# Understanding Neural Networks: A From Scratch Implementation In Julia

Abdelbacet Mhamdi
*Dept. of Electrical Engineering, ISET Bizerte, Tunisia*
abdelbacet.mhamdi@bizerte.r-iset.tn

*Abstract*—**This paper presents an educational implementation of artificial neural networks from scratch using the Julia programming language. By developing neural networks without relying on existing deep learning frameworks, we provide valuable insights into the fundamental concepts and mathematical principles underlying modern machine learning. This implementation serves as both a learning tool and a demonstration of Julia's capabilities in scientific computing.**

*Index Terms*—**Julia, artificial neural network, backpropagation, scientific computing, deep learning**

## I. Introduction

Implementing Artificial Neural Networks (ANN) from scratch in Julia offers numerous benefits, particularly for those seeking a deeper understanding of deep learning principles. By building ANNs from the ground up, learners gain invaluable insights into the inner workings of neural networks, including the intricacies of forward and backward propagation, weight initialization, and optimization techniques. This hands-on approach not only solidifies theoretical knowledge but also enhances practical skills, enabling developers to troubleshoot and optimize models more effectively. Julia's high-performance capabilities and ease of use make it an ideal language for such projects. Additionally, Julia's rich ecosystem and performance-oriented design ensure that even from-scratch implementations can achieve impressive computational efficiency, making it a powerful tool for both educational purposes and experimental research [1].

This paper outlines the importance of from-scratch neural network implementations, discusses the technical benefits of such projects, and highlights the advantages of using Julia for this purpose. We also present a practical guide to using our neural network implementation in Julia, along with results and future directions for the project.

## II. Importance of From-Scratch Implementation

### A. Educational Value

Building neural networks from scratch helps students and practitioners understand startegies of weight initialization, forward and backward propagation algorithms through gradient descent optimization. By delving into these foundational concepts, learners can grasp the mechanics of how neural networks learn and adjust their parameters during training.

Additionally, constructing neural networks from the ground up provides insights into activation functions and their derivatives, loss function computation, and the role of hyperparameters. This comprehensive understanding equips individuals with the knowledge to fine-tune models, troubleshoot issues, and innovate in the field of deep learning.

### B. Technical Benefits

From-scratch implementations provide several significant benefits. Firstly, they offer complete control over the network architecture, allowing developers to tailor the design to specific needs and experiment with innovative structures. Secondly, transparency in the computation process is greatly enhanced, as every step and calculation is explicitly defined and understood. This transparency facilitates better debugging capabilities, making it easier to identify and resolve issues. Additionally, the ability to customize and experiment with different components, such as activation functions and optimization algorithms, fosters a deeper understanding of their impact on the network's performance, as developers can fine-tune every aspect of the implementation to achieve maximum efficiency [2, 3].

### C. Julia's Advantages

Julia offers unique benefits for neural network implementation. Its just-in-time compilation ensures high performance, making it suitable for computationally intensive tasks. The language's clear mathematical syntax allows for intuitive expression of complex algorithms, while native array operations streamline data manipulation. Additionally, Julia's multiple dispatch feature provides flexible design options, enabling developers to write more generic and reusable code. The strong type system enhances code reliability and maintainability, and its excellent numerical computation capabilities make Julia a powerful tool for implementing and optimizing neural networks [1].

## III. Project Implementation and Usage

### A. Core Components

The core components of our neural network implementation, as seen in fig. 1, include the following [4, 5, 6, 7]:

- **Layer Struct:** This object defines the structure of each layer in the neural network, including the number of neurons, activation functions, and weight initialization methods.
- **Activation Functions:** Various activation functions such as ReLU, Sigmoid, and Softmax are implemented to introduce non-linearity into the model.
- **Loss Function:** The loss function measures the difference between the predicted output and the actual target values, guiding the optimization process.
- **Optimizer:** An optimization algorithm, such as Gradient Descent, is used to update the weights of the network to minimize the loss function.
- **Forward and Backward Propagation:** These methods are essential for computing the output of the network and updating the weights based on the loss gradient.

### B. Usage Guide

The neural network is built from scratch and trained using some data. Here is a possible representation of our architecture:

```
model = [ # MLP
    Layer(nf,n1,relu;distribution='n'),
    Layer(n1,n2,relu;distribution='n'),
    Layer(n2,nt,softmax;distribution='n')
    ]
```

where `nf`, `nt` and `n\d+` denote, respectively, the numbers of input features, output targets, and neurons in the hidden layers. The choice of distribution for the weights initialization is either a normal distribution (`n`) or uniform distribution (`u`). Both `Xavier` and `He` methods were implemented.

Some of the hyperparameters are configured as follows:

```
Settings(epochs, batch_size)
```

We can define a `struct` for the regularization as follows:

```
reg = Regularization(method, λ, r, dropout)
```

`method` can be symbol or string of one of the following: `l1`, `l2`, `elasticnet`, or `none`. The `λ` parameter is the regularization parameter. The `r` parameter determines the mix of penalties in case of `elasticnet` method. The `dropout` parameter is the dropout rate. `loss` and `optimizer` are accessed through:

```
Solver(loss, optimizer, η, reg)
```

The `loss` argument can be `:mae`, `:mse`, `:rmse`, `:binarycrossentropy` or `:crossentropy`. `:sgd` is the default optimizer. The model is trained using the following method:

```
TrainNN(model, data_in, data_out,
        x_val, y_val; solver)
```

Under the hood, the `TrainNN` method calls the `FeedForward` and `BackProp` functions. The `FeedForward` method returns the pre-activations `z` and the post-activations `a`, bundled into `data_cache`. The method signature is:

```
data_cache = FeedForward(model,
        data_in; solver::Solver)
```

The `BackProp` method allows to return the `loss` and the gradients of the weights and biases: `∇W` and `∇b`, as follows [5, 6, 7, 8, 9]:

```
loss, ∇W, ∇b = BackProp(model,
        data_cache, data_out; solver::Solver)
```

## IV. Results and Discussion

Julia's version and status of used packages are shown in fig. 2. Our neural network is trained using the classical `iris` data. Our task is to classify flowers into three classes `setosa`, `versicolor` and `virginica` based on the following features: `sepal length`, `sepal width`, `petal length`, and `petal width`. Fig. 3 displays the model's loss for both the training and test sets after each epoch. Fig. 4 provides specifics about confusion matrix, accuracy, precision, recall and f1-score metrics. We can safely say that the model is performing well on both the training and test sets. The implementation is efficient and provides a solid foundation for further experimentation and learning.

## V. Conclusion

Implementing neural networks from scratch in Julia provides valuable insights into the fundamentals of deep learning while leveraging a modern, high-performance programming language. This project demonstrates the educational value of understanding neural networks at their core, while also providing a practical tool for learning and experimentation.

The project has several planned enhancements to improve both performance and functionality:

- Implement batch-level parallelization for backpropagation to replace the current sequential loop implementation;
- Some other optimizers will be implemented to provide more training options.

These enhancements will bring the implementation closer to production-level frameworks while maintaining the educational value of a from-scratch implementation. The additions will also serve as excellent case studies for understanding various optimization techniques in deep learning.

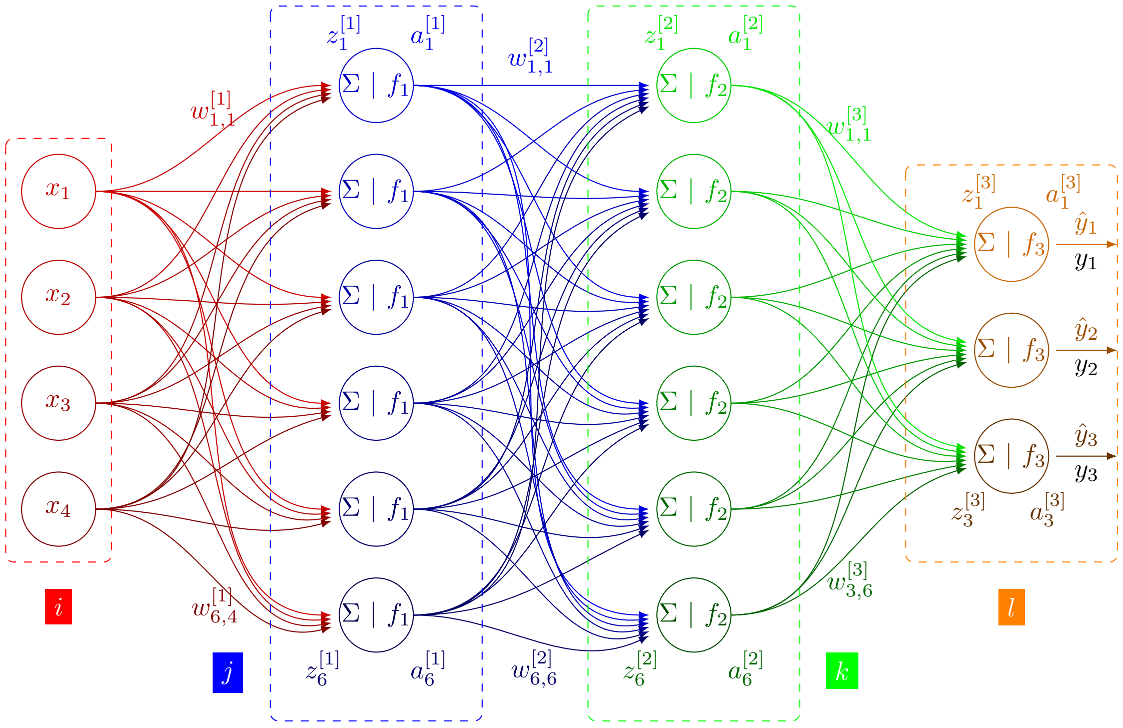
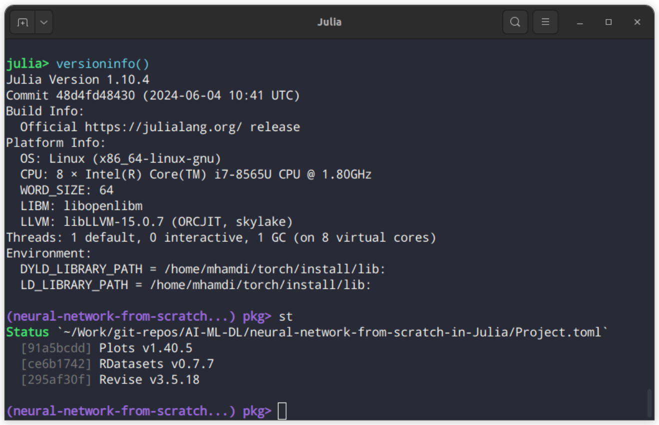


Fig. 1. Structure of the neural network



```
julia> versioninfo()
Julia Version 1.10.4
Commit 48d4fd48430 (2024-06-04 10:41 UTC)
Build Info:
  Official https://julialang.org/ release
Platform Info:
  OS: Linux (x86_64-linux-gnu)
  CPU: 8 × Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz
  WORD_SIZE: 64
  LIBM: libopenlibm
  LLVM: libLLVM-15.0.7 (ORCJIT, skylake)
Threads: 1 default, 0 interactive, 1 GC (on 8 virtual cores)
Environment:
  DYLD_LIBRARY_PATH = /home/mhamdi/torch/install/lib:
  LD_LIBRARY_PATH = /home/mhamdi/torch/install/lib:

(neural-network-from-scratch...) pkg> st
Status `~/Work/git-repos/AI-ML-DL/neural-network-from-scratch-in-Julia/Project.toml`
  [91a5bcdd] Plots v1.40.5
  [ce6b1742] RDatasets v0.7.7
  [295af30f] Revise v3.5.18

(neural-network-from-scratch...) pkg> 
```

Fig. 2. Julia's version and status of packages

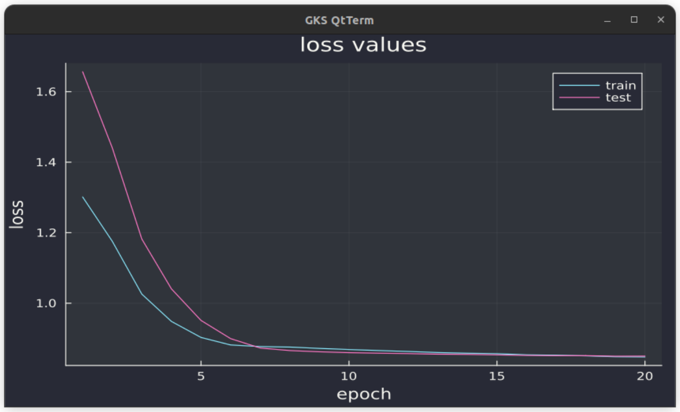
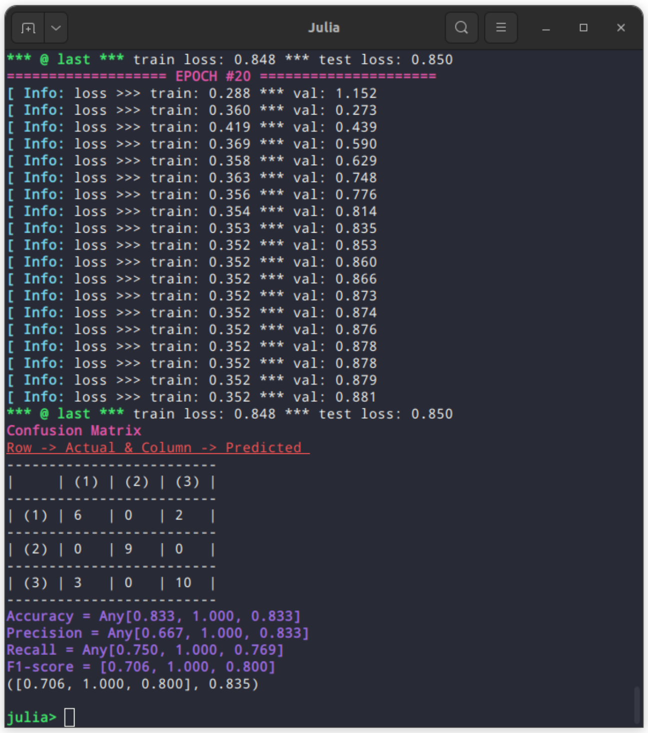


Fig. 3. Graph of loss values

## References

[1] 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.
[2] S. Raschka and V. Mirjalili, *Python Machine Learning*. Packt Publishing Ltd, 2017.
[3] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, "Dive into deep learning."
[4] A. Punnen, "Backpropagation: A gentle introduction," https://medium.com/@alexpunnen/backpropagation-a-gentle-introduction-6e4b73f7c8b4, 2017, accessed: 2023-10-27.
[5] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.
[6] M. Nielsen, *Neural Networks and Deep Learning*. Determination Press, 2015.
[7] M. Mazur, "The backpropagation algorithm," https://madmaze.github.io/2015/03/27/Backpropagation/, 2015, accessed: 2023-10-27.
[8] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986.
[9] Wikipedia contributors, "Backpropagation," https:



```
*** @ last *** train loss: 0.848 *** test loss: 0.850
================== EPOCH #20 ====================
[ Info: loss >>> train: 0.288 *** val: 1.152
[ Info: loss >>> train: 0.360 *** val: 0.273
[ Info: loss >>> train: 0.419 *** val: 0.439
[ Info: loss >>> train: 0.369 *** val: 0.590
[ Info: loss >>> train: 0.358 *** val: 0.629
[ Info: loss >>> train: 0.363 *** val: 0.748
[ Info: loss >>> train: 0.356 *** val: 0.776
[ Info: loss >>> train: 0.354 *** val: 0.814
[ Info: loss >>> train: 0.353 *** val: 0.835
[ Info: loss >>> train: 0.352 *** val: 0.853
[ Info: loss >>> train: 0.352 *** val: 0.860
[ Info: loss >>> train: 0.352 *** val: 0.866
[ Info: loss >>> train: 0.352 *** val: 0.873
[ Info: loss >>> train: 0.352 *** val: 0.874
[ Info: loss >>> train: 0.352 *** val: 0.876
[ Info: loss >>> train: 0.352 *** val: 0.878
[ Info: loss >>> train: 0.352 *** val: 0.878
[ Info: loss >>> train: 0.352 *** val: 0.879
[ Info: loss >>> train: 0.352 *** val: 0.881
*** @ last *** train loss: 0.848 *** test loss: 0.850
Confusion Matrix
Row -> Actual & Column -> Predicted
-------------------------
|     | (1) | (2) | (3) |
-------------------------
| (1) | 6   | 0   | 2   |
-------------------------
| (2) | 0   | 9   | 0   |
-------------------------
| (3) | 3   | 0   | 10  |
-------------------------
Accuracy = Any[0.833, 1.000, 0.833]
Precision = Any[0.667, 1.000, 0.833]
Recall = Any[0.750, 1.000, 0.769]
F1-score = [0.706, 1.000, 0.800]
([0.706, 1.000, 0.800], 0.835)

julia> 
```

Fig. 4. Evaluation results

//en.wikipedia.org/wiki/Backpropagation, 2023, accessed: 2023-10-27.