DSE603 Applied Accelerated Artificial Intelligence

Assignment Report

Anurag Nihal

Saswata Sarkar

Department of Data Science and Engineering

Indian Institute of Science Education and Research, Bhopal

September 21, 2023

Task 1: Dockerized Face Detection Model

In the modern software era, Docker is like a super tool that changes how to make, send, and use computer applications. Imagine it as a special box (a container) that holds everything an app needs to work, like the codes, libraries, environment, and dependencies. Docker provides a consistent and efficient way to create, deploy, and manage these containers, making it an indispensable tool in software development.

Why do we need Docker? Well, before, sending apps was tricky. Sometimes, an application works on one computer but not on another because they have different setups and environments. Docker fixes this by putting the app and all its needs in a container. So, what you make and test on your computer will work the same way on any computer or server. Docker makes everything simpler and stops compatibility problems. You can create a custom Docker image for your Machine Learning and Deep Learning project; you can train it, test it, and deploy it on any machine you want.

Docker offers a plethora of advantages that have made it the go-to choice for many organizations and developers:

1. **Portability**: Docker containers are highly portable. You can run the same container on different environments without worrying about compatibility issues. This portability is especially beneficial in today's multi-cloud and hybrid-cloud infrastructures.

2. **Isolation**: Each container and the host system are isolated, ensuring that changes or issues in one container do not affect others. This isolation enhances security and stability.

3. **Efficiency**: Docker containers are lightweight, consuming fewer resources than traditional virtual machines, allowing you to run more containers on a single server, leading to cost savings and better resource utilization.

4. **Scalability**: Docker makes scaling applications up or down easy by adding or removing containers as needed. This elasticity is crucial for handling varying workloads and traffic spikes.

5. **Version Control**: Docker enables version control for both application code and dependencies. One can version containers, ensuring that one can always return to a previous version if issues arise.

6. **Ecosystem**: Docker has a vibrant ecosystem with a vast repository of pre-built images, making setting up and configuring various software components quick and easy.

In summary, Docker is a game-changing technology that simplifies software development, improves consistency, and enhances scalability. Whether you are a developer, system administrator, ML enthusiast, or anyone involved in the software development process, Docker is a tool you will want to become familiar with to streamline your workflow and deliver robust applications.

In the first task, we needed to Dockerize the model Ultra-Light-Fast-Generic-Face-Detector-1MB. This model is a lightweight face detection model designed for edge computing devices. The model is designed to detect faces both from images and videos. We create a **DockerFile** and use that to create a custom environment in which the Deep Learning model works.

DockerFile

A Dockerfile serves as a script-like blueprint, encompassing a series of essential instructions for constructing a Docker container image. Within this Dockerfile, we meticulously define the desired environment, essential dependencies, and the precise configuration required for a particular application. The Dockerfile allows us to orchestrate a meticulously tailored environment for our application, employing a set of imperative commands to accomplish this task. We have created the following DockerFile for our face detection application.

```
FROM quangbd/dc-pytorch:1.6.0-python3.6-cuda11.0-cudnn8-ubuntu18.04
ARG DEBIAN_FRONTEND=noninteractive
RUN apt-get update
RUN apt-get update && DEBIAN_FRONTEND=noninteractive apt-get install\
    libgl1\
    libgl1-mesa-glx \
    libglib2.0-0 -y

RUN apt update && apt install -y libsm6 libxext6
RUN apt-get install -y libxrender-dev

WORKDIR ./
COPY . ./

RUN python3.6 -m pip install -r requirements.txt
RUN python3.6 -m pip install opencv-python==4.0.0.21
```

Here is the breakdown of the DockerFile:

FROM: We start by importing a base image with `FROM` command that provides the initial environment for our application. It has the following layers of Ubuntu 18.04, Python 3.6, PyTorch 1.6.0, and CUDA 11.0, which are apt for our face detection deep learning model.

RUN: We use the `RUN` command to execute instructions within the container. First, we update the Ubuntu 18.04 system in the container. Then, we installed essential packages like libgl1, libgl1-mesa-glx, and libglib2.0-0 to fix glitches related to opencv-python. Additionally, we installed libsm6 and libxext6 to support the previously installed libraries.

WORKDIR: We create a working directory using `WORKDIR` command inside the container where our application (model) will reside.

COPY: We copy all the files and directories from our local machine's repository to the working directory within the Docker container using the `COPY` command.

Finally, we install all the necessary packages and opencv-python within the container. This DockerFile creates a clean and controlled environment image for our deep learning model, ensuring it has all the required dependencies and packages to run without glitches.

We eventually build the custom Docker image for our deep learning model with the previously created DockerFile using the following bash command:

```
docker build . -f DockerFile --no-cache -t anurag_saswata
```

In this command:

- `docker build` constructs a Docker image based on the instructions specified in the `DockerFile`.
- `-f DockerFile` flag specifies the name of the `DockerFile` to be used for the build. It is worth noting that Docker is case-sensitive, so the filename must match precisely, including the case.
- `-no-cache` flag ensures that the image is built from scratch without utilizing any cached layers from previous builds.
- The `-t anurag_saswata` portion of the command assigns a name and optional tag to the Docker image being built. In this instance, the image is named `anurag_saswata`. This name can be used later to reference the image when running containers.

Subsequently, we employ the `docker run` command to utilize the custom image for our deep learning model:

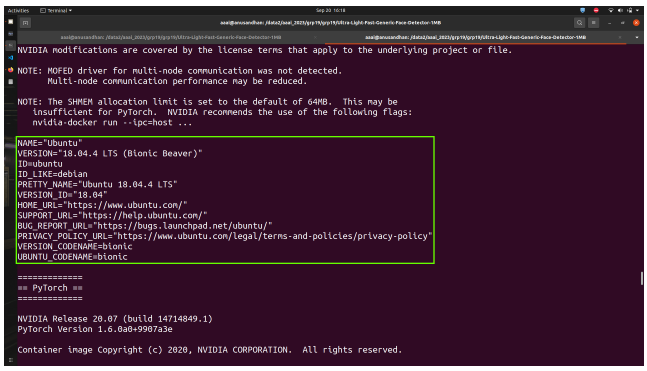
```
docker run -it --gpus all anurag_saswata python3 detect_imgs.py
```

Here's a breakdown of this command:

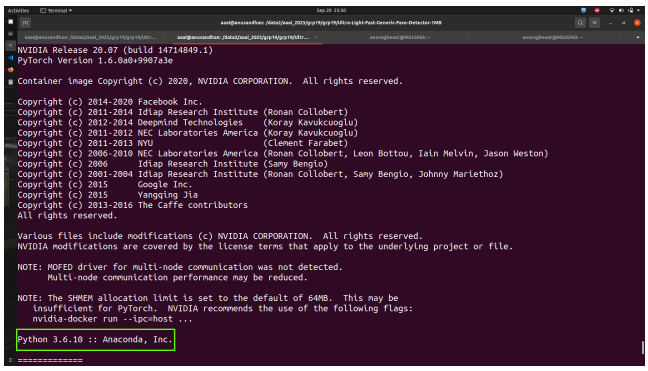
- The `-it` flags stand for `-interactive` and `-tty` (teletypewriter) and are employed to open an interactive terminal within the container, enabling interaction with the container's command-line interface.
- The `--gpus` flag indicates the use of the host machine's GPU.
- The command `python3 detect_imgs.py` is executed within our custom image.

Results and interpretation

In this section, we will display the command line outputs followed by the outputs of the Single Shot MultiBox Detector (SSD) model for 640 input size present in this repository. This model is a lightweight face-detection model designed for edge computing devices. Regarding model size, the default FP32 precision (.pth) file size is 1.04 $\sim$ 1.1MB.



(a) OS Information of Docker Image



(b) Python Version of Docker Image

The Figure above shows the OS information from our custom Docker image. It has `Ubuntu:18.04` and `PyTorch 1.6` (see Figure 1a and 1b) as mentioned in the `DockerFile`. Figure 2 displays the Python packages listed in the Docker container with the command `pip show`.

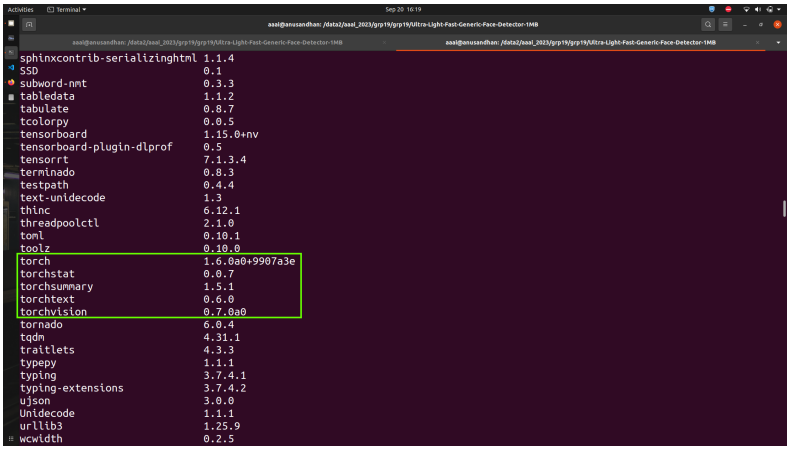
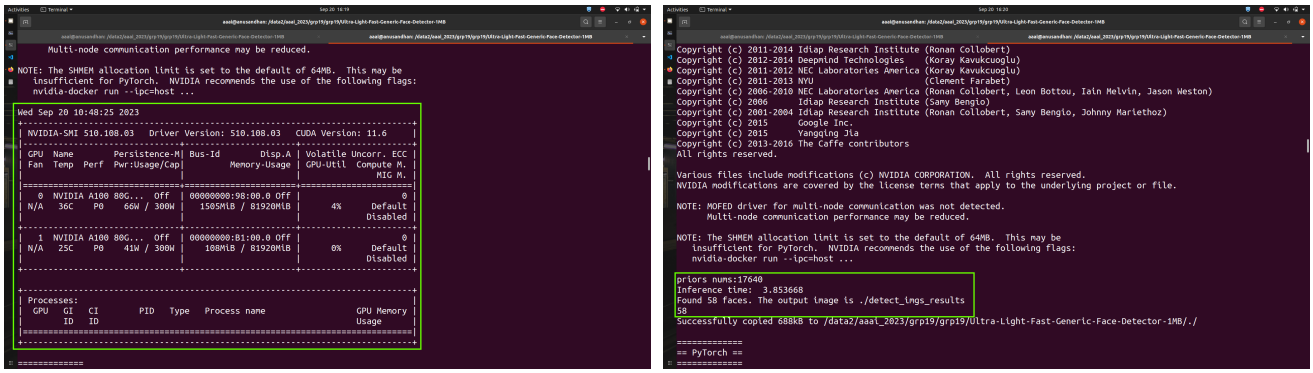


Figure 2: List of Python packages in Docker

Figure 3a illustrates the output of the `nvidia-smi` command within the Docker container, revealing its access to the host machine's GPU from the isolated Docker container. The face detection model processes the image 5a within the Docker container, accurately detecting a total of 58 faces in the image and saving the result as the `/detect_imgs_results` file (see figure 3b. The detected faces are highlighted with a red bounding box as shown in figure 5b.



(a) nvidia-smi output inside Docker (b) Face detection in Docker

Figure 3: nvidia-smi output and face detection output

Figure 4 illustrates the effective operation of the face detection model within the Docker container, specifically showcasing its successful application to video processing. Furthermore, this visualization underscores the seamless functionality of the model as it consistently identifies and stores the processed video frames in the directory `/Detected_images` throughout the process.

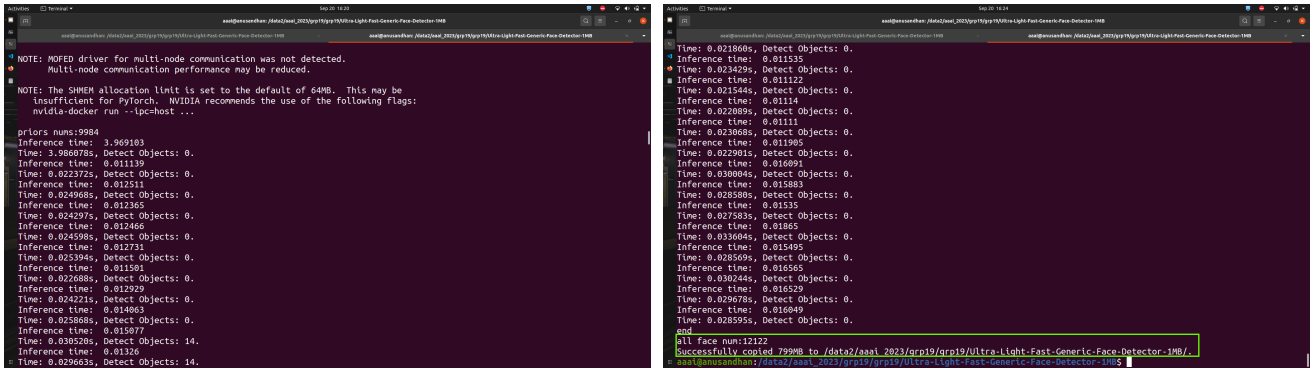
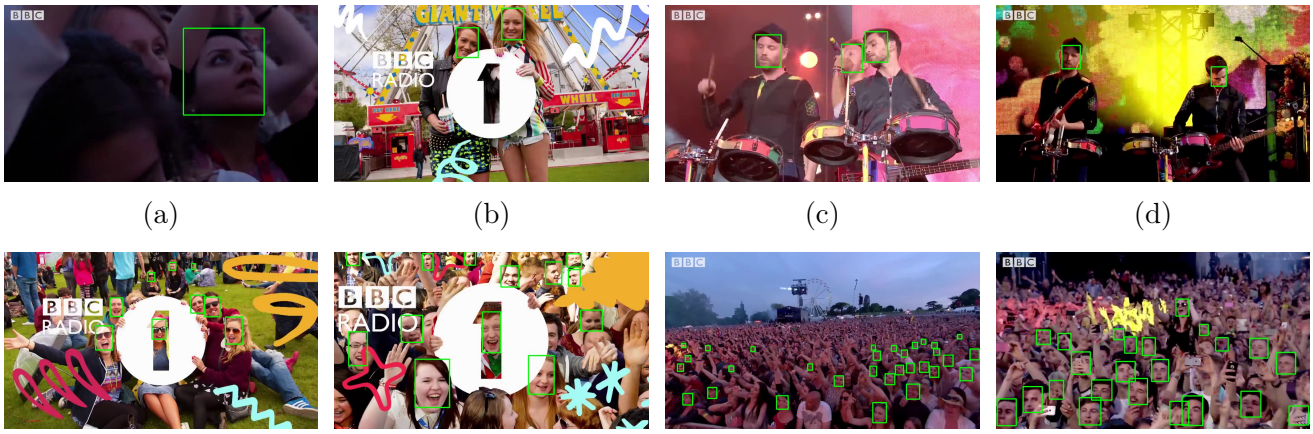


Figure 4: Face detection with video output

(a) Original image (b) Processed image

Figure 5: Comparison of the original and processed image in Docker

In addition to the above, the model showcases its versatility by seamlessly extending its face-detection capabilities to video files. This functionality is encapsulated within the Python script `run_video_face_detect.py`. The script initiates by extracting frames from the video source, followed by the meticulous processing of these frames to accurately detect faces within each frame. The outputs of different frames using the provided input video are presented in Figure 6 with green boxes representing the bounding boxes.



(a) (b) (c) (d)

(e) (f) (g) (h)

Figure 6: Processed frames from the video

Task 2: Matrix Multiplication using multi-threading

The following is the naive $O(N^3)$ optimized function to calculate matrix multiplication. This function is responsible for multiplying a specific block of rows from `matrix_a` by the entire `matrix_b`.

```
def multiply_matrices(start_row, end_row, matrix_a, matrix_b):
  """
  Args:
    start_row: The start row index of the block to multiply.
    end_row: The end row index of the block to multiply.
    matrix_a: The first matrix.
    matrix_b: The second matrix.

  Returns:
    The product of the block and the second matrix.
  """

  C = np.zeros_like(matrix_b)
  for i in range(start_row, end_row):
    for j in range(matrix_b.shape[0]):
      C[i, j] = np.sum(matrix_a[i, :] * matrix_b[:,j].T)

  return C
```

The overall code is present in this Google Colab. We use the important excerpts related to the task objectives.

`num_threads = multiprocessing.cpu_count()` - Determines the number of CPU cores available on the system and assigns it to the variable `num_threads`. `workers = matrix_size // num_threads if matrix_size // num_threads > 1 else 2`: Calculates the number of worker processes to use for parallel computation. It divides the matrix size by the number of CPU cores but ensures that at least 2 workers are used. `pool = multiprocessing.Pool(workers)`: Creates a pool of worker processes for parallel computation. The following loop divides the work into smaller tasks by splitting the rows of `matrix_a` into blocks of rows. Each task is defined by a tuple `(start_row, end_row, matrix_a, matrix_b)` and represents a block of rows to be multiplied. `results = [pool.apply_async(multiply_matrices, args=task) for task in tasks]`: Starts the worker processes asynchronously, with each process executing the `multiply_matrices` function with its assigned task as an argument. It returns a list of results. This sets up and starts multiple worker processes to perform matrix multiplication concurrently. It assigns each worker to process a specific task, and the results of these tasks are collected into the results list. The program can take full advantage of available CPU cores and speed up the matrix multiplication process by doing this asynchronously.

```
    num_threads = multiprocessing.cpu_count()
    # Create a pool of worker processes.
    workers = matrix_size//num_threads if matrix_size//num_threads>1 else 2
    print(workers)
    pool = multiprocessing.Pool(workers)

    # Divide the workload into smaller tasks.
    tasks = []
    for i in range(0, matrix_a.shape[0], workers):
      start_row = i
      end_row = min(i + (workers), matrix_a.shape[0])
      task = (start_row, end_row, matrix_a, matrix_b)
      tasks.append(task)

    # Start the worker processes and get the results asynchronously.
    results = [pool.apply_async(multiply_matrices, args=task) for task in tasks]

    # Combine the results from the worker processes.
    C = np.zeros_like(matrix_a)
    for result in results:
      print("\n result.get()",result.get())
      C += result.get()
```