Parallelizing Plant Health: A Comparative Study of Parallel Deep Learning for Plant Disease Classification

Team 13 – He Zhou, Weiyu Wang

# 1 Introduction

## 1.1 Background

Plant diseases pose a continuous threat to agricultural productivity, causing economic losses of approximately $220 billion annually, as reported by the Food and Agriculture Organization of the United Nations [[[1]](#endnote-2)]. These losses disproportionately affect smallholder farmers in underdeveloped regions, sustaining food insecurity and economic hardships. Early disease detection is crucial, but it is often hindered by the shortage of expert opinions, resulting in delays and extensive crop damage.

In recent years, the integration of technologies, especially deep learning and computer vision, has revolutionized the field of plant disease detection. While conventional methods relied on hand-crafted features, deep learning can automatically extract discriminant features from images, rendering them highly efficient for this task.

## 1.2 Motivation & Goals

Deep learning has made remarkable strides in plant disease classification; however, the substantial challenge exists due to the extensive scale of plant datasets. These datasets, covering a wide variety of plant types and diseases, are often exceptionally large.

In response to this issue, the primary goal of our study is to accelerate the training phases of our model by leveraging the power of parallel techniques, facilitated by the centralized high-performance computing (HPC) Discovery cluster. The second goal is to assess and compare the performance of serial and parallel deep learning, analyzing the potential benefits of parallel approaches. As a result, we aim to provide agricultural stakeholders with timely and precise disease identification, ultimately minimizing crop losses and strengthening agricultural resilience.

# 2 Methodology

The following table outlines the key steps and corresponding tools, offering a structured overview of our project, as shown in Table 1.

**Table 1**

Project Roadmap

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Step | Roadmap | Focuses | Tools | Parallelization Comparison |
| 1 | **Exploratory Data Analysis** | Number of classes | NumPy, Matplotlib |  |
| 2 | **Data Preprocessing** | Mean, std calculation for data normalization | (1) Dask Delayed  (2) PyTorch | √ |
| 3 | **Model Definition** | Resnet18 | Torchvision |  |
| 4 | **Pre-training** | (1) Train with initial hyperparameters on a small number of epochs  (2) Time and Speedup comparison | PyTorch  (1) Serial  (2) multi-process (workers)  (3) multi-thread  (DataParallel) | √ |
| 5 | **Hyperparameter Tuning** | Bayesian optimization | Optuna  (multi-thread) | √ |
| 6 | **Final Training and Model Evaluation** | (1) Train with optimized hyperparameters on extended epochs  (2) assess with classification report and confusion matrix | PyTorch  Scikit-learn  Seaborn |  |

In our study, we employed a set of key methodologies, including:

**Exploratory data analysis:** Conduct a thorough examination of datasets featuring diverse plant diseases. Utilize the pandas feature in Python to classify and count images in specific folders for different diseases. Leverage matplotlib to visually categorize and chart plant diseases based on various plant species.

**Data preprocessing:** Given the augmented dataset, our focus lies in the normalization process. Employ mean and standard deviation to ensure consistent baseline values across the dataset. Implement Dask and PyTorch, two parallel computing methods, in the data standardization process. Document and compare the uptime and efficiency gains by varying the number of workers or processes.

**Model definition:** Utilize Convolutional Neural Networks (CNNs) with the ResNet-18 architecture. Its depth allows it to capture intricate features and patterns in images, which helps to address the vanishing gradient problem and facilitate deep network training.

**Model training:** Conduct model training using both serial and parallel approaches on the Discover cluster. To begin with, implement baseline serial training with PyTorch on a single GPU with one main process. Next, employ multi-process training by leveraging DataLoader’s multi workers. Lastly, explore parallel training with PyTorch DataParallel.

**Hyperparameter training:** Optimize further through hyperparameter tuning using Bayesian search with Optuna. Tune essential parameters including learning rate, step size, weight decay and momentum. Implement a multi-threading study, parallelizing the optimization procedure.

**Parallelization Performance Comparison:** Evaluate and compare serial and parallel approaches using elapsed time and GPU utilization. Calculate speedup and analyze the speedup curve. Visualize metrics using Matplotlib and seaborn for a comprehensive comparison.

**Model Evaluation:** Assess the trained model on the test dataset, presenting a comprehensive evaluation through a classification report and confusion matrix. Scikit-learn will be utilized for metric computation, and Seaborn will aid in visualization.

# 3 Description of Dataset

The New Plant Diseases Dataset (<https://www.kaggle.com/datasets/vipoooool/new-plant-diseases-dataset>) is a comprehensive collection of plant leaf images, inspired by the PlantVillage dataset and curated by Samir Bhattarai.

## 3.1 Dataset Introduction

The total size of this dataset amounts to 1.43 gigabytes. It comprises approximately 87.9k images in JPG format, encompassing both healthy and unhealthy leaves. The leaves were removed from the plant, placed against a gray or black background.

It has been partitioned into a ratio of 80% for training data and 20% for validation data. Additionally, there is a separate test set consisting of 33 images.

## 3.2 Dataset Splitting

The original dataset comprises 33 images designated for testing, representing only 0.37% of the entire dataset. To enhance the efficacy of our training, we have undertaken the creation of a new test subset by partitioning a segment from the original training set. This results in a distribution of 61,494 images for training, 17,572 for validation, and 8,814 for testing, maintaining a balanced ratio of 70% for training, 20% for validation, and 10% for testing, as illustrated in Table 2.

**Table 2**

Distribution of Images in Dataset Subsets After Splitting

|  |  |  |
| --- | --- | --- |
| Subset | Image counts | Percent |
| train | 61494 | 70% |
| valid | 17572 | 20% |
| test | 8814 | 10% |
| Total | **87880** | **100%** |

## 3.3 Exploratory Data Analysis

To initially acquire an understanding and verify the applicability of the dataset, we strategically selected and presented twelve images, each representing a unique plant disease, as shown in Figure 1.

**Figure 1**

Data Samples

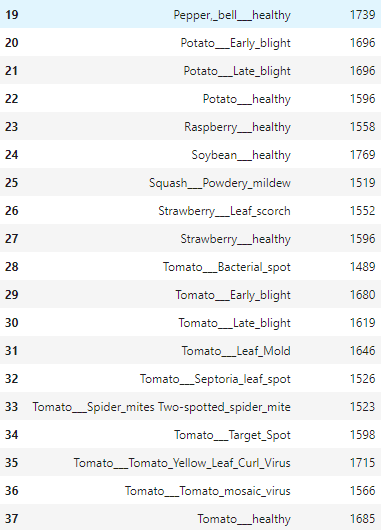
A collage of different leaves

Description automatically generated

The dataset is organized into 38 distinct categories based on plant species and specific diseases, as illustrated in Figure 2.

**Figure 2**

Distinct Categories of the Dataset

There are 14 distinct plant categories, including Apple, Blueberry, Cherry, Corn, Grape, Orange, Peach, Pepper, Potato, Raspberry, Soybean, Squash, Strawberry. We classify plant diseases according to different plant species, which helps to quickly identify patterns and potential anomalies in the data [[[2]](#endnote-3)], as depicted in Figure 3.

**Figure 3**

Plant Diseases grouped by Plant Categories

A screenshot of a graph

Description automatically generated

# 4 Results and Analysis

## 4.1 Data Preprocessing

The dataset comprises augmented and pre-cropped images, each with dimensions of 256x256 pixels. Our primary focus is on the normalization process, where the key objective is to standardize the range of pixel values across the dataset. This standardization is particularly crucial for deep learning models, ensuring that input features (pixel values) maintain a consistent scale. This practice promotes stable and efficient training, contributing to the robust performance of the models on the given data [[[3]](#endnote-4)].

For dataset normalization, we employed Dask and PyTorch, utilizing parallelization to calculate mean and standard deviation respectively. To assess the impact of parallelization, we conducted experiments with 1, 2, 4, and 8 workers.

### 4.1.1 Using Dask

#### (1) Mechanism

We leveraged Dask's process-based task scheduler and dask.delayed function to optimize our workflow involving large-scale data processing. The dask.delayed function defers the execution of tasks, allowing for lazy evaluation. The process-based task scheduler distributes computations across multiple processes, enhancing parallelism and overall efficiency [[[4]](#endnote-5)]. Subsequently, each deferred object, representing a computation task, is transformed into a Dask Array.

#### (2) Hardware

We allocated 4 CPU cores for 1, 2, 4, and 8 processes, as detailed in Table 3.

**Table 3**

*Hardware Information using Dask*

|  |  |
| --- | --- |
| Hardware | Details |
| CPU | Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz  Architecture: X86\_64  Count: 8 |

#### (3) Elapsed time

The elapsed time and speedup are showcased in Table 4, along with visual representations in Figure 4 and Figure 5.

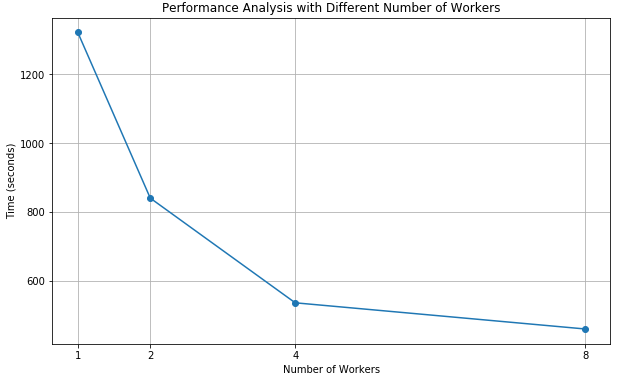
**Table 4**

Calculation Elapsed time and Speedup Comparison Using Dask

|  |  |  |
| --- | --- | --- |
| Number of Workers | Elapsed Time(s) | Speedup |
| 1 | 1321.3 | 1 |
| 2 | 840.0 | 1.57 |
| 4 | 526.0 | 2.47 |
| 8 | 459.9 | 2.87 |

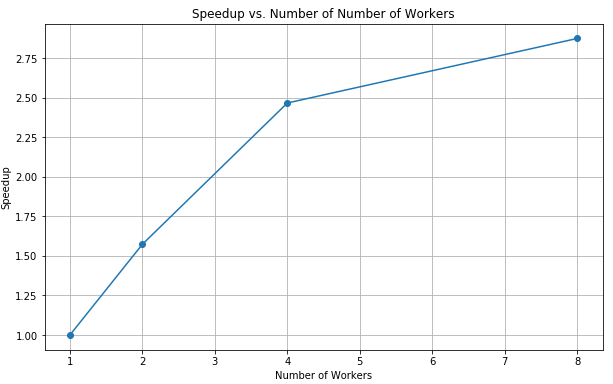
**Figure 4**

*Elapsed time Comparison Plot Using Dask*



**Figure 5**

Speedup Comparison Plot Using Dask



Analyzing the running schedule and line charts above, it's evident that with a single Worker, the process takes the longest time, recorded at 1321.3 seconds, serving as the baseline for parallelization. Doubling the number of workers to two significantly reduces the time to 840.0 seconds, marking a reduction of about 36.5%. Scaling up to four workers further reduces the time to 535.9 seconds, showcasing a reduction of about 59.5% compared to a single Worker and about 36.2% compared to two workers. With eight workers, the runtime decreases to 459.9 seconds, reflecting a 65.2% reduction compared to a single Worker and a 13.2% reduction compared to four workers. However, it's notable that the reduction in runtime diminishes as more workers are added.

### 4.1.2 Using PyTorch

#### (1) Mechanism

We chose PyTorch DataLoader to optimize data loading during model training, enhancing memory efficiency by creating batches and supporting parallel loading with multiple child processes.

PyTorch DataLoader and its num\_workers parameter enables multi-threaded data loading, allowing the code to process multiple batches of images in parallel. This accelerates the preparation of analysis or training data, taking advantage of PyTorch's batch processing capabilities and efficient tensor operations for quick and resource-efficient computation of mean and standard deviation.

#### (2) Hardware

We employ the same hardware configuration as when using Dask, as indicated in Table 3, which is previously referenced in the discussion on serial training.

#### (3) Elapsed time

The elapsed time and speedup are presented in Table 5, accompanied by visual representations in Figure 6 and Figure 7.

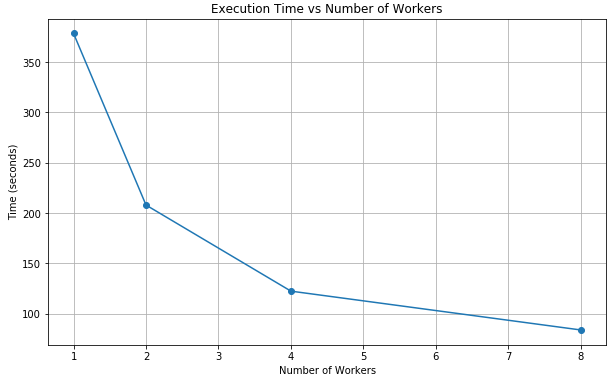
**Table 5**

Elapsed time and speedup Comparison Using PyTorch

|  |  |  |
| --- | --- | --- |
| Number of Workers | Time(s) | Speedup |
| 1 | 378.3 | 1 |
| 2 | 207.8 | 1.82 |
| 4 | 122.3 | 3.09 |
| 8 | 83.7 | 4.52 |

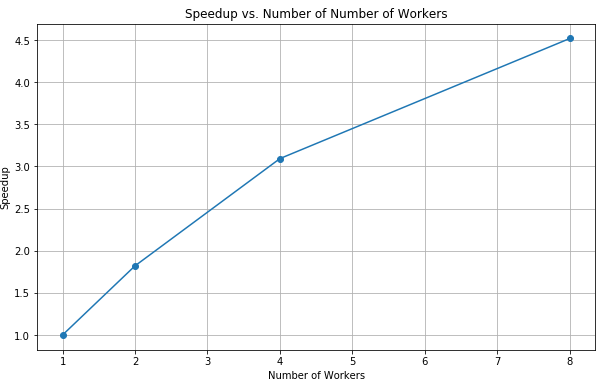
**Figure 6**

Elapsed time Comparison Plot Using PyTorch



**Figure 7**

Speedup Comparison Plot Using PyTorch



In the case of serial running (using a single worker), the elapsed time is 378.3 seconds. Introducing 2 workers reduces the time to 207.8 seconds, marking a 45% improvement over a single worker. While this reduction is significant, it falls slightly short of the desired 50%, likely due to the overhead of managing parallel tasks. With 4 workers, the time is further reduced to 122.3 seconds, indicating a 68% improvement compared to the single-worker scenario and showcasing higher utilization for parallel processing. Moving to 8 workers results in a runtime of 83.7 seconds, representing a 78% improvement over the single-worker scenario. However, diminishing returns become more pronounced, with the improvement not scaling proportionally to the number of workers.

### 4.1.3 Comparison Summary

The initial times with a single worker differ significantly between Dask and PyTorch (1321.3 vs. 378.3 seconds), suggesting variations in the tasks performed or the efficiency of the frameworks. Both frameworks exhibit diminishing returns as the number of workers increases, a common characteristic of parallel processing. However, the point at which returns diminish differs, possibly due to distinct overheads or task parallelizability. The disparities in time reduction and speedup hint at PyTorch's potential suitability for the specific task or better optimization for parallel processing in this context.

While both Dask and PyTorch effectively employ parallel processing to reduce computation time, PyTorch demonstrates a more substantial reduction in time as the number of workers increases. This analysis underscores that, despite both frameworks being capable of parallel processing, their performance can vary considerably based on the task nature and available computing resources.

## 4.2 Pre-training

In this context, "pre-training" refers to the initial training phase before hyperparameter tuning. During this stage, a small number of epochs are executed with an initial set of hyperparameters to obtain an initial evaluation and baseline performance. This baseline serves as a reference point for comparison with performance after hyperparameter tuning.

We assessed both serial and parallel training techniques to identify the approach that achieves optimal training elapsed time.

### 4.2.1 Serial Training

#### (1) Model Architecture

We utilized the ResNet18 implementation from Torchvision without pre-trained weights. The model architecture information depicted in Figure 8 is generated using the torchsummary package.

The architecture begins with a 7x7 convolutional layer followed by batch normalization and ReLU activation, incorporating max pooling for spatial dimension reduction. ResNet18 consists of four residual blocks, progressively increasing the number of filters. Global average pooling is applied after the residual blocks, leading to a 1x1 feature map. The architecture concludes with fully connected layers, including a softmax activation layer for classification [[[5]](#endnote-6)].

**Figure 8**

ResNet18 Model Architecture

A close-up of a number

Description automatically generated

#### (2) Hardware

We utilized 4 CPU cores and 1 T4 GPU for the serial training process. Table 6 provides detailed information about the hardware devices used.

**Table 6**

Hardware Information for Serial Training

|  |  |
| --- | --- |
| Hardware | Details |
| CPU | Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz  Architecture: X86\_64  Count: 4 |
| GPU | Tesla T4  Count: 1 |

#### (3) Elapsed Time

We trained the model for 5 epochs and examined the elapsed time (see [Appendix A](#_Appendix_A:_Training)).The elapsed times per epoch ranged from 427.79 to 445.89 seconds. Across all five epochs, the cumulative elapsed time totaled **2153.49** seconds. This serves as the reference serial time for comparative analysis.

### 4.2.2 Using Multi-process (workers)

#### (1) Mechanism

PyTorch DataLoader optimizes data loading by leveraging its num\_workers parameter, which orchestrates the use of multiple processes to efficiently load and process batch data from disk into memory. This parallel processing capability is particularly beneficial when dealing with large datasets. The designated number of worker processes, often corresponding to the number of available CPU cores, allows concurrent loading and preprocessing of batches, significantly improving data feeding speed [[[6]](#endnote-7)].

Furthermore, the efficiency extends to GPU utilization during the training process. While the data loading occurs in parallel, the main training process can simultaneously consume batches from the queue, minimizing idle time and enhancing overall training iteration speed. This concurrent processing mechanism ensures that both CPU and GPU resources are effectively utilized, leading to improved training efficiency and reduced waiting times during model training. In our experimentation with 5 epochs, we systematically varied the num\_workers parameter, evaluating its impact on time efficiency when set to values such as 0, 2, 4, and 8.

#### (2) Hardware

We utilized four CPUs and one GPU for the multi-processing training process, maintaining the same hardware configuration as in the serial training, as shown in Table 6, which is previously referenced in the discussion on serial training.

#### (3) Elapsed Time

We trained the model using 2, 4, and 8 processes (see [Appendix A](#_Appendix_A:_Training)).

It is noteworthy that when running on 8 processes, a warning was triggered: "This DataLoader will create 8 worker processes in total. Our suggested max number of workers in the current system is 4, which is smaller than what this DataLoader is going to create. Please be aware that excessive worker creation might cause DataLoader to run slowly or even freeze. It's advised to lower the worker number to avoid potential slowness or freeze if necessary."

This warning was issued due to potential over-subscription, indicating that the number of workers exceeds the number of CPU cores. Adjusting the worker number to align with system capabilities is advised by PyTorch to ensure optimal performance and prevent potential slowdowns or freezes.

The elapsed time and speedup are presented in Table 7 and Figure 9.

**Table 7**

Elapsed Time and Speedup Comparison Using Multi-process

|  |  |  |
| --- | --- | --- |
| Count of Processes | Elapsed time (s) | Speedup |
| Serial (1 process) | 2153.4935 | 1 |
| 2 processes | 1169.8231 | 1.8409 |
| 4 processes | 1114.2052 | 1.9328 |
| 8 processes | 1113.5024 | 1.9340 |

**Figure 9**

Elapsed Time and Speedup Comparison Plot Using Multi-process

A graph of a line and a line

Description automatically generated with medium confidence

By employing 2 processes, the elapsed time is substantially reduced to 1169.82 seconds, resulting in a commendable speedup of 1.84. Scaling up the parallelism to 4 processes further enhances efficiency, yielding a reduced elapsed time of 1114.21 seconds and an improved speedup of 1.93. However, increasing the number of processes to 8 under an oversubscription scenario shows minimal gains in speedup, maintaining a nearly consistent performance with an elapsed time of 1113.50 seconds and a corresponding speedup of 1.93.

### 4.2.3 Using Multi-thread (DataParallel)

#### (1) Mechanism

PyTorch's DataParallel is a container designed for a single process, multi-threaded execution on a single machine. This approach efficiently distributes both input data and the model across multiple GPUs, proving particularly advantageous when dealing with large datasets and complex models [[[7]](#endnote-8)].

Throughout the training process, the input data is partitioned across available GPUs, enabling the simultaneous computation of forward and backward passes on each GPU. Once the forward and backward passes conclude on each GPU, the results are aggregated to collectively update the model parameters.

The DataParallel module simplifies the parallelization process, automatically managing the distribution and aggregation of data and model parameters. This facilitates parallel computation across multiple GPUs, streamlining the training of deep learning models.

#### (2) Hardware

We utilized four CPUs and 2 T4 GPUs for the DataParallel training, as detailed in Table 8.

**Table 8**

Hardware Information for DataParallel

|  |  |
| --- | --- |
| Hardware | Details |
| CPU | Intel(R) Xeon(R) Gold 6240 CPU @ 2.60GHz  Architecture: X86\_64  Count: 4 (for 1 processes) |
| GPU | Tesla T4  Count: 2 |

#### (3) Elapsed Time

We recorded the elapsed time during the DataParallel training progress (see [Appendix A](#_Appendix_A:_Training)). The elapsed time and speedup are presented in Table 9 and Figure 10.

**Table 9**

Elapsed Time and Speedup Comparison using DataParallel

|  |  |  |
| --- | --- | --- |
| Count of GPU (T4) | Elapsed time (s) | Speedup |
| Serial (1 GPU) | 2153.4935 | 1 |
| 2 GPU | 1630.4431 | 1.3208 |

**Figure 10**

Elapsed Time and Speedup Comparison Plot using DataParallel

A white sheet of paper with a graph

Description automatically generated

Introducing parallel processing with two GPUs reduced the elapsed time to 1630.4431 seconds, yielding a speedup of 1.3208.

### 4.2.4 Comparison Summary

Table 10, along with Figures 11 and 12, encapsulates a comprehensive summary detailing the comparison of training elapsed time and speedup between serial and parallel techniques.

**Table 10**

Training Elapsed Time and Speedup Comparison Summary

|  |  |  |  |
| --- | --- | --- | --- |
| Training | Resources | Elapsed Time (s) | Speedup |
| Serial | Main process | 2153.4935 | 1 |
| Multi-process  (workers) | 2 processes | 1169.8231 | 1.8409 |
| 4 processes | 1114.2052 | 1.9328 |
| 8 processes | 1113.5024 | 1.9340 |
| Multi-thread  (DataParallel) | 2 GPUs | 1630.4431 | 1.3208 |

**Figure 11**

Training Elapsed Time Comparison Plot

A graph with a line and a number of processes

Description automatically generated

**Figure 12**

Training Speedup Comparison Plot

A graph with a line and a number of processes

Description automatically generated

In the sequential execution with a single main process, the computational task took 2153.4935 seconds to complete.

Transitioning to multi-processing with 2, 4, and 8 processes resulted in notable reductions in elapsed time, achieving speedups of 1.8409, 1.9328, and 1.9340, respectively. Introducing multi-threading with 2 GPUs in a DataParallel configuration led to a further decrease in elapsed time to 1630.4431 seconds, corresponding to a speedup of 1.3208.

Specifically focusing on the 2-process multi-processing scenario, the computational task completed in 1169.8231 seconds, resulting in a notable speedup of 1.8409 compared to the sequential execution. On the other hand, employing multi-threading with 2 GPUs in a DataParallel configuration led to an elapsed time of 1630.4431 seconds, corresponding to a speedup of 1.3208 compared to the single-process sequential execution. Notably, the 2-process multi-processing approach outperformed the 2-GPU DataParallel configuration in terms of speedup.

## 4.3 Hyperparameter Tuning

#### (1) Mechanism

Optuna, an open source hyperparameter optimization framework to automate hyperparameter search, incorporates multithreading to significantly enhance the efficiency of the optimization process. It leverages the Parallel class from the Joblib library to create a parallel execution context and utilizes delayed function to wrap the function that needs to be executed in parallel. This setup enables Optuna to conduct multiple trials simultaneously, each exploring different sets of hyperparameter configurations.

#### (2) Hardware

We employed a high-performance computing setup featuring four AMD EPYC 7543 32-Core Processors and a single A100 GPU, as outlined in Table 11.

**Table 11**

Hardware Information for Hyperparameter Tuning

|  |  |
| --- | --- |
| Hardware | Details |
| CPU | AMD EPYC 7543 32-Core Processor  Architecture: X86\_64  Count: 4 |
| GPU | A100-SXM4-80GB  Count: 1 |

#### (3) Elapsed Time

We tuned four critical hyperparameters—learning rate, step size, weight decay, and momentum—through four sets of experiments, each utilizing 1, 2, 4, and 8 threads, respectively.

Within each experiment, we conducted 8 tuning trials (see [Appendix B](#_Appendix_B:_Tuning)). Additionally, the comparison of elapsed time and speed is provided in Table 12 and visually represented in Figure 13.

**Table 12**

Tuning Elapsed Time and Speedup Comparison

|  |  |  |
| --- | --- | --- |
| Count of Threads | Total Elapsed time (s) | Speedup |
| Serial (1 thread) | 1633.4193 | 1 |
| 2 threads | 954.1967 | 1.7118 |
| 4 threads | 591.6575 | 2.7608 |
| 8 threads | 508.3633 | 3.2131 |

**Figure 13**

Tuning Elapsed Time and Speedup Comparison Plot

A graph of a line

Description automatically generated with medium confidence

In the serial execution with a single thread, the task required 1633.4193 seconds. As the thread count increased to 2, 4, and 8, the total elapsed times saw significant reductions to 954.1967 seconds, 591.6575 seconds, and 508.3633 seconds, respectively. Correspondingly, the speedup metrics indicated enhanced efficiency, measuring at 1.7118, 2.7608, and 3.2131 for 2, 4, and 8 threads, highlighting the substantial performance gains achieved through parallelization.

## 4.4 Final Training and Model Evaluation

In the final training phase, we used the optimized set of hyperparameters obtained through the tuning process to train our model over 10 epochs, as illustrated in Figure 14.

**Figure 14**

Training and Validation Loss and Accuracy

A graph of a graph

Description automatically generated with medium confidence

The best-achieved accuracy on the training set reached 68.43%, while the validation accuracy was 67.36%.

The classification report in Figure 15 and the confusion matrix in Figure 16 provide a comprehensive evaluation of the classification model's performance across various plant diseases and health states.

**Figure 15**

Classification Report

A screenshot of a computer screen

Description automatically generated

**Figure 16**

Confusion Matrix

A close-up of a graph

Description automatically generated

The model demonstrates a reasonable overall accuracy of 67%, underscoring its ability to classify plant diseases and health statuses, though individual class performance varies.

Notably, classes like "Corn\_(maize)\_\_Common\_rust" and "Orange\_\_\_Huanglongbing(Citrus\_greening)" exhibit high precision, recall, and F1-scores, with accuracies reaching 96% and 85%, respectively. These well-classified classes underscore the model's proficiency in accurately identifying instances of certain diseases.

However, some classes, such as "Tomato\_\_\_Late\_blight," "Tomato\_\_\_Septoria\_leaf\_spot," and "Tomato\_\_\_Early\_blight," show lower precision, recall, and F1-scores, indicating the need for further training or model refinement. For instance, "Tomato\_\_\_Late\_blight" has a precision of 49%, recall of 38%, and an F1-score of 43%, suggesting potential challenges in distinguishing this class.

# 5 Conclusion

The application of parallelization extends beyond the acceleration of training in a deep learning project; it encompasses a comprehensive enhancement of the entire process. Based our study we can conclude that:

In the realm of data preprocessing, when calculating mean and std using Dask and PyTorch, both frameworks effectively harness parallel processing to expedite computations and enhance overall efficiency. Compared to Dask, PyTorch exhibits a more substantial time reduction compared to Dask during data preprocessing tasks.

In terms of model training, utilizing multiprocessing yields speedup improvements, but diminishing returns occur when processes exceed CPU core count. Increased GPU count with DataParallel accelerates training times, though multiprocessing outperforms in terms of performance.

In the phase of hyperparameters tuning, Optuna demonstrates notable speedup improvements in hyperparameter tuning with an increasing number of threads.

Note that the conclusions are contingent on the specific hardware utilized in the study, and varying hardware configurations may yield different results (see [Appendix C](#_Appendix_C:_Hardware)). In summary, the study underscores the broad applicability of parallelization techniques across various stages of deep learning workflows.

# **References**

# Appendix A: Training Progresses

Serial Training Progress

A screenshot of a computer

Description automatically generated

Parallel Training Progresses Using Multi-process

|  |  |
| --- | --- |
| Count of  Processes | Training Progress |
| 2 | A screenshot of a computer program  Description automatically generated |
| 4 | A screenshot of a computer  Description automatically generated |
| 8 | A screenshot of a computer program  Description automatically generated |

Parallel Training Progress Using DataParallel

|  |  |
| --- | --- |
| Count of GPUs | Training progress |
| 2 | A screenshot of a computer  Description automatically generated |

# Appendix B: Tuning Progresses

Tuning Progresses

|  |  |
| --- | --- |
| Count of Threads | Tuning Progress |
| 1 | A number and numbers on a white background  Description automatically generated  A number with green and black numbers  Description automatically generated with medium confidence  A number with green and black numbers  Description automatically generated with medium confidence  A number with green and black lines  Description automatically generated with medium confidence  A number with a line  Description automatically generated with medium confidence  A number with green and black lines  Description automatically generated with medium confidence  A number with green and black numbers  Description automatically generated with medium confidence  A number with a dotted line  Description automatically generated with medium confidence  A black numbers and a white background  Description automatically generated |
| 2 | A white background with black and green numbers  Description automatically generated  A white background with black and green numbers  Description automatically generated  A screenshot of a computer code  Description automatically generated  A screenshot of a computer code  Description automatically generated  A screenshot of a computer code  Description automatically generated  A screenshot of a computer code  Description automatically generated  A screenshot of a computer code  Description automatically generated  A screenshot of a computer code  Description automatically generated |
| 4 | A screenshot of a computer program  Description automatically generated        A screenshot of a computer program  Description automatically generated |
| 8 | A screenshot of a computer program  Description automatically generated |

# Appendix C: Hardware Information

|  |  |  |
| --- | --- | --- |
| Experiments | CPU | GPU |
| Mean & STD Calculation | Intel(R) Xeon(R) CPU E5-2680 v4 @2.40GHz  Count: 8 | - |
| Serial Training  Multi-process Training | Intel(R) Xeon(R) Gold 6240 CPU @2.60GHz  Count: 4 | Tesla T4  Count: 1 |
| DataParallel Training | Intel(R) Xeon(R) Gold 6240 CPU @2.60GHz  Count: 4 | Tesla T4  Count: 2 |
| Hyperparameters Tuning | AMD EPYC 7543 32-Core Processor  Count: 4 | A100-SXM4-80GB  Count: 1 |

1. [] Gula, L. T. (2023, February 6). Researchers helping protect crops from pests. National Institute of Food and Agriculture. <https://www.nifa.usda.gov/about-nifa/blogs/researchers-helping-protect-crops-pests> [↑](#endnote-ref-2)
2. [] Ingle, A. (2020, December 15). Plant disease classification - resnet- 99.2%. Kaggle. <https://www.kaggle.com/code/atharvaingle/plant-disease-classification-resnet-99-2> [↑](#endnote-ref-3)
3. [] Brownlee, J. (2019, July 5). How to manually scale image pixel data for deep learning. MachineLearningMastery. <https://machinelearningmastery.com/how-to-manually-scale-image-pixel-data-for-deep-learning/> [↑](#endnote-ref-4)
4. [] Schmitt, M. (n.d.). Understanding dask architecture: Client, scheduler, workers. Datarevenue. [https://www.datarevenue.com/en-blog/understanding-dask-architecture-client-scheduler-workers](https://www.datarevenue.com/en-blog/understanding-dask-architecture-client-scheduler-workers#:~:text=The%20Dask%20Scheduler,who%20should%20do%20which%20tasks). [↑](#endnote-ref-5)
5. [] Modi, R. (2022, February 25). ResNet - understand and implement from scratch. Medium. <https://medium.com/analytics-vidhya/resnet-understand-and-implement-from-scratch-d0eb9725e0db> [↑](#endnote-ref-6)
6. [] Datasets & dataloaders. Datasets & DataLoaders - PyTorch Tutorials 2.1.1+cu121 documentation. (n.d.). <https://pytorch.org/tutorials/beginner/basics/data_tutorial.html> [↑](#endnote-ref-7)
7. [] Zhu, J. (n.d.). Getting started with distributed data parallel. Getting Started with Distributed Data Parallel - PyTorch Tutorials 2.2.0+cu121 documentation. <https://pytorch.org/tutorials/intermediate/ddp_tutorial.html> [↑](#endnote-ref-8)