Politecnico di Torino

Master's Degree in Computer Engineering

Artificial Intelligence and Data Analytics

A.a. 2024/2025

Graduation Session April 2025

One-Shot Image-Conditioned Object Detection Using Transformers

Supervisors:

Prof.Alessandro Rizzo

Dr.Enrico Civitelli

Candidate:

Milena Yahya

Acknowledgements

To my father and my mother,
who are my rock, my guiding light,
and my greatest blessing.

For your unwavering support,
your unconditional love,
and your faith in me,
which has given me so much strength.

Every achievement of mine is a testament
to the love and devotion you have poured
into my life, and to you I dedicate
every one of my successes.

May I forever be the light
that honors your love.

Я вас люблю.

Abstract

Object detection is a critical task in today's world, with applications spanning many diverse fields, including, but not limited to autonomous vehicles, biometric and facial recognition, and industrial automation. Traditional object detection methods heavily rely on ample amounts of labeled training data and require extensive training time. This makes them resource-intensive and less adaptable to rapidly changing scenarios. In this thesis, we define a one-shot object detection framework that overcomes these limitations. Using state-of-the-art pretrained transformer networks, our approach enables real-time detection of novel objects from a single reference image, bypassing the need for a training phase. This results in a scalable and efficient solution for dynamic, data-scarce environments. Full code at: GitHub

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 Context

Object Detection is an essential computer vision task with an ever-increasing demand across a wide range of industries. Its real-world applications include self-driving vehicles, biometrics and facial recognition, inventory and warehousing, medical image analysis, crowd counting, traffic monitoring, and quality control, among others[1]. As many sectors depend on object detection to drive advancements and improve operational efficiency, the task remains an area of active research. Despite significant progress, challenges remain, particularly in optimizing computational efficiency and minimizing resource costs. Researchers continue to explore innovative solutions aimed at enhancing the performance and scalability of object detection systems.

In widely-adopted state-of-the-art object detection models such as Faster R-CNN[2], YOLO[3], and DETR[4], models learn to recognize objects using training datasets of roughly a few hundred examples per each object category. However, achieving high accuracy and robust performance often demands thousands of carefully annotated instances per class. This heavy reliance on large, high-quality, carefully annotated datasets is considered a significant bottleneck because such data is difficult and expensive to obtain. It is time-consuming and resource-intensive.

Moreover, for the aforementioned models, introducing a new object class is rarely a seamless process. It typically requires either full retraining on an updated dataset or fine-tuning with new class-specific data. Alternatively, additional classification heads can be appended at the final layers of the network, or incremental learning can be employed. However, these methods often introduce their own challenges, such as catastrophic forgetting or imbalanced class representation [5]. Despite ongoing research, incremental learning remains an evolving field with many unresolved challenges. Thus, adapting existing solutions to accommodate new object classes is a non-trivial task that does not yet have a straightforward solution.

1.2 Goal

In this thesis, we mitigate these constraints by proposing a one-shot image-conditioned object detection framework. Our approach requires only a single example image of an object, provided in real-time, to enable its detection across diverse scenes. By leveraging Google's pre-trained ***Owlv2*** network, we eliminate any need for training, effectively removing the burden of data collection, annotation, as well as extensive computational resources and training time, all while offering seamless generalization over new categories. This framework not only significantly reduces deployment time and resource requirements, but it also introduces a scalable and flexible solution capable of handling dynamic and data-scarce environments effortlessly.

This thesis was conducted in collaboration with **Comau S.p.A.** who generously provided the necessary resources, support, and mentorship to make this research possible. Their commitment to innovation and technological excellence was crucial in facilitating the successful development and execution of this project.

1.3 Thesis Structure

This thesis is organized in several sections, each building upon the previous one to provide a comprehensive overview of the development process, methodological choices, and achieved results of this project. The organization ensures a logical flow, guiding the reader from the fundamental concepts and motivation behind the work, through the technical implementation and experimentation, to the final analysis and conclusions.

- **Literature Review:** In this section, we present the chronological evolution of object detection models, their architectures, strengths, and shortcomings. We examine the circumstances and conditions under which these models achieve optimal performance, and highlight the challenges they face in dynamic and data-scarce environments. Finally, we introduce the state-of-the-art approaches to ***Few-Shot Object Detection***, setting the stage to introduce our own proposed model.

- **Theoretical Foundations:** This section introduces the building blocks upon which we build our model. It explains Vision Transformers and their Self-Attention mechanisms, detailing their roles in modern object detection systems. Additionally, it delves into key concepts and coined terms in the field, such as *image-conditioned object detection, one-shot object detection, zero-shot object detection, and query embeddings.* These concepts and components will all be referenced in the section demonstrating the architecture of our model.

- **Model Overview:** This section provides a comprehensive introduction to the network utilized in our methodology, focusing on its architecture and training methods. We discuss the transition from OWL-ViT to OWLv2, highlighting improvements and adaptations that make the model suitable for one-shot detection tasks. We also present detailed insights into the components and workflows that enable seamless scalability and adaptability.

- **Implementation:** This section outlines the practical aspects of deploying the proposed solution. It includes details on the model configuration, such as libraries, frameworks, and hyperparameters, as well as the hardware setup used for inference. Additionally, we describe the end-to-end pipeline, emphasizing real-time operation, user-defined control, and solution's ability to seamlessly integrate new object categories, ensuring scalability and adaptability across diverse scenarios. Lastly, we present the custom features implemented that enhance model flexibility, all supported by code snippets and practical examples.

- **Experiments and Results:** Once the framework was fully developed, we proceeded to test its performance. We evalaute the performance on benchmark datasets like **COCO** and **MetaGraspNetv2**. Experiments are also performed on subsets of **Logos in The Wild** and **MVTecAD**. We further conducted a series of experiments simulating some of the target real-word use cases, like bin picking and palletizing, in **Comau's Robolab** utilizing **Comau robots** to validate the system under practical conditions and test the robustness of the server. In this section, we detail the experimental setup, including the testing environment, evaluation metrics, and scenarios considered. We report the results obtained.

- **Conclusion:** We summarize the key contributions and findings of this thesis, reflecting on the advancements made through our one-shot image-conditioned object detection framework. We revisit the research objectives and demonstrate how they were successfully addressed.

- **Future Work:** We outline potential future directions for this research, highlighting areas where further improvements and exploration could lead to even more impactful results in the field of object detection.

Chapter 2

Literature Review

2.1 Early Stages of Object Detection

The earliest object detection algorithms, dating back to the 1990s, employed *template matching* and *window sliding* techniques[6]. While foundational, these methods suffered from some limitations. Besides having poor performance on unseen datasets and low accuracy in general, these methods also relied heavily on hand-crafted features, and were generally very computationally expensive. This made them impractical for real-time applications, but laid the base for future advancements in the field.

In the early 2000s, the object detection landscape saw significant progress with the emergence of the *Viola Jones Algorithm*[7] for real-time face detection. This model utilized a cascade of classifiers and leveraged *Haar-like features*[8] for feature extraction and comparison. Although this method focused particularly on face recognition, it marked a critical turning point, and it set the stage for more generalized object detection methods that would follow in the next years.

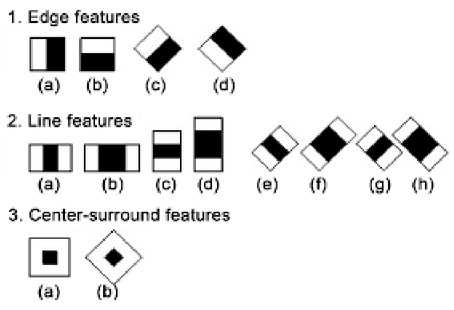


Figure 2.1: Haar features

It had become clear that feature extraction and representation were crucial yet challenging steps in the object detection pipeline, and in 2005, the *HOG (Histogram of Oriented Gradients) feature extractor*[9] revolutionized the object detection scene, which so far had relied on hand-crafted features. The *HOG* provided a more effective way to capture essential object characteristics, such as edges and shapes. As a result, *HOG* became a standard feature extractor, widely adopted by many classification algorithms in object detection tasks.

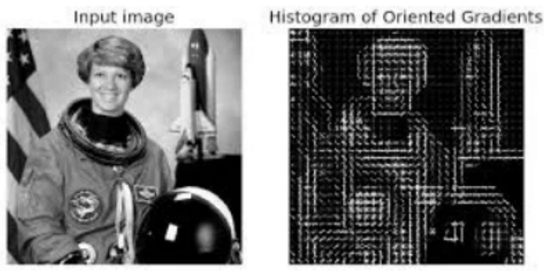


Figure 2.2: HOG feature extractor

In 2008, the *Deformable Parts Model (DPM)*[10] was proposed as an extension of the *HOG*, and it would dominate object detection for years. This model uses the "divide and conquer" strategy, where detecting an object means detecting the different parts that compose it. For example, the problem of detecting a “car” can be broken down to detecting its window, body, and wheels. The training process of *DPM* teaches the model to properly decompose an object into these parts, and then to ensemble the individual detections of different object parts during inference.

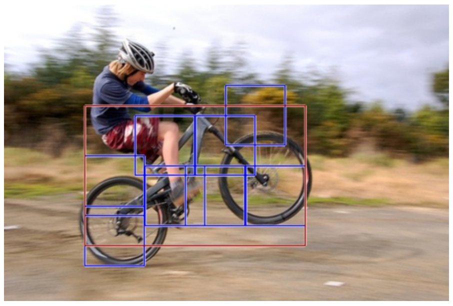


Figure 2.3: DPM concept

After 2010, progress in object detection reached a plateau as research in hand-crafted features became saturated. However, just a few years later, the advent of deep learning revolutionized the field, unlocking new horizons and reshaping object detection into the rapidly evolving and dynamic sphere we know today[11].

2.2 Object Detection in the Deep Learning Era

Deep learning is a subset of machine learning that uses artificial neural networks with multiple layers to understand complex data patterns. Inspired by the human brain, these networks are designed to learn hierarchical data representations and interpret data in a human-like way. Unlike other machine learning paradigms, deep learning models learn their own decision-making, instead of relying on a predefined set of rules. The learning process requires extensive training on large datasets. This ability to autonomously extract meaningful features has made deep learning the driving force behind breakthroughs in computer vision, natural language processing, speech recognition, and other AI-driven fields.

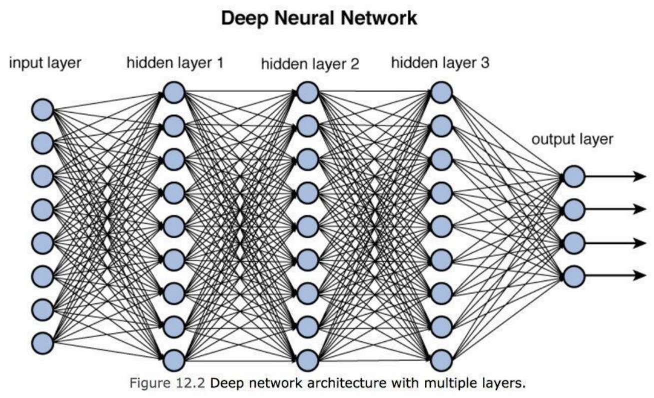


Figure 2.4: A neural network

By 2014, the advancements that came with the rise of deep learning broke the deadlocks of object detection, and redefined the way objects were detected and classified. This marked a significant milestone for object detection, as the challenges that limiting progress in the field were overcome. In this era, object detection models are grouped into two main approaches: "CNN-based models” and “Transformer-based models”[12].

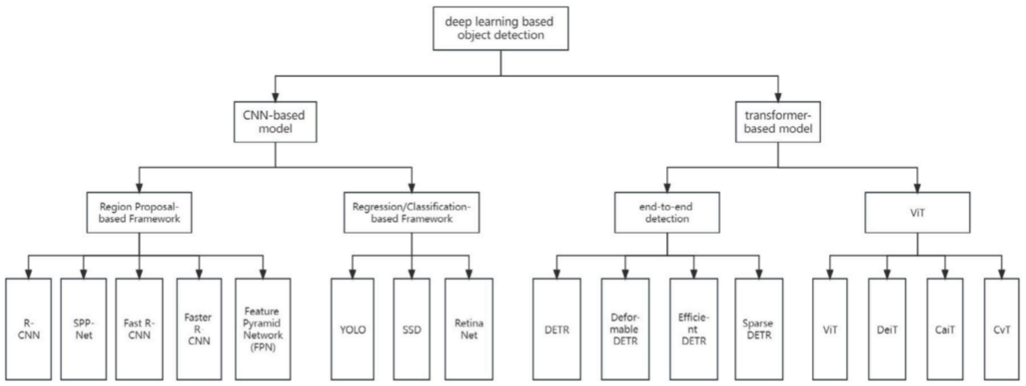


Figure 2.5: Classification diagram of deep learning-based object detection[12]

2.2.1 CNN-based models

A convolutional neural network is a network that "contains three types of layers: convolutional layer, pooling layer, and fully-connected (FC) layer"[12]. The convolutional and hierarchical structure of a CNN renders it capable of learning complex features with higher computational performance. CNN-based object detection methods are divided into two categories: two-stage detectors, and one-stage detectors.

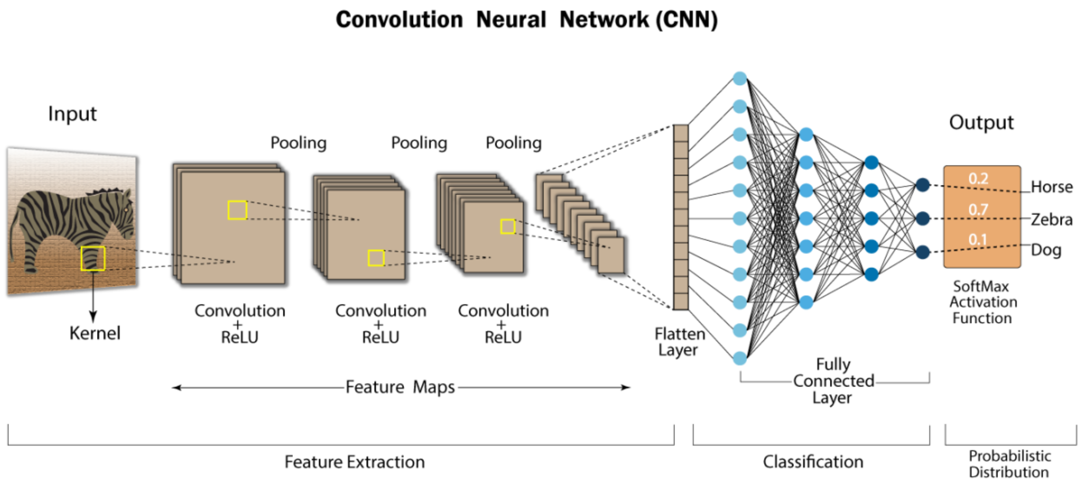


Figure 2.6: An example convolutional neural network

Two-Stage Detectors

These detectors operate in two distinct steps: region proposal and object classification. In the first stage, the model proposes a set of candidate regions, often using a *Region Proposal Network (RPN)*, which identifies areas in the image that are likely to contain objects. In the second stage, these proposed regions are classified into object categories after they are refined, and consequently the predicted bounding boxes are fine-tuned. This separation allows two-stage detectors to focus on high-quality region proposals, leading to accurate object localization and classification. However, this comes at the cost of increased computational complexity and slower inference times, making them less suitable for real-time applications. We briefly touch upon the most famous detectors in this category:

- **R-CNN (Region-based CNN)[13]:** These detectors generate region proposals using **Selective Search**, and extract features from each region using a **CNN**. These proposed regions are classified into object categories using a **Support Vector Machine (SVM) classifier**. This method is accurate but computationally slow, due to the redundant feature extraction of overlapping regions. Other limitations include a fixed input size of 277*277 pixels, low precision of bounding box definitions when using the **SVM classifier**, and requiring "separate training for candidate region generation, feature extraction, and target classification"[12].

- **Fast R-CNN[14]:** These detectors improve upon their predecessors by applying CNN to the entire image once and extracting the feature map, rather than applying a separate CNN to each region.. Then, the regions of interest are extracted and classified using a **RoI pooling** layer, which significantly speeds up the process.

- **Faster R-CNN[2]:** Out-performing all previous detectors, these models employ Region Proposal Networks (RPN) to replace **Selective Search**. The **RPN** is significantly faster and more efficient because it generates region proposals directly from feature maps using learnable convolutional layers. **RPNs** are end-to-end trainable, unlike **Selective Search** which is hand-crafted and heuristic-based. Thus, **Faster R-CNN** allow joint optimization, allowing the network to generate region proposals and to classify objects and refine bounding boxes in a single training process. These detectors produce fewer, higher-quality region proposals.

- **FPN (Feature Pyramid Network[15]):** FPNs tackle another problem: detecting multiscale features or small-scale objects that are generally poorly detected by other models. They propose a pyramid structure with bottom-up and top-down pathways. While the bottom-up link is similarly utilized in other

CNN models, the top-down path adds spatial resolution to the features. Then, a feature fusion model is employed to fuse these the features from both paths. "After the fusion steps on whole features, the Region Proposal Network(RPN) generates proposals for different scales along with the prediction of the presence of objects and bounding box predictions"[12]. These detectors became the standard when dealing with multi-scale applications.

One-Stage Detectors

While R-CNN-based models excel in terms of detection accuracy and computational efficiency, they are unfit for real-time applications as they do not have sufficient speed. Models with simpler structures and faster inference times instead are employed for real-life detection.

- **YOLO (You Only Look Once)[3]:** The **YOLO** algorithm divides each image into an N x N grid, and produces B bounding boxes with their corresponding confidence scores for each grid cell. This allows the model to produce predictions in a single forward pass of the neural network, unlike two-stage detectors, which first generate region proposals and the classify them. This one-stage inference makes **YOLO** suitable for real-time applications, without compromising accuracy significantly. However, the **YOLO** algorithm suffers when detecting adjacent and small-scale objects. Over time, **YOLO** has been significantly improved: **YOLOv2** introduced batch normalization and anchor boxes for better localization, **YOLOv3** enhanced feature extraction with a multi-scale detection approach, it uses multi-label classification to overcome the challenges that **YOLO** faces, and **YOLOv4** optimized training strategies with techniques like self-adversarial training. Subsequent versions, such as **YOLOv5** and **YOLOv7**, have focused on refining model architecture, balancing accuracy and speed, and introducing efficient deployment features.

- **SSD (Single Short Detector)[16]:** This model is another branch of one-stage models parallel to the **YOLO** series. The key innovation of **SSD** lies in the use of multiple feature maps at different scales to detect objects of varying sizes, allowing the network to handle both small and large objects effectively. **SSD** employs default anchor boxes (predefined aspect ratios) at each feature map location, predicting class probabilities and box offsets for each anchor. Compared to **YOLO**, **SSD** offers better accuracy on smaller objects due to its multi-scale feature map design. This combination of efficiency and flexibility makes these detectors a widely used choice in real-time object detection tasks.

2.2.2 Transformer-based Models

In recent years, transformers have become popular in Computer Vision and Natural Language Processing applications. First introduced by Vaswani et al. in 2017 in [17], transformers replace all recurrent and convolutional layers, and rely instead entirely on self-attention mechanisms to process data in parallel. "Transformers utilize the self-attention mechanism to establish global dependencies between different points in sequence"[12]. The general structure of a transformer consists of an **encoder-decoder architecture**, where the encoder maps input sequences into a latent representation, while the decoder generates output sequences based on this representation. Both encoder and decoder stacks are composed of multi-head self-attention layers, which allow the model to focus on different parts of the input simultaneously, and feed-forward layers, which process these attention-weighted representations. Positional encodings are added to the input embeddings to retain some information about the relative or absolute position of the tokens in the sequence. A deeper explanation of the architecture is discussed in Section 3.1.1.

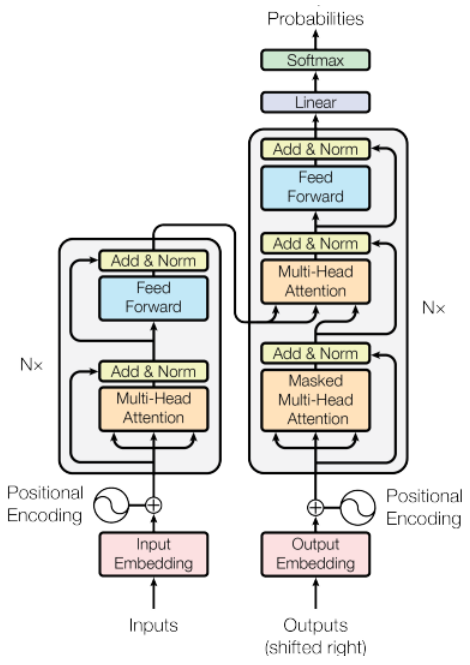


Figure 2.7: Transformer Architecture[17]

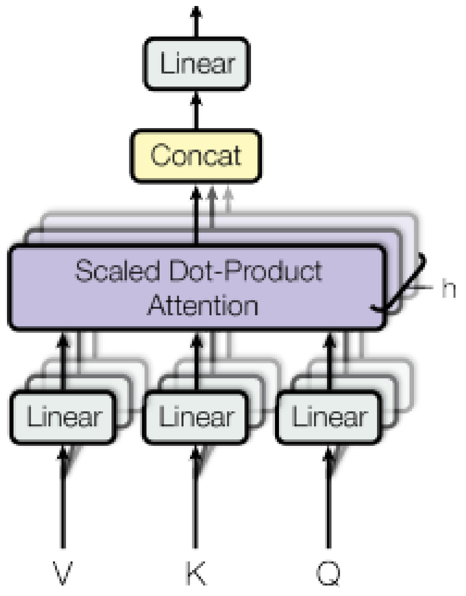


Figure 2.8: Multi-Head Attention[17]

The transformer-based detectors can be categorized as follows: end-to-end detectors like **DETR**, and **ViT**-based detectors.

- **DETR (Detection Transformer)[4]:** This model surpasses the CNN-based detectors with its end-to-end architecture, which eliminates the need for any hand-crafted components, like anchor boxes (e.g **YOLO**), region proposal networks (e.g **Faster R-CNN**), and non-maximum suppression (**NMS**) during the post-processing step. The **DETR** consists of four major components: CNN backbone, transformer encoder, transformer decoder, and final prediction head. The CNN backbone extracts the 2D features from input images, which are then flattened and matched with their positional encodings. The encoder learns the global characteristics of the image from the input sequence, and the decoder uses the output of the encoder, along with a set of learnable object queries to guide the model's attention to different objects. In a final step, the output of the decoder is processed by a feed forward network (**FNN**) to produce the final predictions.

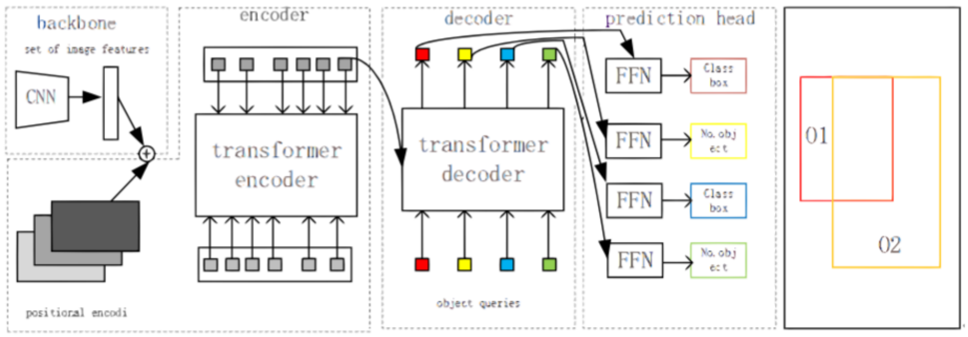


Figure 2.9: DETR workflow[12]

Improved versions of the original **DETR** were proposed to address its slow convergence and computational inefficiencies. The **Deformable DETR** uses sparse attention instead of global self-attention to focus on a small number of key points surrounding each reference point. This imporves detection accuracy, especially for small objects. **Sparse DETR** reduces the number of query points, focusing only on relevant image regions, which speeds up training and reduces memory usage. And finally, **Efficient DETR** optimizes training strategies and architectural design, improving both training speed and inference efficiency without compromising detection performance.

- **ViT-based Object Detectors**: Vision Transformers (ViTs) [18] have been adapted for object detection by leveraging their ability to model long-range dependencies without relying on convolutional backbones. These models divide an image into fixed-size patches, flatten them, and process them as a sequence using transformer layers, with positional embeddings preserving spatial relationships. To perform object detection, ViTs are often coupled with additional components such as region proposal networks (RPNs), feature pyramid networks (FPNs), or detection-specific transformer heads that refine predictions. Architectures like ViTDet[19] integrate these components to enhance multi-scale feature learning, improving their ability to detect objects of varying sizes. By capturing global context and complex relationships across image regions, ViT-based detectors achieve strong performance in both small and large object detection tasks.

2.3 Limitations of Existing Object Detection Methods

The inclusive but not exhaustive list of object detection algorithms we have covered above have significantly advanced the field of computer vision, but they all share certain limitations that hinder their performance in more complex scenarios.

1. The need for extensive amounts of meticulously labeled high-quality data.
2. The drastic declination of performance on novel or unseen classes.
3. High computational cost and inefficiency of CNN-based models when scaling to diverse object categories.
4. Slow convergence and high computational cost of transformer-based models, particularly in low-data scenarios.

These gaps and shortcomings were the driving reason for the emergence of the **Few-Shot Learning** paradigm, which reduces the dependency on large annotated datasets and enables models to adapt to evolving environments and recognize rare or emerging objects more effectively.

2.4 Few-Shot Learning for Object Detection

"Requiring a large number of data samples, many deep learning solutions suffer from data hunger and extensively high computation time and resources. Furthermore, data is often not available due to not only the nature of the problem or privacy concerns, but also the cost of data preparation. Data collection, preprocessing, and labeling are strenuous human tasks"[20]. These challenges have driven the emergence of Few-Shot Learning (FSL) as a critical area of research in machine learning, with a lot of applications in object detection.

Few-Shot Learning aims to enable models to learn effectively from a very limited number of labeled samples per class, typically ranging from just one to a handful.

This is rendered possible by leveraging prior knowledge from previously learned tasks, using **meta-learning**, **transfer-learning**, and **embedding-based** techniques.

In the context of object detection, traditional object detection methods suffer in some real-world scenarios, especially with rare objects, newly emerging categories, and domain-specific tasks where data collection is prohibitively expensive. **FSOD (Few-Shot Object Detection)** presents immense potential because it addresses these limitations by enabling models to detect novel objects using only a few labeled examples, while still maintaining high detection accuracy and generalization capabilities.

Thus, **FSOD** not only reduces the cost and time associated with dataset creation, but also improves model adaptability. This is particularly useful in applications like **autonomous driving**, **medical imaging**, and **surveillance systems**, where encountering previously unseen objects is common, and the ability to quickly adapt to new categories is crucial.

We list the explored methods for Few Shot Learning:

1. **Fine-Tuning/Transfer Learning Methods:** These methods leverage pre-trained models trained on large-scale datasets to serve as a strong foundation for learning. Then, these models are fine-tuned on smaller few-shot datasets, significantly reducing the data and computational requirements, while still achieving high performance and adaptability to new tasks or object classes. [21]

2. **Meta-Learning Methods:** These approaches focus on teaching models how to learn efficiently from limited data by exposing them to few-shot learning scenarios during training. Instead of training on a single task, the model is trained across multiple small tasks, each with only a few labeled examples. This allows the model to become highly adaptive and to quickly fine-tune its parameters or make accurate predictions when presented with new, unseen tasks using minimal data. [21]

3. **Data Augmentation Methods:** Techniques like synthetic feature generation and domain-aware data augmentation are used to increase the diversity and quantity of training samples, compensating the lack of labeled samples, and enabling the model to better adapt to unseen categories.

4. **Model-Oriented Methods:** These methods focus on improving existing model architectures to better handle few-shot scenarios. Innovations often

include improved attention mechanisms, adaptive feature extraction, and dynamic prediction heads tailored for limited data scenarios. Models like **DETR** and **YOLO** have undergone such architectural refinements to help them extract more meaningful representations from sparse data and generalize effectively to novel object categories.

Although this field is still in its early stages of research with no widely-adopted solution, we mention a few prominent models and highlight the gaps identified in these approaches, which lead us to the development of our method.

Hsieh et al. (2019)[22] introduced a novel one-shot object detection framework based on Faster R-CNN, incorporating co-attention and co-excitation mechanisms to enhance feature interactions between the query and target images. In their two-stage detection pipeline, a modified Region Proposal Network (RPN) generates candidate regions from the target image, to which the co-attention mechanism is applied to capture mutual dependencies between the query and target feature maps, guiding the RPN to focus on regions that are most relevant to the query object. Then, the co-excitation module is applied to the feature maps, adaptively re-weighting the channels to emphasize those most pertinent to the query, thus improving the quality of the proposals. These refined regions are then classified and further refined using metric learning with a margin-based ranking loss. This design allows the model to generalize to unseen object categories using a single query example.

Vargas et al. (2024)[23] introduced a novel joint neural network framework for one-shot object recognition and detection. The architecture comprises two interconnected networks of stacked convolutional layers: one for object recognition and the other for object detection. The recognition network processes the query image to extract discriminative features, which are then used to guide the detection network. The detection network generates region proposals from the target image, refining them using the features derived from the recognition network. Both networks are trained jointly using a shared objective, combining classification and bounding box regression tasks into a single loss function, effectively succeeding at the one-shot object detection task.

While the aforementioned methods achieve state-of-the-art performance on benchmark datasets, their architectures remain heavily reliant on convolutional layers and region proposal networks. This reliance introduces inherent limitations, such as sequential computation, which constrains parallelization and increases inference time, and high computational cost, especially in large-scale deployments. Our proposed transformer-based method eliminates these bottlenecks while while achieving performance on par with existing methods and even setting new state-of-the-art benchmarks under specific conditions.

Chapter 3

Theoretical Foundations

This chapter presents the fundamental building blocks, mechanisms, and key terms that form the foundation of our network. These concepts will be referenced throughout the subsequent chapters of this thesis, thus we seek to provide the necessary theoretical context for understanding the proposed model and its components.

3.1 Key Elements

3.1.1 Transformers

Transformers were introduced in *Attention is All You Need*[17] in 2017 as a revolutionary method for processing sequential data. Prior to this, tasks like language modeling and machine translation were typically handled by **Recurrent Neural Networks (RNNs)**[24] and **Long Short-Term Memory (LSTM)**[25] networks. While RNNs and LSTMs were effective for these tasks, they shared a critical limitation: the inability to parallelize computations due to their sequential nature. This bottleneck significantly slowed down training and inference times, especially as datasets grew larger, and rendered these networks difficult to deploy in real-time solutions.

The Transformer model, by contrast, operates on the entire input sequence simultaneously, using a mechanism called *self-attention* to capture dependencies between elements, regardless of their distance in the sequence. This ability to process data in parallel greatly accelerates training and enables the model to scale efficiently. The Transformer architecture has since become the foundation for many state-of-the-art models in natural language processing (NLP) and, more recently, computer vision.

Transformer Architecture

The Transformer is at its very core an Encoder-Decoder structure, both using stacked self-attention and position-wise, fully connected layers.

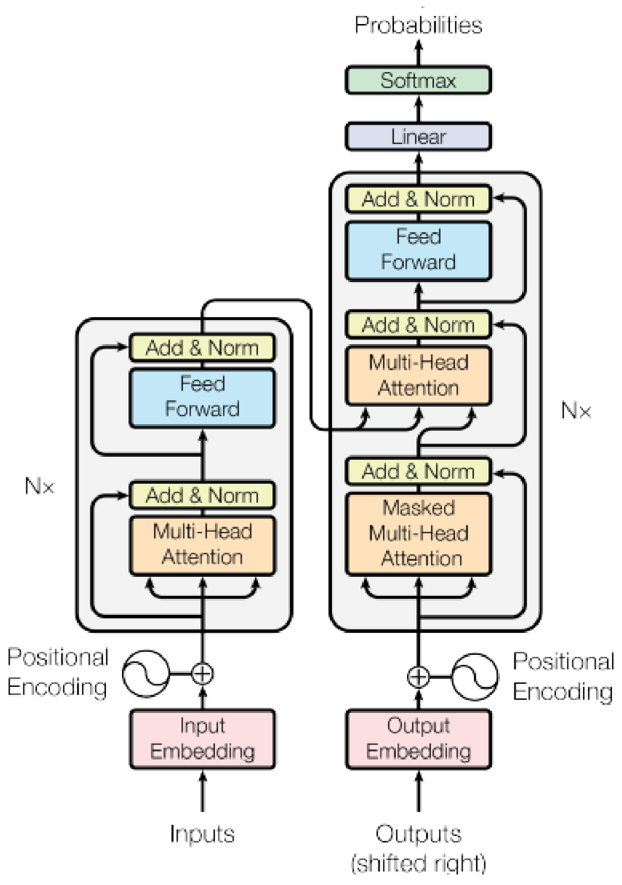


Figure 3.1: Overall Transformer Architecture

- "**Encoder:** The encoder is composed of a stack of N = 6 identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. We employ a residual connection around each of the two sub-layers, followed by layer normalization. [...]
- **Decoder:** The decoder is also composed of a stack of N = 6 identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify

the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with the fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i." [17]

In this architecture, the encoder transforms an input sequence of symbol representations $(x_1, ..., x_n)$ into a sequence of continuous representations $\mathbf{z} = (z_1, ..., z_n)$. Using these continuous representations $\mathbf{z}$, the decoder then generates an output sequence of symbols $(y_1, ..., y_m)$, one symbol at a time. The model operates in an auto-regressive manner, where at each step, it consumes the previously generated symbols as additional input for generating the next symbol in the sequence.

Self-Attention Mechanism

Injected in every layer of the encoder and the decoder, the Self-Attention mechanism maps a query and a set of key-value pairs to an output. It helps us understand the relationships between different elements in a sequence.

- Query **Q**: This matrix represents the focus word for which the context is being determined. By transforming the word representation using the query matrix, the system generates a query vector that will be used to compare against other words in the sentence.
- Key **K**: The key matrix is used to create key vectors for all words in the sentence. These key vectors help the system measure the relevance or similarity between the focus word (using the query vector) and other words in the sentence. A higher similarity score between the query vector and a key vector indicates a stronger relationship between the corresponding words.
- Value **V**: For each element in the sequence, the value matrix generates a value vector which represents the contextual information of the element. Each element is also assigned a weight determined by the similarity scores, and the system finally computes a weighted sum of the value vectors, ensuring that the final contextual representation is influenced more by relevant words.

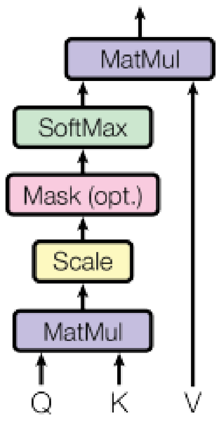


Figure 3.2: Scaled Dot-Product Attention

The attention we use is called "Scaled Dot-Product Attention". The output is computed as follows:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V \tag{3.1}$$

Multi-Head Attention

The transformer architecture uses a multi-head attention mechanism, which consists of multiple self-attention layers running in parallel. Each attention layer has its own set of **Q**, **K**, and **V** matrices, obtained by projecting the original **Q**, **K**, and **V** using separate learned linear projections. The outputs from each layer are concatenated and linearly projected to form the final values. The purpose of multi-head attention is to allow the model to focus on different aspects or relationships between words in the input sequence simultaneously.

$$\text{MultiHead}(Q, K, V) = \text{Concat}(\text{head}_1, \ldots, \text{head}_h)W^O \tag{3.2}$$

$$\text{where head}_i = \text{Attention}(QW_i^Q, KW_i^K, VW_i^V) \tag{3.3}$$

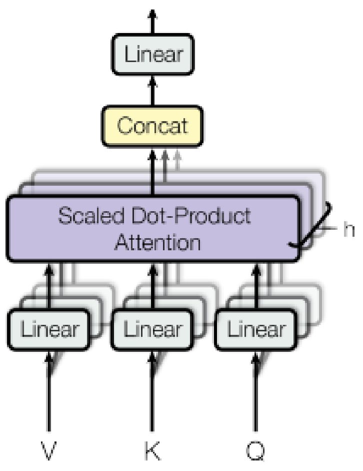


Figure 3.3: Multi-Head Attention

The Transformer uses multi-head attention in three different ways:

1. **Encoder-Decoder Attention:** The queries come from the decoder's previous layer, and the keys and values are from the encoder's output. This allows the decoder to focus on all input positions, similar to traditional sequence-to-sequence models.

2. **Encoder Self-Attention:** Each position in the encoder can attend to all other positions in the encoder. Here, the keys, values, and queries all come from the previous layer of the encoder.

3. **Decoder Self-Attention:** Each position in the decoder attends to all prior positions up to the current one, maintaining the auto-regressive property. This is ensured by masking illegal future connections in the scaled dot-product attention mechanism.

Position-wise Feed-Forward Networks

Each layer of the encoder and the decoder also contains a fully connected feed-forward network, "which is applied to each position separately and identically. This consists of two linear transformations with a ReLU activation in between."[17]

$$\mathrm{FFN}(x) = \max(0, xW_1 + b_1)W_2 + b_2 \tag{3.4}$$

While the linear transformations are the same across different positions, they use different parameters from layer to layer. This step helps the model learn more complex patterns and relationships within the input sequence.

Positional Encoding

"Since our model contains no recurrence and no convolution, in order for the model to make use of the order of the sequence, we must inject some information about the relative or absolute position of the tokens in the sequence. To this end, we add "positional encodings" to the input embeddings at the bottoms of the encoder and decoder stacks. The positional encodings have the same dimension as the embeddings, so that the two can be summed." [17]

Advantages of Self-Attention

Self-attention layers revolutionize sequence processing by directly connecting all positions in a sequence with a constant number of operations, providing unmatched computational efficiency compared to recurrent layers, which require sequential operations proportional to the sequence length. This inherent parallelism allows self-attention to scale more effectively, particularly when the sequence length is shorter than the dimensionality of the representation. Moreover, even for long sequences, the computational load can be further reduced by limiting self-attention to a local neighborhood around each output position without significantly compromising performance.

Unlike convolutional layers, which are inherently more expensive due to their dependency on larger kernels, self-attention achieves comparable or better performance with significantly lower complexity when combined with point-wise feed-forward layers. Optimizations like separable convolutions aim to bridge the gap, but even then, self-attention's efficiency and simplicity remain unmatched.

However, the most important advantage provided by Self-Attention is interpretability. The attention distributions it produces show what parts of a sequence the model focuses on when making decisions. This ability to reveal patterns and relationships makes self-attention not just efficient but also easier to interpret, offering clear advantages over recurrent and convolutional methods.

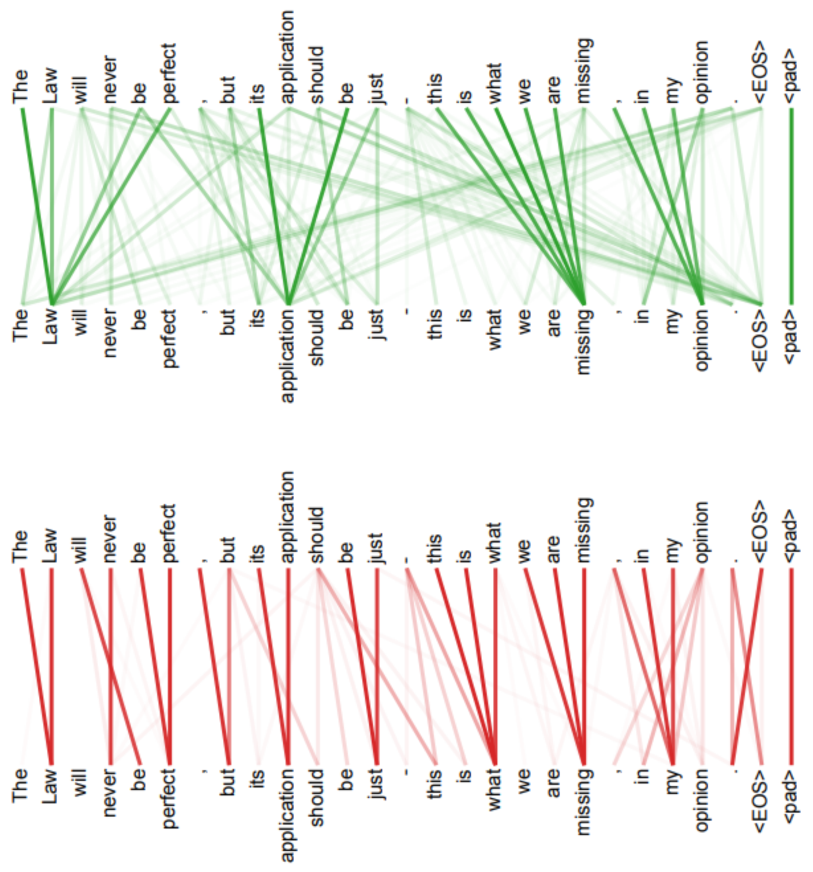


Figure 3.4: "Visualization of how attention heads focus on different relationships between the sequence tokens"

3.1.2 Vision Transformers

Building on the Transformer framework, **Vision Transformers (ViTs)**[18] were developed as a way to apply the Transformer architecture to image data. Unlike traditional convolutional neural networks (CNNs) that operate on pixel grids, ViTs treat an image as a sequence of fixed-size patches. Each patch is linearly embedded into a vector, and the Transformer processes these vectors to learn relationships between different parts of the image. The Vision Transformer has shown impressive performance on computer vision tasks, even outperforming CNNs when trained on large datasets.

ViT Architecture

The intention of this design is to resemble the original Transformer architecture as closely as possible. Thus, similarly to the original Transformer, the ViT consists of alternating layers of multi-headed attention and MLP (Multi-Layer Perceptron) blocks. The MLP blocks replace the FFNs in the original architecture. Layer normalization is applied before every block, and residual connections are added after every block. An overview of the architecture is show in the figure below.

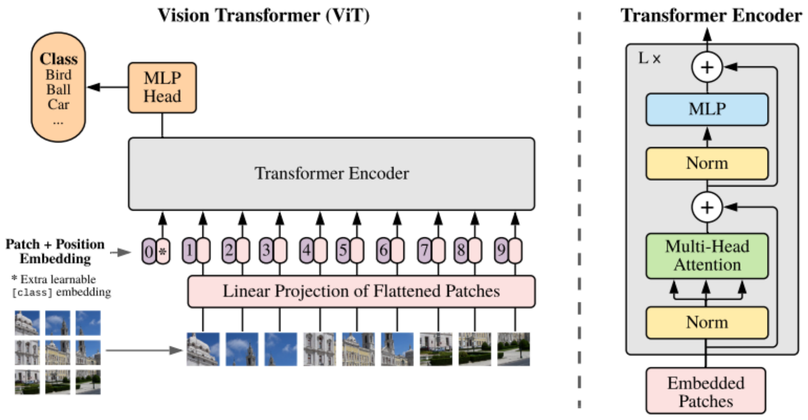


Figure 3.5: Vision Transformer Architecture[18]

We note that the ViT, as the traditional Transformer, relies on the Multi-Headed Self-Attention mechanism. However, we highlight the main differences in architecture:

- The MLP introduced in each block consists of two fully-connected layers with a **GELU (Gaussian Error Linear Unit)** activation function. This activation function adds a smooth non-linearity to the MLP block, and it enhances gradient flow and improves model performance compared to traditional functions like **ReLU (Rectified Linear Unit)**.
- The Layer Normalization applied before every block normalizes the input activations, ensuring stability during the training process by reducing covariate shifts, where the distribution of inputs to each layer changes as the model trains. This helps maintain consistent gradients, allowing for more effective learning across layers.
- Residual Connections, also known as Skip Connections, are added after every block to allow the model to skip some transformations when needed. This facilitates information flow between layers, and prevents vanishing gradients that

can occur in deep architectures. These connections improve both convergence speed and final performance.

Besides the architectural refinements with respect to the traditional Transformer listed above, the way data is preprocessed changes with the ViT. To process 2D images, a series of transformations is applied to the images.

1. First, all the images are resized to a fixed resolution to ensure uniform size across all input images, regardless of their original dimensions.

2. Then, each image is divided into a fixed number of patches, all of the same size, for example 16x16 pixels. These patches do not overlap, and each patch is considered as a "token".

3. Before feeding these tokens to the ViT, each patch is flattened into a 1D vector by concatenating the pixel values. So, for 16x16 pixels patches, if we use 3 color channels (RGB), the resultinf vector with be of size 16x16x3 = 768.

4. Next, a fully connected layer is applied to each of the flattened patches. This is simply a linear transformation which maps the patch vector into an embedding space of fixed size. This step is necessary as Transformers expect embeddings of the data, and not the raw pixel data itself.

5. As Transformers process all patches simultaneously, they are unaware of the sequential order of the patches. Thus, **Positional Encodings** are injected into each patch embedding vector, to provide information about the relative or absolute position of each patch in the image. These positional embeddings can be simply a sine or cosine encoding, or a learnable embedding. Standard learnable 1D position embeddings are used, as no significant performance gains from using more advanced 2D-aware position embeddings were observed.

6. Finally, a sequence of fixed-size patch embeddings with positional encodings is fed to the Vision Transformer.

This is how the ViT adapts the Transformer architecture, originally designed for text, to handle visual data effectively.

3.2 Key Concepts

This section aims to clarify concepts and coined terms that shall be frequently referred to hereafter.

3.2.1 Query Image and Query Embedding

A query image is an image of the object or concept to be detected in a target or test image. It serves as an example of the category we would like our object detector to be able to identify. However, the query image is not passed in its raw format to the object detector. **Query Embeddings** are numerical representations of the query images, obtained using specialized encoders, which output a compact high dimensional vector that captures all the distinctive features of the query image. Query embeddings are designed to retain semantic and structural information, allowing the model to identify similar objects, even under variations in scale, lighting, or orientation. The query embeddings are passed to the object detector, which then looks for the query objects in the test images by comparing the embeddings of the query objects to the test image embeddings.

3.2.2 Image-Conditioned Object Detection

The computer vision task which consists of detecting objects based on a visual example is called Image-Conditioned Object Detection. The visual example is an image of the object to be detected in the test images, and it is used instead of textual queries which are phrases that describe the object to be identified, as well as instead of a fixed predefined set of categories that the detector is pretrained to detect. The example image provided is called the query image. Unlike traditional object detection methods, this approach allows the dynamic identification of objects in test images by comparing their features to those in the query image. The model extracts embeddings from both the test and query images, and then computes similarity measures to locate and classify objects in the scene. This method is particularly useful for scenarios requiring flexibility, such as one-shot or few-shot detection, where labeled data for specific object classes is rare or difficult to obtain. By conditioning on visual queries instead of fixed class definitions, image-conditioned object detection offers a versatile and adaptive solution for detecting novel or previously unseen objects.

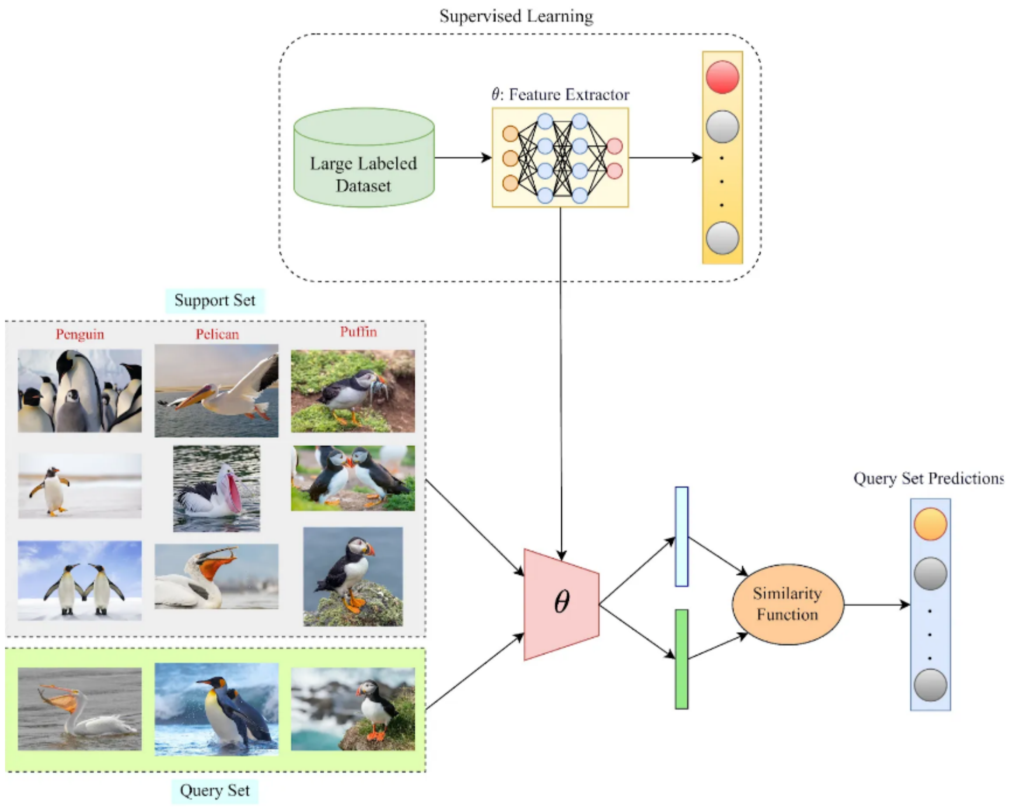


Figure 3.6: Image-Conditioned detection: Features of query images are compared with features of the images from the support (test) set, to produce similarity scores, and eventually class predictions.[20]

3.2.3 Zero-Shot vs Few-Shot Object Detection

Few-Shot Object Detection is an advanced computer vision task whose goal is to detect objects with limited labeled training data available. The model is given few examples of the object to be detected *(often as few as one or five)*. **N-way K-shot detection** refers to the task of detecting "N" novel object categories, with "K" labeled examples available for each of the "N" classes[20]. This is done by using the "K" example images as query images for the detection task.

Zero-Shot Detection, on the other hand, is the task of detecting objects in an image without any labeled examples or prior visual references of the object. This makes zero-shot detection class-agnostic, meaning the model does not rely on specific class labels to identify objects. Instead, it predicts bounding boxes and "objectness" scores, which represent the model's confidence that a bounding box contains an object, regardless of its class.

To accomplish this, zero-shot detection leverages semantic knowledge, such as relationships between objects, attributes, or textual descriptions, rather than relying on visual examples. The model is trained to align object features extracted from images with a pre-existing semantic space. This alignment enables the model to generalize and detect objects from unseen classes by associating visual patterns in the image with their semantic meanings. In simple terms, the model "understands" what an object might be based on its attributes and context, even if it has never seen that specific object before. By using this semantic understanding, the model can identify and localize objects it has no prior examples of, making zero-shot detection a powerful tool for tasks involving unseen categories.

Chapter 4

Model Overview

Our work leverages the pretrained **OWLv2**[26] network, introduced in 2023, which represents the state-of-the-art in one-shot and zero-shot object detection. Our choice to leverage **OWLv2** is driven by its superior performance in generalizing to unseen object categories and its open-vocabulary learning capabilities. The model is designed to detect objects in images based on textual descriptions provided by the user. In our work, we build upon this functionality and extend it by adapting the architecture to detect objects based on reference images rather than textual descriptions. This ability to support both textual and visual inputs demonstrates the flexibility of **OWLv2**, making it an ideal candidate for handling the complex detection tasks relevant to our scope of work.

The **OWLv2** model builds on the foundation of **OWL-ViT**, a vision transformer architecture designed specifically for open-vocabulary learning. By incorporating several optimizations, **OWLv2** enhances both scalability and performance. In the following, we present their respective architectures.

4.1 OWL-ViT

The objective of **OWL-ViT** model is to create a simple and scalable open-vocabulary object detector, i.e, a detector able to recognize objects of novel categories beyond the training vocabulary. The chosen architecture is transformer-focused because of their proven scalability and success in closed-vocabulary detection. Since this model primarily focuses on images, the Vision Transformers described in [18] are utilized. "We present a two-stage recipe:

1. Contrastively pre-train image and text encoders on large-scale image-text data.

2. Add detection heads and fine-tune on medium-sized detection data.

The model can then be queried in different ways to perform open-vocabulary or few-shot detection." [27]

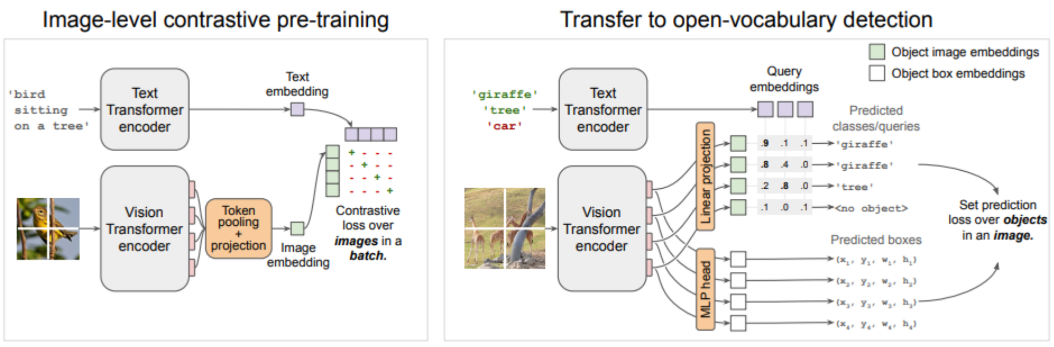


Image-level contrastive pre-training
Transfer to open-vocabulary detection
'bird sitting on a tree'
Text Transformer encoder
Text embedding
Vision Transformer encoder
Token pooling + projection
Image embedding
Contrastive loss over images in a batch.
'giraffe' 'tree' 'car'
Query embeddings
Object image embeddings
Object box embeddings
Predicted classes/queries
Linear projection
MLP head
'giraffe'
'tree'
<no object>
Predicted boxes
Set prediction loss over objects in an image.

Figure 4.1: Overview of the OWL-ViT method: 2-phase recipe[27]

4.1.1 OWL-ViT Architecture

To encode the test images, the model uses a standard Vision Transformer (ViT)[18] as the image encoder and a Transformer-based architecture for encoding text queries, creating a unified framework for object detection through image-text alignment.

4.1.2 Phase I

In the first phase, the image and text encoders are pretrained contrastively. Contrastive training of image-text pairs enables the model to learn consistent visual and language representations from web-derived image and text pairs without the need for explicit human annotations, vastly increasing the available training data, which has led to significant improvements on zero-shot classification benchmarks. The idea of Contrastive Learning is to teach the model to match corresponding image-text pairs, while distinguishing non-matching pairs.

Both encoders are trained from scratch with random initialization with a contrastive loss on the image and text representations.

1. **Image Embedding:** The Vision Transformer (ViT) processes the image by dividing it into patches and generating token representations, where each token corresponds to a specific region of the image. To create a single embedding

for the entire image, Multihead Attention Pooling (MAP) is applied. MAP aggregates all token representations by focusing on the most relevant parts of the image, enabling the model to represent the whole image effectively in a compact vector.

2. **Text Embedding:** The Text Encoder processes the query, and the final end-of-sequence token is used as the text embedding. This embedding captures the overall semantic meaning of the text query in a way that aligns with the visual representations.

Once the image and text embeddings are computed, they are paired and used to train the model with a contrastive loss. This loss function minimizes the distance between embeddings of matching image-text pairs and maximizes the distance between embeddings of non-matching pairs.

At the end of this phase, the image and text encoders are aligned in a shared semantic space, allowing the model to generalize to unseen categories by associating visual features with textual descriptions, making it highly effective for tasks such as zero-shot and few-shot object detection.

4.1.3 Phase II

In the second phase, to adapt the image encoder for object detection, the token pooling and final projection layers are removed. Unlike the pre-training stage, where the output representations of different tokens (representing different parts of the image) are combined into a single image embedding, the Vision Transformer processes the image without aggregating token outputs.

Since each output token from the ViT represents a distinct region of the image, these tokens are linearly projected into individual object embeddings. This allows the model to handle multiple objects within a single image by leveraging these per-token projections. The self-attention mechanism in the ViT ensures that each token not only encodes local information about its specific region but also incorporates global context by attending to other regions in the image.

The query embeddings produced by the text encoder are then used in conjunction with the ViT token outputs to predict class probability scores for the objects. In particular, the query embeddings are compared to the ViT token outputs using a similarity function to associate each region with a class label.

Bounding boxes are generated by passing the ViT output tokens through a small Multi-Layer Perceptron (MLP). The MLP transforms the spatial and contextual features encoded in the tokens into precise bounding box coordinates, localizing the detected objects in the image.

By preserving the token-level granularity of the ViT outputs and aligning them with query embeddings, the model can handle complex scenarios involving multiple objects, overlapping regions, and unseen categories, making it a powerful tool for object detection tasks.

4.2 Transition to OWLv2

The **OWL-ViT** model excels in image classification by leveraging large-scale contrastive learning on image-text pairs, which are abundantly available on the web as weakly supervised data[26]. However, its performance diminishes when applied to object detection tasks. This limitation arises because object detection requires precise localization, a challenge not directly addressed by contrastive learning. Unlike image-text pairs, which are plentiful for classification, naturally occurring data suitable for object localization is scarce and not publicly available, restricting detection performance and scalability.

Additionally, **OWL-ViT**'s dependence on pre-trained representations limits its ability to adapt to the more complex demands of object detection, such as recognizing multiple objects within a single image and managing overlapping regions. Localization tasks demand a precise alignment between visual features and spatial regions, which cannot be effectively learned through contrastive training alone.

To scale detection training, inspiration is drawn from image-level methods, where the principle has been to leverage weak supervision in the largest possible amount. Specifically, a *self-training* approach is adopted: an existing detector generates pseudo-box annotations for image-text pairs, which are then used as training data. This strategy mitigates the scarcity of labeled detection data, enabling more robust and scalable object detection training.

4.3 OWLv2

The **OWLv2** method consists in a simple self-training approach with three steps:

1. Use an existing open-vocabulary detector to predict bounding boxes for a large Web image-text dataset to produce pseudo-annotations.

2. Self-train a new detector on the pseudo-annotations.
3. Optionally, fine-tune the self-trained model briefly on human-annotated detection data.

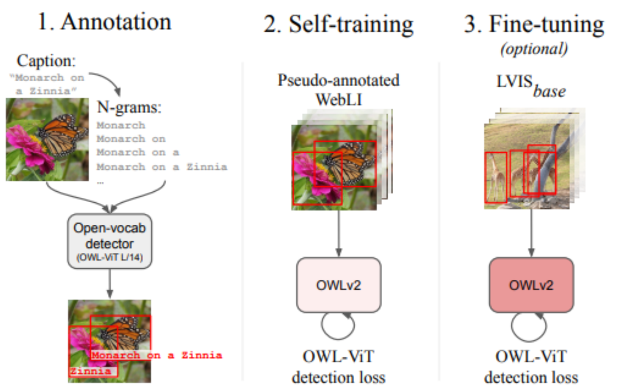


Figure 4.2: Overview of the OWL-V2 method[26]

The self-training approach encounters three primary challenges: *label space selection*, *pseudo-annotation filtering*, and *training efficiency*[26]. The objective of the **OWLv2** method is to address these challenges in a systematic and optimized manner. In the following sections, we provide a detailed description of the *OWLv2* method and how it overcomes these limitations.

4.3.1 Step 1: Generating Pseudo-Annotations

We use the WebLI dataset as the source of weak supervision for self-training. The WebLI dataset consists of approximately 10 billion images and their associated alt-text strings available on the public web. The **OWL-ViT CLIP-L/14** is employed to annotate these images with pseudo-bounding boxes. As an open-vocabulary object detector, **OWL-ViT** first detects objects in a class-agnostic manner and then assigns scores to the detected objects based on their association with free-text queries.

A key design decision in this process is the annotation label space.The two following alternatives are explored[26]:

1. Using a fixed, human-curated label space for all images: We combine all the labels sets from LVIS, Objects365, OpenImagesV4, and Visual Genome datasets and remove duplicates and plural forms. We obtain a total of 2520 categories.

2. Machine-generate per-image queries from image-associated text: We automatically generated queries from the image-associated text, we use no grammatical parsing and simply extract all word N-grams up to length 10 from the text associated with a given image and use them as queries for that image, to avoid biases from grammatical parsing (extracting nouns/verbs for example). An example of an N-gram is the following: if the text describes "a black cat sitting on a red sofa", unigrams, bigrams, trigrams, etc., will be generated:
    - Unigrams: "black", "cat", "sitting", "sofa"
    - Bigrams: "black cat", "red sofa"
    - Trigrams: "black cat sitting", "sitting on sofa"

As for filtering these pseudo-annotations, "Regardless of label space, we ensemble predictions over seven prompt templates such as "a photo of a ". For each predicted box, we keep the query with the highest score as its pseudo-label. For each image, we keep all boxes above a score threshold of 0.3. This threshold was selected after performing a grid search. The pseudo-annotations are used as hard labels for self-training." [26]

It was observed that the machine-generated per-image queries yielded better results than the human-curated label space. A mixture of human and machine-generated label spaces performs well in all settings, but does not significantly outperform the purely machine-generated label space. The broader and more flexible query set generated dynamically for each image enabled better alignment with the diverse and noisy data in WebLI, leading to improved pseudo-annotation quality and more robust training performance in downstream object detection tasks.

4.3.2 Step 2: Self-training at Scale

Self-training represents the primary enhancement introduced in **OWLv2** over **OWL-ViT**, and the following provides an in-depth overview of this process:

Initialization

The model's image and text encoders are initialized from pre-trained contrastive image-text models, such as CLIP. These encoders have already been trained on a large-scale dataset of image-text pairs, enabling them to capture and understand the relationships between visual content and textual descriptions effectively.

The detection heads, responsible for generating bounding boxes and class probabilities, are randomly initialized. Starting from scratch allows the detection heads to learn specifically from the pseudo-annotations generated during training, adapting directly to the object detection task.

Self-training

The self-training process is structured around the following key components:

- **Training data:** The training process begins by exclusively using pseudo-annotations generated by the **OWL-ViT** model, as described in Section 4.3.1. These pseudo-annotations serve as weakly supervised data, allowing the model to improve its detection capabilities without relying on manual annotations.

- **Loss Functions:** The training leverages the same loss functions as the OWL-ViT model, which have been proven effective in guiding the model to make precise decisions. These include contrastive loss for aligning visual and textual embeddings, as well as object detection losses for bounding box regression and classification.

- **Pseudo-negatives:** To further enhance learning, the query embeddings are augmented with pseudo-negatives, which are randomly sampled queries from other images in the dataset. This augmentation mimics the concept of batch negatives in contrastive learning, forcing the model to distinguish between relevant and irrelevant queries. As a result, the model becomes better at identifying correct matches and avoiding false positives, improving its ability to handle diverse and challenging detection scenarios.

- **Token Dropping:** Both natural and Web images contain low-variance areas devoid of useful information, e.g. sky, single-color backgrounds, or padding. These areas do not affect the detection performance of our model. Since ViTs represent images as an unordered sequence of tokens, token can be dropped without changing the model's parameters. Therefore, the image patches are ordered based on their pixel variance and the 50% with the lowest variance are discarded during training. During inference, all patches are preserved to ensure full information is utilized.

- **Instance Selection: OWL-ViT** is an encoder-only architecture, so it predicts one bounding box per encoder token. This is inefficient, since there are typically many more encoder tokens than objects, and most output tokens therefore do not represent objects. We introduce an **objectness head** which predicts the likelihood that an output token actually represents an object, and compute boxes, class scores, and losses only for the *top-k* tokens by objectness. In particular, we select 10% of instances by top objectness during training, but we keep all instances during inference.

- **Mosaics:** During self-training, raw images are combined into grids of up to 6 x 6 to create a single training example. This is done to increase the number of raw images seen for a given fixed model input resolution. We prefer this approach rather than accepting different input image sizes, because the latter requires resizing image position embeddings for each input size, introducing an additional complexity. Moreover, mosaics reduce the average object size, enhancing small-object detection performance.

By incorporating these strategies, the self-training process effectively addresses the challenges of weak supervision, low-resolution data, and inefficiencies, resulting in a model that is more robust and capable of handling diverse detection scenarios.

3. Fine-Tuning

While self-training using pseudo-annotations alone delivers strong performance, briefly fine-tuning on human annotations can yield further significant improvements. However, fine-tuning open-vocabulary models introduces a critical trade-off: it enhances performance on the fine-tuned classes but can reduces the model's open-vocabulary capabilities. Specifically, it is observed that the model's initially high robustness to distribution shifts diminishes after fine-tuning, leading to poorer performance on unseen classes, even as its accuracy on fine-tuned classes improves.

To address this trade-off, an ensemble approach is proposed which averages the model weights before and after fine-tuning. This method requires no additional training, making it a zero-cost solution. The experiments confirm that this **Model Ensembling** strategy achieves the best overall performance, preserving robustness and ensuring balanced performance across both fine-tuned and unseen classes.

In the following section, we describe how we leverage the **OWLv2** model in our work to build and end-to-end pipeline for zero, one, and few-shot detection.

Chapter 5

Implementation

5.1 Model Configuration

As our choice of architecture has fallen upon **OWLv2** described above, we leverage the *HuggingFace* implementation publicly available on GitHub. *HuggingFace* is a robust and highly structured library for state-of-the-art technologies, offering an efficient user-friendly framework for model deployment. It also provides a well-organized, comprehensive, and in-depth documentation, along with numerous examples found at HuggingFace.

For our project, we have utilized the **Owlv2ForObjectDetection** class from the **Transformers** library, which is specifically designed to leverage the **OWLv2** architecture for object detection tasks. This class provides a set of powerful functions that enable us to efficiently process and predict objects in images. Specifically, we utilize the *image_embedder*, which generates image embeddings, the *box_predictor*, which predicts bounding boxes for detected objects, and the *class_predictor*, which assigns class labels to the objects within those boxes. In addition, we employ the *Owlv2Processor*, which integrates both an Image Processor and a CLIP Tokenizer. The *Owlv2Processor* is responsible for preprocessing the input images and tokenizing them in a way that is compatible with the OWLv2 model. Together, these classes enable us to fully utilize the OWLv2 architecture's functionalities, making the model highly effective for our object detection tasks. Although HuggingFace provides a ready-to-use image-guided function for object detection, we opt not to use it, as its abstracted implementation limits user control over critical parameters essential for increasing model adaptability to a wide variety of use cases. Instead, we utilize the base **OWLv2** model and define our own image-guided sequence (see Fig. 5.10), allowing greater flexibility in query embedding extraction, data processing, model inference, and results post-processing.

We have already covered the 2-stage recipe of **OWLv2**, emphasizing that the Encoders are pre-trained contrastively on large amounts of text-image pairs. Our project focuses on the second phase, were we leverage these pre-trained encoders and integrate them in our pipeline for the detection task. Thus, the encoders are initialized from checkpoints pre-trained by Google. In particular, we use the google/owlv2-base-patch16-ensemble checkpoint. We selected this checkpoint due to its *"ensembling"* (see Section 4.3.2) of model weights before and after fine-tuning, a technique that has been shown to deliver the best overall performance over fine-tuned and unseen classes both. Additionally, the 16x16 patch size of the model represents an ideal balance between computational efficiency given the resources at hand, and the ability to capture fine-grained image features, ensuring satisfactory detection performance.

As for the hardware setup, this project was developed on an HP ZBook Power workstation, running Windows 10 Enterprise. The system is equipped with a 13th Generation Intel Core i7-13800H processor, which operates at 2.50 GHz, coupled with 32GB of RAM, providing ample memory for the computational demands of our project. For GPU acceleration, the workstation features an NVIDIA RTX A1000 GPU with 6GB of VRAM, offering robust performance for tasks like deep learning and image processing. However, our code is also adapted to run without GPU support, though at the cost of speed, but it renders the project flexible for systems without dedicated GPU resources.

In the following, we provide a detailed, step-by-step breakdown of the end-to-end pipeline that is our solution, covering all the details and nuances of our implementation.

At a high level, our goal is straightforward: extract the query embeddings from the query images and use them to perform object detection on the test images. To this aim, we present our solution which not only successfully accomplishes this task, but also provides significant user flexibility and control over the various functionalities. This ensures that users can easily tailor the product to meet their specific needs depending on the use case, while maintaining high performance and accuracy. The full code can be found at GitHub

5.2 Zero-Shot Object Detection on Query Images

To initiate the detection process, users must provide their query images and place them in a dedicated directory specified as a parameter to the program. Each query image should strictly contain only one object and must adhere to the following filename format: "**{category_id}_{category_name}_{instance_number}**". This unique naming convention serves two purposes:

1. It enables the program to associate the extracted embeddings with a specific category, which is crucial for zero-shot detection. Since zero-shot detection is class-agnostic, i.e, it identifies objects without prior knowledge of their classes, the filename acts as a guide for mapping the identified object to a known category.

2. It supports few-shot detection by allowing multiple query images for the same category to be distinguished and managed effectively. The program introduces a parameter, k, which specifies the k-shot setting, representing the number of query images provided per class. The inclusion of the **{instance_number}** in the filename ensures unique identification of each query image for a given category, while imposing control over the maximum number of query images allowed per class.

```python
images = []
for image_name in os.listdir(image_dir):
    if image_name.endswith((".png", ".jpg", ".jpeg", ".bmp", "JPEG")):
        category = ID2CLASS[float(image_name.split("_")[0])]

        #Taking only k query images to perfrom k shot detection
        k_hat = image_name.split("_")[-1].split(".")[0]
        if k_shot is not None and int(k_hat) > k:
            continue

        image_path = os.path.join(image_dir, image_name)
        image = cv2.cvtColor(cv2.imread(image_path), cv2.COLOR_BGR2RGB)
        if image is not None:
            images.append((image, category))
    return images

```

Figure 5.1: Extracting classes of query images imposing control on the parameter K using the filename of each image.

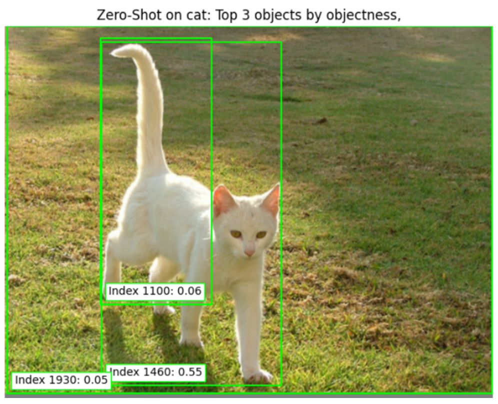
Once the query images and their respective category information are loaded, the program proceeds as follows:

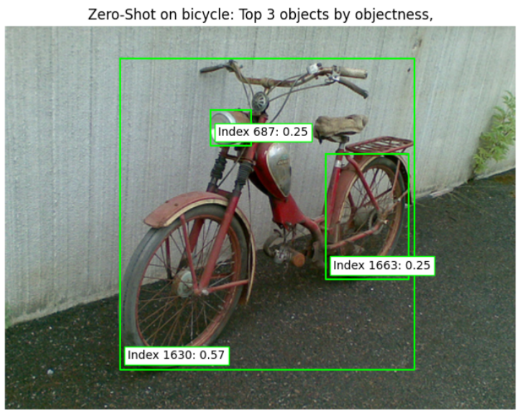
1. **Preprocessing with `Owlv2Processor`** The query images are first passed to the `Owlv2Processor`, which converts them into PyTorch tensors. This transformation ensures compatibility with the model and facilitates batch processing, improving computational efficiency. Additionally, this step allows the tensors to be transferred to a GPU for faster processing.
2. **Feature Extraction and Predictions** The transformed tensors are fed into the model, which extracts features and generates predictions. The model outputs:
   - **Bounding box predictions**, representing the spatial coordinates of detected objects.
   - **Objectness scores**, indicating the model's confidence that a bounding box contains an object, irrespective of its class.

```python
for batch_start in range(0, len(images), args.query_batch_size):

    torch.cuda.empty_cache()
    image_batch = images[batch_start : batch_start +
    ↪   args.query_batch_size]
    source_pixel_values = processor(
        images=image_batch, return_tensors="pt"
    ).pixel_values.to(device)
    with torch.no_grad():
        feature_map = model.image_embedder(source_pixel_values)[0]

        # Rearrange feature map
        batch_size, height, width, hidden_size = feature_map.shape
        image_features = feature_map.reshape(batch_size, height * width,
        ↪   hidden_size).to(device)

        # Get objectness logits and boxes
        objectnesses = model.objectness_predictor(image_features)
        source_boxes = model.box_predictor(image_features,
        ↪   feature_map=feature_map)
        source_class_embedding =
        ↪   model.class_predictor(image_features)[1]
```

At this stage, it is necessary to identify which of the predicted bounding boxes correctly encompasses the query object to extract their corresponding query embeddings as the class embeddings. If the parameter *visualize_query_images* is set to *True*, the program visualizes the *top_k* predicted bounding boxes—ranked by objectness score—on the query image. The *top_k* parameter is user-controlled, enabling flexibility in visualizing the predictions made by the model.

Below are examples of query images with their top 3 zero-shot predictions:





The program provides users with two options for query embedding selection: **automatic** and **manual**, controlled by the boolean parameter *manual_query_selection*.

5.2.1 Automatic Query Selection

In automatic mode, the bounding box with the highest objectness score is selected, and its corresponding query embeddings are assigned as the embeddings of the mapped class.

If the user-provided query images are cropped or zoomed in—such that the entire image represents the query object—the *cls* flag, a model parameter, must be enabled. Specifically:

- Set *cls* to 1 if the image is cropped.
- Set *cls* to 0 if the image contains background elements.
- Alternatively, *cls* can be a bitmap array containing flags for all query images in a batch.

As noted in Section 4.3, token pooling has been removed from the **OWLv2** model. Consequently, there is no token embedding representing the entire image, and aggregating all tokens is infeasible because the resulting embedding lacks meaningful representation. To address this, we represent the whole image by selecting the predicted bounding box with the maximum area, under the following conditions:

1. The area of this box is at least 70% greater than that of the box with the highest objectness score.

2. The objectness score of the maximum-area box exceeds 0.1.

If these conditions are not met, the resulting embeddings will lack meaningful features and risk confusing the model. In this case, we default to selecting the bounding box with highest objectness score.

In Fig. 5.4, we show two examples of cropped query images. Query images may be cropped either because they are extracted using an automated script that crops objects from a dataset, or because the user provides manually cropped images for improved precision during query embedding generation.

```python
for i in range(batch_size):
  # select max objectness score box
  if cls[batch_start + i] == 0:
      max_index = torch.argmax(current_objectnesses[i])
      indexes.append(max_index.cpu().item())
      query_embedding = current_class_embeddings[i, max_index]

  # if selecting cls token, take from the max objectness boxes, the one that has maximum area
  elif cls[batch_start + i] == 1:
      #first find max objectness box
      idx_obj = torch.argmax(current_objectnesses[i])
      box_obj = current_boxes[i, idx_obj]
      area_obj = box_obj[2] * box_obj[3]

      #find max area box
      topk_indices = torch.topk(current_objectnesses[i], args.topk_query).indices
      topk_boxes = current_boxes[i,topk_indices]
      areas = [box[2] * box[3] for box in topk_boxes]
      max_area_index = torch.argmax(torch.tensor(areas))
      max_area = areas[max_area_index]
      objectness = round(float(current_objectnesses[i, topk_indices[max_area_index]]),2)

      #first check: threshold max_area_box on objectness score and compare area
      if objectness < 0.1 or (area_obj/max_area) <= 0.7:
          indexes.append(idx_obj.cpu().item())
          query_embedding = current_class_embeddings[i, idx_obj]

      else:
          idx = topk_indices[max_area_index]
          indexes.append(idx.cpu().item())
          query_embedding = current_class_embeddings[i, idx]

```

Figure 5.2: Class query embeddings extraction.

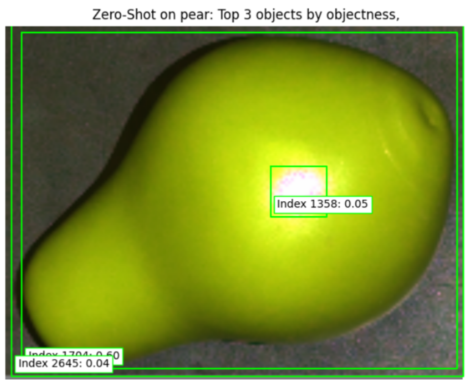


Figure 5.3: The box with maximum area has very low objectness score, while that with maximum objectness score has an area very close to the max area, so we choose the latter.

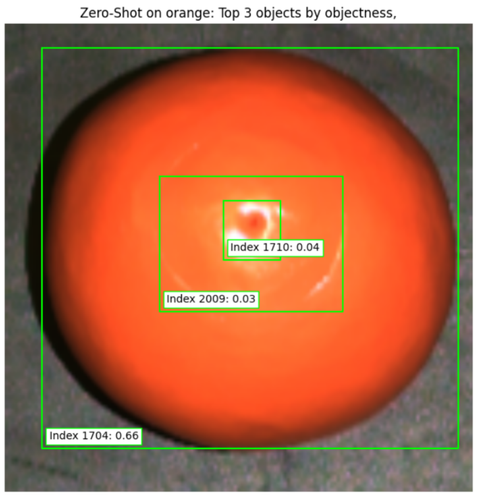


Figure 5.4: The maximum area box coincides with the maximum objectness score, as is the case 90% of the time.

5.2.2 Manual Query Selection

If the automatic query selection algorithms and options fail to select the target box that correctly represents the query object, the user can set the parameter *manual_query_selection* to *True* to perform the selection themselves.

The query embeddings produced by the model have the dimension $[512, 3600]$, 512 being the dimension of the embedding of each patch (hidden size), and 3600 being the number of proposed (possibly overlapping) regions. The bounding boxes have dimension [4,3600], and the query embeddings and boxes are aligned, meaning that the query embedding of bounding box i is found at the index i of the query embeddings array.

It is possible to manually select embeddings for specific classes while leaving automatic predictions for others. To do so, the user must specify the class name they wish to modify and the index of the query embedding corresponding to the patch they want to select for that class (i.e., one of the 3600 predicted regions).

This feature is useful because sometimes the model fails to correctly extract the query object using zero-shot detection. The box containing the object is among the predicted boxes, but the defined heuristics for box selection cannot choose it, because it doesn't have maximum objectness nor maximum area if the image is cropped. Below is an example of such a query image:

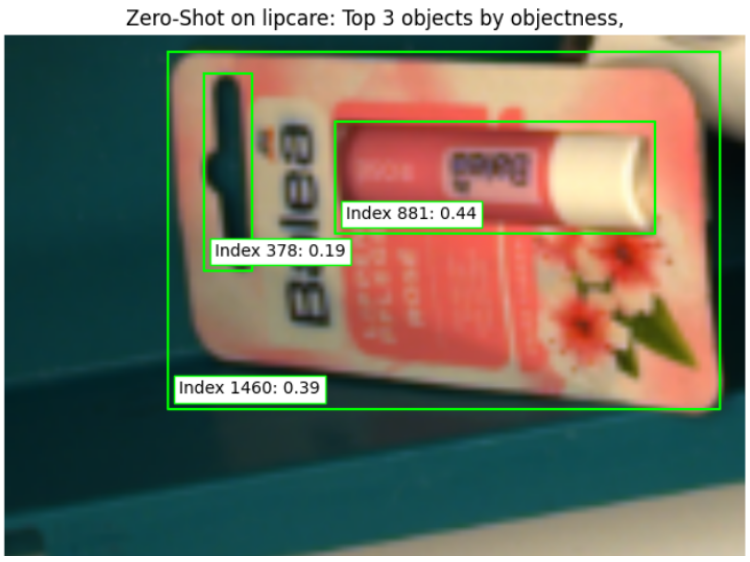


Figure 5.5: Automatic query selection fails to select the correct box with index 1460 and objectness score 0.39.

If the user prefers to avoid manual selection, they could either take/select another query image for the given class, or crop the query image around the object, which will lead the automatic selection to choose the box with the maximum area, i.e, the correct box. Otherwise, we continue to describe the manual selection process:

The program, when performing the zero-shot detection, besides producing the query images with the predicted bounding boxes, also produces a file called "*objectness_indexes_{args.comment}.json*". This file contains for each class, the predicted details about the predicted boxes, namely the *index* and *objectness score* of each of the *top_k* boxes. The entries look like this:

```
[
    {
        "category": "pear",
        "boxes": [
            {
                "index": 1358,
                "objectness": 0.05
            },
            {
                "index": 1704,
                "objectness": 0.6
            },
            {
                "index": 2645,
                "objectness": 0.04
            }
        ]
    },
      {
        "category": "mug",
        "boxes": [
            {
                "index": 1233,
                "objectness": 0.47
            },
            {
                "index": 1819,
                "objectness": 0.16
            },
            {
                "index": 2838,
                "objectness": 0.09
            }
        ]
    }
]
```

Figure 5.6: Details about the top 3 boxes ranked by objectness score.

This file contains the same information as the visualized query images, eg: Figure 5.3, but it is useful as it allows the extraction of the index of the required box in case the boxes are overlapped and the index cannot be clearly read from the image. The manual step thus consists of specifying the class and the corresponding index, as shown below:

```python
if options.manual_query_selection:
        zero_shot_detection(model, processor, options, writer)

        categories = ["sugar_box", "wineglass"]
        idx = [1689, 2166]

        indexes = modify_max_objectness_indices(
            os.path.join(query_dir, f"objectness_indexes_again.json"),
            categories,
            idx)
        indexes = [v for k, v in indexes.items()]
        query_embeddings, classes = find_query_patches_batches(
            model, processor, options, indexes, writer
        )

    else:
        indexes, query_embeddings, classes = zero_shot_detection(
            model,
            processor,
            options,
            writer,
            cls
        )
```

Figure 5.7: Zero-Shot Detection with manual query selection option

At the end of this step, we have *classes* and corresponding *query_embeddings* which we will use to perform One-Shot Detection.

5.2.3 Few-Shot Object Detection

To perform K-Shot detection on test images, we collect K query images for each class. The query embeddings are extracted in the same manner as described for individual images, which means we can choose between automatic or manual embedding selection, as well as cropped or non-cropped images. After extracting these embeddings, two possible approaches can be used to handle them:

1. **Independent Query Embeddings**: In this approach, the query embeddings for each of the K query images of a class are kept independent. This means that each query image has its own embedding, and during inference, the

test image embeddings are compared with each of the K query embeddings as before. Importantly, **Non-Maximum Suppression (NMS)** is applied during post-processing, even if the predictions belong to the same class but originate from different query images.

This approach is particularly useful when the K query images for a class are **significantly different**, such as when they show the object from different angles or perspectives, capturing different features. This diversity ensures that the model has multiple representations of the same class, which can be beneficial in detecting the object under varying conditions or views.

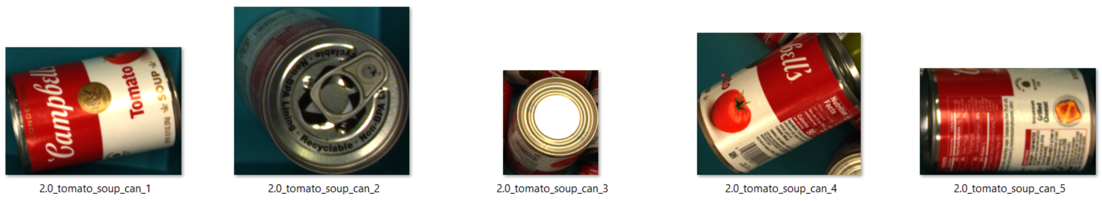


Figure 5.8: Example class where independent query embeddings are required.

2. **Averaged Query Embeddings**: Alternatively, the query embeddings of the K query images for the same class can be averaged. This is done by performing **Average Pooling** on the K query embeddings, resulting in a single combined query embedding for the class.

   This approach is more suitable when the K query images are **very similar** and do not reveal significant new information about the object (e.g., if they are simply different views of the same object from similar angles). By averaging the embeddings, we reduce redundancy and avoid the extra computational overhead that comes from performing **NMS** on multiple similar predictions. This can lead to more efficient processing but may lose some of the diversity present in the first approach.

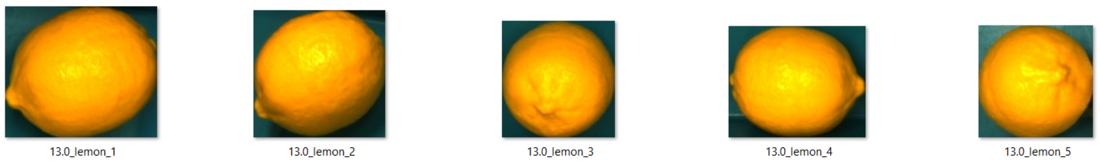


Figure 5.9: Example class where it is better to average query embeddings.

While in the previous case no code modification is required, to perform Average

Pooling of the queries of the same class, we set the parameter *average_queries* to *True*, to execute the following block of code:

```python
    if args.k_shot > 1 and args.average_queries:
        class_embeddings_dict = {}

        # Group queries of same class together
        for embedding, class_label in zip(query_embeddings,
        ↪   classes):
            if class_label not in class_embeddings_dict:
                class_embeddings_dict[class_label] = []
            class_embeddings_dict[class_label].append(embedding)

        query_embeddings = []
        classes = []
        for class_label, embeddings in
        ↪   class_embeddings_dict.items():
            average_embedding = torch.mean(torch.stack(embeddings),
            ↪   dim=0)
            query_embeddings.append(average_embedding)
            classes.append(class_label)

```

5.2.4 Supercategories

Another feature our program offers is the ability to generalize over predicted categories by grouping conceptually or visually similar classes under a common label. This is achieved by mapping individual categories to their corresponding *supercategories*.

This mapping can be applied **before** or **after** inference, with different implications for model performance and result interpretation:

1. **Pre-Inference Mapping** When categories are grouped into supercategories *before* inference, the query embeddings used for each class (now representing a supercategory) are aggregated from multiple individual categories. Since all these embeddings are mapped to the same label, the **Non-Maximum Suppression (NMS)** algorithm will filter them together, significantly impacting the final predictions.

    - This approach may improve robustness by consolidating similar object types, reducing confusion between fine-grained categories.

- However, it may also lead to a loss of specificity, as multiple distinct categories will be treated as a single class.

2. **Post-Inference Mapping** When the mapping is applied *after* inference, the raw detection results remain unchanged, but their labels are generalized at a higher level.

   - This is useful for applications that require more abstract categorization, such as summarizing results in broad object groups (e.g., grouping "lion," "tiger," and "leopard" under "big cats").
   - It can also help align the detected categories with a more human-intuitive taxonomy, making the output more interpretable for downstream tasks.

Thus, whether to apply supercategory mapping before or after inference depends on the specific objectives of the application—whether the goal is to modify the detection behavior or simply to generalize the final labels for easier interpretation.

From a practical point of view, enabling this option requires the user to set the parameter *use_supercategories* to `True` to perform the mapping before inference, or alternatively, to set *generalize_categories* to `True` to perform the mapping after inference.

These two parameters are **mutually exclusive**, meaning they cannot both be set to `True` simultaneously. Additionally, the user must provide the mapping, which is essentially a dictionary where the categories serve as keys and the corresponding supercategories as values.

5.2.5 Seamless addition of a new category

The main objective of this project is to be able to add a new category to be recognized, without collecting labeled data about this category, without training the model on the model, and without re-starting the server.

Our program fully supports this functionality. If the user adds a new query image to the designated query image directory during runtime, the program dynamically integrates the new class into the detection pipeline.

To ensure consistency, the program temporarily **locks** the *query_embeddings* and *classes* resources, preventing their use in the **One-Shot Detection** phase while processing the new query image. It then performs **Zero-Shot Detection** on the newly added image, extracting its embedding and associating it with a new class.

Once this process is complete, the updated *query_embeddings* and *classes* are unlocked, allowing **One-Shot Detection** to resume seamlessly. The key difference is that inference now considers the newly introduced class, without requiring a full restart of the system, nor any data collection and training of the network.

```python
# Create an observer to watch for new query images
event_handler = ImageHandler(model, processor, options, query_embeddings, classes, writer, cls, lock)
observer = Observer()
observer.schedule(event_handler, path=query_directory, recursive=False)
observer.start()

# The function triggered when a new query image is observed
class ImageHandler(FileSystemEventHandler):
def __init__(self, model, processor, options, query_embeddings, classes, writer, cls, lock):
    ...

def on_created(self, event):
    if event.is_directory:
        return
    if event.src_path.endswith((".png", ".jpg", "JPEG")):
        print(f"New query image detected: {event.src_path}")
        with self.lock:
            self.query_embeddings, self.classes = add_query(
                self.model,
                self.processor,
                self.options,
                event.src_path,
                self.query_embeddings,
                self.classes,
                self.writer,
                self.cls
            )
            print("Updated query_embeddings and classes.")

```

Below is a visual representation of the entire flow for extracting the query embeddings:

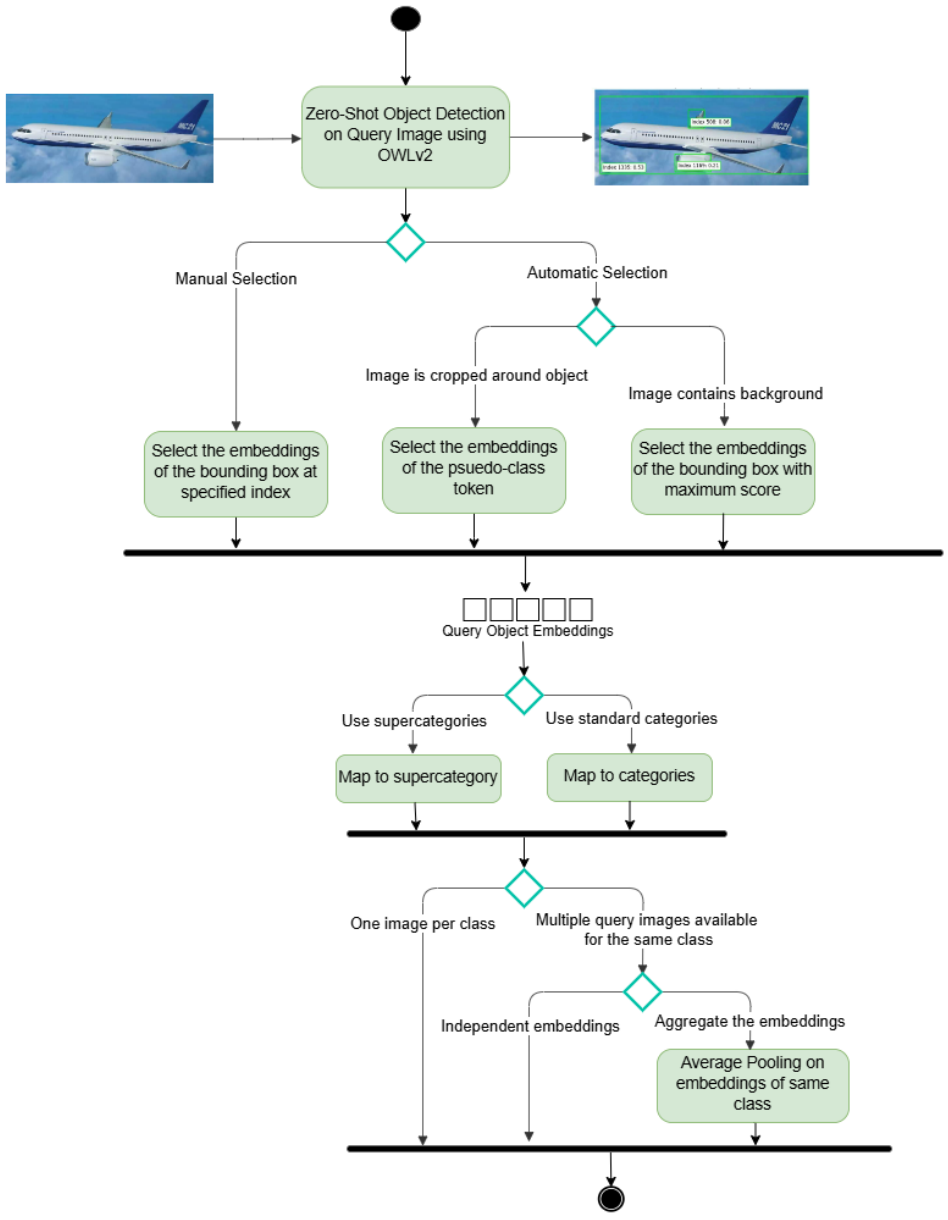

Zero-Shot Object Detection on Query Image using OWLv2
Manual Selection
Automatic Selection
Image is cropped around object
Image contains background
Select the embeddings of the bounding box at specified index
Select the embeddings of the psuedo-class token
Select the embeddings of the bounding box with maximum score
Query Object Embeddings
Use supercategories
Use standard categories
Map to supercategory
Map to categories
One image per class
Multiple query images available for the same class
Independent embeddings
Aggregate the embeddings
Average Pooling on embeddings of same class

Figure 5.10: Query Embedding Extraction – Our Approach

This concludes the **Zero-Shot Detection** phase and all its different features.

5.3 One-Shot Object Detection on Test Images

This is the core part of the application, where the images are provided to the program for object detection. As a starting point, the user must configure a set of parameters which control the behavior of the detection process.

```json
{
    "mode": "test",
    "data": "MGN",
    "source_image_paths": "Queries/Comau_cropped",
    "target_image_paths": "Test/Comau/3D",
    "backbone": "google/owlv2-base-patch16-ensemble",
    "comment": "interface_test",
    "query_batch_size": 8,
    "test_batch_size": 8,
    "topk_query": 3,
    "topk_test": 20,
    "k_shot": 1,
    "average_queries": false,
    "manual_query_selection": false,
    "confidence_threshold": 0.95,
    "visualize_query_images": true,
    "visualize_test_images": true,
    "nms_threshold": 0.3,
    "nms_between_classes": false,
    "write_to_file_freq": 5,
    "generalize_categories": false,
    "use_supercategories": false
}

```

These parameters can either be provided in a *json* configuration file or via the command line.
Below, we explain what each parameter means and how it influences the object detection process:

- `mode`: Defines the operation mode of the program, either `test` or `validation`. The `test` mode evaluates the model on unseen data, while `validation` mode is used to give a quantitative measure to the model's performance, or to tune hyperparameters.

- `data`: Specifies the dataset being used. Possible values include `ImageNet`, `COCO`, `MGN`, or `TestData` for miscellaneous data.

- `source_image_paths`: The directory containing the query images used for object detection.
- `target_image_paths`: The directory containing the target images on which detections will be performed.
- `backbone`: The pre-trained checkpoint with which we initialize our `OWLv2` model, such as *google/owlv2-base-patch16-ensemble*.
- `comment`: An optional field for adding a custom comment to the run, which can help in tracking experiments. It is also used for naming the files generated during runtime.
- `query_batch_size`: The batch size for the query images during processing.
- `test_batch_size`: The batch size for the test images during object detection.
- `topk_query`: The top `k` objectness scores in the query images, used to filter out low-confidence queries.
- `topk_test`: The top `k` predictions by score to keep from the test images after processing.
- `k_shot`: The maximum number of examples used for each object during object detection (K-shot learning).
- `average_queries`: If set to `true` and `k_shot`, this will average the query embeddings that belong to the same class from multiple query images to improve the robustness of the predictions.
- `manual_query_selection`: If set to `true`, the user will manually select the zero-shot prediction on the query images that represents correctly the query object.
- `confidence_threshold`: The minimum confidence score required for a detection to be considered valid. It can be a fixed score for all classes, or a list with class-specific confidence thresholds can be provided.
- `visualize_query_images`: If set to `true`, query images will be visualized during processing.
- `visualize_test_images`: If set to `true`, test images will be visualized during object detection.
- `nms_threshold`: The threshold value for non-maximum suppression (NMS) to filter out redundant bounding boxes based on their overlap.

- `nms_between_classes`: If set to `true`, NMS will be applied between classes to suppress overlapping boxes from different classes.
- `write_to_file_freq`: Specifies how frequently the results should be written to file, in terms of batches processed.
- `generalize_categories`: If set to `true`, categories in the results file will be generalized by mapping them to supercategories after performing prediction.
- `use_supercategories`: If set to `true`, the model will map categories to their corresponding supercategories and use them for predictions instead of individual categories.

All these parameters are aggregated in a custom class, `RunOptions`, that can parse them from *json* files as well as from the command line.

Having provided the necessary settings, we pass to the `one_shot_detection` function the `query_embeddings` and corresponding `classes` computed in the **Zero-Shot Detection** phase, based on which it is required to perform the detection.

The model proceeds as follows:

1. **Preprocessing with `Owlv2Processor`** Just like the query images, the test images are processed by the `Owlv2Processor`, which transforms them into PyTorch tensors. This conversion ensures the images are in a format compatible with the model and optimizes batch processing for improved computational performance. Moreover, this step allows the tensors to be moved to the GPU, enabling faster processing.
2. **Feature Extraction and Predictions** The transformed tensors are fed into the model, which extracts features and generates predictions. In particular:
   - The feature map of each image is passed to the `model.class_predictor`, alongside the tensor containing all the `query_embeddings`, to produce class confidence scores.
   - The `model.box_predictor` produces the proposed bounding boxes, i,e, the regions of the image that are likely to contain an object.
3. **Postprocessing** The results obtained, the bounding boxes and class probability scores, are post-processed using `processor.post_process_object_detection`. This step is necessary because the model resizes and pads the test images all to the same size, and thus the raw predictions it returns are not true to the image real dimensions.

   These few steps in code:

```python
with torch.no_grad():
    feature_map = model.image_embedder(target_pixel_values)[0]
    b, h, w, d = map(int, feature_map.shape)
    target_boxes = model.box_predictor(
        feature_map.reshape(b, h * w, d), feature_map=feature_map
    ) # dimension = [batch_size, nb_of_boxes, 4]

    reshaped_feature_map = feature_map.view(b, h * w, d)

    query_embeddings_tensor = torch.stack(query_embeddings) #
     ↪  Shape: (num_batches, batch_size, hidden_size)

    target_class_predictions, _ =
     ↪  model.class_predictor(reshaped_feature_map,
     ↪  query_embeddings_tensor)  # Shape: [batch_size,
     ↪  num_queries, num_classes]

    outputs = ModelOutputs(logits=target_class_predictions,
     ↪  pred_boxes=target_boxes)
    results
     ↪  =processor.post_process_object_detection(outputs=outputs,
     ↪  target_sizes=target_sizes, threshold=0)

```

4. **Filtering** Since the model generates 3600 predictions per image, a filtering step is necessary to retain only the most relevant and confident detections. This process is controlled by two key parameters:

   (a) `confidence_threshold`: This mandatory parameter ensures that only predictions with a class probability greater than or equal to the specified threshold are retained. The threshold can be applied globally across all classes or set individually for each class, allowing for finer control over the filtering process. A higher threshold reduces false positives, while a lower threshold captures more potential detections.

   (b) `topk_test`: This optional parameter specifies the maximum number of predictions to retain per image. When used, it works in conjunction with the `confidence_threshold` to further refine the selection process. For example, setting `topk_test` to 20 ensures that, among all predictions, only the 20 highest-scoring detections per image are considered, with the additional constraint that they must also meet the confidence threshold requirement. This helps balance precision and recall by limiting the number of predictions while maintaining high-confidence detections.

```python
    scores = torch.sigmoid(target_class_predictions)

    if args.topk_test is not None:
        top_indices = torch.argsort(scores[:, :, 0], descending=True)[:, :args.topk_test]
        scores = scores[torch.arange(b)[:, None], top_indices]
        target_boxes = target_boxes[torch.arange(b)[:, None], top_indices]

    if args.mode == "test":
        if isinstance(args.confidence_threshold, (int, float)):
            top_indices = (scores >= args.confidence_threshold).any(dim=-1)
        else:
            idxs = torch.argmax(scores, dim=-1)
            # Compare the scores with the corresponding class-specific threshold
            predicted_classes = [[classes[idx] for idx in image_idxs] for image_idxs in idxs.tolist()]
            thresholds = [[args.confidence_threshold[class_id] for class_id in image_classes] for image_classes in predicted_classes]
            thresholds_tensor = torch.tensor(thresholds, device=scores.device)
            max_scores = scores[torch.arange(scores.size(0)).unsqueeze(1), torch.arange(scores.size(1)).unsqueeze(0), idxs]
            top_indices = (max_scores > thresholds_tensor).to(device)

        filtered_boxes = [target_boxes[i][top_indices[i]] for i in range(b)]
        filtered_scores = [scores[i][top_indices[i]] for i in range(b)]
```

5. **Padding** After filtering, the number of retained predictions is different for each image. Since our program supports batch processing, this poses a problem. To overcome this, we identify the image with the maximum number of predictions in the batch, and then we pad the predictions of all other images with empty entries so that all images in the batch have the same number of predictions.

6. **Non Maximum Suppression (NMS)** Object detection models often generate multiple overlapping bounding boxes for the same object, each with a different confidence score, all passing the confidence threshold. Non-Maximum Suppression (NMS) is a post-processing technique used to filter these redundant and overlapping predictions.

   NMS works by first sorting all predicted boxes by confidence score in descending order. The highest-scoring box is selected, and all other boxes that overlap with it significantly (as measured by Intersection over Union, IoU) are suppressed. This process is repeated iteratively until no more boxes exceed the predefined NMS threshold, used as IoU threshold.

   Using the boolean parameter `nms_between_classes`, we allow users to choose between class-wise NMS, which removes overlapping boxes both within the same class, and cross-class NMS, which suppresses overlapping boxes across different classes. Class-wise NMS preserves detections of different objects, while cross-class NMS helps reduce confusion between similar objects. This flexibility ensures optimal filtering based on the dataset and application needs.

   To pass the predictions to NMS, we need first to remove the padded empty predictions, and then to flatten them by removing the batch dimension. We use `batched_nms` from the `torchvision library`.

```python

nms_boxes, nms_scores, nms_classes, nms_image_indices = nms_batched(
    flattened_boxes, flattened_scores, flattened_classes, flattened_image_indices, args
    )
```

```python

# NMS function
def nms_batched(boxes, scores, classes, im_indices, args):

'''
Perform non-maximum suppression on a batch of bounding boxes.
Takes boxes in (x1, y1, x2, y2) format for NMS.
Returns boxes (x, y, width, height) format.

'''
if args.nms_between_classes:
    classes_nms = torch.zeros_like(classes)
else:
    classes_nms = classes

indices = batched_nms(boxes, scores, classes_nms, args.nms_threshold)
filtered_boxes = boxes[indices]
filtered_scores = scores[indices]
filtered_classes = classes[indices]
filtered_indices = im_indices[indices]
filtered_boxes_coco = convert_from_x1y1x2y2_to_coco(filtered_boxes)

return  filtered_boxes_coco, filtered_scores, filtered_classes, filtered_indices

```

The results returned by NMS are the final predictions of the model.

7. **Storing the Results** The predicted bounding boxes and class scores are stored in a structured list, where each element is a dictionary containing the detection information. To facilitate standardized evaluation, the results are saved in COCO format, ensuring compatibility with the COCO API for quantitative performance evaluation. For datasets like **COCO** and **MetaGraspNet**, the predictions are given `image_id` that corresponds to the real `image_id` in their corresponding ground truth files. Otherwise, we simply assign the image index.

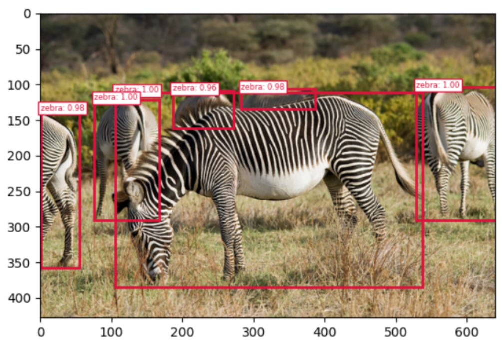
```python
for idx, (box, score, cls, img_idx) in
↪ enumerate(zip(nms_boxes_coco, nms_scores, nms_classes,
↪ nms_image_indices)):

    ... # img_id assignment

    coco_results.append({
        "image_id": img_id,
        "category_id": cls.item(),
        "bbox": rounded_box,
        "score": round(score.item(), 2)
    })
```

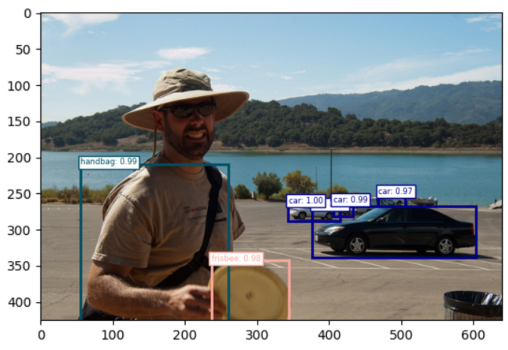
The results are stored in a *json* file named `results_args.comment.json`, where `comment` is a user-defined identifier. For large datasets, results are written to the file incrementally every `args.write_to_file_freq` iterations. This prevents excessive memory usage and ensures efficient resource management, particularly when working with GPU-accelerated processing.

If `generalize_results` discussed in Section 5.2.4 is set to *True*, an abstraction level is added to the results, and the categories are mapped to Supercategories in a post-processing step.

```python
if args.generalize_categories:
    map_supercategories(file)
```

8. **Visualizing the Results** For qualitative evaluation, the predicted bounding boxes can be optionally drawn on the test images and displayed. This visualization helps assess detection accuracy, identify potential misclassifications, and debug model performance intuitively. We demonstrate examples:





5.4 Real-time Deployment

To complete the project and prepare it for industrial applications, we built a server to support the execution of the program on the robot. The goal was to be able to run the application on Comau's Racer3 robot equipped with a Zivid 3D camera and a gripper. The server's functionality can be outlined as follows:

- **Parameter Initialization:** : The system loads user-defined parameters from a `json` configuration file, which are then stored in an instance of the `RunOptions` class. First, the server performs zero-shot detection on the query images present in the designated query directory. This process extracts query embeddings (`query_embeddings`) and their corresponding class labels (`classes`), which are subsequently used for object recognition.

- **Server Deployment** A TCP server is implemented using Python's `socket` module, listening for incoming connections on `localhost` at port 5001. To ensure responsiveness, the server operates in non-blocking mode, allowing it to handle multiple consecutive connections efficiently.

- **Dynamic Query Management** To accommodate new object classes dynamically, a watchdog observer is configured to monitor the query directory. Whenever a new query image is detected, the `add_query` function (detailed in Section 5.2.5) is triggered. This function processes the new image, updates the `query_embeddings`, and integrates the corresponding class into the detection pipeline without requiring a system restart.

```
     signal.signal(signal.SIGINT, git )
    # Create a TCP/IP socket
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

    # Bind the socket to the address given on the command line
    server_address = ('localhost', 5001)
    print('starting up on {} port {}'.format(*server_address))
    sock.bind(server_address)

    # Listen for incoming connections
    sock.listen(1)

    # Set the socket to non-blocking mode
    sock.settimeout(5)

    query_embeddings, classes = zero_shot()

    # Create an observer to watch for new query images
    event_handler = ImageHandler(model, processor, options,
     ↪  query_embeddings, classes, writer, cls, lock)
    # ImageHandler calls add_query
    observer = Observer()
    observer.schedule(event_handler, path=query_directory,
     ↪  recursive=False)
    observer.start()
```

- **Client Connection and Image Acquisition** Once the server is initialized and the query embeddings are prepared, it continuously listens for incoming client connections. Upon establishing a connection, it sequentially receives the image dimensions—first the width, then the height—followed by the actual image data. This structured approach ensures that the received image is correctly reconstructed and ready for processing.

```
    try:
        print("Initializing the connection :D")
        while True:
            try:
                while True:
                    print('waiting for a connection')
                    connection, client_address = sock.accept()
                    print('connection from', client_address)

                    # Receive the size of the image
                    w = int(connection.recv(1024).decode())
                    h = int(connection.recv(1024).decode())
                    image_data = receive_all(connection, w*h*3)
                    if not image_data:
                        break

```

In addition to the image data, the server receives a set of depth and normal information from the robot in the form of `(x, y, z, nx, ny, nz)` for each pixel in the point cloud. These values are essential for converting the 2D bounding box coordinates predicted by our detector into a precise 3D grasping point, enabling the robot to accurately pick up the detected object.

```
    data = []
   for i in range(6):
       d =  receive_all(connection, w*h*4) # 4 bytes for each
       ↪  float
       if not d:
           break
       d = np.frombuffer(d, dtype=np.float32)
       d = d.reshape(h,w,1)
       data.append(d)

```

- **Object Detection** Once all the data has been received from the client, the detection process begins. The system performs One-Shot Object Detection to identify the target objects within the image. The detected 2D bounding boxes are then converted into precise 3D grasping points using the depth and normal information, ensuring accurate object manipulation by the robot.

  We define the grasping point of each object as its centroid, which corresponds to the center of the bounding box. Once the centroid is computed, we extract the corresponding 3D coordinates from the point cloud data associated with the image. To optimize the robot's picking sequence, we prioritize objects closer to the camera. Predictions for these nearer objects are sent first, ensuring the robot picks up the objects that are positioned on top or closer in the 3D space.

```python
def find_grasping_points(data, predictions):
# Get the grasping points for each object
    grasping_points = []
    for i, pred in enumerate(predictions):
        # Get the bounding box for the object
        bbox = pred['bbox']
        category = pred['category_id']
        x, y, w, h = bbox

        # compute centroid
        cx, cy = int(x + w/2), int(y + h/2)

        # find xyz and nx ny nz
        grasping_point = [d[cy,cx,0] for d in data]
        grasping_point.append(category)
        grasping_points.append(grasping_point)

    # sort by increasing value of z
    grasping_points = sorted(grasping_points, key=lambda x:
     ↪  x[2])
    grasping_points = np.array(grasping_points)
    grasping_points = np.array2string(grasping_points,
     ↪  separator=' ', precision=3)

return grasping_points
```

  The predictions for each object are sent as a grasping point along with its associated class information, formatted as `(x, y, z, nx, ny, nz, category_id)`.

This provides both the 3D coordinates and the object classification needed for the robot to perform the grasping action.

Once the predictions are sent, the server continues running, remaining active and ready to receive additional images from the client for further processing.

Chapter 6

Experiments and Results

To evaluate the performance of the proposed system, we conducted both quantitative and qualitative experiments. Quantitative evaluations were performed on benchmark datasets, providing standardized metrics to assess the system's detection capabilities. Qualitative assessments, on the other hand, were carried out in a controlled laboratory environment and on a custom logos dataset, designed to explore the system's adaptability to novel use cases without predefined ground truth annotations.

6.1 Datasets

In this work, we employ two datasets to evaluate our proposed pipeline: COCO and MetaGraspNet. These datasets serve complementary purposes, with COCO providing a standardized benchmark for general object detection and MetaGraspNet focusing on robotic grasp detection.

6.1.1 COCO

The COCO (Common Objects in Context) dataset is a widely used benchmark for object detection, segmentation, and captioning tasks. It contains over 200,000 labeled images spanning 80 object categories, with varying levels of occlusion, lighting conditions, and background complexity. The diversity and richness of COCO make it an ideal dataset for evaluating the generalization capabilities of object detection models. Being a benchmark dataset, it is used to compare systems and establish performance baselines for new models.

6.1.2 MetaGraspNetv2 (MGN)

MetaGraspNet (MGN) is a specialized dataset designed for robotic grasp detection and object manipulation tasks. It features a diverse set of objects, including tools, household items, and industrial components, captured from multiple viewpoints with depth information. In total, it contains 83 object categories. Unlike COCO, which emphasizes general object detection, MetaGraspNet is particularly valuable for assessing how well models can recognize and localize objects relevant to robotic applications. Its inclusion in our evaluation enables us to test the adaptability of our approach to real-world robotic scenarios.

6.2 Performance Metrics

To assess detection performance, we use the mean Average Precision (mAP) metric, a standard evaluation criterion in object detection. The mAP score is computed by averaging the precision-recall curve over all object categories at a specified Intersection over Union (IoU) threshold. In our experiments, we primarily report **mAP@50**, which considers predictions correct if the IoU between the detected and ground-truth bounding boxes is at least 50%. This metric is particularly relevant for labeled datasets such as COCO and MetaGraspNet, where the presence of precise ground-truth annotations enables quantitative performance assessment.

However, for experiments conducted on unlabeled datasets, such as images captured in a laboratory setting where no predefined ground-truth annotations exist, mAP cannot be computed directly. Instead, we adopt a qualitative evaluation approach, in which model predictions are evaluated based on visual inspection of the generated detection results. This method allows us to subjectively analyze the accuracy and robustness of the detections in scenarios where traditional benchmarking is infeasible.

Additionally, for experiments conducted using the robot, performance is measured through the success rate of object retrieval. Specifically, we evaluate whether the robot is able to correctly detect, localize, and grasp the target objects based on the predicted bounding boxes. The effectiveness of our detection pipeline in these settings is therefore inferred from the practical success rate of object pickups rather than numerical metrics derived from labeled datasets.

6.3 Quantitative Evaluation

6.3.1 Numeric Results

We aim to compute the mean Average Precision (mAP) as a quantitative performance measure for our model. To facilitate this evaluation, we developed an automated script designed to streamline the process of fetching query images from various datasets. The script functions as follows:

First, it accesses the ground truth file of the training split of the dataset, and systematically parses the annotations to identify and extract bounding boxes for each object. The scripts crops K bounding boxes of each category, given that the area of the bounding boxes is at least 5000 pixels. These cropped boxes are saved as new images in a specified query directory, with filenames that adhere to the filename format of query images, indicating the object category and instance.

After the acquisition of the query images, we can perform up to K-Shot Detection on the validation split of the dataset.

We now proceed to evaluate the one-shot performance on COCO and MGN. We leverage the COCO API from the library *pycocotools* to facilitate standardized evaluation. We measure $mAP@50 = 49.5\%$ for COCO, and $mAP@50 = 40.7\%$ for MetaGraspNet. These results are consistent with the expected performance of the OWL-V2 model, confirming its state-of-the-art capabilities in one-shot object detection.

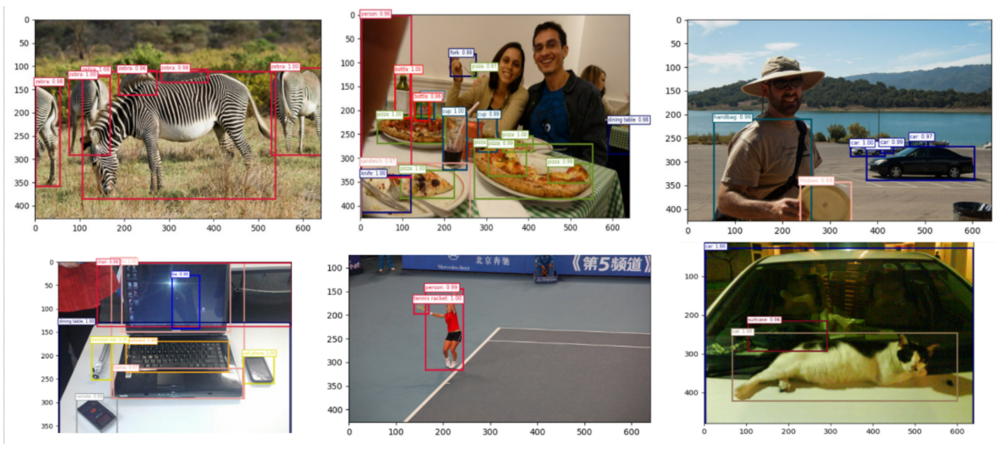


Figure 6.1: One-Shot Object Detection on COCO.

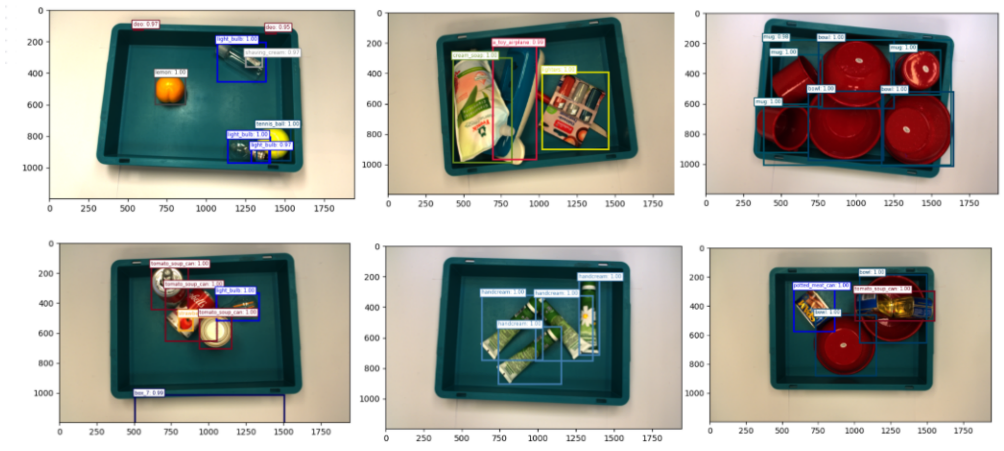


Figure 6.2: One-Shot Object Detection on MGN.

6.3.2 Confidence Threshold Tuning

To further enhance detection performance, we refine the confidence threshold used for detection. The default confidence threshold often does not yield optimal results across all object categories, so we implement a class-specific threshold tuning approach. By optimizing this threshold individually for each category, we aim to improve the trade-off between precision and recall, ultimately boosting overall detection effectiveness.

A key tool in this optimization process is the Precision-Recall (PR) curve, which provides a graphical representation of the trade-off between precision (the proportion of true positive detections among all positive predictions) and recall (the proportion of true positives detected among all ground-truth instances). The PR curve is particularly useful in object detection tasks as it illustrates how changes in the confidence threshold affect model performance.

To quantify this trade-off, we compute the F1-score, a widely used metric that balances precision and recall. The F1-score is defined as:

$$F1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{6.1}$$

By identifying the confidence threshold that maximizes the F1-score for each category, we ensure an optimal balance between correctly identifying objects (recall) and minimizing false positives (precision). This approach enables us to fine-tune our model's decision boundary, improving detection reliability and robustness across different object classes.

Below are the optimal thresholds found for each category:

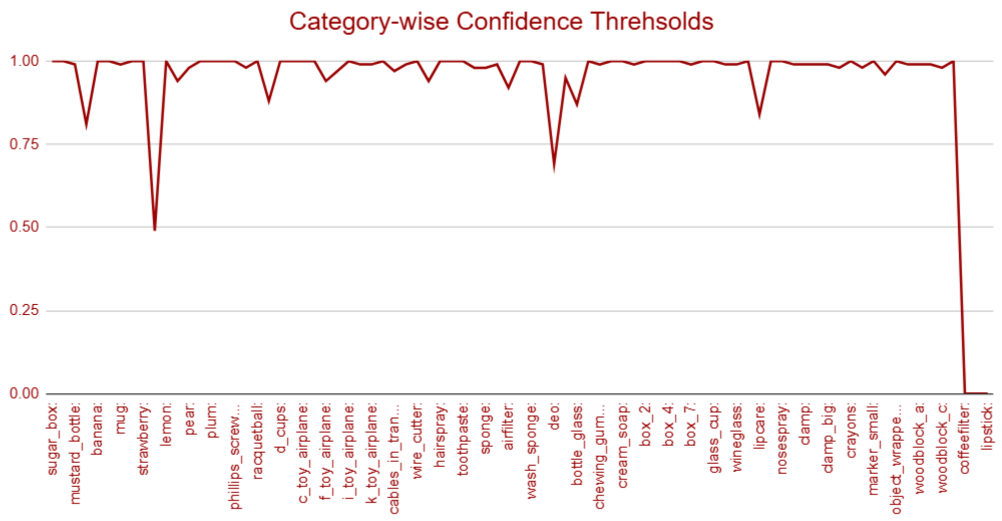


Figure 6.3: Per-Category Thresholds that Maximize the F1-Score.

By leveraging the F1-score-based confidence threshold optimization, we refine the model's detection settings to maximize real-world performance. This adaptive thresholding strategy enhances object detection accuracy beyond the baseline mAP scores, ensuring that the OWL-V2 model operates at its full potential across diverse datasets and object categories.

We demonstrate the improved performance:

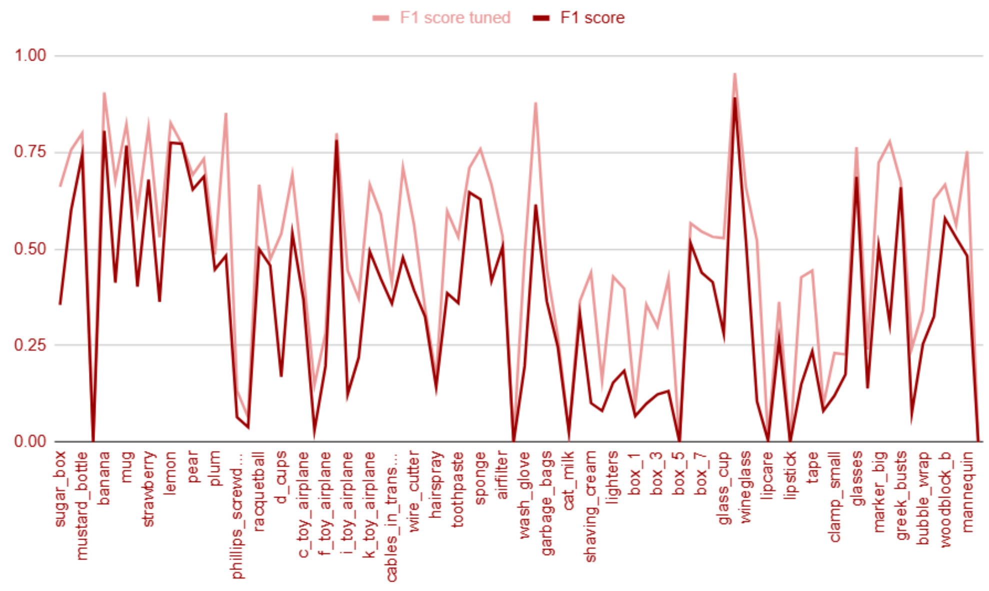


Figure 6.4: F1-Score Before and After Tuning the Confidence Thresholds.

6.3.3 5-Shot vs. 1-Shot Performance

To explore the potential enhancement in detection performance with increased contextual information, we conducted 5-shot detection experiments on the Meta-GraspNet dataset. For each of the 80 classes, five diverse examples showcasing the object from varying perspectives were provided. This approach yielded a considerable improvement, increasing the $mAP@50$ from **40.7%** (1-shot) to **43.7%** (5-shot).

A more detailed per-category analysis revealed that certain object classes with challenging semantics exhibited a significant performance boost under the 5-shot setting. For example, the category `"toydog"` showed an increase from **49.2%** to **86.1%**, suggesting that additional visual context significantly aids in distinguishing complex objects. Conversely, for well-defined objects such as `"banana"`, the improvement was marginal, indicating that the baseline one-shot approach was already sufficient for clear categories. This demonstrates that the effectiveness of few-shot learning varies depending on the complexity and visual ambiguity of object categories.

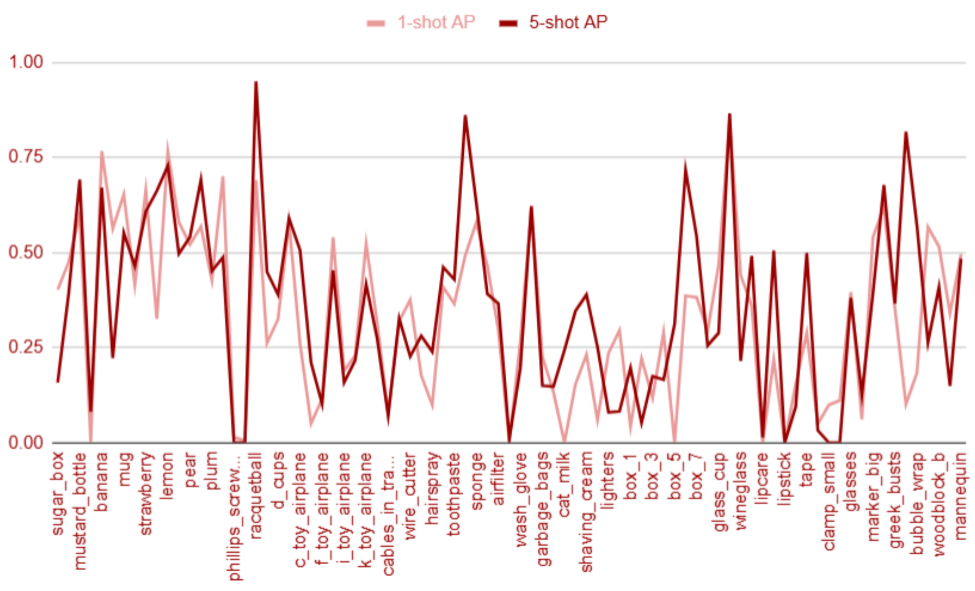


Figure 6.5: 5-Shot vs 1-Shot Performance.

6.4 Qualitative Evaluation

Beyond benchmark datasets, we assessed the system's performance in real-world settings. Two key qualitative experiments were performed:

6.4.1 Controlled Laboratory Environment

The system was tested in a controlled laboratory setting, where objects were arranged under varying conditions such as different lighting angles, occlusions, and backgrounds. These environmental factors were intentionally varied to evaluate the robustness of the system in real-world scenarios. Despite these variations, the results demonstrated that the system was able to effectively identify objects, underscoring its resilience to changes in lighting and background complexity, as well as its ability to handle partial occlusions.

Furthermore, the system was deployed on a physical robot at Comau, the Racer3 robot, where the software's integration with the robotic system was tested. This included evaluating the stability and performance of the server communication, particularly the TCP connection between the robot and the our software. The TCP connection is essential for receiving the query and test images from the robot, and transmitting real-time predicted grasping points and object categories for each object back to the robot. These grasping points are then processed by the robot's manipulation module to guide its object retrieval actions.

In summary, the lab tests not only validated the performance of the object detection system under realistic conditions, but also ensured that the software seamlessly integrated with the robotic hardware, enabling efficient object retrieval even under challenging and dynamic laboratory conditions.

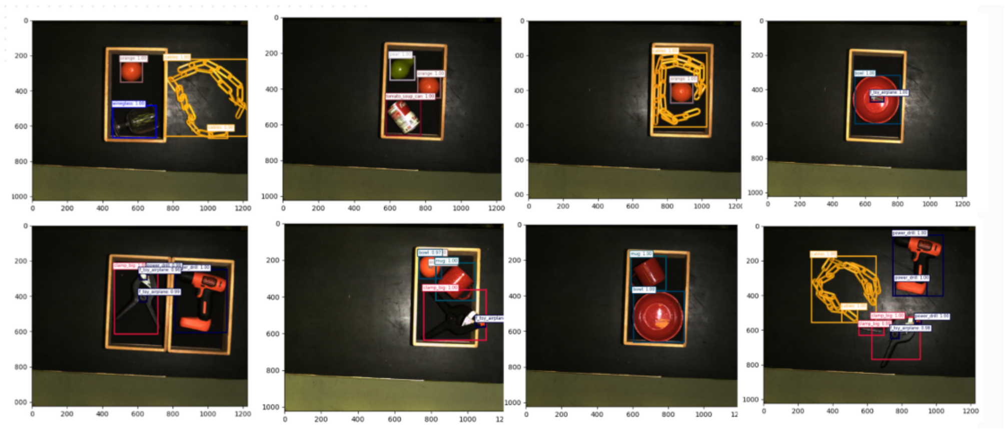


Figure 6.6: One-Shot Object Detection in the Labaratory.

6.4.2 Logo Detection

As a complementary experiment, we performed One-Shot Detection on a subset of the **Logos in the Wild** dataset. We conducted a qualitative evaluation of the visualized predictions to assess whether our system could effectively adapt to brand logos it had never encountered during pretraining.

The results were highly satisfactory, validating that our proposed framework is well-suited for yet another application: verifying that products on a conveyor belt are branded. This demonstrates its potential for automated brand verification and brand presence verification in industrial workflows.

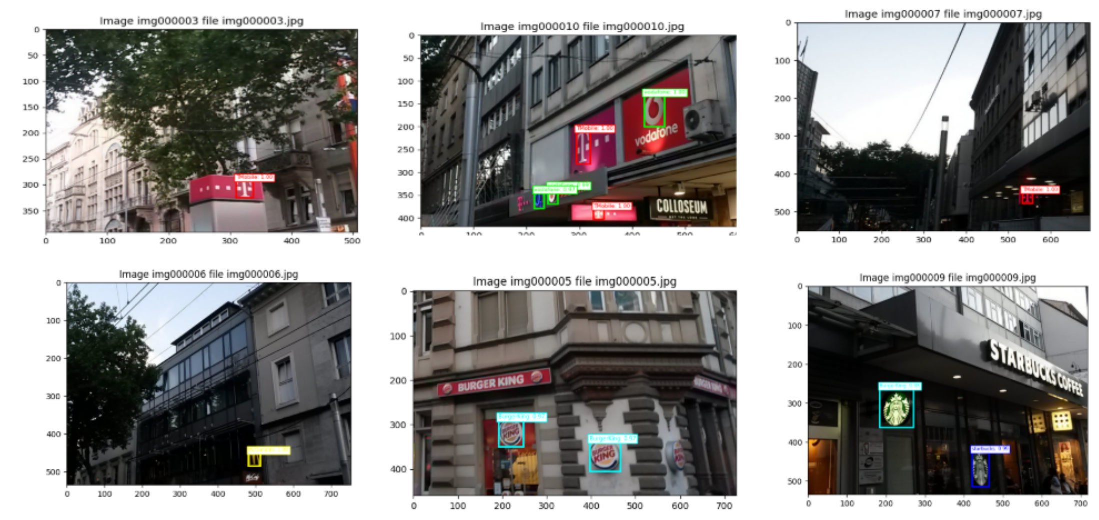


Figure 6.7: One-Shot Object Detection on Logos

6.4.3 Anomaly Detection

Another important industrial application of our framework is anomaly detection, a crucial task in quality control and defect identification. To assess the system's adaptability to this problem, we conducted experiments on a subset of the **MVTec Anomaly Detection (MVTecAD)** dataset, which comprises high-resolution images of industrial objects with various types of real-world defects.

Our qualitative evaluation revealed that the system successfully localized anomalies across different object categories, even without task-specific fine-tuning. The results were **remarkably promising**, demonstrating the robustness of our framework in detecting deviations from normal patterns. Moreover, fine-tuning the model on domain-specific defect distributions could further enhance its performance, enabling precise defect classification and detection.

These findings reinforce the versatility of our approach, showcasing its potential for automated visual inspection in manufacturing environments, where early defect detection is essential for maintaining production quality and minimizing waste. s

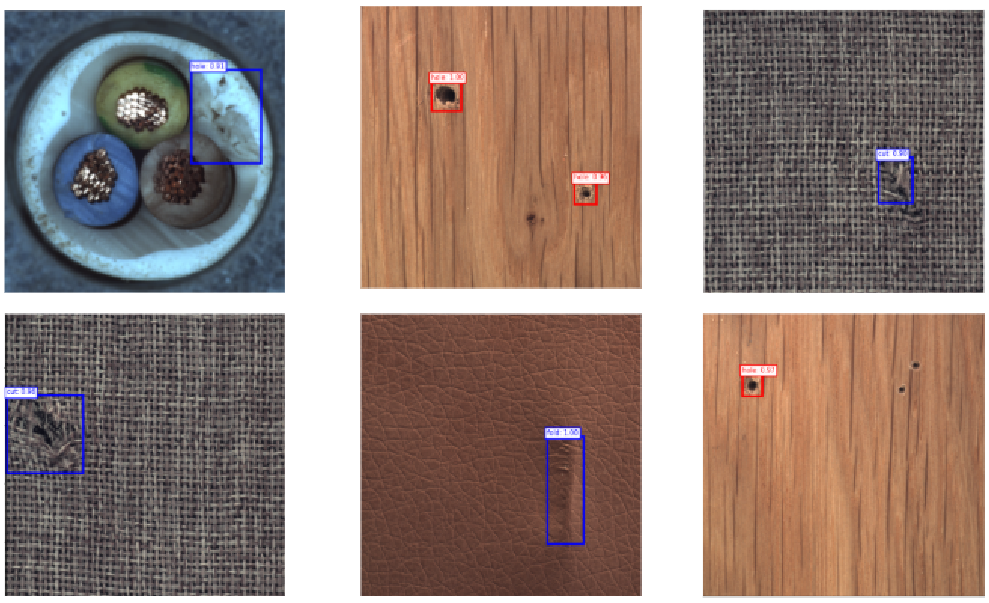


Figure 6.8: One-Shot for Anomaly Detection

6.4.4 Bin Picking and Depalletizing Applications

To further validate the practical applicability of our proposed solution in real-world industrial scenarios, we conducted experiments on bin picking and depalletizing tasks in **Comau's Robolab**. These tasks, fundamental to industrial logistics and warehouse automation, require precise object recognition and localization to ensure efficient robotic grasping.

Our system demonstrated remarkable reliability in these experiments, successfully identifying and retrieving all objects within the test environment. Despite the inherent challenges of occlusions, varying lighting conditions, and diverse object geometries, the model exhibited robust generalization capabilities, effectively guiding the robot in real-time. and adaptability are paramount.

Videos of the experiments can be accessed at the following link: Demo

These results further reinforce the potential of our framework in industrial robotics, paving the way for its integration into autonomous manufacturing and logistics pipelines.

6.5 Efficiency and Performance

One of the key strengths of the system lies in its efficiency and speed. Running on an HP ZBook Power workstation with an Intel Core i7-13800H CPU, 32GB RAM, and an NVIDIA RTX A1000 (6GB VRAM), the system achieves an inference time of $1.7 seconds$ per image, making it suitable for time-sensitive industrial applications. While optimized for GPU acceleration, it remains adaptable to CPU-only execution, although with increased processing time, with about $6 seconds$ per image.

Chapter 7

Conclusion

In this thesis, we have presented a novel one-shot image-conditioned object detection framework leveraging the state-of-the-art OWLv2 model. Addressing the limitations of conventional object detection methods, our approach surpasses the need for extensive dataset annotation and retraining by enabling real-time detection from a single reference image. This capability significantly enhances the adaptability and scalability of object detection in dynamic and data-scarce environments, particularly in industrial applications.

Through rigorous experimentation, we have demonstrated that our detection pipeline achieves competitive performance on benchmark datasets, including COCO and MetaGraspNetv2, with mAP@50 scores of 49.5% and 40.7%, respectively. Furthermore, our analysis of few-shot learning scenarios confirms that incorporating multiple query images improves detection accuracy, particularly for semantically complex object categories. The system's qualitative assessments further validate its efficacy in real-world environments, reinforcing its suitability for automation in industrial robotics, brand verification, and anomaly detection.

The deployment of our detection pipeline via a TCP server ensures seamless integration with industrial workflows. The architecture's flexibility allows for dynamic adaptation, including the incorporation of novel object classes without necessitating server restarts. Moreover, the system's efficiency, with an inference time of 1.7 seconds per image on GPU-accelerated hardware, confirms its applicability in time-sensitive scenarios.

Overall, the findings of this research substantiate the potential of transformer-based architectures for object detection, reaffirming OWLv2's capabilities in facilitating rapid, accurate, and scalable detection. Our contributions pave the way for more autonomous and intelligent vision-based systems, marking a step forward in the evolution of industrial and robotic perception technologies. The combination of high detection accuracy, adaptability to any dataset, and efficient inference time positions our approach as a robust solution for one-shot and few-shot object detection in practical, real-world scenarios.

Chapter 8

Future Work

Building upon our current framework, several enhancements can be introduced to further improve the system's adaptability, performance, and usability across various real-world applications.

One key direction for future work is the integration of segmentation-based techniques to compute optimal grasping points. By leveraging segmentation masks, the system could more effectively analyze object geometries, including non-convex shapes, and determine the most suitable grasping configurations. This would enhance the grasping accuracy and reliability, particularly in unstructured environments where object shapes and orientations vary significantly.

Another valuable improvement is the development of an intuitive and user-friendly interface for fine-tuning the network on task-specific data. This feature would allow users to adapt the model to specialized settings by incorporating domain-specific examples, ultimately enhancing detection and grasping performance in unique industrial or robotic applications. Simplifying the fine-tuning process would make the framework more accessible to non-experts while maintaining the flexibility to refine the model for improved precision and robustness.

Additionally, introducing an abstraction layer to the framework would provide greater modularity and flexibility. Such a layer would decouple the pipeline from the core model, enabling seamless integration of different object detection architectures without altering the underlying process, parameters, or functionalities. This would facilitate experimentation with newer or more advanced models as they become available, ensuring the framework remains adaptable to evolving technological advancements.

By implementing these improvements, the system could achieve greater versatility, ease of customization, and robustness, making it an even more effective solution for real-world object detection and robotic manipulation tasks.

Bibliography

[1] Joseph Redmon and Ali Farhadi. *YOLOv3: An Incremental Improvement.* 2018. arXiv: 1804.02767 [cs.CV]. URL: https://arxiv.org/abs/1804.02767 (cit. on p. 1).

[2] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. *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 (cit. on pp. 1, 8).

[3] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. *You Only Look Once: Unified, Real-Time Object Detection.* 2016. arXiv: 1506.02640 [cs.CV]. URL: https://arxiv.org/abs/1506.02640 (cit. on pp. 1, 9).

[4] Nicolas Carion, Francisco Massa, Gabriel Synnaeve, Nicolas Usunier, Alexander Kirillov, and Sergey Zagoruyko. *End-to-End Object Detection with Transformers.* 2020. arXiv: 2005.12872 [cs.CV]. URL: https://arxiv.org/abs/2005.12872 (cit. on pp. 1, 10).

[5] Da-Wei Zhou, Qi-Wei Wang, Zhi-Hong Qi, Han-Jia Ye, De-Chuan Zhan, and Ziwei Liu. *Class-Incremental Learning: A Survey.* Dec. 2024. DOI: 10.1109/tpami.2024.3429383. URL: http://dx.doi.org/10.1109/TPAMI.2024.3429383 (cit. on p. 1).

[6] C.P. Papageorgiou, M. Oren, and T. Poggio. «A general framework for object detection». In: *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271).* 1998, pp. 555–562. DOI: 10.1109/ICCV.1998.710772 (cit. on p. 4).

[7] P. Viola and M. Jones. «Rapid object detection using a boosted cascade of simple features». In: *Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. CVPR 2001.* Vol. 1. 2001, pp. I–I. DOI: 10.1109/CVPR.2001.990517 (cit. on p. 4).

[8] Luis Arreola, Gesem Gudiño, and Gerardo Flores. *Object recognition and tracking using Haar-like Features Cascade Classifiers: Application to a quad-rotor UAV*. 2019. arXiv: 1903.03947 [cs.RO]. URL: https://arxiv.org/abs/1903.03947 (cit. on p. 4).

[9] 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 (cit. on p. 5).

[10] Pedro Felzenszwalb, David McAllester, and Deva Ramanan. «A discriminatively trained, multiscale, deformable part model». In: *2008 IEEE Conference on Computer Vision and Pattern Recognition*. 2008, pp. 1–8. DOI: 10.1109/CVPR.2008.4587597 (cit. on p. 5).

[11] Chinmoy Borah. *Evolution of Object Detection*. Accessed: 2024-12-26. 2020. URL: https://smarttek.solutions/blog/object-detection-technology/ (cit. on p. 6).

[12] Yibo Sun, Zhe Sun, and Weitong Chen. «The evolution of object detection methods». In: *International Federation of Automatic Control* 133 (July 2024) (cit. on pp. 6–11).

[13] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. *Rich feature hierarchies for accurate object detection and semantic segmentation*. 2014. arXiv: 1311.2524 [cs.CV]. URL: https://arxiv.org/abs/1311.2524 (cit. on p. 8).

[14] Ross Girshick. *Fast R-CNN*. 2015. arXiv: 1504.08083 [cs.CV]. URL: https://arxiv.org/abs/1504.08083 (cit. on p. 8).

[15] Tsung-Yi Lin, Piotr Dollár, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. *Feature Pyramid Networks for Object Detection*. 2017. arXiv: 1612.03144 [cs.CV]. URL: https://arxiv.org/abs/1612.03144 (cit. on p. 8).

[16] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. «SSD: Single Shot MultiBox Detector». In: *Computer Vision – ECCV 2016*. Springer International Publishing, 2016, pp. 21–37. ISBN: 9783319464480. DOI: 10.1007/978-3-319-46448-0_2. URL: http://dx.doi.org/10.1007/978-3-319-46448-0_2 (cit. on p. 9).

[17] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. *Attention is All You Need*. California, USA: Google, 2017 (cit. on pp. 10, 16, 18, 20, 21).

[18] Alexey Dosovitskiy et al. «An Image is Worth 16x16 Words: Transformers for Image Recognition At Scale». In: *Google Reasearch* (2020) (cit. on pp. 11, 22, 23, 28, 29).

[19] Yanghao Li, Hanzi Mao, Ross Girshick, and Kaiming He. *Exploring Plain Vision Transformer Backbones for Object Detection*. 2022. arXiv: 2203.16527 [cs.CV]. URL: https://arxiv.org/abs/2203.16527 (cit. on p. 12).

[20] Archit Parnami and Minwoo Lee. «Learning from Few Examples: A Summary of Approaches to Few-Shot Learning». In: (2022) (cit. on pp. 12, 26).

[21] Zhimeng Xin, Shiming Chen, Tianxu Wuand Yuanjie Shao, Weiping Ding, and Xinge You. «Few-Shot Object Detection: Research Advances and Challenges». In: (2024) (cit. on p. 13).

[22] Ting-I Hsieh, Yi-Chen Lo, Hwann-Tzong Chen, and Tyng-Luh Liu. *One-Shot Object Detection with Co-Attention and Co-Excitation*. 2019. arXiv: 1911.12529 [cs.CV]. URL: https://arxiv.org/abs/1911.12529 (cit. on p. 14).

[23] Camilo J. Vargas, Qianni Zhang, and Ebroul Izquierdo. *Joint Neural Networks for One-shot Object Recognition and Detection*. 2024. arXiv: 2408.00701 [cs.CV]. URL: https://arxiv.org/abs/2408.00701 (cit. on p. 14).

[24] Robin M. Schmidt. *Recurrent Neural Networks (RNNs): A gentle Introduction and Overview*. 2019. arXiv: 1912.05911 [cs.LG]. URL: https://arxiv.org/abs/1912.05911 (cit. on p. 16).

[25] Christian Bakke Vennerød, Adrian Kjærran, and Erling Stray Bugge. *Long Short-term Memory RNN*. 2021. arXiv: 2105.06756 [cs.LG]. URL: https://arxiv.org/abs/2105.06756 (cit. on p. 16).

[26] Matthias Minderer, Alexey Gritsenko, and Neil Houlsby. «Scaling Open-Vocabulary Object Detection». In: *Google DeepMind* (2023) (cit. on pp. 28, 31–33).

[27] Matthias Minderer et al. «Simple Open-Vocabulary Object Detection with Vision Transformers». In: *Google Research* (May 2022) (cit. on p. 29).