Implementing Efficient Automatic Differentiation in Julia for Neural Network Training: A Case Study with MNIST Classification

Mikołaj Zawada
Faculty of Electrical Engineering
Warsaw University of Technology
plac Politechniki 1, 00-661 Warsaw

Michał Szlachta
Faculty of Electrical Engineering
Warsaw University of Technology
plac Politechniki 1, 00-661 Warsaw

***Abstract*—This article presents an implementation of a custom dictionary for Reverse-Mode Automatic Differentiation tailored for Convolutional Neural Networks in Julia. The primary aim is to achieve superior speed, learning accuracy, and minimal memory usage through various optimization techniques. The implementation is rigorously compared against a reference PyTorch program, evaluating performance metrics. The findings of the study underscore the potential of Julia's for efficient deep learning implementations.**

***Index Terms*—Julia programming language, CNN, Automatic Differentiation, MINIST Classification**

I. Introduction

In this study, we explore the integration of automatic differentiation (AD) and neural network training, employing the Julia Programming Language as a foundational tool for developing our computational library. Julia marks a progression in numerical computing by bringing the previously distinct aspects of user friendly high level functionality and low-level computational efficiency. Developed at MIT by Bezanson, Edelman, Karpinski and Shah Julia questions the belief that dynamically typed language has to compromise speed, for flexibility. It accomplishes this by following a design approach that caters, to both the requirements of the machine and the programmer. [1]

Julias' high efficiency alignes with the need for optimized calculations of derivatives when working with large neural networks. There are a number of known solutions to perform differentiation by a machine. The distinction includes finite differentiation, symbolic differentiation and automatic differentiation. However of all those three, automatic differentiation (that instead of focusing on producing the expression of the derivative focuses on calculating its' numerical value) is the only one efficient enough to be considered for the task. [2] Following this idea we can delve deeper into the topic of automatic differentiation.

When performing automatic differentiation we have to consider two modes, forward and reverse. Forward mode identifies the most elementary operations in a function and based on those traverses the function and calculates primals and tangents for each intermediate operations until it returns the sought value (the derivative with respect to one specified input). However forward mode is not a preferred option if we have a substantial amount of arguments to consider. Because in a single pass it calculates derivative on a single argument it is pretty trivial to notice the problem if we have thousands, millions or even billions arguments to consider which is the case in modern neural networks. [2] That's where reverse mode comes into play.

The reverse mode consists of 2 parts. First we perform a forward pass without calculating the derivatives. Then we compute the gradient of one of the the outputs with respect to all inputs in a reverse pass. That gives us the gradient in a single execution of the reverse mode auto differentiation. In context of neural networks, because the number of operations to be performed is now directly connected to the number of outputs, not inputs (thus drastically decreasing the amount of computation required) it's obvious this method will be our method of choice later in the case study.

Because our paper focuses on implementing a custom automatic differentiation framework for neural network training it is crucial that we implement the most efficient solution possible. To achieve this we will consider computation techniques to optimize automatic differentiation. The most basic technique is retaping which focuses on removing reevaluation of intermediate variables at each sweep. However this technique only works if the intermediate variables do not change between sweeps which is the case if we evaluate derivatives at multiple points. [3]

Another area worth optimizing is memory management especially when working with large expression graphs. Solution created to mitigate this issue is checkpointing. The main idea behind it is to divide the computation into sections, with "checkpoints" between each section. These checkpoints allow the program to store intermediate results, reducing the need to store the entire computational graph in memory at once. Region-based memory is another approach when increasing memory usage efficiency. It involves allocating a large block of memory at once, within which all the intermediate variables and necessary computational details are stored. This approach reduces the overhead associated with frequent memory alloca-

tion and deallocation. Once the computation is complete, the entire memory block can be freed in one action. [3]

II. STUDY DESCRIPTION

A. Goal

The purpose of this study was to implement a custom library for reversed automatic differentiation in Julia programming language. Before that we had to create a convolutional neural network that we could use our library on.

The architecture we chose is fairly simple since it does not use bias terms and consists of only 5 layers. The simplicity of it allows for faster and less computationally consuming training and without losing on the final accuracy.

B. Model description

The CNN is designed to work on classification problems that recognize gray-scale hand written numbers from 0 to 9. The dataset used in our study is called MNIST and the hand written numbers are 28x28 gray-scale images. Training of the network uses 60000 images and the test 10,000.

Architecture of the CNN consists of:

- Convolutional layer
- Max pooling layer
- Flattened layer
- Two dense layers
- Loss function

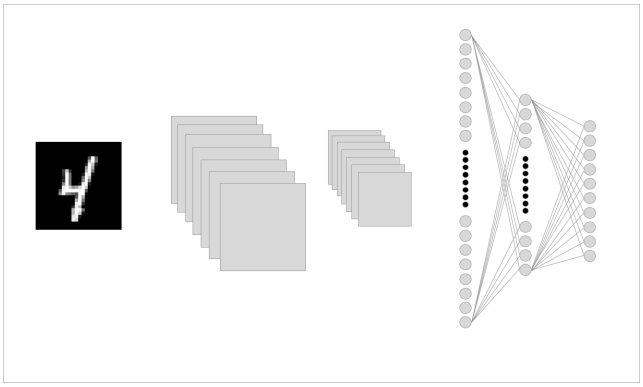

Fig. 1. Simplified visualization of the implemented network

The convolutional layer employs a 3x3 kernel to process inputs of dimensions 28x28x1, converting them from a single channel to six output channels. This layer uses the ReLU activation function and does not include a bias term.

Subsequently, a max pooling layer with a 2x2 window is utilized to down sample the feature maps, thereby reducing the computational load.

The flattening operation then converts the multi-dimensional feature maps into a one-dimensional vector, facilitating the transition from convolutional layers to dense layers.

The first dense layer comprises 84 neurons, without biases, and also uses the ReLU activation function.
The final layer is a dense layer consisting of 10 output units, corresponding to the number of target classes. The activation function for this layer is softmax, which transforms the raw outputs of the neural network into a vector of probabilities.

Finally, the loss is computed using the cross-entropy loss function described by the following formula:

$$-\sum_{c=1}^{M} y_{o,c} \log(p_{o,c})$$

III. IMPLEMENTATION

A. Structures

Our implementation revolves around structures that construct the computational graph. In this graph, nodes represent variables, and edges represent operations between these variables.

- **GraphNode**: The base abstract type for all nodes in the graph.
- **DescentMethod**: An abstract type representing method for updating weights during optimization.
- **Operator**: An abstract type for nodes that perform operations on inputs.
- **Constant**: A structure representing constant values within the graph.
- **Variable**: A generic mutable structure representing variables whose values can change and have associated gradients.
- **ScalarOperator and BroadcastedOperator**: Mutable structures representing scalar and broadcasted operations that can be performed on nodes.

B. Operators and overloading

To construct a computational graph, we employ the `topological_sort` function. This function takes the loss function as input, constructs the graph using the depth-first search (DFS) algorithm, and returns the nodes in topological order. The topological order ensures that each node is processed only after all its dependencies have been processed, which is critical for correctly performing forward and backward passes.

To perform reverse automatic differentiation, we define operators, which are functions that dictate how mathematical operations and functions interact with graph nodes. Each operator is equipped with a `forward` function and a `backward` function. The `forward` function computes the output of the operator given its inputs, while the `backward` function computes the gradients of the inputs with respect to the output gradient.

By overloading standard Julia operators and functions, we can seamlessly integrate these operations into the computational graph. This overloading allows `GraphNode` objects to be used with standard operators, such as `+`, `*`, and custom functions like `convolution` or `relu`.

C. CNN model and training

We begin by loading and preprocessing the MNIST dataset. The training and test sets are reshaped to fit the input requirements of our network, and the labels are one-hot encoded for compatibility with our classification algorithm.

Subsequently, we define key hyperparameters for the network, including the number of epochs, learning rate, and batch size. These parameters are critical for controlling the training process and ensuring efficient convergence of the model.

Following the parameter setup, we define the architecture of the convolutional neural network (CNN). This involves initializing weights for each of the layers and constructing the computational graph. The network architecture includes convolutional layers, activation functions, max-pooling layers, and fully connected layers, designed to effectively process and classify the MNIST images.

Once the network architecture is established, we proceed to the training and testing phases. The training process involves forward and backward passes through the computational graph, updating weights based on the computed gradients. Throughout this process, we monitor key metrics such as training and test accuracy, as well as loss values, to evaluate the performance of the model.

Additionally, the implementation includes functionality to measure and report the time and memory usage, providing insights into the efficiency and resource requirements of our approach.

IV. OPTIMIZATION

In our implementation we used different types of optimization methods. The goal of them was to shorten the time of training and to minimize memory allocation. The following are methods we either implemented or tried to implement without success:

- **Static typing** By defying the type of the variable in code we make life for the compiler much easier. It can make more assumptions about the code such as the size of variables and operations applicable to them. This leads to better memory management, reduced fragmentation and overall improved memory access patterns. Example of static typing in our code can be noticed when analyzing the mutable, generic Variable structure where we use `@with-kw` macro to parameterize based on given type.
- **Region-based memory allocation** In our solution we took advantage of the `@views` macro to create a view over an array without making a copy of the data. This reduces the need for unnecessary memory allocations and copying. Example use of this optimization tactic can be seen in the implementation of the forward function for the maxpool layer. There, `@views` creates x-view as a sub array of x for each pooling operation, which avoids the creation of a new array and only provides a window into the original array x.
- **Mini-batching** We made several attempts to implement mini batching into our solution, unfortunately without any success. Mini-batching splits the entire dataset into small "mini-batches" and processes each mini-batch individually during the training process. Using mini-batching we technically could take advantage of the parallel processing capabilities of modern hardware, thus could make our solution much faster.
- **Einstein summation notation** Using the `@tullio` macro we were able to drastically reduce the time required to train our network. Tullio is a package that provides a macro-based approach for writing concise and efficient tensor operations. It allows multi-dimensional array computations using Einstein summation notation. The package takes care of optimizing the underlying computations, making use of multi-threading and other performance enhancements without requiring manual intervention. [4] Implementation of this approach can be seen in the forward and backward functions of the convolutional layer.

V. RESULTS

After implementing many optimizations into our solution and running it based on the architecture described earlier in the paper we get the following results:

- **Epoch 1:**
  - Train accuracy: 0.93
  - Test accuracy: 0.96
  - Memory usage: 152.6 GiB
  - Time: 50 s
- **Epoch 2:**
  - Train accuracy: 0.974
  - Test accuracy: 0.9713
  - Memory usage: 149.1 GiB
  - Time: 38.8 s
- **Epoch 3:**
  - Train accuracy: 0.981
  - Test accuracy: 0.9784
  - Memory usage: 149.93 GiB
  - Time: 37.7 s
- **Total time:** 126.5 seconds
- **Total memory used:** 452.539 GiB
- **Number of allocations:** 1.44 G

A. Reference solution

The subsequent step involved comparing our solution with a reference implementation in PyTorch, a widely-used machine learning library for Python.

The performance metrics of the PyTorch implementation are as follows:

- **Epoch 1:**
  - Memory usage: 77.8 MiB
  - Test accuracy: 91%
- **Epoch 2:**
  - Memory usage: 0.26 MiB
  - Test accuracy: 94%

- **Epoch 3:**
  - Memory usage: 0.56 MiB
  - Test accuracy: 95%
- **Total time:** 21.49 seconds
- **Total memory used:** 78.62 MiB

These results demonstrate that the PyTorch solution is highly efficient, achieving excellent accuracy while maintaining minimal memory usage, with a peak allocation of approximately 79 MiB.

The results we achieved by implementing the exact same neural network we intent to implement using Julia will work as a baseline we will aim to achieve.

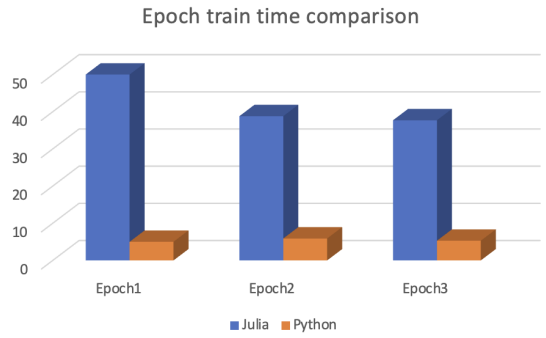

Fig. 2. Epoch train time comparison

B. Comparison

It is trivial to notice a huge performance difference between the two solutions. PyTorch obliterates our implementation with its speed and efficiency. However our implementation achieves over 2.5% higher test accuracy. Moreover, for the comparison to be fair, it is important to point out that our solution was implemented in a matter of weeks, with a small man power in a language we never heard about before. Nonetheless, beating PyTorch is possible. After deep analysis we came to a conclusion that the main cause of the huge performance delta is the lack of mini-batching in our implementation. [1]

VI. SUMMARY

We invested significant effort and dedication, both physically and mentally, into developing an optimized implementation of an automatic differentiation solution tailored for a custom neural network architecture. Furthermore, we conducted a comparative analysis of our work against a custom neural network developed using a well-known third-party library. Initially, the results appeared disappointing as the third-party library outperformed our implementation in terms of both memory usage and efficiency. However, considering the complete context of our project, we are pleased with our achievements. Moreover, we succeeded in optimizing our code, enhancing its performance by 50% and reducing memory consumption by 50% compared to our original version. We can confirm our neural network did not achieve AGI.

[1]The results were obtained on an M1 MAX MacBook with 64 GB of RAM.

REFERENCES

[1] J. Bezanson, A. Edelman, S. Karpinski, and V. B. Shah, "Julia: A fresh approach to numerical computing," *arXiv:1411.1607*, July 7, 2015. [Online]. Available: https://arxiv.org/abs/1411.1607

[2] 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*, vol. 18, no. 153, pp. 1-43, 2017.

[3] C. C. Margossian, "A Review of Automatic Differentiation and its Efficient Implementation," *arXiv preprint arXiv:1811.05031*, 2019.

[4] https://github.com/mcabbott/Tullio.jl