DI SS 2023
Task 3

Vladimir Panin (12238686) Besnik Cani (12243760)
Andrei Florescu (12242987)

July 2023

1 Introduction

In this assignment, a data processing application is developed using Docker, Python, and TensorFlow, with a focus on computation and data offloading. The application is deployed and tested on AWS for efficient computation and scalability. The task provided constists of multiple subtasks. First, an object detection functionality has to be implemented in the provided app.py file, with input as images and output as JSON responses. Then, the application is dockerized, using the provided Dockerfile. Finally, the application is executed locally and remotely, collecting data on average inference time for object recognition on the provided datasets.

2 Problem Overview

The provided code includes a Dockerfile and an app.py file. The Dockerfile specifies the base image as Python 3.7.9 slim-buster and installs various dependencies using apt-get. It also sets up pi wheels for additional package installation. The app.py file contains the implementation of a Flask application for object detection.

The app.py file imports necessary libraries such as PIL, OpenCV, Flask, and others. It defines a `detection_loop` function, which is currently incomplete and marked as a TODO. The `detection_loop` function is expected to process images for object detection and return the results in a specific format.

The Flask application is initialized, and the main route `'/api/detect'` is defined to handle POST and GET requests. Within this route, the JSON data from the request is extracted, including an array of images. The code then prepares the images for object detection, converting them to the appropriate format. However, the actual object detection part is missing and needs to be implemented.

The app.py file also includes a main block that runs the Flask application with debug mode enabled on host `'0.0.0.0'`.

Overall, the code aims to create a Dockerized Flask application for object detection. The current focus is on implementing the object detection logic within the `detection_loop` function and completing the necessary steps to process the images for detection.

3 Methodology and Approach

3.1 Implementation of data processing application

The code consists of two files: `app.py` and `request.py`. The approach involves using a Flask application to perform object detection on images.

In `app.py`, the necessary libraries are imported, including Flask, TensorFlow, and TensorFlow Hub. The Flask application is initialized, and a function called `detection_loop` is defined. This function takes a detector model and a list of images as input. It iterates over each image, decodes it from base64, preprocesses it, and performs object detection using the detector model. The resulting object classes are extracted and stored along with the corresponding image names in a list. Finally, the list is converted to JSON format and returned.

The main route `'/api/detect'` is defined to handle POST and GET requests. Inside this route, the JSON payload containing the images and their names is extracted from the request. The detector model, obtained from TensorFlow Hub, is loaded. The `detection_loop` function is then called with the detector model and the images, and the resulting JSON data is returned as the response.

In `request.py`, the necessary libraries and functions for sending HTTP requests are imported. The script reads all the image files in a specified folder and converts each image to base64 format. The base64 strings and corresponding image names are stored in separate lists. A JSON payload is created using the lists, and a POST request is sent to the `'/api/detect'` route of the Flask application running locally. The JSON response is received and printed for further processing.

Overall, the approach involves setting up a Flask application to handle object detection requests. The `app.py` file defines the detection logic using a TensorFlow Hub model, while the `request.py` file demonstrates how to send image data to the Flask application for detection and retrieve the results.

3.2 Dockerization of application

In order to complete the dockerization of this application we need to create a Dockerfile with the appropriate instructions. Also we need to create a `requirements.txt` file that contains every library that is used by the app.

In the Dockerfile, firstly we need to use the official docker python image and the run a linux update command. Then we copy the current working folder into a new folder on the environment called `app`. We also make sure the `requirements.txt` file is copied into the app folder. Then we set the current working directory to the `app` folder we just created and then we run a `pip install` command to install all the requirements specified in the text file. Lastly, we add a command to run the `app.py` file which contains the main loop of the application.

To make the dockerizing easier, two shell files have been created: `docker_build.sh` shell file and the `docker_run.sh` file. In the first shell file we add the `docker build` command for the app with the name `dic-assignment` and for the second file we add the `docker run` command that runs the dic-assignment app on the port 5000.

3.3 Deployment on Amazon Web Services

We determined we were going to use EC2 as our solution for AWS deployment. The server is running on Amazon Linux OS, whereas performance wise, it has

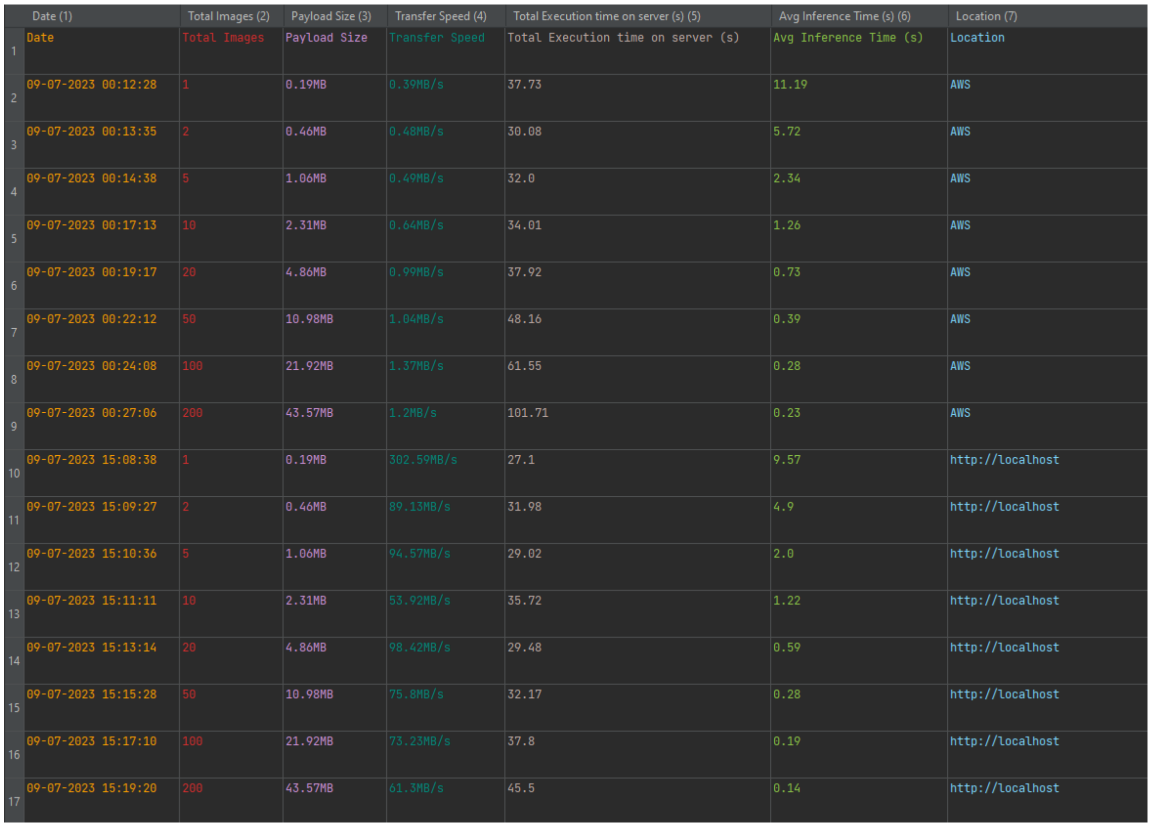
- 8GB of RAM,
- Intel(R) Xeon(R) CPU E5-2686 v4 @ 2.30GHz (2CPUs)
- 50GBs of SSD storage.

We have *sudo* permissions on the instance, which made it easy to do everything. Docker needed to be installed, after that we uploaded the code we had written and launched our application to run on the server. The API was running on port 5000, so the port needed to be opened manually in the security settings of the EC2 instance in order for incoming HTTP connections to be possible.

4 Results

We gathered some information from the executions that we run in order to draw our conclusions. The results were stored in a csv file and can be observed in Figure 1. The tests were run with: 1, 2, 5, 10, 20, 50, 100 and 200 images in order to determine how the parameters vary based on the amount of data to be processed. The results contain the following columns:

- **Date**: In order to keep track of when the executions were ran.



	Date (1)	Total Images (2)	Payload Size (3)	Transfer Speed (4)	Total Execution time on server (s) (5)	Avg Inference Time (s) (6)	Location (7)
1	Date	Total Images	Payload Size	Transfer Speed	Total Execution time on server (s)	Avg Inference Time (s)	Location
2	09-07-2023 00:12:28	1	0.19MB	0.39MB/s	37.73	11.19	AWS
3	09-07-2023 00:13:35	2	0.46MB	0.48MB/s	30.08	5.72	AWS
4	09-07-2023 00:14:38	5	1.06MB	0.49MB/s	32.0	2.34	AWS
5	09-07-2023 00:17:13	10	2.31MB	0.64MB/s	34.01	1.26	AWS
6	09-07-2023 00:19:17	20	4.86MB	0.99MB/s	37.92	0.73	AWS
7	09-07-2023 00:22:12	50	10.98MB	1.04MB/s	48.16	0.39	AWS
8	09-07-2023 00:24:08	100	21.92MB	1.37MB/s	61.55	0.28	AWS
9	09-07-2023 00:27:06	200	43.57MB	1.2MB/s	101.71	0.23	AWS
10	09-07-2023 15:08:38	1	0.19MB	302.59MB/s	27.1	9.57	http://localhost
11	09-07-2023 15:09:27	2	0.46MB	89.13MB/s	31.98	4.9	http://localhost
12	09-07-2023 15:10:36	5	1.06MB	94.57MB/s	29.02	2.0	http://localhost
13	09-07-2023 15:11:11	10	2.31MB	53.92MB/s	35.72	1.22	http://localhost
14	09-07-2023 15:13:14	20	4.86MB	98.42MB/s	29.48	0.59	http://localhost
15	09-07-2023 15:15:28	50	10.98MB	75.8MB/s	32.17	0.28	http://localhost
16	09-07-2023 15:17:10	100	21.92MB	73.23MB/s	37.8	0.19	http://localhost
17	09-07-2023 15:19:20	200	43.57MB	61.3MB/s	45.5	0.14	http://localhost

Figure 1: Obtained results

- **Total Images**: The number of images sent in the request.
- **Payload Size**: The accumulated size of all the images sent in the request.
- **Transfer Speed**:
  - **Transfer Time**: This was measured by timing the moment the request is received on the server and until the request body is fully received and processed,
  - **Transfer Size**: Get the *Content-Length* value from the request header
  - **Speed:** Size divided by Time
- **Total Execution time on server**: This is the time in seconds of how long it took from when the request was received in the server until it was returned to the client.
- **Avg Inference Time**: For each of the images received in the request, we time how long it takes for it to be detected, and then calculate the average.
- **Location**: Where the request was executed

It is important to take into consideration the setups in which the servers run. On AWS we have an Intel Xeon CPU with two cores on 2.3GHz, 8GB of RAM and an SSD, whereas the local machine has a 6 core - 12 threads CPU on 2.30GHz, with 16GB of RAM and an SSD.

5 Conclusions

When we analyze our results, we can see some similarities in the avg inference time, but major discrepancy in the transfer speed. Based on that alone, we can conclude that, in our case, the major bottleneck for the AWS instance is the network speed, which measures only up to 1.37MB/s

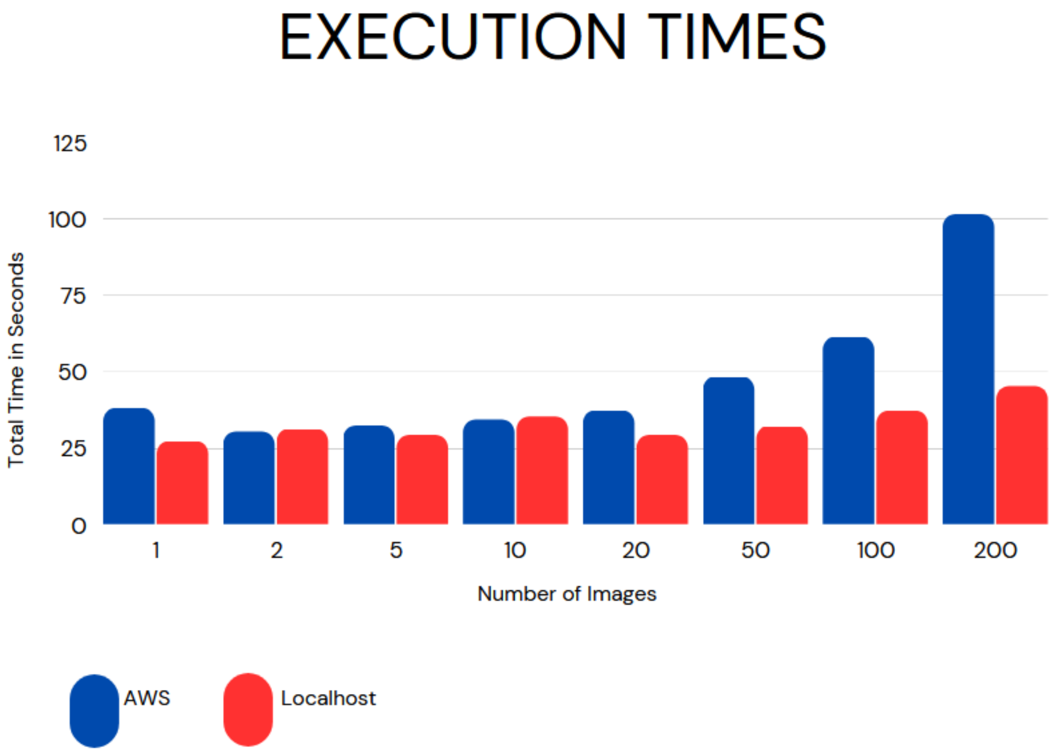


Figure 2: Comparison of execution times between AWS and Localhost

at best, and about 800KB/s on average. As a result, we can see that the *Total Execution Time* parameter is heavily affected as the payload size is increased. Based on this observation, we can conclude that, for our case, it only makes sense to offload in cases when we have less than 2.5 MB of data that we transfer, because, as we see from Figure 1, the results point in favor of Local execution as the size increases.