Parallelization of Error Back-Propagation Algorithm in Neural Networks

Mallegowda M.
Department of Computer Science
(Assistant Professor)
Ramaiah Institute of Technology
Bengaluru,Karnataka

Kushagra Gupta
Department of Computer Science
(Student)
Ramaiah Institute of Technology
Bengaluru,Karnataka

Anushka K
Department of Computer Science
(Student)
Ramaiah Institute of Technology
Bengaluru,Karnataka

Abstract— **Back-propagation training of artificial neural networks is a computationally intensive machine learning procedure. The neural networks can be trained to calculate the error function with regard to all of the weights using the back propagation algorithm. To increase productivity and save time, parallel programming is done using the Open MP architecture. It is used to generate neural networks that are more effective. This method runs the algorithm concurrently. This study provides a comparative analysis of the execution time of serial code and parallel code of the back propagation algorithm.**

Keywords—back propagation algorithm, OpenMP, Neural Network, parallel

I. Introduction

Machine learning is a subset of artificial intelligence that uses complex algorithms to teach computers how to learn from experience and make decisions. It allows computers the possibility to learn without explicit programming. There are various applications of machine learning in artificial intelligence. Neural information processing, marketing and the social sciences, bioinformatics, classifying DNA sequences, and robotics, etc. are some of the specific applications.

Inspired by the structure and function of the brain, artificial neural networks are a subfield of AI. An artificial neural network (ANN) is a type of computer network inspired by the biological neural networks that shape the human brain. Artificial neural networks, like the neurons in a biological brain, consist of nodes that are connected to one another across several levels of the network. These nerve cells are called nodes. The neurons may talk to one another thanks to the connections between them. A neuron's choice is sent to its neighbours. The output of a node is referred to as its activation or node value, and the weight of a link is proportional to the strength of the connection it makes.

Backpropagation is a machine learning technique that finds the value that minimises the loss function by computing the gradient of the loss function. Chain rule calculus is used to determine the gradient as it travels through the layers of a neural network. Small steps in the direction of the gradient get us closer and closer to the minimum value, which we may reach via gradient descent. The gradient of a loss function relative to each weight in the network is calculated in this way.

The optimization technique modifies the weights based on the gradient in order to minimise the loss function. The gradient of the loss function must be calculated in back propagation for each input value in order to determine the intended output.

A method for creating multi-threaded applications is called open MP. It offers a set of compiler pragmas, directives, function calls, and environment variables that are platform neutral so that parallelism can be used. The required number of threads are created by the master thread. To increase the back propagation algorithm's efficiency while using the threads that are available, we use Open MP. Additionally, as the number of threads increases, the software becomes faster. In back propagation, parallelism is possible through node parallelism. Each node in this case is examined to update the weight.

II. Related work

A number of papers on multiprocessing and ANNs methods have been published, demonstrating the usefulness of parallel architectures. In [4], a distributed memory-multiprocessor system is used to model a fully connected multi-layer neural network using a back-propagation technique. The model is divided into subnetworks, and each subnetwork is mapped to a different processor. In [2], two back-propagation using OpenMP implementations are examined; one splits the hidden layers among processors, while the other divides the inputs while keeping a full copy of the network on each CPU. Another GPU-based solution is given in [1], where ANN training is modelled as a matrix multiplication.

In [2], we see a partitioning technique for multilayer networks that employ backpropagation. In order to speed up the learning process, the partitioned network is mapped into a set of workstations.

This CUDA version of an image processing and pattern recognition method uses the NVIDIA CUDA Basic Linear Algebra Subroutines (CUBLAS) to establish and solve the network matrix for the training stage, resulting in a speedup of about 15 times compared to CPU implementations. There is a description of the SpiNNaker MPCS in [9]. It is characterized by highly interconnected processing nodes and tremendous parallelism in data processing. The goal is to get the same kind of computational results as a neural network, and to be able to model brain circuits with 109 neurons.

The authors demonstrate that a biological neural network of the same size would function at the same pace.

In [8], a GPU-based implementation of a zero-order Takagi-Sugeno-Kang (TSK)-type Fuzzy Neural Network (FNN) training method is devised. To accelerate ANN training, a parallel implementation of a Spiking Neuronal Network is presented in [14] by means of the CUDA technology. It was in [23] that the neural network model known as Locally connected Neural Pyramid (LCNP) was first proposed. Using

NVIDIA CUDA, LCNP is tailored for high-throughput, large-scale object identification.

III. Architecture

The architecture for the proposed study is as follows:

1. Input the dataset required for the execution of back propagation algorithm. The BP algorithm is serially computed, meaning that computer code is generated and executed in a serial fashion.
2. Output in the form of changing the weights produces the desired outcome by altering the neural network's weights. Additional strategies include profiling the code using a flat profile and a call graph, among others.
3. It is being done to optimize the code from o2 to o4 levels and to parallelize the code using Open MP. Benchmarking is carried out, which involves evaluating serial and parallel code on various system architectures.
4. It is done to assess how well the system performs in terms of CPU usage, memory usage, and I/O communication.

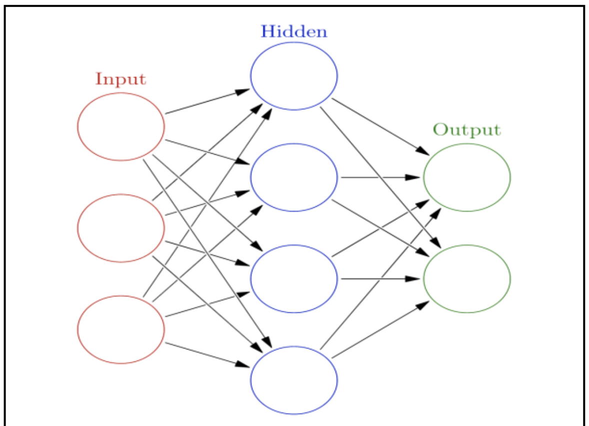

Fig 1. A Simple Artificial Neural Network

IV. Implementation

This Backpropagation method has been trained using data from the fashion MNIST dataset. The data is sampled at various sizes (24MB, 96MB, 250MB, 500MB, 750MB and 1GB). Different picture pixels represent the properties of the dataset. Make use of the Backpropagation algorithm on this data collection. A new estimate can be calculated by revising the weights. Finally, we determine how long it takes to run serial code, how much memory is required, and how long it takes to send and receive data via I/O. In the wake of this process, we profile the code and attempt to enhance it. After that, we parallelize the code using Open MP and compare the execution time, memory footprint, and I/O traffic to that of the original serial code. In the end, we benchmark the serial and parallel programmes on Intel 64-bit and Intel Xeon processors.

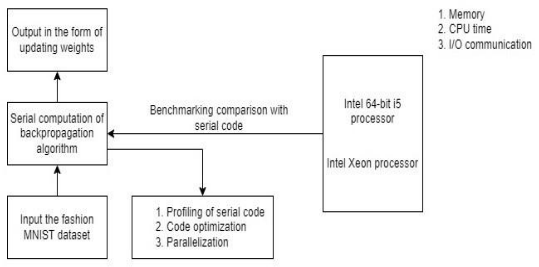
Output in the form of updating weights
Serial computation of backpropagation algorithm
Input the fashion MNIST dataset
Benchmarking comparison with serial code
1. Profiling of serial code
2. Code optimization
3. Parallelization
Intel 64-bit i5 processor
Intel Xeon processor
1. Memory
2. CPU time
3. I/O communication

Fig 2. Basic architectural diagram

V. Results

Table I. Comparison of serial and parallel code with respect to CPU Time (in seconds) (Intel 64-bit processor)

Size of Dataset	Serial code	Parallel code	
		4 threads	8 threads
24MB	20.07	19.37	19.36
96MB	81.54	77.73	76.83
250MB	219.50	212.09	210.44
500MB	442.81	404.22	400.3
750MB	642.76	572.63	567.70
1GB	887.84	801.43	801.66

Table II. Memory (in bytes) used differently by serial vs parallel programs (Intel 64-bit processor)

Size of Dataset	Serial code	Parallel code	
		4 threads	8 threads
24MB	12191	12122	12232
96MB	12324	12384	12430
250MB	7045	6146	6187
500MB	6985	6224	6279
750MB	9495	9521	9565
1GB	9678	9716	9757

Table III. Memory (in bytes) used differently by serial vs parallel programs (Intel Xe-on processor)

Size of Dataset	Serial code	Parallel code	
		4 threads	8 threads
24MB	21.93	19.12	18.92
96MB	86.54	77.28	78.44
250MB	223.57	215.09	213.37
500MB	442.4	424.42	428.28
750MB	633.77	605.47	610.95
1GB	914.83	851.4	854.12

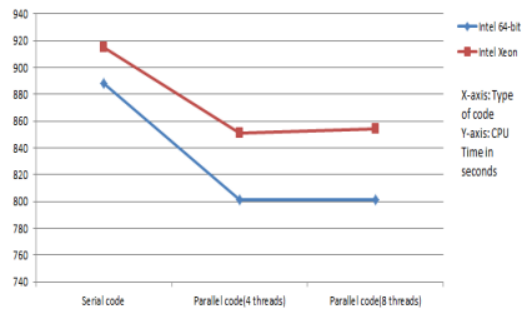

Fig 3. Graph showing the relative CPU time (in seconds) spent on a 1GB dataset while using serial versus parallel code.

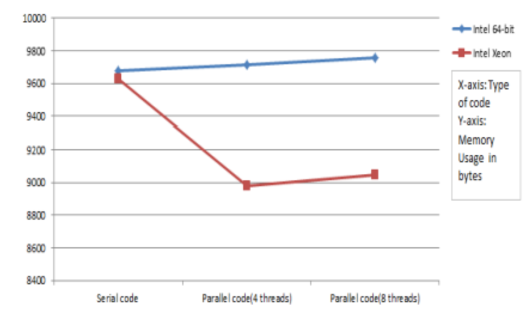

Fig 4. Memory (in bytes) use on a 1 GB dataset, plotted as a comparison between serial and parallel programs.

As we can see in the graphs and the tables presented above parallelization of back propagation algorithm results in lower CPU time. The Intel i5 CPU has significant lower memory usage too.

VI. Conclusion

Several machine learning algorithms were analysed. The most productive algorithm was the back propagation one. This algorithm's serial code was applied to a variety of datasets, and the results were various lengths and running on a 64-bit Intel CPU. In this section, we discuss the results of our performance analysis with regards to central processing unit (CPU) time, memory consumption, and I/O communication. Following that, code optimization and profiling were performed. Then, for each dataset, we run parallel code written in OpenMP and assess its performance based on the aforementioned criteria.

We then run benchmarks on an Intel Xeon CPU to compare the two programmes. The execution time, memory usage, and I/O communications of a serial programme are all larger than those of a parallel programme. Moreover, these variables diminish as the number of threads grows. Furthermore, unlike the Intel 64-bit CPU, the Intel Xeon enables for the seamless execution of serial as well as parallel code.

References

[1] Jiang, P. Chen, C. and Liu, X. 2016. "Time series prediction for evolutions of complex systems: A deep learning approach," 2016 IEEE International Conference on Control and Robotics Engineering (ICCRE), Singapore, pp. 1-6.

[2] V. Boyer and D. El Baz, Recent advances on GPU computing in Operations Research, 27th IEEE International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW 2013), pp. 1778–1787, 2013.

[3] H. Jang and A. Park and K. Jung, Neural network implementation using CUDA and OPENMP, Digital Image Computing: Techniques and Applications (DICTA], pp. 155–161, 2008.

[4] M. M. Khan and D. Lester and L. Plana and A. Rast and X. Jin and E. Painkras and S. Furber, SpiNNaker: mapping neural networks onto a massively-parallel chip multiprocessor, Neural Networks, IEEE World Congress on Computational Intelligence, pp. 2849– 2856, 2008.

[5] D. Kirk, and W. Wen-Mei, Programming massively parallel processors: a hands-on approach, Newnes, 2012.

[6] T. Nowotny, Parallel implementation of a spiking neuronal network model of unsupervised olfactory learning on NVIDIA CUDA The 2010 International Joint Conference on Neural Networks (IJCNN), pp. 1–8, 2010.

[7] Chen, X. and Long, S. 2009. "Adaptive Multi-versioning for OpenMP Parallelization via Machine Learning," 2009 15th International Conference on Parallel and Distributed Systems, Shenzhen, pp. 907-912.