PARALLELIZATION TECHNIQUES IN DEEP LEARNING FOR WEATHER CLASSIFICATION OF IMAGES

Instructor: Prof Handan Liu

Course: CSYE 7105, Parallel Machine Learning and AI

Group number: 9

Student1: Dushyant Mahajan

Student2: Soeb Hussain

Index

Background

Weather image classification presents a unique set of challenges, necessitating the differentiation of various weather phenomena ranging from clear skies with a beautiful Rainbow to the tumultuousness of thunderstorms. The implementation of deep learning models has demonstrated significant potential in effectively handling this task. However, these models often demand substantial computational resources for their training and in preprocessing.

Motivation

The motivation behind enhancing weather image classification models is driven by the increasing demand for both accuracy and efficiency in various pivotal applications

- **Enhanced Weather Forecasting**
- **Climate Change Analysis**
- **Advancements in Computer Vision**

Incorporating parallelization into these models is essential, as it addresses the computational challenges posed by deep learning techniques, enabling faster processing and more scalable solutions for real-time applications.

Goals

The aim of our project is to engineer a deep learning model that operates in parallel to classify weather images. The objectives include:

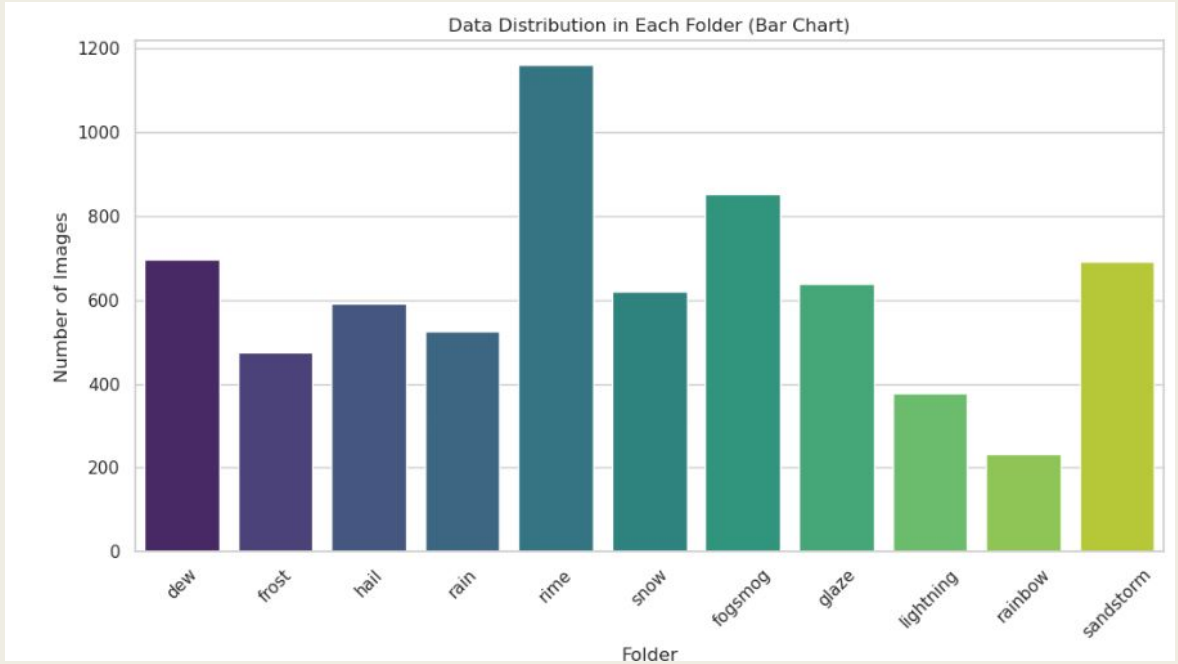
- **Model Design:** Craft a deep learning architecture specifically tailored for the intricate task of weather image classification.
- **Parallelization Implementation:** Apply various parallel computing techniques to bolster the model's training process, thereby increasing speed.
- **Performance Analysis:** Conduct a thorough evaluation and interpret the efficacy of each parallelization strategy across different stages of the model's workflow.

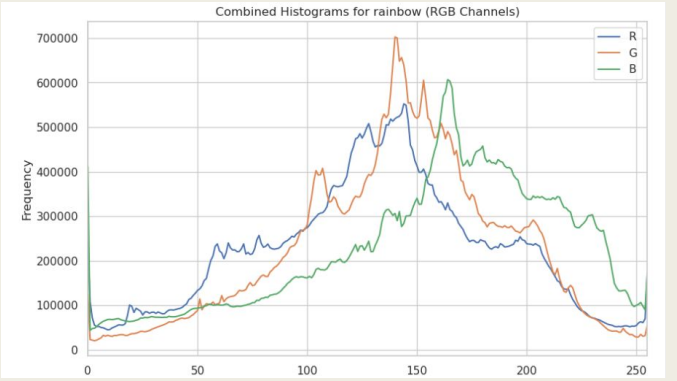
Dataset

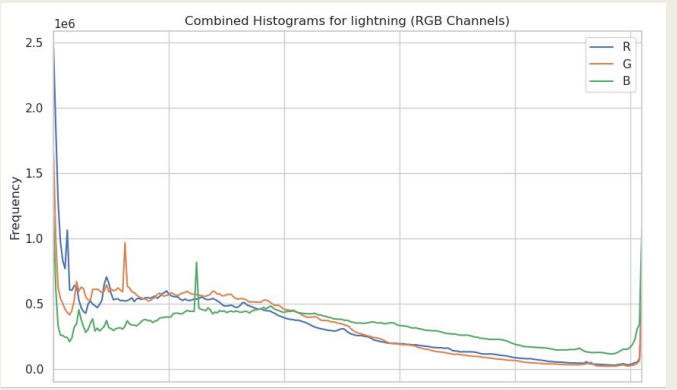
The dataset described in the text is the Weather Phenomenon Database (WEAPD), a collection specifically designed for weather phenomena classification using deep learning techniques. Key characteristics of this dataset include:

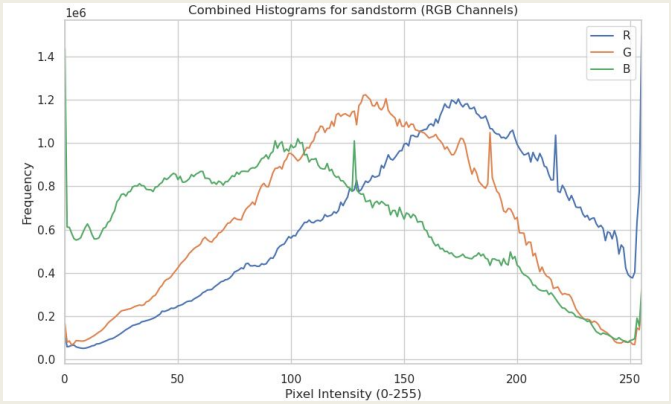
1. **Size and Composition**: WEAPD contains 6,877 images, encompassing 11 different weather phenomena. These phenomena are hail, rainbow, snow, rain, lightning, dew, sandstorm, frost, fog/smog, rime, and glaze.
2. **Purpose and Application**: It's established to facilitate the development and testing of a deep convolutional neural network (CNN) model for classifying weather phenomena

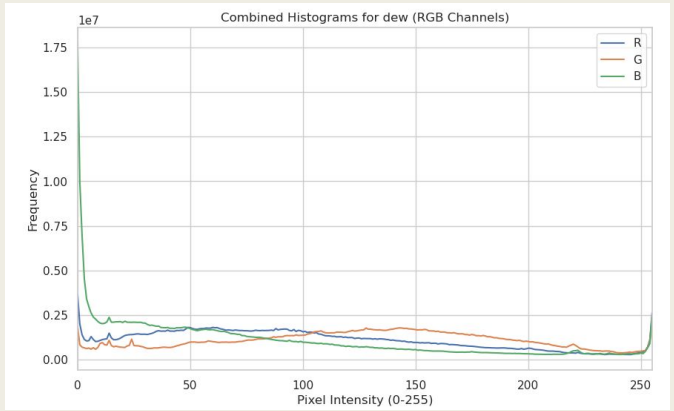
EDA

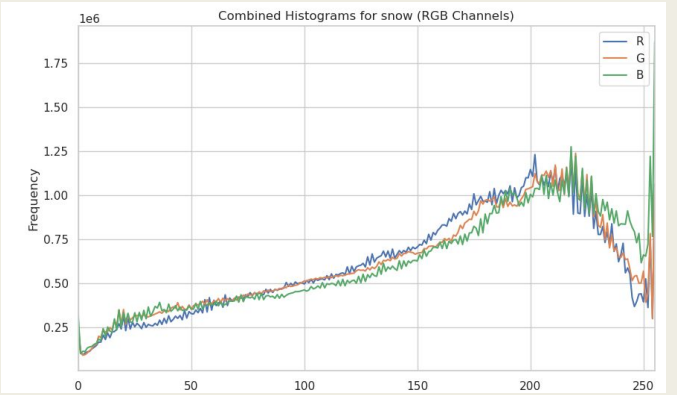

Data Distribution in Each Folder (Bar Chart)
Number of Images
Folder
dew
frost
hail
rain
rime
snow
fogsmog
glaze
lightning
rainbow
sandstorm


Combined Histograms for rainbow (RGB Channels)
Frequency
R
G
B


Combined Histograms for lightning (RGB Channels)
1e6
Frequency
R
G
B


Combined Histograms for sandstorm (RGB Channels)
1e6
Frequency
Pixel Intensity (0-255)
R
G
B


Combined Histograms for dew (RGB Channels)
1e7
Frequency
Pixel Intensity (0-255)
R
G
B


Combined Histograms for snow (RGB Channels)
1e6
Frequency
R
G
B

Methodologies

Various techniques learned in the class were applied to different parts of training.

- **Sequential**
- **Multiprocessing**
- **Dask**
- **GPU**
  - ***Data Parallel***
  - ***Distributed data Parallel***

Sequential Running

Sequential execution in Python is the default mode of operation where instructions are processed one after the other in the order they appear in the code. This means that at any given moment, only one operation is being carried out. For example, if you have a loop that processes images to calculate histograms, each image will be processed one at a time.

- **Global Interpreter Lock (GIL)**: The Global Interpreter Lock, or GIL, is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes at once. This lock is necessary mainly because CPython's memory management is not thread-safe.
- The GIL has a significant impact on multi-threaded Python programs. It allows only one thread to execute in the interpreter at any one time. This means that, even on a multi-core processor, a multi-threaded Python program will usually not run threads in true parallel

Implementation

1. **Process Each Folder**
2. **Process Each Image**
3. **Calculate Histogram f Each Color Channel**
4. **Combine Histograms**

```python
# Process each folder
for folder in (folders):
    folder_path = os.path.join(dataset_path, folder)
    images = os.listdir(folder_path)

    # Process each image
    for image_name in images:
        image_path = os.path.join(folder_path, image_name)
        with Image.open(image_path) as img:
            # Convert image to RGB if not already
            img_rgb = img.convert('RGB')

            # Calculate histogram for each color channel
            for channel, color in zip(range(3), ['R', 'G', 'B']):
                hist = img_rgb.getchannel(channel).histogram()
                histograms[folder][color].append(hist)

# Combine histograms for each color channel in each folder
for folder in histograms.keys():
    for color in ['R', 'G', 'B']:
        histograms[folder][color] = [sum(x) for x in zip(*histograms[folder][color])]
```

Multi Processing

The multiprocessing library in Python is used to create a pool of worker processes.

- The Pool object is created with a specific number of processes (as cpu_counts).
- The map method of the Pool object is used to distribute the work of processing each folder in the dataset across the available processes.
- This method is particularly effective for CPU-bound tasks and can significantly reduce the time required for computation by utilizing multiple CPU cores simultaneously.

Implementation

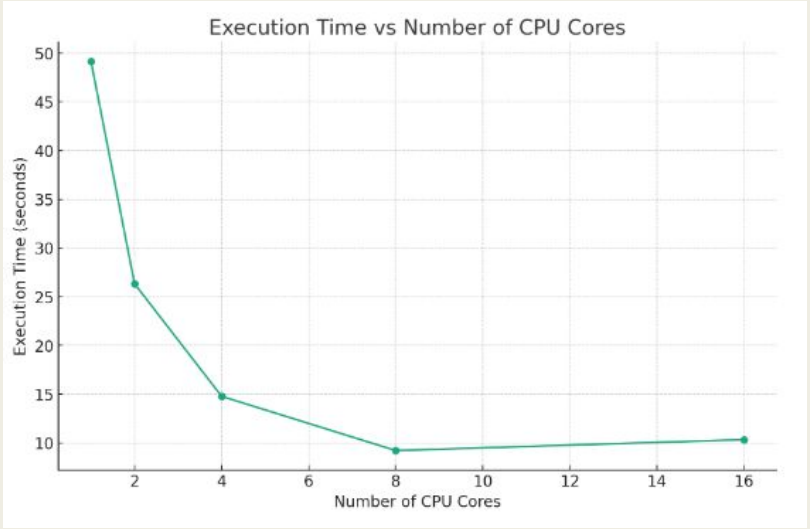
- Initialization process_folder, that processes all images in a given folder to compute and combine their color histograms.
- A multiprocessing Pool is created with the number of processes set to the current CPU count (cpu_count
- The pool.map() function is used to apply the process_folder function to each folder in the folders list.
- This function maps the process_folder function onto the list of folders, distributing the folders across the available processes in the pool.
- Each process in the pool takes a folder, processes all images within it, and returns the combined histograms for each color channel

```
cpu_counts = [1,2, 4, 8,16]  # Number of CPUs to test

for cpu_count in cpu_counts:
    start = time()

    # Using multiprocessing Pool with different number of processes
    with Pool(processes=cpu_count) as pool:
        results = pool.map(process_folder, folders)

    # Reconstructing histograms dictionary from results
    histograms = {folder: data for folder, data in results}

    end = time()
    print(f"Execution Time with {cpu_count} CPU(s): {end - start} seconds")
```

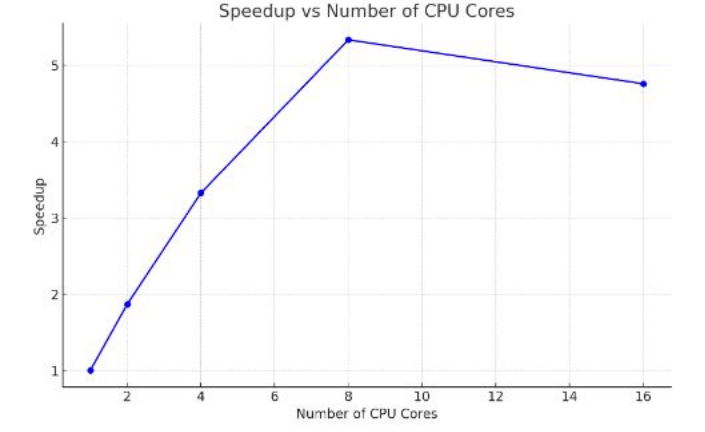
Execution time is 54 sec in sequential run

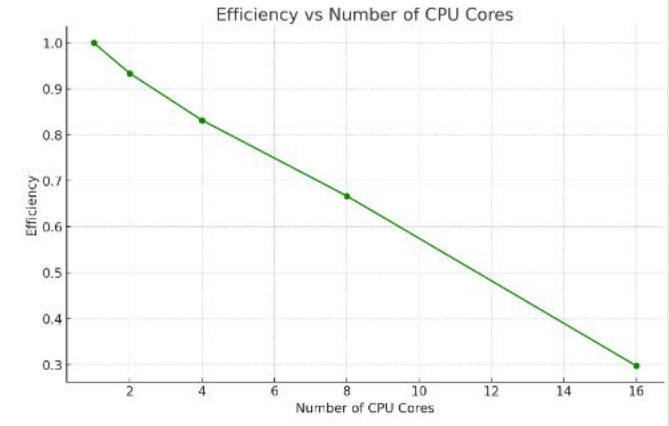
Processing time for EDA and data loading (on a CPU cluster of 8 cores)

Process	Time taken(in sec)
Multiprocessing (with 1 CPU core)	49.15
Multiprocessing (with 2 CPU cores)	26.33
Multiprocessing (with 4 CPU cores)	14.78
Multiprocessing (with 8 CPU cores)	9.214
Multiprocessing (with 16 CPU cores)	10.33



Speedup time and efficiency calculation in multiprocessing





CPU count	Speed up	Efficiency
1 CPU	1.00	1.00
2 CPUs	1.87	0.93
4 CPUs	3.33	0.82
8 CPUs	5.33	0.67
16 CPUs	4.76	0.30

Interpretation of Results

- **Standard Python (Sequential Execution):** The entire computation is done in a single thread, utilizing only one core
- **Multiprocessing with 2 CPUs:** Noticeable performance improvement. Parallel processing starts to show benefits as two cores are used
- **Multiprocessing with 4 CPUs:** With four cores, the process is nearly four times faster than the sequential approach
- **Multiprocessing with 8 CPUs**: Utilizes all available cores, showing a substantial speed-up.
- **Multiprocessing with 16 CPUs:** Performance degradation compared to 8 CPUs. This is because your machine has only 8 cores, and specifying more processes than cores leads to context switching and increased overhead

Using Dask

Dask is a parallel computing library that provides advanced parallelization capabilities, particularly suited for tasks that are both CPU-bound and memory-bound.

- The @delayed decorator is used to make the process_folder function a lazy operation, which means it doesn't compute immediately but waits to be executed.
- The list tasks is created with delayed calls to process_folder for each folder, allowing Dask to manage task scheduling and execution.
- Dask's compute function triggers the parallel computation of all tasks, leveraging multiple threads or processes to execute the tasks concurrently.
- Dask is beneficial when working with large datasets that might not fit entirely into memory, as it can optimize computation by streaming data and using task graphs to execute computations efficiently.

Implementation

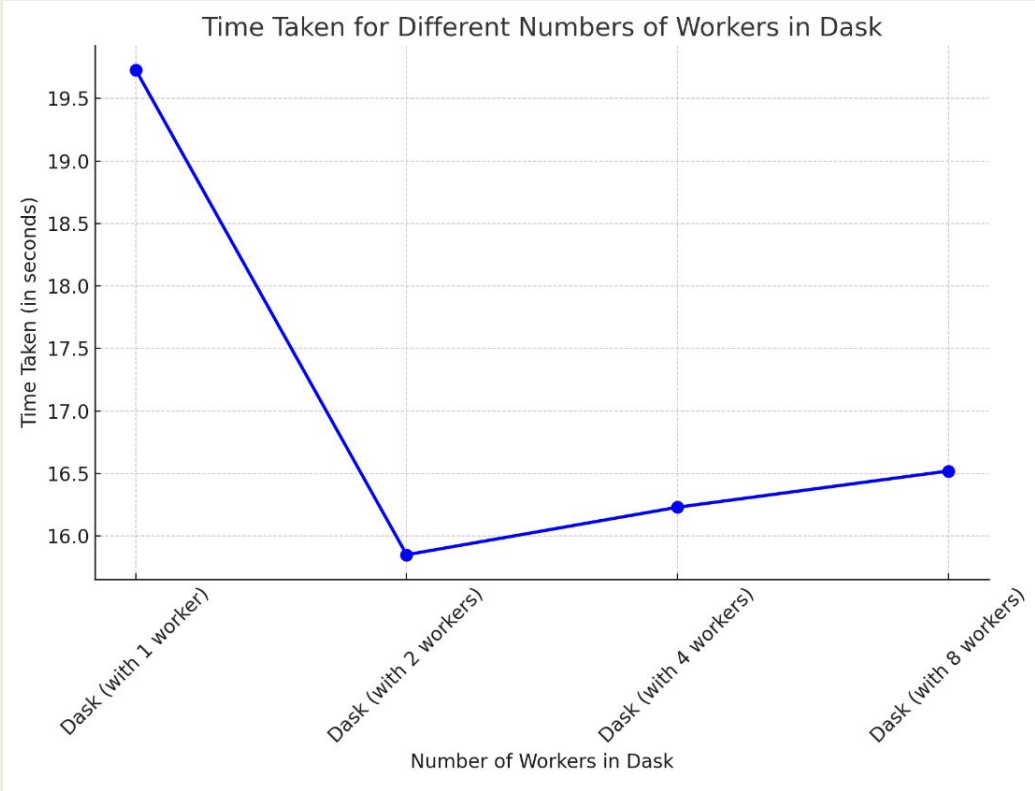
Processing Folders in Parallel:

- A list of tasks is created, where each task is a delayed execution of process_folder for each folder.
- compute(*tasks) is used to execute these tasks in parallel, utilizing the number of workers specified by the current Client.
- After computation, results (folder names and their histograms) are collected and transformed into a dictionary histograms.

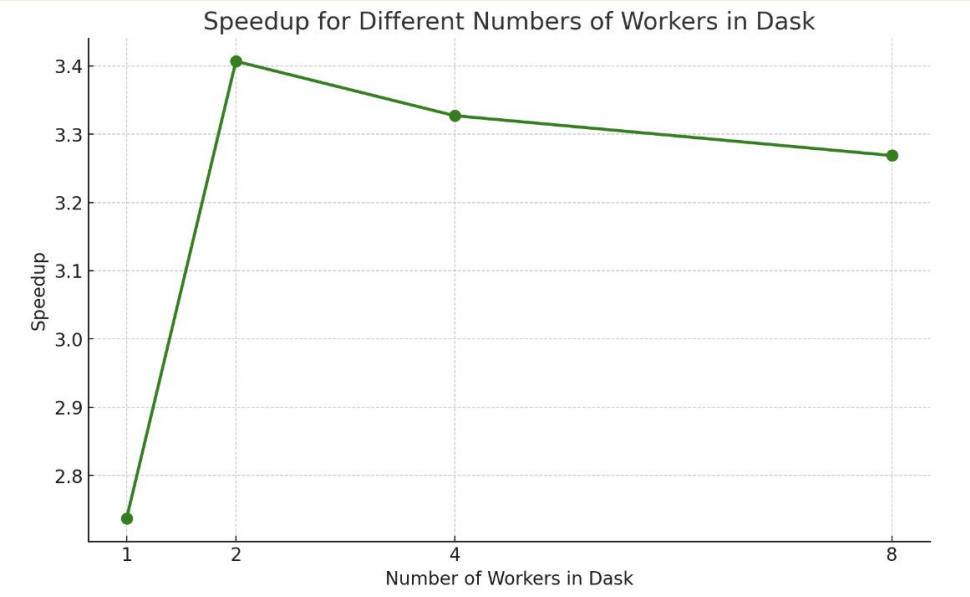
```
cpu_counts = [1, 2, 4, 8]

# Loop over each CPU configuration
for cpu_count in cpu_counts:
    start = time.time()

    # Initialize Dask client with the current number of workers
    client = Client(n_workers=cpu_count)

    # Create a list of delayed tasks
    tasks = [process_folder(folder) for folder in folders]

    # Compute all tasks in parallel
    folder_histograms_results = compute(*tasks)

    # Convert the results to a dictionary
    histograms = {folder: data for folder, data in folder_histograms_results}

    # Shutdown the client
    client.shutdown()

    end = time.time()
    print(f"Execution Time with {cpu_count} CPU(s): {end - start} seconds")
```

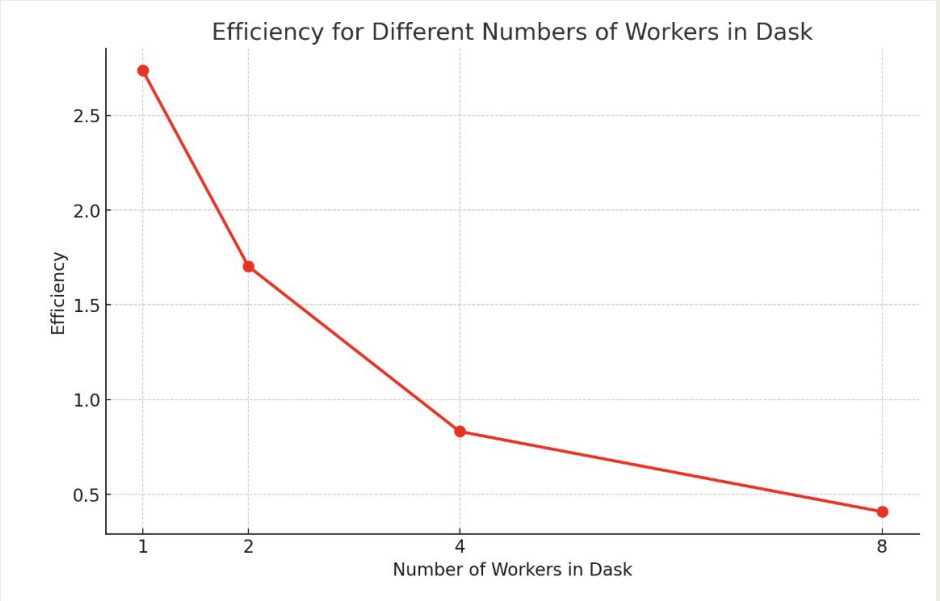
Processing Time for Dask

Process	Time taken(in sec)
Dask (with 1 worker)	19.73
Dask (with 2 worker)	15.85
Dask (with 4 worker)	16.23
Dask (with 8 worker)	16.52



Speedup time and efficiency with Dask





Dask

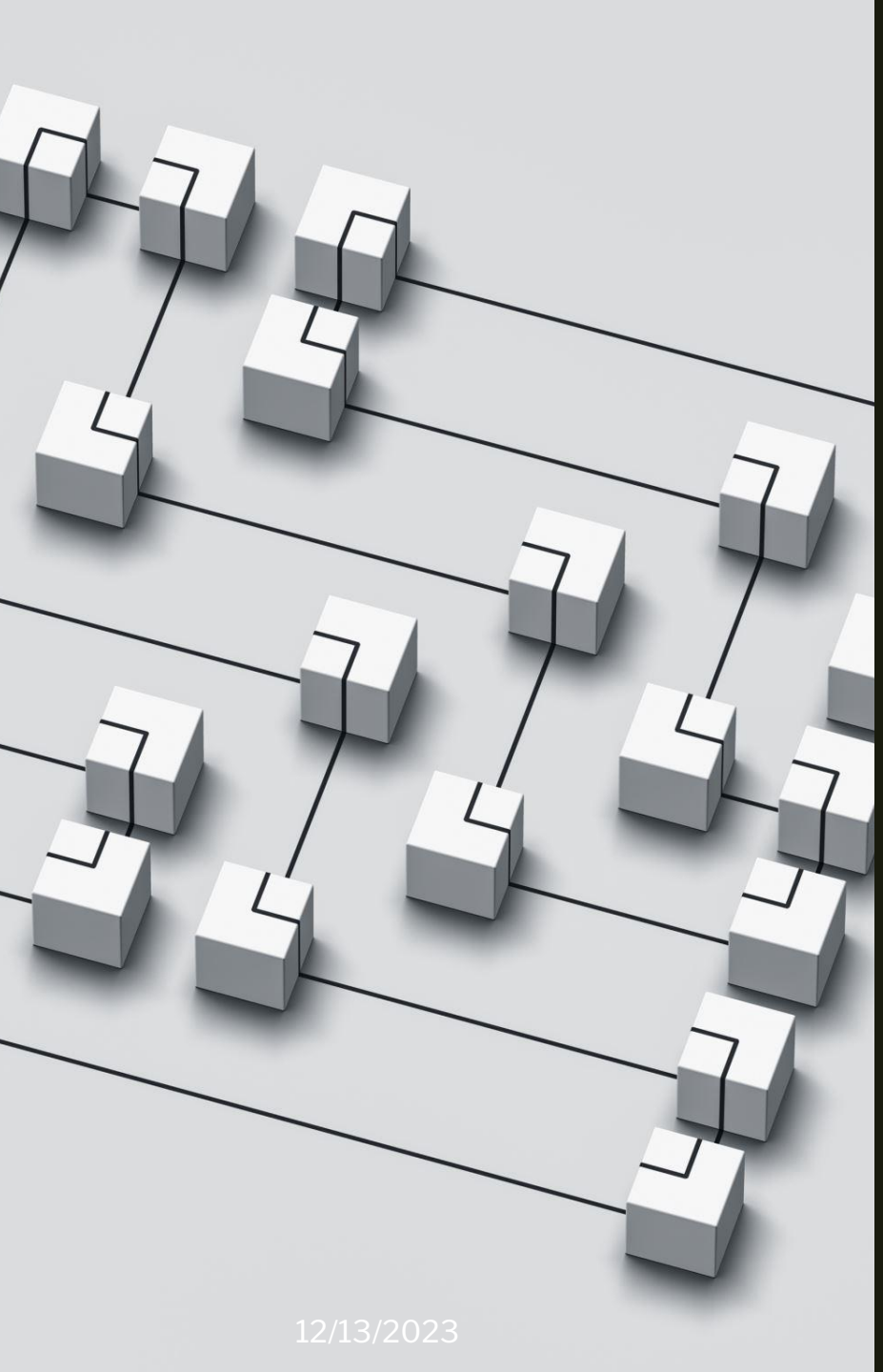
1. **Time Taken by Different Dask Workers**:
    1. As the number of workers increased to 2, there was a notable decrease in time taken, indicating effective parallel processing.
    2. However, increasing workers to 4 and then to 8 did not significantly decrease the time taken compared to 2 workers. This suggests a diminishing return in performance gains with more workers for this specific task.
2. **Speedup**:
    1. The highest speedup was observed with 2 workers, followed closely by 4 and 8 workers. This indicates that the task benefits from parallelization up to a certain point.
    2. The speedup didn't proportionally increase with the number of workers beyond 2. This could be due to overhead costs associated with managing more workers or due to the nature of the task not being perfectly parallelizable.

The reason for the diminishing returns in performance gains with the increasing number of workers beyond two

3. Overhead Costs: Adding more workers can introduce additional overhead costs associated with managing these workers.
4. Task Parallelizability: Not all tasks are perfectly parallelizable.

Data Parallel

- Data Parallelism is when we split the mini-batch of samples into multiple smaller mini-batches and run the computation for each of the smaller mini-batches in parallel.
- Data Parallelism is implemented using `torch.nn.DataParallel`. One can wrap a Module in `DataParallel` and it will be parallelized over multiple GPUs in the batch dimension.
- In the forward pass, the module is replicated on each device, and each replica handles a portion of the input. During the backwards pass, gradients from each replica are summed into the original module. This container parallelizes the application of the given `module` by splitting the input across the specified devices by chunking in the batch dimension (other objects will be copied once per device).



Distributed Data Parallel

Module provided by PyTorch to facilitate distributed training of deep learning models. The key features of DDP in PyTorch:

1. Parallelism Across Multiple GPUs and Nodes: DDP enables parallel training across multiple GPUs and even across multiple nodes (machines). It can distribute model and data across different GPUs and synchronize the gradients during training.
2. Synchronous Gradient Update: In DDP, each process (typically one per GPU) computes the gradients independently and then synchronizes them across all processes. This ensures that each model replica is updated with the same gradients and thus remains consistent.

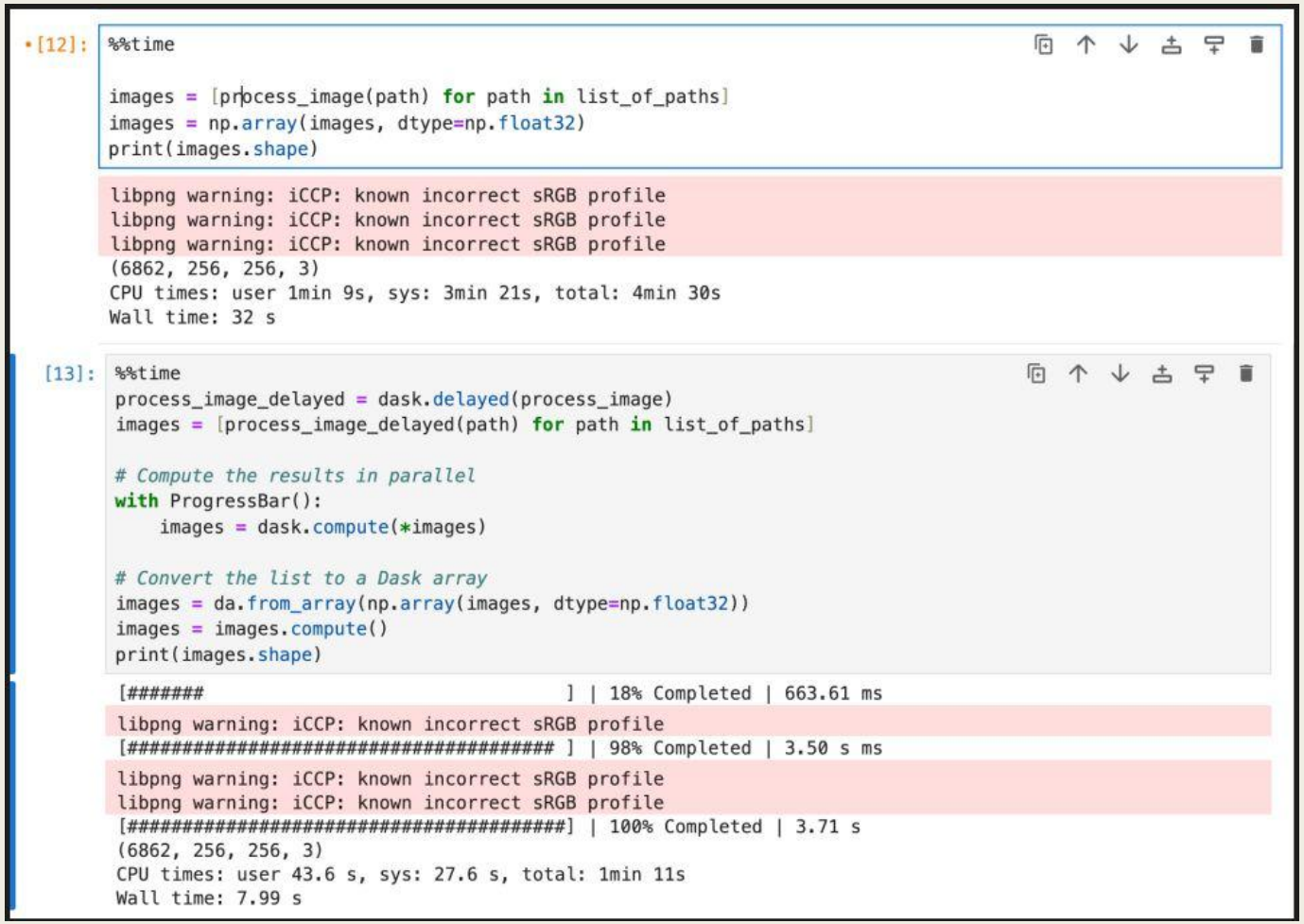
- To use DistributedDataParallel on a host with N GPUs, you should spawn up N processes, ensuring that each process exclusively works on a single GPU from 0 to

Data preprocessing

- The image dataset needs to be converted into readable format to be handled by PyTorch.
- We used OpenCV to read the images and resized them to the dimension size of 256 x 256 x 3 from different shapes
- Each image was then converted into float32 NumPy arrays using **Single-threaded Processing** and **Parallel Processing**
- **Single-threaded Processing** used list comprehension to sequentially transform images to NumPy array
- Whereas **Parallel Processing with Dask** employed Dask's delayed functionality for parallel computation.
- Using `**dask.delayed()**` and executing computations in parallel significantly improved performance.

Results for Data preprocessing

- Single-threaded Processing took **32** seconds to walk through the directory and convert all the images to NumPy arrays
- Parallel Processing with Dask completed the same task in just **8** seconds
- The speedup was **4x**
- In conclusion, parallel processing with Dask improves computational efficiency, especially for larger datasets or complex operations



```
[12]: %%time

images = [process_image(path) for path in list_of_paths]
images = np.array(images, dtype=np.float32)
print(images.shape)

libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: known incorrect sRGB profile
(6862, 256, 256, 3)
CPU times: user 1min 9s, sys: 3min 21s, total: 4min 30s
Wall time: 32 s

[13]: %%time
process_image_delayed = dask.delayed(process_image)
images = [process_image_delayed(path) for path in list_of_paths]

# Compute the results in parallel
with ProgressBar():
    images = dask.compute(*images)

# Convert the list to a Dask array
images = da.from_array(np.array(images, dtype=np.float32))
images = images.compute()
print(images.shape)

[#######                                 ] | 18% Completed | 663.61 ms
libpng warning: iCCP: known incorrect sRGB profile
[######################################## ] | 98% Completed | 3.50 s ms
libpng warning: iCCP: known incorrect sRGB profile
libpng warning: iCCP: known incorrect sRGB profile
[########################################] | 100% Completed | 3.71 s
(6862, 256, 256, 3)
CPU times: user 43.6 s, sys: 27.6 s, total: 1min 11s
Wall time: 7.99 s
```

Model Architecture

- We employed a CNN based Neural Network using PyTorch to perform the multi-label classification task on images
- The layer configuration consisted of convolutional layers, batch normalization, activation functions (Tanh), pooling layers (AvgPool2d), and fully connected layers
- A few dropout layers were also added to avoid overfitting in the model
- The model was designed for image classification tasks, especially viable for datasets with moderate input sizes

```python
class ConvolutionalModel(nn.Module):
    def __init__(self, output_size):
        super().__init__()
        self.net = nn.Sequential(
            ## ConvBlock 1
            nn.Conv2d(3, 6, kernel_size=4, stride=1, padding=0),
            # Input: (b, 3, 256, 256) || Output: (b, 6, 250, 250)
            nn.BatchNorm2d(6),
            nn.Tanh(),
            nn.AvgPool2d(kernel_size=5, stride=5, padding=0),
            # Input: (b, 6, 250, 250) || Output: (b, 6, 50, 50)

            ## ConvBlock 2
            nn.Conv2d(6, 16, kernel_size=5, stride=1, padding=0),
            # Input: (b, 6, 50, 50) || Output: (b, 16, 46, 46)
            nn.BatchNorm2d(16),
            nn.Tanh(),
            nn.AvgPool2d(kernel_size=2, stride=2, padding=0),
            # Input: (b, 16, 46, 46) || Output: (b, 16, 23, 23)

            ## ConvBlock 3
            nn.Conv2d(16, 32, kernel_size=8, stride=1, padding=0),
            # Input: (b, 16, 23, 23) || Output: (b, 32, 16, 16)
            nn.BatchNorm2d(32),
            nn.Tanh(),
            nn.AvgPool2d(kernel_size=4, stride=4, padding=0),
            # Input: (b, 32, 16, 16) || Output: (b, 32, 4, 4)

            ## ConvBlock 4
            nn.Conv2d(32, 120, kernel_size=4, stride=1, padding=0),
            # Input: (b, 32, 4, 4) || Output: (b, 120, 1, 1)
            nn.BatchNorm2d(120),
            nn.Tanh(),
            nn.Flatten(),  # flat to a vector
            # Input: (b, 120, 1, 1) || Output: (b, 120*1*1) = (b, 120)

            nn.Dropout(p=0.32), # Avoid Overfitting
            ## DenseBlock
            nn.Linear(120, 84),
            # Input: (b, 120) || Output: (b, 84)
            nn.Tanh(),
            nn.Linear(84, output_size)
            # Input: (b, 84) || Output: (b, 10)
        )
```

Implementing Data Parallel Approach

- Distributes model training across available GPU devices simultaneously.
- PyTorch's 'DataParallel' module simplifies multi-GPU training.
- Single line of code (nn.DataParallel(net)) scales the model across GPUs
- Optimizes GPU usage by dividing data and computations among devices
- Allows easy scalability for increased model complexity or dataset size

```
if torch.cuda.device_count() > 1:
    net = nn.DataParallel(net)
```

Implementing Distributed Data Parallel Approach

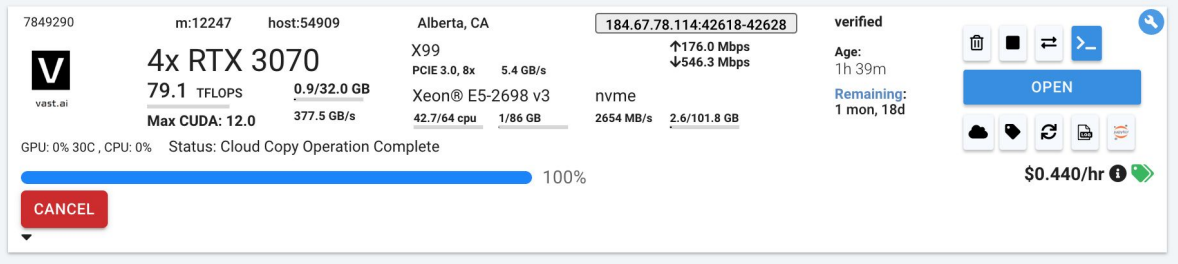
- **GPU Enumeration**:
  - Identifies the number of available GPUs (**world_size**) to facilitate parallel processing and distributed training.
- **Environment Setup**:
  - Establishes the communication infrastructure by defining the address and port for inter-process communication, vital for coordination among different processes.
- **Model Configuration**:
  - Initializes the neural network model, loss function (criterion), and optimizer, essential components for training neural networks.
- **Device Allocation**:
  - Assigns specific devices (GPUs) based on the process rank, ensuring each process operates on its designated GPU for parallel execution.
- **DDP Integration**:
  - Utilizes PyTorch's DistributedDataParallel module to wrap the model, enabling parallel execution across multiple GPUs by synchronizing parameters and gradients during training.

```
# DDP Initialization
world_size = torch.cuda.device_count()
print(world_size)

os.environ['MASTER_ADDR'] = 'localhost'
os.environ['MASTER_PORT'] = '12355'
torch.distributed.init_process_group("nccl", rank=rank, world_size=world_size)

# Initialize the model, optimizer, criterion...
device = torch.device("cuda", rank) if torch.cuda.is_available() else torch.device("cpu")
output_size = len(unique_labels)

net = ConvolutionalModel(output_size).to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)

net = nn.parallel.DistributedDataParallel(net, device_ids=[rank])
```

Training configurations

- 1 x P100 GPU
- 2 x P100 GPU
- 1 x RTX3070 GPU
- 2 x RTX3070 GPU
- 3 x RTX3070 GPU
- 4 x RTX3070 GPU

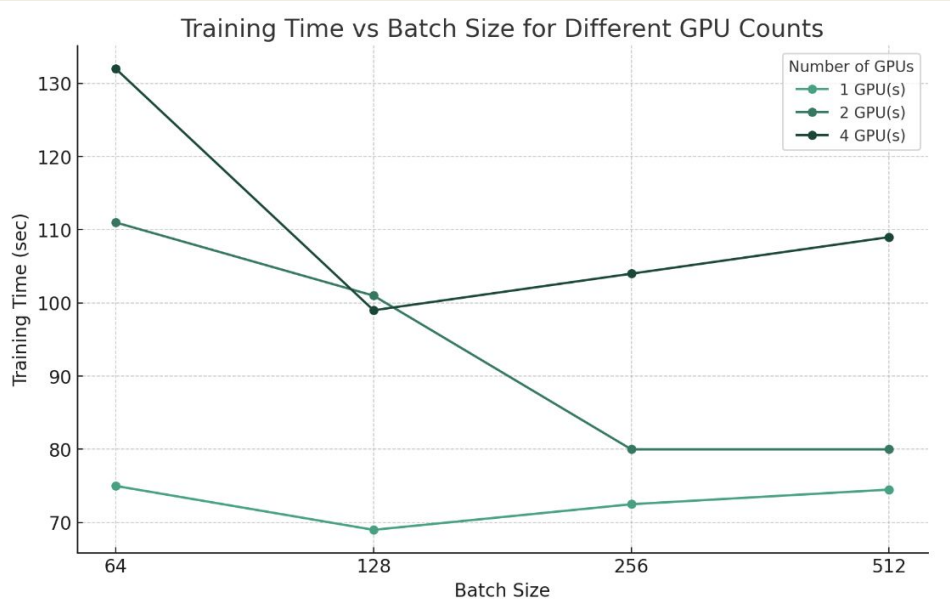
- As the Northeastern cluster was unavailable, we used a paid service to get GPUs.




```
root@C.7849290:/home$ nvidia-smi
Thu Dec 14 03:09:10 2023
+-----------------------------------------------------------------------------+
| NVIDIA-SMI 525.125.06   Driver Version: 525.125.06   CUDA Version: 12.0     |
|-------------------------------+----------------------+----------------------+
| GPU  Name        Persistence-M| Bus-Id        Disp.A | Volatile Uncorr. ECC |
| Fan  Temp  Perf  Pwr:Usage/Cap|         Memory-Usage | GPU-Util  Compute M. |
|                               |                      |               MIG M. |
|===============================+======================+======================|
|   0  NVIDIA GeForce ...  On   | 00000000:03:00.0 Off |                  N/A |
|  0%   38C    P2    51W / 240W |   4143MiB /  8192MiB |     15%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   1  NVIDIA GeForce ...  On   | 00000000:04:00.0 Off |                  N/A |
|  0%   36C    P2    43W / 240W |   2357MiB /  8192MiB |     22%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   2  NVIDIA GeForce ...  On   | 00000000:05:00.0 Off |                  N/A |
|  0%   39C    P2    58W / 240W |   2357MiB /  8192MiB |     76%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+
|   3  NVIDIA GeForce ...  On   | 00000000:81:00.0 Off |                  N/A |
|  0%   39C    P2    64W / 240W |   2357MiB /  8192MiB |     35%      Default |
|                               |                      |                  N/A |
+-------------------------------+----------------------+----------------------+

+-----------------------------------------------------------------------------+
| Processes:                                                                  |
|  GPU   GI   CI        PID   Type   Process name                  GPU Memory |
|        ID   ID                                                   Usage      |
|=============================================================================|
+-----------------------------------------------------------------------------+
```

Training Performance Analysis DP

Model trained on multiple RTX3070 GPU in different batch sizes using Data Parallelism and without Data Parallelism



Batch size	64	128	256	512
1 GPU no DP	75s	69s	72.5s	74.5s
2 GPU with DP	111s	101s	80s	79s
4 GPU with DP	132s	99s	104s	109s

Time analysis on different GPU configurations

Speed up & Efficiency using DP

2 x GPU with DP

Batch size	64	128	256	512
Speed Up	0.67	0.74	0.93	0.94
Efficiency	0.33	0.37	0.46	0.47

4 x GPU with DP

Batch size	64	128	256	512
Speed Up	0.56	0.75	0.72	0.68
Efficiency	0.14	0.18	0.18	0.17

1 GPU x RTX3070

Batch size - 32

```
Rank: 0 || Epoch: 6 || Loss: 1.385 || Accuracy:  0.55 || Time: 4.43
Rank: 0 || Epoch: 7 || Loss: 1.306 || Accuracy:  0.58 || Time: 4.52
Rank: 0 || Epoch: 8 || Loss: 1.236 || Accuracy:  0.61 || Time: 4.47
Rank: 0 || Epoch: 9 || Loss: 1.194 || Accuracy:  0.61 || Time: 4.55
Rank: 0 || Epoch: 10 || Loss: 1.131 || Accuracy:  0.63 || Time: 4.48
Rank: 0 || Epoch: 11 || Loss: 1.092 || Accuracy:  0.64 || Time: 4.47
Rank: 0 || Epoch: 12 || Loss: 1.074 || Accuracy:  0.65 || Time: 4.30
Rank: 0 || Epoch: 13 || Loss: 1.038 || Accuracy:  0.66 || Time: 4.32
Rank: 0 || Epoch: 14 || Loss: 1.024 || Accuracy:  0.65 || Time: 4.31
Rank: 0 || Epoch: 15 || Loss: 0.994 || Accuracy:  0.67 || Time: 4.53
Rank: 0 || Epoch: 16 || Loss: 0.987 || Accuracy:  0.67 || Time: 4.52
Rank: 0 || Epoch: 17 || Loss: 0.957 || Accuracy:  0.69 || Time: 4.53
Rank: 0 || Epoch: 18 || Loss: 0.946 || Accuracy:  0.69 || Time: 4.50
Rank: 0 || Epoch: 19 || Loss: 0.926 || Accuracy:  0.69 || Time: 4.62
91.9847617149353
```

Batch size - 64

```
Rank: 0 || Epoch: 8 || Loss: 1.338 || Accuracy:  0.58 || Time: 4.57
Rank: 0 || Epoch: 9 || Loss: 1.319 || Accuracy:  0.58 || Time: 4.63
Rank: 0 || Epoch: 10 || Loss: 1.245 || Accuracy:  0.61 || Time: 4.60
Rank: 0 || Epoch: 11 || Loss: 1.230 || Accuracy:  0.61 || Time: 4.60
Rank: 0 || Epoch: 12 || Loss: 1.196 || Accuracy:  0.61 || Time: 4.58
Rank: 0 || Epoch: 13 || Loss: 1.154 || Accuracy:  0.62 || Time: 4.56
Rank: 0 || Epoch: 14 || Loss: 1.139 || Accuracy:  0.63 || Time: 4.62
Rank: 0 || Epoch: 15 || Loss: 1.125 || Accuracy:  0.63 || Time: 4.60
Rank: 0 || Epoch: 16 || Loss: 1.090 || Accuracy:  0.64 || Time: 4.63
Rank: 0 || Epoch: 17 || Loss: 1.083 || Accuracy:  0.65 || Time: 4.54
Rank: 0 || Epoch: 18 || Loss: 1.043 || Accuracy:  0.66 || Time: 4.57
Rank: 0 || Epoch: 19 || Loss: 1.023 || Accuracy:  0.67 || Time: 4.57
Number of available GPUs: 1
93.80670881271362
```

Batch size - 128

```
Rank: 0 || Epoch: 9 || Loss: 1.475 || Accuracy:  0.54 || Time: 4.46
Rank: 0 || Epoch: 10 || Loss: 1.452 || Accuracy:  0.54 || Time: 4.45
Rank: 0 || Epoch: 11 || Loss: 1.414 || Accuracy:  0.56 || Time: 4.41
Rank: 0 || Epoch: 12 || Loss: 1.400 || Accuracy:  0.57 || Time: 4.42
Rank: 0 || Epoch: 13 || Loss: 1.364 || Accuracy:  0.58 || Time: 4.40
Rank: 0 || Epoch: 14 || Loss: 1.345 || Accuracy:  0.58 || Time: 4.37
Rank: 0 || Epoch: 15 || Loss: 1.314 || Accuracy:  0.59 || Time: 4.45
Rank: 0 || Epoch: 16 || Loss: 1.320 || Accuracy:  0.59 || Time: 4.42
Rank: 0 || Epoch: 17 || Loss: 1.311 || Accuracy:  0.59 || Time: 4.48
Rank: 0 || Epoch: 18 || Loss: 1.304 || Accuracy:  0.58 || Time: 4.47
Rank: 0 || Epoch: 19 || Loss: 1.234 || Accuracy:  0.61 || Time: 4.46
Number of available GPUs: 1
91.07798266410828
```

Batch size - 256

```
Rank: 0 || Epoch: 9 || Loss: 1.604 || Accuracy:  0.50 || Time: 4.55
Rank: 0 || Epoch: 10 || Loss: 1.578 || Accuracy:  0.51 || Time: 4.51
Rank: 0 || Epoch: 11 || Loss: 1.553 || Accuracy:  0.52 || Time: 4.52
Rank: 0 || Epoch: 12 || Loss: 1.531 || Accuracy:  0.52 || Time: 4.53
Rank: 0 || Epoch: 13 || Loss: 1.499 || Accuracy:  0.54 || Time: 4.61
Rank: 0 || Epoch: 14 || Loss: 1.483 || Accuracy:  0.54 || Time: 4.57
Rank: 0 || Epoch: 15 || Loss: 1.546 || Accuracy:  0.52 || Time: 4.93
Rank: 0 || Epoch: 16 || Loss: 1.483 || Accuracy:  0.54 || Time: 4.64
Rank: 0 || Epoch: 17 || Loss: 1.443 || Accuracy:  0.55 || Time: 4.91
Rank: 0 || Epoch: 18 || Loss: 1.427 || Accuracy:  0.56 || Time: 4.61
Rank: 0 || Epoch: 19 || Loss: 1.392 || Accuracy:  0.57 || Time: 4.63
Number of available GPUs: 1
93.35435628890991
```

2 GPU x RTX3070

Batch size - 32

```
Rank: 1 || Epoch: 11 || Loss: 1.277 || Accuracy:  0.58 || Time: 4.18
Rank: 0 || Epoch: 11 || Loss: 1.274 || Accuracy:  0.59 || Time: 4.19
Rank: 0 || Epoch: 12 || Loss: 1.222 || Accuracy:  0.61 || Time: 4.05
Rank: 1 || Epoch: 12 || Loss: 1.190 || Accuracy:  0.62 || Time: 4.05
Rank: 0 || Epoch: 13 || Loss: 1.195 || Accuracy:  0.61 || Time: 4.01
Rank: 1 || Epoch: 13 || Loss: 1.203 || Accuracy:  0.62 || Time: 4.01
Rank: 1 || Epoch: 14 || Loss: 1.138 || Accuracy:  0.63 || Time: 4.08
Rank: 0 || Epoch: 14 || Loss: 1.145 || Accuracy:  0.63 || Time: 4.08
Rank: 0 || Epoch: 15 || Loss: 1.158 || Accuracy:  0.62 || Time: 3.95
Rank: 1 || Epoch: 15 || Loss: 1.122 || Accuracy:  0.63 || Time: 3.96
Rank: 0 || Epoch: 16 || Loss: 1.147 || Accuracy:  0.63 || Time: 4.07
Rank: 1 || Epoch: 16 || Loss: 1.119 || Accuracy:  0.64 || Time: 4.07
Rank: 0 || Epoch: 17 || Loss: 1.114 || Accuracy:  0.63 || Time: 4.02
Rank: 1 || Epoch: 17 || Loss: 1.113 || Accuracy:  0.64 || Time: 4.02
Rank: 1 || Epoch: 18 || Loss: 1.049 || Accuracy:  0.66 || Time: 4.57
Rank: 0 || Epoch: 18 || Loss: 1.047 || Accuracy:  0.66 || Time: 4.57
Rank: 0 || Epoch: 19 || Loss: 1.035 || Accuracy:  0.66 || Time: 4.18
97.1252875328064
Rank: 1 || Epoch: 19 || Loss: 1.029 || Accuracy:  0.66 || Time: 4.18
97.12546634674072
```

Batch size - 64

```
Rank: 0 || Epoch: 16 || Loss: 1.337 || Accuracy:  0.58 || Time: 3.69
Rank: 1 || Epoch: 16 || Loss: 1.346 || Accuracy:  0.57 || Time: 3.69
Rank: 1 || Epoch: 17 || Loss: 1.331 || Accuracy:  0.58 || Time: 3.51
Rank: 0 || Epoch: 17 || Loss: 1.305 || Accuracy:  0.60 || Time: 3.51
Rank: 1 || Epoch: 18 || Loss: 1.289 || Accuracy:  0.59 || Time: 3.52
Rank: 0 || Epoch: 18 || Loss: 1.303 || Accuracy:  0.59 || Time: 3.52
Rank: 1 || Epoch: 19 || Loss: 1.281 || Accuracy:  0.60 || Time: 3.49
Number of available GPUs: 2
79.61142706871033
Rank: 0 || Epoch: 19 || Loss: 1.276 || Accuracy:  0.60 || Time: 3.49
Number of available GPUs: 2
79.61195635795593
```

Batch size - 128

```
Rank: 0 || Epoch: 14 || Loss: 1.490 || Accuracy:  0.54 || Time: 3.45
Rank: 1 || Epoch: 14 || Loss: 1.491 || Accuracy:  0.54 || Time: 3.45
Rank: 1 || Epoch: 15 || Loss: 1.458 || Accuracy:  0.54 || Time: 2.74
Rank: 0 || Epoch: 15 || Loss: 1.455 || Accuracy:  0.55 || Time: 2.74
Rank: 0 || Epoch: 16 || Loss: 1.459 || Accuracy:  0.55 || Time: 2.66
Rank: 1 || Epoch: 16 || Loss: 1.471 || Accuracy:  0.54 || Time: 2.66
Rank: 0 || Epoch: 17 || Loss: 1.444 || Accuracy:  0.56 || Time: 3.09
Rank: 1 || Epoch: 17 || Loss: 1.437 || Accuracy:  0.56 || Time: 3.09
Rank: 1 || Epoch: 18 || Loss: 1.433 || Accuracy:  0.55 || Time: 2.84
Rank: 0 || Epoch: 18 || Loss: 1.419 || Accuracy:  0.56 || Time: 2.84
Rank: 0 || Epoch: 19 || Loss: 1.395 || Accuracy:  0.57 || Time: 2.68
Number of available GPUs: 2
63.36701250076294
Rank: 1 || Epoch: 19 || Loss: 1.383 || Accuracy:  0.58 || Time: 2.68
Number of available GPUs: 2
63.366753578186035
```

Batch size - 256

```
Rank: 1 || Epoch: 14 || Loss: 1.666 || Accuracy:  0.49 || Time: 3.26
Rank: 0 || Epoch: 14 || Loss: 1.676 || Accuracy:  0.48 || Time: 3.26
Rank: 1 || Epoch: 15 || Loss: 1.653 || Accuracy:  0.48 || Time: 3.20
Rank: 0 || Epoch: 15 || Loss: 1.643 || Accuracy:  0.50 || Time: 3.20
Rank: 0 || Epoch: 16 || Loss: 1.610 || Accuracy:  0.50 || Time: 3.09
Rank: 1 || Epoch: 16 || Loss: 1.636 || Accuracy:  0.49 || Time: 3.09
Rank: 1 || Epoch: 17 || Loss: 1.616 || Accuracy:  0.50 || Time: 3.26
Rank: 0 || Epoch: 17 || Loss: 1.637 || Accuracy:  0.48 || Time: 3.26
Rank: 1 || Epoch: 18 || Loss: 1.599 || Accuracy:  0.51 || Time: 3.47
Rank: 0 || Epoch: 18 || Loss: 1.593 || Accuracy:  0.50 || Time: 3.47
Rank: 0 || Epoch: 19 || Loss: 1.592 || Accuracy:  0.50 || Time: 3.37
Number of available GPUs: 2
66.64922547340393
Rank: 1 || Epoch: 19 || Loss: 1.578 || Accuracy:  0.51 || Time: 3.38
Number of available GPUs: 2
66.64951729774475
```

3 GPU x RTX3070

Batch size - 32

```
Rank: 0 || Epoch: 15 || Loss: 1.296 || Accuracy:  0.59 || Time: 7.48
Rank: 2 || Epoch: 16 || Loss: 1.238 || Accuracy:  0.60 || Time: 7.58
Rank: 1 || Epoch: 16 || Loss: 1.245 || Accuracy:  0.59 || Time: 7.58
Rank: 0 || Epoch: 16 || Loss: 1.211 || Accuracy:  0.61 || Time: 7.58
Rank: 1 || Epoch: 17 || Loss: 1.219 || Accuracy:  0.60 || Time: 7.80
Rank: 0 || Epoch: 17 || Loss: 1.204 || Accuracy:  0.61 || Time: 7.80
Rank: 2 || Epoch: 17 || Loss: 1.207 || Accuracy:  0.61 || Time: 7.81
Rank: 0 || Epoch: 18 || Loss: 1.194 || Accuracy:  0.61 || Time: 7.19
Rank: 2 || Epoch: 18 || Loss: 1.199 || Accuracy:  0.61 || Time: 7.18
Rank: 1 || Epoch: 18 || Loss: 1.194 || Accuracy:  0.61 || Time: 7.19
Rank: 2 || Epoch: 19 || Loss: 1.174 || Accuracy:  0.62 || Time: 7.66
Number of available GPUs: 3
136.14690804481506
Rank: 1 || Epoch: 19 || Loss: 1.092 || Accuracy:  0.66 || Time: 7.66
Number of available GPUs: 3
136.14783596992493
Rank: 0 || Epoch: 19 || Loss: 1.186 || Accuracy:  0.62 || Time: 7.66
Number of available GPUs: 3
136.1485390663147
```

Batch size - 64

```
Rank: 2 || Epoch: 17 || Loss: 1.430 || Accuracy:  0.56 || Time: 5.84
Rank: 0 || Epoch: 18 || Loss: 1.404 || Accuracy:  0.55 || Time: 4.78
Rank: 2 || Epoch: 18 || Loss: 1.415 || Accuracy:  0.57 || Time: 4.78
Rank: 1 || Epoch: 18 || Loss: 1.407 || Accuracy:  0.57 || Time: 4.78
Rank: 1 || Epoch: 19 || Loss: 1.392 || Accuracy:  0.57 || Time: 4.72
Number of available GPUs: 3
101.59220385551453
Rank: 2 || Epoch: 19 || Loss: 1.368 || Accuracy:  0.58 || Time: 4.72
Number of available GPUs: 3
101.59203910827637
Rank: 0 || Epoch: 19 || Loss: 1.397 || Accuracy:  0.57 || Time: 4.72
Number of available GPUs: 3
101.5934100151062
```

Batch size - 128

```
Rank: 2 || Epoch: 17 || Loss: 1.591 || Accuracy:  0.50 || Time: 4.21
Rank: 2 || Epoch: 18 || Loss: 1.565 || Accuracy:  0.51 || Time: 3.93
Rank: 0 || Epoch: 18 || Loss: 1.519 || Accuracy:  0.53 || Time: 3.93
Rank: 1 || Epoch: 18 || Loss: 1.548 || Accuracy:  0.51 || Time: 3.93
Rank: 1 || Epoch: 19 || Loss: 1.530 || Accuracy:  0.52 || Time: 4.36
Number of available GPUs: 3
87.89397931098938
Rank: 0 || Epoch: 19 || Loss: 1.552 || Accuracy:  0.51 || Time: 4.36
Number of available GPUs: 3
87.89493036270142
Rank: 2 || Epoch: 19 || Loss: 1.552 || Accuracy:  0.51 || Time: 4.36
Number of available GPUs: 3
87.89747834205627
```

Batch size - 256

```
Rank: 1 || Epoch: 16 || Loss: 1.730 || Accuracy:  0.46 || Time: 4.02
Rank: 2 || Epoch: 16 || Loss: 1.725 || Accuracy:  0.46 || Time: 4.02
Rank: 0 || Epoch: 17 || Loss: 1.696 || Accuracy:  0.47 || Time: 3.99
Rank: 2 || Epoch: 17 || Loss: 1.706 || Accuracy:  0.47 || Time: 3.99
Rank: 1 || Epoch: 17 || Loss: 1.743 || Accuracy:  0.46 || Time: 3.99
Rank: 1 || Epoch: 18 || Loss: 1.699 || Accuracy:  0.46 || Time: 4.62
Rank: 2 || Epoch: 18 || Loss: 1.741 || Accuracy:  0.45 || Time: 4.62
Rank: 0 || Epoch: 18 || Loss: 1.715 || Accuracy:  0.46 || Time: 4.63
Rank: 0 || Epoch: 19 || Loss: 1.675 || Accuracy:  0.46 || Time: 3.97
Rank: 2 || Epoch: 19 || Loss: 1.650 || Accuracy:  0.48 || Time: 3.97
Number of available GPUs: 3
Number of available GPUs: 3
88.003422498703
88.00323367118835
Rank: 1 || Epoch: 19 || Loss: 1.687 || Accuracy:  0.45 || Time: 3.97
Number of available GPUs: 3
88.00375008583069
```

4 GPU x RTX 3070

Batch size - 32

```
Rank: 0 || Epoch: 17 || Loss: 1.261 || Accuracy:  0.61 || Time: 12.35
Rank: 2 || Epoch: 17 || Loss: 1.284 || Accuracy:  0.59 || Time: 12.35
Rank: 1 || Epoch: 18 || Loss: 1.245 || Accuracy:  0.61 || Time: 12.48
Rank: 0 || Epoch: 18 || Loss: 1.228 || Accuracy:  0.61 || Time: 12.48
Rank: 3 || Epoch: 18 || Loss: 1.234 || Accuracy:  0.61 || Time: 12.48
Rank: 2 || Epoch: 18 || Loss: 1.254 || Accuracy:  0.60 || Time: 12.48
Rank: 2 || Epoch: 19 || Loss: 1.210 || Accuracy:  0.62 || Time: 12.59
Number of available GPUs: 4
249.65013790130615
Rank: 3 || Epoch: 19 || Loss: 1.218 || Accuracy:  0.62 || Time: 12.59
Number of available GPUs: 4
249.65002608299255
Rank: 1 || Epoch: 19 || Loss: 1.225 || Accuracy:  0.62 || Time: 12.59
Number of available GPUs: 4
249.650949716568
Rank: 0 || Epoch: 19 || Loss: 1.229 || Accuracy:  0.61 || Time: 12.59
Number of available GPUs: 4
249.65137434005737
```

Batch size - 64

```
Rank: 2 || Epoch: 18 || Loss: 1.481 || Accuracy:  0.54 || Time: 11.97
Rank: 0 || Epoch: 19 || Loss: 1.481 || Accuracy:  0.52 || Time: 11.50
Number of available GPUs: 4
236.01985788345337
Rank: 3 || Epoch: 19 || Loss: 1.464 || Accuracy:  0.55 || Time: 11.50
Number of available GPUs: 4
236.0199897289276
Rank: 1 || Epoch: 19 || Loss: 1.448 || Accuracy:  0.56 || Time: 11.50
Number of available GPUs: 4
236.02083134651184
Rank: 2 || Epoch: 19 || Loss: 1.434 || Accuracy:  0.56 || Time: 11.50
Number of available GPUs: 4
236.02105331420898
```

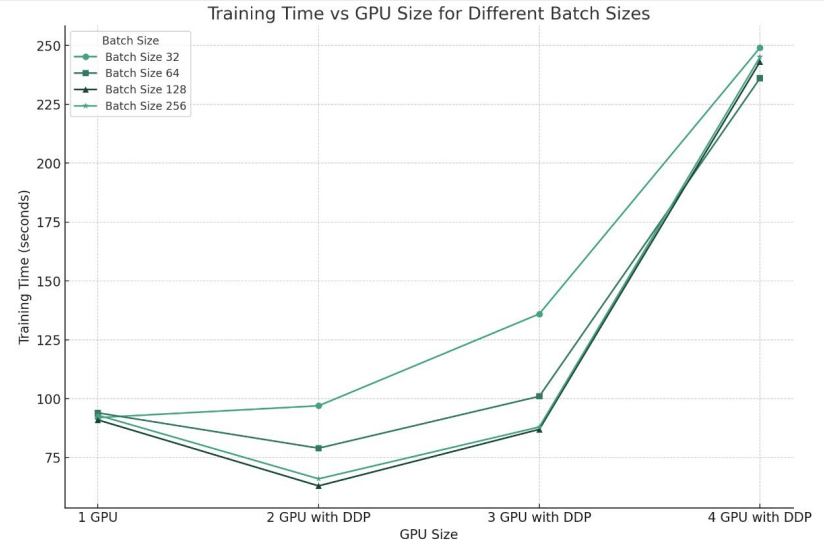
Batch size - 128

```
Rank: 1 || Epoch: 18 || Loss: 1.766 || Accuracy:  0.45 || Time: 12.46
Rank: 2 || Epoch: 18 || Loss: 1.737 || Accuracy:  0.48 || Time: 12.46
Rank: 0 || Epoch: 19 || Loss: 1.735 || Accuracy:  0.47 || Time: 11.95
Number of available GPUs: 4
245.20328736305237
Rank: 3 || Epoch: 19 || Loss: 1.714 || Accuracy:  0.48 || Time: 11.95
Number of available GPUs: 4
245.20352578163147
Rank: 1 || Epoch: 19 || Loss: 1.767 || Accuracy:  0.46 || Time: 11.95
Number of available GPUs: 4
245.20385456085205
Rank: 2 || Epoch: 19 || Loss: 1.749 || Accuracy:  0.46 || Time: 11.95
Number of available GPUs: 4
245.20393633842468
```

Batch size - 256

```
Rank: 0 || Epoch: 17 || Loss: 1.768 || Accuracy:  0.45 || Time: 12.41
Rank: 1 || Epoch: 17 || Loss: 1.742 || Accuracy:  0.47 || Time: 12.41
Rank: 2 || Epoch: 17 || Loss: 1.755 || Accuracy:  0.47 || Time: 12.41
Rank: 3 || Epoch: 18 || Loss: 1.740 || Accuracy:  0.46 || Time: 12.46
Rank: 0 || Epoch: 18 || Loss: 1.744 || Accuracy:  0.46 || Time: 12.46
Rank: 1 || Epoch: 18 || Loss: 1.766 || Accuracy:  0.45 || Time: 12.46
Rank: 2 || Epoch: 18 || Loss: 1.737 || Accuracy:  0.48 || Time: 12.46
Rank: 0 || Epoch: 19 || Loss: 1.735 || Accuracy:  0.47 || Time: 11.95
Number of available GPUs: 4
245.20328736305237
Rank: 3 || Epoch: 19 || Loss: 1.714 || Accuracy:  0.48 || Time: 11.95
Number of available GPUs: 4
245.20352578163147
Rank: 1 || Epoch: 19 || Loss: 1.767 || Accuracy:  0.46 || Time: 11.95
Number of available GPUs: 4
245.20385456085205
Rank: 2 || Epoch: 19 || Loss: 1.749 || Accuracy:  0.46 || Time: 11.95
Number of available GPUs: 4
245.20393633842468
```

Training Performance Analysis DDP



Batch size	32	64	128	256
1 GPU	92s	94s	91s	93s
2 GPU with DDP	97s	79s	63s	66s
3 GPU with DDP	136s	101s	87s	88s
4 GPU with DDP	249s	236s	243s	245s

Speed up & Efficiency using DDP

2 x GPU with DDP

Batch size	32	64	128	256
Speed Up	0.95	1.19	1.44	1.41
Efficiency	0.47	0.59	0.72	0.70

3 x GPU with DDP

Batch size	32	64	128	256
Speed Up	0.676	0.931	1.046	1.057
Efficiency	0.225	0.310	0.349	0.352

4 x GPU with DDP

Batch size	32	64	128	256
Speed Up	0.37	0.40	0.37	0.38
Efficiency	0.09	0.10	0.09	0.09

Results

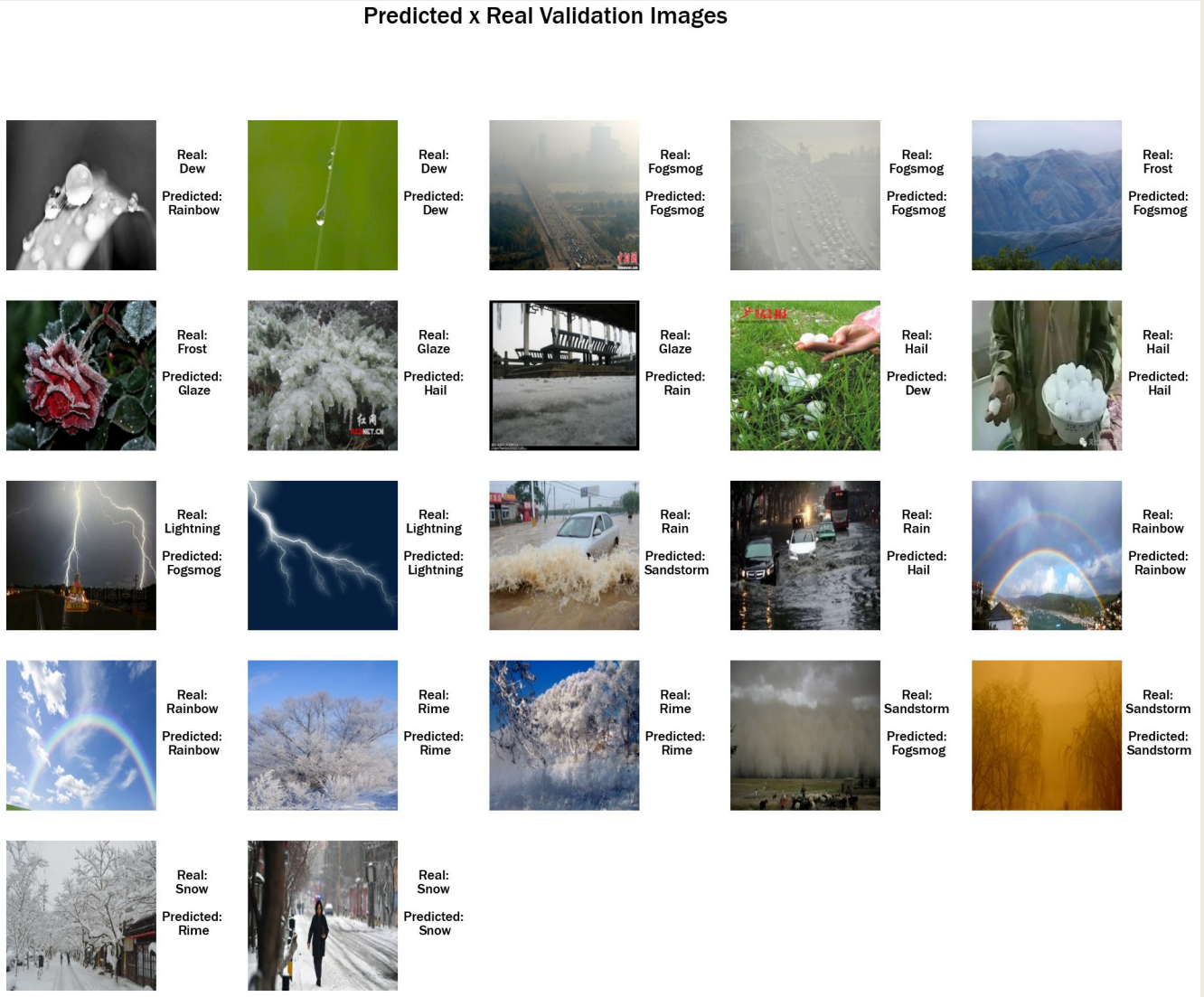
- **Varied Class Performance**: The model exhibits diverse performance across different classes. Some classes, like '**Fogsmog**,' **'Sandstorm'** show higher precision and recall, indicating more accurate predictions. However, other classes, such as **'Snow'** and **'Rainbow**,' display lower precision or recall, suggesting challenges in correctly identifying these weather phenomena.
- **Imbalanced Class Issues**: There's an indication of imbalanced class issues, as seen in disparities in support (number of instances) among classes. This imbalance might affect the model's performance.
- **Trade-off Between Precision and Recall**: The F1-Score, a balance between precision and recall, varies across classes. This highlights a potential trade-off between precision and recall in the model's predictions. Some classes achieve a good balance, while others might prioritize precision over recall or vice versa, affecting overall model performance.

	Precision	Recall	F1-Score	Support
Dew	0.722973	0.849206	0.781022	126.000000
Fogsmog	0.514970	0.924731	0.661538	186.000000
Frost	0.600000	0.170455	0.265487	88.000000
Glaze	0.406250	0.314516	0.354545	124.000000
Hail	0.586538	0.429577	0.495935	142.000000
Lightning	0.758065	0.635135	0.691176	74.000000
Rain	0.589744	0.460000	0.516854	100.000000
Rainbow	0.230088	0.565217	0.327044	46.000000
Rime	0.698565	0.610879	0.651786	239.000000
Sandstorm	0.650000	0.787879	0.712329	132.000000
Snow	0.636364	0.241379	0.350000	116.000000
Accuracy	0.576111	0.576111	0.576111	0.576111
Macro Avg	0.581232	0.544452	0.527974	1373.000000
Weighted Avg	0.601291	0.576111	0.558984	1373.000000

Classification Report

Results analysis

- Validation of model shows 55% avg accuracy
- Peer-Peer transport using PyTorch Distributed backend for communication between devices was not supported
- Synchronizing gradients and calculations increases complexity and can reduce accuracy
- Overhead causing bottlenecks for data transfer



Conclusion

- As verified before, the model mistakes Frost with Glaze and Rime because of their similarities. The remainder labels generalized very well.
- The models have an accuracy of 55%-68% on new images.
- The CNN could be improved by maybe using a pre-trained model or introducing other methods of regularization to ensure it doesn't overfit the training data
- **Efficiency Trade-off:**
  - Data parallelism across multiple GPUs does not always showcase superior performance compared to single GPUs due to overhead.
- **Optimized Computation Time:**
  - Dask and multiprocessing exhibit significant improvements in execution time compared to serial execution, with Dask Array notably enhancing NumPy Array computation.
- **Strategic GPU Utilization:**
  - Optimal performance is achieved by leveraging Dask and multiprocessing on a configuration with 4 GPUs, balancing efficiency and computational speed.

References

1. https://docs.dask.org/en/stable/
2. https://agupubs.onlinelibrary.wiley.com/doi/full/10.1029/2020EA001604
3. https://arxiv.org/pdf/2212.07936.pdf
4. https://docs.python.org/3/library/multiprocessing.html
5. https://pytorch.org/docs/stable/index.html
6. https://vast.ai/
7. https://pytorch.org/tutorials/intermediate/ddp_tutorial.html

THANK YOU