Methods for Speeding Up the Retraining of Neural Networks

Sergey S. Varykhanov[1], Dmitry M. Sinelnikov[2], Viktor V. Odintsev[3], Mikhail M. Rovnyagin[4], Elza F. Mingazhitdinova[5]
National Research Nuclear University MEPhI (Moscow Engineering Physics Institute)
Moscow, Russia
[1]masmx64@gmail.com, [2]dsinelnikov96@gmail.com, [3]zakhams@gmail.com, [4]m.rovnyagin.2015@ieee.org, [5]mingazhitdinova@gmail.com

***Abstract*—Nowadays machine learning is widespread and becoming more complex. Developing and debugging neural networks is becoming more and more time-consuming. Distributed solutions are often used to speed up learning process. But these solutions don't solve the problem of retraining model from zero if the learning fails. This paper presents a new approach for training models on a large datasets, which can save time and resources during the development. This approach is splitting model's learning process on separate layers. Each of these layers can be modified and reused for the next layers. The implementation of this approach is based on transfer learning and distributed machine learning techniques. To create reusable network layers, it is proposed to use the methods of automating code parallelization for hybrid computing systems described in the article. These methods include: tracking the readiness and dependencies in the data, speculative execution at the kernel level, creating a DSL**

Keywords—ML, neural networks, hybrid computing systems, code optimization

I. INTRODUCTION

Nowadays, there are a lot of machine learning frameworks. One of the most popular, convenient and at the same time reliable framework is TensorFlow [1].

TensorFlow was developed by the Google Brain team for internal Google use in research and production. In 2015 Google released the initial version under the Apache License 2.0 [2].

In September 2019, Google released the updated version TensorFlow 2.0. New version provides a lot of improvements comparing the old one and also provides Keras library [3] as a high-level API.

Today, training model on the single computer takes a lot of time for complex models. Distributed machine learning solves this problem by performing training on the cluster. One of the most interesting distributed machine learning frameworks is TensorFlowOnSpark [4] released by Yahoo in 2019. It provides API layer to perform model training on the Apache Spark [5] cluster using TensorFlow framework.

But despite the framework, the neural network developers are often facing the problem of building and training model from the beginning.

This problem becomes stronger when it comes to train large model on a huge datasets. Distributed machine learning partially avoids this problem. But still, in this case, any mistake in the model or dataset costs a lot of time and resources.

Modern computing systems have reached a certain limit of the computation speed per core. The decrease in those processes and the increase in frequency are limited by the laws of physics [6]. If we look at the architecture of a modern Intel 11th generation Tiger Lake (2020) processor and compare it with the 2nd generation Sandy Bridge (2011), we will see. That the improvements over the past almost 10 years are more quantitative than qualitative. The architecture of ZEN processors from AMD is somewhat different from Intel, but the basic principles are very similar there.

The performance improvements over the past 10 years have been achieved by reducing the process technology, which in turn reduces heat generation, and also allows more transistors to be placed on the chip. This allows more cores to be placed on the die. From this we can conclude that the current trend to increase the number of cores is likely not going anywhere.

To date, there is no fully automatic program parallelization algorithm. This is due to the complexity of deep analysis of data flows within the program. This is partly due to the fact that most of the libraries are stored in the form of machine code, not source programs. In addition, existing programming languages are usually designed for manual parallelization by creating threads [7]. Some languages, such as Go lang or Qt, take some of this responsibility. However, the huge variety of different algorithms makes the task of fully automatic parallelization extremely difficult, due to the need to build chains of instructions that can be executed in parallel. At the moment, a search is underway for various solutions to this problem.

In order to write high-performance programs on modern computing systems, it is necessary to parallelize the algorithm so that it makes the most of all available computing resources. Unfortunately, in the general case, the process of manually parallelizing an algorithm and implementing it in the form of a program code is an extremely nontrivial task that requires a creative approach and knowledge in the field of parallel programming.

II. RELATED WORKS

There are multiple ways of reducing model development time. The most common way is to reduce training time using

978-1-6654-0993-3/22/$31.00 ©2022 IEEE

distributed machine learning technique. Distributed architecture also allows to achieve better availability [8].

Distributed machine learning can be performed in two ways: data-parallelism and model-parallelism.

In the data-parallelism approach [9] the model is copied across all cluster nodes. Then the training data is splitted into batches and every node trains model on the specific batch. In the end, trained models are merged into one. This approach is the most common during its simplicity and flexibility.

The model-parallelism approach [10] has more complex implementation. In this approach model is splitted into multiple parts and every cluster node trains specific model part. But, unfortunately, this approach lacks of flexibility and supports very limited number of learning algorithms.

One of the interesting approaches to the distributed machine learning is Federated Learning [11]. In Federated Learning multiple clients collaborate to solve traditional distributed ML problems under the coordination of the central server without sharing their local private data with others. This approach provides a secure way to organize distributed machine learning.

Using weighted round robin task distribution can increase learning speed on heterogeneous clusters with GPGPU [12].

Another way of reducing model developing time is transfer learning technique [13]. This approach allows to change the domain of already trained model to another domain. Using this approach developers can reuse the existing model and re-purpose it to their needs.

III. Layers-based development approach

In this article, we propose a way to speed up model development and training. This approach is based on transfer learning and distributed machine learning techniques.

Transfer learning allows to re-purpose already trained models to do another specific task. Using this technique, we propose new definition – layer.

Layer is a trained model on a specific domain. During development model can have multiple layers. Each layer except the first one is based on the previous one. And the final layer is a final trained model. The layers model architecture is shown in the figure 1.

This approach allows developers to rollback model to the previous layer in case of mistake. In addition, any layer can be shared with other developers, that lets them not to start from the beginning, but reorganize and retrain already prepared model to their needs.

Beyond that, TensorFlowOnSpark has been chosen to achieve higher training speed using distributed learning and data-parallelism.

To achieve high-availability on the Apache Spark cluster, Kubernetes [14] has been chosen as a cluster manager.

IV. Experiments

Beyond that, TensorFlowOnSpark has been chosen to achieve higher training speed using distributed learning and data-parallelism.

To test the workflow of the implemented platform, a test case was developed.

To measure performance improvements of the proposed approach two experiments were performed. The first experiment was to measure the dependence of time from training accuracy for the classic development workflow. And the second one was to take the same measurements for every layer in layers-based development workflow.

The experiments were performed on the MNIST DB [15].

All measurements were taken on homogeneous Apache Spark cluster consisting of four nodes.

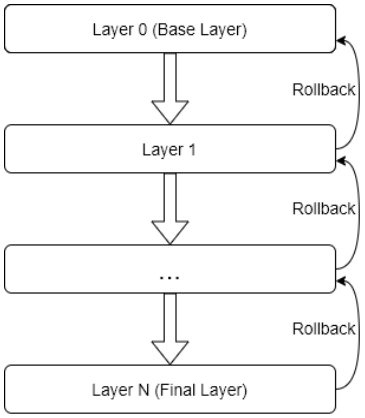

Fig. 1. The layers model architecture

The result of the experiments is shown in the figure 2.

As can be seen from the experiment results, the classic way of training is slower then each of the layers training. And every next layer is faster than the previous one. If developers need to retrain only the last layer it will take less time than retraining model from the beginning.

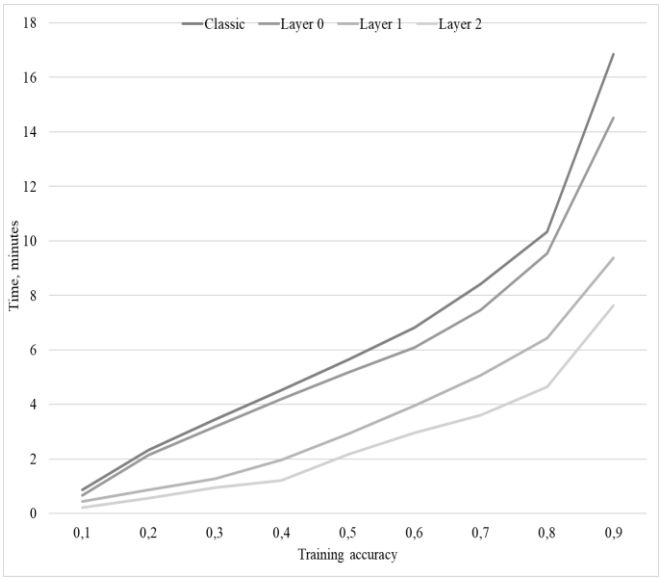

Fig. 2. The result of the experiments

V. Low-level Optimization Approach

As an experiment, it was decided to develop a programming language. It is a C-like language, the main feature of which is the presence of a number of restrictions associated primarily with the operation of pointers and the limited use of global variables. In addition, language-level synchronization mechanisms are expected to be introduced. The main limitations are as follows:

1) Pointers can be formed only from variable addresses or be NULL;
2) The meaning of the output values of the function depends only on the input values;
3) Prohibiting the use of global variables.

As a result of the introduction of these restrictions, it becomes possible to consider a program as a computational circuit consisting of a set of related, independent functions with multiple inputs and outputs. A fragment of such a scheme is shown in Figure 3.

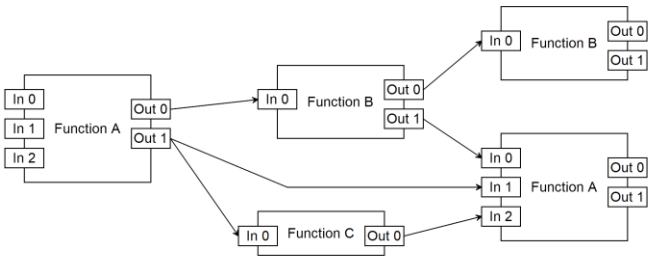

Fig 3. Computing stack fragment.

Such schemas can be very efficiently parallelized. Each of the functions is calculated when the input data is ready. A pool of several threads and a queue of functions ready for computation are created. Initially, this queue contains functions that have no inputs, or depend only on the input values of the program. Each of the threads performs the following algorithm:

1. Wait for at least one function ready for computation to appear in the queue;

2. Pick up a function from the queue for execution;

3. Execute the function;

4. Bypass all functions that require the results of the calculation of this function and mark the corresponding inputs that the data is ready;

5. If all the input data of the function are ready, then add it to the queue.

To implement this method, you need to use thread synchronization tools, for example, you need to use semaphores to organize access to the queue, but these details are omitted in this article. If a block of statements is under a condition in the original program, then the condition itself is executed as a separate function with one logical output and, depending on the result, the functions that were in the body of the condition in the program are ignored or executed.

The situation with loops is a little more complicated. In fact, the loop can have an unlimited number of iterations and, accordingly, an unlimited number of functions for each iteration. As a solution, dynamic function generation is used several iterations ahead.

Various approaches and specific algorithms need to be developed and tested before starting to build a basic software tool for automatic parallelization. For these purposes, a test prototype of the program was developed, the main purpose of which is to create controlled conditions for testing. In the future, this prototype will be called an emulator.

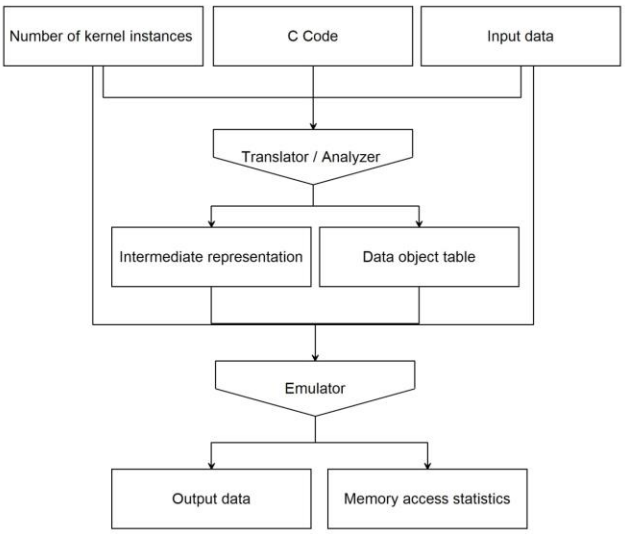

Fig 4. Emulator schema

The emulator translates the C source code into an intermediate representation in the form of a C operator tree. C++ features are not currently supported. This C code is an analogue of the CUDA kernel function, therefore, in what follows, the code for the emulator will also be called the kernel. Instead of a main () function, the kernel must contain a kernel () function with an arbitrary set of parameters. These parameters currently require additional descriptions outside the body of the program. The program is represented by the n-th number of threads without dividing into blocks and excluding warp. Another limitation concerns dynamic memory allocation within the kernel. In CUDA programs, dynamic memory allocation within the kernel is, in principle, possible starting from the compute capability 2.0, but it is not very much in demand. The reason is very simple. The main memory used is register / local, shared, or global video memory. Ideally, all kernel variables fit in register memory and are shared between CUDA threads at compile time [16]. The shared data for the cores is stored in the global video memory, and the shared data is used either as a replacement for the local or for the shared data of the CUDA block. Allocating and accessing the global video memory is long and

can significantly impact performance when calling many CUDA threads. As a result, if the amount of data is known, it is easier to allocate memory from the main program. Of course, there are times when this mechanism is irreplaceable, but at this stage of testing, We decided to ignore it. The test environment translator was developed using Lex and Yacc software and is an LL (1) translator [17].

The input is a kernel program, a description of the parameters, the parameters themselves for launching and the number of kernel instances. Then the code is translated into an intermediate representation in the form of a syntax tree. At this stage, a table of data objects is built, taking into account their type. The code is run and executed, while statistics are recorded on accesses to different types of memory and some other data [18].

The implementation of the emulator allows you to trace data streams and divide the program into a set of independent functions, which in turn can be parallelized using the method described above.

VI. Conclusion

In this work, neural network model development methods were analyzed and a new approach was proposed. The layers approach in the development workflow provides faster and flexible way of model training. Although, this approach can slow down training process, it saves a lot of time during model development for large datasets and provides a flexible way for model sharing.

References

[1] Singh P., Manure A. *Learn TensorFlow 2.0: Implement Machine Learning and Deep Learning Models with Python*, 1st ed. New York: Apress, 2020, ISBN: 978-1484255605.

[2] Apache License, Version 2.0, [Online]. Avialable: https://www.apache.org/licenses/LICENSE-2.0 (visited: 10/14/2021).

[3] Atienza R., *Advanced Deep Learning with TensorFlow 2 and Keras*, 2nd ed. United Kingdom, Birmingham: Packt Publishing, 2020, ISBN: 978-1838821654.

[4] Open Sourcing TensorFlowOnSpark: Distributed Deep Learning on Big-Data Clusters, [Online]. Available: https://developer.yahoo.com/blogs/157196317141/ (visited: 09/04/2021).

[5] Verbraeken J., Wolting M., Katzy J. & Kloppenburg J., Verbelen T., Rellermeyer J. A Survey on Distributed Machine Learning. ACM Computing Surveys, 2020, V. 53, pp. 1-33, doi: 10.1145/3377454.

[6] Ravishekhar Banger, Koushik Bhattacharyya. OpenCL Programming by Example. M.: Packt Publishing, 2013. – 304 c.

[7] Sanders Jason, Kandrot Edward. CUDA by Example: An Introduction to General-Purpose GPU Programming. Prentice Hall, 2010. – 290 c.

[8] Rovnyagin M. M., Sinelnikov D. M., Odintsev V. V., Varykhanov S. S. Distributed Fault-tolerant Platform for Web Applications. 2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus), 2020, pp. 482-485, doi: 10.1109/EIConRus49466.2020.9039529.

[9] Shallue C. J., Lee J., Antognini J., Sohl-Dickstein J., Frostig R., Dahl G. E. Measuring the Effects of Data Parallelism on Neural Network Training. Journal of Machine Learning Research 20, 2019, pp. 1-49.

[10] Chi-Chung C., Chia-Lin Y., Hsiang-Yun C. Efficient and Robust Parallel DNN Training through Model Parallelism on Multi-GPU Platform. arXiv: 1809.02839 [cs.DC], 2019.

[11] Kai H., Yaogen L., Xia M., Jiasheng W., Meixia L., Shuai Z., Liguo W. Federated Learning: A Distributed Shared Machine Learning Method. Complexity, 2021, V. 2021, doi: 10.1155/2021/8261663.

[12] Kamenskikh I. S., Sinelnikov D. M., Kalintsev D. S., Kozlov A. A., Rovnyagin M. M., Shulga D. A. Software development framework for a distributed storage and GPGPU data processing infrastructure. 2016 IEEE NW Russia Young Researchers in Electrical and Electronic Engineering Conference (EIConRusNW), 2016, pp. 216-219, doi: 10.1109/EIConRusNW.2016.7448158.

[13] S. J. Pan and Q. Yang. A Survey on Transfer Learning. IEEE Transactions on Knowledge and Data Engineering, V. 22, pp. 1345-1359, 2010, doi: 10.1109/TKDE.2009.191.

[14] N. Poulton, *The Kubernetes book.* United Kingdom: Nigel Poulton, 2018, ISBN: 978-1521823637.

[15] The MNIST Database, [Online]. Available: http://yann.lecun.com/exdb/mnist/ (visisted: 08/17/2021).

[16] Holewinski J., Pouchet L.-N., Sadayappan P. High-performance code generation for stencil computations on GPU architectures // Proceedings of the 26th ACM international conference on Supercomputing - ICS '12. New York, New York, USA: ACM Press, 2012. P. 311.

[17] Levine John. Flex and Bison. M.: Wiley, 2009. – 292 p.

[18] Ferreira A.B., Matias R., Maciel V.F. An exploratory study on patterns in dynamic memory allocations // Brazilian Symposium on Computing System Engineering, SBESC. IEEE Computer Society, 2016. Vol. 0. P. 40–47.