UCLouvain

École polytechnique de Louvain

Object Detection with Scarce Data: Scans of Entomological Boxes using Active Learning

Authors: **Tobias Löwenthal, Alexandr Usoltsev**
Supervisor: **Sébastien Jodogne**
Readers: **Edouard Chatzopoulos, Patrick Semal**
Academic year 2024–2025
Master [120] in Computer Science and Engineering

Abstract

Insects are essential to ecosystem stability, yet global warming threatens their populations worldwide. Monitoring insect biodiversity is critical but challenging, especially using physical entomological collections. Digitizing these collections and applying AI for automatic insect detection can revolutionize ecological studies.

This thesis focuses on memory and data-efficient AI methods for detecting and counting insects in digitized entomological box images. Leveraging a custom dataset of 139 images and limited hardware (8 GB RAM MacBook Air), three approaches were developed: (1) post-detection filtering of pre-trained models, (2) fine-tuning lightweight YOLOv8 models via a dynamic multi-phase method, and (3) active learning strategies to reduce required training data. Notably, AL enabled achieving comparable detection performance to full dataset training while using 50% less data.

Our best model sets a new state-of-the-art in insect detection, reaching an map@50 of 0.799, while significantly lowering computational complexity, memory usage and hence overall electricity consumption. This work advances sustainable AI applications in biodiversity monitoring and contributes foundational tools for large-scale digitized insect collections.

Acknowledgements

First and foremost, we would like to express our sincere gratitude to UCLouvain Professor Sébastien Jodogne for his invaluable guidance and continuous support throughout the semester. His insights and perceptiveness greatly enriched this master thesis and were instrumental in navigating its various challenges.

We are also deeply thankful to Professor Patrick Semal, Curator at the Royal Belgian Institute of Natural Sciences (IRSNB), for providing the dataset that formed the foundation of this work. His clear articulation of the museum's needs and his readiness to participate in regular calls were key to aligning our research with real-world applications.

Special thanks go to Gautier Black and Nicolas Delcommune, whose master thesis provided a solid basis upon which our work was built. Their prior contributions were an essential reference and source of inspiration throughout our research.

Additionally, one of us would like to acknowledge Professors Matteo Matteucci and Giacomo Boracchi from Politecnico di Milano for their outstanding courses in computer vision, which played a key role in shaping the motivation behind pursuing this thesis topic.

Finally, we are grateful to Assistant Edouard Chatzopoulos for kindly agreeing to be a reader for this thesis. His involvement is greatly appreciated.

Contents

1 Introduction and State of the Art **1**
- 1.1 Introduction . . . 1
- 1.2 Problem definition . . . 3
- 1.3 State-of-the-art in computer vision . . . 4
  - 1.3.1 Neural networks . . . 4
  - 1.3.2 Convolutional neural networks . . . 5
  - 1.3.3 YOLO . . . 6
  - 1.3.4 Faster-RCNN . . . 8
  - 1.3.5 EfficientNet . . . 8
  - 1.3.6 Active learning . . . 9
- 1.4 State-of-the-art for insect detection . . . 11
  - 1.4.1 First master thesis (2024) . . . 11
  - 1.4.2 Second master thesis (2025) . . . 12

2 Detection **14**
- 2.1 Adapting the state-of-the-art . . . 14
  - 2.1.1 Memory Analysis . . . 14
  - 2.1.2 Label bias in the state-of-the-art . . . 17
  - 2.1.3 Non-ML false positive filtering . . . 18
  - 2.1.4 Binary post-detection classifier . . . 21
- 2.2 Fine-tuning on YOLOv8s . . . 25
  - 2.2.1 Problem definition . . . 26
  - 2.2.2 Memory analysis . . . 27
  - 2.2.3 Fine-tuning algorithm . . . 30
  - 2.2.4 Dealing with memory overloads . . . 35

3 Active Learning **38**
- 3.1 Merging strategies . . . 39
  - 3.1.1 Validation split . . . 40
  - 3.1.2 Validation replacement . . . 41
- 3.2 Subsampling strategies . . . 44

3.2.1 Random subsampling . . . 45
3.2.2 Uniform partition subsampling . . . 45
3.2.3 Subsampling based on uncertainty . . . 47
3.2.4 Subsampling based on diversity . . . 49
3.2.5 Supervised hybrid stategy . . . 53
3.3 Retraining strategies . . . 56
3.3.1 Full retraining . . . 57
3.3.2 Gradual fine-tuning . . . 57

4 Measures and results **59**
4.1 Test set and metrics . . . 59
4.1.1 Test set selection . . . 59
4.1.2 Metrics . . . 59
4.2 Detection performance . . . 63
4.2.1 Post-detection classifier . . . 63
4.2.2 Detectors . . . 66
4.3 Active learning performances . . . 69
4.3.1 Active learning simulation . . . 69
4.3.2 Comparison of subsampling strategies . . . 70
4.4 Limitations and Future Work . . . 76

5 Conclusion **78**

6 Bibliography **80**
A Code . . . 86
B YOLOv8 architecture . . . 87
C Post-detection classifier architecture . . . 88
D Typical prediction . . . 89

Chapter 1

Introduction and State of the Art

1.1 Introduction

Global warming is one of the most pressing challenges humanity faces today. An increase of a few degrees in average temperatures could have a catastrophic impact on life on earth at large. The equilibrium of nature is made of intricate relationships between species, with some being indispensable to the life of others. This forms a large-scale - and hence fragile - tree of dependencies between forms of lives: the disappearance of a given species can cause similar fates to any other species that were dependent on it.

This is particularly the case for insects, which are instrumental to the cycle of life itself: to name only a few of their vital roles, they ensure pollination, recycling of biological material and they serve as prey for a huge variety of small predators. If the scientific community wishes to find ways to mitigate the impact of climate change on insects, it first needs to study them thoroughly. Understanding their mutual roles, and gathering information about their population is both needed to evaluate their respective importance and to identify the risks that they incur. This preliminary stage is crucial when it comes to implementing effective and targeted solutions for their preservation.

In this context, entomological boxes serve an important role. They ensure the collection and preservation of large samples of specimens across time. With the help of entomological boxes, scientists can compare samples of the past with present samples, and evaluate the evolution of populations.

Many museums across the world possess such collections of entomological boxes. The Royal Belgian institute of Natural Sciences (IRSNB), for instance, is home to

more than 69 000 boxes, containing approximately 17 million specimens. While this collection is significant, it mostly contains samples gathered by Belgian entomologists, either within Belgium or during expeditions organized by the Belgian crown. There still remains many species to gather for the IRSNB collection to be complete and exhaustive of insect life.

This task of gathering all existing species of insects into a single collection is the dream of many entomologists. However, it is an ideal that is hard to achieve. Nevertheless, there exists a way to get closer to this ideal: digitization. If all entomological collections from museums and laboratories across the globe were gathered into a single digital database, scientists would be ever-closer to reaching that ideal. Moreover, digital databases have the significant advantage of being less vulnerable to the passage of time than physical collections: digital images can essentially remain unaltered for centuries.

Furthermore, such digital databases pave the way for advanced and automatic image processing opportunities. For instance, AI-driven models could take those images as input and extract precise statistics on the specimen count and sizes. An algorithm could automatically detect insects in each box, and deduce information about their number, size and species. In more practical terms, an object detection model could first identify and count insects, as well as hand-written labels for each box. Then, using this information and reading labels with an OCR[1], the algorithm could extract precise statistics about the number of specimens for every species. Moreover, a classification model could be trained on cropped detections, allowing the development of automatic and precise species classifiers. Such models would help entomologists in their dichotomous identification of species, which can be a complex and tedious task.

In this master thesis, we will tackle the preliminary step of this digital process: the detection and counting of insects in each box. However, to stay in line with the essentially environmental nature of the subject at hand, an emphasis will be put onto memory efficiency and data efficiency. Indeed, in recent years there has been growing concerns over the substantial greenhouse emissions generated by AI. Data centers, for instance, which are integral to AI operations and training, account for approximately 2.5% to 3.7% of global greenhouse gas emissions, surpassing those of the aviation industry[32]. Using AI for insect detection with no regard for its environmental impact would be contradictory in essence.

As such, in this master thesis we identify two ways in which AI-driven object

[1]Optical Character Recognition algorithm

detectors can be used at a reduced environmental cost: firstly, a reduction in model complexity, which in turn decreases both the number of operations and the memory needed for training. Secondly, a reduction in the amount of training data, effectively decreasing training time. Both aspects have a direct impact on electricity consumption. Hence, reducing them can indeed lead to reduced emissions.

However, decreasing model complexity and dataset size typically leads to worse performances. The ambitious goal of this thesis is to achieve the opposite result: obtaining better performances with a reduced model complexity and dataset size. As such, we will only train on a custom dataset of 139 images from the IRSNB collection, and using a MacBook Air with 8 GBytes of RAM.

Three efforts will be made in this regard. First, we will attempt to adapt pre-existing insect detectors without retraining them, thereby saving training time and energy, by using various *post-detection filters* to reduce the number of false positives. We will show that this approach, which only necessitates the training of a lightweight classifier for 10 epochs, increases f1-score from 0.765 to 0.876.

Secondly, we will attempt the fine-tuning of a YOLO version of greatly reduced complexity and resolution (from YOLOv8x 2016×2016 to YOLOv8s 640×640) on our local and highly-memory constrained machines, by using our *dynamic multi-phase fine-tuning* method. This method will result in an f1-score of 0.861 for an 11 times reduced inference time and with a much smaller model.

Thirdly, we will study different active learning (AL) strategies, aiming to select the optimal samples so as to obtain good performance with significantly less data. We will show that some of our methods match the performance of a full set training with 50% less data.

1.2 Problem definition

The goal of this thesis is thus to find automatic ways to identify and localize insects in entomogical boxes, with an emphasis on both memory efficiency and data efficiency. This task falls within the domain of computer vision, more specifically under the category of object detection.

Object detection is a fundamental task in computer vision that involves identifying and localizing instances of semantic objects—such as people, animals, or, in our case, insects—within an image. Formally, given an input image I, the goal is

to predict a set of N bounding boxes

$$\{(b_i, c_i, s_i) \mid i = 1, \ldots, N\},$$

where each bounding box $b_i \in \mathbb{R}^4$ is represented as $b_i = (x_i, y_i, w_i, h_i)$, with (x_i, y_i) denoting the center coordinates, and w_i, h_i the width and height of the box[2]. Each detection is also associated with a class label $c_i \in \mathcal{C}$ from a predefined set of categories, and a confidence score $s_i \in [0, 1]$ indicating the detector's certainty about the presence and classification of the object. The task therefore combines both classification (assigning object categories) and localization (determining spatial extent).

1.3 State-of-the-art in computer vision

1.3.1 Neural networks

Neural networks are computational models inspired by the brain, composed of layers of interconnected nodes (neurons) that process data through weighted connections. Each neuron receives inputs, applies a linear transformation using weights and biases, and passes the result through a nonlinear activation function. Stacking layers - especially fully connected (*dense*) layers where every neuron connects to all neurons in the previous layer - allows the network to model complex relationships between input and output data.

Training a neural network involves adjusting the weights to minimize a loss function, which measures the difference between the model's predictions and the ground-truth labels. This optimization is done using *gradient descent*, an iterative algorithm that computes the gradient of the loss with respect to each weight and updates the weights in the direction that reduces the error. Variants like stochastic gradient descent (*SGD*) or *Adam* improve convergence by using techniques such as mini-batches or momentum.

To evaluate and guide learning, the dataset is typically split into a *training set* and a *validation set*. The model learns patterns from the training data, while the validation set assesses how well it performs on unseen data, helping to detect issues like *overfitting* - that is, when the model memorizes the training data instead of learning general patterns. Overfitting results in low error on the training set but poor performance on new data.

[2]This is YOLO format, other specifications exist as well, like top-left and bottom-right coordinates

The goal of training is to achieve *generalization*: the ability of the network to make accurate predictions on data it has not seen before. Good generalization means the model has captured the relevant general features, allowing it to stay accurate even in contexts that differ from those of the training set. Regularization techniques, dropout, and early stopping are often used to promote generalization and prevent overfitting.

1.3.2 Convolutional neural networks

Convolutional Neural Networks (CNNs) represent the state-of-the-art approach for image-related tasks due to their ability to efficiently handle high-dimensional input data. Traditional fully connected (dense) neural networks struggle with image data, as the number of parameters required scales rapidly with input size, leading to overfitting and computational inefficiency. CNNs address these issues through convolutional layers, which apply learnable kernels to local regions of the input image. This allows the network to extract spatially localized features, such as edges or textures, while drastically reducing the number of parameters. The weight sharing and local connectivity in CNNs not only enforce spatial hierarchies but also improve generalization. Stacking multiple convolutional layers enables the network to learn increasingly abstract representations of the input, which are then typically passed to dense layers for final prediction tasks.

CNNs gradually reduce the spatial resolution of feature maps through a combination of operations, primarily strided convolutions, pooling layers, and, less commonly, downsampling techniques like average pooling or learned transformations. Strided convolutions apply filters with a step greater than one, effectively skipping pixels and shrinking the output size. Pooling layers, such as max pooling, summarize local regions by selecting the most prominent feature, reducing resolution while retaining important information. These operations help the network build hierarchical feature representations, capturing broader spatial context at deeper layers while reducing computational complexity and memory usage.

A CNN can be divided into two parts. The first is the backbone, composed primarily of convolutional layers, which creates a lower dimensionality representation of the image, called an embedding. The second is the head, which is a dense neural network that uses the embedding as input and accomplishes its given task (classification, detection, etc.). Therefore, it is common practice to re-use task-specific state-of-the-art backbones, which have been trained with large datasets and powerful computational infrastructure, and adapt them to a custom task. This is called *transfer learning*, as we have "transferred" knowledge from another model.

Adaptation to the task at hand is called *fine-tuning*, and it can be done with a fixed backbone (*linear probing*) or a trainable one (*full fine-tuning*).

A key challenge of neural networks is their lack of interpretability. Due to their complexity, it is often unclear how decisions are made, which can lead to misleading results—such as a model exploiting background patterns instead of relevant features. For example, if insect images are consistently taken in distinct box types depending on the collector, the network might learn to classify boxes rather than insects. And even without such a bias, users may still want to understand whether predictions rely on known traits or novel patterns. Saliency maps help address this by highlighting which regions of an input image most influence the model's decision. An example from Zhou et al [58], shown in Figure 1.1, has been built by using *class activation mapping*. Red regions in the saliency maps correspond to regions which contribute the most to the prediction of each class, namely "brushing teeth" and "cutting trees".

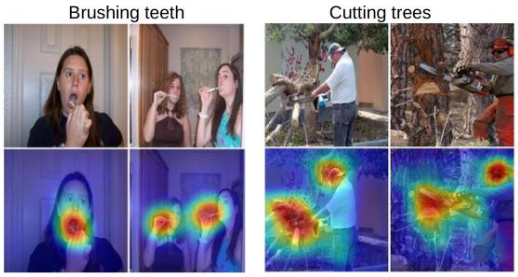


Figure 1.1: Example of a saliency map

1.3.3 YOLO

YOLO (short for You Only Look Once) is a family of object detection CNNs developed by Ultralytics and made for real time detection. They were trained on the COCO (Common Objects in Context) dataset, a large-scale object detection dataset designed to encourage research on a wide variety of object categories and commonly used for benchmarking computer vision models[50]. The main characteristic of YOLO, as the name suggests, is that it only takes a single pass over a network to predict bounding boxes and classify the objects within them, in contrast with methods like Faster RCNN (see Subsection 1.3.4).

To do this, YOLO separates the image into a grid of predefined size cells. Each of those cells uses a combination of convolutional and dense layers to predict the presence, location and class probabilities of objects within it. Then YOLO combines all the cells' bounding box and class predictions into a final list of detected objects.

Since several cells' output predictions could refer to the same object, a non-max suppression (NMS) algorithm is applied, which only selects the boxes with the highest confidence and removes lower confidence boxes that heavily overlap them.

The YOLO version that we use is YOLOv8. It is one of many upgrades to the original YOLO algorithm published by Redmon et al[39]. The main updates of YOLOv8 to the previous versions are anchor-free detections, a change in mosaic data augmentation, a decoupled head and a change from the C3 module used in the backbone to a C2f layer[57].

In previous versions of YOLO, bounding boxes were predicted as offsets from anchor boxes. With predefined sizes and aspect ratios, anchor boxes are the initial localization guesses of the detector: during training, those anchor boxes will be "refined". Specifically, for every grid cell, the network will predict a center, width and height offset for each anchor box, as well as an object probability (probability that the anchor box effectively contains an object).

YOLOv8 is anchor-free, which means that it predicts bounding boxes directly: for every grid cell, it computes a single set of values, for the center, width and height. This approach reduces the number of redundant bounding boxes and decreases the load on NMS. Moreover, it proved to improve generalization on custom datasets[57]. Indeed, predefined anchor boxes are class-specific, which might not suit a dataset that substantially differs from COCO.

Mosaic augmentation is a technique that combines 4 images into one. It has shown to improve generalization when used in the early training stage, so in YOLOv8 a *close_mosaic* parameter was added to stop mosaic augmentation for the last given number of epochs.

Unlike previous versions, in which a single head outputted values for localization and classification, YOLOv8 has two distinct heads: one for localization and one for classification. Those two tasks are different in nature, and separating them led to increases in performance.

Lastly, both the previously used C3 and the new C2f modules consist in a series of bottlenecks. Bottlenecks are submodules with a pair of 3x3 convolution layers and an optional residual connection (addition of the module's input to its output, so that the module learns a *residual function*). The difference between C2f and C3 is that C2f concatenates all the bottlenecks' outputs, and then aggregates them with a 1x1 convolution. The C2f module is thus able to learn how much of each

bottleneck representation it should include in its output.

1.3.4 Faster-RCNN

Faster-RCNN is another well-known state-of-the-art architecture in object detection. However, inference of previous insect detectors before ours was relatively slow using YOLO (see Subsection 1.4), and it is notable that YOLO is faster than Faster-RCNN[39][40]. Therefore, to speed up inference time, we decided not to use Faster-RCNN. We will thus not delve into too much detail concerning its architecture.

Faster R-CNN is a two-stage object detector that first generates region proposals and then classifies them. In the first stage, a *Region Proposal Network* (RPN) scans the image using a shared convolutional backbone to suggest candidate object regions. In the second stage, these proposals are refined and classified into object categories. The use of shared features between the RPN and classifier makes it more efficient than earlier models like R-CNN and Fast R-CNN. Compared to single-stage detectors like YOLO, Faster R-CNN typically achieves slightly higher accuracy, but at the cost of slower inference speed.

1.3.5 EfficientNet

EfficientNet is a family of CNNs designed for high accuracy with minimal computational cost. Its key innovation is compound scaling, which defines a precise strategy to optimally expand a given CNN. Once a satisfactory baseline model has been found, one can increase its complexity by uniformly scaling depth (number of layers), width (number of channels), and input resolution using this principled method.

The creators of this network initially searched for their baseline model, called EfficientNet-B0, using an algorithm called Neural Architecture Search (NAS). NAS finds an optimal architecture without relying on trial and error: instead, it uses reinforcement learning to find the optimal configuration. Larger EfficientNet versions were then found by applying compound scaling. Compared to earlier models like ResNet, EfficientNet achieves better accuracy with fewer parameters and FLOPs, making it highly efficient.

Its efficiency also stems from several architectural innovations. Depthwise separable convolutions split standard convolutions into two steps: first, a depthwise convolution, which applies distinct filters per input channel rather than mixing them

up. There is thus one output per input channel. Then, a pointwise convolution (1×1) combines the outputs. This technique drastically reduces the number of parameters.

Squeeze-and-Excitation (SE) blocks improve representational power by dynamically weighting each channel. First, spatial dimension is "squeezed" with global average pooling. The resulting vector is passed into a tiny dense neural network, which learns the relative importance of each channel. The resulting attention weights will then multiply their corresponding channels: the channels are "excited".

Finally, Swish activation (defined as $f(x) = x \cdot sigmoid(x)$) outperforms ReLU by allowing small negative values and smoothing gradient flow, contributing to better convergence and accuracy. Together, all these elements make EfficientNet both compact and powerful.

1.3.6 Active learning

Supervised machine learning, while greatly effective in many domains, has a certain cost, namely the need of a large annotated dataset for training. This data is rarely readily available and generating it can often be time consuming, especially for tasks where the annotation is complex, like bounding boxes for object detection. Moreover, larger datasets need to be trained for longer periods of time and increase the number of operations per epoch, which in turn increases energy consumption.

Active learning (AL) is a technique that helps reduce the required amount of data by only selecting the most useful unannotated samples from a pool and asking a human (called an *oracle*) to annotate them. In our work, we found crucial insight in this review of active learning in the context of object detection (OD) by Garcia et al[17]. The study distinguishes several different techniques for choosing which samples to select, as shown in figure 1.2.

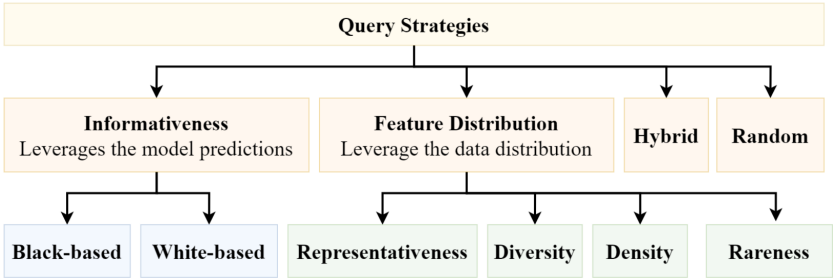


Figure 1.2: Query strategies identified in the review[17]

The informativeness-based approaches select new data to be labeled by looking at model predictions and trying to pick those that are near the model's decision boundary. In more OD-specific terms, we are looking for samples for which the current model is least confident. Garcia et al distinguish two classes of informativeness-based methods: black-box and white-box.

Black-box methods rely solely on the outputs of the model and their confidence scores. They are called 'black-box' because they do not use internal feature representations, nor do they attempt to interpret the model's internal workings. They will typically sample instances which yield the highest model uncertainty score. This score can be computed in various ways: a straightforward approach is to select instances whose best class confidence is the lowest. Other approaches exist, like the margin-based method (uncertainty is high when there is a large difference between the two highest confidence class scores), and the entropy-based method (high entropy reflects high uncertainty, because confidence scores are evenly distributed between classes).

White-box methods, on the other hand, pick a candidate based on internal information of the network and its architecture. Separate additional models are typically trained to predict uncertainty from the internal feature representations of the main detector.

While those uncertainty-based approaches have proven to be effective in many applications, their strictly informativeness-based heuristic is at risk of selecting candidates that are not representative of the dataset at large. Feature distribution based strategies, on the other hand, specifically study instances and their distribution to make selections. However, they sometimes do so using drastically opposite heuristics.

For instance, diversity-based methods select samples that are very different from the labeled pool according to some distance metric. On the other hand, representativeness-based methods select candidates that are close to the middle of "data clusters", thus better representing the "average image" to process. Those diametrically opposed strategies are a typical case of the exploration-exploitation dilemma(cf.[4]): while diversity-based methods explore the data space by looking for images that diverge from the current labeled set, representativeness-based approaches exploit the model knowledge by feeding it "typical" images, discarding any anomaly.

Rareness- and density-based strategies operate on similar principles. The for-

mer focuses on exploration by selecting anomalous or rare instances, while the latter emphasizes exploitation by sampling from dense regions of the data space to reinforce the model's existing knowledge.

Lastly, hybrid methods combine different heuristics to get the best increases in performance. Typically, such methods aim to find an optimal exploration-exploitation tradeoff.

1.4 State-of-the-art for insect detection

Two previous works exist on our specific application: insect detection in entomological boxes. Namely, our report will build upon the 2024 master thesis of Black, G.[6] and the 2025 one of Delcommune, N.[14].

The former took an exploratory approach to the subject, testing several different techniques for both detection and species classification, while the latter sought to improve results, focusing on deep learning-based techniques. As such, the detector developed by Delcommune is the state-of-the-art for this application. In our report, we will refer to it as the *state-of-the-art detector*, and to both authors as the *state-of-the-art authors*. However, it will be understood that their detector is in fact the *previous* state-of-the-art, since the method introduced in our report demonstrates significant improvements (see Chapter 4).

Although both authors contributed to advancements in insect detection and species classification, this report will focus exclusively on their work related to object detection, as it is the primary focus of our study.

1.4.1 First master thesis (2024)

The first detection method studied by Black consists of a series of image processing operations. First, a mask isolates all pixels above or below a certain threshold of hue saturation and value. This separates the objects in the image (mostly the specimens but also pins, text and some other elements) from the background of the box. Then, an erosion operation is performed on the blurred mask. Erosion removes small, thin black regions such as text or pin outlines. These two preprocessing steps are designed to eliminate non-insect elements that may still be present in the mask. After these operations, the mask ideally contains only insect shapes. To extract their outlines, contour detection is applied using a

method based on the Canny edge detection algorithm. This algorithm estimates the intensity gradient at each pixel and identifies edges by selecting those with the highest gradient magnitudes, thereby outlining the contours of the insect bodies.

This approach has the significant advantage of not requiring any sort of training. However, a human user must manually set mask thresholds for each individual image, which makes it hard to automate. Empirical observations indicate that the time required to set these thresholds—primarily through trial and error—is comparable to that of traditional manual image annotation, highlighting a significant limitation in the method's practical utility.

Secondly, the author went on to explore the Viola-Jones object detection framework. This method, originally designed for rapid face detection, uses a cascade of simple Haar-like features combined with AdaBoost for feature selection, and a sliding window approach to scan the image at multiple scales. Despite its efficiency and success in detecting well-defined, consistent shapes such as faces, the method did not yield satisfactory results in this context. The high variability in insect shapes and orientations likely contributed to its poor performance.

Finally, the author explored deep learning and state-of-the-art object detectors. Neither YOLOv8 nor Faster-RCNN, discussed respectively in sections 1.3.3 and 1.3.4, were able to achieve satisfying results out of the box. Therefore, a small dataset was labeled, with the help of the author's first approach (mask and Canny algorithm), and some hand-corrections. YOLOv8m was then fully trained on the small sample, with default training settings. This last approach yielded the best performances.

1.4.2 Second master thesis (2025)

This second paper by Delcommune, N. sought to build upon the previous work of Black, G. However, the study employed a narrower methodological scope, concentrating primarily on deep learning approaches and CNNs.

In order to improve the performances of the 2024 detector, the author significantly increased the model's resolution, width and depth. Input resolution was set at 2016×2016, and the largest version of YOLOv8, namely YOLOv8x, was used. Such a configuration necessitated a lot of memory and computing power, and for training the author was granted access to the *Lucia* supercomputer from Cenaero, Belgium[21]. As expected, such an increase in model complexity led to notable improvements. A similar approach was used with Faster-RCNN, but it

yielded worse results.

Another lead tested by the author was the use of *hard negative annotations* - that is, carefully labeled non-insect objects. Specifically annotating them as non-insects can reduce the number of false positives. However, this technique led to marginal improvements. Figure 1.3 shows an example of such annotations (highlighted in blue).

Figure 1.3: Example of hard negative annotation

Chapter 2

Detection

2.1 Adapting the state-of-the-art

The previous state-of-the-art in the matter of insect detection in entomological boxes used a detection model based on the YOLOv8x architecture. The latter model was completely retrained, all layers unfrozen, on a small labeled dataset[14].

Note that in this report we will only mention the best model that the state-of-the-art authors were able to build, namely, a retraining of YOLOv8x. With a mean relative error of -2.72, it significantly outperformed the retraining of Faster-RCNN (mean relative error of -13.27)[14]. On top of this, YOLO has been shown to be 9 times faster at inference than Faster-RCNN[39][40]. Finally, as will be described in more details in Subsection 2.1.4, our contribution will include a post-detection classifier, effectively adding another stage to the inference pipeline. Since Faster-RCNN is already in itself a two-stage detector, this additional stage would have likely led to excessive overhead.

On a second note, complexity analysis in this report will focus on memory, rather than execution time. We are equipped with an 8-core M1 chip, which achieves a peak theoretical throughput of 2.6 TFlops[3]. With such performances, training time should not be a problem. On the other hand, with our 8 GBytes of RAM, we are highly constrained by memory, calling for precise memory analysis to select optimal training configurations.

2.1.1 Memory Analysis

Retraining the largest YOLO model version, YOLOv8x, with all layers unfrozen is an expensive endeavour in terms of computation. Calculating formally the GPU

memory needed to do this will be instrumental in motivating our later choices.

Yolov8x contains 68.2 million parameters [52]. The dataset used in the previous state-of-the-art contained 51 high-resolution images of dimensions 8688×5792 pixels in full precision (32 bits per pixel), all resized to 2016×2016 pixels for training. The default yolo batch size (16) and initial learning rate (0.01) were used, and 100 epochs were completed without early stopping.

The memory needed to store one batch and store all model's parameters can thus be calculated as follows :

$68.2 \times 4 = 272.8$ MBytes	Memory needed to store model's parameters
$16 \times (2016 \times 2016 \times 3) \times 4 = 780.0$ MBytes	Memory needed to store one batch of images

Note that for each parameter a gradient and two *optimizer states* (first and second moment when using optimizer *Adam*) must also be stored. Therefore this effectively makes $4 \times 272.8 = 1.09$ *GBytes* of memory needed just for parameter updates.

However what really necessitates a lot of memory when training a convolutional network are the feature maps activations, which all need to be kept in memory for gradient descent in the backward pass. This must be done for every image in the batch. Calculating the number of feature maps and their resolutions requires us to have clear knowledge of YOLO's network architecture.

The architecture of YOLOv8 consists of 22 layers of different types : classical convolution layers, *C2f* layers, *SPPF*, *Detect* and *Upsample* blocks. The image from Appendix B does not only give a valuable overview of the whole network architecture, it also allows us to calculate the memory necessary to store the feature maps for a batch of images, whatever version of YOLOv8 we are using[8].

Using this figure, we can first compute the memory needed to train on one 640×640 image, by carefully adding up all the activations needed between each layer and sublayer. This includes the feature map output of each layer, but also any additional tensor which will be needed in practice. For example, the *MaxPool2d* layer needs an additional *pool indices* map of the same size of its output, to keep track of the selected maximum cell in the backward pass. To make a careful

computation of all of these feature maps, while taking into account the factors to apply when converting from one model size to another (see table on the top-right corner of the image in Appendix B, specifying depth, width and ratio), we made a Python script called *memory_analysis.py* which gives the theoretical memory needed to train a model of a given YOLO size (n, m, s, l or x) on images of a given resolution. Running the script with YOLO size x and image resolution 2016, we obtain a value of 12.42 GBytes to store the feature maps of a single image. Multiplying by batch size 16 and adding up the memory needed to store the parameters and the batch itself, we thus get :

$12.42 \times 16 = 198.75$ GBytes	Memory needed to store feature maps of one batch
$198.75 + 1.09 + 0.78 = 200.62$ GBytes	Memory needed to train on one batch, including parameters and optimizer states

All in all, this means 200.62 GBytes will be needed to be stored in the GPU memory during training in this configuration. And even though the average RAM capacity on home computers is increasing, it was of only 11.8 GBytes in 2024[31]. This highlights the inability of the vast majority of personal GPU's to store the amount of data needed for this model's training. We are ourselves equipped with a M1 GPU on a 2022 MacBook Air, and our 8 GBytes of RAM were of course way too few to attempt any retraining in this configuration. Upon trying, our program kept crashing without completing a single epoch.

It is thus no surprise that the authors of the previous state-of-the-art, willing to train on YOLOv8x and in the given configuration, had recourse to an external, more powerful hardware, in the name of the Lucia supercomputer (Cenaero, Belgium). Lucia, which achieves a peak performance of about 4 PFlops and is equipped with a flash tier memory of 200 TBytes[21], trained the model effortlessly and in a matter of minutes.

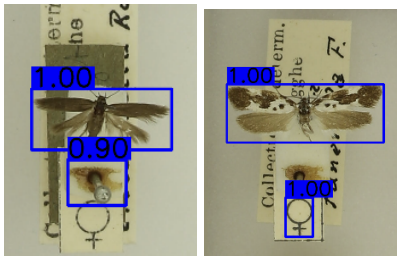
The performance of most computer vision networks is very dependent on computing power, so much so that some argue future progress in deep learning will slow down because of computational limitation[48]. Moreover, higher image resolution has proven to have a large impact on accuracy, when computing power is sufficient[56]. Therefore, our first intuition was that we could probably not achieve better results than the state-of-the-art by retraining a YOLO detector locally on our M1 chip : we could simply not match the computing power and image resolution they were able to exploit. Instead, we decided to train a light-weight

binary classifier that would take as input the detections of the state-of-the-art model and filter out as many false positives as possible (more details in Subsection 2.1.4).

2.1.2 Label bias in the state-of-the-art

On a small but diverse test set of 30 entomological boxes from the IRSNB collection, we tested the state-of-the-art model *coleo_lepido.pt* and noticed it achieved a high average recall per image of 0.96 and a low average precision of 0.52 under a standard confidence threshold of 0.5 (details and explanations on test set selection, performance metrics and measurements are available in Chapter 4). Even using the model's optimal point on the PR curve, at a confidence threshold of 0.83, recall was at a strong 0.85 and precision at a low 0.75. Moreover, the very fact that this optimal threshold was relatively high indicated that there were many false positives to filter out.

Those false positives have been reported by the state-of-the-art authors. The model tends to mistake handwritten labels, pins and small objects for insects. Such mistakes are shown in the figures below, which were predicted on our test set with the model at its optimal confidence threshold 0.826:



Incorrect detection of pin and symbol

Incorrect detection of label and object

Efforts have been made by the state-of-the-art authors to solve this problem. As mentioned, they tried to add hard-negative annotations on anything that their model could have mistaken for an insect. Hard negatives consist in an explicit annotation of potential false positives, labeled as "background", and added to the label files to increase detection models' precision. This technique has been used successfully in detection tasks as early as 2005, in a human detection task by Dalal and Triggs[12], and in detectors of historical importance such as DPM[15] and RCNN[18].

The approach resulted in slight improvements in their measured performance metrics (mean and median relative error w.r.t. true insect count) but those were

not very significant and they did not solve the problem entirely.

We believe that the core of this problem was to the way the state-of-the-art authors labeled small insects, especially those of the order *coleoptera*. Their bounding boxes, instead of showing a precise location for small insects, rather outlined the larger rectangular paper supports those insects were lying on. The rationale might have been that for such small insects, the information of this paper support is instrumental for the detection, even for human annotators. However, in doing so, they failed to realize that convolutional detectors like YOLO are designed to take into account elements of context even outside bounding boxes. Moreover, those imprecise labels triggered mistakes because their detector was now not only identifying insects, it was also detecting labels and rectangles of paper, leading to a substantial amount of false positives.

Therefore, we decided not to use the labeled dataset used by the authors of the state-of-the-art. Instead, we hand-annotated 167 images of entomological boxes from the IRSNB collection. When we encountered such small insects of the *coleoptera* order, we drew precise bounding boxes around them. Below is a comparison of the state-of-the-art labels (left) with ours (right).

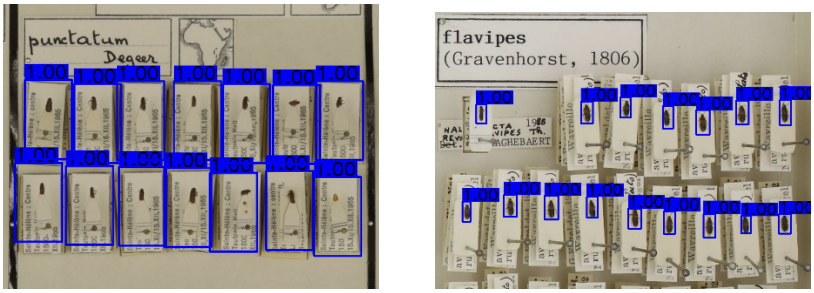


Figure 2.2: Side-by-side comparison of labels

2.1.3 Non-ML false positive filtering

So as to harness the significant computing power and network complexity deployed by the state-of-the-art authors, while mitigating the relatively low precision of their detector, we looked for ways to process the output detections of their model and decide if they are more likely to be false positives or true positives.

This kind of filtering is typically used when the initial detector achieves a high recall but a relatively low precision, with a substantial amount of false positives,

which is exactly the case here.

As a first step, we looked for non-ML methods to filter out false positives, allowing the posterior classifier to focus on harder instances. Upon looking at the detector's predictions on our training set, we noticed that a significant number of false positives could already be filtered out by using two simple non-ML methods. Firstly, a *non-maximum suppression* (NMS) on overlapping regions, based on a custom overlap threshold which we called *intersection over smallest area* (IoSA). Secondly, a filtering of severe outliers in terms of box area, using Z-score.

NMS was implemented to solve the state-of-the-art model's tendency to make multiple overlapping detections of the same insect. When a group of overlapping detections were observed, all non-maximum predictions in terms of confidence level were filtered out. The standard metric used to detect overlaps in object detections is the *Intersection over Union* (IoU), which measures the ratio of the boxes' intersection area over their union's area. However, this metric was not sufficient to detect the typical overlaps found in the detector's predictions : the model would often output very small faulty detections inside true larger ones. Those kinds of overlap, because of the larger area being considerably greater than the intersection, were not detected by an IoU threshold. Below is an example of such typical overlaps:

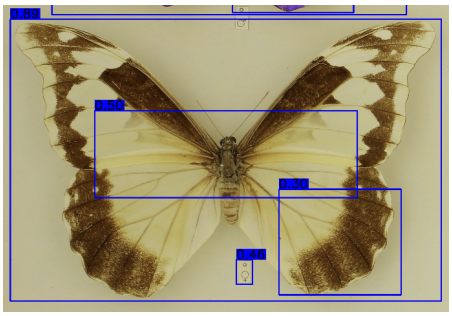


Figure 2.3: Example of typical detection overlap

This is why we introduced a custom overlap threshold, the IoSA, which measures the ratio of the intersection over the area of the smaller box. When this ratio is high, it means that the smaller box is largely contained in the larger one. When this ratio is 1, the smaller box is completely enclosed in the larger box. Below is a visual depiction of the IoU threshold on the top, and our custom IoSA threshold on the bottom.

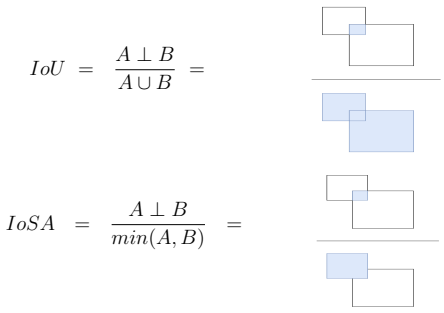


Figure 2.4: IoU and IoSA illustration

Applying an overlap threshold of 0.7 (which is the value we chose for overlap detection) on the example of Figure 2.3, specifically between the larger 0.89 confidence level box and the smallest 0.46 confidence level box, calculating IoU would have given a value of 0.004, thus way below the threshold. On the other hand IoSA gives a value of 1, and the overlap is detected, filtering out the false positive by performing NMS.

Secondly, the model sometimes predicted very large false detections, which were significantly larger than the true detections. We were able to automatically detect those ones using the image distribution of bounding box areas. The Z-score of an individual bounding box of area X is computed as follows:

$$Z = \frac{X - \mu}{\sigma} \quad \text{where } \mu \text{ is the sample mean area and } \sigma \text{ the standard deviation}$$

In a normal distribution, 99.7% fall within a Z-score of $\pm$ 3. We therefore chose a Z-score threshold of +5 to ensure the removal of severe outliers, significantly larger than the other boxes.

Although it is theoretically possible that a small insect would lie on a larger one, or that a very large insect would be put next to much smaller ones in a box, in which case the methods described above would erroneously filter out true positives, performance measures have shown that the methods had a positive impact on the inference precision, raising it to 0.58 and leaving the recall unchanged at 0.96 for the standard confidence threshold of 0.5. If this precision of 0.58 can be considered low, it is still a significant improvement compared to the initial 0.52.

2.1.4 Binary post-detection classifier

Even using the filtering methods described above, precision was still quite low. Therefore, we decided to implement a light-weight binary classifier which would take as input the detections of the state-of-the-art model, filtered by the non-ML methods, and decide which are true positives and which are false positives.

This technique is not new and is commonly referred to in the literature as *post-detection filtering*. Cheng et al called it *Decoupled Classification Refinement* and significantly increased precision and MaP (mean average precision, see Section 4 for explanations on metrics) on VOC and COCO datasets by performing *hard false positive suppression* on the output of Faster-RCNN[10].

Since the average state-of-the-art model's inference time per image on our test set was at a high 5.45 seconds, and because of our limited GPU memory capacity, we decided to choose EfficientNet as our pretrained model. As already mentioned, EfficientNet is an ideal pretrained model for fast inference and low training memory needs because it uses compound scaling, depthwise separable convolutions, and squeeze-and-excitation blocks to optimize performance and efficiency.

In order to train a convolutional model, all training images have to have the same resolution. This is not the case of cropped bounding boxes, of course, because they vary in size. Therefore, they need to be all resized to the same resolution. An input size of 256×256 was chosen for the model, close to the original 224×224 resolution EfficientNetB0 was trained on, but with some additional pixels to account for padding. Indeed, we chose to resize the bounding boxes "with pad", as performed by the tensorflow method *resize_with_pad()*, so as to preserve their original width-to-height ratio, or aspect ratio, preventing distortions that could occur from stretching or squashing. We chose this resizing method over the more classical bilinear interpolation method, which does not preserve aspect ratio. Our rationale was that discrimination between small insects and handwriting or pins at low resolution can be challenging, especially when distortions are involved. Experiments with the standard bilinear interpolation resizing method and no padding confirmed our intuition, as validation cross-entropy converged twice slower to its minimum, and never reached the performance of the resize-with-pad method (see Figure 2.7).

The EfficientNetB0 network, pretrained on the ImageNet dataset, remained frozen, so as to take advantage of the meaningful feature maps it is able to build. On top of this frozen network, a classification head was added, composed of a global average pooling layer (reducing dimensionality to a 1D vector), a batch nor-

malization layer, and then a succession of 2 dense layers with gelu activations, each paired with a dropout layer (dropout of 0.2). These choices were deliberately made to support efficient gradient flow. Experiments have shown that our classifier had some trouble converging, with its convergence being highly dependent on training data label distribution. However, when it did converge, it achieved very quickly a high validation accuracy, without any dramatic overfitting. In the convergence vs overfit-risk trade-off, we decided to favour model convergence, hence the gelu activation function for its smoothness and a dropout value small enough to allow model stabilization.

The dense layers have 256 and 128 neurons respectively, making the network able to learn non-linear combinations of the EfficientNet feature maps to discriminate between insects and non-insects. A softmax layer with two neurons was finally added. Although the sigmoid is more standard for binary classification, we opted for a softmax to facilitate potential future extension to multi-class classification (e.g., distinguishing between coleoptera, lepidoptera, and background). With softmax, this can be achieved simply by adding additional output neurons, without altering the underlying architecture. Appendix C shows a diagram of the classifier's architecture, generated with the tensorflow *plot_model()* method (the *Functional* block is the pretrained EfficientNetB0).

Our dataset for the classifier training was generated using the predictions of the state-of-the-art model at a minimum confidence level of 0.25. The vast majority of predictions below this threshold were easily detectable false positives which unnecessarily increased data load without fostering classification accuracy. True positives were identified using the IoSA threshold rather than the standard IoU, because of the prior label bias in the state-of-the-art dataset (c.f. Subsection 2.1.2): indeed, the detectors tended to draw excessively large bounding boxes enclosing the paper supports around small insects. We decided nevertheless to treat those detections as true positives, since in this task the focus is rather on sole insect detection than insect localization (this choice is motivated in more details in Chapter 4). True insect bounding boxes being totally contained in the larger predictions, this usually gave an IoSA score of 1, exceeding our chosen threshold of 0.7 and hence being considered as true positives.

80% of the entomological boxes were randomly selected for training, while the remaining 20% were used for validation. This was done to monitor training and ensure generalization, since the entire boxes chosen for validation remained "unseen" in the classifier's training. Below is a random sample of 10 predictions from the training set, along with their corresponding binary label.

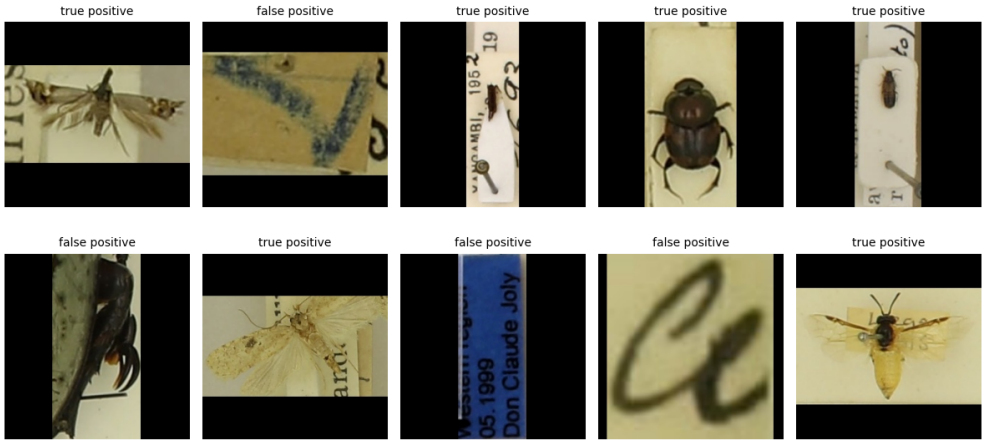


Figure 2.5: Sample of true and false positives

Because we noticed that the classifier achieved good accuracy even with a small training set, we limited the training set size to 5000 samples to decrease memory load. If this threshold is exceeded by a difference of δ, δ images will be deleted from the training set, while making sure that the label balance remains unchanged.

Moreover, we noticed that model convergence was highly dependent on training set label balance, as is often the case of convolutional classifiers (class imbalance has a significant negative impact on gradient convergence[9][16]). In our case, we observed a counterintuitive dynamic where the classifier achieved optimal convergence when trained on a strongly imbalanced dataset consisting of 80% true insect instances. Training set was thus undersampled to reach this positive label ratio. Even in such a setup, the model generalized well on balanced test sets.

In contrast, convergence failed entirely when training on a class-balanced training set. This behavior was likely caused by high intra-class variance in negative samples. Our negative class of non-insects represents a broad and heterogeneous set of false positives, such as texture variations and background objects. Hence, they lack of a real distinguishable semantic meaning. The results suggest that the model implicitly engaged in *positive-class modeling with anomaly rejection*, a strategy akin to *one-class learning*, where a compact, semantically coherent positive class is learned while all others are treated as outliers[35][41]. When the negative class was overrepresented or equally weighted, the model's capacity was overwhelmed by noisy and inconsistent gradients caused by the variability of our false positive class.

Since our model achieved high validation accuracy in a few number of epochs, thanks to the meaningful representations given by the pretrained EfficientNet, training for only 10 epochs was sufficient. Moreover, we monitored validation loss (measured with cross-entropy) with an *early stopping* mechanism, choosing a patience of 5. If the validation loss had not decreased for the last 5 epochs, the training was stopped, and the best model weights were restored. In our case, this already happened at the 7^{th} epoch, since minimum cross-entropy had been recorded at epoch n$^{\text{o}}$ 2.

For early stopping, we chose to monitor validation loss over validation accuracy to account for confidence levels of the classification (also known as *calibration error*[5]), and not only its outcome (or *refinement error*). The graphs below shows the evolution of validation accuracy and cross-entropy for each epoch. As you can see, our model converged very quickly to a validation accuracy of 0.93 at epoch n$^{\text{o}}$ 2.

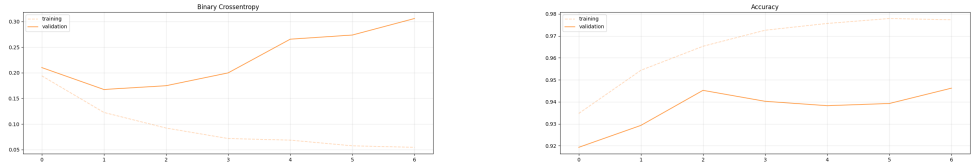


Figure 2.6: Evolution of cross-entropy (left) and accuracy (right)

As a comparison, below is the cross-entropy evolution for a model trained with bounding boxes resized with the classical bilinear method, which does not preserve aspect ratio. Minimum cross-entropy is only 0.2 (vs 0.174 for our resize-with-pad method) and is reached at epoch n$^{\text{o}}$ 4, thus slowing down convergence rate by a factor of 2.

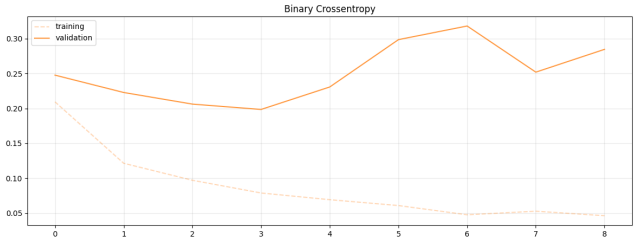


Figure 2.7: Evolution of cross-entropy with the bilinear resizing method

All in all, this light-weight classifier was very effective at improving the perfor-

mance of the state-of-the-art detector. Although average recall dropped down to 0.913, precision increased significantly to 0.870, for an increased average f1-score of 0.876 (those measures were made at the optimal point on the PR curve, with a confidence treshold of 0.646. More details in Chapter 4).

However, our classifier has the substantial disadvantage of significantly increasing inference time, up to 25.83 seconds per image on average. This is largely because all bounding boxes found by the detector have to be individually resized before being passed to the classifier. While this is still faster than manual counting (entomological boxes can contain hundreds of specimens), this inference time is too long for us to be able to claim a new state-of-the-art.

2.2 Fine-tuning on YOLOv8s

Although a significant improvement to the state-of-the-art model was achieved by applying post-detection filtering to its output, we believed that we could push performances even further by fine-tuning YOLOv8 more carefully on our local machine, and using a custom dataset from the IRSNB, free of any label bias (c.f. 2.1.2).

An obvious line of improvement was the inference time. As already mentioned, state-of-the-art detection on its own takes on average 5.45 seconds per image, and inference time goes up to 25.83 seconds when post-detection filtering is used.

Therefore, we decided to search for a new training configuration which could decrease inference time while still achieving good performance. Insects in entomological boxes are usually well organized and clearly separated in the boxes, with uniform lighting and few overlaps between instances. If it wasn't for labeled data scarcity, this object detection task would have likely been relatively simple. In our opinion, a much lighter and simpler model was enough to reach similar performances, if not better. With this in mind, along with our strong hardware constraints (M1 GPU and 8GBytes RAM), we decided to change several aspects of the state-of-the-art training configuration:

- We reduced image resolution from 2016×2016 down to 640×640 (see Subsection 2.2.2)
- We trained YOLOv8s rather than YOLOv8x (see Subsection 2.2.2)
- We fine-tuned on our own dataset from the IRSNB (139 images), which we carefully hand-annotated ourselves to avoid any label bias.

- We did not specify a *close_mosaic* value k when training. This argument disables mosaic augmentation[1] for the last k epochs[52]. Authors of the previous state-of-the-art completely disabled mosaic augmentation in the whole training. However, mosaic augmentation has been shown to improve model generalization, making detection less context-dependent[7]. We believed that this could be beneficial for our application, since our small dataset size poses risks of overfitting. Therefore, we kept the *close_mosaic* argument at its default value of 10.
- We carefully fine-tuned YOLO with gradual layer unfreezing, so as to avoid losing potentially meaningful representations obtained on the COCO dataset. (see Subsection 2.2.3)

2.2.1 Problem definition

Every version of YOLO is pretrained on the COCO dataset. COCO only has 91 classes, which are of course not exhaustive enough to be adapted to all real-life applications. And indeed, none of its categories correspond to an "insect" class, nor to any given insect species. The figure below shows depictions of all the categories included in COCO[11]:

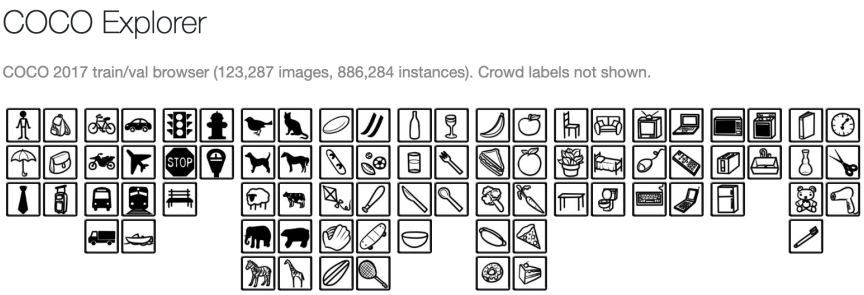


Figure 2.8: COCO categories

Because our class of interest is not included in the COCO categories, we cannot use direct transfer learning with a frozen backbone (*linear probing*), like we did with EfficientNet in Subsection 2.1.4. EfficientNet was trained on the ImageNet dataset, which has dozens of classes corresponding to insects or species of insects, so its representations can be directly used the way they are. In contrast, YOLO, which was trained on COCO, needs to be fine-tuned more deeply to be able to detect insects.

[1]Mosaic augmentation combines 4 images into one, typically at early training stages[51]

The state-of-the-art authors trained a binary detector, which identified two classes, *coleoptera* and *lepidoptera*. This was a sensible choice, as both orders can differ significantly from each other in terms of appearance. However, aiming for model simplicity and easier image labelling, we decided to make our model single-class: it would label all detected instances as "insect". Our rationale was that all insects share some visual features, like body segmentation or antennae, which makes us able, as humans, to confidently group various species under the "insect" umbrella. And even when two kinds of insects are not at all alike (e.g. a butterfly and a caterpillar), some prior knowledge makes us able to put them in the same semantic category, despite their differences. We believe neural networks possess a similar capacity to learn shared features and group visually dissimilar objects into the same category. Since our goal is, as a first step, to detect all instances of insects more than distinguishing between them, we chose a single class to reduce annotation effort and model complexity.

Note that, when evaluating performance, we are effectively computing a single aP (average precision) rather than MaP (mean average precision) because the MaP is the average over all classes, and we only have one class (More details in Chapter 4).

2.2.2 Memory analysis

As already mentioned, we chose to reduce dimensionality from 2016×2016 down to 640×640. This means much faster inference times, and much less memory-intensive training. That said, this also means that the detector may miss fine details, detectable only at high resolution. But YOLO has been trained natively on images of size 640×640, and increasing resolution beyond this value has not yielded better results in some research applications[27][44]. In general, it has been shown that while higher spatial resolutions can enhance detection accuracy, the improvements plateau beyond a certain point[56]. Using those empirical insights, a resolution of 640 seemed a good trade-off.

But our main motivation in choosing this resolution was our desire to promote memory efficiency, and the limited RAM available on our machine. Because we wanted to make the most out of our hardware, we designed our script *memory_analysis.py* to be able to search for the optimal training configuration given the RAM available. Making a grid-search on a series of YOLO model sizes and input image resolutions, the script tries to minimize the difference between the "available RAM" ($total\ RAM - system\ reservations$, with system reservations set at 1.5 GB by default) and the training memory needed, when this difference is positive. This allowed us to choose a training configuration which would use the

most out of our RAM capacity without overloading it. Running the script with the *–ram* parameter set at 8 GBytes (which is the RAM capacity of our MacBook Air), we got the following results :

```
Optimal resolution for model YOLOv8n is 640 which uses 3.157715968 GBytes
Optimal resolution for model YOLOv8s is 640 which uses 6.245364736 GBytes
Optimal resolution for model YOLOv8m is 320 which uses 3.0618149546666666 GBytes
Optimal resolution for model YOLOv8l is 320 which uses 4.728329216 GBytes
Optimal resolution for model YOLOv8x is 320 which uses 6.11832832 GBytes
```

Figure 2.9: Output of the *memory_analysis.py* grid-search

Therefore, it seemed that training with a resolution greater than 640 was not an option anyway. We discarded the option of training on 320×320 images, which would have been too few to detect boxes with large numbers of small insects. Instead, we chose to downgrade the YOLO version from YOLOv8X to YOLOv8s, thereby reducing the number of parameters from 68.2 down to 11.2 million parameters, as well as reducing network depth in the *C2f* layers by a factor of 3, and its width in all layers by a factor of 2.5 (see Appendix B). As a reminder, network depth refers to the number of layers, and network width to the number of channels in the layers outputs. As you can see in the figure above, the chosen configuration needs a minimum available RAM of 6.24 GBytes, which is reasonable for our 8 Gbytes RAM, and much below the the 200 GBytes demanded by the state-of-the-art configuration.

However, in practice, the additional memory needs of the OS and other processes, in conjunction with the overhead induced by fragmentation, can be unstable, sometimes exceeding this 1.5 GBytes "security net" which we called *system reservations*. Therefore, every now and then, memory on our machine was still overloaded, and training crashed. Because we did not want to downgrade even more our training configuration, we found two solutions to this problem. Firstly, we gradually decreased batch size every time we unfroze model layers. Secondly, we monitored system failures and reran training automatically when needed, regularly keeping training checkpoints in memory. The first solution (batch decreasing) will be explained in the current subsection, while the second one (reruns) will be detailed in Subsection 2.2.4.

Memory needed for training decreases linearly with batch size, since feature maps for every image in the batch must be stored when using *Adam* optimizer. Decreasing batch size is therefore a very effective way of reducing memory needs. However, it also considerably slows down model convergence. For very low batch sizes (smaller that 16), it may simply not converge at all. Research on batch size has shown that very small batch sizes can result in unstable training if not paired

with an appropriate learning rate[29]. A practical compromise is thus to firstly ensure convergence, and proximity to a "local minimum" in terms of loss, by using a larger batch size, and only then reduce the batch size to fit in memory constraints. A first memory-intensive training step ensures convergence, and the following steps exploit the first step's solution to improve loss on smaller batch sizes. This staged training approach was employed in our work, where we first trained on a completely frozen backbone and batch size 16, and then gradual unfroze the model and reduced batch size.

More specifically, we divided the fine-tuning of YOLOv8s in 23 steps, unfreezing the pretrained model layers one-by-one, so as to avoid loosing the meaningful feature maps of the original YOLO model (more information on the fine-tuning algorithm in Subsection 2.2.3). As well as enabling a gradual tuning, frozen layers do not require any computation, and they also reduce the memory load. Indeed, optimizer states do not need to be stored into memory anymore, since those layers weights will not be updated. The graph below shows the evolution of theoretical memory need with *layer unfreezing* for a constant batch size of 16. It was also plotted using the script *memory_analysis.py*.

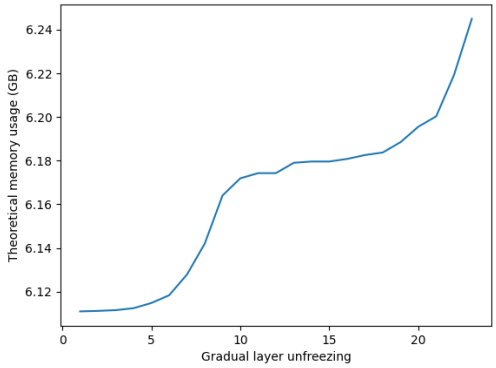


Figure 2.10: Impact of layer unfreezing on memory usage

The difference in memory usage between a frozen model and an unfrozen one is approximately 120 MBytes. If this may seem few, relatively to the total memory usage (more than 6 GBytes), those additional MBytes were often the decisive ones which made our training overload memory. It was observed that, using a constant batch size, every subsequent fine-tuning step had a higher chance of crashing. But this is not solely due to layer unfreezing. The culprits for this behaviour are most probably memory leaks and inefficient garbage collections, as has been reported

in similar GPU-related cases[24][38]. Therefore, we decided to gradually decrease batch size down to 8 for the completely unfrozen model. As already mentioned, the first iterations ensured convergence, and the following ones exploited prior solutions with smaller batch sizes, hence decreasing memory load to account for layer unfreezing, memory leaks and inefficient garbage collection. The graph below shows the evolution of theoretical memory usage with fine-tuning steps where both the number of frozen layers and the batch size are decreased.

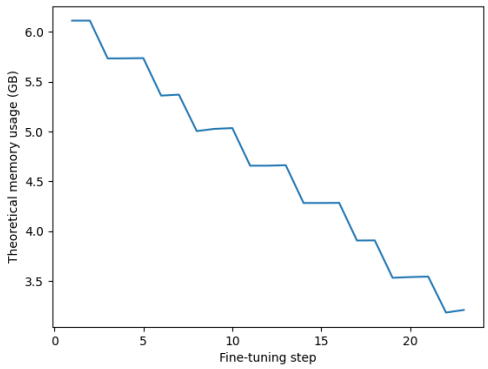


Figure 2.11: Evolution of theoretical memory usage vs. fine-tuning step

In practice however, the decrease in memory usage was much less significant. We found that this gradual batch size decrease was ideal to account for the increasing memory needs of subsequent training steps, induced by memory leaks and inefficient garbage collection.

2.2.3 Fine-tuning algorithm

Fine-tuning is a transfer learning technique in which the parameters of a pre-trained neural network model are trained on new data. The key difference between full fine-tuning and linear probing is the fact that, for full fine-tuning, all layers of the network are trained on the new data, and not only the newly attached head. The weights are effectively initialized to the pretrained model's parameters, and then they are updated with gradient descent. When training our *post-detection filter* in Subsection 2.1.4, linear probing was sufficient to achieve a good validation accuracy. In the current detection task however, training only the detection heads of YOLOv8s was not sufficient to achieve a good validation MaP. This is because our custom dataset of entomological boxes differs substantially from the images in

the COCO dataset, both in terms of object context and in terms of output classes (as mentioned in Subsection 2.2.1). To adapt the pretrained detector to our dataset, we also train the YOLO backbone, enabling weight updates even for shallow layers.

The same fine-tuning approach was used by the previous state-of-the-art authors, as they completely retrained YOLOv8x on their dataset. However, we believed that the small size of our dataset (139 images for training) called for a more careful, gradual fine-tuning. A full retraining with the default YOLO learning rate (0.01) exposed to risk of overfitting and *catastrophic forgetting*[2] with such a small dataset. Carefully tuning the YOLO pretrained weights towards an optimal solution for insect detection was crucial to ensure good generalization.

Specifically, we decided to fine-tune YOLOv8s in several steps, gradually unfreezing layers, rather than training them all at once. This approach, referred to in the literature as *gradual tuning* or *multi-phase fine-tuning*, has shown promising results in several computer vision studies, significantly reducing the risk of *catastrophic forgetting*[30]. In a *Comparison of fine-tuning strategies for transfer learning in medical image classification*, scholars have shown that such gradual unfreezing was the most effective method for many medical applications[13]. The same approach has been used in various computer-vision applications[22][42], including one entomological application of *insect species identification*[27].

This gradual tuning technique was combined with what is referred to in the literature as *dynamic fine-tuning*[13]: the learning rate is adjusted adaptively during the fine-tuning process. This is done both between each fine-tuning steps, and within each of them.

During a single training (one fine-tuning step), YOLO implements by default an adaptative learning-rate over time, going from an initial $lr0$ to a smaller $lr0 \times lrf$ (with $0 < lrf < 1$). Moreover, we chose to use *Adam* optimizer rather than *SGD*. *Adam* enables a dynamic adaptation of the learning rate by taking into account the gradients of the previous iterations. Although *Adam* is slightly[3] less memory-efficient than *SGD*, because it needs two additional optimizer states per parameter (first and second momentum), it has been shown to converge faster than *SGD* in many deep learning applications, especially in those presenting sparse features[26][33]. Since our dataset consists of boxes of very diverse insect species,

[2]catastrophic forgetting describes a scenario in which a neural network, after being trained on a new task, completely or substantially forgets the information related to previously learned tasks[2]

[3]We say "slightly" because optimizer state storage is small compared to batch feature maps, see Subsection 2.2.2

we believed that the gain in epochs allowed by a faster *Adam* convergence would make up for the slight memory disadvantage it has against *SGD*. As outlined in Subsection 2.2.2, epoch number has an impact on memory usage because subsequent iterations have an increasing chance of overloading RAM.

Moreover, we also explicitely adapted learning rate between each fine-tuning step. Starting with $lr0 = 0.01$ and $batch = 16$, we progressively reduced those values down to $lr0 = 0.001$ and $batch = 8$ for the final step. The rationale is two-fold: firstly, research consistently indicates that smaller batch sizes benefit from a smaller learning rate, to mitigate gradient noise and training instability[28][29]. Secondly, we expect shallow layers of the YOLO backbone to have learned general features which are good for a wide range of applications, including ours. On the other hand, the deep layers (close to the detection heads) make the final decisions to classify instances among the COCO categories. Therefore, the former shallow layers should only be slightly updated (low learning rate), while the latter deep ones are to be entirely modified to adapt to our entomological application (high learning rate).

Our iterative and adaptive fine-tuning technique, which we call *dynamic multi-phase fine-tuning*, is presented as a pseudo-code in Algorithm 1 (next page).

Algorithm 1 Fine-tuning algorithm

```
1: procedure DYNAMIC_MULTI_PHASE_FT(dataset, ft_steps)
2:     best ← yolov8s
3:     frozen_layers ← 22
4:     lr0 ← 0.01
5:     batch ← 16
6:
7:     frozen_decrement ← 23 ÷ ft_steps
8:     lr_decrement ← (0.01 − 0.001) ÷ ft_steps
9:     batch_decrement ← (16 − 8) ÷ ft_steps
10:
11:    for i = 1 to ft_steps do
12:        model ← TRAIN_AND_MONITOR(dataset, best, frozen_layers, lr0, batch)
13:
14:        if MAP@50(model) > MAP@50(best) then
15:            best ← model
16:        end if
17:
18:        frozen_layers ← frozen_layers − frozen_decrement
19:        lr0 ← lr0 − lr_decrement
20:        batch ← batch − batch_decrement
21:    end for
22:
23:    return best
24: end procedure
```

Note that at each step, weights are initialized to the best model obtained so far in terms of validation map@50, and not necessarily the output of the previous iteration. This is done to prevent catastrophic forgetting, where one step would obtain poor results and pass on a poor initialization to the subsequent steps. Instead, all steps which do not improve validation map@50 are effectively skipped, and only the best models are used for weight initialization.

The procedure *TRAIN_AND_MONITOR* in line 12 of the peudo-code refers to the system failure monitoring explained in Subsection 2.2.4. A pseudo-code of the procedure is available in Algorithm 2.

Finally, we chose to train each step for 20 epochs and used early-stopping with patience 5 to avoid unnecessary iterations which do not increase performance, and to ensure best weight selection.

The graph below shows the evolution of validation map@50 with fine-tuning steps, using this dynamic multi-phase technique :

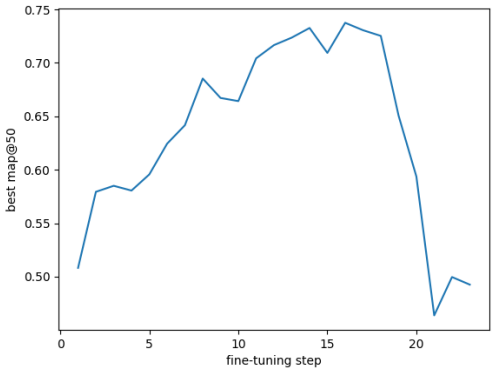


Figure 2.12: Evolution of validation map@50 vs. fine-tuning step

MaP increases gradually with fine-tuning steps until its peak at 0.738 in step n° 16, where it starts to decrease at a fast rate. This confirms our intuition that shallow YOLO layers, especially up to layer n° 7, benefit from remaining frozen since they represent low-level features that are needed for a broad set of detection tasks. Updating their weights is thus not likely to result in an increase in performances, and in fact it is the opposite that occurs. Deep to middle-depth layers (layer n° 8 to 22), however, can be successfully fine-tuned to such small datasets when paired with an appropriate learning rate.

This graph highlights the strength of this technique when fine-tuning YOLO on a small custom dataset, especially when we compare it to other fine-tuning techniques. Upon trying linear probing, where we only train the detection head and keep the backbone frozen, map@50 stayed consistently below 0.55, even when trained for 40 epochs. Full fine-tuning, with all layers unfrozen, overloaded memory with batch 16, and performed very poorly with lower batches. This highlights the need for gradual fine-tuning when dealing with small datasets.

All in all, fine-tuning locally with this method allowed us to set the new state-of-the-art in insect detection in entomological boxes, achieving an average f1-score per image of 0.861 for a minimum confidence threshold of 0.110, the optimal point on the PR curve. As a comparison, average f1-score of the previous state-of-the-art was 0.765. Moreover, inference coupled with non-ML filtering methods runs in

0.47 seconds on average, which is more than 10 times faster than the previous state-of-the-art which ran in 5.45 seconds.

The same post-detection classifier as the one described in Subsection 2.1.4 was trained on our new detector's predictions. It increased average f1-score up to 0.889, at the cost of an average inference time of 9.66 seconds. All those performance metrics will be detailed in Chapter 4.

2.2.4 Dealing with memory overloads

This subsection is about the second solution we found to recurrent memory overloads: we monitored system failures and implemented reruns with reduced memory load when those were detected.

More specifically, every fine-tuning step was run using the python library *subprocess*, which allows to spawn new processes, connect to their input, output and error pipes, and obtain their return codes[37]. This allowed the parent process, responsible for the global multi-phase tuning, to continue running even when one of its child processes had crashed. YOLO allows for extensive feedback during training, which it puts into a *run* directory. Among other files, the directory contains a csv describing every epoch, a copy of the best model found based on validation MaP, and various statistic files and plots which are added upon training completion. Even when the training has crashed at some epoch, the best model found is stored in a *best.pt* file. Using those feedbacks, the parent process can work out if one of its child processes had completed or not.

Specifically, when the global process detects a failure, it copies the incomplete *run* directory to an *attempts* folder, and reruns the fine-tuning step with a decremented batch size, thereby decreasing memory load. Upon four consecutive failures, the run yielding the best *best.pt* model file in the *attempts* folder is selected to be passed on to subsequent fine-tuning steps. And even if a rerun has completed, the parent script will check in the *attempts* folder if a previous failed attempt has yielded better validation results than the completed one. The model passed on to the next fine-tuning steps will always be the one yielding the best model, regardless of if its training has completed or not. In between each fine-tuning step, the best *run* directory is copied to a *detect* folder and the *attempt* folder is deleted. Upon total completion of the multi-phase fine-tuning algorithm, this *detect* folder will contain the best (re)run for every fine-tuning step.

Below is a pseudo-code describing this failure monitoring. It defines the method

TRAIN_AND_MONITOR which was used in Algorithm 1.

Algorithm 2 System failure monitoring

1: **procedure** TRAIN_AND_MONITOR($dataset$, $best$, $frozen_layers$, $lr0$, $batch$)
2: $i \leftarrow 1$
3: $model \leftarrow$ TRAIN($dataset, best, frozen_layers, lr0, batch$)
4: MAKE_DIR($attempt_folder$)
5: COPY_TO($model, attempt_folder$)
6:
7: **while** HAS_CRASHED($model$) and $i < 4$ **do**
8: $model \leftarrow$ TRAIN($dataset, best, frozen_layers, lr0, batch - i$)
9: COPY_TO($model, attempt_folder$)
10: $i \leftarrow i + 1$
11: **end while**
12:
13: $best_model \leftarrow$ FIND_BEST_MODEL($attempt_folder$)
14: COPY_TO($best_model, detect_folder$)
15: REMOVE_DIR($attempt_folder$)
16: **return** $best_model$
17: **end procedure**

Note that in line 7, *TRAIN* is called with $batch - i$, effectively reducing batch size in order to decrease memory load. In a perfect-case scenario, every training step completes and no re-running occurs. In practice however, memory was regularly overloaded, leading to reruns with reduced batch size. The following graph shows the evolution of batch size for every fine-tuning step, both in theory (perfect-case scenario with no re-runs) and in practice (reduced batch size upon crashing).

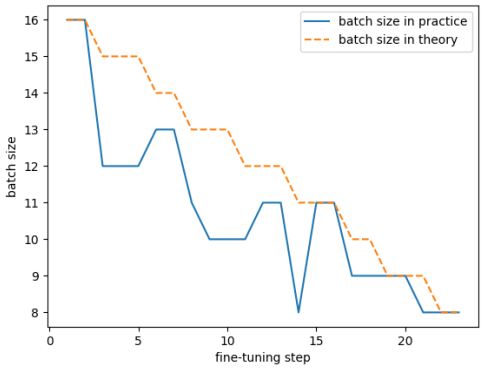


Figure 2.13: Evolution of batch size vs. fine-tuning step

At every point where the two lines do not meet, a re-run had to occur, since batch size was reduced. This actually happened for most of the fine-tuning steps, and only 8 steps out of the 23 completed at first try. Six steps even required the maximum number of tries (4 attempts). This shows the importance of memory leaks and inefficient garbage collection: if in theory our 8 GBytes RAM could handle such memory load, the result was very different in practice. This failure monitoring strategy allowed us to deal with memory overloads, without having to downgrade our training configuration to lower resolutions or model sizes.

Chapter 3

Active Learning

Dataset size plays a critical role in CNN performance. Several studies about the impact of the number of training images on model accuracy confirm this insight. One study referred to this phenomenon as the *Unreasonable Effectiveness of Data in Deep Learning*, spurring the vision community to develop collective efforts in building larger datasets[46]. In general, studies show a logarithmic impact of dataset size on model accuracy[43][47] - that is, the performance gains are substantial initially but diminish as the dataset grows larger. Given our current dataset of only 139 training images, we are situated near the origin of this curve, especially when compared to established datasets (COCO has 330k images and ImageNet over 14 million[45][50]). This means that increasing the size of our dataset is almost guaranteed to have a significant effect on our model performance. With this in mind, we decided to prepare our dataset and model for easy labeled image addition and retraining, so as to pave the way for performance enhancement in the future of our application.

However, this perspective of hand-annotating more and more images in the global context of computer visions poses financial, ethical and environmental concerns, as well as being repetitive and time-consuming.

Most well-established large-scale datasets had recourse to crowdsourcing platforms like *Amazon Mechanical Turk* to label their data, where online workers follow detailed instructions and are paid per task. This can be expensive, and the annotation of ImageNet can be estimated to have cost several hundred thousand dollars just to pay off workers[54][55]. Moreover, such recourse to crowdsourcing platforms has raised ethical questions. Reports indicate that the data labeling industry often exploits workers in developing countries, with some earning as little as a few cents per task[1], and their median hourly wage reaching only $2.83 when accounting for the unpaid tasks of searching for work and managing payments[49].

Lastly, the increasing need for large-scale neural networks, which need to be trained on extensive datasets, has raised environmental concerns[34]. Storing and processing these datasets necessitates the use of large-scale data centers, which require a lot of electricity as well as water for cooling purposes. Moreover, bigger datasets also mean a longer and more energy-hungry training, since every image in the dataset must be processed at every epoch. As such, energy consumption associated with training can be expected to scale at least linearly with dataset size, though the exact relationship depends on various factors including model complexity and training parameters.

Therefore, if dataset growth can be viewed as a promising option for performance enhancement in the field of deep learning, it must be used with care because of its financial, ethical and environmental implications. Active learning techniques have emerged in this context, aiming at selecting images optimally from unlabeled data. Thereby, they reduce the amount of data needed to achieve high performance. As a result, active learning can lower annotation costs, diminish reliance on potentially exploitative crowdsourcing platforms, and decrease the energy demands of training. Active learning is therefore a promising lead towards sustainable and ethical AI.

Given the limited size of our current dataset (139 images), acquiring additional labeled data is essential to improving the detector's performance. However, the annotation budget of the IRSNB is highly constrained; the dataset provided to us consisted entirely of unlabeled entomological box images, requiring us to annotate our sample manually. Exploring active learning techniques is thus both relevant for the future of our detector, as well as in the bigger picture of research towards sustainable AI.

3.1 Merging strategies

We implemented two merging strategies, which define the way new images are going to be added to the dataset. Specifically, we are deciding how many new images are going to be copied to the validation subfolder, and how many will be put in the training subfolder. Defining a strict merging procedure is important when fine-tuning on a pre-trained model, because we want to avoid leakage. Leakage refers to the inclusion of information during model training that wouldn't be available when making real-world predictions[25]: typically, testing on "seen" images, which were in the model's training set. This leads to overly optimistic performance metrics and gives a misleading impression of the model's effectiveness.

In our case, this could happen when we are re-splitting the new grown dataset: if in the new validation set there are images which used to be in the previous training set, validation performance monitoring will not be computed on "unseen" images anymore, and this will likely lead to overfitting. Therefore, a first strong requirement of our merging strategies is to never put into the new validation set an image that was in the previous training set.

In the following subsections, S will refer to the original size of the dataset (training set + validation set), and k to the number of new images. When written in bold, $\mathbf{S}$ and $\mathbf{k}$ will refer to actual sets of images, and not simply their size. Therefore, $|\mathbf{S}| = S$. The proportion of training images selected for validation will be referred to as val_split, and will be comprised between 0 and 1. In practice, we chose a value of 0.2 for the validation split ratio.

Lastly, extensions and subscripts will be used together with S and k. The extensions $_train$ and $_val$ will refer to the number of images put into the training and validation set. For example, S_train refers to the number of images in the original training subfolder, and k_val to the number of new images which will be copied to the validation subfolder. The temporal subscripts t will refer to dataset size at a given iteration. Therefore, $S_{t+1} = S_t + k_t$, as the dataset size at iteration $t + 1$ results from the merging of the original dataset and the new images of the previous iteration.

3.1.1 Validation split

This strategy is the most straightforward one: the split of the new images will correspond to the validation split of the original dataset. This means a random selection of $k_val = k \times val_split$ will be put into the validation set, and the remaining $k_train = k \times (1 - val_split)$ will be copied to the training set. Using set notation, we have $\mathbf{S_val}_{t+1} = \mathbf{S_val}_t \cup \mathbf{k_val}_t$.

Although this merging strategy has the advantage of simplicity, it can slow down the effectiveness of retraining. Indeed, training performance monitoring is based on the validation MaP. In our active learning case, new samples of images are typically added little by little to the dataset, and so very often we have $k \ll S$. Therefore, we also have:

$$
\begin{aligned}
k_t \times val_split &\ll S_t \times val_split \\
k_val_t &\ll S_val_t \\
S_val_{t+1} &= S_val_t + k_val_t \\
\frac{S_val_t}{S_val_{t+1}} &\simeq 1
\end{aligned}
$$

And since with this merging strategy we have $\mathbf{S_val}_t \subset \mathbf{S_val}_{t+1}$, we can conclude that the new validation set will be sensibly the same as the previous one, with only a very small number of new images, relatively to the original validation set size.

Best model selection is based on validation MaP, and in this configuration training will select models which have stayed very similar to the original pretrained one, since validation set has barely changed. In practice, using the active learning simulation described in Chapter 4, we noticed that performance tended to plateau when using this merging strategy.

3.1.2 Validation replacement

In this merging strategy, we move images from the original validation subfolder to the training subfolder, and replace them by the new images. New images will thus form the new validation set.

Formally, the desired new validation size is $S_val_{t+1} = val_split \times S_{t+1} = val_split \times (S_t + k_t)$. The proportion of new images added to the validation set will be 1 if $k_t < S_val_{t+1}$ (all new images are copied to the validation set), and S_val_{t+1} otherwise (k_t exceeds the desired validation size). Therefore, we have $k_val_t = min(k_t, S_val_{t+1})$. The number of new images added to the training set will be $k_train_t = k_t - k_val_t$.

Let's define by $\mathbf{M_t}$ the *moved images*. They are the images from the original dataset S_t which will be moved from the validation set S_val_t to the training set S_train_{t+1}. These images, originally in the validation set, will be effectively "replaced" by the new images. Formally, we define $\mathbf{M_t}$ as:

$$\mathbf{M_t} \equiv \mathbf{S_val_t} \cap \mathbf{S_train_{t+1}}$$

The number of moved images $|\mathbf{M_t}| = M_t$ depends on the desired validation size S_val_{t+1} and on the number of new validation images k_val_t. There are two case-scenarios.

- $k_val_t = S_val_{t+1}$. This means that the number of new images is sufficient to reach the desired validation set size. In this case, all images from the original validation set $\mathbf{S_val_t}$ will be moved to the training set $\mathbf{S_train_{t+1}}$, giving $M_t = S_val_t$.
- $k_val_t < S_val_{t+1}$. This means that k_val_t is not sufficient to reach the desired validation size. In order to reach this desired size, some images from the original validation set $\mathbf{S_val_t}$ will have to stay in the validation set $\mathbf{S_val_{t+1}}$, hence reducing the number of moved images M_t. Specifically, we will first select k_val_t images to be "replaced" by the new validation images, and then we will randomly remove $(S_val_{t+1} - S_val_t)$ images from the selection, so that they stay in the validation set and we can reach the desired validation size. M_t is thus given by the following formula: $M_t = k_val_t - (S_val_{t+1} - S_val_t)$.

Therefore, we have the following formula for M_t:

$$M_t = \begin{cases} k_val_t - (S_val_{t+1} - S_val_t) & \text{if } k_val_t < S_val_{t+1} \\ S_val_t & \text{if } k_val_t = S_val_{t+1} \end{cases}$$

Using the formulae described above for k_val_t, S_val_t and S_val_{t+1}, we can simplify this formula and get M_t in terms of S_t, k_t and val_split:

$$M_t = \begin{cases} (1 - val_split) \times k_t & \text{if } (1 - val_split) \times k_t < val_split \times S_t \\ val_split \times S_t & \text{otherwise} \end{cases}$$

Using those formulaes, the algorithm for the *validation replacement* merging strategy can be implemented with ease. The strategy is described in the pseudo-code below, where sets are written in bold text and cardinalities in regular text, just like before:

Algorithm 3 Validation replacement merging strategy

procedure VALIDATION_REPLACEMENT($\mathbf{S}$, $\mathbf{k}$, val_split)
    $k_val \leftarrow min(k, val_split \times (S + k))$
    $\mathbf{k_val} \leftarrow$ RANDOM_SELECT($\mathbf{k}, k_val$)
    $\mathbf{k_train} \leftarrow (\mathbf{k} - \mathbf{k_val})$

    $condition \leftarrow (1 - val_split) \times k < val_split \times S$
    if *condition* **then**
        $M \leftarrow (1 - val_split) \times k$
    else
        $M \leftarrow val_split \times S$
    end if

    $\mathbf{S_train} \leftarrow$ TRAIN_FOLDER($\mathbf{S}$)
    $\mathbf{S_val} \leftarrow$ VAL_FOLDER($\mathbf{S}$)
    $\mathbf{M} \leftarrow$ RANDOM_SELECT($\mathbf{S_val}, M$)

    $\mathbf{S_val} \leftarrow \mathbf{S_val} - \mathbf{M}$
    $\mathbf{S_train} \leftarrow \mathbf{S_train} \cup \mathbf{M}$
    $\mathbf{S_train} \leftarrow \mathbf{S_train} \cup \mathbf{k_train}$
    $\mathbf{S_val} \leftarrow \mathbf{S_val} \cup \mathbf{k_val}$
end procedure

Below is a diagram depicting the merging strategy visually. M and k_val are given by the formulae detailed above.

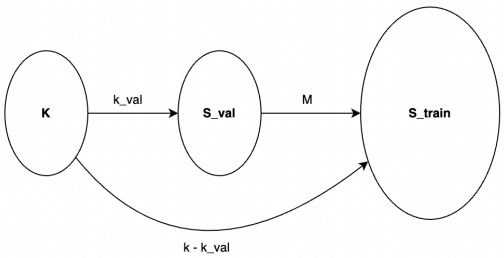


Figure 3.1: Validation replacement strategy diagram

Validation replacement solves the problem encountered in the previous *validation split* strategy, because now in the case $k \ll S$, we have $k_val = k$ and

$M = (1 - val_split) \times k$. All new images will thus be added to the validation set, and M images will be removed from it. This effectively changes the validation set on which best model selection is based, enabling weights to really be updated significantly. This theoretical insight was confirmed empirically, as we observed performance increase at each iteration using validation replacement. Moreover, the strategy is crucial for the *supervised hybrid strategy* introduced in Subsection 3.2.5.

3.2 Subsampling strategies

Subsampling strategy is the central idea behind active learning. The goal is to select optimally a subsample of images from the unlabeled pool of images. This subsample will then be annotated, and added to the training set for a subsequent retraining of the detector.

Researchers in active learning have used various pool-based sampling techniques, with varying levels of success. In general, efficient active learning heuristics include informativeness (black-based or white-based), representativeness, diversity, density and rareness[17]. In this report we will focus on informativeness in an uncertainty-based strategy (Subsection 3.2.3), and on diversity for our uniform partition (Subsection 3.2.2) and our diversity-based subsampling (Subsection 3.2.4). Finally, we will introduce our new supervised hybrid strategy, which uses labeled validation images to maximize informativeness, and combines it with diversity-based subsampling.

Note that all of our strategies are exploratory rather than exploitative. Specifically, we focus on selecting images on which our model performs badly, either by directly evaluating the model's confidence on them (informativeness), or by selecting images that differ significantly from the current training set (diversity). Our rationale was that exploitation is only relevant if there is a strong basis to exploit, and in the context of active learning this means a large and representative training set. With a training set of only 139 images, we cannot really claim representativeness, and therefore we believed that it was best to focus on sampling diverse and informative images when enlarging our dataset. However, we will see in Chapter 4 that excessive exploitation slows down performance for feature distribution-based strategies.

The notation for sets will be the same as before. $\mathbf{S}$ is the original training set and $\mathbf{k}$ is the additional images sampled from the unlabeled pool $\mathbf{U}$. Subscripts t will denote iterations. Therefore, we have:

$$\mathbf{U_{t+1}} = \mathbf{U_t} - \mathbf{k_t}$$
$$\mathbf{S_{t+1}} = \mathbf{S_t} \cup \mathbf{k_t}$$

In the following subsections, it will be understood that the *validation replacement* strategy will be used to merge $\mathbf{S}$ and $\mathbf{k}$ at each iteration.

3.2.1 Random subsampling

This strategy consists simply in selecting a random subsample $\mathbf{k}$ from the unlabeled pool $\mathbf{U}$. The motivation for such a trivial strategy is two-fold.

Firstly, it will be used to test and compare our retraining strategies (see Section 3.3). The way we retrain the detector on the enlarged training set at each iteration is crucial to increase performance. Random subsampling will allow us to compare our retraining strategies on even ground, minimizing the effect of the subsampling to assess the impact of sole retraining, isolated from any other strategy.

Secondly, random subsampling will be used as a baseline to prove the positive impact of active-learning subsampling strategies on the retrained model performance. If an AL strategy significantly outperforms random-subsampling, we will be able to conclude that it is effective (see Chapter 4).

3.2.2 Uniform partition subsampling

This strategy is highly specific to our entomological application, and to our dataset of images from the IRSNB. As such, it is not portable to other contexts, but its simple mechanism and ease of implementation makes it a topic of interest.

Images in our dataset were loaded in such a way that their names have a path-like structure, which includes the subfolders they were in, in their original unlabeled dataset. Those subfolders were classified by families, and sometimes even by species. Therefore, boxes from the same subfolder usually contain similar species. A simple alphanumerical sorting of their filenames is thus sufficient to have similar images close by in the image listing, and diverse boxes far away from each

other. For example, in the screenshot below of a small part of our entire dataset image name listing, insect of the family *staphylinidae* are grouped together.

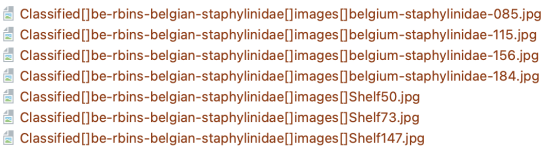


Figure 3.2: Image names in our dataset

Therefore, maximizing the pairwise index distance between each image name already guarantees a certain level of diversity. We define the *index distance* as the difference between the index of the two images' names in the dataset listing. For instance, in Figure 3.2 above, the index distance between *Classified[]...staphylinidae-085.jpg* and *Classified[]...Shelf147.jpg* is 6.

Subsampling k images from an unlabeled pool of size U by enforcing each selected index to be a mutliple of $U \div k$, rounded to the closest integer, is enough to maximize pairwise index distance between each image. So as to include some randomness in the strategy, and to avoid the first image to be systematically selected, we add a random offset to each selected index. Therefore, given a random offset $offset \in [0, U-1]$, the set of selected image indices $\mathbf{I}$ is given by:

$$\mathbf{I} = \{i \times \lfloor U \div k \rceil + offset \mid 0 \leq i < k\}$$

This strategy is easy to implement and runs in approximately $\mathcal{O}(1)$, since it does not require any pass into the unlabeled pool, nor does it need any image processing. It achieves a certain level of sample diversity in a very fast and straightforward way, but by definition it is highly dependent on image names and it will fail to identify certain similarities and differences. For instance, it will identify all mixed boxes (their file names starting with "Mixed") as similar, when this actually means that their content is diverse in terms of species. Moreover, it will consider species starting with the same letter to be similar, when this is of course not necessarily true.

Note that this strategy was originally used to sample our dataset of 139 training images and 28 test images. It seemed to be the most simple way to promote diversity in our set without having recourse to any sort of image processing. A little twist was added however to the strategy, to ensure that 25% of our training

images were "mixed" boxes, that is, boxes which contained different species of insects within them. The rationale was that if the detector was able to correctly detect insects of different species, even in a single box, it would have a higher chance of achieving good generalization.

3.2.3 Subsampling based on uncertainty

This subsampling strategy aims to maximize the informativeness of new instances, that is, the effectiveness of that instance on reducing the uncertainty of the model[20]. Typically, such strategy will select instances on which the model is most "uncertain". Labeling those problematic instances and adding them to the training set is expected to increase performance, as the focus will be put on solving the model's weaknesses.

Uncertainty can be defined in various ways, depending on the AL informativeness-based method in use. Because of their simplicity and proven effectiveness, we will focus on the traditional black box based methods of computing uncertainty. Those methods involve inferring on unlabeled instances with the current model and computing uncertainty for every image from the sole softmax output of the detector, which effectively gives confidence levels for each class. Since our problem is single class, there will be only two softmax outputs per image: confidence levels for *insect* and *background*. Therefore, there is no need for computing uncertainty based on entropy or margin of confidence between the top-2 rated classes, as those metrics are only relevant in multi-class contexts. Instead, we simply define uncertainty of a prediction i as:

$$U_i = 1 - P_i(class = insect)$$

With P_i given by the softmax output of the *insect* class. To compute a global uncertainty for each image, we need to define how to aggregate the uncertainty scores of each prediction in the image. We chose to make an average over all predictions, taking into account every single bounding box. This enables us to compare different images on even ground, even when their number of predictions differs significantly. Therefore, with c being the number of predictions in the image, we define image uncertainty score as:

$$U = \frac{\sum_{i=1}^{c} U_i}{c}$$

Our uncertainty-based subsampling strategy will compute uncertainty for all images in the unlabeled pool, and extract the k images with highest uncertainty score.

This strategy has however one clear disadvantage, though it is specific to our dataset. Our unlabeled pool contains a significant number of empty entomological boxes, which are not going to be of great help for model retraining. We observed empirically that detectors, even when trained on extremely small datasets (12 images), will usually never predict a high confidence bounding box in an empty image. In other words, processing empty boxes is not a challenging task for our detectors, and adding empty boxes to the dataset will likely not result in increases in performance. The problem is that our uncertainty-based subsampling method will tend to promote those empty boxes for selection. Indeed, although our model will not predict high-confidence boxes in empty images, it will however predict boxes with low confidence scores. Empty images uncertainty scores will thus be high, meaning that they will be selected by the subsampling method.

However this weakness can be put into perspective in two ways. Firstly, this is actually more about our specific dataset than about the subsampling method itself, as most object detection datasets do not contain empty images. Moreover, in practice, empty boxes selected for annotation will by definition be very easy to annotate (no annotation needed) and our merging strategy automatically discards any images without annotations. Therefore, the subsampling strategy can at the same time be used as a tool to discard empty images. If *k_empty* out of the k images selected are empty, at the next training iteration we will have:

$$\mathbf{S_{t+1}} = \mathbf{S_t} \cup (\mathbf{k_t} - \mathbf{k_empty_t})$$
$$\mathbf{U_{t+1}} = \mathbf{U_t} - \mathbf{k_t}$$

At iteration $t + 1$, empty images in $\mathbf{k_empty_t}$ are effectively permanently deleted from the dataset, which is what we want. Note that in our sample of 167 images, we did not include any empty boxes.

Figure 3.3 shows the image with highest uncertainty in our test set (uncertainty of 0.973), together with our model's predictions. This selection can be easily explained, as it is actually a cropped single insect and not an entomological box. The

insect being way closer to the camera lens than any of the examples our detector has seen in its training set, it effectively struggles to detect it. Due to our model's bias towards more distant insects, this image is thus a very informative one for our detector.

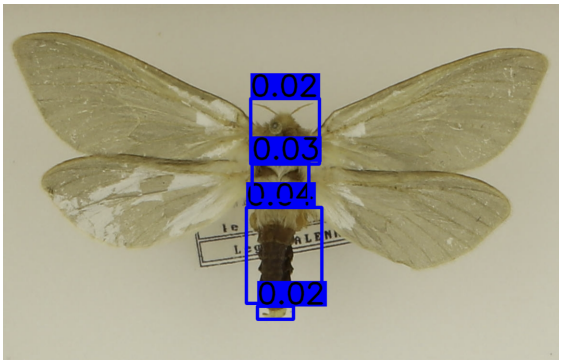


Figure 3.3: Highest uncertainty image in test set

3.2.4 Subsampling based on diversity

Here, we are trying to maximize the diversity of new images, as well as their semantic distance from **S**. In other words, we want the strategy to select species that are morphologically different, both from each other (other images in **k**) and from the instances of the training set **S**. Therefore, it is a purely exploratory strategy.

Distance definition

To be able to mathematically claim that an image is different from another, we need to define how to compute distance between them. Typically, in the context of object detection, this is either done at the detector's feature level or at the label level[17]. In our single-class case, label-level distance is of course irrelevant. On the other hand, feature-level distance could work but it can be hard to interpret. Moreover, deep layers of our single-class detector are at risk of loosing information about key morphological differences between insects. Indeed, since all insects are treated as a single class, features may only indicate the presence of an insect, with no regards for morphological differences between instances of that class.

To avoid this, we decided to define a "representation" for each entomological box, which we call *average detection*. This representation is obtained through the

following process: first, if we are dealing with unlabeled images, we infer on them with our current model. Second, we select a small sample out of those predicted bounding boxes, by choosing those with the highest confidence level to minimize the number of false positives in the *average detection* (if we are dealing with labeled images, we select this sample randomly from the ground-truth labels). Third, we resize cropped bounding boxes from the small sample to resolution (256, 256), and we make a pixel-wise average of them. This *average detection* will be used as a representation of what kinds of insects the box effectively contains.

The reason for computing *average detection* on a small subset of the predictions is two-fold: first, resizing boxes is costly, and for boxes with hundreds of insects in them this becomes problematic. Second, *average detection* informativeness is lost when combining too many predictions. Below is a comparison of *average detection* computed on a sample of 5 predictions (left) and on 10 predictions (right).

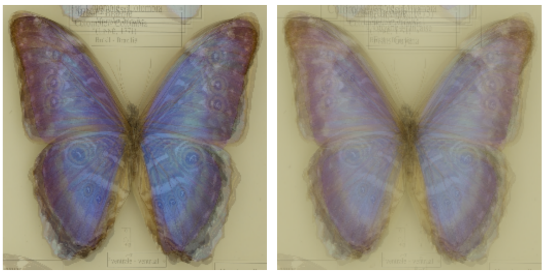


Figure 3.4: Average detections computed on 5 and 10 bounding boxes

And this blurring effect on the *average detection* worsens with increasing prediction sample size. However, several bounding boxes are still needed if we want to have a representative *average detection* of the content of the box. This is especially true for mixed entomological boxes (several species in a single box), where averaging on several predictions is successful in representing the different species it contains.

To make average detections semantically meaningful, we pass them into the EfficientNetB0 model pretrained on ImageNet, from which we have detached the classification head. As a result we end up with a feature map for each average detection. This is important because computing pairwise distance between two feature maps will actually evaluate the difference between what both average detections contain semantically. Those feature maps will be largely invariant on unimportant differences, like a slight translational shift or a difference in lighting for example. On the other hand, a bare pixel-wise distance between average detections would have deemed those differences important, and could have erroneously considered

far apart detections of the same species.

Distance between two feature maps is evaluated by first flattening both maps to 1D vectors and then computing Euclidean distance between them. Therefore, for flattened feature maps x and y of size n, we have:

$$d(x, y) = \sqrt{\sum_{i=1}^{n}(x_i - y_i)^2}$$

To sum up, we define distance between entomological boxes as the euclidean distance between flattened feature maps, which were extracted by EfficientNetB0 from the *average detection* of the image (i.e. pixel-wise average of the five highest-confidence bounding boxes in the image).

Subsample selection

Now that we have clarified how distance among images is computed, we must still explain how the diversity-based selection is performed. As a reminder, we aim to maximize both the diversity among the k images of the selection, as well as their distance from the original dataset $\mathbf{S}$.

This problem can be reformulated as a variation of *coreset selection*. In computational geometry, a coreset of an input set is a subset of points, such that solving a problem on the coreset yields similar results as solving the problem on the entire input set[23]. The small variation that our specific case adds to this problem is the fact that we have a fixed original dataset $\mathbf{S}$, to which we have to add a selection $\mathbf{k}$ so as to obtain a coreset of the total pool of images $\mathbf{S} \cup \mathbf{U}$. The coreset is thus effectively $\mathbf{S} \cup \mathbf{k}$, but $\mathbf{S}$ is fixed and we can only sample k additional images from $\mathbf{U}$.

Coreset selection is typically performed according to a set function $f(\mathbf{C})$, with $\mathbf{C}$ being the selected subset. The function f will measure the subset's ability to represent the entire input set. In our case, since $\mathbf{S}$ is fixed, f will be a function of $\mathbf{k}$. Formally, we define our diversity-based set function as:

$$f(\mathbf{k}) = \frac{1}{k(k+S-1)} \sum_{i \in [1,k]} \Big(\sum_{j \in [i,k]} distance(\mathbf{k}_i, \mathbf{k}_j) + \sum_{m \in [1,S]} distance(\mathbf{k}_i, \mathbf{S}_m) \Big)$$

Where distance is computed according to the subsection above. For each image in $\mathbf{k}$, distance is thus computed with other images in $\mathbf{k}$, as well as with images in

S. All in all, this makes $k(k + S - 1)$ distinct distances summed up, so we divide by this value to get the average distance between images.

However, we know that those kinds of problems - where the goal is to find a subset with the maximum benefit - are NP-hard[23]. Finding an optimal solution is thus computationally challenging. Instead, we used a suboptimal heuristic algorithm to find a local maximum to our diversity-based set function f. Specifically, we start with the original dataset $\mathbf{S}$ and an empty sample $\mathbf{k}$, and then we incrementally add an image x to $\mathbf{k}$ which maximizes $f((\mathbf{S} \cup \mathbf{k}) \cup \{x\})$, until we reach the desired sample size k. This is known as *greedy selection*.

An efficient way to find x at each iteration is by using the *max-min algorithm* described in this article about most distant points selection[19]. For each image in $\mathbf{U}$, we compute the distance to all previously selected images $\mathbf{S} \cup \mathbf{k}$ and we identify the minimum out of those distances. Thus, to every image in $\mathbf{U}$, we assign a score corresponding to their distance from the closest image in $\mathbf{S} \cup \mathbf{k}$. Then, we select x as the image in $\mathbf{U}$ with the maximum assigned score. To make this a bit more clear, we provide a pseudo-code below, where *DIST* is the image distance computed like specified in the latter subsection, and *nested list comprehensions* (for loops inside of lists) are used in a python way[36]:

Algorithm 4 Greedy diversity-based selection

```
1: procedure GREEDY_SELECT(S, U, k)
2:     k ← {}
3:     size ← 0
4:
5:     while size < k do
6:         distances ← [[DIST(m, n) for m in S ∪ k] for n in U − k]
7:         closest_images ← [MIN(m) for m in distances]
8:         i ← ARGMAX(closest_images)
9:         k ← k ∪ U_i
10:        size ← size + 1
11:    end while
12:
13:    return k
14: end procedure
```

And this runs in polynomial time, specifically in $O(Uk(S + k))$ multiplied by the distance function complexity. Below is an example of the output of our

diversity-based selector, using a small dataset of 12 images (sampled with uniform partition from our training set) for **S**, our test set for **U** and $k = 4$. The images selected are portrayed by their *average detections*.

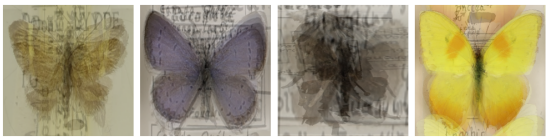


Figure 3.5: Output of diversity-based subsampling with $k = 4$

Note that if this solution contains only *lepidopterae* and *microlepidopterae*, it is because the small sampled **S** used for testing happened to contain mostly *coleopterae*. Therefore, diversity subsampling was successful in identifying new boxes which differed from those in **S**.

3.2.5 Supervised hybrid stategy

Both uncertainty-based and diversity-based methods explained above involve inferring with the current model on the unlabeled images in **U**, and putting some trust in the level of confidence it is claiming to have. Indeed, the uncertainty-based strategy relies entirely on those confidence levels, and the diversity-based method takes the five highest-confidence boxes to make *average detections*. But what if the detector fails to provide reasonable confidence levels ? What if it detects a significant number of false positives with a high confidence threshold ? This could interfere negatively with how those subsampling strategies work.

However, the very essence of AL resides in the idea of selecting optimal samples from an *unlabeled* pool. In the context of object detection, this means we have no information of where the objects of the image might be. The only way to extract the information necessary to maximize a subsampling heuristic is thus to use an object detector, even if it performs poorly. But what if we had access to labeled images that were unseen in our model's training to evaluate the types of objects our detector struggles on ? We could then sample images from the unlabeled pool that are the most similar to those identified problematic instances.

In fact, we do have access to such unseen labeled images; the validation set. *Supervised hybrid strategy* is based on the idea of identifying problematic instances in a supervised way by leveraging the labels of the validation set. Specifically, we will infer with our current model on each validation image, and comparing with the ground-truth labels we will compute an MaP-score for each of them. Then, the 10% worst performing images will be used to guide subsampling, and for each of those we will compute a representation using *average detection*. The subsampling heuristic will then be to find images in $\mathbf{U}$ that are the most similar to at least one image in this worst-performing sample, which we call $\mathbf{w}$. We define the heuristic set function to maximize as:

$$f(\mathbf{k}) = -\frac{1}{k} \sum_{i \in [1,k]} argmin_{j \in [1,w]}\ distance(\mathbf{k}_i, \mathbf{w}_j)$$

Notice the minus on the right-hand side of the equation. This is because it is actually a minimization problem, but we convert it to a maximization problem to stay coherent with the previous subsection.

In English, we want to minimize the average distance of each image in $\mathbf{k}$ from their respective closest image in $\mathbf{w}$. To do this, we first compute all distances from images in $\mathbf{U}$ to images in $\mathbf{w}$. Then, to every image in $\mathbf{U}$, we assign a score corresponding to the distance to the closest image in $\mathbf{w}$. Finally, we select the image in $\mathbf{U}$ with the minimal score. We then repeat the process until obtaining a sample of size k. The algorithm is depicted in the pseudo-code 5.

Merging with the original dataset will then be done using the *validation replacement* strategy, which replaces the images of the validation set. This highlights the importance of *validation replacement* for this strategy: $\mathbf{w}$ effectively changes at each AL iteration.

However, we identified a clear weakness to this strategy: say our current model particularly struggles on a subset $\mathbf{A}$ of the entire pool of images $\mathbf{S} \cup \mathbf{U}$. If $\mathbf{S_val} \cap \mathbf{A} \neq \emptyset$, our strategy will most likely select images in $\mathbf{A}$, and there is even a chance that the selection will be entirely composed of such images. At the next iteration, since the validation set will be composed of only images in $\mathbf{A}$, worst-performing images $\mathbf{w}$ will still be in $\mathbf{A}$. The subsampling strategy will thus continuously sample images in $\mathbf{A}$ until there is no more and $\mathbf{A} \cap \mathbf{U} = \emptyset$. This behaviour will continue even though the model might actually have become good at detecting objects in images from this subset $\mathbf{A}$, slowing down general performance increases in a counter-productive way.

Therefore, we decided to make this strategy *hybrid*. The supervised strategy focuses on informativeness, but combining it with a diversity-base heuristic could solve the problem described above. A proportion *supervised_split* of the new sample **k** will be chosen similar to the problematic instances in the validation set (informativeness), and the remaining $(1 - supervised_split) \times k$ images of the sample will be chosen to maximize diversity both with regards to those $supervised_split \times k$ first images, as well as with images in **S** (diversity). Hence the term *hybrid*, because we are using two different heuristics. Using the example described in the previous paragraph, we are now certain that the subsampling strategy will not make a selection $\mathbf{k} \subset \mathbf{A}$, because the diversity heuristic enforced selection of images which are outside of **A**. Below is a pseudo-code describing the entire hybrid strategy, with *GREEDY_SELECT* being the diversity method from pseudo-code 4.

Algorithm 5 Supervised hybrid strategy

```
1:  procedure SUPERVISED_SELECT(S, U, k, current_model, supervised_split)
2:      S_val ← VALIDATION_SET(S)
3:      predictions ← INFER(current_model, S_val)
4:      scores ← GET_MAP_SCORES(predictions, S_val_labels)
5:      worst_indices = GET_MIN_INDICES(scores, size = S_val ÷ 10)
6:      w ← S_val_{i∈worst_indices}
7:
8:      distances_to_worst = [[DIST(m, n) for m in w] for n in U]
9:      closest_distances ← [MIN(m) for m in distances_to_worst]
10:     selected_indices ← GET_MIN_INDICES(closest_distances,
          size = supervised_split × k)
11:     k ← U_{i∈selected_indices}
12:
13:     k ← k ∪ GREEDY_SELECT(S ∪ k, U − k, (1 − supervised_split) × k)
14:
15:     return k
16: end procedure
```

And we chose 0.75 as our default value for *supervised_split*. Using our entire training dataset of 139 images, which has a validation set of 28 images, the two "worst-performing" images are represented below by their *average detections*.

Figure 3.6: Worst performing images of the validation set

The algorithm correctly identified the main weakness of our model: very small *coleopterae*, which are hard to detect given the chosen 640×640 resolution (more on this in Chapter 4). When subsampling 4 images with *supervised hybrid strategy* from our test set, we obtain the *average detections* below:

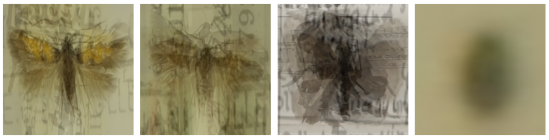


Figure 3.7: Supervised hybrid sample of 4 images

The *average detection* on the right represents the only box with very small *coleopterae* from our test set. Therefore, the strategy accurately samples images on which our model struggles. The image sampled with the diversity heuristic is represented by the 3$^{\text{rd}}$ *average detection* (from left to right).

3.3 Retraining strategies

The goal of AL sampling techniques is to maximize performance gains obtained by retraining on datasets of increasing size. If sample selection is indeed important in that regard, the way we retrain the current model is also crucial. We define two retraining strategies, *full retraining* and *gradual fine-tuning*.

Note that for both strategies we fully retrain a *post-detection classifier* with EfficientNetB0 weight initialization on the detections of the current model, similarly

to what we did in Subsection 2.1.4. This is because the training time of the classifier is negligible compared to the detector's training time. Therefore, for the sake of simplicity and because it adds negligible overhead, the classifier is fully retrained.

3.3.1 Full retraining

This approach is very straight-forward: we retrain YOLOv8s on $\mathbf{S} \cup \mathbf{k}$ using the *dynamic multi-phase strategy* described in subsection 2.2.3. This has the advantage of simplicity, and prevents a potential risk of *catastrophic forgetting*, where one bad iteration of the process would trigger the downfall of all subsequent iterations. If a model m_i diverges from the local optimum, this does not mean that m_{i+1} will do the same, since retraining is initialized with the YOLOv8s COCO weights.

However, such retraining does not re-use the information gained by the model at the previous iteration. This is quite inefficient, because convergence must be obtained from the very initial state. And since the training set size increases, training will take longer and longer at each step.

3.3.2 Gradual fine-tuning

In order to re-use information gained at previous steps, *gradual fine-tuning retraining* retrains on the previous model, that is, weights are initialized to m_{t-1}. So as to avoid diverging too quickly from the local optimum approached by m_{t-1}, we reduce learning rate $lr0$ by multiplying the initial 0.01 by a $factor$ given by the following formula:

$$factor = max(\frac{k}{S+k}, 0.01)$$

The rationale is that the larger the new sample of images $\mathbf{k}$, the more additional information is added to $\mathbf{S}$, and the more updates to m_{i-1} are going to be needed. Therefore, learning rate will be larger. If, on the contrary, k is very small, then we will barely update the weights of m_{i-1}. However, a minimum factor of 0.01 is set to prevent a retraining that will have no effect on the model weights.

The same rationale is applied to the number of fine-tuning steps, epochs and the patience value. This is to avoid a too long retraining when it is not needed because k is small:

$$
\begin{aligned}
fine_tuning_steps_i &= max(factor \times fine_tuning_steps_{i-1},\ 3) \\
epochs_i &= max(factor \times epochs_{i-1},\ 3) \\
patience_i &= max(epochs_i \div 3,\ 2)
\end{aligned}
$$

This method has the clear advantage of reducing retraining time significantly and re-using information from the previous models. However, there could be a risk of *catastrophic forgetting* where training of m_i is initialized on the bad weights of m_{i-1}, leading to a decrease of performance in comparison with previous steps. Nevertheless, we taylored our *dynamic multi-phase fine-tuning* technique so as to minimize this risk, with careful learning rate tuning and layer unfreezing, early stopping and best model selection according to validation MaP. Empirically, we measured that *gradual fine-tuning* reached performances that were similar to those of *full retraining*, but with a much faster retraining time. In fact, it even outperformed *full retraining* is some cases (see Chapter 4).

Chapter 4

Measures and results

4.1 Test set and metrics

4.1.1 Test set selection

The test set of 28 images was sampled from the unlabeled IRSNB dataset in the same way as the training set. We used *uniform partition sampling* (see subsection 3.2.2) to obtain a diverse and representative selection without the need of image processing. Moreover, we constrained the selection to ensure that it contained 25% of mixed boxes, that is, boxes that contain several distinct species of insect. Good performances on such boxes indicate strong generalization.

The test set was sampled *after* we trained all previous models. This was done to be absolutely certain that no *leakage* had occured. As a result, we can be 100% confident that test images have not been seen in the models' training, and as such they can be used for unbiased performance measurements.

4.1.2 Metrics

The previous state-of-the-art authors evaluated performance by computing the mean and median error with respect to the true count of insects in each box. In contrast, we preferred the more standard object detection metrics of precision, recall and f1-score.

Unlike count-based metrics such as the mean error with respect to the true count, precision, recall, and f1-score consider the spatial accuracy and object-wise correctness of detections. A model could achieve a low mean count error by predicting a number of detections close to the ground truth, even if those detections are

inaccurately placed or correspond to no actual object. In contrast, precision and recall are sensitive to the correctness of each individual detection, making them better suited for assessing real-world performance in object detection tasks. As such, it is those metrics, combined with MaP, that will be used for model comparison.

The first objective of this subsection is to formally define those metrics. Then, we will also explain the decision boundary used to determine if a detection is a true or a false positive.

Precision, Recall and F1-score

Precision, recall, and f1-score are standard evaluation metrics in object detection tasks. Below, we give formal definitions for each of them.

Precision. Precision measures the proportion of correctly identified positive instances (true positives) among all positive predictions made by the model. It is defined as:

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

where TP is the number of true positives (correct detections), and FP is the number of false positives (incorrect detections). High precision indicates a low rate of false alarms, which is particularly desirable in applications where false positives are costly.

Recall. Recall measures the proportion of actual positive instances that were correctly detected by the model. It is given by:

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

where FN is the number of false negatives (missed detections). High recall indicates that most of the relevant objects were detected, which is critical when missing a detection is more problematic than a false detection.

F1-Score. The f1-score is the harmonic mean of precision and recall. It provides a single metric that balances both aspects:

$$\text{F1-Score} = 2 \cdot \frac{\text{Precision} \cdot \text{Recall}}{\text{Precision} + \text{Recall}}$$

The f1-score is particularly useful when both false positives and false negatives are undesirable, giving a more balanced view of performance.

Mean Average Precision and Precision-Recall curve

Mean Average Precision (MaP) is a comprehensive metric widely used for evaluating object detection models. It incorporates both precision and recall across different confidence thresholds, providing a single-number summary of detection performance.

Precision-Recall Curve. The Precision-Recall (PR) curve is a graphical representation that illustrates the trade-off between precision and recall for different confidence thresholds of a model. By varying the threshold, we can observe how precision and recall change, which helps in understanding the model's performance across different operating points.

An ideal model maintains high precision and recall across all thresholds, resulting in a PR curve that stays close to the top-right corner of the plot. Below is a visual depiction of an example PR curve from [53], where two points are highlighted along with their confidence thresholds.

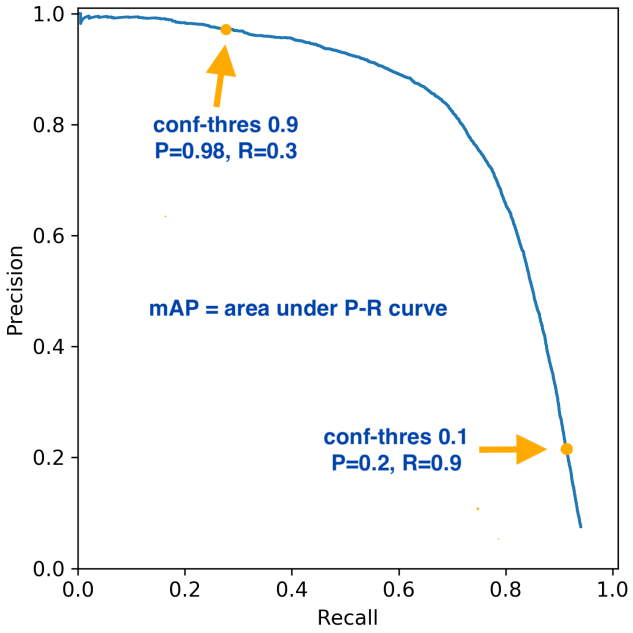


Figure 4.1: Example of a Precision-Recall curve for an object detection model.

Average Precision (aP). For a given class, the Average Precision (aP) summarizes the shape of the PR curve into a single value, representing the area under the curve. It is computed as:

$$\text{aP} = \int_0^1 p(r)\, dr$$

where $p(r)$ is the precision as a function of recall r. In practice, this integral is approximated using numerical methods, such as the 101-point interpolation or all-point interpolation, depending on the evaluation protocol. In this report, we will use the 101-point interpolation.

Mean Average Precision (MaP). Mean Average Precision is the mean of the Average Precision values across all object classes. If there are N classes, it is defined as:

$$\text{MaP} = \frac{1}{N} \sum_{i=1}^{N} \text{aP}_i$$

This averaging ensures that each class contributes equally to the final score, regardless of how frequently it appears in the dataset. In our single-class case, MaP is actually just aP.

MaP is widely regarded as a more holistic and reliable metric for benchmarking object detection systems in both academic and industrial contexts, and as such it will also be used in this report for performance evaluation.

Decision boundary for True Positive determination

To determine whether a detection is a true positive or a false positive, a spatial overlap criterion is used. The most common is the Intersection over Union (IoU), which we have already described in Subsection 2.1.3.

A detection is considered correct if its IoU with a ground-truth bounding box exceeds a predefined threshold (commonly 0.5). Some benchmarks, such as COCO, compute MaP at multiple IoU thresholds (from 0.5 to 0.95 in steps of 0.05 for map@50:95) and aggregate them to a single scalar (map@50:95 computes the average). This provides a more nuanced assessment of localization precision. In this performance evaluation, we will compute map@50 and map@0.5:0.95 under those strict standard requirements.

However, the IoU threshold does not allow a fair and unbiased performance comparison with the previous state-of-the-art. Indeed, the previous state-of-the-art

dataset suffered from a label bias described in Subsection 2.1.2. Therefore, as already mentioned, their detector will tend to draw bounding boxes around the paper supports on which small insects are lying, because that is what it has been trained to detect. True positive determination based on the IoU would consider those detections as false positives, leading to poor performance metrics for their detector.

Therefore, for the sake of fair comparison, we will determine true positives according to the relaxed overlap criterion of Intersection over Smaller Area (IoSA, see Subsection 2.1.3). We will show that even under this relaxed criterion, our model significantly outperforms the previous state-of-the-art on our test set.

This essentially means that larger bounding boxes enclosing small insects will be treated as true positives. In an application where our goal is to count and crop every insect for future numerization and species classification, we believe that such a relaxed localization criterion is actually well-adapted. Therefore, we use IoSA both for unbiased comparison purposes, and because it is adapted to the problem at hand.

Specifically, precision, recall and f1-score will be computed with the IoSA criterion. On the other hand, map@50 and map@0.5:0.95 will be computed formally using IoU.

4.2 Detection performance

4.2.1 Post-detection classifier

Here we will assess the performances of our binary *post-detection classifier* from Subsection 2.1.4.

We are focusing here on the performances of the classifier *final_23.keras*, trained on all detections of our final detection model called *final_23.pt*[1], with minimum confidence threshold 0.01. However, this performance evaluation is also applicable to the *post-detection classifier* trained on the output of the previous state-of-the-art detector (Subsection 2.1.4), which achieves similar results.

First, we inferred on our test set with *final_23.pt*, and labeled the cropped output bounding boxes as true and false positives. We then compared those ground-

[1]We named it *final_23.pt* because it was our final model, and it was trained in 23 fine-tuning steps (remember Subsection 2.2.3)

truth values to the predictions of the *post-detection classifier*. *final_23.keras* achieved a test accuracy of 0.946. Below is a confusion matrix portraying those results:

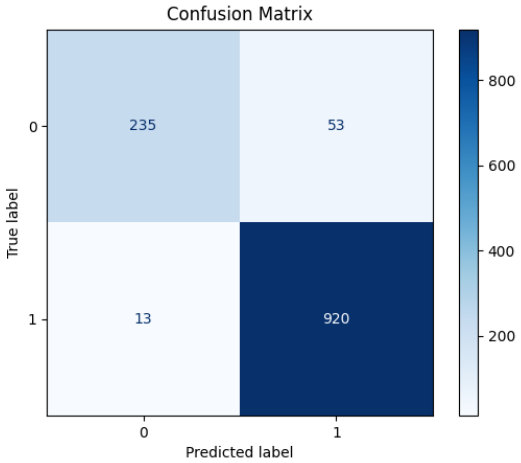


Figure 4.2: Confusion matrix of the *post-detection classifier*

The classifier is slightly biased towards the positive class, with 53 instances misclassified as true positives. This is not surprising given the balance of 80% true positives it was trained on to ensure model convergence. All in all, the global accuracy of 0.946 shows the classifier's effectiveness in increasing the precision of the detector.

Below is an example of 3 randomly sampled misclassifications by the model:

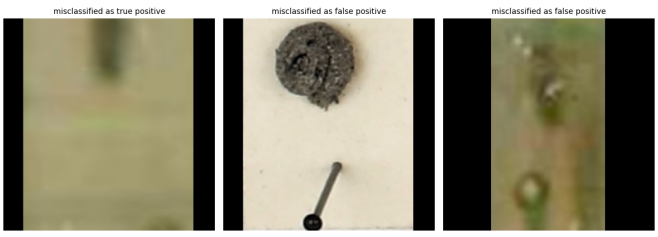


Figure 4.3: Example of errors made by the classifier

All three are examples that are hard to identify even for a human annotator. The left and right images have insufficient resolution, due to the very small size of the bounding boxes. On the other hand, the central image could be easily confused with a cotton ball, or another object. This further proves the strength of our classifier; misclassifications happen on properly hard instances.

By converting dense layers of our classifier to convolutional layers with a single 1×1 filter, and by removing the global average pooling layer, we can actually make our network fully-convolutional - that is, entirely made of convolutional layers. Fully convolutional neural networks (FCNNs) have the advantage of accepting inputs of arbitrary resolution. If we convert our classifier to an FCNN, and then feed it a high-dimensional image, each channel of the output will be the *saliency map* of a given class (remember Subsection 1.3.2).

Below, we superimposed the input (an entire entomological box) and its saliency map for the class *true positive*:

Figure 4.4: Saliency map on an entire box for the class "true positive"

Regions with high scores for the "true positive" class seem to align with actual insect locations. This suggests that our classifier has learned spatially localized features relevant to insect presence, making it effectively applicable to *weakly*

supervised localization despite being trained solely for classification.

4.2.2 Detectors

Here we will compare the performances of 4 distinct inference pipelines: (1) the previous state-of-the-art detector (*coleo_lepido.pt*), (2) the previous state-of-the-art combined with our additional *post-detection filters*, (3) our own fine-tuned detector *final_23.pt* and (4) our own detector combined with its *post-detection classifier*.

The following table shows performance metrics on our test set for each of those pipelines. Note that they are average performance metrics *per image*. This means that single scores are computed for each test image, and then all scores are averaged. Averaging on all images gives a more accurate idea of the model's performance on a single image. Those performance metrics have been computed using the optimal point on the PR curve for each model predictions, that is, the confidence level which yields the closest point to the ideal (1, 1) in the diagram. Each of those optimal confidence levels are specified in the "Optimal Confidence" row of the table. The very last row gives the average inference times per image.

Metric	**coleo_lepido.pt**	**coleo_lepido.pt + classifier**	**final_23.pt**	**final_23.pt + classifier**
Optimal Confidence	0.826	0.646	0.110	0.0111
Precision	0.744	0.870	0.902	0.878
Recall	0.854	0.913	0.882	0.919
F1 Score	0.765	0.876	0.861	0.889
Inference Time (s)	5.45	25.8	0.469	9.66

Table 4.1: Average performance metrics (rounded to 3 significant figures) across different models.

For a fair comparison with the previous state-of-the-art model (*coleo_lepido.pt*) and its label bias, we can assess the f1-score, which was computed using IoSA for true positive determination. Each of the three detection pipelines we have introduced significantly outperform *coleo_lepido.pt*'s f1-score of 0.765. The best pipeline in terms of f1-score is our model *final_23.pt* fine-tuned on YOLOv8s and combined with a binary post-detection classifier, which achieves an average f1-score per image of 0.889. Our model *final_23.pt* reaches a slightly lower f1-score of 0.861

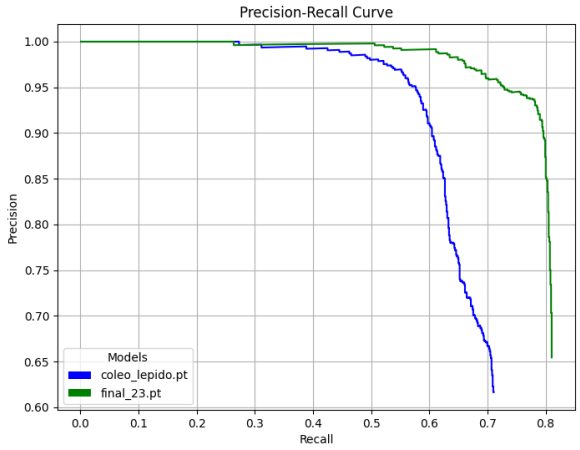
but is 11.6 times faster than *coleo_lepido.pt*.

Mean average precisions, because they incorporate both precision and recall across different confidence thresholds, must by definition not be computed under a fixed threshold, as is done in the table above. Therefore, when evaluating MaP we do not discard any low-confidence predictions. Moreover, MaP is typically computed as a dataset-level metric - that is, all predictions from all test images are collected and marked as true or false positives, and then we compute MaP on the entire dataset. In other words, we do not compute per-image scores as we did for the metrics above. MaP is widely considered as the most comprehensive performance metric for object detection, and as such we compute it under its strict specifications (see Table 4.2 below).

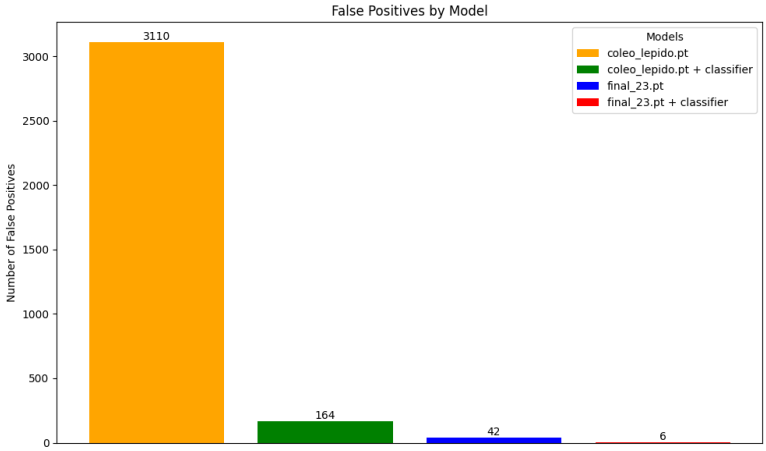
Metric	**coleo_lepido.pt**	**coleo_lepido.pt + classifier**	**final_23.pt**	**final_23.pt + classifier**
mAP@50	0.679	0.712	0.799	0.793
mAP@50:95	0.367	0.354	0.560	0.558

Table 4.2: Mean Average Precision (mAP) metrics across different models, rounded to 3 significant figures.

Across both types of MaP, *final_23.pt* consistently gets the best results. Note that the impact of bad labeling is apparent in this table; map@50 and map@50:95 are minimal for all inference pipelines based on *coleo_lepido.pt*. This is because here we are using the standard IoU-based true positive determination, as opposed to our custom IoSA. The PR curve on which the map@50 values were computed are available in the plot below (only for the pure detectors, without post-detection filtering).



We only compared pure detectors in this PR curve because MaP is actually not the best metric for assessing the performance of a detector coupled with a post-detection filter. Indeed, MaP is based on the confidence threshold of the detector, and ignores any confidence associated with false positive filtering. It is thus not portraying the actual levels of confidence of the whole pipeline. To accurately measure the filtering effect of post detection classifiers, we can simply compare the number of false positives for each pipeline. The following graph shows the number of false positives for each pipeline's predictions on our test set, where any prediction with confidence level smaller than 0.1 was discarded.



This shows the effectiveness of the post-detection classifier in reducing the number of false positives. This number was reduced by a factor of 19 for the high-recall low-precision *coleo_lepido.pt* and by a factor of 7 for our model *final_23.pt*. The

effect of post-detection filtering is smaller on our model simply because *final_23.pt* achieves high precision even for low confidence levels (see Table 4.1).

All of those results confirm that we have effectively set the new state-of-the-art in the field of insect detection in entomological boxes.

4.3 Active learning performances

The effectiveness of active learning subsampling techniques is measured by the rate at which the detector's performance increases when retrained on the newly selected instances. To measure this increase rate, we have to simulate a real-life active-learning scenario: we start with a small subset of our labeled dataset, and treat the rest of the dataset as the unlabeled pool **U**. Then, we incrementally select samples **k** from **U**, retraining our model on our updated **S**. For comparison and to avoid leakage, performance must be measured according to a fixed exterior test set **Te**.

4.3.1 Active learning simulation

Here we explain the exact configuration of our active learning simulation. We will refer to our actual entire training set as **Tr**. If we decide to simulate n subsampling iterations, we first need to select an initial training set $\mathbf{S}_1$ of size $Tr \div n$.

This initial set is common to all simulated strategies, because subsampling strategies are only used at iteration 2. The first sample $\mathbf{S}_1$ is selected using the uniform partition strategy from Subsection 3.2.2. This is to simulate our actual entire labeled dataset **Tr** which was sampled with this method. The uniform sampling strategy has the advantage of introducing some diversity in the initial dataset, without requiring the use of a detector. We can then also define $\mathbf{U}_1 = \mathbf{Tr} - \mathbf{S}_1$, and train our first model m_1 on $\mathbf{S}_1$.

Then, for every subsampling strategy, we iteratively sample $\mathbf{k}_i$ from $\mathbf{U}_i$ according to the strategy, we merge $\mathbf{k}_i$ with $\mathbf{S}_i$ using *validation replacement* (see Subsection 3.1.2), and we retrain m_{i+1} with *gradual fine-tuning* (see Subsection 3.3.2). Finally, for each iteration we measure the performances on our test set **Te**.

For our simulation we chose $n = 10$, so approximately $k = 14$ images are added at each step. We choose map@50 and map@50:95 as performance metrics, in order

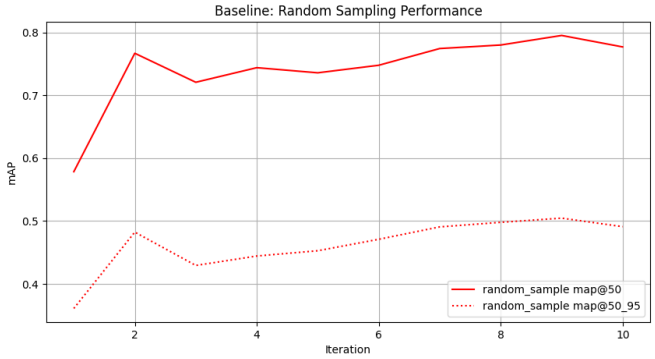
to evaluate performance across all confidence thresholds. In the graphs of the next Subsection 4.3.2, MaPs are computed per image in **Te** and then averaged. This is not standard but still appropriate for comparison purposes.

4.3.2 Comparison of subsampling strategies

Random subsampling

Random subsampling is used as a baseline to prove the effectiveness of AL techniques, as well as to evaluate the sole effect of incremental retraining without subsampling strategies.

The following graph shows the evolution of MaPs with every subsampling iteration, from step 1 to 10:

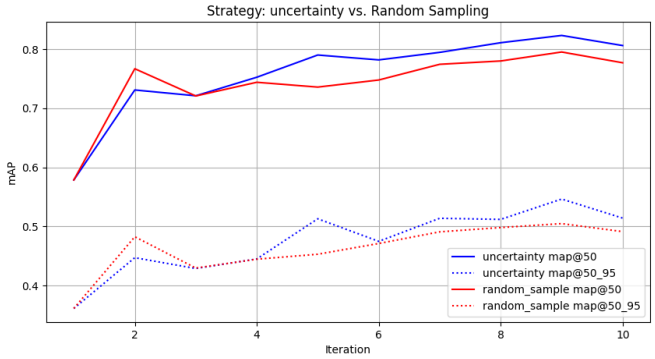


It is apparent than the *gradual fine-tuning* retraining strategy is successful in increasing the model's performance. Full training of *final_23.pt* on the entire **Tr** actually yielded an average map@50 per image of 0.785[2]. Here, the random sample strategy gets 0.777 at iteration 10, and even 0.795 at the previous iteration. This shows that *gradual fine-tuning* is an effective retraining strategy, yielding similar results to full retraining with fewer fine-tuning steps and epochs. In fact, we will see later that, when paired with appropriate AL subsampling strategies, it even performs better than full retraining.

[2]The value is different from the one in Table 4.2 (0.799) because here we calculate average MaP per image, and not global MaP on all predictions.

Uncertainty based subsampling

The following graph shows the performance gains brought by the uncertainty-based strategy (blue), compared against the baseline (red):



Note that any performance increase before step 3 is not due to the subsampling strategy. Indeed *validation replacement* makes it so that the sample $\mathbf{k}_1$ is actually put into the validation set of $\mathbf{S}_2$, hence not directly contributing to the model's training. Therefore, results of the subsampling strategy will only be visible from step 3 and on.

Hence, it is apparent that *uncertainty based subsampling* performs favorably against the baseline. Map@50 at step 10 is at 0.806, and at step 9 it reaches its maximum of 0.823. Considering the fact that only a few iterations with small additional samples were needed to make that difference, this is significantly better than the baseline maximum of 0.795. Moreover, it also proves to be better than full retraining on **Tr** (map@50 of 0.785).

For comparison purposes we define the average MaP increase rate as:

$$\text{increase_rate} = \frac{(map@50)_8 - (map@50)_3}{6}$$

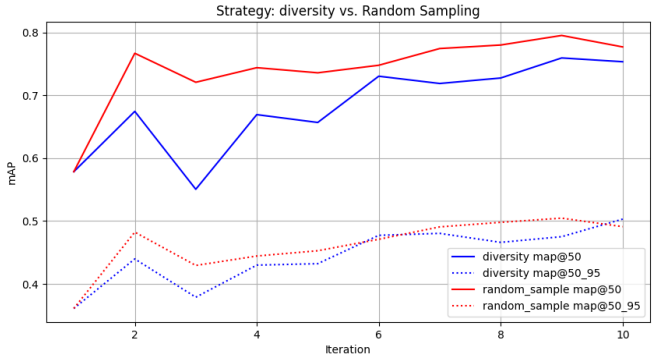
Where subscripts refer to the MaP iteration. We decided to start at iteration 3 and end at 8 to actually measure the effect of the subsampling strategy. Below iteration 3, samples have not entered **S_train**, and above iteration 8 all strategies start to have very similar training sets.

The baseline MaP increase rate is 0.00983, meaning that random subsampling gives on average an map@50 increase of approximately 0.01 per iteration. In

contrast, the MaP increase rate for the uncertainty-based subsampling method is 0.0150, significantly outperforming the baseline by a factor of more than 50%.

Diversity based subsampling

The following graph compares diversity-based subsampling (blue) with the baseline (red):



The diversity subsampling strategy does not increase performances, and in fact it seems to do the opposite. The diversity curve consistently stays below the baseline curve, underperforming against it by a significant margin (except for map@50-95 at iteration 6). Maximum map@50 is obtained at step 9, where diversity subsampling reaches 0.759, which is considerably lower than the baseline maximum score (0.795).

Interestingly, the average MaP increase rate for diversity is at a high 0.0293. This is because performance drastically dropped in iteration 3, and then recovered at a relatively fast rate. One possible explanation is that the strategy introduced too much exploration early on, selecting highly varied images before the model had developed sufficient representational capacity. At such an early stage, the training set **S** may not have been large or consistent enough to allow the model to benefit from diversity. A more incremental exploration approach — gradually introducing diverse samples after establishing a solid initial set — might yield better results.

Lastly, the fast performance increase must be interpreted with care. Gains are naturally easier to achieve when recovering from a low point, as seen after the iteration 3 trough. As such, this recovery may not indicate superior long-term strategy performance.

Uniform partition subsampling

Uniform partition subsampling is another diversity-based strategy which is is specific to our IRSNB dataset and its filenames. Its performances are shown in the graph below.

Similarly to the previous diversity-based strategy, small original datasets **S** at the first few iterations (up to iteration 5) seem to suffer from the introduction of too diverse images, with performances staying significantly below the baseline. However from iteration 6 and on, map@50 grows rapidly, reaching its peak of 0.823 at iteration 8, only slightly below the uncertainty-based strategy's peak. Moreover, due to its early low performances and its quick recovery, it achieves an impressing average MaP increase rate of 0.0235. In the end, it appears to significantly outperform the baseline from iteration 6 and on, making it a viable AL strategy.

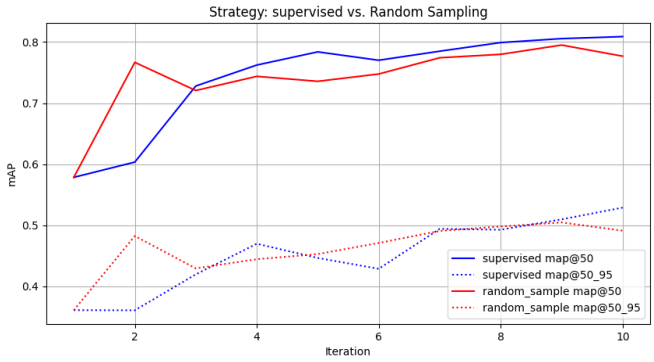
One possible explanation for its success compared to the earlier diversity-based method is the implicit randomness introduced by uniform partitioning. For instance, it is entirely arbitrary that a species starting with the letter "a" should be considered similar to species starting with "b". While the uniform partition subsampling ensures that selected samples will not contain twice the same species, its simple implementation does not enforce morphological diversity in the strictest sense. Therefore, This approximation of diversity, tempered by randomness, may have made the strategy less aggressively explorative, enabling a better model adaptation.

Interestingly, unlike most other strategies, map@50-95 for the uniform partition strategy does not exhibit a clear upward trend alongside map@50, and consistently stays below the baseline. This suggests that while detection performance improves

when evaluated at the standard IoU threshold of 0.5, localization accuracy, reflected in the stricter map@50-95 metric, remains relatively low. As a result, the apparent success of the uniform strategy should be interpreted with care: it may lead to better coarse object detection without necessarily improving localization.

Supervised hybrid subsampling

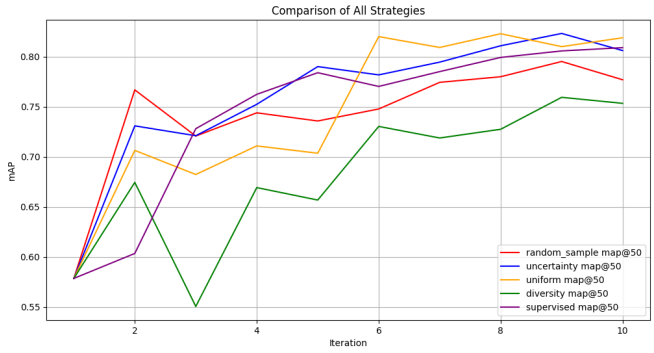
For the supervised hybrid method, we have the following graph:



Similarly to the uncertainty-based strategy, *supervised hybrid strategy* performs favorably against the baseline, reaching its peak at iteration 10 with map@50 0.809, when the baseline peak is only 0.795. Average MaP increase rate is 0.0118, overperforming against the baseline's increase rate (0.00983). This shows our strategy's effectiveness in boosting performance increases.

Comparison of all strategies

To compare all strategies against each other, we plotted all curves on a single graph. Note that map@50:95 is omitted for readability, as its trend closely mirrors that of map@50 for most strategies.



From this graph, it is apparent that informativeness-based strategies yield the best results, both in terms of performance peak and stability (note that supervised hybrid strategy is 75% informativeness-based, see Subsection 3.2.5). The uncertainty-based method got the highest performance peak of all simulations, at an map@50 of 0.823. Moreover, it achieved a high MaP increase rate of 0.0150.

The very objective of active learning is to obtain performances that are similar to those of a larger-scale model, but with significantly less data. In that regard, both uncertainty-based and supervised hybrid strategies are the fastest to reach the average MaP of a full training on **Tr** (0.785). This threshold is reached at iteration 5, effectively matching performances with 50% less data.

As already mentioned, diversity-based strategy was probably too explorative for such small initial datasets. Uniform partition subsampling partly solved this problem with its implicit randomness, but this did not translate to strict localization and map@50:95. Moreover, the uniform strategy is highly specific to our dataset, and not really portable to other contexts.

It would be interesting to measure both diversity-based strategies performances with a larger-scale simulation, and see if their fast MaP increase rates persist over more iterations. However, such a larger-scale simulation could only have been carried out with a larger labeled dataset.

Generally, if our active learning simulation effectively proves the superiority of our informativeness-based methods over the baseline, a more extensive simulation could offer a more robust and representative comparison. For instance, although the uncertainty and supervised hybrid strategies exhibit similar performance trends from iteration 3 onward, a simulation of larger scale would be necessary to draw definitive conclusions about their relative effectiveness.

4.4 Limitations and Future Work

This section is about the various leads which could improve our detection pipelines and active learning strategies.

First of all, as mentioned in Subsection 3.2.5, our model struggles on entomological boxes containing very small coleopterae. An example of a prediction from our new state-of-the-art *final_23.pt*, combined with post-detection filters, on such a box from our test set is shown below. Only a small portion of the box has been cropped, and confidence levels are not shown to ensure visibility.

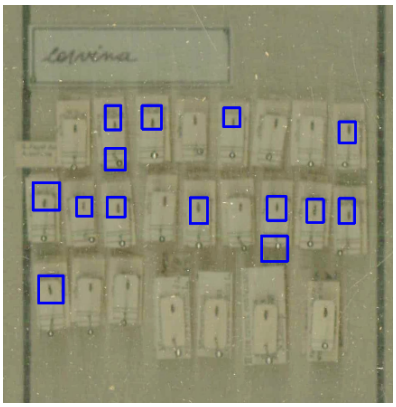


Figure 4.5: Example of bad detection

It is clear that our model struggles on such small insects. On the entire box of the example, it scored an f1-score of only 0.610.

An easy escape from that weakness would be to claim that AL techniques, especially the supervised hybrid strategy, will in the future specifically sample such boxes and our model will get better at detecting them. However, we believe that this behavior resides in the smaller resolution choice we chose for our detector: 640×640. While it enabled us to train locally in a memory-efficient way, and to speed up inference time by a factor of 11, it came at the cost of loosing some fine-grained details in the images. For very small insects of the order *coleoptera*, and especially on poor-quality images such as the example above (observe the numerous reflections and defects), this loss of details resulted in poor performance.

It is important to note that this example represents our worst-case test scenario. For a more typical model prediction with average performance, see Appendix D.

Secondly, our *post-detection classifier* adds substantial inference time overhead. When combined with *final_23.pt*, it increased average inference time by a factor of more than 20 (from 0.469 seconds up to 9.66 seconds). This is due to the expensive resizing of all bounding boxes to be passed into the classifier. This overhead could be mitigated with purely implementation-related optimizations. Another lead could be to make the classifier fully convolutional (remember Subsection 4.2.1), and use a GAP layer to convert each output feature map to a scalar, thereby making the classifier accept input of arbitrary size. This technique would prevent the pipeline from having to perform those costly bounding box resizing operations.

Thirdly, to draw more definitive conclusions about the effectiveness of AL subsampling strategies—especially informativeness-based methods—a more extensive and long-term evaluation is warranted. One approach could be to alternate between subsampling strategies during actual AL cycles and monitor the incremental performance improvements. Over time, this would enable a statistically meaningful assessment of each method's efficacy.

Lastly, a promising direction would be to combine the diversity-based subsampling approach with a more exploitative strategy in a hybrid manner. Such a combination could mitigate the weaknesses previously discussed in Subsection 4.3.2.

Chapter 5

Conclusion

The objective of this master thesis was to pave the way for an automatic processing of digital entomological collections. Specifically, we wished to tackle the first stage of this endeavor: the automatic detection and localization of individual insects in entomological boxes.

Moreover, to align with the environmental motivations of this work, we focused on memory and data-efficient approaches. The goal was to achieve satisfying performance with a highly constrained hardware and dataset.

To do this, we first sought to adapt existing detectors by using post-detection filters, including a binary classifier pretrained on EfficientNetB0. This allowed us to drastically increase the average f1-score of the previous state-of-the-art detector, from 0.765 up to 0.876.

Next, with a small labeled dataset of 139 images and a constrained RAM of 8 GBytes, we used dynamic multi-phase fine-tuning, a technique combining a gradual layer unfreezing with a dynamic adaptation of batch size and learning rate during training. This allowed us to set the new state-of-the-art in insect detection, reaching an map@50 of 0.799 and reducing inference time of previous detectors by a factor of 11.

However, while our insect detector outperforms pre-existing ones, it is still not perfectly accurate and it would benefit from enhancements. Therefore, we defined and implemented active learning strategies to pave the way for future dataset enlargements and model retraining.

We studied and defined merging, retraining and subsampling strategies which all aimed to maximally increase performance gains. Subsampling strategies in-

cluded a standard uncertainty-based strategy, a novel diversity-based strategy which used distances between feature maps of average detections, and a supervised hybrid strategy which leveraged the labeled images of the validation set to sample problematic unlabeled instances. Then, using an active learning simulation, we concluded that the uncertainty-based and the supervised hybrid strategies were the best-performing ones: they reached the full dataset training performance with 50% less data.

Thus, we conclude that post-detection classifiers can increase the precision of high-recall detectors. Moreover, a dynamic multi-phase fine-tuning can enable the training of YOLOv8 on very small datasets. Lastly, active learning techniques based on informativeness heuristics allow to reach good performances with significantly less data.

However, to achieve this pursued goal of a complete and automatic processing of digital entomological collections, additional work is needed. For instance, the detector could be extended to detect handwritten labels. Those could be read by an OCR, allowing the creation of a large-scale classification database for the training of an accurate insect species classifier. By automatizing the species identification process and extracting precise and invaluable statistics on entomological collections, such works would be of great help in the global study of insects.

All in all, this thesis brings forward foundational tools to such an ambitious and promising endeavor, and does so using techniques which contribute to crucial advances in sustainable AI.

Chapter 6

Bibliography

[1] Milagros Miceli Adrienne Williams and Timnit Gebru. *The Exploited Labor Behind Artificial Intelligence*. 2022. URL: https://www.noemamag.com/the-exploited-labor-behind-artificial-intelligence/.

[2] Nightfall AI. *Catastrophic Forgetting: The Essential Guide*. 2025. URL: https://www.nightfall.ai/ai-security-101/catastrophic-forgetting#.

[3] Apple. *M1 Overview*. 2021. URL: https://www.apple.com/ua/business/mac/pdf/Apple-at-Work-M1-Overview.pdf.

[4] Oded Berger-Tal et al. "The exploration-exploitation dilemma: a multidisciplinary framework". en. In: *PLoS One* 9.4 (Apr. 2014), e95693.

[5] Eugène Berta et al. *Rethinking Early Stopping: Refine, Then Calibrate*. 2025. arXiv: 2501.19195 [cs.LG]. URL: https://arxiv.org/abs/2501.19195.

[6] Gauthier Black. "Analyse de boites entomologiques : détection et classification d'insectes". MA thesis. UCLouvain, 2024.

[7] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. *YOLOv4: Optimal Speed and Accuracy of Object Detection*. 2020. arXiv: 2004.10934 [cs.CV]. URL: https://arxiv.org/abs/2004.10934.

[8] Gaudenz Boesch. *YOLOv8: A Complete Guide*. 2024. URL: https://viso.ai/deep-learning/yolov8-guide/.

[9] Mateusz Buda, Atsuto Maki, and Maciej A. Mazurowski. "A systematic study of the class imbalance problem in convolutional neural networks". In: *Neural Networks* 106 (Oct. 2018), pp. 249–259. ISSN: 0893-6080. DOI: 10.1016/j.neunet.2018.07.011. URL: http://dx.doi.org/10.1016/j.neunet.2018.07.011.

[10] Bowen Cheng et al. *Decoupled Classification Refinement: Hard False Positive Suppression for Object Detection*. 2020. arXiv: 1810.04002 [cs.CV]. URL: https://arxiv.org/abs/1810.04002.

[11] COCO. *COCO Explore*. 2025. URL: https://cocodataset.org/#explore.

[12] 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)*. Vol. 1. 2005, 886–893 vol. 1. DOI: 10.1109/CVPR.2005.177.

[13] Ana Davila, Jacinto Colan, and Yasuhisa Hasegawa. "Comparison of fine-tuning strategies for transfer learning in medical image classification". In: *Image and Vision Computing* 146 (June 2024), p. 105012. ISSN: 0262-8856. DOI: 10.1016/j.imavis.2024.105012. URL: http://dx.doi.org/10.1016/j.imavis.2024.105012.

[14] Nicolas Delcommune. "Automatisation de la classification des spécimens d'insectes provenant de boîtes entomologiques de collections muséales". MA thesis. UCLouvain, 2025.

[15] Pedro Felzenszwalb, David Mcallester, and Deva Ramanan. "A Discriminatively Trained, Multiscale, Deformable Part Model". In: vol. 8: June 2008. DOI: 10.1109/CVPR.2008.4587597.

[16] Emanuele Francazi, Marco Baity-Jesi, and Aurelien Lucchi. *A Theoretical Analysis of the Learning Dynamics under Class Imbalance*. 2024. arXiv: 2207.00391 [stat.ML]. URL: https://arxiv.org/abs/2207.00391.

[17] Dibet Garcia et al. "Ten Years of Active Learning Techniques and Object Detection: A Systematic Review". In: *Applied Sciences MDPI* (2023). DOI: 10.3390/app131910667.

[18] Ross Girshick et al. "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation". In: *2014 IEEE Conference on Computer Vision and Pattern Recognition*. 2014, pp. 580–587. DOI: 10.1109/CVPR.2014.81.

[19] Rabia Gondur. "Most Distant Points; Understanding the Max-Min algorithm". en. In: *Medium* (2024).

[20] Xiaoqiang Gui, Xudong Lu, and Guoxian Yu. "Cost-effective Batch-mode Multi-label Active Learning". In: *Neurocomputing* 463 (2021), pp. 355–367. ISSN: 0925-2312. DOI: https://doi.org/10.1016/j.neucom.2021.08.063. URL: https://www.sciencedirect.com/science/article/pii/S0925231221012534.

[21] Cenaero HPC. *Lucia System Overview*. 2025.

[22] Jaafar Jaafari et al. “Towards more efficient CNN-based surgical tools classification using transfer learning”. In: *Journal of Big Data* 8.1 (Aug. 2021), p. 115. ISSN: 2196-1115. DOI: 10.1186/s40537-021-00509-8. URL: https://doi.org/10.1186/s40537-021-00509-8.

[23] Ibrahim Jubran, Alaa Maalouf, and Dan Feldman. *Introduction to Coresets: Accurate Coresets*. 2019. arXiv: 1910.08707 [cs.LG]. URL: https://arxiv.org/abs/1910.08707.

[24] evanfields on Julia forum. *GPU memory usage increasing on each epoch (Flux)*. 2024. URL: https://discourse.julialang.org/t/gpu-memory-usage-increasing-on-each-epoch-flux/112942.

[25] Shachar Kaufman, Saharon Rosset, and Claudia Perlich. “Leakage in Data Mining: Formulation, Detection, and Avoidance”. In: vol. 6. Jan. 2011, pp. 556–563. DOI: 10.1145/2020408.2020496.

[26] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2017. arXiv: 1412.6980 [cs.LG]. URL: https://arxiv.org/abs/1412.6980.

[27] Tautvydas Kvietkauskas and Pavel Stefanovič. “Influence of Training Parameters on Real-Time Similar Object Detection Using YOLOv5s”. In: *Applied Sciences* 13.6 (2023). ISSN: 2076-3417. DOI: 10.3390/app13063761. URL: https://www.mdpi.com/2076-3417/13/6/3761.

[28] Shuaipeng Li et al. *Surge Phenomenon in Optimal Learning Rate and Batch Size Scaling*. 2024. arXiv: 2405.14578 [cs.LG]. URL: https://arxiv.org/abs/2405.14578.

[29] Dominic Masters and Carlo Luschi. *Revisiting Small Batch Training for Deep Neural Networks*. 2018. arXiv: 1804.07612 [cs.LG]. URL: https://arxiv.org/abs/1804.07612.

[30] Guglielmo Montone, J. Kevin O’Regan, and Alexander V. Terekhov. *Gradual Tuning: a better way of Fine Tuning the parameters of a Deep Neural Network*. 2017. arXiv: 1711.10177 [cs.AI]. URL: https://arxiv.org/abs/1711.10177.

[31] Jowi Morales. “Analysts say average laptop RAM quota will reach 11.8GB in 2024, up 12% year on year”. In: *Tom’s hardware* (2024).

[32] Ochuko Orikpete, Akeeb Fawole, and Daniel Ewim. “Impact of Data Centers on Climate Change: A Review of Energy Efficient Strategies”. In: *The Journal of Engineering and Exact Sciences* 9 (Aug. 2023), pp. 1–15. DOI: 10.18540/jcecvl9iss6pp16397-01e.

[33] Yan Pan and Yuanzhi Li. *Toward Understanding Why Adam Converges Faster Than SGD for Transformers*. 2023. arXiv: 2306.00204 [cs.LG]. URL: https://arxiv.org/abs/2306.00204.

[34] David Patterson et al. *Carbon Emissions and Large Neural Network Training*. 2021. arXiv: 2104.10350 [cs.LG]. URL: https://arxiv.org/abs/2104.10350.

[35] Pramuditha Perera and Vishal M. Patel. "Learning Deep Features for One-Class Classification". In: *IEEE Transactions on Image Processing* 28.11 (Nov. 2019), pp. 5450–5463. ISSN: 1941-0042. DOI: 10.1109/tip.2019.2917862. URL: http://dx.doi.org/10.1109/TIP.2019.2917862.

[36] Python. *Data Structures*. 2025. URL: https://docs.python.org/3/tutorial/datastructures.html.

[37] Python. *subprocess — Subprocess management*. 2025. URL: https://docs.python.org/3/library/subprocess.html.

[38] Yash_Rathi on Pytorch forum. *Memory usage increasing with number of steps while training*. 2022. URL: https://discuss.pytorch.org/t/memory-usage-increasing-with-number-of-steps-while-training/143694.

[39] Joseph Redmon et al. *You Only Look Once: Unified, Real-Time Object Detection*. 2016. arXiv: 1506.02640 [cs.CV]. URL: https://arxiv.org/abs/1506.02640.

[40] Shaoqing Ren et al. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. 2016. arXiv: 1506.01497 [cs.CV]. URL: https://arxiv.org/abs/1506.01497.

[41] Lukas Ruff et al. "Deep One-Class Classification". In: *Proceedings of the 35th International Conference on Machine Learning*. Ed. by Jennifer Dy and Andreas Krause. Vol. 80. Proceedings of Machine Learning Research. PMLR, Oct. 2018, pp. 4393–4402. URL: https://proceedings.mlr.press/v80/ruff18a.html.

[42] Noha Sarhan, Mikko Lauri, and Simone Frintrop. "Multi-phase Fine-Tuning: A New Fine-Tuning Approach for Sign Language Recognition". In: *KI - Künstliche Intelligenz* 36.1 (Mar. 2022), pp. 91–98. ISSN: 1610-1987. DOI: 10.1007/s13218-021-00746-2. URL: https://doi.org/10.1007/s13218-021-00746-2.

[43] Saleh Shahinfar, Paul Meek, and Greg Falzon. *How many images do I need? Understanding how sample size per class affects deep learning model performance metrics for balanced designs in autonomous wildlife monitoring*. 2020. arXiv: 2010.08186 [cs.CV]. URL: https://arxiv.org/abs/2010.08186.

[44] Jisu Song et al. “Determination of Optimal Dataset Characteristics for Improving YOLO Performance in Agricultural Object Detection”. In: *Agriculture* 15.7 (2025). ISSN: 2077-0472. DOI: 10.3390/agriculture15070731. URL: https://www.mdpi.com/2077-0472/15/7/731.

[45] Princeton University Stanford University. *Imagenet*. 2021. URL: https://www.image-net.org/.

[46] Chen Sun et al. *Revisiting Unreasonable Effectiveness of Data in Deep Learning Era*. 2017. arXiv: 1707.02968 [cs.CV]. URL: https://arxiv.org/abs/1707.02968.

[47] Yee Liang Thian, Dian Wen Ng, and James Thomas Hallinan. “Effect of training data volume on performance of convolutional neural network pneumothorax classifiers”. en. In: *J. Digit. Imaging* 35.4 (Aug. 2022), pp. 881–892.

[48] Neil C. Thompson et al. *The Computational Limits of Deep Learning*. 2022. arXiv: 2007.05558 [cs.LG]. URL: https://arxiv.org/abs/2007.05558.

[49] Carlos Toxtli, Siddharth Suri, and Saiph Savage. “Quantifying the Invisible Labor in Crowd Work”. In: *Proceedings of the ACM on Human-Computer Interaction* 5.CSCW2 (Oct. 2021), pp. 1–26. ISSN: 2573-0142. DOI: 10.1145/3476060. URL: http://dx.doi.org/10.1145/3476060.

[50] Ultralytics. *COCO Dataset*. 2025. URL: https://docs.ultralytics.com/datasets/detect/coco/.

[51] Ultralytics. *Data Augmentation using Ultralytics YOLO*. 2024. URL: https://docs.ultralytics.com/guides/yolo-data-augmentation/#flip-up-down-flipud.

[52] Ultralytics. *Explore Ultralytics YOLOv8*. 2025.

[53] glenn-jocher github user from ultralytics. *PRECISION-RECALL CURVE*. 2020. URL: https://github.com/ultralytics/yolov3/issues/898.

[54] Admon W. *How Much Do Data Annotation Services Cost? The Complete Guide 2025*. 2025. URL: https://www.basic.ai/blog-post/how-much-do-data-annotation-services-cost-complete-guide-2025.

[55] Wikipedia. *ImageNet*. 2025. URL: https://en.wikipedia.org/wiki/ImageNet.

[56] Eddie Yan, Liang Luo, and Luis Ceze. “Characterizing and Taming Resolution in Convolutional Neural Networks”. In: *2021 IEEE International Symposium on Workload Characterization (IISWC)*. 2021, pp. 189–200. DOI: 10.1109/IISWC53511.2021.00027.

[57] Muhammad Yaseen. *What is YOLOv8: An In-Depth Exploration of the Internal Features of the Next-Generation Object Detector*. 2024. arXiv: 2408.15857 [cs.CV]. URL: https://arxiv.org/abs/2408.15857.

[58] Bolei Zhou et al. *Learning Deep Features for Discriminative Localization*. 2015. arXiv: 1512.04150 [cs.CV]. URL: https://arxiv.org/abs/1512.04150.

Appendices

A Code

Code for this project is open-source and available at this link:

Visit Our GitHub Repo

Other appendices are in the following pages, because they are images in full format.

B YOLOv8 architecture

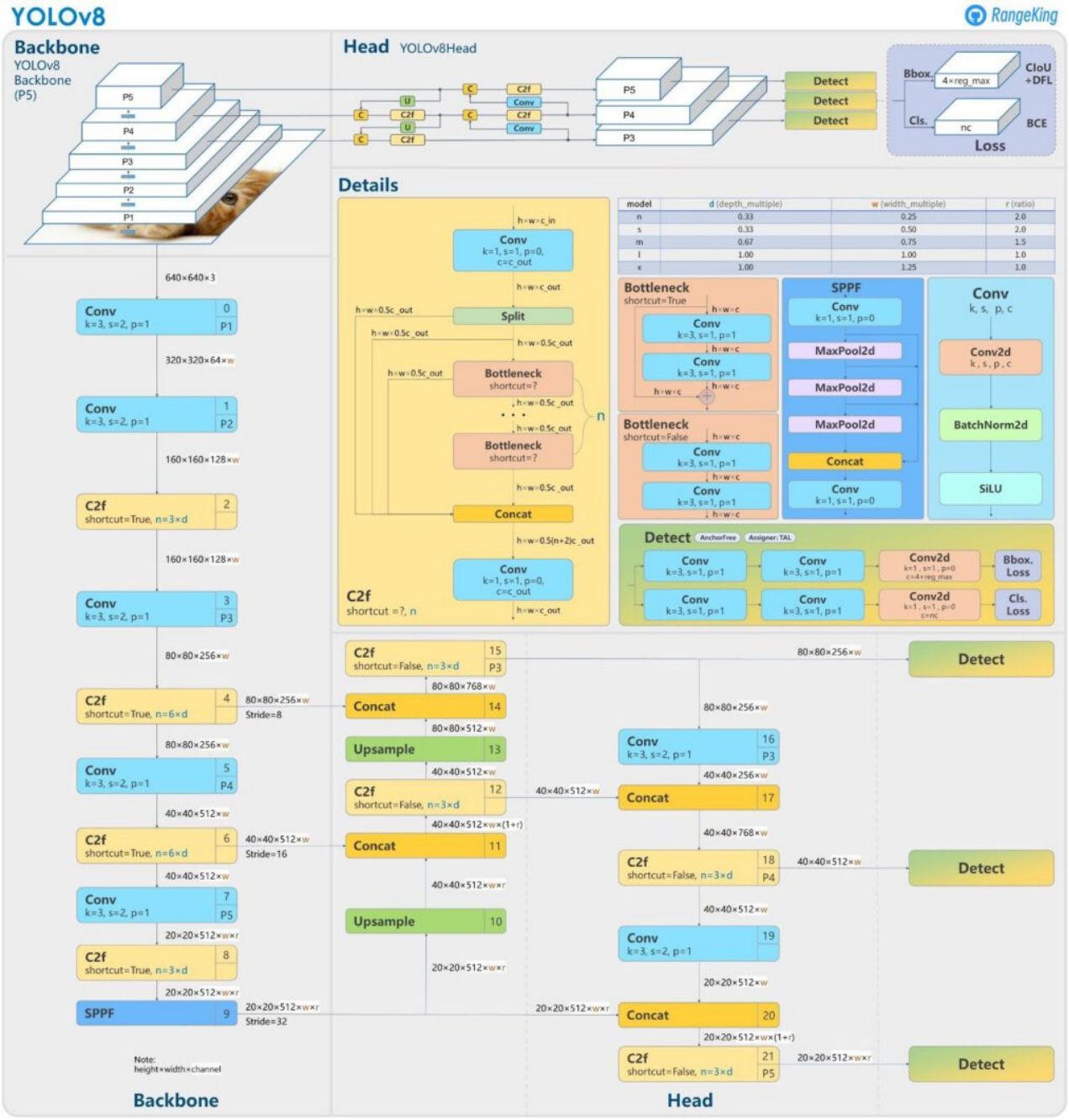
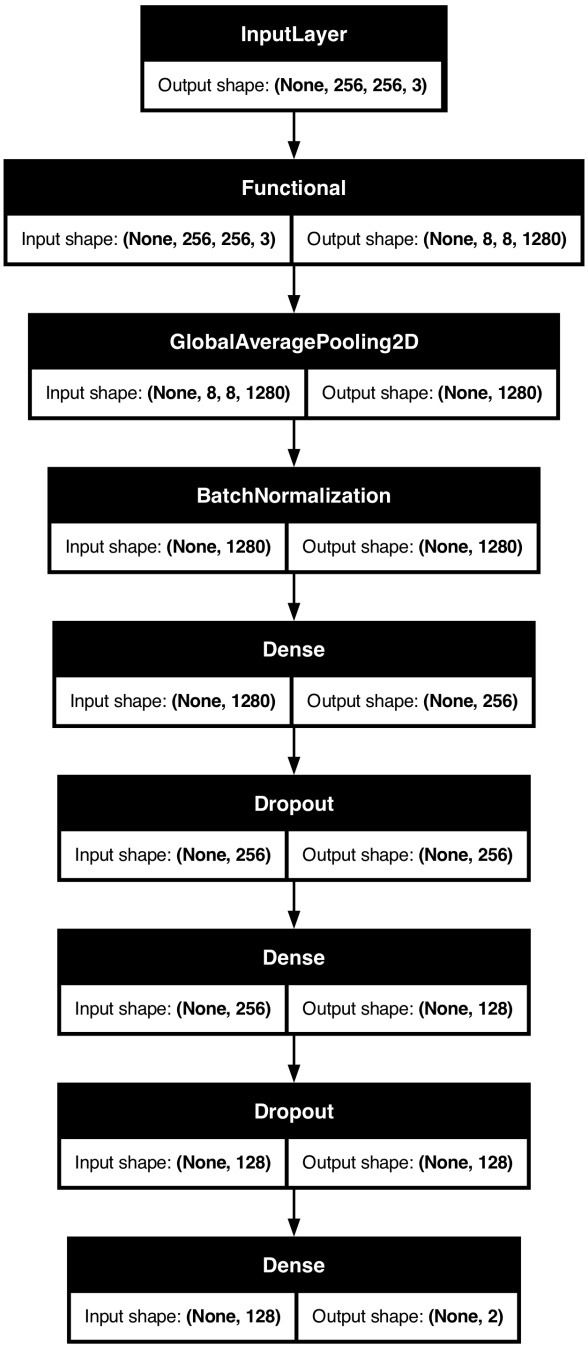


Figure 1: Yolov8 architecture, originally presented by GitHub user RangeKing

C Post-detection classifier architecture



D Typical prediction