DeepDendro: Parallel C++ Neural Network Framework

Yaroslav Korch
Faculty of Applied Sciences
Ukrainian Catholic University
L'viv, Ukraine
yaroslav.korch@ucu.edu.ua

Matvii Prytula
Faculty of Applied Sciences
Ukrainian Catholic University
L'viv, Ukraine
matvii.prytula@ucu.edu.ua

Anastasiia Senyk
Faculty of Applied Sciences
Ukrainian Catholic University
L'viv, Ukraine
anastasiia.senyk@ucu.edu.ua

***Abstract*—Deep learning models often require large amounts of data to be trained effectively. Hierarchical clustering algorithms can be used to group similar data points together and reduce the amount of data required for training. However, traditional hierarchical clustering algorithms can be computationally expensive and not scalable to large datasets. In this paper, we propose DeepDendro, a parallel system designed specifically for deep learning applications. We implement DeepDendro using C++ and Intel's TBB [1] library for parallelization. Our algorithm uses a PipeDream [2] approach for parallel training of Neural Networks. The proposed algorithm achieves significant speedup compared to traditional hierarchical clustering algorithms while maintaining high accuracy in results.**

***Index Terms*—parallel and concurrent programming, multithreading, C++, machine learning, high-performance computing**

I. Introduction

Deep neural networks (DNN) have achieved remarkable success in various fields, such as computer vision, natural language processing, and speech recognition. However, training DNNs can be computationally expensive and time-consuming, especially for large datasets and complex models. The traditional approach of training DNNs on a single machine can take days or even weeks to converge to an acceptable level of accuracy. This long training time can be a bottleneck for many applications that require fast and efficient training of DNNs.

To address this problem, researchers have proposed various techniques to speed up the training process of DNNs. Some strategies focus on dividing the training data across different machines. This type of parallelism is called *data parallelism*. However, it has its bottlenecks, such as synchronizations. Moreover, it is much more often applicable when dealing with enormous datasets, where one machine with cannot fit all the data into its memory. On the other hand, new complex models also require a lot of parameters to keep track of in order to make them more sophisticated and applicable in real-world scenarios. This is where *model parallelism* can help. It is based on the idea of distributing the model across different devices, each storing its part of the model and updating corresponding parameters. Nevertheless, such models are even more rare in the case of the average user, who trains its deep neural network.

In this report, we present a parallel C++ neural network library that leverages *pipeline parallelism* to enable faster DNN training. Our library is inspired by PipeDream, a system that uses pipeline parallelism to optimize DNN training. However, our library goes beyond PipeDream by providing a user-friendly interface for building and training DNN models in C++. Our library also supports various optimization techniques, such as data parallelism and model parallelism, to further improve the performance of DNN training.

II. Explored Approaches

As the Neural Networks become more and more complex, so does the time required for their training. Also, the computational capabilities of the processors develop with time but not as fast as the market demands. Especially the AI market. There is no chance to wait to improve the learning time of the deep neural network by improving the hardware part. For these reasons, there exists a lot of optimization algorithms to optimize the neural network learning process.

When dealing with a huge amount of data, the first thing that comes to mind is to divide it into smaller parts. In neural network terminology, this technique is called minibatches [3]. They are beneficial for a number of reasons. At first, they allow the use of smaller portions of matrix multiplication, which can be more easily parallelized and thus optimized. Moreover, it allows the application of the crucial part of neural network learning, weight update, to be performed more often. This means that, in general, fewer learning epochs (iterations) are needed in order to achieve the same predicting accuracy. Also, based on minibatch, there exist a lot of elegant optimization algorithms.

A. Intra-Batch parallelism

Intra means in. So, it refers to the parallelization within one mini-batch. That is, when the size of the mini-batch is still large, we can divide it into even smaller parts.

Intra-batch parallelism involves dividing a single batch of input data into smaller subsets and processing these subsets in parallel using different workers or processing units. Each worker processes a different subset of the input data in parallel and computes the gradients for the corresponding subset of the

data. The gradients are then combined across all the workers to update the model parameters.

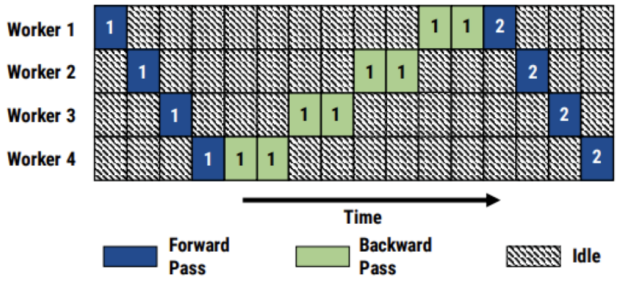

Fig. 1. Intra-Batch Parallelism across 4 workers [2].

As shown in Fig. 1, each smaller subset of the input batch is being processed by a separate worker. However, we can observe a huge amount of idle time. For the record, here it is assumed that, on average, the amount of time spent on backward propagation is twice as much as the amount of time spent on forward propagation for each microbatch.

B. Inter-batch parallelism

In inter-batch parallelism, the input data is divided into multiple batches, and each batch is processed in parallel using different workers or processing units. Each worker trains the model independently on their assigned batch of data, computes the gradients, and updates the model parameters based on the gradients.

Unlike intra-batch parallelism, inter-batch parallelism does not divide the input data into smaller subsets within each batch. Instead, it divides the input data into multiple batches and trains the model on each batch in parallel. The Inter-batch input data batches are typically larger than the subsets used in intra-batch parallelism. Each batch is then processed independently and in parallel by different workers or processing units. This allows for the training of the model on larger amounts of data in parallel, which can accelerate the training process and improve scalability.

C. Inter-batch vs. Intra-batch

The main difference between intra-batch parallelism and inter-batch parallelism is in how the input data is divided for parallel processing.

In intra-batch parallelism, a single batch of input data is divided into smaller subsets, such as mini-batches or micro-batches, and these subsets are processed in parallel by different workers or processing units. The gradients are then combined across all workers to update the model parameters.

In inter-batch parallelism, the input data is divided into multiple larger batches, and each batch is processed independently and in parallel by different workers or processing units. The results are then combined to update the global model parameters.

D. Further improvements – GPipe

GPIPE is a Google project that does a unique thing. It combines the two mentioned types of parallelism, inter-batch and intra-batch. That is, for the purpose of resource allocation, we divide each of our miniBatch into microBatch. At the same time, it gives a gain by reducing the time when one worker waits for the result of another.

Fig. 2. GPipe [4].

According to Fig. 2, as soon as the zero workers have processed the zero microbatch, he passes the result to the next one and starts work on the first one himself, and so on. In this case, we even have periods when all workers simultaneously perform some calculations.

It is important to note that synchronization or connection of all results occurs during the loss calculation in the stage before the back prop. Then, the back prop happens according to the same logic.

III. Asynchronous solution – PipeDream

PipeDream elegantly solves the pipeline bubbles problem during neural network training. It does this using an asynchronous update of the weights, which does not interfere with the process of forward and backward passing in any way.

A. Partitioning Layers Across Machines

Given the model and the set of Machines, PipeDream's first challenge is to automatically partition the layers of the model across available machines so as to minimize overall training time.

The thing is that, usually, starting layers are bigger, while ending layers are smaller, so equally dividing layers between machines is not a good idea. There are three main characteristics given for each layer that can help divide the model equally:

- input size
- output size
- activation time

With these three things, we can figure out how to partition and organize this self-efficient profiling effectively.

B. Work Scheduling

Unlike traditional pipelines, PipeDream is bi-directional. Thus, every machine has to make the choice between two options:

- performs the forward pass, thus pushing the mini-batch to downstream machines,
- performs the backward pass for a different mini-batch, thus ensuring progress in learning.

Always prioritizing forward or backward passes is bad. Therefore, PipeDream alternates between two choices.

C. Implementation

As for our requirements, there was no particular reason for implementing a profiler at this stage. However, the solution for a bidirectional pipeline was highly required. Furthermore, it had to be able to perform asynchronous weight updates, without interrupting forward and backward propagation for each microbatch

1) Pipeline organization: By conventional means, the implementation of a bidirectional pipeline is impossible. Even the flow graph must still be acyclic. However, this mention led us to the idea of expanding the pipeline into a unidirectional one. It is commonly known that the backward propagation follows the forward propagation, and it cannot be changed. Therefore, we do not need to, as the workers, perform both operations of forward and backward passing. Instead, we could involve more workers, where some would perform forward pass, some would perform backward pass, and some would perform the weight update. The worker management, therefore, could be transferred to another library. In our case, that was oneTBB from Intel [1].

The structure of the pipeline is formed out of the TBB's *flow graph* where forward and backward pass form a linear regular pipeline that is two times longer than the bidirectional would be. Moreover, the weight updates are totally different branches of the graph.

2) Weight stashing: In order to implement an asynchronous weight update, it has to somehow affect the forward and backward passes without interrupting them. Therefore, in our implementation, layers serve as storage for weights, activations, and gradients, while nodes of the flow graph are the functions themselves (such as backward propagation, forward propagation, pipeline populate, weight update, and loss function calculation).

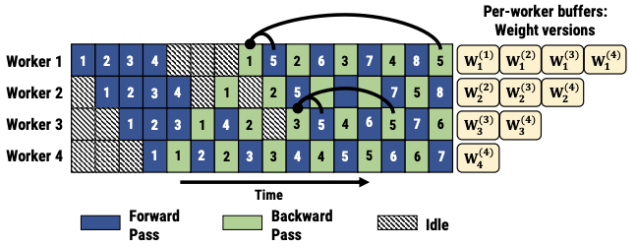

Fig. 3. PipeDream Weight stashing [2].

If we closely look at Fig. 3, we can see that even when the microbatch was processed by the worker in forward pass, it still has to be run through the backward pass node. And the crucial thing here is that the weights used for forward and backward passes have to be the same. Now, imagine that an asynchronous update happened in the middle of the process of a microbatch traveling from the forward node to the backward one. This means that the weights in the first case will differ from those in the second. This could cause our model some problems. Most likely, the loss function would not decrease.

The solution to this problem is weight stashing, which means that for the microbatches available in the pipeline at the moment, we store the version of weights by which it performed its forward pass. By doing this, we can always perform backward propagation using the version of weights. It does require additional memory. However, there is an endless trade-off between memory, time, and correctness.

3) Results measuring: Since the work is being spread among different threads for the time computation decrease aims, the process of time, accuracy, and loss function decrease measuring is quite challenging. Furthermore, each worker is processing a different microbatch, which is a different portion of data. That is, to clearly measure such values, we would need to synchronize them, then measure and keep on training. However, synchronization here means either flushing the pipeline or stopping it. Both cases would result in the efficiency loss. The further results are displayed in the corresponding section.

IV. Convolutional Neural Networks

A. Convolutional Layers

The convolutional layers in NN's are widely used for image classification, facial recognition, autonomous driving, etc. They thrive because of the computations' efficiency and the kernels' reusability. Once the CNN has trained a kernel, it is used for the whole input rather than in one place, as in ANNs.

We have implemented 2D and 3D Convolutional Layers that use 3D and 4D tensors, respectively. One extra dimension is for the number of elements in one batch. For example, for classifying 2D images of the MNIST dataset, one would use $(batch\_size, 28, 28)$ tensors. The constructor of a layer accepts the number of filters, a filter's shape, the activation function (optional), and the input shape.

B. Max Pooling

There are two most popular types of pooling in CNNs — Max Pooling and Average Pooling. However, Max Pooling has been more useful during the last decade.

Max Pooling allows one to shrink the input shape while keeping the essential features of the input. It is fast, requires no learning, and is easy to use. For example, 2×2 max pool will reduce the input's size by 4.

C. Flattening

When the convolution-related parts of NN's architecture have ended, one needs to connect the agglomerated input to the ANN, and this is where flattening layers help. They are used to reshape the input to a convenient dimension for dense layers (usually 2D matrices) during forward propagation and to transform the gradient back to the original shape during backpropagation.

V. Results and analysis

A. MicroBatch loss function variation

Since the whole dataset is divided into smaller parts, the loss function decrease is not as smooth as one would be in the sequential model presented in Fig. 4.

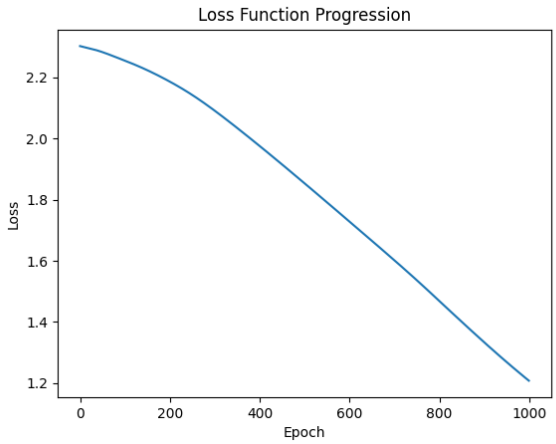

Fig. 4. Loss function decrease plot, Sequential model

On different workers with different microbatches, it rather looks like on this Figure 5, with noticeable noise. However, we can clearly see that, in general, it is decreasing. The epochs axis now does not really represent the progress we want for the parallel model, as a weight update is being done after some number of microbatches have been processed.

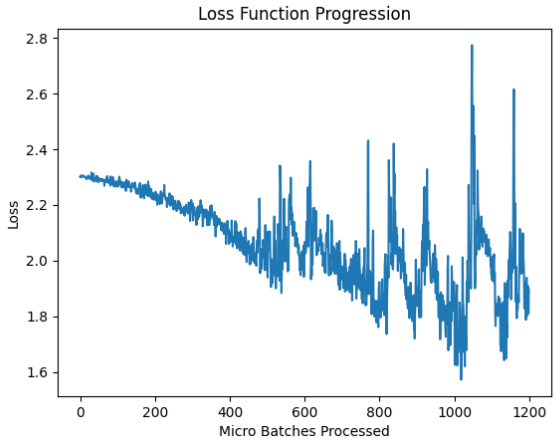

Fig. 5. Loss function decrease plot, Parallel model + Microbatches

B. Productivity

Productivity comparison depicted on Fig. 6 was done on the same dataset MNIST [5] with the same network configuration:

- first inner layer 16 neurons, ReLU activation function,
- second inner layer 8 neurons, ReLU activation function.

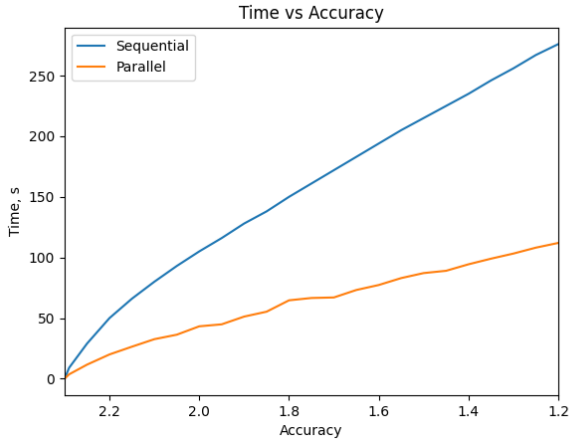

Fig. 6. Productivity comparison, Parallel and Sequential Model.

The loss on microbatches was taken by averaging each 1000 processed microbatches and their loss function.

As it can be seen, the parallel model is near **2.5** times faster than the sequential one.

C. Results of CNN

We have chosen to test our framework on the MNIST dataset. Let us consider the following sequential model:

- ConvolutionalLayer2D: 4 filters, each has shape 5×5, batch size $= 10$.
- FlatteningLayer2D.
- DenseLayer: 32 neurons, activation function — ReLU.
- DenseLayer: 10 neurons (as classes of digits), activation — Softmax.

In Fig. 7, each batch included ten images (to use vectorization), and overall 18000 iterations were performed, which resulted in only 3(!) epochs. The loss function values are averaged within 100 elements to cover the spikes.

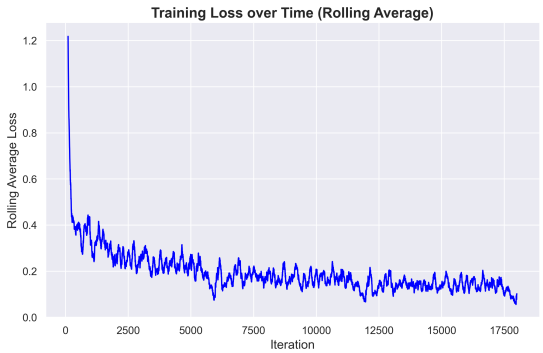

Fig. 7. Averaged Loss function, ConvLayer + DenseLayers.

Time: 40.7948 seconds, accuracy: 95.2778%. After only $\approx$ 40 seconds on the machine, the accuracy is 95%+, which is great, as this test did not include parallelization, and the architecture consisting only of dense layers for this task took

7 minutes to achieve $\approx 91\%$ accuracy. This shows the power of convolutional layers and the speed of C++ language.

D. Comparison with other frameworks

One can compare our approach with Python's TensorFlow [6]:

```
Epoch 1/5
 13s - loss: 0.2910 - accuracy: 0.9143

Epoch 2/5
 12s - loss: 0.1553 - accuracy: 0.9545
```

It can be seen that TensorFlow's implementation took 25 seconds to achieve the same accuracy score. Our team considers this a not-bad performance, given the difference of 60%.

VI. Conclusion

This paper introduces DeepDendro, a parallel neural framework designed specifically for deep learning applications.

Implementing DeepDendro using C++ and Intel's TBB library and combined with the novel algorithm based on PipeDream's approach enables efficient parallelization and brings significant advantages over traditional hierarchical clustering algorithms. In addition, the framework also provides the possibility of using convolutional layers, which are widely used in image classification tasks.

The findings of this study offer insights into the impact of parallelization on various aspects of neural network training, such as data parallelism and model parallelism. By evaluating the performance, speedup, and productivity of DeepDendro compared to traditional approaches, this study provides empirical evidence of the advantages and trade-offs associated with parallelization in deep learning tasks.

The code of the project is available at https://github.com/anastasiasenyk/DeepDendro.

References

[1] "Intel onetbb documentation," Intel Corporation, 2023, accessed on May 30, 2023. [Online]. Available: https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html%23gs.z664wb

[2] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: Generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, ser. SOSP '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 1–15. [Online]. Available: https://doi.org/10.1145/3341301.3359646

[3] X. Peng, L. Li, and F. Wang, "Accelerating minibatch stochastic gradient descent using typicality sampling," *CoRR*, vol. abs/1903.04192, 2019. [Online]. Available: http://arxiv.org/abs/1903.04192

[4] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, "Memory-efficient pipeline-parallel DNN training," *CoRR*, vol. abs/2006.09503, 2020. [Online]. Available: https://arxiv.org/abs/2006.09503

[5] Y. LeCun and C. Cortes, "MNIST handwritten digit database," http://yann.lecun.com/exdb/mnist/, 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[6] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: https://www.tensorflow.org/