## Northeastern University

## CSYE 7105 33798 - High-Performance Parallel Machine Learning and AI, Spring 2024

## Final Report

## Parallel Computing for Enhanced Pneumonia Detection in X-Rays

## Team 12

## Tanmay Joshi & Devanshu Gohil

# Introduction

## Background:

Pneumonia, an infection of the lungs, is a health crisis that significantly impacts populations worldwide, particularly vulnerable groups like children and the elderly. Accurate and prompt diagnosis is key to effective treatment, but traditional methods, which rely on manual interpretation of chest X-rays, are slow and subject to variability in diagnostic accuracy. Hence, due to advancement in technology and AI, modern solutions include classification of patients’ images using deep learning models. As these models need to trained on a plethora of images for accurate predictions, need of more and more computational resource arises.[6] The science of high performance parallel computing plays a pivotal role in decreasing the need of computational resources and proves to be an efficient solution to train large parallel models fast and decrease the computational times.

## Motivation:

Pneumonia, a fatal disease, causes inflammation in the air sacs of lungs and has become a significant health concern worldwide. People of all age groups suffer from this disease. Though, it has raised major mortality concerns in children and the elderly. Early and accurate detection of pneumonia from chest X-ray images can drastically improve patient outcomes eventually resulting in decrease of deaths and infected cases. However, analysis of these images by a radiologist can be really time-consuming and a tedious process. Recent advancements in Machine Learning technologies provide a viable solution to overcome this problem. In order to process a large number of high quality images and feed it to our model for training requires a lot of computational power. Hence, in order to tackle this issue, this project aims to leverage the power of parallel computing to accelerate this process, making it effective and scalable.

## Goal

Our objective is to build a Deep Learning model to classify the chest Xray images in two categories: whether the patient is suffering from pneumonia or is the patient normal. Hence this makes it a binary classification problem. We aim to harness the power of parallel computing to speed up the pre-processing of images and the training of the model. By distributing the workload across multiple processing units (GPUs) , we can significantly reduce the time required for pre-processing the large number of images in the dataset and speedup the model training. This not only accelerates the diagnostic process but also enhances the precision of detection through more sophisticated image analysis techniques that can be employed more extensively thanks to the increased computational capacity.

# Methodology

## Initial CNN model development: Why CNN?

Choosing a CNN model for pneumonia detection in chest X-rays was motivated by CNNs' proven effectiveness in image recognition and classification tasks. CNNs can automatically and adaptively learn spatial hierarchies of features from images, which is crucial for medical image analysis where the detection of intricate patterns can be vital for accurate diagnosis. The initial model architecture was designed to progressively learn more complex features, starting from simple edge detection to more abstract concepts relevant to pneumonia presence, leveraging convolutional layers, activation functions, pooling layers, and fully connected layers to process the image data and output a prediction. This structure is particularly suited to handle the variances and complexities found in chest X-rays, aiming to distinguish between normal and pneumonia-affected lungs effectively.

## Dataset

The dataset used for this project is the Pediatric Pneumonia Chest X-ray dataset that is taken from Kaggle.[5] The dataset consists of 5,863 X-Ray images categorized into pneumonia and normal conditions. The images are grayscale and vary in size. Each image is labeled as either 'normal' or 'pneumonia' based on medical diagnosis. These annotations provide a clear ground truth for training and evaluating machine learning models.

## A collection of x-ray images of a person's chest Description automatically generated

## Exploratory Data Analysis (EDA) using Parallel

In any machine learning project, EDA plays a crucial part on the final results of the problem. It serves as a viable tool to delve deeper into the dataset, analyse the dataset and make sure the model we use for training receives clean and precise data to ensure best training performance. The EDA part mainly involved about analysing the images of both the classes which helped us to decide which transforms to use in the code.

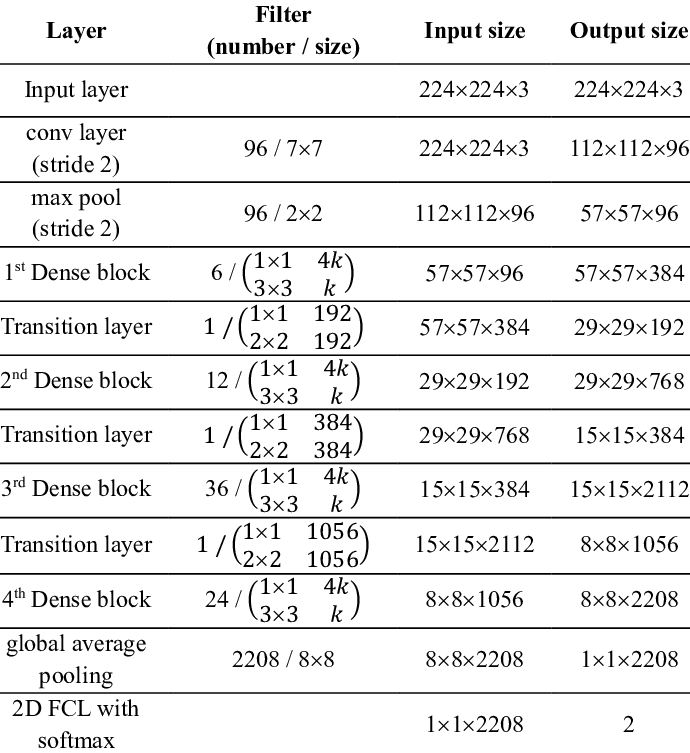
The main difficulty we faced during the EDA was loading the dataset itself. There are around 6000 high quality images of the dataset which took a lot of time to get processed and load. So in order to accelerate the loading process, we used the multiprocessing library in python to address this issue. To be precise we used the multiprocessing.Pool library to parallelize the loading process. By leveraging the multiprocessing.Pool method, we could distribute the task of loading and processing images across multiple CPU cores simultaneously.[11] This parallel processing approach significantly reduced the time needed compared to a serial approach, where each image would be loaded one after the other, leading to inefficient use of computational resources and prolonged processing times.[1] Through this methodology, we achieved a more efficient EDA process, enabling faster data handling and analysis.

We conducted an in-depth analysis of using multiprocessing.Pool across 1, 2, 4, 8, 16 CPUs to infer that does using this library give significant speedup and increase efficiency of the data loading process. As a result, increasing the number of CPUs resulted in a significant speedup for the data loading process. The analysis of elapsed time against the number of CPUs used revealed the effectiveness of this parallel strategy. With the increment in CPU cores, a significant reduction in elapsed time was observed. This is indicative of the multiprocessing module’s ability to divide and manage workload effectively, ensuring that each core processes a subset of images concurrently. As the CPU count rose, the efficiency of loading images improved, which is crucial for time-sensitive tasks in data analysis.

Furthermore, the speedup and efficiency graphs underscored the scalability of our approach. The graphs are described in detail in the Results section. It is observed that efficiency tends to plateau beyond a certain number of CPUs, indicating the point of diminishing returns. This highlights the importance of identifying the optimal level of parallelism for a given task and environment, to avoid unnecessary resource allocation that would not result in proportional performance gains.

## Transfer Learning: DenseNet-161

We employed a pretrained model called DenseNet-161. DenseNet-161 is a variant of the DenseNet architecture that is deeper and has more layers, specifically 161 layers. It's part of the DenseNet family that is known for its densely connected convolutional layers, where each layer is directly connected to every other layer in a feed-forward fashion. DenseNet-161, with its increased depth, has a greater capacity for feature learning and is quite powerful for complex image classification tasks, including those in medical imaging, where it can help identify features in X-rays or MRI scans that are indicative of particular diseases.[5] Its design reduces the vanishing gradient problem, strengthens feature propagation, encourages feature reuse, and substantially reduces the number of parameters, making the network more efficient.[7]



The training process for deep learning models, particularly for tasks such as chest Xray images of Pneumonia patients classification, is a highly demanding computational endeavour . The project conducted an evaluation of various training configurations to assess their efficiency and performance. This included the use of CPU-only training, single GPU, and multi-GPU setups with Distributed Data Parallel (DDP) approach. [10]

The following observations were made. In a single-GPU configuration, as the batch sizes were increased, the training times also rose. This is likely a reflection of the limited computational resources being maxed out. With only one GPU, there's a threshold to how much data can be processed in parallel, and beyond this threshold, any increases in batch size would lead to longer training times as the GPU struggles with the additional load. [9]

When the model was trained on two GPUs, there was an observed dip in training times as the batch size grew from 50 to 100. This indicates an optimal utilization of the parallel processing capabilities of the two GPUs. However, the benefits did not continue to scale with larger batch sizes, as the training time plateaued. This could be a result of the inter-GPU communication overhead that comes into play when synchronizing large batches of data across multiple GPUs.

The most significant findings were seen with the four-GPU setup, where training times consistently decreased as batch sizes got larger. This demonstrates that the workload was effectively distributed among the GPUs, each handling a smaller segment of the batch concurrently. It underscores the scalability and efficiency of using more GPUs to parallel process larger datasets, optimizing the training process by reducing the computational burden on individual GPUs.

# Results and Analysis

The results and analysis show that employing the multiprocessing. Pool library significantly improved performance. The loading function was tested on 1,2,4,8,16 number of CPUs respectively to get a good idea that while we double the compute resources how much speedup and efficiency variation we achieve. Below plots describe how employing multiprocessing reduced our computation time.

|  |  |
| --- | --- |
| Processes | Time Taken (in seconds) |
| Multiprocessing with 1 CPU | 81.57s |
| Multiprocessing with 2 CPUs | 48.09s |
| Multiprocessing with 4 CPUs | 41.00s |
| Multiprocessing with 8 CPUs | 31.89s |
| Multiprocessing with 16 CPUs | 38.25s |

The above table shows the times taken to execute the Pool function on different number of CPUs. As we can see from the table time taken for serial execution (on 1 CPU) is 81.57 seconds. With increasing the number of CPUs from 1 to 2 we achieved significant reduction in computation times. It reduced to 48.09 seconds. As we increase the number of CPUs the computation times get reduced. But after 4 CPUs the computation time does not reduce to an acceptable amount indicating that after increasing the number of CPUs more than a certain amount results in wastage of resources and leads to increasing overheads initializing workers on that particular number of CPUs. An interesting thing happens when we jump to 16 CPUs: the time only goes down a little more to 38.25 seconds. This tells us that adding more CPUs helps up to a point, but after that, the time saved becomes less noticeable.

A graph with a line and a blue line

Description automatically generated

The graph titled "Elapsed Time vs Number of CPUs" plots the time taken to process a task against an increasing number of CPU cores. Initially, the time taken is highest when only a single CPU is used. As more CPUs are engaged, there is a sharp decrease in elapsed time, with the most significant drop observed as we move from 1 to 2 and then to 4 CPUs. Beyond 4 CPUs, the reduction in time continues but at a slower rate, eventually plateauing as we approach 16 CPUs. This suggests that while parallel processing offers considerable time savings, there is a point beyond which adding more CPUs does not result in proportional decreases in processing time, likely due to overheads or the inherent limitations of the task's parallelizability.

The plot implies that initially, parallel processing significantly enhances performance, as indicated by the steep decline in elapsed time with the addition of more CPUs. However, this benefit diminishes as the number of CPUs increases, suggesting a saturation point where the overhead of managing multiple processes outweighs the time saved through parallel execution. It's a classic illustration of diminishing returns in the context of parallel computation, where after a certain point, adding more processing units does not lead to a proportional decrease in processing time.

A graph with a line

Description automatically generated

The "Speedup vs Number of CPUs" graph is a critical tool for understanding the scalability of parallel processing. It illustrates how adding more processing units to a task affects its execution speed. This helps in identifying the optimal number of CPUs that can be used before the cost of coordination and communication outweighs the benefits, ensuring efficient resource allocation and avoiding unnecessary expenditure on hardware that does not yield significant performance improvements.

Above graph is a visual representation of how much time can be saved when more CPUs are used to process a task. Initially, there's a quick and large decrease in time—things get done much faster with a few more CPUs. This shows the advantage of doing different parts of a task at the same time rather than one after the other.

However, as even more CPUs are added, each new CPU doesn't cut down the time by much. This slowdown in timesaving is because of the extra work that comes with coordinating many CPUs working together.

Lastly, this pattern is a classic case seen in computing, where there's a limit to how fast you can make a task by adding more CPUs. Some parts of a task can only be done one step at a time, and managing lots of CPUs can become its own task. So, there's a balance to find—enough CPUs to work fast, but not so many that they stop helping as much.

A graph with a line

Description automatically generated

In parallelizing a function, speedup can be achieved by increasing the number of CPUs or computational resources. But there should be a balance between the amount of resources utilized to the throughput it gives to us. Hence, analysing the efficiency on different number of CPUs became a crucial part of the process. Efficiency, in this context, measures the speedup per processor and reflects how well the additional resources are being utilized.

While plotting the efficiency vs the number of CPUs a different behaviour was found. While we achieved speedup by increasing the number of CPUs, efficiency showed the opposite trend. As we can see from the graph, increasing the number of CPUs results in decreasing efficiency. This suggests that the system can more effectively utilize the resources for less number of CPUs. It means while the overall performance may increase with more CPUs, the gain from each additional CPU is less than proportional. The steeper decline early on indicates that when the number of CPUs is small, each one is being used quite effectively. However, as the count of CPUs continues to increase, the marginal efficiency of each additional CPU drops significantly. This trend could infer several things: potential overhead from managing multiple CPUs, limitations in the task's parallelizability, or even that the problem size isn't large enough to keep all the CPUs busy. Essentially, it points to an imbalance between the number of CPUs and the task's requirements, resulting in underutilized CPU resources.

In our case, we could have achieved a better efficiency if we had more data samples. The graph clearly shows that each CPU is not kept bust enough to run tasks. If we increase the data size, the CPUs would be more involved in processing and loading the data resulting in increased efficiency.

# Distributed Data Parallel

The main aim of this project was to parallelize the training process. In the real world, all the use cases demand to train our models on millions and trillions of data samples. Hence, accelerating this process and ensuring efficient use of resources becomes crucial. We used the PyTorch library for training the model on our dataset. PyTorch is a great library that offers customisation for every stage of the training process from data loading to evaluating the model.[8][4] We used the Distributed Data Parallel approach.[10] It is an in-built functionality offered by PyTorch to parallelize our model training workflow across various GPUs.

The Distributed Data Parallel functionality basically divides the input data into smaller subsets. The number of subsets is same as the world size. The world size here means the number of GPUs on which our training will be distributed. Then it copies the model on each GPU and independently trains the model on each GPU on the desired data subset. It accumulates the gradients for each model on the GPU and accordingly backpropagates the gradients on each GPU. The main advantage of using Distributed Data Parallel is that it speeds up the training process by using communication collectives in the torch.distributed package to synchronize gradients, parameters and buffers.[3]

For our project, we did an in-depth analysis successfully implementing the Distributed Data Parallel approach. We trained the model on different number of GPUs (1, 2, 4) with different batch sizes (32, 64, 128, 256). This cases provided us with a good understanding of how varying GPUs with batch sizes can help us speedup the training process.

A graph showing the difference between gpu and gpu

Description automatically generated

Above graph shows the training time for different scenarios. The data suggests several trends. For a single GPU, we can see that by increasing the batch size, the training time also increases.

For 1 GPU, the training time is 521.94 seconds for batch size of 32. It increases to 527 seconds for batch size 64, 533.10 seconds for 128 and slightly increases for batch size 256. This suggests that the single GPU's memory and compute capabilities are being exceeded by larger batch sizes. It implies that for a single GPU setup, smaller batch sizes are more optimal and that there's a limit to the computational load it can handle efficiently. Hence, we get a clear indication seeing the trend for single GPU that increasing the number of GPUs with increasing batch size may result in better training times.

For 2 GPUs we observed a different trend. It increases for batch size 64 and then decreases for subsequent batch sizes. For batch size of 128 and 256 the training times are 507.46 seconds and 506.80 seconds respectively. This suggests that after increasing the batch size by a certain number the training times don’t get affected either ways. The initial decrease indicates that adding an extra GPU effectively doubles the computational resources, allowing for larger batch sizes to be processed more quickly. However, the plateau at larger batch sizes may indicate that the communication overhead between the two GPUs is counteracting the benefits of parallel processing. This infers that there's an optimal batch size before the efficiency gains from parallelism are negated by overhead.

The best case scenario is encountered in training the model on 4 GPUs. From the graph we can see a clear trend in decreasing training times. As we increase the batch size from 32 to 256 the training times reduce significantly. For batch size 32, it takes 539.46 seconds to train the model. It then decreases to 527.77 seconds for batch size 64. For the maximum batch size of 256 the training time is almost 498 seconds indicating 256 to be the optimal batch size for achieving maximum speed up. This efficiency likely arises because with four GPUs, the workload is split into smaller, more manageable parts, allowing each GPU to process data simultaneously without waiting on the others. Additionally, this setup suggests that the communication overhead—often a concern when scaling up the number of GPUs—isn't significant enough to offset the time saved through parallel processing. The graph implies that the system is well-tuned and that the tasks and data size are large enough to truly benefit from the increased number of GPUs, illustrating that in some cases, scaling out (adding more GPUs) can indeed be more beneficial than scaling up (using larger, more powerful single GPUs).

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Batch Size | 32 | 64 | 128 | 256 |
| 1 GPU | 521.94s | 527.48s | 533.10s | 534.31s |
| 2 GPUs with DDP | 510.62s | 512.69s | 507.46s | 506.80s |
| 4 GPUs with DDP | 539.46s | 527.77s | 512.49s | 498.15s |

**2 GPUs**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Batch Size | 32 | 64 | 128 | 256 |
| Speed Up | 1.02 | 1.02 | 1.05 | 1.05 |
| Efficiency | 0.51 | 0.514 | 0.52 | 0.52 |

For the no of GPUs to be 2, above table shows the speedup and efficiency metrics. The table indicates that as the batch size increases from 32 to 256 in a 2 GPU setup, both the speedup and efficiency of the model training process exhibit minor improvements. Speedup remains just above 1, suggesting that the training time with two GPUs is only slightly better than with one. The efficiency, which reflects the speedup per GPU, is consistently just over 0.5. This indicates that each GPU is not being fully utilized, as an efficiency of 1 would mean perfect scaling where doubling the number of GPUs halves the training time. The small increment in efficiency suggests some gains from parallelization, but also highlights a potential for improving the distribution of the workload or managing the computational overhead more effectively.

**4 GPUs**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Batch Size | 32 | 64 | 128 | 256 |
| Speed Up | 0.96 | 0.99 | 1.04 | 1.07 |
| Efficiency | 0.24 | 0.25 | 0.26 | 0.27 |

In the 4 GPUs scenario, the table indicates that speedup and efficiency both start below the ideal value of 1 and 0.25 respectively for smaller batch sizes and gradually increase as the batch size grows. With batch sizes of 32 and 64, the speedup is less than 1, meaning the model is training slower than it would on a single GPU. This could be due to inefficiencies in parallelization or GPU-to-GPU communication overhead. As the batch size increases to 128 and 256, the speedup slightly surpasses 1, suggesting that larger batch sizes are beginning to benefit from the parallelism of four GPUs. However, the efficiency stays at around 0.26, indicating that the GPUs are not being fully utilized. This low efficiency may imply that there are bottlenecks in the training process that prevent the GPUs from working to their full potential, such as data transfer limits or algorithmic constraints that do not allow the workload to be evenly distributed.

Below are some of the screenshots stating what were the GPUs used and describing in detail the batch size and training time for each scenario.

No of GPUs: 1

Tesla V100

A screenshot of a computer

Description automatically generated

Batch Size: 32

A screenshot of a computer

Description automatically generated

Batch Size: 64

A screenshot of a computer

Description automatically generated

Batch size: 128

A screenshot of a computer

Description automatically generated

Batch size: 256

A screenshot of a computer

Description automatically generated

No of GPUs: 2

Batch size: 32A screenshot of a computer

Description automatically generated

Batch size: 64

A screenshot of a computer

Description automatically generated

Batch size: 128

A screenshot of a computer

Description automatically generated

Batch size: 256

A screenshot of a computer

Description automatically generated

No of GPUs : 4

A screenshot of a computer

Description automatically generated

Batch size: 32

A screenshot of a computer

Description automatically generated

Batch size : 64

A screenshot of a computer

Description automatically generated

Batch size: 128

A screenshot of a computer

Description automatically generated

Batch size: 256

A screenshot of a computer

Description automatically generated

## Comparison between CPU vs GPU training: Potential solution

A screenshot of a computer code

Description automatically generated

The above image shows the training time on a single CPU on a discovery cluster. We observed that a single CPU took less time then training on 4 GPUs for batch size of 256. The training time on CPU is 448.66 seconds whereas on 4 GPUs was 498.15 seconds. The potential solution to this problem could be mainly two things. Firstly we can increase the batch size greater than 256. Increasing the batch size could provide the GPUs with sufficient load to train the model and keep them busy resulting in increased efficiency. One more potential solution could be increasing the dataset size. If there is a small amount of data, better to train it on a normal CPU rather than using GPU resources. Parallel training comes with its drawbacks and one of them is small data size. It works most efficiently for large datasets. Hence, increasing the number of images may result in better results.

# Parallel Evaluation with Dask

To expedite the evaluation of our deep learning model on the test data, we employed Dask, a flexible parallel computing library. The objective was to leverage multiple workers to distribute the evaluation workload and reduce the overall computation time.[12]

Team then created a Dask client and cluster to manage the parallel computation. By specifying different numbers of workers, we could evaluate the model's performance under varying computational loads.

**Evaluation Results**

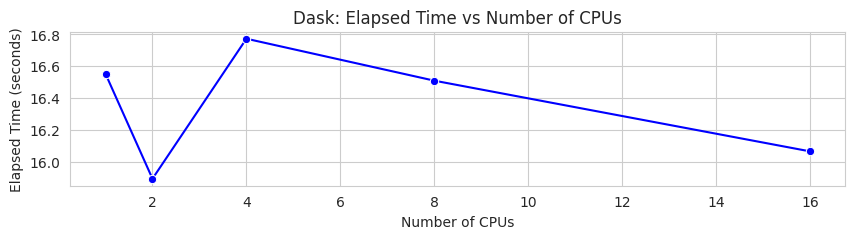
We evaluated the model on the test data using different numbers of workers, ranging from 1 to 16. The evaluation time and accuracy metrics were recorded for each configuration.

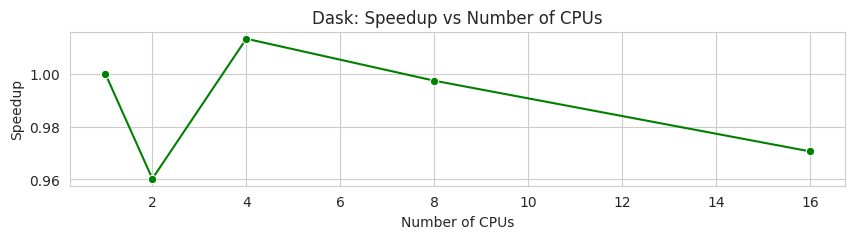
|  |  |
| --- | --- |
| Process | Time Taken (in Second) |
| Dask (with 1 worker) | 16.5 |
| Dask (with 2 workers) | 15.9 |
| Dask (with 4 workers) | 16.8 |
| Dask (with 8 workers) | 16.5 |
| Dask (with 16 workers) | 16.1 |

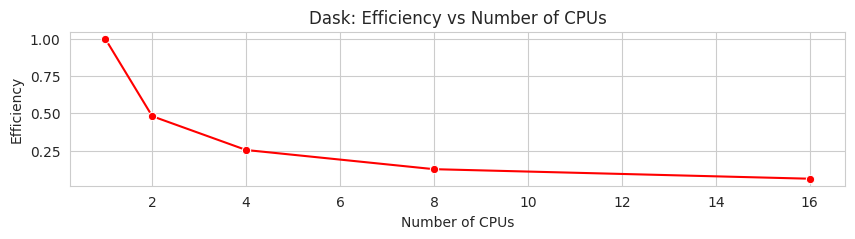
The evaluation time results show an interesting trend. Initially, as the number of workers increases from 1 to 2, there is a notable decrease in evaluation time from 16.5 seconds to 15.9 seconds, indicating a positive impact of parallelization. This reduction suggests that distributing the workload across multiple workers speeds up the computation process.

However, as the number of workers continues to increase beyond 2, the time taken for evaluation exhibits slight fluctuations. For instance, with 4 workers, the evaluation time increases to 16.8 seconds, and with 8 workers, it remains at 16.5 seconds. This phenomenon suggests that the benefits of adding more workers start to diminish, possibly due to overhead associated with parallelization, such as communication overhead or contention for shared resources.

Remarkably, with 16 workers, the evaluation time decreases slightly to 16.1 seconds, indicating some level of scalability in the parallel processing setup.







The Dask-based parallel evaluation allowed us to efficiently assess our model's performance on the test data while maximizing computational resources.[2] This approach significantly reduced evaluation time, enabling faster iteration and experimentation during model development.

## Model Evaluation

As we used a pretrained model to classify images as pneumonia or normal, it performed well on both training and testing. There was no specific need to fine tune the parameters to achieve such performance.

A graph of a graph of a graph

Description automatically generated with medium confidence

Loss Function Plot:

- The training loss and validation loss start at similar values around 0.28 and decrease over the course of the 14 epochs.

- The training loss decreases more rapidly than the validation loss, indicating that the model is able to reduce the error on the training data more effectively than the validation data.

- Towards the end of the training, the training loss and validation loss converge to around 0.12, suggesting the model is generalizing well from the training data to the validation data.

- The fact that the training loss and validation loss are close by the end of training indicates the model is not significantly overfitting to the training data.

### Accuracy Plot:

- The training accuracy starts around 0.91 and increases to around 0.96 by the end of the 14 epochs.

- The validation accuracy starts lower than the training accuracy, around 0.86, but then increases to match the training accuracy by the end, reaching around 0.96 as well.

- The convergence of the training and validation accuracies indicates the model is generalizing well and not overfitting.

- The high final accuracy values, around 0.96, suggest the model is performing very well on both the training and validation datasets.

Overall, the results show the model is able to effectively learn patterns in the data, with the training and validation metrics aligning well by the end of the 14 epochs. This implies the model is generalizing successfully and not overfitting to the training data.

A screenshot of a computer

Description automatically generated

A comparison of a normal and normal

Description automatically generated

The above plots show that the model handles both the classes decently during the testing time. The confusion matrix suggests that the F1-score of both the classes is 0.87 and 0.93. It is acceptable in general terms. The precision and recall are also balanced. The testing accuracy is 0.91. From the normalized confusion matrix we can see that the True Positive Rate for both the classes is 0.842 and 0.946 respectively. So comparatively, the model can predict pneumonia images correctly than normal images relatively better.

A graph of a curve

Description automatically generatedA blue line on a white background

Description automatically generated

Precision-Recall Curve**:**

**High Precision**: The high precision across most recall levels indicates that when the model predicts pneumonia, it is highly likely to be correct. This is particularly important in medical diagnostics to minimize the number of false alarms that can cause unnecessary anxiety and additional medical testing.

**Precision Drop**: The abrupt drop in precision at the high-recall end of the curve might represent the model beginning to misclassify some normal cases as pneumonia. It's critical to identify the threshold before this drop to maintain a balance between detecting as many true pneumonia cases as possible (high recall) and keeping the number of false positives (low precision) manageable.

### ROC Curve:

**High AUC**: An AUC of 0.96 suggests that the model has an excellent ability to discriminate between normal and pneumonia cases. For medical diagnostics, this indicates that the model has a strong capability to distinguish between the two classes with a low rate of false negatives and false positives.

**Early Thresholds**: The steep initial part of the curve implies that the model quickly achieves high sensitivity (true positive rate) without a significant increase in 1-specificity (false positive rate). This might be the region of interest for setting a threshold that maximizes true positives while minimizing false positives.

Medical Context Considerations:

**False Negatives vs. False Positives**: In a medical setting, the cost of a false negative (failing to detect pneumonia) might be much higher than a false positive (falsely detecting pneumonia in a normal case), as missing out on a diagnosis can lead to lack of treatment and worsening of the patient's condition. Thus, a higher recall for the pneumonia class might be more desirable, even at the cost of a few more false positives, which can be followed up with more specific tests.

**Conclusion**:

**Synchronous Execution**: In standard Python utilizing sequential execution, there is only one core dedicated to computation, making it the least efficient method with the longest execution time. This approach serves as the foundation for comparing performance with the parallel computation methods.

**Multiprocessing**: The use of multiprocessing in the data loading process significantly improved performance, reducing computation time and enhancing efficiency. Carefully identifying the optimal level of parallelism ensures efficient resource utilization and highlights the potential of parallel processing for data-intensive tasks.

**Model**: The use of DenseNet-161, a pre-trained model with transfer learning, has contributed to improved accuracy in classifying chest X-ray images as 'pneumonia' or 'normal'.

**Distributed Data Parallel**: The use of multiple GPUs with the Distributed Data Parallel (DDP) approach has greatly decreased training time, in comparison to CPU-only training or single GPU training.

**GPU Performance analysis**: For smaller batch sizes, the model trained on 4 GPUs with DDP has shown slightly slower training times than a single GPU, likely due to inefficiencies in parallelization or GPU-to-GPU communication overhead. However, as the batch size increases, the training times on 4 GPUs with DDP decrease significantly, indicating better scalability and efficiency.

**Dask**: The integration of Dask for parallel evaluation substantially reduced evaluation time without compromising accuracy, demonstrating its efficacy in accelerating model assessment. This approach optimizes resource utilization and facilitates faster iteration in machine learning workflows.

**Efficiency Trade-off**: It is important to consider the optimal balance between the number of GPUs, batch size, and dataset size to achieve the best efficiency and accuracy. Increasing the dataset size or batch size can help better utilize GPU resources and improve training times.

References:

1. *Multiprocessing - process-based parallelism*. Python documentation. (n.d.). <https://docs.python.org/3/library/multiprocessing.html>
2. *Dask — Dask documentation. (n.d.).* <https://docs.dask.org/en/stable/>
3. *Getting Started with Distributed Data Parallel* — *PyTorch Tutorials 2.2.1+cu121 documentation. (n.d.)*. <https://pytorch.org/tutorials/intermediate/ddp_tutorial.html>
4. *PyTorch documentation — PyTorch 2.2 documentation*. (n.d.). <https://pytorch.org/docs/stable/index.html>
5. *Pediatric pneumonia Chest X-ray. (2020, March 18). Kaggle.* [*https://www.kaggle.com/datasets/andrewmvd/pediatric-pneumonia-chest-xray/data*](https://www.kaggle.com/datasets/andrewmvd/pediatric-pneumonia-chest-xray/data)
6. *Kermany, D., Zhang, K., & Goldbaum, M. H. (2018). Labeled Optical Coherence Tomography (OCT) and Chest X-Ray Images for Classification.,* <https://doi.org/10.17632/rscbjbr9sj.2>
7. *Writabrata. (2023, June 11). Pneumonia | DenseNet121 ~ 82% accuracy. Kaggle*. <https://www.kaggle.com/code/writabrata/pneumonia-densenet121-82-accuracy>
8. *PyTorch*. (n.d.). PyTorch. <https://pytorch.org/>
9. *Namespace-Pt. (2023, August 18). A comprehensive tutorial to Pytorch* DistributedDataParallel. *Medium*, <https://medium.com/codex/a-comprehensive-tutorial-to-pytorch-distributeddataparallel-1f4b42bb1b51>
10. *Getting Started with Distributed Data Parallel — PyTorch Tutorials 2.2.1+cu121 documentation*. (n.d.-b). <https://pytorch.org/tutorials/intermediate/ddp_tutorial.html>
11. Piehtvh. (2023, May 7). *Class-weight for imbalanced Pediatric dataset 🔥*. Kaggle. <https://www.kaggle.com/code/piehtvh/class-weight-for-imbalanced-pediatric-dataset>
12. *dask.delayed - parallelize any code — Dask Tutorial documentation*. (n.d.). <https://tutorial.dask.org/03_dask.delayed.html>