# Library for Automatic Differentiation In Julia adapted to use in Recurrent Neural Network

Marcin Latawiec
*Faculty of Electrical Engineering*
Warsaw University of Technology
marcin.latawiec.stud@pw.edu.pl

***Abstract*—This article references the issue of automatic differentiation as a task for a machine learning model. Among other things, the document presents a library for creating a recurrent neural network was compared with other reference solutions. The project was carried out in the course Algorithms in Data Engineering.**

***Index Terms*—Recurrent Neural Network, Model, Reverse-Mode, Julia Language**

## I. Introduction

The rapid advancement in machine learning (ML) and artificial intelligence (AI) has led to the development of sophisticated neural network architectures capable of handling a variety of complex tasks. Among these architectures, recurrent neural networks (RNNs) have proven particularly effective in processing sequential data, such as time series, natural language, and speech. The ability of RNNs to maintain an internal state or memory makes them uniquely suited for tasks that require context or temporal dependencies.

One of the key challenges in implementing RNNs is efficiently computing the gradients needed for training. This is where automatic differentiation (AD) comes into play. AD is a set of techniques to numerically evaluate the derivative of a function specified by a computer program. It is a fundamental tool in the optimization of neural networks, as it allows for the efficient and accurate calculation of gradients.

This article explores a machine learning library specifically designed to leverage automatic differentiation for recurrent neural networks. By integrating AD, this library simplifies the process of training RNNs, enabling more effective and faster development cycles for models that require handling sequential data. Whether you are dealing with tasks such as language modeling, time series forecasting, or speech recognition, this library offers powerful tools to streamline your workflow and improve the performance of your RNN models.

### A. Recursive Neural Networks (RNN) at Glance

RNN is one of the two main categories of artificial neural networks, distinguished by how information flows between its layers. Unlike the one-way feedforward neural network, an RNN is bi-directional, meaning it allows the output from certain nodes to influence the subsequent input to the same nodes. This capability to use internal state (or memory) for processing sequences of inputs makes RNNs suitable for tasks like recognizing unsegmented, connected handwriting or speech.

Recurrent Neural Networks (RNNs) require a specialized method to automatically calculate derivatives in reverse mode, known as Backpropagation Through Time (BPTT). This method extends the backpropagation algorithm to train RNNs, addressing their temporal characteristics and allowing them to learn from sequential data by unrolling the network over time. BPTT is closely associated with one of the major challenges in training simple recurrent neural networks.

Both infinite-impulse and finite-impulse networks can incorporate additional stored states controlled directly by the network, or these states can be managed by another network or graph that includes time delays or feedback loops. These controlled states are known as gated states or gated memory and are integral to long short-term memory networks (LSTMs) and gated recurrent units (GRUs), also known as Feedback Neural Networks (FNNs). Recurrent neural networks are theoretically Turing complete, meaning they can execute arbitrary programs to process sequences of inputs.

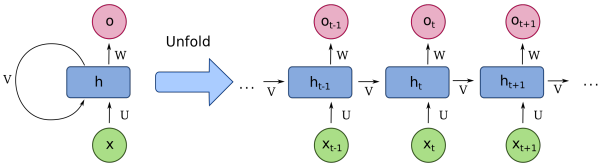


Fig. 1. Compressed (left) and unfolded (right) basic recurrent neural network

### B. Gradient descent in RNN

Gradient descent is a first-order iterative optimization algorithm used to find the minimum of a function. In the context of neural networks, it is employed to minimize the error term by adjusting each weight in proportion to the derivative of the error concerning that weight, assuming the non-linear activation functions are differentiable. A significant challenge with using gradient descent in standard RNN architectures is the vanishing gradient problem. This issue arises because the error gradients diminish exponentially as the time lag between significant events increases, making it difficult for the network to learn long-term dependencies effectively.

### C. Julia Language

The Julia programming language has gained significant traction in the scientific computing and machine learning communities due to its high performance, ease of use, and dynamic nature. One of Julia's standout features is its ability to seamlessly integrate with C and Fortran libraries while providing the high-level functionality needed for developing complex algorithms. This makes it an ideal candidate for building libraries for automatic differentiation, a crucial component in training neural networks, especially recurrent neural networks.

## II. RNN Graph implementation

The RNN implementation uses a computational graph approach for automatic differentiation. This flexible approach simplifies the orchestration of the model. The primary goal of this implementation is to create an efficient and generic recurrent neural network that functions as a classifier. The network architecture is relatively straightforward, comprising one unrollable recurrent layer and one dense layer, which are described in more detail in the following section.

### A. Layers

**Recurrent Layer**: The recurrent layer is of key importance, because it enables backpropagation through time mechanism. It is possible by the layer by handling two primary inputs: the current (t) data input and the previous (t-1) hidden state. It produces an output state that is used throughout all these hidden time states sequentially. When this layer is unrolled, it forms the structure depicted in Fig. 2.

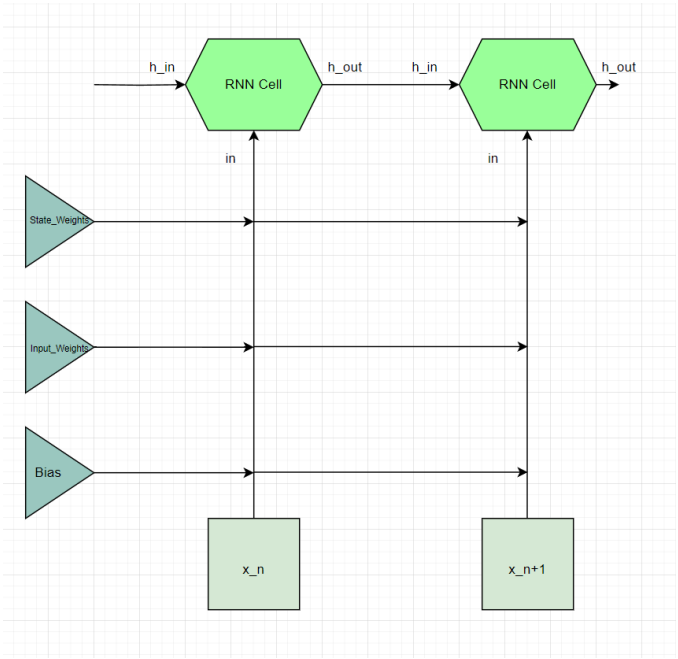


Fig. 2. Implementation schema of Recurrent Layer (unfolded)

**Dense Layer**: The dense layer functions as a intermediate layer, applying a linear transformation to generate the desired features. This layer and its associated computations are incorporated into the computational graph. There are two optimizable parameters associated with this layer: input weights and biases. In RNN implementation it uses softmax function, which formula is written below:
Formula for calculating Activation function($\sigma$):

$$h_t = \sigma(W * h_{t-1} + b)$$

**Loss function**: At the end of the computational graph, there is a node dedicated to the loss function. This node assesses the accuracy of the network's predictions during the forward pass and plays a crucial role in optimizing gradients during backpropagation. The solution described involves computing the loss based on a probabilities vector. Specifically, the output from the dense layer is fed into the cross-entropy loss function using a numerically stable Softmax operation. This approach is necessary because the explosive nature of RNN gradients can lead to the generation of matrices containing NaNs, which would compromise the model's performance.

The formula for calculating the loss function:

$$\frac{\partial L(\Theta)}{\partial W} = \sum_{t=1}^{T} \frac{\partial L(\Theta)}{\partial W}$$

### B. Weights initialization

Proper initialization of weights plays a critical role in mitigating issues related to the exploding gradient problem. The current standard approach for initialization of the weights of neural network layers and nodes that use the Sigmoid or TanH activation function is called "glorot" or "xavier" initialization. It was selected as the preferred method. This approach ensures that the weights are initialized in a manner that balances the variance of activations across layers, thereby promoting stable and effective training of neural networks.

### C. Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent (SGD) operates by randomly selecting one data point from the entire dataset during each iteration, thereby drastically reducing computational overhead. An enhancement to this approach involves sampling a small batch of data points rather than just one at each step, known as "mini-batch" gradient descent. Mini-batch gradient descent aims to strike a balance between the accuracy improvements of traditional gradient descent and the computational efficiency of SGD. This method efficiently updates model parameters by leveraging the average gradient computed from a subset of data points, making it a widely adopted optimization technique in training neural networks.

## III. Optimization

The optimization discussed in this article focuses on managing the allocated memory and the time required to train the model. It aims to ensure that the model remains flexible

enough to handle larger datasets with more parameters, while still completing training within a reasonable timeframe on typical computer systems. This approach is crucial for making machine learning models practical and scalable, enabling them to tackle complex tasks efficiently without overwhelming computational resources.

### A. Type stability

In particular situation, when the type of a local variable cannot be inferred from the types of the arguments, the compiler will produce machine code full of "if"s, covering all options of what the type of each variable could be. The loss in performance is often significant, easily by a factor of 10.
If we would infer the type of every local variable, and every expression in a method (or script) from the types (not the values) of the arguments or from constants in the code, the function will be typestable. Actually, as will be seen below, this inference of types is also allowed access to struct declarations, and to the types of the return values of functions called by the function.
In the optimized implementation of mine, Wherever was it possible arrays with a concrete element type were used.

### B. Static Arrays and Matrices

In the final version of the project, neither static arrays nor matrices were utilized due to their significant limitations in the current state. Basic operations between a matrix and a static array are impractical and impossible, rendering this feature unusable and irrelevant for the project's requirements.

### C. Cache memory

Additionaly the final version of the project covers 'caching', which is a technique used throughout the program to minimize unnecessary creations - it creates a structurre only once per run if possible. This approach ensures that objects are instantiated only once whenever possible, thereby limiting the need for garbage collection and reducing both the time spent on object creation and the overall memory footprint.
For instance, in scenarios like resetting the state, rather than repeatedly generating a large matrix of zeros for each batch, a single matrix of zeros is created initially and reused. This optimization not only streamlines the program's execution but also conserves memory by eliminating redundant allocations. This strategy of caching contributes significantly to enhancing the efficiency and performance of the application.

### D. Syntactic Loop Fusion

Writing code in a "vectorized" style, akin to what is familiar in Matlab, Numpy, R, and similar environments, is key in this project. The dots allow Julia to recognize the "vectorized" nature of the operations at a syntactic level (before e.g. the type of x is known). This optimization has been extensively applied in both forward and backward functions within the computational graph. Not only does it accelerate program execution, but it also minimizes memory allocations. By performing all operations within the same loop, we eliminate the necessity to allocate memory for new vectors or matrices, contributing to overall efficiency and performance.

## IV. Evaluation

The evaluation of given project was conducted with a usage of widely recognized MNIST image dataset. It was used both for training and testing the network. The MNIST database (Modified National Institute of Standards and Technology database) is a large database of handwritten digits that is commonly used for training various image processing systems. In order to provide a comprehensive comparison, results from reference solutions are also presented. The evaluation focuses on three primary metrics: accuracy of predictions on the test data, data loss and memory allocation during training.
To ensure a fair comparison, all solutions were configured with identical network architectures and learning parameters, including parameters like learning rate and batch size. This consistency allows for a direct assessment of how each approach performs in terms of computational efficiency and predictive accuracy.
Parameters were set as follows:
*batchsize* = 100
*learning rate* = $15 * 10^{-3}$
*epochs* = 5, each containing 600 batches
Each epoch worked on randomly picked batch data permutations from MNIST dataset, which was driven by the desire to generalize the dataset and minimize different trends in classification.
Below are the graphs of the accuracy and loss functions in the batch domain:

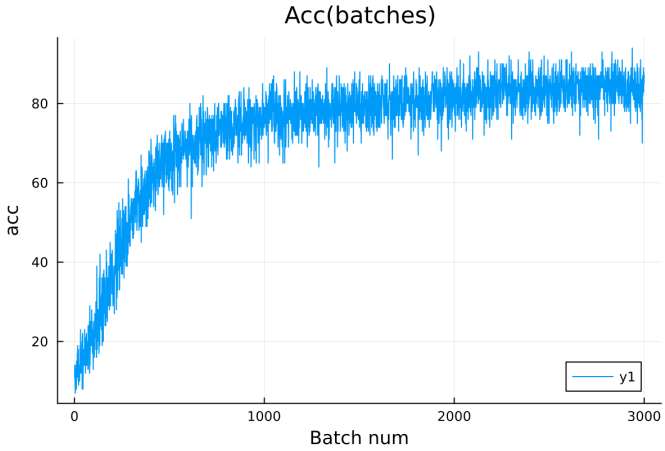


Fig. 3. Accuracy function over batches before optimalization process

For the earlier unoptimized architecture, it should be felt that the number of 5 epochs is insufficient to adequately train the network, the accuracy rate is constantly increasing.

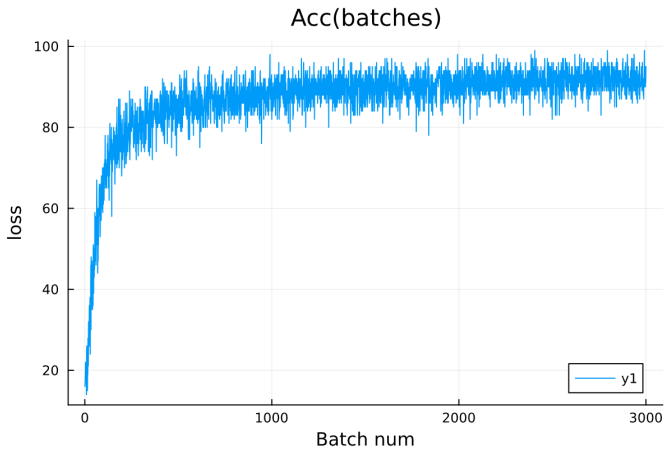


Fig. 4. Accuracy function over batches after optimalization process

After optimalization process, what is observed is that, Accuracy gradually increases and after a few batch rounds it reaches a decent value and oscillates up to 92%.

Below I show the relationship of the loss value in relation to the subsequent batch, the relationship is inversely proportional to the previously shown accuracy.

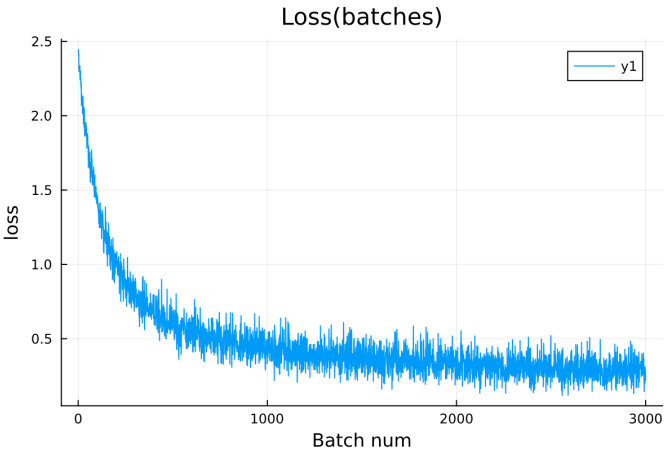


Fig. 5. Loss function over batches after optimalization process

| Tests | Domain | Value |
|---|---|---|
| Before optimalization | **Test accuracy** | **85.35** |
| After optimalization | | **92.21** |
| Before optimalization | **Data Loss** | **0.5108** |
| After optimalization | | **0.27** |
| Before optimalization | **Memory Allocation dur. Training** | **6.18 GB** |
| After optimalization | | **5.11 GB** |

After 5 epochs of training the RNN achieves 92.24% accuracy. The training takes allocates on 5th Epoch about 0.906 GB of memory space.

### A. Referal to another models

Below is a table that includes results and indicators for 2 reference RNN models from the Flux and Torch libraries

| Library | Domain | Value |
|---|---|---|
| ***Flux*** | **Test accuracy** | **0.9441** |
| ***Torch*** | | **0.9418** |
| ***Flux*** | **Loss** | **0.198** |
| ***Torch*** | | **0.194** |
| ***Flux*** | **Allocated Memory (In total)** | **14.537 GB** |
| ***Torch*** | | **31.57 GB** |
| ***Flux*** | **Cumulative Training Time** | **18.28s** |
| ***Torch*** | | **16.24s** |

*1) Flux:* First reference solution utilized *Flux* library. Flux is a robust and versatile machine learning framework for the Julia programming language, designed to offer a user-friendly interface for constructing and training neural networks. Comparing the obtained results with those obtained using Flux is particularly significant, as both solutions were tested within the same environment. This allows for a direct and fair evaluation of the performance and efficiency of implemented library relative to Flux.

Flux excelled in achieving the highest accuracy among the tested solutions. The difference is around 2 points. It is worth to add, that the solution described in this article consumes less memory.

*2) Torch:* Torch is Python written library, widely used in Machine Learning projects. Torch is highly esteemed for its flexibility and ease of use, which makes it a popular choice in deep learning applications. Detailed memory allocation is measured using the PyTorch Profiler, providing valuable insights into the resource usage and efficiency of the implementations. PyTorch also scores better accuracy than presented implementation. However, it also required significantly more memory compared to the Julia-based solutions. In terms of training time, PyTorch performed similarly to the other implementations, maintaining comparable efficiency.

## V. Summary

To conclude, the implementation of the RNN detailed in this article demonstrates commendable accuracy and satisfactory performance. The training time is comparable across the board, with proprietary implementation having a slight edge. However, the reference libraries, on average, provide better test data predictions: circle about 94% in both cases vs 92.2%.

The discussed RNN implementation leverages a computational graph for automatic differentiation, facilitating seamless gradient calculation via backpropagation through time. The recurrent layer is adept at analyzing data sequences, adjusting its weights through backpropagation through time. Meanwhile, the dense layer processes the data and directs it towards generating the final, probability-based predictions. The computational graph culminates in a loss function node, which assesses the network's prediction accuracy and initiates the calculation of backward gradients. To prevent the exploding gradient problem, proper initial weights are essential, to which Xavier weights initialization method was fitted.

Overall, Julia proves to be an excellent language for numerical computing and data analysis, offering both high performance and flexibility. It provides a robust foundation for a wide range of machine learning tasks.

## 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. [Online]. Available:

[5] J. Revels, M. Lubin, and T. Papamarkou, 'Forward-Mode Automatic Differentiation in Julia'. arXiv, Jul. 26, 2016. Accessed: Mar. 05, 2024.

[6] Geeks for Geeks, 'Difference between Batch Gradient Descent and Stochastic Gradient Descent', https://www.geeksforgeeks.org/difference-between-batch-gradient-descent-and-stochastic-gradient-descent/ Accessed: June, 19, 2024.

[7] Jason Brownlee, Weight Initialization for Deep Learning Neural Networks, https://machinelearningmastery.com/weight-initialization-for-deep-learning-neural-networks/ Accessed: June, 19, 2024.

[8] Frames White, Miles Lubin, Guillaume Dalle, 'Differentiation tools in Julia' https://juliadiff.org/

[9] Philippe Mainçon 'Writing type-stable Julia code' https://www.juliabloggers.com/writing-type-stable-julia-code/

[10] The Julia Programming Language Documentation, https://docs.julialang.org/en/v1/

[11] PyTorch documentation, https://pytorch.org/docs/stable/index.html