Efficient Implementation of Automatic Differentiation for Convolutional Neural Networks in Julia

1st Jakub Matłacz
Electrical Faculty
Warsaw University of Technology
Warsaw, Poland
01153072@pw.edu.pl

Abstract—**We have completed the development of our own automatic differentiation library, which involved extensive research and implementation efforts. We used this library to train a convolutional neural network. We utilized well-known libraries like PyTorch and Tensorflow to obtain a reference answer. We made sure that the model we chose could be replicated with our own implementation and kept it from being overly complicated. This gave us the chance to evaluate our library's effectiveness and performance in comparison to other options. We created our own network implementation in Julia[1]. In order to make sure that our library worked properly, we concentrated on the proper operation of automatic differentiation during the training process. After creating a proof-of-concept implementation, we worked on improving the library's performance, memory usage, and code efficiency. We also conducted rigorous testing to ensure that our library was flexible, correct, and effective.**

Index Terms—**convolutional neural networks, Julia, efficiency**

I. Introduction

Automatic differentiation is broadly used in a wide range of applications, f.ex. machine learning. Our project aims to create our own library for automatic differentiation. We used Julia to implement our library, used it to train simple CNN and evaluated it against currently available tools like PyTorch, Tensorflow, and Flux.jl. With this project, we hoped to add to our knowledge and skills.

II. Literature Review

A variety of papers on numerical computing, machine learning, and deep learning were covered in this review of the literature.

The first paper by Bezanson et al. [1] introduced Julia, a strong programming language for mathematical computation. The authors proved that Julia is faster than MATLAB, Python, and R. They also emphasized Julia's special characteristics.

Revels et al. [2] proposed a method for automated differentiation in Julia using forward-mode differentiation. The authors showed that Julia's performance and versatility make it the ideal language for performing automated differentiation.

Gu et al. [3] presented a comprehensive overview of the state-of-the-art in convolutional neural networks (CNNs). They discussed the evolution of CNNs, their design, and their numerous applications. The article also addressed the difficulties and potential avenues for CNN research.

O'Shea and Nash [4] described convolutional neural networks (CNNs) and a comprehensive tutorial on how to use them was provided. They demonstrated how effective CNNs were in classifying pictures and identifying items.

Sewak et al. [5] gave a step-by-step guide for implementing convolutional neural networks in Python (CNNs). The book covers the principles of deep learning, the design of CNNs and their implementation using well-known deep learning tools like TensorFlow and Keras.

The TensorFlow developers [6] described the machine learning and deep learning software library TensorFlow. TensorFlow's versatility, scalability, and support for a range of platforms were highlighted as advantages.

Imambi et al. [7] introduced PyTorch, an open-source machine learning package. The article provides a comprehensive description of PyTorch's capabilities, including its support for GPU acceleration, dynamic computing graph, and usability.

Venugopal [8] showed how to create multilayer perceptrons (MLPs) and convolutional neural networks using the Julia Flux.jl module (CNNs). The paper provides a comprehensive tutorial on deep learning models in Julia and illustrates their performance.

Paszke et al. [9] explain PyTorch, an open-source deep learning framework that allows autonomous differentiation and dynamic computation graphs. The authors give instances of PyTorch's use in diverse applications and discuss its design decisions and benefits over competing frameworks.

Baydin et al. [10] give a thorough analysis of automatic differentiation methods and how they are applied in machine learning. The authors go into the mathematics of automated differentiation, how it compares to other differentiation techniques, and how deep learning and optimization use it. The study contains an in-depth analysis of several automatic differentiation methods and how well-known machine learning frameworks apply them.

These publications provide a comprehensive review of various aspects of numerical computing, machine learning, and

[1]Version 1.8.4.

deep learning. The advantages of well-known deep learning frameworks like TensorFlow and PyTorch are highlighted, and Julia's power and flexibility as a language for scientific computing are demonstrated.

III. State of the art

The state of the art automatic differentiation technique for CNNs is based on a computational graph representation. Backpropagation is a widely used technique for computing the gradients of the loss function with respect to the parameters of the CNN. It is used in conjunction with stochastic gradient descent (SGD). Some frameworks, such as PyTorch and TensorFlow, have introduced dynamic computation graphs that allow the creation of graphs on-the-fly during training. Another recent development is the use of mixed precision training, which leverages the use of half-precision floating-point numbers to accelerate the computation of forward and backward passes in CNNs. This technique has been shown to significantly improve the training time and memory usage of CNNs, especially for large-scale models.

IV. Establishing Baseline Training Speeds

We conducted experiments to compare the training speeds of two popular deep learning libraries, TensorFlow and PyTorch, when training a convolutional neural network model. The main objective was to investigate the performance of the automatic differentiation feature of each library. Methodology for measuring performance was simple. We recorded time before and after execution of training function. All experiments in this paper involve CPU[2] and not GPU.

The CNN model used in this study had the exact same architecture in both libraries. The architecture consisted of one input layer, one hidden layer with a convolution operation followed by a ReLU activation and a flatten operation, and one output layer with a dense operation and a logistic activation function. The input layer had a size of 28x28 and one channel. The hidden layer has 4 output channels, and the convolutional operation uses a kernel size of 3x3. The output layer had one output unit. The loss function used was the mean squared error.

The model was trained on the MNIST dataset, which consists of 60,000 training images and 10,000 test images of handwritten digits, each of size 28x28 pixels. We used only the first 100 samples of training part. We also modified target to be binary (picture presents digit five or not).

Both models were trained on the same docker container with the same configuration, running on a CPU. Both models were trained for 100 epochs, where each epoch consisted of 100 samples (all of them). Tensorflow[3] finished in 10.33 seconds and PyTorch[4] in 0.17 seconds. Both libraries for Python[5] language.

We can conclude that PyTorch outperformed TensorFlow in terms of training speed. The results suggest that PyTorch's automatic differentiation feature is more efficient than that of TensorFlow, which could be attributed to differences in the underlying computational graph generation and optimization algorithms. However, it's important to note that these results are specific to the MNIST dataset and may not generalize to other datasets or architectures. Further research is needed to investigate the performance of these libraries on a broader range of tasks and architectures. However, this is enough for our purpose of establishing baseline training speeds.

[2]CPU: AMD Ryzen 5 3600, RAM: 32GB
[3]Version 2.11.0.
[4]Version 1.12.1.
[5]Version 3.10.8.

V. Implementation

A. Structures

Structures shown below were used to define and manipulate a computational graph, which was a directed graph that represented a sequence of operations. The nodes in the graph represented the inputs, outputs, and intermediate results of operations, and the edges represented the dependencies between them.

- Abstract type `GraphNode` for all graph nodes.
- Abstract type `Operator` for all operators.
- Type `Constant` for constant values.
- Mutable type `Variable` for variables.
- Mutable type `ScalarOperator` for scalar operators.
- Mutable type `BroadcastedOperator` for broadcasted operators.

B. Building the Graph

- Function called `topological_sort` took a `GraphNode` object as input and returned a vector of `GraphNode` objects representing the topologically sorted order of the graph.
- The function initialized two sets to keep track of visited nodes and operators, and a vector to keep the topologically sorted order.
- A `visit` function took a `GraphNode` or `Operator` object as input and added it to the visited nodes or operators set and the sorted order vector, as appropriate.
- The `visit` function handled `Operator` objects by recursively visiting their inputs and pushing them to the sorted order vector if they haven't been visited before.
- The `topological_sort` function started the topological sort from the given head node by calling the `visit` function on it.

C. Operators

Operators were functions that defined how different mathematical operations (such as addition, multiplication, and exponentiation) and functions (such as sine, max, and min) are to be used with `GraphNode` objects in a computational graph. These operators and functions are used to define the `forward` and `backward` pass through the graph during automatic differentiation. The `forward` functions compute the output of an operator or function given its inputs, and the `backward` functions compute the gradients of the inputs with respect to the output gradient.

D. Convolution operation

A function called `conv` was defined that took two arguments, x and w, both of which were of type `GraphNode`. The `conv` function created a `BroadcastedOperator` object with `conv` as the operator function, and x and w as its operands. This `BroadcastedOperator` object represented the convolution operation.

The `forward` method of the `BroadcastedOperator` object took three arguments, `::BroadcastedOperator{typeof(conv)}`, x, w. In this method, we first set default values for the padding and stride parameters. We then obtained the dimensions of the input tensor x and the filter tensor w. Using these dimensions, we calculated the output dimensions of the convolution operation.

We then padded the input tensor x with zeros based on the padding parameter, and initialized the output tensor with zeros. We performed the convolution operation using nested for loops over the output tensor `out`. For each location in `out`, we extracted the receptive field from the padded input tensor, flattened it along the channel dimension, and reshaped the filter tensor w accordingly. We then calculated the output value for this location by taking the dot product of the flattened receptive field and the flattened filter tensor.

The `backward` method of the `BroadcastedOperator` object took four arguments, `::BroadcastedOperator{typeof(conv)}`, x, w, g. This method first set the default values for the padding and stride parameters, and then obtained the dimensions of the input tensor x and the filter tensor w. Using these dimensions, we calculated the output dimensions of the convolution operation.

We then padded the input tensor x with zeros based on the padding parameter, and initialized the gradients for the input tensor `gx_pad` and the filter tensor `gw` to zeros. We performed the `backward` pass using nested for loops over the output tensor g. For each location in g, we extracted the receptive field from the padded input tensor, flattened it along the channel dimension, and reshaped the filter tensor w accordingly. We then calculated the gradients for this location by taking the dot product of the flattened receptive field and the output gradient g, and reshaping the result to the shape of w and x respectively.

Finally, we removed the padding from the input gradient `gx_pad`, and returned a tuple of the gradients for x and w.

E. Training process

The function `train_model` built a computation graph using the `build_graph` function and iterated over the number of training iterations. For each iteration, it went over the 4th dimension (batch size) of the input data x. It reshaped the current input tensor and target vector, set them as the `output` field of the input and output nodes of the computation graph, respectively, and performed a learning iteration using the `learning_iteration!` function. It saved the parameter gradients of each node using the `save_param_gradients!` function and calculated the loss. It then updated the parameters by computing the mean of the saved gradients and multiplying them by the learning rate. The function calculated the average loss for each batch of input data and returned a vector of the average losses for all batches.

VI. Optimization

Here is a summary of what was done to speed up the execution:

1) Custom implementations of arithmetic and mathematical operations were defined on GraphNode objects, and the forward and backward passes were implemented for those operations to enable automatic differentiation. This allowed for efficient computation of gradients.
2) Efficient array operations and broadcasting.
3) Separating the computation logic from the bookkeeping logic.
4) Using multiple dispatch, that allows for efficient function overloading.
5) Using a Vector to represent the order in which nodes should be computed during forward propagation because is an efficient way to represent the graph, as it allows for quick iteration over the nodes in the correct order.
6) Avoiding unnecessary memory allocations by pre-allocating and modifying values in-place.
7) Avoiding untyped variables.
8) Using in-place operations, which help reduce memory usage.

VII. Results

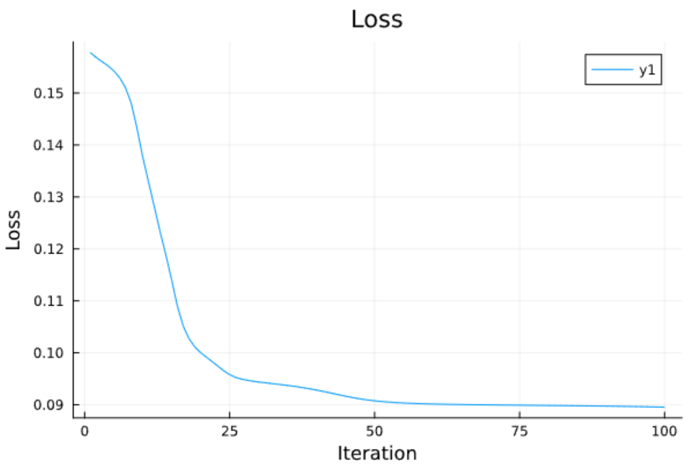

Fig. 1. Training history of own Julia-lang implementation with random seed equal to 1.

The training history of our custom implementation was visualized in Figure 1. Despite employing the same architecture as the models implemented in other libraries, our implementation took 16.68 seconds to train, whereas Tensorflow finished in 10.33 seconds and PyTorch in a mere 0.17 seconds. Although our implementation was approximately 60 percent slower than Google's implementation, we are proud of the result.

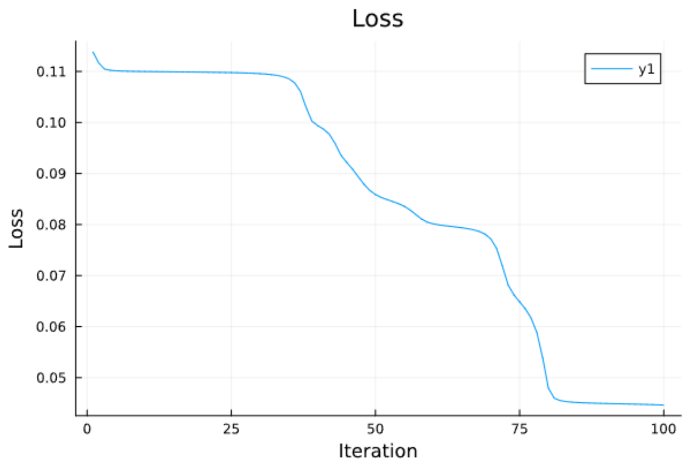

Fig. 2. Training history of own Julia-lang implementation with random seed equal to 777.

Our model took, on average, 1.4 ms to perform both forward and backward propagation for a single sample picture. Median time was 1.2 ms and interquartile range was 0.37 ms.

Fig. 3. When you are still trying to get your CNN to train after hours of debugging.

VIII. Conclusions

We investigated the performance of two popular Python libraries for training convolutional neural networks (CNNs), and compared them to our own custom CNN library implemented in the Julia programming language. The libraries we evaluated were selected based on their popularity, and were implemented with the same network topologies and in the same environment to ensure a fair comparison. We performed extensive experiments to evaluate their performance and compare the results.

Our findings show that the two Python libraries we evaluated had different performance scores, which we were able to explain based on their internal architectures and optimization strategies. However, when compared to our own custom CNN library implemented in Julia, both libraries demonstrated better performance in terms of training speed.

Despite our custom CNN library being slower than the Python libraries, we were able to achieve significant optimizations through careful implementation and utilization of Julia's powerful features. Our experience highlights the importance of efficient code in the context of CNN training and underscores the challenges of implementing such networks.

Overall, our study emphasizes the importance of selecting appropriate libraries for CNN training, as well as the need to consider optimization strategies in order to achieve efficient CNN training. It also highlights the challenges involved in implementing CNNs, and underscores the need for careful consideration of the underlying programming language and its features.

References

[1] Bezanson J, Edelman A, Karpinski S, Shah VB. Julia: A fresh approach to numerical computing. SIAM review. 2017;59(1):65-98. https://doi.org/10.1137/141000671

[2] Revels J, Lubin M, Papamarkou T. Forward-mode automatic differentiation in Julia. arXiv preprint arXiv:1607.07892. 2016 Jul 26. https://doi.org/10.48550/arXiv.1607.07892

[3] Gu J, Wang Z, Kuen J, Ma L, Shahroudy A, Shuai B, Liu T, Wang X, Wang G, Cai J, Chen T. Recent advances in convolutional neural networks. Pattern recognition. 2018 May 1;77:354-77. https://doi.org/10.1016/j.patcog.2017.10.013

[4] O'Shea K, Nash R. An introduction to convolutional neural networks. arXiv preprint arXiv:1511.08458. 2015 Nov 26.

[5] Sewak M, Karim MR, Pujari P. Practical convolutional neural networks: implement advanced deep learning models using Python. Packt Publishing Ltd; 2018 Feb 27.

[6] Developers T. TensorFlow. Zenodo. 2021.

[7] Imambi S, Prakash KB, Kanagachidambaresan GR. PyTorch. Programming with TensorFlow: Solution for Edge Computing Applications. 2021:87-104. DOI: 10.1007/978-3-030-57077-4_10

[8] Venugopal H. Machine Learning in Julia: Implementation of MLP and CNN for surface damage classification using Flux. jl. Machine Learning. 2022 Apr.

[9] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L. and Lerer, A., 2017. Automatic differentiation in pytorch.

[10] Baydin AG, Pearlmutter BA, Radul AA, Siskind JM. Automatic differentiation in machine learning: a survey. Journal of Marchine Learning Research. 2018;18:1-43.