International Conference on AI and the Digital Economy (CADE 2024) Venice, Italy
24–26 June 2024

Improving CNN Model Training Time Efficiency using MPI-driven Parallelization and Ensemble Learning

Zia Ur Rehman[1,*], Saif ul Islam[2], Uzair Hassan[1], Jalil Boudjadar[3], and Rajkumar Buyya[4]

[1] *Institute of Space Technology, Islamabad, Pakistan,*
[2] *WMG, University of Warwick, Coventry, UK,*
[3] *Aarhus University, Aarhus, Denmark,*
[4] *The University of Melbourne, Melbourne, Australia*
**Email: zia.rahman@ist.edu.pk*

Abstract

Convolutional Neural Networks (CNNs) have proven remarkably effective in various computer vision applications, such as object detection, image segmentation, medical imaging, and classifying handwritten numbers. On the other hand, CNN training on large datasets can be highly compute-intensive, resulting in lengthy training times. To address this problem, this paper proposes a novel method to reduce CNN training time for classification problems. To achieve this, we divide the training dataset across several CPU cores using Message Passing Interface (MPI) parallelization. Furthermore, we employ an ensemble learning method that combines the predictions of separate models trained on various subsets of the dataset using a majority voting scheme. We consider the MNIST dataset for the experiments. The results of our experiments show that, while retaining promising accuracy, the approach we propose significantly reduces the CNN model's training time up to 45.13% and 91.67% depending upon the machine specifications compared to the sequential approach.

1 Introduction

In recent years, deep learning models, particularly Convolutional Neural Networks, have shown remarkable performance in various fields such as computer vision, natural language processing, and weather forecasting [1]. CNNs have demonstrated outstanding performance across various applications, significantly transforming the landscape of computer vision [2]. Specifically, CNN model for RGB-D image-based high-accuracy 6D pose estimation, an unsupervised anomaly localization by CNN framework, and an object classification by Voxel Graph CNN [3–5]. CNNs are excellent at identifying features from images, which enables them to interpret deep patterns and connections in the data.

Due to their complexity, CNNs for object recognition and classification are usually tested as prototypes on powerful computing platforms such as GPU cards. Although these systems allow many algorithms to be implemented in real time, they have a major power consumption disadvantage. To implement CNNs as effectively as possible, there has been a recent rise in the creation of specialized processors and hardware accelerators emphasizing decreasing the loss of power and inference time [6].

Utilizing multiple CPU cores can divide the training process into smaller tasks that can be executed simultaneously. This parallel processing allows for faster training times and improved overall performance of the CNN model. Furthermore, parallelization of training on CPU cores can also help overcome the speed and power consumption limitations often associated with CNN models [6]. This is important in real-life applications with critical time and power constraints. By utilizing CPU core parallelization, the training time for CNN models can be significantly reduced, allowing for faster model development and deployment in various applications.

Training CNNs can be costly and time-consuming, especially with large datasets. This is primarily due to the sequential nature of typical CNN training, which processes data one sample at a time. Modern multi-core processors are limited in their ability to be used due to their sequential nature, which slows training and prevents widespread use for time-sensitive real-world tasks. This paper investigates how CPU core parallelization using MPI affects CNN model training time. We demonstrate the efficiency of our proposed method using the MNIST dataset [7], a popular benchmark for handwritten digit classification. We evaluate the accuracy and training time of a CNN model trained sequentially against a model trained using MPI parallelization. According to our research, MPI parallelization can reduce training times without affecting classification accuracy, allowing CNNs to be trained and used more quickly in practical applications.

In this paper, we investigate the performance and training time of the Convolutional Neural Network (CNN) model on a CPU with and without parallelization. Processor efficiency drops when a CPU's single core is used instead of multiple cores, which results in slower execution times and reduced overall performance. CPU parallelization resolves this problem and saves energy and time by utilizing MPI to divide work across several CPU cores. Initially, we implemented a CNN model for the MNIST dataset without parallelization to measure test accuracy and training time. Next, we implement MPI to create a distributed CNN training method in which each MPI process trains a local model on a subset of the MNIST dataset. Each process receives a portion of the dataset and uses it to construct and train a CNN model locally. Following training, all trained models are combined at the root process, using a majority voting scheme to generate predictions on the test dataset. This produces an ensemble model with the same accuracy as a model without parallelization but requires less training time.

The rest of the paper is organized as follows: In section 2, we discuss related work consisting of CNN classifiers, MPI, and ensemble learning and their detailed discussion. Section 3 presents the methodology, which involves the proposed MPI-based technique and algorithm for parallelization. Section 4 presents the experimental evaluation, which includes a discussion related to the dataset, model configurations, hardware and software setup, results, and discussions. Finally, section 5 concludes the paper.

2 Related Work

Computer vision is an essential component of artificial intelligence. It uses human vision patterns to recognize and interpret things in pictures and videos. It has a wide range of applications, including facial recognition, self-driving automobiles, and medical diagnostics.

2.1 Convolutional Neural Networks

Convolutional Neural Networks are powerful instruments for image-oriented tasks. They effectively utilize encoded characteristics for picture classification and contribute to the advancement of computer vision technology [8]. A CNN is an effective deep-learning technique for image analysis that minimizes the need for manual preprocessing by using filters acquired during training to identify significant features. It uses convolution to extract features and pooling to minimize dimensions, frequently with maximum or average pooling, to excel in image classification and sequential data applications like natural language processing [8].

In CNN, the input layer contains pixel data. ReLU activation comes after the convolutional layer computes neuron outputs based on local input areas. Pooling lowers spatial dimensions, fully connected layers produce class scores, and ReLU enhances interlayer performance. CNN layer parameters use learnable kernels that slide across input dimensions to produce 2D activation maps in convolutional layers. By dividing the input into 2D spaces and using techniques like max and average pooling, pooling layers reduce dimensionality and improve noise robustness.

Activation function ReLU improves nonlinear features and trains more quickly because of its computational efficiency and simplicity than other CNN activation functions like sigmoid or hyperbolic tangent. In a CNN, fully connected layers consider every component of the input feature and compute the weighted sum of the previous layer features to identify particular targets.

Marina et al. [6] investigate CNN requirements for a variety of applications, with an emphasis on inference time and power consumption. The study elaborates on the importance of keeping time and power limits to set up CNN effectively. It also emphasizes the necessity of effective hardware use by highlighting the connection between execution time and power consumption. It also highlights how important low power consumption is, especially for battery-powered computers, and how different application domains have varied time limitations.

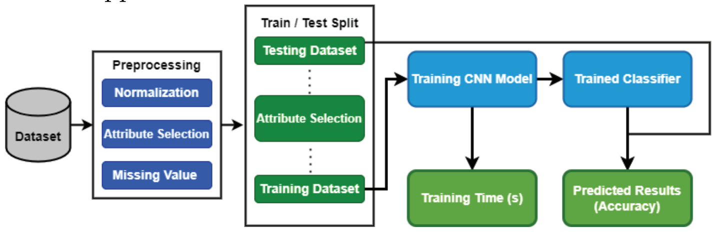

Figure 1: Sequential Technique for Training a CNN Model.

Figure 1 illustrates the traditional sequential technique for training a CNN model, starting with preprocessing steps like normalization, attribute selection, and handling missing values. The dataset is then split into training and testing sets, with the training set used to train the CNN model and the testing set to evaluate its performance. The resulting trained classifier is assessed based on training time and the accuracy of the predicted results.

2.2 Massage Passing Interface

Message Passing Interface, or MPI, is a standardized message-passing system that can operate on various parallel computers. For scientific languages like Fortran, C, $C++$, and Python, it enables parallel programming in multi-core systems and defines syntax and semantics for library functions.

Multiprocessing becomes possible for Python by MPI, which expands these capabilities. It allows for effective parallel processing in Python and has an object-oriented interface that corresponds precisely with MPI-2 C++ connectors [9].

MPI enables parallel programming on multi-core systems by allowing communication and coordination between multiple processes. Python programs can use MPI to split tasks among several cores, using each core's processing capacity. This allows for the effective running of experiments, data processing jobs, and parallel computer programs, which results in large computation time savings. Additionally, MPI's standardized message-passing protocol guarantees flawless portability and interoperability across multiple parallel computing architectures.

Zina et al. explore the development of Python parallel programming, focusing on the historical issues brought up by the Global Interpreter Lock (GIL) [10]. They investigate various Python packages and frameworks that facilitate multiprocessing and parallel processing. A special focus is on MPI, a necessary tool for distributed memory and multi-core systems to achieve parallelism. The study focuses on how MPI helps accelerate processing in many fields, such as cyber security, multimedia, high-performance computing, and optimization methods. The research also emphasizes the importance of using MPI in Python ecosystems to utilize parallel computing resources efficiently [10].

The study [11] presents mpi4py futures, a utility that uses the MPI to speed up Python task execution. This tool is easy to integrate into existing codebases because it closely mimics the interface of Python's concurrent futures package. Performance tests on distributed and shared memory systems showed that mpi4py.futures outperformed Python's native concurrent futures package in terms of throughput and bandwidth. Furthermore, on a Cray XC40 system, mpi4py futures performed better overall than Dask, a well-known Python parallel computing library, in several scenarios [11].

The article [12] examines the features that modern machine learning and deep learning frameworks provide to facilitate development. It explores how training and inference can be greatly accelerated by adjusting these aspects, particularly parallelism.

2.3 Ensemble Learning

Through numerous voting mechanisms and various data predictions, ensemble learning uses many machine learning or deep learning algorithms to improve predictions than those produced by individual algorithms [13].

The fundamental concept of a typical ensemble classification model is shown in Figure 2, which involves two primary steps. First, a consistency function combines the various classification outcomes using several weak classifiers. The final forecast results from this procedure are obtained through various voting techniques. Ensemble models improve predicted accuracy and performance in classification problems by utilizing the combined expertise of several classifiers [13].

Deep learning architectures are now beating standard models in ensemble learning, which integrates multiple models for enhanced adaptability. Deep ensemble models improve generalization by combining ensemble and deep learning methods. The authors in [14] categorize modern deep ensemble models and their applicability in different fields. They also suggest possible directions for future ensemble learning research.

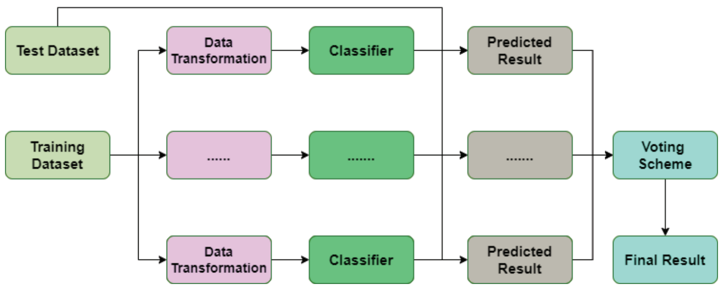

Figure 2: Ensemble Learning Classification [13].

3 Proposed Method

As shown in Figure 3, the proposed technique divides the training dataset into subsets and distributes them to multiple CPU cores. The proposed approach trains a local CNN model on each core with an assigned data subset. After training, the cores communicate to gather the trained models on the root process.

The gathered models from each core are integrated using an ensemble learning technique, such as majority voting, to generate the final model. Finally, the overall performance of this final model is assessed using a different testing dataset.

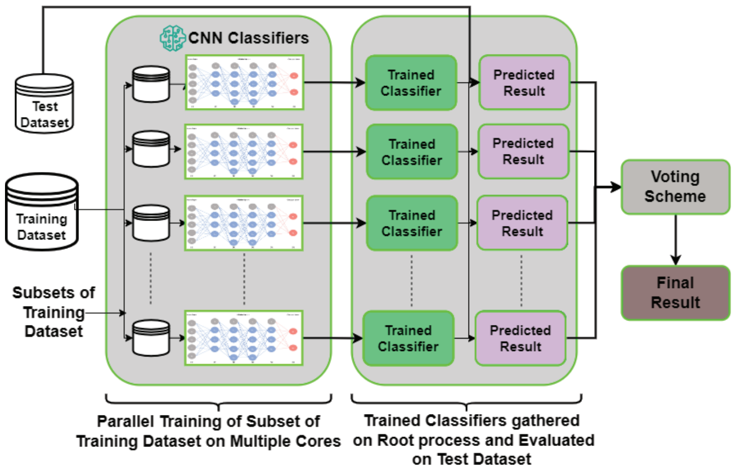
CNN Classifiers
Test Dataset
Training Dataset
Subsets of Training Dataset
Trained Classifier
Predicted Result
Voting Scheme
Final Result
Parallel Training of Subset of Training Dataset on Multiple Cores
Trained Classifiers gathered on Root process and Evaluated on Test Dataset

Figure 3: Proposed MPI Based Ensemble Learning Model for CNN Classification on multi-core systems.

Algorithm 1: MPI Based Parallel Ensemble Learning

```
 1: Input: MNIST Dataset
 2: Output: Average training time of models, ensemble
    model test accuracy, precision, recall, F1-score, and accuracy
 3:
 4: procedure ParallelCNNModelTraining
 5:     MPI Initialization
 6:     Load and preprocess the MNIST dataset
 7:     Data distribute among MPI processes
 8:     Define the CNN model architecture
 9:     Build the local CNN model
10:     Train the local CNN model on the training data chunk
11:     Gather total training times from all processes to process 0
12:     Gather all trained models to process 0
13:     if $rank = 0$ then
14:         Predict classes for the test dataset using all models
15:         Majority voting scheme for Ensemble prediction
16:         Calculate ensemble model test accuracy
17:         Calculate precision, recall, F1-score, and accuracy
18:         Print ensemble model test accuracy, precision, recall, F1-score, and accuracy with average training time
19:     end if
20: end procedure
```

Algorithm 1 works as follows: MPI is initialized to enable coordination and communication between parallel processes. Subsequently, the total number of processes is identified, and each process is given a distinct rank (a unique identification number). The number of MPI processes determines the splitting of the training dataset into chunks. The model is trained on each process's unique subset of data.

TensorFlow/Keras is used to build the CNN model architecture. To normalize pixel values and one-hot encode labels, the MNIST dataset is loaded and preprocessed. The main function trains the CNN model using a specified subset of the training data and gathers the models on the root process. Every MPI process simultaneously builds its local model and trains it on the assigned chunk of the data. The time module determines how long each process takes to train the model. The MPI function *comm.gather* is used at the root process to collect the total training time of each process. Following model training, each model makes predictions on the test dataset. The final ensemble prediction is then made by combining all of the predictions by a majority voting scheme. The model's accuracy is determined by comparing the ensemble model's predictions with the test dataset's actual true labels. The MPI-based parallelization approach significantly reduces average training time without sacrificing model accuracy, making it appropriate for large-scale deep learning tasks on multi-core systems. It divides the training workload among multiple processes and aggregates the results using the ensemble learning technique.

4 Performance Evaluation

This section presents the experimental setup and discusses the results. In the experimental setup, we explain dataset and model configurations and specify hardware and software details in Table 1 and Table 2. Later on, we discuss experimental results.

4.1 Experimental Setup

4.1.1 Dataset

Handwritten digit recognition has long been used as a benchmark for machine learning and deep learning algorithms, and it forms a fundamental challenge in recognizing optical characters. The widely used MNIST database makes evaluating such algorithms easier, which provides pre-processed handwritten digits for comparison [7]. The MNIST database was created by adapting and modifying the NIST database. It consists of 10,000 test photos taken from the same distribution and 60,000 training images, some of which can be used for cross-validation. They are normalized in size and center with a fixed size, 28x28 pixels, and black and white numbers in each image. As a result, every picture sample vector has 784 dimensions and binary elements. For practitioners looking to use machine learning and deep learning algorithms on real-world data with no preparation work, MNIST provides an easy-to-use dataset [7].

4.1.2 Model Configurations

The CNN model used in our research recognizes handwritten digits on images with a pixel size 28x28. The architecture gradually reduces spatial dimensions by starting with three convolutional layers and ending with max pooling layers. The output shape gradually changes after each convolutional layer, going from 26x26x32 to 11x11x64 and finally 3x3x64. The output is then reshaped by a Flattened layer into a 576-size one-dimensional array to make it easier for Dense layers to process. The retrieved features are refined by these dense layers with ReLU activation, leading to a final Dense layer that outputs probabilities for 10 classes (digits 0-9). Using the Adam optimizer and categorical cross-entropy loss, the model, which has 93,322 parameters in total, is trained on the MNIST dataset.

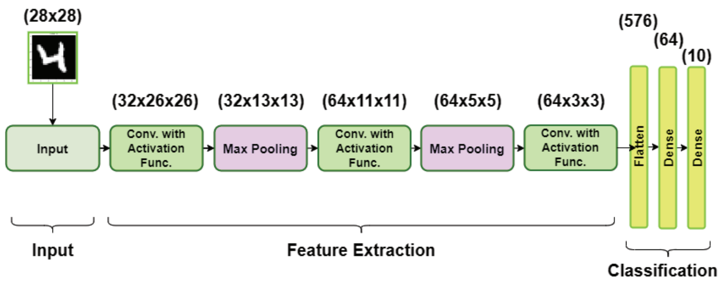

Figure 4: CNN model and configurations used in experiments.

Table 1: Machine 1 hardware & software specifications

Hardware	**Details**
System	Lenovo K14 Gen 1
Processor	11th Gen Intel(R) Core(TM) i5-1135G7 @ 2.40GHz 2.42 GHz (8 cores)
RAM Capacity	8.00 GB (7.70 GB usable)
Software	**Details**
Operating System	Windows 11 Pro 64-bit
Integrated Development Environment	Visual Studio Code
Programming Language	Python
Jupyter Notebook Extension	Jupyter extension for Visual Studio Code
Programming Language	Python (version: 3.11.4)
Tensorflow	(Version:2.15.0)
mpi4py	(Version: 3.1.5)

Table 2: Machine 2 hardware & software specifications

Hardware	**Details**
System	Supermicro GPU SuperServer SYS-220GP-TNR
Processor	Intel(R) Xeon(R) Gold 6354 CPU @ 3.00GHz (72 cores)
RAM Capacity	256 GB
Software	**Details**
Operating System	Ubuntu
Integrated Development Environment	Visual Studio Code
Programming Language	Python
Jupyter Notebook Extension	Jupyter extension for Visual Studio Code
Programming Language	Python (version: 3.11.4)
Tensorflow	(Version:2.15.0)
mpi4py	(Version: 3.1.5)

4.2 Results and Discussions

We consider the accuracy [15] and training time as evaluation metrics to evaluate the performance of the proposed approach against sequential technique.

The experimental findings show that the combination of MPI-based parallelization and ensemble learning approaches significantly improves the effectiveness of the Convolutional Neural Network (CNN) training procedure.

Figure 5 presents the accuracy in % and training time in seconds for sequential technique on two machines. The graph shows that Machine 2 has a slightly higher accuracy (98.53%) compared to Machine 1 (98.20%) using the sequential technique. Additionally, Machine 2 achieves a significantly lower training time (58.55 seconds) than Machine 1 (77.65 seconds). This suggests that Machine 2 produces marginally better accuracy and trains the model much faster than Machine 1.

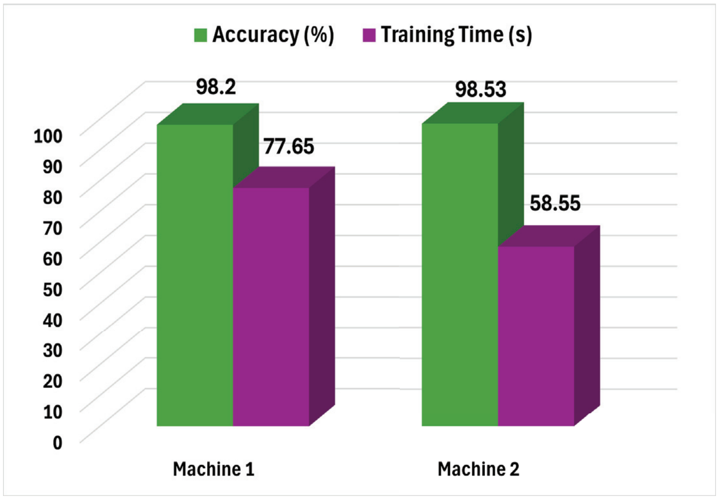

Figure 5: Accuracy and training time for sequential technique on Machine 1 and 2.

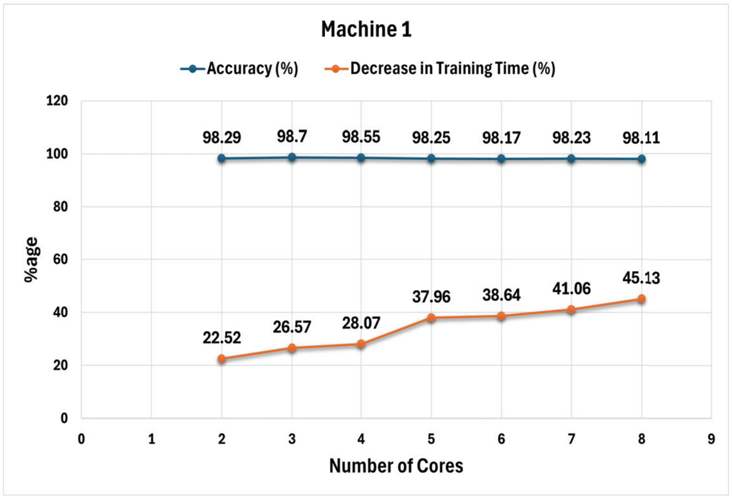

Figure 6: Decrease in training time (%age) of the proposed approach in comparison to the sequential technique and accuracy when training on MNIST Dataset against the number of Cores on Machine 1.

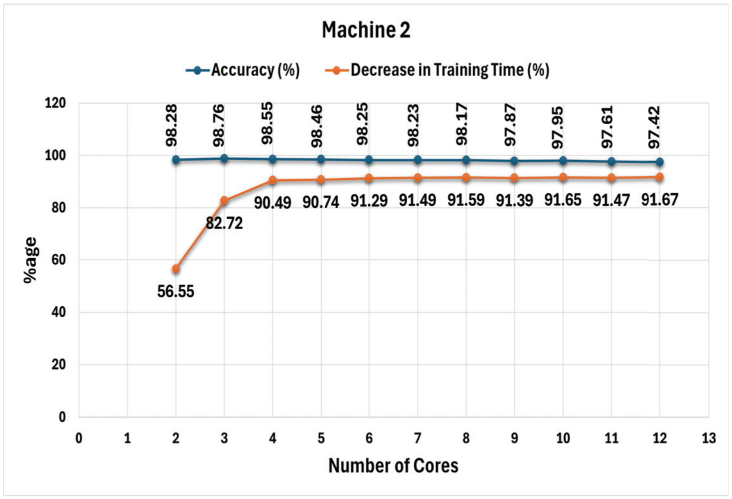

Figure 7: Decrease in training time (%age) of the proposed approach in comparison to the sequential technique and accuracy when training on MNIST Dataset against the number of Cores on Machine 2.

In Figure 6, the accuracy on Machine 1 shows a minimal variance between 98.11% and 98.29% when increasing the number of cores, revealing that increasing cores has a small impact on the accuracy of the model.

Training time decreases significantly as the number of cores increases: as shown in Figure 5, drops are 22.52% for two cores, 26.57% for three cores, 28.07% for four cores, 37.96% for five cores, 38.64% for six cores, 41.06% for seven cores, and 45.13% for eight cores. This shows that increasing the number of cores on Machine 1 substantially cuts down on training time; the decrease becomes more noticeable as the number of cores exceeds 5.

As a result, on Machine 1, training time significantly decreases when more cores are used, indicating that Machine 1 can efficiently take advantage of multi-core processing to speed up the training process, even though accuracy is unaffected by the number of cores.

In the graph shown in Figure 7 for Machine 2, the accuracy slightly fluctuates and overall remains stable across different numbers of cores, ranging from 97.42% to 98.98%, indicating that the number of cores does not significantly influence the model's accuracy. The training time shows drops of 17.69% with 2 cores, 56.55% with 3 cores, 82.72% with 4 cores, 90.49% with 5 cores, 90.74% with 6 cores, 91.29% with 7 cores, 91.49% with 8 cores, 91.39% with 10 cores, 91.65% with 11 cores, 91.47% with 12 cores as the number of cores increases. This shows a steep decrease initially, with a major drop at 4 cores and less significant improvements beyond 5 cores. When comparing Machine 2 to Machine 1, we observe that both machines maintain stable accuracy regardless of the number of cores. However, the decrease in training time shows a different pattern.

Both machines show that increasing the number of cores can reduce training time without compromising accuracy as much. Machine 1 shows a more linear improvement with additional cores, while Machine 2 shows a rapid improvement up to 4 cores with diminishing returns thereafter. These observations can help determine the optimal number of cores to use for efficient training on each machine, maximizing resource utilization without unnecessarily increasing hardware costs. This suggests that parallelization and ensemble learning did not compromise the quality of the trained models when using the proposed approach.

This notable boost in training efficiency shows the value of parallel computing paradigms, particularly in utilizing the computational capability of multi-core platforms. The computational resources are used more efficiently when the training effort is split over numerous processes using MPI, significantly reducing the total training time without sacrificing the model's predictive performance. Moreover, the trained model's robustness and reliability are further improved by incorporating ensemble learning approaches. By employing ensemble prediction strategies like majority voting, the model's accuracy is maintained while taking advantage of the various forecasting abilities of several separately trained models.

The results demonstrate the potential of ensemble learning and MPI-based parallelization as effective tactics for speeding up the training of deep learning models on sizable datasets. When training time is a significant barrier, these strategies provide valuable options for researchers and practitioners looking to maximize the computational efficiency of their machine-learning and deep-learning algorithms.

5 Conclusion

Our study highlights the effectiveness of ensemble learning and MPI-based parallelization in reducing CNN training time. We retained excellent accuracy while significantly reducing training time by spreading the training workload over multiple CPU cores. This method provides a feasible way to optimize deep learning workflows, especially when training time is crucial. The results demonstrate how ensemble learning and parallel computing can speed up deep learning model training on datasets. The proposed method has the potential to advance research and be applied in a variety of fields where fast model training is crucial for timely decision-making. The proposed approach is not limited to this use case and can be applied to various machine learning and deep learning models to reduce training time.

Software Availability:

The paper source code can be downloaded from: https://github.com/ZiaUrRehman-bit/CNN-Using-MPI

References

[1] Tongxue Zhou, Su Ruan, Stéphane Canu: 'A review: Deep learning for medical image segmentation using multi-modality fusion', *Array*, **3-4**, pp. 100004, 2019, ISSN: 2590-0056

[2] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436-444, May 2015.

[3] Xiaoke Jiang, Donghai Li, Hao Chen, Ye Zheng, Rui Zhao, Liwei Wu: 'Uni6D: A Unified CNN Framework Without Projection Breakdown for 6D Pose Estimation', in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2022, pp. 11174-11184.

[4] Ying Zhao: 'OmniAL: A Unified CNN Framework for Unsupervised Anomaly Localization', in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2023, pp. 3924-3933.

[5] Yongjian Deng, Hao Chen, Hai Liu, Youfu Li: 'A Voxel Graph CNN for Object Classification With Event Cameras', in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2022, pp. 1172-1181.

[6] Mariana Eugenia Ilas, Constantin Ilas: 'Towards real-time and real-life image classification and detection using CNN: a review of practical applications requirements, algorithms, hardware and current trends', in *2020 IEEE 26th International Symposium for Design and Technology in Electronic Packaging (SIITME)*, pp. 225-233, 2020,

[7] L. Deng, "The MNIST database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141-142, 2012.

[8] A. A. Elngar, M. Arafa, A. Fathy, B. Moustafa, O. Mahmoud, M. Shaban, and N. Fawzy, "Image classification based on CNN: a survey," *Journal of Cybersecurity and Information Management*, vol. 6, no. 1, pp. 18–50, 2021.

[9] Lisandro Dalcin, "MPI for Python," Release, 2019.

[10] Zina A. Aziz, Diler Naseradeen Abdulqader, Amira Bibo Sallow, and Herman Khalid Omer, "Python Parallel Processing and Multiprocessing: A Review," *Academic Journal of Nawroz University*, vol. 10, no. 3, pp. 345–354, Aug. 2021.

[11] Marcin Rogowski, Samar Aseeri, David Keyes, and Lisandro Dalcin, "mpi4py.futures: MPI-Based Asynchronous Task Execution for Python," *IEEE Transactions on Parallel and Distributed Systems*, vol. 34, no. 2, pp. 611-622, 2023.

[12] Y. E. Wang, C.-J. Wu, X. Wang, K. Hazelwood, and D. Brooks, "Exploiting Parallelism Opportunities with Deep Learning Frameworks," *ACM Trans. Archit. Code Optim.*, vol. 18, no. 1, articleno. 9, Mar. 2021, pp. 1-23.

[13] Xibin Dong, Zhiwen Yu, Wenming Cao, Yifan Shi, and Qianli Ma, "A survey on ensemble learning," *Frontiers of Computer Science*, vol. 14, no. 2, pp. 241-258, 2020.

[14] M.A. Ganaie, Minghui Hu, A.K. Malik, M. Tanveer, and P.N. Suganthan, "Ensemble deep learning: A review," *Engineering Applications of Artificial Intelligence*, vol. 115, pp. 105151, 2022.

[15] Meysam Vakili, Mohammad Ghamsari, Masoumeh Rezaei: 'Performance analysis and comparison of machine and deep learning algorithms for IoT data classification', in *arXiv preprint arXiv:2001.09636*