training time, this option achieves similar performance to previous ones.

## VIII. SUMMARY

The implementation presented in this article achieves decent accuracy and works with satisfying performance. In terms of memory allocation it wins against Flux and PyTorch solutions. In case of time spent on training, results are very close to eachother, with our implementation taking slight lead. Unfortunately, on average, both reference libraries give better test data prediction (around 2 pp. difference) than this.

Considered RNN implementation uses a computational graph for automatic differentiation, enabling easy gradient calculation via backpropagation. The recurrent layer allows the analysis of sequences of data. It adapts its weights thanks to backpropagation through time, whereas the dense layer wraps up and feeds data forward to place, which is responsible for defining final, probability-based prediction. The computational graph concludes with a loss function node, where the network's prediction correctness is evaluated, and backward gradients calculation starts.

Proper initial weights are crucial to avoid the exploding gradient problem. Glorot initialisation is used to maintain balanced variance and stabilise the training process.

The optimisation process consisted of many steps. Using stable types, caching, or utilising Julia's dot notation increased the program's overall performance. Ensuring that model resource usage is optimal is crucial, allowing it to handle larger datasets or be configured with more parameters while still completing training in an acceptable time frame.

Overall, our final implementation was satisfactory in recognising MNIST numbers. It performs with relatively high accuracy and consumes a reasonable amount of resources. The library can still be enhanced in multiple ways, such as using different gradient optimisation algorithms or mini-batching approaches.

To sum up, Julia is a well-suited language for numerical computing and data analysis, where high performance and flexibility are essential. It grants a solid foundation for many machine-learning tasks.

## REFERENCES


[1] 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. https://ntnuopen.ntnu.no/ntnu-xmlui/bitstream/handle/11250/2624618/no.ntnu:inspera:2497321.pdf, Accessed: May 23, 2024

[2] B. van den Berg, T. Schrijvers, J. McKinna and A. Vandenbroucke, 'Forward- or reverse-mode automatic differentiation: What is the difference?'. ScienceDirect, Jan. 2024. https://www.sciencedirect.com/science/article/pii/S0167642323000928, Accessed: May 23, 2024

[3] H. Salehinejad, S. Sankar, J. Barfett, E. Colak, and S. Valaee' Recent Advances in Recurrent Neural Networks'. arXiv, Dec. 29, 2017. https://arxiv.org/abs/1801.01078, Accessed: May 23, 2024

[4] R. Pascanu, C. Gulcehre, K. Cho and Y. Bengio' How to Construct Deep Recurrent Neural Networks'. arXiv Dec 20, 2013. https://arxiv.org/abs/1312.6026, Accessed: May 23, 2024

[5] A. G. Baydin, B. A. Pearlmutter, A. Radul, J.M. Siskind' Automatic differentiation in machine learning: a survey'. ArXiv, Feb 5, 2018. https://arxiv.org/abs/1502.05767, Accessed: June 20, 2024

[6] C. C. Margossian' A review of automatic differentiation and its efficient implementation'. Wiley interdisciplinary reviews: data mining and knowledge discovery, 9(4), e1305, 2019. https://wires.onlinelibrary.wiley.com/doi/abs/10.1002/widm.1305, Accessed: June 20, 2024

[7] P. J. Werbos, 'Backpropagation through time: what it does and how to do it' in Proceedings of the IEEE, vol. 78, no. 10, pp. 1550-1560, Oct. 1990, doi: 10.1109/5.58337. https://ieeexplore.ieee.org/abstract/document/58337, Accessed: June 20, 2024

[8] Y. Hu, A.E. Huber, J. Anumula, S. Liu' Overcoming the vanishing gradient problem in plain recurrent networks', 2018. ArXiv, abs/1801.06105. https://arxiv.org/abs/1801.06105, Accessed: June 20, 2024

[9] X. Glorot, Y. Bengio' Understanding the difficulty of training deep feedforward neural networks', 2010. Journal of Machine Learning Research - Proceedings Track. 9. 249-256. https://proceedings.mlr.press/v9/glorot10a.html, Accessed: June 20, 2024

[10] S. Osowski, R. Szmurło 'Matematyczne modele uczenia maszynowego w językach MATLAB i PYTHON', 2024. Oficyna Wydawnicza Politechniki Warszawskiej.