Offloading in Object Detection using AWS

Report for Exercise 3 in Data Intensive Computing, Group 49

Adam Domoslawski (12241331),
Dario Giovannini (01227603),
Hela Bouabdeli (01435477)

July 2023

Contents

1 Introduction

The ability to efficiently process and analyze large volumes of data is crucial in today's data-driven world. To achieve this, the integration of cutting-edge technologies and innovative approaches becomes essential. In this assignment we aim to build a robust image processing application that shows the power of Docker applications with their dependencies, ensuring consistency across different environments and TensorFlow which is a popular open-source machine learning framework that will facilitate the efficient processing and analysis of complex data sets. The application developed with python will use TF object detection API (TensorFlow) module to detect and locate instances of objects on given images. Besides, we will leverage the cloud infrastructure of Amazon Web Services (AWS) to deploy and test our data processing application. AWS offers a wide range of scalable and reliable services, providing an ideal environment for hosting and executing computationally intensive tasks. In fact, we aim to achieve seamless scalability and efficient computation, enabling our application to handle large workloads effectively.

2 Implementation

The application consists of the principal "app.py" script, which contains the main function to which post requests get routed by flask. This function extracts the image from the posted json object, as well as the request time. The extracted images are then decoded from a string back into an image and then a numpy array.

The images are then passed to the "detection_loop" function, which contains the functionality for preforming object detection on the passed images. To this end, a module from the TensorFlow hub is used - specifically, mobilenet V2 was chosen as it is a lightweight model with lower memory requirements than some others, which was deemed useful for the remote offloading task given limited resources there.

The actual process of inference using this model is very straightforward - the decoded image is converted into a tensorflow tensor and passed to the model, which returns the predicted bounding boxes. These get converted into a list for JSON-serializability and together with execution time information combined into a response JSON object.

2.1 Client

The client script provides the convenient ObjectDetectionClient class, which serves as a client for the flask server instantiated with "app.py". Specifically, it loads an image from a filename, encodes it in a json-serializable way, and sends the request to the specified url. It then returns the response object as-is, but provides a further convenience function for plotting the bounding boxes on top of the original image, as shown in image 1.

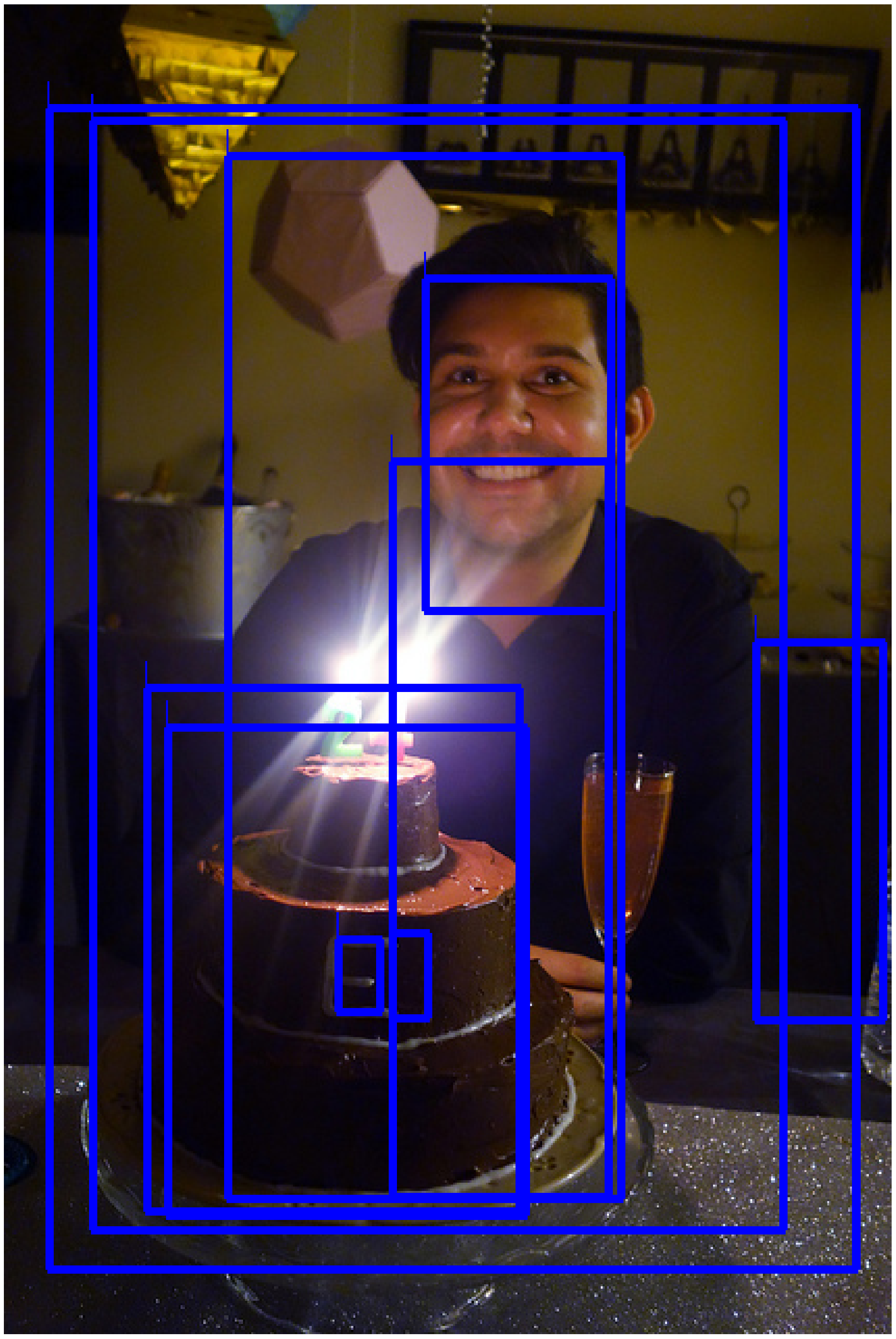
Figure 1: Bounding boxes on the first image of the dataset.

The client provides functions for doing detections using a single image, provided as a filename-string, or multiple images at once by passing a list of such filename strings. Either way the images are encoded one-by-one and sent to the app as a list of such encoded strings.

2.2 Docker

In this step, we have used Docker which is a platform that allows user to package an application all with its dependencies into a more standardized unit which is knows as container. It is like a lightweight virtual machine that runs your application with all its necessary libraries and settings. Once, we had the Dockerfile and requirements.txt file ready, we executed the following commands respectively in the terminal to build the docker image and to run the docker locally :

```
docker build -t dic-assignment .
docker run -d -p 8080:8080 dic-assignment
```

2.3 Remote

In order to perfom the remote execution of our containerized application on Amazon Web Services (AWS) and collect data on transfer time and inference time, we followed the following steps:

1. set up our AWS account and signed in to the AWS Management Console.
2. We chose the appropriate AWS deployment option which is Elastic Compute Cloud (EC2).
3. set up an EC2 instance and configure it with the necessary security groups, networking, and storage options.
4. Deploy and run the containerized application running on the EC2 instance.
5. Data Offloading to the remote AWS environment, we had established a secure connection from the client to the EC2 instance issues.

3 Experimental Setup

3.1 Inference and Transfer Time

Inference time is calculated within the "detection_loop" function using the "time" package. While looping over the images that have been passed to this function, the time is recorded at the start of each loop and then subtracte from the time at the end. This difference is the inference time, and added to the cumulative inference time. The average inference time is then calculated by dividing the total cumulative inference time by the number of images in the loop.

Transfer or upload time is calculated similarly, though the start-time for this is recorded by the client when a detection function is called, before any image loading or encoding happens. This timestamp is then passed to the app with the detection request, and at the beginning of the "detection_loop" call, the current time is subtracted from this timestamp to calculate the upload time.

3.2 Hardware

3.2.1 Local

The local experiments were performed on a windows PC. Cuda was not used as detection performance was quite fast already and it could give a better impression of the intention of offloading - if a local machine is powerful enough to run inference itself, as is the case if it were Cuda-capable, then the offloading process is pointless overhead.

3.2.2 Remote

The EC2 instance used for remote execution was equipped with a high-performance CPU with multiple cores to handle the computational workload efficiently.Besides, for the network Bandwidth,we ensured a reliable and high-speed network connection between the client and the AWS EC2 instance.

4 Results

4.1 Repeated single detection vs batch-detection

In local experiments, sending images to the app one at a time or all at once in a list via the discussed functions led to no appreciable difference in performance. In the case of single detection, the average total time over the 296 images in the SMALL dataset is just under 0.1s (0.097s), while the average total time per image when sending them all at once is 0.0971s, though with slightly higher inference time - which may be due to random variance - and slightly lower upload time.

5 Conclusion

Based on our findings, offloading execution to the remote AWS cluster is worth considering but the decision depends on different factors such as the network conditions, device capabilities and data size. Moreover, we found that we can improve the performance of both remote and local execution by optimizing the containerized application code, employing hardware accelerators such as GPUs and ,last but not least, Utilizing edge computing infrastructure to minimize data transfer time.