1. Introduction

Object recognition has become an important part of computer vision, allowing machines to recognize and locate objects within images or video. This project focuses on implementing an efficient object detection system using a lightweight model. The goal is to develop a system that can accurately identify objects in real-time scenarios while maintaining optimal performance.

The project uses state-of-the-art techniques and state-of-the-art deep learning frameworks to achieve robust and accurate object detection. By leveraging a lightweight model, we aim to strike a balance between accuracy and computational efficiency, making the system suitable for deployment on resource-limited devices or platforms.

In addition to model development, this project also highlights the practical application of object detection by creating a web application. The web application provides a user-friendly interface where users can upload images or videos and receive real-time object detection results. This application is designed to be scalable and accessible, allowing users to easily interact with the object detection system without requiring extensive technical knowledge.

To ensure the scalability and reliability of the web application, it is deployed on Amazon Web Services (AWS) using Docker. Docker enables the creation of simple and portable containers that encapsulate an application and its dependencies, ensuring consistent behavior across different environments. Using AWS services such as Elastic Container Registry (ECR) and Elastic Compute Cloud (EC2), we ensure seamless application deployment and management.

The combination of a simple object detection model, easy to use web application, and AWS deployment provides a comprehensive solution for object detection tasks. This project aims to demonstrate the practicality and effectiveness of object detection in real-world scenarios, allowing users to take advantage of the capabilities of the system for various applications such as surveillance, object recognition, and automation.

Throughout this report, we will discuss the methodology used, the design and implementation of the object detection system and web application, as well as the challenges encountered and insights gained from the project. By analyzing the performance and evaluating the results, our goal is to demonstrate the effectiveness and potential of the developed system in advancing object detection applications.

2. Problem Overview

Object detection is a crucial task in computer vision, involving the identification and localization of objects within images or videos. The challenge lies in efficiently processing large-scale datasets and deploying object detection models in real-world scenarios. In this project, we aim to address these challenges by combining object detection, Docker, and AWS technologies.

The primary objective is to develop an accurate and lightweight object detection system that can operate in real-time scenarios. By leveraging cutting-edge deep learning techniques, we seek to achieve optimal performance while maintaining high precision and recall rates. Additionally, we aim to optimize the model's efficiency, making it suitable for deployment on resource-constrained devices or cloud environments.

To facilitate the practical application of the object detection system, we employ Docker for containerization. Docker enables the creation of self-contained, portable containers that encapsulate the application, its dependencies, and the required environment. This approach ensures consistency and reproducibility, allowing for seamless deployment across different platforms and environments.

Furthermore, we utilize AWS services, specifically Amazon Elastic Container Registry (ECR) and Amazon Elastic Compute Cloud (EC2), to deploy and manage the containerized object detection system. AWS provides scalable infrastructure and robust computing resources, ensuring high availability and efficient utilization of resources. By leveraging these cloud services, we can seamlessly scale the system to handle varying workloads and effectively serve multiple users.

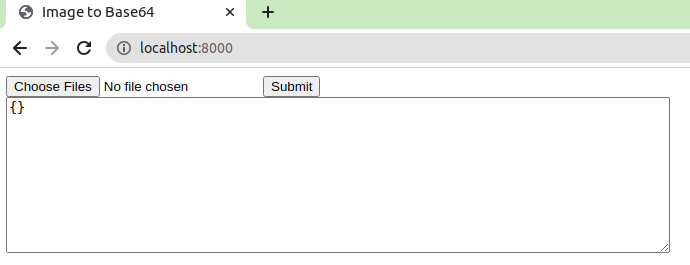
The project also focuses on the development of a user-friendly web application that enables users to interact with the object detection system. Through the web application, users can upload images or videos and receive real-time object detection results. This intuitive interface simplifies the usage of the system, making it accessible to users with diverse technical backgrounds.

Throughout this project, we address the challenges of object detection performance, deployment, and accessibility. By combining lightweight object detection models, containerization with Docker, and deployment on AWS, we provide an end-to-end solution that facilitates efficient object detection at scale. The project report covers the methodology, implementation details, and performance analysis, showcasing the benefits and potential applications of the developed system.

3. Methodology and Approach

3.1. Implementation of data processing application

We initially utilized a CNN model implemented in Tensorflow documentation. The original documentation utilizes FasterRCNN+InceptionResNet V2 for the detection. The size of this model is larger than 200 MB so it takes a bit long to load the model. So, we utilized a more lightweight model ssd+mobilenet. I downloaded this model on the local and loaded the model before the flask app started.

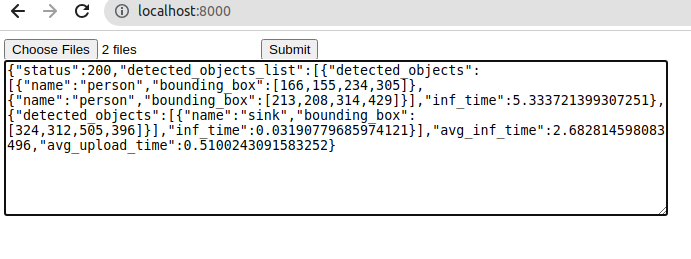
The frontend of the web application is as following:  


You can choose multiple image files using the “Choose Files” button. Once the images are selected you should push the “Submit” button to send the request to the backend. We load the images in base64 string format. For multiple images, we create a list and push these base64 strings to that list. And then, we convert this list into a json format and send the request to /api/detect.

In the backend, we get the request from the frontend, which is in a base64 array format. We convert this into a numpy array list and utilize the ssd+mobilenet model to detect objects in the images. The result is in the following json format:

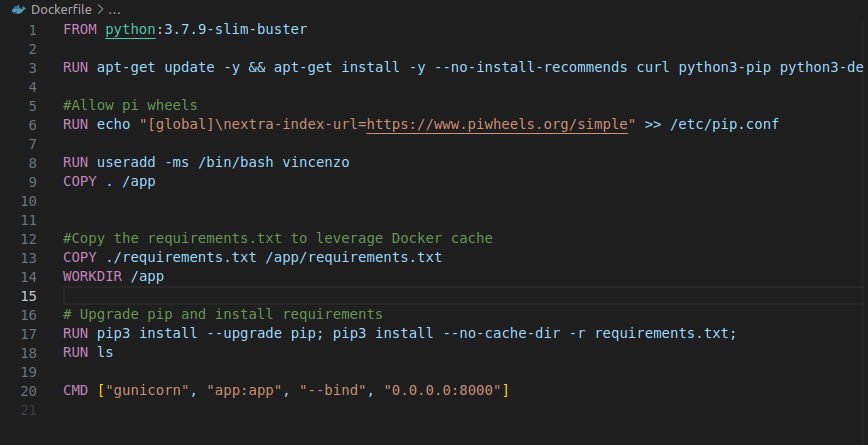
{"status":200,"detected\_objects\_list":[{"detected\_objects":[{"name":"person","bounding\_box":[166,155,234,305]},{"name":"person","bounding\_box":[213,208,314,429]}],"inf\_time":5.333721399307251},{"detected\_objects":[{"name":"sink","bounding\_box":[324,312,505,396]}],"inf\_time":0.03190779685974121}],"avg\_inf\_time":2.682814598083496,"avg\_upload\_time":0.5100243091583252}

Detected\_objects\_list contains the list of detected objects. If we input 10 images, this list contains 10 dictionaries. In each iteration, we calculate the inference time, class category and bounding boxes. After the whole process finishes, we calculate the average inference time and average upload time.

Then, we send the result to the front end to show it. The result is as following:  


We can run this flask app using the “python3 app.py” or “python app.py” command. In windows, we use python command and in linux, we use python3 command.

3.2. Dockerization of application

We utilized docker to build a container for the flask app we built. In order to run the flask app on server, we utilized Python gunicorn library. And the server starts running with "gunicorn app:app --bind 0.0.0.0:8000" command on the server. We utilized flask 2.1.3 for this app. The dockerfile looks as following:  


We utilized the “docker build -t dic-assignment .” command to build the docker image.

3.3. Local and remote execution

- Local execution

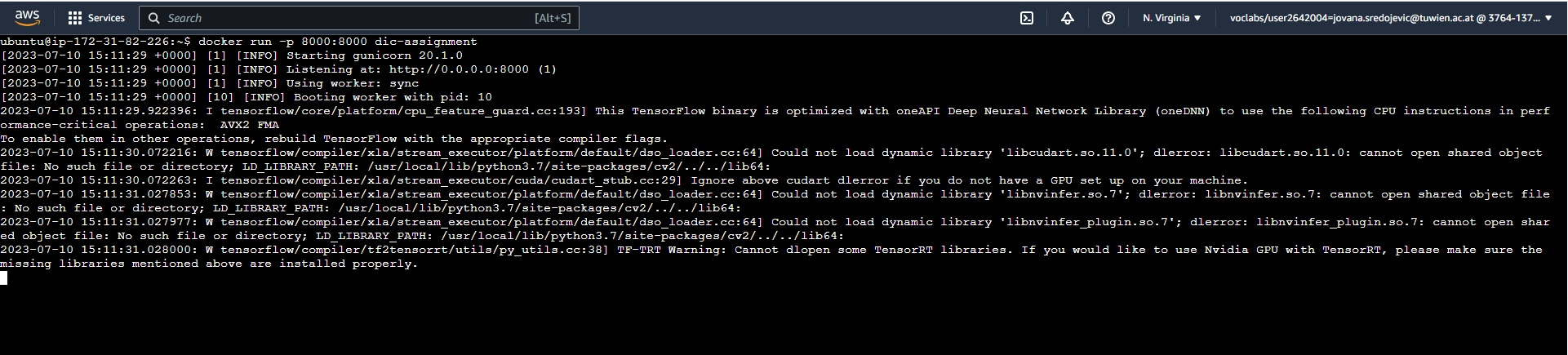
We can run the flask app on a local machine using the “python3 app.py” command. Or we can use "gunicorn app:app --bind 0.0.0.0:8000".

In order to run the docker image on the local machine, we can use the “docker run -p 8000:8000 dic-assignment” command to run the docker image.

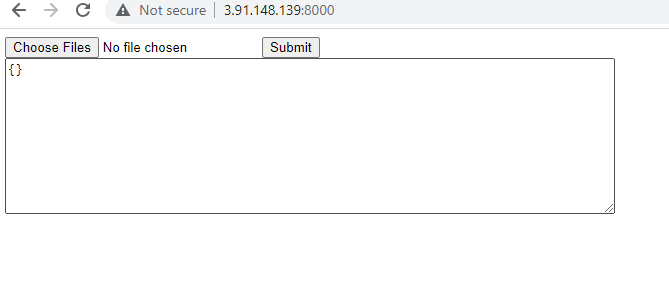
- Execution on EC2

First, we created an EC2 instance on AWS. We named the instance object\_detection\_docker\_EC2. We created a t2.large instance with 2 CPUs and 24 GB of additional storage.

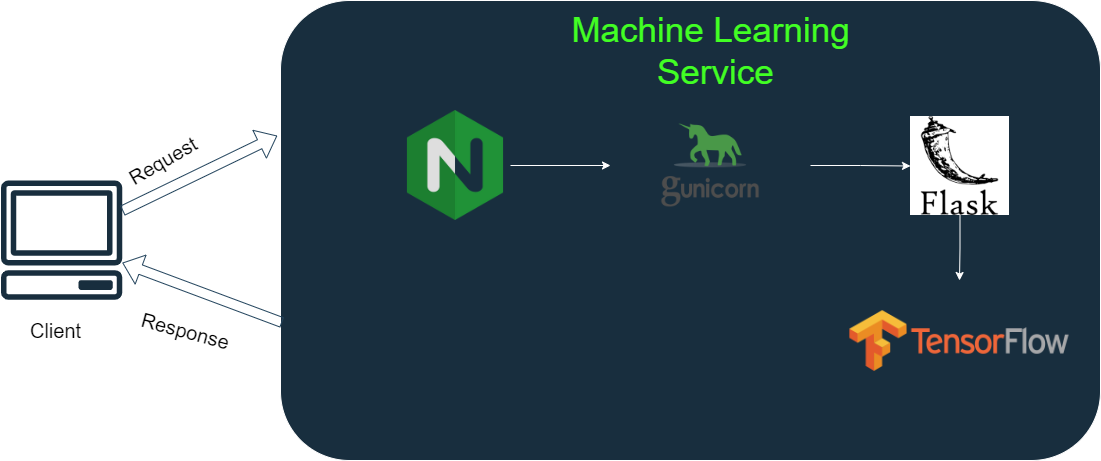
We utilized git to upload the code on EC2. Here, we installed the docker on EC2 instance and built the docker image over there. The process is the same as the local. After the docker image is ready, we run the app with “docker run -p 8000:8000 dic-assignment” command. Once the docker image is running, we can see the following commands from the EC2.



In order to deploy the application on the AWS EC2, we should reverse the proxy using nginx. We can run the web browser and type the ip address and port of this EC2 instance to open this app. The command is http://<ip\_address>:8000/. For example, <http://3.91.148.139:8000/>. The result is as follows:



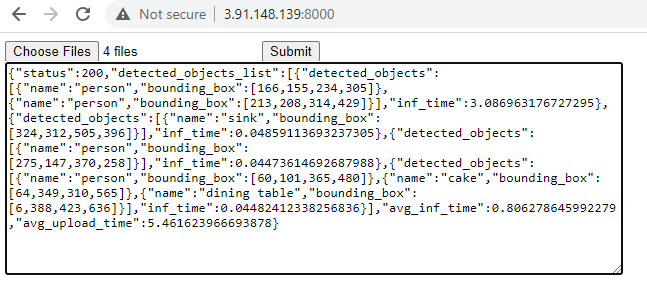
The next diagram shows the main infrastructure of the flask app on AWS EC2.



Note: The student account is not secure, so the IP address changes once the session expires. So, we should check the IP address before running the flask app.

4. Results

Once we select the images and send the request on the app deployed in the cloud, the result looks like this.



The average upload time and average inference time take longer than on local execution. Because we should transfer data from local machine to AWS server and from AWS server to local machine as well.

5. Conclusions.

In this project, we utilized a ssd+mobilenet model to detect the object from multiple images. We utilized docker to build the docker image and deployed it on the AWS platform. We used the services such as EC2, ECR and ECS for the deployment. To run the server on AWS, we utilized gunicorn python library.

As time goes by, the object detection systems are getting more and more accurate and lightweight. We can improve the efficiency of this model using the state-of-the-art yolov8 model as well. We can improve the UI/UX of the frontend of this web application for better customer satisfaction.