TECHNISCHE UNIVERSITÄT WIEN

194.048 Data-intensive Computing

Assignment 3 - Report

Group 7

Members:

Makai Andras, 0928348, e0928348@student.tuwien.ac.at
Mathis Tobias Gallus, 01621473, e1621473@student.tuwien.ac.at
Nichitov Tudor , 12045663, e12045663@student.tuwien.ac.at

Vienna, December 18, 2023

1 Introduction

Object detection involves the analysis of images and video frames to classify and locate different objects in an image. It is useful (or could be useful) in numerous applications such as medical applications, autonomous driving or surveillance systems.
Our solution relies on a TensorFlow model and while the most computationally intensive part is the training of such models, getting a prediction from a multi-layer deep learning network is also resource-intensive, which could be especially important in real-time applications (such as driving). For certain applications offloading the computation to scalable entities such as AWS with way more computing power and RAM than any single PC (they even run quantum computers!) could pay off for computationally expensive processes.
Clearly, if time is the metric that we care about than the cost of the offloading will be the transfer-time. This in turn depends on one's own internet connection as well as the bandwidth of the AWS instance (which can be wider for more $$$)
In this report, we will explore the implementation of an object detection application using Docker, Python, and TensorFlow, and examine the performance differences between local execution and remote execution on AWS. We will analyze the trade-offs between local and offloaded execution based on metrics such as inference time and transfer time, providing insights into the conditions where data offloading proves advantageous.

2 Problem Overview

2.1 Tasks

We are given the following (sub-)tasks to solve

- Object detection We are required to do object detection on various image collections and return the class and bounding box for the objects found.
- Dockerisation: We have to create a docker container for the application to run in and we send requests by POSTing the data to that.
- Local and remote execution: We need to deploy the docker container both locally as well as on AWS remotely. We collect transfer as well as inference times.
- Evaluation: Based on the collected time data, we need to discuss whether it's worth deploying the container remotely or locally based on the resources used and time consumed.

2.2 Data set

We are using the data linked in TUWEL, which involves 3 different datasets each composed of varying amounts of jpg images.

- BIG: 4913
- MEDIUM: 2070
- SMALL: 296

The image files had different sizes ranging from 14 kilobytes to 495 kilobytes.
JPG images usually encode 3 different colour channels (RGB), albeit it also supports grayscale images where only one intensity channel is encoded. JPG images are compressed using a lossy compression technique that utilises the specifics of human vision as a trade-off with storage space.

3 Methodology and Approach

3.1 Model

In the app we use an already pretrained model: Mobilenet SSD V2. It is a Machine Learning model for fast object detection specifically optimized for mobile devices.
The model takes as input an image (converted to array), computes the bounding boxes and categories for each object. Due to the small size and fast detection time the model provides the possibility of integration in an EDGE or Cloud (code offloading) architecture.

3.2 Object detection app

The 'app.py' file contains the Flask application that exposes an API endpoint for object detection using the 'run_detector' function. A first step in our application is represented by decoding the base64 encoded images. We then store the array of images in a dictionary called 'filename_image'.
Then a route is defined using the '@app.route' decorator for the /api/detect endpoint, which accepts both POST and GET requests. The 'detection_loop' function is called, passing filename_image and 'uploadtime' as arguments, and the result is stored in the out variable, which is an array of dictionaries converted to json. Each entry of the dictionary stores the following information:

1. picture ID
2. status (set to 200 for a successfull inference)
3. class ID of the object
4. bounding boxes
5. inference time (inference time of a single image)
6. average inference time
7. upload time
8. average upload time

After processing all images, the function calculates the average inference time by summing up the inference times of all entries and dividing it by the total number of entries. The detection function, 'run_detector' from the 'detect.py' resizes the image to match the input shape of the detector model, then creates the input tensors and assigns the image data to it and then simply performs the inference. The output is represented by the tensors containing bounding box coordinates, class IDs, and confidence scores.

3.3 Dockerization

We used docker as it was prescribed in the assignment and one can only speculate about what the true intentions of the author was, but in general terms dockers are a useful tool for compartmentalising code, making it portable and independent of other containers or operating systems. Each Docker container will have it's own file-system, network stack and other resources. In other applications it would simplify scalability, but as we are sending all our requests to the same docker container instead of some load-distributor, we aren't profiting from this aspect.

Running the docker container works by running the bash script located under the following location: `project3/docker/build_run_docker.sh`
Running the local or remote environment works by running the bash script under the following location: `project3/run.sh` it is currently set to run the tiny dataset, but one can change to any other directory that's in the `data` folder.
One can alter the server as well as the dataset used, both are located and commented in the `run.sh` script. As the IP address constantly changed, we needed to adjust this manually.

3.4 Metrics

1. The inference time is measured for every input using a time counter and it records the time it takes for the object detection model to produce an output. The data is then added to each entry of the dictionary.

2. After all inference times are recorded, average inference time is calculated by iterating over all elements of the array of dictionaries and calculating the average.

3. In the same manner, upload time is received as output from the flask app and is stored in each entry of the dictionary.

4. The average upload time is calculated in the same manner as the average inference time by iterating over all dictionaries in the dictionary array.

3.5 Experimental Setup

3.5.1 Specs Local Execution

Locally the datasets ran on an Ubuntu 20.04 LTS with the specs shown in Table 1.

Table 1: System Specifications for the local machine

#CPU cores	RAM (GB)	GPU present?
12	16	0

3.5.2 Specs Remote Execution

Remotely the datasets ran on an Ubuntu 20.04 LTS with the specs shown in Table 2

Table 2: System Specifications for the AWS EC2 instance used

#CPU cores	RAM (GB)	GPU present?
2	8	0

1 vCPU is equivalent to one thread on a 3.3 GHz Intel Xeon Scalable processor (Haswell E5-2676 v3 or Broadwell E5-2686 v4).

3.5.3 Setup on AWS

We deployed the same docker container that embedded the flask application, from the local part.
We opted to use an EC2 service running an Ubuntu version with a docker installation (according to the official installation guide, as the snap install didn't work). We tried

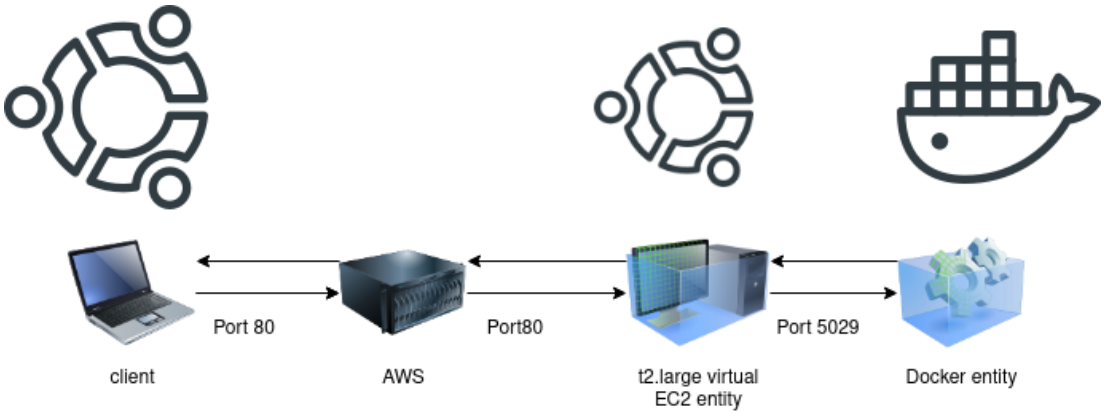

Figure 1: Architecture and communication channels of our solution. The operating systems are shown above and the ports between the entities.

both the t2.micro as well as the t2.large instances. We first ran t2.micro, but it was rather slow and returned an empty response for the MEDIUM dataset and would've taken more than 4 hours to run the BIG dataset, which would've resulted in a newer IP address being assigned, which would've broken our curl request.
We ended up using t2.large entity which has significantly more RAM and computing power than the default t2.tiny instance.
We opened up port 5029 as a custom TCP security exception so that we could send something to our docker container from the clients. When switching to t2.large one can simply copy the security settings from a previously configured instance.

4 Results

Figure 2 presents the results, clearly showing how local execution performs better than remote execution in all the tested scenarios. This is even true when comparing only the time it takes for the system to detect the objects (see inference time). The local execution consistently outshines the remote execution in this aspect. Moreover, when considering both inference time and the time it takes to upload data, the advantage of using the local approach becomes even more evident. Table 3 provides additional evidence to support our conclusion. It displays the average values obtained from our experiments, further reinforcing the idea that pursuing remote execution is in our case not worthwhile. Finally, given that our local setup was somewhat stronger than the remote one, we can imagine cases where the difference in the local and remote speeds are greater in favour of the remote machine.

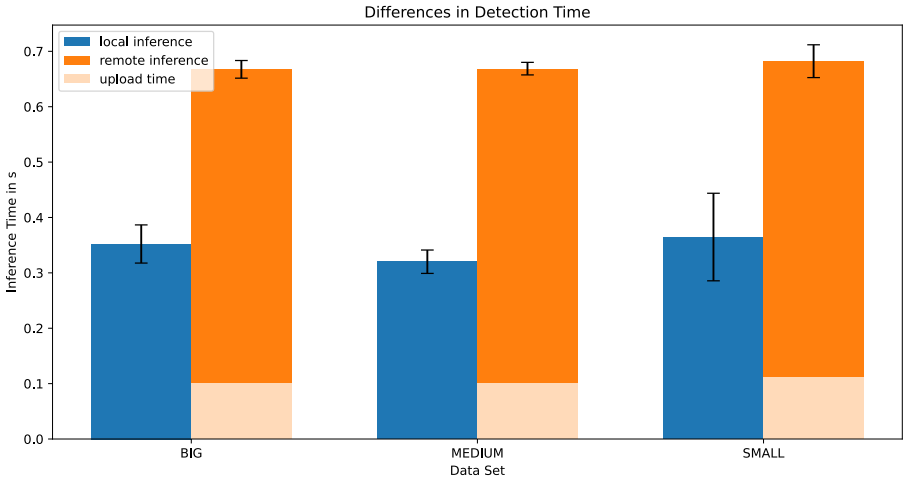

Figure 2: Graphical Overview of detection time in different settings

Table 3: Measured mean detection time in different settings: l-local, r-remote

data - location	BIG l	BIG r	MED l	MED r	SMA l	SMA r
`inference time [s]`	0.35	0.57	0.32	0.57	0.36	0.57
`upload time [s]`	0	0.10	0	0.10	0	0.11
`total time [s]`	0.35	0.67	0.32	0.67	0.36	0.58

5 Conclusions

In conclusion we want to comment on the following statements:

Is it worth offloading execution on the remote cluster? If not, why? Given the data of our experiments we have to conclude that in our case offloading yields no benefit. It has not only made the execution time longer but also made the whole process more complex and dependent on other externalities, such as stable internet connection and available workload and costs related to AWS. In addition, since images often contain sensitive data, it is easier to resolve privacy issues if the data remains on the local

machine. Sadly, the compute power on AWS was not high enough. The upload time was around 0.1 s per image, which is around 1/3 of the inference time locally. Thus with faster remote computing power, we would have been able to outperform the local execution.

What would be needed to improve the performance of remote and local execution? As our comparison showed, there are two main factors in cloud computing that influence the results: the computing power and the upload time. Increasing the first on aws and decreasing the latter (either with higher bandwidth or better data compression)can greatly increase the attractiveness of offloading specific computing intensive tasks.

Can you think of a scenario where offloading improves performance? Although in this particular task offloading was not beneficial, there are a lot of scenarios in which offloading is not only useful but absolutely necessary. One could think of scenarios where only one sensor (e.g., a small camera) is connected to a weak microcomputer and therefore has to rely on external computing capabilities. Furthermore, as 5G becomes more and more established as a standard, higher transmission speeds can be expected in the near future, and this trend will continue.