Implementation of Automatic Differentiation in Julia

Daniel Stańkowski
Faculty of Electrical Engineering
Warsaw University Of Technology
Warsaw, Poland
01159493@pw.edu.pl

Tomasz Gryczka
Faculty of Electrical Engineering
Warsaw University Of Technology
Warsaw, Poland
01159252@pw.edu.pl

***Abstract*—This article delves into implementing reverse-mode automatic differentiation in a CNN neural network for digit image classification with the MNIST dataset. It explores the strengths and limitations of reverse-mode automatic differentiation and techniques for optimizing its efficiency. Computational graphs simplify derivative operations via the chain rule. Leveraging Julia's automated formula translation and performance capabilities ensures effective implementation, highlighting Julia's importance in fast, efficient automatic differentiation algorithms.**

***Index Terms*—automatic differentiation, backpropagation, forward mode automatic differentiation, machine learning, Julia**

I. Introduction

A. Automatic Differentiation

Automatic differentiation (AD) is a computational technique for efficiently and accurately evaluating derivatives of functions [1]. Unlike numerical differentiation methods, which approximate derivatives through finite differences and suffer from numerical errors [2], automatic differentiation computes derivatives with machine precision [3]. It does so by decomposing complex functions into a sequence of elementary operations, for which derivatives are known, and then applying the chain rule recursively [4].

Automatic differentiation (AD) employs two primary modes: forward mode and reverse mode. These modes offer distinct approaches to efficiently compute derivatives of functions, catering to different computational needs and structures [5].

Forward mode, evaluates the derivative of a function by propagating derivatives forward through the computational graph from inputs to outputs. At each intermediate operation, the derivative of the function with respect to each input variable is computed recursively using the chain rule. This mode is particularly useful when the function has a small number of inputs but a large number of outputs, as it computes the derivative with respect to each input individually. [6]

Reverse mode, also known as backward mode or backpropagation, computes the derivative of a function by traversing the computational graph from outputs to inputs. It efficiently computes gradients of the function with respect to multiple variables simultaneously by leveraging the chain rule and the concept of adjoints. In reverse mode, the computational graph is traversed in a bottom-up fashion, computing the derivatives of intermediate variables with respect to the output variables. This process involves storing intermediate values and gradients during the forward pass and then efficiently computing gradients during the backward pass using the chain rule [7]. Reverse mode is particularly well-suited for functions with a large number of inputs but a small number of outputs, such as neural networks in deep learning.

B. Julia

Julia has emerged as a powerful language for scientific computing, offering both the flexibility of high-level scripting languages and the performance of low-level languages. Developed with a strong emphasis on numerical computing, Julia combines the ease of use of languages like Python with the speed of languages like C or Fortran. One of Julia's key features is its just-in-time (JIT) compilation, which allows it to achieve performance comparable to statically compiled languages [8]. This performance is crucial for automatic differentiation, where efficiency is paramount for handling large-scale optimization problems. Julia's type system enables developers to write generic code that is both efficient and abstract [8]. This flexibility is particularly advantageous for automatic differentiation, as it allows for the creation of generic algorithms that can operate on a wide range of data types and structures.

II. Computational graph

The training of neural networks requires the use of an automatic differentiation technique. To achieve this, automatic differentiation with backpropagation was chosen. This approach has been integrated into many popular machine learning frameworks such as TensorFlow, Torch and others [9]. Backpropagation often uses a forward and backward pass in the computational graph to compute gradients.

A computational graph is a directed graph where nodes symbolize mathematical operations, and edges between nodes illustrate the data (or tensors) flowing from one operation to another. It is a way to visually represent and compute mathematical expressions [10]. Computational graphs are a fundamental concept in the implementation of neural networks because they provide a structured and efficient way

to represent and compute complex network architectures. The specific graph illustrating this concept is depicted in Figure 1, representing the function defined in Equation 1.

$$f(x, y, z) = (x + y) \times z \tag{1}$$

To calculate the derivative of a function (compute gradients in a neural network), two passes are performed: forward pass and backward pass.

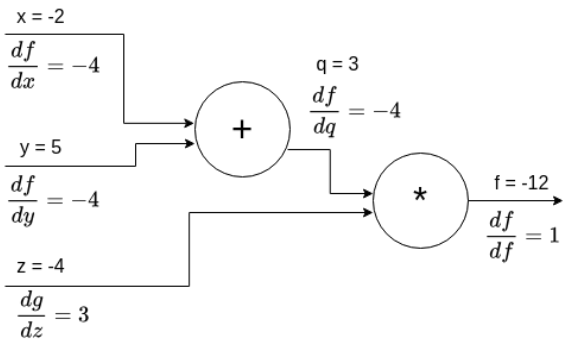

Fig. 1. Simple computational graph representing equation 1

A. Forward pass

The forward pass in a computational graph is the process of computing the output of the graph given a specific set of input values. This method involves moving from the input nodes through the graph's operations (nodes) to ultimately produce the output nodes. During the forward pass, input values are propagated forward through operations, with each node applying its function to the input it receives, generating output that serves as input for subsequent nodes [11]. This sequential computation continues layer by layer until the final output is computed. The forward pass essentially captures the flow of information through the graph, transforming inputs into predictions or outputs. In the example graph depicted in Figure 1, three inputs (x = -2, y = 5, z = -4) were used, and an output (f) of -12 was obtained. Intermediate values and operations computed during the forward pass are stored and utilized during the backward pass.

B. Backward pass

During the backward pass (backpropagation), which is part of automatic differentiation, the graph is traversed in reverse order starting from the output node(s) back towards the input nodes. The process involves applying the chain rule of calculus iteratively to compute gradients at each node [12]. The backward pass starts with the computation of the derivative of the final output relative to itself. This differentiation yields an identity derivative, equating to a value of one. Subsequently, the backward pass through the multiplication operation will be conducted, during which gradients will be calculated for q and z. Furthermore, since f is equal to $q \times z$, it can be inferred that $\frac{df}{dz} = q = 3$ and $\frac{df}{dq} = z = -4$, because the values of z and q are already known from the forward pass. The next step is to calculate gradients at x and y. To calculate these values efficiently, the chain rule of differentiation will be employed and the derivatives presented in equation 2 and 3 are obtained.

$$\frac{df}{dx} = \frac{df}{dq} \cdot \frac{dq}{dx} = -4 * 1 = -4 \tag{2}$$

$$\frac{df}{dy} = \frac{df}{dq} \cdot \frac{dq}{dy} = -4 * 1 = -4 \tag{3}$$

The exact same method is utilized when computing gradients in a neural network that is constructed as a computational graph. Once the gradients of the loss function with respect to each parameter have been computed, these gradients can be used to update the parameters in the direction that minimizes the loss function [13].

III. Optimization

The performance of the neural network was improved by modifying the implementation of certain operations responsible for its learning process. This led to a 3-fold reduction in network training time, as well as approximately a 2.5-fold decrease in overall memory consumption by the neural network.

A. Convolution Operation

Notable advancement came with the transition from naive convolution implementation using nested loops to more efficient method, that leverages matrix multiplication. Originally, convolution operation in CNN was performed by iterating through the input data using nested loops, applying a filter to extract features. However, this approach was computationally intensive, especially on large datasets.

The introduction of the "im2col" technique transformed this process by reshaping the input data into a matrix format [14]. The input data is divided into patches corresponding to the receptive field of the kernel. Next, these patches are rearranged into columns, forming a new matrix where each column represents a receptive field as a flattened vector. Simultaneously, the filter is transformed into a row vector. The convolution operation then becomes a matrix multiplication between this reshaped input matrix and the flattened filter, resulting in an output matrix of convolved features.

By replacing nested loops with matrix multiplication, the "im2col" algorithm significantly accelerates convolution operations in CNNs. This approach benefits from optimized linear algebra libraries, making CNN training and inference more efficient and scalable [15].

B. Vectorized operations and views

To optimize the code, element-wise operators were utilized where applicable. Julia's unique dot syntax was leveraged to convert scalar functions into vectorized function calls and operators into vectorized operations. This dot syntax enables the fusion of nested "dot calls" into a single loop at the syntax level, eliminating the need for temporary array allocations [16]. By using . = and similar assignment

operators, results can be stored in-place in a pre-allocated array. In the realm of linear algebra, although operations like vector addition $(vector + vector)$ and scalar multiplication $(vector * scalar)$ are defined, it can be more advantageous to utilize $vector. + vector$ and $vector. * scalar$ instead. This approach allows the resulting loops to fuse with surrounding computations, potentially enhancing performance.

Another performance optimization involved using views for array slices. Normally, when a slice expression like $array[1:5,:]$ is used, a copy of the data is created, which can lead to inefficiency for multiple operations due to allocation costs [17]. Instead, the creation of a "view" of the array using view() or @views (for a whole expression or block of code) enables direct referencing of the original array's data without copying. Modifications to a view also result in reflections in the original array's data, rendering this approach memory-efficient and performance-enhancing for complex operations on slices.

C. Graph building

Previously, the graph was routinely rebuilt with new values of input data and target outputs for each training iteration. Transitioning to a strategy where the graph is initialized once and then dynamically updated with new input and target values during each iteration allows us to leverage static optimizations and minimize the costs associated with frequent graph creation. The introduction of a more efficient method involves constructing the graph once with constant nodes that hold input and output data, respectively. During training, these constant nodes are dynamically updated with fresh input-output pairs while maintaining the same graph structure. This optimization reduces unnecessary graph reconstruction, contributing to enhanced computational efficiency.

IV. Model Comparison

The performance and efficiency of the convolutional neural network architecture were evaluated using both Python with PyTorch and Flux referencing solutions. Both implementations were assessed under identical learning parameters and datasets to ensure a fair comparison.

A. Model architecture

The architecture employed in all reference models follows a standard structure commonly used in convolutional neural networks for image classification tasks. It comprises several layers designed to progressively extract features from input images and make predictions. These layers include convolutional layers, which apply learnable filters to input images to extract features, followed by pooling layers that reduce spatial dimensions to capture important features efficiently. Additionally, flattening is applied to reshape the output from previous layers into a one-dimensional vector, facilitating input to fully connected layers responsible for classification. These fully connected layers use non-linear activation functions to introduce complexity and learn representations from the extracted features. Finally, the output layer generates class predictions based on the learned representations. Visual repsentation of the architecture can be seen in Figure 2.

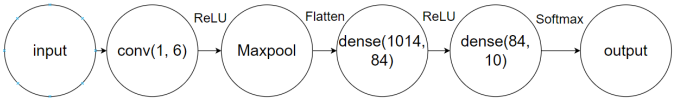

Fig. 2. Model architecture

B. Flux reference model

The Flux-based model demonstrates the capabilities of this machine learning library within the Julia ecosystem. With Flux's straightforward syntax, defining neural network architectures becomes intuitive, facilitating the construction of complex models. Leveraging Flux's modular design, the model seamlessly integrates within a Chain module, allowing for the sequential composition of layers tailored to diverse tasks. The MNIST dataset was loaded and preprocessed, with training and test data reshaped to include a trivial channel dimension.

C. PyTorch reference model

PyTorch provides developers with a flexible framework for building neural networks, offering ease in creating intricate architectures. Leveraging PyTorch's modular approach, developers can stack layers sequentially, customizing networks for specific applications. PyTorch's dynamic computational graph enhances efficiency during training and inference, providing developers with flexibility in model design and experimentation. The MNIST dataset was loaded and transformed similarly to the Flux approach.

D. Comparison

The comparison of performance metrics across the three models, Flux, PyTorch, and our implementation, provides valuable insights into their respective efficiencies. Flux demonstrates the fastest execution time, followed closely by PyTorch and then our own implementation. Flux showcases moderate memory usage, whereas our model appears to utilize significantly more memory. Unfortunately, PyTorch does not provide any information about memory usage, but given its strong optimization and underlying implementation in C, it likely performs comparably to Flux. On the bright side, our implementation achieved the highest accuracy, a trend observed consistently during testing. It's important to note that our model was developed from scratch by two individuals, whereas the other two solutions are maintained by teams of experts. For a detailed breakdown of performance metrics, refer to Table I.

TABLE I
COMPARISON OF MODEL PERFORMANCE

Model	Time (s)	Allocations (GiB)	Accuracy (%)
Flux	33.5	15.80	94.67
PyTorch	36.9	-	94.59
Own	168.7	250.632	95.59

V. CONCLUSION

In conclusion, this article has delved into the implementation of reverse-mode automatic differentiation in a convolutional neural network (CNN) for digit image classification using the MNIST dataset. By harnessing Julia's capabilities in automatic differentiation and optimization techniques, we have demonstrated the power of computational graphs in simplifying derivative operations. Julia's performance and flexibility have been showcased in handling large-scale optimization problems, underscoring its importance in fast and efficient automatic differentiation algorithms.

Furthermore, the comparison of performance metrics across different implementations, including Flux, PyTorch, and our own, was not aimed at achieving competitive performance. Instead, it provided valuable context for exploring Julia's capabilities in machine learning. While our implementation may have lagged behind established frameworks in execution time and memory usage, the primary objective was to gain insights into developing and optimizing machine learning models in Julia.

This exploration has revealed valuable lessons about leveraging Julia's features for machine learning tasks, highlighting its potential for developing custom solutions. Moving forward, these findings pave the way for further research and optimization within this domain, solidifying Julia's position as a language of choice for innovative machine learning solutions.

REFERENCES

[1] Louis B. Rall, George F. Corliss, "An Introduction to Automatic Differentiation"

[2] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, 'Automatic Differentiation in Machine Learning: a Survey'.

[3] Arun Verma, "An introduction to automatic differentiation"

[4] 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.

[5] Christian H. Bischof, H. Martin Bucker, "Computing derivatives of computer programs"

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

[7] 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.

[8] 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.

[9] Ch. Yang, Y. Deng, J. Yao, Y. Tu, H. Li and L. Zhang, 'Fuzzing Automatic Differentiation in Deep-Learning Libraries', 2023.

[10] A. Savine, 'Computation Graphs for AAD and Machine Learning Part I: Introduction to Computation Graphs and Automatic Differentiation', 2018.

[11] A. Kohan, E. A. Rietman, and H. T. Siegelmann, 'Signal Propagation: The Framework for Learning and Inference in a Forward Pass', 2023.

[12] X. Zhou, W. Zhang, Z. Chen, S. Diao, T. Zhang, 'Efficient Neural Network Training via Forward and Backward Propagation Sparsification', 2021.

[13] J. Lee and G. AlRegib, Gradients as a Measure of Uncertainty in Neural Networks, 27th IEEE International Conference on Image Processing (ICIP), Abu Dhabi, United Arab Emirates (UAE), 2020.

[14] Y. Zhang and X. Li, Fast Convolutional Neural Networks with Fine-Grained FFTs, 2020.

[15] G. Alaejos, A. Castello, P. Alonso-Jorda, Enrique S. Quintana-Orti 'Convolution Operators for Deep Learning Inference: Libraries or Automatic Generation?', 2021.

[16] J. Bezanson, J. Chen, B. Chung, S. Karpinski, V. B. Shah, J. Vitek, L. Zoubritzky, 'Julia: Dynamism and Performance Reconciled by Design', 2018.

[17] A. Rahman. D. Brinto, R. Shakya, R. Pandita Come for Syntax, Stay for Speed, Understand Defects: An Empirical Study of Defects in Julia Programs