**Melanoma Detection using Deep Learning and Parallelism**

Team 6 - Saiyam Doshi & Govind Mudavadkar

Instructor: Prof. Handan Liu

## **Introduction**

## **Background**

Melanoma, the most aggressive form of skin cancer, poses a significant threat due to its ability to spread rapidly throughout the body. Early detection is crucial for improving survival rates and treatment outcomes. When caught in its early stages, melanoma has a 5-year survival rate of 99%, but this rate drops dramatically as the cancer progresses. Early detection allows for less invasive treatments and prevents metastasis, which occurs when cancer cells spread beyond the skin to other organs.

One of the most promising advancements in melanoma detection is the integration of artificial intelligence (AI) with various imaging modalities. A systematic review of AI-based approaches applied to non-invasive imaging for early melanoma detection revealed significant progress in this field. For dermoscopy, AI algorithms consistently demonstrated performance comparable to or better than dermatologists, with mean sensitivity and specificity of 83.01% and 85.58%, respectively5. Even more impressive results were observed with other imaging techniques: AI-based algorithms achieved 95% accuracy when used in conjunction with Optical Coherence Tomography (OCT), and 82.72% accuracy with Reflectance Confocal Microscopy (RCM)5. These advancements are not limited to clinical settings; innovative solutions like the Health of Things Melanoma Detection System (HTMDS) have achieved over 98% accuracy for detection and over 99% accuracy for skin cancer segmentation in dermoscopic images1. Such technologies have the potential to revolutionize melanoma screening by improving accessibility, reducing the need for unnecessary biopsies, and enhancing the overall efficiency of the diagnostic process.

## **Motivation**

Skin cancer is a prevalent and potentially life-threatening condition, making early and accurate detection crucial for effective treatment. Traditional manual inspection of skin lesions by dermatologists, while effective, can be time-consuming and subject to human error, especially when analyzing a large number of cases. The use of automated image-based diagnostic tools offers a promising solution by leveraging machine learning to assist in the accurate identification of malignant skin lesions. Such tools not only provide objective and consistent analysis but also serve as a valuable aid to dermatologists, enhancing diagnostic accuracy and improving patient outcomes.

This project aims to develop a robust machine learning model capable of analyzing high-resolution skin lesion images to differentiate between benign and malignant cases. By integrating advanced parallel computation techniques, the model will process large image datasets more efficiently, supporting real-time detection and scalability for widespread clinical use. The combination of state-of-the-art image processing algorithms, deep learning architectures, and parallel processing will ensure that the system remains both accurate and responsive. Ultimately, this approach seeks to streamline the diagnostic process, empowering healthcare professionals with cutting-edge technology to combat skin cancer effectively.

Top of Form

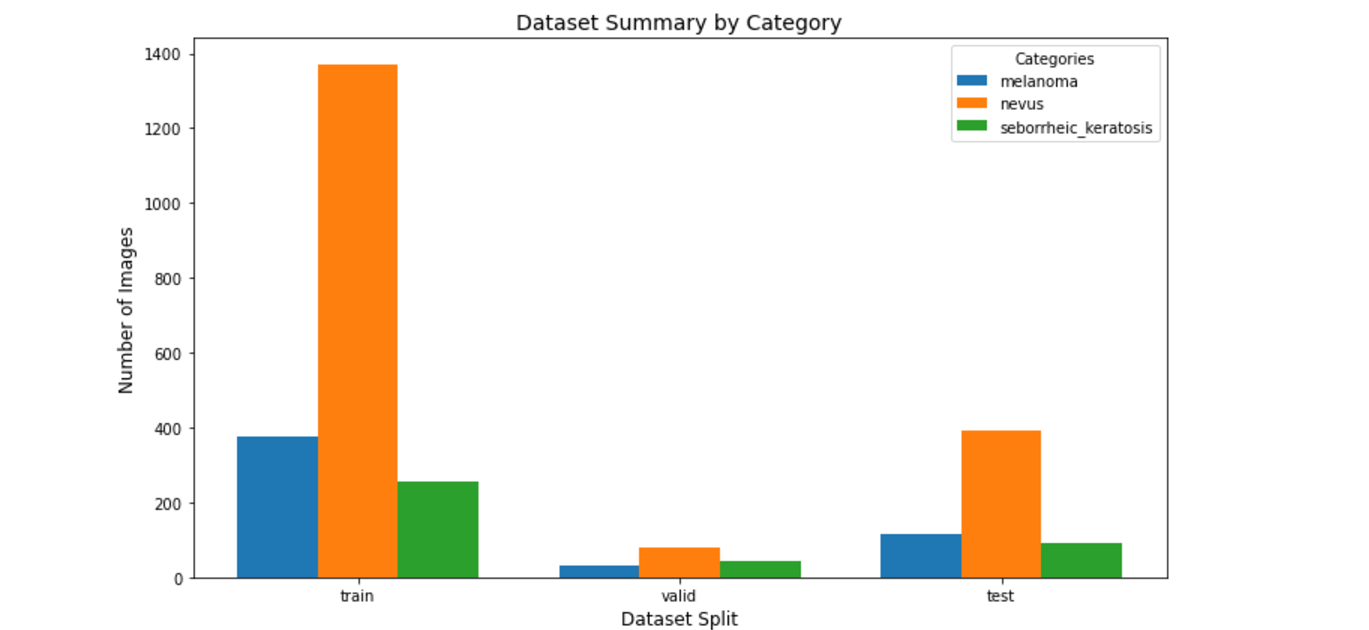
### **Goals**

This project aims to develop a machine learning-based system for accurate detection and classification of melanoma and other skin lesions using dermatoscopic images. Leveraging the "Skin Lesion Analysis Towards Melanoma Detection" dataset, the goal is to build a deep learning model that can distinguish between malignant and benign lesions, providing critical support for early melanoma diagnosis. The primary focus is on identifying melanoma, a serious and potentially life-threatening form of skin cancer, while also classifying various other skin lesions, such as benign nevi and seborrheic keratosis. To achieve this, the project will utilize Convolutional Neural Networks (CNNs), which are well-suited for image analysis, to train the model to automatically assess and categorize skin lesions, thereby enhancing diagnostic accuracy, minimizing human error, and enabling quicker clinical assessments.

Additionally, this project will integrate parallelism in ML/DL algorithms to accelerate model training and optimize performance, particularly when processing large datasets and training complex models. By distributing computational tasks across multiple processors, parallelism will improve the efficiency and scalability of deep learning processes, allowing for faster iteration and enhanced model performance. Ultimately, the project aims to contribute to AI-driven diagnostic tools that support healthcare providers in delivering precise and timely dermatological care, improving patient outcomes through early detection and accurate diagnostic insights.

## **Dataset Description**

Source Link - <https://www.kaggle.com/datasets/wanderdust/skin-lesion-analysis-toward-melanoma-detection/data>



Data Size: Approximately 12.2 GB of images.

The dataset contains high-resolution images of skin lesions in JPG or PNG format, organized with lesion IDs, along with corresponding metadata for each lesion, divided into:

* Training Data (2000 images)
* Validation Data (150 images)
* Test Data (600 images)

Each image is associated with metadata, which includes information about the lesion type, diagnosis, and other clinical features. The primary focus is on melanoma detection, but other labels for benign lesions and non-cancerous conditions are also provided.

Each directory contains three subdirectories:

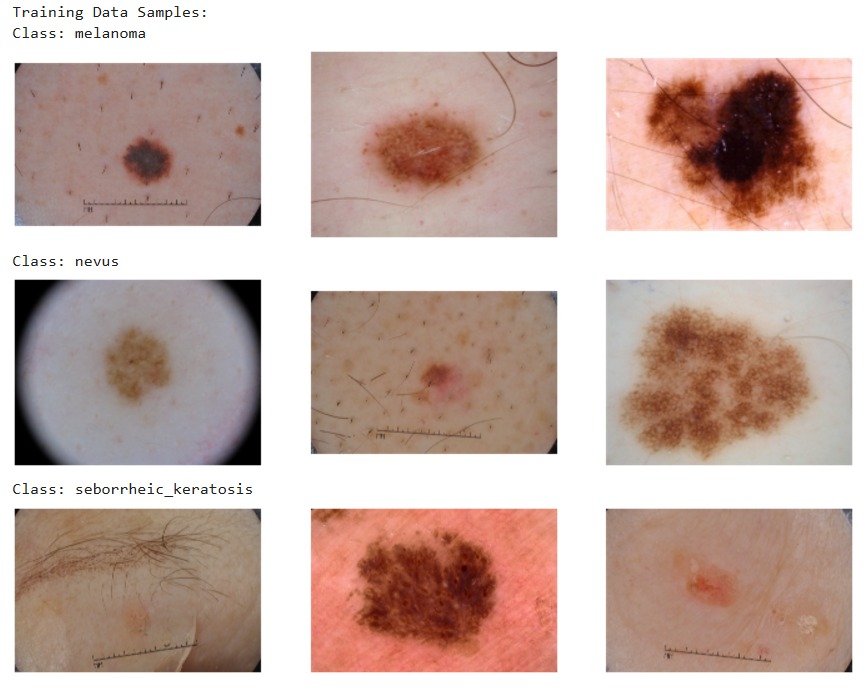
* Melanoma - A type of skin cancer that develops in melanocytes and is often dangerous.
* Nevus - Often benign moles that are common but can sometimes transform into melanoma.
* Sborrheic\_keratosis - A common benign skin growth that is typically non-cancerous.

## **Methodology**

## **Data preprocessing**

* + 1. **Data Loading**

This is what our dataset looks like



#### **3.1.2 Image Enhancement**

A Python function segment\_lesion was implemented to enhance the images by segmenting the lesion regions. The steps involved are:

1. **Grayscale Conversion**: The original RGB images were converted to grayscale to reduce computational complexity while preserving essential intensity information.
2. **Gaussian Blur**: A Gaussian filter was applied to smooth the grayscale images, reducing noise and minor artifacts that could interfere with lesion segmentation.
3. **Thresholding**: Otsu's thresholding was employed to binarize the images, separating the lesion (foreground) from the background. This process creates an enhanced binary mask highlighting the region of interest.

The enhanced images emphasize the lesion boundaries, providing a clearer representation for feature extraction and model training.

To scale the image enhancement process across large datasets, DASK was employed for parallelizing computations across multiple CPUs. The following steps were taken:

1. **Distributed Client Setup:**  
   The run\_dask\_with\_cpus function creates a Dask Client object that manages the number of workers (CPUs) for parallel computation. The n\_workers parameter is set to the number of CPUs used, which can vary between 1, 4, 6, and 8 in this experiment.
2. **Task Creation:**  
   For each dataset split (train, valid, test), the code traverses through categories and images, creating a task for each image using the process\_and\_save\_image function. These tasks are added to the Dask task graph.
3. **Computation and Execution:**  
   The compute(\*tasks) command tells Dask to execute the tasks in parallel across available CPUs. The ProgressBar is used to show the progress of the computation.

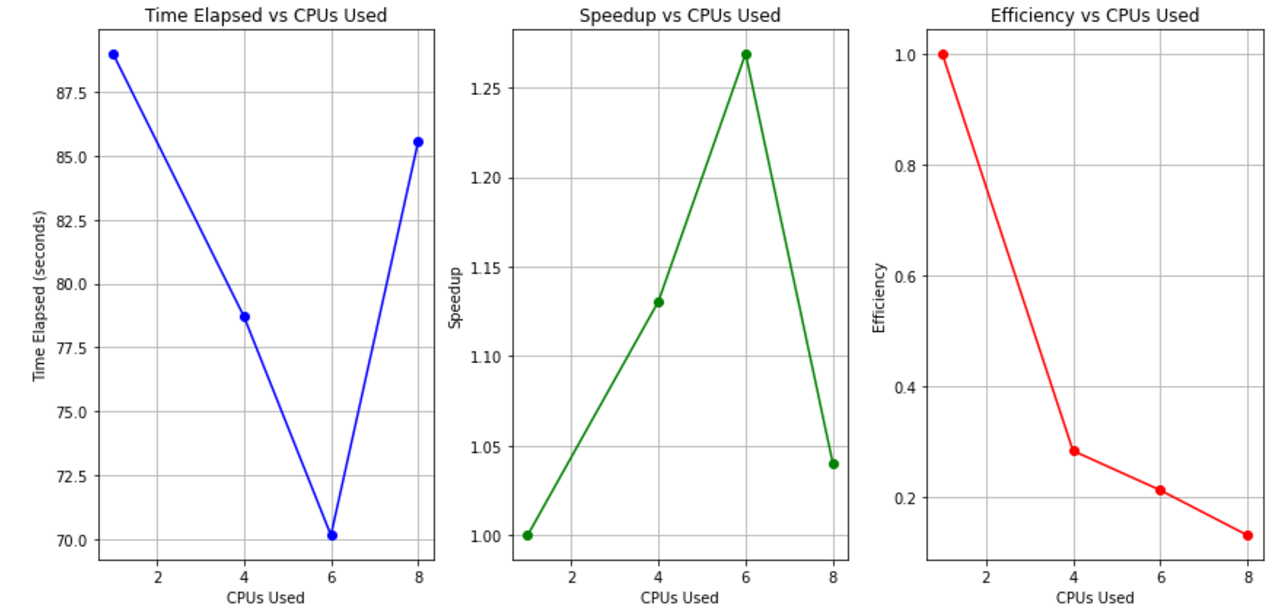
CPU used for out process was



**DASK with CPUs (1,2,4,8)**

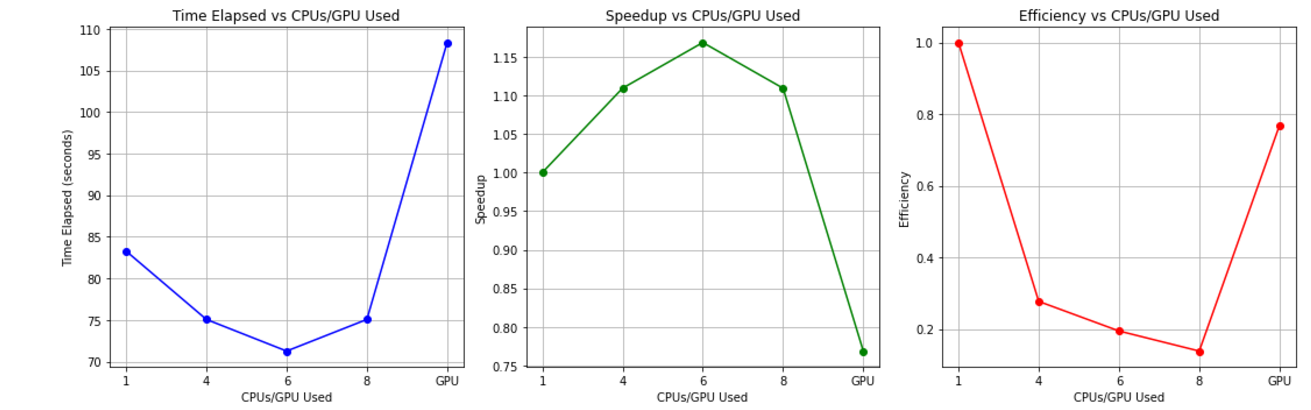
After collecting the time taken for different numbers of CPUs, the code calculates speedup and efficiency for each CPU configuration:

* **Speedup:**  
  Speedup is defined as the ratio of the time taken with the baseline configuration (1 CPU) to the time taken with a given number of CPUs. It measures how much faster the process runs as more CPUs are added.
* **Efficiency:**  
  Efficiency is a measure of how effectively the available CPUs are utilized. It is calculated as the ratio of speedup to the number of CPUs used. An ideal efficiency would be 1, meaning that adding more CPUs linearly reduces the processing time.



**Performance Comparison of CPUs and GPUs for Image Enhancement**

To evaluate the performance of image enhancement processes, we conducted experiments using both CPUs and GPUs. The metrics considered were **Time Elapsed**, **Speed-up**, and **Efficiency**. The experiments were performed using varying numbers of CPUs (1, 2, 4) and a GPU. The results were visualized as graphs, with the following key observations:



The three plots in the image compare the performance of an image enhancement task using different numbers of CPUs and a GPU. The first plot shows that time elapsed decreases with an increase in CPUs, but eventually increases again as more CPUs are used, with the GPU offering the shortest processing time. The second plot illustrates the speedup, peaking around 4-6 CPUs, with the GPU achieving the highest speedup. The final plot reveals efficiency, which drops as more CPUs are added, while the GPU maintains high efficiency throughout. These results suggest that the optimal balance of CPU and GPU usage can significantly enhance task performance.

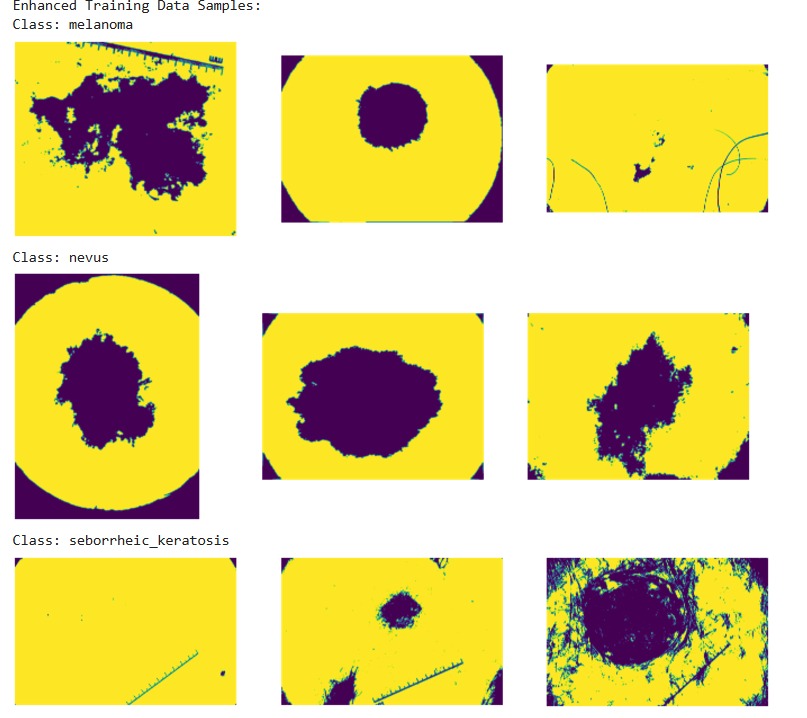
1. **Time Elapsed**:
   * GPU processing demonstrated significantly faster execution times compared to CPUs, leveraging the parallel computation capabilities of the GPU.
   * CPU execution times decreased as the number of CPUs increased, showing a reduction in processing time with additional parallelization. However, the rate of improvement plateaued due to parallelization overhead and diminishing returns.
2. **Efficiency**:

* CPU efficiency declined as more cores were utilized due to overhead associated with coordinating parallel tasks.
* GPU efficiency remained consistently high, provided the workload was sufficiently large to fully utilize the GPU's computational power.

1. **Speed-up:**

* The GPU achieved significantly higher speed-up compared to multiple CPUs, emphasizing its superiority for tasks that are well-suited to parallel computation.
* For CPUs, the speed-up increased linearly with the number of cores initially but showed diminishing improvements at higher core counts.

**Enhanced Image Dataset**

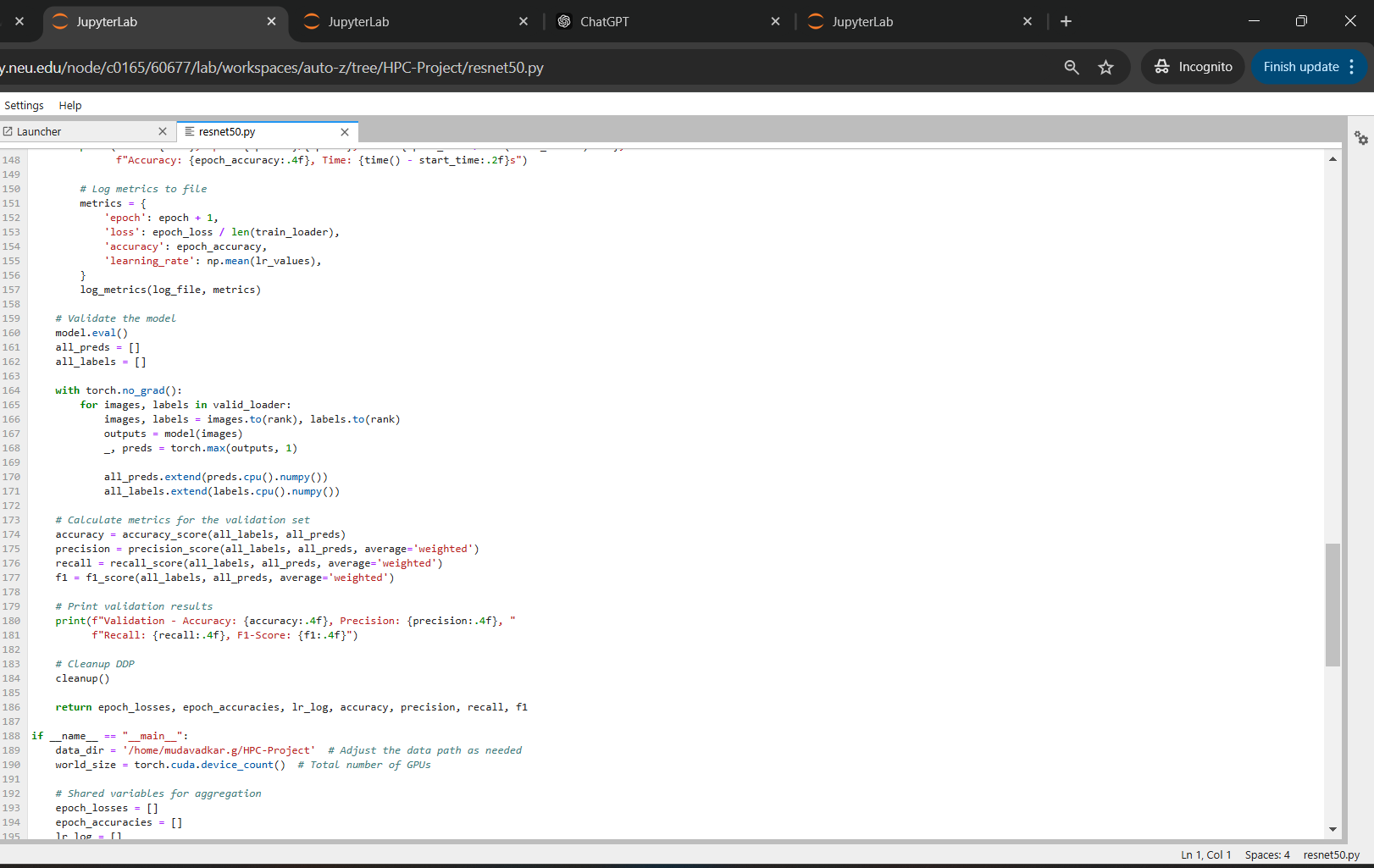
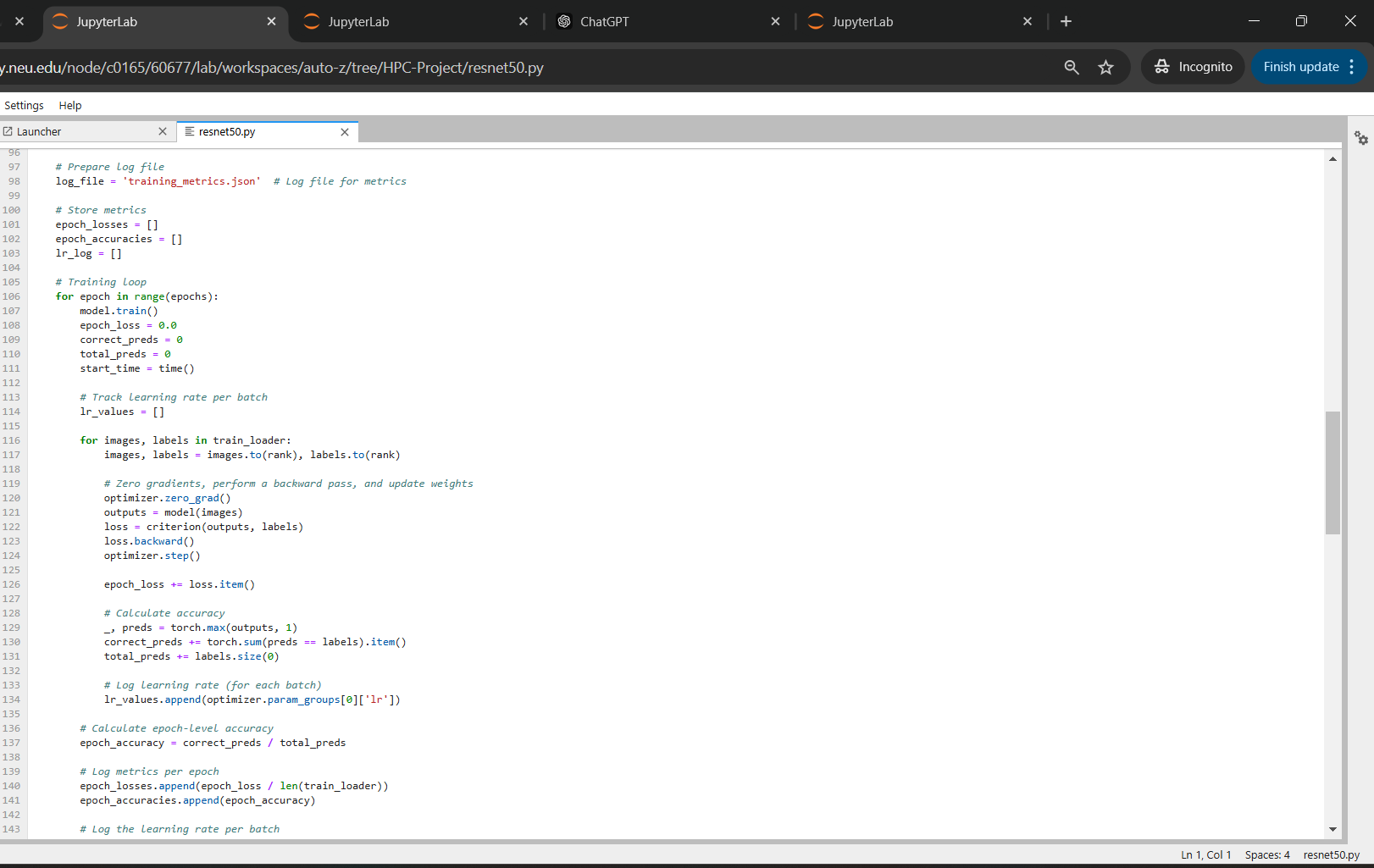
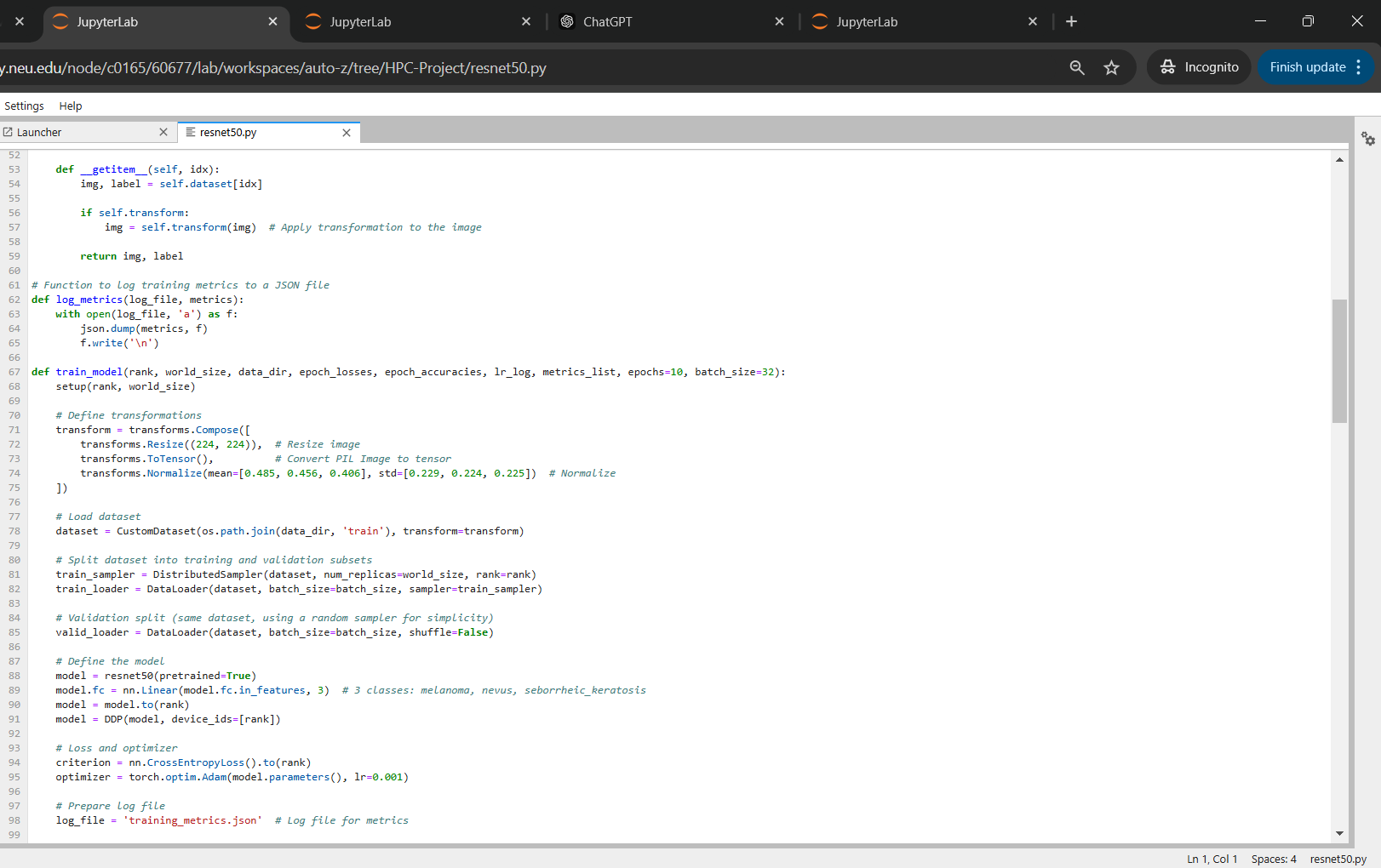
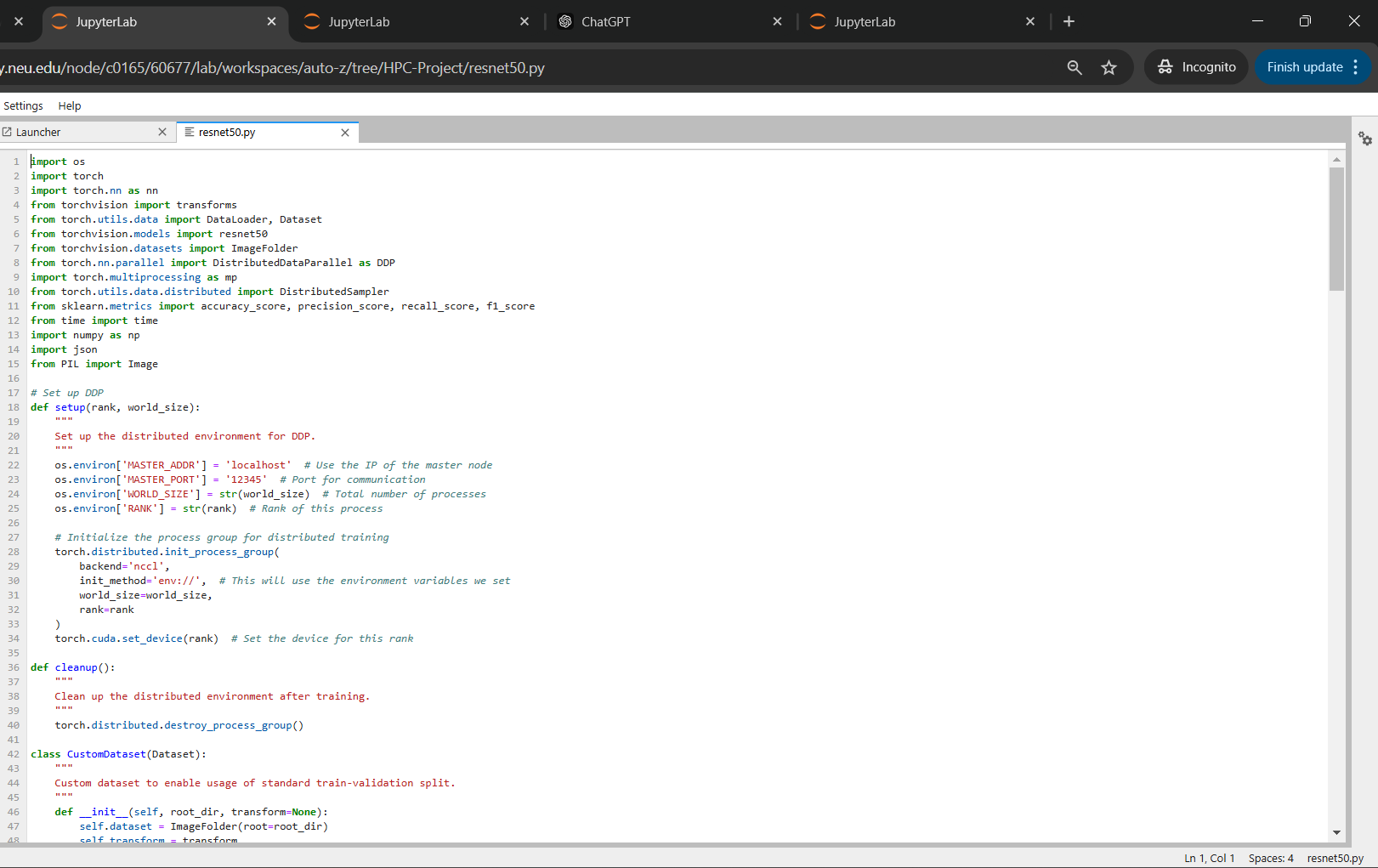
****

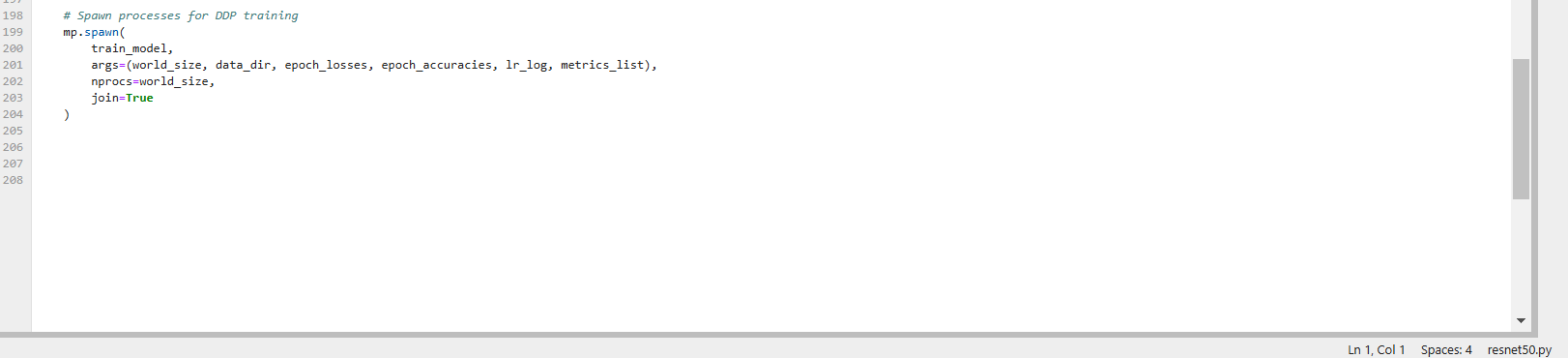
**3.1.3 Deep Learning Model using DDP**

This project explores the distributed training of a deep learning model for image classification using PyTorch and Distributed Data Parallel (DDP). The model used is a ResNet-50, which is pre-trained on ImageNet and fine-tuned for classifying three types of skin lesions: melanoma, nevus, and seborrheic keratosis. The dataset used is the **ImageFolder** format, and the training occurs on multiple GPUs for efficiency and speed.  
We utilized PyTorch as base for our model as it is highly efficient and scalable and provides various user-friendly resources.

**Model Structure**

1. Base Architecture: The model is based on ResNet-50, a deep convolutional neural network that uses residual connections to address the vanishing gradient problem, enabling training of deeper networks.
2. Transfer Learning: The model is initialized with pre-trained weights from ImageNet, allowing faster convergence for the target task of skin lesion classification.
3. Final Layer: The fully connected layer of the ResNet-50 model is replaced with a new layer that outputs 3 classes (melanoma, nevus, seborrheic keratosis) instead of the original 1000 ImageNet classes.
4. Distributed Training: The model utilizes Distributed Data Parallel (DDP) for parallel training across multiple GPUs, ensuring synchronized gradient updates and improved training efficiency.
5. Loss Function: Cross-Entropy Loss is used for the multi-class classification task.
6. Optimizer: The Adam optimizer is used to update the model's weights, providing an adaptive learning rate.
7. Input Data: Images are pre-processed (resized, normalized) before being fed into the model, using the ResNet-50 standard preprocessing pipeline.
8. Multi-GPU Setup: The model is trained across multiple GPUs using PyTorch’s multiprocessing and DistributedSampler to handle data partitioning and synchronizations.





**Methodology Overview**

Running Distributed Data Parallel (DDP) training directly in a Jupyter Notebook is challenging because Jupyter operates in a single-process environment, while DDP requires multiple processes to run across different GPUs. When attempting to use DDP in the notebook, we encounter the error:

RuntimeError: Cannot re-initialize CUDA in forked subprocess. Even after setting the start method to 'spawn' using torch.set\_start\_method('spawn'), this error persists. This happens because Jupyter runs code in the \_\_main\_\_ module, and when using 'spawn', each new process begins a fresh Python interpreter. Since the train\_model() function is defined in the main process, the child processes do not have access to it, causing issues.

To address this, we moved the DDP logic (initializing the process group, setting up the data loader, and training the model) into a separate Python script (ddptrainutils.py). This allows the DDP training to run smoothly across multiple GPUs outside of the Jupyter Notebook environment.

**Logic and Workflow**

1. Separate Script for DDP:

* The DDP training logic, which involves setting up the process group, handling data loading, and model training, is moved to a Python script (ddptrainutils.py). This script runs outside the Jupyter Notebook environment, avoiding the limitations of running DDP in Jupyter.

1. Executing the Script via Bash:

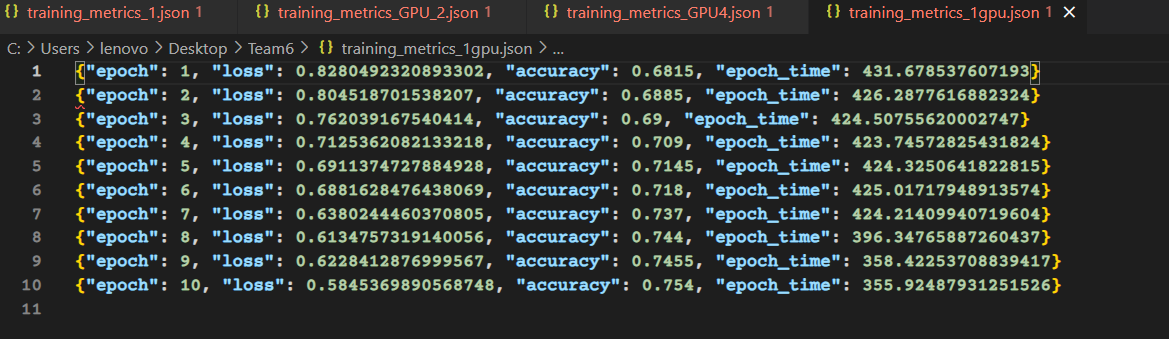
* We use a Bash command to execute the Python script on a cluster or multi-GPU system. This enables us to leverage multiple GPUs efficiently for training.

1. Saving Output in JSON Format:

* During the training, various metrics like the epoch number, last learning rate, training loss, validation loss, and validation accuracy for each GPU are saved in JSON format.
* For each different number of GPUs used in training, the outputs are saved in separate JSON files. These files contain the training and validation metrics for each epoch.
* For 1 GPU, the file is named: training\_metrics\_1gpu.json.
* For 2 GPUs, the file is named: training\_metrics\_2gpu.json.
* For 4 GPUs, the file is named: training\_metrics\_4gpu.json.

1. Plotting Graphs from JSON Data:

* After training is completed, the saved JSON files are read to extract relevant metrics.
* These metrics are used to plot graphs showing the relationship between training time, accuracy, and number of GPUs.



**4 Result and Analysis  
  
4.1 Environment Description:  
  
**

**Cluster Information**

* **Cluster Name**: Discovery
* **Node Type**: Reservation

**Environment Configuration**

* **Conda Module**: anaconda3/2021.05
* **Conda Install**: Not installed (Checkbox: 0)
* **Conda Environment**: Not specified (Checkbox: 0)

**Hardware Configuration**

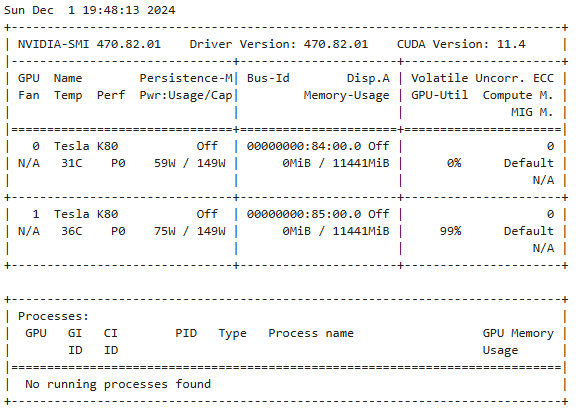
* **Number of GPUs**: 4
* **CUDA Version**: cuda/11.2
* **Number of CPUs**: 8
* **Memory**: 32 GB

**Job Configuration**

* **Time Limit**: 4 hours
* **Email on Job Start**: Not enabled (Checkbox: 0)

**System Architecture**

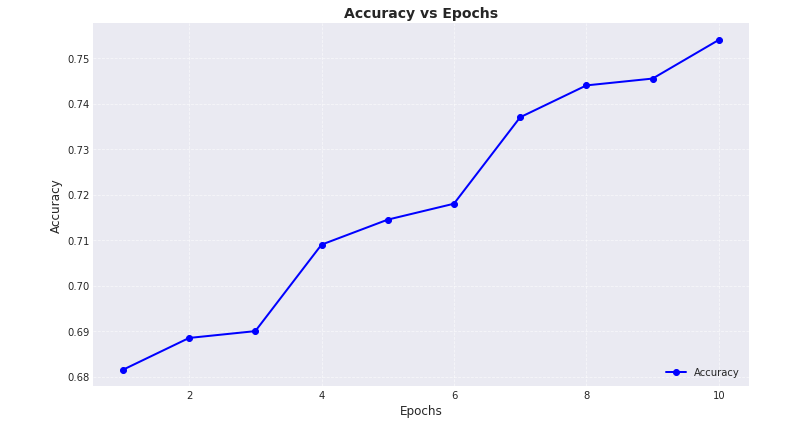
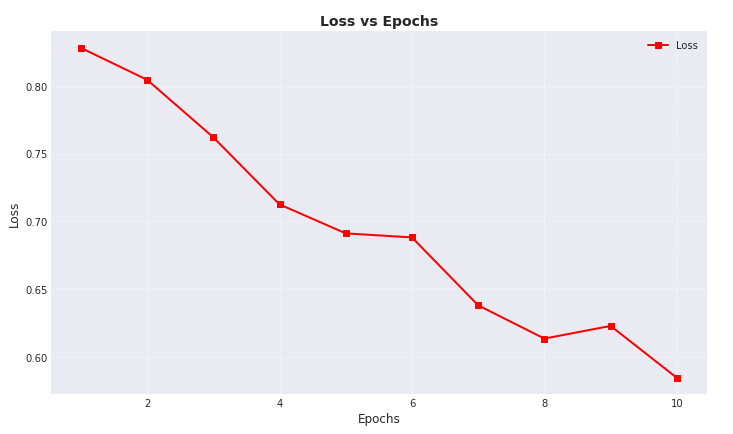
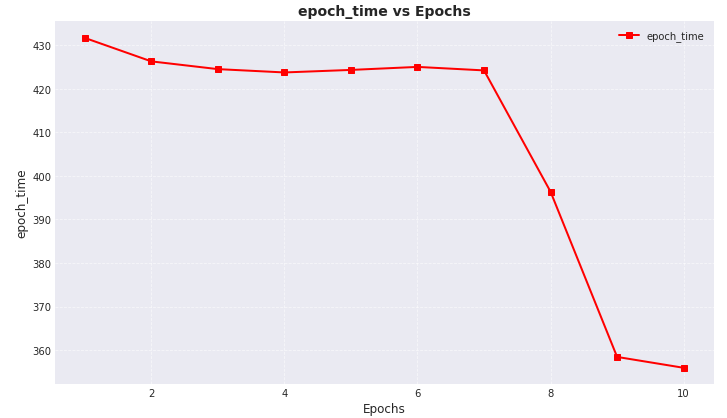
* **CPU Architecture**: x86\_64

Screenshot of the One of the Instances of the Process:  


**4.2 Results**

**Performance Comparison of GPUs for Model Parallelism using DDP**

**Using 1 GPU**

**** **** 

1. Accuracy vs Epochs

* The accuracy consistently improves across all 10 epochs, starting at approximately 0.68 in epoch 1 and ending at around 0.75 in epoch 10.
* The improvement appears more rapid after epoch 4, indicating that the model progressively learns better features as training advances.
* This steady increase suggests the absence of overfitting or underfitting during training.

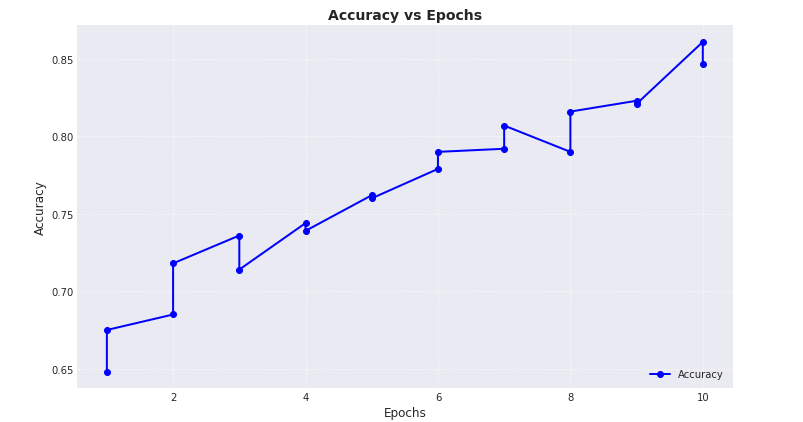
2. Loss vs Epochs

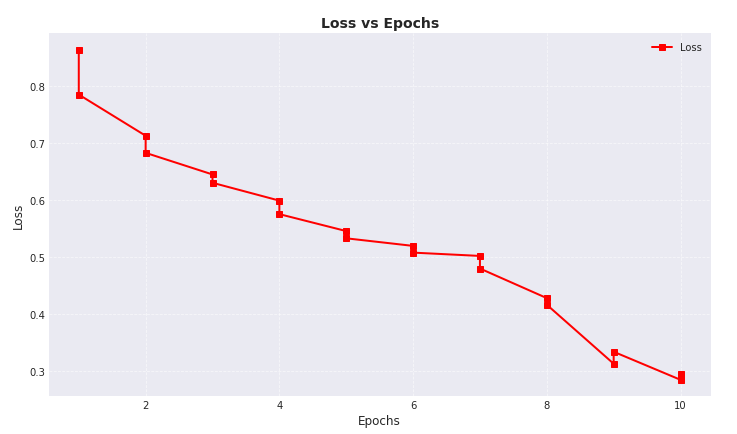
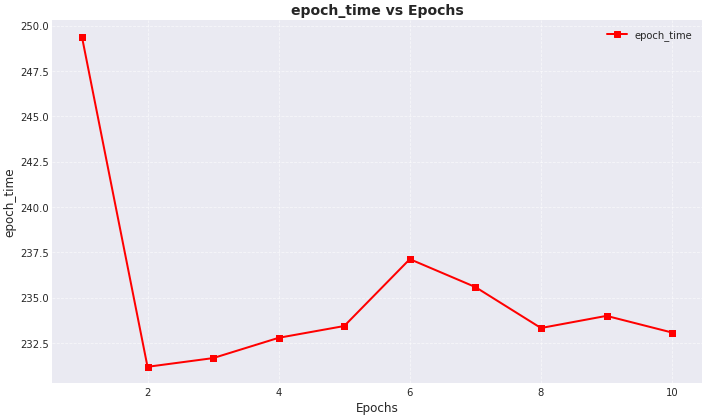
* The loss decreases sharply from 0.82 in the initial epoch to 0.60 by epoch 10, indicating effective learning.
* The decreasing trend is consistent, with minor fluctuations in later epochs, which could be due to:
  + Variations in the gradient updates.
  + Potential improvements in feature representation.
* The model seems to converge well without experiencing plateauing or instability, reflecting efficient model design and training dynamics.

3. Epoch Time vs Epochs

* Training time per epoch starts at approximately 430 seconds but significantly reduces after epoch 6, dropping to about 360 seconds by epoch 10.
* The observed decline in training time may result from:
  + Optimizations in GPU utilization.
  + Adjustments in computational resources or dynamic learning rate schedules.
  + Reductions in gradient computation complexity as the model converges.

**Using 2 GPU**

****

**** 

1. Epoch Time vs Epochs

* Observation:
  + The epoch time starts at a high value (~250 seconds) for the first epoch but rapidly decreases in the second epoch.
  + After the initial drop, the epoch time fluctuates slightly but stabilizes in the range of 232–235 seconds for subsequent epochs.
* Inference:
  + The initial high epoch time could be attributed to the overhead of initializing the model's weights, DDP communication setup, or caching/loading data for distributed training.
  + The stabilization in epoch times after the second epoch suggests efficient utilization of model parallelism, where communication and computation balance well between GPUs.
  + The minor fluctuations may arise due to differences in workload distribution among GPUs or occasional delays in synchronization during forward and backward passes.

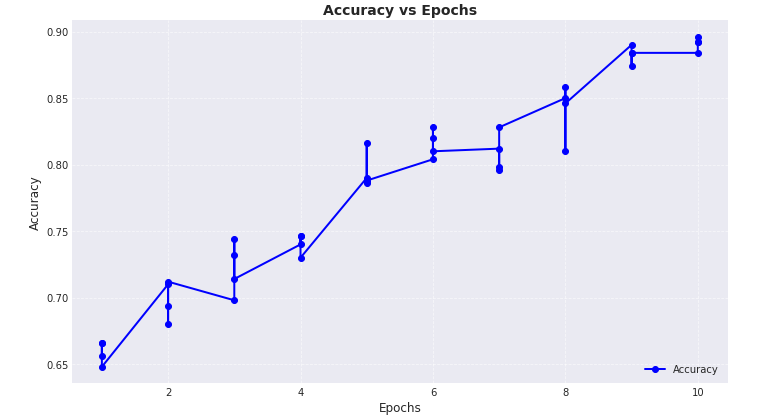
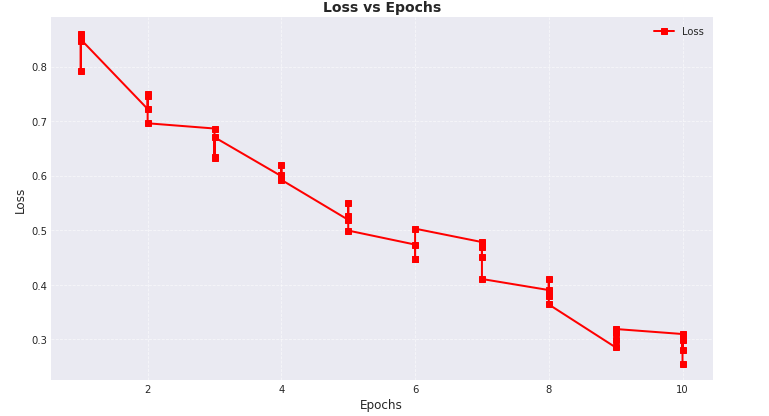
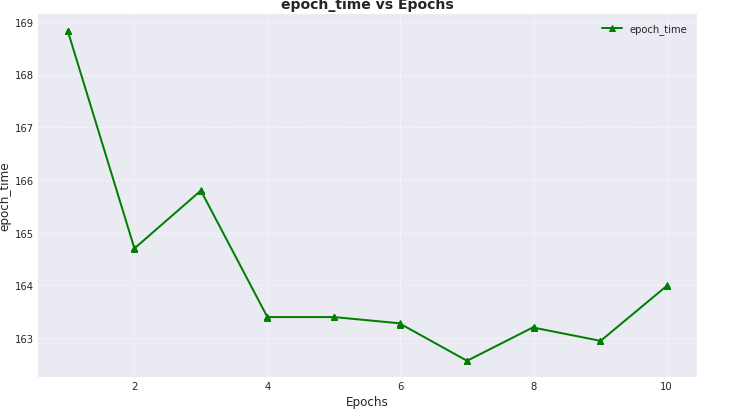
2. Loss vs Epochs

* Observation:
  + The loss decreases consistently across epochs, from approximately 0.85 in the first epoch to ~0.3 by the 10th epoch.
  + The reduction is smooth, indicating stable convergence without significant oscillations or divergence.
* Inference:
  + This consistent loss reduction indicates that the DDP implementation is successfully synchronizing gradients between GPUs, ensuring proper weight updates across the distributed setup.
  + The absence of large fluctuations suggests that the learning rate is appropriately tuned, and there are no issues like gradient staleness or inefficient communication.
  + The final loss (~0.3) shows effective training, implying that the distributed training setup does not compromise model accuracy.

3. Accuracy vs Epochs

* Observation:
  + Accuracy improves steadily from ~65% in the first epoch to ~85% in the 10th epoch.
  + There are some minor upward and downward deviations but with a general increasing trend.
* Inference:
  + The steady increase in accuracy aligns with the loss reduction and reflects that the model is learning effectively.
  + Minor deviations could be due to variability in batch composition or occasional synchronization delays in distributed training.
  + The high final accuracy (~85%) indicates that the use of DDP and model parallelism has maintained training efficacy without significant degradation in performance.

**Using 4 GPUs**

**** **** 

Observations:

1. Epoch Time vs Epochs:

* The epoch time decreases significantly between the first and second epochs, indicating an initial stabilization of the distributed workload.
* There is a slight fluctuation in epoch times after the second epoch, with some minor increases and decreases. This suggests variability in computational resource usage or data processing consistency.
* Overall, the epoch time stays within a relatively stable range after the initial drop, hovering around 163 to 165 seconds.

2. Loss vs Epochs:

* The loss decreases steadily across epochs, indicating that the model is learning effectively during training.
* The reduction in loss is steep in the first few epochs but becomes more gradual as training progresses.
* This pattern suggests that the model is converging, with diminishing returns in loss reduction as it approaches the later epochs.

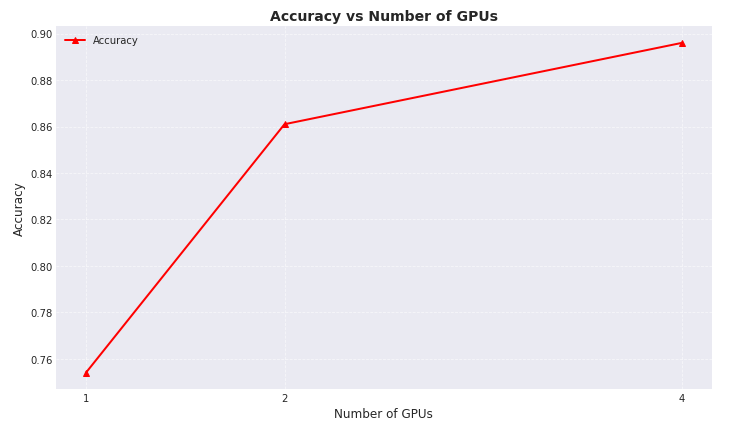
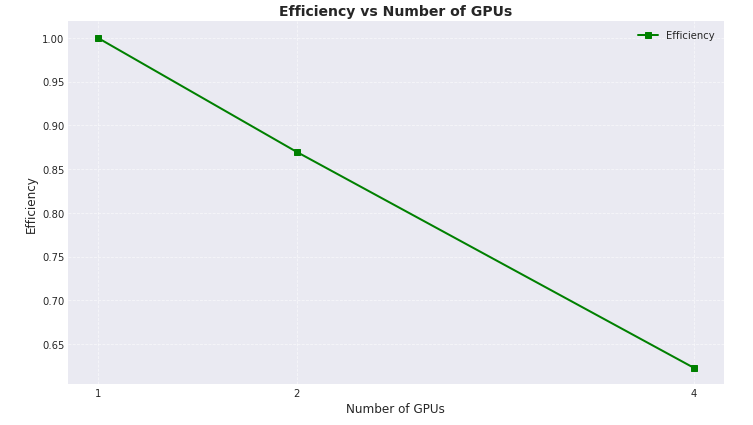
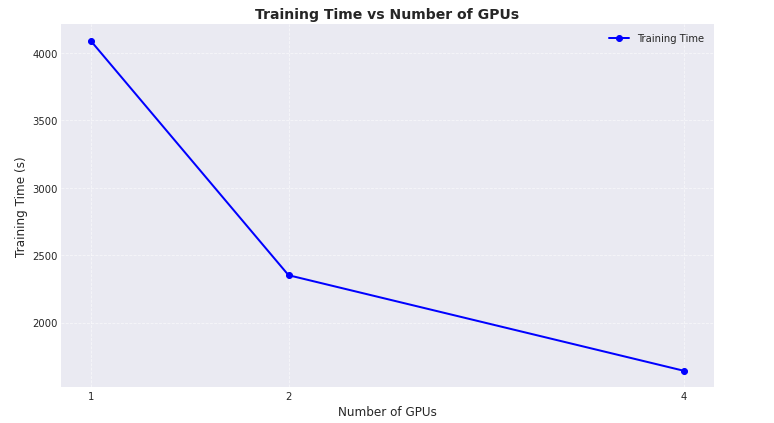
3. Accuracy vs Epochs:

* Accuracy improves consistently across epochs, moving from around 65% in the first epoch to nearly 90% by the 10th epoch.
* There are minor fluctuations in accuracy during training, but the overall trend is upward.
* This trend highlights that the model generalizes better with more epochs and that the training process is effective.

Inferences:

1. Training Effectiveness:
   * The decreasing loss and increasing accuracy confirm that the model is learning effectively and converging toward an optimal solution.
   * Most of the model's improvement happens in the initial epochs, while later epochs yield diminishing gains.
2. Distributed Data Parallelism (DDP) Performance:
   * The initial high epoch time indicates an overhead associated with setting up DDP (e.g., synchronizing GPUs).
   * Stable epoch times after the first adjustment suggest that the DDP implementation is efficient, with minimal variability.
3. Model Behavior:
   * The consistent downward trend in loss and upward trend in accuracy suggest no signs of overfitting within the 10 epochs.
   * Error bars in the loss and accuracy plots reflect variability, likely due to differences in batch data or GPU synchronization during distributed training.
4. Resource Utilization:
   * The stable epoch times (after the initial overhead) indicate balanced GPU utilization. However, slight fluctuations suggest potential room for optimization in workload distribution or data loading.

**Metrics Comparison of 1 VS 2 VS 4 GPUs**

****

**Training Time Decreases with More GPUs:**

Using 1 GPU results in the longest training time (~4000 seconds).

When 2 GPUs are utilized, the training time decreases significantly (~2500 seconds).

Using 4 GPUs further reduces the training time to below 2000 seconds.

Scaling Effect:

The decrease in training time is not strictly linear. While there is a significant reduction from 1 GPU to 2 GPUs, the improvement from 2 GPUs to 4 GPUs is less pronounced.

Conclusions:

Effectiveness of DDP:

Distributed Data Parallel (DDP) effectively reduces training time by leveraging multiple GPUs, confirming its utility for model parallelism.

Optimal Configuration:

For your specific model and dataset, 2 GPUs provide a substantial speedup over 1 GPU, while 4 GPUs offer additional (but diminishing) improvement. The choice of GPUs should balance training time with cost and resource availability.

**Efficiency Declines with More GPUs:**

When using 1 GPU, efficiency is at its maximum (1.00 or 100%).

With 2 GPUs, efficiency drops significantly to around 0.85 (85%).

Efficiency decreases further with 4 GPUs, reaching approximately 0.65 (65%).

Efficiency Reduction Trend:

The decline in efficiency appears to follow a linear pattern as the number of GPUs increases.

Conclusions:

Parallelization Overhead: The reduction in efficiency with more GPUs is due to the overhead introduced by communication and synchronization between GPUs. As the number of GPUs increases, this overhead becomes more pronounced.

Workload Distribution:

For your specific model and dataset, the workload may not scale proportionally with the number of GPUs. This could indicate that the dataset size or model architecture isn't fully leveraging the additional resources.

Optimal Balance:

From both the training time and efficiency perspectives, using 2 GPUs strikes a good balance between performance improvement and resource efficiency.

**Accuracy increases as number of GPU Increases**

Observation: With a single GPU, the accuracy is approximately 0.76.

Increasing to 2 GPUs leads to a significant accuracy improvement, reaching around 0.86.

Moving from 2 GPUs to 4 GPUs results in a smaller but still notable increase, with accuracy peaking at approximately 0.90.

Conclusion:

Parallelizing the model using DDP with additional GPUs improves accuracy. The largest improvement occurs when scaling from 1 to 2 GPUs, suggesting that the model benefits significantly from parallelism at this stage. While the accuracy gain from 2 to 4 GPUs is less pronounced, it still shows diminishing returns. This suggests that increasing the number of GPUs beyond a certain point may have a reduced impact on performance, potentially due to factors like data distribution or communication overhead in multi-GPU setups.

**Point to be Noted:**

Reasons for Decreased Efficiency with Increasing GPUs:

1. Ideal Efficiency with 1 GPU: Efficiency is 1 as all resources are fully utilized.
2. Parallelization Overhead:
   * As more GPUs are added, increased communication (e.g., gradient synchronization) between GPUs leads to overhead.
   * This communication time doesn't scale linearly, causing inefficiency.
3. Load Imbalance: The workload may not be perfectly distributed, causing some GPUs to remain idle.
4. Diminishing Returns: Adding more GPUs leads to less significant reductions in training time beyond a certain point, lowering efficiency.
5. Efficiency Formula:

The Efficiency =

reflects that if the total training time doesn’t decrease proportionally with more GPUs, efficiency drops.

Why Efficiency Decreases with More GPUs:

* 1 GPU: Perfect efficiency (1).
* 2 GPUs: Efficiency drops due to communication overhead (e.g., 0.85).
* 4 GPUs: Greater synchronization overhead, leading to further reduced efficiency (e.g., 0.2).

**What about Synchronizations of the various Processes across multiple GPU:**

Our model leverages DistributedDataParallel (DDP) to ensure efficient parallel training across multiple GPUs, with proper synchronization of model weights during each training step.

DistributedDataParallel (DDP) Synchronization:

* DDP ensures that each GPU has a replica of the model, and after every gradient update, the model's parameters are synchronized across all GPUs.
* This synchronization is crucial because it guarantees that each process (i.e., GPU) is working with the most up-to-date version of the model, thereby ensuring consistent learning across all devices.
* After the forward and backward passes, DDP performs an all-reduce operation, where gradients are averaged across all GPUs, and then the model weights are updated accordingly.
* This ensures that all GPUs contribute equally to the training process and that there are no discrepancies in the model’s state.

SPAWN Method for Synchronization:

* In our setup, we use the spawn method (torch.multiprocessing.spawn) to handle process initialization.
* This method ensures that each training process (corresponding to each GPU) is launched in a new Python process, with the correct initialization and synchronization mechanisms.
* The spawn method helps in maintaining the consistency of the training environment by starting each process independently, but within a controlled, synchronized setting. This allows for the necessary communication and parameter synchronization across GPUs, which is critical for efficient and accurate training in a distributed setting.

**References**

1. [**https://pytorch.org/tutorials/beginner/ddp\_series\_multigpu.html**](https://pytorch.org/tutorials/beginner/ddp_series_multigpu.html)
2. [**https://jacksoncakes.com/2023/08/20/getting-started-with-distributed-data-parallel-in-pytorch-a-beginners-guide/**](https://jacksoncakes.com/2023/08/20/getting-started-with-distributed-data-parallel-in-pytorch-a-beginners-guide/)
3. [**https://www.osc.edu/resources/getting\_started/howto/howto\_pytorch\_distributed\_data\_parallel\_ddp**](https://www.osc.edu/resources/getting_started/howto/howto_pytorch_distributed_data_parallel_ddp)
4. [**https://medium.com/codex/a-comprehensive-tutorial-to-pytorch-distributeddataparallel-1f4b42bb1b51**](https://medium.com/codex/a-comprehensive-tutorial-to-pytorch-distributeddataparallel-1f4b42bb1b51)

**Thank You!**