Міністерство освіти і науки України

Національний університет «Києво-Могилянська академія»

Факультету інформатики

Кваліфікаційна робота

освітній ступінь – бакалавр

на тему:

“Object detection model pruning with application to human recognition in UAV footage”

Виконав: студент 4-го року навчання,

Освітньої програми

«Комп'ютерні Науки», 122

Безбородов Владислав Павлович

Керівник ст. викладач Кузьменко Д.О.

Рецензент к.ф.м-н,____

Кваліфікаційна робота захищена

з оцінкою _____

Секретар ЕК _____

«____» _____ 2025 р.

Київ-2025

Міністерство освіти і науки України
Національний університет «Києво-Могилянська академія»

Факультет інформатики

Кафедра інформатики

ЗАТВЕРДЖУЮ

Зав.кафедри інфомратики,
доцент, к. фіз.-мат. наук

_____*Гороховський С.С.*

(підпис)

“____”_____ 2021

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ для
кваліфікаційної роботи
студенту 4-го курсу, факультету інформатики
Безбородов Владислав Павлович

Тема: «Метаевристичні алгоритми для обрізки нейронних мереж» Зміст кваліфікаційної роботи:

Abstract

1. Introduction
2. Fundamentals of object detection models
3. Model compression techniques
4. Experiments
5. Results
6. Conclusions

Дата видачі “ ____”_____ 2025 Керівник _____

(підпис)

Завдання отримав _____

(підпис)

Графік підготовки кваліфікаційної роботи до захисту

Графік узгоджено « _____» _____ 2025р.

Календарний план виконання роботи

Тема: Обрізка моделі виявлення об'єктів із застосуванням до розпізнавання людей на знімках з БПЛА

Календарний план виконання роботи:

№ п/п	Назва етапу дипломного проекту (роботи)	Термін виконання етапу	Примітка
1.	Отримання завдання на курсову роботу	15 жовтня 2024 р.	
2.	Аналіз предметної області	1 грудня 2024 р.	
3.	Опрацювання літератури	1 лютого 2025 р.	
4.	Виконання практичної частини	1 квітня 2025 р.	
5.	Написання текстової частини кваліфікаційної	20 травня 2025 р.	
6.	Оформлення презентації для захисту	25 травня 2025 р.	
7.	Передзахист кваліфікаційної роботи	28 травня 2025 р.	
8.	Захист кваліфікаційної роботи	5 червня 2025 р.	

Студент: Безбородов В. П

Керівник: Кузьменко Д.О.

“_____” _____

Table of Contents

Contents

1 Abstract

This thesis explores the application of the model optimization techniques in object detection field with a focus on human recognition from UAV footage. Limitations of resource constrained devices and deployment of accurate yet lightweight models is a challenging task. To address this, we examine three core optimization approaches: quantization, pruning, and knowledge distillation. Each method is investigated and applied in the context of YOLOv8-based detectors. Through experimental evaluation and comparative analysis with models trained from scratch, we demonstrate these techniques can significantly reduce model size and inference latency while preserving favorable performance.

2 Introduction

Over the past two decades, computational powers have advanced at an extraordinary pace, enabling breakthroughs in artificial intelligence. Deep Neural Networks (DNNs) have achieved noteworthy success in core computer vision tasks, including object detection, which has allowed the creation of numerous architectures with a range of complexity for different applications.

However, the power and accuracy of modern neural networks come at the cost of high computational problems and memory usage. This makes them computationally expensive and memory-intensive, which is challenging when deploying on resource-constrained edge devices. High-end GPUs or cloud servers can easily handle such models, but edge platforms like mobile phones, and unmanned aerial vehicles (UAVs) have limited processing capabilities and strict energy budgets. In particular, small drones cannot carry heavy hardware due to payload and power limitations as their compute capacity and power supply are relatively low.

Deploying a state-of-art object detector in real-time on a drone's embedded processor is extremely demanding and results in poor performance, namely slow inference speeds and increased latency, which might cause missed detections or ineffective real-time performance. For instance, Suo et al. [1] in their study benchmarked the YOLOv3 object detector on NVIDIA's Jetson Xavier NX and Nano platforms, achieving a throughput of 6.3 and 2 frames per second respectively with a 608x608 input. Such figures emphasize the gap between large DNN models and the processing capabilities of representative UAV hardware.

Moreover, complicated computations can quickly drain the limited battery of a drone, reducing its flight time. UAV imagery also presents challenging conditions such as small object scales, low image resolutions, and indirect angles which can require even more processing to achieve high accuracy. Although modern deep learning models propose high performance for human recognition and other computer vision tasks, their size and computation demand present an obstacle for deploying on edge devices such as drones.

To address the issue, this work will focus on model optimization techniques that can compress and accelerate neural networks without significant performance degradation. To be more concrete, we examine:

- Quantization that reduces the numerical precision of weights and activations to decrease memory usage and improve inference speed;
- Pruning, which removes redundant parameters in the model to create sparser, more sufficient networks;
- Knowledge distillation, a teacher-student method that transfers information from a large model to a smaller, lightweight one.

These approaches offer promising solutions for transforming powerful object detectors for real-time scenarios on edge devices. During this research, we apply the model optimization techniques to the problem of human recognition in natural landscapes from aerial footage, where speed and efficiency are crucial for success. The main objective of our work is to evaluate whether optimized versions of a standard detection model can outperform or match the performance of a smaller model trained from scratch. To this end, we select a reference architecture and conduct experiments to produce a set of compressed models using the above-mentioned methods.

The following work is organized as follows:

Section 3 provides a brief overlook of YOLOv8 as the model selected for our study and some enhancements that can be applied to it to have a better impact on the performance.

Section 4 describes the theoretical part of our research and explores model optimization techniques in detail – the intricacies of quantization, pruning, and knowledge distillation.

Section 5 reviews the experimental setup, selected datasets, hardware resources, and training flow where we discuss our ways of optimizing for great performance on a custom dataset.

Section 6 analyses the results of our findings. The conclusions are outlined in Section 7.

3 Fundamentals of Object Detection Models

Object detection is a fundamental task in the fields of computer vision and image analysis. Its goal is to identify and locate instances of specific object categories within digital images or videos Zhao et al. (2019) [2]. In the context of our work, the focus is on human recognition from aerial or UAV-captured footage. Object detection combines two essential computer vision problems: image classification, which assigns class labels to detected objects, and object localization, which determines their spatial positions within the frame. By integrating these capabilities, object detection enables the prediction of bounding boxes that encapsulate objects, along with the corresponding class to which each object belongs.

3.1 Evolution of Object Detection Models

We can categorize modern deep-learning object detectors into two paradigms such as two-stage and one-stage approaches. Two-stage detectors first generate region proposals and then classify each region. For example, R-CNN designed by Ross Girshick, et al.[3] pioneered this method using selective search to generate approximately 2000 region proposals and then applied a CNN to each proposal for classification and bounding box refinement. R-CNN performed well but was very slow. Ross Girshick further in his next research[4] introduced Fast R-CNN improving speed by running convolutional feature extraction on the full image once and then pooling features for each region. Later Faster R-CNN[5] was developed to integrate a Region Proposal Network (RPN) using CNN as a tool for generating proposals. These two-stage models set state-of-art accuracy benchmarks at the cost of heavier computation.

In contrast, one-stage detectors perform localization and classification in a single pass without a separate proposal stage. The YOLO (You Only Look Once) series [6] illustrates this approach. As Hrishitva Patel described in the research [7] YOLO splits the image into a grid of cells and directly predicts bounding-box coordinates and class probabilities for each cell. SSD (Single Shot MultiBox Detector) [8] similarly predicts boxes and scores at multiple scales from convolutional feature maps. As they eliminate the

region-proposal stage, these one-stage models have better performance in terms of real-time inference speed and latency, enabling YOLO and SSD models to export on budget platforms. Initially, one-stage detectors trailed two-stage methods in precision, but new algorithms and innovations have narrowed the gap. As a result, modern one-stage detectors like YOLOv8 and improved SSD variants now offer competitive accuracy with high throughput, making them perfect for UAV-based detection.

To better comprehend the evolution of object detection models from conventional approaches to modern architectures, Figure 1 provides a chronological visual overview.

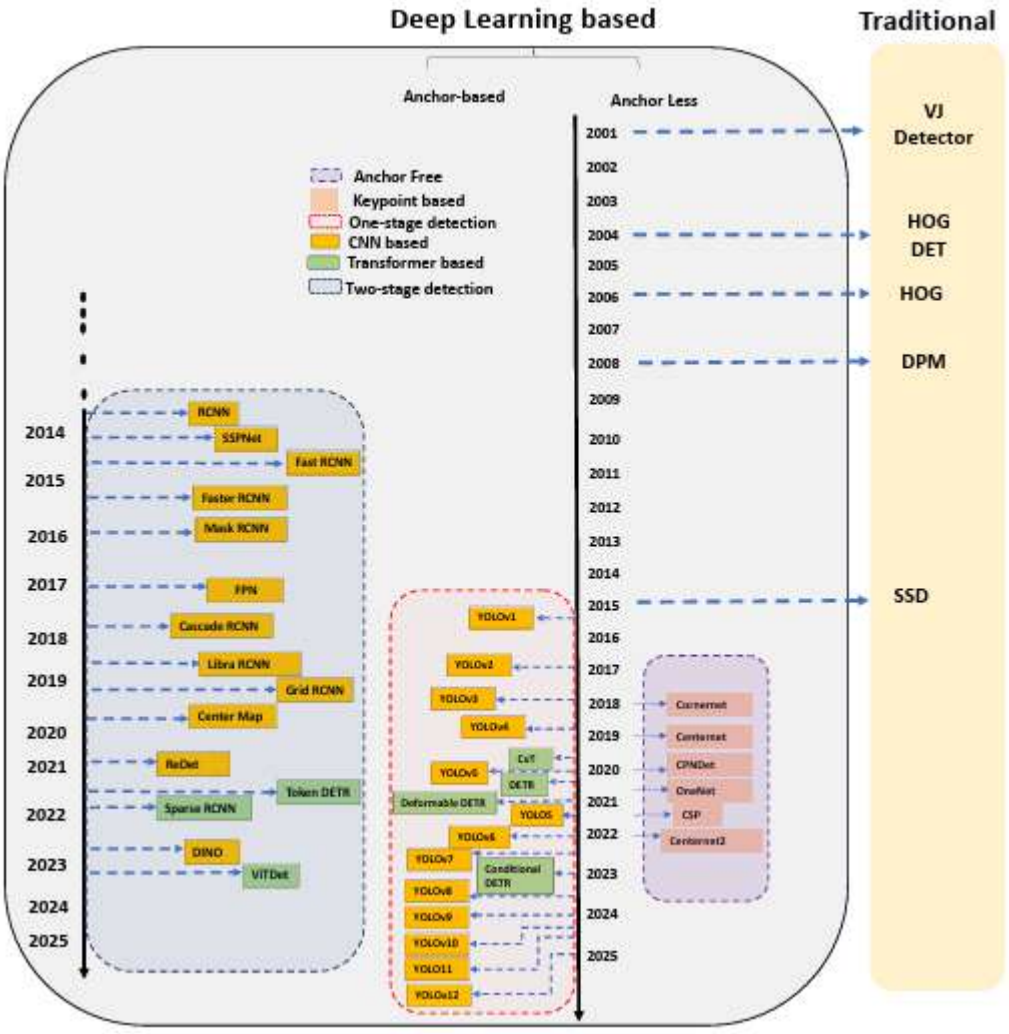


Figure 1. Timeline of object detection model evolution from traditional methods to deep learning-based approaches [9]

3.2 YOLOv8

YOLOv8 is a recent iteration of YOLO family object detectors, developed and released by Ultralytics in 2023. Ranjan Sapkota, et al. [9] describe in their research the model as a user-friendly solution designed in PyTorch making it accessible for a variety of applications. From an architecture standpoint, Ranjan Sapkota, et al. [11] discuss the novelties of YOLOv8 that involve three main components: a Backbone, a Nech, and a Head. For the backbone, they enhanced CSP-based modules for spatial feature extraction from the input image. The neck of YOLOv8 incorporates a simplified Feature Pyramid Network (FPN), which strengthens the model's ability to capture features at multiple spatial scales. This is particularly important for detecting objects of different sizes within the same image.

Furthermore, YOLOv8 employs a decoupled head structure, where object classification, bounding box regression, and objectness scoring are handled separately. This separation improves accuracy while maintaining computational efficiency. Overall, the architecture is designed to be lightweight and effective, offering a strong balance between detection speed and precision. These characteristics make YOLOv8 suitable for real-time applications on resource-limited platforms, as shown in Figure 2.

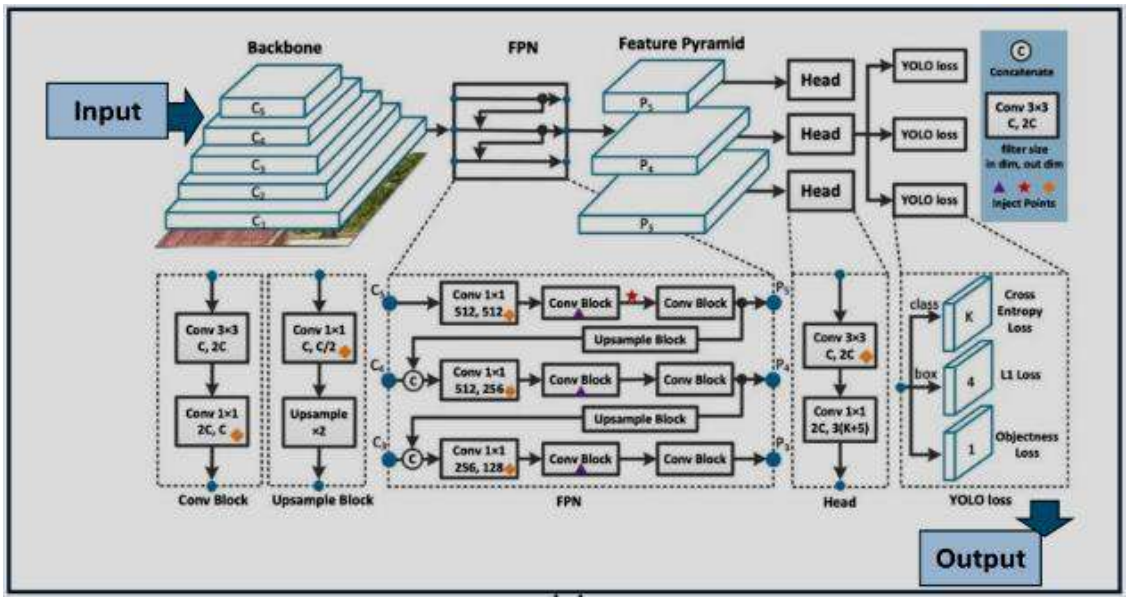


Figure 2. YOLOv8 architecture featuring a CSP-based backbone, streamlined FPN, and anchor-free decoupled head for efficient multi-scale object detection [11]

Although contemporary models such as YOLOv11 and YOLOv12 are superior to the model we selected for our investigation, YOLOv8 remains a strong and practical choice for object detection applications. One of the main advantages is its widespread integration and proven robustness. Since the release of the model, YOLOv8 has been extensively studied, tested, and deployed in various real-world scenarios. The fallout is a rich ecosystem of pre-trained models, public benchmarks, and integration tools, which significantly reduce development and experimentation time. On top of that, YOLOv8 delivers a favorable trade-off between detection accuracy and inference speed, making it suitable for deployment on edge devices.

However, there are noticeable limitations to selecting YOLOv8. Newer models introduce innovative architectural solutions leading to superior precision and efficiency. Therefore, while YOLOv8 remains a reliable and performant baseline, it might no longer represent the forefront of object-detection models.

3.3 CBAM for YOLOv8

Convolutional Block Attention Module[12] is a lightweight attention module that can improve CNN feature representation. Sequentially, CBAM applies two sub-modules: channel attention and spatial attention modules. As illustrated in Figure 3, given an intermediate feature map $F \in R^{CxHxW}$, CBAM proceeds in two stages :

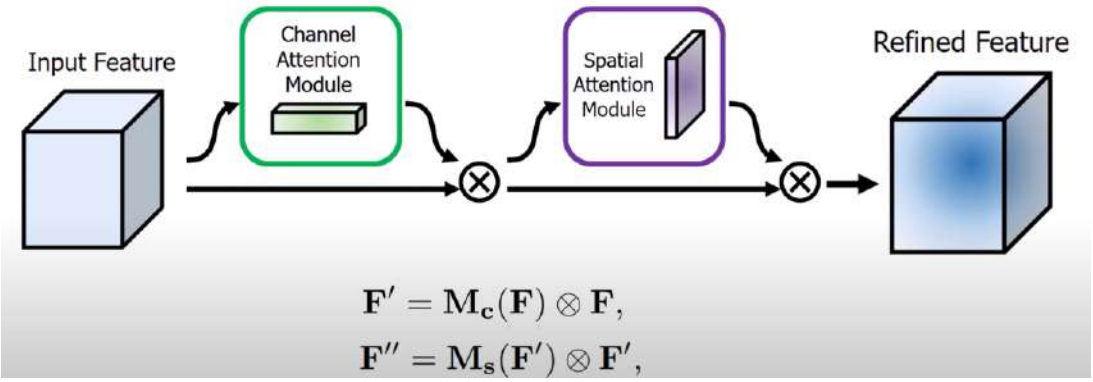


Figure 3. The CBAM architecture

- Channel Attention

- Pooling: Two descriptors of shape R^c are computed using average-pooling and max-pooling over the spatial dimensions.
- Shared MLP: A two-layer perceptron passes each descriptor to produce per-channel weight vectors.
- Fusion: The two vectors are summed and flattened by the sigmoid function to yield channel attention weights $M_c \epsilon [0,1]^C$.
- Refinement: The input map is reweighted by $M_c : F' = M_c \otimes F$

- Spatial Attention
  - Pooling: Max- and average-pooling generate two feature maps of shape $R^{1\times H\times W}$ across the channel dimension of F'.
  - Concatenation and Convolution: These two maps are concatenated along the channel axis and convolved with a 7×7 kernel to produce a spatial attention mask $M_s \epsilon [0,1]^{H\times W}$.
  - Refinement: The output of the channel stage is further reweighted spatially: $F'' = M_s \otimes F'$.

In a nutshell, the channel attention map indicates what feature channels are most important. The spatial attention module then pools the input features along the channel axis to generate two 2D descriptors, concatenates them, and applies a convolution to produce a 2D spatial attention map. This spatial attention map demonstrates where in the features map to focus. As CBAM is end-to-end trainable, it can be integrated into any convolutional backbone with an insignificant impact on inference cost. Recent studies conducted such experiments, obtaining measurable accuracy gains[13].

4 Model Compression Techniques

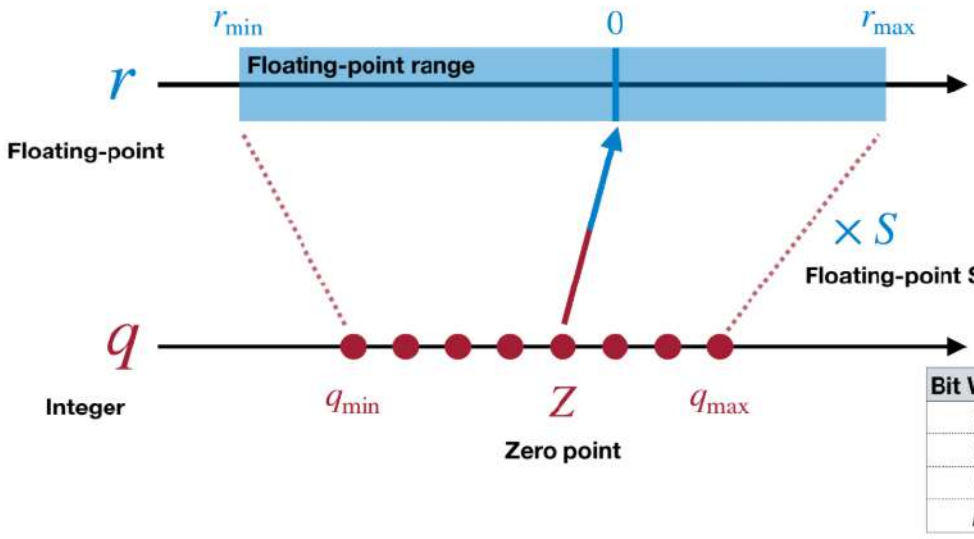
In the context of deploying object detection models on resource-constrained hardware such as drones, model optimization techniques have emerged as essential tools to reduce memory usage, inference time, and energy consumption with a slight accuracy degradation. In this section, we present a deep dive into three prominent approaches in the field, examining each from a theoretical perspective.

4.1 Quantization

Linear Quantization is a powerful optimization technique that reduces the numerical precision used to represent weights and activations in a neural network, thereby reducing this we can unlock model size reductions and accelerate inference. In practice, 32-bit floating point weights and activations are mapped to lower-bit representations, usually 8-bit integers, while trying to preserve the model's accuracy. The core idea is to exploit the efficiency of integer arithmetic as operation on low bit integers can execute faster and require less memory bandwidth, leading to a smaller model size and enhanced inference latency. Formally, we can describe quantization as an affine mapping of integers to real numbers[14]:

$$r = S(q - Z) \tag{1}$$

where r is the 32-bit float weights , q is the quantized to integer weights, S is a scaling factor that scales a dynamic range, and Z is the zero-point, quantization parameter that allows real number when $r = 0$ be exactly representable by a quantized integer Z, so Z will be mapped to zero.



Bit Width	q_{min}	q_{max}
2	-2	1
3	-4	3
4	-8	7
N	-2^{N-1}	$2^{N-1}-1$

Figure 4. Illustration of linear quantization, mapping the floating-point range onto the integer range[15]

Figure 4 demonstrates key parameters of quantization, including:

- Real-value bounds

We have a floating-point dynamic range elements lie in $[r_{min}, r_{max}]$ highlighted in blue. The goal is to map $r_{min} \rightarrow q_{min}$ and $r_{max} \rightarrow q_{max}$ exactly

- Integer bounds

For an N-bit signed two's-complement representation, the allowable integer values are $q_{min} = -2^{N-1}, q_{max} = 2^{N-1} - 1$. These fixed endpoints define integer's dynamic range shown in red

- Scale factor

Fulfilling these endpoint correspondences gives a scale:

$$S = \frac{r_{max} - r_{min}}{q_{max} - q_{min}} \quad (2)$$

Intuitively, S defines how many real-value units match one integer step.

- Zero Point

To ensure that the real value $r = 0$ aligns with an integer code $q = Z$, we compute the zero-point with given equations (1) and (2):

$$Z = round(q_{min} - \frac{r_{min}}{S}) \quad (3)$$

This guarantees that when $q = Z$, the reconstructed real value is 0.

The quantized affine mapping is most frequently used in linear quantized fully-connected and convolutional layers. In a fully-connected layer, the output is computed by performing matrix multiplications, accumulating into 32-bit integer registers to prevent overflow, adding a quantized bias, and then applying final fixed-point rescale followed by the output zero-point. A quantized convolutional layer follows the same strategy but replaces matrix multiplications with integer convolution operations.

Building upon the computational foundation, we would also like to introduce the practical application of quantization. One of the widely operated approaches is Post-Training Quantization [10]. This is mostly wielded by Convolutional Neural Networks. The workflow core lies in the following: we modify the model, calibrate it based on some dataset, and then convert from the calibrated model to a quantized one to get high performance. At a lower level of abstraction, this process involves quantizing weights and activations either for the entire model or for selected sub-modules. This can be achieved using integrated observers into the network, storing the statistics of spectated parameters and activations during the calibration stage. These statistics are then used to identify appropriate scale and zero point values required for thriving quantization. As a result, it tends to provide faster compute also uses less memory both for runtime and storage. his process is shown in Figure 5, where the calibration and quantization phases are shown as distinct stages of the model transformation.

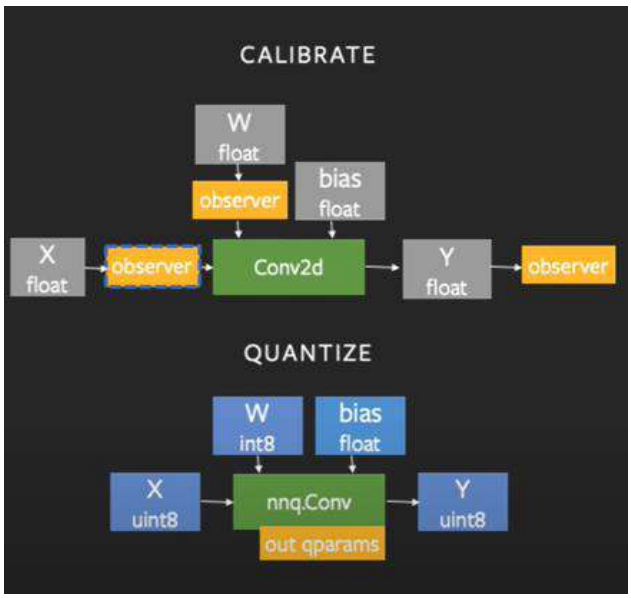


Figure 5. Workflow illustrating the calibration of floating-point layers using observers (top) and the subsequent transformation into an integer-based inference model (bottom)

4.2 Model Pruning

Having reviewed the first approach of model optimization, we would like to introduce neural network pruning which reduces the parameters number of neural networks by more than 90%, decreasing the storage requirements and improving computation efficiency of neural networks. If quantization manipulates with the data types of weights and activations, model pruning operates directly with them using different methods how to remove them. We will go through all stages of pruning and establish different granularities and criterions of this model compression technique.

The core idea of pruning lies in making neural network smaller by removing synapses and neurons into sparse neural network. Figure 6 demonstrates the overview of this process intuitively.

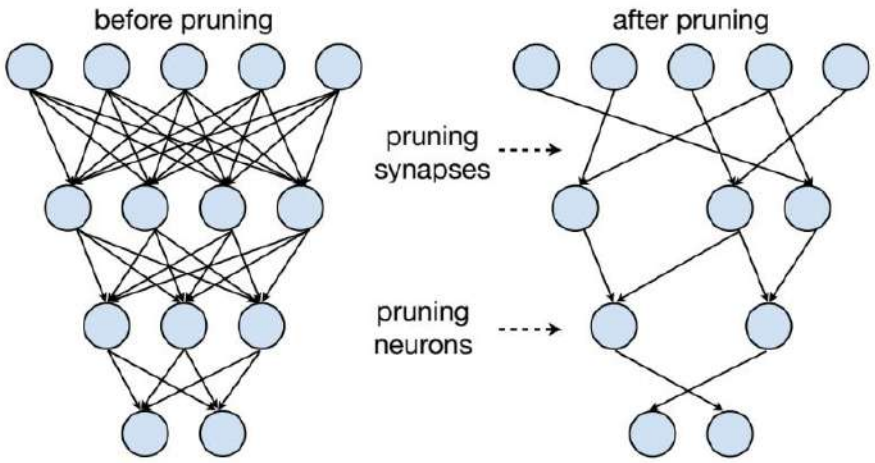


Figure 6. Neural network pruning[15]

First, we train a model to convergence to recognize which neuron is either important or redundant. Then we prune away some of the connections and the more we prune, the more accuracy drop appears. Before pruning, the model usually forms roughly normal distribution of the weights and after pruning we are removing small parameters. We examine if the weights absolute value is less than the certain threshold than we remove those connections. Therefore, more cropping the model, sparser the model becomes with a larger gap of weights distribution.

When the accuracy after performing this approach decreases significantly we can apply post-training to our remaining weights to recover precision. Furthermore, it is possible to apply iterative pruning and training of survived weights, to reduce volume of the model even more without accuracy degradation.

In general, we could formulate the pruning as follows:

$\arg\min L(x; W)$ – standard neural network approach

$\arg\min L(x; W_p)$ – pruned neural network

subject to $||W_p||_0 < N$

where L represents the objective function for neural network training; x is input, $W$is original weights, W_p is pruned weights; $||W_p||_0$ calculates the non zeros in W_p and $N$is the threshold

Neural network pruning can be applied in different patterns or granularities with a certain trade-off to balance the number of weights we pruned away and boosting speed we achieve with the hardware.

Figure 7. Example of fine-grained pruning[15]

Figure 7 presents purely unstructured pruning[16] where we prune away some of the weights which is in white color after pruning. It is a flexible approach that enables removing whichever weight. However, it is challenging to accelerate on GPUs, as it is very irregular in order to store this matrix. We have to preserve not only the matrix weights, but also their positions using different formats. The advantage of the method lies in full flexibility to decide which weight to prune, therefore, the pruning ratio is high. If the target of the research is just to reduce the model's size, it can be an attractive approach.

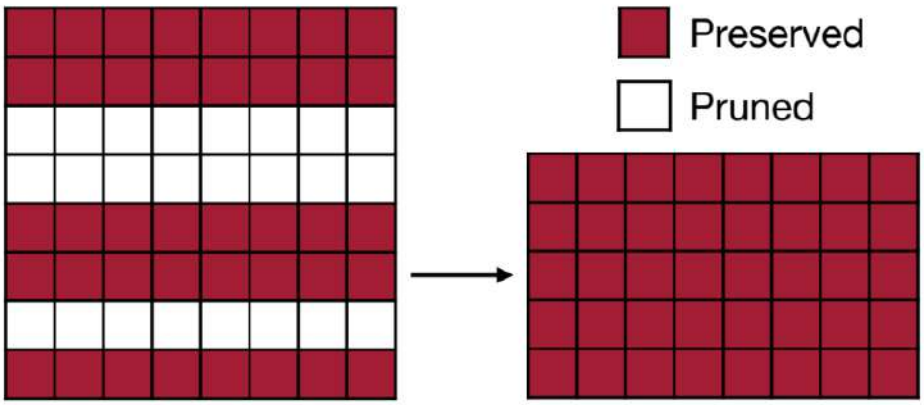


Figure 8. Example of coarse-grained pruning[15]

On the other hand, structured pruning follows a different pattern. We prune away entire rows, columns, so that we can condense weight matrix to a smaller size as illustrated in Figure 8. As a result, the actual computation reduces enabling acceleration on the hardware. Nevertheless, this approach has certain downsides such as its rigidity which can lead to significant accuracy degradation.

Last, but not least, the method we focus on is channel pruning - one of the most widely used strategies for speeding up inference on edge devices. Formally, entire channels are pruned what results in direct acceleration due to reduced channel numbers as the neural networks possess smaller channels. The disadvantage of this method is its relatively low compression ratio, which is typically between 10% and 30%, compared to higher ratios (e.g., 3× to 10×) in other approaches. In addition, compared to a uniform reduction in all channel sizes, channel pruning provides better performance when the compression ratio is optimized individually for each channel rather than applying a fixed ratio for all layers. Figure 9 exemplifies this concept by estimating uniform shrink and uneven pruning, where each layer is cut to an individual ratio.

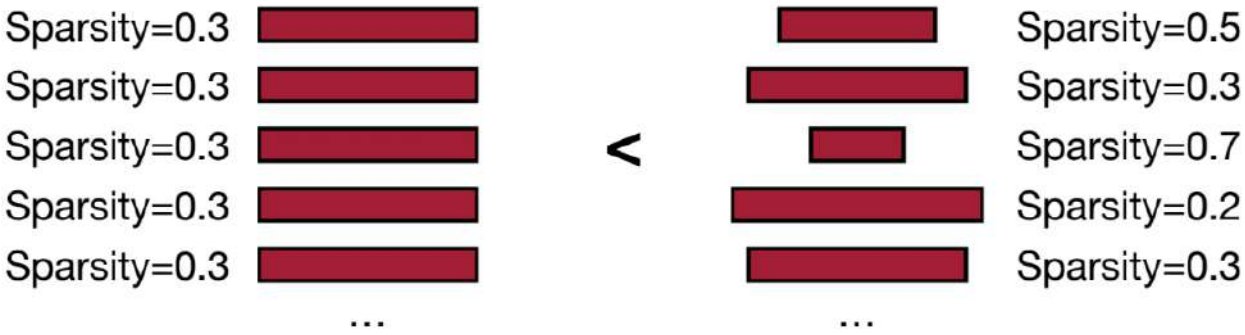


Figure 9. Comparison of channel pruning with constant sparsity(left) with non-uniform(right)[15]

4.3 Knowledge Distillation

Last model optimization technique explored in our work is knowledge distillation. It addresses the question: "How can we train small model to run efficiently on tiny edge devices with the assistance of larger model?" So, the core idea lies in exploitation of complex neural network architecture as a teacher to train a lightweight model, our student[17].

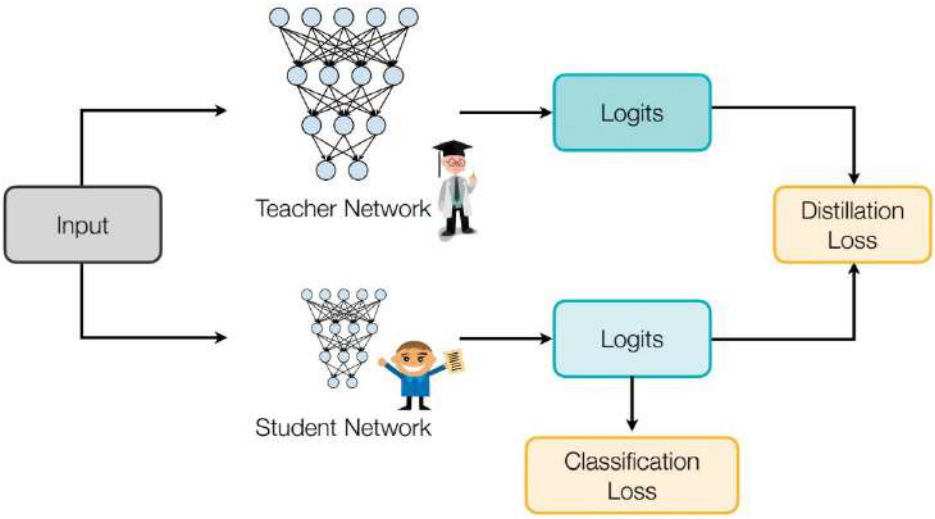


Figure 10. Knowledge distillation[15]

Figure 10 depicts the structure of this approach. Given an input sample, both the pre-trained teacher network and the smaller student network generate logit vectors. In addition to the standard classification loss calculated on the student's logits, a distillation loss term is introduced to encourage the distribution of the student's output to match that of the teacher. Although the student network can correctly predict the correct class, its confidence is usually lower than that of the teacher. Therefore, distillation loss 'boosts' the student's prediction probabilities, minimizing the discrepancy between the student's and teacher's softened distributions of the input data.

To enable this smoothing, temperature parameter T is added to the SoftMax function. For a logit z_i corresponding to a class i, the smoothed probability is defined as:

$$P(z_i, T) = \frac{\exp(z_i/T)}{\sum_j \exp(z_j/T)} \quad (4)$$

Here $i, j = 0,1,2, \ldots, C-1$, where C is the number of classes. T is the temperature, which is normally set to 1. Meaning of the formula: for different class with a different temperature the probability $\exp(z_i/T)$ for the particular index is divided by $\sum_j \exp(z_j/T)$

When T=1, this reduces to the standard SoftMax; as T increases, the probability distribution becomes smoother, thereby revealing more information about the relative similarity between all classes. During training, the teacher and student logits are divided into the same temperature T before applying SoftMax. Then, the distillation loss is calculated as the cross-entropy between these two smoothed distributions. In this way, the student network learns not only to assign high probabilities to the correct class, but also to mimic the full distribution of the teacher's output, improving generalization and allowing the smaller model to approach the teacher's performance. Summarizing, the goal of knowledge distillation is to align the class probability distributions from teacher and student networks.

Moving on, we focus on intermediate features we can match between teacher and student networks. One of the simplest approaches refers to matching output logits. As figure 10 demonstrates after we obtained prediction outputs, our output logits, we can counterpart them, using Cross entropy loss $E(-p_t \log p_s)$ or L2 loss $E(||p_t - p_s||_2^2)$. Beyond matching outputs, we can also match intermediate tensors as depicted in figure 11.

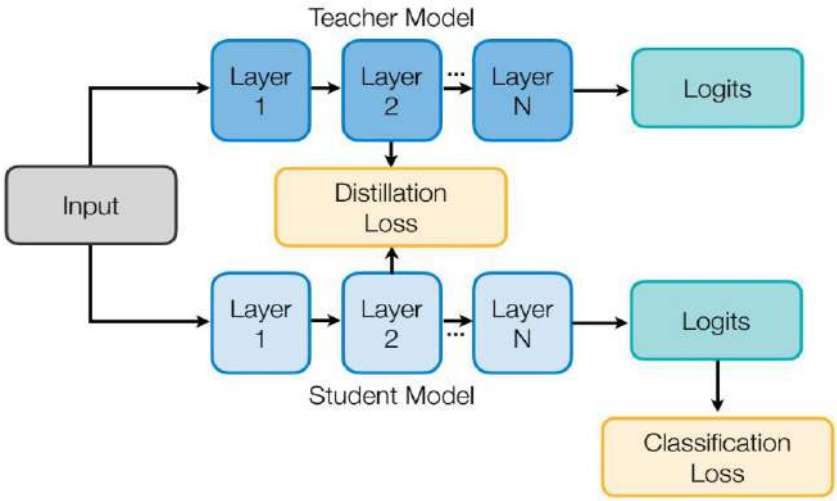


Figure 11. Aligning intermediate layer tensors from the student to the teacher

Using this approach we can pair the intermediate weights. The challenge of the method consists in different tensor dimensions, Since the teacher model is large and the student model is tiny, the incompatibility occurs, and a direct matching is not possible. To address this, we introduce a learnable projection that maps the student's weight tensor into the teacher's weight space before computing L_1 or L_2 loss. It can be formulated as follows[18]:

$$W_S^* = argmin(\ L_T(W_S, W_r)\ ) \tag{5}$$

where W_S is the student's layer weights; W_r is a small projection network that maps the student's guided-layer weights into the same space as the teacher's layer; the loss $L_T(W_S, W_r)$ measures how far the student's projected activations are from the teacher's outputs.

Another instance of intermediate that we can match are the features. The intuition lies in that teacher and student networks should have similar features distributions, not only probability distributions.

Next matching method is associated with the gradient[19]. During training of the model backpropagation performs two heavy operations, namely, computing the weight gradient and activation gradient. To match the former is a very challenging task to perform, however, it is possible to execute with the latter. To be concrete, we counter the activation gradient for the first layer for the input, since teacher and student share it. Therefore, we can compute the attention of a CNN map x that is defined as $\frac{dL}{dx}$, where L is the learning objective. If $\frac{dL}{dx_{i,j}}$ is large, a small disturbance at i, j will significantly affect the final output. As a result, the network is putting more attention to the position i, j.

After accessing matching output logics, we would like to introduce to channel-wise knowledge distillation, departs from traditional approaches to spatial alignment by focusing on distributions encoded in each individual feature channel rather than on the correspondence of pixel-based feature activations. In their work, Shu et al. [20] note that for dense prediction tasks such as semantic segmentation, many convolutional channels act as implicit saliency detectors — the activations of each channel tend to highlight regions

corresponding to certain semantic categories. By normalizing the $H \times W$ activation map of each channel using SoftMax over spatial locations, the method transforms raw feature responses into probability maps that highlight the most informative spatial regions for that channel. Instead of imposing pointwise or pairwise similarity between teacher and student feature maps, channel distillation minimizes the divergence between these channel-wise probability distributions.

A distinctive feature of channel distillation is its ability to implicitly transfer the teacher's attention patterns at the level of individual semantic detectors. Instead of forcing the learner to accurately copy the high or low values of the teacher's feature map at individual points, channel distillation encourages the learner to create similar spatial heat maps for each feature channel. Since each channel often corresponds to the detection of different object boundaries or textures, this alignment ensures that the student learns to pay attention to the most salient regions for each semantic concept.

5 Experiments

5.1 Datasets

For the practical survey we chose 2 main datasets[21][22] from Kaggle and Roboflow Universe merging them into big single unit[23]. The datasets were selected correspondingly to our objective and stored only class “human”. Figure 12 depicts a sample of the dataset. After data cleaning process and joining the total number of the images contained over 20,000 images. Then we split the dataset into train, valid, and test sub-folders with 0.7, 0.2 and 0.1 ratios, accordingly.

Figure 12. Dataset test image

5.2 Resources

All trials were conducted both locally and virtually. For the former environment we used PyCharm for IDE and GPU NVIDIA GeForce GTX 1070 8192MiB. For the latter environment – Kaggle Notebooks for experiments and for GPU Tesla P100-PCIE16GiB.

5.3 Model Selection

In previous sections we already covered the model we chose for our experiments – YOLOv8. Ultralytics YOLOv8-x[24] trained on COCO[25] achieves 53.9 mAP and has

68.2 M parameters. Due to its reliable performance and deep researched architecture it is a great candidate for our investigation goals. We will large architecture using afore-mentioned techniques and compare with YOLOv8-m 50.2 mAP, 25.9 parameters and YOLOv8-s 44.9 mAP and 11.2 parameters to discover which approach is more relevant – pruning the model or training from scratch.

5.4 Evaluation metrics

Essential indicators we will take into account are mean average precision with different IoU (Intersection over Union) value, inference time[26] estimated on GPU and model's size. Additionally, we will track how long it takes to perform each stage of the experiments to compare in the results section.

5.5 Training Flow

5.5.1 Transfer learning

Before we start investigating how model optimization techniques affect model's performance we have to prepare the initial baseline model we will utilize. Although there are already pretrained models available by Ultralytics, but they are based on COCO dataset what has multiple irrelevant classes for our objective. Therefore, we decided first to find the YOLOv8 model on VisDrone[27] dataset, one of the most corresponding sources to assemble our model for the main task. Finally, having obtained the appropriate pretrained YOLOv8x model[28] and having integrated the CBAM module into the architecture to enhance the performance we initiate our training.

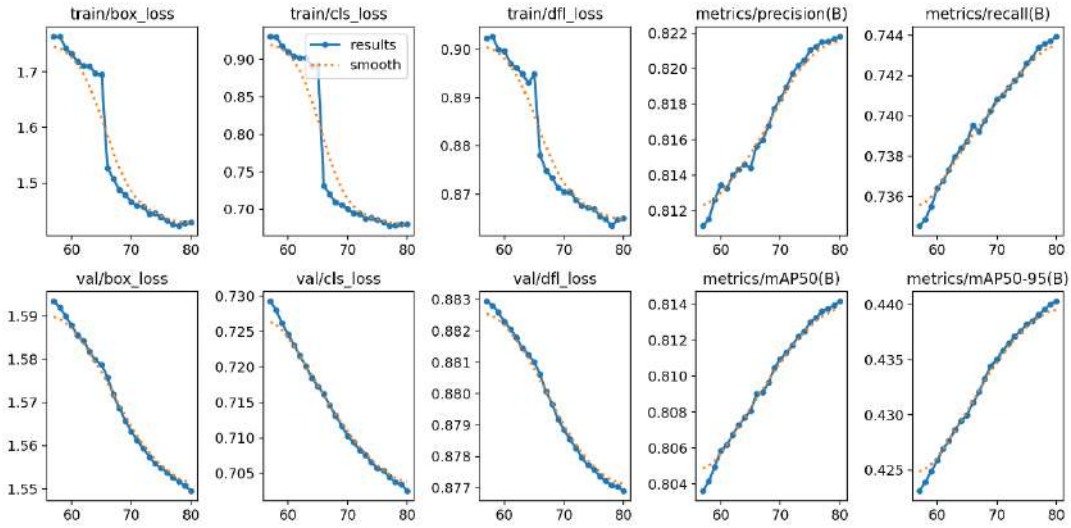


Figure 13. YOLOv8x with pretrained weights results

As illustrated in figure 13 the training curves show stable convergence of YOLOv8x on a custom-built dataset, with a consistent decrease in loss during training and validation for all components. Accuracy and reproducibility improve with each epoch, and mAP50 and mAP50-95 increase smoothly, indicating effective fine-tuning and stable generalization, reaching over 75% of accuracy threshold. Depicted in the table 1 results of the baseline model are as follows:

Model name	mAP@50(%)	mAP@50-95(%)	Speed GPU(ms)	Training time(hours)	Model size (MB)
Yolov8x	78.9	38.6	42.4	~40	136.7

Table 1. YOLOv8x baseline model results

5.5.2 Training models from scratch

Before deep-diving into the specifics of each model optimization technique's application we will train YOLOv8m and YOLOv8s for final comparison with other models. In table 2, indicators we will consider after finishing model compression stage. The values are the following:

Model name	mAP@50(%)	mAP@50-95(%)	Speed GPU(ms)	Training time(hours)	Model size (MB)
Yolov8m	52.1	17.8	14.0	~24	52.01

Yolov8s	**63.9**	**24.4**	**8.2**	**~12**	**22.51**

Table 2. YOLOv8m and YOLOv8s comparison

What captivating was that the smaller model performed significantly better than the larger in terms of architectural depth and number of parameters. The accuracy difference is more than 10%, the inference time for YOLOv8s model is better as well, so are the other parameters.

5.5.3 Quantization

For quantization, we use ONNX and OpenVINO tools, each of which offers separate mechanisms for reducing model accuracy. Using the ONNX Runtime quantization toolkit, we apply static quantization after training directly to the exported YOLOv8x model. This process converts the model's 32-bit floating-point weights and activations to 8-bit integers using a calibration dataset, reducing the model size and computational requirements while maintaining accuracy. After quantization in ONNX, we further optimize the model using OpenVINO's post-training optimization tool. OpenVINO provides hardware-aware optimization for INT8 inference, enabling improved performance on Intel architectures. It supports fine-grained control over quantization parameters and includes a precision-aware pipeline to ensure minimal quality degradation, making it ideal for edge deployment scenarios. Table 3 provides comparison of those methods and we can conclude that OpenVINO implementation performs slightly better in terms of obtained indicators:

Model name	mAP@50(%)	mAP@50-95(%)	Speed GPU(ms)	PostTraining time(hours)	Model size (MB)
ONNX	78.3	28.4	860.2	~5	272.66
OpenVINO	**78.4**	**29.3**	**594.8**	**~1**	**66.8**

Table 3. Comparison of Quantization approaches

5.5.4 Pruning

For pruning, we use the built-in PyTorch module **torch.nn.utils.prune**, which provides tools for both unstructured and structured pruning. This library enables to directly modify model weights by applying masks to specific layers, enabling flexible experimentation with different thinning strategies.

5.5.4.1 Unstructured pruning

Unstructured pruning in PyTorch is the process of removing individual weights from a neural network based on certain criteria, most often their magnitude. For example, using functions such as **prune.l1_unstructured**, you can prune a fixed proportion of the weights with the smallest magnitude in selected layers, such as convolutional or linear layers. The result is a weight matrix with a large number of zero elements, while the overall structure of the network remains unchanged. Although this approach allows for a high degree of compression, the resulting irregular matrix structure limits performance on standard hardware unless special libraries for computing irregular matrices are used.

5.5.4.2 Structured pruning

In contrast, structured pruning in PyTorch removes entire filters, channels, or neurons, effectively modifying the model architecture. This is done using **prune.ln_structured**, which allows pruning along specific dimensions, such as the output channels of a convolutional layer. Unlike unstructured pruning, the latter changes the shape of the network itself, creating a more compact and hardware-friendly model. Although it may not achieve the same level of compression as unstructured pruning, it results in a noticeable increase in inference speed and memory efficiency on most hardware platforms.

Before examining the results depicted in table 4, we would like to outline that during experiments we discovered that applying pruning in the very first convolutional layer or in the detection head of the architecture might lead to significant accuracy drop. To be concrete, model's precision decreased drastically from 78% of accuracy to 3%

Model name	mAP@50(%)	mAP@50-95(%)	Speed GPU(ms)	PostTraining time(hours)	Model size (MB)
unstr_0.10	78.7	38.0	32.4	~5	133.2
unstr_0.15	78.4	37.9	32.4	~5	133.2
unstr_0.20	78.5	37.9	32.4	~5	133.2
unstr_0.25	**78.5**	**37.9**	**32.4**	**~5**	**133.2**
unstr_0.3	78.5	37.8	32.4	~5	133.2
unstr_0.4	78.4	37.8	32.4	~5	133.2
unstr_0.5	78.2	37.8	32.4	~5	133.2
str_0.10	**78.0**	**37.4**	**32.4**	**~5**	**133.2**
str_0.15	77.7	36.9	32.4	~5	133.2
str_0.20	77.3	36.4	32.4	~5	133.2
str_0.25	77.0	36.1	32.4	~5	133.2
str_0.3	76.7	35.5	32.4	~5	133.2
str_0.4	75.5	34.4	32.4	~5	133.2
str_0.5	74.0	32.7	32.4	~5	133.2

Table 4. Comparison of pruning application

Reviewed, we can underline among unstructured variants, the model with pruning ration in 25% is chosen as most performant model, maintaining the same inference speed and model size, and representing the highest sparsity still preserving almost full accuracy. In contrast, for structured pruning the best trade-off appeared for model with 10% pruning ration with slight precision degradation. These both model are selected into final table observation alongside quantised and distilled versions.

Figure 14 shows that up to approximately 30% pruning, unstructured pruning has almost no effect on mAP, while structured pruning begins to degrade accuracy by 1–2%. Above 30%, the accuracy of structured pruning continues to decline more rapidly (almost -5% at 50% pruning), while unstructured pruning still maintains close to its initial performance at very high sparsity levels.

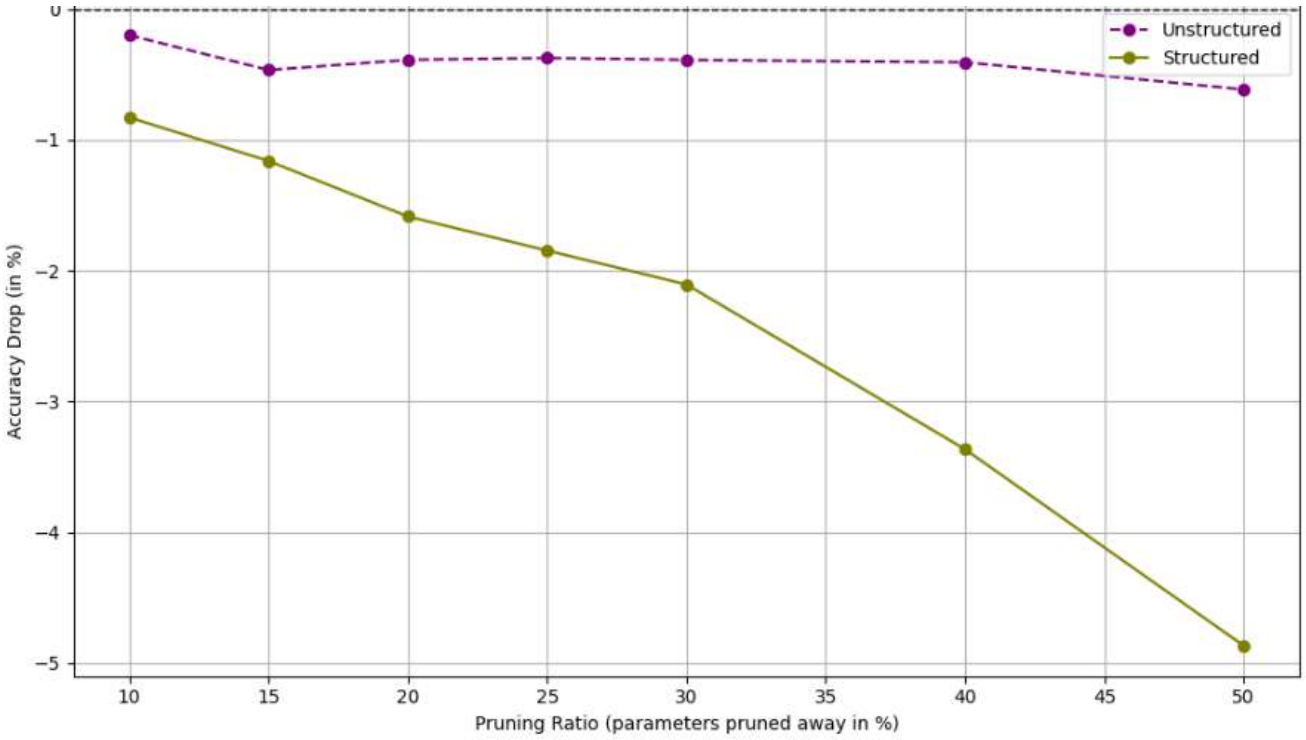


Figure 14. Pruning impact on YOLOv8x accuracy

5.5.5 Knowledge distillation

Implementing our final model optimization technique we employed the open-source repository which extends Ultralytics with modular teacher-student training framework. For this experiment we used our baseline model YOLOv8x as a teacher and for student we selected YOLOv8s in order to compare with a trained one.

The framework supports several distillation implementations, including channel-wise distillation, where feature maps from chosen layers are matched not only spatially, but also along the channel dimension. This allows the student model to learn detailed activation patterns that improve feature representation. Temperature and weight loss scaling could be adjusted through configuration, ensuring seamless integration.

The results of teacher-student training are illustrated in the table5.

Model name	mAP@50(%)	mAP@50-95(%)	Speed GPU(ms)	Training time(hours)	Model size (MB)
yolov8x_kn	68.9	29.3	7.6	~12	22.51

Table 5. Distillation knowledge results

6 Results

In this section, we summarize the results of each stage of our experiments and provide a generalized comparison of all final models in Table 6.

Model name	mAP@50(%)	mAP@50-95(%)	Speed GPU(ms)	(Post)Training time(hours)	Model size (MB)
yolov8x	78.9	38.6	42.4	~40	136.7
yolov8_quant	78.4	29.3	594.8	~1	66.8
yolov8_unstr	78.5	37.9	32.4	~5	133.2
yolov8_str	78.0	37.4	32.4	~5	133.2
yolov8_kn	**68.9**	**29.3**	**7.6**	**~12**	**22.51**
yolov8s_scratch	63.9	24.4	8.2	~12	22.51

Table 6. Final results of model optimization techniques

Among all the compressed variants of YOLOv8x, the model based on knowledge extraction offers the best compromise: its mAP@50 and mAP@50-95 scores exceed those of YOLOv8 trained from scratch by approximately 5 percentage points and show a slight improvement in inference speed. Other approaches performed well with minimal accuracy degradation, especially at higher IoU thresholds.

Unfortunately, in our practical survey the quantization method showed drastically bad performance in terms of inference speed, increasing more than 10 times. The possible reason might lie in OpenVINO's default int8 kernels that are often optimized for CPU or for Intel GPUs, not necessarily for NVIDIA GPUs. As a result, OpenVINO ends up executing many operations in software or falling back to slower code paths. In other indicators quantization cuts the model significantly with minimal accuracy losses, what makes this method still considerable.

Structured and unstructured pruning showed one of the best performances in terms of preserving of all parameters beating baseline model at every aspect. However, following our objective pruning approach loses to distilled model in inference time with difference of 25 milliseconds. Nonetheless, the technique demonstrates excellent achievements.

7 Conclusions

In this thesis, we explored the application of model optimization techniques to object detection with the objective of employing the approaches to enhance model's performance in the field of human recognition from UAV footage. We comprehended how each application functions from the inside and provided comparative analysis to address one question: "*Can compressed large architectures outperform small trained from scratch models?*" As a result, we can confirm that implementing optimization methods such as quantization, pruning and knowledge distillation is worth it if the task expects on edge device such metrics as inference time, precision and model's size are essential.

We would like to outline that the work contains certain limitations. Even though the dataset provided numerous images in nature, it has some vulnerabilities. For instance, the humans wearing camo clothes might lead to confusing models and misdetections. Moreover, the dataset does include a huge variety of different landscapes that cover the whole world what challenges the model in such scenarios. The solution is continuous development of the dataset to correspond to any case. Alternatively, many models for contrasting biomes have to be created to simplify heavy tasks for the deep learning architecture.

Furthermore, we would like to address the future scope of the paper. We need to execute auxiliary experiments with every optimization technique to achieve better performance, overcoming the obstacles encountered and improving gained results. Furthermore, deploying and testing the optimized models directly on resource-constrained devices will help validate their performance in real-world scenarios and provide insights for future improvements.

References

[1] J. Suo, X. Zhang, W. Shi, and W. Zhou, “E³-UAV: An Edge-based Energy-Efficient Object Detection System for Unmanned Aerial Vehicles,” arXiv preprint arXiv:2308.04774v2 [cs.RO], Dec. 2, 2023. [Online]. Available: https://arxiv.org/abs/2308.04774

[2] Z. -Q. Zhao, P. Zheng, S. -T. Xu and X. Wu, "Object Detection With Deep Learning: A Review," in IEEE Transactions on Neural Networks and Learning Systems, vol. 30, no. 11, pp. 3212-3232, Nov. 2019, doi: 10.1109/TNNLS.2018.2876865.

[3] Girshick, R., Donahue, J., Darrell, T., & Malik, J. (2013). Rich feature hierarchies for accurate object detection and semantic segmentation. arXiv preprint arXiv:1311.2524.

[4] Girshick, R. (2015). Fast R-CNN. arXiv preprint arXiv:1504.08083. https://arxiv.org/abs/1504.08083

[5] Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster R-CNN. arXiv preprint arXiv:1506.01497. https://arxiv.org/abs/1506.01497

[6] Redmon, J., & Divvala, S. (2015). You only look once: Unified, real-time object detection. arXiv preprint arXiv:1506.02640. https://arxiv.org/abs/1506.02640

[7] H. Patel, “A Comprehensive Study on Object Detection Techniques in Unconstrained Environments,” The University of Texas at San Antonio, 2023.

[8] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, “SSD: Single Shot MultiBox Detector,” in Computer Vision – ECCV 2016, vol. 9905, B. Leibe, J. Matas, N. Sebe, and M. Welling, Eds. Cham: Springer, 2016, pp. 21–37. doi: 10.1007/978-3-319-46448-0_2.

[9] R. Sapkota, R. Qureshi, M. Flores-Calero, C. Badgujar, U. Nepal, A. Poulose, P. Zeno, U. B. P. Vaddevolu, S. Khan, M. Shoman, H. Yan, and M. Karkee, “YOLOv12 to Its Genesis: A Decadal and Comprehensive Review of the You Only Look Once (YOLO) Series,” unpublished manuscript, Cornell University, 2023.

[10] J. Liu, L. Niu, Z. Yuan, D. Yang, X. Wang, and W. Liu, “PD-Quant: Post-Training Quantization Based on Prediction Difference Metric,” unpublished manuscript, Huazhong University of Science & Technology and Houmo AI, 2024.

[11] R. Sapkota, Z. Meng, M. Churuvija, X. Du, Z. Ma, and M. Karkee, “Comprehensive Performance Evaluation of YOLOv12, YOLO11, YOLOv10, YOLOv9 and YOLOv8 on Detecting and Counting Fruitlet in Complex Orchard Environments,” unpublished manuscript, 2024.

[12] S. Woo, J. Park, J.-Y. Lee, and I. S. Kweon, “CBAM: Convolutional Block Attention Module,” arXiv preprint arXiv:1807.06521v2 [cs.CV], Jul. 18, 2018.

[13] M. S. Mia, A. A. B. Voban, A. B. H. Arnob, A. Naim, M. K. Ahmed, and M. S. Islam, “DANet: Enhancing Small Object Detection through an Efficient Deformable Attention Network,” unpublished manuscript, Nanjing University of Information Science and Technology & Beijing Institute of Technology, 2024.

[14] B. Jacob, S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko, “Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference,” Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR), 2018, pp. 2704–2713.

[15] TinyML and Efficient Deep Learning Computing, 6.5940, Fall 2024,” MIT Course Website, 2024. [Online]. Available: https://efficientml.ai

[16] H. Mao et al., "Exploring the Granularity of Sparsity in Convolutional Neural Networks," 2017 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), Honolulu, HI, USA, 2017, pp. 1927-1934, doi: 10.1109/CVPRW.2017.241.

[17] G. Hinton, O. Vinyals, and J. Dean, “Distilling the Knowledge in a Neural Network,” arXiv preprint arXiv:1503.02531, Mar. 2015.

[18] A. Romero, N. Ballas, S. E. Kahou, A. Chassang, C. Gatta, and Y. Bengio, “FitNets: Hints for Thin Deep Nets,” arXiv preprint arXiv:1412.6550, Dec. 2014.

[19] S. Zagoruyko and N. Komodakis, “Paying More Attention to Attention: Improving the Performance of Convolutional Neural Networks via Attention Transfer,” arXiv preprint arXiv:1612.03928, Dec. 2016.

[20] C. Shu, Y. Liu, J. Gao, Z. Yan, and C. Shen, “Channel-wise Knowledge Distillation for Dense Prediction,” arXiv preprint arXiv:2011.13256, Nov. 2020.

[21] I. E. Lassakeur, “HERIDAL,” Kaggle Datasets, [Online]. Available: https://www.kaggle.com/datasets/imadeddinelassakeur/heridal

[22] maxlee18.98@gmail.com, “SAR_1: Dataset Version 7,” Roboflow Universe, [Online]. Available: https://universe.roboflow.com/maxlee18-98-gmail-com/sar_1/dataset/7

[23] V. Bezborodov, “HERIDAL-2_SAR_1,” Kaggle Datasets, [Online]. Available: https://www.kaggle.com/datasets/vladyslavbezborodov/heridal-2-sar-1

[24] Ultralytics, “YOLOv8,” Ultralytics Documentation, [Online]. Available: https://docs.ultralytics.com/models/yolov8/

[25] Ultralytics, “COCO Dataset,” Ultralytics YOLO Documentation, [Online]. Available: https://docs.ultralytics.com/datasets/detect/coco/

[26] Ultralytics, “Real-time Inference,” Ultralytics Glossary, [Online]. Available: https://www.ultralytics.com/glossary/real-time-inference

[27] Ultralytics, “VisDrone Dataset,” Ultralytics YOLO Documentation, [Online]. Available: https://docs.ultralytics.com/datasets/detect/visdrone/

[28] Mahadih534, “YoloV8-VisDrone,” Hugging Face, Jun. 26, 2024. [Online]. Available: https://huggingface.co/Mahadih534/YoloV8-VisDrone

[29] ONNX Runtime, “Quantize ONNX models,” ONNX Runtime Documentation, [Online]. Available: https://onnxruntime.ai/docs/performance/model-optimizations/quantization.html

[30] OpenVINO, “Basic Quantization Flow,” OpenVINO Documentation, 2025. [Online]. Available: https://docs.openvino.ai/2025/openvino-workflow/model-optimization-guide/quantizing-models-post-training/basic-quantization-flow.html

Appendix

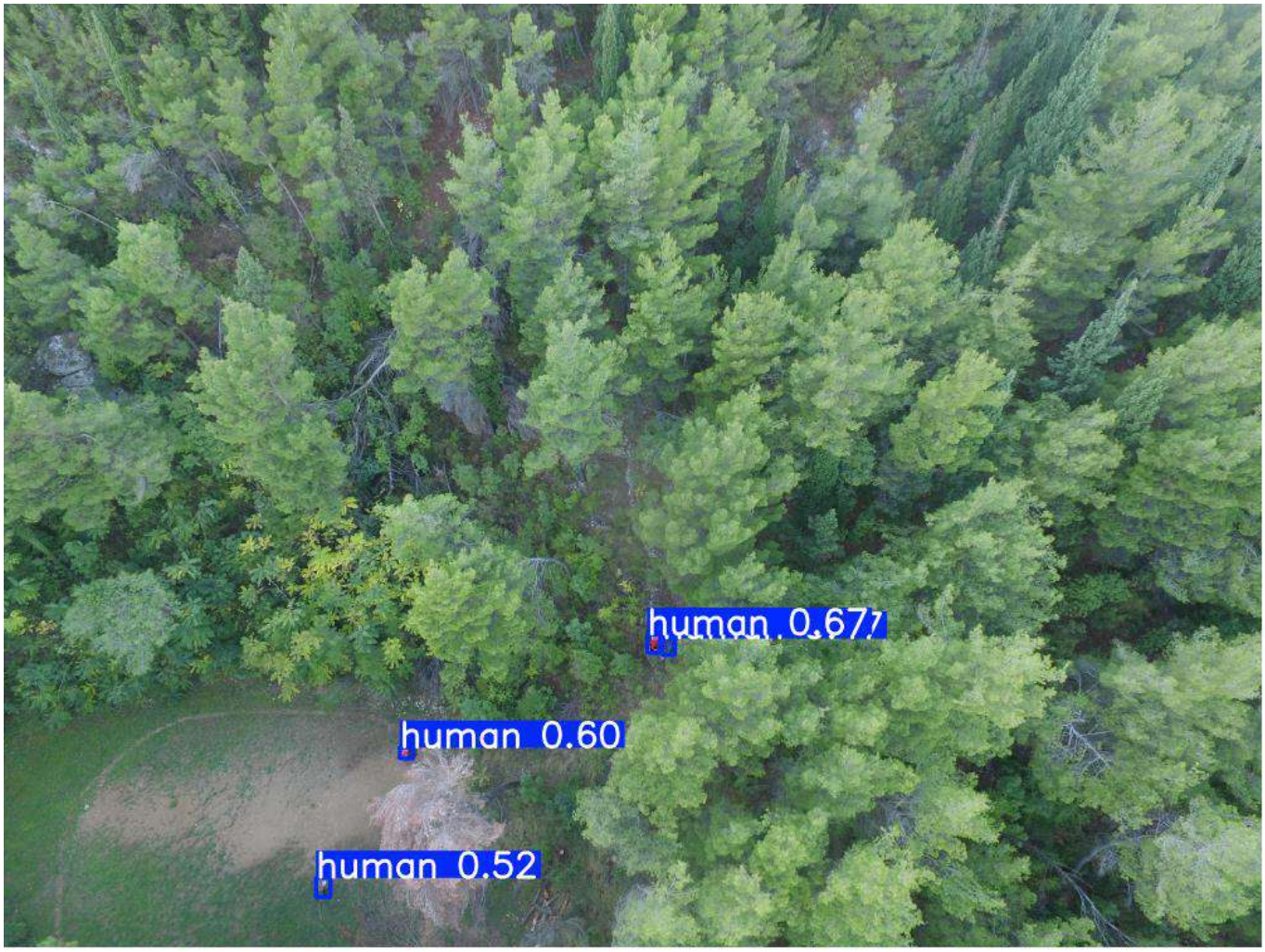


Figure. YOLOv8x object detection sample

All the models from the experiments section are located on Kaggle platform