Automatic differentiation library in Julia

***Abstract*—This article presents a library for automatic differentiation (AD) and neural networks implemented in Julia. It reviews existing tools and compares various implementation approaches. The library utilizes reverse-mode AD based on computation graphs to efficiently compute gradients. This work demonstrates the potential of Julia as a high-performance platform for neural network development and automatic differentiation.**

***Index Terms*—automatic differentiation, Julia, CNN**

I. Introduction

Automatic differentiation (AD) is a fundamental technique in scientific computing and machine learning, where efficient and accurate gradient computation is essential for model optimization. Unlike symbolic or numerical differentiation, AD offers a practical balance between precision and performance, making it especially valuable in training neural networks.

Julia is a modern programming language designed for high-performance numerical computing. Its support for operator overloading, just-in-time (JIT) compilation, and intuitive syntax makes it well-suited for implementing AD tools and machine learning frameworks. Julia combines the ease of a high-level language with execution speed comparable to low-level languages, enabling rapid development without sacrificing performance.

The goal of this work is to develop a lightweight library for automatic differentiation and neural networks in Julia. The implementation uses reverse-mode AD with computation graphs to enable efficient gradient calculation. This paper compares the proposed solution with existing AD tools, explores various implementation strategies, and describes the design and development of the implemented library.

II. Literature review

In recent years, deep learning has emerged as one of the fastest-growing and most influential technologies worldwide. A fundamental component of deep learning is the computation of function derivatives. To perform these calculations efficiently, automatic differentiation (AD) is used. Without AD, the optimization process would be considerably slower, hindering the progress of deep learning and driving up costs. Given its critical role, AD has become the focus of extensive research, with numerous studies dedicated to improving and refining its methods.

There are three primary methods for computing derivatives: symbolic differentiation, finite difference methods, and AD. Rall and Corliss [1] discuss the advantages and limitations of each. Their article highlights that, although symbolic differentiation provides exact values, it requires significant computational resources and cannot be applied to functions defined within computer programs or subroutines. They also note that finite difference methods introduce greater numerical errors. They conclude that automatic differentiation is the preferred approach for achieving both efficiency and accuracy.

Automatic differentiation has two types: forward mode and reverse mode. "Forward-mode automatic differentiation in Julia" [2] propose a forward-mode AD approach. Thanks to various optimizations, their implementation outperforms a naive C++ version. However, it struggles with large inputs and remains slower than reverse mode. As pointed out by Grstad [3] forward mode requires computation of the full Jacobian matrix, which scales poorly for large inputs. In contrast, reverse mode computes only the Vector-Jacobian Product (VJP), making it more efficient for functions with high-dimensional inputs.

In his lecture [4], Roger Grosse discusses a popular Python automatic differentiation library and explains how to implement one from scratch. He outlines the key steps of reverse-mode automatic differentiation: constructing the computation graph, the forward pass, computing vector-Jacobian products, and backward pass. The lecture also compares how computation graphs are created in two libraries: TensorFlow and Autograd. TensorFlow provides a mini-language for graph construction, whereas Autograd traces the forward pass, enabling an interface that is nearly indistinguishable from NumPy. Autograd's approach is effective because it offers a familiar experience and eliminates the need for learning a new syntax.

In "Automatic Differentiation in Machine Learning: A Survey" [5] we can find major implementation strategies and a survey of existing tools for different programming languages. The survey mentions a way of implementing AD in modern programming languages through operator overloading. This approach allows redefinition of basic operators, such as +, making the method more intuitive to use without requiring additional learning. Additionaly, the survey points out possible challenges that AD isn't immune to, like floating point arithmetic.

Creating a custom library would provide a lightweight and well-suited solution that can be easily improved. Moreover, as Roger Grosse [4] points out, while most people may never need to know how to implement such library themselves, doing so can help them better understand and utilize well-established existing libraries. Based on the referenced literature, the best approach for the library would be to use reverse-mode automatic differentiation. Moreover, using julia is an excellent choice, as it supports operator overloading and can achieve performance on par with C.

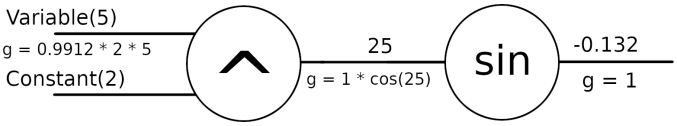

Fig. 1. Computation graph

III. Implementation

A. Reverse-Mode Automatic Differentiation and Computation Graphs

The implemented library uses reverse-mode automatic differentiation (AD) based on computation graphs. This approach is both easy to understand and extend, making it suitable for library. The differentiation process consists of three main steps: building the computation graph, performing the forward pass to evaluate the function, and executing the backward pass to compute gradients.

The computation graph consists of three types of elements: Constants, which represent fixed numerical values; Variables, which correspond to mutable values; Operators, which represent operations and store information about their inputs. Thanks to operator overloading in Julia, we do not need to manually define each Operator and its inputs, this process is handled automatically during computation. To construct the graph, we take an Operator and recursively visit its inputs. Each visited node is added to the end of an array once all its inputs have been visited.

During the forward pass, we iterate over the computation graph. If a node is an Operator, we execute its associated function to compute the output based on its inputs. Operator overloading allows us to use a single forward function, which automatically dispatches the correct computation for each node type. Once the forward pass is complete, all outputs are stored within the respective nodes.

The backward pass follows the reverse topological order of the graph. Similar to the forward pass, we call the backward function at each Operator node to compute and propagate gradients to its input nodes. Thanks to the use of reverse-mode automatic differentiation, there is no need to compute the entire Jacobian matrix. Instead, gradients are propagated efficiently using the chain rule. At each node, the local gradient is multiplied by the gradient received from the subsequent node in the graph, as shown in Formula 1.

$$\frac{df}{dx} = \frac{df}{dg} \cdot \frac{dg}{dx} \tag{1}$$

The process for equation 2 is depicted on image 1. On top of lines are outputs of forward pass, below lines are gradients and circles are Operators.

$$f(x) = sin(x^2) \tag{2}$$

B. Structure

The code is devided into 3 projects: differential, net and main. Differential is a library for automatic differentiation, net is a library for neural networks, and the main project is used to run the program. In Julia, each project follows a standard structure consisting of /src and /test folders, along with two configuration files: Project.toml and Manifest.toml.

The /src folder in the differential project is organized into two main parts: the /struct folder, which contains code defining the graph nodes, and the /operations folder, which includes the implementation of graph construction, the forward pass, and the backward pass. In the net project, the /src directory is divided into multiple files based on their functionality, such as data preparation, weight initialization and layers preparation. Finally, in the /src directory of the main project, there are three files: one for launching the custom implementation, one for running the Flux-based reference implementation, and one for executing the PyTorch version.

C. Challenges

When implementing this library, several challenges were encountered. One notable issue was related to data storage. In Julia, it is optimal to use immutable structures since they are easy to access and efficient. However some operations produce scalar results that when stored in immutable structures can't be changed. To work around this limitation numbers are wrapped in a single-element vectors.

Another challenge was was optimizing multidimensional array multiplication. In the library, operations such as convolution or gradient optimization ultimately reduce to matrix multiplications, which account for the majority of the computation time. Therefore, they are the most important to optimize.

IV. Optimization

A. Types

Julia can optimize code very effectively, but it requires precise type information to do so. It is important to avoid using the Any type whenever possible, as it prevents the compiler from applying many of its advanced optimization techniques. In the AD library, types within the structures representing graph nodes are defined using parametric types. Compared to using a Union of multiple types or assigning specific types manually, parametric types offer greater flexibility and provide better performance. A problem arises when assigning types to the Operator struct. Sometimes, when inputs have different types, the result type is promoted to the one with higher dimensions. For instance, the result of an element-wise addition (.+) between a matrix and a vector will be a matrix. However, this behavior is not consistent across all operations and can be difficult to predict. For example, the result of the sum operation is of type Number, regardless of the input. As a result, the type in an Operator node depends both on the input types and the specific operation being performed. In some cases, type resolution must be hardcoded to handle such inconsistencies.

B. Keeping proper types

In neural networks, data is usually of type Float32 or even Float16. By default, floating-point numbers in Julia are of type Float64. Whenever there is an operation between different types, Julia needs to convert between them, which can slow down performance and increase memory usage. Therefore, in this project, all variables inside library are of type Float32. If the data is of another type, such as a BitMatrix, it need to be converted into a Float32 matrix.

C. @views and @inbounds

When doing slicing, Julia creates a copy of the sliced array by default. This can lead to unnecessary memory usage and reduced performance. By using the @views macro, slices are instead treated as references, which avoids copying and improves efficiency. The @inbounds macro disables bounds checking for array accesses. While it must be used with caution to avoid runtime errors, it can speed up performance.

D. Parallelization

Many operations in the program can be executed in parallel. Julia's @threads macro allows for easy parallelization of loops by distributing iterations across a thread pool initialized at the start of the program. This enables efficient use of multiple CPU cores without requiring complex thread management. For optimal performance, the number of threads should typically match the number of physical cores available on the CPU.

E. Storage

For certain operations, it is beneficial to store intermediate data. For example, during the forward pass of a max-pooling operation, a mask is created to record the positions of maximum values in the input matrix. This mask is then reused during the backward pass to correctly propagate gradients. Another optimization involves storing constant values that do not change between passes. This eliminates the need for repeated reinitialization, improving both efficiency and performance.

V. Corectness tests

When training a neural network, it is often difficult to understand what is happening internally. Typically, a programmer only sees high-level statistics such as accuracy or loss. While these metrics show whether the network is learning, they provide little insight into the possible issues with algorithm. Because detecting the source of errors during training can be challenging, it is crucial that all individual components function correctly. This ensures that if the network underperforms, the programmer can focus on improving the model architecture rather than debugging low-level implementation issues.

In the case of an automatic differentiation library, the two most important components to test are the forward and backward passes. Although additional tests, such as verifying the correctness of the computation graph construction, can offer more detailed insights into potential problems, validating the forward and backward computations is sufficient to confirm the library's correctness.

For simple operations like addition or multiplication, expected results can be hardcoded and easily verified. However, this becomes more difficult for complex operations such as convolution or max-pooling on large matrices. To address this, the tests compare the gradients computed by the custom implementation with those produced by Zygote, an open-source automatic differentiation library in Julia.

In case of neural network library it is important to verify that the library correctly handles user input and raises appropriate errors when inputs are invalid. Additionally, the computation graph should be initialized properly based on the input structure. Another important aspect to test is weight initialization, and optimization algorithm since they significantly impact the result of neural network.

VI. Comparison

In order to test performance and efficiency, created solution is compared against equivalent models built using Julia's Flux and PyTorch.

A. Model architecture

Models are trained on imdb dataset using network structure:

- Batch size: 64
- Layers:
  - Embedding
  - Permutation (in created solution it is already in embedding)
  - Convolution
  - MaxPool
  - Flatten
  - Dense

On machine:

- Cpu: Intel i54460
- Ram: 16GB
- Threads: 4

B. Memory

TABLE I
Comparison of memory allocations

Model	Allocations (GB)
own	31.37
Flux	62.9
Pytorch	-

In Julia, memory allocations were measured using the @allocated macro. For pytorch there was no way to measure allocations. As shown in table I, the custom solution has fewer allocations. The most impactful optimizations for Julia included the use of @view to avoid unnecessary data copying and the saving values between iterations to reduce redundant computations.

TABLE II
COMPARISON OF PERFORMANCE

Model	Epoch1	Epoch2	Epoch3	Epoch4	Epoch5
own	20.7	3.8	3.8	3.8	3.8
Flux	36.5	11	10.7	10.7	10.8
Pytorch	6.2	4.7	4.8	5	4.8

C. Performance

From Table II, we observe that the first epoch for the Julia-based models is significantly longer than the subsequent ones. This is due to Julia's use of just-in-time (JIT) compilation. Pytorch model achieved better performance than flux model, primarily because it relies heavily on numpy and is well optimized.

D. Accuracy

TABLE III
COMPARISON OF ACHIEVED ACCURACY

Model	Accuracy(train)	Accuracy(test)
own	0.95	0.87
Flux	0.95	0.87
Pytorch	0.94	0.88

All models achieved simillar accuracy as shown in table III. The test accuracy is nearly 90%, indicating that the custom library is correctly implemented.

VII. CONCLUSION

The developed solution achieves high accuracy, comparable to the reference implementations. Compared to the Flux module it is faster, primarily due to optimizations tailored for tested model architecture and improved parallelization, but it is not as flexible.

One of the main drawbacks of the created library is its limited functionality. While it supports CNNs and MLPs, it does not offer much beyond these basic architectures. Additionally, certain features used in CNN, such as dilation in convolutions are not implemented. Developing and optimizing a library is time-consuming and complex process. Compared to well-established libraries that have been refined over many years by large teams of developers, a solution built by a single person cannot match their level of versatility.

Moreover, if the library were to be used by others, it would need to support GPU compilation. Although CPUs are better for general-purpose tasks, GPUs offer massive parallel processing capabilities, allowing them to perform neural network computations many times faster. Modern deep learning networks rely heavily on GPUs and, in many cases, utilize clusters of hundreds of them.

In Julia, GPU support can be added relatively easily using the CUDA.jl package. However, this must be done with care. In the case of the referenced network, enabling GPU acceleration may not lead to performance improvements and could even result in slower execution. While computations on the GPU are generally faster, they require transferring data between the CPU and GPU, which introduces overhead. Due to batching, the matrices involved in each computation step are relatively small, for references structure approximately 130×50×64, making the data transfer cost potentially outweigh the benefits of GPU acceleration.

Another possible step in the library's development is making it open-source. There is a limit to what one person can achieve, and for the library to compete with others, it would need many more features.

REFERENCES

[1] L. B. Rall and G. F. Corliss, "An introduction to automatic differentiation," Computational Differentiation: Techniques, Applications, and Tools, vol. 89, pp. 1–18, 1996.

[2] J. Revels, M. Lubin, and T. Papamarkou, "Forward-mode automatic differentiation in Julia," arXiv preprint arXiv:1607.07892, 2016.

[3] S. Grøstad, "Automatic Differentiation in Julia with Applications to Numerical Solution of PDEs," Master's thesis, NTNU, 2019.

[4] Roger Grosse, *CSC321 Lecture 10: Automatic Differentiation*, https://www.cs.toronto.edu/~rgrosse/courses/csc321_2018/slides/lec10.pdf, accessed: March 11, 2024.

[5] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, "Automatic Differentiation in Machine Learning: A Survey," *Journal of Machine Learning Research*, 2018.