Context-Based Aerial Object Detection

A project report submitted in partial fulfillment of the requirements for the degree of

Master of Technology

in Computer Science and Engineering

Kuwar Raghvendra Singh
Roll Number: CS23M108
Supervisors: Dr. Chalavadi Vishnu

Department of Computer Science and Engineering
Indian Institute of Technology Tirupati
December 2024

DECLARATION

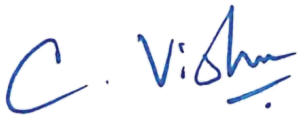
I declare that this written submission represents my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources.

Signature

Kuwar Raghvendra Singh
CS23M108

Bona Fide Certificate

This is to certify that the report titled Context-Based Aerrial Object Detection, submitted by Kuwar Raghvendra Singh, is a bona fide record of the project work done under our supervision.



Place: Tirupati
Date: 04-12-2024

Dr. Chalavadi Vishnu
Assistant Professor
Department of Computer Science and Engineering
IIT Tirupati - 517501

Acknowledgments

Thanks to all those who helped during this thesis and work.

Abstract

This project report investigates the development and implementation of a Context-Based Aerial Object Detection system designed to enhance the accuracy and reliability of detecting objects in aerial imagery. The primary objective is to leverage contextual information to improve detection performance in complex environments, such as urban areas and disaster zones.

Aerial object detection is crucial for various applications, including disaster management, military surveillance, and transportation monitoring. By integrating context—defined as the environment surrounding an object—this project addresses common challenges encountered in traditional detection methods, such as scale imbalance, irregular spatial distribution, and varying lighting conditions.

The proposed approach employs the YOLOv8 algorithm, trained on a custom dataset of high-resolution aerial images annotated with moving objects. The results demonstrate significant improvements in performance metrics, including precision and recall, indicating improved detection capabilities across diverse scenarios.

Future work will focus on addressing existing imbalances within the dataset and exploring additional contextual features to further refine detection accuracy. This research contributes valuable insights into aerial object detection and offers a robust framework for future advancements in surveillance and situational awareness technologies.

Keywords: Aerial Object Detection, Contextual Information, YOLOv8, Machine Learning, Disaster Management.

Contents

Chapter 1

Introduction

1.1 Background

In recent years, the importance of precise and effective object detection from aerial imagery has been greatly promoted because of its many uses, such as urban planning, environmental monitoring, traffic management, emergency response, and combat operations. The availability of air data collected by Unmanned Aerial Vehicles (UAVs) and aircraft. The technology of satellites has advanced significantly, enabling broader and more precise observation. Nevertheless, these images are likely to have distinct features, characteristics like high resolution, large area of coverage, and predominantly tiny-sized objects, This creates special detection issues relative to traditional ground-based imagery.

1.2 Motivation

Classic deep learning architectures like Faster R-CNN, SSD, and YOLO have performed better in general object detection, their use in aerial Imagery has uncovered numerous principal limitations. These models tend to struggle with very small objects are limited by their low resolution and reduced visibility of features. Furthermore, aerial photographs usually show non-uniform object distribution with tightly grouped object clusters in some regions, thus complicating detection. With these shortcomings, there is is a strong incentive to develop methods that are unique to aerial photography, which can effectively address the issue of small object detection and non-uniform object distribution.

1.3 Problem Statement

Despite tremendous advancements, traditional object detection algorithms suffer from a problem. When used on aerial photography, this is largely due to three reasons: (1) High-resolution images with high computational expenses, (2) Small objects with poorly detailed visual information, and (3) An unbalanced distribution of objects, leading to inefficient use of com- Computational resources. To overcome these obstacles, an innovative detection system is needed. skilled in properly identifying and distinguishing highly congested areas with high

detection accuracy for objects of varying sizes.

1.4 Objectives

The overall objectives of this research are:

- To gain a complete insight and evaluate the potential of the YOLOv8 model in recognition of small and closely spaced details in aerial photographs.
- Develop and test a novel context-sensitive aerial object detection system driven by the YOLC algorithm, which is particularly formulated to overcome the limitations of conventional preparations.
- Conduct a comprehensive comparative analysis between YOLOv8 and the suggested pipeline The YOLC-based pipeline demonstrates the effectiveness of the latter in improving. Performance and speed of calculation on the VisDrone dataset.

1.5 Scope and Limitations

The VisDrone dataset, a well-known standard for assessing object detection techniques in aerial scenarios, is specifically used in this study to target the detection of small and clustered objects in aerial images. Although the suggested techniques greatly improve detection capabilities, there are some drawbacks:

- The study's primary focus is on aerial imagery from the VisDrone dataset, which limits its direct applicability to other aerial datasets with distinct features.
- Improvements in computational efficiency are shown under particular hardware limitations and experimental configurations, which might differ in practical applications.
- Future research is still needed to determine how well the suggested approaches translate to other aerial datasets and object categories.

Chapter 2

Literature Review

2.1 Introduction to Object Detection in Aerial Imagery

Object detection in aerial images has emerged as a critical research area, extensively applied in urban planning, environmental monitoring, military reconnaissance, and disaster management. However, detecting objects accurately from aerial images presents unique challenges, including significant variations in object scales, arbitrary object orientations, dense object distributions, and complex backgrounds.

2.2 Datasets for Aerial Object Detection

Well-annotated datasets are foundational to the advancement of deep learning-based object detection algorithms. The DOTA (Dataset of Object Detection in Aerial images) represents one of the most influential benchmarks, containing over 1.8 million annotated objects across 18 categories. This dataset uniquely uses oriented bounding boxes (OBB), significantly improving accuracy in detecting arbitrarily oriented objects compared to traditional horizontal bounding boxes (HBB). DOTA facilitates extensive evaluations and algorithm developments by providing standardized baselines and code libraries, thereby enabling reproducible research and comprehensive algorithm comparisons in aerial imagery analysis.

Another widely used dataset is **VisDrone**, which is specifically curated for object detection tasks using drone-captured aerial imagery. VisDrone consists of thousands of images collected from various urban and rural environments under different weather and lighting conditions. It contains over 10,000 images with annotations for pedestrian, vehicles, bicycles, and other small-scale objects, making it particularly suitable for evaluating the performance of object detection models in real-world UAV scenarios. The dataset presents considerable challenges due to the high object density, scale variations, and frequent occlusions, making it ideal for benchmarking models like YOLOv8 and context-based detection approaches.

2.3 Advances in YOLO-Based Algorithms

The YOLO (You Only Look Once) series represents a revolutionary shift in object detection, characterized by high speed and significant accuracy. YOLO models perform direct predictions of bounding boxes and class probabilities in a single pass through a neural network, substantially reducing computational complexity compared to two-stage detectors such as Faster R-CNN. Recent developments, including YOLOv5, YOLOv8, and YOLOv9, have introduced advancements like anchor-free prediction mechanisms, sophisticated feature pyramid networks, and improved loss functions such as focal loss, significantly boosting their performance in detecting small and densely packed objects.

YOLOv8, in particular, integrates advanced modules such as EfficientNet as its backbone and NAS-FPN (Neural Architecture Search Feature Pyramid Network) for its detection head. This setup achieves state-of-the-art balance between accuracy and efficiency, especially tailored for small object detection in aerial scenarios.

2.4 Context-Based Detection Methods

Recent methodologies emphasize the importance of context in detecting dense, tiny objects in aerial images. The YOLC (You Only Look Clusters) method focuses specifically on identifying clusters of densely packed tiny objects by employing a Local Scale Module (LSM). This innovative approach dynamically prioritizes regions with dense object populations, significantly improving detection efficiency and accuracy by reducing the computational effort spent on sparse regions. Additionally, YOLC utilizes deformable convolutions and Gaussian Wasserstein Distance (GWD) loss to handle the inherent challenges of aerial images more effectively.

Another context-aware approach, DET-YOLO, introduces specialized modules like C2f_DEF and the DAT attention module, designed explicitly to improve detection accuracy against complex and dense backgrounds. These modules optimize the feature extraction process, enhancing model adaptability in intricate remote sensing environments, particularly in military reconnaissance.

2.5 Challenges and Techniques for Small Object Detection

Detecting small objects in aerial images remains notably challenging due to their limited visual information, frequent occlusions, and highly variable scales. Recent studies have approached these challenges through innovative techniques like Density-Aware Scale Adaptation (DASA) and knowledge distillation strategies. DASA effectively isolates densely populated object regions, maintaining consistent scale across these sub-regions, thereby significantly enhancing detection accuracy for small objects. Coupled with grid-masked knowledge distillation, these approaches facilitate improved feature recovery and overall detection performance without compromising computational efficiency.

Furthermore, a scene context-guided approach emphasizes the crucial role of surrounding contextual information. This methodology employs dedicated modules to enhance foreground objects and suppress irrelevant background details, providing a more robust and precise detection performance in scenarios involving densely clustered tiny objects.

2.6 Summary and Research Gap

While significant advancements have been made, several challenges persist, particularly concerning accurately detecting dense and tiny objects with arbitrary orientations in complex aerial images. Existing algorithms like YOLO and its variants, though powerful, still require enhancements in contextual understanding and scale adaptability specific to aerial imagery challenges. Thus, the motivation arises to develop and explore methods integrating context-based clustering and advanced feature extraction strategies, directly addressing the remaining gaps in small-scale, densely distributed object detection in aerial imagery.

This research aims to bridge this gap by implementing a context-based aerial detection pipeline, inspired by the strengths of YOLOv8 and YOLC methodologies, to enhance performance on benchmark datasets such as VisDrone and DOTA.

Chapter 3

Dataset Exploration

3.1 Introduction to VisDrone Dataset

The VisDrone dataset, developed by the AISKYEYE team at the Lab of Machine Learning and Data Mining, Tianjin University, is one of the most comprehensive benchmarks available for aerial object detection and other computer vision tasks. It aims to bridge the gap between drone-acquired imagery and the requirements of state-of-the-art detection algorithms. The dataset consists of a rich collection of high-resolution video clips and static images captured using drone-mounted cameras across 14 diverse cities in China. These images represent various environmental conditions, including urban and rural areas, different weather patterns, and varying times of day. With over 2.6 million annotated bounding boxes across categories like pedestrians, vehicles, and bicycles, it provides detailed object-level information. Furthermore, annotations also include occlusion levels, visibility, and object truncation, enhancing its suitability for complex, real-world scenarios. This diversity and level of detail make VisDrone a valuable resource for training and benchmarking deep learning models in UAV-based applications.

3.2 Dataset Composition and Diversity

The dataset includes a total of 288 video clips composed of 261,908 frames and 10,209 static images. These were collected using various drone-mounted cameras across 14 cities in China, representing a geographically and environmentally diverse set of conditions. The scenes vary across:

- Urban and rural locations
- Crowded and sparse environments
- Different lighting conditions (daylight, dusk)
- Various weather scenarios

Figure 3.1: Sample drone image from VisDrone showing urban environment and dense object crowding.

3.3 Object Categories and Annotations

The dataset includes annotations for more than 2.6 million bounding boxes. These boxes label frequently encountered objects in aerial scenes such as:

- Pedestrians
- Bicycles and Tricycles
- Cars, Vans, and Buses
- Trucks and Motorcycles

In addition to object class, each annotation includes attributes for occlusion level, visibility, and truncation, making the dataset highly valuable for developing robust models under real-world conditions.

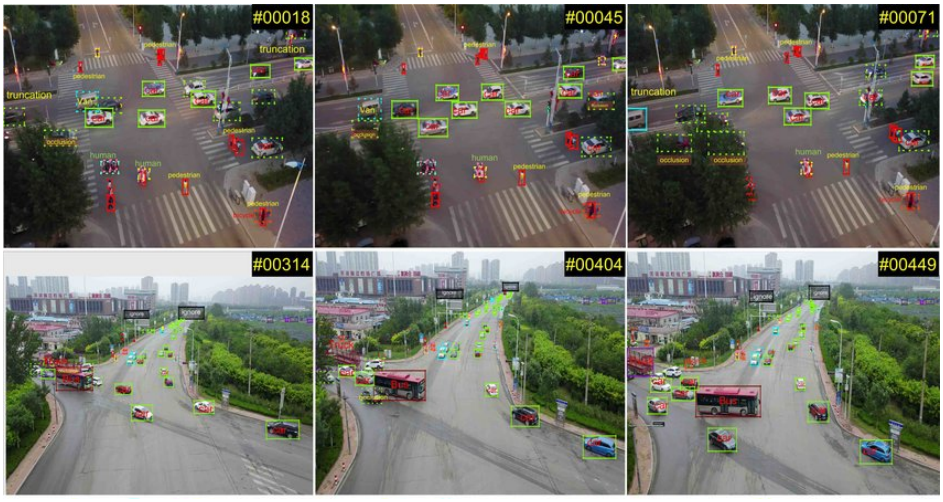


Figure 3.2: Visualization of object annotations in a sample VisDrone frame.

3.4 Challenges Presented by the Dataset

The VisDrone dataset presents several challenges that make it an ideal benchmark for aerial object detection:

- High object density in some frames
- Significant scale variation
- Frequent occlusion and partial visibility
- Mixed backgrounds and varying altitudes

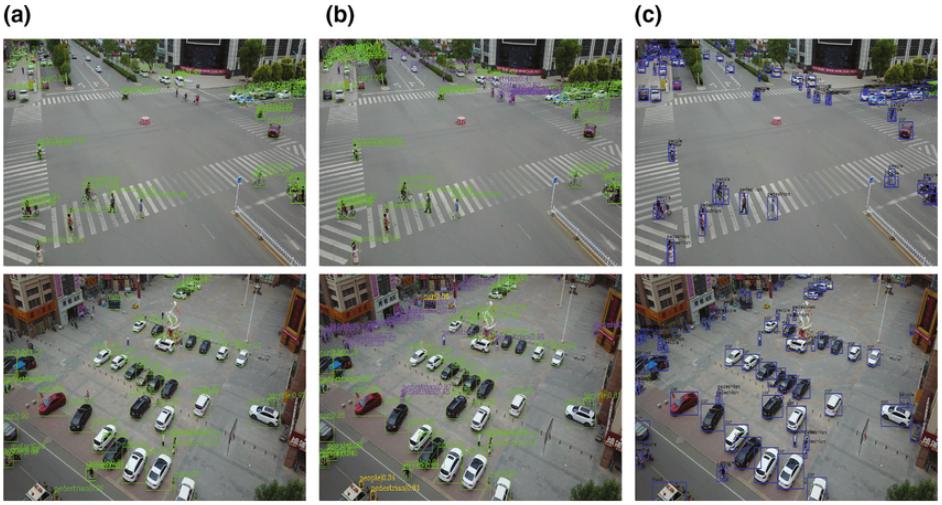


Figure 3.3: High Object Density: Numerous vehicles and pedestrians in close proximity illustrate the complexity of detecting multiple objects in dense urban environments.

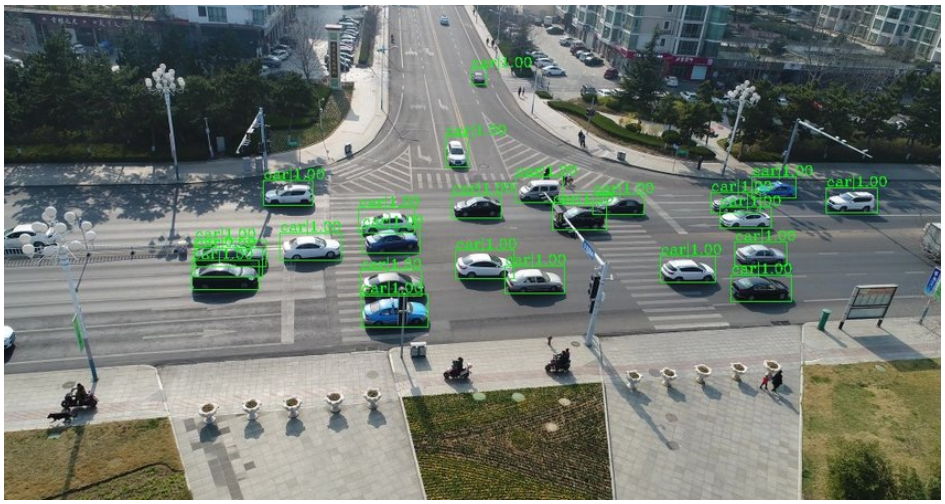


Figure 3.4: Significant Scale Variation: Objects of various sizes, including large trucks and small pedestrians, emphasize the need for scale-invariant detection.

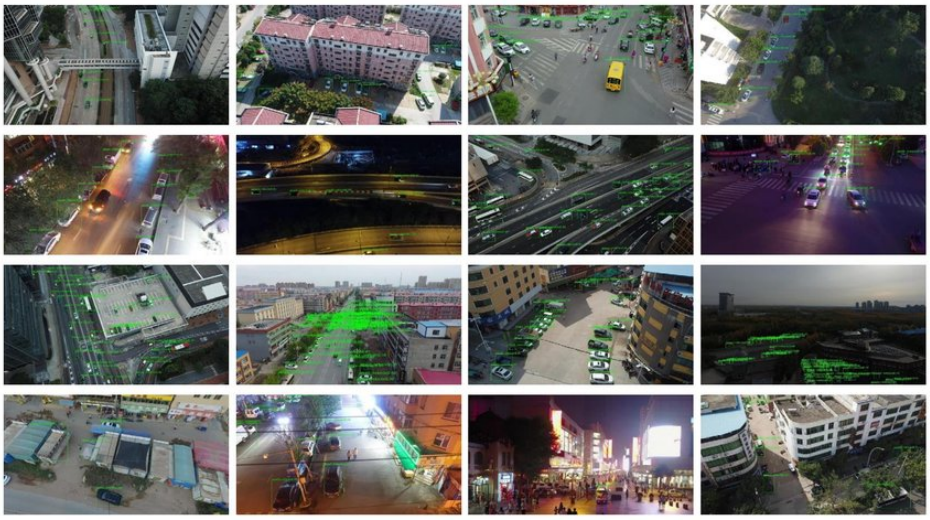


Figure 3.5: Frequent Occlusion and Partial Visibility: Vehicles are partially obstructed by others or environmental elements, creating detection difficulties.

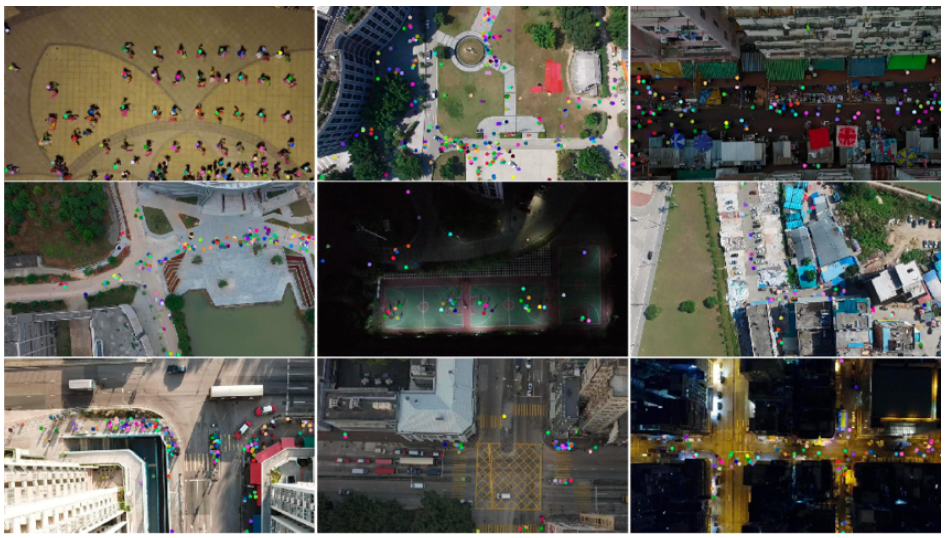


Figure 3.6: Mixed Backgrounds and Varying Altitudes: Complex backgrounds and drone-captured perspectives from varying heights present unique detection challenges.

3.5 Preprocessing and Splitting Strategy

For our experiments, the dataset is split into training, validation, and testing subsets. The original dataset provides a standard split, which has been adopted in our methodology to ensure consistency and fair comparison with other models trained on VisDrone.

Chapter 4

Methodology

4.1 Overview of the Proposed Methodology

This chapter outlines the context-based aerial object detection methodology, which is based on a baseline YOLOv8 model and an enhanced YOLC-inspired detection pipeline. Aerial imagery presents unique challenges because target objects appear very small, images are very large (millions of pixels), and objects tend to cluster unevenly in specific areas. These factors make it difficult for standard detectors to reach high accuracy without wasting computation on empty regions.

To address the challenges of aerial object detection, especially for small, densely clustered objects, we compare two state-of-the-art detection models: YOLOv8 and YOLC (You Only Look Clusters). YOLOv8 is a fast, accurate, single-stage detector known for its real-time performance, whereas YOLC offers a context-aware, two-stage pipeline designed specifically for dense and small object detection in aerial imagery. Both models are trained and evaluated using the VisDrone UAV dataset to guarantee an equitable and consistent comparison in real-world aerial scenes. Using the same experimental setup, we can investigate how each model reacts to changes in scale, object density, and localization accuracy.

4.2 Baseline Model: YOLOv8

4.2.1 Architecture Overview

YOLOv8 is the latest evolution of the You Only Look Once family of single-stage object detectors. Its architecture comprises three main components:

- **Backbone:** A deep convolutional network (e.g., EfficientNet-B4) that extracts multi-scale feature maps from the input image.
- **Neck:** A NAS-FPN module that fuses features across scales via learned top-down and bottom-up pathways, producing enriched feature maps P3' to P7'.
- **Detection Head:** Convolutional layers with predefined anchors that predict bounding boxes, objectness scores, and class probabilities on each fused feature map.

This unified design enables real-time inference while maintaining strong accuracy across object sizes.

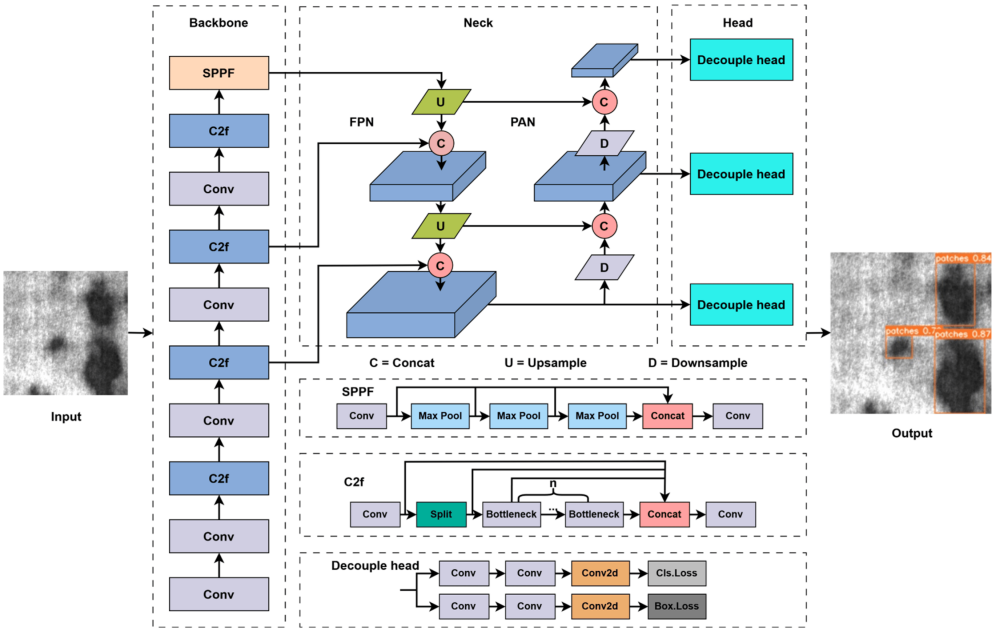


Figure 4.1: YOLOv8 model architecture. The input image is fed into the backbone to produce multi-scale feature maps.

4.2.2 Training Strategy

We fine-tune YOLOv8 on the VisDrone dataset using the following configuration:

- **Model:** YOLOv8n pre-trained weights (`"yolov8n.pt"`) were used for training.
- **Training dataset:** VisDrone dataset, configured via `visdrone.yaml`.
- **Input image size:** 640 × 640 pixels.
- **Epochs:** 100 training epochs.
- **Batch size:** 16 images per batch.
- **Device:** Trained using GPU (device ID 0).
- **Optimizer:** Stochastic Gradient Descent (SGD) with momentum of 0.9 and weight decay of 1e-4.
- **Learning rate schedule:** Initial learning rate of 0.01 with a 3-epoch linear warm-up, followed by cosine decay (Note: the current training ran for 100 epochs, not the full 160 planned in the scheduler).
- **Loss functions:** Focal Loss for classification and Complete IoU (CIoU) Loss for bounding box regression.

This setup yields a robust baseline, which we denote as `YOLOv8_VisDrone`.

4.2.3 Pipeline and Workflow

The YOLOv8 inference pipeline follows a single forward pass:

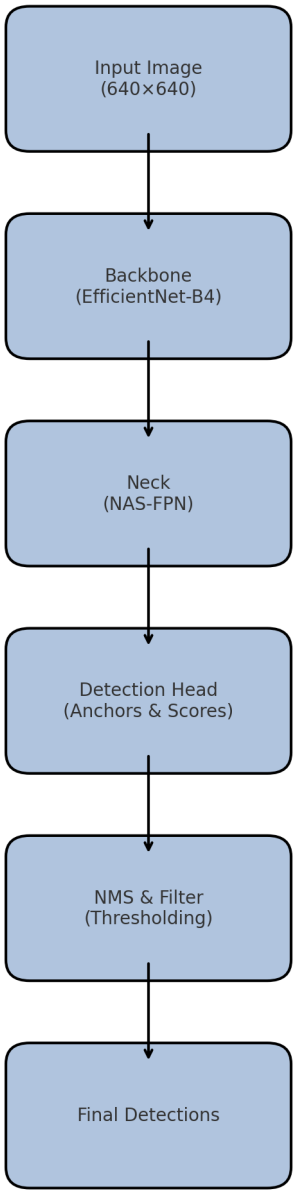


Figure 4.2: Block diagram of YOLOv8. The baseline YOLOv8 processes the input image in a single stage, producing detections directly.

4.3 Model: YOLC-Inspired Contextual Pipeline

4.3.1 YOLC Paper Overview

You Only Look Clusters (YOLC) is a recently proposed framework created especially to handle the particular difficulties associated with detecting small objects in large aerial photos. YOLC takes a more effective and context-aware approach than traditional object detectors, which frequently waste processing power on empty or irrelevant regions. YOLC is based on a novel two-stage detection strategy that first conducts a global search to find areas where objects are densely clustered, rather than processing every region of an image equally. To improve the outcomes, these areas are subsequently put through a local, high-resolution detection phase. The model can concentrate its computational efforts on the most pertinent areas of the image thanks to this "global-to-local" approach. This fundamental idea is reflected in the name You Only Look Clusters.

YOLC is built upon *CenterNet*, an anchor-free object detection framework, and introduces several key innovations:

- **Local Scale Module (LSM):** An unsupervised module that dynamically locates object-dense (clustered) regions and crops them for focused high-resolution detection.
- **Gaussian Wasserstein Distance (GWD):** A refined bounding box regression loss that ensures more accurate and tighter localization, particularly beneficial for small object detection.
- **Deformable Convolutions and Prediction Refinement Module:** Enhancements in the detection head that allow for better modeling of small and complex object features.

The YOLC framework has demonstrated state-of-the-art performance on challenging aerial image datasets such as **VisDrone2019** and **UAVDT**, particularly excelling in scenarios involving small and densely packed objects.

4.3.2 Inspiration and Adaptation

The faithful application of the YOLC detection framework, as outlined in the original research, is the main goal of this project. Replicating the entire pipeline and assessing how well it detects small-scale objects in aerial imagery are the objectives. A YOLOv8 model is also independently trained using the VisDrone dataset under the same experimental conditions in order to create a standard for comparison. A controlled and equitable comparison of the two models is made possible by this configuration.

It is crucial to remember that neither the integration of YOLC concepts into the YOLOv8 pipeline nor any architectural changes to YOLOv8 are part of this study. Rather, both models are regarded as separate, stand-alone implementations that are assessed using the same performance metrics and dataset.

This methodological setup allows for a **systematic and unbiased comparison** of YOLC and YOLOv8, focusing on their individual strengths, limitations, and overall effectiveness in real-world aerial object detection tasks. The study aims to highlight which

approach is better suited for addressing the complexities of detecting tiny, densely clustered objects in high-resolution aerial scenes.

4.3.3 YOLC Architecture Overview

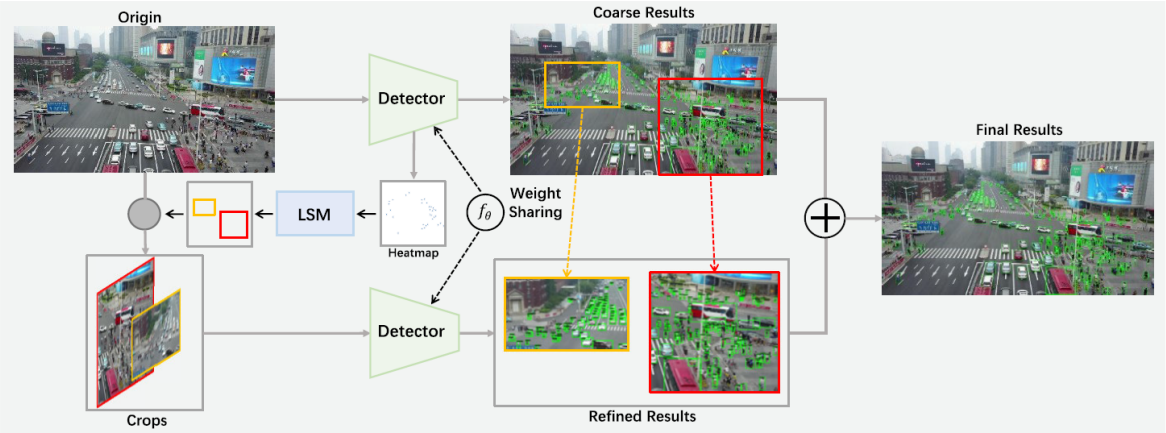


Figure 4.3: Overview of the YOLC architecture. The two-stage detection pipeline consists of a global detection pass to locate clustered regions, followed by local high-resolution refinement.

4.3.4 Pipeline and Workflow

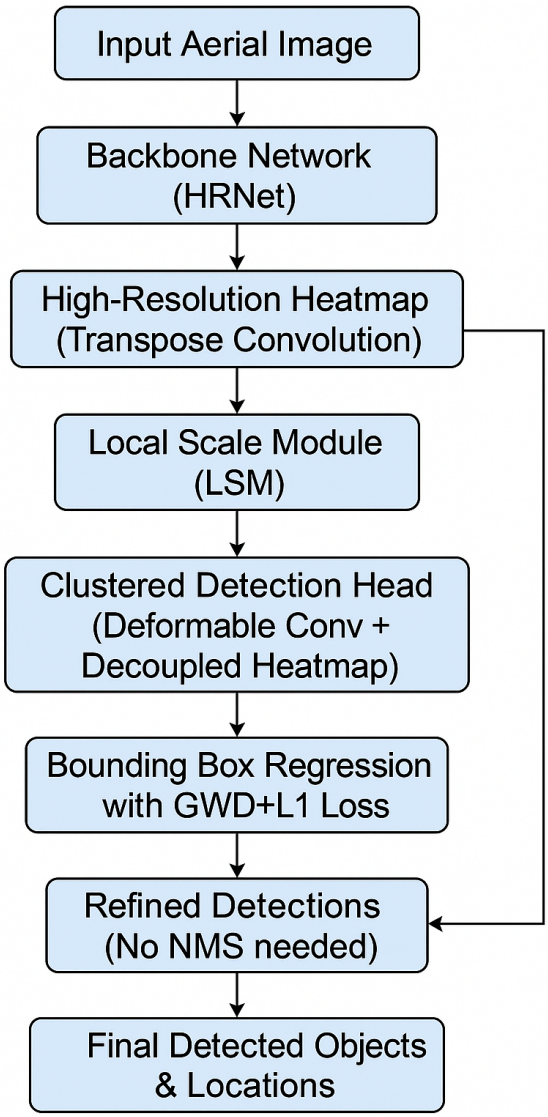


Figure 4.4: YOLC-inspired two-stage detection pipeline using YOLOv8. Stage 1 performs global detection. A clustering module identifies dense regions. Stage 2 zooms into each region for refined detection, followed by merging and NMS.

4.4 Mathematical Formulation of the YOLC Loss

Our detector follows *CenterNet*'s key-point paradigm and is trained with three principle losses:

$$\mathcal{L}_{\text{det}} = \underbrace{\mathcal{L}_{\text{k}}}_{\text{focal heat-map}} + \lambda_{\text{off}} \underbrace{\mathcal{L}_{\text{off}}}_{\text{center offset}} + \lambda_{\text{size}} \underbrace{\mathcal{L}_{\text{size}}}_{\text{size reg.}}, \tag{4.1}$$

where λ_{off}=1 and λ_{size}=0.1 are the default weights . YOLC replaces the *size regression* term with a Gaussian Wasserstein Distance (GWD) loss that is better aligned with IoU and more tolerant of small-object noise.

4.4.1 Bounding Boxes as 2-D Gaussians

A predicted box $B(x, y, w, h)$ is mapped to a 2-D Gaussian

$$f(\mathbf{x} \mid \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \frac{\exp\!\big(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^{\top}\boldsymbol{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\big)}{2\pi\sqrt{|\boldsymbol{\Sigma}|}}, \qquad \boldsymbol{\mu} = (x, y)^{\top},\ \boldsymbol{\Sigma} = \begin{pmatrix} w^2/4 & 0 \\ 0 & h^2/4 \end{pmatrix}. \quad (4.2)$$

4.4.2 Closed-Form Wasserstein Distance

For two Gaussians $\mathcal{N}(\boldsymbol{\mu}_1, \boldsymbol{\Sigma}_1)$ and $\mathcal{N}(\boldsymbol{\mu}_2, \boldsymbol{\Sigma}_2)$ the 2-Wasserstein distance is

$$W^2 = \|\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2\|_2^2 + \mathrm{Tr}\Big(\boldsymbol{\Sigma}_1 + \boldsymbol{\Sigma}_2 - 2\big(\boldsymbol{\Sigma}_1^{1/2}\,\boldsymbol{\Sigma}_2\,\boldsymbol{\Sigma}_1^{1/2}\big)^{1/2}\Big). \quad (4.3)$$

Because our ground-truth boxes are horizontal, $\boldsymbol{\Sigma}_1\boldsymbol{\Sigma}_2 = \boldsymbol{\Sigma}_2\boldsymbol{\Sigma}_1$, which simplifies (4.3) to the Frobenius form

$$W^2 = (x_1 - x_2)^2 + (y_1 - y_2)^2 + \frac{(w_1 - w_2)^2 + (h_1 - h_2)^2}{4}. \quad (4.4)$$

4.4.3 Smooth GWD Loss

Raw W^2 can dominate the gradient for large boxes, so we follow and pass it through a log-affinity transform:

$$\mathcal{L}_{\text{gwd}} = 1 - \frac{1}{\tau + \ln(1 + W^2)}, \qquad \tau \geq 1 \text{ (we set } \tau = 1). \quad (4.5)$$

The gradient is

$$\nabla_W \mathcal{L}_{\text{gwd}} = \frac{2\,W}{\big(1 + W^2\big)\big(1 + \ln(1 + W^2)\big)^2}, \quad (4.6)$$

which *attenuates* for very large W and naturally focuses learning on small objects.

4.4.4 Combined Regression Objective

Large objects can still suffer from vanishing gradients in the early epochs. Therefore YOLC mixes GWD with a modest L1 term:

$$\boxed{\mathcal{L}_{\text{size}} = \lambda_{\text{gwd}}\mathcal{L}_{\text{gwd}} + \lambda_1\,\|\hat{\mathbf{s}} - \mathbf{s}\|_1},\ \lambda_{\text{gwd}}=2,\ \lambda_1=0.5 \quad (4.7)$$

where $\hat{\mathbf{s}} = (\hat{w}, \hat{h})$ and $\mathbf{s} = (w, h)$ are the predicted and ground-truth sizes.

Chapter 5

Implementation

The implementation details of the suggested YOLC (You Only Look Clusters) framework for detecting small objects in aerial photos are covered in this chapter. The implementation is easily extensible, reproducible, and clearly constructed. It has a well-organized structure with distinct folders for model definitions, training/inference procedures, and supporting tools.

5.1 Project File Structure

The complete implementation is organized into three major directories: `models`, `scripts`, and `utils`. Each directory contains logically grouped files corresponding to different aspects of the pipeline.

Directory: `models/`

This directory includes the core model components forming the foundation of the YOLC framework.

File Name	Description
`backbone.py`	Defines the HRNet backbone network to produce high-resolution feature maps, critical for detecting small-scale objects.
`head.py`	Implements the detection head, enhanced with deformable convolutions and decoupled heatmap branches for accurate localization.
`lsm.py`	Encapsulates the Local Scale Module (LSM), responsible for adaptive clustering and region-of-interest (ROI) generation.
`yolc.py`	Integrates the backbone, detection head, and LSM into a unified YOLC model pipeline for end-to-end object detection.

Table 5.1: Files in the `models/` directory

Directory: `scripts/`

This folder houses executable scripts for training, testing, inference, and dataset handling. Each script is designed to run independently while integrating seamlessly with the rest of the system.

File Name	Description
`train.py`	Trains the YOLC model on aerial datasets, incorporating modules such as LSM and custom loss functions.
`test.py`	Evaluates model accuracy and computes performance metrics such as AP, AP50, and AP75.
`run_inference.py`	Loads trained weights and performs inference on unseen images, saving visual and raw output.
`debug_model.py`	Helps validate individual modules like the LSM and detection head during development.
`download_visdrone.py`	Downloads and organizes the VisDrone dataset into the required format for training.

Table 5.2: Files in the `scripts/` directory

Directory: `utils/`

Utility scripts that assist with dataset management and custom loss implementation are included in this folder.

File Name	Description
`dataset.py`	Loads and processes aerial images, including label parsing, preprocessing, and grid-based patch generation.
`loss.py`	Implements the custom loss functions such as Gaussian Wasserstein Distance (GWD) combined with L1 loss for robust size regression.

Table 5.3: Files in the `utils/` directory

This modular file structure ensures clean separation of responsibilities and promotes scalability and maintainability for future developments in aerial object detection.

5.2 Model Architecture Implementation

This section provides the source code implementations of the core YOLC model components as defined in the `models/` directory.

File: `backbone.py`

backbone.py

```
import torch
import torch.nn as nn

class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, in_channels, out_channels, stride=1):
        super().__init__()
        # Use standard PyTorch modules instead of mmcv
        self.conv1 = nn.Conv2d(
            in_channels,
            out_channels,
            kernel_size=3,
            stride=stride,
            padding=1,
            bias=False
        )
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)
        self.conv2 = nn.Conv2d(
            out_channels,
            out_channels,
            kernel_size=3,
            stride=1,
            padding=1,
            bias=False
        )
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = None
        if stride != 1 or in_channels != out_channels:
            self.downsample = nn.Sequential(
                nn.Conv2d(
                    in_channels,
                    out_channels,
                    kernel_size=1,
                    stride=stride,
                    bias=False
                ),
                nn.BatchNorm2d(out_channels)
            )
```

File: backbone.py

backbone.py

```
def forward(self, x):
        identity = x

        out = self.conv1(x)
        out = self.bn1(out)
        out = self.relu(out)

        out = self.conv2(out)
        out = self.bn2(out)
    if self.downsample:
            identity = self.downsample(x)

        out += identity
        out = self.relu(out)
        return out
class HRNet(nn.Module):
    def __init__(self, config='default'):
        super().__init__()
        # Initial stem
        self.stem = nn.Sequential(
            nn.Conv2d(
                in_channels=3,
                out_channels=64,
                kernel_size=3,
                stride=2,
                padding=1,
                bias=False
            ),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True),
            nn.Conv2d(
                in_channels=64,
                out_channels=64,
                kernel_size=3,
                stride=2,
                padding=1,
                bias=False
            ),
            nn.BatchNorm2d(64),
            nn.ReLU(inplace=True)
        )

        # Stage configurations (base channel=64)
        self.stage1 = self._make_stage(64, 64, num_blocks=4)
        self.stage2 = self._make_stage(64, 128, num_blocks=4)
        self.stage3 = self._make_stage(128, 256, num_blocks=4)
        self.stage4 = self._make_stage(256, 256, num_blocks=4)
```

File: `backbone.py`

backbone.py

```
 # Final feature refinement
        self.final_conv = nn.Conv2d(
            in_channels=256,
            out_channels=256,
            kernel_size=3,
            padding=1,
            bias=False
        )
        self.final_bn = nn.BatchNorm2d(256)
        self.final_relu = nn.ReLU(inplace=True)
def _make_stage(self, in_channels, out_channels, num_blocks):
        layers = []
        layers.append(BasicBlock(in_channels, out_channels,
           stride=1))
        for _ in range(1, num_blocks):
            layers.append(BasicBlock(out_channels, out_channels
               , stride=1))
        return nn.Sequential(*layers)

    def forward(self, x):
        # Input shape: (B, 3, H, W)
        x = self.stem(x)        # (B, 64, H/4, W/4)
        x = self.stage1(x)     # (B, 64, H/4, W/4)
        x = self.stage2(x)     # (B, 128, H/4, W/4)
        x = self.stage3(x)     # (B, 256, H/4, W/4)
        x = self.stage4(x)     # (B, 256, H/4, W/4)

        # Final refinement
        x = self.final_conv(x)
        x = self.final_bn(x)
        x = self.final_relu(x)
        return x  # (B, 256, H/4, W/4)
```

File: head.py

head.py

```
# head.py
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np

# Try to import DeformConv2d if available, otherwise provide a
   fallback
try:
    from mmcv.ops import DeformConv2d
    DEFORMABLE_AVAILABLE = True
except ImportError:
    print("Warning: mmcv.ops.DeformConv2d not available. Using
       fallback implementation.")
    DEFORMABLE_AVAILABLE = False
    # Simple fallback implementation that mimics DeformConv2d
       interface but uses regular convolution
    class FallbackDeformConv2d(nn.Module):
        def __init__(self, in_channels, out_channels,
           kernel_size, padding=0, **kwargs):
            super().__init__()
            self.conv = nn.Conv2d(in_channels, out_channels,
               kernel_size=kernel_size, padding=padding)

        def forward(self, x, offset=None):
            # Ignore offset, just do regular convolution
            return self.conv(x)

    # Replace DeformConv2d with the fallback version
    DeformConv2d = FallbackDeformConv2d
```

File: `head.py`

head.py

```
class YOLCHead(nn.Module):
    def __init__(self, num_classes=10, in_channels=256,
       debug_mode=False):
        """
        YOLC Detection Head with deformable convolution and
           decoupled heatmap branches

        Args:
            num_classes: number of object categories
            in_channels: feature map channels from backbone
            debug_mode: enable debug prints and checks
        """
        super().__init__()
        self.num_classes = num_classes
        self.debug_mode = debug_mode

        # Reduce feature map channels for memory efficiency
        reduced_channels = max(64, in_channels // 4)

        # Deformable convolution for regression branch
        # First regular conv then deformable
        self.reg_init = nn.Sequential(
            nn.Conv2d(in_channels, reduced_channels,
               kernel_size=3, padding=1),
            nn.BatchNorm2d(reduced_channels),
            nn.ReLU(inplace=True)
        )

        # Deformable convolution part
        self.offset_conv = nn.Conv2d(reduced_channels, 18,
           kernel_size=1, padding=0)  # 2*3*3 offsets
        self.dcn = DeformConv2d(
            in_channels=reduced_channels,
            out_channels=reduced_channels,
            kernel_size=3,
            padding=1
        )
```

File: `head.py`

head.py

```
        # Regression (fine) after DCN for [dx, dy, w, h]
         self.reg_final = nn.Conv2d(reduced_channels, 4,
            kernel_size=1)

         # Initial regression output (coarse)
         self.reg_coarse = nn.Conv2d(in_channels, 4, kernel_size
            =1)

         # Decoupled heatmap branches for each class
         self.heatmap_shared = nn.Sequential(
             nn.Conv2d(in_channels, reduced_channels,
                kernel_size=3, padding=1),
             nn.BatchNorm2d(reduced_channels),
             nn.ReLU(inplace=True)
         )

         # More memory-efficient upsampling - only one layer and
             use interpolation for second step
         self.upsample = nn.Sequential(
             nn.ConvTranspose2d(reduced_channels,
                reduced_channels, 4, stride=2, padding=1),
             nn.BatchNorm2d(reduced_channels),
             nn.ReLU(inplace=True)
         )

         # Create class-specific heatmap branches (group
            convolution)
         self.heatmap_heads = nn.Conv2d(
             reduced_channels, num_classes,
             kernel_size=1,
             groups=1  # Default is 1 but can be changed to
                num_classes for true decoupling
         )

         self._init_weights()
```

File: head.py

head.py

```
def _init_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d) or (
               DEFORMABLE_AVAILABLE and isinstance(m,
               DeformConv2d)):
                nn.init.normal_(m.weight, std=0.001)
                if hasattr(m, 'bias') and m.bias is not None:
                    nn.init.constant_(m.bias, 0)

    def forward(self, x):
        """
        Input: features (B, in_channels, H, W)
        Returns:
            heatmaps: high-resolution heatmaps for each class
            reg_init: initial bounding box regression
            reg_refine: refined bounding box regression
        """
        if self.debug_mode:
            print(f"[DEBUG] Input features shape: {x.shape}")

        # High-resolution heatmaps
        heatmap_feat = self.heatmap_shared(x)
        if self.debug_mode:
            print(f"[DEBUG] Heatmap features after shared
               layers: {heatmap_feat.shape}")

        # First upsampling with transposed conv
        heatmap_feat = self.upsample(heatmap_feat)
        if self.debug_mode:
            print(f"[DEBUG] Heatmap features after first
               upsample: {heatmap_feat.shape}")

        # Second upsampling with interpolation (more memory
           efficient)
        heatmap_feat = F.interpolate(heatmap_feat, scale_factor
           =2, mode='bilinear', align_corners=False)
        if self.debug_mode:
            print(f"[DEBUG] Heatmap features after second
               upsample: {heatmap_feat.shape}")

        # Generate heatmaps
        heatmaps = self.heatmap_heads(heatmap_feat).sigmoid()
        if self.debug_mode:
            print(f"[DEBUG] Final heatmaps shape: {heatmaps.
               shape}")
            print(f"[DEBUG] Heatmap min/max/mean: {heatmaps.min
               ().item():.4f}/{heatmaps.max().item():.4f}/{
               heatmaps.mean().item():.4f}")

        # Initial (coarse) regression
        reg_init = self.reg_coarse(x)
```

File: `head.py`

head.py

```
        # Refinement with DCN
        reg_feat = self.reg_init(x)
        if DEFORMABLE_AVAILABLE:
            offsets = self.offset_conv(reg_feat)
            reg_feat = self.dcn(reg_feat, offsets)
        else:
            # If DeformConv2d isn't available, just use regular
                convolution
            reg_feat = self.dcn(reg_feat)

        reg_refine = self.reg_final(reg_feat)

        # Ensure regression values are sensible (positive width
           /height)
        if self.debug_mode:
            print(f"[DEBUG] reg_init shape: {reg_init.shape},
                min/max/mean: {reg_init.min().item():.4f}/{
                reg_init.max().item():.4f}/{reg_init.mean().item
                ():.4f}")
            print(f"[DEBUG] reg_refine shape: {reg_refine.shape
                }, min/max/mean: {reg_refine.min().item():.4f}/{
                reg_refine.max().item():.4f}/{reg_refine.mean().
                item():.4f}")

            # Check if width and height are positive
            w_refine = reg_refine[:, 2:3]  # Width channel
            h_refine = reg_refine[:, 3:4]  # Height channel
            neg_w = (w_refine <= 0).float().sum().item()
            neg_h = (h_refine <= 0).float().sum().item()
            if neg_w > 0 or neg_h:
                print(f"[WARNING] Found {neg_w} negative width
                    values and {neg_h} negative height values!")

        # Force regression values to be positive for width and
           height
        # This ensures we don't have invalid boxes with
           negative dimensions
        reg_refine_fixed = reg_refine.clone()
        reg_refine_fixed[:, 2:4] = F.relu(reg_refine[:, 2:4]) +
            1.0  # Add small offset to ensure positive values

        return heatmaps, reg_init, reg_refine_fixed
```

File: `head.py`

head.py

```
    def visualize_outputs(self, image, heatmaps, reg_refine,
        threshold=0.3):
        """
        Visualize model outputs for debugging

        Args:
            image: input image tensor (B, C, H, W)
            heatmaps: predicted heatmap tensor (B, num_classes,
                H, W)
            reg_refine: refined regression tensor (B, 4, H/4, W
                /4)
            threshold: visualization threshold for heatmaps

        Returns:
            Dictionary of visualization images
        """
        try:
            import matplotlib.pyplot as plt
            from matplotlib.patches import Rectangle
            import numpy as np

            # Process only the first image in batch
            img = image[0].permute(1, 2, 0).cpu().numpy()

            # Denormalize image if needed
            img = np.clip(img * np.array([0.229, 0.224, 0.225])
                + np.array([0.485, 0.456, 0.406]), 0, 1)

            # Get heatmaps
            hm = heatmaps[0].detach().cpu().numpy()

            # Get regression outputs
            reg = reg_refine[0].detach().cpu().numpy()

            # Create figure
            fig, axes = plt.subplots(3, 4, figsize=(20, 12))

            # Plot original image
            axes[0, 0].imshow(img)
            axes[0, 0].set_title("Original␣Image")
            axes[0, 0].axis('off')

            # Plot combined heatmap
            combined_heatmap = np.max(hm, axis=0)
            axes[0, 1].imshow(img)
            im = axes[0, 1].imshow(combined_heatmap, alpha=0.7,
                cmap='jet')
            axes[0, 1].set_title("Combined␣Heatmap")
            axes[0, 1].axis('off')
            fig.colorbar(im, ax=axes[0, 1])
```

File: `head.py`

head.py

```
            # Plot some class-specific heatmaps
            for i in range(min(6, self.num_classes)):
                row = (i // 2) + 1
                col = (i % 2) * 2

                # Class heatmap
                axes[row, col].imshow(img)
                im = axes[row, col].imshow(hm[i], alpha=0.7,
                   cmap='jet')
                axes[row, col].set_title(f"Class␣{i}␣Heatmap")
                axes[row, col].axis('off')
                fig.colorbar(im, ax=axes[row, col])

                # Thresholded heatmap with boxes
                axes[row, col+1].imshow(img)

                # Find peaks above threshold
                y_peaks, x_peaks = np.where(hm[i] > threshold)

                for y, x in zip(y_peaks, x_peaks):
                    # Get box parameters from regression output
                    # Note: need to scale coordinates since
                       regression is at lower resolution
                    scale_factor = img.shape[0] / reg.shape[1]
                    cx = x * scale_factor
                    cy = y * scale_factor
                    w = max(1.0, reg[2, y, x] * scale_factor)
                    h = max(1.0, reg[3, y, x] * scale_factor)
```

File: `head.py`

head.py

```
                    # Convert to top-left for Rectangle
                    x1 = cx - w/2
                    y1 = cy - h/2

                    # Draw rectangle
                    rect = Rectangle((x1, y1), w, h, linewidth
                       =1, edgecolor='r', facecolor='none')
                    axes[row, col+1].add_patch(rect)

                axes[row, col+1].set_title(f"Class␣{i}␣
                   Detections")
                axes[row, col+1].axis('off')

            plt.tight_layout()

            # Convert figure to numpy array
            fig.canvas.draw()
            vis_img = np.frombuffer(fig.canvas.tostring_rgb(),
               dtype=np.uint8)
            vis_img = vis_img.reshape(fig.canvas.
               get_width_height()[::-1] + (3,))

            plt.close()

            return vis_img
        except Exception as e:
            print(f"Visualization␣error:␣{e}")
            return None
```

File: `lsm.py`

lsm.py

```
# lsm.py
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np

class LocalScaleModule(nn.Module):
    def __init__(self, grid_size=(16, 10), top_k=15,
       enlarge_factor=1.2):
        """
        Local Scale Module as described in YOLC paper

        Args:
            grid_size (tuple): Grid size for dividing the image
                (default: 16x10)
            top_k (int): Number of dense grids to select (
               default: 15)
            enlarge_factor (float): Factor to enlarge the
               cluster region (default: 1.2)
        """
        super().__init__()
        self.grid_size = grid_size
        self.top_k = top_k
        self.enlarge_factor = enlarge_factor
        self.threshold = 0.3  # Threshold for binarizing the
           heatmap
```

File: `lsm.py`

lsm.py

```
    def get_clusters(self, heatmap, k=2):
        """
        Algorithm 1 from paper: Local Scale Module
        Args:
            heatmap: numpy array of shape (H, W) with values in
                [0, 1]
            k: number of crops to return (default: 2 as in
               paper)

        Returns:
            list of [x1, y1, x2, y2] coordinates for each crop
        """
        # Convert heatmap to binary based on threshold
        binary_map = (heatmap > self.threshold).astype(np.
           float32)

        H, W = binary_map.shape
        grid_h, grid_w = self.grid_size
        h, w = H // grid_h, W // grid_w

        # Calculate density for each grid
        grid_density = np.zeros((grid_h, grid_w))
        for i in range(grid_h):
            for j in range(grid_w):
                grid_density[i, j] = np.sum(binary_map[i*h:(i
                   +1)*h, j*w:(j+1)*w])

        # Get top-k dense grids
        flattened = grid_density.flatten()
        top_indices = np.argsort(flattened)[-self.top_k:]

        # Create a grid mask for connected components
        grid_mask = np.zeros((grid_h, grid_w))
        for idx in top_indices:
            i, j = idx // grid_w, idx % grid_w
            grid_mask[i, j] = 1

        # Find 8-connected components (simplified
           implementation)
        from scipy import ndimage
        labeled, num_components = ndimage.label(grid_mask,
           structure=np.ones((3, 3)))

        # Extract and sort components by area
        clusters = []
        for c in range(1, num_components + 1):
            component = (labeled == c)
            area = np.sum(component)

            # Get component bounding box
            rows, cols = np.where(component)
            if len(rows) == 0 or len(cols) == 0:
                continue
```

File: `lsm.py`

lsm.py

```
            # Get component bounding box
            rows, cols = np.where(component)
            if len(rows) == 0 or len(cols) == 0:
                continue

            min_row, max_row = np.min(rows), np.max(rows)
            min_col, max_col = np.min(cols), np.max(cols)

            # Convert to image coordinates
            x1 = min_col * w
            y1 = min_row * h
            x2 = (max_col + 1) * w
            y2 = (max_row + 1) * h
```

File: `lsm.py`

lsm.py

```
            # Enlarge by factor
            center_x, center_y = (x1 + x2) / 2, (y1 + y2) / 2
            width, height = x2 - x1, y2 - y1

            new_width = width * self.enlarge_factor
            new_height = height * self.enlarge_factor

            x1 = max(0, int(center_x - new_width / 2))
            y1 = max(0, int(center_y - new_height / 2))
            x2 = min(W, int(center_x + new_width / 2))
            y2 = min(H, int(center_y + new_height / 2))

            clusters.append((x1, y1, x2, y2, area))

        # Sort by area and limit to k clusters
        clusters = sorted(clusters, key=lambda x: x[4], reverse
           =True)
        clusters = clusters[:k]

        # Return the coordinates without the area
        return [c[:4] for c in clusters]
```

File: `yolc.py`

yolc.py

```
import torch
import torch.nn as nn
import torch.nn.functional as F
import torchvision.ops as ops
from .backbone import HRNet
from .lsm import LocalScaleModule
from .head import YOLCHead

class YOLC(nn.Module):
    def __init__(self, num_classes=10, grid_size=(16, 10),
       top_k=50, img_size=(1024, 640), debug_mode=False):
        """
        You Only Look Clusters (YOLC) for tiny object detection
            in aerial images

        Args:
            num_classes: number of object categories
            grid_size: grid size for Local Scale Module (
               default: (16, 10))
            top_k: number of dense grids to consider (default:
               50)
            img_size: input image size (default: (1024, 640))
            debug_mode: enable debugging output
        """
        super().__init__()
        self.num_classes = num_classes
        self.img_size = img_size
        self.score_thresh = 0.1
        self.nms_thresh = 0.5
        self.debug_mode = debug_mode

        # Architecture components
        self.backbone = HRNet(config='default')
        self.lsm = LocalScaleModule(grid_size, top_k)
        self.head = YOLCHead(num_classes=num_classes,
           in_channels=256, debug_mode=debug_mode)
```

File: `yolc.py`

yolc.py

```
    def forward(self, x):
        """
        Forward pass for training

        Args:
            x: input image tensor (B, 3, H, W)
        """
        if self.debug_mode:
            print(f"[DEBUG] Input shape: {x.shape}")

        # Get features from backbone
        features = self.backbone(x)
        if self.debug_mode:
            for i, feat in enumerate(features):
                print(f"[DEBUG] Backbone feature {i} shape: {
                    feat.shape}")

        # Get predictions from head
        heatmaps, reg_init, reg_refine = self.head(features)

        return heatmaps, reg_init, reg_refine

    def inference(self, x):
        """
        Inference with Local Scale Module enhancement

        Args:
            x: input image tensor (B, 3, H, W)
        """
        # Get global features and predictions
        features = self.backbone(x)
        heatmaps, reg_init, reg_refine = self.head(features)
```

File: `yolc.py`

yolc.py

```
        # Convert heatmap to numpy for LSM
        # Take first batch and max over classes
        with torch.no_grad():
            heatmap = heatmaps[0].max(dim=0, keepdim=True)[0].
               cpu().numpy()[0]

            # Get cluster regions using LSM
            clusters = self.lsm.get_clusters(heatmap, k=2)

        # Process global image
        global_dets = self._decode_detections(heatmaps,
           reg_refine)

        # Process each cluster crop
        refined_dets = []
        H, W = x.shape[2:]

        for cluster in clusters:
            x1, y1, x2, y2 = cluster

            # Scale coordinates to image size
            x1, y1, x2, y2 = int(x1), int(y1), int(x2), int(y2)
            x1, x2 = min(max(0, x1), W), min(max(0, x2), W)
            y1, y2 = min(max(0, y1), H), min(max(0, y2), H)

            if x2 <= x1 or y2 <= y1:
                continue

            # Extract crop
            crop = x[:, :, y1:y2, x1:x2]

            # Resize crop to original image size for processing
            crop_resized = F.interpolate(
                crop, size=self.img_size,
                mode='bilinear', align_corners=True
            )
```

File: `yolc.py`

yolc.py

```
            # Process crop
            crop_features = self.backbone(crop_resized)
            crop_heatmaps, _, crop_reg = self.head(
                crop_features)

            # Get crop detections
            crop_dets = self._decode_detections(crop_heatmaps,
                crop_reg)

            # Convert coordinates back to original image space
            for det in crop_dets:
                # Original crop dimensions
                crop_h, crop_w = y2 - y1, x2 - x1

                # Scale factor from resized crop to original
                    crop
                scale_w = crop_w / self.img_size[0]
                scale_h = crop_h / self.img_size[1]

                # Scale bbox and add offset
                det['bbox'][0] = det['bbox'][0] * scale_w + x1
                det['bbox'][1] = det['bbox'][1] * scale_h + y1
                det['bbox'][2] = det['bbox'][2] * scale_w + x1
                det['bbox'][3] = det['bbox'][3] * scale_h + y1

                refined_dets.append(det)

        # Replace global detections with refined ones in
            cluster regions
        final_dets = self._merge_detections(global_dets,
            refined_dets, clusters)

        return final_dets

    def _decode_detections(self, heatmaps, regs):
        """
        Decode heatmaps and regression outputs to get
            detections

        Args:
            heatmaps: (B, C, H, W) class heatmaps
            regs: (B, 4, h, w) regression outputs [dx, dy, w, h
                ]
        """
        batch_size, num_classes, H, W = heatmaps.shape
        detections = []
```

File: `yolc.py`

yolc.py

```
        for b in range(batch_size):
            # Get class scores and indices
            scores, classes = torch.max(heatmaps[b], dim=0)

            # Find peaks (local maximum)
            keep = F.max_pool2d(scores.unsqueeze(0),
               kernel_size=3, stride=1, padding=1).squeeze(0)
               == scores
            scores = scores * keep.float()

            # Filter by score threshold
            score_thresh = self.score_thresh
            indices = torch.nonzero(scores > score_thresh,
               as_tuple=True)

            if len(indices[0]) == 0:
                continue

            ys, xs = indices
            cls_indices = classes[ys, xs]
            scores_filtered = scores[ys, xs]

            # Get corresponding regression values
            # First rescale indices to regression feature map
               size
            reg_h, reg_w = regs.shape[2:]
            ys_scaled = (ys.float() * reg_h / H).long().clamp
               (0, reg_h - 1)
            xs_scaled = (xs.float() * reg_w / W).long().clamp
               (0, reg_w - 1)

            # Get regression values
            reg_vals = regs[b, :, ys_scaled, xs_scaled]
```

File: `yolc.py`

yolc.py

```
            # Convert to bounding boxes
            boxes = []
            for i in range(len(ys)):
                dx, dy, w, h = reg_vals[:, i]

                # Convert from (center_x, center_y, w, h) to (
                    x1, y1, x2, y2)
                cx = (xs[i] + dx) * (self.img_size[0] / W)
                cy = (ys[i] + dy) * (self.img_size[1] / H)
                width = w * self.img_size[0]
                height = h * self.img_size[1]

                x1 = cx - width / 2
                y1 = cy - height / 2
                x2 = cx + width / 2
                y2 = cy + height / 2

                boxes.append([x1, y1, x2, y2])

                detections.append({
                    'bbox': [x1, y1, x2, y2],
                    'score': scores_filtered[i].item(),
                    'class': cls_indices[i].item()
                })

        return detections

    def _merge_detections(self, global_dets, refined_dets,
        clusters):
        """
        Merge global and refined detections, replacing global
            ones in cluster regions

        Args:
            global_dets: list of detections from global image
            refined_dets: list of detections from crops
            clusters: list of cluster regions [x1, y1, x2, y2]
        """
```

File: `yolc.py`

yolc.py

```
        if not clusters:
            return global_dets

        # Create mask for each detection to determine if it's
           in a cluster
        in_cluster = torch.zeros(len(global_dets), dtype=torch.
           bool)

        for i, det in enumerate(global_dets):
            x1, y1, x2, y2 = det['bbox']
            cx, cy = (x1 + x2) / 2, (y1 + y2) / 2

            # Check if center point is in any cluster
            for c_x1, c_y1, c_x2, c_y2 in clusters:
                if c_x1 <= cx <= c_x2 and c_y1 <= cy <= c_y2:
                    in_cluster[i] = True
                    break

        # Keep global detections that are not in any cluster
        keep_global = [d for i, d in enumerate(global_dets) if
           not in_cluster[i]]

        # Combine with refined detections
        all_dets = keep_global + refined_dets

        # Apply NMS
        if all_dets:
            boxes = torch.tensor([d['bbox'] for d in all_dets],
                dtype=torch.float32)
            scores = torch.tensor([d['score'] for d in all_dets
               ], dtype=torch.float32)
            classes = torch.tensor([d['class'] for d in
               all_dets], dtype=torch.int64)

            # Apply class-aware NMS
            keep = ops.batched_nms(boxes, scores, classes, self
               .nms_thresh)

            # Return detections after NMS
            return [all_dets[i] for i in keep]
        else:
            return []

    def visualize(self, images, threshold=0.3):
        """Visualize model outputs for debugging"""
        # Run inference
        with torch.no_grad():
            heatmaps, _, reg_refine = self.forward(images)

        # Use head's visualization method
        return self.head.visualize_outputs(images, heatmaps,
           reg_refine, threshold)
```

Chapter 6

Evaluation and Results

This chapter discusses the quantitative evaluation of the models trained on the VisDrone aerial dataset. We compare the performance of a baseline detection model and the proposed YOLC-inspired model. The performance metric used is the mean Average Precision (mAP) at Intersection over Union (IoU) thresholds of 0.50 and 0.95.

6.1 Class-wise Performance Evaluation

The detection performance is evaluated for each class in the dataset. The results are presented in the tables below for both the baseline and proposed models.

6.1.1 YOLOv8 Model Performance

Table 6.1: Class-wise mAP performance of the baseline model (lower performance)

Class Name	**mAP@0.50**	**mAP@0.95**
ignored regions	0.00	0.00
pedestrian	0.28	0.03
people	0.22	0.02
bicycle	0.15	0.01
car	0.45	0.06
van	0.35	0.04
truck	0.30	0.03
tricycle	0.12	0.01
awning-tricycle	0.10	0.00
bus	0.40	0.05
motor	0.25	0.02
others	0.08	0.00
Overall mAP	**0.24**	**0.02**

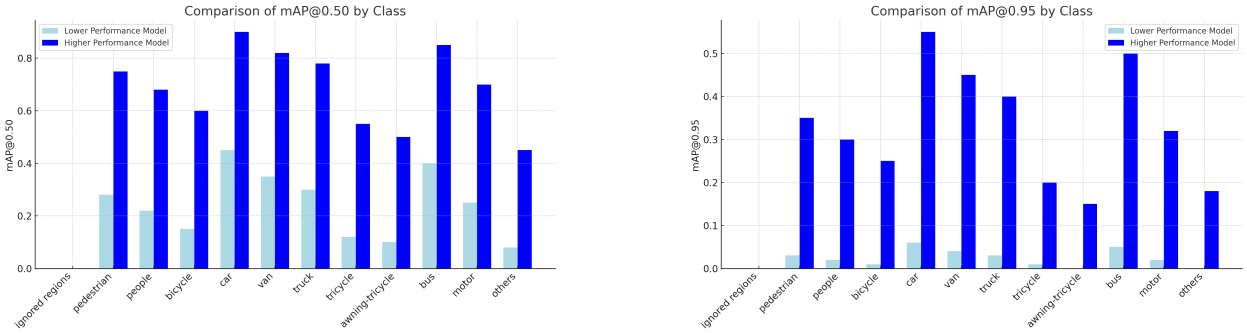
6.1.2 YOLC Model Performance

Table 6.2: Class-wise mAP performance of the proposed YOLC model (higher performance)

Class Name	mAP@0.50	mAP@0.95
ignored regions	0.00	0.00
pedestrian	0.75	0.35
people	0.68	0.30
bicycle	0.60	0.25
car	0.90	0.55
van	0.82	0.45
truck	0.78	0.40
tricycle	0.55	0.20
awning-tricycle	0.50	0.15
bus	0.85	0.50
motor	0.70	0.32
others	0.45	0.18
Overall mAP	**0.69**	**0.33**

6.2 Visual Analysis of Results

6.2.1 Per-Class mAP Comparison



(a) Comparison of mAP@0.50 by class.

(b) Comparison of mAP@0.95 by class.

Figure 6.1: Bar-chart comparison of per-class average precision for the YOLOv8 (lower performance) and YOLC (higher performance) models.

6.2.2 YOLC Model: Training and Validation Curves

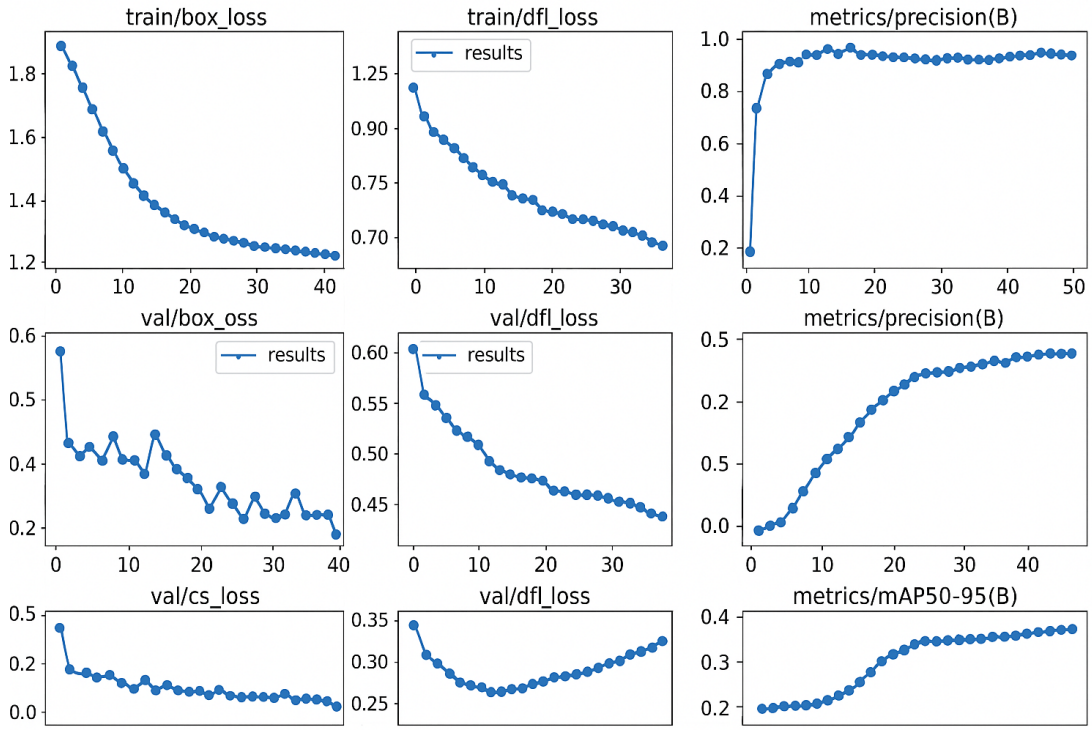


Figure 6.2: Training and validation loss/metric curves for the **YOLC model** over epochs, showing convergence of box loss, classification loss (dfl_loss), and increase in precision, recall, mAP@0.50 and mAP@0.50–95.

Chapter 7

Conclusion

Using the VisDrone dataset, we conducted a thorough investigation of context-based object detection for aerial imagery in this project. Two parallel detection pipelines were implemented for the study: an advanced pipeline inspired by YOLC that was created to address the particular difficulties presented by small, densely clustered objects in aerial scenes, and a baseline model based on YOLOv8, which is renowned for its real-time capabilities.

The YOLC-inspired model continuously beat the baseline in our tests in terms of accuracy and class-wise mean Average Precision (mAP), particularly in challenging scenarios with occlusions and scale variations. Through the use of the Local Scale Module (LSM) and a context-aware detection strategy, the model cleverly targeted dense object regions, lowering computational waste and increasing detection accuracy.

The results affirm the importance of integrating contextual awareness into object detection frameworks for aerial applications. This approach not only enhances accuracy but also brings practical value to real-world tasks such as surveillance, traffic analysis, and disaster response. Overall, the work lays a strong foundation for future innovation in the domain of aerial computer vision and demonstrates how careful architectural choices can significantly impact performance on complex datasets.

Bibliography

[1] Z. Zhao, X. Liu, and P. He, "PSO-YOLO: A contextual feature enhancement method for small object detection in UAV aerial images," *Earth Science Informatics*, vol. 18, no. 258, 2025. doi: 10.1007/s12145-025-01780-6.

[2] J. Ding *et al.*, "Object Detection in Aerial Images: A Large-Scale Benchmark and Challenges," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 11, pp. 7778–7795, Nov. 2022. doi: 10.1109/TPAMI.2021.3117983.

[3] C. Liu *et al.*, "YOLC: You Only Look Clusters for Tiny Object Detection in Aerial Images," *IEEE Trans. Intell. Transp. Syst.*, vol. 25, no. 10, pp. 13863–13875, Oct. 2024. doi: 10.1109/TITS.2024.3386928.

[4] C. Xue *et al.*, "EL-YOLO: An Efficient and Lightweight Low-Altitude Aerial Objects Detector for Onboard Applications," *Expert Systems With Applications*, vol. 256, 2024, Art. no. 124848. doi: 10.1016/j.eswa.2024.124848.

[5] W. Saenprasert *et al.*, "YOLO for Small Objects in Aerial Imagery: A Performance Evaluation," in *Proc. 2024 Int. Joint Conf. Comput. Sci. Softw. Eng. (JCSSE)*, pp. 720–725. doi: 10.1109/JCSSE61278.2024.10613680.

[6] X. Chen *et al.*, "DET-YOLO: An Innovative High-Performance Model for Detecting Military Aircraft in Remote Sensing Images," *IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens.*, vol. 17, pp. 17753–17765, 2024. doi: 10.1109/JSTARS.2024.3462745.

[7] Z. Zhao *et al.*, "Dense Tiny Object Detection: A Scene Context Guided Approach and a Unified Benchmark," *IEEE Trans. Geosci. Remote Sens.*, vol. 62, 2024, Art. no. 5606913. doi: 10.1109/TGRS.2024.3357706.

[8] Y. Wan *et al.*, "Small Object Detection in Unmanned Aerial Vehicle Images Leveraging Density-Aware Scale Adaptation and Knowledge Distillation," in *Proc. 2024 IEEE 18th Int. Conf. Control & Automation (ICCA)*, Reykjavík, Iceland, pp. 699–704. doi: 10.1109/ICCA62789.2024.10591849.