Automating Tree Counting and Detection with High Resolution Satellite Imagery on Urban Areas Using a Modified YOLO Framework

Osman Hussien Fadl Ali[1]

MSc Data Science and Machine Learning

Primary Supervisor: Marta Betcke

Submission date: 09 09 2024

[1] This report is submitted as part requirement for the MSc Degree in Data Science and Machine Learning at University College London. It is substantially the result of my own work except where explicitly indicated in the text. The report may be freely copied and distributed provided the source is explicitly acknowledged.

Acknowledgements

First and foremost, I would like to thank Allah (subhanahu wa ta'ala) for granting me the strength, knowledge, and perseverance to complete this project. Alhamdulillah. This past year has been the most challenging year of my life, as my beloved country, Sudan, has endured the hardships of war for over 500 days. Despite this, I feel incredibly privileged to have pursued this course, and I dedicate this work to Sudan and to my family, whose unwavering support has carried me to this point.

I am sincerely grateful to Dr. Marta Betcke for her support and feedback throughout the project. I would also like to extend my heartfelt thanks to my colleagues at QBE for welcoming me into the Pricing Data Science team and providing me the opportunity to work on this topic. Special thanks to Sudheer Pamula and Thibaud du Crest for their mentorship and thoughtful advice, which have been instrumental to my growth during this journey. Finally, thank you to all my friends who have been a part of this journey.

Abstract

In recent years, the task of tree counting using satellite imagery has become increasingly important for environmental monitoring and climate change mitigation. However, accurately detecting trees is a complex task due to the inherent challenges posed by low-resolution satellite images and the natural characteristics of trees—such as dense growth patterns, overlapping canopies, the presence of shadows, and seasonal variations in colour and foilage—which complicate detection from aerial perspectives. In this work, we approach tree counting as an object detection problem by training and evaluating three state-of-the-art deep learning models—SSD, RetinaNet, and YOLOv8—on a publicly available tree dataset. Among these models, YOLOv8 emerged as the best performer in terms of both accuracy and speed. Building on the strengths of YOLOv8, we propose the novel YOLOv8-small model, a compact version of the original architecture, achieved by simply removing the layers in the network associated with generating predictions for low-resolution feature maps. This modification allowed the lightweight model to retain high-resolution features, effectively addressing the challenges posed by low-resolution satellite imagery. The YOLOv8-small outperformed the original model in terms of accuracy, speed, recall, and precision, while also reducing training time to just 10 minutes. During evaluation, we observed that none of the trained models generalised well to random satellite images, revealing an out-of-distribution problem. To address this, we created and manually annotated a geospatial dataset comprising satellite imagery from various countries, including challenging edge cases such as shadows, seasonal variations, and overlapping canopies. We introduce a simple yet effective baseline method using canopy density data and a straightforward formula, benchmarking it against all models. Our final results show the YOLOv8-small model outperforming all other methods in this study, demonstrating the potential of architecture-specific optimisations tailored to the task of tree detection.

Contents

1 Introduction **2**

1.1 Motivation . . . 2
1.2 Research Aim and Objectives . . . 3
1.3 Industry Partner: QBE . . . 3
1.4 Contributions . . . 4

2 Literature Review **5**

2.1 Tree Counting Techniques . . . 5
2.2 Deep Learning for Tree Counting . . . 6
2.3 Metrics . . . 6
2.4 Object Detection . . . 8
2.4.1 Traditional Detectors . . . 8
2.4.2 Two-stage Detectors . . . 8
2.4.3 One-stage Detectors . . . 9
2.4.4 Non-Maximum Suppression . . . 12
2.5 YOLO . . . 13
2.5.1 YOLOv1: The Foundation . . . 13
2.5.2 Evolution of YOLO Frameworks . . . 15
2.5.3 YOLOv8 . . . 16

3 Model Design and Experiments **19**

3.1 Datasets . . . 19
3.1.1 Tree Dataset . . . 19
3.1.2 Geospatial Dataset . . . 22
3.2 Baseline . . . 24
3.3 Object Detection Model Training . . . 26
3.3.1 Google Cloud Platform . . . 26
3.3.2 SSD . . . 27
3.3.3 RetinaNet . . . 27
3.3.4 YOLOv8 . . . 28
3.4 Model Architecture Modifications . . . 29
3.5 Geospatial Training . . . 32

4 Results and Evaluation **33**
4.1 Model Training on Tree Dataset 33
4.2 YOLOv8 Architecture Evaluation 36
4.3 Geospatial Dataset Evaluation 40

5 Discussion **45**
5.1 Baseline Method 45
5.2 Architecture Modification 46
5.3 Datasets 47

6 Conclusion **48**

A Hyperparameter Tuning **54**

B Model Architectures **56**
B.1 SSD 56
B.2 RetinaNet 58
B.3 YOLOv8s 65
B.4 Modified YOLOv8-small 65

C Predictions on Tree Dataset **66**

Chapter 1

Introduction

1.1 Motivation

In the past few years, sustainable tree resource management has become increasingly important due to the impacts of biodiversity loss and climate change [1]. Policies addressing climate change mitigation, sustainable wood production, and biodiversity must rely on timely and reliable information about the number of trees, their species and tree canopy distribution. This information is commonly derived from remote sensing data, such as satellite images, and is often combined with national forest inventories (NFI) [2]. However, most inventories and satellite-based studies do not include systematic assessments of trees outside forests [3], and recent studies have shown that trees in urban and agricultural landscapes constitute a considerable part of global tree resources [4, 5]. The importance of accounting for trees outside traditional forests has been highlighted by recent reports on the Tesla Gigafactory in Germany, where satellite images revealed that approximately 500,000 trees were cut down, sparking significant environmental concerns and a widespread debate [6].

With the advent of high-resolution satellite imagery and advancements in deep learning, there is now an unprecedented opportunity to expand the scope of tree monitoring to include urban and suburban areas. However, this expansion brings several challenges. Remote sensing data, such as satellite imagery, is inherently geospatial, often containing multiple spectral bands that require specialised processing techniques different from traditional image processing. Although recent developments in higher-resolution satellite imagery have been substantial, the resolution is still often insufficient to clearly distinguish individual trees in dense, cluttered environments. Trees are densely packed by nature, with smaller trees frequently overshadowed by larger ones, making it hard to detect individual trees from an aerial viewpoint. Additionally, trees grow, are cut down, and seasonal variations cause trees to appear differently throughout the year. Therefore, to develop a viable solution to the problem, one must take into account the dynamic nature of tree populations and the complexities of geospatial data. In this work, we tackle all these challenges by leveraging state-of-the-art deep learning techniques to detect trees in satellite imagery.

1.2 Research Aim and Objectives

The aim of this thesis is to implement and refine current state-of-the-art deep learning methods to accurately detect and count the number of trees in urban satellite imagery. We outline the following objectives needed to achieve this aim:

1. Formulate the problem of tree counting as an object detection task and review current state-of-the-art deep learning methods for this use case.
2. Develop a baseline method for tree detection to serve as a reference point for evaluating the performance of the deep learning models.
3. Prepare and annotate tree datasets using publicly available sources, ensuring diverse coverage of urban and suburban areas.
4. Train deep learning models on the prepared datasets and conduct a thorough evaluation of their generalisation capabilities across different types of satellite imagery, including those with varying resolutions and environmental conditions.
5. Propose and implement architecture modifications aimed at enhancing the accuracy and efficiency of the selected deep learning models

By pursuing these objectives, we are effectively answering two research questions: "How can we leverage deep learning for the complex task of tree detection?" and "What architectural modifications can be made to deep learning models to circumvent the challenges associated with satellite imagery?". Through this research, we not only explore the potential of deep learning for tree detection but also contribute to the ongoing discourse on improving model architectures for specialised tasks.

1.3 Industry Partner: QBE

This research was done in collaboration with QBE, a global insurance company that has presence in 27 countries [7]. Within QBE's European headquarters in London, the Pricing Data Science team is at the forefront of exploring advanced data-driven techniques to enhance their risk assessment models. The team is particularly interested in the application of deep learning methods for detecting trees in satellite imagery as a method of risk assessment for the properties they insure. Accurate detection and analysis of tree cover in urban and suburban areas is important for assessing risks related to natural disasters, such as storms and wildfires, which can have significant financial implications for the insurance industry. By partnering with QBE, this research benefits from direct access to industry expertise, compute and the opportunity to test and refine the developed models in a practical setting.

1.4 Contributions

In this thesis, there are three main contributions:

1. **Baseline Method:** The first contribution is the development of a novel and straightforward benchmarking tool, referred to as the Baseline Method. This method provides a rough yet effective estimate of the number of trees within any defined region of interest. By leveraging low-resolution satellite imagery and basic canopy density data, the Baseline Method offers a quick and computationally efficient means of establishing a benchmark for the deep learning models.

2. **YOLOv8-small Model:** The second major contribution is the development of a novel, compact version of the YOLOv8 architecture, named YOLOv8-small. Initially, three deep learning models—SSD, RetinaNet, and YOLOv8—were trained and evaluated for the task of tree detection. After determining that YOLOv8 outperformed the other architectures, it was further refined to better address the unique challenges of detecting trees in urban satellite imagery. This refinement involved the strategic removal of layers responsible for processing downsampled feature maps at medium and low resolutions. To justify this approach, it's important to note that the original YOLOv8 model processes feature maps at multiple resolutions, combining low, medium and high-resolution features to detect objects of varying sizes. However, since satellite imagery is already low in resolution, further downsampling diminishes the detail needed for detecting small or densely packed trees. By removing layers responsible for processing downsampled feature maps, the model prioritised high-resolution features crucial for accurate detection. The resulting YOLOv8-small model not only delivers superior accuracy and efficiency in tree detection but also offers a robust, specialised solution tailored to the problem of environmental monitoring in urban areas.

3. **Geospatial Dataset:** The third major contribution is the curation and annotation of a geospatial dataset, consisting of high-resolution satellite images from various urban regions. This dataset has been meticulously labeled to include a wide range of challenging scenarios, such as small trees, shadows, and varying tree densities. The primary purpose of this dataset is to evaluate how well models generalise to challenging satellite imagery, improve their performance on edge cases, and enable a direct comparison between our baseline method and advanced deep learning models.

Chapter 2

Literature Review

This chapter aims to give a comprehensive overview of the existing methods for tree counting and detection, highlighting the evolution from traditional tree counting to modern deep learning approaches with a focus on object detection. The chapter discusses key algorithms, metrics, and the progression of loss functions used in object detection, while differentiating between one-stage and two-stage detectors. The goal of this chapter is to familiarise the reader with the foundational work and methodologies that underpin the research presented in this thesis.

2.1 Tree Counting Techniques

Traditional tree counting methods primarily relied on field sampling techniques, which involved systematic or random sampling within forest stands [8]. This approach requires manually measuring parameters such as tree location, basal area, height, tree species, and crown size before using extrapolation methods for larger areas. With the advent of aerial photography, experts began manually identifying and counting trees from aerial images, a process that was labor-intensive and time-consuming [8]. As a result, several algorithms were developed to automate this process. Local Maximum (LM) filtering is an example used for identifying tree crowns in spatial imagery by recognising the points with the greatest brightness within a search window that scans the entire image [9]. Choosing the right window size is critical for the success of LM techniques. Template Matching (TM) involves correlating image sections with pre-defined models of tree shapes, yet it is computationally intensive and may struggle with variations of tree sizes [9]. Contour-based (CB) methods delineate tree boundaries using edge detection, although these methods can be affected by the complexity of forest backgrounds [8]. For example, small trees that are overshadowed by larger trees may not be accurately detected and dense forests can cause overlapping crowns which can be interpreted as a single entity, leading to underestimation. One way to address these challenges is by developing approaches that combine edge detection with marker-controlled watershed segmentation [8]. The former helps in identifying individual tree tops and the latter helps in delineating tree crowns.

Recent advances in remote sensing technologies have significantly enhanced the ability to map and monitor trees across large areas with unprecedented precision and efficiency. Remote sensing involves the acquisition of data from satellite imagery and aerial platforms, providing a wealth

of information that traditional field sampling methods cannot achieve alone. Techniques such as Light Detection and Ranging (LiDAR), Synthetic Aperture Radar (SAR), and multi-spectral and hyperspectral imaging have revolutionised tree detection by offering detailed information about tree structure, biomass, and species composition [10]. LiDAR, for instance, provides three-dimensional representations of forest canopy and structure, enabling accurate height measurements and crown delineation, which are essential for estimating tree density and biomass. These methods have facilitated the development of large data-driven models, like the one developed by Meta [11], which uses LiDAR to create a high-resolution map of global tree canopy height. Similarly, Google developed Google Earth Engine [12], a platform that provides access to remote sensing data and geospatial datasets for analysis and visualisation. The baseline method in this study was built on this platform.

2.2 Deep Learning for Tree Counting

Given the limitations of traditional approaches and the continuous improvement of GPU hardware, deep learning has been vigorously developed, showing significant advantages for tasks like tree detection. Li et al. developed a model based on the U-Net architecture, which provides location, crown area, and height for individual overstory trees from aerial images [13]. This multitask deep neural network has two branches: one for predicting tree counts by regressing density maps and another for segmenting crowns by classifying pixels as either a tree or a background pixel. The counting branch predicted the tree count by regressing density maps, where each tree crown was represented by a small sample point located at its centre on the density map [13]. Despite its robustness, the model faced challenges related to tree detection which were also mentioned in other literature, such as a high bias with tall trees and difficulties in interpreting results at a low spatial resolution. Yao et al. evaluated four different classical convolutional neural networks (CNNs): AlexNet, VGGNet, a combined network of shallow and deep layers, and an Encoder-Decoder network (U-Net) [14]. They found that the U-Net Encoder-Decoder Network achieved the best performance with a Mean Absolute Error (MAE) significantly lower than the other models. However, the study highlighted that all models tended to underestimate tree densities in areas with high ground truth values due to partial overlap and the multi-scale characteristics of trees. Therefore, when evaluating tree detection models, it is crucial to use a comprehensive dataset that includes edge cases such as trees with shadows, small trees, and trees of varying resolutions and types.

2.3 Metrics

In terms of metrics, the choice of evaluation metrics primarily depends on whether the task of tree detection is formulated as a semantic segmentation problem, or an object detection problem. The two aforementioned papers treated the problem as a segmentation task. In doing so, they employed metrics like Intersection over Union (IoU), which evaluates the overlap between predicted and ground truth regions. IoU measures the ratio of the intersection area to the union area of the predicted and ground truth bounding boxes, providing a normalised measure of accuracy [15]. On the other hand, object detection mostly relies on the mean Average Precision (mAP) metric,

which provides a comprehensive measure of the model's performance in terms of both precision and recall by averaging the precision-recall curve across different IoU thresholds [16]. In other words, Average Precision (AP) is the area under a precision-recall curve. Precision in object detection is defined as the number of true positives divided by the sum of true positives and false positives, where a true positive occurs when the IoU between the predicted box and ground truth exceeds a predefined threshold, and a false positive occurs when it is below that threshold [17]. Recall, on the other hand, is defined as the number of true positives divided by the sum of true positives and false negatives, with a false negative occurring when the model fails to detect the object within the bounding box.

Calculating mAP involves mapping each detection box to its most-overlapping ground-truth object instance and calculating precision and recall values for increasingly large subsets of detections, thus providing a detailed assessment of the model's accuracy across varying conditions. The most common IoU thresholds for mAP are 0.50 (mAP-0.50) and 0.95 (mAP0.5:0.95) [16], and these will be used in this paper to evaluate the tree detection models. For mAP0.5:0.95, we take steps of 0.05 starting from an IoU threshold of 0.5 and stopping at 0.95. The average precision over this interval is the class average precision. We do this for all classes and take the average over them to generate the mAP50-95. The equations below summarise the metrics discussed used in this study. We include the F-1 score which is a byproduct of Precision and Recall:

$$\text{MAE} = \frac{1}{N}\sum_{j=1}^{N}|\text{Prediction}_j - \text{True Value}_j| \tag{2.1}$$

$$\text{IoU} = \frac{\text{Area of Overlap}}{\text{Area of Union}} = \frac{B_p \cap B_{gt}}{B_p \cup B_{gt}} \tag{2.2}$$

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2.3}$$

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{2.4}$$

$$\text{F-1 Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2.5}$$

$$\text{AP} = \int_0^1 \text{Precision}(r)\, dr \tag{2.6}$$

$$\text{mAP} = \frac{1}{N}\sum_{i=1}^{N}\text{AP}_i \tag{2.7}$$

where:

- N: Total number of predictions or classes, depending on the context.
- B_p: Predicted bounding box
- B_{gt}: Ground-truth bounding box
- TP: True Positives
- FP: False Positives
- FN: False Negatives
- r: Recall Threshold

2.4 Object Detection

In this paper, the task of tree counting and detection will be formulated as an object detection problem. During the past two decades, object detection has undergone historical strides, cementing it as an integral component of modern-day computer vision. To understand the work in this paper, one must become familiar with the evolution of object detection techniques and the current methods employed in literature.

2.4.1 Traditional Detectors

The Viola-Jones detector (VJ), introduced in 2001, was the first real-time object detection system, utilising a cascade of classifiers trained with AdaBoost for rapid and efficient face detection [18,19]. The VJ detector was tens or even hundreds of times faster than other algorithms in its time under comparable detection accuracy. It followed the most straightforward way of detection, i.e., sliding windows: to go through all possible locations in an image to see if any window contains a human face. In 2005, Dalal and Triggs proposed the histogram of oriented gradients (HOG) feature descriptor [20]. HOG can be considered an important improvement over methods like the scale-invariant feature transform, as it focuses on the distribution of intensity gradients and edge directions within localised portions of the image. In essence, HOG divides an image into small, connected regions called cells, and for each cell, it computes a histogram of gradient directions. These histograms are then concatenated to form a feature vector that describes the local shape of the object. By normalising these histograms across larger regions called blocks, HOG increases the robustness of the feature against changes in illumination and shadowing, a common challenge in this project.

The Deformable Part-Based Model (DPM), introduced by Felzenszwalb et al. in 2008, marked another significant advancement in traditional object detection methods [21]. DPM builds upon the HOG descriptor and follows a strategy where the training can be simply considered as the learning of a proper way of decomposing an object, and the inference can be considered as an ensemble of detections on different object parts [22]. For example, the problem of detecting a "car" can be decomposed to the detection of its wheels and windows. Although today's object detectors have since surpassed DPM in detection accuracy, many of them are still deeply influenced by its legacy. Concepts such as part-based representation continue to inform modern detection frameworks including advanced neural network architectures like CNNs.

In 2010s, deep convolutional networks significantly transformed the landscape of object detection when they were able to learn high-level feature representations of an image, resulting in substantial improvements in accuracy and speed over traditional methods [23]. CNN object detection systems can be broadly classified into two-stage and one-stage detectors, where the former formulates the detection as a "coarse-to-fine" process, and the latter frames it as to "complete in one step" [22].

2.4.2 Two-stage Detectors

The two-step framework divides the detection process into a region proposal and a classification stage. In the first step, the detector first proposes several object candidates using reference boxes

which are known as anchors. During the classification stage, the proposals are classified and their localisation is refined [24]. In R-CNN, the pioneer model that utilised this framework [25], an external selective search was used to generate the object proposals before being rescaled to a fixed-size image and fed into a pre-trained CNN model to perform classification and bounding box regression. Faster-RCNN proposed a scheme in which features are shared between both stages, achieving a significant efficiency improvement without compromising accuracy [26]. The main innovation was introduction of a Region Proposal Network (RPN) that shares full-image convolutional features with the detection network, thus enabling nearly cost-free region proposals. Faster R-CNN has inspired many other works that tried to improve detection accuracy with various approaches, such as designing better backbones that can obtain richer representations of the input image. For instance, feature pyramid networks (FPN) were proposed in order to crop object candidate features from different levels depending on the scale [27]. While many deep learning models use popular backbones such as VGG [28], or AlexNet [29], FPN has now become the basic building block of many latest detectors.

2.4.3 One-stage Detectors

The one-stage framework contains a single feed-forward fully convolutional network that directly provides the bounding boxes and the object classification [24]. The Single Shot MultiBox Detector (SSD) was among the first to propose a single unified architecture that retrieves all detected objects in one-step inference [30]. This is advantageous as it allows for real-time detection with accuracy comparable to two-stage detectors, all while reducing computational complexity. The SSD architecture consists of VGG16 backbone, followed by several convolutional layers that predict object categories and bounding boxes at multiple scales. These convolutional layers progressively decrease in size, allowing the model to capture objects of various sizes. The predictions are made using a set of default anchor boxes, each associated with multiple aspect ratios and scales, which are tiled across the feature maps as seen in figure 2.1. The key innovation of SSD lies in its approach to multireference and multiresolution detection techniques, which significantly improved the detection accuracy of one-stage detectors [22], especially for small objects like a tree in an aerial image. By leveraging feature maps from different layers of the network, SSD effectively handles objects of varying sizes and aspect ratios, thereby enhancing its robustness across diverse scenarios [30]. SSD outperformed traditional sliding window detectors, achieving high precision and recall rates, demonstrated by its competitive performance on challenging datasets like COCO, with a mAP of 46.5% at an IoU threshold of 0.5 [30]. The SSD loss is derived from the MultiBox objective [31, 32] but extended to handle multiple object categories. The overall objective loss function is a weighted sum of the localisation loss (loc) and the confidence loss (conf):

$$L(x, c, l, g) = \frac{1}{N} \left(L_{\text{conf}}(x, c) + \alpha L_{\text{loc}}(x, l, g) \right) \tag{2.8}$$

where N is the number of matched default boxes, α is a weight term set by cross-validation, c represents the class confidences, l are the predicted box parameters, g represents the ground truth box parameters and x serves as a binary indicator variable that evaluates to 1 when the default box matches a ground truth box, and 0 otherwise. This is common in object detection models where each box is either matched to a ground truth object or considered a background.

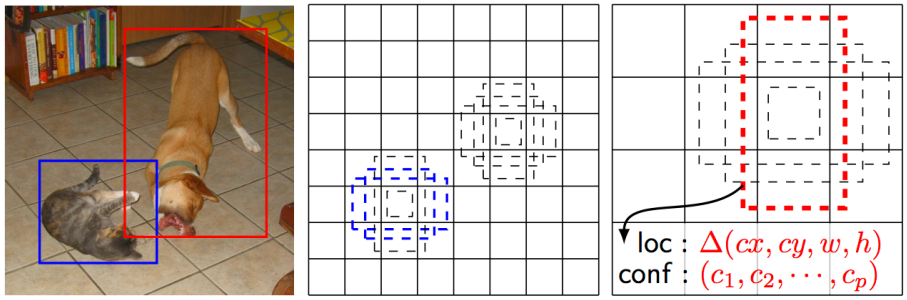
The localisation loss L_{loc} is a Smooth L1 [33] loss between the predicted box parameters and the ground truth box parameters, regressing to offsets for the center (c_x, c_y), width (w) and height (h). It is smooth in the sense that it avoids the sharp transitions of the L1 loss by using a quadratic function for small errors and a linear function for larger errors. The double sum first iterates over all matched default boxes and then over the bounding box parameters (c_x, c_y, w, h), where l_m and g_m represent the predicted and ground truth parameters, respectively:

$$L_{\text{loc}}(x, l, g) = \sum_{i \in \text{Pos}} \sum_{m \in \{c_x, c_y, w, h\}} x_{ij} \, \text{Smooth}_{L1} \left(l_m^i - \hat{g}_m^j \right) \quad \text{where} \quad \text{Smooth}_{L1}(z) = \begin{cases} 0.5\, z^2 & \text{if } |z| < 1, \\ |z| - 0.5 & \text{otherwise} \end{cases} \tag{2.9}$$

The confidence loss L_{conf} is the softmax loss over multiple class confidences. The class confidence c_p^i represents the predicted probability that default box i belongs to class p, computed via a softmax over all possible classes including the background:

$$L_{\text{conf}}(x, c) = -\sum_{i \in \text{Pos}} x_{ij}^p \log(\hat{c}_p^i) - \sum_{i \in \text{Neg}} \log(\hat{c}_0^i) \text{ where: } \hat{c}_p^i = \frac{\exp(c_p^i)}{\sum_p \exp(c_p^i)} \tag{2.10}$$

The localisation loss computes the errors in predicted bounding box coordinates, while the confidence loss measures the accuracy of the predicted class labels. The figure below shows how SSD makes model predictions using the loss functions above:



(a) Image with GT boxes (b) 8×8 feature map (c) 4×4 feature map

Figure 2.1: SSD framework [30]. (a) Input image with ground truth boxes for a cat and a dog. (b) An 8×8 feature map with default boxes of various aspect ratios at each location. (c) A 4×4 feature map demonstrating how SSD predicts both localization $(\Delta(cx, cy, w, h))$ and confidence $((c_1, c_2, \cdots, c_p))$ for each default box. During training, default boxes are matched to ground truth boxes, with matched boxes treated as positives in blue and the rest as negatives in red. The model's loss is a combination of localization loss (Smooth L1) and confidence loss (Softmax).

In the example above, there are only two objects in the input image while the rest is classified as background. This uneven distribution between foreground objects and background is known as the class imbalance problem [34]. This issue causes the model to predominantly learn from the vast number of easy negative examples, thereby reducing its effectiveness in identifying positive examples (the cat and the dog). Although SSD employs a technique called Hard Negative Mining [35] to mitigate this problem by selecting a subset of negative examples with high loss for training, it still does not surpass the accuracy of two-stage detectors. Lin et al. [36] extensively investigated this issue and attributed the inferior accuracy of one-stage detectors compared to two-stage detectors

to the class imbalance problem. They proposed a novel loss function called Focal Loss, designed to tackle class imbalance by down-weighting the loss assigned to well-classified examples, thereby focusing the learning process on hard, misclassified examples. The focal loss (FL) is essentially a modification of the standard cross entropy loss (CE):

$$\mathrm{CE}(p_t) = -\log(p_t) \tag{2.11}$$

$$\mathrm{FL}(p_t) = -(1-p_t)^{\gamma}\log(p_t) \tag{2.12}$$

where p_t is defined as:

$$p_t = \begin{cases} p & \text{if } y = 1 \\ 1-p & \text{if } y = 0 \end{cases} \tag{2.13}$$

Here, y specifies the ground-truth class and p is the model's estimated probability for the class with label $y = 1$. The subscript t in p_t stands for "target", as it represents the probability of the target (correct) class. The focusing parameter γ adjusts the rate at which easy examples are down-weighted, and can be rescaled to focus more on hard, misclassified examples.

Building on this innovation, Lin et al. introduced the RetinaNet architecture, which effectively integrates the Focal Loss into its design [36]. RetinaNet is a one-stage detector that retains the speed advantage of single-stage methods while addressing the class imbalance problem, thereby achieving accuracy levels greater than two-stage detectors. This was the first one-stage detector that was able to outperform the existing state-of-the-art (SOTA) two-stage detectors in both speed and accuracy [36], marking a significant shift in the computer vision community towards the adoption of one-stage detectors for object detection tasks [22]. The RetinaNet architecture consists a FPN backbone on top of ResNet architecture with a novel head design that predicts both class and box outputs at every pyramid level. The FPN is responsible for constructing a multi-scale feature pyramid from a single-resolution input image, which allows for better object detection at various scales. The classification and regression tasks are handled by two separate subnetworks that are attached to each level of the pyramid. The classification subnet predicts the probability of an object being present at each spatial position for each anchor box, while the regression subnet calculates the offsets required to adjust the anchor boxes to fit the detected objects more accurately. This architecture, combined with the Focal Loss, allows RetinaNet to focus on hard, misclassified examples, leading to superior detection performance. The RetinaNet architecture is seen below:

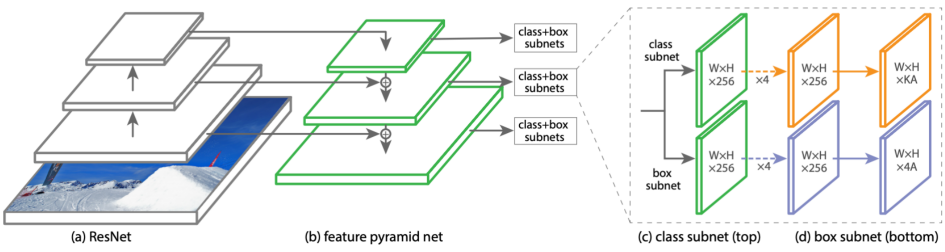


Figure 2.2: The one-stage RetinaNet network architecture uses a Feature Pyramid Network (FPN) backbone on top of a feedforward ResNet architecture (a) to generate a rich, multi-scale convolutional feature pyramid (b). To this backbone RetinaNet attaches two subnetworks, one for classifying anchor boxes (c) and one for regressing from anchor boxes to ground-truth object boxes (d) [36].

2.4.4 Non-Maximum Suppression

Most object detection models end up detecting multiple bounding boxes around the objects of interest (see Figure 2.1). Non-Maximum Suppression (NMS) is a post-processing technique used in object detection algorithms to reduce the number of overlapping bounding boxes and improve the overall detection quality [37]. NMS filters out redundant and irrelevant bounding boxes using a two-step process: first, it selects the bounding box with the highest confidence score, and then it suppresses all other bounding boxes that have a significant overlap with the selected one, based on a predefined Intersection over Union (IoU) threshold. This process is repeated until all bounding boxes have either been selected or suppressed. Algorithm 1 describes this procedure in detail and Figure 2.3 shows NMS applied to an aerial image containing a tree. The purpose of NMS is to ensure that each object is detected only once, thus improving the precision of the model by eliminating multiple detections of the same object.

Algorithm 1 Non-Maximum Suppression Algorithm [38]

Require: Set of predicted bounding boxes B, confidence scores S, IoU threshold τ, confidence threshold T
Ensure: Set of filtered bounding boxes F

$F \leftarrow \emptyset$
Filter the boxes: $B \leftarrow \{b \in B \mid S(b) \geq T\}$
Sort the boxes B by their confidence scores in descending order
while $B \neq \emptyset$ **do**
    Select the box b with the highest confidence score
    Add b to the set of final boxes F: $F \leftarrow F \cup \{b\}$
    Remove b from the set of boxes B: $B \leftarrow B - \{b\}$
    for all remaining boxes r in B **do**
        Calculate the IoU between b and r: $iou \leftarrow IoU(b, r)$
        if $iou \geq \tau$ **then**
            Remove r from the set of boxes B: $B \leftarrow B - \{r\}$
        end if
    end for
end while

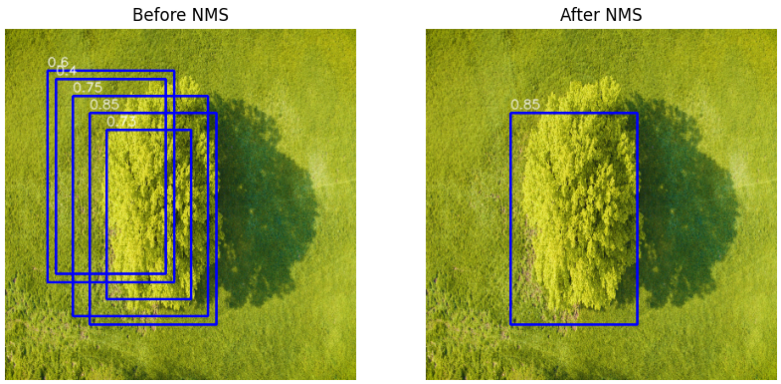


Figure 2.3: Illustration of Non-Maximum Suppression (NMS) applied to an aerial image containing a tree. First figure shows multiple overlapping bounding boxes around the tree with associated confidence scores. Figure on the right demonstrates the result after applying NMS, where redundant boxes are suppressed, leaving only the most confident box that accurately encompasses the object.

All the models used in this paper employ NMS as a standard post-processing step to handle multiple detections and to ensure that only the most relevant bounding boxes are retained. In the case of RetinaNet, NMS filters out low-confidence detections that could otherwise clutter the final output. Similarly, SSD and YOLO (next section) also use NMS to refine their detection results, particularly in scenarios where there are many overlapping bounding boxes due to the presence of multiple objects in close proximity. Since trees often grow closely together, it is essential to leverage NMS to effectively differentiate individual trees and prevent multiple detections of the same tree in densely forested areas.

2.5 YOLO

A substantial amount of this work is based on the YOLO (You Only Look Once) framework. Therefore, it is instructive to dedicate this section to an exploration of the early versions of YOLO to discuss the major changes in network architecture before looking at the current SOTA YOLO methods. The YOLO family has evolved through multiple iterations, each addressing limitations and enhancing performance, ultimately solidifying its position as a leading choice in real-time object detection tasks.

2.5.1 YOLOv1: The Foundation

YOLO by Joseph Redmon et al. was published in CVPR 2016 [39]. It presented the first real-time end-to-end approach for object detection. The first YOLO variant was the YOLOv1 [39], which was based on a modified version of the GoogLeNet model [40], featuring 24 convolutional layers followed by 2 fully connected layers. YOLO reframed the object detection problem as a single regression problem by directly predicting bounding boxes and class probabilities from full images using a single neural network. This was done by dividing the image into a $S \times S$ grid, with each grid cell predicting B bounding boxes and their associated class probabilities for C different classes. Each bounding box prediction is characterised by five components: P_c, b_x, b_y, b_h, and b_w. Here, P_c represents the confidence score, indicating the likelihood that the box contains an object. The coordinates b_x and b_y denote the center of the box relative to the grid cell, while b_h and b_w represent the height and width of the box relative to the entire image. The final output of YOLO is a tensor with dimensions $S \times S \times (B \times 5 + C)$. Figure 2.4 demonstrates an example where for each cell, YOLO predicts the probability of an object being present (P_c), the bounding box coordinates (b_x, b_y, b_h, b_w), and class labels (c_1, c_2, c_3). In this case, the cell with the center of the dog predicts bounding boxes with associated confidence scores and class probabilities for each predefined class. The output of this example is a $3 \times 3 \times 8$ which is followed by NMS to remove duplicate detections.

Similar to SSD, YOLOv1 utilises a loss function that combines both a localisation loss and a confidence loss, together with a classification loss. However, there are some key differences in how these losses are computed and combined. YOLOv1's loss function is composed of multiple sum-squared errors, with each component weighted differently to balance their contributions to the overall loss.

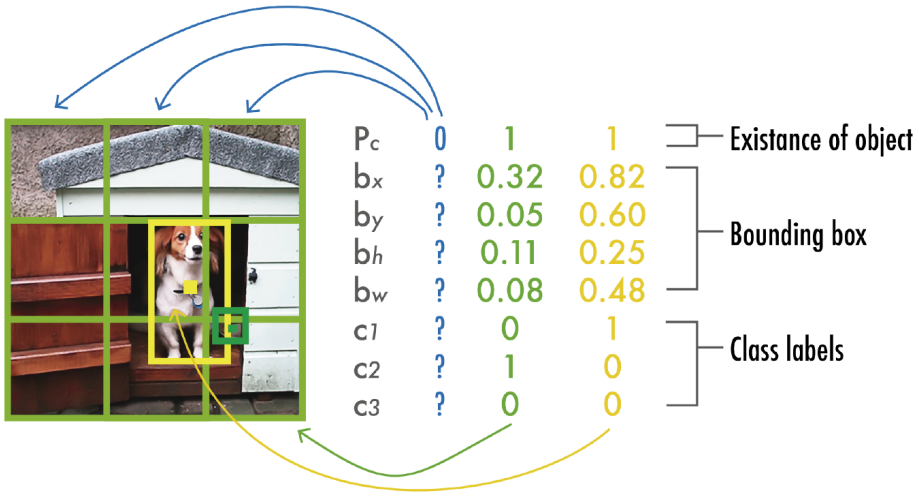


Figure 2.4: YOLOv1 prediction example [38]. The image is divided into a $S \times S$ grid. Each grid cell is responsible for predicting a bounding box if the center of an object falls within it.

$$
\begin{aligned}
\mathrm{L}(\mathbf{x}, \mathbf{y}, \mathbf{w}, \mathbf{h}, \hat{\mathbf{x}}, \hat{\mathbf{y}}, \hat{\mathbf{w}}, \hat{\mathbf{h}}, \hat{\mathbf{C}}, \mathbf{C}, \hat{\mathbf{p}}, \mathbf{p}) = {} & \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbf{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
& + \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbf{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right] \\
& + \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbf{1}_{ij}^{\text{obj}} \left(\hat{C}_i - C_i \right)^2 \\
& + \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbf{1}_{ij}^{\text{noobj}} \left(\hat{C}_i - C_i \right)^2 \\
& + \sum_{i=0}^{S^2} \mathbf{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} \left(\hat{p}_i(c) - p_i(c) \right)^2
\end{aligned}
\tag{2.14}
$$

In this loss function, the parameter λ_{coord} acts as a scaling factor that increases the emphasis on the accuracy of the bounding box predictions, while λ_{noobj} reduces the impact of errors from boxes that do not contain objects. The first two terms of the loss function correspond to the localisation loss, which computes the error between the predicted bounding box coordinates $(\hat{x}, \hat{y})$ and dimensions $(\hat{w}, \hat{h})$ and their corresponding ground truth values (x, y) and (w, h). These errors are calculated only for the bounding boxes that contain objects (indicated by the indicator function $\mathbf{1}_{ij}^{\text{obj}}$), meaning that the loss penalises errors only when an object is present in a grid cell. The third and fourth terms represent the confidence loss. The third term measures the confidence error for boxes containing objects (again indicated by $\mathbf{1}_{ij}^{\text{obj}}$), while the fourth term measures the confidence error for boxes that do not contain objects (indicated by $\mathbf{1}_{ij}^{\text{noobj}}$). As most boxes are empty, this portion of the loss is down-weighted by the λ_{noobj} factor. The final component of the loss function is the classification loss, which measures the squared error of the predicted class probabilities, but only for cells containing objects.

2.5.2 Evolution of YOLO Frameworks

This section will focus on the major advancements introduced across key YOLO versions, with a particular emphasis on YOLOv8. These improvements span various areas such as network design, loss function adjustments, anchor box adaptations, and input resolution scaling.

YOLOv2, released in 2017 [41], introduced several enhancements over YOLOv1. One of the most notable changes was the adoption of anchor boxes, similar to the SSD framework, which enabled the network to predict bounding boxes with varying aspect ratios and scales more effectively, addressing a limitation in YOLOv1 where the network directly predicted box coordinates, which was less flexible. Like YOLOv1, the model was initially pre-trained on ImageNet [23] with a resolution of 224 $\times$ 224. However, YOLOv2 introduced a new step by fine-tuning the model for ten epochs on ImageNet using a higher resolution of 448 $\times$ 448, which significantly improved the network's performance on high-resolution inputs. Architecturally, YOLOv2 replaced the fully connected layers used in YOLOv1 with a fully convolutional network, simplifying the design and improving efficiency. It also introduced batch normalisation across all convolutional layers, which not only accelerated the training process but also helped in regularising the model, thereby reducing overfitting. In terms of network design, YOLOv2 adopted the Darknet-19 backbone [41], which consists of 19 convolutional layers followed by 5 max-pooling layers. This was a shift from the GoogLeNet-inspired architecture used in YOLOv1. Darknet-19 was designed to be faster and more efficient, with fewer operations per layer, making it more suitable for real-time applications.

YOLOv3 [42] was in many ways similar to YOLOv2, using a similar backbone but introducing Darknet-53, a more powerful and efficient architecture with 53 convolutional layers. This deeper backbone allowed YOLOv3 to extract richer feature representations while maintaining speed. The network also incorporated a modified Spatial Pyramid Pooling (SPP) block that concatenates multiple max pooling outputs without subsampling (stride = 1), each with different kernel sizes (k $\times$ k where k = 1, 5, 9, 13), effectively expanding the model's receptive field. YOLOv3 was able to predict bounding boxes at three different scales, leveraging a FPN-like approach. By doing so, YOLOv3 significantly improved its performance on small objects—a notable weakness in previous versions. The YOLOv3 loss function remained largely unchanged from its predecessors. However, when compared to RetinaNet's Focal Loss, RetinaNet excelled in accuracy and handling class imbalance while YOLOv3 achieved comparable AP scores with significantly faster inference times.

YOLOv4 [43] and **YOLOv5** [44] were developed by different authors but followed the same philosophy of single shot, real-time, open source, and a darknet framework. However, they marked a significant shift in architecture, adhering to the design of modern object detectors with three main components: a backbone, a neck and head. The backbone CNN carries out feature extraction with lower-level features (e.g., edges) extracted in the earlier layers and higher-level features (e.g., object parts and semantic information) extracted in the deeper layers. The neck focuses on refining these features to obtain an spatial and semantic representation of the input image by including additional convolutional components like a FPN. Lastly, the head uses these representations to generate final predictions. Figure 2.5 illustrates the high-level design.

One notable architectural component of YOLOv5 is the Cross Stage Partial (CSP) Net. CSP-Net, a variant of the ResNet architecture, is designed to improve the learning efficiency of the network by dividing the feature map into two parts: one part passes through a dense block of

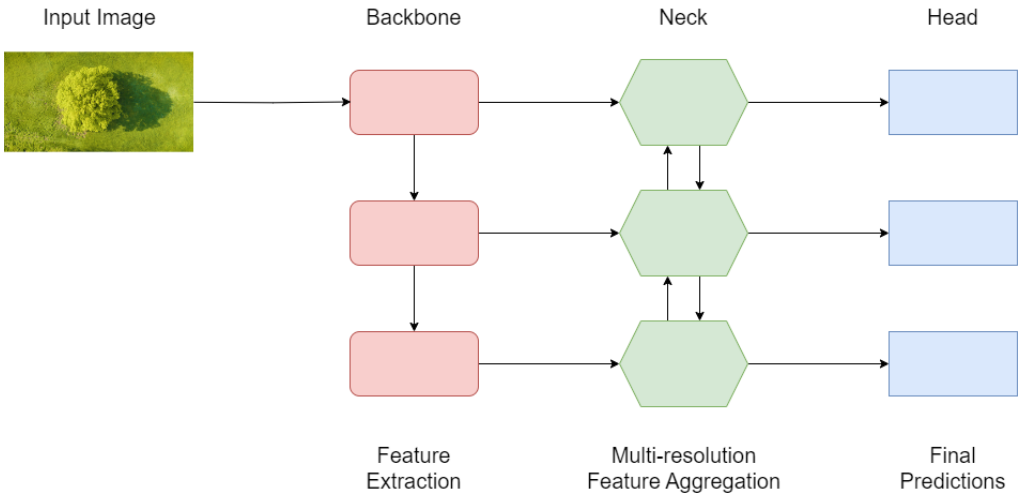


Figure 2.5: The modern object detection architecture comprising three main components: a backbone for feature extraction, a neck for refining features, and a head for generating final predictions.

convolutions, and the other is concatenated directly with the output of the dense block. This design not only reduces computational complexity but also maintains a higher level of accuracy than YOLOv4 and YOLOv3 [44]. The backbone includes SPP block to help with the feature extractions and the neck includes a Path Aggregation Network (PAN) that allows feature fusion and information flow across different scales of the network using upsampling and downsampling paths.

2.5.3 YOLOv8

Subsequent versions of YOLO continued to introduce iterations and changes to improve performance, including the incorporation of better backbones for enhanced feature extraction and modifications in the loss functions within the detection head. **YOLOv8** [45], released in 2023 by a company called Ultralytics, introduced significant changes, most notably the shift from anchor-based to anchor-free detection. As mentioned earlier, previous YOLO versions used anchor boxes which were predefined and used as reference points to predict object locations within an image. This method, while effective, introduced complexities related to the selection and optimisation of anchor box sizes, especially when dealing with objects that can come in many sizes. To mitigate this, YOLOv8 employs an anchor-free architecture with a decoupled head, allowing it to independently handle classification and regression tasks. This design allows each branch to focus on its task and improves the model's overall accuracy. In the output layer of YOLOv8, the sigmoid function is used as the activation function for the objectness score, indicating the probability that a bounding box contains an object, while the softmax function is utilised for class probabilities, representing the likelihood of the object belonging to each possible class. YOLOv8 continues to use a bigger version of the CSPDarknet backbone coupled with an improved PAN in the neck, which is responsible for feature fusion across different scales. A significant addition in YOLOv8 is the introduction of the C2f (Cross Stage Partial with two fusion) module, which is integrated into the CSPDarknet backbone. The C2f module builds upon the CSPNet concept in YOLOv5 by incorporating additional fusion stages. This module splits the feature map into several parts,

processes these parts through bottleneck layers, and then fuses them back together. The C2f module enhances the model's ability to extract and combine fine-grained features, leading to improved detection accuracy. A high-level overview of the architecture is shown in Figure 2.6. The YOLOv8 architecture will be explored in more detail in section 3.4.

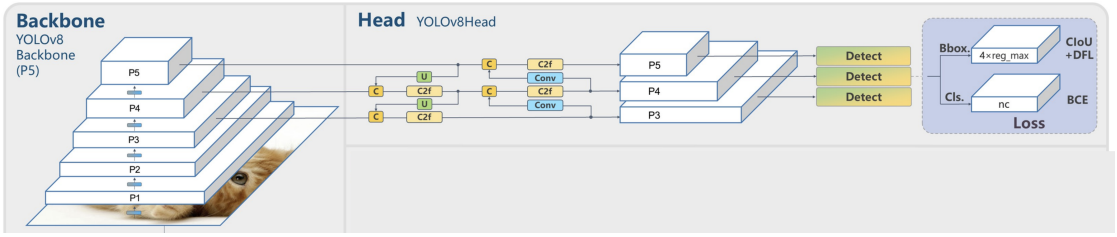


Figure 2.6: YOLOv8 High-Level Architecture. The architecture includes a backbone, a neck (not explicitly labelled), a head and the loss components [17]

Furthermore, YOLOv8 introduces advanced loss functions specifically designed for its anchor-free architecture. Among these, the Complete Intersection over Union (CIoU) [46] loss is employed for bounding box regression. This loss extends the capabilities of the localisation loss (Equation 2.9) by not only considering the overlap between predicted and ground truth boxes but also factoring in the distance between their centers, aspect ratio, and scale, thereby improving the precision of object localisation. In addition to CIoU, YOLOv8 includes a modification of RetinaNet's focal loss called the Distribution Focal Loss (DFL) [47]. DFL enhances the model's regression performance by focusing on harder-to-classify examples by improving its ability to accurately distinguish between different classes. The final loss function in YOLOv8 is a combination of the bounding box regression loss, which includes both CIoU and DFL, and the classification loss, which uses Binary Cross Entropy (BCE) to ensure accurate class predictions. The equations are as follows:

$$L(\mathbf{b}, \mathbf{b}^{gt}, p, \hat{p}) = \lambda_{\text{bbox}}(L_{\text{CIoU}}(\mathbf{b}, \mathbf{b}^{gt}) + L_{\text{DFL}}(y, \hat{p})) + \lambda_{\text{cls}} L_{\text{BCE}}(y, p) \tag{2.15}$$

$$L_{\text{CIoU}}(\mathbf{b}, \mathbf{b}^{gt}) = 1 - \text{IoU} + \frac{\rho^2(\mathbf{b}, \mathbf{b}^{gt})}{d^2} + \alpha v \tag{2.16}$$

$$L_{\text{BCE}}(y, p) = -\sum_{i=1}^{C} y_i \log(p_i) + (1 - y_i) \log(1 - p_i) \tag{2.17}$$

$$L_{\text{DFL}}(y, \hat{p}) = -\sum_{x,y} \sum_{c \in \text{classes}} y_{x,y,c} \log(\hat{p}_{x,y,c}) + (1 - y_{x,y,c}) \log(1 - \hat{p}_{x,y,c}) \tag{2.18}$$

where:

- $L_{\text{CIoU}}(\mathbf{b}, \mathbf{b}^{gt})$ and $L_{\text{DFL}}(y, \hat{p})$: The Complete Intersection over Union (CIoU) loss and the Distribution Focal Loss (DFL), both used for bounding box regression.
- $L_{\text{BCE}}(y, p)$: The Binary Cross Entropy loss, used for the classification task.
- $\lambda_{\text{bbox}}, \lambda_{\text{cls}}$: The weights associated with the bounding box regression and classification loss components, respectively, balancing their contributions to the total loss.

- $\mathbf{b}, \mathbf{b}^{gt}$: Represent the predicted and ground truth bounding boxes, respectively.
- ρ^2: The squared Euclidean distance between the central points of the predicted and ground truth bounding boxes.
- d^2: The diagonal length of the smallest enclosing box among the predicted and ground truth boxes.
- αv: A term that accounts for the consistency in aspect ratio between the predicted and ground truth boxes.
- y_i and p_i: The ground truth and predicted probabilities for class i, respectively.
- $\hat{p}_{x,y,c}$: The predicted probability for class c at pixel coordinates (x, y).

Chapter 3

Model Design and Experiments

This chapter begins by outlining the datasets used in this study, which include a combination of publicly available and manually annotated datasets. The chapter delves into how the datasets were curated, labeled and augmented using geometric, colour and geospatial augmentations. Inspired by traditional methods discussed in Section 2.1, we present the baseline model as an initial framework for tree count estimation without relying on deep learning. Subsequently, the chapter conducts a performance comparison of three state-of-the-art object detection architectures, namely SSD, RetinaNet, and YOLOv8 to identify the best model for the task of tree detection. Next, we improve detection accuracy by modifying the YOLO architecture to better suit our dataset's unique characteristics.

3.1 Datasets

3.1.1 Tree Dataset

The main dataset used to train the object detection models in this study is a combination of three separate datasets published by Roboflow Universe, an open source computer vision community. It consists of a total of 489 aerial images containing urban and suburban environments with varying densities of tree coverage [48–50]. These images span different geographical locations, showcasing a variety of urban landscapes, including residential areas, roadsides, and parks. Each dataset originally had its own respective split, but they were combined to create a unified dataset with 380 images for training, 59 images for validation, and 50 images for testing (approximately a 77-13-10 split). Figure 3.1 shows sample images of the dataset.

The dataset is annotated with bounding boxes identifying the locations of trees, with labels available in both YOLO and XML formats. The YOLO format is specifically designed for training the YOLO model, where each bounding box is represented by a label followed by normalised coordinates: the center of the box (x, y) and its dimensions ($width$, $height$). The XML format stores similar information but in a different structure. In XML, the bounding box is defined by the coordinates of the top-left corner ($xmin$, $ymin$) and the bottom-right corner ($xmax$, $ymax$). Despite these differences, both formats provide precise information on the location and dimensions of each tree within the images, enabling the model to learn and predict tree positions effectively.

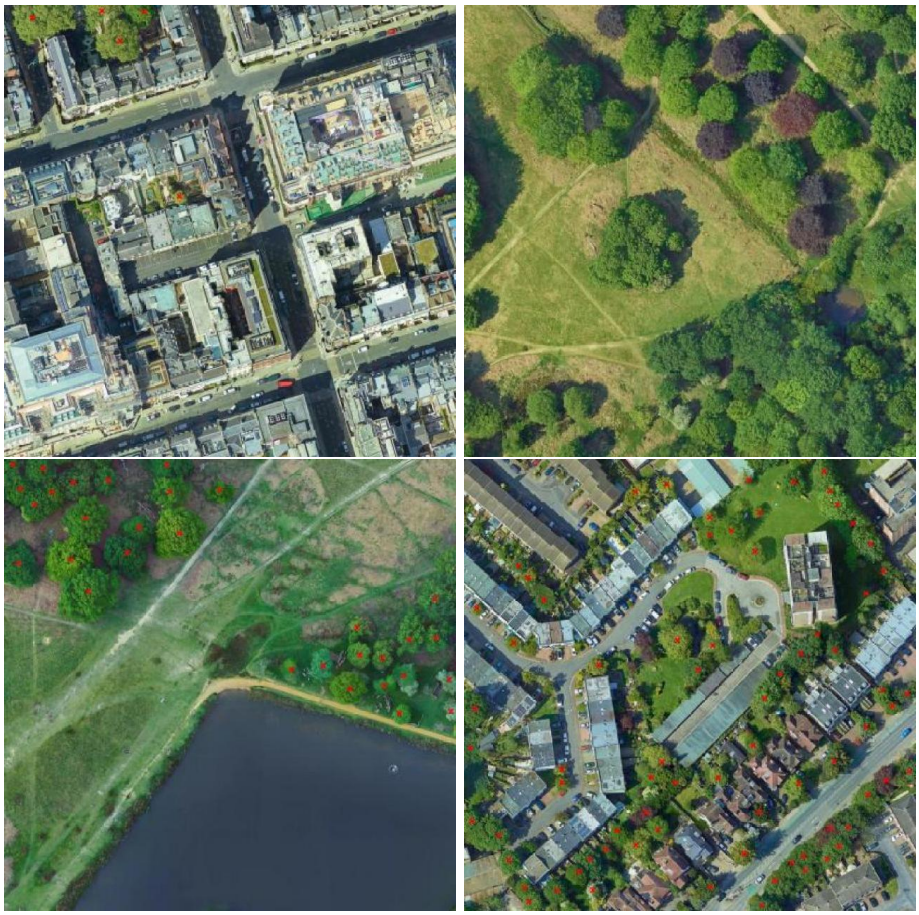


Figure 3.1: Sample images from the dataset used for training the object detection models.

Figure 3.2: Comparison between the YOLO and XML formats for bounding box annotations. Figure shows annotation for the bounding box specified by the arrow.

Tree Dataset Augmentation

To ensure enough variety in the dataset and reduce the risk of overfitting, we applied a set of geometric and colour transformations using Roboflow Augmentation Tool [51] to augment the training data. In this section, we outline these augmentations, parameter values and provide justification for their application to the dataset. The augmentation parameters, which range from 0.0 to 1.0 and represent the probability of each transformation being applied, were determined through a random hyperparameter search followed by manual tuning, as detailed in Appendix A. The final parameter values are stated below:

Table 3.1: Summary of the Main Augmentations Applied to the Tree Dataset

Argument	Type	Value	Description
hsv_h	Colour	0.015	Adjusts hue to simulate different lighting conditions and enhance color variability. Useful since trees can appear different depending on time of the day or season.
hsv_s	Colour	0.7	Alters saturation to reflect varying environmental conditions, improving model adaptability.
hsv_v	Colour	0.4	Modifies brightness to account for different light intensities, helping with shadowed or bright areas.
translate	Geometric	0.1	Apply slight shifting to ensure the model doesn't overfit to specific tree locations in the training set.
scale	Geometric	0.5	Rescales images to simulate trees at different distances, aiding size variation learning.
fliplr	Geometric	0.5	Horizontal flip helps increase orientation variability since aerial imagery can be taken from an angle.
mosaic	Geometric	1.0	Combines multiple images to create complex scenes, improving the model's ability to understand varied environments.

The mosaic augmentation, which combines multiple images to create a single complex image, was particularly created for aerial imagery applications [52], and was a key component in the improved performance of all models, hence we used a value of 1.0. The effectiveness of mosaic augmentation in aerial imagery is largely due to its ability to simulate different scene compositions and object interactions, which are common in overhead views where multiple trees are seen together in various arrangements. Figure 3.3 illustrates a simplified example of a mosaic augmentation. It should be noted that in practice, images used in mosaic augmentation are typically not perfectly aligned and combinations can include less or greater than four images. It is worth noting that mosaic augmentation can create discontinuities at the edges where different images are combined, but this did not degrade model performance. Additionally, several colour augmentations were considered during the hyperparameter search. For instance, mixup, which blends two images and their labels into one, was evaluated for its potential to increase variability. However, it was found to introduce too much label noise, leading to reduced model performance. Similarly, BGR, which flips the colour channels from RGB to BGR, was tested but did not prove beneficial for this task.

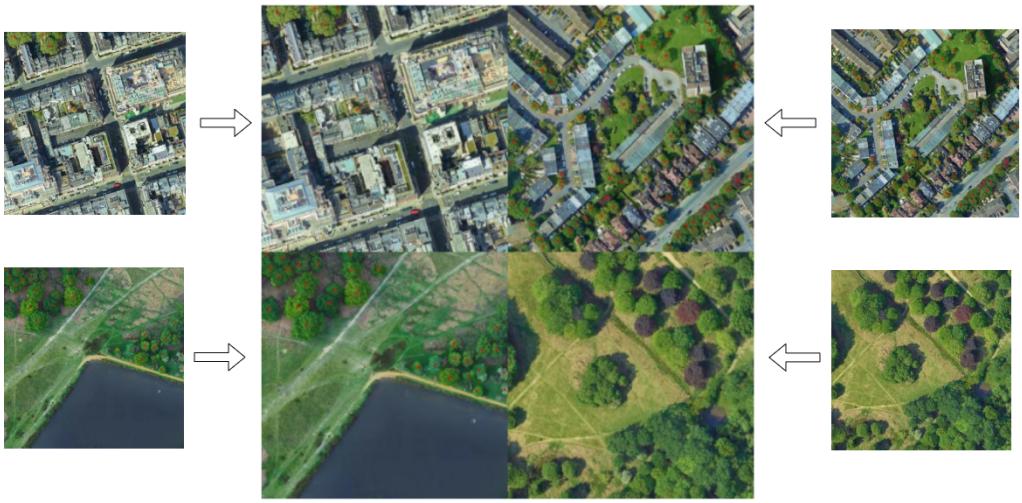


Figure 3.3: Simplified example of mosaic augmentation combining four different images into one, creating a richer and more complex training image with diverse tree arrangements.

3.1.2 Geospatial Dataset

While the aerial tree dataset contains images of dense trees in various urban landscapes, it has a few shortcomings. Firstly, the vast majority of images in the dataset feature hundreds of well-labeled trees, leading to a potential class imbalance. This imbalance could result in models that generalise poorly on images with fewer trees, which are common in certain urban areas. Secondly, the lack of documentation regarding the precise locations of the trees in the aerial dataset makes it impossible to cross-reference the model's predictions with the baseline model or other tree counting algorithms. This limitation hinders the ability to compare and contrast the performance of deep learning models with non-deep learning methods. Thirdly, the aerial tree dataset is limited to RGB bands, while many satellite images provide additional bands, such as near-infrared (NIR), which can be crucial for accurately identifying and analysing vegetation. This lack of spectral diversity in the training data could potentially limit performance on complex satellite imagery acquired from other sources.

To address these limitations, we created and labeled a geospatial satellite dataset consisting of 70 GeoTIFF satellite images exported directly from Google Earth Engine. GeoTIFF is a file format that includes metadata that allows georeferencing information such as coordinate data to be embedded directly within the image files [53]. The dataset was split into 70% training, 20% validation and 10% testing images. For the labeling process, we utilised the Roboflow platform [51], a comprehensive tool for computer vision dataset management and annotation. Since GeoTIFF files are not compatible with Roboflow, we first converted each image into PNG format before uploading the dataset to the Roboflow platform. Using Roboflow's web-based interface, we manually annotated each image by drawing bounding boxes around individual trees and assigning them the class label "tree". After completing the annotation process, we used Roboflow's export functionality to export the dataset in YOLO and XML formats. Figure 3.4 shows an example.

Figure 3.4: Example of tree labeling in the geospatial dataset using the Roboflow tool. The yellow bounding boxes indicate the locations of individual trees, which were annotated to create the dataset.

These images cover regions from various countries and include edge cases such as shadows, small trees, overlapping canopies, and images with additional spectral bands (e.g., NIR). This variation was introduced by utilising geospatial metadata of the exported regions. Table 3.2 summarises the composition of the geospatial dataset.

Table 3.2: Summary of Geospatial Dataset Composition

Country	Number of Images	Seasonal Aug.	Spectral Aug.	Image Resolution	Year
Latvia	14	No	Yes	20cm	2018
Netherlands	32	Yes	No	7.50cm	2021/2022
Spain	10	No	No	10cm	2019
Switzerland	14	No	No	10cm	2020

Geospatial Dataset Augmentation

The augmentation for this dataset came from being able to utilise the geospatial metadata of the exported regions to introduce variety on the dataset. For instance, trees can look different depending on the time of year — they may appear yellow in the fall, green in the spring, or sparse with fewer leaves in the winter. To account for these seasonal variations, we implemented a "seasonal augmentation" strategy by selecting images of the same areas captured at different times of the year using the available metadata. This enabled the model to learn and adapt to the natural seasonal changes in tree appearance. Additionally, we incorporated Near-Infrared (NIR) bands available for the Latvia regions to further enrich the dataset, providing additional spectral information. Figure 3.5 shows sample images from the geospatial dataset that illustrate the seasonal and spectral augmentation present in the dataset.

Figure 3.5: Illustration of Seasonal and Spectral Augmentation in the Geospatial Dataset. The image shows the diversity introduced in the dataset: the top two images represents Seasonal Augmentation with images captured at different times of the year, while the bottom two images demonstrates Spectral Augmentation by comparing RGB and Near-Infrared (NIR) band combinations.

3.2 Baseline

Precise tree counting can be a very resource-intensive task, often requiring detailed field surveys and high spatial-resolution satellite imagery. As a preliminary step, this paper proposes a baseline method to provide a rough estimate of tree count within any region of interest using publicly available low-resolution satellite imagery. Specifically, we use NASA's Global Forest Cover Change (GFCC) dataset [54], which provides global tree canopy cover at a 30-meter resolution. The dataset contains a band named 'tree-canopy-cover' that indicates the percentage of canopy cover per pixel.

To define the baseline approach, the dataset was first loaded and filtered in google earth engine to select images from 2015. The tree canopy cover band was extracted so that each pixel would indicate the percentage of tree canopy cover within the 30-meter resolution. Next, the region of interest (ROI) was constructed by drawing a square of size 50mx50m to allow for preliminary tree

count analysis. The estimated tree count is calculated using the following equation:

$$\text{Estimated Tree Count} = \left(\frac{\text{Area of ROI}}{\text{Average Tree Canopy Area}} \right) \times \text{ROI Canopy Cover Proportion} \quad (3.1)$$

An average tree canopy area of 60m^2 was chosen for this method, based on a study that reported a similar mean canopy area value [55]. The canopy cover data from the band was then used to compute the mean canopy cover of the ROI, which was subsequently converted into a canopy cover proportion by dividing by 100. This facilitated the final calculation of the estimated tree count within the defined ROI.

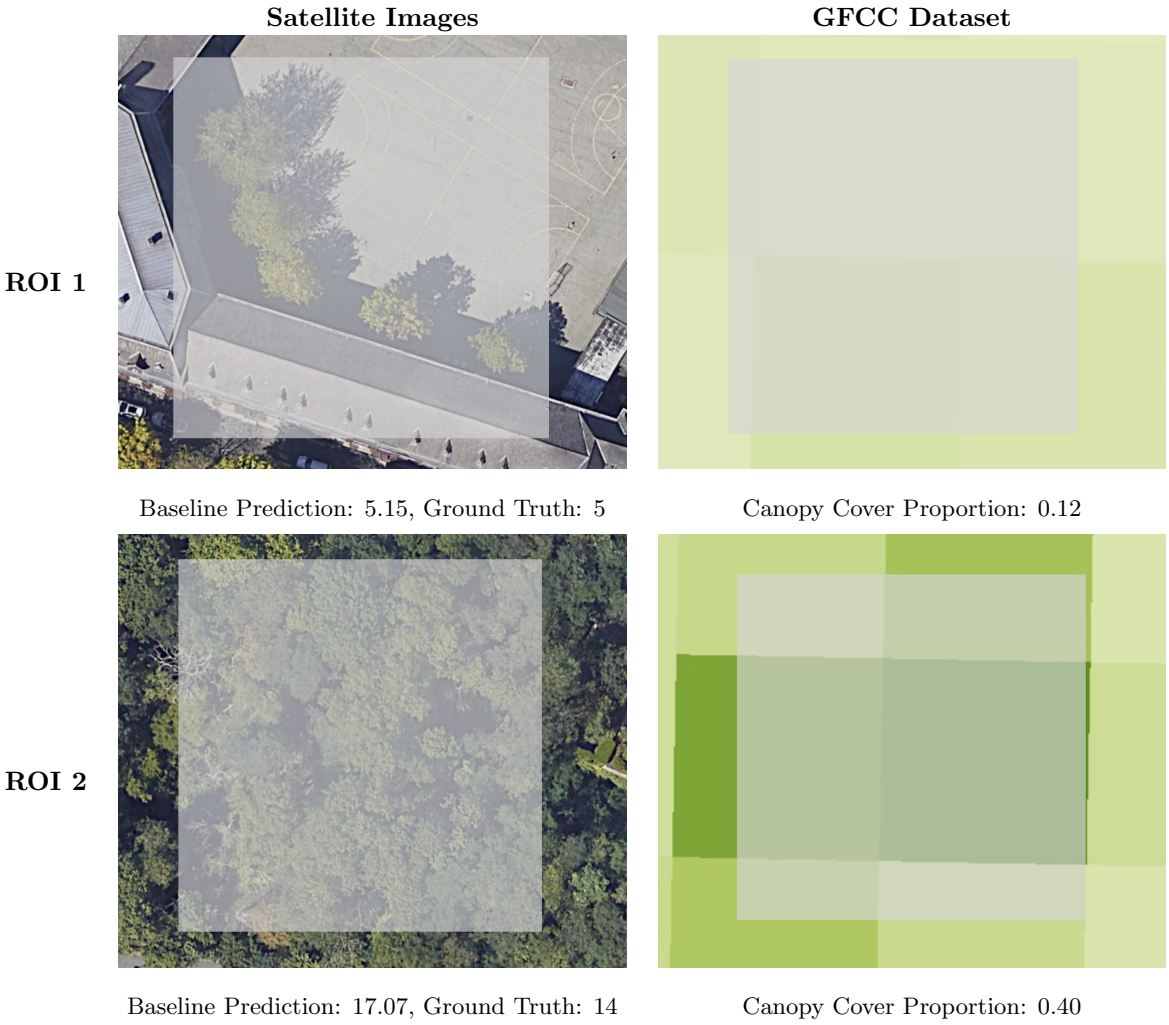


Figure 3.6: Baseline model predictions on sparse and dense Regions of Interest (ROI). The figures show satellite imagery and corresponding GFCC dataset overlays for the two regions. Baseline predictions and ground truth tree counts are provided along with the canopy cover proportion.

Figure 3.6 shows the baseline method applied to two distinct urban regions: ROI 1, with a sparse tree count of 5, and ROI 2, with a denser cover of 14 trees. The method accurately estimated 5.15 trees for ROI 1 and 17.07 trees for ROI 2, reflecting the higher canopy cover in denser areas and showing good estimations, particularly in regions with sparse tree cover. We ensured this

accuracy of the tree count estimates by carefully aligning the ROI with the grid of the satellite imagery dataset. Specifically, we adjusted the ROI to match the grid resolution of the dataset, which has a 30-meter resolution. This alignment was achieved by snapping the ROI boundaries to the grid, ensuring that the entire area was accurately represented within the grid cells of the dataset. By doing so, we avoided the common issue of partial pixel inclusion, where some pixels might only be partially within the ROI, leading to inaccuracies in the canopy cover calculation. One of the limitations of this approach is fixing the tree canopy area, which does not account for the variability in tree sizes across different regions and species. This, among other limitations, will be discussed in the next sections. Nevertheless, the baseline method offers a simple, efficient approach to obtain rough estimates of tree count in any region without requiring extensive computational resources or complex algorithms. This helped establish performance benchmarks, an important first step in the project. It is important to note that this method can only infer tree count when precise coordinates of tree locations and regions are available. Therefore, the comparison between this baseline method and deep learning models will only be conducted after the deep learning models have been trained on the geospatial dataset. This comparison will provide insights into how well deep learning approaches perform relative to non-deep learning approaches.

3.3 Object Detection Model Training

The purpose of this section is to identify the best model architecture for the task of tree counting. In this experiment, we train and test three object detection models: SSD, RetinaNet and YOLOv8 on the tree dataset. First, we look into how we set up a virtual environment for model training on the cloud. Second, we explain the configuration and training processes for the three models.

3.3.1 Google Cloud Platform

Training deep learning models and algorithms requires extensive computational resources, both in terms of processing power and memory. This is due to the large number of parameters the models need to learn and update during the training process. To meet these demands, we use Google Cloud Platform (GCP) to provide the necessary infrastructure for model training. A virtual environment was set up on GCP using a high-performance virtual machine (VM) equipped with a GPU. Specifically, the VM was configured with the n1-standard-4 machine type, which includes 4 virtual CPUs and 15 GB of RAM. Additionally, one NVIDIA T4 GPU was attached to the VM, significantly accelerating the training process. Using Python's virtual environment tools, dependencies specific to the three models (SSD, RetinaNet, and YOLOv8) were managed effectively. Required libraries and frameworks such as Pytorch and OpenCV were installed and configured within the VM. For efficient data management, a Google Cloud Storage bucket was set up to store and access all datasets during model training. The geospatial dataset, consisting of 70 GeoTIFF satellite images, was first exported to this bucket from Google Earth Engine before being downloaded onto the VM and converted to PNG format. Similarly, the tree dataset, sourced from Roboflow, was uploaded to the Google Cloud Storage bucket. From there, the images were downloaded into the VM's environment. All the computational resources and infrastructure for this project were kindly provided and supported by QBE.

3.3.2 SSD

For this experiment, we used the SSD300 architecture, which was initially designed to operate on 300x300 pixel input images. This architecture is paired with a VGG16 backbone, a well-established model known for its 16 convolutional layers, which acts as the feature extractor. The SSD300 model was initialised using pre-trained weights from the COCO dataset [56] to leverage transfer learning, which is beneficial given the relatively small size of the tree dataset. These weights were loaded into the VGG16 backbone, which was fine-tuned end-to-end to adapt to the dataset. The model's anchor boxes were configured to cover a range of aspect ratios and scales, ensuring that the model could detect trees of various sizes. In terms of preprocessing, the input images of trees were resized to 640x640 pixels to ensure consistency with the input requirements of the YOLO and RetinaNet models. Using a bigger image size also helped better capture the spatial features of trees at different resolutions. The images were then normalised to scale pixel values between 0 and 1, which helps stabilise the training process by ensuring that the inputs are within a standard range and reducing the likelihood of exploding gradients during training.

The training process was carried out using a stochastic gradient descent (SGD) optimizer with a learning rate of 0.0001, momentum of 0.9, and Nesterov acceleration enabled. This configuration helped in speeding up model convergence. The learning rate was further adjusted using a StepLR scheduler, which decreased the learning rate by a factor of 0.1 every 15 epochs. A batch size of 4 was used to balance efficient memory utilisation with sufficient gradient updates per iteration. The model was trained for 100 epochs, during which the training loss and validation metrics were monitored. A custom class was employed to keep track of the average training loss per epoch, providing a clear indication of the model's learning curve. The validation performance was assessed using the Mean Average Precision (mAP) metric. Early stopping was implemented with a patience threshold of 10 epochs, meaning that if the model did not show improvement for 10 consecutive epochs, the training process was automatically terminated. The weights of the best model were saved based on the highest validation mAP achieved during training. A seed of 42 was used for reproducibility. The training process was visualized using progress bars to track the loss and mAP in real-time.

3.3.3 RetinaNet

The configuration of the RetinaNet and SSD models is almost identical. For both models, the number of classes was set to 2, corresponding to the 'background' and 'tree' classes. To address the foreground-background class problem, we employ the focal loss function to down-weight the contribution of dominant background class. It was hypothesised that this would help the model learn better representations for the minority tree class, especially in images with less than 100 trees.

Similar to the SSD model, we initialised the RetinaNet model with pre-trained weights from the COCO dataset using Pytorch. These weights were loaded into a ResNet50 backbone with a FPN to leverage transfer learning, ensuring that the model starts with a strong baseline of feature extraction capabilities. When inspecting the model architecture source code, we found that RetinaNet uses 9 anchor boxes per feature map location, compared to SSD's 6. This means that RetinaNet evaluates more possible bounding boxes for each spatial location in the image,

potentially leading to better detection performance. These anchors help the model propose regions of interest that are more likely to contain objects, which are then refined during the training update. The model was then fine-tuned end-to-end on the tree dataset using the same learning rate, momentum and optimiser as SSD before training for 100 epochs with early stopping. The full model architectures of SSD and RetinaNet are available in the appendix.

3.3.4 YOLOv8

YOLOv8 was developed and released by Ultralytics [45], and it comes with several variants listed in Table 3.3. These models share the same architecture but differ in the number of three key parameters: depth multiple (d), width multiple (w) and the maximum channels (mc). The value of d scales the depth of the network, dictating how many layers the network has. The value of w scales the width of the network, dictating the number of channels in each layer and mc is the maximum number of channels used by the network. After testing all variants, it was found that the difference in accuracy via mAP is not significant but the difference in training time was substantial. As a result, the YOLOv8s was chosen as it offered the best trade-off between accuracy and computational efficiency for tree detection. The architecture for this model is available in the appendix.

Model variant	**d (depth_multiple)**	**w (width_multiple)**	**mc (max_channels)**
n	0.33	0.25	1024
s	0.33	0.50	1024
m	0.67	0.75	768
l	1.00	1.00	512
xl	1.00	1.25	512

Table 3.3: YOLOv8 Model Variants

The YOLOv8s model was initialised using pre-trained weights from the Ultralytics repository. These weights were downloaded directly within the training script and were also trained using the COCO dataset. The model was trained for 100 epochs on the tree dataset, with a batch size of 4 and an image size of 640x640 pixels, consistent with the configurations used for SSD and RetinaNet. However, unlike the other models, YOLOv8 uses a AdamW optimiser [57] which employs adaptive learning rates and weight decay to stabilise the learning process. The initial learning rate was set to 0.002, with a momentum of 0.9 and a weight decay of 0.0005. Instead of using a scheduler, the AdamW optimiser takes care of dynamically adjusting the learning rate during training.

It was noted during the initial phase of training that certain data augmentations and hyper-parameter configurations were more effective for some architectures than others. Given that the purpose of this experiment is to identify the best model for the task of tree counting and detection, no data augmentation techniques were used. For YOLOv8, this was achieved by simply setting 'augment=False' in the training configuration. The final chosen parameters and architectural details are summarised in Table 3.4.

Table 3.4: Architecture and Hyperparameters Comparison

Method	SSD300	RetinaNet	YOLOv8s
Input network resolution	640 × 640	640 × 640	640 × 640
Number of layers	29	161	225
Number of parameters	24,037,928	34,738,573	11,135,987
Backbone	VGG16	ResNet50 + FPN	CSPDarknet53
Anchor boxes	6 per feature map	9 per feature map	None
Optimizer	SGD	SGD	AdamW
Learning rate	0.0001	0.0001	0.002
Momentum	0.9	0.9	0.9
Weight decay	0.0005	0.0005	0.0005
Batch size	4	4	4
Scheduler	StepLR (0.1 every 15 epochs)	StepLR (0.1 every 15 epochs)	None

3.4 Model Architecture Modifications

Section 2.5.3 provided a high-level overview of the YOLOv8 architecture, emphasising the transition to an anchor-free detection mechanism and the improvements introduced by it's backbone, neck and head modules. In this section, we present the YOLOv8's architecture in a bit more detail to provide a deeper understanding of why it performs well and also set the stage for the architectural modifications employed to improve the accuracy of the model. Inspired from Terven et. al. [38], we present the core modules of the YOLOv8 architecture in Figure 3.7.

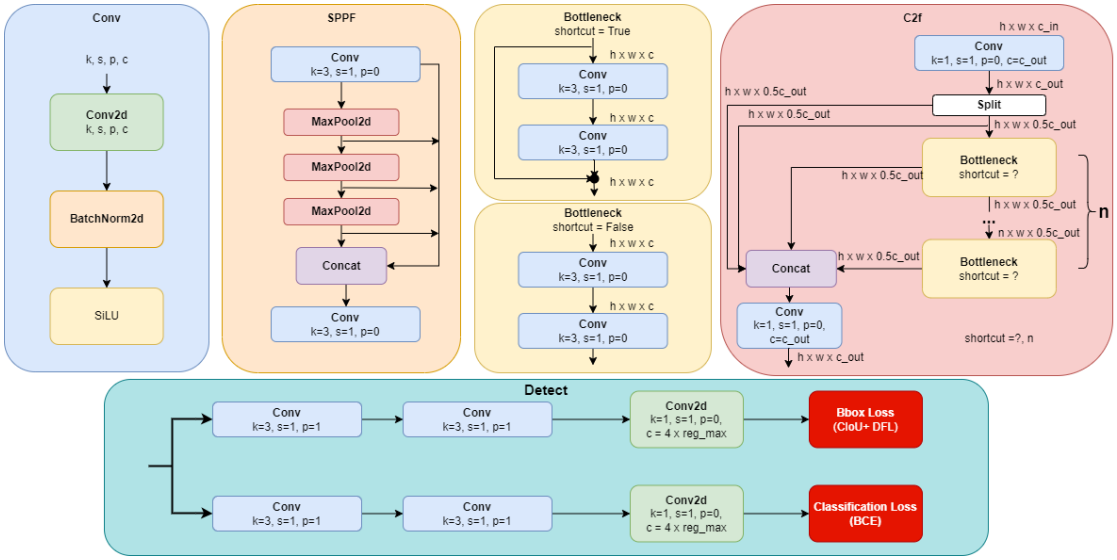


Figure 3.7: YOLOv8 Architecture Modules. The figure illustrates the core modules of the YOLOv8 architecture. The Conv module performs the initial feature extraction, followed by the SPPF module which enhances feature representation. The C2f module further refines features before passing them to the detection head, where bounding boxes and class predictions are made. Note: 'h x w x c' refers to the multiplication of height, width and channels, respectively.

The Conv module is the initial feature extraction block that consists of a convolution layer followed by batch normalisation and a SiLU activation function. The symbols 'k, s, p, c' refer to the kernel size, stride, padding and the channels of the 2d convolutional filter. The role of the Conv

is to capture low-level features such as edges from the input image. The SPPF (Spatial Pyramid Pooling) module applies multiple max pooling operations to the same input, concatenates the results, and then processes them through another convolution layer. This enhances the feature representation by pooling features at different scales. The bottleneck module consists of two configurations: with and without shortcuts (residual connections). The bottleneck layers help in reducing the number of parameters by applying convolutions while maintaining the number of channels of the network. As mentioned in Section 2.5.3, the C2f module helps in capturing the fine-grained features as it divides the feature maps into several parts, processes these parts through bottleneck layers, and then fuses them back together. The detect module is where the predictions are made. It processes the refined features from the C2f modules in the neck and outputs bounding boxes and class scores. These scores are optimised using the CIoU, DFL and BCE loss functions. Now, we show how these modules fit together in the Figure 3.8.

The 640x640 input image is passed on the Conv modules which downsample the spatial dimensions while doubling the number of channels, a pattern that is repeated across the architecture. This process of downsampling and increasing channels continues until the feature maps reach the maximum number of channels (mc) specified for the YOLOv8s variant, which is 1024. The backbone is shown in the left hand side of the figure, represented by modules 0-8. To reiterate, the C2f modules in the backbone split the feature maps from the input into multiple parts, process them independently through bottleneck layers, and then merge them back together. These processed feature maps are passed through the SPPF module and to the neck, shown by modules 10-21. The neck consists of additional C2f modules, upsampling operations, and concatenation layers that combine high-level (low-resolution) features with low-level (high-resolution) features. This fusion of features from different scales is critical for detecting objects of various sizes within the image. The upsampling steps in the neck (modules 10 and 13) restore the resolution of the feature maps, allowing the network to maintain spatial context that is essential for accurate localisation. Finally, the refined features reach the head of the architecture, where the detection modules are located (modules 22-24). These modules are responsible for predicting bounding boxes, confidence scores, and class probabilities. We notice that there are three detect modules. According to Reis et. al. [17], each detect module focuses on predicting different feature scales. Specifically, module 24 processes feature maps with dimensions of 80x80x256, which retains a higher spatial resolution. The larger spatial dimensions provide more detailed information about the position and boundaries of small objects, making this module particularly adept at detecting them. Conversely, module 23 and module 22 process progressively downsampled feature maps (40x40x512 and 20x20x1024, respectively) thus they are used to detect medium-sized and big objects, respectively. Given that the tree dataset contains multiple small trees, we propose an architecture modification of removing all the modules and layers relating to blocks 22 and 23, shown in Figure 3.9. This approach helped in simplifying the network by reducing the number of operations in the neck and head, and most importantly, it helped retain more high-resolution features throughout the network. We call this model **YOLOv8-small**. By maintaining higher-resolution feature maps and reducing the network's complexity, it was hypothesised that these changes would allow the model to focus on detecting small trees more effectively since the network would concentrate on the specific scale of objects present in the aerial image, rather than diluting its capacity by predicting objects at scales irrelevant to the dataset. This modification was done by downloading the 'YOLOv8.yaml'

from the Ultralytics repository [58]. This file includes all the core modules connected together in one script and can be modified easily.

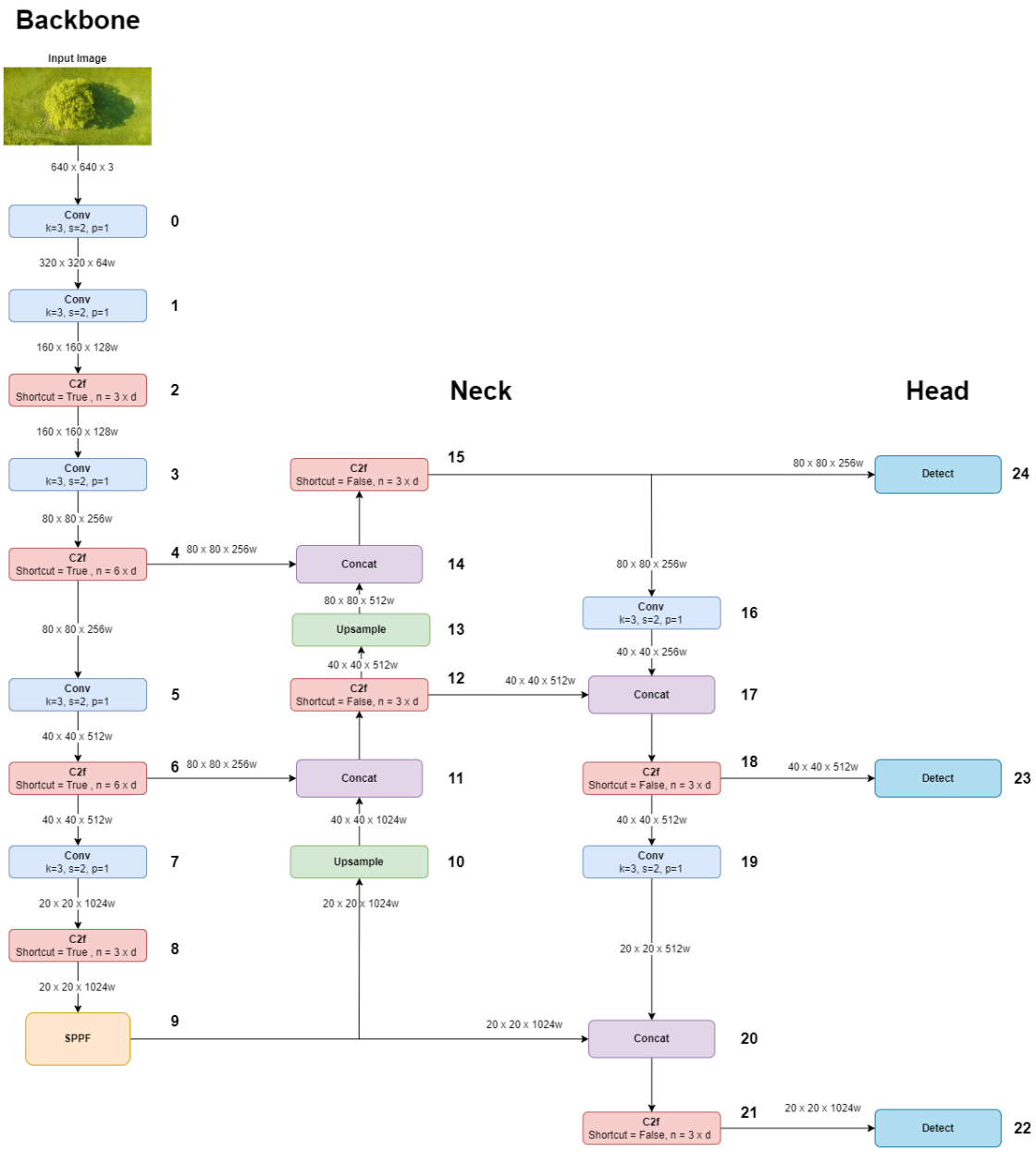


Figure 3.8: YOLOv8 Full Architecture: The figure illustrates the detailed flow of computation in the YOLOv8 architecture, starting from the backbone for feature extraction, through the neck for feature fusion, and finally to the head for predictions. Each module is carefully designed to downsample the input image, extract features, and refine them for accurate object detection. Diagram inspired by [38].

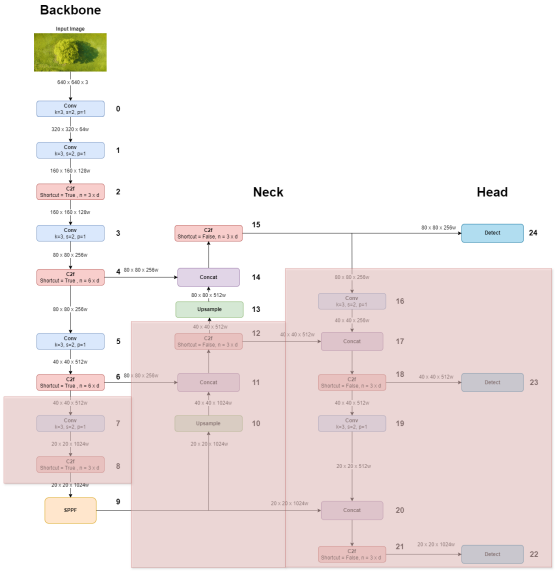


Figure 3.9: Full YOLOv8 architecture with removed modules highlighted in red. The red blocks indicate the layers and modules that have been removed in the modified version.

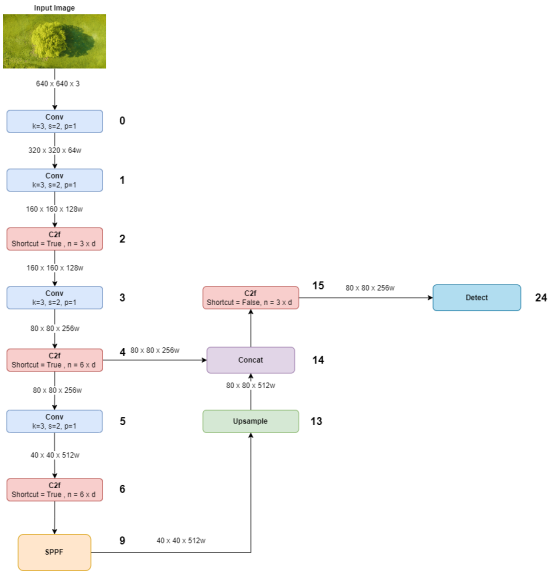


Figure 3.10: YOLOv8-small Architecture. The final modified architecture after removing the redundant modules for small object detection, optimising the model for the tree dataset.

The table below summarises the difference in architecture between the original YOLOv8 and the YOLOv8-small model. The modified model has fewer modules, layers and parameters.

Table 3.5: Comparison of Original and Modified YOLOv8 Architectures

Model	Original YOLOv8	YOLOv8-small
Number of Layers	225	120
Number of Parameters	11,135,987	8,880,017
Conv Modules	6	4
C2f Modules	9	4
SPPF Module	1	1
Upsample Operations	2	1
Concat Layers	4	1
Detect Modules	3	1

3.5 Geospatial Training

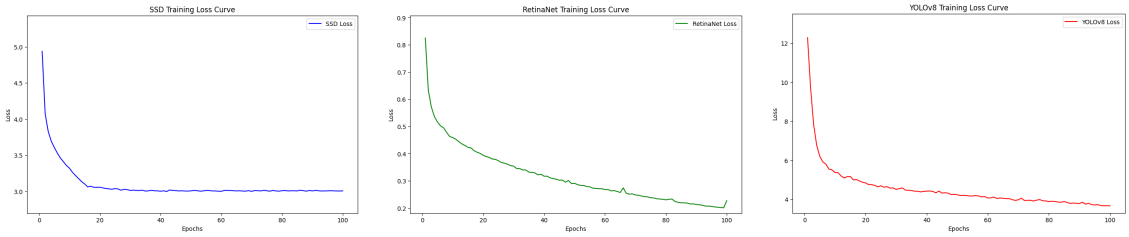
To thoroughly evaluate model performance, we train SSD, RetinaNet, YOLOv8, and the modified YOLOv8-small on the geospatial dataset, alongside the baseline method. This dataset serves as a method to assess how well the models generalise to satellite imagery and also highlight the discrepancies in performance between the deep learning models and the baseline. We start by training a YOLOv8s base model—one that had not been pre-trained on the tree dataset. This allowed us to isolate and observe the impact of using the tree dataset as a form of pre-training. Next, we trained all the aforementioned models on the geospatial dataset using identical training configurations, including the same learning rates, optimisers, and number of epochs, as previously applied in the tree dataset experiment.

Chapter 4

Results and Evaluation

4.1 Model Training on Tree Dataset

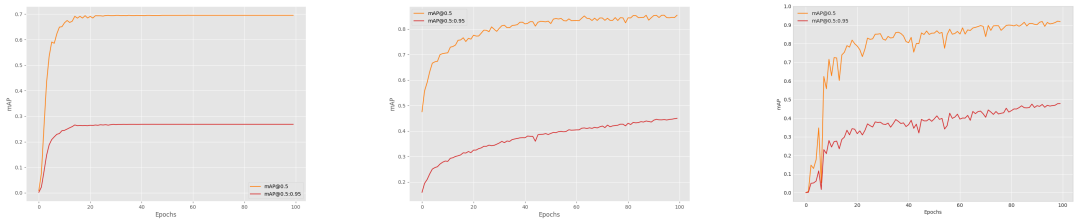
The primary objective of the first experiment was to identify the most effective model architecture for tree detection, based on the configurations outlined in Table 3.4. The training progression of each model—SSD, RetinaNet, and YOLOv8—was carefully monitored by examining both the training loss curves and the validation metrics. For the validation metrics, we use the mAP metric at different IoU thresholds, specifically 0.50 (mAP@0.5) and 0.95 (mAP@0.5:0.95). This provided a detailed view of how each model's detection capabilities evolved after every epoch. The figure below shows the training loss curves for each model:



(a) SSD Training Loss Curve **(b)** RetinaNet Training Loss Curve **(c)** YOLOv8 Training Loss Curve

Figure 4.1: Training Loss Curves for SSD, RetinaNet, and YOLOv8 Models

Each training loss curve shows an expected trend represented by a sharp loss decline in the first 10 epochs, followed by a gradual tapering off as the models approach convergence. It is apparent that SSD plateaus much quicker compared to the other models. This suggests that while SSD is effective at quickly capturing initial patterns, the model is not learning much after 20 epochs. In contrast, RetinaNet and YOLOv8 losses continues to refine its learning over more epochs, which could indicate better generalisation. The fact that their losses are still decreasing suggests that further training could yield even better results. However, the aim of this experiment is not to achieve the highest possible accuracy but to determine the best model architecture for the task. The YOLOv8 loss starts from a relatively higher value because it is a combination of the DFL, CIoU, and BCE losses. Next, we observe how the validation mAP metrics evolved with training time.



(a) SSD mAP Curves **(b)** RetinaNet mAP Curves **(c)** YOLOv8 mAP Curves

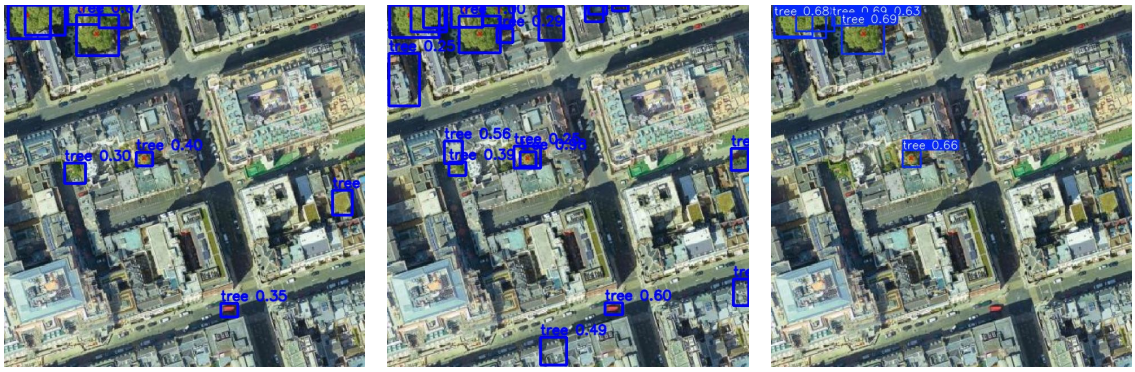
Figure 4.2: Validation mAP@0.5 and mAP@0.5:0.95 Curves for SSD, RetinaNet, and YOLOv8 Models

As expected, the SSD mAP validation does not show significant improvement after 20 epochs, aligning with the observed plateau in the loss function. The final mAP@0.50 stabilises around 0.69, while the mAP@0.50:0.95 levels off at approximately 0.27. This indicates that SSD quickly reaches its capacity to generalise on the validation set, with limited learning beyond the initial stages of training. RetinaNet and YOLOv8 follow a similar trajectory but display different patterns in their fluctuations. RetinaNet demonstrates more stable learning, as indicated by the steady increase in both mAP validation curves, achieving an mAP@0.50 of 0.86 and an mAP@0.50:0.95 of 0.45. RetinaNet's stable learning can be attributed to its robust feature extraction through the ResNet50 backbone, which is known for its reliability. On the other hand, YOLOv8 demonstrates less stability as seen by the erratic fluctuations in the first 20 epochs. These fluctuations are likely due to the challenging balance between the multiple loss functions (DFL, CIoU, BCE) that are optimised using the AdamW optimiser after every iteration. Despite this, YOLOv8 demonstrates impressive performance, reaching a validation mAP@0.50 of 0.92, which corresponds to 92% accuracy after 100 epochs. Notably, YOLOv8's performance stabilises after epoch 80, suggesting that the model could achieve even greater consistency after further training. While these metrics offer valuable insights into each model's detection capabilities, it's essential to evaluate their performance on test data, which represents unseen examples that provide a more accurate reflection of how the model will perform in real-world scenarios. We test each model on the test set and calculate the MAE on all 50 test images. The MAE provided a quantitative measure of the models' prediction accuracy, as it indicates how closely the predicted number of trees aligns with the ground truth. Before computing the MAE, a visual analysis of detection inferences on unseen data is essential to gain a comprehensive understanding of model performance.

Figure 4.3 illustrates the models' predictions on two test images: one featuring a sparse distribution of trees and another with a dense tree population, highlighting each model's ability to handle images with varying tree density. Each bounding box in the figure is accompanied by a probability score, representing the confidence level of the detection. A detection threshold of 0.25 was applied, meaning any detection with a probability below 25% was discarded to reduce false positives. In the sparse image with 7 trees, YOLOv8 accurately identifies all 7 trees without any false positives. This suggests that YOLOv8's object detection capabilities are effective in less cluttered environments. RetinaNet overestimates the number of trees, detecting 20 objects, many of which are false positives. Notably, both SSD and RetinaNet mistakenly identify a red car in the bottom center of the image as a tree. This likely occurs because most of the tree centers in the image dataset were labeled with a red 'x' to mark their locations, which may have inad-

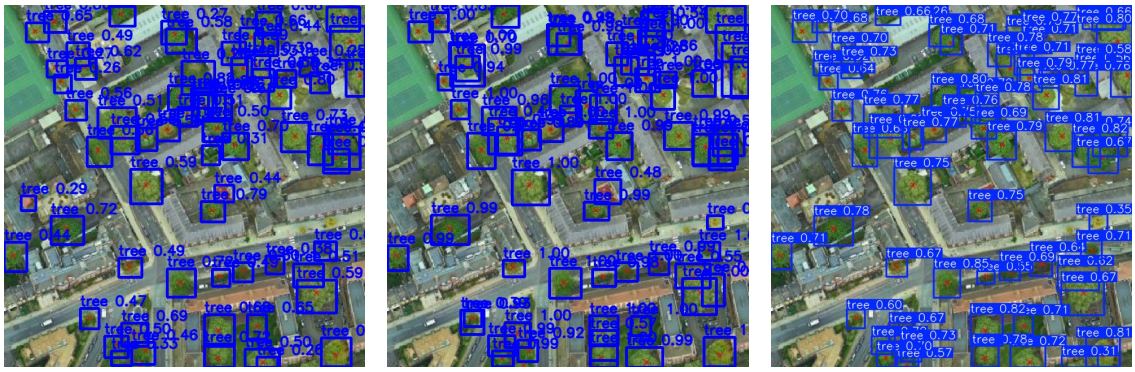
vertently trained the SSD and RetinaNet models to associate the red colour with tree centers, leading to incorrect detections. YOLO, which does not use anchor boxes, avoids this issue by directly estimating object locations, making it less prone to such visual cues like the red 'x'. The predictions on the dense image were more accurate across all models, with each showing improved performance compared to the sparse image scenario. All models successfully avoid falsely identifying a green field in the left part of the dense image as containing trees, indicating that they can distinguish between trees and other green objects. Surprisingly, SSD outperformed RetinaNet in both the sparse and dense image scenarios, despite achieving a lower mAP on the validation set. This suggests that RetinaNet might have overfitted to the training data, leading to worse generalisation on the unseen test data.

Sparse Image with Ground Truth: 7 trees



(a) SSD Prediction: 9 **(b)** RetinaNet Prediction: 20 **(c)** YOLOv8 Prediction: 7

Dense Image with Ground Truth: 48 trees



(d) SSD Prediction: 53 **(e)** RetinaNet Prediction: 59 **(f)** YOLOv8 Prediction: 52

Figure 4.3: Comparison of model predictions on two test images: a sparse image with a ground truth of 7 trees and a dense image with a ground truth of 48 trees.

Next, we compute the MAE on all 50 test images. YOLOv8 achieves the lowest MAE of 5.44, outperforming both SSD and RetinaNet, which have MAEs of 9.98 and 13.18, respectively. The predictions for each image in the test set are detailed in Appendix C.1. It is noteworthy that YOLOv8 not only delivers the best performance but also completes training in the shortest amount of time, despite having more layers. The training time for YOLOv8 is just 0.233 hours (14 minutes), significantly faster than the other models. The final results of this experiment are summarised in Table 4.5.

Table 4.1: Summary of Model Performance on Tree Detection Task

Model	mAP@0.5	mAP@0.5:0.95	Training Time (hrs)	Inference Speed (ms/img)	Final Test MAE
SSD	0.69	0.27	1.38	18.40	9.98
RetinaNet	0.86	0.45	2.65	70.90	13.18
YOLOv8	0.92	0.48	0.23	4.30	5.44

Given that the YOLOv8 model outperforms SSD and RetinaNet across several critical metrics—including mAP accuracy, training and inference speed, and Mean Absolute Error (MAE)—we selected the YOLOv8 architecture as the primary framework for this study. This strong performance motivated the decision to focus on modifying and optimising the architecture further, aiming to enhance its capability in detecting trees with greater accuracy and efficiency.

4.2 YOLOv8 Architecture Evaluation

In Section 3.4, we introduced YOLOv8-small and outlined the modifications made to the YOLOv8 architecture, specifically the removal of certain layers designed for detecting larger objects, which are not relevant to the tree dataset. By focusing on the layers used to detect small objects, such as trees in large satellite images, the YOLOv8-small model offers a more compact structure that retains high-resolution features and potentially makes more accurate predictions. These modifications reduced the model from 225 to 120 layers and decreased the number of parameters from approximately 11 million to 8.8 million (see Table 3.5). In this section, we compare the performance of the modified YOLOv8-small model with the original YOLOv8 by analysing the loss components during training and various evaluation metrics during validation and testing.

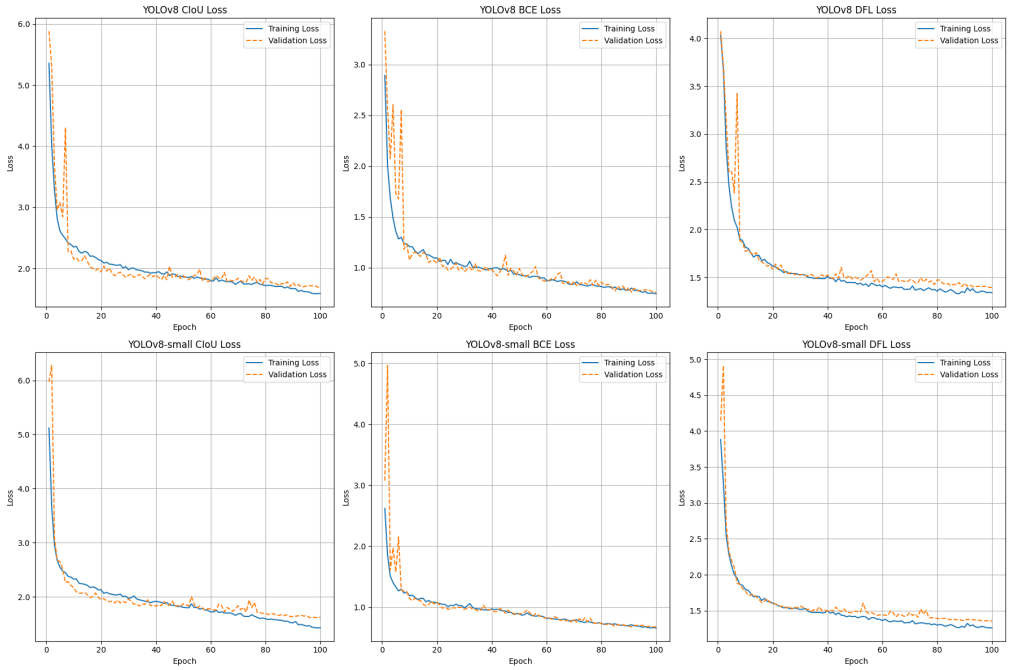


Figure 4.4: Comparison of YOLOv8 and YOLOv8-small Losses. The plots show the training and validation losses for CIoU, BCE, and DFL components across 100 epochs.

As expected, the training time of the YOLOv8-small model was shorter than that of the original YOLOv8 model, with a training time of 0.169 hours (10 minutes), making it approximately 26.5% faster, and a better inference speed of 3.3 milliseconds. During training, we analysed each loss component separately (CIoU, DFL, BCE) to determine if the removal of layers had any effect on the model's learning process. The learning progression across 100 epochs is shown for each loss component in Figure 4.4. Upon examining the graphs for each loss component, it appears that the architectural modifications did not have a significant impact on the loss components or disrupt the typical learning process of the YOLOv8 model. All the graphs in the figure show the same trend as before; a sharp drop in loss in the first 20 epochs, followed by a gradual decrease as training progresses. The validation losses for the YOLOv8-small model appear slightly more stable compared to the original YOLOv8, as indicated by the reduced spikes in the beginning of training. Additionally, both the training and validation losses of the YOLOv8-small model are marginally lower than those of the original model. This suggests that the reduction in layers may have streamlined the model's learning process, allowing it to focus more effectively on the relevant features of the dataset without compromising its performance. Next, we compare the quantitative and visual performance of both models on the validation set. First, we compute the precision, recall, F-1 score and the resulting mAP metrics for both models. Figure 4.5 shows 5 plots illustrating this comparison.

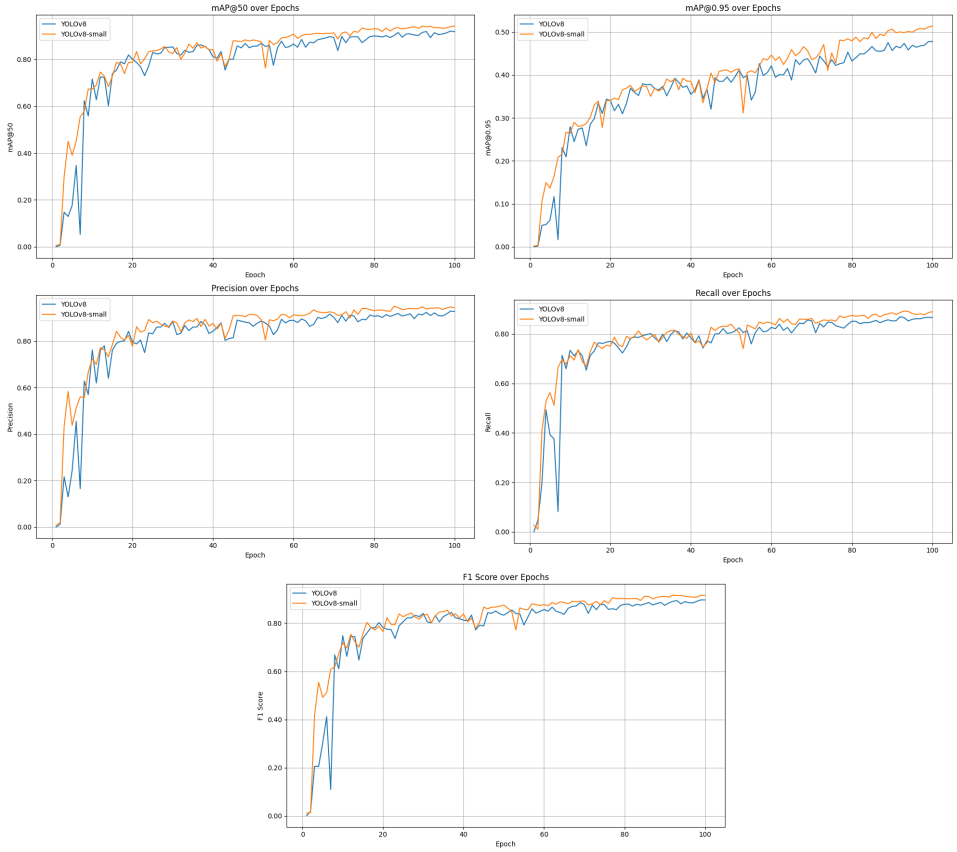


Figure 4.5: Comparison of YOLOv8 and YOLOv8-small across mAP, Precision, Recall, and F1 Score metrics over 100 epochs.

The YOLOv8-small model consistently outperformed the original YOLOv8 model across all metrics, as demonstrated by the metrics' plots. The higher mAP scores across different IoU thresholds indicate that YOLOv8-small has a better capacity to accurately detect and localise trees. The slightly better precision and recall values further suggest that the model is more reliable in distinguishing trees from other objects, reducing the likelihood of both false positives (incorrectly identifying non-trees as trees) and false negatives (failing to detect actual trees). Next, we analyse to how both models perform visually. To do so, we utilise the built-in Ultralytics plotting package, which provides a clear depiction of how the YOLO models make predictions on the validation set. Figure 4.6 presents examples of predictions on images from the validation set.

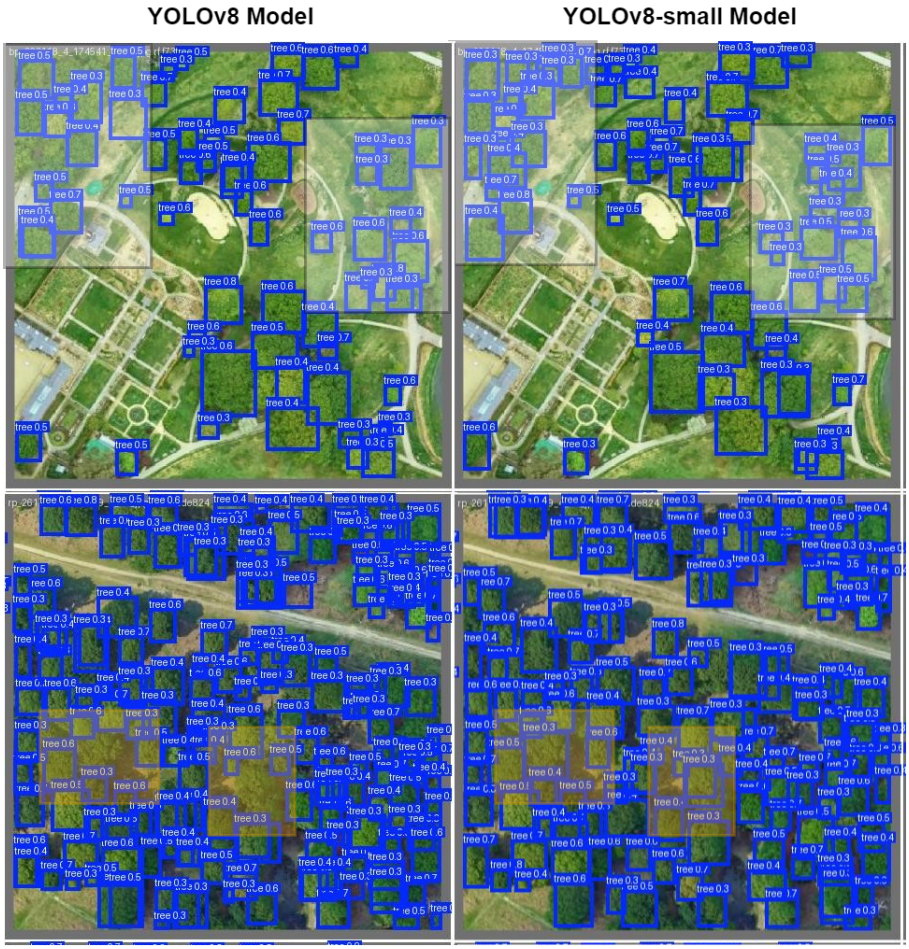


Figure 4.6: Visual Comparison of YOLOv8 and YOLOv8-small Model Predictions on Validation Images. The highlighted regions in the first row indicate areas where the YOLOv8-small model makes more predictions than the YOLOv8 model. In the bottom row, the YOLOv8-small model demonstrates better localisation, with fewer overlapping boxes and improved detection of tree crowns, as compared to the YOLOv8 model.

We can observe some key differences between the two models. In the first row of images, the highlighted regions reveal that the YOLOv8-small model tends to make more predictions in certain areas compared to the original YOLOv8 model. This suggests that YOLOv8-small is more sensitive to detecting smaller or less distinct tree crowns, which the original model may have overlooked. In the second row, the differences become more pronounced. The YOLOv8-

small model demonstrates better localisation, as evidenced by fewer overlapping bounding boxes and more accurately defined tree crowns in the entire image. The original YOLOv8 model, on the other hand, shows several overlapping boxes covering the single entities, which can lead to overestimation. The highlighted regions in the second row show that the YOLOv8-small model successfully detects some tree canopies that the original YOLOv8 model misses entirely, although this does not occur a lot. One general observation across many validation images is that the YOLOv8-small model tends to make more confident predictions, with slightly higher probability scores associated with its bounding boxes. Overall, while both models perform well, the visual analysis shows that the YOLOv8-small model offers improvements in localisation and prediction, making it a more refined tool for detecting trees in large satellite images. Finally, we compare the original and the modified model on the test set. We calculate the difference between the predicted the number of trees and the ground truth for each image. The results are shown in Figure 4.7, and their average (MAE) is presented as well.

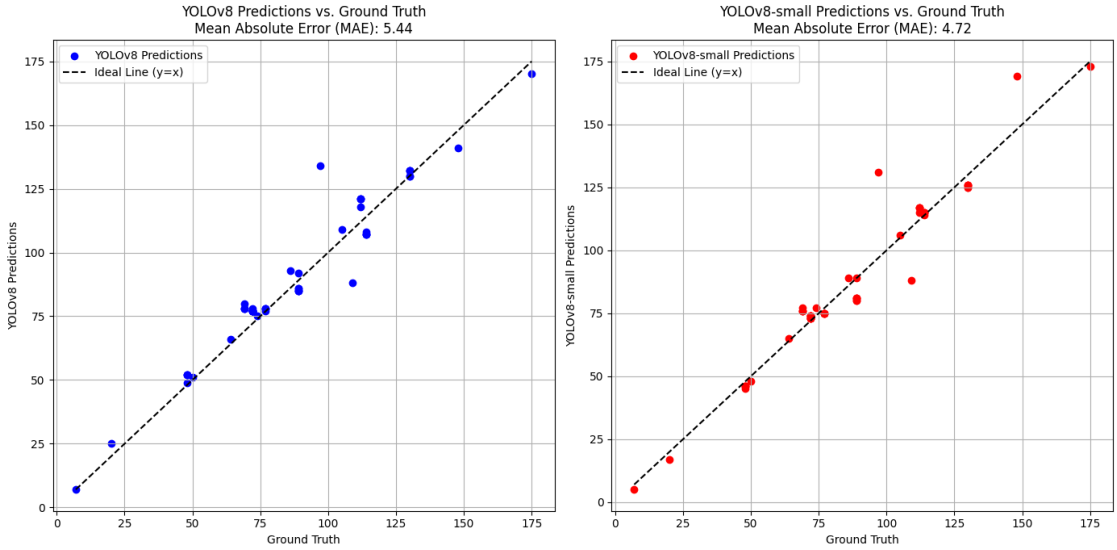


Figure 4.7: Comparison of YOLOv8 and YOLOv8-small predictions against the ground truth. The left plot shows the YOLOv8 predictions (blue dots) compared to the ground truth, with a Mean Absolute Error (MAE) of 5.44. The right plot shows the YOLOv8-small predictions (red dots) compared to the ground truth, with a Mean Absolute Error (MAE) of 4.72. The dotted line in both plots represents the ideal case where predictions match the ground truth exactly ($y = x$).

The predictions on unseen test data are very similar for both models, with the YOLOv8-small model slightly outperforming the YOLOv8 model with a MAE of 4.72. This small difference suggests that the YOLOv8-small model is slightly more accurate in its predictions. However, both models have a similar distribution of prediction errors, as shown by the alignment of the predictions with the ideal line, where predictions perfectly match the ground truth. This line was initially drawn with the intention of seeing whether any of the models severely overestimate or underestimate their predictions. The YOLOv8-normal model tends to have more overestimations than underestimations, whereas the YOLOv8-small model displays a more balanced distribution of errors, with the exception of 3 data points that deviate greatly from the ground truth. We summarise all the comparison results from this section in Table 4.2.

Table 4.2: Summary of YOLOv8 and YOLOv8-small Model Performance on Tree Detection Task

Model	mAP@0.5	mAP@0.5:0.95	Precision	Recall	F1-Score	Training Time (hrs)	Final Test MAE
YOLOv8	0.92	0.48	0.92	0.87	0.90	0.23	5.44
YOLOv8-small	0.94	0.51	0.94	0.89	0.92	0.17	4.72

Given the insights gained from the comparisons of the YOLOv8 and YOLOv8-small models, it becomes evident that the architectural adjustments made to YOLOv8 have led to significant improvements in performance of all metrics. The fact that these improvements come with a reduction in both the number of layers and the number of parameters is particularly noteworthy. It suggests that the YOLOv8-small model is not only more effective but also more efficient—requiring less computational power and time to train. In the next section, we explore the performance of all the trained models in this study on the geospatial dataset.

4.3 Geospatial Dataset Evaluation

In this section, we evaluate all trained models on a manually labeled geospatial dataset. The primary motivation for creating this dataset was to assess how well the top-performing models generalise to satellite imagery from a new source. This step was necessary because even the best YOLO models encountered challenges with edge cases—such as trees with shadows, trees with sparse foliage, and trees in various colours or seasons—that were not adequately represented in the original training data. Figure 4.8 demonstrates how the trained models from this study were not able to generalise well to a satellite image exported from Google Earth Engine and includes trees with varying characteristics. We follow this with baseline estimation on the image.

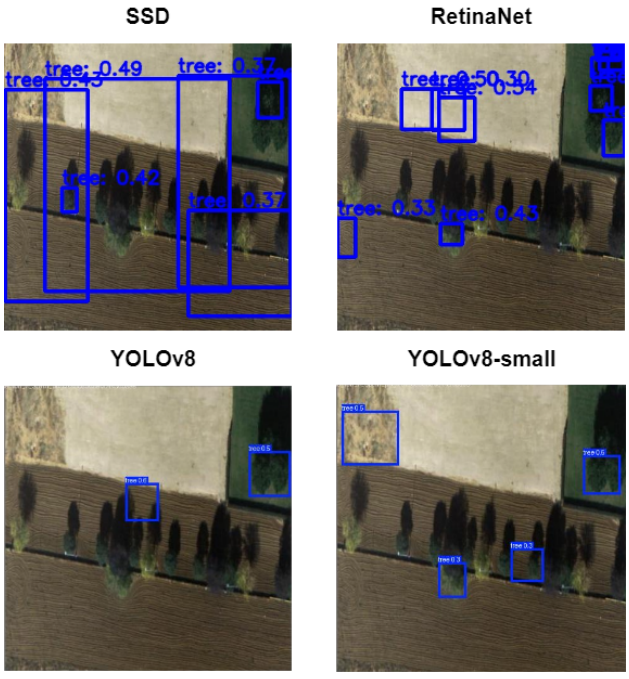


Figure 4.8: The figure shows the performance of SSD, RetinaNet, YOLOv8, and YOLOv8-small models on a satellite image containing small trees, shadows and other varying characteristics.

We evaluate how the baseline method performs on this region using the equation 3.1, the baseline estimation formula. The area of the region was determined using Google Earth Engine and the Canopy Proportion was extracted from the GFCC dataset:

$$\text{Estimated Tree Count} = \left(\frac{\text{Area of ROI}}{\text{Average Tree Canopy Area}} \right) \times \text{ROI Canopy Cover Proportion}$$

Given the values:

$$\text{Area of ROI} = 2659.87 \text{ square meters}$$

$$\text{ROI Canopy Cover Proportion} = 0.2$$

We calculate tree count:

$$\text{Estimated Tree Count} = \left(\frac{2659.87}{60} \right) \times 0.2$$

$$\text{Estimated Tree Count} \approx 8.87$$

For the sample image, the baseline method predicts around 9 trees, with the ground truth being around 13 trees. In this particular case, the baseline provides the closest estimation to the ground truth compared to the trained models, which struggled to generalise to this satellite image. This result highlights a significant challenge in deep learning models when applied to out-of-distribution data. The implications of this are crucial for applications in geospatial analysis, where models must often be deployed on data that significantly differs from the training set. This highlights two key considerations: the necessity of fine-tuning the trained models on a dedicated geospatial dataset, and the importance of comparing deep learning models to traditional baseline methods.

To further investigate the baseline method's performance, we evaluate it on the training and validation images of the geospatial dataset. During this process, we identified several anomalies, particularly in regions from Spain and the Netherlands. In the Netherlands, three regions showed extreme overestimations of the tree count, exceeding the ground truth by over 100 trees. This occurred because the regions had large areas of interest (ROI), which, when combined with a fixed average tree canopy area, resulted in disproportionately high estimated tree counts. This overestimation in large regions of interest is a limitation of the baseline method.

In the case of Spain, many regions exhibited very low or even zero canopy cover proportions due to inadequate coverage by the GFCC dataset. This led to significant underestimations in tree counts. To address this, we implemented a fallback mechanism: if the canopy cover proportion was less than 0.05, we incremented it by 0.2. This adjustment helped mitigate the issue as seen in Tables 4.3 and 4.4.

Table 4.3: Original Data MAE by Country

Country	**MAE**
Latvia	7.46
Netherlands	31.62
Spain	13.00
Switzerland	9.69

Table 4.4: MAE by Country Post-processing

Country	**MAE**
Latvia	7.46
Netherlands	5.32
Spain	5.00
Switzerland	9.69

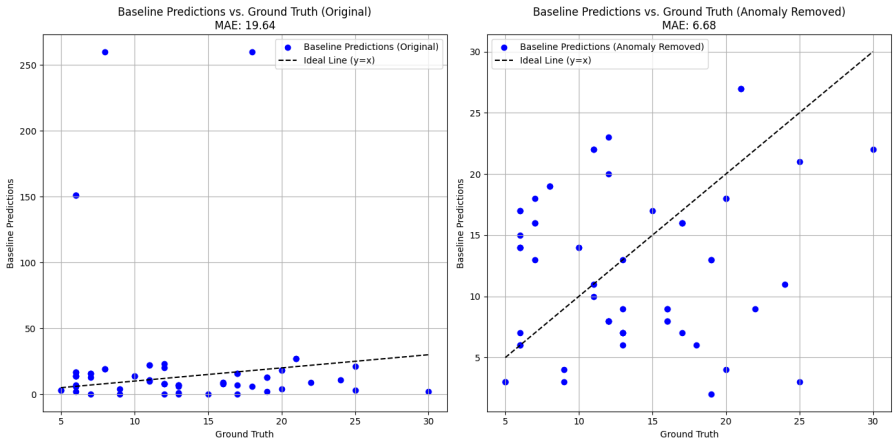


Figure 4.9: Comparison of Baseline Predictions vs. Ground Truth before (left) and after (right) anomaly removal.

Figure 4.9 shows the how the baseline results have changed after addressing the anomalies. The comparison of baseline predictions before and after anomaly removal reveals the method's sensitivity to region-specific characteristics. By addressing these anomalies, we achieved more accurate and consistent predictions, demonstrated by the reduction of the MAE from 19.64 to 6.68. Overall, the baseline performance on the satellite images was good considering its simplicity and the post processing measure taken. The issues in the Netherlands regions were more related to outliers that could be identified and removed, but the problems observed in Spain reflect a more systematic issue inherent in the dataset. This serves as a caution against over-reliance on such datasets, which may lack comprehensive coverage, ultimately affecting prediction accuracy.

In the final experiment of this project, we train all deep learning models on the geospatial dataset and we present a direct comparison of methods in this study by analysing their performance on the test set. Despite the small size of the dataset, the test set includes six carefully selected images, ensuring that it contains difficult examples that will challenge the models' generalisation capabilities. See Figure 4.10 below.

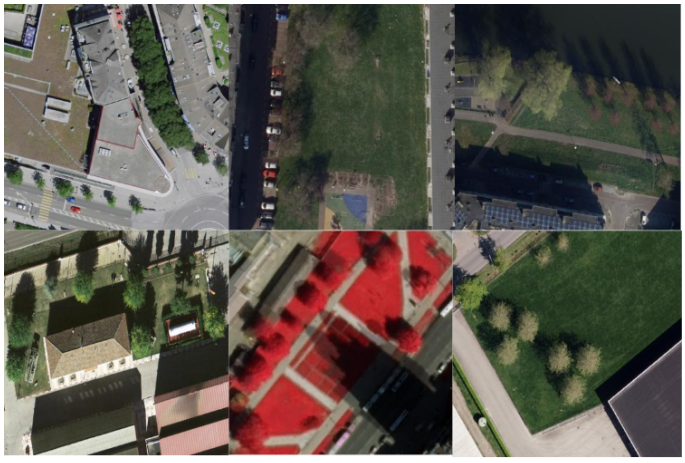


Figure 4.10: Test set images used in the geospatial dataset in order. The selected images include challenging examples with different tree types, colours, shadows and spectral bands.

For each image in the test set, we perform inference using the trained models: Baseline, SSD, RetinaNet, YOLOv8, YOLOv8-small, and YOLOv8-base. The YOLOv8-base model, unlike the others, was trained solely on the geospatial dataset without any prior training on the tree dataset. This comparison allows us to evaluate the impact of training on the tree dataset as a form of "pre-training" for the downstream task of predicting trees in the final satellite test images. The results are shown in Figure 4.11. The x-axis shows images 1-6 from Figure 4.10 in order.

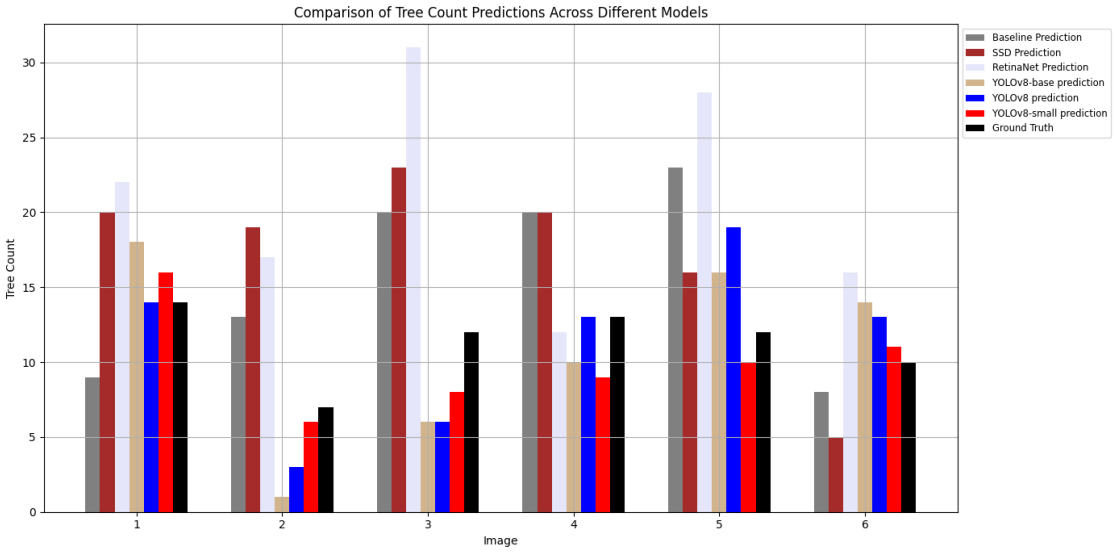


Figure 4.11: Comparison of tree count predictions across different models on the test images. The models are represented by different colours.

Across all test images, the YOLOv8 (blue) and YOLOv8-small (red) models demonstrate superior performance, with predictions that closely match the Ground Truth, represented in black. YOLOv8 shows remarkable accuracy in images 1 and 4, where the tree canopies are distinctly green, managing to predict the exact tree count as the ground truth. This suggests that the YOLOv8 model is particularly effective in scenarios where the tree canopies are clearly defined. On the other hand, the YOLOv8-small model outperforms in the other four images, where the tree canopies are less vibrant or have been altered due to seasonal and spectral augmentations. This adaptability indicates that YOLOv8-small, with its modified architecture, might be better suited to handling variations in tree canopy appearance, making it a robust choice for diverse environmental conditions. Both YOLOv8 and YOLOv8-small outperform the YOLOv8-base model, which was trained solely on the geospatial dataset without training on the tree dataset. This highlights the importance of the tree dataset as an effective pretraining resource that allows better generalisation on tree detection tasks.

In contrast, RetinaNet (lavender) consistently underperforms, especially in scenarios with challenging conditions in images 3 and 5, highlighting its limitations in generalising across different types of tree canopies and colour bands. SSD (brown) performs moderately, better than RetinaNet but not as well as the YOLOv8 models. Interestingly, the baseline method (grey), despite its simplicity, holds up surprisingly well against the more complex models. We present a summary of the final results from the geospatial experiment and include the mAP metrics obtained in training:

Table 4.5: Model Performance Summary on the Geospatial Dataset

Model	**mAP@0.5**	**mAP@0.5:0.95**	**Test MAE**	**Training Time (hrs)**
Baseline	-	-	6.50	-
SSD	0.74	0.34	7.50	0.08
RetinaNet	0.85	0.43	10.00	0.36
YOLOv8	0.75	0.35	3.33	0.05
YOLOv8-base	0.79	0.37	4.50	0.05
YOLOv8-small	0.78	0.36	2.33	0.03

Given the high mAP values achieved by RetinaNet during training (mAP@0.5 of 0.85 and mAP@0.5:0.95 of 0.43) and its poor performance on the test set, as indicated by an MAE of 10.00, it is evident that RetinaNet has likely overfitted to the training dataset. This overfitting is not entirely unexpected, considering the small size of the geospatial dataset and the fact that RetinaNet has the highest number of parameters among all the models evaluated. The YOLOv8-small model demonstrates the best overall performance, with a test MAE of 2.33, slightly outperforming the original YOLOv8 model, although the YOLOv8 model demonstrates better performance when the tree canopies are distinctly green, as previously discussed. Interestingly, the baseline method, despite its simplicity, slightly outperforms the SSD model after post-processing, achieving a test MAE of 6.50 compared to SSD's MAE of 7.50. This highlights that even simple methods, when carefully tuned and processed, can rival deep learning models especially when the dataset is small. Size remains one limitation of the geospatial dataset.

Chapter 5

Discussion

5.1 Baseline Method

Inspired by traditional detection methods, the baseline method provided an excellent tree estimation benchmark for any region of interest. Its performance on the geospatial test set was surprisingly strong, achieving a commendable Mean Absolute Error (MAE) of 6.50, which outperformed both RetinaNet and SSD—two widely recognised deep learning models. As previously discussed, RetinaNet's poor performance can likely be attributed to overfitting, a consequence of its complex architecture combined with the small size of the geospatial dataset. The decision to maintain the same training configuration for the geospatial experiment as was used for the tree dataset, particularly the 100-epoch training duration, may not have been optimal. While the baseline method slightly outperformed the SSD model, the margin was just 1.0, and given the limited scope of the analysis—only 6 test images—we cannot definitively conclude that the baseline method is superior to SSD.

Despite its impressive performance, the baseline method has several limitations. Firstly, it is heavily dependent on the GFCC dataset, which provides global forest canopy cover data. This reliance is a significant limitation, as certain regions, like those in Spain highlighted in the final experiment, are not well-covered. Relying on a single dataset for tree estimation can introduce significant biases, particularly when some areas are better represented than others. This suggests the need for careful consideration of the data sources used in geospatial analysis. Secondly, the baseline method assumes a fixed tree canopy area across all regions, yet in reality, individual tree canopies can vary significantly in size depending on their species and location. One study examines the distributional equity of urban tree canopy in multiple US states and found significant variation in canopy distribution between states and even among cities within the same state [59]. Mathematically, fixing the tree size also leads to overestimations as we increase the area of the region in the baseline equation. However, having a large area does not necessarily mean larger tree count, especially since this study mainly covers urban areas and not forested regions. Thirdly, this baseline approach assumes a uniform distribution of tree canopies within the ROI, which is rarely the case in natural settings. Trees are often clustered or unevenly spaced, leading to inaccuracies when using a fixed canopy size. It is also worth mentioning that we were only able to compare the baseline method with the deep learning models on the geospatial dataset because we had the

coordinates of the regions. Metadata such as coordinates and location are rarely provided with image datasets, making it challenging to apply baseline predictions in comparison to deep learning models.

While the baseline offers valuable initial insights, its accuracy is inherently tied to the fixed canopy size assumption, making it less reliable in heterogeneous environments. A potential improvement would be to adopt a dynamic tree canopy size that takes into account factors such as region and tree species. Another improvement would be the integration of multiple datasets for canopy cover proportion that are regularly updated, rather than relying on a single dataset. Despite these limitations, the baseline method provided a simple yet effective approach to tree estimation, capable of producing results that rival deep learning models trained specifically on tree data. Being able to obtain a rough estimate of tree counts across any region of interest was a significant milestone in the research phase of this project, as it allowed for meaningful comparisons with different techniques.

5.2 Architecture Modification

The main findings of this paper center around the significant modifications applied to the YOLOv8 architecture, resulting in the development of YOLOv8-small—a more compact, yet highly effective version of the original model. This novel version not only surpasses the original YOLOv8 in terms of accuracy, speed, precision, recall, and F-1 score, but also demonstrates the potential of architecture-specific optimisations tailored to the challenges posed by satellite imagery, particularly in the task of tree detection.

In the first experiment, the goal was to identify the most effective model architecture for detecting trees within the tree dataset. Among the three models evaluated—SSD, RetinaNet, and YOLOv8—the YOLOv8 architecture demonstrated superior performance across multiple key metrics, including accuracy, training time, and inference speed. These results were achieved despite YOLOv8 having a higher layer count compared to SSD and RetinaNet, underscoring its well-optimised architecture. The use of the AdamW optimiser and the decision to apply mosaic augmentations contributed to this result. Given YOLOv8's impressive performance, it became apparent that a deeper exploration of its architectural design was necessary. By understanding how the input image is processed through the backbone and neck for feature extraction and aggregation, and how predictions are made using detect modules at different scales, we identified opportunities to tailor the architecture more specifically to the dataset. By strategically removing two of the three detection modules, which were originally designed to process downsampled feature maps at medium and low resolutions, the modified YOLOv8-small architecture retained only the module that processes higher-resolution feature maps, thereby enhancing its focus on the small-scale objects in the dataset. Initially, there was a concern that this change could disrupt the model's learning process by preventing the network from capturing low-resolution representations of the input image. On the contrary, the YOLOv8-small model exhibited more stable validation losses and marginally lower overall loss values compared to the original model, indicating a more streamlined learning process.

5.3 Datasets

The performance of the trained models on random satellite images exported from Google Earth Engine highlighted a significant issue: the out-of-distribution problem. Even though the Roboflow tree dataset was well-labelled, the trained models struggled to generalise when faced with new, unseen data, particularly satellite imagery containing edge cases like extensive shadows or trees in different seasonal states. This out-of-distribution problem is a well-documented challenge in computer vision, particularly in object detection where models are often deployed on data that significantly deviates from the training set [60]. Fine-tuning for specific downstream tasks is one of the solutions proposed in the literature to address this problem [60]. This, among other reasons, prompted the creation of the geospatial dataset, which included satellite images exported from various countries using Google Earth Engine. Therefore, fine-tuning on the geospatial dataset was a critical step towards improving model generalisation by incorporating edge cases, such as trees with varying colors and shadows, which were not adequately represented in the original tree dataset.

It is important to draw some distinctions and limitations between both datasets. The Roboflow tree dataset is well-labeled and curated, featuring complex forestry scenes with overlapping trees, whereas the geospatial dataset was manually labelled and lacks the same level of precision and expertise. This discrepancy likely contributed to the inferior performance of YOLOv8-base, which was trained exclusively on the geospatial dataset, compared to the original YOLOv8 model. The fact that the YOLOv8 and YOLOv8-small models performed better than YOLOv8-base also suggests that pre-training on the tree dataset provided a more robust foundation for the models, although the YOLOv8-base was trained for lesser epochs. Additionally, the size of both datasets is a significant limitation in this project. Due to time constraints, the geospatial dataset comprised only 70 images, and although the test images were carefully selected to challenge the models' generalisation capabilities, 6 images are insufficient to draw comprehensive conclusions about all the models. Even the tree dataset is limited in size; according to the YOLO training documentation, a minimum of 1,500 well-labelled images is recommended for effective model training [61]. A considerable amount of time was spent at the beginning of this project searching for appropriate tree datasets, highlighting the scarcity of well-annotated tree datasets, particularly for urban areas. Therefore, the need for larger, more diverse, and accurately labeled tree datasets is pivotal for advancing the development of robust models capable of detecting trees with good accuracy and reliability.

Chapter 6

Conclusion

To conclude, this study carried out the objectives necessary to fulfill the project's aim: to implement and refine state-of-the-art deep learning methods to accurately detect and count trees in satellite imagery. The development of the YOLOv8-small model, with its tailored architectural modifications, addressed the unique challenges posed by satellite imagery, such as detecting trees with spectral and seasonal augmentation with varying resolutions. This model consistently outperformed other models in all experiments, demonstrating the effectiveness of the proposed modifications. The Baseline Method provided a valuable point of comparison, setting a solid standard that, in some cases, rivaled the performance of the more complex deep learning models, particularly in the geospatial dataset. However, the question of how the results would change if the geospatial dataset was larger and more diverse remains unanswered. The potential impact of dataset size on the generalisation and accuracy of models in satellite imagery is a critical area that warrants further exploration.

In terms of future work, several avenues are worth exploring. Firstly, the geospatial dataset should be expanded to include a broader range of environmental conditions, tree species, and urban landscapes, ideally with annotations provided by experts. This would offer a more comprehensive evaluation of model generalisation and provide a clearer understanding of which models perform best across diverse scenarios. Another promising direction is the incorporation of height prediction for each detected tree using LiDAR and canopy height maps, which are publicly available. Predicting tree height is particularly relevant in insurance analysis, as it helps quantify the risk associated with tree falls in urban areas. All in all, this thesis lays the groundwork for future research aimed at further improving the accuracy and applicability of tree detection models in various real-world scenarios.

Bibliography

[1] Marcus Lindner, Michael Maroschek, Sigrid Netherer, Antoine Kremer, Anna Barbati, Jordi Garcia-Gonzalo, Rupert Seidl, Sylvain Delzon, Piermaria Corona, Marja Kolström, Manfred J. Lexer, and Marco Marchetti. Climate change impacts, adaptive capacity, and vulnerability of european forest ecosystems. *Forest Ecology and Management*, 259(4):698–709, 2010. Adaptation of Forests and Forest Management to Changing Climate.

[2] Erkki Tomppo, Håkan Olsson, Göran Ståhl, Mats Nilsson, Olle Hagner, and Matti Katila. Combining national forest inventory field plots and remote sensing data for forest databases. *Remote Sensing of Environment*, 112(5):1982–1999, 2008. Earth Observations for Terrestrial Biodiversity and Ecosystems Special Issue.

[3] Sebastian Schnell, Christoph Kleinn, and Göran Ståhl. Monitoring trees outside forests: a review. *Environmental Monitoring and Assessment*, 187, 09 2015.

[4] Federica Baffetta, Piermaria Corona, and Lorenzo Fattorini. Assessing the attributes of scattered trees outside the forest by a multi-phase sampling strategy. *Forestry: An International Journal of Forest Research*, 84(3):315–325, 07 2011.

[5] C. Krishnankutty, K. Thampi, and M Chundamannil. Trees outside forests (tof): A case study of the wood production—consumption situation in kerala. *International Forestry Review - INT FOR REV*, 10:156–164, 06 2008.

[6] The Guardian. Trees cut down at site of tesla gigafactory in germany, satellite images reveal. *The Guardian*, 2024. Accessed: 2024-08-22.

[7] QBE Insurance Group. About qbe, 2024. Accessed: 2024-09-01.

[8] Le Wang, Peng Gong, and Gregory Biging. Individual tree-crown delineation and treetop detection in high-spatial-resolution aerial imagery. *Photogrammetric Engineering Remote Sensing*, 70:351–357, 03 2004.

[9] Marília Gomes and Philippe Maillard. Detection of tree crowns in very high spatial resolution images. *Intech*, page 32, 06 2016.

[10] Pengcheng Han, Cunbao Ma, Jian Chen, Lin Chen, Shuhui Bu, Shibiao Xu, Yong Zhao, Chenhua Zhang, and Tatsuya Hagino. Fast tree detection and counting on uavs for sequential aerial images with generating orthophoto mosaicing. *Remote Sensing*, 14(16), 2022.

[11] Jamie Tolan, Hung-I Yang, Benjamin Nosarzewski, Guillaume Couairon, Huy V. Vo, John Brandt, Justine Spore, Sayantan Majumdar, Daniel Haziza, Janaki Vamaraju, Theo Moutakanni, Piotr Bojanowski, Tracy Johns, Brian White, Tobias Tiecke, and Camille Couprie. Very high resolution canopy height maps from rgb imagery using self-supervised vision transformer and convolutional decoder trained on aerial lidar. *Remote Sensing of Environment*, 300:113888, 2024.

[12] Noel Gorelick, Matt Hancher, Mike Dixon, Simon Ilyushchenko, David Thau, and Rebecca Moore. Google earth engine: Planetary-scale geospatial analysis for everyone. *Remote Sensing of Environment*, 202:18–27, 2017.

[13] Sizhuo Li, Martin Brandt, Rasmus Fensholt, Ankit Kariryaa, Christian Igel, Fabian Gieseke, Thomas Nord-Larsen, Stefan Oehmcke, Ask Holm Carlsen, Samuli Junttila, Xiaoye Tong, Alexandre d'Aspremont, and Philippe Ciais. Deep learning enables image-based tree counting, crown segmentation, and height prediction at national scale. *PNAS Nexus*, 2(4):pgad076, 03 2023.

[14] Ling Yao, Tang Liu, Jun Qin, Ning Lu, and Chenghu Zhou. Tree counting with high spatial-resolution satellite imagery based on deep neural networks. *Ecological Indicators*, 125:107591, 2021.

[15] Seyed Hamid Rezatofighi, Nathan Tsoi, JunYoung Gwak, Amir Sadeghian, Ian D. Reid, and Silvio Savarese. Generalized intersection over union: A metric and A loss for bounding box regression. *CoRR*, abs/1902.09630, 2019.

[16] Paul Henderson and Vittorio Ferrari. End-to-end training of object class detectors for mean average precision. *CoRR*, abs/1607.03476, 2016.

[17] Dillon Reis, Jordan Kupec, Jacqueline Hong, and Ahmad Daoudi. Real-time flying object detection with yolov8, 2024.

[18] Paul Viola and Michael Jones. Rapid object detection using a boosted cascade of simple features. In *CVPR 2001*, volume 1, pages I–511, 02 2001.

[19] P. Viola and M. Jones. Robust real-time face detection. In *Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001*, volume 2, pages 747–747, 2001.

[20] N. Dalal and B. Triggs. Histograms of oriented gradients for human detection. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 1, pages 886–893 vol. 1, 2005.

[21] Pedro Felzenszwalb, David McAllester, and Deva Ramanan. A discriminatively trained, multiscale, deformable part model. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2008.

[22] Zhengxia Zou, Zhenwei Shi, Yuhong Guo, and Jieping Ye. Object detection in 20 years: A survey. *CoRR*, abs/1905.05055, 2019.

[23] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. *Commun. ACM*, 60(6):84–90, may 2017.

[24] Manuel Carranza-García, Jesús Torres-Mateo, Pedro Lara-Benítez, and Jorge García-Gutiérrez. On the performance of one-stage and two-stage object detectors in autonomous vehicles using camera data. *Remote Sensing*, 13(1), 2021.

[25] Ross B. Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR*, abs/1311.2524, 2013.

[26] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015.

[27] Tsung-Yi Lin, Piotr Dollár, Ross B. Girshick, Kaiming He, Bharath Hariharan, and Serge J. Belongie. Feature pyramid networks for object detection. *CoRR*, abs/1612.03144, 2016.

[28] Shuying Liu and Weihong Deng. Very deep convolutional neural network based image classification using small training sample size. In *2015 3rd IAPR Asian Conference on Pattern Recognition (ACPR)*, pages 730–734, 2015.

[29] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012.

[30] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: single shot multibox detector. *CoRR*, abs/1512.02325, 2015.

[31] Dumitru Erhan, Christian Szegedy, Alexander Toshev, and Dragomir Anguelov. Scalable object detection using deep neural networks. *CoRR*, abs/1312.2249, 2013.

[32] Christian Szegedy, Scott E. Reed, Dumitru Erhan, and Dragomir Anguelov. Scalable, high-quality object detection. *CoRR*, abs/1412.1441, 2014.

[33] Ross B. Girshick. Fast R-CNN. *CoRR*, abs/1504.08083, 2015.

[34] Nieves Crasto. Class imbalance in object detection: An experimental diagnosis and study of mitigation strategies, 2024.

[35] Maxime Bucher, Stéphane Herbin, and Frédéric Jurie. Hard negative mining for metric learning based zero-shot classification. *CoRR*, abs/1608.07441, 2016.

[36] Tsung-Yi Lin, Priya Goyal, Ross B. Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection. *CoRR*, abs/1708.02002, 2017.

[37] Jan Hendrik Hosang, Rodrigo Benenson, and Bernt Schiele. Learning non-maximum suppression. *CoRR*, abs/1705.02950, 2017.

[38] Juan Terven, Diana-Margarita Córdova-Esparza, and Julio-Alejandro Romero-González. A comprehensive review of yolo architectures in computer vision: From yolov1 to yolov8 and yolo-nas. *Machine Learning and Knowledge Extraction*, 5(4):1680–1716, November 2023.

[39] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.

[40] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott E. Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[41] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. *CoRR*, abs/1612.08242, 2016.

[42] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018.

[43] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *CoRR*, abs/2004.10934, 2020.

[44] Glenn Jocher. Yolov5 by ultralytics, 2020.

[45] Glenn Jocher, Ayush Chaurasia, and Jing Qiu. Ultralytics yolov8, 2023.

[46] Zhaohui Zheng, Ping Wang, Wei Liu, Jinze Li, Rongguang Ye, and Dongwei Ren. Distance-iou loss: Faster and better learning for bounding box regression. *CoRR*, abs/1911.08287, 2019.

[47] Xiang Li, Wenhai Wang, Lijun Wu, Shuo Chen, Xiaolin Hu, Jun Li, Jinhui Tang, and Jian Yang. Generalized focal loss: Learning qualified and distributed bounding boxes for dense object detection. *CoRR*, abs/2006.04388, 2020.

[48] Tree Counting. Tree counting dataset. https://universe.roboflow.com/tree-counting/tree-counting-gz3xz , mar 2023. visited on 2024-08-14.

[49] Project. Tree counting dataset. https://universe.roboflow.com/project-s402o/tree-counting-qiw3h , mar 2023. visited on 2024-08-14.

[50] tree species. tree species dataset. https://universe.roboflow.com/tree-species/tree-species-a4jme , apr 2024. visited on 2024-08-14.

[51] B. Dwyer, J. Nelson, T. Hansen, et al. Roboflow (version 1.0), 2024. Computer vision software.

[52] Zhiwei Wei and Chenzhen Duan. Amrnet: Chips augmentation in areial images object detection. *CoRR*, abs/2009.07168, 2020.

[53] Sk. Sazid Muhammad and R. Ramakrishnan. Geotiff - a standard image file format for gis applications. In *Map India 2003 - Image Processing & Interpretation*, Ahmedabad, India, 2003. Space Applications Centre (ISRO). Presented at the Map India 2003 Conference.

[54] NASA LP DAAC at the USGS EROS Center. Global forest canopy cover (gfcc) dataset version 3, 2021. Accessed: 2024-06-15.

[55] M. Bouvier, S. Durrieu, R.A. Fournier, and J.-P. Renaud. Generalizing predictive models of forest inventory attributes using an area-based approach with airborne lidar data. *Remote Sensing of Environment*, 156:322–334, 2015. Accessed: 2024-06-29.

[56] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context, 2015.

[57] Ilya Loshchilov and Frank Hutter. Fixing weight decay regularization in adam. *CoRR*, abs/1711.05101, 2017.

[58] Glenn Jocher, Ayush Chaurasia, and Jing Qiu. Ultralytics YOLO, January 2023.

[59] Kirsten Schwarz, Michail Fragkias, Christopher G. Boone, Weiqi Zhou, Melissa McHale, J. Morgan Grove, Jarlath O'Neil-Dunne, Joseph P. McFadden, Geoffrey L. Buckley, Dan Childers, Laura Ogden, Stephanie Pincetl, Diane Pataki, Ali Whitmer, and Mary L. Cadenasso. Trees grow on money: Urban tree canopy cover and environmental justice. *PLOS ONE*, 10(4):e0122051, April 2015.

[60] Stanislav Fort, Jie Ren, and Balaji Lakshminarayanan. Exploring the limits of out-of-distribution detection. *CoRR*, abs/2106.03004, 2021.

[61] Ultralytics. Train mode documentation - ultralytics yolov8, 2024. Accessed: 2024-08-30.

Appendix A

Hyperparameter Tuning

To determine the best hyperparameter configuration for the models trained, we first implemented a random search strategy on the YOLOv8 model using the `model.tune()` method from the Ultralytics repository [58]. This method leverages mutation-based techniques to explore the hyperparameter space by applying small, random changes to existing hyperparameters, thereby generating new candidates for evaluation.

This method was used to train the model with different configurations over 120 iterations, each running for 30 epochs. The process took approximately 9 hours in total. The fitness metric used in this tuning process was the mean Average Precision (mAP). The performance of the model, in terms of fitness, is shown in the figure below. As the iterations progressed, we observed a clear improvement in the mAP, reflecting the effectiveness of the hyperparameter search.

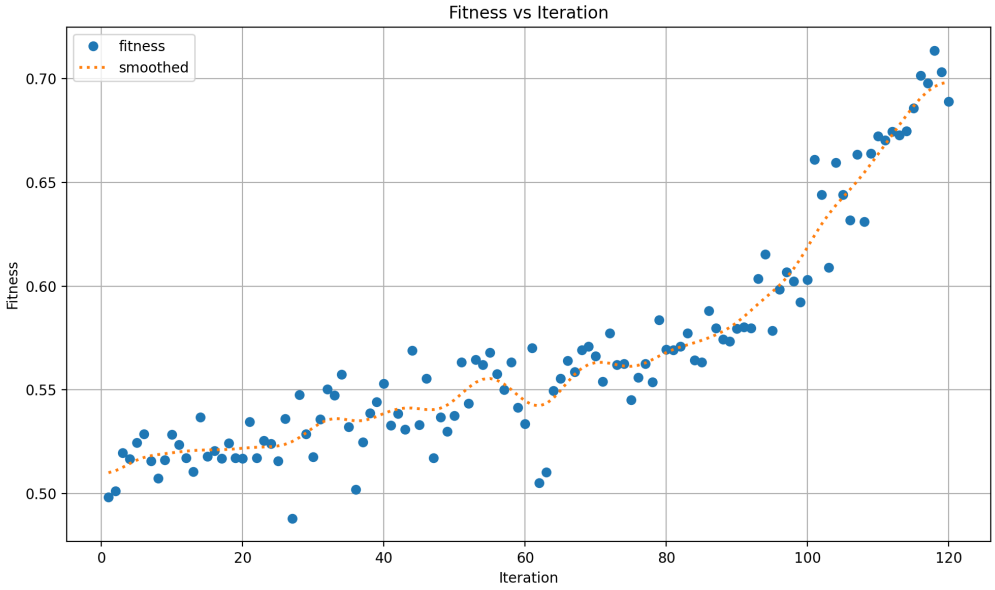


Figure A.1: Fitness vs. Iteration during Hyperparameter Tuning. The fitness score, represented by mAP, improves as the iterations progress, demonstrating the effectiveness of the random search strategy. The highest fitness was achieved in iteration 118.

The highest fitness achieved was at iteration 118, representing the best hyperparameters shown below:

```
# 120/120 iterations complete ✅ (32366.53s)
# Results saved to runs/detect/tune
# Best fitness=0.71368 observed at iteration 118
# Best fitness metrics are {'metrics/precision(B)': 0.95854, 'metr
95(B)': 0.68434, 'val/box_loss': 0.88642, 'val/cls_loss': 0.29984,
# Best fitness model is runs/detect/yolov8s_normal_tuned118
# Best fitness hyperparameters are printed below.

hsv_h: 0.011
hsv_s: 0.17527
hsv_v: 0.16854
degrees: 0.0
translate: 0.02675
scale: 0.09048
shear: 0.0
perspective: 0.0
flipud: 0.0
fliplr: 0.19722
bgr: 0.0
mosaic: 0.7526
mixup: 0.0
copy_paste: 0.0
```

Figure A.2: Best hyperparameters identified during the tuning process. These values reflect the best settings found for augmentations such as hue, saturation, brightness, and mosaic, before manual tuning took place.

This tuning process worked particularly well with YOLOv8, yielding a set of good hyperparameters. However, it did not perform as well for SSD and RetinaNet models, highlighting the importance of model-specific tuning strategies. Given that these augmentation values did not work equally for all models, the best hyperparameters were manually tweaked until further improvements in performance were observed. As a result, we did not include this section in the main report. The final tweaked parameters are shown in Table 3.1 in the main report.

Appendix B

Model Architectures

B.1 SSD

```
SSD(
  (backbone): SSDFeatureExtractorVGG(
    (features): Sequential(
      (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (1): ReLU(inplace=True)
      (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (3): ReLU(inplace=True)
      (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (6): ReLU(inplace=True)
      (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (8): ReLU(inplace=True)
      (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (11): ReLU(inplace=True)
      (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (13): ReLU(inplace=True)
      (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (15): ReLU(inplace=True)
      (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=True)
      (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (18): ReLU(inplace=True)
      (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (20): ReLU(inplace=True)
      (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (22): ReLU(inplace=True)
    )
    (extra): ModuleList(
```

```
    (0): Sequential(
      (0): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
      (1): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (2): ReLU(inplace=True)
      (3): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (4): ReLU(inplace=True)
      (5): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      (6): ReLU(inplace=True)
      (7): Sequential(
        (0): MaxPool2d(kernel_size=3, stride=1, padding=1, dilation=1, ceil_mode=False)
        (1): Conv2d(512, 1024, kernel_size=(3, 3), stride=(1, 1), padding=(6, 6), dilation=(6, 6))
        (2): ReLU(inplace=True)
        (3): Conv2d(1024, 1024, kernel_size=(1, 1), stride=(1, 1))
        (4): ReLU(inplace=True)
      )
    )
    (1): Sequential(
      (0): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1))
      (1): ReLU(inplace=True)
      (2): Conv2d(256, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
      (3): ReLU(inplace=True)
    )
    (2): Sequential(
      (0): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1))
      (1): ReLU(inplace=True)
      (2): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))
      (3): ReLU(inplace=True)
    )
    (3): Sequential(
      (0): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1))
      (1): ReLU(inplace=True)
      (2): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1))
      (3): ReLU(inplace=True)
    )
    (4): Sequential(
      (0): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1))
      (1): ReLU(inplace=True)
      (2): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1))
      (3): ReLU(inplace=True)
    )
  )
)
(anchor_generator): DefaultBoxGenerator(aspect_ratios=[[2], [2, 3], [2, 3], [2, 3], [2], [2]],
```

```
  clip=True, scales=[0.07, 0.15, 0.33, 0.51, 0.69, 0.87, 1.05],
  steps=[8, 16, 32, 64, 100, 300])
  (head): SSDHead(
    (classification_head): SSDClassificationHead(
      (module_list): ModuleList(
        (0): Conv2d(512, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): Conv2d(1024, 12, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (2): Conv2d(512, 12, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): Conv2d(256, 12, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (4): Conv2d(256, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (5): Conv2d(256, 8, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      )
    )
    (regression_head): SSDRegressionHead(
      (module_list): ModuleList(
        (0): Conv2d(512, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (1): Conv2d(1024, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (2): Conv2d(512, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (3): Conv2d(256, 24, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (4): Conv2d(256, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (5): Conv2d(256, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
      )
    )
  )
  (transform): GeneralizedRCNNTransform(
      Normalize(mean=[0.48235, 0.45882, 0.40784],
      std=[0.00392156862745098, 0.00392156862745098, 0.00392156862745098])
      Resize(min_size=(640,), max_size=640, mode='bilinear')
  )
)
```

B.2 RetinaNet

```
RetinaNet(
  (backbone): BackboneWithFPN(
    (body): IntermediateLayerGetter(
      (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
      (layer1): Sequential(
        (0): Bottleneck(
          (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
```

```
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (downsample): Sequential(
      (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (1): Bottleneck(
    (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (2): Bottleneck(
    (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
)
(layer2): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (downsample): Sequential(
      (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
```

```
  )
  (1): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (2): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (3): Bottleneck(
    (conv1): Conv2d(512, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
)
(layer3): Sequential(
  (0): Bottleneck(
    (conv1): Conv2d(512, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
    (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
    (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
    (downsample): Sequential(
      (0): Conv2d(512, 1024, kernel_size=(1, 1), stride=(2, 2), bias=False)
      (1): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
```

```
(1): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(2): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (conv3): Conv2d(256, 1024, kernel size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
)
(3): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track running_stats=True)
  (conv2): Conv2d(256, 256, kernel size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track running_stats=True)
  (conv3): Conv2d(256, 1024, kernel size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track running_stats=True)
  (relu): ReLU(inplace=True)
)
(4): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track running_stats=True)
  (conv2): Conv2d(256, 256, kernel size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  (conv3): Conv2d(256, 1024, kernel size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  (relu): ReLU(inplace=True)
)
(5): Bottleneck(
  (conv1): Conv2d(1024, 256, kernel size=(1, 1), stride=(1, 1), bias=False)
  (bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  (conv2): Conv2d(256, 256, kernel size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
  (bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
  (conv3): Conv2d(256, 1024, kernel size=(1, 1), stride=(1, 1), bias=False)
  (bn3): BatchNorm2d(1024, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
```

```
      (relu): ReLU(inplace=True)
    )
  )
  (layer4): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(1024, 512, kernel size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
      (conv2): Conv2d(512, 512, kernel size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
      (conv3): Conv2d(512, 2048, kernel size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(1024, 2048, kernel size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
      (conv2): Conv2d(512, 512, kernel size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
      (conv3): Conv2d(512, 2048, kernel size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
      (conv2): Conv2d(512, 512, kernel size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
      (conv3): Conv2d(512, 2048, kernel size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track running stats=True)
      (relu): ReLU(inplace=True)
    )
  )
)
(fpn): FeaturePyramidNetwork(
  (inner_blocks): ModuleList(
    (0): Conv2dNormActivation(
      (0): Conv2d(512, 256, kernel size=(1, 1), stride=(1, 1))
    )
    (1): Conv2dNormActivation(
```

```
        (0): Conv2d(1024, 256, kernel size=(1, 1), stride=(1, 1))
      )
      (2): Conv2dNormActivation(
        (0): Conv2d(2048, 256, kernel size=(1, 1), stride=(1, 1))
      )
    )
    (layer_blocks): ModuleList(
      (0): Conv2dNormActivation(
        (0): Conv2d(256, 256, kernel size=(3, 3), stride=(1, 1), padding=(1, 1))
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(256, 256, kernel size=(3, 3), stride=(1, 1), padding=(1, 1))
      )
      (2): Conv2dNormActivation(
        (0): Conv2d(256, 256, kernel size=(3, 3), stride=(1, 1), padding=(1, 1))
      )
    )
    (extra_blocks): LastLevelP6P7(
      (p6): Conv2d(2048, 256, kernel size=(3, 3), stride=(2, 2), padding=(1, 1))
      (p7): Conv2d(256, 256, kernel size=(3, 3), stride=(2, 2), padding=(1, 1))
    )
  )
)
(anchor_generator): AnchorGenerator()
(head): RetinaNetHead(
  (classification_head): RetinaNetClassificationHead(
    (conv): Sequential(
      (0): Conv2dNormActivation(
        (0): Conv2d(256, 256, kernel size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (1): GroupNorm(32, 256, eps=1e-05, affine=True)
        (2): ReLU(inplace=True)
      )
      (1): Conv2dNormActivation(
        (0): Conv2d(256, 256, kernel size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (1): GroupNorm(32, 256, eps=1e-05, affine=True)
        (2): ReLU(inplace=True)
      )
      (2): Conv2dNormActivation(
        (0): Conv2d(256, 256, kernel size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
        (1): GroupNorm(32, 256, eps=1e-05, affine=True)
        (2): ReLU(inplace=True)
      )
      (3): Conv2dNormActivation(
```

```
          (0): Conv2d(256, 256, kernel size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (1): GroupNorm(32, 256, eps=1e-05, affine=True)
          (2): ReLU(inplace=True)
        )
      )
      (cls_logits): Conv2d(256, 18, kernel size=(3, 3), stride=(1, 1), padding=(1, 1))
    )
    (regression_head): RetinaNetRegressionHead(
      (conv): Sequential(
        (0): Conv2dNormActivation(
          (0): Conv2d(256, 256, kernel size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (1): GroupNorm(32, 256, eps=1e-05, affine=True)
          (2): ReLU(inplace=True)
        )
        (1): Conv2dNormActivation(
          (0): Conv2d(256, 256, kernel size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (1): GroupNorm(32, 256, eps=1e-05, affine=True)
          (2): ReLU(inplace=True)
        )
        (2): Conv2dNormActivation(
          (0): Conv2d(256, 256, kernel size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (1): GroupNorm(32, 256, eps=1e-05, affine=True)
          (2): ReLU(inplace=True)
        )
        (3): Conv2dNormActivation(
          (0): Conv2d(256, 256, kernel size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
          (1): GroupNorm(32, 256, eps=1e-05, affine=True)
          (2): ReLU(inplace=True)
        )
      )
      (bbox_reg): Conv2d(256, 36, kernel size=(3, 3), stride=(1, 1), padding=(1, 1))
    )
  )
  (transform): GeneralizedRCNNTransform(
      Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225])
      Resize(min_size=(800,), max_size=1333, mode='bilinear')
  )


)
```

B.3 YOLOv8s

Index	From	n	Params	Module (Arguments)
0	-1	1	928	Conv [3, 32, 3, 2]
1	-1	1	18560	Conv [32, 64, 3, 2]
2	-1	1	29056	C2f [64, 64, 1, True]
3	-1	1	73984	Conv [64, 128, 3, 2]
4	-1	2	197632	C2f [128, 128, 2, True]
5	-1	1	295424	Conv [128, 256, 3, 2]
6	-1	2	788480	C2f [256, 256, 2, True]
7	-1	1	1180672	Conv [256, 512, 3, 2]
8	-1	1	1838080	C2f [512, 512, 1, True]
9	-1	1	656896	SPPF [512, 512, 5]
10	-1	1	0	Upsample [None, 2, 'nearest']
11	$[-1, 6]$	1	0	Concat [1]
12	-1	1	591360	C2f [768, 256, 1]
13	-1	1	0	Upsample [None, 2, 'nearest']
14	$[-1, 4]$	1	0	Concat [1]
15	-1	1	148224	C2f [384, 128, 1]
16	-1	1	147712	Conv [128, 128, 3, 2]
17	$[-1, 12]$	1	0	Concat [1]
18	-1	1	493056	C2f [384, 256, 1]
19	-1	1	590336	Conv [256, 256, 3, 2]
20	$[-1, 9]$	1	0	Concat [1]
21	-1	1	1969152	C2f [768, 512, 1]
22	$[15, 18, 21]$	1	2116435	Detect [1, [128, 256, 512]]

Table B.1: YOLOv8s Model Architecture

YOLOv8s summary: 225 layers, 11,135,987 parameters, 11,135,971 gradients, 28.6 GFLOPs

B.4 Modified YOLOv8-small

Index	From	n	Params	Module (Arguments)
0	-1	1	928	Conv [3, 32, 3, 2]
1	-1	1	18560	Conv [32, 64, 3, 2]
2	-1	1	29056	C2f [64, 64, 1, True]
3	-1	1	73984	Conv [64, 128, 3, 2]
4	-1	2	197632	C2f [128, 128, 2, True]
5	-1	1	295424	Conv [128, 256, 3, 2]
6	-1	2	788480	C2f [256, 256, 2, True]
7	-1	1	296192	SPPF [256, 512, 5]
8	-1	1	0	Upsample [None, 2, 'nearest']
9	$[-1, 4]$	1	0	Concat [1]
10	-1	1	180992	C2f [640, 128, 1]
11	[10]	1	410577	Detect [1, [128]]

Table B.2: Modified YOLOv8s-small Model Architecture

YOLOv8s-small summary: 120 layers, 2291825 parameters, 2291809 gradients, 18.2 GFLOPs

Appendix C

Predictions on Tree Dataset

Image	Ground Truth	SSD Prediction	RetinaNet Prediction	YOLOv8 Prediction
image_1	86	89	87	93
image_2	7	9	20	7
image_3	48	53	61	49
image_4	48	53	59	52
image_5	48	53	59	52
image_6	48	53	59	52
image_7	64	69	73	66
image_8	69	74	69	78
image_9	69	74	69	78
image_10	69	74	69	78
image_11	69	73	69	80
image_12	105	92	90	109
image_13	114	100	97	107
image_14	114	100	97	107
image_15	114	100	97	107
image_16	114	98	96	108
image_17	109	79	58	88
image_18	97	70	136	134
image_19	175	133	131	170
image_20	20	20	46	25
image_21	72	84	90	77
image_22	72	84	90	77
image_23	72	80	92	78
image_24	72	84	90	77
image_25	89	84	85	86
image_26	89	84	85	86
image_27	89	87	82	85
image_28	89	87	82	85
image_29	89	87	82	85
image_30	77	79	81	78
image_31	77	79	81	78
image_32	77	79	81	78
image_33	77	81	79	77
image_34	148	52	141	141
image_35	112	113	124	121
image_36	112	113	128	118
image_37	112	113	124	121
image_38	112	113	124	121
image_39	72	73	96	77
image_40	72	73	93	77
image_41	72	73	96	77
image_42	72	73	96	77
image_43	74	84	79	75
image_44	50	39	52	51
image_45	130	114	117	132
image_46	130	114	120	130
image_47	130	114	120	130
image_48	130	114	117	132
image_49	130	114	117	132
image_50	89	84	78	92
		MAE: 9.98	**MAE: 13.18**	**MAE: 5.44**

Table C.1: Comparison of model predictions with ground truth across SSD, RetinaNet, and YOLOv8s-normal models.