MŰEGYETEM 1782

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Telecommunications and Media Informatics

Gusztáv Ferenczi

SOFTWARE AS A SERVICE FOR OBJECT DETECTION: EVALUATING PERFORMANCE, SCALABILITY AND DEPLOYMENT AGAINST LOCAL SOLUTIONS

SUPERVISOR

Dr. László Toka

BUDAPEST, 2024

Contents

STUDENT DECLARATION

Budapest, 31 May 2024

Ferenczi Gusztáv

....

Gusztáv Ferenczi

Summary

This thesis presents a comparative analysis of object detection solutions for vehicular environments, evaluating the performance, scalability, and deployment of Software as a Service (SaaS) offerings against local implementations. The thesis focuses on three SaaS providers - *Microsoft Azure Computer Vision, Google Cloud Vision*, and *Amazon Rekognition* - and two local solutions based on *Ultralytics' YOLOv5* and *YOLOv8* models.

The methodology involves converting and filtering multiple datasets, including Audi A2D2, Roboflow Vehicular Dataset, and nuScenes, into a standardized format for consistent evaluation. A modular inference pipeline was designed to process images through the various object detection models and store results for comparison.

The comparative analysis covers several key aspects, including performance metrics such as Intersection over Union (IoU) and inference time, cost implications, and the impact of different configurations like image resolution, server location, and batch processing on the performance of the object detection solutions. Fine-tuned versions of the models are created based on the dataset and are compared with their more generalized counterparts.

By providing a comprehensive evaluation of SaaS and local object detection methods, this thesis aims to offer valuable insights and guidance for selecting the most suitable approach for vehicular applications.

Összefoglaló

Ebben a szakdolgozatban különböző objektumfelismerési megoldásokat hasonlítunk össze közlekedési adathalmazok használatával. Kiértékeljük a teljesítményt, a skálázhatóságot és az implementációt a "Szoftver mint Szolgáltatás" (SaaS) kínálatok és a lokális megoldások esetében. A szakdolgozat három SaaS szolgáltatóra - *Microsoft Azure Computer Vision*, *Google Cloud Vision* és *Amazon Rekognition* - valamint két helyi megoldásra, az *Ultralytics YOLOv5* és *YOLOv8* modelljeire fókuszál.

A módszertan magában foglalja több adathalmaz, köztük az Audi A2D2, egy Roboflow adathalmaz és a nuScenes átalakítását és szűrését egy szabványos formátumba a konzisztens értékelés érdekében. Kifejlesztettem egy moduláris pipeline-t, amely feldolgozza a képeket a különféle objektumfelismerő modellekkel, valamint tárolja és kiértékeli az eredményeket.

Az összehasonlítás során számos kulcsfontosságú szempontot veszünk figyelembe, beleértve az "Intersection over Union" (IoU) teljesítménymutatót, a válaszidőt, a költséget, valamint különböző beállítások hatását az objektumfelismerő megoldások teljesítményére. Ezek magukba foglalják a képek felbontását, a szerverek helyét, a batch feldolgozást és a modellek továbbtanítását. A modellek tanítását a fent említett adatokon végezzük, majd összehasonlítjuk az eredményeket a modellek eredeti változataival.

A szakdolgozat átfogó értékelést nyújt a SaaS és a helyi objektumfelismerési módszerekről, értékes betekintést adva a legmegfelelőbb megoldások kiválasztásához közlekedési környezetekben történő objektumfelismeréshez.

1 Introduction

In today's rapidly evolving world of autonomous vehicles and self-driving cars, the processing of image and radar data using artificial intelligence is crucial for further advancements. Cloud services and Software as a Service (SaaS) solutions are increasingly being explored to enhance the capabilities of these systems in vehicular environments.

During the completion of the "*Project Laboratory*" subject in 2023, I gained extensive experience working with Convolutional Neural Networks (CNNs) in the context of self-driving cars. The primary objective of my project was to detect objects in image data, while simultaneously expanding my knowledge of convolutional networks.

For this project, I primarily used *Keras* to build and train neural networks, alongside various other libraries for object recognition and data processing. I successfully created two functioning models, although they only partially achieved the desired results. One model was developed entirely using *Keras*, while the other was based on a pre-trained model. These models could detect specific pre-trained objects within input images, albeit with significant inaccuracies.

I also implemented data preprocessing and developed an easily adjustable and reconfigurable solution. This allowed for the modification of the object type being searched for, the structure of the model layers, and the training process.

My previous work demonstrated that a theoretically complex and seemingly inaccessible field has become increasingly approachable and learnable, thanks to the availability of high-level APIs developed by major companies (such as *TensorFlow* and *PyTorch*), learning opportunities (like *Kaggle*), and specifically designed development environments and free resources (such as *Google Colab*).

1.1 Objectives

Expanding on the above-mentioned foundational knowledge and experience, my current thesis project seeks to thoroughly explore and investigate object detection solutions for vehicular applications, emphasizing the potential of cloud services and SaaS offerings. This research will involve a comprehensive analysis and comparison of various approaches to determine the most effective methods for use in self-driving car applications.

1.2 Scope of the Study

In this chapter we list the main objectives of the thesis.

1.2.1 Comparison of SaaS Solutions vs. Local Implementations

Evaluate the performance of various Software as a Service (SaaS) solutions (*Microsoft Azure Computer Vision*, *Google Cloud Vision*, and *Amazon Rekognition*) against local CNN implementations (*Ultralytics' YOLOv5* and *YOLOv8*). This involves assessing key metrics such as inference time and Intersection over Union (IoU).

1.2.2 Effectiveness in Object Detection

Determine the effectiveness of each solution in detecting objects within vehicular environments, specifically for self-driving car applications. This will include evaluating the precision of bounding boxes and the overall detection accuracy. Also evaluate the performance gains of fine-tuning pretrained models.

1.2.3 Cost, Scalability, Deployment, and Processing Methods

Analyze the cost implications of using SaaS solutions compared to local implementations. This includes considering the cost of cloud services, data transfer, and computational resources required for local processing.

Examine the scalability of SaaS solutions for managing large datasets and handling numerous inference requests. Compare batch processing performance and efficiency with single file processing. Evaluate different image upload methods to cloud services, measuring inference times and determining optimal batch sizes for efficient processing. Assess the ease of deployment and flexibility of SaaS solutions versus local implementations. Compare the efficiency of uploading images sequentially in binary format to pre-uploading images to cloud storage and referencing them for inference.

1.2.4 Evaluation Across Different Datasets and Resolutions

Assess the impact of image resolution on the performance and accuracy of object detection. This includes comparing inference times and IoU for different image resolutions to determine the trade-offs between processing speed and detection precision.

Use multiple vehicular datasets (*Audi A2D2*, a *Roboflow Vehicular Dataset*, and *nuScenes*) to ensure robustness and generalizability of the results. This involves converting datasets into a unified format, running the object detection pipeline, and evaluating the performance of each solution on these datasets.

1.3 Organization of the Thesis

This thesis begins with an overview of neural networks, convolutional neural networks (CNNs), and object detection models, including the tools and environments used in the research. The *methodology* chapter covers the selection of SaaS providers and local solutions, the datasets utilized, and the data format and pipeline design.

In the *implementation* chapter, the focus is on data conversion, the creation of object detection functions, and the development of the inference pipeline. The evaluation chapter presents performance metrics, hardware information, and result visualizations. The *comparative analysis* chapter provides in-depth comparisons between local YOLO models and SaaS, performance across different environments, and the effects of various parameters on inference time and accuracy.

At the end, we conclude our research and interpret our findings. Future research options are also explored.

2 Theoretical Framework and Technologies

2.1 Neural Networks

Neural networks are essential to this thesis because they form the backbone of modern object detection models, enabling them to accurately identify and classify objects within images. Understanding the principles and applications of neural networks is crucial for evaluating the performance and scalability of different object detection solutions, both local and SaaS-based.

In this chapter, we will discuss some theoretical concepts related to neural networks, providing a partial overview, and introducing fundamental elements, as a comprehensive exploration is beyond the scope of this thesis.

2.1.1 Fundamentals of Neural Networks

Neural networks are computational models inspired by the human brain, designed to recognize patterns and make decisions based on data. They consist of layers of interconnected nodes (neurons) that process and transmit information. Each neuron receives inputs, applies weights to them, sums them up, and passes the result through an activation function to produce an output.

The basic operation of a neuron can be represented mathematically as:

$$y = f\left(\sum_{i=1}^{n} w_i x_i + b\right)$$

where y is the output, x_i are the inputs, w_i are the weights, b is the bias, and f is the activation function[1].

2.1.1.1 Architecture of Neural Networks

Neural networks are typically organized into three types of layers:

1. **Input Layer:** The first layer that receives the raw input data.
2. **Hidden Layers:** Intermediate layers that perform complex transformations of the input data[2][3].
3. **Output Layer:** The final layer that produces the network's output.

2.1.1.2 Activation Functions

Activation functions introduce nonlinearity into the neural network, enabling it to learn more complex patterns. Common activation functions include:

- **Sigmoid**: $\sigma(x) = \frac{1}{1+e^{-x}}$
- **Tanh**: $\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
- **ReLU (Rectified Linear Unit)** $\text{ReLU}(x) = \max(0, x)$

2.1.1.3 Training Neural Networks

Training a neural network involves adjusting the weights and biases to minimize the difference between the predicted output and the actual target values. This process uses backpropagation and optimization algorithms like gradient descent. Backpropagation calculates the gradient of the loss function with respect to each weight, allowing for updates that reduce the overall error. Gradient descent adjusts the weights iteratively to minimize the loss function.

2.1.1.4 Overfitting and Regularization

Overfitting occurs when a model performs well on training data but poorly on new data. Regularization techniques help mitigate overfitting by adding constraints to the model[4]. Common methods include:

- **L2 Regularization**: Adds a penalty proportional to the square of the magnitude of weights.
- **Dropout**: Randomly sets a fraction of the neurons to zero during training to prevent over-reliance on specific neurons.

2.1.2 Deep Learning and Neural Networks

Deep learning, a subset of machine learning, focuses on neural networks with many layers, known as deep neural networks (DNNs). These networks effectively model complex and abstract features in data, making them suitable for tasks like image and speech recognition, natural language processing, and autonomous driving.

DNNs consist of multiple hidden layers, each learning different abstraction levels. The network's depth allows it to transform input data through hierarchical layers, with each layer extracting higher-level features from the previous one. This hierarchical learning enables DNNs to recognize intricate patterns and make sophisticated decisions.

Training DNNs requires large datasets and significant computational power, typically provided by GPUs and specialized hardware accelerators. Advanced optimization techniques, such as stochastic gradient descent (SGD) and adaptive learning rate methods like Adam, efficiently train these models.

A key advantage of deep learning is its automatic feature extraction capability. Traditional machine learning approaches rely on hand-crafted features, which need domain expertise and are time-consuming to develop. In contrast, deep learning models extract relevant features directly from raw data, reducing the need for manual feature engineering.

Despite their power, DNNs face challenges, such as overfitting, especially with small datasets. Techniques like dropout, batch normalization, and data augmentation help improve generalization and prevent overfitting[5].

Deep learning has revolutionized various fields by advancing tasks that were previously difficult for machine learning models. Its ability to learn complex representations from large datasets makes it central to modern artificial intelligence research and applications.

2.2 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are a class of deep neural networks specifically designed for processing structured grid-like data such as images. Unlike traditional neural networks, which treat each pixel in an image independently, CNNs leverage the spatial structure of the data, making them particularly effective for tasks like image recognition and classification.

A typical CNN architecture consists of several key layers: convolutional layers, pooling layers, and fully connected layers.

A typical CNN architecture consists of:

- **Convolutional Layers:** Core building blocks that apply convolution operations using learnable filters to detect features like edges, textures, or patterns. The result is a set of feature maps highlighting different aspects of the input.

- **Pooling Layers:** Layers that follow convolutional layers and perform down-sampling to reduce spatial dimensions and parameters, mitigating overfitting. Max pooling is the most common operation.
- **Fully Connected Layers:** After several convolutional and pooling layers, these layers integrate the extracted features to produce the final output. They have neurons connected to every neuron in the previous layer.

Activation functions like ReLU introduce non-linearity, while batch normalization and dropout accelerate training and improve generalization.

2.3 Object Detection Models

Object detection aims to identify and localize objects within an image or video by assigning labels and bounding boxes. It is more challenging than image classification due to the dual task of classification and localization. Deep learning, particularly CNNs, has revolutionized this field with end-to-end trainable models that simultaneously classify and localize objects accurately[6].

Several influential deep learning-based object detection models offer different trade-offs between speed and accuracy:

- **R-CNN, Fast R-CNN, Faster R-CNN:** Region proposal-based methods with increasing speed improvements.
- **SSD:** Eliminates region proposals for faster detection but can sacrifice some accuracy compared to two-stage methods.
- **YOLO:** Treats detection as a single regression problem, predicting bounding boxes and class probabilities directly for very high speeds.
- **Mask R-CNN:** Extends Faster R-CNN with a branch for predicting object masks, enabling instance segmentation[7].

2.3.1 YOLO (You Only Look Once) Models

The YOLO family represents a significant advancement in real-time object detection by predicting bounding boxes and class probabilities in a single evaluation. It formulates object detection as a regression problem, dividing the image into a grid and having each cell predict bounding boxes, confidence scores, and class probabilities.

YOLO's design philosophy emphasizes speed and simplicity, making it suitable for real-time applications.

Various YOLO versions have introduced improvements:

- **YOLOv1**: Introduced the single regression approach but struggled with small and closely packed objects.
- **YOLOv2 (YOLO9000):** Improved accuracy with batch normalization, high-resolution classifiers, and anchor boxes.
- **YOLOv3**: Enhanced accuracy and speed with multi-scale predictions and a deeper *Darknet-53* architecture[8].
- **YOLOv4**: Integrated features like *CSPDarknet53* backbone, Mosaic data augmentation, and self-adversarial training.
- **YOLOv5**: Developed by *Ultralytics* with optimizations in performance, ease of use, and flexible model sizes.
- **YOLOv6 and YOLOv7**: Continue refining the architecture and incorporating advanced training techniques.
- **YOLOv8**: One of the latest versions of YOLO, also developed by *Ultralytics*, builds upon previous versions with further improvements in performance, flexibility, and ease of use[9][10].

YOLO's real-time object detection capabilities make it particularly relevant for self-driving car applications, where fast and accurate identification of objects in the vehicle's environment is crucial for safe navigation.

2.3.2 SaaS-Based Object Detection Solutions

Software as a Service (SaaS) has revolutionized object detection by offering scalable, ready-to-use solutions from major cloud providers like Microsoft, Google, and Amazon.

- **Microsoft Azure Computer Vision**: Azure's API provides robust object detection, offering bounding boxes and confidence scores. It's scalable and easy to integrate, supporting use cases from retail to security[11].

- **Google Cloud Vision**: Google's API delivers advanced image analysis, identifying objects and their locations. It's highly scalable, ideal for digital asset management and content moderation[12].
- **Amazon Rekognition**: Offers comprehensive image and video analysis, detecting objects, scenes, and activities with bo[^eunding boxes and confidence scores. It integrates with other AWS services, ensuring scalability and easy workflow integration[13].

SaaS-based object detection solutions offer key advantages:

- **Scalability**: Handle large-scale image processing without major infrastructure investments.
- **Ease of Use**: Integrate object detection via simple API calls, requiring no deep machine learning expertise.
- **Cost-Effectiveness**: Pay-as-you-go pricing manages costs by charging only for used resources.
- **Continuous Improvement**: Regular updates ensure access to the latest machine learning advancements.

These solutions enable quick deployment of image analysis, faster development cycles, and focus on core business objectives.

2.4 Tools and Environments

In my project, various tools and environments were utilized to support the implementation and comparison of object detection methods. These tools include *Jupyter Notebook*, *Google Colab*, Python libraries and packages, and integrated development environments (IDEs). Each of these tools and environments played a crucial role in the development, testing, and evaluation phases of the project.

2.4.1 Jupyter Notebook

Jupyter Notebook is an open-source web application for creating and sharing documents that combine live code, equations, visualizations, and narrative text. It supports various programming languages, including Python, which was used exclusively in this project. Its flexibility and interactivity make Jupyter Notebooks ideal for data analysis, machine learning, and scientific computing[14].

For this project, Jupyter Notebook was chosen over Google Colab for fair testing of SaaS providers. Running tests on Google Colab, part of the Google ecosystem, might introduce bias due to potential interconnectivity with Google Cloud servers. Using Jupyter Notebook in VSCode provided a neutral and consistent testing environment.

2.4.2 Google Colab

Google Colab is a cloud based Jupyter Notebook environment provided by Google, offering access to powerful computing resources like GPUs and TPUs at no cost[15]. It's popular for machine learning, data analysis, and other computational tasks due to its ease of use and accessibility.

Initially, Google Colab was used for this project because of its convenience and high-performance hardware. However, concerns arose about potential bias in performance measurements since Google Colab operates within Google's ecosystem. This could lead to faster image upload speeds due to more efficient interconnections with Google Cloud servers.

For fair comparison among all SaaS providers, the project was migrated to Jupyter Notebook running locally on Visual Studio Code. This change provided consistent and unbiased testing conditions, ensuring each SaaS provider was evaluated under the same circumstances.

2.4.3 Visual Studio Code

In this project, VSCode (Visual Studio Code) served as the primary IDE for developing and testing the code. VSCode is a lightweight yet powerful source code editor with built-in support for various programming languages, including Python. Its rich ecosystem of extensions enhances functionality, offering support for Jupyter Notebooks, an integrated terminal, and debugging tools. Using VSCode enabled switching between Jupyter Notebooks and traditional Python scripts, facilitating a smooth development workflow[16].

Leveraging these tools and environments allowed for efficient implementation, testing, and evaluation of different object detection solutions, ensuring a robust and comprehensive analysis.

2.4.4 RunPod

To evaluate the performance of YOLO models on high-end GPUs, I used RunPod, a cloud-based service providing access to dedicated GPU hardware. RunPod allows users to rent powerful GPUs, making it ideal for computationally intensive tasks like object detection[17].

Once the environment was set up, I uploaded my preprocessed datasets (Roboflow Vehicular, and nuScenes) to the RunPod instance. I deployed the YOLOv5 and YOLOv8 models, loaded the pre-trained weights, and set up the inference scripts.

3 Methodology

3.1 Selection of SaaS Providers

This section details the selection criteria and features of the three prominent SaaS providers chosen for this study: Microsoft Azure Computer Vision, Google Cloud Vision, and Amazon Rekognition. These providers were selected based on their capabilities, pricing, scalability, and ease of integration.

3.1.1 Microsoft Azure Computer Vision

Microsoft Azure Computer Vision offers comprehensive object detection services, providing bounding boxes and confidence scores. It also includes OCR, face recognition, image tagging, and moderation. Azure's versatility makes it suitable for retail inventory management, medical imaging analysis, and vehicle tracking, and it integrates well with other Azure services.

3.1.2 Google Cloud Vision

Google Cloud Vision API leverages advanced machine learning models for object detection and classification. It also offers OCR, face detection with emotional attributes, logo and landmark recognition, image property analysis, and safe search detection. This makes it ideal for surveillance monitoring, crop monitoring in agriculture, or vehicular applications.

3.1.3 Amazon Rekognition

Amazon Rekognition provides robust object detection and recognition, identifying objects, scenes, and activities within images. It supports label detection, face analysis, text detection, celebrity recognition, content moderation, and person tracking in videos. Rekognition integrates with AWS services, making it suitable for retail product recognition, security surveillance, and quality control.

3.1.4 Comparison

Provider	**Free Tier**	**Paid Tier**	**Rate Limits**	**Image Size**
Microsoft Azure Computer Vision[18]	5,000 images per month	$1 per 1,000 images	Free: 20 transactions per minute, Paid: 20 transactions per second	Up to 20MB
Google Cloud Vision[19]	1,000 units per month	$1.50 per 1,000 images	Up to 1,800 images or requests per minute	Up to 20MB
Amazon Rekognition[20]	5,000 images per month	$1 per 1,000 images	No specified transaction per second limit	Sent as byte array: 5MB, S3 reference: 15MB

Figure 1: This table summarizes the pricing and rate limitations for the selected SaaS providers.

Microsoft Azure Computer Vision, Google Cloud Vision, and AWS Rekognition were selected for their comprehensive features, competitive pricing, and ease of integration. These providers suit various object detection applications in vehicular environments.

Microsoft Azure offers robust object detection and integration within the Azure ecosystem, making it versatile for enterprises. Google Cloud Vision leverages advanced machine learning for high-throughput applications. AWS Rekognition excels in integration with AWS, ideal for large-scale deployments.

Other providers considered but not selected include Clarifai, which was too expensive at $0.10 per image[21] beyond the free tier, and IBM Watson Visual Recognition, which has been discontinued[22]. AWS Lookout for Vision was unsuitable for the project's general object detection needs[23].

3.2 Selection of Local Solutions

In addition to exploring various SaaS options for object detection, this project integrates local solutions, specifically Ultralytics' YOLOv5 and YOLOv8 models. These models are renowned for their accuracy, speed, and efficiency in object detection tasks, making them suitable benchmarks against SaaS offerings. This section outlines the selection and integration of these local solutions into the evaluation pipeline.

3.2.1 Ultralytics' YOLOv5

YOLOv5 (You Only Look Once version 5) is an open-source object detection model developed by Ultralytics. It was chosen for its impressive balance between detection accuracy and inference speed, crucial for real-time applications such as object detection in vehicular environments. YOLOv5 runs efficiently on both CPUs and GPUs, offering versatility in deployment[24].

YOLOv5 has extensive community support and comprehensive documentation, on troubleshooting, model training, and implementation. Its pre-trained models on the COCO dataset can be easily fine-tuned for specific applications, allowing rapid adaptation without the need for training from scratch.

Integration involves setting up the development environment, installing necessary Python packages like PyTorch, and configuring Jupyter Notebook. Pre-trained YOLOv5 models are then downloaded and configured for object detection tasks. Datasets such as Audi A2D2, Roboflow Vehicular, and nuScenes are converted to a YOLOv5-compatible format. Custom Python scripts handle image loading, inference, and post-processing results to align with evaluation metrics.

3.2.2 Ultralytics' YOLOv8

Building on YOLOv5, Ultralytics developed YOLOv8 with enhanced features and performance improvements, offering higher accuracy and faster inference times. YOLOv8's improved architecture handles complex scenes better and provides higher detection accuracy, essential for robust object detection in dynamic environments.

YOLOv8 is optimized for both edge devices and high-end GPUs, providing flexibility for deployment in various environments. Its features include improved handling of small objects, enhanced non-maximum suppression (NMS), and refined training algorithms.

The integration process for YOLOv8 mirrors that of YOLOv5, starting with setting up the development environment, installing necessary Python packages, and configuring the environment. Pre-trained YOLOv8 models are acquired and configured for object detection tasks. The same datasets used for YOLOv5 are prepared in the YOLOv8-compatible format. Custom Python scripts for YOLOv8 leverage its advanced features for optimal performance and accuracy[25].

3.3 Dataset Selection and Preparation

In the context of this research, selecting and preparing suitable datasets was critical for the effective evaluation of object detection models. The datasets chosen were specifically tailored for vehicular environments to simulate self-driving car scenarios accurately. The process involved thorough investigation, selection, conversion, and validation of these datasets to make sure they were fit for the experimental framework. Here, we discuss three primary datasets: Audi A2D2, Roboflow Vehicular Dataset, and nuScenes.

The Waymo Open Dataset, introduced by Waymo (formerly the Google self-driving car project) in 2019, was also considered for this research due to its extensive and detailed annotations. However, its complexity and the dataset’s size and substantial computational resources required to make use of it led to its exclusion from the datasets used in this project.

3.3.1 Audi A2D2 Dataset

The Audi A2D2 dataset, created by Audi in 2019, is a comprehensive dataset designed for autonomous driving research. It includes high-resolution images captured by multiple cameras mounted on an Audi vehicle, providing a detailed 360-degree view of the car's surroundings. The dataset contains 3D bounding box annotations for various objects, which are crucial for 3D object detection tasks[26].

Initially, the Audi A2D2 dataset seemed promising due to its rich data and detailed annotations. However, upon further analysis, it became apparent that the 2D bounding boxes derived from the 3D annotations were not precise enough for this study's requirements. The bounding boxes were often not tightly fitted around the objects, leading to potential inaccuracies in the object detection evaluation. This limitation necessitated the search for alternative datasets that offered more precise 2D annotations. Consequently, the Audi A2D2 dataset was ultimately dropped from further consideration.

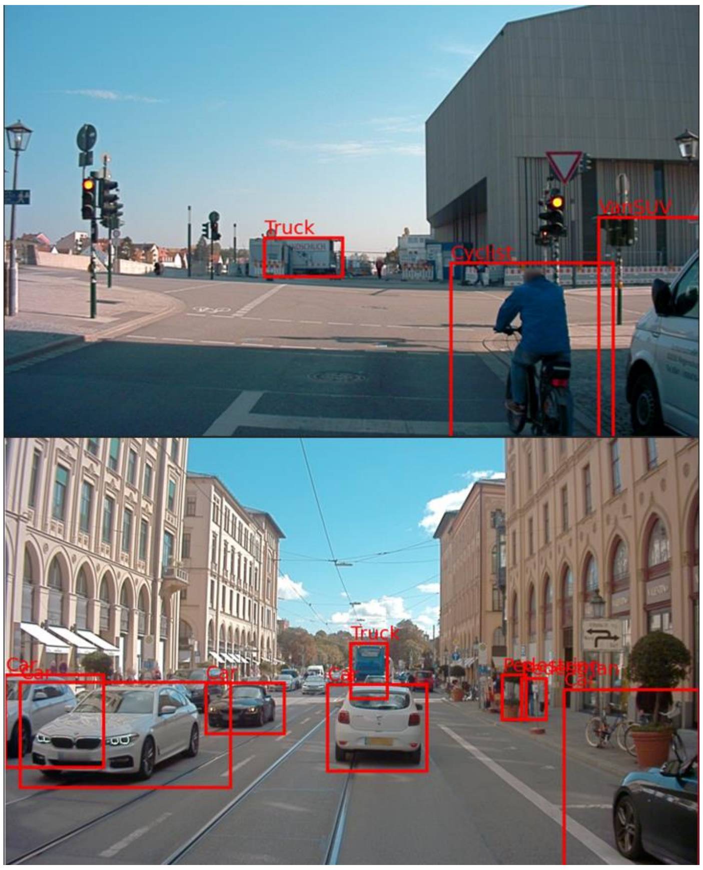


Figure 2: Example images of the A2D2 dataset and its 2D annotations.

3.3.2 Roboflow Vehicular Dataset

The Roboflow Vehicular Dataset is an enhanced version of the original Udacity Self-Driving Car Dataset. Roboflow, a platform that specializes in managing, preprocessing, augmenting, and versioning datasets for computer vision, undertook the task of re-labeling this dataset to correct errors and omissions. This improved dataset includes precise annotations for thousands of pedestrians, bikers, cars, and traffic lights, which were missing in the original dataset, addressing critical gaps that could affect model performance and safety[27].

This dataset contains 97,942 labels across 11 classes and spans 15,000 images, with 1,720 null examples (images with no labels). All images are high resolution at 1920x1200, though a downsampled version at 512x512 is also available. Annotations have been hand-checked for accuracy by Roboflow, ensuring high-quality data. Despite these improvements, some duplicated bounding boxes remain, particularly for stoplights, which could affect model performance. This dataset is released under the MIT License, making it easily accessible and usable for various applications.

The Roboflow dataset provided a solid foundation for initial experiments due to its comprehensive annotations and ease of use across various machine learning frameworks. The dataset's detailed metadata and multiple format options enabling integration into the experimental pipeline. However, the discovery of the nuScenes dataset, which offered even more detailed and structured data, led to its adoption as the primary dataset for this research.

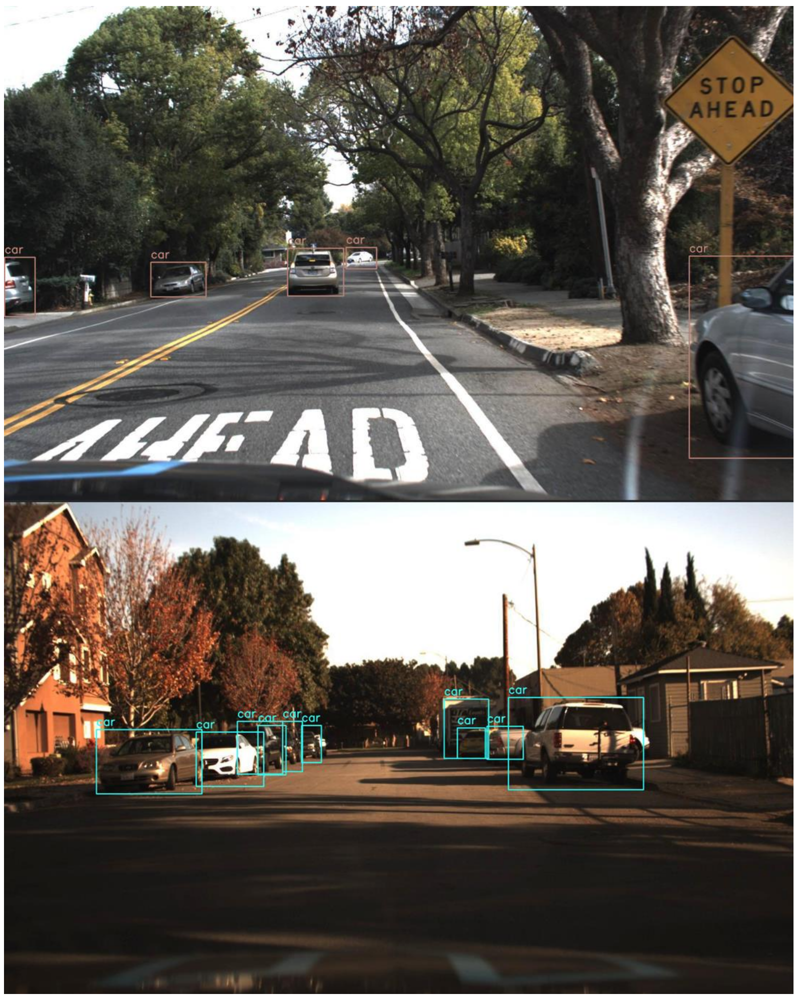


Figure 3: Examples of the Roboflow Vehicular dataset and it's better fitted annotations compared to A2D2, though still not perfect.

3.3.3 nuScenes Dataset

The nuScenes dataset, developed by Motional and nuTonomy in 2019, is a comprehensive resource for autonomous driving research. It includes 1,000 scenes from Boston and Singapore, capturing diverse weather conditions, times of day, and traffic densities. The dataset offers multimodal sensor data, including LiDAR, RADAR, and high-resolution images from six cameras.

One major advantage of the nuScenes dataset is its precise 2D bounding box annotations for a wide range of objects, making it ideal for object detection tasks. The bounding boxes are tightly fitted and well-categorized, ensuring accurate data for training and evaluation. The dataset's detailed class naming scheme includes categories like vehicles, pedestrians, animals, and various static and movable objects. For this project, only the human and vehicle categories were used, excluding the ego car category[28].

The nuScenes dataset is well-structured and pre-divided into training, validation, and test sets, encouraging straightforward data handling and consistent model evaluation. Specifically, the nuImages subset was used, focusing on the "samples" packages, which contain annotated keyframes. Unlike the "sweeps" part, which includes non-annotated images between keyframes, the "samples" provide well-annotated frames for robust training and evaluation.

The official nuScenes Python package and GitHub repository streamlined data compilation and conversion processes, offering comprehensive support for accessing, visualizing, and processing the data. These tools, along with extensive documentation, enabled efficient data preparation, making nuScenes the primary dataset for this project.

3.4 Data Format and Pipeline Design

The design of the data format and the processing pipeline is crucial for the successful comparison of various object detection solutions. Our objective was to create a modular and efficient system that can handle different datasets and object detection methods.

3.4.1 Custom Data Format

To manage multiple datasets and ensure consistency across different object detection methods, I developed a custom data format. This format was designed to standardize the input data, making it easier to compare results from various models and services. The custom data format includes several key elements: image data, bounding box data, and output/results data.

The image data in our format is stored in a standardized resolution and format. This ensures that the object detection models receive uniform input, for fair comparison. Each image is identified by an *image_id,* which is a unique identifier for the image within the dataset.

The bounding box data is standardized to include normalized coordinates and dimensions. Each bounding box is defined by the coordinates (x, y) of the top-left corner, the width w, and the height h of the bounding box. This uniform representation simplifies the process of calculating evaluation metrics like Intersection over Union (IoU). Additionally, each bounding box is associated with a *class_name*, indicating the type of object detected within the box.

In the output additional data is added after object detection and includes more information such as the time taken for inference, or the time taken for uploading the images to the server of the SaaS provider, both measured in milliseconds. IoU is also added for each image after calculating it and confidence values for each inferred bounding box returned are also stored in this output format. This way we can easily save our results and retrieve them for later analysis.

In this format, each detection service (e.g., Google, Azure, Rekognition, YOLOv5) has its own entry for each image, within a list.

An example of the standardized data format, described in a JSON-like template, is as follows:

Input:

(Dataset's annotation format after conversion)

```
[
    {
        "image_id": "path/to/image.jpg",
        "objects": [
            {
                "class_name": "object_class",
                "bbox": [
                    x_normalized,
                    y_normalized,
                    width_normalized,
                    height_normalized
                ]
            }
        ]
    }
]
```

Output:

(Converted data returned from inference and stored for evaluation)

```
[
    {
        "service_name": {
            "image_id": "path/to/image.jpg",
            "objects": [
                {
                    "class_name": "object_class",
                    "bbox": [
                        x_normalized,
                        y_normalized,
                        width_normalized,
                        height_normalized
                    ],
                    "confidence": confidence_value
                }
            ],
            "inference_time": inference_time_in_ms
            "upload_time": upload_time_in_ms
        }
    }
```

3.4.2 Pipeline Overview

The object detection pipeline was designed to manage data preprocessing, model inference, and post-processing systematically. The process is divided into several stages, each with specific tasks.

The first stage involves downloading and preparing datasets from sources like Audi A2D2, Roboflow, and nuScenes. Each dataset was converted to a custom format, ensuring consistency across the pipeline. Irrelevant or low-quality data were filtered out, and datasets were split into necessary subsets.

Once prepared, datasets were standardized into a common format for uniform handling throughout the pipeline. Next, Python packages for various object detection models and cloud services were imported.

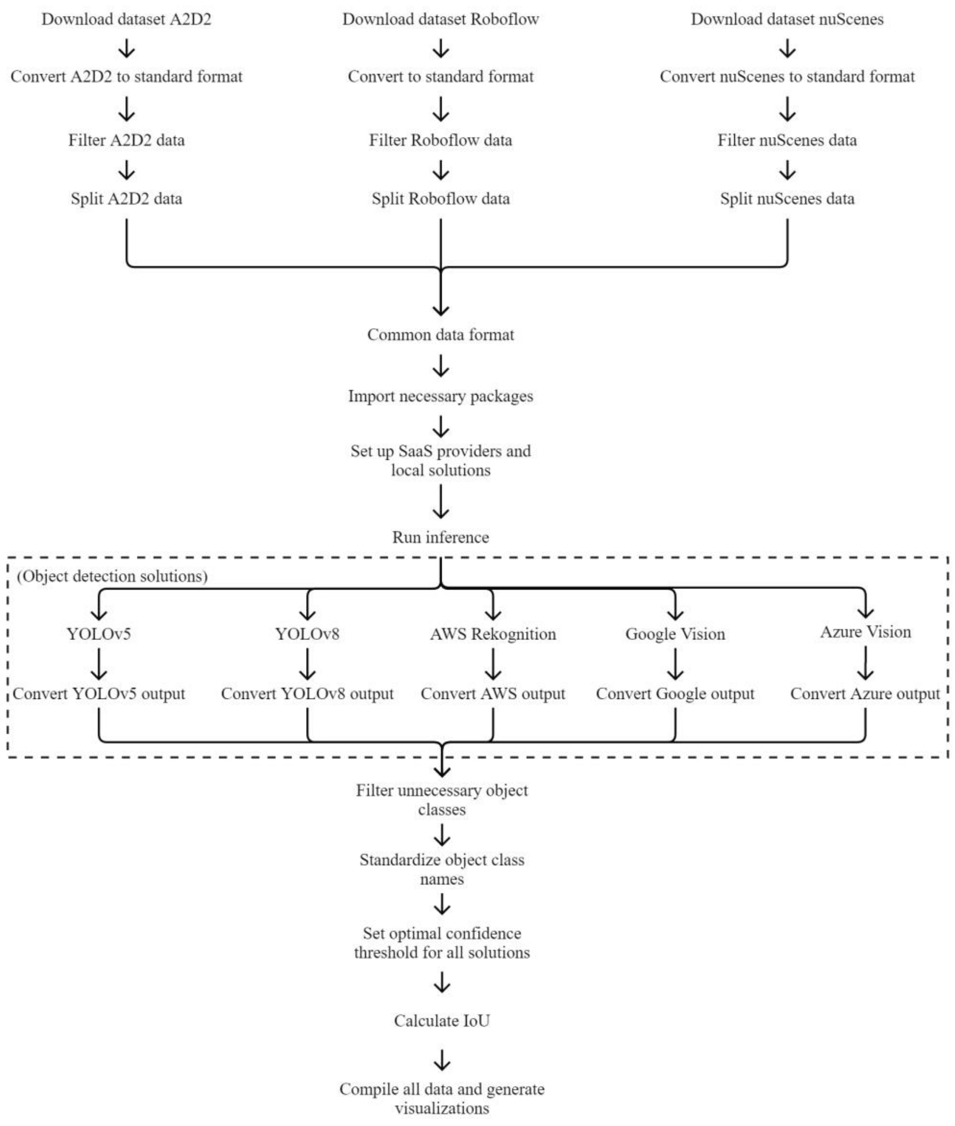
I then configured SaaS providers and local solutions, setting up access keys and endpoints for SaaS providers and loading local object detection models. Proper configuration was crucial for communication with cloud services and correct local model setups.

The inference process began by randomly selecting a subset of images. These images underwent object detection using functions defined for each SaaS provider and local solution. Results were returned in a standardized output format and stored in a single list.

Post-processing involved converting detection results to a common format, filtering out unnecessary object classes, and standardizing class names. Optimal confidence thresholds were determined by calculating the average IoU for different thresholds and selecting the highest. An optimized inference result was created by filtering out bounding boxes below the optimal confidence threshold.

To evaluate and present results, I collected hardware information (CPU, RAM, GPU), calculated average inference times and IoUs for each method, and saved results in a timestamped folder. This included original inference results in a JSON file and a text file with summarized results and hardware information. Visualizations compared the performance of different methods, including bar plots for average inference times and IoUs, line plots showing convergence over multiple inference calls, and histograms of confidence values, inference times, and IoUs.

This structured approach ensured efficient processing and meaningful comparisons across object detection solutions. The pipeline's modularity allowed easy integration of new datasets and models.



Download dataset A2D2
Download dataset Roboflow
Download dataset nuScenes
Convert A2D2 to standard format
Convert to standard format
Convert nuScenes to standard format
Filter A2D2 data
Filter Roboflow data
Filter nuScenes data
Split A2D2 data
Split Roboflow data
Split nuScenes data
Common data format
Import necessary packages
Set up SaaS providers and local solutions
Run inference
(Object detection solutions)
YOLOv5
YOLOv8
AWS Rekognition
Google Vision
Azure Vision
Convert YOLOv5 output
Convert YOLOv8 output
Convert AWS output
Convert Google output
Convert Azure output
Filter unnecessary object classes
Standardize object class names
Set optimal confidence threshold for all solutions
Calculate IoU
Compile all data and generate visualizations

Figure 4: Diagram of the pipeline used.

3.5 Environment Setup

3.5.1 Python Packages and Libraries

To implement the object detection pipeline, a range of Python packages and libraries were utilized. These included boto3 for interacting with AWS Rekognition, *google.cloud.vision* for Google Cloud Vision, *azure.cognitiveservices.vision.computervision* for Microsoft Azure Computer Vision, and *torch* for leveraging the PyTorch-based YOLO models. Additional libraries such as *cv2* (OpenCV) for image processing, *matplotlib.pyplot* for data visualization, and numpy for numerical operations were also essential.

During the initial setup, conflicting package versions presented challenges. This was resolved by creating a new Python virtual environment, isolating the dependencies, and ensuring compatibility across all required packages. The virtual environment allowed for controlled installation of each library using pip, mitigating version conflicts, and simplifying the management of dependencies. The libraries installed included *io* for input/output operations, *json* for handling JSON data, *os* for interacting with the operating system, *shutil* for some file operations, *random* for random number generation, *time* for measuring SaaS server upload/response times, *ultralytics* for YOLO model utilities, *IPython* for interactive computing, *psutil* for system and process utilities, *gpustat* and *cpuinfo* for displaying and saving hardware information, and *PIL* for image handling.

3.5.2 Access Keys and Endpoints Setup

Setting up access keys and endpoints for the SaaS providers was a critical step in configuring the environment. This process involved creating accounts and configuring resources for each provider—Amazon Rekognition, Google Cloud Vision, and Microsoft Azure Computer Vision. Each provider required specific setup steps to enable API access and storage integration.

For AWS Rekognition, an *IAM root user* with appropriate permissions was created, and access keys were generated. An *S3 bucket* was configured to store images, for the upload-and-reference testing approach. Google Cloud Vision required the creation of a service account with the necessary permissions, followed by generating a JSON key file. A Google Cloud *Storage bucket* was also set up for image storage. Similarly, Microsoft Azure Computer Vision necessitated the creation of a *Cognitive Services* account, obtaining the subscription key, and setting up a *Blob Storage container*.

These access credentials and endpoints were integrated into the code, ensuring easy but secure communication with each SaaS provider. The setup process involved careful management of these credentials, storing them securely, and ensuring they were correctly referenced in the code to enable API interactions and data storage operations. This way, the object detection pipeline could efficiently leverage both local and cloud-based resources.

4 Implementation

4.1 Data Conversion and Filtering

In this chapter, I describe the steps taken to convert and filter the datasets used in this thesis. Three datasets were initially considered: Audi A2D2, Roboflow, and nuScenes. However, due to issues with bounding box precision and missing labels, the Audi A2D2 dataset was ultimately excluded.

The Roboflow and nuScenes datasets were utilized, with specific processing applied to ensure their compatibility and reliability for object detection tasks. The standardized format described previously in the thesis was what their labels were converted into.

4.1.1 Conversion of Audi A2D2 Dataset

The Audi A2D2 dataset was initially considered for this project. However, upon closer inspection, the 2D bounding boxes provided were not sufficiently precise, as they were primarily intended for 3D inference training. Sometimes less important objects further away from the observer were labeled while ones located closer to the camera/car were not. Still, the conversion process involved normalizing the bounding box coordinates and filtering out unused object classes while converting the data into our standardized format.

Despite the effort to convert the dataset, the imprecision of the bounding boxes led to the decision to exclude the Audi A2D2 dataset from this project.

4.1.2 Conversion of Roboflow Dataset

The Roboflow dataset required significant processing to make sure it was suitable for use. This involved removing duplicate labels based on overlapping bounding boxes and filtering out unnecessary classes such as traffic lights. The bounding boxes' coordinates were normalized, and the data was reformatted to match our desired input structure.

Filtering involved removing overlapping bounding boxes with an Intersection over Union threshold, checking each bounding box pair in an image, to eliminate duplicates and retaining only relevant classes:

```
# Example: Filtering script based on IoU and class relevance
def process_json_file(json_path):
    with open(json_path, 'r') as file:
        data = json.load(file)

    valid_classes = ['pedestrian', 'car', 'biker', 'truck']
    preferred_classes = {'truck': 'car', 'biker': 'pedestrian'}
    filtered_data = []

    for image_data in data:
        filtered_objects = []
        for obj in image_data['objects']:
            if obj['class_name'] in valid_classes:
                should_keep = True
                for other_obj in filtered_objects:
                    # if the “obj” objects’ class name is valid:
                    # if obj is “truck”, while other_obj is “car”, we only
                    # keep “truck”, which is preferred
                        if iou > 0.65:
                            should_keep = False
                            break
                if should_keep:
                    filtered_objects.append(obj)

        if filtered_objects:
            image_data['objects'] = filtered_objects
            filtered_data.append(image_data)

    with open(json_path, 'w') as file:
        json.dump(filtered_data, file, indent=4)
```

In the code above, one of the longer *if* conditions was replaced with pseudocode, to make it more readable what our conditions were. So, if one of the overlapping (IoU over 65%) bounding boxes were in the “preferred_classes” dictionary, I only kept the one which was of the class deemed more specific, which were “truck” or “biker” over “car” and “pedestrian”, respectively.

Although the Roboflow dataset improved in quality after processing, it still had some missing labels and less precisely fitted bounding boxes compared to the nuScenes dataset.

4.1.3 Conversion of nuScenes Dataset

The nuScenes dataset provided the most reliable and precise data for this project. The dataset included tightly fitted bounding boxes and comprehensive labeling, which significantly improved the quality of the analysis. The conversion process involved filtering out classes that were not relevant to the vehicular context, such as objects outside the vehicle or human categories.

First, I downloaded and imported the official nuScenes devkit library from their GitHub repository which contains excellent in-depth documentation on its usage and makes loading and handling of the dataset an easier task:

```
from nuimages.nuimages import NuImages
# Initialize NuImages for each subset
nuim_train     =     NuImages(dataroot=data_path,     version='v1.0-train',
verbose=True)
nuim_val = NuImages(dataroot=data_path, version='v1.0-val', verbose=True)
```

This time I did not have to make any complex filters with IoU calculations due to the labels' accuracy, I just had to again just normalize the bounding box coordinates like before with my previous datasets and convert it into our standard data input format, then save it:

```
# Function to save images and annotations to the specified subset
def save_to_subset(subset, sample_data, nuim, data_path, output_folder):
    image_path = os.path.join(data_path, sample_data['filename'])
    with Image.open(image_path) as img:
        image_width, image_height = img.size
        # Retrieve and filter annotations for the sample_data
        image_annotations = {
            "image_id": os.path.basename(image_path),
            "objects": []
        }
        anns = [ann for ann in nuim.object_ann if ann['sample_data_token']
== sample_data['token']]
        valid_annotations = False
        for ann in anns:
            category = nuim.get('category', ann['category_token'])
            class_name = category['name']
            if             (class_name.startswith('human')             or
(class_name.startswith('vehicle') and class_name != 'vehicle.ego')):
                bbox     =     normalize_bbox(ann['bbox'],     image_width,
image_height)
                image_annotations["objects"].append({
                    "class_name": class_name,
                    "bbox": bbox
                })
                valid_annotations = True
        if valid_annotations:
            # Save image to output folder
            output_image_path    =    os.path.join(output_folder,    subset,
os.path.basename(image_path))
            img.save(output_image_path)
            annotations[subset].append(image_annotations)
```

Any object whose class was not a subtype of “vehicle” or “human”, except for the “vehicle.ego” category, which corresponds to the car that has taken the images and was filtered from the dataset.

4.2 Object Detection Functions

In this chapter, we go into the specifics of implementing object detection functions for the three SaaS providers selected.

These functions were designed to enable integration and comparison with local solutions, namely YOLOv5 and YOLOv8. Each SaaS provider required unique handling for both sequential byte-array uploads and batch processing methods.

Each function is designed to perform object detection using a specific provider's API. The functions are categorized into three versions: sequential byte array upload for immediate inference, upload to a cloud storage service followed by inference, and batch processing where multiple images are uploaded and inferred in batches. The latter is only available for the Google Vision API as the other two services do not support batch requests for label detection specifically.

For proper integration with the respective SaaS APIs, the setup process involves initializing the API clients for each provider. This setup block establishes connections to AWS, Google Cloud, and Microsoft Azure services using appropriate authentication credentials.

The code excerpts can be found in the “Annex” section of this document or on my GitHub repository[29] found in “References”.

4.2.1 AWS Rekognition

The first version of the function for AWS Rekognition uploads images as byte arrays sequentially for immediate inference. The image is read as bytes and sent directly to the Rekognition API, which returns the detected objects and their bounding boxes.

The second version uploads the image to an S3 bucket first, then runs inference by referencing the image in the bucket. This method separates upload and inference times for detailed performance analysis.

4.2.2 Google Vision

Sequential Byte Array Upload

For Google Cloud Vision, the sequential upload function reads the image as bytes, sends it to the Vision API, and retrieves the detected objects.

Upload to Cloud Storage and Then Inference

This version uploads the image to Google Cloud Storage before referencing it for object detection, allowing the separation of upload and inference times.

Batch Processing

For batch processing, multiple images are uploaded to Google Cloud Storage and then processed together. This method improves efficiency for large datasets. Only Google's API supports online synchronous batch image annotation this way. It allows up to 16 images to be referenced for object localization tasks. After uploading all selected images, the same way as before, by looping over them and calling *blob.upload_from_filename(image_path),* we can then run inference on them all at once, while measuring the time for a response to be received.

To eliminate unnecessary cloud storage costs, we must immediately after delete the data stored on the servers.

4.2.3 Microsoft Azure Computer Vision

Sequential Byte Array Upload

Like our previous functions, the function for Microsoft Azure Computer Vision uploads images as byte arrays sequentially and performs immediate inference using the API.

Upload to Cloud Storage and Then Inference

Like before, this second version uploads the image to Azure Blob Storage before performing object detection, separating upload and inference times. For the sake of brevity, I only wrote down the important lines we need to add to our code to use the service in this way, in the Annex section of this document.

4.2.4 YOLOv5 and YOLOv8

For both YOLOv5 and YOLOv8, I developed functions to load the models, run inference on images, and convert the results into a standardized format for comparison with SaaS solutions. The functions are designed to be modular and efficient, allowing for easy integration into the main pipeline.

4.2.4.1 Inference Function

The function for running local inference reads an image from a given path, runs the YOLOv5/v8 model to detect objects, and returns the results along with the inference time. This function is optimized to handle various image sizes and configurations, making it flexible for different datasets and testing conditions.

The images were already converted to our desired size so I just set the inference resolution to that (otherwise the default height would be 640 pixels for every image). This is important as the SaaS solutions do the inference at whatever the resolution of the uploaded images was, without needing to declare the desired inference resolution, so this is necessary for a fair comparison.

4.3 Inference Pipeline

The inference pipeline is a critical component of the project, for the processing of images through various object detection models. The pipeline integrates multiple stages, ensuring a seamless flow from image selection to inference execution and results storage.

4.3.1 Conversion of Images

Without deleting the original images, new images are temporarily saved after being converted to a set resolution. This way we can test all SaaS/local object detection solutions' speed and performance at various image resolutions and compare them later.

We can set the image height and this script will resize our images to the specified size at the beginning of our pipeline, while keeping the aspect ratio the same.

4.3.2 Inference from All Sources

The core of the inference pipeline is the function that executes object detection across all selected models and SaaS providers. The *inferenceFromAllSources* function calls the detection functions for each model and consolidates the results.

In the code, we can easily comment out, replace, or add new methods we want to try out for object detection. The results are stored separated based on inference method.

4.3.3 Storing Inference Results

After running the inferences, the results are stored in a structured manner. This involves appending each inference result to a list, which is then used for further analysis and visualization. The results include details such as detected objects, bounding boxes, and confidence scores, later adding IoU performance metrics and inference times too.

4.4 Post-Processing

4.4.1 Class Name Conversion and Filtering

Class name conversion is necessary to standardize the output from different object detection methods. Different models might use different names for the same class of objects, so a mapping system is employed to keep uniformity. The conversion process involves using a dictionary to map alternative class names to a common name, which simplifies further analysis and comparison.

As seen in the code found in the Annex section, the data is simplified from all sources, including the original dataset into only four wider categories: “car”, “truck”, “person”, and “biker”.

As the different SaaS providers, YOLO, and the two different datasets I ended up using have wildly different classification schemes, I had to specify very wide categories.

There is some ambiguity in categorizing objects like ambulances, construction equipment, and others, as initial dataset classification requires setting boundaries between object types, which can not cover all real-life edge cases perfectly. This issue is even harder with different classification schemes from various sources. To mitigate this, I declared only four categories as relevant and converted all objects into our naming format.

4.4.2 Confidence Thresholds Optimization

I initially disabled the confidence threshold for any solutions that I could. This was easily done for local solutions, setting their thresholds to 0% for both YOLO implementations used.

As for the SaaS solutions, while Amazon Rekognition allows setting the confidence threshold, Google and Microsoft Azure do not. Google Vision only allows setting the maximum number of detected objects which I set to a value of 100. Rekognition’s confidence threshold was also set to 0% in the request.

While these low thresholds obviously return too many incorrectly detected objects in an image, the purpose of doing this was to remove the differing confidence scores from unfairly impacting the performance results of our inference methods. And even though Google and Microsoft do not allow their users to lower their minimum thresholds, they still return confidence scores for every detected object.

This allowed us to dynamically set the confidence thresholds for every object localization solution to their respective optimal values (for our datasets and methodology).

This might introduce unfair bias towards Rekognition and YOLO, because only they can go below the default 50% confidence threshold, but the limitations of the other two solutions simply do not allow us to retrieve the necessary data for a better and more equal comparison.

The *find_optimal_confidence_thresholds* function iterates over a range of threshold values, calculating the average IoU for each threshold. The threshold that yields the highest average IoU is selected as the optimal threshold.

After finding the thresholds that yield the best average IoU across the given dataset, all detected objects with a confidence score below that thresholds are filtered out, for each detection method.

4.4.3 IoU Calculation

4.4.3.1 Introduction

The Intersection over Union (IoU) is a critical metric in evaluating the performance of object detection models. It quantifies how accurately the predicted bounding boxes (by the model) overlap with the ground truth bounding boxes (actual labels). An IoU score closer to 1 indicates a high degree of overlap, signifying accurate object detection, while a score closer to 0 indicates little to no overlap.

IoU is defined as the ratio of the intersection area of the predicted and ground truth bounding boxes to their union area. Mathematically, it is expressed as: $[\mathrm{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}}]$

Area of Overlap: The area where the predicted and ground truth bounding boxes intersect. *Area of Union*: The total area covered by both the predicted and ground truth bounding boxes.

4.4.3.2 Challenges of multi-class IoU

There were several problems to tackle before creating a good IoU calculating function, one which was that I had multiple classes of objects to deal with and compare between the inferred and original bounding box results.

One possible solution considered was to calculate intersections and unions only between objects of the same class, and then take the mean average of our results.

This would not be a fair comparison however as different classes are not represented equally across an image, so we must weigh them accordingly. Also, an object might be detected incorrectly that does not actually appear on the image at all, heavily swaying our results closer to zero.

We must consider intersections only when two objects of the same class overlap, while for unions we can also consider objects that do not have any pairs between the inferred and the original solutions.

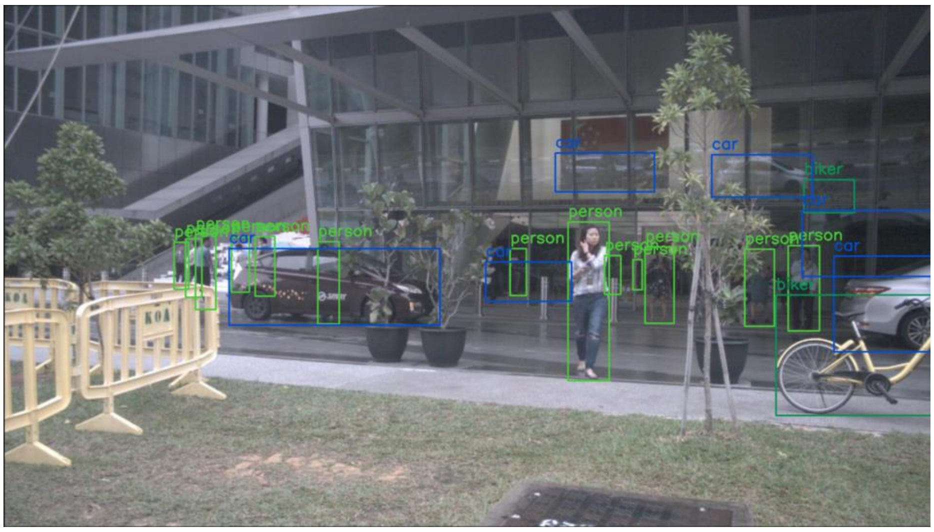


Figure 5: Example of an image with many overlapping objects of identical class from the nuImages dataset. The dataset creators' choice to include labels of reflections is also notable.

Another issue that comes up is that even when we only consider the inference results, without comparing them to the original, there may be overlaps between objects of the same class. For example, if we detect three overlapping cars (covering each other from the camera's perspective) in an image, but only two cars overlap in that location in the original data, then we must still penalize for the oversensitivity of the model.

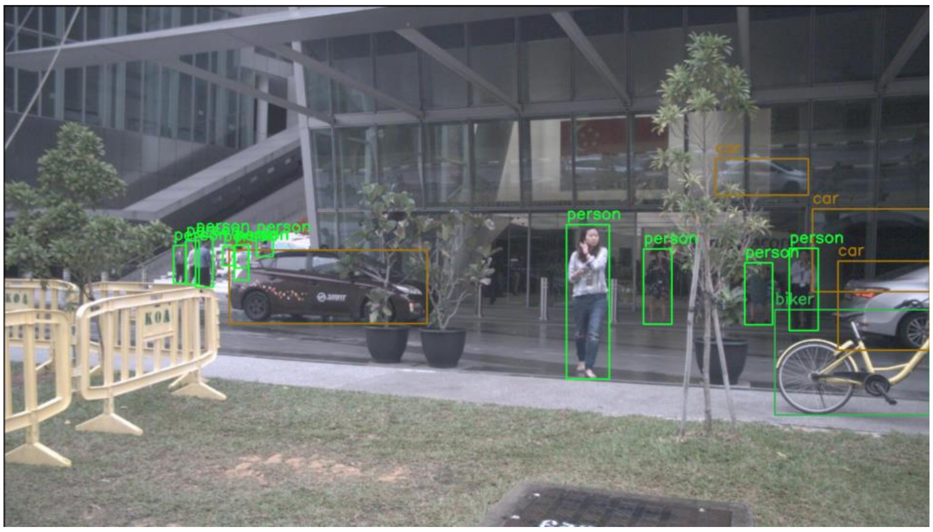


Figure 6: The results of the YOLOv8 model after inference and processing. The IoU metric of this measured against the original data was 0.725.

4.4.3.3 Solution and its implementation

The solution for this was to create a 2D integer array the size of the image for all classes of both the original and the inferred data. First, I had to convert the normalized pixel coordinates back to pixel coordinates:

```
def convert_to_pixel_coords(bbox, image_width, image_height):
    x, y, w, h = bbox
    x_min = int(x * image_width)
    y_min = int(y * image_height)
    x_max = int((x + w) * image_width)
    y_max = int((y + h) * image_height)
    return x_min, y_min, x_max, y_max
```

It still makes sense to store the bounding box coordinates normalized, as we may want to use lower image resolution for detection rather than their original sizes.

Then we increment the value in every pixel where we know an object is located:

```
def create_class_masks(objects, image_width, image_height):
    class_masks = {}
    for obj in objects:
        class_name = obj['class_name']
        if class_name not in class_masks:
            class_masks[class_name]        =        np.zeros((image_height,
image_width), dtype=int)
        x_min, y_min, x_max, y_max = convert_to_pixel_coords(obj['bbox'],
image_width, image_height)
        class_masks[class_name][y_min:y_max, x_min:x_max] += 1
    return class_masks
```

This way, if for example three cars overlap then the pixels where they do so will have an integer value of 3. After doing this for all classes, we have created weighted “masks” of the image for each of them. From then on, it is much simpler to calculate intersection and union between the different data sources:

```
def    calculate_iou(inferred_objects,    original_objects,    image_width,
image_height):
    inferred_masks=
 create_class_masks(inferred_objects, image_width, image_height)
    original_masks=
 create_class_masks(original_objects, image_width, image_height)

    total_intersection = 0
    total_union = 0

    # Calculate intersection for each class
    for class_name in inferred_masks:
        if class_name in original_masks:
            intersection=
 np.minimum(inferred_masks[class_name], original_masks[class_name])
            total_intersection += np.sum(intersection)
```

By going over each pixel’s value for both data sources and taking the lower value of the two for each class, we get a 2D map of all the intersections. For the union, we do the same but keep the higher values instead:

```
# Calculate union across all classes
    all_classes                                                         =
set(inferred_masks.keys()).union(set(original_masks.keys()))
    for class_name in all_classes:
        inferred_mask=
 inferred_masks.get(class_name,    np.zeros((image_height,    image_width),
dtype=int))
        original_mask=
 original_masks.get(class_name,    np.zeros((image_height,    image_width),
dtype=int))
        union = np.maximum(inferred_mask, original_mask)
        total_union += np.sum(union)
    iou = total_intersection / total_union if total_union > 0 else 0
```

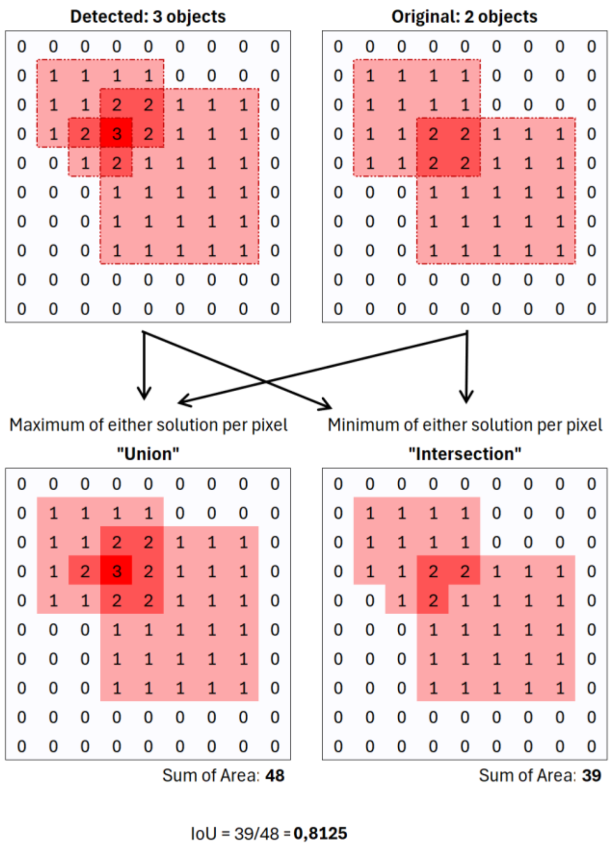


Figure 7: Visual explanation of the method used to calculate IoU in the script. The process is repeated for all classes.

4.5 Fine-tuning

For my project, I fine-tuned two object detection models: Microsoft Azure Custom Vision and YOLO Ultralytics, using the same dataset of 1000 training images and a separate validation dataset (already provided separately from the nuScenes website) for later comparison, from the nuImages dataset.

4.5.1 Microsoft Azure Custom Vision

Microsoft Azure Custom Vision is a cloud-based service that enables the creation and deployment of custom image classifiers and object detectors. I used my Azure account and created a Custom Vision project for object detection. I uploaded the 1000 training images along with their bounding box annotations and uploaded image tags for the objects using the Custom Vision API[30].

After configuring the project, I initiated the training process through the Custom Vision interface, which provided real-time feedback on metrics like precision, recall, and mean average precision (mAP). Once training was completed, I published the model to an endpoint for real-time inference[31].

Google Vision and AWS Rekognition also offer fine-tuning, but for this section for simplicity I only decided to compare one YOLO model against one SaaS offering.

4.5.2 YOLO Ultralytics Fine-tuning

For fine-tuning the YOLO (You Only Look Once) Ultralytics model, I utilized an RTX 4090 GPU rented from RunPod for efficient training. I set up the environment, installed necessary dependencies, and configured training scripts. The annotations were converted to the YOLO format. I trained the model with early stopping (patience of 10 epochs) and slight regularization (weight decay of 0.0005). The training process ensured that the model was optimized for the specific dataset.

Both models were trained using the same dataset, and a separate validation dataset was later used for performance comparison, providing insights into the strengths and weaknesses of each approach.

5 Comparative Analysis

In this chapter, we conduct a comprehensive comparative analysis of the various object detection solutions evaluated in this study. The comparisons are based on several key aspects, including inference time, Intersection over Union (IoU), cost implications, and different configurations such as batch processing and server locations. By systematically comparing the results from different runs, we aim to identify the strengths and weaknesses of each approach and provide insights into the most suitable solutions for object detection in vehicular environments.

5.1 Local YOLO Models vs. SaaS

This section compares the performance of local YOLO models with Software as a Service (SaaS) solutions. The focus is on inference time and IoU, with different resolutions and datasets being evaluated.

5.1.1 Performance comparison

When using the more accurately labeled nuImages dataset and running on 200 images as input, I had the following results:

Method	**Average Image IoU (%)**	**Optimal Confidence Threshold (%)**
Google	61.60	0
Azure	55.17	0
Rekognition	72.05	75
YOLOv5	60.19	14
YOLOv8	62.95	20

Figure 8: Results on the nuImages dataset.

Amazon Rekognition seemed to have by far the best performance among the detection methods tested in this scenario while Azure Vision performed the worst. Rekognition seemed to perform the best with an exceptionally high confidence threshold set at 75% while the local YOLO models performed the best using a very low confidence threshold value.

As Google Vision and Microsoft Azure do not allow lowering their confidence threshold, it seems like they performed best with no detected objects being filtered from their responses.

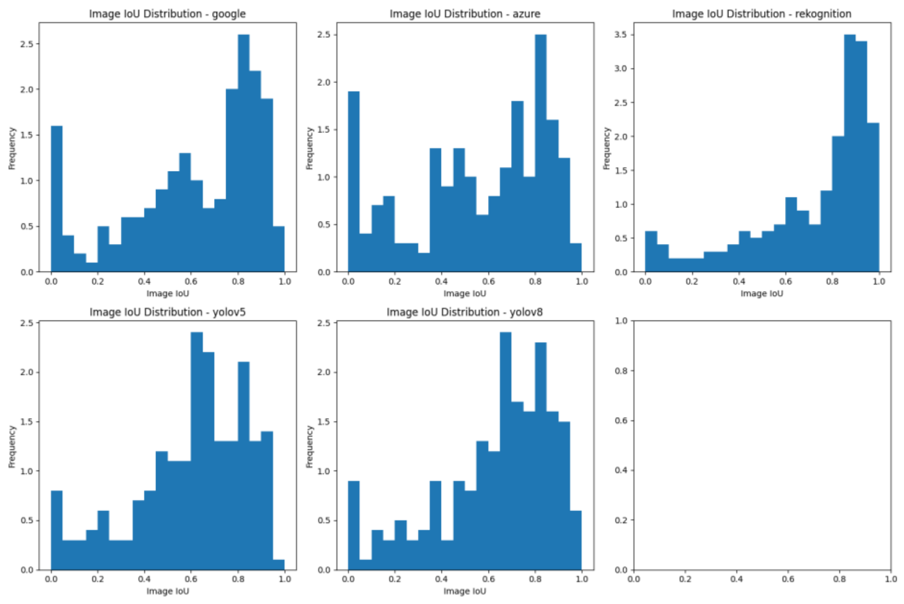


Figure 9: Distribution of IoU scores across all 200 images (nuImages) and detection methods.

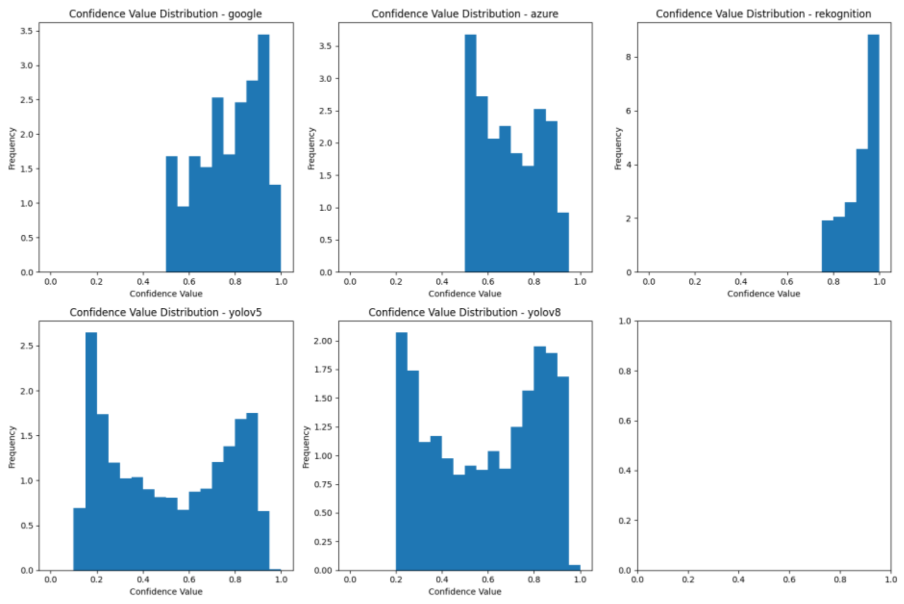


Figure 10: Distribution of confidence scores across all the detected objects.

As we can see, YOLO models tend to produce lower while Rekognition outputs higher confidence scores on this particular dataset.

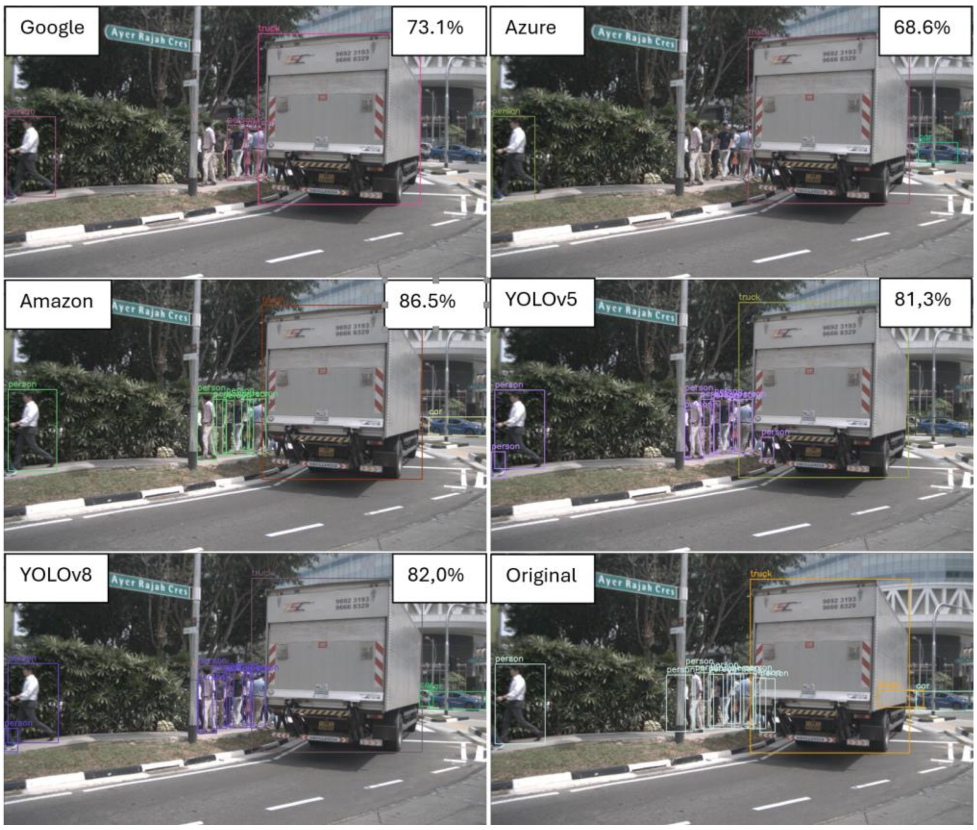


Figure 11: Comparison of all methods against the original data based on a single image, with IoU score calculated for said single image only (from nuImages dataset).

Because of the subpar quality of the Roboflow dataset compared to the nuImages one, all object localization solutions performed worse compared to our earlier results:

Method	**Avg Image IoU (%)**	**Optimal Confidence Threshold (%)**
Google	48.50%	0%
Azure	43.93%	0%
Rekognition	65.37%	82%
YOLOv5	56.98%	48%
YOLOv8	58.66%	28%

Figure 12: Comparison based on 200 images of the Roboflow dataset.

5.1.2 Speed comparison

As inference times can easily be retrieved from the YOLO models' outputs but is not provided in the server responses of SaaS providers in this case, they had to be measured differently.

In these findings, *Inference Time* when taken in the context of SaaS providers means the time until a response was received from the service provider after sending a request for inference. While in the context of local models like YOLO, it is still the "Inference time" taken straight from the model's output.

Obviously, network communications produce a considerable overhead on these results, but this is expected and will be a problem in any practical usage of these SaaS object detection methods. Still, we must consider that the inference itself is much faster than the measured timings below for SaaS and so as we will see in later section of our results, are less adversely affected by factors like resolution or even batch processing.

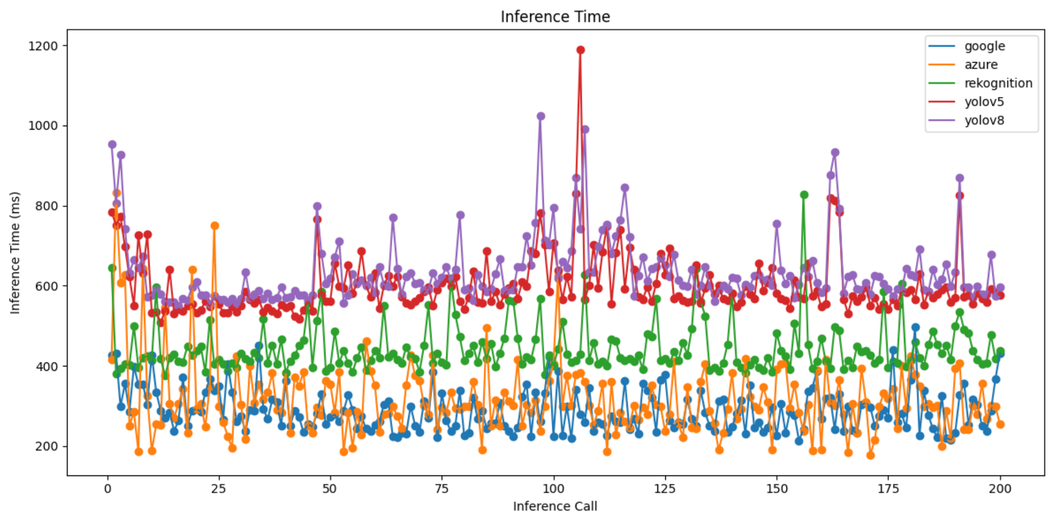


Figure 13: Inference times of all images. AWS: Frankfurt, Azure: North Switzerland; nuImages

All SaaS requests were sent from Budapest, Hungary via wired internet connection, to their respective server locations. I used the official Python libraries for each provider to upload images to cloud storage and to send object detection requests, the latter of which uses HTTP requests to communicate with their endpoints.

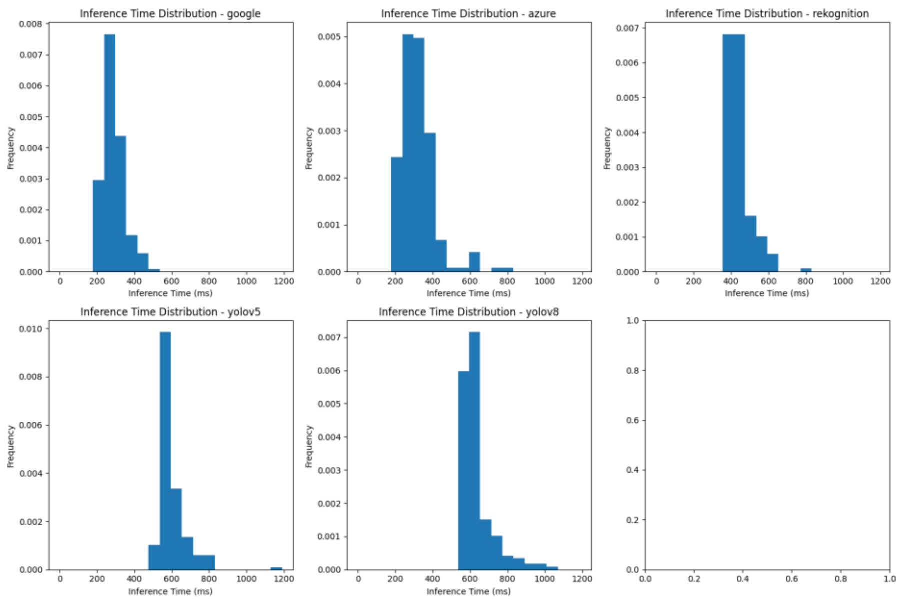


Figure 14: Distribution of inference times for all images, from the same run as Figure 13.

The specific server locations used will always be specified whenever speed measurements are mentioned. Only Google does not allow setting server locations, beyond specifying EU vs. US locations for GDPR compliance reasons.

Method	**Hardware**	**Average Inference Time (ms)**	**Average Upload Time (ms)**
Google	SaaS (Google Cloud Europe)	289.17	293.41
Azure	SaaS (Azure Northern Switzerland)	318.04	288.95
Rekognition	SaaS (AWS Frankfurt)	443.73	174.94
YOLOv5	Local (Ryzen 5 4500U)	599.91	0.00
YOLOv8	Local (Ryzen 5 4500U)	635.05	0.00
YOLOv5	Local (RTX 4090)	4.75	0.00
YOLOv8	Local (RTX 4090)	4.44	0.00
YOLOv5	Local (RTX A4000)	11.72	0.00
YOLOv8	Local (RTX A4000)	12.45	0.00

Figure 15: Comparison of upload and response times (SaaS) and inference times (YOLO) in milliseconds.

As seen from the table above, the speeds of SaaS solutions almost equal to a YOLO model running on my personal laptop, which is equipped only with a Ryzen 4500U chip and does not have a dedicated graphics card or CUDA support, only AMD integrated graphics.

When switching to higher-end and dedicated hardware, like an NVIDIA RTX 4090 or an RTX A4000 graphics card, rented from RunPod, local inference becomes orders of magnitude faster compared to SaaS solutions.

Considering the above numbers, using high-end dedicated hardware can be over a hundred times faster than sending requests to a SaaS provider.

5.1.3 Price comparison

<u>SaaS Costs per 1,000 Images:</u>

- Google Cloud Vision: $1.50
- Microsoft Azure: $1.00
- Amazon Rekognition: $1.00

<u>Prices of dedicated hardware evaluated (from Nvidia[32]):</u>

- RTX 4090: *$2050*
- RTX A4000: *$1000*

Using the minimum SaaS inference cost of $0.001 per image, the breakeven point for the RTX A4000 comes after *one million inferred images* and for the RTX 4090 at around *two million inferred images*, but other costs would need to be considered besides the graphics card, like building the whole system itself, research and development, and whether or not the environment we want to use it in (for example, a car) can even reliably support it. Reliability of a system is of course also a big concern for network communication with SaaS providers, but considerations regarding the development of a whole object detection system in a vehicular environment fall outside the scope of this thesis.

5.1.4 Different Hardware using YOLO

The performance of YOLO models can vary significantly depending on the hardware used. YOLO benefits greatly from CUDA-enabled GPUs, which allow for highly parallel processing, making the inference process extremely fast. For instance, when running on an NVIDIA RTX 4090, the average inference time for YOLOv5 and YOLOv8 drops to mere milliseconds (4.75 ms and 4.44 ms, respectively). This is a stark contrast to running the same models on a CPU like the AMD Ryzen 5 4500U, where the average inference times are significantly higher (599.91 ms for YOLOv5 and 635.05 ms for YOLOv8). The RTX A4000, while not as fast as the 4090, still offers substantial performance improvements with average inference times of 11.72 ms for YOLOv5 and 12.45 ms for YOLOv8.

These results show that dedicated hardware, especially high-end GPUs, can provide substantial speed advantages, making local YOLO models far more efficient for real-time object detection tasks compared to SaaS solutions.

5.2 Effect of Image Resolution

This section examines the impact of different image resolutions on performance. Two sets of experiments were conducted using the NuImages dataset at resolutions of 1600x900 and 800x450. Each configuration was assessed with exactly 200 images.

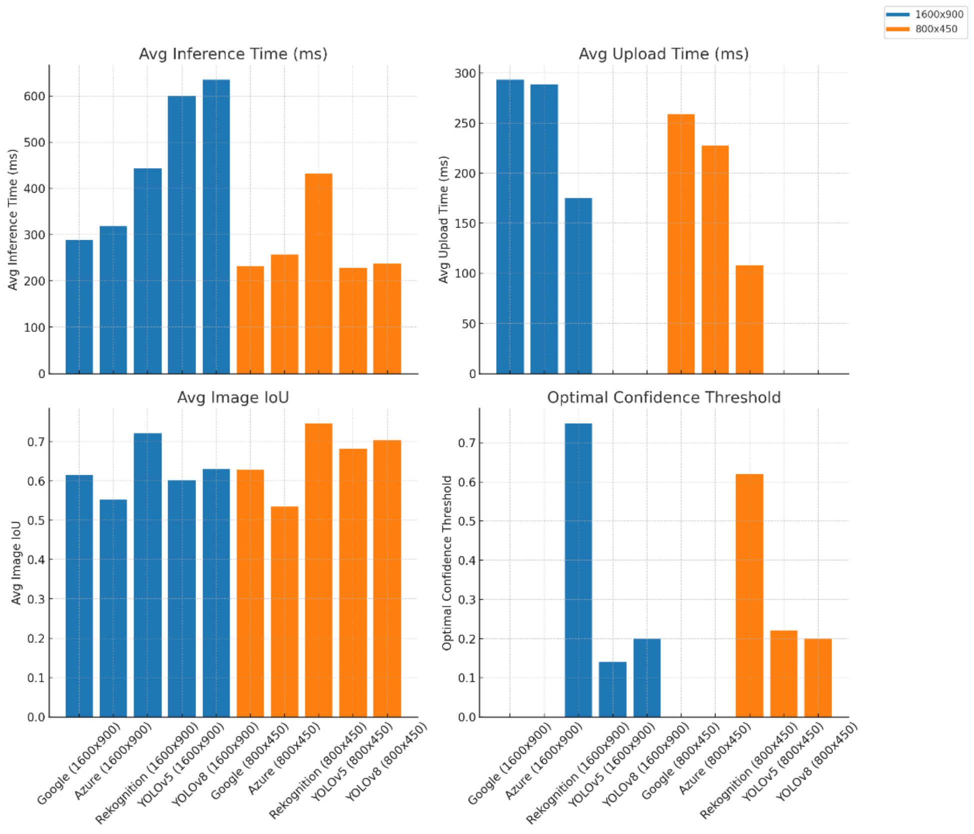


Figure 16: Performance and speed comparison between localization methods at two resolutions.

As you can see from the above graphs, most detection methods seemed to perform about the same or even worse at higher resolutions when using IoU as performance metric.

Average upload time increased as expected while "inference time" increased by a significantly larger amount for the local inference than the SaaS solutions, to almost twice as much as the lower resolution runs. This implies that there might be an advantage to using SaaS for object localization in the case of higher resolution data.

5.3 Performance Across Different Datasets

As per the results detailed in previous sections ("5.1.1.1 Performance comparison"), the quality and characteristics of the dataset significantly impact the performance of both local YOLO models and SaaS solutions. In particular, the more accurately labeled nuImages dataset generally resulted in higher IoU scores across all methods compared to the Roboflow dataset. This shows the critical role of precise annotations in achieving high detection accuracy.

When comparing the performance on the nuImages dataset, Amazon Rekognition stood out with the highest average IoU, indicating superior object detection accuracy under the optimal confidence threshold. The YOLOv8 model outperformed YOLOv5, highlighting the advancements in the newer version of the model. In contrast, Google Cloud Vision and Microsoft Azure Computer Vision had lower IoU scores, partly due to their inability to adjust confidence thresholds, which limited their performance optimization.

On the other hand, the Roboflow dataset, which has less precise annotations, yielded lower IoU scores across all methods. The average IoU scores were significantly reduced for each method, highlighting how annotation quality can influence detection performance. Despite the lower overall performance, Amazon Rekognition still led in terms of IoU but required a higher optimal confidence threshold to achieve its best performance. YOLOv8 maintained a slight edge over YOLOv5, though both models performed worse on this dataset compared to nuImages. The SaaS solutions, particularly Google and Azure, showed greater sensitivity to the quality of the dataset annotations, which affected their detection accuracy more noticeably.

5.4 Different Server Locations (SaaS)

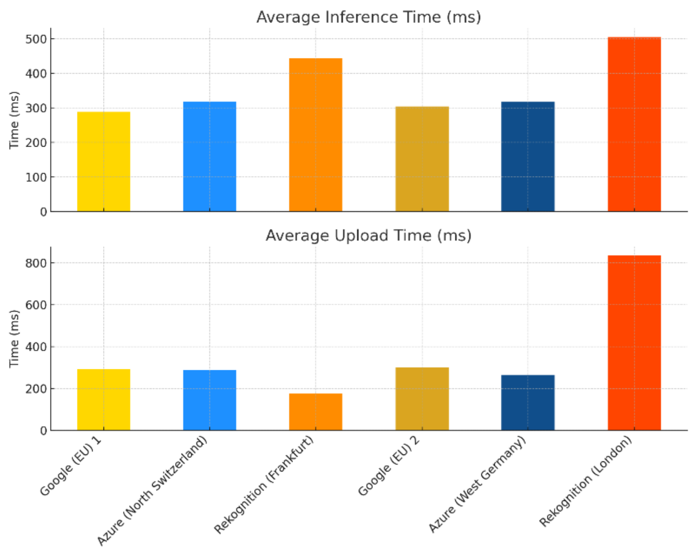


Figure 17: Time to upload and time to receive a response for inference for each provider at a set location, except Google. Data from 400 inference requests using nuImages dataset at 1600x900.

The results show varying performance across different SaaS providers and server locations. Google Cloud Vision API had moderate inference and upload times across its EU servers, giving stable but average performance. Microsoft Azure's services in North Switzerland and West Germany were similar, with slightly higher upload times in West Germany. AWS Rekognition had the biggest difference, with much longer upload times in London compared to Frankfurt. Testing from Budapest on my laptop using a wired connection ensured consistent network conditions, making the performance differences between cloud providers more apparent.

Setting up new resources for all providers was challenging, involving multiple steps for both vision APIs and storage solutions. Each provider needs different configurations, access management, and service integrations, adding complexity. Google Cloud restricts server location settings by choice, only allowing US vs. EU for GDPR compliance, and even this is only available for their OCR service[33]. This limitation reduces flexibility in optimizing performance based on specific regional needs compared to AWS and Azure, which offer more precise geographic options.

5.5 Byte Array vs. Upload and Reference (SaaS)

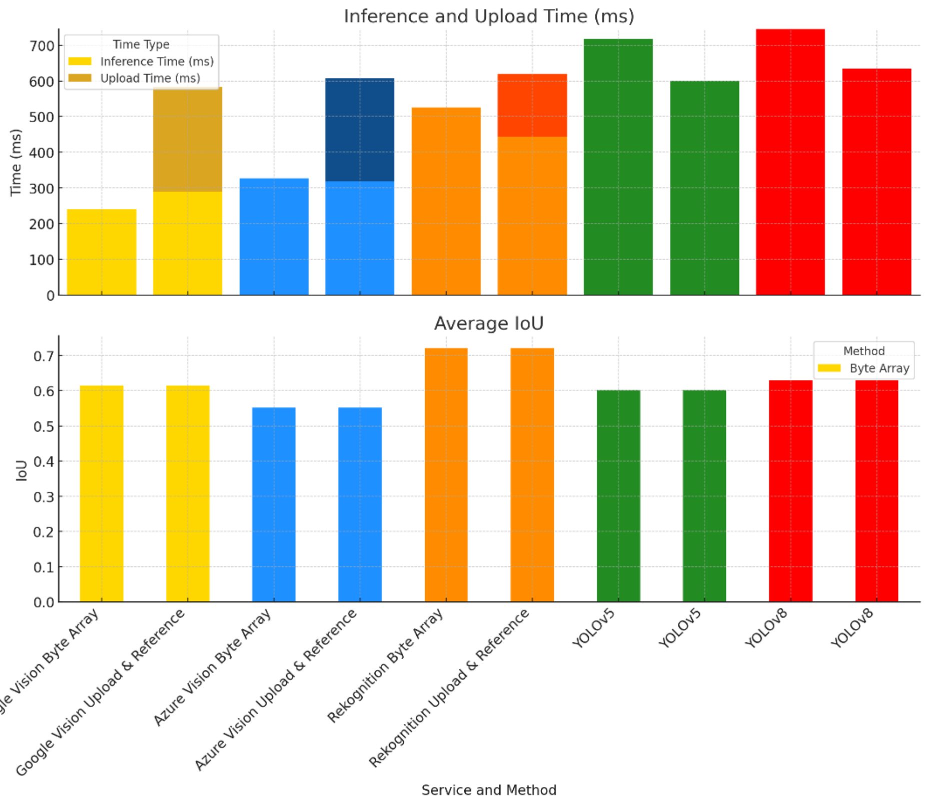


Figure 18: Graph comparing response times of upload and refer method vs. sending to object detection service as byte array.

For Google Vision, the byte array method results in faster total processing time (240.39 ms) compared to the upload and reference method (582.58 ms). For Azure Vision, the upload and reference method also has a much higher total time (606.99 ms vs. 326.61 ms).

Amazon Rekognition had closer results between the two runs but was still slower when uploading separately.

The YOLO models were run using the same settings both times, the slight time variations must be due to environmental factors independent of dataset and resolution.

All methods were assessed on 200 images each, on the same images twice, as can be seen from the identical IoU results. YOLO models ran on a laptop equipped with only an AMD Ryzen 4500U.

Overall, sending the images as bytes arrays in the HTTP request seems to be faster than uploading them to the providers' respective cloud storage solutions and then referencing them in the request. That is why it is preferable to use the first method for better response times, unless we want to also store the images for later in cloud storage. This can also be useful for object detection done in batches, detailed in the next section.

5.6 Batch Inference

Out of the three SaaS object detection providers, only Google Vision supports batch requests. Two types are supported[34]:

- Online (synchronous) requests, where up to 16 images can be sent in a single batch and a response will be sent immediately.
- Offline (asynchronous) requests, where up to 2000 images can be sent for annotation but they are scheduled only, and no immediate response is sent to the caller.

I evaluated the performance of synchronous requests, dividing the 200 images into batches of 16 images, uploaded them to Google Cloud Storage and the ran object detection on them:

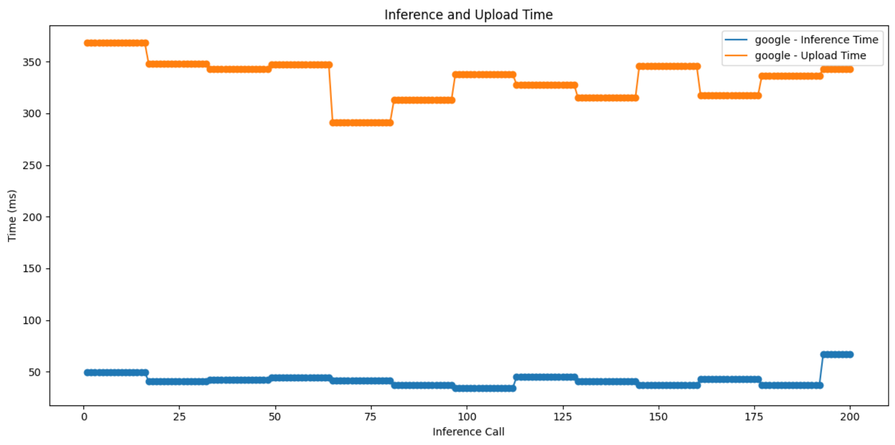


Figure 19: Google Vision object labeling done in batches, inference and upload times averaged to single images.

As the graph above shows, when considering the average time it took for a single image to be evaluated the batch solution had much faster inference times compared to single image annotations running sequentially. Upload times stayed about the same on average but inference times when greatly reduced, comparable to a locally running YOLO model on dedicated hardware.

This shows that Google Vision API is capable of using parallelization to its advantage for object localization jobs and this is an advantage of it over its competitors who do not offer such solution.

5.7 Fine-tuning

Name	**Inference Time**	**Optimal Confidence Threshold**	**Avg IoU**
Googel Vision	154.32 ms	0%	60.69%
Azure Vision	770.91 ms	0%	54.05%
AWS Rekognition	1271.57 ms	56%	71.20%
YOLOv5	6.91 ms	20%	57.27%
YOLOv8	6.20 ms	17%	60.00%
YOLOv8 (Fine-tuned)	6.12 ms	42%	**62.59%**
Azure Custom Vision (Fine-tuned)	1769.55 ms	83%	**70.56%**

Figure 20: Table showing the averaged-out results of a run on 200 images of nuImages' validation subset. Local YOLOs' training and prediction ran on an RTX 4090 equipped RunPod instance.

As we can see from the above results, the fine-tuned models were able to outperform their original counterparts, with Azure Custom Vision gaining an over 16% in average IoU score over the default Azure Vision, almost reaching up to Amazon Rekognition in performance.

6 Conclusion

6.1 Summary of Findings

Our analysis revealed that Amazon Rekognition consistently outperformed other solutions in IoU metrics, especially with higher confidence thresholds. Microsoft Azure had the lowest performance. Local YOLO models, while effective, showed significant performance improvements with high-end GPUs, achieving much faster inference times than SaaS solutions.

Cost-wise, SaaS solutions are economical for lower volumes, but local solutions become more cost-effective with high usage, despite their higher upfront costs.

Higher image resolutions increased inference times more for local solutions than for SaaS, showing SaaS efficiency at handling high-resolution images.

Server location and processing methods affected performance, with AWS and Azure offering more regional flexibility than Google Cloud. Sending images as byte arrays generally resulted in faster response times compared to uploading and referencing.

Google’s batch inference capability demonstrated significant performance gains through parallel processing, surpassing its competitors and enhancing efficiency for large-scale tasks.

With fine-tuning I was able to further improve the performance of these models, showing that it is a worthwhile investment to make our detection models application specific.

These findings highlight the trade-offs between SaaS and local solutions, providing guidance for selecting the most suitable approach for vehicular object detection applications.

6.2 Further Research

To enhance the scope of this research, further studies should explore evaluating a broader range of hardware configurations. Evaluating performance across different types of CPUs and GPUs could offer deeper insights into the scalability and efficiency of object detection models. In addition, incorporating more models, beyond those currently analyzed, would allow for a comprehensive understanding of their strengths and weaknesses in various contexts.

Moreover, testing with more varied datasets and scenarios would improve the robustness of the findings. Using datasets with different types of objects, environments, and image qualities could help in assessing the generalizability of the models. Better fine-tuning techniques should also be explored, including optimizing hyperparameters and using advanced training algorithms. Furthermore, fine-tuning other models and SaaS solutions could reveal their potential when tailored to specific applications, potentially unlocking higher performance and accuracy.

References

[1] *Neural Networks and Deep Learning* by Michael Nielsen, http://neuralnetworksanddeeplearning.com/ (revision 21:31, 29 May 2024)

[2] Coursera: *AI and Machine Learning articles*, https://www.coursera.org/articles/category/ai-and-machine-learning (revision 21:33, 29 May 2024)

[3] Deepai: *Understanding Hidden Layers in Neural Networks*, https://deepai.org/machine-learning-glossary-and-terms/hidden-layer-machine-learning (revision 21:34, 29 May 2024)

[4] AWS: *What is Overfitting?*, https://aws.amazon.com/what-is/overfitting/ (revision 21:35, 29 May 2024)

[5] Datacamp: *What is Deep Learning?* , https://www.datacamp.com/tutorial/tutorial-deep-learning-tutorial (revision 21:36, 29 May 2024)

[6] Folio3AI: *Object Detection in 2023: The Definitive Guide*, https://www.folio3.ai/blog/object-detection-guide/ (revision 21:38, 29 May 2024)

[7] V7labs: *The Ultimate Guide to Object Detection*, https://www.v7labs.com/blog/object-detection-guide (revision 21:40, 29 May 2024)

[8] Datacamp: *YOLO Object Detection Explained*, https://www.datacamp.com/blog/yolo-object-detection-explained (revision 21:41, 29 May 2024)

[9] Ultralytics: *Ultralytics YOLOv8 Docs*, https://docs.ultralytics.com/quickstart/ (revision 21:43, 29 May 2024)

[10] LearnOpenCV: *YOLOv8 : Comprehensive Guide to State Of The Art Object Detection*, https://learnopencv.com/ultralytics-yolov8/ (revision 21:44, 29 May 2024)

[11] Microsoft: *Azura AI Vision*, https://azure.microsoft.com/en-us/products/ai-services/ai-vision (revision 21:45, 29 May 2024)

[12] Google: *Cloud Vision documentation*, https://cloud.google.com/vision/docs (revision 21:51, 29 May 2024)

[13] AWS: *How Amazon Rekognition works*, https://docs.aws.amazon.com/rekognition/latest/dg/how-it-works.html (revision 21:55, 29 May 2024)

[14] Jupyter: *Project Jupyter Documentation*, https://docs.jupyter.org/en/latest/ (revision 22:11, 29 May 2024)

[15] Google: *Colaboratory Frequently Asked Questions* https://research.google.com/colaboratory/faq.html (revision 22:12, 29 May 2024)

[16] Visual Studio Code Docs: *Python Interactive* window https://code.visualstudio.com/docs/python/jupyter-support-py (revision 22:12, 29 May 2024)

[17] RunPod: *Runpod Documentation: Overview*, https://docs.runpod.io/overview (revision 22:13, 29 May 2024)

[18] Microsoft: *Azure AI Vision pricing*, https://azure.microsoft.com/en-us/pricing/details/cognitive-services/computer-vision/ (revision 22:15, 29 May 2024)

[19] Google: *Cloud Vision pricing*, https://cloud.google.com/vision/pricing (revision 22:16, 29 May 2024)

[20] AWS: *Amazon Rekognition pricing*, https://aws.amazon.com/rekognition/pricing/ (revision 22:17, 29 May 2024)

[21] Clarifai: *Pricing*, https://www.clarifai.com/pricing (revision 22:18, 29 May 2024)

[22] IBM Cloud: *2020 What's new*, https://jp-tok.dataplatform.cloud.ibm.com/docs/content/wsj/getting-started/whats-new-2020.html (revision 22:21, 29 May 2024)

[23] AWS: *Lookout for Vision about page*, https://aws.amazon.com/lookout-for-vision/ (revision 22:22, 29 May 2024)

[24] Github: *Ultralytics' YOLOv5 releases*, https://github.com/ultralytics/yolov5/releases (revision 22:32, 29 May 2024)

[25] Google Colab: *Official Ultralytics YOLOv8 Tutorial*, https://colab.research.google.com/github/ultralytics/ultralytics/blob/main/examples/tutorial.ipynb (revision 22:34, 29 May 2024)

[26] A2D2: *Audi Autonomous Driving Dataset*, https://arxiv.org/pdf/2004.06320v1 (revision 22:40, 29 May 2024)

[27] Roboflow: *Udacity Self Driving Car Dataset*, https://public.roboflow.com/object-detection/self-driving-car (revision 22:41, 29 May 2024)

[28] nuScenes: *A multimodal dataset for autonomous driving*, https://arxiv.org/pdf/1903.11027 (revision 22:45, 29 May 2024)

[29] Personal GitHub repository containing all Jupyter notebook code used for inference and all test run results: https://github.com/fgusztav/thesis2024 (revision 01:45, 30 May 2024)

[30] Microsoft: *Use cases for Custom Vision*, https://learn.microsoft.com/en-us/legal/cognitive-services/custom-vision/custom-vision-cvs-transparency-note (revision 22:46, 29 May 2024)

[31] Microsoft: *How to improve your Custom Vision model*, https://learn.microsoft.com/en-us/azure/ai-services/custom-vision-service/getting-started-improving-your-classifier (revision 22:47, 29 May 2024)

[32] Nvidia: *Nvidia Store*, https://store.nvidia.com/en-us/ (revision 22:48, 29 May 2024)

[33] Google Cloud: *Multi-regional support*, https://cloud.google.com/vision/docs/ocr#regionalization (revision 22:51, 29 May 2024)

[34] Google Cloud: *Small batch file annotation online*, https://cloud.google.com/vision/docs/file-small-batch (revision 22:52, 29 May 2024)

Annex

Object Detection Functions

Setting up all clients:

```
# Amazon Rekognition
rekognition_client = boto3.client(
    "rekognition",
    aws_access_key_id=AWS_ACCESS_KEY_ID,
    aws_secret_access_key=AWS_SECRET_ACCESS_KEY,
    region_name=AWS_REGION
)
s3_client = boto3.client(
    "s3",
    aws_access_key_id=AWS_ACCESS_KEY_ID,
    aws_secret_access_key=AWS_SECRET_ACCESS_KEY,
    region_name=AWS_REGION
)

# Google Vision
google_client = vision.ImageAnnotatorClient(
    client_options={"api_key":      GOOGLE_API_KEY,      "quota_project_id":
GOOGLE_PROJECT_ID}
)
storage_client = storage.Client(project=GOOGLE_PROJECT_ID,
                                client_options={"api_key":  GOOGLE_API_KEY,
"quota_project_id": GOOGLE_PROJECT_ID})

# Microsoft Azure
azure_client           =           ComputerVisionClient(MICROSOFT_ENDPOINT,
CognitiveServicesCredentials(MICROSOFT_SUBSCRIPTION_KEY))
```

AWS Rekognition

Inference request sent as byte array:

```
def detect_objects_with_rekognition(image_path: str) -> dict:
    with open(image_path, "rb") as image_file:
        image_bytes = image_file.read()

    start_time = time.time()
    response = rekognition_client.detect_labels(
        Image={"Bytes": image_bytes},
        MinConfidence=REKOGNITION_CONFIDENCE_THRESHOLD
    )
    inference_time = (time.time() - start_time) * 1000
    response["inference_time"] = inference_time
    return response
```

Uploading first to S3 bucket, then referencing:

```
def detect_objects_with_rekognition(image_path: str) -> dict:
    bucket_name = # S3 bucket’s name
    object_key = os.path.basename(image_path)

    start_upload_time = time.time()
    s3_client.upload_file(image_path, bucket_name, object_key)
    upload_time = (time.time() - start_upload_time) * 1000

    start_time = time.time()
    response = rekognition_client.detect_labels(
        Image={"S3Object": {"Bucket": bucket_name, "Name": object_key}},
        MinConfidence=REKOGNITION_CONFIDENCE_THRESHOLD
    )
    inference_time = (time.time() - start_time) * 1000
    response["inference_time"] = inference_time
    response["upload_time"] = upload_time
    return response
```

Google Vision

Inference request sent as byte array:

```
def detect_objects_with_google_vision(image_path: str) -> dict:
    bucket_name = "your-bucket-name"
    blob_name = os.path.basename(image_path)

    bucket = storage_client.bucket(bucket_name)
    blob = bucket.blob(blob_name)

    start_upload_time = time.time()
    blob.upload_from_filename(image_path)
    upload_time = (time.time() - start_upload_time) * 1000

    image                                                             =
vision.Image(source=vision.ImageSource(gcs_image_uri=f"gs://{bucket_name}/
{blob_name}"))
    start_time = time.time()
    response = google_client.object_localization(image=image)
    inference_time = (time.time() - start_time) * 1000

    return   {"response":   response,   "inference_time":   inference_time,
"upload_time": upload_time}
```

Batch Processing:

```
    # Perform batch inference on all images
    image_requests = [
        vision.AnnotateImageRequest(
image=vision.Image(source=vision.ImageSource(gcs_image_uri=f"gs://{bucket_
name}/{blob.name}")),

features=[vision.Feature(type=vision.Feature.Type.OBJECT_LOCALIZATION,
max_results=GOOGLE_MAX_RESULTS)]
        )
        for blob in blobs
    ]

    start_time = time.time()
    vision_responses=
google_client.batch_annotate_images(requests=image_requests)
```

```
    inference_time  =  (time.time()  -  start_time)  *  1000   #  Convert  to
milliseconds
```

Deleting the blob:

```
# Clean up the blob
blob = bucket.blob(response[blob_name])
blob.delete()
```

Microsoft Azure Computer Vision

Sequential Byte Array Upload:

```
def detect_objects_with_azure(image_path: str) -> dict:
    with open(image_path, "rb") as image_file:
        content = image_file.read()

    image_data = io.BytesIO(content)
    start_time = time.time()
    analysis       =       azure_client.analyze_image_in_stream(image_data,
visual_features=[VisualFeatureTypes.objects])
    inference_time = (time.time() - start_time) * 1000
    return     {"analysis":     analysis.as_dict(),     "inference_time":
inference_time}
```

Upload to Cloud Storage and Then Inference:

```
    container_name = # name of our storage container
    blob_service_client  =  BlobServiceClient.from_connection_string("your-
connection-string")
    container_client=
blob_service_client.get_container_client(container_name)
    blob_name = os.path.basename(image_path)
    with open(image_path, "rb") as data:
        blob_client = container_client.get_blob_client(blob_name)
        blob_client.upload_blob(data, overwrite=True)

    image_url=
f"https://{blob_service_client.account_name}.blob.core.windows.net/{contai
ner_name}/{blob_name}"
    analysis=
azure_client.analyze_image(image_url,visual_features=[VisualFeatureTypes.o
bjects])
```

YOLOv5/v8

```
def detect_objects_with_yolov5(image_path: str) -> dict:
    image = cv2.imread(image_path)
    results  =  yolov5_model(image_path,  conf=YOLOV5_CONFIDENCE_THRESHOLD,
imgsz=image.shape[:2])
    return {
        "results": results,
        "inference_time": results[0].speed['inference']
    }
```

Conversion of Images

```
desired_y_resolution = 900
for image_file in os.listdir(original_image_folder):
    if image_file.endswith(".png") or image_file.endswith(".jpg"):
        image_path = os.path.join(original_image_folder, image_file)
        image = PIL.Image.open(image_path)

        original_width, original_height = image.size

        new_width   =   int((desired_y_resolution   /   original_height)   *
original_width)
        resized_image = image.resize((new_width, desired_y_resolution))

        out_p = os.path.join(image_folder, image_file)
        resized_image.save(out_p)
```

Class Name Conversion and Filtering

```
def convert_class_names(objects):
    class_mapping = {
        "car": ["automobile", "taxi", "vehicle", "suv", "jeep", "sedan",
"van", "land vehicle", "vehicle.car", "vehicle.emergency.police"],

        "truck": ["truck", "lorry", "bus", "shuttle bus", "pickup truck",
"vehicle.truck",         "vehicle.bus.bendy",         "vehicle.bus.rigid",
"vehicle.trailer", "vehicle.construction", "vehicle.emergency.ambulance"],

        "person":         ["person",         "pedestrian",         "human",
"human.pedestrian.adult",                         "human.pedestrian.child",
"human.pedestrian.construction_worker",
"human.pedestrian.personal_mobility",    "human.pedestrian.police_officer",
"human.pedestrian.stroller", "human.pedestrian.wheelchair"],

        "biker": ["bicycle", "bike", "biker", "motorcycle", "motorbike",
"vehicle.bicycle", "vehicle.motorcycle"]
    }

converted_objects = []
    for obj in objects:
        class_name = obj["class_name"].lower()
        obj["class_name"] = class_name
        for common_name, alt_names in class_mapping.items():
            if class_name == common_name or class_name in alt_names:
                obj["class_name"] = common_name
                converted_objects.append(obj)
                break
    return converted_objects
```

Confidence Thresholds Optimization

```
def find_optimal_confidence_thresholds(inference_results, bbox_data,
threshold_step=0.01):
    optimal_thresholds = {}
    for method in inference_results[0].keys():
        max_avg_iou = 0
        optimal_threshold = 0
        for threshold in np.arange(0, 1 + threshold_step, threshold_step):
            total_iou = 0
            image_count = 0
            for inference_result in inference_results:
                image_file =
os.path.basename(inference_result[method]["image_id"])
                bbox_data_item = next((data for data in bbox_data if
data["image_id"] == image_file), None)
                if bbox_data_item is not None:
                    filtered_objects = [obj for obj in
inference_result[method]["objects"] if obj["confidence"] >= threshold]
                    image_iou = calculate_iou(filtered_objects,
bbox_data_item["objects"])
                    total_iou += image_iou
                    image_count += 1
            avg_iou = total_iou / image_count if image_count > 0 else 0
            if avg_iou > max_avg_iou:
                max_avg_iou = avg_iou
                optimal_threshold = threshold
        optimal_thresholds[method] = round(optimal_threshold, 2)
    return optimal_thresholds
```