Creation of the library used in training of convolutional neural networks

inż. Piotr Szczepański
Wydział Elektryczny
Politechnika Warszawska
Warszawa, Polska
01148811@pw.edu.pl

inż. Karol Zieliński
Wydział Elektryczny
Politechnika Warszawska
Warszawa, Polska
karol.zielinski.stud@pw.edu.pl

***Index Terms*—AI, Artificial Inteligence, CNN, Convolutional Neural Network, Julia, Julia Programming Language, Automatic Differentiation**

I. Introduction

The action taken by the world's biggest technology companies such as Microsoft and Google clearly indicate the direction of development of the modern world. Since the launch of ChatGPT artificial intelligence has gained popularity in society and reinvigorated speculations about its capabilities. Most recently developed deep learning techniques have obtained well performance into many fields of application, for example visual data processing or audio processing (including processing of the natural language). Another, one of the most popular, approach to the deep learning is convolutional neural network [1]. Convolutional neural networks, created on the basis of biological neural networks, are being applied in a range of different fields, including computer vision, pattern recognition, speech processing and face recognition. Such networks are feedforward networks in which information flow takes place in only one direction (from input to output) [2]. It also uses the idea of weight sharing and because of that the number of parameters that need to be trained is reduced - that helps in easier training. A general model of convloutional neural network consists [3]:

- convolution layer
- pooling layer
- activation function
- fully connected layer

With the widespread popularisation of artificial intelligence, many solutions have emerged that can even be created at home. One of these is the ability to create your own artificial neural network. Because of that reliable computation of derivatives has become crucial, because machine learning and other based on gradient optimization problems mostly rely on it. A programming language that can be used for this purpose is Julia which can be described as high level, high performance and dynamic. It is open source, has been created in 2012 and designed for numerical computing (what is great advantage when it comes to serious calculations). Julia also integrates open source C and Fortran libraries for wide range of functions [4]. One of the packages that extends the usage of Julia programming language toward artificial intelligence is ForwardDiff - package for forward-mode automatic differentiation which uses JIT (just-in-time) compilation (unlike some similar tools developed in other high-level languages). It implements to Julia multidimensional dual number[5] .Automatic differentiation is universally used for deep learning because of its great performance of gradient-based optimization procedures compared to the other methods which does not use derivatives. It works very well when it comes to the continuous dependence of parameters, yet programs that have a discrete connection between a result and the parameters are challenging. [8] Automatic differentiation is a powerful tool when it comes to the calculation of derivatives, what is often required while using numerical methods.[6] It is, among other methods of automatic calculation of derivatives (like finite differentiation and symbolic differentiation), the best performing - when applied to complex functions [6]. Current availability of general purpose automatic differentiation really simplifies the implementation of algorithms such as deep neural networks by making them regular program that use differentiations and is created with convolutional and recurrent elements [7]. The availability of methods and ways to implement neural networks is encouraging the search for new solutions and extending the capabilities of existing ones. However, how do custom solutions compare to existing ones created using popular libraries?
In this article, a comparative analysis of the common implementation of automatic differentiation contained in the Keras library will be made with the custom one realised in the Julia programming language. The basis for comparison will be the implementation of a convolutional neural network.

II. CNN Implementations

Since the aim of this article is to compare the performance of two implementations of the automatic differentiation contained in a convolutional neural network, two solutions that work the same way had to be prepared. More specifically, they were to consist of the same components (individual layers, as well as their activation function, loss function and optimiser). However, before the construction of the reference neural network is presented, it is worth mentioning that not only pure code is important when comparing implementations. Equally important is the software and hardware, meaning all the computers components that are used to run the programmes.

A. Hardware Components

Starting with the hardware, a desctop computer was used to run the neural networks. This computer is equipped with an Intel Core i5-13600KF CPU, which is a fourteen-core, twenty-thread processor. It has a base frequency of 3.50 GHz and a turbo boost frequency of 5,10 GHz. In terms of graphics, it features an NVIDIA GeForce GTX 3060 with 12GB of onboard memory and a core clock rate of 1807 MHz. This computer also comes with 32GB of DDR4 RAM clocked at 4000 MHz and its storage is a 1TB M.2 PCIe NVMe 4.0 x4 SSD.

B. Software Components

The computer runs on Windows 11, the latest operating system developed by Microsoft. For running neural networks, Jupyter Notebook was utilized as the chosen environment, integrated within the Visual Studio Code. Visual Studio Code is an rich text editor created by Microsoft which offers a wide range of features and extensions that enhance the coding experience, including intelligent code completion or debugging. It supports multiple programming languages. Version 1.78.0 of the Visual Studio Code was used while working on the neural networks.

C. Reference Solution

The reference convolutional neural network was built in Python programming language (version 3.10.8). It is designed to classify images from the Fashion MNIST dataset which is dataset of Zalando's article images (which is online retailer of articles of clothing). This dataset consists a training set of 60000 examples and a test set of 10000 examples. Each image is a 28x28px gray-scale image associated with a label from 10 classes: t-shirt, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag and ankle boot. The Fashion MNIST dataset is an alternative to the original MNIST dataset, which contains exact same structure and image size but the contents of the images are different - they are handwritten digits. Both training and test data sets have 785 columns - first one consists the class number and the other 784 contain values representing greyscale of the image (values from 0 to 255). The Fashion MNIST is released under the MIT license which allows free use, modification, distribution, and commercial use of the dataset. The neural network has been created using the Keras which is a deep learning API written in Python running on top of Tensorflow 2.0 - developed by Google open-source library for numerical computation and large-scale machine learning. TensorFlow enables developers to construct dataflow graphs, which serve as representations of the flow of data through a network of processing nodes. In this graph, each node corresponds to a specific mathematical operation, while the connections or edges between nodes are multidimensional data arrays. Keras works as its interface and allows users to define and train neural networks, by providing a wide range of pre-built layers, activation functions, loss functions, optimizers and other utilities that make creating custom neural network easier. The version of Keras used while implementing reference solution was 2.9.0., and Tensorflow version was 2.9.1. While creating reference solution, sequential model from Keras library has been used as foundation. This is one of the simplest type of models, as it performs linear stack of layers - each layer is sequentially connected to the next layer. Subsequently following layers were then identified:

- Conv2D - convolutional layer using ReLU activation which extracts features from input images,
- Flatten - layer that converts multi-dimensional output from Conv2D layer into a one-dimensional vector,
- Dense - fully connected layer using ELU activation which provides non-linear transformation of the output,
- Dense - final, fully connected layer that uses softmax activation and provides probability distribution over the classes

Created model of convolutional neural network is then compiled using caterogical cross-entropy loss function and SGD optimizer. In this case caterogical cross-entropy measures the dissimilarity between the predicted probability distribution and the true probability distribution. The loss value in this function is calculated for each class, and the overall loss is the average of these particular losses. Through the optimisation of this loss function, the model aims to produce more accurate and reliable multi-class forecasts. Regarding the SGD (Stochastic Gradient Descent) optimizer, it is well-known and widely used algorithm which updates the models parameters iteratively based on the gradients of the loss function with respect to those parameters. SGD is used to update the weights and biases of the model during the training process. The models parameters are adjusted to minimize the categorical cross-entropy loss function between the predicted probabilities and the true labels. What is worth mentioning is that SGD is not perfect optimizer as it might get stuck in local minimum extremes. The last activity performed with the reference convolutional neural network model was its training. The batch of the train data was used in this purpose for going through multiple epochs (iterations). For each batch (which size can be changed), the model generates predictions, computes the loss based on them, and then updates the models parameters (weights and biases) using the SGD optimizer. The model 'learns' from the training data, and after the training comes to the end cnn can be used to make predictions on test data.
This is how a reference model of a convolutional neural network created in Python language with use of external library has been implemented. To its performance an implementation of our own network written in the Julia programming language will be compared.

D. Implementing a custom neural network in Julia

The development of the custom solution, which was to create a convolutional neural network using automatic differentiation, was divided into 2 stages. The first was the preparation of a proof-of-concept - a preliminary implementation of the network. The second stage was its refinement - preparation of

the final implementation.
As mentioned above, implementation of custom neural network was created in Julia programming language, specifically using version 1.8.5. To maintain consistency Jupyter Notebook via Visual Studio Code has been used as well. As the aim is to compare the use of automatic differentiation, the neural networks, although created in different programming languages, had to be the same. Otherwise, the differences involved would have reduced the reliability of the comparison results. The approach, then, in developing own solution was to replicate the reference network as accurately as possible. A library, based on computational graphs (computed forward and backward), was then developed and used to create a network model like the one described above in the reference example. To be specific, this involves the individual layers in correct order, their activation functions, the caterogical cross-entropy loss function and the SGD optimizer. Obviously also the same dataset was used in both cases. To achieve similarity, some not significantly affecting the functioning external packages were needed. Below is a table with them and their versions included : Package MLDatasets was used to import Fashion MNIST dataset, and Flux's functionality onehotbatch was used to adapt data representation.

Package	**Version**
MLDatasets	0.7.9
Flux	0.13.15

TABLE I: Packages used in Julia and their versions

Once the CNN was set up in Julia and assured of its proper functioning, it was possible to move on to the next stage, which was to improve its quality and efficiency. Macros available in the Julia programming language or in its BenchmarkTools package were used to achieve this. Accordingly, several changes had to be made:

- Minimizing the occurrence of allocations in the program (which, as seen in the studies, was insufficient)
- Utilization of macros @. and @inbounds
- Utilization of views where necessary
- General quality of code improvements

After the adjustment of Julia implementation, the final version became ready for comparison with reference solution created in Python.

III. Performance Experiments

Implementation comparisons will mainly be considered between the reference example and the final version of the network in Julia. The example from before the refinement will be included more as a curiosity about the optimisation, as it is not an optimal solution.

In order to compare the custom convolutional neural network implementation with the one using an external Keras library, several tests were carried out:

A. Comparison of network learning rates as a function of training set size

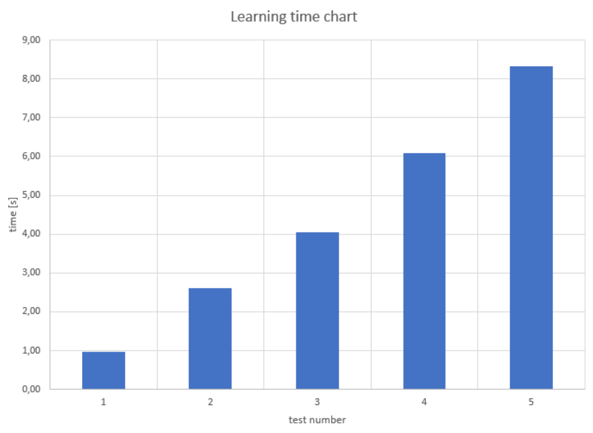

Fig. 1: A chart of learning times in the Python program

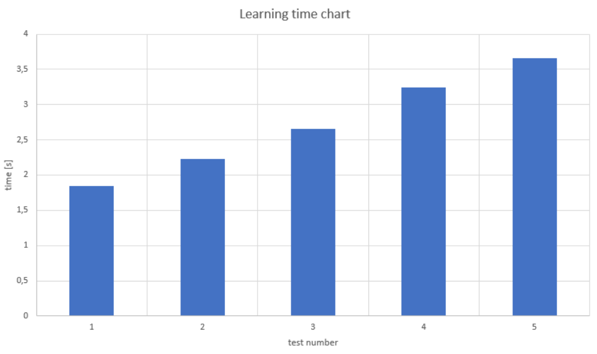

Fig. 2: A chart of learning times in the Julia program

The results of the study revealed interesting findings regarding the training times of the CNNs using Python and Julia. For smaller dataset sizes, the Python implementation exhibited faster training times compared to the Julia implementation. This outcome can be attributed to the mature ecosystem and optimized libraries available in Python for deep learning tasks.

However, as the dataset size increased, the Julia implementation demonstrated superior performance in terms of training times. It achieved faster training times compared to the Python implementation. This observation suggests that Julia's high-performance computing capabilities become advantageous when dealing with larger and more complex datasets. The observed difference in training times may also stem from the simplicity of the solution in the Julia language.

B. Comparison of the loss function

In the second study, we investigated the learning performance of the trained neural network models and their ability to minimize the loss function. The loss function used in this study was the cross-entropy loss, a commonly employed metric for classification tasks.

The neural network models implemented in both Python, serving as the reference solution, and Julia were trained using the respective programming languages. It was observed that the Julia implementation, due to its simplicity, exhibited slower convergence in minimizing the cross-entropy loss function.

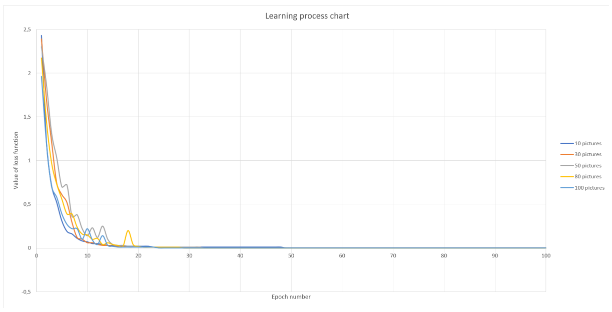

Fig. 3: A chart of learning progress in the Python program

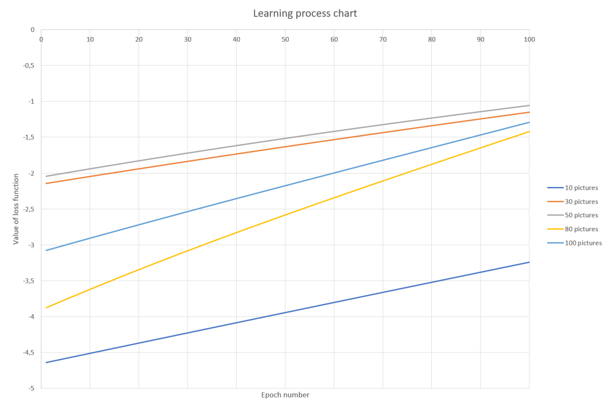

Fig. 4: A chart of learning progress in the Julia program

The training process required a significantly larger number of epochs for the Julia program to achieve comparable results.

In contrast, the Python implementation demonstrated excellent performance and achieved satisfactory results after approximately 20 epochs. The reference solution, benefiting from the mature ecosystem and optimized libraries available in Python, showcased efficient convergence in minimizing the loss function.

These findings highlight the impact of programming language choice on the efficiency of the training process and the speed of convergence. While the Julia program may have certain advantages in terms of simplicity, it may require additional optimization techniques or adaptations to improve its learning capabilities and reduce training time.

C. *Comparison of allocations*

The purpose of the final study was to investigate the memory allocation behavior of the Python and Julia programs during their execution. Specifically, the study aimed to determine the amount of memory allocated by the programs and how it was influenced by the size of the training dataset. The results revealed notable differences in memory allocation between the two implementations. The Python program consistently allocated approximately 5 MB of memory throughout its execution, regardless of the dataset size used for training. This finding indicated that the memory allocation of the Python program remained relatively constant and unaffected by changes in the dataset size. In contrast, the Julia program

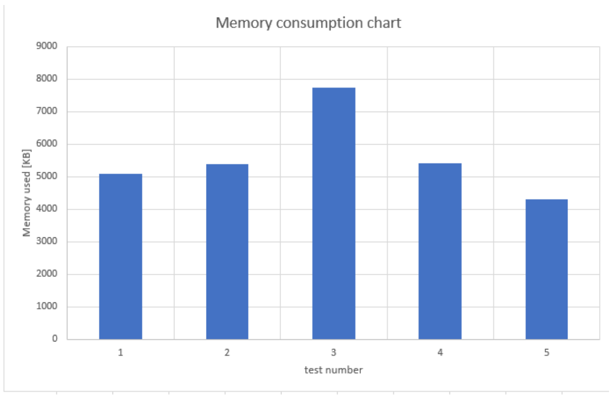

Fig. 5: A chart of memory usage in the Python program

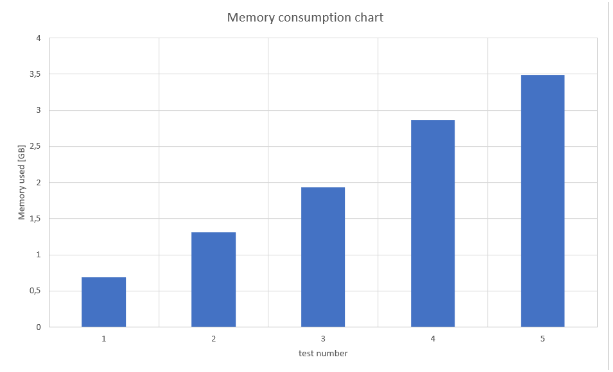

Fig. 6: A chart of memory usage in the Julia program

exhibited distinct memory allocation patterns. Not only did the amount of memory allocated by the Julia program depend on the dataset size, but it also showcased significantly larger memory allocations. On average, the Julia program allocated around 2GB of memory during its execution.

IV. Conclusion

In this article, we examined the quality and performance of a custom implementation of a library for automatic differentiation in Julia, in comparison to a reference solution implemented in Python. Our focus was on evaluating the key aspects of training convolutional neural networks (CNNs), including training time, learning performance, and memory allocation.

In the analysis of results, it is necessary to take into account the difference in automatic differentiation methods between both programs.

When comparing the automatic differentiation methods employed in Python and Julia programs, we observe distinct approaches: symbolic differentiation in Keras and graph nodes in Julia. Symbolic differentiation, as implemented in Keras, offers advantages in terms of computational efficiency and optimization. The use of symbolic expressions allows for precompiled computations, resulting in improved performance and utilization of hardware resources. Additionally, symbolic differentiation enables the application of advanced compilation optimizations, further enhancing the overall computational efficiency. On the other hand, Julia's graph nodes approach offers flexibility and expressiveness. The graph nodes provide a way to define and manipulate computational graphs, allowing for greater control over the differentiation process. This flexibility can be advantageous when working with complex models and custom operations.

REFERENCES

[1] L. Alzubaidi et al., 'Review of deep learning: concepts, CNN architectures, challenges, applications, future directions', J Big Data, vol. 8, no. 1, p. 53, Mar. 2021, doi: 10.1186/s40537-021-00444-8.

[2] Waseem Rawat, Zenghui Wang, "Deep Convolutional Neural Networks for Image Classification: A Comprehensive Review", 2017.

[3] S. Indolia, A. K. Goswami, S. P. Mishra, and P. Asopa, 'Conceptual Understanding of Convolutional Neural Network- A Deep Learning Approach', Procedia Computer Science, vol. 132, pp. 679–688, 2018, doi: 10.1016/j.procs.2018.05.069.

[4] T. A. Cabutto, S. P. Heeney, S. V. Ault, G. Mao, and J. Wang, 'An Overview of the Julia Programming Language', in Proceedings of the 2018 International Conference on Computing and Big Data, Charleston SC USA: ACM, Sep. 2018, pp. 87–91. doi: 10.1145/3277104.3277119.

[5] J. Revels, M. Lubin, and T. Papamarkou, 'Forward-Mode Automatic Differentiation in Julia'. arXiv, Jul. 26, 2016. Accessed: Apr. 02, 2023. [Online]. Available: http://arxiv.org/abs/1607.07892

[6] C. C. Margossian, 'A Review of automatic differentiation and its efficient implementation', WIREs Data Mining Knowl Discov, vol. 9, no. 4, Jul. 2019, doi: 10.1002/WIDM.1305.

[7] Atılım G¨une¸s Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, Jeffrey Mark Siskind," Automatic Differentiation in Machine Learning: a Survey", April 2018.

[8] G. Arya, M. Schauer, F. Schäfer, and C. Rackauckas, 'Automatic Differentiation of Programs with Discrete Randomness'.