Reverse-mode automatic differentiation in Julia

Wiktor Kepiński
Faculty of Electrical Engineering
Warsaw, Poland
wiktor.kepinski.stud@pw.edu.pl

***Abstract*—In the field of machine learning, efficient gradient computation is crucial for optimizing neural networks. Reverse-mode automatic differentiation (AD) has become an essential tool for this purpose. This paper explores the implementation of reverse-mode AD in the Julia programming language, focusing on its application to recurrent neural networks (RNNs). A custom Julia AD package was developed and compared with the Flux.jl library in Julia and the PyTorch library in Python. The study employs comprehensive benchmarking to evaluate the performance, memory allocation, and training accuracy of each implementation. This approach allows for the identification of the advantages and trade-offs associated with each solution.**

***Index Terms*—Reverse-mode automatic differentiation, Julia, RNN**

I. Introduction

Automatic differentiation (AD) is a set of techniques used to evaluate the derivatives of functions specified by computer programs. Unlike numerical differentiation, which approximates derivatives using finite differences, or symbolic differentiation, which manipulates mathematical expressions, AD computes exact derivatives efficiently through algorithmic means [1]. AD is based on the way computers solve complex equations, namely decomposition into individual operations such as $+ - /*$ and basic functions such as $\sin$, $\cos$, $\exp$ etc. The resulting decomposition implements a chain rule. The chain rule can be employed in two distinct ways, resulting in two distinct forms of AD: forward mode and reverse mode [3].

The role of AD has expanded in parallel with the advancement of machine learning (ML), where the structures of artificial neural networks (ANNs) have become increasingly complex, with thousands of parameters influencing the final result. AD enables the determination of the derivatives of complex expressions such as activation functions, due to the breakdown into single operations. Subsequently, AD enables the determination of the impact of each parameter on the ANN error, which can then be optimised through gradient algorithms, such as gradient descent, to update the parameter weights.

This paper will concentrate on the reverse mode method, as it is more accurate when there are more input variables than output variables. This is in contrast to the forward method [4].

The objective of this paper is to create a own package for reverse mode automatic differentiation in the Julia programming language and to compare it to the Flux [5] and PyTorch [9] libraries. The Julia programming language has rapidly gained popularity in the scientific and technical computing communities due to its unique combination of performance and ease of use. Julia is designed to bridge the gap between low-level languages like C, which offer high performance, and high-level languages like Python, which are easy to write and understand [2]. The functionality of the prepared package will be evaluated using the example of an RNN [10] trained on the MINST set [11].

Recurrent neural networks (RNNs) represent a type of artificial neural network that is particularly well-suited for processing sequential data. In contrast to traditional feedforward neural networks, RNNs possess connections that form cycles, thereby enabling them to maintain a form of memory of previous inputs. This capability renders RNNs effective for a wide range of applications, including time series forecasting, natural language processing, and speech recognition. The training of RNNs, however, presents a challenging task in that it involves computing gradients through time. This is efficiently handled by reverse-mode AD [10].

Package implementations can be achieved through two distinct methodologies: automatic code generation or computational graph creation. In this work, the computational graph approach is selected. This approach comprises several key elements. Upon receipt of the problem posed by the MINST set, the initial step is to construct a computational graph. Subsequently, in a single pass of the graph, the values at each node are calculated using the forward method. This is followed by the reverse method, which is employed to calculate the gradient in accordance with the loss function.

The final stage of the process before comparing the obtained results will be the optimization of the algorithm, which will be carried out using the capabilities of the Julia environment. These include the ability to create multiple function definitions, which is known as multiple dispatch. Additionally, mathematical methods for neural networks will be employed, such as the appropriate initialization of weights, which can be achieved through techniques such as Xavier's method [8]. Furthermore, the prevention of gradients from exploding through clipping will be considered [14].

II. Automatic differentiation

The paper employs the technique of automatic differentiation (AD) with back propagation utilising the graph method. This method involves the construction of a computational graph in which nodes represent operations and edges represent data flow. The process comprises two graph transitions. Initially, a forward transition is executed to calculate the value of a function. Subsequently, a backward pass is conducted to

calculate the gradient of the loss function with respect to each parameter, utilising the chain rule. By applying the chain rule in a systematic manner, back-propagation enables gradients to propagate backward through the network, thus facilitating the optimization of model parameters through gradient descent. The structured approach of the graph method ensures that intermediate values are stored during the forward transition, allowing for accurate and efficient gradient calculations during the backward transition. The paper distinguishes three basic types of graph nodes: Constant, Variable, Operator. The Constant node contains only value information, the Variable node stores information about its value and gradient, and the Operator node represents a function acting on its arguments. The algorithm employed the multiple dispatch mechanism of the Julia language, which greatly simplified the process of dynamic graph generation.

III. Artificial Neural Network

Recurrent neural networks (RNNs) are a class of neural networks designed to recognize patterns in sequences of data, such as time series or natural language. The case study, namely the recognition of handwritten digits, is not an optimal application for a network of this type. However, research indicates that satisfactory accuracy results can be obtained. Unlike feedforward neural networks, RNNs have connections that form directed cycles, allowing them to maintain a hidden state that captures information from previous time steps. This allows RNNs to exhibit dynamic temporal behavior, rendering them particularly well-suited for tasks where context or sequence dependence is crucial. Each neuron in an RNN receives input not only from the current data point, but also from a hidden state from the previous step, effectively creating a memory of previous input data. The term backpropagation through time (BPTT) is associated with the learning of recurrent networks, although it is the same algorithm that is used in non-recursive neural networks. The hidden states and outputs are updated recursively, allowing the network to learn and model complex temporal patterns. The study employed a classic form of RNN, the vanilla RNN, which comprises four recursive states with a $\tanh$ activation function. Two issues are associated with learning recurrent networks: a vanishing gradient and an exploding gradient [14]. The exploding gradient was encountered in the study. The solution involved the addition of clipping functions of the calculated gradient values by the backpropagation algorithm, in addition to the use of Xavier's method [8] to initialize the weights.

IV. Implementation

Julia, with its high-performance capabilities, allows for the efficient implementation of these processes using a computational graph.

A computational graph is a directed acyclic graph (DAG) [13] where nodes represent operations or variables, and edges represent dependencies between these operations. This graphical representation is crucial because it visually and structurally organizes the sequence of operations, making it easier to trace and debug computations. In a computational graph nodes represent various computational entities, such as constants, variables, or operations. Edges represent the data flow between these nodes, indicating the dependency of one node's output on another node's input. The primary advantage of using a computational graph is its ability to handle complex chains of operations efficiently, facilitating both the forward and backward passes required in automatic differentiation and neural network training.

Nodes in the computational graph are categorized into three primary types: Constant, Variable, and Operator.

Constant nodes represent a fixed value in the computation and do not change during training or differentiation.

Variable nodes represent values that can change and for which gradients must be computed. Variables are typically the parameters or inputs to the model that are optimized.

Operator nodes represent functions or operations that take other nodes as inputs and produce an output. A node of this type represents a function or operation that takes other nodes as inputs and produces an output. This could be any mathematical operation, such as addition, multiplication, or more complex functions like activation functions in neural networks.

These definitions help in structuring the computational graph and delineating the roles of different types of nodes, which is crucial for both forward and backward propagation.

The next essential step in processing a computational graph is topological sorting. It involves ordering the nodes in such a way that each node appears before any nodes it points to (its dependencies). This ordering is crucial because it ensures that when performing forward propagation, each node's inputs are already computed by the time the node itself is processed. Similarly, during backward propagation, each node's gradients can be propagated back to its inputs correctly. Topological sorting prevents cycles in the graph, ensuring that the graph is a directed acyclic graph (DAG). This step is critical for the correctness and efficiency of the forward and backward passes.

Before starting the backward propagation, it is necessary to reset the gradients of all nodes in the graph. This step ensures that gradients from previous iterations do not interfere with the current computation. In neural network training and automatic differentiation, gradients are accumulated over iterations. Without resetting these gradients, the accumulation from previous iterations can lead to incorrect updates, which might hinder or even prevent convergence. Resetting gradients involves setting the gradient attribute of each node (except constants) to nothing or zero. This prepares the graph for a fresh round of gradient computation.

The forward pass involves traversing the graph in topological order and computing the output of each node based on its inputs.

With regard to *constants* and *variables*, the output is typically the value itself. In contrast, for *operators*, the output is computed based on the values of its input nodes and the specific operation it represents (e.g., addition, multiplication). The forward pass is crucial for generating the final output of

the computational graph, which is then used to compute the loss or the objective function. In this instance, the loss function is cross-entropy loss, which is implemented separately for the forward and backward passes. This output serves as the basis for the backward pass, where gradients are computed.

In the process of developing the model, the phenomenon of exploding gradients was encountered. In order to minimise the impact of this phenomenon, a method called gradient clipping was employed. Gradient clipping is a technique used to prevent the gradients from becoming too large during backpropagation, which can cause instability in the training process. Large gradients can lead to issues such as exploding gradients, where the model parameters can become excessively large and destabilize the learning process. Exploding gradients is a phenomenon that is particularly common in the implementation of recurrent neural networks (RNNs). Clipping gradients involves capping the gradients at a maximum and minimum threshold value. If a gradient exceeds this value, it is set to the threshold value. This technique helps maintain stable training and can improve the convergence properties of the model. In the context of a computational graph, gradient clipping is applied to the gradients of all relevant nodes after they have been computed in the backward pass.

The backward pass, or backpropagation, is the process of computing the gradients of the output with respect to each input by traversing the graph in reverse topological order. The gradient of the output node is typically initialized to a seed value of 1.0. This value represents the output of the loss function with respect to the loss function. Then, for each node, the gradient is propagated backward, starting with the last node and moving toward the initial node. This process involves computing the partial derivatives of the node's output with respect to its inputs and multiplying by the gradient of the output node. It should be noted that *constants* do not require gradient updates. In contrast, *variables* undergo gradient computation and accumulation if they already possess gradients. *Operators* are subject to gradient computation based on their inputs, which are then propagated back to each input node. The backward pass is crucial for training neural networks and performing optimization, as it provides the necessary gradients to update the model parameters.

The final stage of the process is to update the weights of the neural network model.

V. Optimalization

Julia is designed for numerical and computational science and engineering, and it provides a sophisticated compiler, distributed parallel execution, numerical accuracy, and an extensive mathematical function library. One of the most compelling features of Julia is its ability to combine the ease of writing high-level code with the performance of low-level languages like C or Fortran. This combination is achieved through several optimization strategies [2].

Julia uses the LLVM-based Just-In-Time (JIT) compiler which allows it to generate optimized machine code on the fly. This compilation step happens the first time a function is called, enabling subsequent calls to be extremely fast. This approach provides both the flexibility of dynamic languages and the speed of statically compiled languages. While Julia is dynamically typed, specifying types can significantly enhance performance. Type declarations help the compiler generate more efficient code by reducing the overhead of type inference. The paper adheres to this principle by ensuring that functions possess specific types of variables.

Another Julia mechanism for creating optimal code is multiple dispatch. This is a core Julia feature that allows the selection of method implementations based on the types of all function arguments. This can be used to write generic code that remains efficient. Multiple dispatch ensures that the most specific method is selected for execution, optimizing performance.

Julia offers a comprehensive set of macros designed to enhance code efficiency and facilitate in-place operations. The macros utilized in this work include @inbounds, which omits the necessity of verifying whether the index is within the bounds of the matrix, and @views, which generates a view of the matrix without the need to copy and allocate a new matrix. It is of vital importance to utilise memory efficiently in order to achieve rapid code execution. Julia has a built-in garbage collector that is responsible for the allocation and deallocation of memory. However, manually counter-checking memory allocation, as exemplified by the aforementioned macros, can further enhance code performance.

Benchmarking tools are essential for the identification of performance bottlenecks. Julia provides a suite of powerful tools, including @time and @btime macros from the BenchmarkTools.jl and Base.jl packages, which are used to measure and improve code performance. Additionally, Julia offers packages such as JET or ProfileView, which are employed to visualize the flow of information in code and assist in the identification of so-called bottlenecks, where code runs slower.

VI. Results

This section presents the results obtained in comparison with popular AD solutions in the form of the Flux library for Julia and PyTorch of the Python environment. The same conditions were prepared for each of these variants, including the network structure, learning loop parameters, and optimizer.

The learning network has a vanilla RNN structure, consisting of a recursive part with a $\tanh$ activation function and input dimension 196 and output dimension 64, and a dense layer with an $\mathrm{identity}$ activation function and output in the form of a 10-element vector with class scores. In this case, the digits $0:9$ were used as the class types.

The loss function in each case was $\mathrm{logitcrossentropy}$ which is a numerically stable combination of $\mathrm{crossentropy}$ and $\mathrm{softmax}$ functions [5].

The learning function comprised a primary loop of five epochs and an inner loop responsible for training the network in the form of a batches of size 100. Following each batch, the weights of the network were updated, and at the end of each epoch, the accuracy on the learning and test set was calculated.

The network weights were updated based on a simple gradient descent algorithm with a learning step of $15 \cdot 10^{-3}$.

To evaluate the performance of the learning function, the following measures were employed: the accuracy on the learning and test sets after each epoch, the amount of allocated memory and the execution time after each epoch, as well as the allocated memory and time of the entire train function.

In the table below, AD Package symbolizes the author's implementation discussed in the article.

Epoch	Measures	Flux	PyTorch	AD Package
1	Train accuracy (%)	89.35	83.0	87.57
	Test accuracy (%)	90.46	79.23	87.59
	Allocated memory (GiB)	4.02	0.003	3.32
	Time (s)	14.35	16.96	4.85
2	Train accuracy (%)	92.03	87.0	90.93
	Test accuracy (%)	92.38	87.13	90.93
	Allocated memory (GiB)	2.63	0.003	3.09
	Time (s)	1.01	16.63	1.76
3	Train accuracy (%)	92.99	91.0	92.22
	Test accuracy (%)	93.15	90.3	91.73
	Allocated memory (GiB)	2.63	0.003	3.09
	Time (s)	0.95	17.4	1.41
4	Train accuracy (%)	93.87	93.0	92.85
	Test accuracy (%)	94.02	90.31	92.6
	Allocated memory (GiB)	2.63	0.003	3.09
	Time (s)	1.01	17.33	1.45
5	Train accuracy (%)	94.4	93.0	92.78
	Test accuracy (%)	94.55	91.24	92.47
	Allocated memory (GiB)	2.63	0.003	3.09
	Time (s)	1.28	18.0	1.45
	Allocated memory (GiB)	17.80	0.003	23.04
	Summary time (s)	22.45	89.32	15.68

TABLE I
COMPARISON WITH EXISTING SOLUTIONS

The allocated memory values and time in the case of the Julia language were measured with the @time macro, and in the case of Python, the time and tracemalloc modules [12] were employed.

After five epochs of learning, the highest accuracy on the test set was achieved by Flux, followed by the AD Package implementation in Julia and PyTorch. It is noteworthy that the initial epoch in the case of Julia exhibited the lowest performance and the highest memory usage. This is attributed to the necessity of compiling the entire training function. The larger memory allocation observed in the Flux solution relative to the AD Package is attributed to the more complex library. The PyTorch solution exhibited the lowest performance but the lowest memory usage. This may be attributed to the characteristics of the Python language, which is an interpretable language.

The AD Package implementation exhibited the fastest runtime, although it allocated the largest amount of memory. However, when considering only the accuracy results of the trained network, the author's implementation demonstrated no significant deviation from the presented solutions.

VII. CONCLUSION

Automatic Differentiation is a crucial component of the advancement of neural networks, machine learning, and artificial intelligence (AI). The computational graph method represents a convenient and transparent approach to its implementation. Julia represents an intriguing alternative to existing machine learning (ML) solutions, which are primarily in Python. It offers numerous advantages, including multiple dispatch and developed usage tracking tools. However, it is characterized by fast operation, which comes at the expense of the amount of memory used. This trade-off between execution speed and memory allocation is an important consideration for selecting the appropriate tool for specific machine learning tasks. The prepared implementation demonstrated competitive performance in the studied case.

Future work will focus on further optimizing memory management and expanding the capabilities of the AD package to support a wider range of neural network architectures and machine learning applications.

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.

[5] M. Innes, 'Flux: Elegant machine learning with Julia', JOSS, vol. 3, no. 25, p. 602, May 2018, doi: 10.21105/joss.00602.

[6] C. C. Margossian, 'A review of automatic differentiation and its efficient implementation', WIREs Data Min & Knowl, vol. 9, no. 4, p. e1305, Jul. 2019, doi: 10.1002/widm.1305.

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

[8] Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." Proceedings of the thirteenth international conference on artificial intelligence and statistics. 2010.

[9] Automatic differentiation in PyTorch, Paszke, Adam and Gross, Sam and Chintala, Soumith and Chanan, Gregory and Yang, Edward and DeVito, Zachary and Lin, Zeming and Desmaison, Alban and Antiga, Luca and Lerer, Adam,2017.

[10] Alex Sherstinsky, Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) network, http://dx.doi.org/10.1016/j.physd.2019.132306

[11] Deng, L., 2012. The mnist database of handwritten digit images for machine learning research. IEEE Signal Processing Magazine, 29(6), pp.141–142.

[12] Van Rossum, G. , Drake, F.L., 2009. Python 3 Reference Manual, Scotts Valley, CA: CreateSpace.

[13] Digitale JC, Martin JN, Glymour MM. Tutorial on directed acyclic graphs. J Clin Epidemiol. 2022;142:264-267. doi:10.1016/j.jclinepi.2021.08.001

[14] Pascanu, Razvan, Tomas Mikolov and Yoshua Bengio. "On the difficulty of training recurrent neural networks." (2012).