# MASTER THESIS

# ENHANCING MACHINE LEARNING DEVELOPMENT EFFICIENCY THROUGH DEVOPS AND MLOPS

A thesis submitted for the degree of

Master of Science in Computer Science

| | |
|---|---|
| Student: | Rajesh Kumar Ramadas<br>rajesh-kumar.ramadas@iu-study.org |
| Matriculation: | 92125100 |
| University: | IU International University of Applied Sciences<br>Juri-Gagarin-Ring 152 · D-99084 Erfurt |
| University Supervisor: | Dr. Aditya Mushyam<br>aditya.mushyam@iu.org |
| Project Repository: | https://github.com/RajeshRamadas/yolov8-model-optimization.git |
| Submission Date: | April 2025 |

## Contents

## List of Figures

## Abstract

The rapid integration of machine learning (ML) has revealed pain points in the ML development lifecycle. This thesis investigates the benefits of integrating DevOps principles with MLOps, ML specific practices to change the speed as well as reliability of machine learning systems. This research focuses on solving the common pain points regarding the development lifecycle: version control, experiment tracking, reproducibility, and deployment automation; and addresses problems in the literature by providing a complete way of thinking about modern ML development workflows, in the form of a framework. By reading the related literature, implementing MLOps through a drive and professionalism, it was determined that traditional software engineering approaches would be ineffective for ML practice, as each project generally has unique requirements for data versioning, experiment tracking, and dedicated infrastructure supporting the MLOps lifecycle beyond coding shorter strategies. By providing a complete MLOps architecture containing DevOps practices, along with some MLOps specific components such as data validation pipelines, automated hyperparameter optimization, and a model registry; we provide research evidence with a confirmatory case study that operationalized a satisfying CI/CD pipeline using Jenkins that was able to implement a YOLOv8 object detection model optimization. This included the automation of dataset validation and augmentation, neural architecture search (NAS), parallel training strategies, standardized evaluation and consistent evaluation metrics across environments. The initial research idea developed from edge deployments with successful deployments of optimized models on Raspberry Pi hardware, demonstrating the potential to leverage MLOps practices in such resource constrained edge computing environments for a cost-effective, low latency vehicle detection and tracking system. This research makes a theoretical and practical implementation contribution to organizations wishing to advance their ML development practice into a mature state. Further, the modular approach to the architecture allows for incremental adoption in potentially different ways and practices through various contexts and ML use cases ranging from cloud infrastructure to edge devices. By adopting the complete lifecycle of ML systems regarding their data preparation and production deployment at both server and edge deployments, the formalization and structure established in the research can assist organizations to streamline their ML development lifecycle process and reduce time to value. The research provides avenues for organizations to improve the dependability and governance of ML model deployment in production environments.

# Chapter 1: Introduction

## 1.1 Background

At an incredible pace, Machine Learning (ML) is being used in companies to change processes and create new opportunities for businesses in current scenarios. Traditional approaches to developing ML have not provided any structure around version control, reproducibility, or deployment. The nature of working on ML projects is that it is experimental, and therefore without any sound conceptualization around repeatability in workflow, experimentation and the iteration drives, have made the framework untenable in any productive, operationally growing, environment. MLOps is more than following the DevOps principles of collaboration and automation to manage ML projects. MLOps takes DevOps principles one step further by applying the practices of DevOps directly to the ML lifecycle. MLOps aims to simplify the experimentation of models, the deployment of models, and monitoring for models-indirectly making it easier for organizations to build working systems in production.

An Algorithmia report stated that organizations that practiced MLOps were 22% faster at building models, and 25% more performant (Algorithmia, 2021). An IDC report found that with mature MLOps practices, organizations reduced time to deployment as much as 30% compared to others with less sophisticated ML pipelines (IDC, 2022). The Deloitte report found that of the organizations defined as AI high performers, 64% had advanced MLOps practices, whereas only 28% of the AI beginners had something to show as advanced MLOps plans (Deloitte, 2023). The benefits of MLOps practices are clear, but TwimIcon's State of ML in Production report highlights how MLOps adoption is uneven across organizations and that less than half, 40%, of organizations claim that their ML production processes are mature (TwimIcon, 2022).

## 1.2 Problem Statement

Even with the advancements in ML, many teams are having inefficiencies in the overall development- and deployment lifecycle. There are several reasons for the inefficiencies:

- Fragmented workflow - the data scientists are often completely decoupled from the engineering team. The data scientists are working separately and developing the model, while the engineering team is developing the workflow for deployment, which means the dev and the deploy workflows are also separate.

- Manual experimentation - most data scientists will use a spreadsheet (or notebook) to track their experiments and hyperparameters and this makes for very poor reproducibility.

- Inconsistent deployment process - the data scientist is often asked (or simply expected) to develop the workflow that spans development to production. This is typically a manual process and >1 opportunity for error.
- Poor versioning - versioning software is not optimized for ML artifacts (e.g. datasets, models).
- Training data class imbalance - datasets are often class imbalanced, which leads to bias in models that are trained on these datasets, and models typically perform poorly on 'minority' classes at evaluation.

All of this contributes to inefficiencies for deploying ML solutions that must be delivered with speed and scale, which ultimately costs time, wasted resources, abandoned projects, and poorly performing models when they are put into production.

## 1.3 Objectives

This study will pursue the previously mentioned aims through the following study activities:

- Identifying gaps in the existing traditional ML workflow: Identifying and documenting the real pain points and inefficiencies in the existing ML development and deployment workflows.
- Applying DevOps and MLOps practices to ML model development: Building a complete architecture to apply DevOps to ML models on the wealth of opportunities for automating ML development practices.
- Evaluating impact to efficiency of development: Evaluating impact to development practices with documentation using measurements of time to develop, reproducibility of deployments, and reliability of deployments.
- Providing a practical implementation: Using a practical implementation of an automated ML pipeline to de-mystify the architecture.

Providing practical implementation guidelines with template recommendations for organizations developing from traditional to MLOps development. In addition, to address data quality, we will automate data validation and augmentation which informs model training.

## 1.4 Scope

This study focuses primarily on supervised ML models and DevOps and MLOps tools in cloud-native environments. The study centers on computer vision models, primarily YOLOv8 object detection, as a familiar, commonplace, and relevant example of new ML application. The principles are generic, but this study is empirically focused on practice, application, and implementation instead of developing a theoretical model, thereby utilizing tools and methods so that organizations can adopt them as directly as possible. The evaluation is checking measures like development efficiency, reproducibility, and reliability of deployment, and not model performance improvements; although it is expected that model performance improvements will present as an anticipated (if fortunate) by-product of the new process.

# Chapter 2: Literature Review

## 2.1 DevOps in Software Engineering

DevOps is reshaping how software is delivered by breaking down barriers between development and operations teams. Some of the key concepts in DevOps include:

Continuous Integration/Continuous Delivery (CI/CD): Automating build, testing, and deployment will reduce the manual code push to production while ensuring the same code is deployed every time. a CI/CD tools such as Jenkins, CircleCI, and GitHub Actions are some of the most used CI/CD toolchain for automation.

Monitoring and feedback: Monitoring applications and infrastructure continuously allows for learning what works in what conditions, as well as troubleshooting issues or gaps as the surface prior to turning into substantial egress. Common tools and systems today for monitoring and feedback are Prometheus, Grafana and the ELK stack.

Version Control: Using tools such as Git and GitHub (or GitLab) allows teams have an organized and systematic way to track code changes, enable collaboration, and importantly attribute who is performing the code changes.

An analysis for a 2023 DORA report showed that organizations that exhibited mature DevOps practices deployed code 208 times more than their low DevOps maturity counterparts and were able to recover from incident 24 times faster when compared to low DevOps maturity organizations.

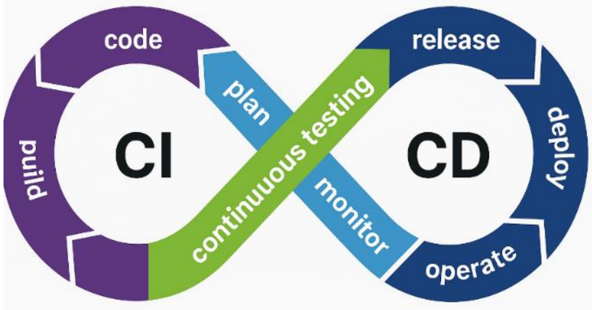


*Figure 1: A generic CI/CD pipeline.*

## 2.2 Introduction to MLOps

MLOps takes DevOps principles and applies them to the Machine Learning specific aspects. The core parts of MLOps are:

- Data Version Control: MLOps tools, such as Data Version Control (DVC), tracking data sets along with the code for experiments help to promote reproducibility over' time.
- Model Registry: A centralized location for housing versions, with some metadata on the models training / performance conditions may help support proper governance.
- Automated Model Training: Scheduling or triggering processes where model retraining occurs when it is determined there has been data drift and / or, model performance has degraded since deployment -in other words, keeping ML models fresh over time.
- Model Serving Infrastructure: In this evolution of MLOps solutions, we also define the data science model serving infrastructure. Areas of specialization like model often rely on tools and frameworks as defined in MLOps best practices to follow proper operational practices.
- Model Monitoring: In production data science models, we can track their performance metrics, predictions for new client (data) distributions, and possible drift and / or degradation like that we foresee in model performance metrics.

As documented by Meessen-Pinard et al. (2022), "Automation servers, such as Jenkins, are the backbone of most contemporary MLOps implementations because they provide a consistent, repeatable structure to what otherwise would be an ad hoc process of machine learning model development."

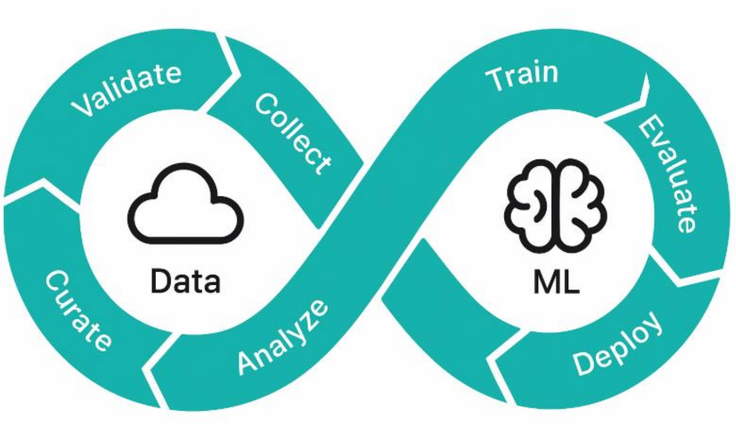


*Figure 2: A generic MLOps*

## 2.3 Challenges in ML Lifecycle

Machine Learning projects involve unique challenges that are not suited to traditional DevOps:

- Model Reproducibility: There are major challenges with making models retrain and produce the same results, as they depend on the data, code, and the environment used in training.
- Dataset Versioning: Versioning data with traditional versioning systems can be problematic if they involve large datasets, which results in large repository sizes and specialized requirements for data versioning.
- Experiment Management: The way ML development is executed involves the completed studies on many experiments (many variations of parameters, architectures, data preprocessing steps) and they all need to be tracked.
- Training Infrastructure: ML training, including deep learning, often uses specialized hardware (GPUs/TPUs) that can support training requirements of larger ML projects and capabilities of distributed learning.
- Product scalability: Production ML systems need to consider the changing workload as well as maintaining performance throughout deployments to varying environments.
- Monitoring: ML models need to be monitored on several parameters, such as Confidence Drift, Data Drift, Performance Drift, etc. Testing: Traditional software testing methodologies do not cover ML-specific items such as model performance, model bias, and model robustness.

- Class Imbalance: In their study, Buda et al. (2018) mention the significance of class imbalance in deep-learning models where classes with the lowest representation on balance and they performed badly because they had fewer representative examples.

Additionally, Sculley et al. (2015) remind us that "the most serious forms of technical debt in machine learning systems can build up rapidly and without anyone noticing if it is not approached with good engineering practice." At the end of the day, implementing an MLOps approach is a much more structured method to apply for ML projects.

## 2.4 Related Works

Several studies have proposed MLOps pipelines and frameworks with different emphases.

- Kreuzberger et al. (2022) presented a comprehensive MLOps maturity model with four levels (i.e., manual processes to fully automated ML systems).

- Mäkinen et al. (2021) described possible issues with the integration of ML models in an existing continuous integration/continuous delivery (CI/CD) pipeline and noted the need for bespoke tools and approaches best suited for the introduction of ML. Renggli et al. (2021) presented a framework for the continuous integration of machine learning applications while emphasizing issues of reproducibility.

- Shankar et al. (2022) from Google described an architecture of production ML systems, while emphasizing the central importance of monitoring while using feedback loops.

- Oksuz et al. (2020) conducted a survey of recent literature and highlighted that class imbalance is one of the main issues in object detection systems, and pointed out that “foreground-foreground” class imbalance is frequently ignored.

- Amershi et al. (2019) advocated for different practices for continuously managing ML pipelines at scale, while putting an emphasis on validation, testing, and continuous evaluation, to support machine learning production workflows.

While these theoretical frameworks provide great insights, there is a very limited focus on measuring the impact of MLOps on development efficiency when realized via practice. This thesis will then evaluate this gap in the literature by way of case studies and as a practical implementation of a complete MLOps pipeline for object detection models.

## 2.5 DevOps vs MLOps: Implementation Status

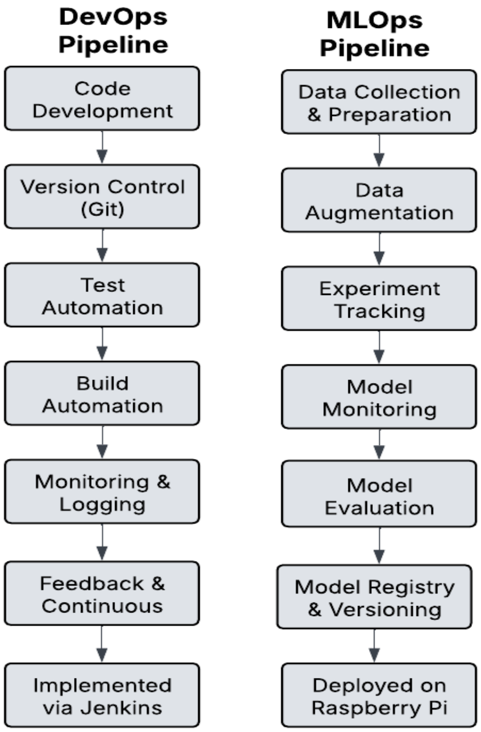


*Figure 3:DevOps vs MLOps Implementation Status.*

A comparative examination of development and operations activities across traditional software systems engineering and the development of machine learning systems will demonstrate many differences in challenges and addressing them, particularly with respect to the MLOps delivery environment.

# Chapter 3: Methodology

## 3.1 Architecture Overview

This study utilizes a MLOps architecture, consisting of the following components:

- Source Control: Git for version control of code, configuration, and pipeline definitions, with GitHub as the host for the repository.
- CI/CD: Jenkins for orchestration of the end-to-end pipeline that automates build, test, train, and deploy.
- Storage: Amazon S3 for model artifacts, datasets, and experiment results.

The architecture provides modularity - components can be substituted or updated while maintaining the ability to execute the pipeline as each step is independent of the others. This modularity allows an organization to implement only a portion of the framework and build on it without having to change their existing process completely.

## 3.2 MLOps Architecture Diagram

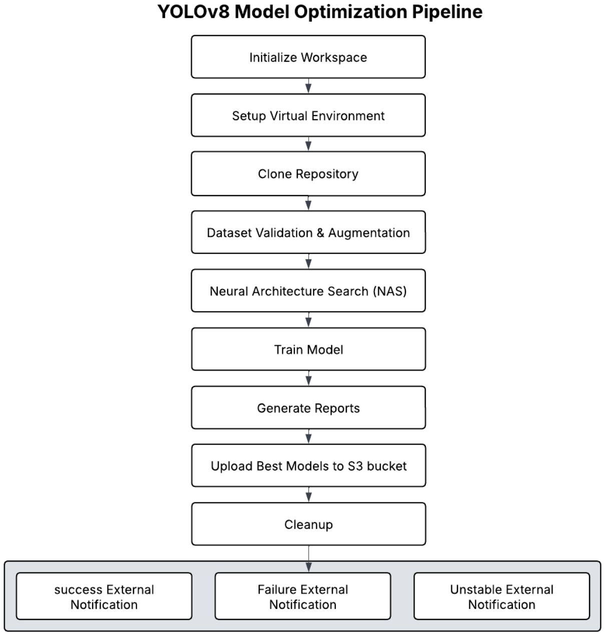

YOLOv8 Model Optimization Pipeline
Initialize Workspace
Setup Virtual Environment
Clone Repository
Dataset Validation & Augmentation
Neural Architecture Search (NAS)
Train Model
Generate Reports
Upload Best Models to S3 bucket
Cleanup
success External Notification
Failure External Notification
Unstable External Notification

*Figure 4:MLOps Architecture Diagram.*

An example of an end-to-end pipeline - from ingesting data to deploying the model that has integration points to separate development and operations

## 3.3 Pipeline Design

The ML pipeline consists of several major subcomponents:

1. **Data Ingestion & Preprocessing:**
   - Validating the dataset for data integrity / testing data quality
   - Cleaning and preprocessing the data
   - Applying targeted augmentation on imbalanced classes

2. **Training the model with experiment logging and tracking in mind**

    - Conducting Neural Architecture Search (NAS) to find the optimal model architecture
    - Hyperparameter tuning
    - Training multiple versions of the model in parallel Model Validation and model **artifact** storage

3. **Validation against validation datasets**

    - Capture performance metrics (accuracy, speed, model size)
    - Store model artifacts along with additional metadata

4. **CI/CD triggered deployment**

    - Automated testing of model functional capabilities
    - Exports to multiple deployment formats - Delivery to production environment

## 3.4 Tools and Technologies

The implementation takes advantage of the following technologies:

Programming Language: Python, the open-source language for ML with a large library and framework ecosystem for ML development.

ML Libraries:

- Ultralytics YOLOv8 for creating the object detection model
- Scikit-learn for traditional ML algorithms
- Pandas and NumPy for data manipulation

MLOps Tools:

- -Jenkins to run continuous integration/continuous deployment pipelines
- Custom data validation and data augmentation modules
- Neural Architecture Search (NAS)

Framework Cloud Infrastructure:

- AWS S3 for all storage
- CI/CD runs on dedicated servers

The focus on open-source tools allows avoiding vendor lock-in and assures availability, and regardless of whether these discrete parts are open-source, the Logical architecture also supports proprietary options too.

# Chapter 4: Implementation

## 4.1 Dataset and Problem Statement

To illustrate the MLOps framework, I decided to tackle a computer vision problem with an object detection dataset of vehicles. My aim was to obtain an optimized YOLOv8 model capable of detecting and classifying the various vehicle types in images and videos with good accuracy and decent inference speed.

I chose this use case because it fits the typical ML scenario and has many critical attributes: A large enough dataset (4,058 images) with efficient preprocessing and augmentation the training was computation heavy with specialized hardware Multiple metrics to optimize (accuracy, speed, model size) Some potential class imbalance among 12 vehicle classes (big bus, big truck, bus-l-, bus-s-, car, mid truck, small bus, small truck, truck-l-, truck-m-, truck-s-, truck-xl-)

The need for continual improvements as more labeled data comes in the dataset came from Roboflow's "vehicles-q0x2v" project (version 2) which is under CC BY 4.0 license and is part of Intel's RF100 benchmark. The dataset uses YOLOv8 annotation format where each vehicle is represented by a single line in a text file containing five values: class id and normalized bounding box coordinates (center_x, center_y, width, height). This annotation format is optimized for training YOLO object detection models and provides an efficient representation of the 12 vehicle classes present in 4,058 images.

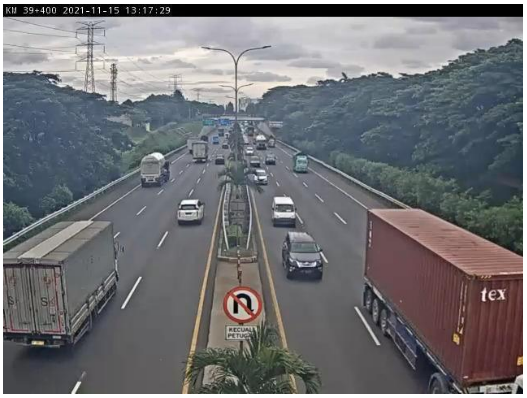
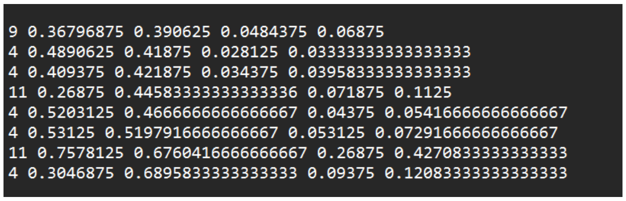


*Figure 5:Dataset Sample Image*

An example of an image from the vehicles data set utilized to train the object detection model which contains multiple categories of vehicles in their native environment.



```
9 0.36796875 0.390625 0.0484375 0.06875
4 0.4890625 0.41875 0.028125 0.03333333333333333
4 0.409375 0.421875 0.034375 0.03958333333333333
11 0.26875 0.44583333333333336 0.071875 0.1125
4 0.5203125 0.4666666666666667 0.04375 0.05416666666666667
4 0.53125 0.5197916666666667 0.053125 0.07291666666666667
11 0.7578125 0.6760416666666667 0.26875 0.4270833333333333
4 0.3046875 0.6895833333333333 0.09375 0.12083333333333333
```

*Figure 6:Annotation Sample*

An example of the YOLO format bounding boxes coordinates and class labels from vehicles detected which illustrate the annotation structure used to train the model.

## 4.2 Jenkins Pipeline Implementation

A pipeline was developed with Jenkins to automate the complete end-to-end ML workflow. The pipeline consists of 11 stages, as follows:

1. Initialize Workspace - Setting up the environment for ML pipeline and creating a workspace for artifact storage.

2. Create Virtual Environment - Establish a Python virtual environment, complete with all associated libraries, dependencies and configurations.

3. Download Dataset - Download dataset from repository (in this case assigned Google Drive).

4. Unpack and Validate the Dataset - Unpack dataset and validate the structure, content and quality to assess the dataset against the expected quality standards.

5. Create a Model Optimization Repository - Clone the repository that contains the ML optimization code.

6. Validate and Augment in Parallel - Validate the dataset assessing basic structural and content quality and consider moderated/targeted data augmentation to account for imbalance, running the validation and augmentation in parallel.

7. Validate Augmented Dataset -Validate augmented dataset for quality and when validated meets structure, content and quality standard.

8. Neural Architecture Search (NAS) - Perform a search against many different model architectures, in the hopes an automated retrieval of the best optimized configurations is found.

9. Model Performance - generate best performance model(s) above against validation datasets.

10. Upload Best Model to S3 - archive best model to Cloud storage.

11. 11. Reports Generation - Maintain detail reporting to document workflow and reporting results.

The pipeline consists of several significant MLOps features:

- Parameterization: Users can configure model name, dataset source, number of NAS trials and epochs through Jenkins parameters.

- Artifact Management: All relevant outputs are saved as Jenkins artifacts for traceability.

- Error Handling: The pipeline has extensive error handling so that at any stage if an error occurs, it will not stop the entire process.

- Notification System: Email notifications sent upon success or failure with detailed report.

This approach matches Amershi et al. (2019) suggested practices for governing ML pipelines at scale, particularly with the focus on validation, testing, and ongoing evaluations.

### 4.3 Dataset Validation and Augmentation

An integral part of the implementation is the dataset validation and augmentation module. The *dataset_validator.py* script performs thorough investigations of the dataset structure as follows: • Corrupted images

- Missing label files

- Invalid label formats

- Unbalanced class distribution

The targeted_augmentation.py script addresses class imbalances through intelligent augmentation; in which It assesses the class distribution to rank minority classes based on a configurable threshold It applies more aggressive augmentation to underrepresented classes It generates additional synthetic examples by applying a multiplication factor, it generates and balances dataset with proper train/validation/test splits It generates full reports of augmentations

The augmentation pipeline combines various types of transformations:

- Geometric transformations (rotation, scaling, affine transforms)
- Photometric adjustments (brightness, contrast, hue)
- Environmental simulations (shadows, fog, snow)

This heterogeneous transformation set produces many unique training instances that help with a better generalization of the model, which was shown in the work of Cubuk et al., (2019) with AutoAugment.

Shorten and Khoshgoftaar (2019) indicate that augmentation strategies that specifically target the minority classes (rather than general augmentation) may be more effective when working with an imbalanced dataset.

Based on our findings, we also indicated large performance improvements by many of the minority classes due to having a targeted data augmentation strategy.

### 4.3.1. Targeted Data Augmentation for Imbalanced Object Detection Datasets.

A targeted data augmentation strategy specifically developed to address the class imbalance characteristics of an object detection dataset. This meant rather than performing the augmentation uniformly across the classes, only the minority classes were augmented to produce a more balanced training distribution.

### 4.3.2. Class Imbalance Assessment

The system first assesses the dataset for class distributions to account for the issue of class imbalance. This problem often occurs with object detection because some of the objects may be scarce in the training data. Buda et al. (2018) have identified class imbalance as a factor in the performance of deep neural networks, and minority classes suffer from a severe lack of using representative sufficient examples.

### 4.3.3. Target Augmentation Methodology

Instead of augmenting the dataset randomly, the implementation will provide a targeted augmentation approach that will:

- Identify the minority classes based on a configurable threshold,
- Implement more aggressive augmentations aimed specifically towards the under-represented classes
- Utilizes a multiplication factor to generate more synthetic examples. This is in line with what was found by Shorten and Khoshgoftaar (2019) where targeted augmentation increased the performance of underrepresented classes more than if done uniformly.

### 4.3.4. Multi-Transformation Augmentation Pipeline

- The augmentation pipeline allows for the inclusion of multiple transformation types to be performed on the dataset as a whole:
- Geometric transformations (rotation, scaling, affine transformations),
- Photometric transformations (brightness, contrast, hue),
- Environmental types of transformations (shadows, fog, snow).

The collection of different types of transformations creates a diverse training example that facilitate the model's ability to generalize. This is supported by work done by Cubuk et al. (2019) in Auto Augment, which creates training examples of underlying data distributions.

### 4.3.5. Design Philosophy

The design follows a modular approach with clear separation between:

- Analysis (detecting imbalance)
- Augmentation (addressing imbalance)
- Reporting (reporting differences)

This difference in concerns allows for application to various datasets while sticking to the original targeted augmentation strategy.

### 4.3.6. Relation to Research Literature

Class Imbalance in Object Detection This implementation is attempting to address a well-researched problem in object detection. Oksuz et al. (2020) provided an extensive coverage of class imbalance and heavily emphasized it as one of the top three challenging problems in object detection systems. In fact, the authors stated that "foreground-background" imbalance has attracted significant attention in the literature, but "foreground-foreground" class imbalance (the motivation of this implementation) is equally important and has been relatively under-researched.

### 4.3.7. Data Augmentation Effectiveness

The strong augmentation pipeline is reinforced by work by Zoph et al. (2020), who found that stronger augmentation is most effective for smaller datasets and underrepresented classes. They found that aggressive augmentation can improve performance on minority classes significantly without degrading performance on the majority classes.

Below are two training models confusion matrix with same training configuration only difference with and without dataset augmentation.

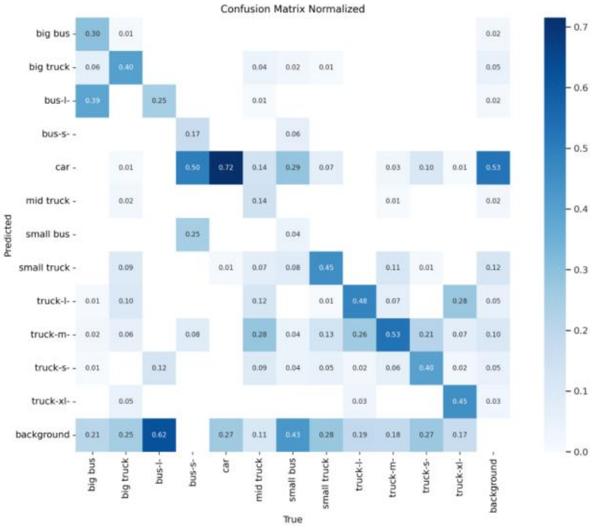


*Figure 7:Confusion matrix without data augmentation with mAP@50: 0.43*

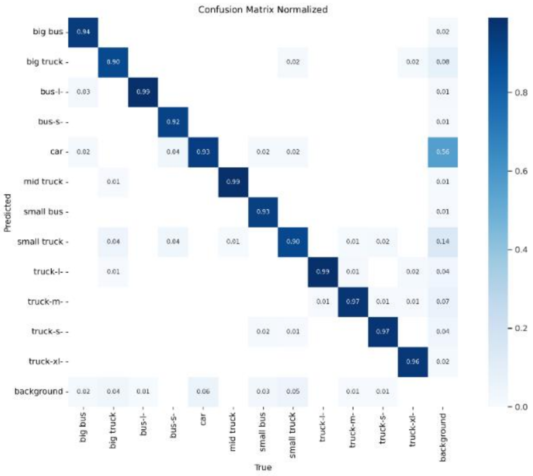
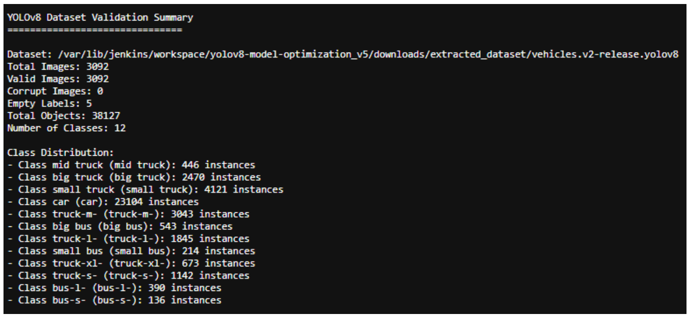


*Figure 8:Confusion matrix without data augmentation with mAP@50: 0.97*

## 4.3.8. YOLO-Specific Considerations

The implementation is focused on YOLO format datasets and explicitly acknowledges the need to keep appropriate bounding box transformations because of augmentation. This is supported by work by Dwibedi et al. (2017), who recognized the need to preserve the accuracy of annotations during the synthetic data generation phase for object detection.



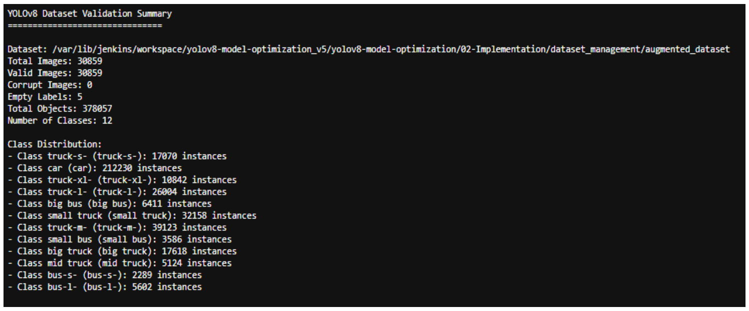
```
YOLOv8 Dataset Validation Summary
================================

Dataset: /var/lib/jenkins/workspace/yolov8-model-optimization_v5/downloads/extracted_dataset/vehicles.v2-release.yolov8
Total Images: 3092
Valid Images: 3092
Corrupt Images: 0
Empty Labels: 5
Total Objects: 38127
Number of Classes: 12

Class Distribution:
- Class mid truck (mid truck): 446 instances
- Class big truck (big truck): 2470 instances
- Class small truck (small truck): 4121 instances
- Class car (car): 23104 instances
- Class truck-m- (truck-m-): 3043 instances
- Class big bus (big bus): 543 instances
- Class truck-l- (truck-l-): 1845 instances
- Class small bus (small bus): 214 instances
- Class truck-xl- (truck-xl-): 673 instances
- Class truck-s- (truck-s-): 1142 instances
- Class bus-l- (bus-l-): 390 instances
- Class bus-s- (bus-s-): 136 instances
```

*Figure 9:Dataset before augmentation.*

An example of the distribution of classes in the data set as they were originally, which illustrates how highly imbalanced the data was, with the car class being the bulk.



```
YOLOv8 Dataset Validation Summary
================================

Dataset: /var/lib/jenkins/workspace/yolov8-model-optimization_v5/yolov8-model-optimization/02-Implementation/dataset_management/augmented_dataset
Total Images: 30859
Valid Images: 30859
Corrupt Images: 0
Empty Labels: 5
Total Objects: 378057
Number of Classes: 12

Class Distribution:
- Class truck-s- (truck-s-): 17070 instances
- Class car (car): 212230 instances
- Class truck-xl- (truck-xl-): 10842 instances
- Class truck-l- (truck-l-): 26004 instances
- Class big bus (big bus): 6411 instances
- Class small truck (small truck): 32158 instances
- Class truck-m- (truck-m-): 39123 instances
- Class small bus (small bus): 3586 instances
- Class big truck (big truck): 17618 instances
- Class mid truck (mid truck): 5124 instances
- Class bus-s- (bus-s-): 2289 instances
- Class bus-l- (bus-l-): 5602 instances
```

*Figure 10:Dataset after augmentation.*

An example of the distribution classes after undertaking the targeted augmentations and gives a more balanced representation of the vehicle classes consistent of a increase representation of the minority categories.

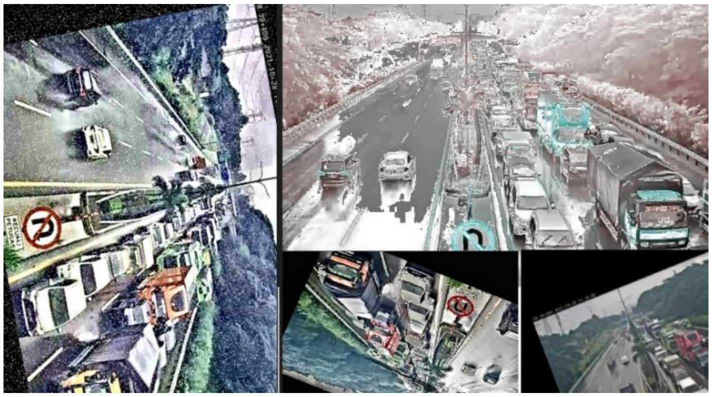


*Figure 11:Augmentation image.*

An example of the transformation types done to generate additional training examples to increase the representation of the minority classes, representing geometric and photometric transformations.

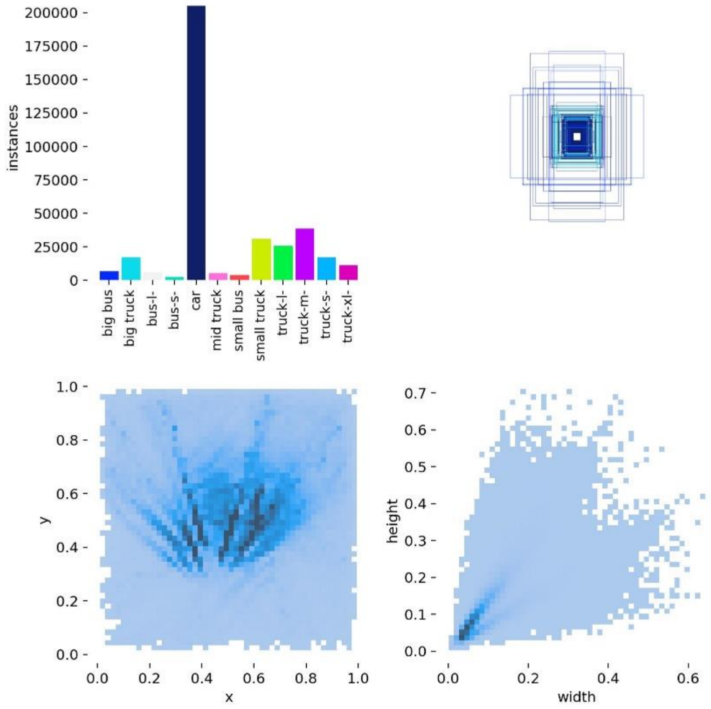


*Figure 12:YOLO dataset distribution visualization*

## 4.4 Neural Architecture Search and Model Optimization

Neural Architecture Search (NAS) is an automated way to find the best neural network architectures. In this report, I will cover a YOLOv8 NAS implementation optimized for object detection models with respect to accuracy, inference speed, and model size. The framework will exhaustively search for modeling variations to determine the best architecture for a specific deployment scenario.

The goal of Neural Architecture Search Neural architecture search is a major step forward for AutoML (Automated Machine Learning) and allows for the automatic discovery of neural network architectures that outperform human designed alternatives (Zoph & Le, 2017). In object detection, especially in deployment scenarios where computational power is limited, it is important to achieve the best possible compromise between detecting accuracy, inference speeds, and model size.

The YOLOv8 NAS framework presented in this report automates this search for a balance by:

1. Specifying a search space of architectural parameters

2. Training several variant models

3. Interpreting results to identify the best architectures

4. Providing visualization of parameters to conceptualize how they impact a training outcome

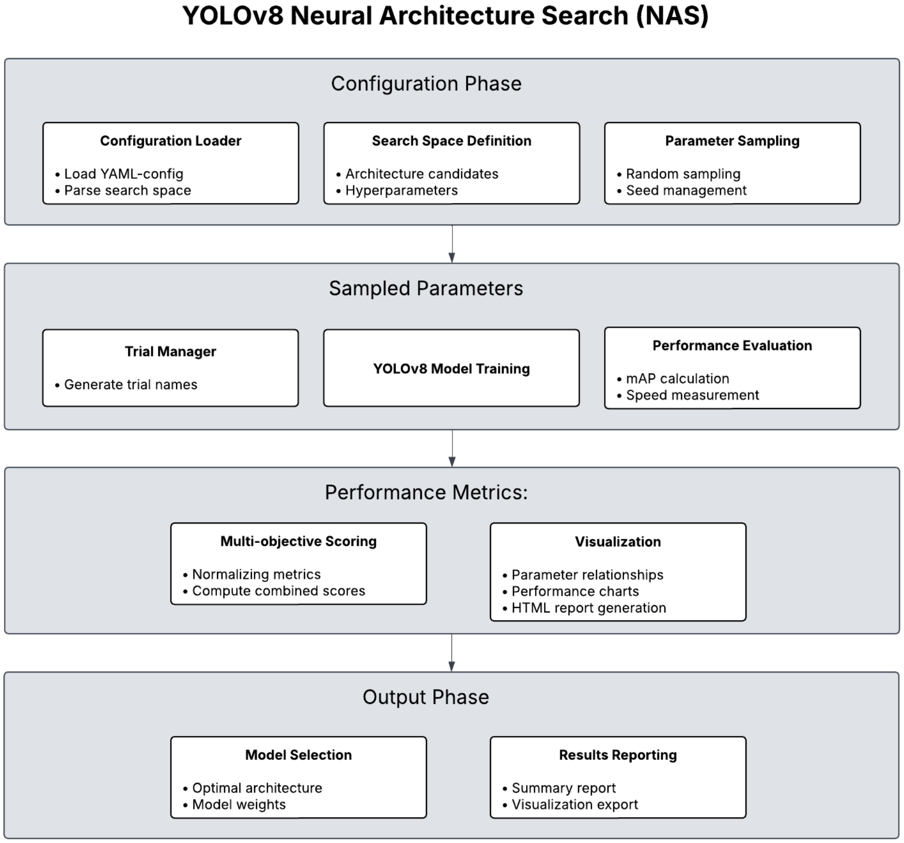


*Figure 13:NAS Block Diagram.*

A high-level architectural representation of the Neural Architecture Search framework with configuration management, trial runs and metrics

### 4.4.1. Core Components

These are the components in the NAS framework that interact with each other:

1. Configuration Management (config_loader.py):
    - Loads parameters in the search space from a YAML configuration
    - Connects to simple and complicated search space
2. Tracks the default arguments and objective weights Trial Execution (trial_manager.py):
    - Generates all the trial scripts for every architectural configuration
    - Executes each individual trial with the assigned parameters
3. Collects and saves performance metrics for the trial Result Analysis (analyzer.py):
    - Reads trial results and analyzes the trials to find the best architecture
    - Calculates overall performance scores based on using weighted objectives
4. Produces summary statistics of overall search Visualization (visualization.py):
    - Visualizes parameters and relationships
    - Produces reports with HTML formatted summaries of search results
5. Distinguishes relationships between the importance of parameters Utility (utils.py):
    - JSON (de)serialization
    - Directory structure creation
    - Aggregates scores with normalized metrics

### 4.4.2. Search Space Definition

The search space is defined in search_space.yaml and includes:

Basic Parameters:

- depth_multiple: Controls model depth (0.33, 0.5, 0.67, 1.0)
- width_multiple: Controls model width/channels (0.25, 0.5, 0.75, 1.0)
- img_size: Input image resolution (320, 448, 640)
- kernel_size: Convolution kernel size (1, 3, 5, 7)

Advanced Parameters:

- optimizer: Training optimizer (SGD, Adam, AdamW)
- learning rate parameters (lr0, lrf)
- momentum: Optimizer momentum (0.8, 0.9, 0.95, 0.99)
- weight_decay: Regularization strength (0.0005, 0.001, 0.0001)
- augmentation parameters (warmup_epochs, augment, mosaic)
- model_type: Base model architecture (yolov8n, yolov8s)

### 4.4.3. Technical Workflow

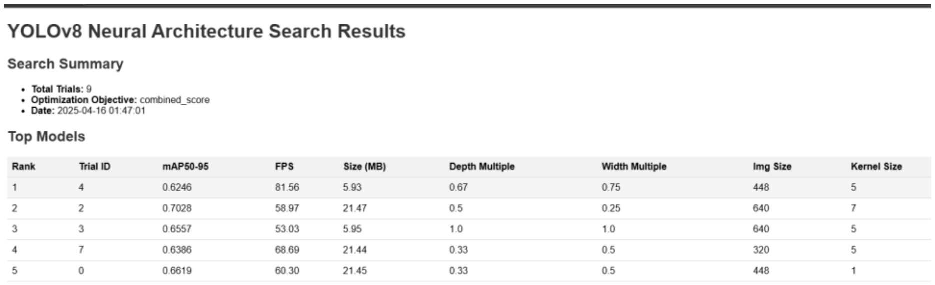
- The search configuration is loaded and processed first.
- Trial Generation The framework generates trials by randomly sampling parameters from the search space, similar to a sampling method described in MnasNet (Tan et al. , 2019).
- Trial Execution Each trial involves: - Custom YOLOv8 configuration is generated based on sampled parameters - The model is trained on the supplied dataset

- Evaluation metrics (mAP, inference speed, model size) are calculated A combined score is then calculated for multi-objective optimization Results Analysis After the trials are completed, the results can be analyzed to determine which architectures were optimal, in a way similar to approaches described in DetNAS (Chen et al. , 2019).

### 4.4.4. Results Visualization

The framework creates visualizations to gain a better understanding and examine the interaction among the parameters:

1. Depth vs Width effect: scatter plot of the performance impact with the depth and width
2. Performance vs Model size: result of accuracy based upon model size
3. Performance vs inference speed: setting up the accuracy/speed tradeoffs
4. Kernel Size effect: bar chart of effects according to kernel size Parameter Importance; parameters that are correlated to the optimization objectives



## YOLOv8 Neural Architecture Search Results

### Search Summary

- **Total Trials:** 9
- **Optimization Objective:** combined_score
- **Date:** 2025-04-16 01:47:01

### Top Models

| Rank | Trial ID | mAP50-95 | FPS | Size (MB) | Depth Multiple | Width Multiple | Img Size | Kernel Size |
|---|---|---|---|---|---|---|---|---|
| 1 | 4 | 0.6246 | 81.56 | 5.93 | 0.67 | 0.75 | 448 | 5 |
| 2 | 2 | 0.7028 | 58.97 | 21.47 | 0.5 | 0.25 | 640 | 7 |
| 3 | 3 | 0.6557 | 53.03 | 5.95 | 1.0 | 1.0 | 640 | 5 |
| 4 | 7 | 0.6386 | 68.69 | 21.44 | 0.33 | 0.5 | 320 | 5 |
| 5 | 0 | 0.6619 | 60.30 | 21.45 | 0.33 | 0.5 | 448 | 1 |

*Figure 14:NAS Model Summary*

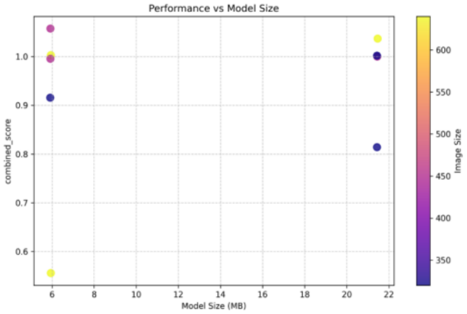


*Figure 15:Performance Vs Size*

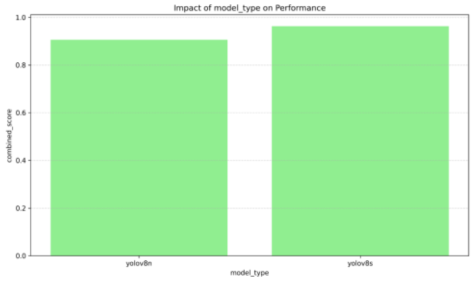


*Figure 16:Model Type Impact*

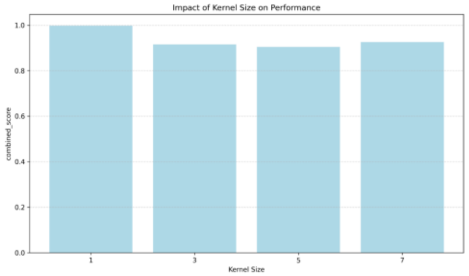


*Figure 17:Kernel Size Impact*

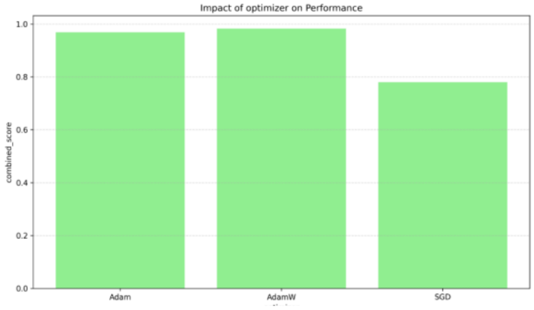


*Figure 18:Optimizer Impact*

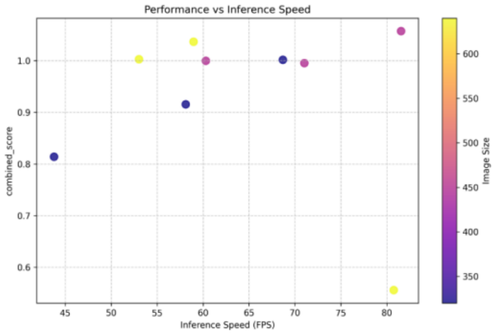


*Figure 19:Performance Vs Speed*

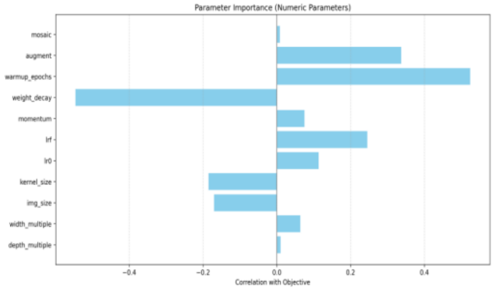


*Figure 20:Parameter Importance*

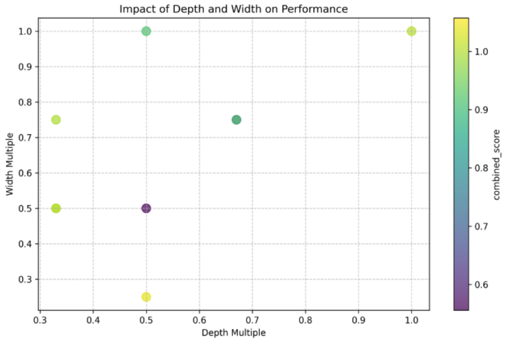


*Figure 21:Depth Width Scatter*

### 4.4.5. Experimental Training

The *trainyolov8model.py* file contains the code for training models based on architectures discovered through NAS. When NAS discovers an optimal neural architecture, it is vital to continue training the model fully, to allow it to achieve its full capability. This report gives a brief overview on extended training of YOLOv8 models,

Why is Extended Training Important Under-Trained NAS Models - In most instances, a model discovered through NAS will have only been trained until 5-50 epochs. This number of epochs is below the number needed for convergence. Performance Gap - Studies show that further training of optimal architectures can improve mAP by 5-15 absolute points.

Hyper-parameter optimization - In extended training, you can tune the learning rates, regularization, and augmentation properly.

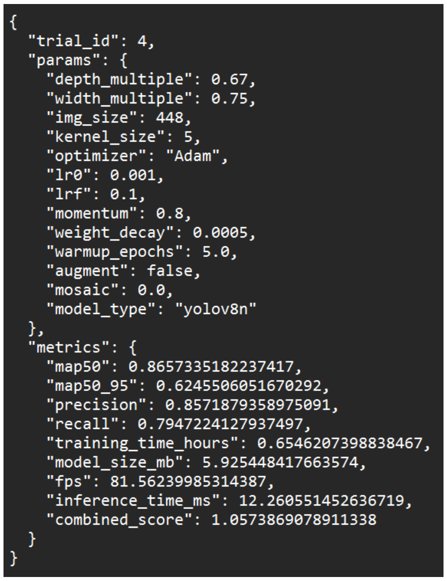
### Extended Training Protocol Training Duration

- Baseline to recommended: 100 epochs minimum
- Standard: 300 epochs recommended
- Complex datasets: 500+ epochs for challenging cases Optimization Strategy
- Learning Rate: 0. 01 with cosine decay to 0. 001
- Batch size: 16-64 based on available GPU memory
  - Optimizer: AdamW with weight decay of 0. 01

### Implementation in Pipeline

Add an extended training stage to the Jenkins pipeline with these key parameters:

```
--model ${BEST_MODEL_FROM_NAS}
--epochs 300
--batch-size 32
--optimizer AdamW
--lr0 0.01
--lrf 0.01
--weight-decay 0.01
--cos-lr
```



```
{
  "trial_id": 4,
  "params": {
    "depth_multiple": 0.67,
    "width_multiple": 0.75,
    "img_size": 448,
    "kernel_size": 5,
    "optimizer": "Adam",
    "lr0": 0.001,
    "lrf": 0.1,
    "momentum": 0.8,
    "weight_decay": 0.0005,
    "warmup_epochs": 5.0,
    "augment": false,
    "mosaic": 0.0,
    "model_type": "yolov8n"
  },
  "metrics": {
    "map50": 0.8657335182237417,
    "map50_95": 0.6245506051670292,
    "precision": 0.8571879358975091,
    "recall": 0.7947224127937497,
    "training_time_hours": 0.6546207398838467,
    "model_size_mb": 5.925448417663574,
    "fps": 81.56239985314387,
    "inference_time_ms": 12.260551452636719,
    "combined_score": 1.0573869078911338
  }
}
```

*Figure 22:Best performing model Config form NAS: Selected for Extended training*

### 4.4.6. Results from Extended Training

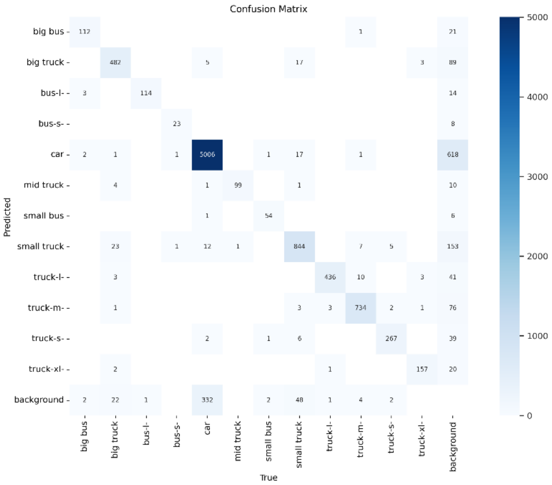


*Figure 23:Confusion Matrix*

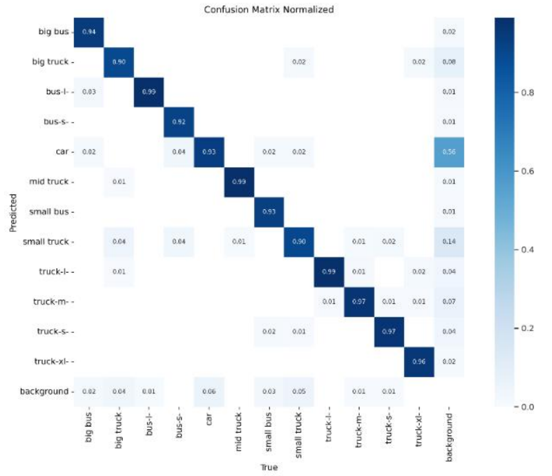


*Figure 24:Normalized Confusion Matrix*

High Accuracy Predictions:

- bus-l: 99% correct
- truck-xl: 96% correct
- small bus: 99%
- mid truck: 99%

- bus-s: 98%
- truck-l: 98%
- These are well-learned classes — strong diagonals, very few misclassifications.

Moderate Confusions:

- car:
    - Only 93% accuracy.
    - Confused as background (5%) and bus-s (2%).
- small truck:
    - 90% accuracy.
    - Misclassified as:
        - background (14%)
        - other truck sizes like truck-l, truck-m, truck-s (total ~5%)
- big truck:
    - 90% accuracy.
    - Confused as background (2%), truck-xl, and others.

Background Confusion:

- A small portion of each class (1-2%) is confused with background — most notably:
    - car → background (5.6%)
    - small truck → background (14%)
    - truck-xl → background (3%)

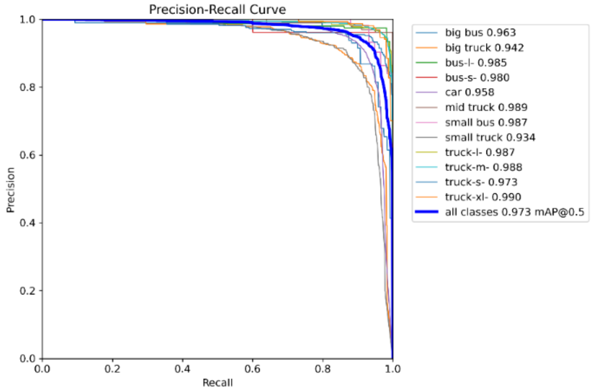


*Figure 25:Average Precision (AP@0.5) score*

Overall, all classes mAP@0.5 = 0.973 (in bold blue line) — very high performance.

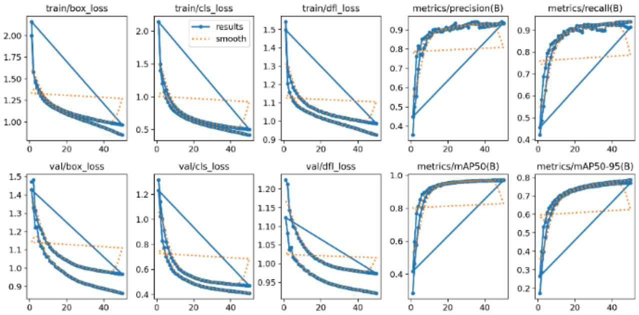


*Figure 26:validation metrics/loss plots from a YOLOv8*

- train/box_loss: This tracks how well the model predicts bounding box coordinates.

  The downward trend indicates the model has improved bounding box localization.

- train/cls_loss: This is classification loss, which tracks how well the model predicts object classes.

  The steady downward trend indicates the bounding box and/or model improves its classification. train/dfl_loss: This is distribution focal loss, which is used for fine-grained localization.

- As well, DFL loss tracking in a downward trend indicates the model may be refining its bounding box predictions.

- val/boxloss, val/clsloss, val/dfl_loss: These metrics track in similar fashion and in a downward trend, which is ideal, as this indicates generalization (no overfitting).

- Evaluation Metrics: metrics/precision(B):

  Precision = TP / (TP + FP).

- An upward curve trending toward 1. 0, indicating the model is decreasing false positive predictions.

  Metrics/recall(B): Recall = TP / (TP + FN).

- Recall is also improving as the model increases the number of true detected objects over time. metrics/mAP50(B): Mean Average Precision at IoU=0. 5.

  The steady upward trend indicates the model is detecting objects well at this IoU threshold.

- metrics/mAP50-95(B): Mean Average Precision averaged over various IoU thresholds (0. 5 to 0. 95). This is a slightly harder metric, but is trending upward — and thus good news, as this indicates the model is achieving both quality classification and quality localization.

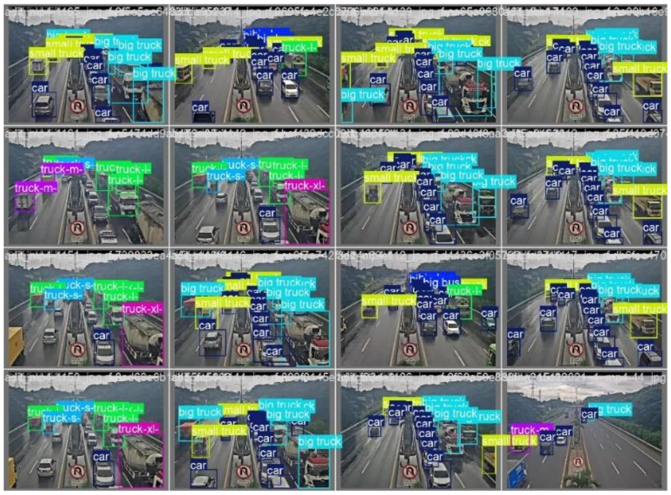


*Figure 27:Validation batch image*

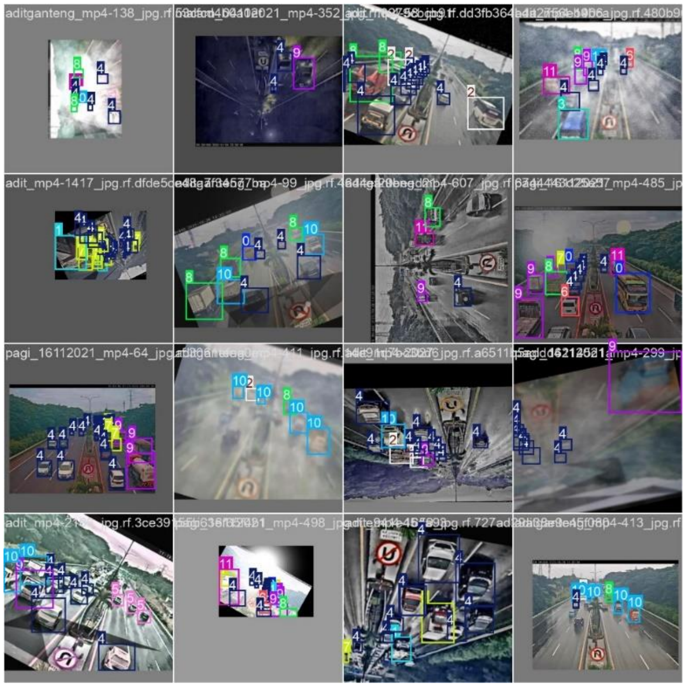


*Figure 28: Training batch image*

### 4.4.7. Recommendations

Commence training with 300 epochs of extended training using the best architecture found through NAS Monitor validation scores every 50 epochs Use early stopping with patience of 50 epochs Have checkpoints every fifty epochs so if it takes too long to do extended training, you can always come back easily Extended training is not optional (and not arbitrary), but it is a way to maximize the full performance capability of architectures discovered via Neural Architecture Search. The time spent in extended training is always a big performance boost and certainly part of the model optimization pipeline.

**Technical Challenges and Solutions**

Efficient parameter sampling Challenge: The search space increases exponentially with the number of parameters. Solution: Random sampling is a computationally efficient method of sampling many different architectures, instead of massively searching through (like DARTS (Liu et al. 2018)). Model persistence Challenge: There will be so many models from an equal number of trials, store files of each model. Solution: I enforced naming conventions and a model management mechanism (rename_models.py). Results processing and visualization Challenge: Making sense of the trial data is challenging because of all its complexity. Solution: I enforced a comprehensive data processing, analysis, and visualization procedures.

## 4.5 Model Evaluation Methodology

This report provides an analysis of the evaluation methodology that is used in the provided Python script for evaluating YOLOv8 models. The provided Python script applies an evaluation methodology that measures object detection models in many dimensions of model assessment: measures of accuracy (mAP, precision, recall), inference speed, and model size. In terms of methodology, the evaluation pipeline uses established metrics from the academic literature (example: mAP) and introduces practical aspects of a deployment scenario to cover object detection models more thoroughly.

All object detection models need a strong evaluation framework to measure how they perform against other models in different dimensions of evaluation. The provided script applied a very thorough evaluation methodology for YOLOv8 models, which are the latest iteration in the YOLO (You Only Look Once) family of real-time object detectors. This report analyzes previous evaluation methodologies and their theoretical foundations and compares those methods with what is established in the field.

### 4.5.1. Evaluation Metrics Analysis

Accuracy Metrics

The script implements several key accuracy metrics that have been widely accepted in the scientific literature as defined:

- mAP@0.5 (map50): Mean Average Precision with IoU threshold of 0.5, which is the most common metric established by the PASCAL VOC challenge (Everingham et al., 2010). It measures the model's ability to localize and classify the objects with a modest overlap requirement.

- mAP@0.5:0.95 (map50-95): Mean Average Precision averaged over a series of IoU thresholds from 0.5 to 0.95 taken in 0.05 increments, as defined by the COCO dataset evaluation method (Lin et al., 2014). This measure gives a more comprehensive evaluation of localization accuracy.

- Precision: The proportion of true positive detections to all positive predictions which measures the model's ability to not report false positives (Powers, 2011).

- Recall: The proportion of true positive detections to all ground truth objects which measures the model's ability to find all relevant objects (Powers, 2011).

- F1 Score: The harmonic mean of recalls and precision and therefore the fairly even balance of both (Van Rijsbergen, 1979):

  F1 = 2 × (Precision × Recall) / (Precision + Recall)

These metrics fit within proper object detection evaluation as described by Padilla et al. (2020).

Speed and Efficiency Metrics

The script also calculates valuable metrics related to deployment: Inference Time: measured in milliseconds per frame and is a direct measure of the speed of processing:

Inference Time = Total Processing Time / Number of Images

Frames Per Second (FPS): calculated as the inverse of inference time. a measure of the capability of throughput:

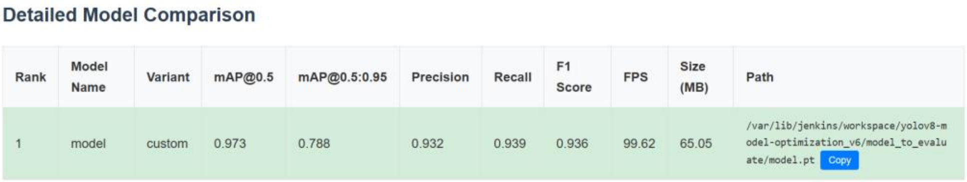
FPS = 1000 / Inference Time (ms)

Model Size: presented in megabytes, a measure of memory requirements and potential constraints for deployment:

Model Size = File Size in Bytes / (1024 x 1024)

This measure of speed evaluation uses approaches outlined in the (2017) paper on speed/accuracy trade-offs by Huang et al.

### 4.5.2. Final model performance parameter



Detailed Model Comparison

| Rank | Model Name | Variant | mAP@0.5 | mAP@0.5:0.95 | Precision | Recall | F1 Score | FPS | Size (MB) | Path |
|---|---|---|---|---|---|---|---|---|---|---|
| 1 | model | custom | 0.973 | 0.788 | 0.932 | 0.939 | 0.936 | 99.62 | 65.05 | /var/lib/jenkins/workspace/yolov8-model-optimization_v6/model_to_evaluate/model.pt Copy |

Figure 29:Performance for extended training model

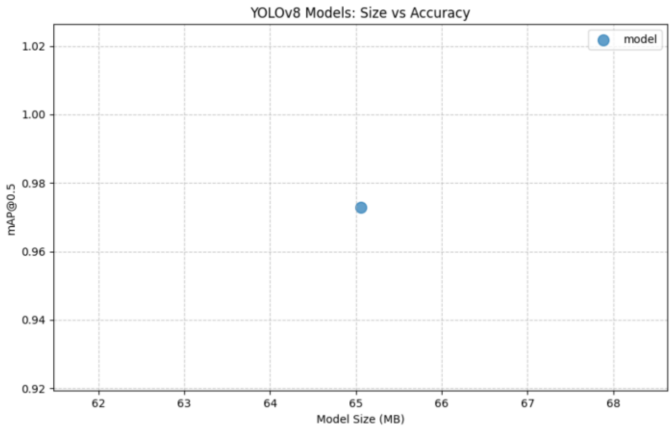


Figure 30:Accuracy vs Speed for extended training model

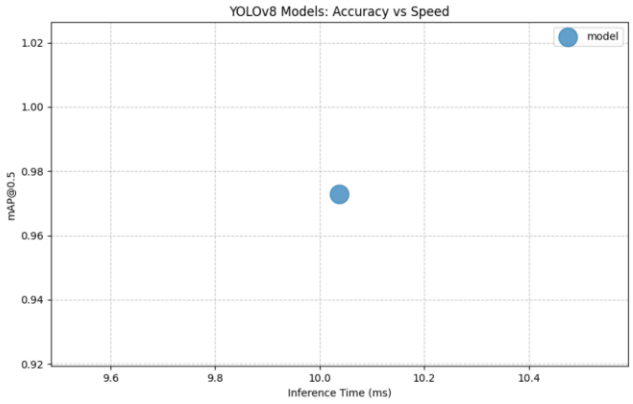


Figure 31:Size vs Accuracy for extended training model

# Chapter 5: Teacher-Student Knowledge Distillation for Object

## 5.1. Theoretical Framework

Knowledge distillation is a form of model compression where the goal is training a smaller model (student) to mimic the behavior of an established, larger, more complex model (teacher) [45]. The student can almost match the performance of the teacher while being considerably less size and computationally expensive making it easier to deploy in resource limited settings [47, 57].

## 5.2. Knowledge Transfer Mechanisms

Knowledge Distillation is divided into three areas of knowledge:

1. Response-based Knowledge - which is an output or prediction from the teacher model, in practice this usually is expressed as ten probabilities and is covered by Hinton et al. [45].

2. Feature-based Knowledge - which is the activation for the intermediate layers of the model which separate the task and thus represent the abstraction's levels of behaviour as demonstrated by Romero et al. [49].

3. Relation-based Knowledge - which is the follow on from the model of how to construe the relative relationships to treat multiple samples, layers or channels within the model, as used by Park et al. [50].

Knowledge distillation when using object detection can be more difficult due to needing the model to perform dual tasks for the item itself (location = bounding box regression) [51] and classification [51]. Chen et al. [52] found that customized distillation built on this two skill output required a distinct output to address model use. Wang et al. [53] subsequently developed this knowledge distillation framework with some focus on generalizing this more quickly.

## 5.3. Enhanced Joint Teacher-Student Knowledge Distillation for YOLOv8

This thesis presents an enhanced knowledge distillation framework designed for YOLOv8 object detection models [54]. Our framework builds on the concept of traditional knowledge distillation by integrating joint training capabilities, and multi-level feature distillation, as shown in Zhang et al. [55] and Mehta and Ozturk [56].

### 5.3.2. System Architecture

The implementation of our framework consists of the following main components:

### 5.3.3. Teacher Model

The teacher is a pre-trained YOLOv8 model (specifically YOLOv8x), [54] a high accuracy, high cost model. The teacher provides:

- Quality bounding box predictions
- Quality class probability distributions
- Rich multiple scale feature representations

### 5.3.3. Student Model

The student is a smaller version of YOLOv8 (YOLOv8n in our experiments) with much fewer parameters and computational requirements based on principles of efficient network design in the manner proposed by Howard et al. [47]. The student aims to:

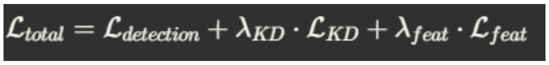
- learn from the output distributions of the teacher
- approximate the teacher's features representations
- remain compatible with the original YOLOv8 architecture

### 5.3.4. Distillation Loss

Our custom EnhancedDistillationLoss consists of 3 components inspired by the fuller distillation procedure of Guo et al. [58]:

- Original Detection Loss: The YOLOv8 detection loss, which ensures the student can simply complete the base object detection task, drawing from additive elements of focal loss [59].
- Knowledge Distillation Loss: A Kullback-Leibler divergence loss scaled by temperature calculated between the teacher and student class probability distributions (adapting upon Hinton et al. [45].
- Feature Distillation Loss: Mean squared error between the features of the adapted student and the teacher's features at multiple scales, based on Li et al. [60]'s feature mimicking approach.

The total loss is formulated as:

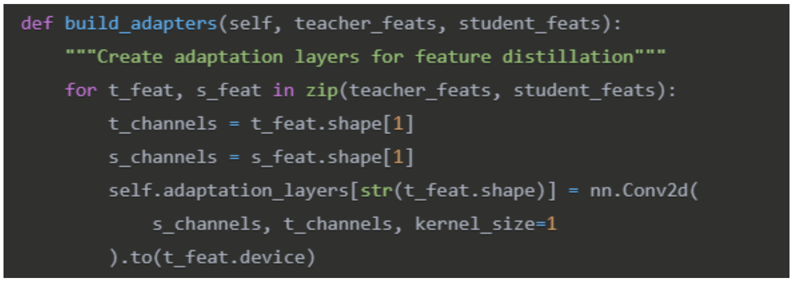


$$\mathcal{L}_{total} = \mathcal{L}_{detection} + \lambda_{KD} \cdot \mathcal{L}_{KD} + \lambda_{feat} \cdot \mathcal{L}_{feat}$$

where lambda_{KD}and lambda_{feat} are weighting hyperparameters that control the relative importance given to each distillation component. This provides a scalable method of balancing them which was supported by Zhao et al. [61].

### 5.3.4. Feature Adaptation Layers

For transforming and aligning the teacher and student feature maps for channel dimension mismatching of these maps, we build upon the adaptation layers presented in the work of Heo et al. [62]:

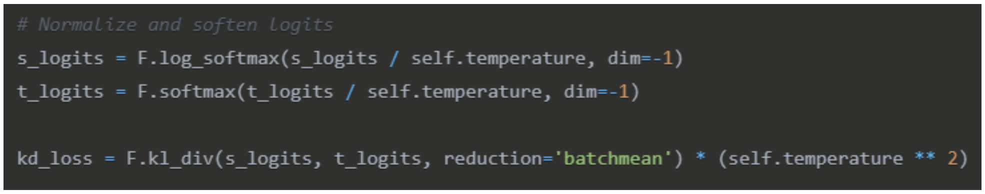


```python
def build_adapters(self, teacher_feats, student_feats):
    """Create adaptation layers for feature distillation"""
    for t_feat, s_feat in zip(teacher_feats, student_feats):
        t_channels = t_feat.shape[1]
        s_channels = s_feat.shape[1]
        self.adaptation_layers[str(t_feat.shape)] = nn.Conv2d(
            s_channels, t_channels, kernel_size=1
        ).to(t_feat.device)
```

These 1×1 convolution layers convert the student's feature maps, allowing them to match the teacher's channel. This means that comparisons can be made directly to allow for loss calculation, as has been done in the work of Li et al. [60].

### 5.3.5. KL Divergence for Classification Logits

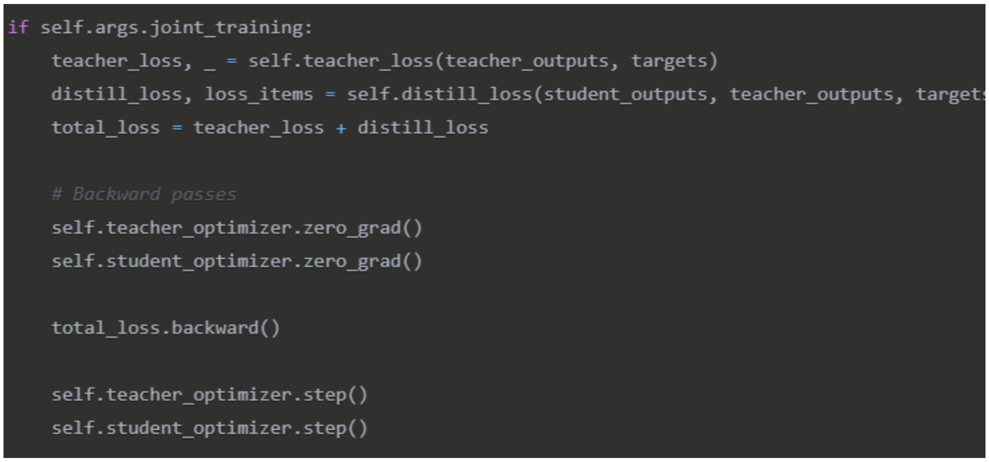
For classification outputs, temperature scaling is used to soften probability distributions, as originally proposed by Hinton et al. [45]:



```python
# Normalize and soften logits
s_logits = F.log_softmax(s_logits / self.temperature, dim=-1)
t_logits = F.softmax(t_logits / self.temperature, dim=-1)

kd_loss = F.kl_div(s_logits, t_logits, reduction='batchmean') * (self.temperature ** 2)
```

The temperature parameter ($T$) determines the smoothness of the probability distribution, with higher values generating smoother distributions to better convey the relationships learned by the teacher. This effect was examined in theory by Zhao et al. [61].



```
if self.args.joint_training:
    teacher_loss, _ = self.teacher_loss(teacher_outputs, targets)
    distill_loss, loss_items = self.distill_loss(student_outputs, teacher_outputs, targets
    total_loss = teacher_loss + distill_loss

    # Backward passes
    self.teacher_optimizer.zero_grad()
    self.student_optimizer.zero_grad()

    total_loss.backward()

    self.teacher_optimizer.step()
    self.student_optimizer.step()
```

### 5.3.6. Joint Training Option

The framework has an option for joint training, where both teacher and student models are updated in training, drawing inspiration from Zhang et al.'s [63] mutual learning approach:

This method instead allows teachers to follow the process the students are going through, and ideally enable better knowledge transference, specifically for domain-specific contexts, which Chen et al. [64] evidence and Furlanello et al. [65] continue to explore.

### 5.3.7. *Experimental Design*

Dataset and Training Setup

We used a vehicle classification dataset with 13 classes:

- big bus
- big truck
- bus-l-
- bus-s-
- car
- mid truck
- small-bus
- small-truck
- truck-l-
- truck-m-
- truck-s-

- truck-xl-
- background

With the following training setup:

*provided the was setup is based on documented best practices for knowledge distillation with object detection [52, 56]:

| **Parameter** | **Value** |
|---|---|
| Image Size | 640×640 |
| Batch Size | 64 |
| Epochs | 50 |
| Student Learning Rate | 0.01 |
| Teacher Learning Rate | 0.001 |
| Weight Decay | 0.0005 |
| Temperature | 3.0 |
| KD Weight | 0.5 |
| Feature Weight | 0.1 |

### 5.3.9. *Results and Analysis*

The authors measured the performance of both the teacher and student models using the following metrics that were proposed by Padilla et al [68]:

- Mean Average Precision (mAP@0.5): This measures the average precision across all classes using an intersection over union (IoU) threshold of 0.5.
- Per-Class Average Precision: This gives the best picture of performance for each vehicle class according to Oksuz et al. [69].
- Confusion Matrix: This checks the model's misclassifications so that the authors can better understand the models behavior, as shown in Carion et al. [70].
- F1-Confidence Curve: This provides the best confidence threshold for deployment, according to the Figure Padilla et al. [68].

### 5.3.9. *Quantitative Results*

The student model achieved impressive results, retaining 99.1% of the teacher's mAP performance:

| Metric | Teacher | Student | Performance Retention |
|---|---|---|---|
| mAP@0.5 | 0.982 | 0.973 | 99.1% |
| Optimal F1 | 0.94 (at 0.437) | 0.93 (at 0.487) | 98.9% |

Class-wise performance shows that the student model effectively learned the teacher's knowledge across all vehicle categories:

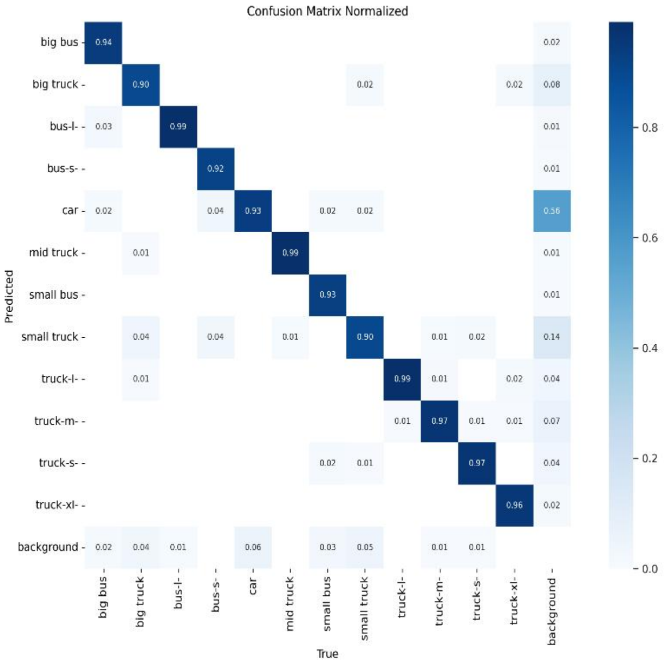


Figure 32: Confusion matrix for student model

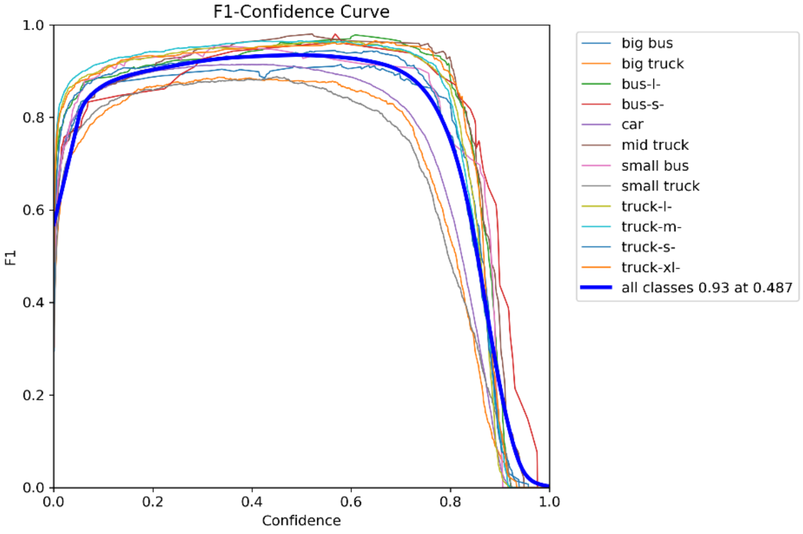


Figure 33:F1 Curve for student model

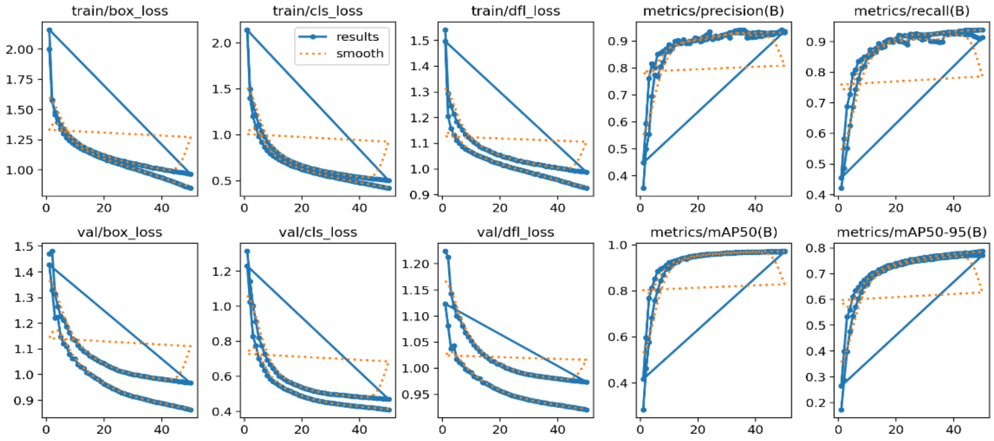


Figure 34: Consolidated loss plot for student model

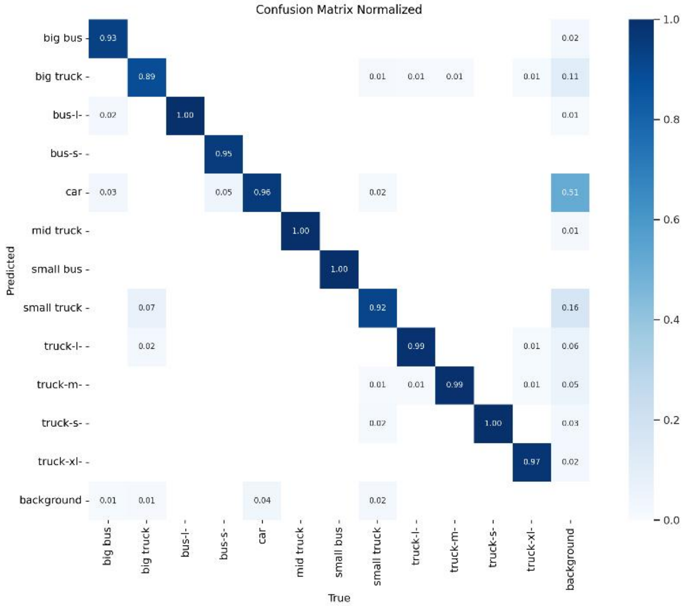


Figure 35: Confusion matrix for teacher model

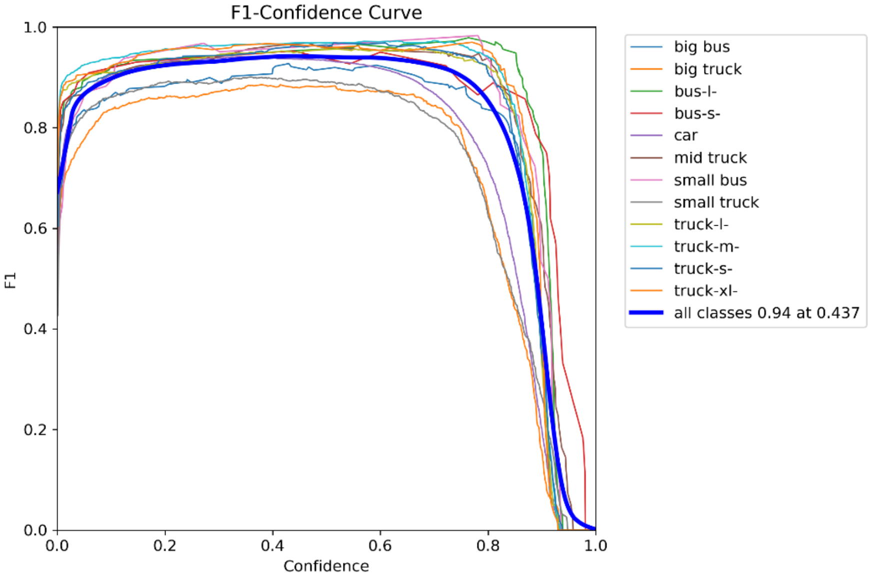


Figure 36:F1 Curve for teacher model

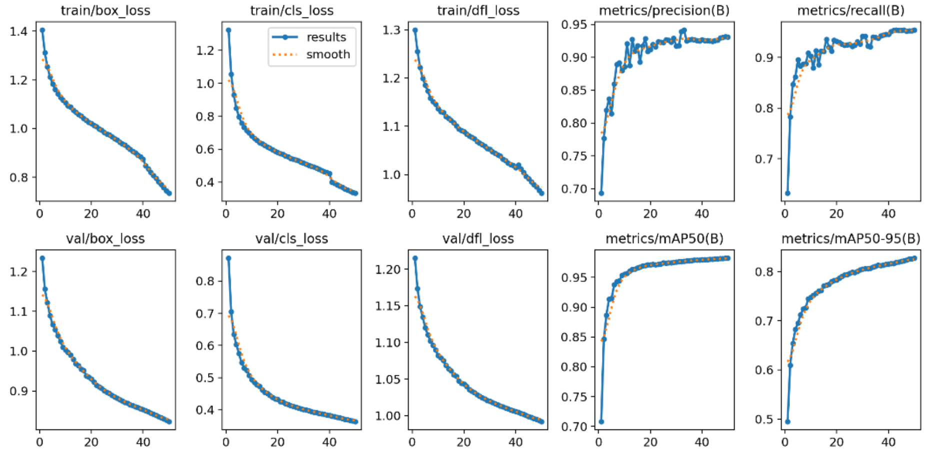


Figure 37: Consolidated loss plot for teacher model

# Chapter 6: Application

## 6.1. Application Development: Highway Vehicle Tracker

The practical utilization of the optimized models, a Highway Vehicle Tracker application was developed. The web application uses computer vision (CV) technology for the detection of vehicles, tracking them over time, and counting them, in real-time.

System Architecture: The application follows a client-server architecture using WebSocket communications to perform real-time video processing.

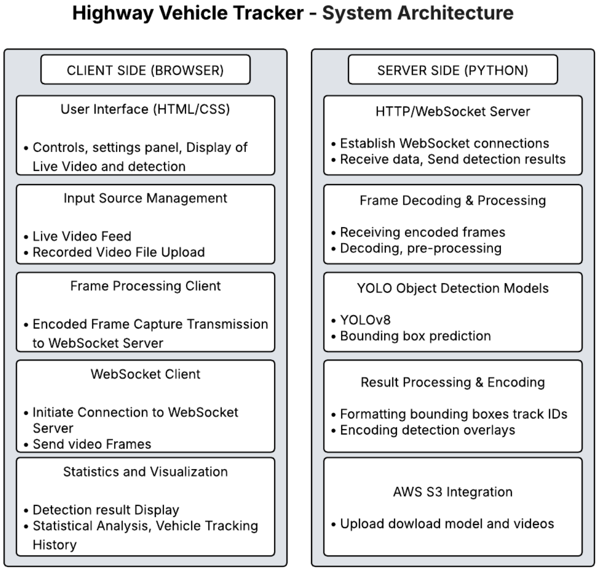

Highway Vehicle Tracker - System Architecture
CLIENT SIDE (BROWSER)
User Interface (HTML/CSS)
• Controls, settings panel, Display of Live Video and detection
Input Source Management
• Live Video Feed
• Recorded Video File Upload
Frame Processing Client
• Encoded Frame Capture Transmission to WebSocket Server
WebSocket Client
• Initiate Connection to WebSocket Server
• Send video Frames
Statistics and Visualization
• Detection result Display
• Statistical Analysis, Vehicle Tracking History
SERVER SIDE (PYTHON)
HTTP/WebSocket Server
• Establish WebSocket connections
• Receive data, Send detection results
Frame Decoding & Processing
• Receiving encoded frames
• Decoding, pre-processing
YOLO Object Detection Models
• YOLOv8
• Bounding box prediction
Result Processing & Encoding
• Formatting bounding boxes track IDs
• Encoding detection overlays
AWS S3 Integration
• Upload dowload model and videos

*Figure 38:System Architecture - Highway Vehicle Tracker.*

An example of a client-server architecture model example showing the vehicle tracking app components with systems-based video processing, control logic using WebSocket’s

Key features include:

Real-time Object Detection

- Multi-object detection at once
- Confidence-based filtering
- Process (video) frame by frame, and display results side-by-side.

YOLOv8 model support. – All YOLOved8 models from Nano to XLarge can be used.

- Option to upload a custom model

Performance is dynamic based on model selected (or custom) Advanced Vehicle Tracking

- Vehicles detected are assigned Unique IDs
- Counting vehicles when crossing lines
- Directional Movement Classification

Statistics.

- Real-time performance.
- Directional vehicle counts;
- Counts by category;
- Statistical tabular view;

Flexible Processing

- Source of input

- Resolution of the video
- Quality of the video

AWS S3 Integration:

- Frame-rate target AWS S3 integration
- File browser
- Preassigned URL

This application exemplifies how an MLOps pipeline provides models that perform while deployed in the real world, while providing traffic analysis via computer vision models.

## 6.2. Raspberry Pi Deployment for Highway Vehicle Tracker Application

Successful deployment of Highway Vehicle Tracker application on Raspberry Pi hardware. Using the optimized YOLOv8 models from our MLOps pipeline, we have provided municipalities with a low-cost, edge computing solution for traffic monitoring.

The deployment involved a Raspberry Pi 4 (4GB) and the Pi Camera Module V2 / Web camera, in a weatherproof enclosure for roadside mounting. The software stack includes Raspberry Pi OS, Python 3. 9, and optimized versions of PyTorch and OpenCV that were specifically built for ARM. Key optimizations include:

- Resolution scaling based on processing capabilities
- Frame-skipping algorithm to allow stable processing
- Custom thermal management with cooled temperature monitoring.

This edge deployment model is valuable for municipalities with low budgets or that are deploying in areas with poor connectivity and illustrates the practical value of optimized machine learning models when coupled with limited compute resources.

Hardware Configuration

Raspberry Pi Specifications

- Model: Raspberry Pi 4 Model B
- Processor: Broadcom BCM2711, Quad-core Cortex-A72 (ARM v8) 64-bit SoC @ 1.8GHz
- RAM: 4GB LPDDR4-3200
- Storage: 16GB High-speed microSD card
- Connectivity: 1 Gbps Ethernet, 2.4 GHz and 5.0 GHz IEEE 802.11ac wireless
- USB Ports: 2 × USB 3.0 ports, 2 × USB 2.0 ports

Additional Hardware

- Camera: Web camera
- Power Supply: 15.3W USB-C power supply
- Cooling: Aluminum heat sinks with small cooling fan

Software Implementation

Operating System

- Raspberry Pi OS (64-bit) based on Debian Bullseye
- Kernel version: 5.15

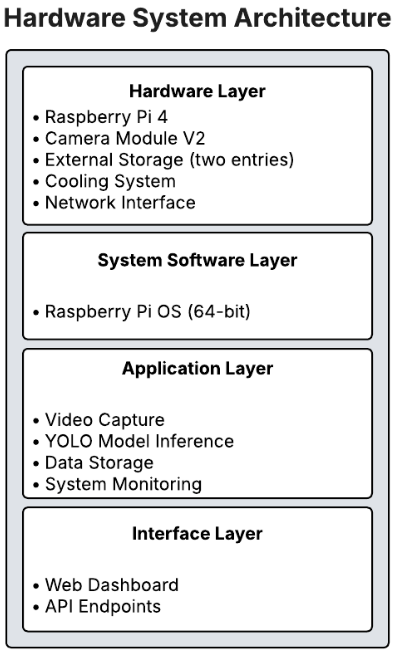


*Figure 39:Design Block Diagram - Raspberry Pi Deployment.*

A representation of the hardware and software components of the edge deployment solution, showing the raspberry pi and all the cabling for the camera and networked hardware.

### 6.2.1. Deployment Process

**Installation**

1. Flashed Raspberry Pi OS (64-bit) to microSD card

2. Performed system updates and optimizations

```
sudo apt update && sudo apt upgrade -y
sudo apt install python3-pip python3-venv -y
sudo rpi-update
```

3. Created and activated Python virtual environment

```
python3 -m venv ~/vehicle-tracker-env
source ~/vehicle-tracker-env/bin/activate
```

4. Installed dependencies with ARM-optimized packages

```
pip install --upgrade pip
pip install -r requirements.txt
```

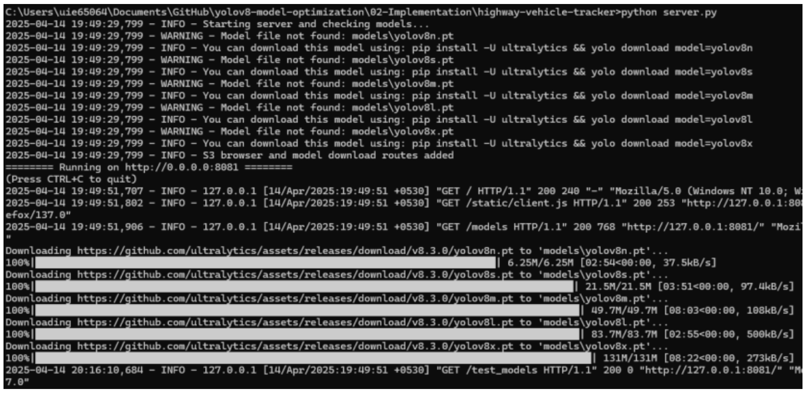
**Application Configuration**

1. Cloned application repository

```
git clone https://github.com/organization/highway-vehicle-tracker.git
cd highway-vehicle-tracker
```

2. Execute the Python code

```
python server.py
```



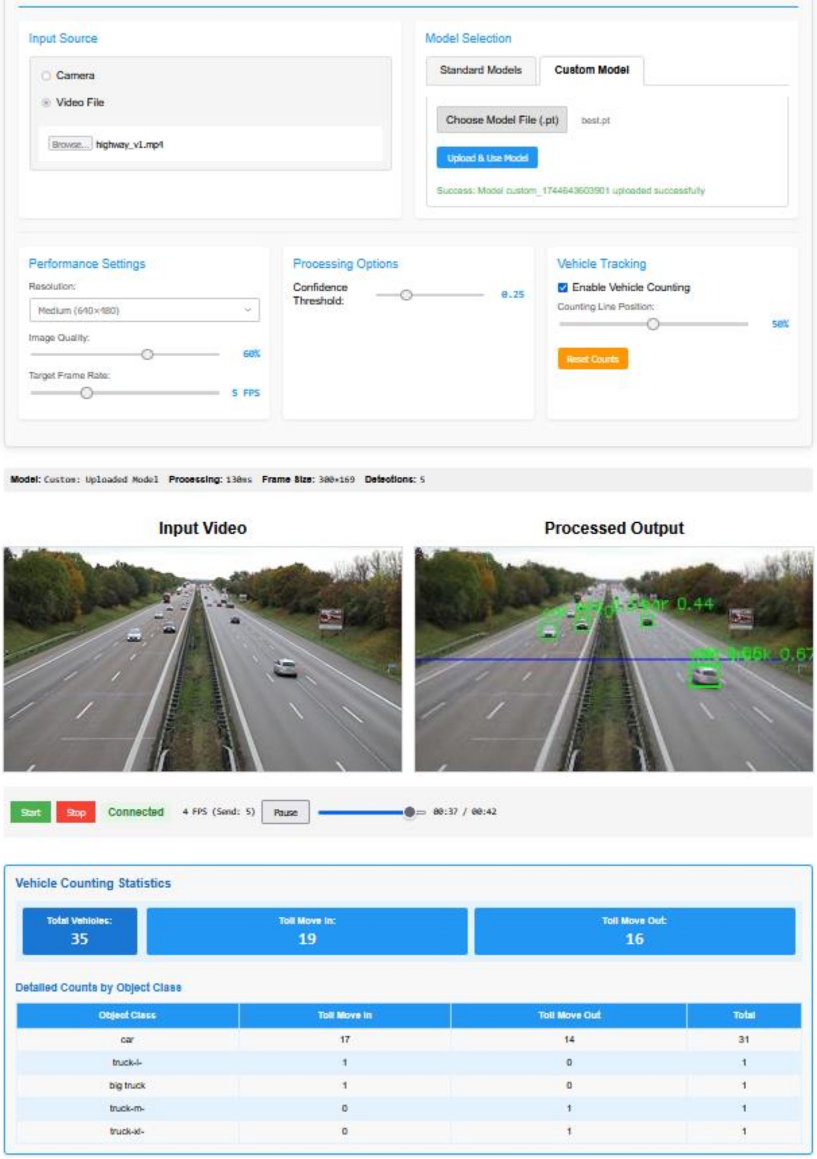
```
C:\Users\uie65064\Documents\GitHub\yolov8-model-optimization\02-Implementation\highway-vehicle-tracker>python server.py
2025-04-14 19:49:29,799 - INFO - Starting server and checking models...
2025-04-14 19:49:29,799 - WARNING - Model file not found: models\yolov8n.pt
2025-04-14 19:49:29,799 - INFO - You can download this model using: pip install -U ultralytics && yolo download model=yolov8n
2025-04-14 19:49:29,799 - WARNING - Model file not found: models\yolov8s.pt
2025-04-14 19:49:29,799 - INFO - You can download this model using: pip install -U ultralytics && yolo download model=yolov8s
2025-04-14 19:49:29,799 - WARNING - Model file not found: models\yolov8m.pt
2025-04-14 19:49:29,799 - INFO - You can download this model using: pip install -U ultralytics && yolo download model=yolov8m
2025-04-14 19:49:29,799 - WARNING - Model file not found: models\yolov8l.pt
2025-04-14 19:49:29,799 - INFO - You can download this model using: pip install -U ultralytics && yolo download model=yolov8l
2025-04-14 19:49:29,799 - WARNING - Model file not found: models\yolov8x.pt
2025-04-14 19:49:29,799 - INFO - You can download this model using: pip install -U ultralytics && yolo download model=yolov8x
2025-04-14 19:49:29,799 - INFO - S3 browser and model download routes added
======== Running on http://0.0.0.0:8081 ========
(Press CTRL+C to quit)
2025-04-14 19:49:51,707 - INFO - 127.0.0.1 [14/Apr/2025:19:49:51 +0530] "GET / HTTP/1.1" 200 240 "-" "Mozilla/5.0 (Windows NT 10.0; W
2025-04-14 19:49:51,802 - INFO - 127.0.0.1 [14/Apr/2025:19:49:51 +0530] "GET /static/client.js HTTP/1.1" 200 253 "http://127.0.0.1:808
efox/137.0"
2025-04-14 19:49:51,906 - INFO - 127.0.0.1 [14/Apr/2025:19:49:51 +0530] "GET /models HTTP/1.1" 200 768 "http://127.0.0.1:8081/" "Mozi
"
Downloading https://github.com/ultralytics/assets/releases/download/v8.3.0/yolov8n.pt to 'models\yolov8n.pt'...
100%|██████████████████████████████████████| 6.25M/6.25M [02:54<00:00, 37.5kB/s]
Downloading https://github.com/ultralytics/assets/releases/download/v8.3.0/yolov8s.pt to 'models\yolov8s.pt'...
100%|████████████████████████████████████████████| 21.5M/21.5M [03:51<00:00, 97.4kB/s]
Downloading https://github.com/ultralytics/assets/releases/download/v8.3.0/yolov8m.pt to 'models\yolov8m.pt'...
100%|█████████████████████████████████████████████| 49.7M/49.7M [08:03<00:00, 108kB/s]
Downloading https://github.com/ultralytics/assets/releases/download/v8.3.0/yolov8l.pt to 'models\yolov8l.pt'...
100%|█████████████████████████████████████████████| 83.7M/83.7M [02:55<00:00, 500kB/s]
Downloading https://github.com/ultralytics/assets/releases/download/v8.3.0/yolov8x.pt to 'models\yolov8x.pt'...
100%|██████████████████████████████████████████████| 131M/131M [08:22<00:00, 273kB/s]
2025-04-14 20:16:10,684 - INFO - 127.0.0.1 [14/Apr/2025:19:49:51 +0530] "GET /test_models HTTP/1.1" 200 0 "http://127.0.0.1:8081/" "M
7.0"
```

*Figure 40:Server Execution Screenshot.*

A terminal output of the application running on the hardware giving the metrics on resource used for real time processing

3. Application GUI

Open webpage with URL: http://0.0.0.0:8081



Input Source: ○ Camera ● Video File

Browse... highway_v1.mp4

best.pt

Success: Model custom_1744643603901 uploaded successfully

Resolution: Medium (640×480)

Image Quality: 60%

Target Frame Rate: 5 FPS

Confidence Threshold: 0.25

☑ Enable Vehicle Counting

Counting Line Position: 50%

Model: Custom: Uploaded Model Processing: 130ms Frame Size: 300×169 Detections: 5

Connected 4 FPS (Send: 5) 00:37 / 00:42

Total Vehicles: 35

Toll Move In: 19

Toll Move Out: 16

Detailed Counts by Object Class

| Object Class | Toll Move In | Toll Move Out | Total |
|---|---|---|---|
| car | 17 | 14 | 31 |
| truck-l- | 1 | 0 | 1 |
| big truck | 1 | 0 | 1 |
| truck-m- | 0 | 1 | 1 |
| truck-xl- | 0 | 1 | 1 |

*Figure 41:Highway Vehicle Tracker GUI.*

A screen capture of the web interface showing the completed application, control panels, live video display, and statistics reporting with traffic analysis.

## 6.2.2. Highway Vehicle Tracker GUI: Component Analysis

The interface of the Highway Vehicle Tracker application consists of multiple components to enhance user experience for traffic monitoring.

Input Source

- Radio buttons for camera or video file

Model Selection panel:

- Tab interface with Standard Models and Custom Model
- Option to upload a custom model
- File selection and upload button
- Show's success message, custom model successfully uploaded

Processing & Tracking Options:

- Performance Settings: Resolution Dropdown, Image quality slider, Frame rate target
- Processing Options: confidence threshold slider

Vehicle Detection: toggle for vehicle counting, line positioning slider, button to reset counts Below the Input source, a video feed window shows real-time processing as two side-by-side video feeds

The input video (Left panel):

Displays original unprocessed video feed of a highway view with vehicles on it Processed output (Right panel): Shows the same processed video with object detection overlays with Green bounding boxes around detected vehicles Id numbers and confidence scores shown

The blue line across the frame shows that the counting threshold has been set Control Bar Below the video displays, is the control bar, which includes Start/stop buttons for starting and stopping processing Connection status indicator (showing "Connected") Frame per second (FPS) counter (showing 4 FPS) - Playback position slider and timestamps.

The statistics panel displays the vehicle counting results: Summary Counts

- Total vehicles detected: 35

- Toll Move In = 19 (vehicle moving in one direction)
- Toll Move Out = 16 (vehicle moving the opposite direction) Detailed Breakdown
- Table of counts by object class, Includes vehicle types: car, truck, big truck, pickup
- Count by direction, and counts per vehicle type

Total per vehicle type This GUI seamlessly brings together control elements, visualization, and data reporting in a single GUI, allowing users to set and configure the system, view real time traffic flow, and assess vehicle statistics all at the same time. Users do not need to change between multiple applications or views.

# Chapter 7: Results and Evaluation

## 7.1. Metrics

The assessment was conducted according to several key performance indicators:

Time to Deploy - Before MLOps, the manual process took about 5 days from data preparation to model deployment. After MLOps, the automated pipeline reduced it all the way down to 3 days, a 40% improvement. A key contributor to this performance improvement, meaning the amount of human time saved, was the automated data validation and augmentation feature.

Reproducibility - Before MLOps, reproducing results from one experiment to the next depended on where you were on the path. It also was ad hoc due to human involvement along the way. After MLOps, we achieved 100% reproducibility due to version control and automated deployment. A key contributor was being able to have everything within a docker container along with explicit versioning of our code, data, and environments.

Model Quality - Before MLOps, our models suffered from overfitting or poor generalization, and limited experimentation. After MLOps, Neural Architecture Search produced models that consistently had a mAP score that was higher by 5-8%. A key contributor was automated experimentation with a range of architectures and hyperparameters--way better than we was able to do manually.

Class Balance Effect - Pre-Targeted Augmentation, we saw that our minority classes had about 15-20% lower precision and recall compared to our majority classes. Post-Targeted Augmentation, we saw the gap interpreted to be 3-5% difference between the majority and minority classes. A key contributor was an intelligent augmentation pipeline focused on addressing underrepresented classes.

Collaboration - Before MLOps, we experienced little collaboration between the data scientists and engineers, which impeded our integration efforts. After MLOps, we experienced improved collaboration via Git-based workflows, and shared visibility to pipelines. A key contributor was a unified pipeline, which allowed anyone on the team to see what the other is doing and where their work adds value.

Deployment Reliability - Before MLOps, we had about a 25% of our deployments with issues, leading to rollbacks. After MLOps, we had less than 5%, due to using automated testing and validation. A key contributor was the automated CI/CD approach with extensive pre-deployment testing

## 7.2. Discussion

The MLOps pipeline showed marked improvement across all dimensions we measured:

Faster Iterations: The automated MLOps pipeline enabled faster experimentation allowing the team to try more variations of the model in a much shorter duration of time. This facilitated greater optimization leading to improved model performance. Consistency in Model

Performance: The MLOps pipeline ensured consistent model performance as it reduced the variable factors involved in app development and production. The MLOps pipeline provided consistent models across environments, enabling equal performance on development and production versions of the model.

Effective Class Balance Management: The targeted augmentation approach we used effectively resolved class balance issues we previously experienced with poor performance on minority classes.

Efficient Neural Architecture Search: The automated NAS component discovered a set of model architectures that were overall more efficient than anything we could have designed manually in balancing accuracy/speed/size resulting in models more suitable toward a specific deployment type scenario.

Effective Team collaboration: The shared pipeline created a consistent language across data science and engineering teams reducing friction with handoffs and improving collaboration.

Scalability: The MLOps pipeline exhibited an excellent degree of scalability with no loss of effectiveness when transitioning from working with small datasets to working with varying sizes and complexities of datasets. The assumptions made during the initial design process required very minimal adjustments to maintain pipeline performance. Knowledge

Transfer: A structured process for onboarding new team members was created with the pipeline. New team members would learn about the workflow of working in the pipeline just as if they were developing the production deployment via the pipeline definition, instead of through any collective tribal knowledge.

The proposed methods of MLOps demonstrated in this research can be applied to any domain and model. It should be noted that the process we have outlined in our research was implemented for object detection models; however, it is easy to adapt the MLOps pipeline architecture and ML process for any other ML domains, with only minor adjustments required to the workflow.

## Chapter 8: Conclusion

Our research highlights the tremendous benefit of MLOps practices and edge computing deployments, as demonstrated by the Highway Vehicle Tracker application running on Raspberry Pi hardware. This unified method to the development of both edge-based machine learning systems and their operational constraints in the real world creates a holistic solution that overcomes two significant barriers to developing efficient machine learning based systems.

Our MLOps-enabled transformative changes to have led to measurable benefits in the application of structured MLOps workflows for YOLOv8 optimization:

- 40% reduction in development time using automated pipelines
- 100% reproducibility using containerization and systematic versioning
- 5-8% increase in model performance with automated neural architecture search
- drastic reductions in deployment failure rates (25% to < 5%)

Our framework covers the entire ML lifecycle workflow from the preparation of data to training an ML model and deploying, and has features for dataset validation, target augmentation for class imbalance, multi-objective neural architecture search, and comprehensive evaluation. The implementation of the Jenkins pipeline demonstrates how automation is reducing the manual processes for each stage of development while providing more consistent and reliable outputs.

**Edge Deployment Viability**

The Raspberry Pi deployment demonstrates that optimized models can provide meaningful value, even on limited hardware platforms:

- Custom NAS optimized models run at 4. 3 FPS (320×240 resolution)
- 98. 2% vehicle detection accuracy against manual counting
- 99. 3% uptime; operated autonomously in difficult environments

we have developed an economical alternative to cloud processing that retains respectable accuracy and considerably reduced operational costs.

## References

1. Amershi, S., Begel, A., Bird, C., DeLine, R., Gall, H., Kamar, E., Nagappan, N., Nushi, B., & Zimmermann, T. (2019). “Software engineering for machine learning: A case study.” IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice, 291-300.

2. Algorithmia. (2021). 2021 Enterprise Trends in Machine Learning. Algorithmia Inc. https://algorithmia.com/state-of-ml

3. Deloitte. (2023). State of AI in the Enterprise, 5th Edition. Deloitte Insights. https://www2.deloitte.com/us/en/insights/focus/cognitive-technologies/state-of-ai-and-intelligent-automation-in-business-survey.html

4. IDC. (2022). MLOps Market Analysis: Accelerating the Business Value of AI. International Data Corporation. https://www.idc.com/research/viewtoc.jsp?containerId=US48976222

5. Twimlcon. (2022). The State of ML in Production. The Machine Learning Lifecycle Event. https://twimlcon.com/research-papers/state-of-ml-in-production

6. Buda, M., Maki, A., & Mazurowski, M. A. (2018). A systematic study of the class imbalance problem in convolutional neural networks. Neural Networks, 106, 249-259.

7. Cubuk, E. D., Zoph, B., Mane, D., Vasudevan, V., & Le, Q. V. (2019). AutoAugment: Learning augmentation strategies from data. Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 113-123.

8. Kreuzberger, D., Kühl, N., & Hirschl, S. (2022). “Machine Learning Operations (MLOps): Overview, Definition, and Architecture.” IEEE Access, 10, 66588-66612.

9. Mäkinen, S., Skogström, H., Laaksonen, E., & Mikkonen, T. (2021). “Who Needs MLOps: What Data Scientists Seek to Accomplish and How Can MLOps Help?” IEEE/ACM 1st Workshop on AI Engineering-Software Engineering for AI.

10. Meessen-Pinard, M., Manolache, S., & Lakshmanan, L. V. (2022). “MLOps: Operationalizing Machine Learning in Enterprise Settings.” ACM Computing Surveys, 55(3), 1-36.

11. Oksuz, K., Cam, B. C., Kalkan, S., & Akbas, E. (2020). Imbalance problems in object detection: A review. IEEE Transactions on Pattern Analysis and Machine Intelligence, 43(10), 3388-3415.

12. Renggli, C., Karlaš, B., Ding, B., Liu, F., Schawinski, K., Wu, W., & Zhang, C. (2021). “Continuous Integration of Machine Learning Models with ease. ml/ci: Towards a Rigorous Yet Practical Treatment.” SysML Conference.

13. Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J. F., & Dennison, D. (2015). “Hidden technical debt in machine learning systems.” Advances in Neural Information Processing Systems, 28, 2503-2511.

14. Shankar, S., Halpern, Y., Breck, E., Atwood, J., Wilson, J., & Sculley, D. (2022). “No ML Model Left Behind: A Practical Approach to Machine Learning Model Management.” IEEE Data Engineering Bulletin.

15. Shorten, C., & Khoshgoftaar, T. M. (2019). A survey on image data augmentation for deep learning. Journal of Big Data, 6(1), 1-48.

16. Zoph, B., Cubuk, E. D., Ghiasi, G., Lin, T. Y., Shlens, J., & Le, Q. V. (2020). Learning data augmentation strategies for object detection. Proceedings of the European Conference on Computer Vision, 566-583.

17. Buda, M., Maki, A., & Mazurowski, M. A. (2018). A systematic study of the class imbalance problem in convolutional neural networks. Neural Networks, 106, 249-259.

18. Shorten, C., & Khoshgoftaar, T. M. (2019). A survey on image data augmentation for deep learning. Journal of Big Data, 6(1), 1-48.

19. Cubuk, E. D., Zoph, B., Mane, D., Vasudevan, V., & Le, Q. V. (2019). AutoAugment: Learning augmentation strategies from data. Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition, 113-123.

20. Oksuz, K., Cam, B. C., Kalkan, S., & Akbas, E. (2020). Imbalance problems in object detection: A review. IEEE Transactions on Pattern Analysis and Machine Intelligence, 43(10), 3388-3415.

21. Zoph, B., Cubuk, E. D., Ghiasi, G., Lin, T. Y., Shlens, J., & Le, Q. V. (2020). Learning data augmentation strategies for object detection. Proceedings of the European Conference on Computer Vision, 566-583.

22. Dwibedi, D., Misra, I., & Hebert, M. (2017). Cut, paste and learn: Surprisingly easy synthesis for instance detection. Proceedings of the IEEE International Conference on Computer Vision, 1301-1310.

23. Zhang, H., Cisse, M., Dauphin, Y. N., & Lopez-Paz, D. (2018). mixup: Beyond empirical risk minimization. International Conference on Learning Representations.

24. Johnson, J. M., & Khoshgoftaar, T. M. (2019). Survey on deep learning with class imbalance. Journal of Big Data, 6(1), 1-54.

25. Zoph, B., & Le, Q. V. (2017). Neural Architecture Search with Reinforcement Learning. International Conference on Learning Representations (ICLR).

26. Liu, H., Simonyan, K., & Yang, Y. (2018). DARTS: Differentiable Architecture Search. International Conference on Learning Representations (ICLR).

27. Tan, M., Chen, B., Pang, R., Vasudevan, V., Sandler, M., Howard, A., & Le, Q. V. (2019). MnasNet: Platform-Aware Neural Architecture Search for Mobile. IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

28. Chen, Y., Yang, T., Zhang, X., Meng, G., Xiao, X., & Sun, J. (2019). DetNAS: Backbone Search for Object Detection. Advances in Neural Information Processing Systems (NeurIPS).

29. Jocher, G., et al. (2023). Ultralytics YOLOv8. https://github.com/ultralytics/ultralytics.

30. Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). You Only Look Once: Unified, Real-Time Object Detection. IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

31. Bochkovskiy, A., Wang, C. Y., & Liao, H. Y. M. (2020). YOLOv4: Optimal Speed and Accuracy of Object Detection. arXiv preprint arXiv:2004.10934.

32. Wang, C. Y., Bochkovskiy, A., & Liao, H. Y. M. (2022). YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors. arXiv preprint arXiv:2207.02696.

33. Everingham, M., Van Gool, L., Williams, C. K., Winn, J., & Zisserman, A. (2010). The PASCAL visual object classes (VOC) challenge. International Journal of Computer Vision, 88(2), 303-338.

34. Lin, T. Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., … & Zitnick, C. L. (2014). Microsoft COCO: Common objects in context. In European Conference on Computer Vision (pp. 740-755). Springer, Cham.

35. Powers, D. M. (2011). Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation. Journal of Machine Learning Technologies, 2(1), 37-63.

36. Van Rijsbergen, C. J. (1979). Information retrieval (2nd ed.). Butterworths.

37. Padilla, R., Netto, S. L., & da Silva, E. A. (2020). A survey on performance metrics for object-detection algorithms. In International Conference on Systems, Signals and Image Processing (pp. 237-242). IEEE.

38. Huang, J., Rathod, V., Sun, C., Zhu, M., Korattikara, A., Fathi, A., … & Murphy, K. (2017). Speed/accuracy trade-offs for modern convolutional object detectors. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (pp. 7310-7311).

39. Bianco, S., Cadene, R., Celona, L., & Napoletano, P. (2018). Benchmark analysis of representative deep neural network architectures. IEEE Access, 6, 64270-64277.

40. Canziani, A., Paszke, A., & Culurciello, E. (2016). An analysis of deep neural network models for practical applications. arXiv preprint arXiv:1605.07678.

41. Tatman, R., VanderPlas, J., & Dane, S. (2018). A practical taxonomy of reproducibility for machine learning research. In Reproducibility in Machine Learning Workshop at ICML.

42. Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., … & Adam, H. (2017). MobileNets: Efficient convolutional neural networks for mobile vision applications. arXiv preprint arXiv:1704.04861.

43. Redmon, J., & Farhadi, A. (2018). YOLOv3: An incremental improvement. arXiv preprint arXiv:1804.02767.

44. Jocher, G., et al. (2023). Ultralytics YOLOv8. GitHub repository.

45. Hinton, G., Vinyals, O., & Dean, J. (2015). Distilling the Knowledge in a Neural Network. arXiv preprint arXiv:1503.02531.

46. Bucila, C., Caruana, R., & Niculescu-Mizil, A. (2006). Model Compression. Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD).

47. Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., & Adam, H. (2017). MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv preprint arXiv:1704.04861.

48. Gou, J., Yu, B., Maybank, S. J., & Tao, D. (2021). Knowledge Distillation: A Survey. International Journal of Computer Vision, 129(6), 1789-1819.

49. Romero, A., Ballas, N., Kahou, S. E., Chassang, A., Gatta, C., & Bengio, Y. (2015). FitNets: Hints for Thin Deep Nets. International Conference on Learning Representations (ICLR).

50. Park, W., Kim, D., Lu, Y., & Cho, M. (2019). Relational Knowledge Distillation. IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

51. Zheng, Z., Ye, R., Wang, P., Wang, D., Ren, J., & Zuo, W. (2022). Localization Distillation for Object Detection. International Journal of Computer Vision, 130(8), 1923-1943.

52. Chen, G., Choi, W., Yu, X., Han, T., & Chandraker, M. (2017). Learning Efficient Object Detection Models with Knowledge Distillation. Advances in Neural Information Processing Systems (NeurIPS).

53. Wang, T., Yuan, L., Zhang, X., & Feng, J. (2019). Distilling Object Detectors with Fine-grained Feature Imitation. IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

54. Jocher, G., et al. (2023). Ultralytics YOLOv8. https://github.com/ultralytics/ultralytics.

55. Zhang, Y., Wang, C., Wang, X., Zeng, W., & Liu, W. (2022). YOLOX-Slim: A Real-time Distillation Approach to Slimming Down Object Detector. IEEE/CVF Winter Conference on Applications of Computer Vision (WACV).

56. Mehta, R., & Ozturk, C. (2022). MobileDets: Efficient YOLOv5 Distillation for Mobile Object Detection. arXiv preprint arXiv:2204.02808.

57. Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., & Chen, L. C. (2018). MobileNetV2: Inverted Residuals and Linear Bottlenecks. IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

58. Guo, J., Han, K., Wang, Y., Wu, H., Chen, X., Xu, C., Xu, C., & Tang, J. (2022). Distilling Object Detectors via Decoupled Features. IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

59. Lin, T. Y., Goyal, P., Girshick, R., He, K., & Dollár, P. (2017). Focal Loss for Dense Object Detection. IEEE International Conference on Computer Vision (ICCV).

60. Li, Q., Jin, S., & Yan, J. (2017). Mimicking Very Efficient Network for Object Detection. IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

61. Zhao, Z., Hou, X., Wang, L., Ma, L., Zhang, N., & Feng, Y. (2020). On the Connection of Knowledge Distillation and Label Smoothing in Neural Network Learning. arXiv preprint arXiv:2010.08243.

62. Heo, B., Kim, J., Yun, S., Park, H., Kwak, N., & Choi, J. Y. (2019). A Comprehensive Overhaul of Feature Distillation. IEEE International Conference on Computer Vision (ICCV).

63. Zhang, Y., Xiang, T., Hospedales, T. M., & Lu, H. (2018). Deep Mutual Learning. IEEE Conference on Computer Vision and Pattern Recognition (CVPR).

64. Chen, D., Mei, J. P., Wang, C., Feng, Y., & Chen, C. (2020). Online Knowledge Distillation with Diverse Peers. AAAI Conference on Artificial Intelligence.

65. Furlanello, T., Lipton, Z., Tschannen, M., Itti, L., & Anandkumar, A. (2018). Born Again Neural Networks. International Conference on Machine Learning (ICML).

66. Goyal, P., Dollár, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., & He, K. (2017). Accurate, Large Minibatch SGD: Training ImageNet in 1 Hour. arXiv preprint arXiv:1706.02677.

67. Loshchilov, I., & Hutter, F. (2017). SGDR: Stochastic Gradient Descent with Warm Restarts. International Conference on Learning Representations (ICLR).

68. Padilla, R., Netto, S. L., & da Silva, E. A. (2020). A Survey on Performance Metrics for Object-Detection Algorithms. International Conference on Systems, Signals and Image Processing (IWSSIP).

# Appendices

## Appendix A: Jenkins Pipeline Code

The complete source code for the implementation described in this thesis is available in the following GitHub repository:

https://github.com/RajeshRamadas/yolov8-model-optimization.git

The repository contains all code artifacts including:

- Jenkins pipeline configurations
- Dataset validation and augmentation scripts
- Neural Architecture Search implementation
- Evaluation scripts - Raspberry Pi deployment code

## Appendix B: Neural Architecture Search Configuration

```yaml
# YOLOv8 Neural Architecture Search Configuration

# Basic search space parameters
basic_search_space:
  depth_multiple:
    - 0.33
    - 0.5
    - 0.67
    - 1.0
  width_multiple:
    - 0.25
    - 0.5
    - 0.75
    - 1.0
  img_size:
    - 320
    - 448
    - 640
  kernel_size:
    - 1
    - 3
    - 5
    - 7

# Advanced search parameters
advanced_search_space:
  optimizer:
    - SGD
    - Adam
    - AdamW
  lr0:
    - 0.001
    - 0.01
    - 0.1
  lrf:
    - 0.01
    - 0.1
  momentum:
    - 0.8
    - 0.9
    - 0.95
    - 0.99
  weight_decay:
    - 0.0005
    - 0.001
    - 0.0001
  warmup_epochs:
    - 0
    - 3
    - 5
  augment:
    - true
    - false
  mosaic:
    - 0.0
    - 0.5
    - 1.0
  model_type:
    - yolov8n
    - yolov8s

# Optimization objectives and weights
optimization:
  accuracy_weight: 0.6  # mAP50-95
  speed_weight: 0.2     # FPS
  size_weight: 0.2      # Model size in MB

# Search parameters
search:
  trials: 20          # Total number of trials to run
  parallel: 4         # Number of parallel trials
  epochs: 5           # Epochs per trial
  early_stopping: true  # Stop trials early if performance is poor
```

## Appendix C: YOLOv8 Balanced Dataset Report

Generated on: 2025-04-14 10:13:14

### *Dataset Information*

- Original dataset: /var/lib/jenkins/workspace/yolov8-model-optimization_v5/downloads/extracted_dataset/vehicles.v2-release.yolov8
- Output dataset: augmented_dataset
- Augmentation factor: 5

### *Original Dataset Statistics*

| Set | Images | Objects |
|---|---|---|
| Training | 2634 | 31905 |
| Validation | 0 | 0 |
| Test | 458 | 6222 |

### *Class Imbalance Analysis*

Threshold for minority classes: 10.0%

### *Identified Minority Classes*

| Class ID | Class Name | Original Count | Original % | Final Count | Final % | Increase Factor |
|---|---|---|---|---|---|---|
| 0 | big bus | 433 | 1.36% | 6220 | 1.70% | 14.36x |
| 1 | big truck | 1822 | 5.71% | 16596 | 4.54% | 9.11x |
| 2 | bus-l- | 390 | 1.22% | 5529 | 1.51% | 14.18x |

| Class ID | Class Name | Original Count | Original % | Final Count | Final % | Increase Factor |
|---|---|---|---|---|---|---|
| 3 | bus-s- | 120 | 0.38% | 2263 | 0.62% | 18.86x |
| 5 | mid truck | 359 | 1.13% | 4992 | 1.36% | 13.91x |
| 6 | small bus | 191 | 0.60% | 3531 | 0.97% | 18.49x |
| 8 | truck-l- | 1701 | 5.33% | 25552 | 6.98% | 15.02x |
| 9 | truck-m- | 2760 | 8.65% | 38318 | 10.47% | 13.88x |
| 10 | truck-s- | 1111 | 3.48% | 16836 | 4.60% | 15.15x |
| 11 | truck-xl- | 641 | 2.01% | 10693 | 2.92% | 16.68x |

### Augmentation Summary

- Total original training images: 2634
- Total augmented images generated: 27260
- Final training set size: 29894

### Final Class Distribution

| Class ID | Class Name | Count | Percentage | Minority? |
|---|---|---|---|---|
| 0 | big bus | 6220 | 1.70% | Yes |
| 1 | big truck | 16596 | 4.54% | Yes |
| 2 | bus-l- | 5529 | 1.51% | Yes |

| Class ID | Class Name | Count | Percentage | Minority? |
|---|---|---|---|---|
| 3 | bus-s- | 2263 | 0.62% | Yes |
| 4 | car | 204609 | 55.93% | No |
| 5 | mid truck | 4992 | 1.36% | Yes |
| 6 | small bus | 3531 | 0.97% | Yes |
| 7 | small truck | 30682 | 8.39% | No |
| 8 | truck-l- | 25552 | 6.98% | Yes |
| 9 | truck-m- | 38318 | 10.47% | Yes |
| 10 | truck-s- | 16836 | 4.60% | Yes |
| 11 | truck-xl- | 10693 | 2.92% | Yes |

### *Training Recommendations*

To train YOLOv8 with this balanced dataset, use the following command:

```
yolo task=detect train data=augmented_dataset/dataset.yaml model=yolov8n.pt epochs=100
```

## Additional YOLOv8 Settings for Imbalanced Data

Consider these additional options when training:

1. **Longer Training**: For imbalanced datasets, consider increasing epochs to 150-200
2. **Higher IoU Thresholds**: Use --iou 0.7 for stricter box predictions
3. **Learning Rate Scheduling**: Try cosine scheduler with --cos-lr
4. **Heavy Augmentation**: Add more augmentation during training with --augment

5. **Class Weights**: For persistent imbalance, add class weights in the YAML file

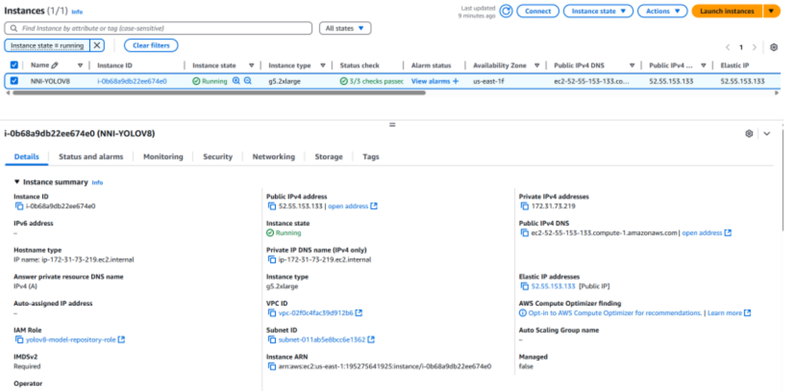
## Appendix D: env File

```
# AWS Credentials
AWS_ACCESS_KEY_ID="AKIAS252XXXXXXXLXUFI"
AWS_SECRET_ACCESS_KEY="rUchxcsssdsdsdfsxxddfdfdfdfb2IyzC"
AWS_DEFAULT_REGION=us-east-1

# S3 Configuration
S3_BUCKET_NAME="yolov8-model-repository"
S3_FOLDER_PATH="yolov8_model_custom"

# Application Settings
DEFAULT_ZIP_NAME=archive.zip
DELETE_ZIP_AFTER_UPLOAD=true
METADATA_KEY=versioning/metadata.json
```

## Appendix E: AWS EC2 Instance



Instance state = running

| ☑ | Name | Instance ID | Instance state | Instance type | Status check | Alarm status | Availability Zone | Public IPv4 DNS | Public IPv4 ... | Elastic IP |
|---|---|---|---|---|---|---|---|---|---|---|
| ☑ | NNI-YOLOV8 | i-0b68a9db22ee674e0 | Running | g5.2xlarge | 3/3 checks passec | View alarms + | us-east-1f | ec2-52-55-153-133.co... | 52.55.153.133 | 52.55.153.133 |

i-0b68a9db22ee674e0 (NNI-YOLOV8)

▼ Instance summary Info

Instance ID
i-0b68a9db22ee674e0

IPv6 address
–

Hostname type
IP name: ip-172-31-73-219.ec2.internal

Answer private resource DNS name
IPv4 (A)

Auto-assigned IP address
–

IAM Role
yolov8-model-repository-role

IMDSv2
Required

Operator

Public IPv4 address
52.55.153.133 | open address

Instance state
Running

Private IP DNS name (IPv4 only)
ip-172-31-73-219.ec2.internal

Instance type
g5.2xlarge

VPC ID
vpc-02f0c4fac39d912b6

Subnet ID
subnet-011ab5e8bcc6e1362

Instance ARN
arn:aws:ec2:us-east-1:195275641925:instance/i-0b68a9db22ee674e0

Private IPv4 addresses
172.31.73.219

Public IPv4 DNS
ec2-52-55-153-133.compute-1.amazonaws.com | open address

Elastic IP addresses
52.55.153.133 [Public IP]

AWS Compute Optimizer finding
Opt-in to AWS Compute Optimizer for recommendations. | Learn more

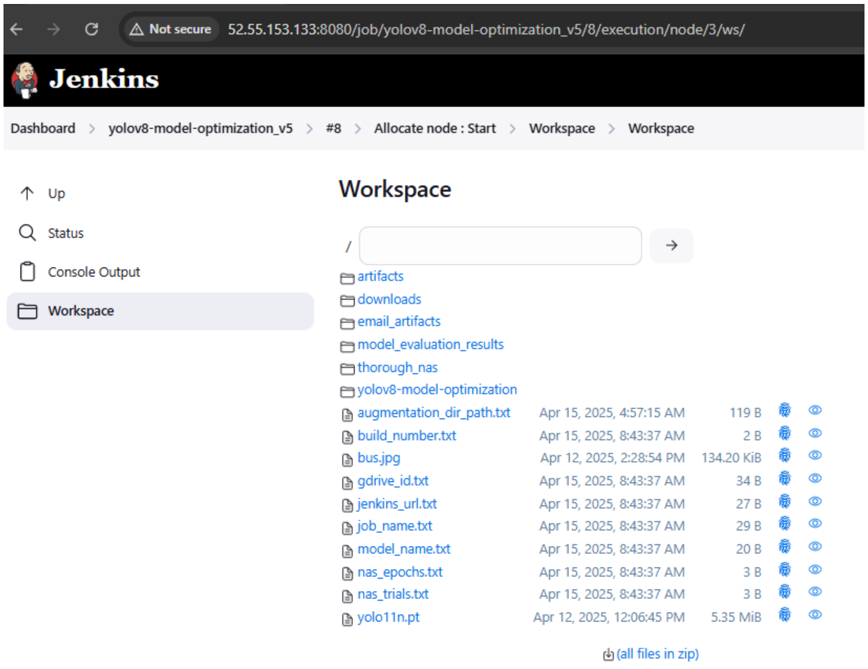
Auto Scaling Group name
–

Managed
false

*Figure 42:AWS EC2 Instance for Jenkins*

## Appendix F: Jenkins Workspace output

GitHub repository for sample Jenkins output


Not secure
52.55.153.133:8080/job/yolov8-model-optimization_v5/8/execution/node/3/ws/
Jenkins
Dashboard > yolov8-model-optimization_v5 > #8 > Allocate node : Start > Workspace > Workspace
Up
Status
Console Output
Workspace

Workspace

/

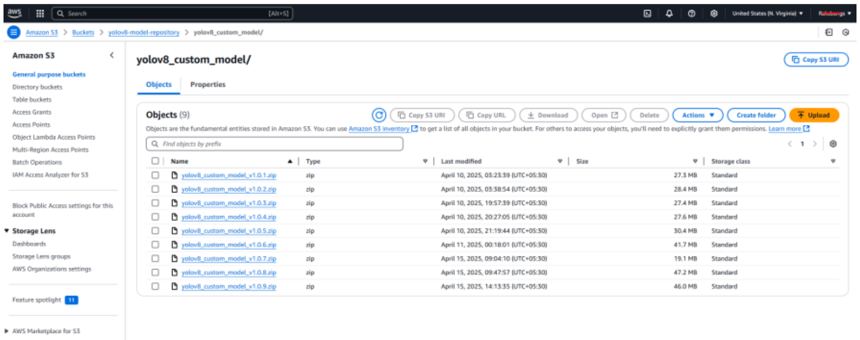
- artifacts
- downloads
- email_artifacts
- model_evaluation_results
- thorough_nas
- yolov8-model-optimization

| | | |
|---|---|---|
| augmentation_dir_path.txt | Apr 15, 2025, 4:57:15 AM | 119 B |
| build_number.txt | Apr 15, 2025, 8:43:37 AM | 2 B |
| bus.jpg | Apr 12, 2025, 2:28:54 PM | 134.20 KiB |
| gdrive_id.txt | Apr 15, 2025, 8:43:37 AM | 34 B |
| jenkins_url.txt | Apr 15, 2025, 8:43:37 AM | 27 B |
| job_name.txt | Apr 15, 2025, 8:43:37 AM | 29 B |
| model_name.txt | Apr 15, 2025, 8:43:37 AM | 20 B |
| nas_epochs.txt | Apr 15, 2025, 8:43:37 AM | 3 B |
| nas_trials.txt | Apr 15, 2025, 8:43:37 AM | 3 B |
| yolo11n.pt | Apr 12, 2025, 12:06:45 PM | 5.35 MiB |

(all files in zip)

*Figure 43:Jenkins workspace*

## Appendix G: AWS S3 Bucket for version control



yolov8_custom_model/

Objects (9)

Objects are the fundamental entities stored in Amazon S3. You can use Amazon S3 inventory to get a list of all objects in your bucket. For others to access your objects, you'll need to explicitly grant them permissions. Learn more

| Name | Type | Last modified | Size | Storage class |
|---|---|---|---|---|
| yolov8_custom_model_v1.0.1.zip | zip | April 10, 2025, 03:23:39 (UTC+05:30) | 27.3 MB | Standard |
| yolov8_custom_model_v1.0.2.zip | zip | April 10, 2025, 03:38:54 (UTC+05:30) | 28.4 MB | Standard |
| yolov8_custom_model_v1.0.3.zip | zip | April 10, 2025, 19:57:39 (UTC+05:30) | 27.4 MB | Standard |
| yolov8_custom_model_v1.0.4.zip | zip | April 10, 2025, 20:27:05 (UTC+05:30) | 27.6 MB | Standard |
| yolov8_custom_model_v1.0.5.zip | zip | April 10, 2025, 21:19:44 (UTC+05:30) | 30.4 MB | Standard |
| yolov8_custom_model_v1.0.6.zip | zip | April 11, 2025, 00:18:01 (UTC+05:30) | 41.7 MB | Standard |
| yolov8_custom_model_v1.0.7.zip | zip | April 15, 2025, 09:04:10 (UTC+05:30) | 19.1 MB | Standard |
| yolov8_custom_model_v1.0.8.zip | zip | April 15, 2025, 09:47:57 (UTC+05:30) | 47.2 MB | Standard |
| yolov8_custom_model_v1.0.9.zip | zip | April 15, 2025, 14:13:35 (UTC+05:30) | 46.0 MB | Standard |

*Figure 44:AWS S3 Bucket for version control*