Project Documentation

This is the documentation about the implementation of an object detection system for industrial environments using YOLOv5. The project focuses on detecting objects such as tuggers, cabinets, STRs, boxes, and forklifts.

Hassan Khadra

HassanKhxd@gmail.com

YOLO Object Detection

InMind Final Project

Table Of Contents

[Executive Summary 5](#_Toc194424932)

[Introduction 6](#_Toc194424933)

[Part 1: Data Preparation & Visualization 6](#_Toc194424934)

[1. Loading the Dataset 6](#_Toc194424935)

[a. Code Implementation: Load Dataset 7](#_Toc194424936)

[2. Visualization of Labeled Images 7](#_Toc194424937)

[a. Code Implementation: Visualization 8](#_Toc194424938)

[3. Dataset Augmentation 8](#_Toc194424939)

[a. Code Implementation: Data Augmentation 8](#_Toc194424940)

[4. Augmentation Results 10](#_Toc194424941)

[5. Dataset Splitting 10](#_Toc194424942)

[a. Code Implementation: Dataset Splitting 11](#_Toc194424943)

[Part 2: Model Training & Evaluation 12](#_Toc194424944)

[1. YOLOv5 Model Training 12](#_Toc194424945)

[a. Dataset Preparation for YOLOv5 12](#_Toc194424946)

[b. YOLOv5 Dataset Configuration 13](#_Toc194424947)

[c. Training Process 13](#_Toc194424948)

[d. Model Evaluation and Hyperparameter Tuning 14](#_Toc194424949)

[2. TensorBoard Visualization 16](#_Toc194424950)

[a. Model Performance Comparison 16](#_Toc194424951)

[b. F1 Score Curves 16](#_Toc194424952)

[c. Mean Average Precision (mAP) 18](#_Toc194424953)

[b. Loss Comparison 18](#_Toc194424954)

[c. Learning Rate 19](#_Toc194424955)

[d. Confusion Matrices 20](#_Toc194424956)

[Part 3: Model Deployment & Inference 22](#_Toc194424957)

[1. Export Models to ONNX Format 22](#_Toc194424958)

[2. Network Visualization with Netron 22](#_Toc194424959)

[3. REST API Implementation with FastAPI 23](#_Toc194424960)

[a. API Endpoints 23](#_Toc194424961)

[b. Architecture 23](#_Toc194424962)

[c. Sample JSON Output 24](#_Toc194424963)

[d. Testing FastAPI Endpoints 24](#_Toc194424964)

[4. Docker Containerization 26](#_Toc194424965)

[a. Dockerfile 26](#_Toc194424966)

[b. Docker Compose 26](#_Toc194424967)

[c. Running the Containerized API 27](#_Toc194424968)

[Conclusion 28](#_Toc194424969)

[References 29](#_Toc194424970)

[Appendix 30](#_Toc194424971)

[System Requirements 30](#_Toc194424972)

[Model Configuration Details 30](#_Toc194424973)

Table Of Figures

[Figure 1: Implementation of dataset loading functionality 7](#_Toc194425009)

[Figure 2: Implementation of bounding box visualization 8](#_Toc194425010)

[Figure 3: Implementation of data augmentation techniques 9](#_Toc194425011)

[Figure 4: Example visualized image after augmenting it. 10](#_Toc194425012)

[Figure 5: Implementation of dataset splitting functionality 11](#_Toc194425013)

[Figure 6: Yolov5 repository setup and initial preparation steps 12](#_Toc194425014)

[Figure 7: Dataset configuration YAML for YOLOv5 13](#_Toc194425015)

[Figure 8: Training command execution and initial output 13](#_Toc194425016)

[Figure 9: Model architecture summary for YOLOv5s 14](#_Toc194425017)

[Figure 10: Training completion output with final metrics 14](#_Toc194425018)

[Figure 11: Evaluation results of the trained model on test dataset 15](#_Toc194425019)

[Figure 12: Training with modified hyperparameters 15](#_Toc194425020)

[Figure 13: Completion output of hyperparameter-tuned training 16](#_Toc194425021)

[Figure 14: F1 score curve for the standard model 17](#_Toc194425022)

[Figure 15: F1 score curve for the hyperparameter-tuned model 17](#_Toc194425023)

[Figure 16: Comparison of mAP@0.5 and mAP@0.5:0.95 between both models 18](#_Toc194425024)

[Figure 17: Box loss, class loss, and object loss during training for both models 18](#_Toc194425025)

[Figure 18: Validation losses for both models 19](#_Toc194425026)

[Figure 19: Learning rate schedules for both models 19](#_Toc194425027)

[Figure 20: Confusion matrix for the standard model 20](#_Toc194425028)

[Figure 21: Confusion matrix for the hyperparameter-tuned model 21](#_Toc194425029)

[Figure 22: Netron visualization of the standard YOLOv5s model architecture 22](#_Toc194425030)

[Figure 23: Netron visualization of the hyperparameter-tuned model architecture 23](#_Toc194425031)

[Figure 24: FastAPI docs page showing project endpoints 24](#_Toc194425032)

[Figure 25:' /models' endpoint returning the available models 25](#_Toc194425033)

[Figure 26: '/Inference' Endpoint showing the json output which includes the bounding boxes 25](#_Toc194425034)

[Figure 27: Image inference endpoint showing the image and the returned bounding boxes 26](#_Toc194425035)

[Figure 28: Docker Compose build process initiation 27](#_Toc194425036)

[Figure 29: Successful completion of the Docker container build 27](#_Toc194425037)

[Figure 30: Container logs showing API endpoint requests and responses 28](#_Toc194425038)

# Executive Summary

This Final Project implements an object detection system designed specifically for industrial environments. This documentation details the development of a computer vision solution capable of detecting various industrial objects including tuggers, cabinets, STRs, boxes, and forklifts. The project utilizes the YOLOv5 architecture, a state-of-the-art real-time object detection framework.

The documentation covers the complete workflow from data preparation and model training to deployment and inference implementation. Performance evaluation metrics are presented, comparing standard and hyperparameter-tuned models. The final solution includes a containerized REST API for seamless integration into industrial systems.

# Introduction

Object detection in industrial environments presents unique challenges due to varying lighting conditions, occlusions, and the need for high accuracy to ensure safety and efficiency. The Project addresses these challenges by implementing a customized YOLOv5-based detection system.

This project is structured in three main parts:

1. Data Preparation & Visualization
2. Model Training & Evaluation
3. Model Deployment & Inference

Each section describes the methodologies, implementation details, and results achieved throughout the development process.

# Part 1: Data Preparation & Visualization

## Loading the Dataset

A custom dataset loader was implemented using PyTorch's DataLoader to efficiently process images and annotations for training. The loader handles batch processing and ensures proper formatting of the input data.

### Code Implementation: Load Dataset

Figure 1: Implementation of dataset loading functionality

## Visualization of Labeled Images

To verify the correctness of annotations, a visualization function was developed. This function overlays bounding boxes on original images, allowing for visual inspection of the labeled data.

### Code Implementation: Visualization



Figure 2: Implementation of bounding box visualization

## Dataset Augmentation

Data augmentation techniques were applied to improve model generalization and robustness. The Albumentations library was used to implement various transformations including horizontal flips, rotations, and contrast adjustments.

### Code Implementation: Data Augmentation

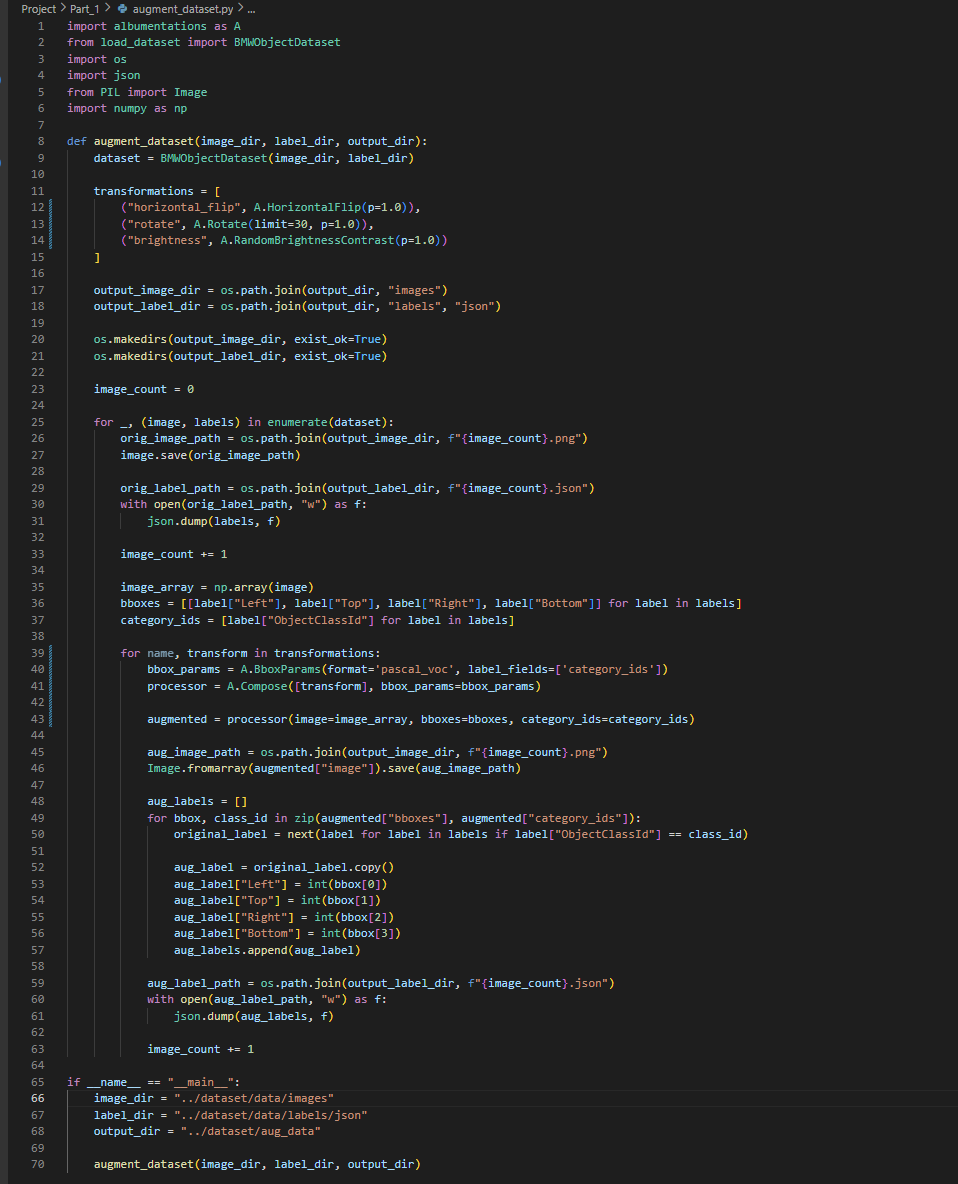


Figure 3: Implementation of data augmentation techniques

## Augmentation Results

Figure 4: Example visualized image after augmenting it.

## Dataset Splitting

The dataset was divided into training and validation sets using an 80/20 split ratio. This ensures proper evaluation of model performance on unseen data.

### Code Implementation: Dataset Splitting



Figure 5: Implementation of dataset splitting functionality

# Part 2: Model Training & Evaluation

## YOLOv5 Model Training

The YOLOv5 architecture was selected for its balance of accuracy and inference speed. The model was trained using transfer learning from pre-trained weights.

### Dataset Preparation for YOLOv5

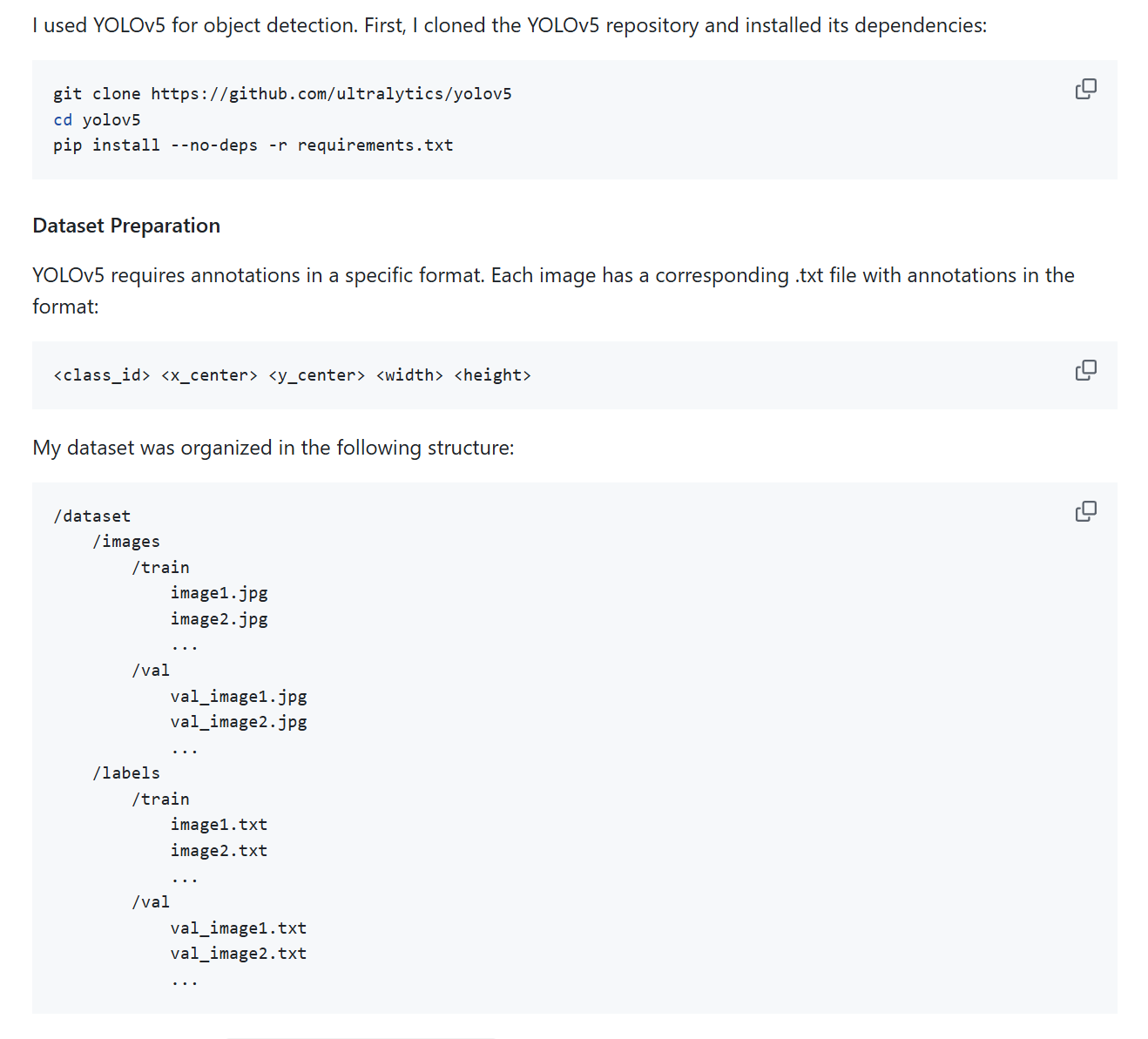
YOLOv5 requires a specific format for annotations, where each image has a corresponding text file containing normalized bounding box coordinates and class IDs. The dataset structure follows the YOLOv5 convention with separate directories for training and validation images and labels.

Figure 6: Yolov5 repository setup and initial preparation steps

### YOLOv5 Dataset Configuration

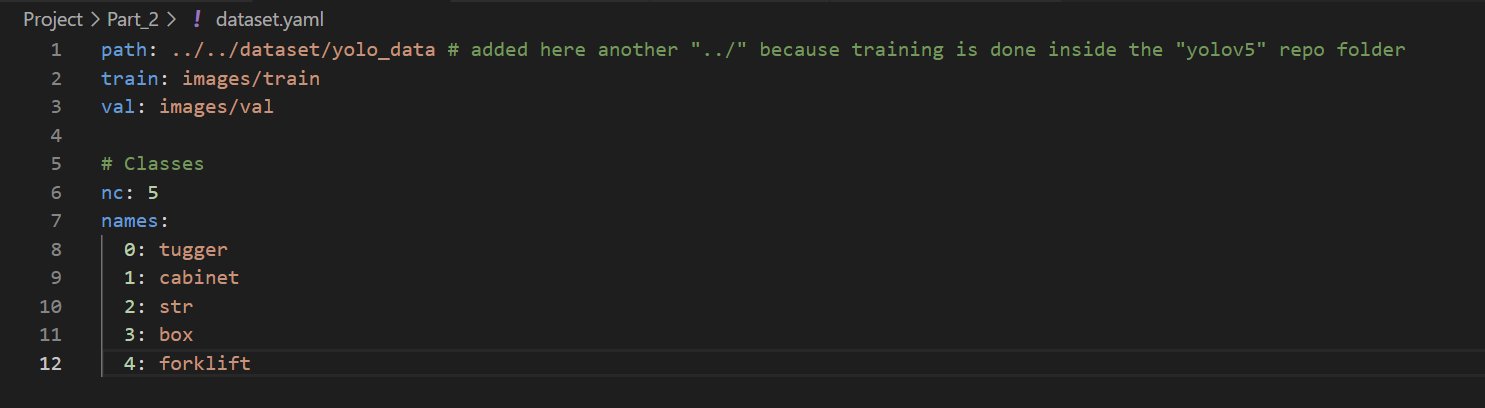


Figure 7: Dataset configuration YAML for YOLOv5

### Training Process

The model was trained using the YOLOv5 training script with the following parameters:

* Image size: 640×640 pixels
* Batch size: 16
* Number of epochs: 50
* Weights: YOLOv5s pre-trained model

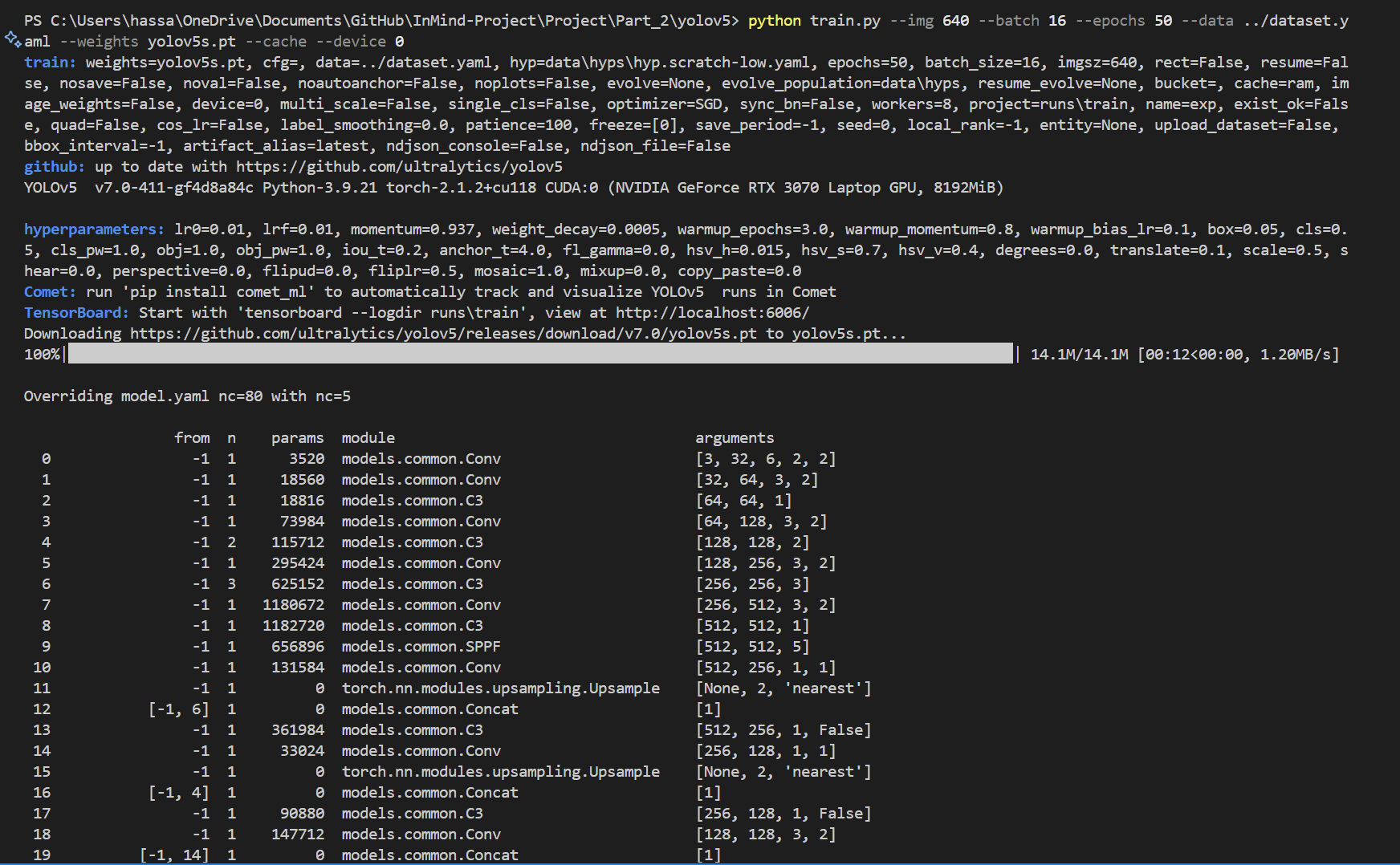


Figure 8: Training command execution and initial output

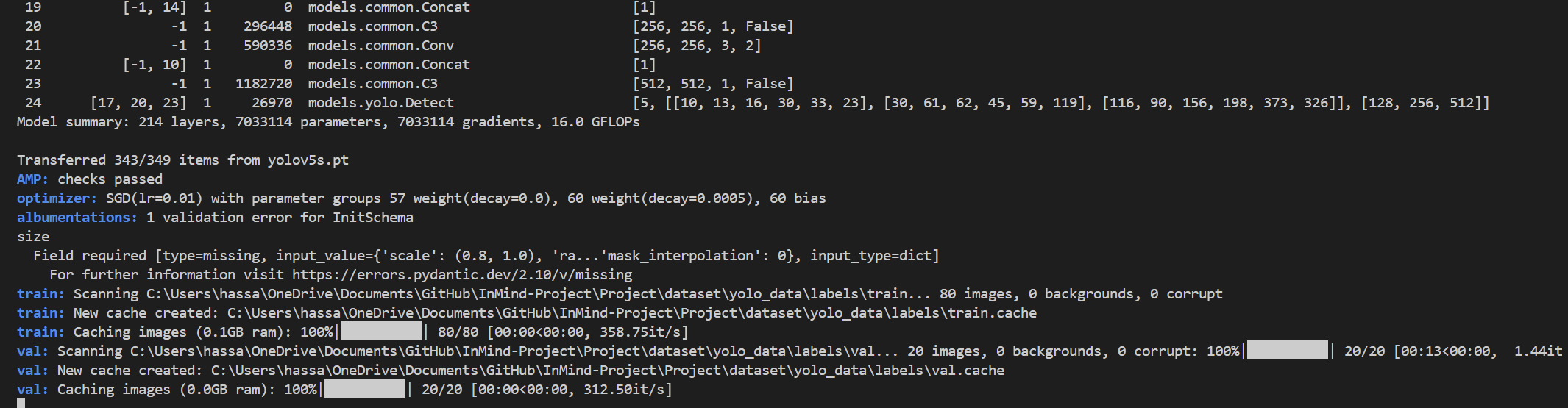
Model architecture:

Figure 9: Model architecture summary for YOLOv5s

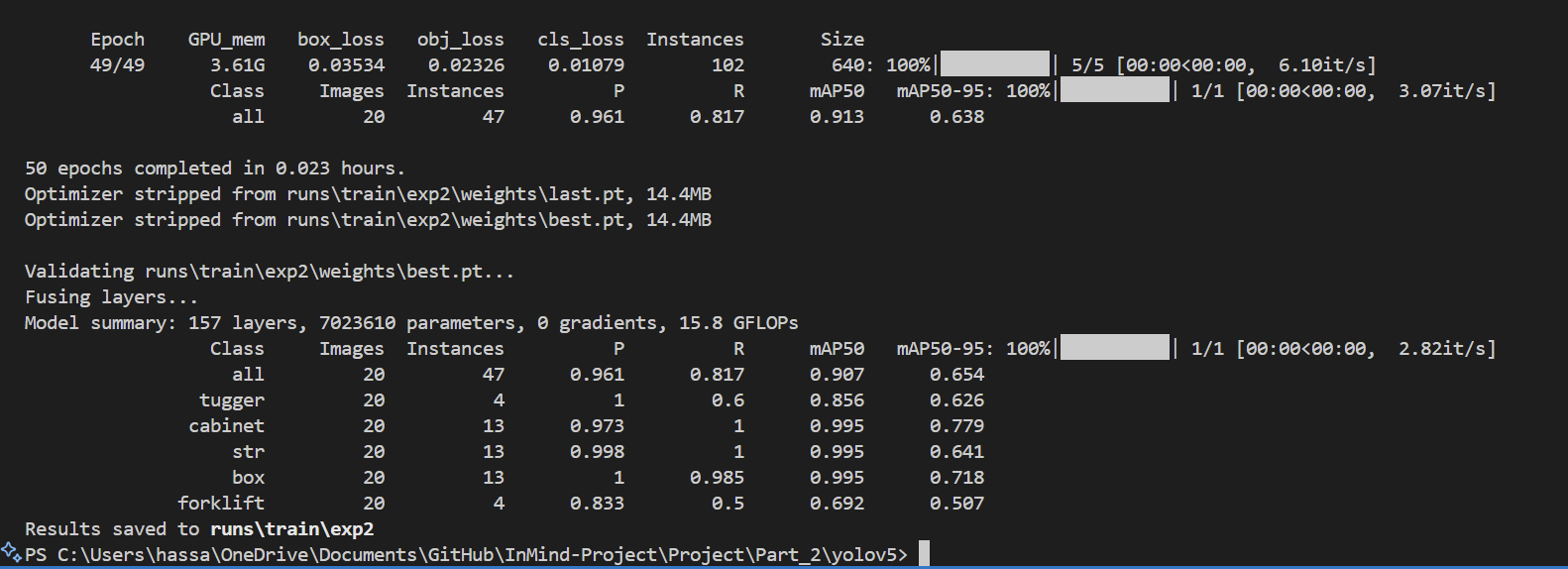


Figure 10: Training completion output with final metrics

### Model Evaluation and Hyperparameter Tuning

After initial training, the model was evaluated on the test dataset to assess its performance. Based on these results, hyperparameter tuning was performed to potentially improve detection accuracy.

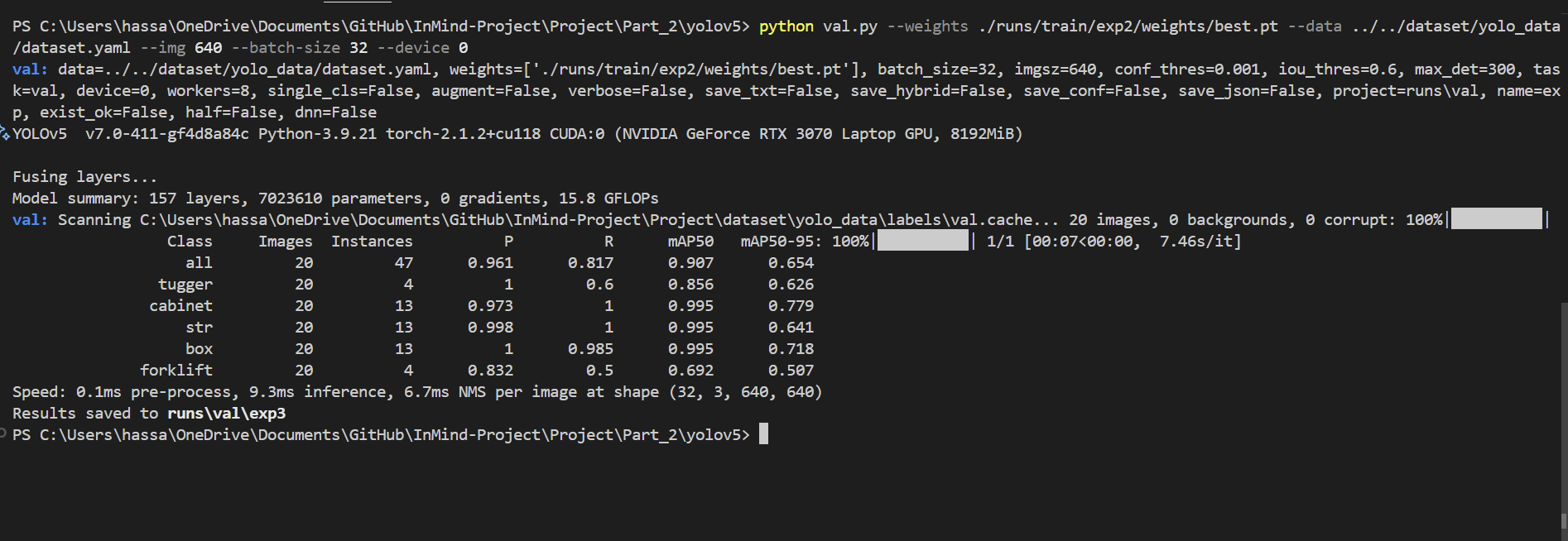
A second model was trained with modified hyperparameters using the hyp.scratch-high.yaml configuration, which includes higher augmentation settings.

Figure 11: Evaluation results of the trained model on test dataset



Completion output of hyperparameter-tuned training:

Figure 12: Training with modified hyperparameters



Figure 13: Completion output of hyperparameter-tuned training

## TensorBoard Visualization

TensorBoard was used to visualize and compare training metrics between the standard and hyperparameter-tuned models. Both models were trained with the same dataset but different hyperparameter configurations.

### Model Performance Comparison

Various metrics were used to compare the performance of both models, including F1 score, mean average precision (mAP), and loss curves.

### F1 Score Curves

Standard model:

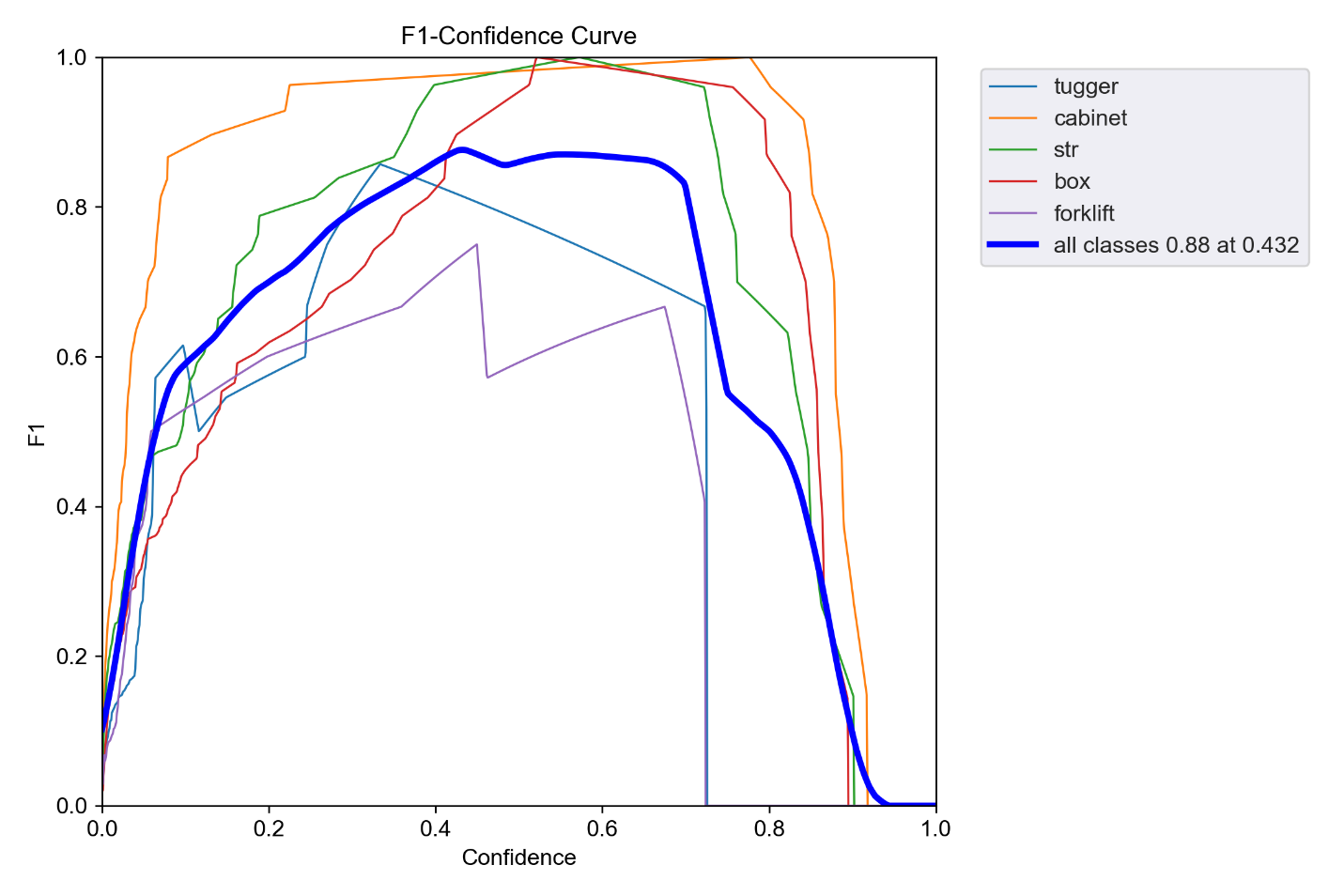


Figure 14: F1 score curve for the standard model

Hyperparameter-tuned model:

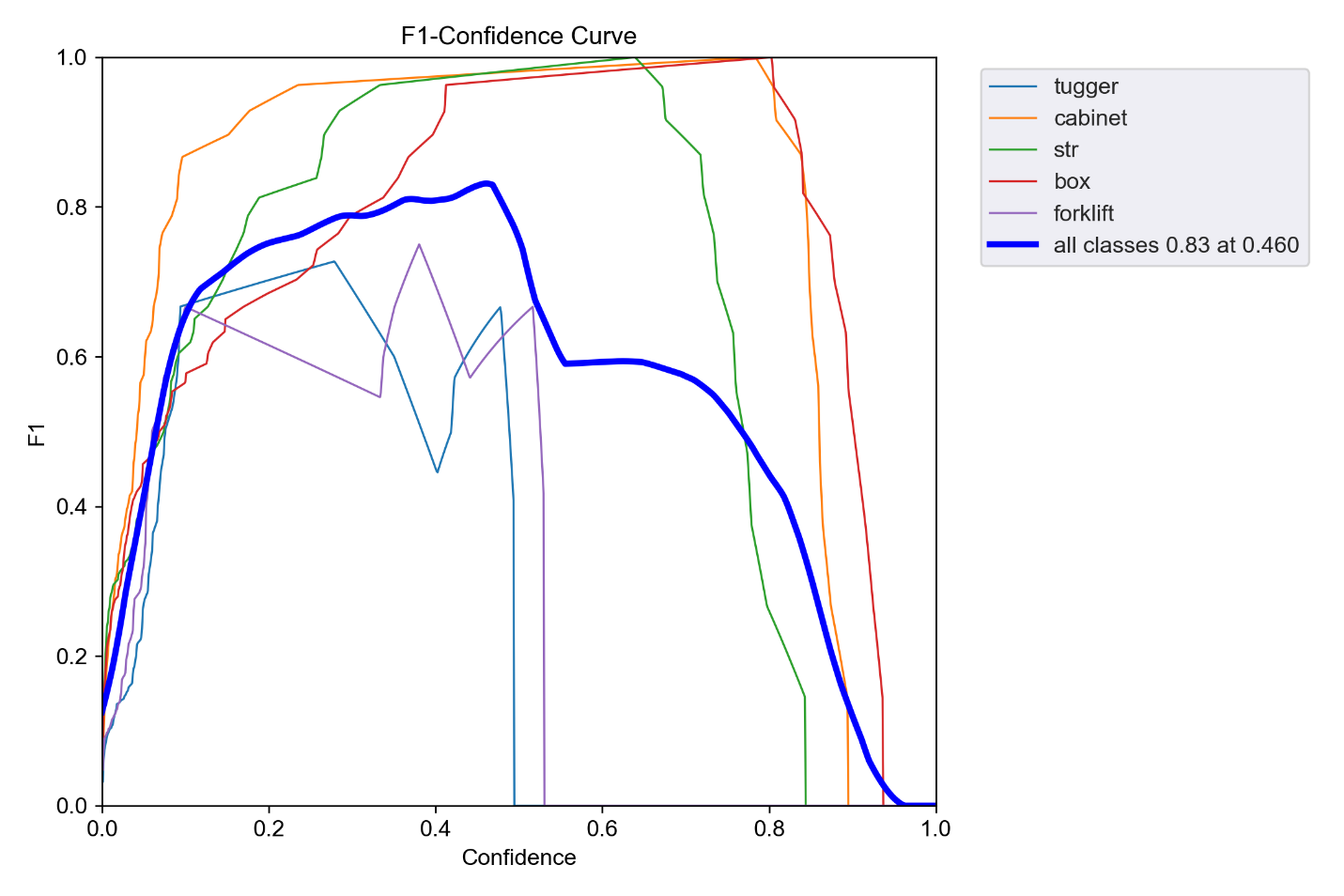


Figure 15: F1 score curve for the hyperparameter-tuned model

### Mean Average Precision (mAP)

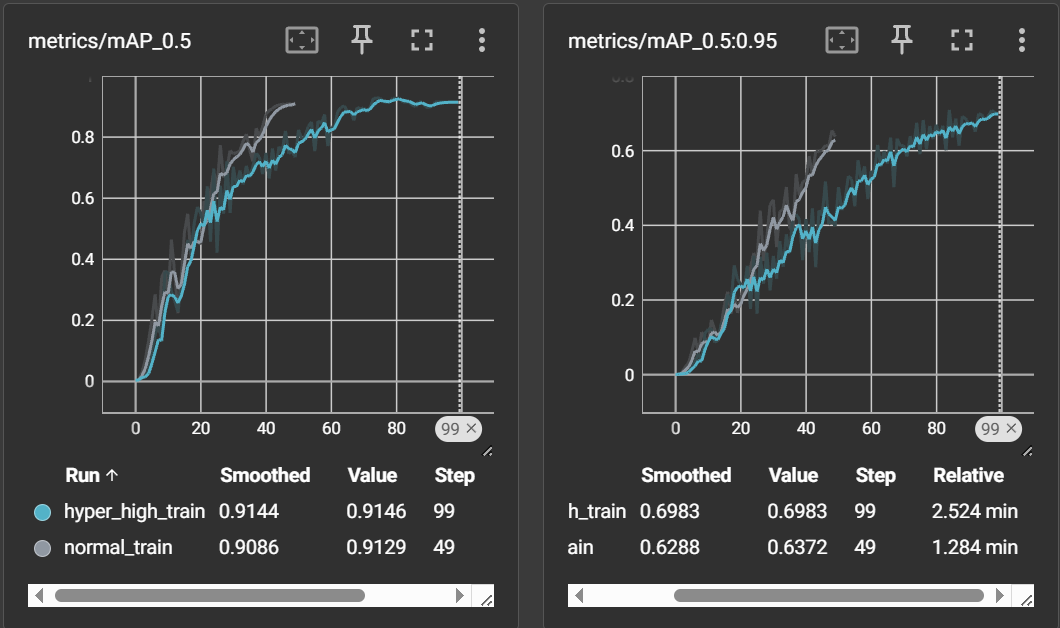


Figure 16: Comparison of mAP@0.5 and mAP@0.5:0.95 between both models

### Loss Comparison

Loss during training for both models:

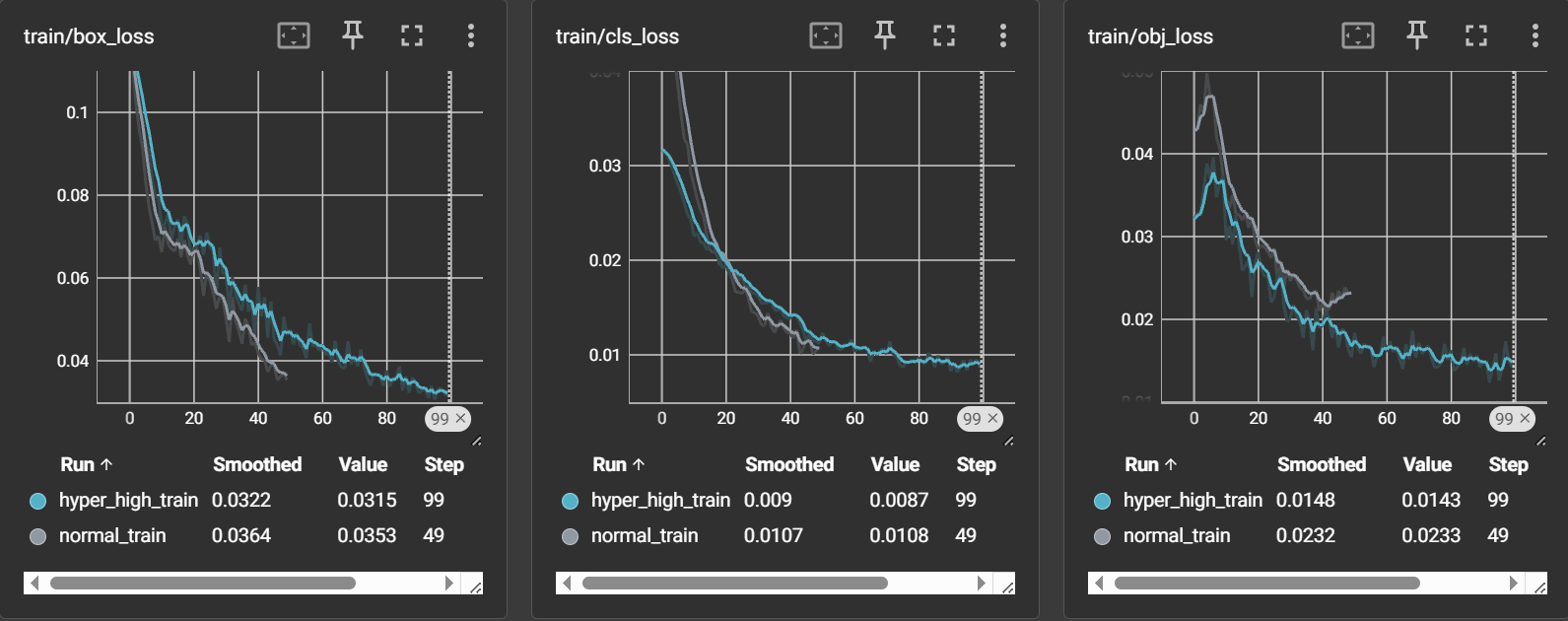


Figure 17: Box loss, class loss, and object loss during training for both models

Validation losses for both models:

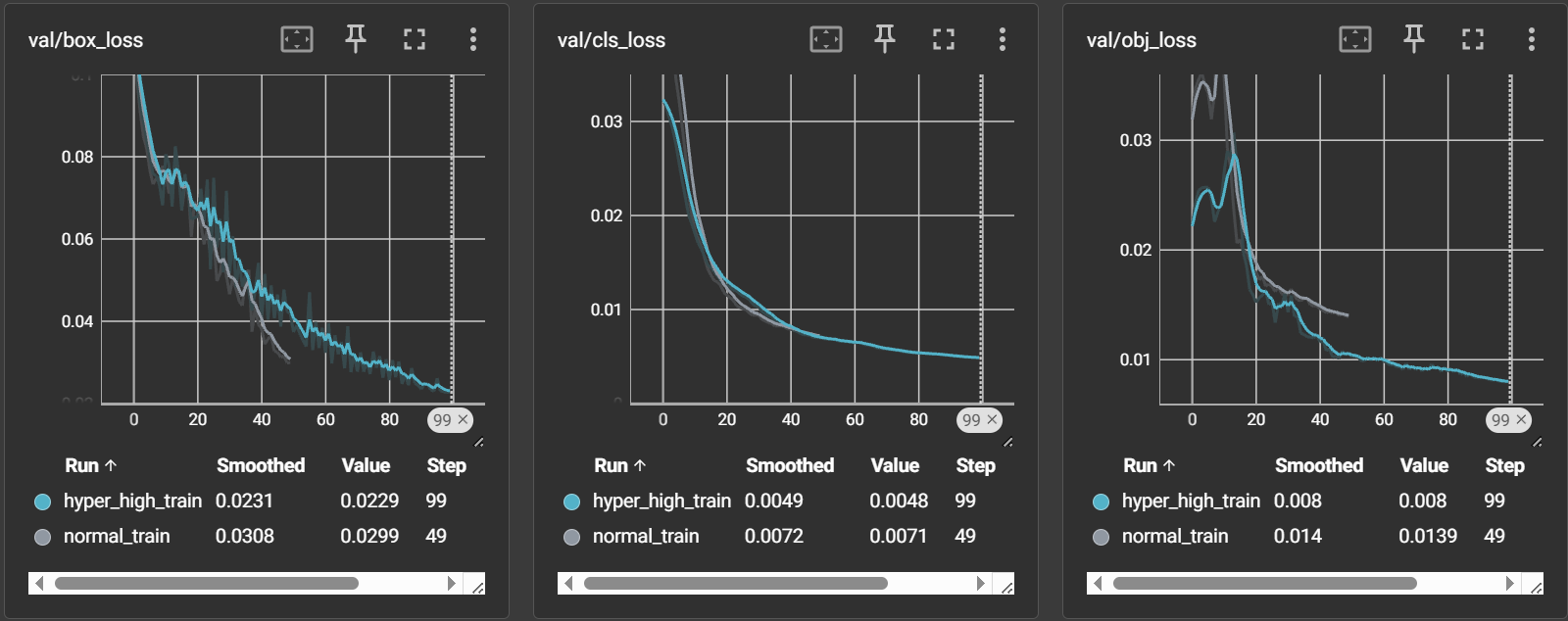


Figure 18: Validation losses for both models

### Learning Rate

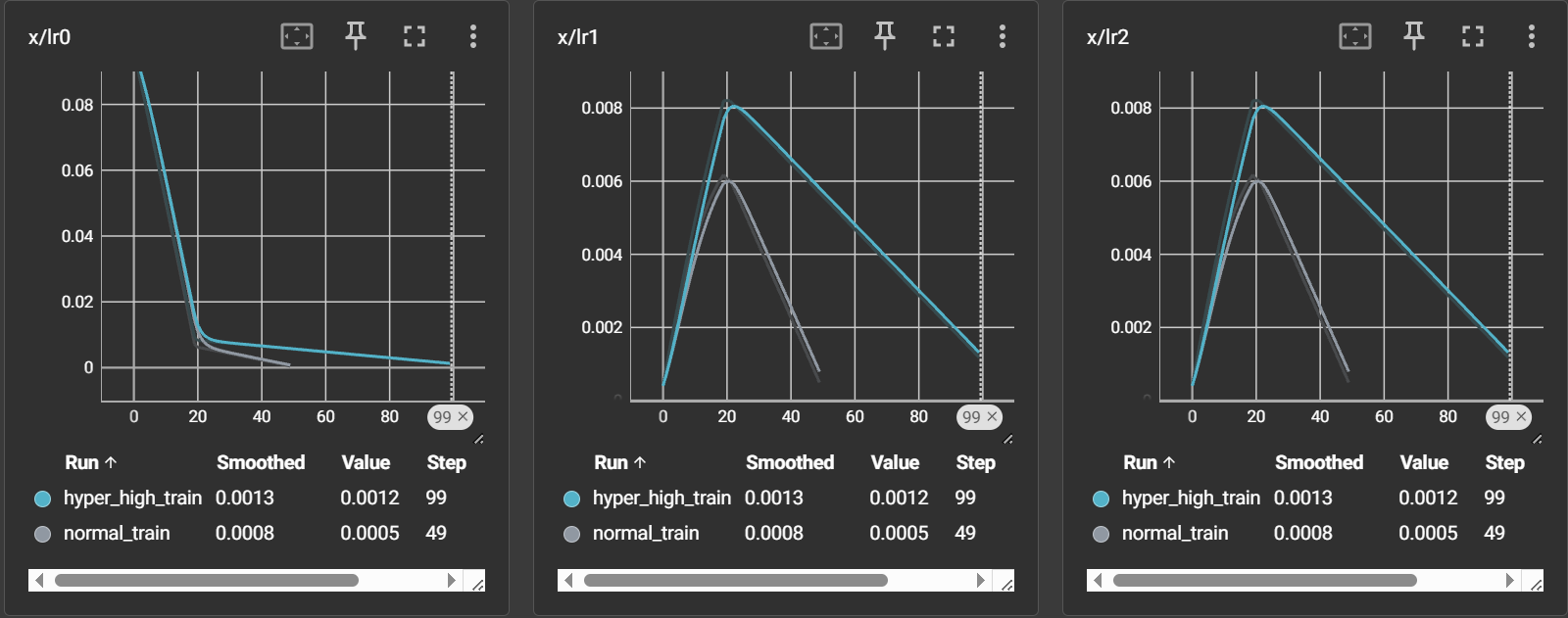


Figure 19: Learning rate schedules for both models

### Confusion Matrices

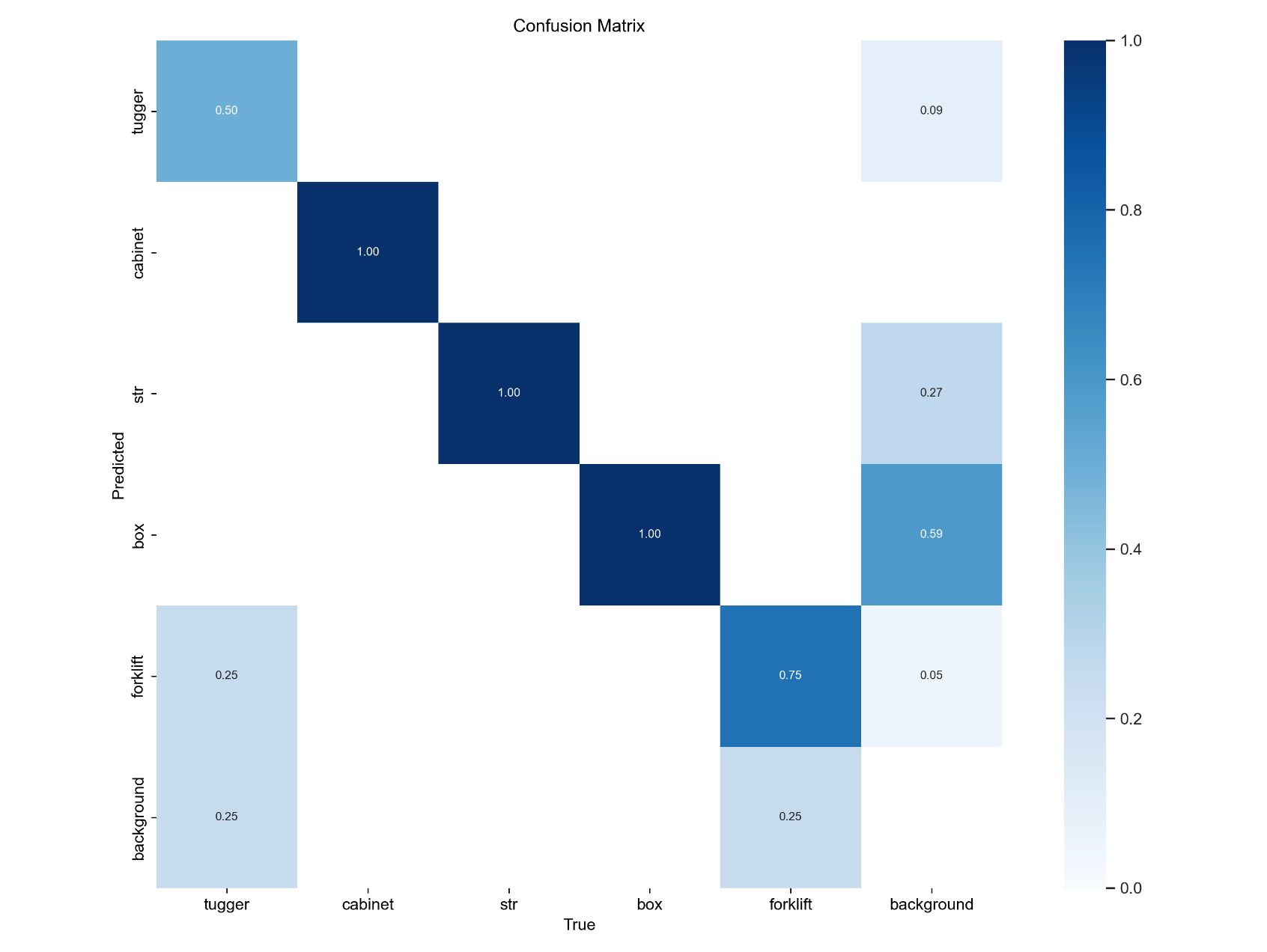
Standard model:

Figure 20: Confusion matrix for the standard model

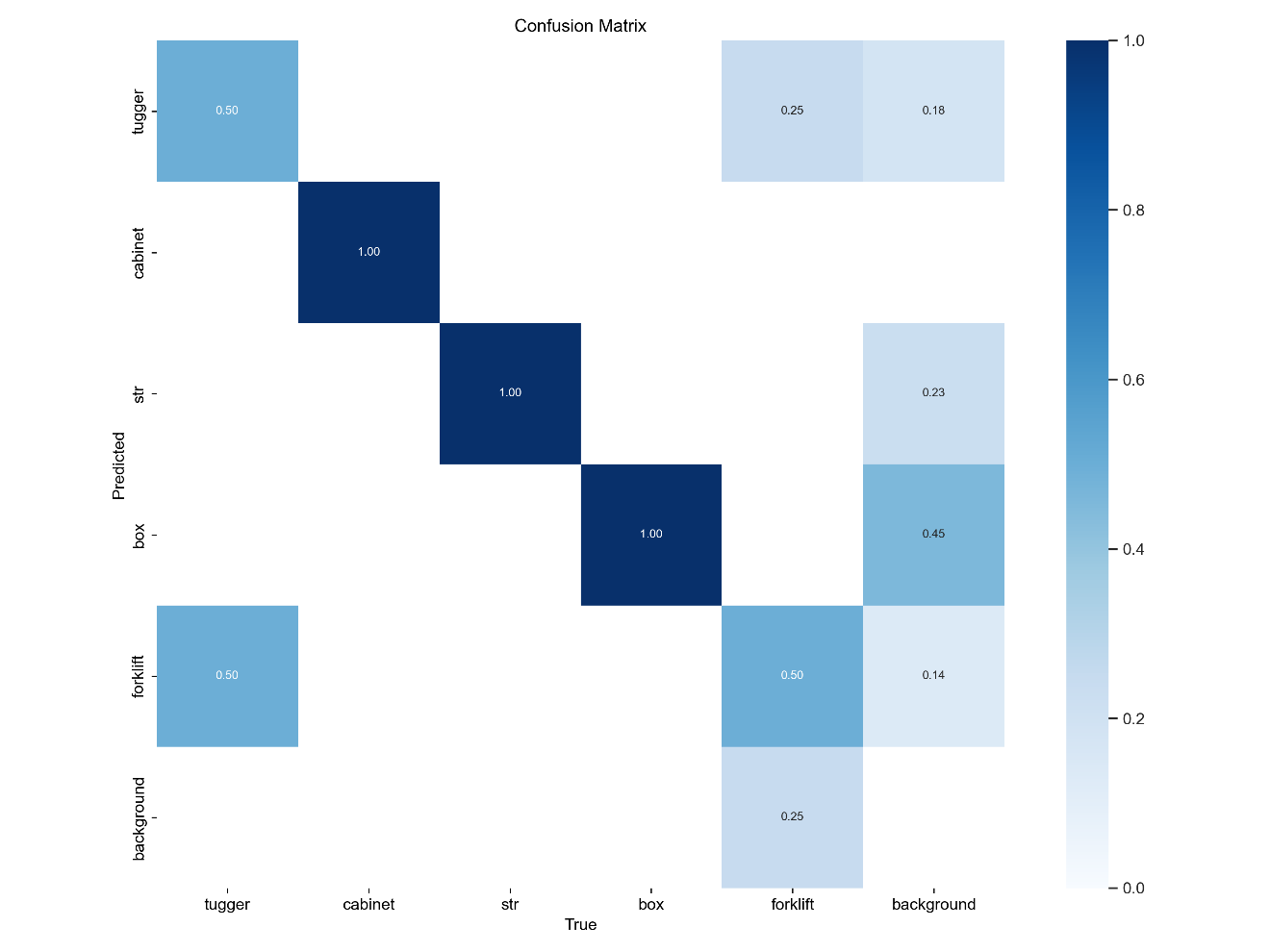
Hyperparameter-tuned model:

Figure 21: Confusion matrix for the hyperparameter-tuned model

# Part 3: Model Deployment & Inference

## Export Models to ONNX Format

To enable efficient inference on various platforms, both trained models were exported to the ONNX (Open Neural Network Exchange) format. This conversion allows the models to be used with a wide range of inference engines.

The export process maintained the input size of 640×640 pixels and used dynamic axes to support variable batch sizes during inference.

## Network Visualization with Netron

The exported ONNX models were visualized using Netron, a web-based neural network visualization tool. This provides insight into the model architecture and helps verify the correct export of the models.

Standard YOLOv5s model architecture:

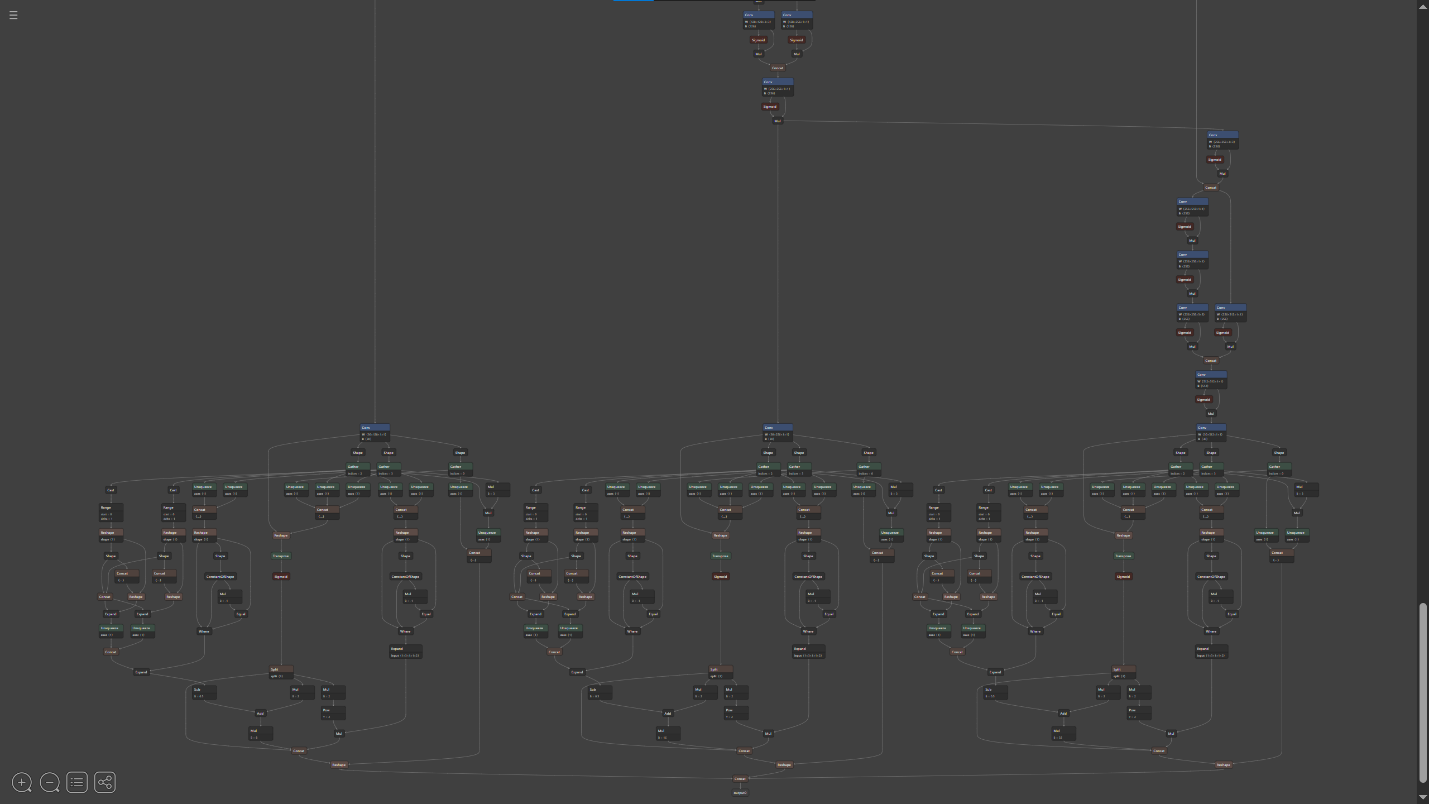


Figure 22: Netron visualization of the standard YOLOv5s model architecture

Hyperparameter-tuned model architecture:

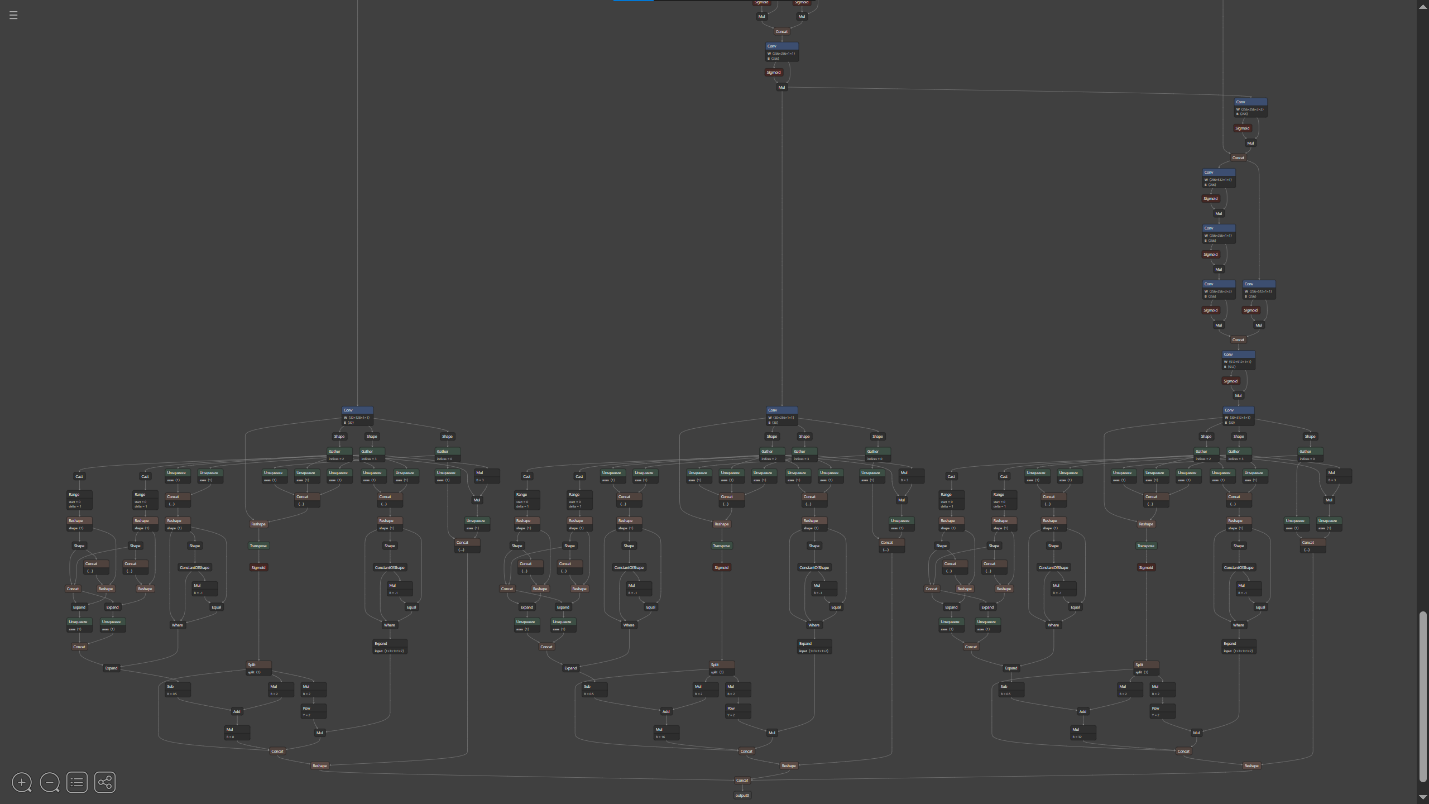


Figure 23: Netron visualization of the hyperparameter-tuned model architecture

## REST API Implementation with FastAPI

A REST API was developed using FastAPI to serve the trained object detection models. The API provides endpoints for real-time inference on uploaded images, returning both JSON output and visualized results.

### API Endpoints

The API offers the following endpoints:

1. **GET /**: Redirects to the API documentation
2. **GET /models**: Lists all available ONNX models
3. **POST /inference**: Accepts an image and returns detected objects as JSON
4. **POST /inference\_image**: Accepts an image and returns the same image with bounding boxes drawn

### Architecture

The API implementation includes:

* Non-Maximum Suppression (NMS) for filtering redundant detections
* Custom logic for processing YOLOv5 output tensors
* Visualization functionality for displaying detection results

### Sample JSON Output

[

{

"Id": 172779,

"ObjectClassName": "cabinet",

"ObjectClassId": 2,

"Left": 398,

"Top": 23,

"Right": 652,

"Bottom": 427,

"x\_center": 0.2734,

"y\_center": 0.2083,

"width": 0.1323,

"height": 0.3741,

"Confidence": 0.8976

}

]

### Testing FastAPI Endpoints

Figure 24: FastAPI docs page showing project endpoints

Models Endpoint:

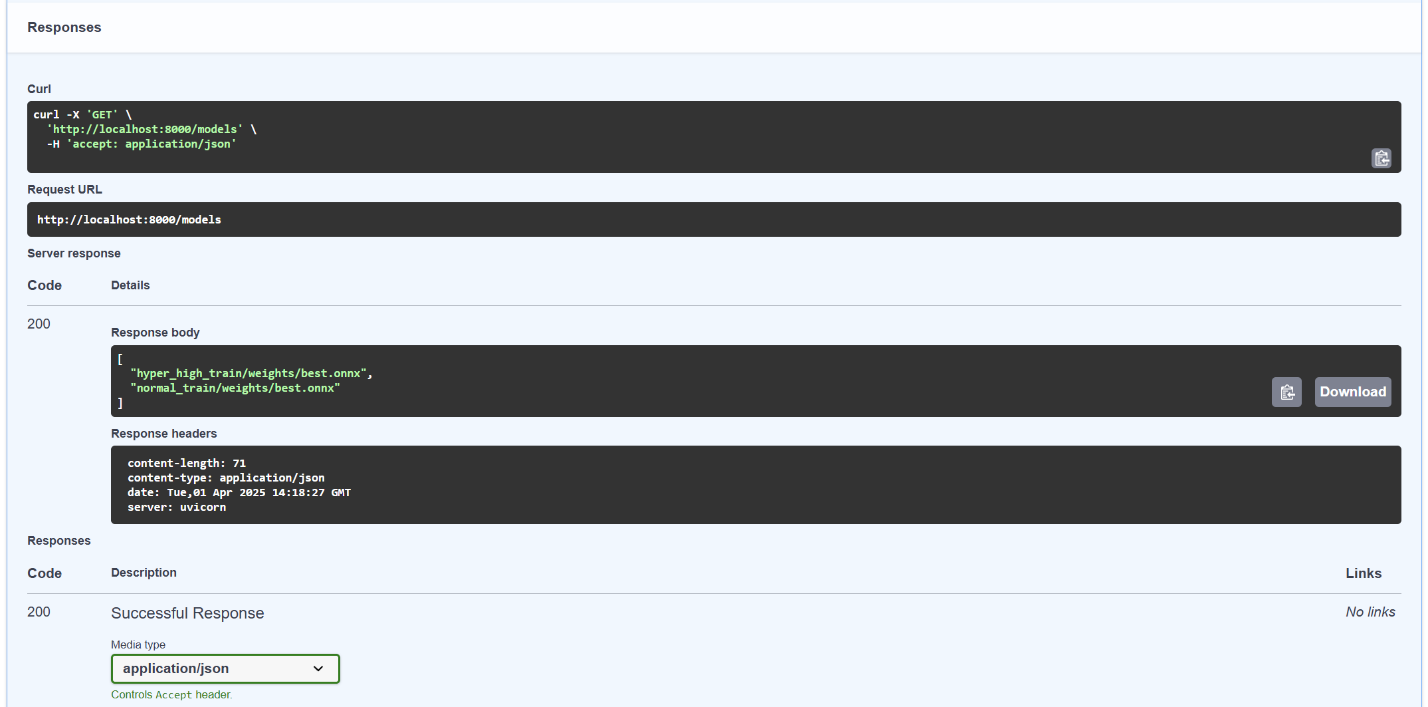


Figure 25:' /models' endpoint returning the available models

Inference Endpoint:

Figure 26: '/Inference' Endpoint showing the json output which includes the bounding boxes

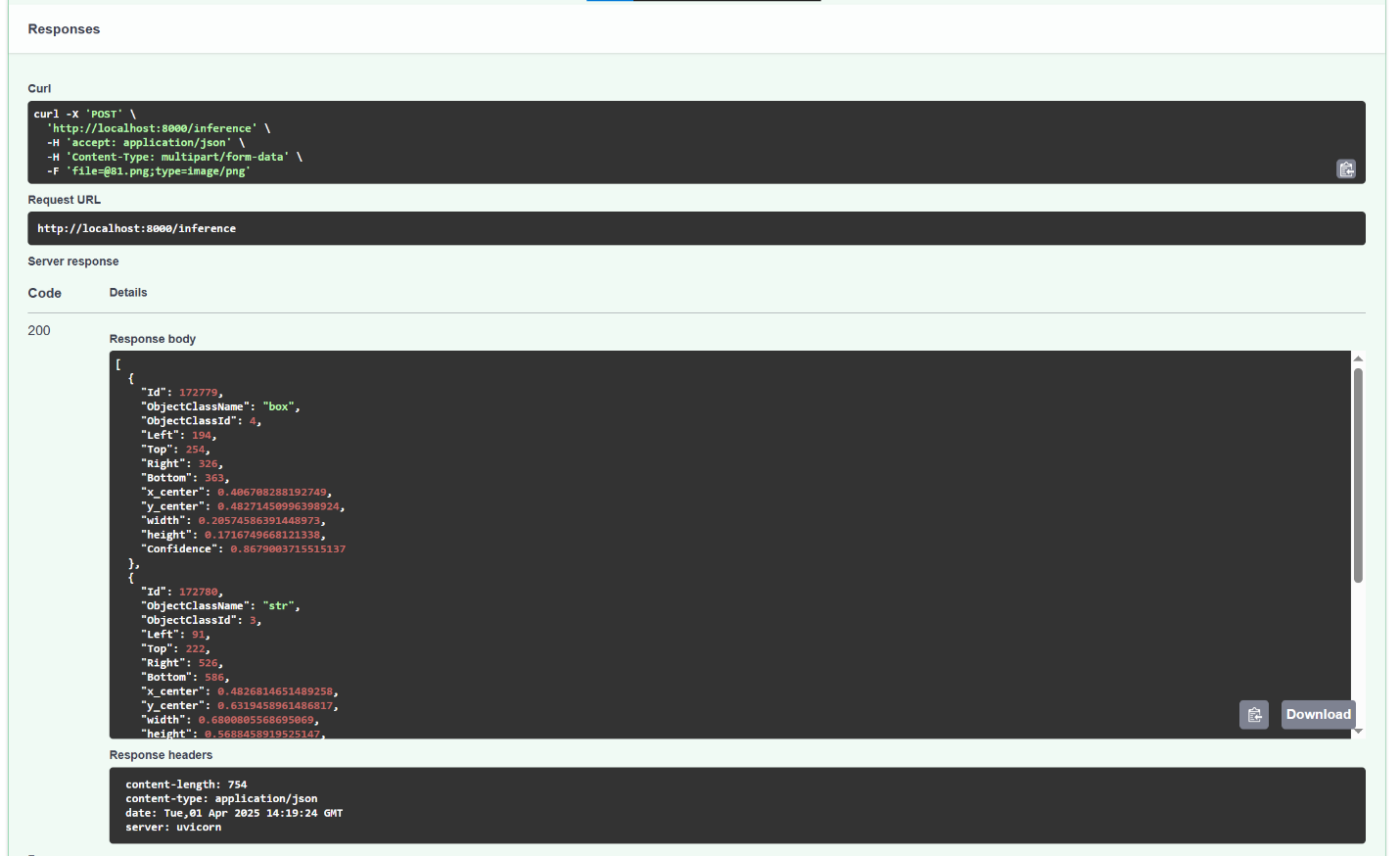


Image Inference Endpoint:

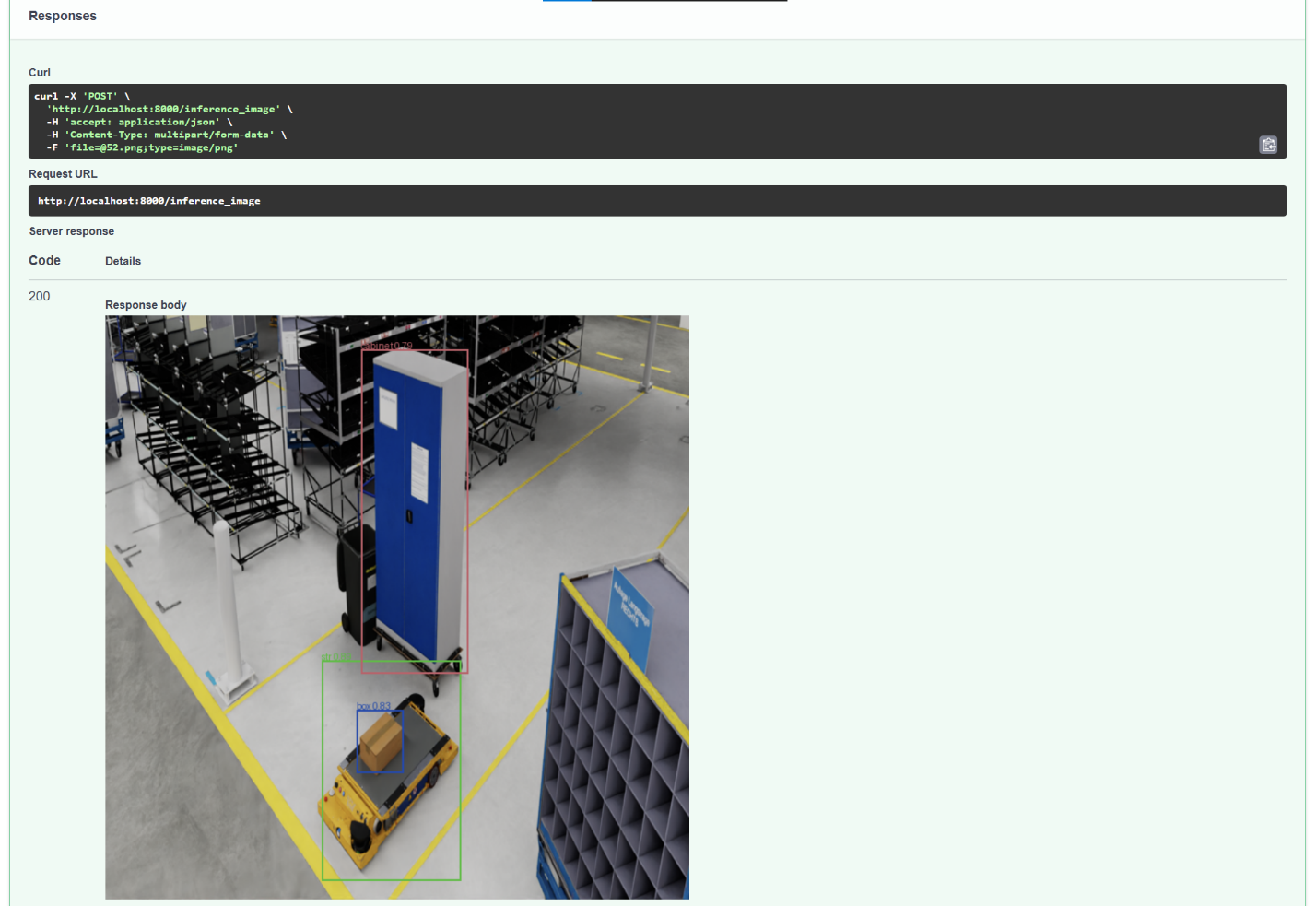


Figure 27: Image inference endpoint showing the image and the returned bounding boxes

## Docker Containerization

The API was containerized using Docker to ensure consistent deployment across different environments. This approach simplifies installation and eliminates potential dependency issues.

### Dockerfile

A Dockerfile was created to specify the Python environment and dependencies required for the API.

### Docker Compose

A docker-compose.yml file was configured to simplify the deployment process, including port mapping and volume mounting for model files.

### Running the Containerized API

The containerized API can be started using Docker Compose, providing an easy-to-deploy solution for inference.

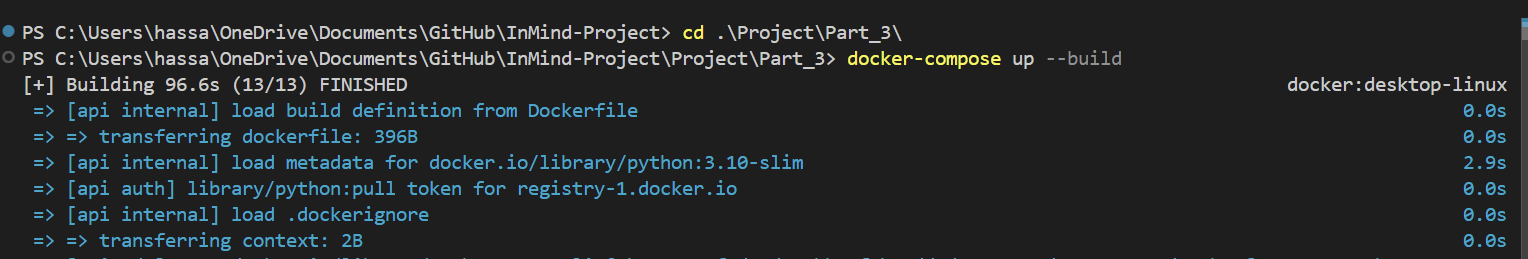
Docker Compose build process initiation:

Figure 28: Docker Compose build process initiation

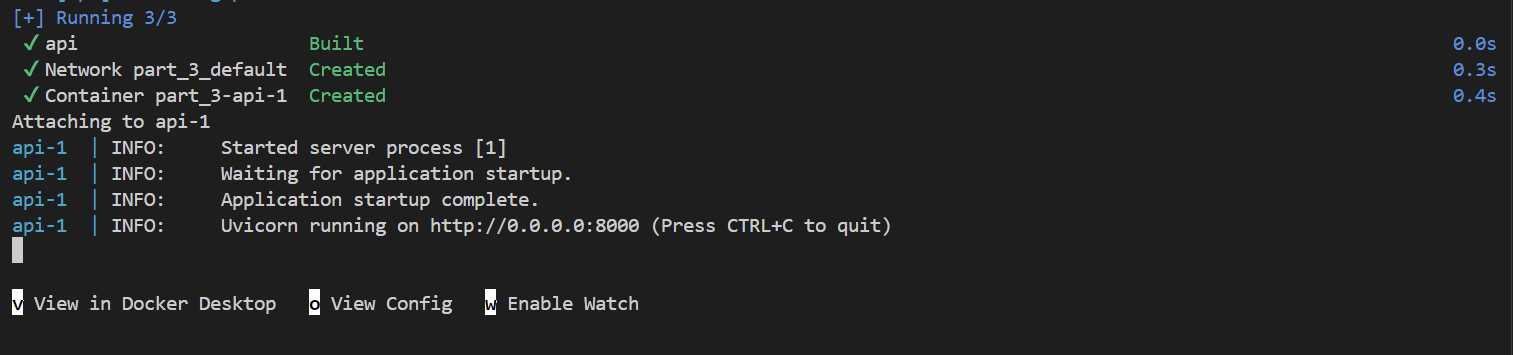
Completion of the Docker container build:

Figure 29: Successful completion of the Docker container build

Container logs in the Docker desktop app:

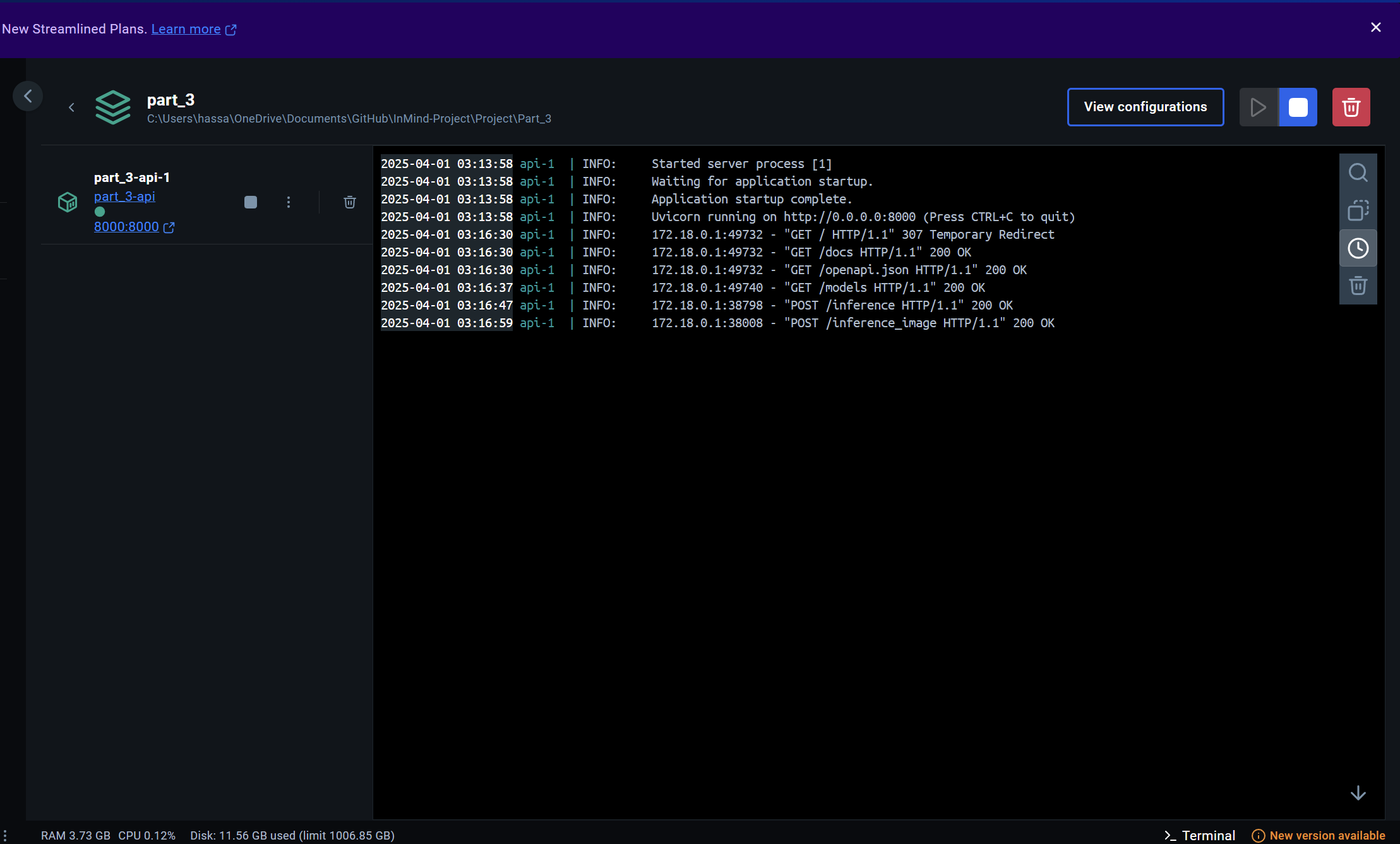


Figure 30: Container logs showing API endpoint requests and responses

# Conclusion

The final object detection Project successfully implemented an object detection system for industrial environments using YOLOv5. The project encompassed the complete machine learning pipeline from data preparation to model deployment.

Performance evaluation showed that while hyperparameter tuning was applied, the improvements were minimal due to the limited size and imbalanced nature of the dataset. Future work could focus on expanding the dataset and exploring more advanced architectures.

The final solution provides a containerized API that can be easily integrated into existing industrial systems for real-time object detection.

# References

1. Ultralytics YOLOv5. (2022). GitHub repository. https://github.com/ultralytics/yolov5
2. ONNX Runtime. (2022). GitHub repository. https://github.com/microsoft/onnxruntime
3. FastAPI. (2022). Documentation. https://fastapi.tiangolo.com/
4. Albumentations. (2022). Documentation. https://albumentations.ai/docs/

# Appendix

## System Requirements

The project dependencies include:

* torch==2.1.2+cu118
* torchvision==0.16.2+cu118
* matplotlib
* albumentations
* numpy
* tensorboard
* onnx
* onnxruntime
* fastapi
* uvicorn
* python-multipart

## Model Configuration Details

The detailed configuration of the YOLOv5 models including layer architecture and hyperparameters can be found in the model summary output and the hyperparameter configuration files.