Masaryk University
Faculty of Informatics

Detecting and Extracting Information from Images on the Web

Master's Thesis

Bc. Martin Galajda

Brno, Fall 2019

Masaryk University
Faculty of Informatics

Detecting and Extracting Information from Images on the Web

Master's Thesis

Bc. Martin Galajda

Brno, Fall 2019

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. Martin Galajda

Advisor: doc. RNDr. Pavel Matula Ph.D.
Consultant: Mgr. Zdeňka Šeděnka

Acknowledgements

I would like to express my sincere gratitude to the thesis supervisor doc. Pavel Matula, Ph.D. for all of his valuable advice. I am also grateful for the advice and guidance I received from my thesis consultant Mgr. Zdeňka Šeděnka. Moreover, my thanks go also to Gauss Algorithmic company which dedicated some resources to help me with my thesis.

Abstract

New advancements in computer vision have been successfully applied to solve various problems. This thesis is concerned with applying new techniques to extract useful information from the images present on the web to support targeted advertising. The thesis contains research on state-of-the-art techniques for object detection, which leverage deep learning. The researched techniques are implemented and compared on available public datasets. Moreover, they are compared on the dataset of images collected from the web sites on which targeted advertising is used. The collected dataset provides new research opportunities for detecting which images on the web pages are relevant for advertising.

Keywords

deep learning, computer vision, convolutional neural networks, machine learning, object detection, image classification, targeted advertising

Contents

List of Tables

List of Figures

Introduction

In the last decade, there has been a lot of research in the area of computer vision that was concerned with classifying and localizing objects in images [1] [2] [3]. The area has seen many successes due to the rise of deep learning [4]. The applications of new advancements include optical character recognition, face detection, identity verification, self-driving cars, and many more. One particular application that will be the topic of this thesis is extracting useful information from the images on the web to support targeted advertising.

At the time of writing, approximately 4.4 billion people are using the internet and presumably browsing World Wide Web [5]. Many websites that people visit include images containing information that is important for the visitors. However, not all images are relevant to the content of a website, such as images containing advertisements or background images. The thesis will be concerned with tackling the problem of distinguishing important images in the context of a web page and extracting useful information from them.

Our motivation for analyzing images on the web is to support targeted advertising. Targeted advertising is a popular form of online advertising that targets audiences with certain traits, based on the product or service the advertiser is promoting [6]. These traits can be demographic, which consist of attributes like race, economic status, and age. Alternatively, they can be psychographic based on attributes like the consumer's personality, values, and interests. This kind of advertising reduces wastage since only consumers who are likely to be interested in product or service will receive the advertisement. The thesis will aim to support targeted advertising by discovering consumer's interests using images from the visited web pages. It has become relatively common to track which websites users visit using browser cookies. Predicting user interests is then achieved by extracting various information from the visited pages such as text, tags, and structure of the uniform resource locator (URL). However, some web pages also contain images that can be used for analyzing the content of a visited web page.

The thesis is done in collaboration with Gauss Algorithmic [7], a company providing cutting-edge services in the field of big data

analysis, machine learning, and predictive analysis. The company is already tracking users on websites and provides targeted advertising based on the textual content on the websites. However, the problem that the company faces is that some domains serve web pages containing documents that do not contain too much text, or information that is present in images is not contained in the text. The data in the form of images present on the websites is not utilized. One of the goals of the thesis is to identify the domains where analyzing images can help with advertising.

After identifying the relevant domains, there is still a need for solving the problem of distinguishing important images on the websites. This can be a challenging research problem in itself. The problem is that there is no available dataset containing annotations for important images in the web page. A dataset like that could be used for finding patterns or rules that could identify those important images. Therefore, the goal of the thesis will also be to build a dataset that could help tackle this problem in the future.

One of the core goals of the thesis will be researching computer vision methods for extracting meaningful information from the images. The thesis will explore state-of-the-art techniques in this area. It will include the necessary background and research that has been done in the past. Applying researched computer vision techniques will require an appropriate dataset. Therefore, the research into available public datasets will also be part of solving the problem. In order to report results and compare different models, the thesis will discuss appropriate metrics that are used to benchmark techniques. The thesis will then compare researched techniques using these metrics on the available public datasets.

However, since the primary goal of the thesis is to help the company to extract meaningful information from the context of the web pages that they use for advertising, metrics will also be evaluated on the dataset that will be built as part of the thesis. The primary goal of the thesis will be to compare the performance of the techniques on this dataset.

One of the practical outcomes of the thesis will consist of the inference pipelines for detecting objects located on the image. As a result, the implementation should provide an easy off-the-shelf inference that the company can use for targeted advertising.

The rest of the thesis is structured as follows. In Chapter 1, the thesis contains a theoretical background concerning machine learning, computer vision, and deep learning. Chapter 2 contains the research of literature about metrics used for evaluating object detection techniques and about the state of the art techniques for object detection. Chapter 3 discusses public datasets with annotations for object detection. Chapter 4 presents problems and methods concerning the process of building our own dataset of important images. Implementation of the state of the art object detection techniques is the content of Chapter 5. Chapter 6 presents the results from the publicly available benchmarks and from our own experiments that compare the implemented object detection methods. In Chapter 7, the thesis contains discussions about the collected dataset and comparison results. Chapter 8 concludes the thesis with key findings and a summary of what was achieved.

1 Background

1.1 Computer vision

Computer vision is an interdisciplinary scientific field that is concerned with collecting, processing and analyzing digital images. The high-level goal of computer vision is to extract useful information from images [8]. The field combines knowledge and ideas from multiple fields such as physics, biology, psychology, computer science, mathematics, and engineering [4]. It has become an interesting and challenging research field in the last decades. The true challenge proved to be solving tasks that are easy for people to perform, like detect faces in images, but are extremely hard to describe formally and to automate [9].

From the biological science point of view, the goal of computer vision is to discover computational models of the human visual system. From the engineering point of view, the goal of computer vision is to come up with autonomous systems that could perform some of the tasks that the human visual system can perform [10]. Recently, computer vision algorithms were even able to outperform humans on certain tasks. Identifying indicators for cancer in blood and tumors in MRI scans is one example [11].

Applications of computer vision range from reproducing human visual abilities, such as recognizing faces, to creating entirely new categories of visual abilities [9, p. 452]. As an example for the new visual ability, there has been an research into recognizing sound waves from the vibrations they induce in objects visible in a video [12]. Perhaps most popular applications which are worth mentioning include [13]:

- Optical character recognition - using machines for reading handwritten postal codes or license plate signs
- Machine inspection - inspecting images to enable quality assurance, e.g., using stereo vision with specialized illumination to measure tolerances on aircraft wings
- Medical imaging - finding relationships between various medical images and medical conditions

- Retail - detecting objects for automating checkout lanes
- 3D model building - constructing 3D models from aerial photos, e.g. in Bing maps
- Automotive safety / autonomous driving - using object detection techniques to detect obstacles on the road
- Match move - merging computer-generated imagery with live-action footage, a technique widely used in Hollywood

There are many areas in computer vision, but we will be mostly focusing on understanding images in terms of the objects they contain. Tasks which have been a very active area of research in recent years and are related to detecting objects in the image include:

- Image classification
- Object localization
- Object detection
- Instance segmentation
- Semantic segmentation

In all of the tasks mentioned above, our inputs are made of digital images usually consisting of RGB pixels with values ranging from 0 to 255. The tasks differ in outputs they are supposed to produce. In **image classification**, the objective is to output the list of classes of objects that are contained in the image. For some applications, it might be useful to have a model for deciding binary classification, which tells us, for example, whether an image contains a tumor or not. For some applications, it might be required to do multiclass classification - detecting the presence of more than two classes. It is worth mentioning that when we talk about classification, there are two more types of tasks that we distinguish:

1. Simple (one label) classification task - output is just 1 class, all classes are assumed to be mutually exclusive

2. Multilabel classification task - output is an arbitrary number of classes that are not mutually exclusive

In **object localization**, we are performing a simple (one label) classification task, but we are also predicting the location of the single object in the image. Simply put, we are detecting at most one object associated with one class and one bounding box.

In **object detection**, we are detecting multiple instances of objects. Every object is usually associated with one class and one bounding box.

In **instance segmentation**, we are detecting multiple instances of objects at a pixel level, meaning that every pixel will be associated with some instance of an object (or background). Similarly, in **semantic segmentation**, we are performing classification for each pixel in the image. However, the difference is that we do not differentiate between multiple instances of objects of different classes. For example, in **instance segmentation**, if we have three cats in the image, our goal is to detect three different instances of objects with the class cat. In **semantic segmentation**, our goal is to only output the cat class for each pixel where the cat is located, without differentiating different instances of cats.

The difference between these tasks can be seen in Figure 1.1.

1.2 Machine learning

Machine learning has seen a lot of improvement in the last two decades. There is a number of reasons for that, but probably most relevant is the increase in processing power, memory, and storage capacity of computers [8]. Due to the increase in processing power, the field of machine learning has seen a lot of successes everywhere and has also played a significant role in computer vision research. The reason why machine learning proved to be very popular in all sorts of areas is that it is able to learn programs from data, without being explicitly programmed [14].

Machine learning represents a set of algorithms that are able to learn patterns in data [9, p. 99]. Discovered patterns allow us to build programs for performing various tasks. It proved effective in fields like computer vision because it is hard to come up with a reliable

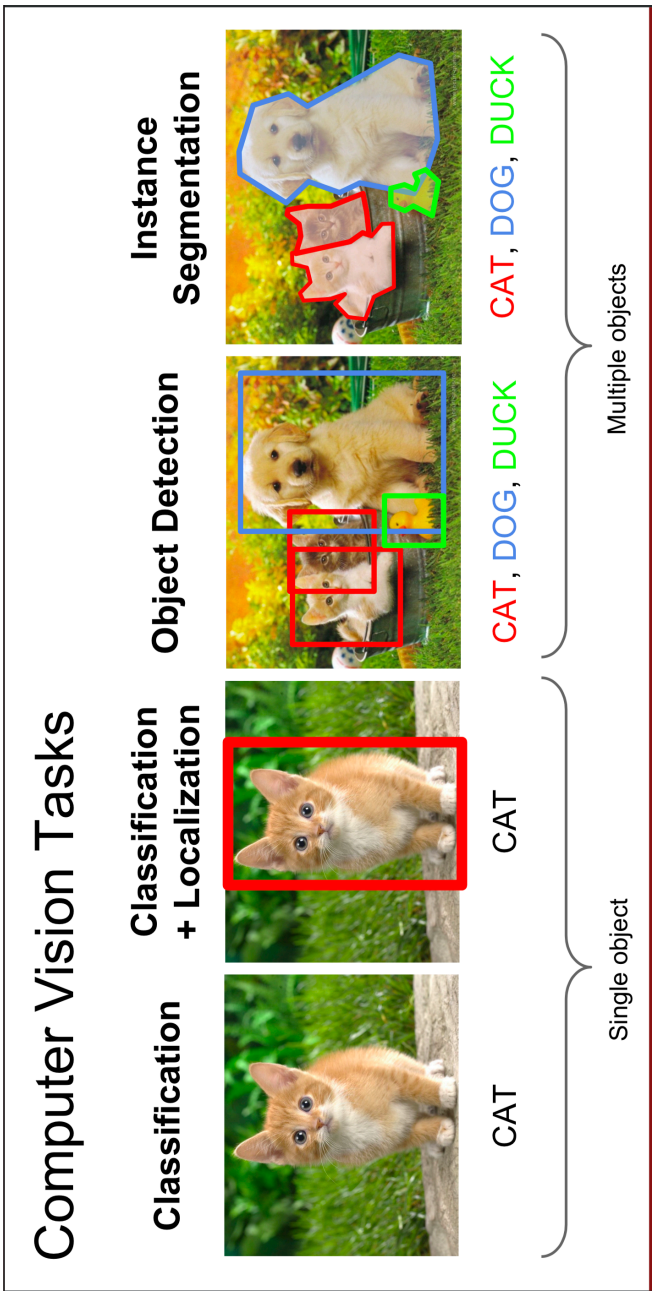


Figure 1.1: Common computer vision tasks (from [4])

algorithm consisting of specific instructions for performing a task [9, p. 99]. Consider, for example, the task of classifying cars in an image. Although the task is easy for a human, it is hard for computers. We know that cars have wheels, so we could design the presence of wheels as one of our features. However, it is fairly complicated to describe the wheel in terms of raw pixels [9, p. 3]. It has a simple geometric shape (circle), but in a real image, the wheel might be occluded or the sun might be glaring off the metal parts of the wheel [9, p. 3]. Moreover, it can be located in different parts of the image. Also, images are usually taken in different illumination conditions - different interactions of light, shadow, and material. This all makes it pretty hard to describe wheels in terms of raw pixels, and consequently, it becomes infeasible to come up with a precise set of instructions that would allow a computer to classify the object in the image as a car.

In the past, most successful computer vision algorithms relied upon hand-crafted features developed by experts in the problem area [15]. This proved to be a limitation as slight variations in data would cause the algorithm to perform poorly. Also, that meant that for different problems, we had to have different feature extractors. Before the emergence of deep learning, it was common to use random forests or support vector machines to perform classification and localization tasks, which usually relied upon these expertly crafted features. The performance of classical machine learning algorithms depends heavily on the appropriate representation and many tasks can be solved with hand-crafted features with satisfactory results. However, in computer vision, this tends to not work as well as some applications require. The reason is that images, which are our inputs, lie in high-dimensional space and "classical" machine learning algorithms do not perform well in the high-dimensional input space since finding appropriate and robust representation is not easy. This is a well-known problem in the field of machine learning known as "curse of dimensionality". Informally, it means that data points in high dimensional space are too far away from each other, making it exponentially harder to separate them reliably. Moreover, an intuition that we develop in 3D space, for example, for models like k-nearest neighbor, does not work in space with a higher dimension [16, p. 36].

Although it is true that images represent high dimensional inputs, pixels from natural images lie on a lower-dimensional manifold

embedded in this high dimensional space [16, p. 173]. By natural images, we mean images that come from the real world. Therefore, one solution to this problem is to use machine learning not just for discovering mapping from the input representation to desired outputs, but also for learning representation itself [9]. This is known as **representation learning** and it allows us to learn important features from the input data automatically. This was partly tackled with the use of unsupervised learning techniques like autoencoders. However, it proved to be insufficient and it has been surpassed by a technique called deep learning. The ability to learn various appropriate representations automatically is the reason why deep learning became so popular recently and is regarded as the state-of-the-art for many vision tasks. The evolution throughout time can be seen in Figure 1.2.

1.3 Deep learning

Deep learning is a subset of machine learning that uses specialized models called deep **neural networks**. We call these models neural networks because their creation was originally inspired by biological neural systems, and they are typically represented by composing multiple functions [9, p. 168]. Deep learning allows us to learn a representation of the world as a nested hierarchy of concepts, with each concept defined in relation to simpler concepts, and more abstract representations computed in terms of less abstract ones [9]. The term "deep" comes from the fact that we construct a computational graph that is organized into many layers that are stacked on top of each other. That means that the networks contain a lot of layers. These layers represent the hierarchy of concepts and allow us to extract appropriate representation from our inputs. In our work, we will be concerned with one particular kind of neural networks called feedforward. However, there are other kinds, such as recurrent neural networks, which are used in different applications.

Feedforward networks, which are also called multilayer perceptrons, are models that are used to approximate some function f. In supervised learning, we are trying to approximate function which maps inputs from our training dataset to target values (e.g., class in image classification).

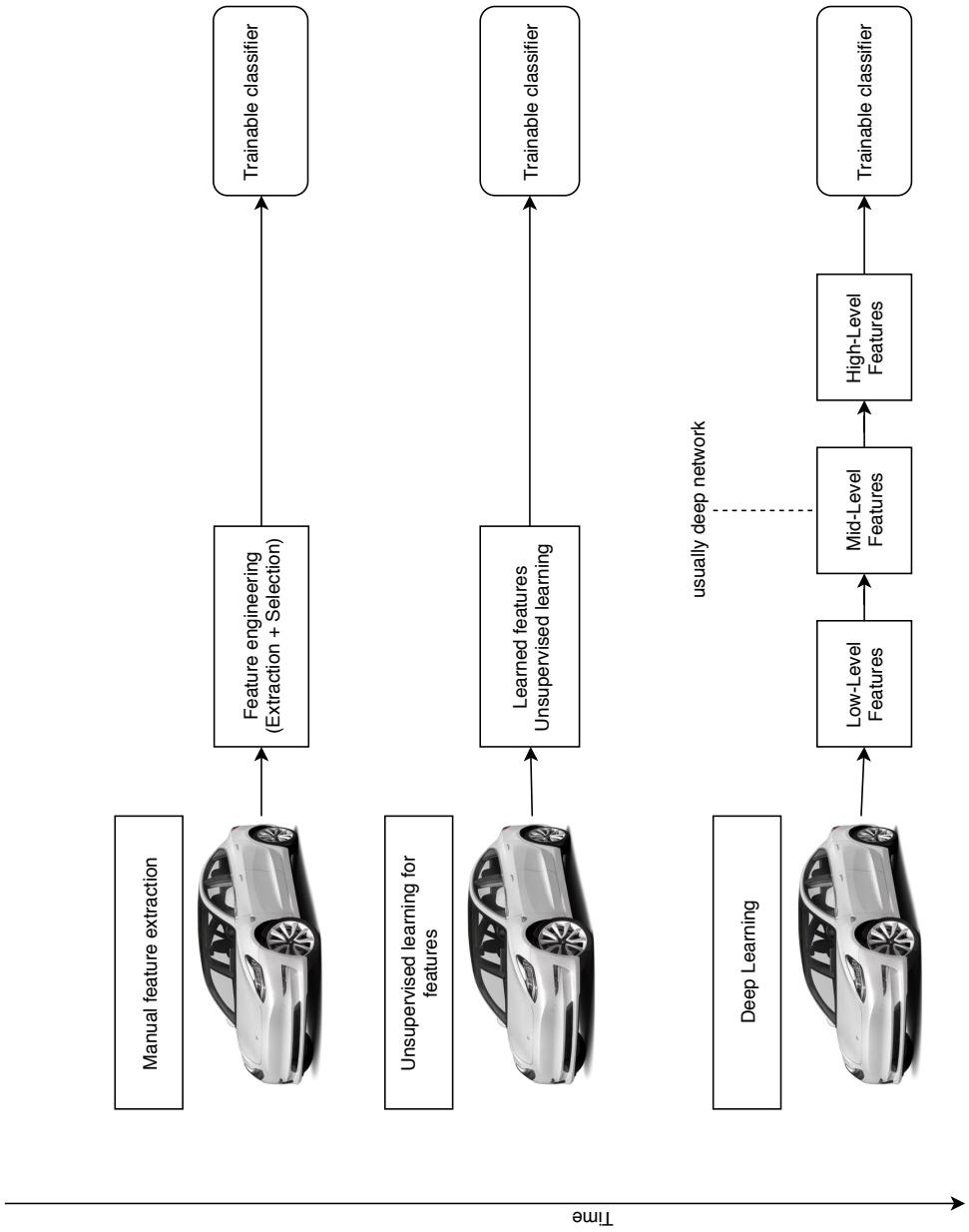


Figure 1.2: Evolution of approaches towards extracting features

The basic unit of computation in neural networks is called a neuron. The reason for the naming is that neural networks have originally been inspired by the goal of modeling biological neural systems. However, after some time, it turned out that trying to mimic biological neurons was not the best decision. Since then, it diverged from the concept of a biological neuron and became more a matter of engineering [4].

Neural networks are organized into layers that are composed of these neurons. **Visible** or **input** layer is the input fed into a network. The layer which is producing outputs is called **output** layer. All layers between the input and output layer are called **hidden** layers. Each neuron takes the output from some preceding layer(s) as its input and computes some function which outputs a value that is fed to the next layer(s). Most commonly, each neuron computes a weighted sum of all of its inputs and applies some non-linear function called **activation** function. Therefore, we can say that each layer computes a function of preceding layers. This organization allows us to build a hierarchy of concepts and learn a representation that explains relationships in the data. For example, in the case of image classification, the first hidden layer can learn to identify edges by comparing the brightness of neighboring pixels. The second hidden layer can learn to represent corners and extended contours as a collection of edges. From the corners and contours, the third hidden layer can learn to detect whole parts of different objects. Finally, the output layer can learn to output object classes by combining parts of the object that the previous layer identified. The whole concept can be seen in Figure 1.3.

The depth of the neural network is usually defined by how many layers it contains. The width of the network is usually determined by how many neurons are present in one layer. In the past, it was very popular to use neural networks with just one hidden layer. The reason was that it had been proved that a 2-layer neural network could approximate any reasonable continuous function [4]. However, it has been discovered later that those networks can be quite wide and require a lot of parameters. Furthermore, due to the required size of those 2-layer networks, training those networks turned out to be impossible for practical applications. By introducing more depth in the architecture, we can exponentially decrease the number of parameters needed meanwhile still maintaining expressiveness. That makes training the network way more practical and it is one the reasons why

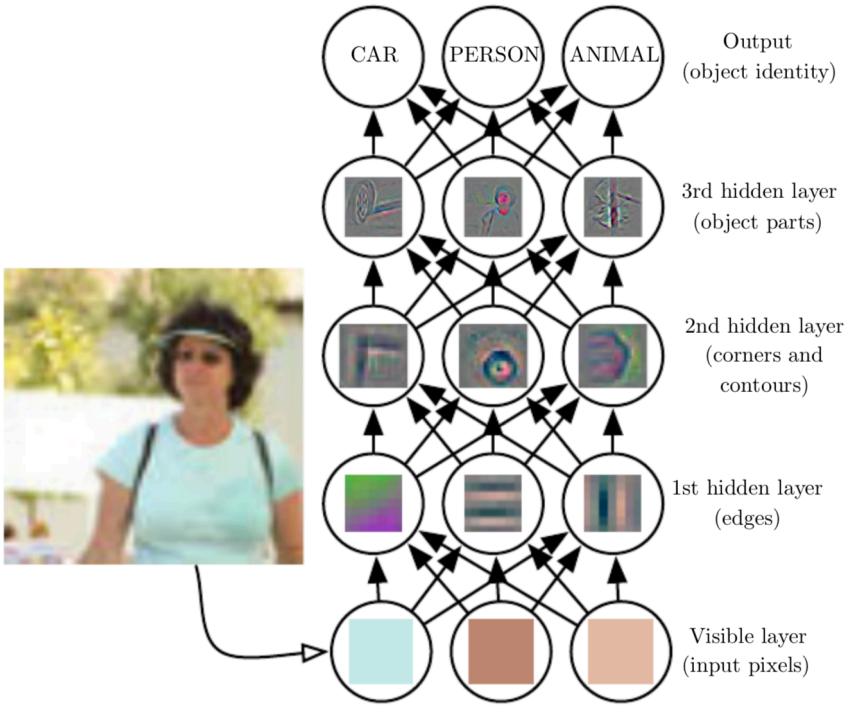


Figure 1.3: Architecture of neural network (from [9])

deep learning became so popular in the last decades. However, it is true that with deeper networks, we have to deal with new problems that arise like vanishing and exploding gradient problem.

1.4 Convolutional neural networks

For computer vision problems, there is one specialized kind of neural network, called convolutional, that proved to be very successful in practical applications. Convolutional networks are networks which use convolution, a mathematical operation which has three desirable properties [9, p. 335]:

- sparse interactions/connectivity,
- parameter sharing,
- equivariancy to some functions.

In the most common fully connected neural networks, we perform multiplication between the whole input to the layer and the weight matrix of the layer. Features in one layer are connected with every feature in the next layer through learnable parameters contained in the weight matrix. Therefore, the size of the layer weight matrix is necessarily tied to the size of the input and output layer. In convolutional neural networks (CNN), we perform convolution operation between arbitrarily big portions of the input to the layer and weight matrices. In convolutional networks, weight matrices in convolutional networks are called kernels. The convolution operation is usually performed on a smaller part of the input. Kernels are, therefore, usually significantly smaller in size than the input layer, which results in **sparse interactions** between inputs and outputs of the layer. That means that each input is connected only with part of the output. This leads to a reduction in memory requirements and improvement in statistical efficiency [9, p. 338]. In the context of computer vision, this is particularly useful since we are able to detect salient features such as edges on images consisting of thousands of pixels only with small kernels. Using deep convolutional networks, which means stacking multiple convolutional layers on top of each other, we can still indirectly interact with a large portion of the input. With deep convolutional networks,

we are then able to describe complicated interactions between many variables by constructing the interactions from simple building blocks that are made only of sparse interactions [9]. This is the reason why deep convolutional networks are able to learn useful representations in the context of computer vision.

Convolutional networks provide us also with an ability to share parameters of the model in more than one function that is computed within the model. In other words, we are using parameters in multiple interactions between different inputs and outputs of the layers. The reason is that we usually perform convolution operation multiple times using different parts of the input but using the same kernel. This also supports the argument that they improve statistical efficiency and reduce storage requirements [9, p. 338], but it also results in another useful property called equivariance to translation. **Equivariancy to some function** means that when the input is changed according to some function, outputs are changed in the same way. This is useful for computer vision since we want to be able to detect objects regardless of their position on the image.

Another notable advantage of convolutional neural networks is that they are able to work with the input of arbitrary size. In our context, this is particularly useful, because we can analyze images that come in different sizes and shapes with one model.

2 Object detection - literature research

One of the core goals of the thesis was to extract meaningful information from the images on the web. As we discussed in Chapter 1, many computer vision methods tackle this problem. We were considering framing the problem as multi-label classification or object detection. The reason for ruling out segmentation was that we did not need pixel-level precision for detecting objects. For solving our problem, we also did not need directly to know the location of the detected object that object detection provides. However, it could be helpful to know how big is the portion of the image that the object is covering. We argue that more important objects would usually take up a bigger portion of the image. That information could be used when building a user profile for targeted advertising. Furthermore, classification models allow us to detect only one instance of the object per image. If the image contains multiple instances of the object, the classifier will not provide us with that information. That is also valuable information. Moreover, the research for object detection was quite successful in recent years, and the classification performance of the object detection models is comparable to the models performing just classification. Therefore, we decided to use object detection as the method for extracting the information.

In this chapter we provide definition for metrics that are used for evaluating object detection models. Moreover, we present research from the literature about techniques that perform object detection. In Chapter 5, we will discuss the implementation of the techniques that are content of this chapter.

2.1 Object detection metrics

Since our goal is to evaluate and compare models for object detection, we needed to research appropriate metrics that will reflect the quality of the models. Our research showed that the popular metric that is used commonly in different public competitions is mean average precision (mAP) [17] [18] [19]. In this section, we will provide a summary of the theoretical background that we gathered from the research of

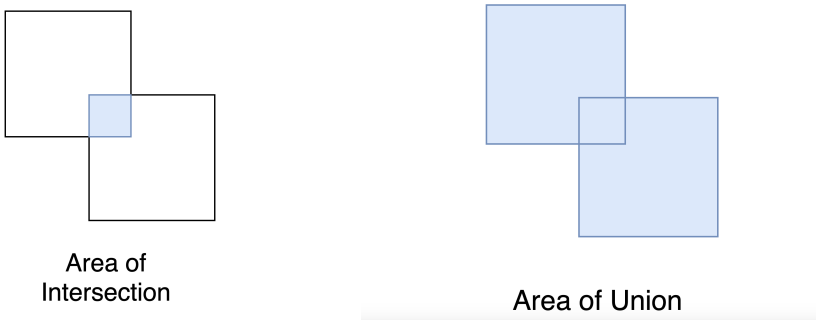


Figure 2.1: Illustration of the intersection and union area used for computing IOU

existing literature. We will describe how this metric is computed and what it represents.

We need to provide definitions for certain terms that are used when we are discussing metrics for object detection. The first one is the intersection over union (IOU). IOU is a score which quantifies how good is the predicted location for a certain object. It is computed between 2 boxes that represent objects in the image. Usually, one box contains a true location for the object that we would like our models to infer, and the second one is the actual box predicted by the model. In order to compute the score, we compute the area of intersection and union of the two boxes. The intersection and union area for the boxes is illustrated in Figure 2.1. The score is then computed as:

$$IOU(\text{true box}, \text{predicted box}) = \frac{\text{area of intersection}}{\text{area of union}} \tag{2.1}$$

The IOU of 0 represents the case when the two boxes do not overlap at all. The IOU of 1 represents the ideal case when the two boxes overlap perfectly.

When we evaluate the object detector, we consider predictions that the model makes on the given image. For evaluation, we need a test dataset consisting of images with manually annotated rectangular boxes specifying the location and the class of the box. We will refer to the boxes that are part of this test dataset as the ground truth (GT). Ideally, we want bounding boxes predicted by object detector to

match the ground truth boxes. We distinguish 2 cases when classifying predictions. The predicted box is considered to be a true positive (TP) if its IOU with the GT box is greater than the specified threshold. Otherwise, it is considered to be a false positive (FP). Moreover, we also consider false negatives (FN) when the model does not detect the ground truth box, which happens when none of the predicted boxes overlap with the GT box with the required IOU. In the object detection task, we usually do not consider true negative (TN), which would mean not detecting certain bounding box, since there are many possibilities for box predictions and rewarding model for not detecting so many of them is not useful.

Precision and recall are important for measuring object detection models since a good detector will have high precision, meaning that a large portion of the predicted boxes will be true positives. It will also have a good recall, which means that a large portion of the ground truth boxes will be detected. Thus it will have a low number of false negatives. In simple terms, precision tells us how good is the model at predicting only relevant objects (by having a low number of FP). Recall tells us how good is the model at finding and detecting all the relevant boxes (by having a low number of FN). Equations (2.2) and (2.3) show how precision and recall is computed.

$$Precision = \frac{TP}{TP + FP} \tag{2.2}$$

$$Recall = \frac{TP}{TP + FN} \tag{2.3}$$

The precision/recall curve is a type of the plot where, on one axis, we have a recall, and on another axis, we have a precision. The curve tells us how is the precision changed when we increase recall. The precision of the good detector will stay high as we increase recall. However, it is fairly hard to tell which model performs better just by looking at the precision/recall curve of different models. Therefore, for evaluation, we usually use composite scores that try to summarize precision and recall, such as F1 score, average precision, or area under a curve (AUC).

In object detection competitions, it is common to use average precision (AP) as the composite score. However, different competitions

provide slightly different definitions for computing this score, and we will use one from PASCAL VOC competition [19]. The Open Images challenge [17] uses definition for computing AP that requires additional annotation metadata. It is based on the PASCAL VOC definition, but the annotators were instructed to mark boxes with special attributes such as whether the box covers more than five instances of the same class, which heavily occlude each other. These attributes are then used in the AP equation for the Open Images challenge. However, since our annotation tool did not have support for annotating dataset with these special attributes, that was not a feasible option for us.

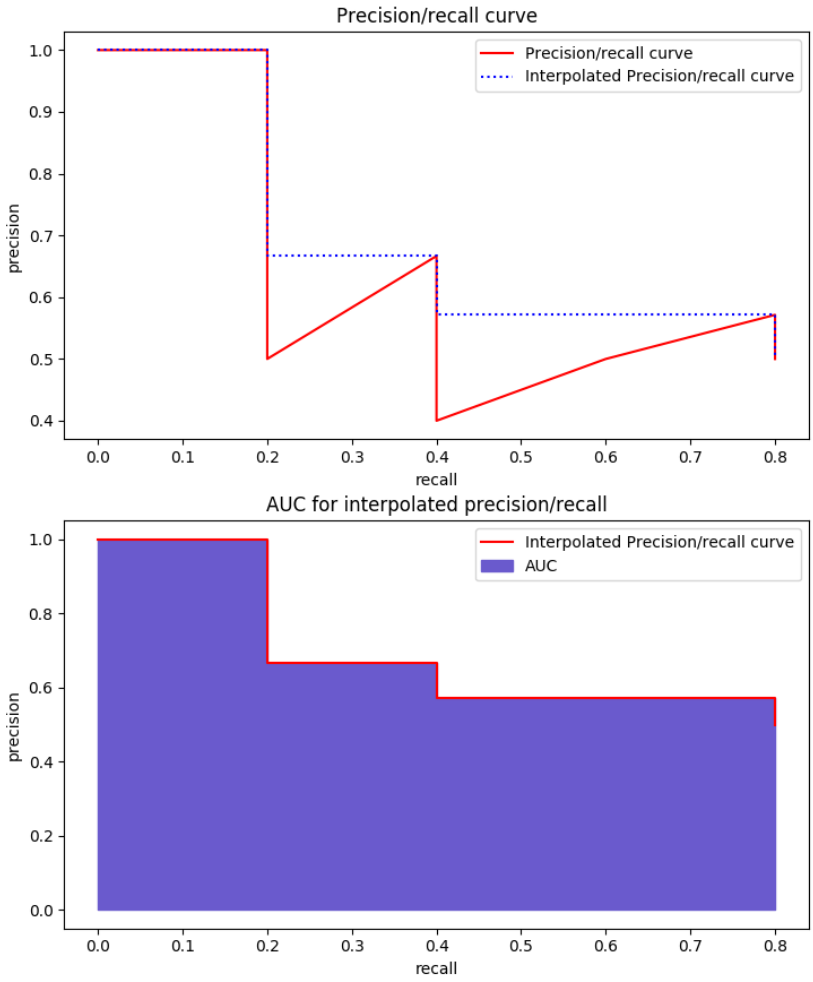
The equation (2.4) shows how AP is computed. We define I as a set of indices for unique recall values. In the computation, we consider all unique recall values, and for each recall value, we compute maximum precision by considering recall values that are equal or greater than the given recall value. Also, for notational convenience, we define $recall_0 = 0$. The interpretation for computing AP is then the area under the interpolated precision/recall curve. Plots illustrating the precision/recall curve and AUC can be observed in Figure 2.2.

$$\begin{aligned} AP &= \sum_{i=1}^{I} (recall_i - recall_{i-1}) * \rho_{interpolated}(recall_i) \\ \rho_{interpolated}(recall_i) &= \max_{\tau:\tau \geq recall_i} precision(\tau) \end{aligned} \tag{2.4}$$

It is important to realize that average precision considers always bounding boxes just for one class. However, since object detectors are usually able to detect more than one label, it is fairly common to use mean average precision (mAP). This score is computed as computing AP for each class that the object detector recognizes and then taking mean of all those APs. The resulting score summarizes precision and recall across all the classes. The equation for computing mAP is shown in (2.5).

$$mAP = \frac{\sum_{c=1}^{C} AP_c}{C} \tag{2.5}$$

where AP_c represents AP considering only bounding boxes (predicted and ground truth) with associated class c.



The upper plot shows the precision/recall curve. The bottom plot shows AUC from the interpolated precision/recall curve which represents AP metric.

Figure 2.2: Precision/Recall and AUC

2.2 History

The field of computer vision, and object detection specifically, has seen a lot of action in the last 15 years. The first important landmark that is worth mentioning is from the year 2005 and was introduced in the paper Histograms of Oriented Gradients for Human Detection [20]. Authors came with an approach that consisted of computing special features, called histograms of oriented gradients (HOG), on each window that was obtained by running the sliding window algorithm on the pyramid representation of the image. After computing the features, they used support vector machines for performing the task of human detection. This was an important landmark because these features proved inexpensive to compute and were practical for various tasks in the area of computer vision.

However, the manual feature extraction has been surpassed by deep convolutional neural networks (CNN) which are able to learn complex representations automatically. Even within the framework of using CNN for object detection, there are two different methods:

- region-based methods,
- regression-based methods.

2.3 Region-based methods

Region-based methods were a natural evolution of HOG methods where we replaced the engineered features with features learned by CNNs that served as feature extractors. However, the problem was that the number of windows generated by the sliding window algorithm was too big for CNN. It was computationally expensive and slow. The problem with the complexity arose because the sliding window approach divides an image into many smaller rectangular images that serve as bounding boxes. We call these generated bounding boxes as region proposals. Those region proposals are run through a classifier that decides what object that part of the image contains (or background). The number of possibilities for diving images into the region proposals is huge, so performing an exhaustive search is impossible. Therefore we have to hand-pick shapes and aspect ratios

that we want to try in each image. Still, to achieve good performance, we have to generate many rectangular areas from the input image and run all of them through a classifier.

The solution, with which the first popular architecture called R-CNN [21] came up, was to use a selective search [22] algorithm to generate region proposals. That decreased the number of generated region proposals and increased their quality. The region proposals were then fed into the CNN as the input. The selective search algorithm uses cues such as similarities in texture, brightness, or colors for finding candidate regions. Region proposals are then candidates for possible bounding boxes, and CNN is responsible for extracting useful features from the proposals. In this architecture, linear support vector machines (SVMs) were used for classifying proposals using features extracted by the CNN. The process of generation of region proposals that specify candidate bounding boxes for objects, and which are consequently fed into a CNN that extracts features for classification, characterizes the family of region-based methods.

After the initial success of R-CNN, it was determined that it is still rather slow for some applications, and the bottleneck turned out to be the generation of region proposals. The next region-based approach that aimed to improve the speed of R-CNN was an architecture that introduced the concept of spatial pyramid pooling within the context of CNN. The authors called this architecture SPP-Net [23]. The main drawback of R-CNN was that it still used CNN that had a fully connected output layer and that required to have a fixed input size. That meant that images that are fed into the CNN had to have a fixed size. Models used techniques like cropping or warping to produce fixed-size input. For example, R-CNN used simple affine warping. Moreover, each region proposal had to be run separately through the whole CNN computation. That means that for 2000 region proposals (original number used in the R-CNN paper), we had to run 2000 computations through the whole CNN. Spatial pyramid pooling, which was introduced in SPP-net, allowed using inputs that were of arbitrary size. Moreover, SPP-net was able to share computations between region proposals since it ran the whole image through a large part of the CNN just once. It then used features from the feature maps for classification of the region proposal. SPP-Net used max-pooling for extracting the portion of the feature map that corresponded with the

location of the region proposal. This resulted in 24-102x speedup over R-CNN [23, p. 2].

The next evolution in region-based methods was Fast R-CNN [24]. It took work that SPP-Net has done and applied it to R-CNN. Moreover, SPP-Net had some problems with training that Fast R-CNN managed to solve better, and that led to an improvement in the mAP. Fast R-CNN introduced a region of interest pooling layer (ROI) that extracts a fixed-size feature vector from the feature map for each proposal. That was something that allowed the model to share computations. Also, similar to SPP-Net, running computation for each region proposal through the whole CNN was no longer needed. Moreover, Fast R-CNN introduced multi-task loss. The network had two sibling output layers: one responsible for the classification and another one for correcting the location of the region proposal via regression. The paper also used a few tricks for training that became popular at the time, such as initialization from the ImageNet pre-trained networks. This all led to significant improvement in speed and mAP [24, p. 2].

The last step in the evolution of region-based methods was Faster R-CNN [2]. Since Fast R-CNN and SPP-Net exposed computation of region proposals as a bottleneck, Faster R-CNN wanted to speed up the generation of region proposals. One reason for the slow computation of the region proposals was that it was implemented only on the CPU. On the other hand, CNNs took advantage of fast GPU computations. Although reimplementation of the region proposal computation on GPU could speed up object detectors, Faster R-CNN decided to take another approach. Instead of selective search, authors decided to use another fully-convolutional neural network for predicting region proposals, which they called region proposal network (RPN). One advantage of using RPN for computing region proposals is that it can share convolutional features with the object detection network, making the generation of region proposals more efficient. RPN was responsible for predicting object bounds for region proposals and objectness of the given region proposal. RPN uses so-called anchor boxes to predict region proposals. Anchor boxes are reference boxes that consist of coordinates for the center point, width, and height of the box. RPN is responsible for predicting whether the given anchor, which is projected on convolutional features, contains an object (objectness). It is also responsible for predicting correction for the location

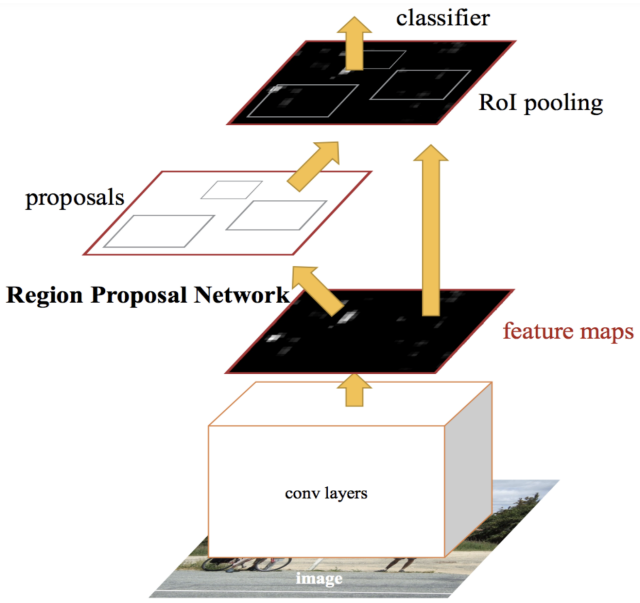


Figure 2.3: Faster R-CNN architecture [29]

of the anchor: how much and in what direction shift the center of the anchor and how to scale width and height to match the predicted box. Region proposals that were predicted by RPN were processed by the ROI pooling layer and fed into Fast R-CNN object detector. The architecture of Faster R-CNN can be seen in Figure 2.3. It is worth noting that there exist different variations of Faster R-CNN architecture based on what CNN is used as a feature extractor. The original paper used VGG-16 [25] and ZF [26], but people used also ResNet [27] or Inception[28]. Choosing underlying feature extractor, commonly called also backbone, represents a trade-off between accuracy and inference speed.

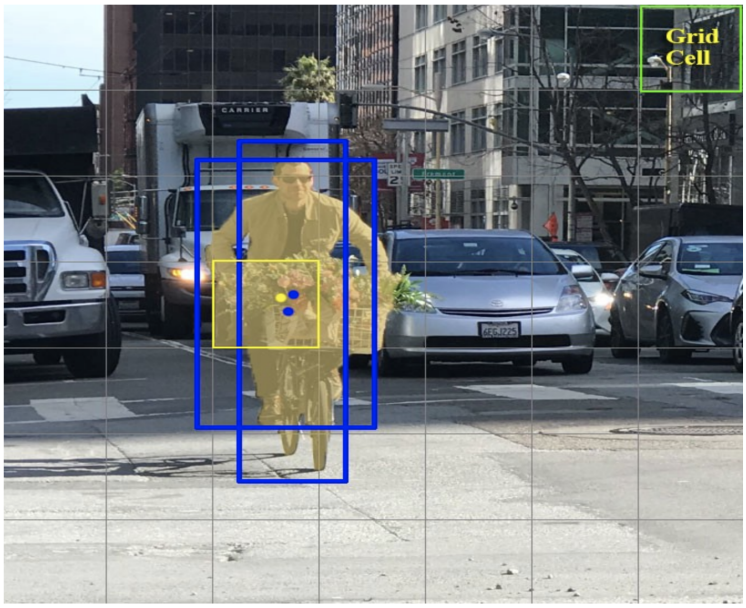
2.4 Regression-based methods

Another popular approach towards object detection consists of framing the problem as a regression problem. We call these methods sometimes as single-shot detectors since they use only one neural network for predicting bounding boxes and class probabilities, taking only

raw image pixels as the input. The whole object detection network is trained end-to-end on detection performance, making it very efficient. One of the most popular architectures, which is mainly known for its incredible inference speed, is called You Only Look Once (YOLO). Authors reiterated on their ideas and it has now three versions. We will discuss briefly the most important specifics about the architecture and how it evolved.

YOLOv1 [1] was a new architecture that achieved an inference speed of 45 frames per second (FPS) on GPU. Unlike region-based methods that used parts of the image as input for detection, YOLOv1 used the whole image as the input. This can be advantageous since the network could learn to encode contextual information about classes. For example, human eyes are almost always present in the context of a human face. Natural images are intrinsically hierarchical, which was also one of the main ideas used in the development of Selective Search [22, p. 1]. Therefore, the context of classes and relationships between them can be helpful, and YOLO could learn to use this information. YOLOv1 architecture had fully connected layers, so the original paper used fixed-size images of size 448 x 448 as input. YOLOv1 divides image into S x S grid. Each grid cell predicts B bounding boxes that consist of 5 predictions: x and y coordinate, width and height and confidence score for the bounding box. The confidence score expresses how likely is the predicted bounding box to contain an object and how accurate the prediction is. During training, a grid cell is responsible for predicting certain ground truth bounding box if the center of the box falls within the cell. Also, since each grid cell predicts multiple bounding boxes with various sizes and aspect ratios, the prediction that has the highest intersection over union (IOU) with ground truth was responsible for predicting the box when computing loss during training. This leads to specialization between predictions since each prediction can learn to detect specific sizes and aspect ratios. The illustration of how grid cells predict bounding boxes can be seen in Figure 2.4.

In the second version of YOLO [31], called YOLO9000, authors wanted to focus on improving localization and recall. The first trick that YOLO9000 used is batch normalization, which allowed authors to get rid of other forms of regularization and led to a 2% improvement in mAP. YOLO9000 got rid of the fully-connected layer, so the network



The image is divided into 7x7 grid cells. We can observe that the person on the image is supposed to be detected by the grid cell that contains the center of the person. Also, we can observe how the cell predicts multiple bounding boxes. Original YOLO used two bounding boxes per each grid cell.

Figure 2.4: Grid structure of YOLO predictions [30]

became fully convolutional. Instead of using direct predictions for bounding boxes, it adopted the approach of Faster R-CNN and used anchor boxes. However, predictions for coordinates of the center of the box for the given anchor in Faster R-CNN are unconstrained and can fall anywhere in the image, meanwhile, YOLO9000 constrains coordinates to fall within the location of the grid cell by using the sigmoid function. Therefore, YOLO9000 uses only the width and height of anchor boxes for making predictions and the center is inferred purely by the network. Since the network is fully-convolutional, authors experimented with multi-scale training, randomly resizing inputs for the network every ten batches. Moreover, authors came up with their own backbone network for extracting features called Darknet-19 [31, p. 4]. Also, authors decided to use passthrough layers, concatenating features from earlier layers with features from the layers deeper in the network with a goal to improve the detection of smaller objects by having more fine-grained features.

YOLOv3 [32] was an incremental improvement and it took ideas that other popular architectures had and applied it to YOLO. YOLOv3 is a little bigger and slower than YOLOv2, but it is more accurate. One new and important thing is that authors decided to frame class prediction as multi-label classification. That means that there is a possibility to detect multiple classes within one bounding box (which is not possible, for example, in Faster R-CNN). Authors specifically claim that this architectural decision helps when moving to more complex domains like Open Images dataset [17] since it contains a lot of overlapping labels. YOLOv3 also predicts boxes across three different scales using a similar concept to feature pyramid networks [33]. Next, the authors introduce a new feature extractor called Darknet-53, which combines ideas from Darknet-19, and from residual networks such as ResNet [27].

3 Available datasets

In order to implement and evaluate models that will be used for extracting meaningful information from the images, we had to choose an appropriate public dataset for our problem. Neural networks are data-hungry models and their quality is affected by the quality and quantity of data that was used for their optimization [9, p. 20]. Therefore, we conducted research with a goal to find an appropriate dataset. Our goal was to find some dataset that will be able to generalize on large-scale since our goal was to detect objects on websites with a great amount of diversity. The appropriate dataset should enable training a model that is able to detect a big amount of diverse objects present in various settings.

3.1 Overview

At the time of writing, there are multiple popular public datasets for the object detection task. However, there are some specialized datasets that are specifically designed for autonomous driving or other specialized domains, and we will not be considering them. We will be concerning ourselves only with large-scale and diverse datasets that can help us tackle the general problem of detecting objects in images on the web. Public object detection datasets are commonly used for competitions to enable evaluation of different models and algorithms developed by researchers.

The oldest high-scale dataset used in the domain of computer vision is **ImageNet** [34], which is a dataset organized according to WordNet [35] hierarchy. Each concept in WordNet, possibly described by multiple words or word phrases, is labeled as a "synonym set" or shorter "synset". ImageNet contained around 14,000,000 annotated images organized by the semantic hierarchy of WordNet as of August 2014. ImageNet populated approximately 22,000 synsets of WordNet with an average of 650 manually verified and full resolution images [36]. Bounding boxes for images are also provided that allows training of object detection models. However, bounding boxes are available just for a subset of 3000 popular synsets. ImageNet claims to have on average 150 images with a bounding box for each synset that is

associated with object detection task [37]. This leads us to believe that they provide approximately 450 000 images for object detection.

Another dataset that was perhaps one of the most important for research in the years 2010-2014 is from the competition called **ImageNet Large Scale Visual Recognition Challenge (ILSVRC)** [36]. It uses ImageNet as a backbone and it has multiple versions from years 2010-2017. It had competitions for different vision tasks such as image classification, object localization, object detection, and scene classification. The most popular is classification dataset containing 1000 categories. In that dataset, each image is associated with exactly one category, so it is a task for simple one-label classification. It has become fairly common to pre-train networks for other tasks (possibly on different datasets) on this ILSVRC classification dataset. The reason is that it contains 1000 diverse categories, so the headless part of the network trained on this dataset can serve as a good feature extractor.

Another challenge that comes with an associated dataset for computer vision tasks is **The Pascal Visual Object Classes (VOC)** [19]. The challenge and evolution of its dataset started in 2005 and ended in 2012. The website of challenge contains multiple datasets from these years. It contains data for classification, instance segmentation, object detection, and for two subsidiary tasks on action classification and person layout. For the object classification and detection, it is annotated with 20 classes of objects. It can be used for training the multi-label classification model as one image is annotated with multiple classes. The dataset for classification and detection contains 11,540 images and 31,561 objects for training.

In 2014, new challenge called **Microsoft COCO: Common Objects in Context** [18] was introduced. The motivation that authors provide for creating this new challenge is that they wanted to create a challenge for improving state-of-the-art object recognition. The challenge aims to achieve its goal by placing the question of object recognition in the context of the broader question of scene understanding. Images present in the dataset are placed within complex scenes with common objects occurring in their natural context. The classification and detection dataset is labeled with 91 different classes.

The last dataset that we discovered for the task of object detection goes by name **Open Imagaas** [38]. It is a large-scale dataset containing 9.2M images, 30.1M image-level labels with 19.8k concepts, 15.4M

bounding boxes with 601 classes. It also includes 375k visual relationship annotations involving 57 classes [17]. The dataset contains an unprecedented amount of annotations for object detection, 15 x more than the next large datasets. The images were collected by annotating real images from Flickr [1] that were identified to have Commons Attribution license. Authors claim that the dataset is more complex and harder than previous datasets because they remove images that appear elsewhere on the internet and that removes "easy" images that appear in search engines. Although the dataset contains 19.8k image-level labels, only about 7.1k classes have enough associated images to be considered trainable. The unique thing about this dataset is the fact that annotations for image-level classification and object detection are provided on the same set of images, making it unified. Also, images are not scraped based on a predefined list of classes, which helps to avoid the initial design bias. The object detection dataset contains a set of 601 diverse classes.

The dataset statistics related to the object detection task are summarized in Table 3.1. We can observe that Open Images contains most bounding box annotations. It is worth noting that statistics provided by ImageNet are not complete and might not be accurate. They claim to have annotated over 3000 synsets with an average of 150 images for each synset, which gives us around 450k annotated images [37]. At the same time, they claim to have around 1M images annotated with bounding boxes which is contradictory with the previous statistic [39]. We did not find statistics about the number of bounding box annotations.

Table 3.1: Comparison of object detection datasets

Type	VOC	COCO	ILSCVR	Open Images	ImageNet
Classes	20	80	200	600	3000
Boxes	27,5k	887k	535k	15,5M	N/A
Images	11,6K	124K	477K	1.9M	1M
Boxes/image	2.4	7.2	1.1	8.1	N/A

1. Online photo management and sharing application `https://flickr.com`.

3.2 Choosing the dataset

We analyzed and compared all the datasets that were available for the object detection task. We decided to use Open Images (v4) for multiple reasons. The most important reason was that it had significantly more images and bounding box annotations for object detection than any other public dataset, making it the largest public dataset for object detection. Also, the fact that all the images have a CC-BY license made them available for commercial usage. The number of classes for detection was also highest which allows models trained on the dataset to detect a fairly broad number of classes. For object detection, it contains annotations for 600 classes that are organized into a hierarchy.

ImageNet also seemed to have a lot of annotated images with bounding boxes, but it is not a competition dataset. Therefore, the dataset does not explicitly contain a test subset of images that should not be used for training, just for evaluating accuracy of the model. That means that researchers are less likely to release models trained on this dataset since there is no easy way to compare the model. Moreover, released statistics on the website of the dataset were 9 years old and that suggested that it is not that actively maintained as Open Images which seemed to have frequent updates in recent years.

4 Building the dataset of important images

One of the goals of the thesis was to build a dataset of important images within the context of web pages that the company uses for targeted advertising. There were two main reasons for building the dataset. The first one was to tackle the problem of detecting important images on the web page. The hypothesis was that the important images could be detected based on their location within the HTML document. The company did not possess the dataset that could be used for finding patterns or rules that identify the important images. Building such a dataset was considered as an important step for solving the problem in the future. Therefore, the dataset needed to contain annotations specifying the location of important images in the HTML. The second reason, which was more related to the topic of the thesis, was the need to evaluate object detection models in our domain of interest. Our goal was to use the models for targeted advertising on the web pages served from the specific domains. Therefore, we needed to compare the quality of the models on the images from those web pages.

4.1 Collecting URLs of relevant web pages

The company provided us with a list of domains that it uses for tracking users for the purposes of targeted advertising. The list was provided in the form of comma-separated values (CSV) file that contained 563 domains. From the list, we were supposed to choose domains that were likely to contain websites where images provide some extra information that is not present in the text. The illustration of the important and unimportant images can be seen in Figure 4.1.

After choosing domains, our goal was to crawl them and collect a dataset consisting of uniform resource locators (URLs) to the web pages. The outcome from this phase was supposed to be a list of page URLs that would get processed in the next phase of collecting important images.

Deciding which domains contain websites with important visual content is a challenging problem and there is a certain subjective bias towards what a person could evaluate as important. Our goal was to minimize this subjectiveness and come up with an approach that

Figure 4.1: Illustration of important and not important images on websites. Unimportant are surrounded by the red rectangle and important are surrounded by the green rectangle.

makes sense at least intuitively. Initially, we reviewed websites on all the domains that we received and tried to filter out domains that were not appropriate. Specifically, we identified certain types of domains where analyzing images does not make sense, like for example e-commerce websites selling products. Those websites usually contain specific textual information about products and images serve just as an illustration for the design of the product. We included information from the expert in the company in our decision to eliminate certain domains.

After we got rid of the domains that did not make sense, we were still left with quite a lot of candidate domains. Our goal was to have a list of approximately 20 domains for evaluation. Therefore, we implemented a web crawler that would crawl the remaining domains and count an average number of images on the web page present on a given domain. Following this approach, we constructed a list of domains ordered by the average number of images on the website on the given domain. Afterward, we manually reviewed domains from the top of the list and picked 24 domains from which we would collect URLs for annotation. We tried to choose domains with different types

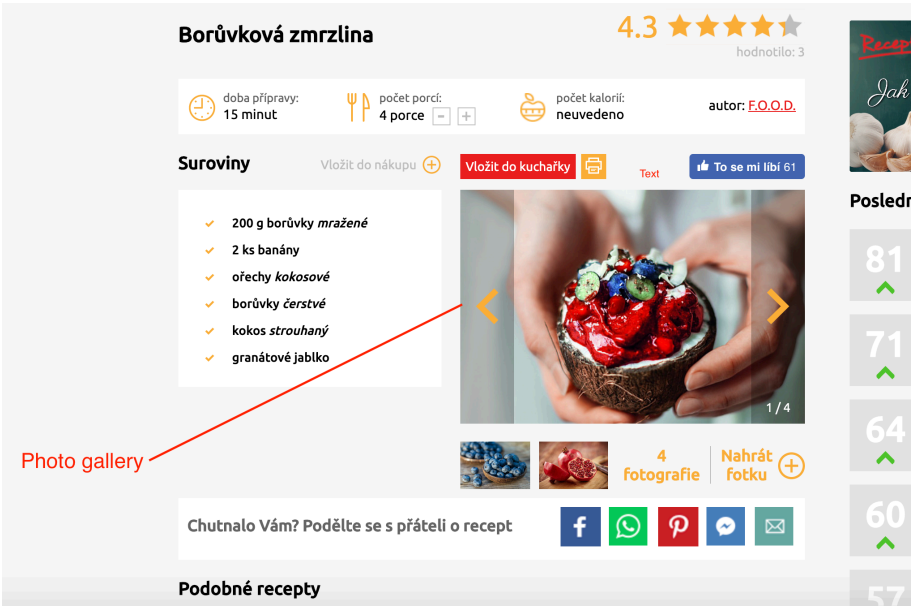


Figure 4.2: Example of a webpage with photo gallery

of content to promote diversity in our evaluation. We chose domains containing web pages related to lifestyle, news, recipes, professional photography, video streaming websites, sport, and entertainment.

After we constructed the list of domains, we still had to crawl them and get a list of URLs for web pages containing relevant images for advertising. It is fairly common for websites to have some pages that are not specific to the reason why a typical user decides to view the website, such as Terms and conditions, About or Contacts page.

One possible approach could be to come up with some mechanism that would blacklist those pages. However, it seemed like coming up with a reliable mechanism for blacklisting all the unimportant pages would take a lot of time since it would require us identifying this kind of web page across 24 domains. The next approach that came naturally to our mind was whitelisting webpages that we know to be interesting for us. For whitelisting, we needed some common denominator for the web page with important images. We found one in the form of photo galleries that are present on web pages. The example can be seen in Figure 4.2.

```
{
    "url": "https://prozeny.blesk.cz",

    "pageWithGalleryContainsAnchorHrefRegexp": "https://prozeny\\.blesk\\.cz/galerie/.*/.*",
    "pageWithGalleryContainsImgSrcRegexp": ".*cdn.dopc.cz.*",
    "pageWithGalleryUrlMatchesRegexp": null,

    "galleryUrlMatchesRegexp": "https://prozeny.blesk.cz/galerie/.+"
},
```

An example of JSON configuration used for detecting pages with photo galleries when scraping the domains for URLs that are suitable for our processing. This JSON is parsed by a crawler, the URL field in this JSON represents substring of URLs for which this configuration should be used. Other fields specify regexp expressions for detecting patterns in the HTML document that identify the page as having a photo gallery. This configuration was created for every domain that was used for crawling.

Figure 4.3: Configuration file for scraping

Therefore, we decided to implement a crawler that crawled selected domains. The crawler collected URLs referencing websites where the photo gallery was present. Another problem that we had to solve was detecting the presence of a photo gallery on a given website. We looked at the structure of HTML documents on the domains and we found out that for each website, we could come up with some combination of HTML tags and attributes that would uniquely identify page which has a photo gallery. Sometimes we were also able to detect pages with a photo gallery based on the structure of the URL. Therefore, we created a JSON file containing configurations for matching the photo gallery page on each of the chosen 24 domains. The spider crawling the domains was implemented to take the configuration file into account when visiting the given domain. Just for illustration, we provide 1 example of such configuration in Figure 4.3.

We restricted the crawler to collect a maximum of 10000 URLs per domain as some domains contained many webpages and it would take a lot of time and computational resources to finish crawling. The crawler which was collecting URLs for annotation was also counting the number of images that it found on the given website. Afterward, we ended up with a long list of URLs for each domain (approximately

10000 URLs for each domain). We sorted the list by the number of images that were present on the given URL and picked 250 URLs uniformly from each domain.

For crawling, we used Scrapy [40]. Scrapy is an application framework for crawling web sites and extracting structured data. It can be used for a variety of applications such as data mining, information processing, or general-purpose web crawling. The source code and configuration used for crawling will be made available in attachments.

4.2 Marking important images on the Web

We needed to build a dataset from the selected web pages that would contain the location of the important image in the HTML documents. This proved to be a rather challenging problem. We looked at available tools for annotating images and none of them supported annotating dynamically loaded websites. Moreover, none of them supported anything like marking the location of the image within the HTML document. Therefore, we had to come up with some custom solution.

First, we thought of creating a desktop application in Electron [41], which is an open-source library for building cross-platform desktop applications. Electron combines Chromium [42] and Node.js [43] into one runtime. It is basically an environment for building web applications and running them in the desktop environment. That stroke us as something that could fit our use case for building application which interprets dynamically loaded web applications using a predefined list of URLs and allows marking images in the HTML document. Moreover, Electron contains natively WebView element [44], which is able to embed whole web pages inside the app using just its URL. However, there was a note in the docs that the architecture of the WebView element is not stable and it is undergoing dramatic strategic changes. Moreover, after we started to build some prototype, we did found some issues that could be caused because of strict cross-origin sharing (CORS) policy on certain websites. Also, it seemed like a pretty complex task to build something that would work reliably.

Therefore, we continued to explore other ideas. The next idea was to create a browser extension that happens to have access to the whole HTML document of the website user is visiting. If the user gives

consent, extensions are able to view and edit the HTML document on any website. The problem that we found with the browser extension was that it is unable to save data on user hard drive. The reason is that the capabilities of browser extensions are limited due to security concerns. However, extensions are able to make HTTP requests, so we could store annotations on a remote server. We decided not to go with the standard solution and develop a backend for storing the annotations as it would require us to deploy it somewhere and manage the infrastructure. Our goal was to develop something minimalistic that would be sufficient for our needs to collect the data. For that reason, we decided to use Firestore as our backend [45]. Firestore is a real-time cloud data store that is modeled as a NoSQL document database. It is free for applications that require less than 50000 reads and 20000 writes per day that fitted our situation. It is a cloud database, so we did not need to deploy it anywhere. Setting up Firestore is fairly easy and it has software development kits (SDK) available in multiple languages. It also provides a REST API.

The fact that a chrome extension can be easily installed from the Chrome browser just by one click store also seemed advantageous. The chrome extensions are cross-platform applications as they can run anywhere where Chrome can run. The chrome extensions are also provided natively with Chrome APIs. One of them is the authentication API, which allows us to use a Google account for authentication. Taking into account the maturity and stability of the development environment, ease of installation, and APIs that are available, we decided to develop a chrome extension for marking important objects in websites.

4.3 Building the chrome extension for creating the dataset

Chrome extensions [46] are small software programs that customize Google Chrome [47], a cross-platform web browser. They enable users to tailor Chrome functionality and behavior to individual needs or preferences. Extensions are built on web technologies such as HTML, JavaScript, and CSS.

4.3.1 Functional requirements

We were not given an explicit list of requirements for the chrome extension. The only requirement was that it should be able to mark important images on websites and it would be something that would be pleasant and easy to work with. Therefore, we came up with some basic set of functions that we considered as necessary for the given task:

1. Ability to sign in and log out using Google account.
2. Ability to maintain and save progress.
3. Ability to detect and mark important images on websites.
4. Ability to choose a dataset of URLs for processing.
5. Ability to navigate on the list of URLs that are supposed to get processed.
6. Ability to retrieve the collected data later on.

The chrome extension can be seen in Figure 4.4.

4.3.2 Architecture

Each chrome extension comes with its JSON-formatted manifest file. The manifest file provides metadata for the extension, such as name, version, permissions for scripts, location of scripts within folder, cross-origin resource sharing policy, and many more.

Chrome extensions consist of 3 scripts that run in separate processes:

- Background script
- Content script
- Popup script

The scripts can communicate via an asynchronous message exchange. Extensions are event-based programs. Events are browser triggers, such as navigation to a new page or receiving a message from another script. Background scripts are useful for monitoring these events and installing handlers for reacting appropriately. The background script is loaded when it is needed and unloaded when it goes idle. Background scripts can register event handlers for various types of events and when the event is triggered, the script becomes active and executes the event handler. We used the background script for communication between the popup and content page and for communication with Firestore.

The popup page has its own graphical user interface that can also be seen in Figure 4.4. It allows the user to sign in and sign out. To avoid bloating the browser of user, the functionality of marking important images with extension is always active just for 1 URL, which is the one that should be processed next. Initially, it is the first URL in the list of all URLs in the chosen dataset, but the user can navigate through the list and change the active URL. The information about which URL is active is associated with the work session for the dataset, so the work on each dataset is shared. Changing the active URL is also possible from the popup page. The user can navigate to the current or next page, which opens the URL in a new tab where the content page script of extension gets activated. The user can also change the active dataset using the settings on the popup page. The popup page also shows information about how many URLs have already been processed and what is the name of the currently active dataset.

The content script is responsible for the core functionality of the extension: marking the important images. It gets activated when the browser navigates to the currently active URL. When active, the content script is periodically scanning the HTML document, looking for images every 1.5 seconds. Images are detected via *href* attribute of *img* elements or via *background-image* attribute of any other HTML element. After detecting elements that contain an image, the red border is added to each such element and event handlers for clicking on the element are overridden. If the user clicks on the image element, the image is marked as important in the browser storage and its border changes color to green. After clicking on the save and proceed button, information about all marked images for the URL is stored in Firestore

and the next URL in the list becomes the active URL for the extension. It is not possible to use Firestore directly from the content script, so the background script acts as a proxy for communication with Firestore. Along with the information which images were important on the given URL, the extension also stores metadata about where the important images were located in the HTML document. This data can be further used for discovering patterns for locating important images in the HTML documents.

Content script and popup script can exchange messages with the background script in an asynchronous manner. The communication between scripts is shown in Figure 4.5. The content script also shows information about how many images have been already marked, how many URLs have been processed, and information about navigation state in the current dataset. It provides users also with an option to navigate in the dataset without saving progress in Firestore by clicking on arrows in the GUI. The GUI for content script can be seen in Figure 4.4.

4.3.3 Data storage

We decided to use Firestore as our backend. The reason for choosing Firestore was that it is a cloud-hosted service which means that it does not require too much setup and maintenance of the underlying infrastructure. The only thing that was required for using Firestore in our app was generating a secret service account key in the form of a JSON file.

Firestore is a NoSQL database where documents are a basic unit of storage. Documents contain key-value pairs of data. They are organized into collections which are containers for documents that support efficient, expressive, and flexible querying. The database itself is optimized for storing collections of a large number of small documents. Since it is a NoSQL database, the documents have a flexible structure. That served us well as we could iterate quickly on the design of our data model in the early phase of development. Integrating Firestore into our codebase was fairly easy and seamless as it comes with its own JavaScript software development kit (SDK). The service also has a lightweight and nice GUI for modifying and viewing data in the web browser. The database supports real-time communication by listening

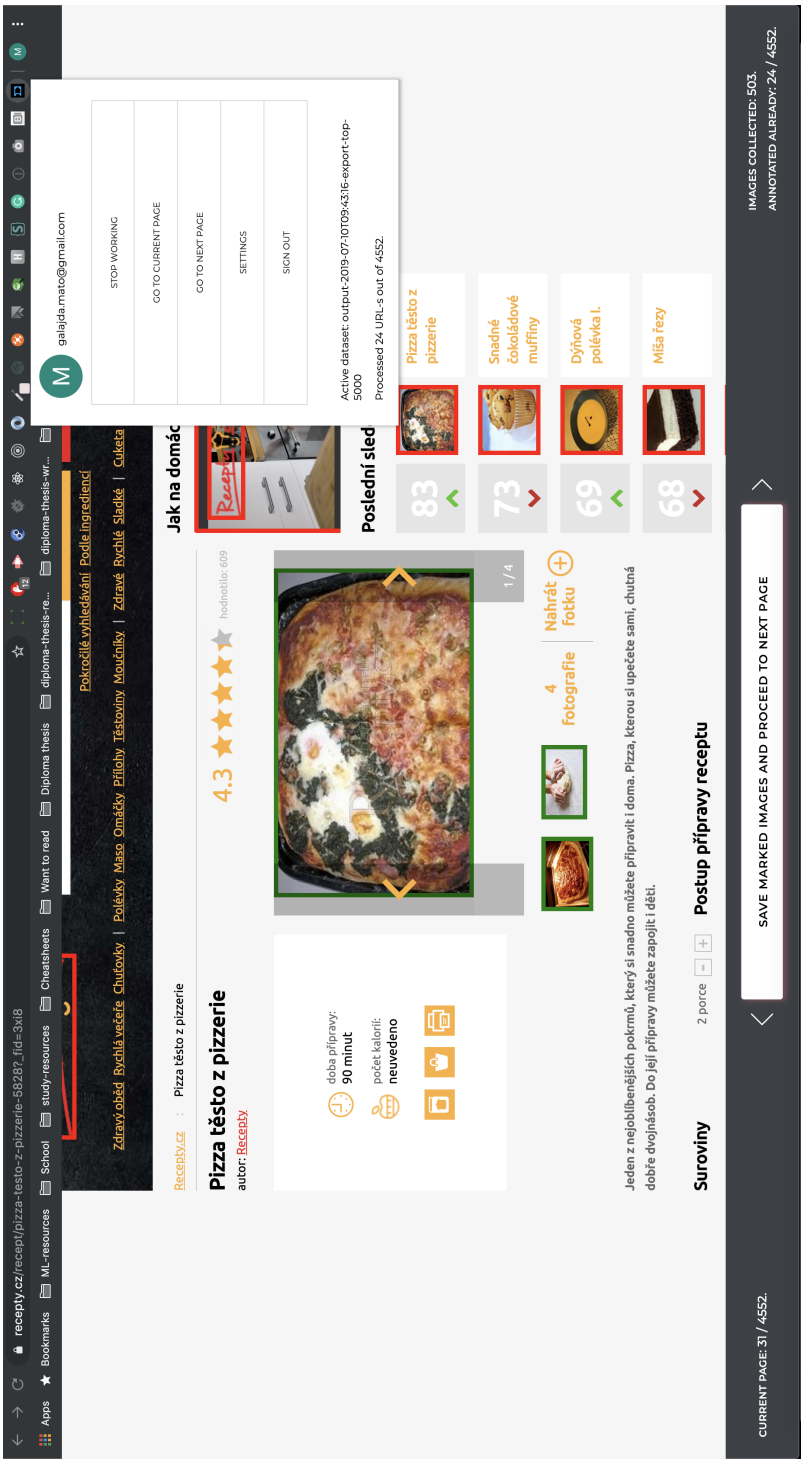


Figure 4.4: GUI of the chrome extension

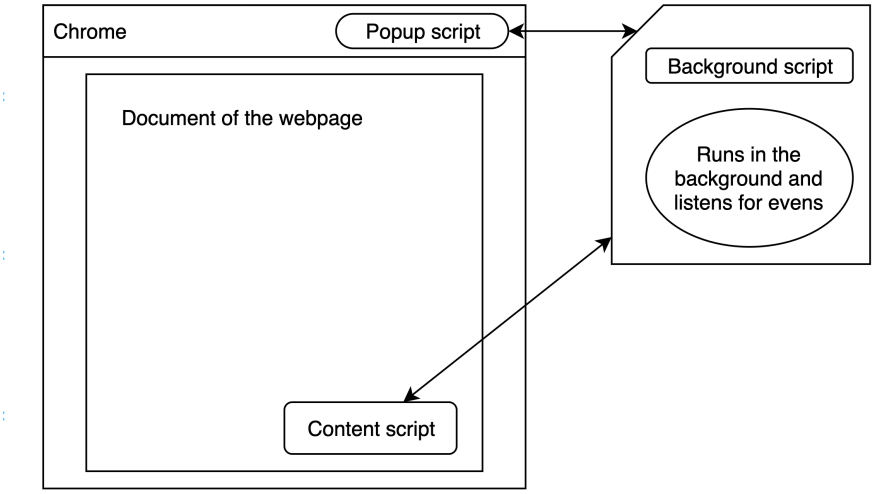


Figure 4.5: Chrome extension architecture

to events that are fired when the data is updated on the backend. However, it was not very relevant for our use case and we did not need too much real-time functionality since we had just one human annotator. However, it could be useful if we would have multiple annotators that would like to work at the same time since the progress on datasets is shared between users.

4.3.4 Technology stack

For building and bundling the extension, we used Webpack [48]. Webpack is a tool that allows us to bundle all of our JavaScript files into one big file to avoid multiple HTTP requests when fetching the content of the website. With the help of Webpack, we were able to structure our application reasonably into multiple files but serve it just as one file after bundling it. This proved to be useful and we did spend some time coming up with a good structure for our app that supports sharing code between the background, popup, and the content script. We used React [49] for building a user interface (UI) for the popup and content page. React is a popular UI library for building single-page web

applications. It simplifies frontend development with a concept of developing reusable UI components. The component-oriented approach allowed us to manage the complexity of the UI for our application by splitting it into multiple smaller components that have a well-defined purpose in the UI and can be reused in different parts.

4.4 Labeling objects in images

After we collected important images from websites using the developed Chrome extension, we needed to annotate objects on the extracted images to evaluate our object detection models. For the purpose of labeling objects, we looked at few options and chose Labelbox [50] since it had a nice web-based GUI. Labelbox also had convenient import and export options. It also has a Web API that allows to automate certain tasks and offers more flexibility than GUI.

We also needed some utility programs that would be responsible for processing and moving data between Firestore, Labelbox and our scripts used for evaluating models. For that purpose, we developed small utility programs with command-line user interface (CLI) in Go programming language. The reason for using Go was that we liked the Go version of SDK for Firestore. Go also seemed like a good compromise between having a type-safe language and being able to develop the utility program quickly. The CLI utility is provided in attachments and is available in the form of open source on GitHub [51].

4.5 Results

The statistics for the dataset that was built are shown in Table 4.1. Note that the number of images that were marked as important and the number of annotated images for object detection is different. The reason is that the annotator was instructed to skip annotating the image if it did not contain any object with Open Images class.

Table 4.1: Resulting statistics about our dataset

Statistic	Quantity
Collected web pages for annotating	4552
Annotated web pages	26
Collected important images	503
Annotated important images	384
Bounding boxes	3891
Bounding boxes per image	10.13
Classes	102

5 Implementing object detectors

After researching state-of-the-art methods for object detection and available datasets, we were ready to implement the inference pipelines. We decided to use Python. Our research on datasets led us to the conclusion that we need models that are built on Open Images dataset [17]. Since this is really a large-scale dataset, the possibility of reproducing papers and training models on our own was not an option, as the training process requires a lot of GPUs that cost a lot of money that we did not have. Also, it would not be very beneficial to train models if they have already been trained and made public. Therefore, the goal was to implement object detectors that would use already trained weights that have been published by the research community. Based on our research on the state-of-the-art object detection models, we decided to take one representative of the region-based method, Faster R-CNN, and one representative of the regression-based method, YOLOv3.

We invented an abstract base class that would be implemented for both models and would serve as an interface for common functionality. This interface would be used in code for measuring inference speed and mAP on our test dataset that we collected with the Chrome extension and Labelbox. Moreover, the abstract class would contain logic for shared functionality, such as loading image from the file path

or formatting the outputs. Having a well-defined interface also helps with drawing and visualizing results of detected bounding boxes for debugging and experimenting. Also, it can be used later on in production for out of the box inference, without really knowing the internals of the underlying models. The abstract interface consists of 2 methods:

1. method for producing box coordinates, indices of classes, probability scores for each detected object,

2. method for producing a dictionary that maps class index to human readable class.

We needed to have both models implemented with same deep learning technology to have controlled environment for a comparison of the methods. Based on the numbers from the poll that we found [52], we decided that our models will be implemented in TensorFlow [53] and Keras [54]. The reason is that it is the most popular deep learning technology at the time of writing and it is suitable for production usage.

5.1 Faster R-CNN inference pipeline

Implementing the inference pipeline for Faster R-CNN architecture turned out to be fairly straightforward. Luckily for us, we found a GitHub repository [55] that contained already trained Faster R-CNN model on Open Images dataset. The repository contained a link for downloading the serialized TensorFlow inference graph with weights already baked in the graph. It could be used for off-the-shelf inference. Everything that was needed to integrate with our abstract class for object detection was to compile protobuf files [56] into Python classes, deserialize the TensorFlow graph from the file and do a little of post-processing for the outputs from the graph to match the outputs expected by our object detection abstract class. The instructions for setting up and running Faster R-CNN object detector are provided in the `README` file provided in the attachments.

5.2 YOLOv3 inference pipeline

The situation with YOLOv3 turned out to be a bit trickier. Authors of the YOLO use their own framework called Darknet [57]. It is a custom deep learning framework in C. The framework is good for doing experiments and research since authors have complete control of the environment and computations. However, it is not ideal for doing object detection in production. Looking at the commit history on GitHub suggests the framework has just one contributor and it is not actively maintained. Moreover, the authors mentioned in the paper that the framework itself contained a bug that reduced model accuracy [32]. We also found a lot of commented out code in the framework and the quality of the code did not seem good. Therefore, we had strong arguments for reimplementing the model in TensorFlow.

In the official GitHub repository of the Darknet framework, we found serialized weights for the model trained on the Open Images v4 dataset [17]. The repository also contained a configuration file for the network used for training, which was in the framework-specific format. We created a parser that recreates a TensorFlow model based on the configuration file for the Darknet framework. The parser also loads weights into the TensorFlow model from the Darknet-specific file containing serialized weights. This turned out to be fairly complicated since we had to through the source code of the Darknet. We had to understand how are the weights serialized. We also had to understand specifics of the CNN implementation in order to be able to reconstruct the model in TensorFlow. One example of the complication with the reconstruction of the model in TensorFlow was the strategy used for padding, which is different in TensorFlow. Also, weights for the convolution layers were serialized in format `(output_depth,input_depth,height,width)`. TensorFlow convolution weights are serialized in format `(height,width,input_depth,output_depth)`. In the reconstruction of the model, we took some inspiration from repositories that have already done it [58]. However, our experiments showed that the reconstructions available on GitHub were not working with the model trained on the Open Images dataset.

Another challenge was that papers do not mention details of preprocessing and post-processing that they do on images, and it showed to be very important. YOLOv3 downsamples input 32 times, so input

dimensions have to be divisible by 32. Also, the configuration file contained information that the size of the images that were used for training was 608 x 608. First, we tried to resize images into a fixed size of 608 x 608 using bicubic interpolation. However, the outputs of the model were really poor compared to what was produced by the original Darknet model. We found out that authors used a technique called "letterboxing" for resizing images into fixed dimensions. The illustration of the technique can be seen in Figure 5.1. Also, by examining the source code of the Darknet framework, we found out that authors used linear interpolation. Our experiments showed that the interpolation strategy was not that important. However, the "letterboxing" technique proved to be very significant for the performance. Impacts on model quality due to different pre-processing strategies can be observed in Figure 5.2.

In our final implementation of the model, we use the "letterboxing" technique with bicubic interpolation, exactly as the official implementation. Also, we cross-validated our implementation in Keras and TensorFlow with official Darknet implementation by running inference on random images.

The instructions for setting up and using the YOLOv3 object detector for off-the-shelf inference are provided in the `README` file provided in the attachments.

We resize images of smaller dimensions into images of size 608 x 608 using linear interpolation meanwhile maintaining aspect ratio. We add a gray background as padding along a smaller dimension to match the size of 608. In the picture, we add a gray background along the y-axis (height of the image).

Figure 5.1: Letterboxing technique used in YOLOv3

In the first image, we can observe predictions made by YOLOv3 when using bicubic interpolation for resizing the image. In the second image, we can observe predictions made when using the "letterboxing" technique. Clearly, the accuracy drops since the animal is detected with a much lower probability score and it is shifted a bit.

Figure 5.2: Impact of the pre-processing

6 Comparison of object detection techniques

After we implemented the inference pipelines for one state-of-the-art representative of the region-based and regression-based object detector, we needed to evaluate them. We provide comparisons on the public datasets and also on our collected dataset. The metrics that are usually used for comparison of different object detectors and would guide us in our evaluation are precision/recall curve and inference speed. As we discussed in Chapter 2, the composite score that is used for evaluation precision/recall of the object detectors is mean average precision (mAP).

The goal is to have an accurate and fast detector. For evaluating models on our dataset, we have implemented evaluation scripts for these metrics inside Jupyter notebooks [59], and we executed them on Google Colaboratory [60] which makes our experiments reproducible for anybody who owns Google account. With some effort, results can also be reproduced locally by running Jupyter notebooks. Google Colaboratory allowed us to test inference speed on CPU, TPU, and GPU. The notebooks are provided in attachments and also available in the Google drive.

In section 6.1, we present publicly available benchmarks that were published previously by other researchers. The presented results are for the same architecture of the models as our object detectors use, but the implemented models differ since they were trained on different datasets. In section 6.2, we present results from the experiments that we conducted. The discussions about the results are presented in Chapter 7.

6.1 Publicly available benchmarks

In this section, we present publicly available benchmarks on two public datasets that were published for the models that used the same architecture as our models.

6.1.1 Results on Open Images v2

First, we present the benchmarks on **Open Images version 2**. The mAP can be seen in Table 6.1. Note that the computed metric is not exactly the same as mAP defined in Section 2.1. It is defined by Open Images Challenge. The metric is slightly more complicated because it uses more information than just simple bounding box annotations. Also, the models presented are not exactly the same as our implemented models. They cannot be, because the second version of the dataset contained only 500 classes, and our models were trained on the fourth version that has 600 classes. They are merely models built on the same architectures with the same technologies as our models.

Table 6.1: Comparison of mAP[1] on Open Images v2 (from [61] [62])

Model architecture	mAP@0.5
YOLOv3	42.407
Faster R-CNN	37

6.1.2 Results on COCO

Next, we present results from the **COCO** dataset. In Table 6.2 we present mAP results as defined by the COCO competition [18]. The results include mAP with the IOU threshold for the detection of 0.5. Moreover, we present the averaged mAP with IOU thresholds ranging from 0.5 to 0.95.

Table 6.2: Comparison of mAP [2] on COCO (from [32])

Model architecture	mAP@0.5	mAP(averaged)
YOLOv3	57.9	33.0
Faster R-CNN	55.5	34.7

1. mAP that is reported is defined by the Open Images challenge
2. mAP that is reported is defined by the COCO competition

6.2 Our benchmarks

In this section, we present our own results for computing mAP metrics. The results presented are for two models that we implemented for the inference. We have created and used our own implementation for computing mAP since we did not find any existing open-source implementation that was easy to use and flexible enough for use in Google Colab. However, to avoid mistakes, we cross-validated our results with one popular implementation that we found available on GitHub [63]. We couldn't use the implementation in Google Colab since it could be executed just as standalone program from a terminal. Note that the implementation uses mAP defined by PASCAL VOC [64] that we discussed in Section 2.1.

6.2.1 Results on Open Images v4

In this subsection, we present results for computing mAP metrics on the subset of the test dataset from the Open Images version 4. The test subset contains images which models did not see during training. We used approximately 1000 images. We did not use the whole test dataset because we had a limited amount of storage and computing power available. The comparison of the overall mAP can be seen in Figure 6.1.

6.2.2 Results on our dataset

In this subsection, we present results for computing mAP metrics on our collected dataset. The comparison of the overall mAP can be seen in Figure 6.2.

However, PASCAL VOC [19] considers a detected box as a true positive if it overlaps a ground truth box with IOU threshold of 0.5. IOU of 0.5 is quite generous and by varying IOU thresholds for matching detected boxes as true positives, we can be observe the localization capability of models. Therefore, we looked also at mAP by varying IOU from 0.5 to 0.95. The results are summarized in Table 6.3.

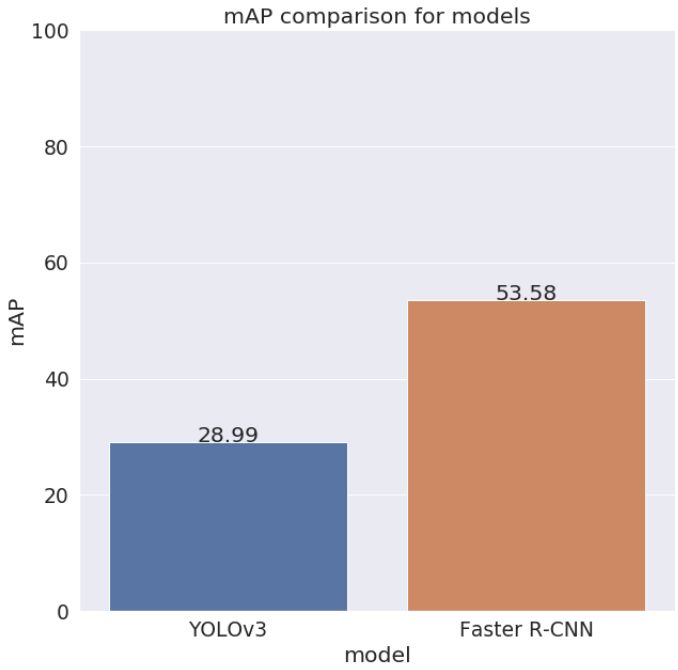


Figure 6.1: mAP@0.5 results on Open Images v4

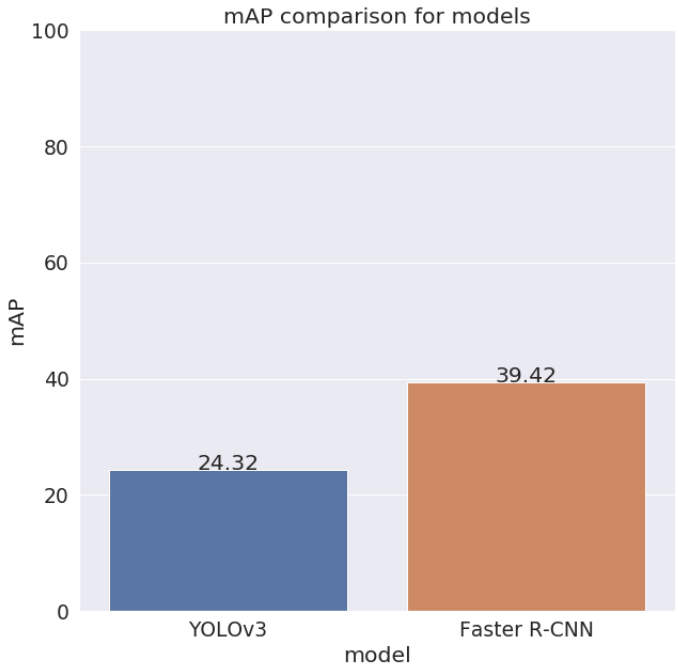


Figure 6.2: mAP@0.5 results on our dataset

Table 6.3: mAP results on our dataset by varying IOU threshold

metric	Faster R-CNN	YOLOv3
mAP@0.5	39.4	24.3
mAP@0.55	38.0	23.0
mAP@0.6	34.9	21.6
mAP@0.65	31.6	19.40
mAP@0.7	29.29	18.3
mAP@0.75	23.79	16.0
mAP@0.8	17.29	12.1
mAP@0.85	10.10	8.4
mAP@0.9	3.2	0.8
mAP@0.95	1.09	0.1

6.3 Inference speed results

For benchmarking and comparing the inference speed of the models, we used three images with various sizes: 3002 x 3000 (big), 284 x 177 (small), 512 x 512 (medium). For measurements on GPU and TPU, we ran inference on each image for 100 iterations to get a good average and a rough idea about the standard deviance of the model inference speed. For measurements on CPU, we ran inference on each image only for ten iterations as it was taking significantly more time to run those benchmarks. In our review of the literature, we did not find any information about whether the size of the image influences the inference speed. Therefore, to answer our question, we decided to use three images consisting of different sizes.

The overall comparison results for the average inference speed can be seen in Figure 6.4. We ran benchmarks for all models on available CPU, GPU, and TPU devices in Google Colaboratory. Tesla K80 was the GPU that was used for benchmarks. For CPU benchmarks, Google Colab provided us with two Intel(R) Xeon(R) CPU @ 2.20GHz processors. For TPU, we were using eight TPU-s that are available in Google Colab.

It is worth noting that after initial inference speed benchmarks, we found YOLOv3 slower than expected. After some time spent profiling, we identified the bottleneck - performing non-max suppression in Numpy (on CPU). We implemented another version of YOLOv3 that performed non-max suppression on GPU using TensorFlow implementation. This led to a significant speedup of the model inference.

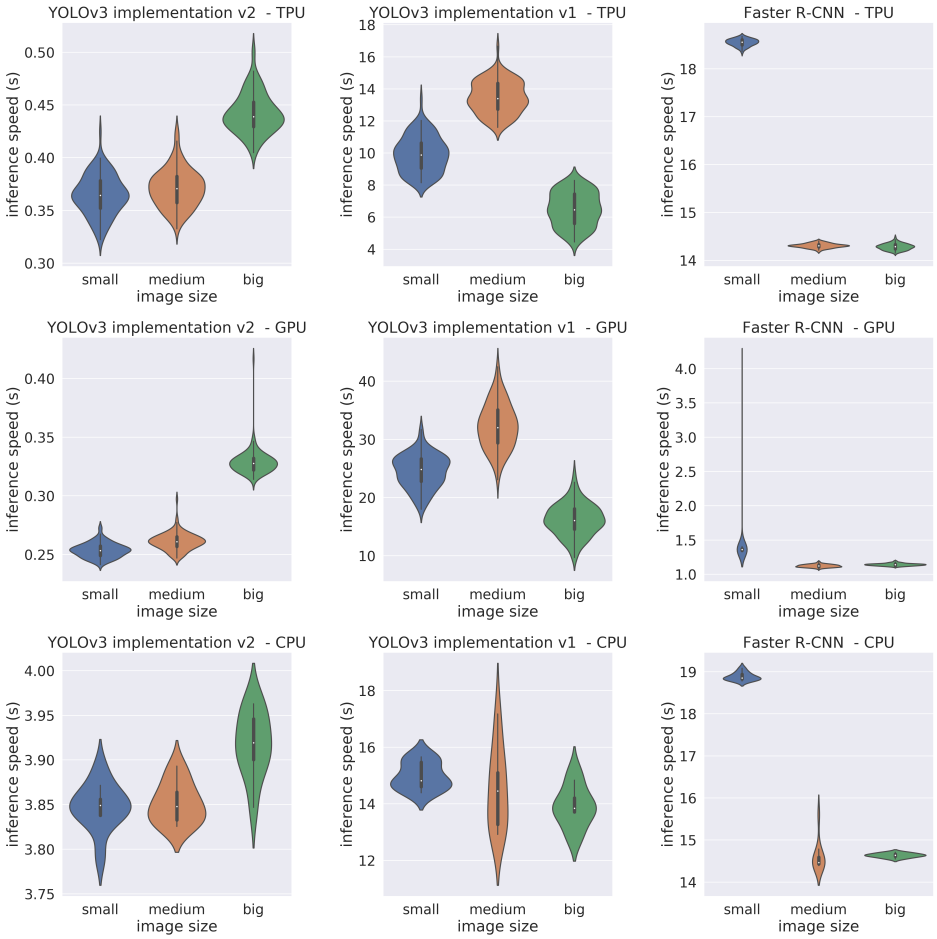


Figure 6.3: Violin plots demonstrating inference speed on 3 images with different size by different models and devices

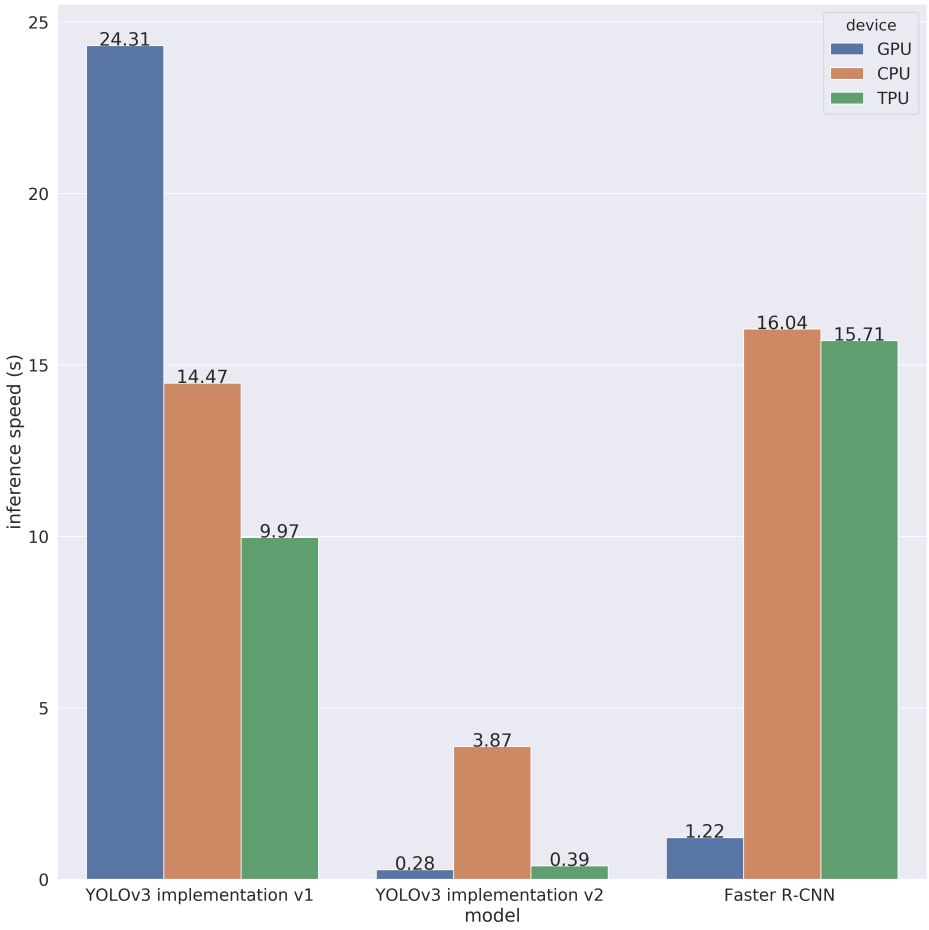


Figure 6.4: Comparison of average inference speed

7 Discussions

7.1 Collected dataset

We collected 4552 web page URLs for annotating important images in the HTML documents. Using the tool that we developed, we annotated 26 web pages with 503 important images. This is most likely not enough for discovering patterns for determining important images in the web pages. The reason for not annotating more web pages is that we prioritized annotating collected images with bounding boxes to compare our object detectors. We got enough images after 26 web pages. Therefore, we focused instead on collecting annotations for objects in the images. However, the tool is ready for annotating the remaining URLs that we collected, and it can be used for creating a bigger dataset.

The dataset that was built for comparing object detectors contains annotations for roughly 400 images. That is not too much, but we believe that it is enough to compare and draw some conclusions about our object detectors. The reason for not annotating more images is that the annotation process for object detection is very time consuming and we had a limited amount of resources for the task. The annotated bounding boxes contain 102 different classes altogether out of 601 Open Images v4 classes. The distribution of the classes can be seen in Figure 7.1. We can observe that classes are not uniformly distributed, and 92 classes contribute to only 21% annotations. This places our dataset into a slightly different context than Open Images, which is carefully designed to contain annotations with uniformly distributed classes. Collecting and annotating more images would most likely result in more distributed classes.

It is worth noting that annotating images with 601 classes that our object detectors support is not an easy task. The annotator needed to be aware of 601 classes that can be present and should be marked on the image. To be able to annotate it with the highest possible quality, we would need considerably more time and annotators than we had. For example, annotators of Open Images dataset had to be properly trained, and each annotator was responsible for annotating only 1 class at a time [17]. Moreover, the annotations were peer-reviewed. Therefore, it is fair

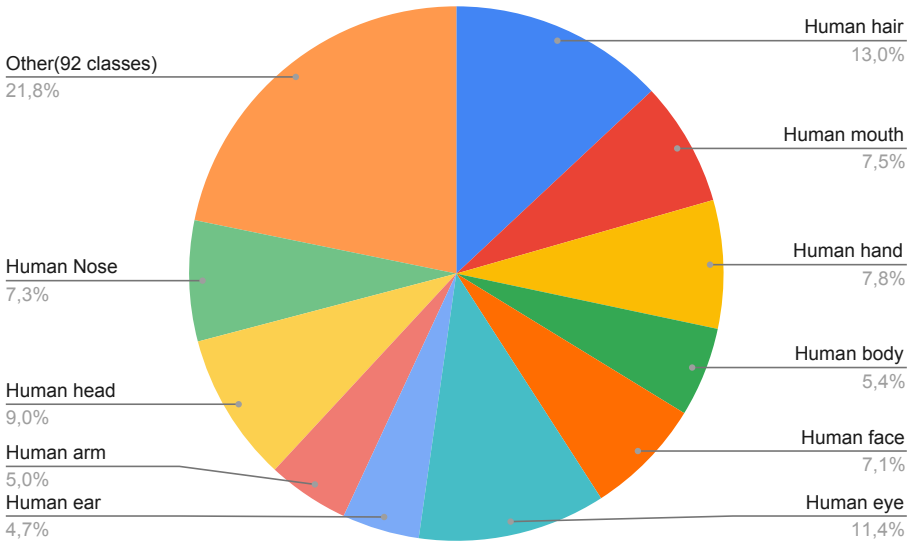


Figure 7.1: Distribution of classes in our dataset

to say that in comparison to competition datasets such as Open Images, our annotations are not perfect. It is possible that the annotator missed some bounding boxes, but those that were created should be valid. Taking into account the fact that the average number of bounding boxes per image in our dataset is even higher than in Open Images suggests that the dataset can provide meaningful insights. Consequently, we believe that our annotations can help us make informed conclusions about the performance of our models in our domain, but the resulting numbers should be interpreted with caution (meaning that more rigorous annotation process could possibly produce slightly different and more accurate results).

7.2 Object detection models

The publicly available benchmarks suggest that the performance of YOLOv3 and Faster R-CNN is similar with regards to mAP. However, our benchmarks on Open Images v4 show that the Faster R-CNN significantly outperforms YOLOv3. We are unsure why our benchmarks

do not align exactly with the previous public benchmarks. We cross-validated predictions made by our implementation in TensorFlow with the Darknet implementation. Therefore, our implementation should not be the source of any mistakes. One possibility is that the released YOLOv3 model might not be adequately trained to the convergence. Another possibility is that the architecture was simply not able to generalize well on such a complex dataset as the Open Images v4 is. The results for the released model were never published, which leads us to believe that might have not been that good.

The benchmarks that we did on our own dataset also suggest the superior performance of Faster R-CNN with regards to the mAP metric. They seem to align with our results on Open Images v4. As we already mentioned in the previous section, our collected annotations are not perfect. Therefore, the mAP for both models could be higher if we would have more resources to annotate it with high quality. Nevertheless, the gap between the mAP of Faster R-CNN and YOLOv3 leads us to believe that Faster R-CNN is qualitatively a better predictor in our domain. Moreover, when we look at how mAP decreases when we increase the IOU threshold for detection, both models seem to lose precision in a similar fashion. However, at mAP with IOU threshold of 0.85, both models have similar results. That suggests that for localizing objects with highly precise bounding boxes, both models have similar performance.

We performed a manual error analysis to better understand predictions made by models. We used Labelbox API to import predictions made by both models, and we manually compared some of them. We saw the pattern of YOLOv3 detecting only more general classes of objects. On the other hand, Faster R-CNN seemed pretty good at detecting even more concrete classes. One example would be an image containing a horse. YOLOv3 would usually detect an object with a class animal. Faster R-CNN would correctly detect a horse. Moreover, Faster R-CNN seemed far better at predicting smaller objects in the image.

If we look at the inference speed of both models, YOLOv3 is clearly the winner. That was expected since the purpose of YOLOv3 has always been real-time object detection. It is 4.37 times faster than Faster R-CNN. That also means that for predicting objects on the same number of images, YOLOv3 consumes roughly four times less GPU. That

is important because GPU is nowadays an expensive resource. Our experiments show that the inference speed of the models is affected by the quality of the implementation. Not running non-max suppression on GPU greatly decreased the inference speed of YOLOv3.

As can be seen in Figure 6.3, YOLOv3 is slightly slower on images with a bigger size but the difference is not that significant. The reason is probably that we run the computations for "letterboxing" image on CPU. Surprisingly, Faster R-CNN seemed to be significantly slower on smaller images. We assume that it might be because of the upsampling that is performed. The violin plots demonstrate that inference speed across different devices grows proportionally as expected. TPU times for Faster R-CNN are almost the same as times on CPU, which leads us to believe that the Faster R-CNN model was not able to utilize available TPU devices.

To sum up, our results suggest that the implemented Faster R-CNN is a more accurate predictor than YOLOv3. However, if the inference speed is important, YOLOv3 might be a better option.

8 Conclusions

The thesis was concerned with the problem of extracting useful information from the images on the web using computer vision techniques. The primary motivation was to use the extracted information for discovering consumer's interests to support targeted advertising.

After researching different computer vision techniques, we decided to use object detection as the technique for extracting information from the images. We researched and compared public datasets that are available for the object detection task, and that we could use for commercial purposes. The dataset that was determined as the best for our purpose was Open Images v4, which is at the time of writing considered as the hardest dataset for detection [32]. After that, we performed research on state-of-the-art techniques for object detection. We produced a brief summary of the research that contains information about the development of the most successful techniques.

We built a dataset of images that are important for targeted advertising in the context of users browsing the web. We annotated HTML documents from the web with information about which images were important. For annotating the documents, we used our custom-made solution in the form of a Chrome extension. The extension was released in the Chrome store, and it is available for use by anyone using the Chrome browser. The data that was collected also contains information about the location of the important images in the HTML document. It can be used in future work to analyze patterns for locating relevant images in the HTML document. Moreover, the tool that we developed can be used for creating a bigger dataset.

As part of our work, we also implemented a scraper that can crawl domains and collect URLs of pages with images that are useful for advertising. We used the scraper for collecting URLs that were used by the Chrome extension. Moreover, we researched tools for annotating images for object detection. Based on the research, we chose one tool that was used for annotating our dataset of images for object detection.

We implemented two state-of-the-art models for object detection that are built on the Open Images v4 dataset. Each detector represents one of the two most prevalent architectures for the object detection task. We created scripts and Jupyter notebooks for evaluating and com-

paring the models. Specifically, we looked at the inference speed and mean average precision which are the metrics commonly discussed in the research papers and are used in object detection competitions [32] [2]. Our implementation for building and comparing models was made flexible enough to support other models in the future. Object detection is still an active area of the research, and new models pop up each year. Our codebase can be used in the future for adding new models that will be created, and they can be evaluated on our collected dataset.

Public benchmarks on different datasets suggested that both architectures were similarly good at detecting objects. We ran our own benchmarks for comparing models on a publicly available dataset and on our own dataset. Our experiments suggest that one model is qualitatively better at predictions, and the other one is significantly faster. This leads us to the conclusion that choosing a model represents a trade-off between the quality and costs. Our experiments clearly show that Faster R-CNN outperformed YOLOv3 in detecting the objects in our domain. However, if the associated costs and time needed for extracting information from the images are more important than the drop in the accuracy, YOLOv3 might be a better option. Both object detectors that we implemented are ready for off-the-shelf inference.

All the source code that was produced has been made available in the form of open-source on GitHub [65] [66] [51] [67].

Bibliography

1. REDMON, Joseph; DIVVALA, Santosh Kumar; GIRSHICK, Ross B.; FARHADI, Ali. You Only Look Once: Unified, Real-Time Object Detection. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2015, pp. 779–788.
2. REN, Shaoqing; HE, Kaiming; GIRSHICK, Ross B.; SUN, Jian. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *CoRR*. 2015, vol. abs/1506.01497. Available from arXiv: 1506.01497.
3. LIU, Wei; ANGUELOV, Dragomir; ERHAN, Dumitru; SZEGEDY, Christian; REED, Scott E.; FU, Cheng-Yang; BERG, Alexander C. SSD: Single Shot MultiBox Detector. *CoRR*. 2015, vol. abs/1512.02325. Available from arXiv: 1512.02325.
4. *CS231n: Convolutional Neural Networks for Visual Recognition* [online]. Standford, CA, USA: Standford University, 2019 [visited on 2019-07-01]. Available from: http://cs231n.stanford.edu/.
5. *Internet World Stats* [online] [visited on 2019-07-03]. Available from: https://www.internetworldstats.com/stats.htm.
6. PLUMMER, Joseph; RAPPAPORT, Steve; HALL, Taddy; BAROCCI, Robert. *The Online Advertising Playbook: Proven Strategies and Tested Tactics from the Advertising Research Foundation*. New York, NY, USA: John Wiley & Sons, Inc., 2007. ISBN 9780470051054.
7. *Gauss Algorithmic* [online] [visited on 2019-09-22]. Available from: https://www.gaussalgo.com/en/.
8. PRINCE, Simon J. D. *Computer Vision: Models, Learning, and Inference*. 1st. New York, NY, USA: Cambridge University Press, 2012. ISBN 1107011795, 9781107011793.
9. GOODFELLOW, Ian; BENGIO, Yoshua; COURVILLE, Aaron. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.
10. HUANG, T. *Computer Vision: Evolution And Promise*. CERN, 1996. Available from DOI: 10.5170/CERN-1996-008.21.

11. RAZZAK, Muhammad Imran; NAZ, Saeeda; ZAIB, Ahmad. Deep Learning for Medical Image Processing: Overview, Challenges and Future. *CoRR*. 2017, vol. abs/1704.06825. Available from arXiv: 1704.06825.

12. DAVIS, Abe; RUBINSTEIN, Michael; WADHWA, Neal; MYSORE, Gautham J.; DURAND, Frédo; FREEMAN, William T. The Visual Microphone: Passive Recovery of Sound from Video. *ACM Trans. Graph.* 2014, vol. 33, no. 4, pp. 79:1–79:10. ISSN 0730-0301. Available from DOI: 10.1145/2601097.2601119.

13. SZELISKI, Richard. *Computer Vision: Algorithms and Applications*. 1st. Berlin, Heidelberg: Springer-Verlag, 2010. ISBN 1848829345, 9781848829343.

14. DOMINGOS, Pedro. A Few Useful Things to Know About Machine Learning. *Commun. ACM*. 2012, vol. 55, no. 10, pp. 78–87. ISSN 0001-0782. Available from DOI: 10.1145/2347736.2347755.

15. SRINIVAS, Suraj; SARVADEVABHATLA, Santosh Ravi Kiran; MOPURI, Konda Reddy; PRABHU, Nikita; S.S. KRUTHIVENTI, Srinivas; BABU, R. An Introduction to Deep Convolutional Neural Nets for Computer Vision. In: 2017, pp. 25–52. ISBN 9780128104088. Available from DOI: 10.1016/B978-0-12-810408-8.00003-1.

16. BISHOP, Christopher M. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006. ISBN 0387310738.

17. KUZNETSOVA, Alina et al. The Open Images Dataset V4: Unified image classification, object detection, and visual relationship detection at scale. *arXiv:1811.00982*. 2018.

18. LIN, Tsung-Yi et al. Microsoft COCO: Common Objects in Context. *CoRR*. 2014, vol. abs/1405.0312. Available from arXiv: 1405.0312.

19. EVERINGHAM, M.; ESLAMI, S. M. A.; VAN GOOL, L.; WILLIAMS, C. K. I.; WINN, J.; ZISSERMAN, A. The Pascal Visual Object Classes Challenge: A Retrospective. *International Journal of Computer Vision*. 2015, vol. 111, no. 1, pp. 98–136.

20. DALAL, Navneet; TRIGGS, Bill. Histograms of Oriented Gradients for Human Detection. In: *Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1 - Volume 01*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 886–893. CVPR '05. ISBN 0-7695-2372-2. Available from DOI: 10.1109/CVPR.2005.177.

21. GIRSHICK, Ross B.; DONAHUE, Jeff; DARRELL, Trevor; MALIK, Jitendra. Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR*. 2013, vol. abs/1311.2524. Available from arXiv: 1311.2524.

22. UIJLINGS, J. R.; SANDE, K. E.; GEVERS, T.; SMEULDERS, A. W. Selective Search for Object Recognition. *Int. J. Comput. Vision*. 2013, vol. 104, no. 2, pp. 154–171. ISSN 0920-5691. Available from DOI: 10.1007/s11263-013-0620-5.

23. HE, Kaiming; ZHANG, Xiangyu; REN, Shaoqing; SUN, Jian. Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition. *CoRR*. 2014, vol. abs/1406.4729. Available from arXiv: 1406.4729.

24. GIRSHICK, Ross B. Fast R-CNN. *CoRR*. 2015, vol. abs/1504.08083. Available from arXiv: 1504.08083.

25. SIMONYAN, Karen; ZISSERMAN, Andrew. Very Deep Convolutional Networks for Large-Scale Image Recognition. *CoRR*. 2014, vol. abs/1409.1556. Available also from: http://arxiv.org/abs/1409.1556.

26. ZEILER, Matthew D.; FERGUS, Rob. Visualizing and Understanding Convolutional Networks. *CoRR*. 2013, vol. abs/1311.2901. Available from arXiv: 1311.2901.

27. HE, Kaiming; ZHANG, Xiangyu; REN, Shaoqing; SUN, Jian. Deep Residual Learning for Image Recognition. *CoRR*. 2015, vol. abs/1512.03385. Available from arXiv: 1512.03385.

28. SZEGEDY, Christian; LIU, Wei; JIA, Yangqing; SERMANET, Pierre; REED, Scott E.; ANGUELOV, Dragomir; ERHAN, Dumitru; VANHOUCKE, Vincent; RABINOVICH, Andrew. Going Deeper with Convolutions. *CoRR*. 2014, vol. abs/1409.4842. Available from arXiv: 1409.4842.

29. *R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms* [online] [visited on 2019-09-11]. Available from: https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e.

30. *R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms* [online] [visited on 2019-09-12]. Available from: https://medium.com/@jonathan_hui/real-time-object-detection-with-yolo-yolov2-28b1b93e2088.

31. REDMON, Joseph; FARHADI, Ali. YOLO9000: Better, Faster, Stronger. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016, pp. 6517–6525.

32. REDMON, Joseph; FARHADI, Ali. YOLOv3: An Incremental Improvement. *CoRR*. 2018, vol. abs/1804.02767. Available from arXiv: 1804.02767.

33. LIN, Tsung-Yi; DOLLÁR, Piotr; GIRSHICK, Ross B.; HE, Kaiming; HARIHARAN, Bharath; BELONGIE, Serge J. Feature Pyramid Networks for Object Detection. *CoRR*. 2016, vol. abs/1612.03144. Available from arXiv: 1612.03144.

34. *ImageNet* [online] [visited on 2019-07-01]. Available from: http://image-net.org/.

35. MILLER, George A. WordNet: A Lexical Database for English. *Commun. ACM*. 1995, vol. 38, no. 11, pp. 39–41. ISSN 0001-0782. Available from DOI: 10.1145/219717.219748.

36. RUSSAKOVSKY, Olga et al. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV)*. 2015, vol. 115, no. 3, pp. 211–252. Available from DOI: 10.1007/s11263-015-0816-y.

37. *ImageNet - Download the Object Bounding Boxes* [online] [visited on 2019-09-10]. Available from: http://image-net.org/download-bboxes.

38. KRASIN, Ivan et al. OpenImages: A public dataset for large-scale multi-label and multi-class image classification. *Dataset available from https://storage.googleapis.com/openimages/web/index.html*. 2017.

39. *Imagenet - Summary and Statistics (updated on April 30, 2010)* [online] [visited on 2019-09-10]. Available from: http://image-net.org/about-stats.
40. *Scrapy | A fast and Powerful Scraping and Web Crawling Framework* [online] [visited on 2019-09-22]. Available from: https://scrapy.org.
41. *Electron Documentation* [online] [visited on 2019-07-05]. Available from: https://electronjs.org/docs/tutorial/about.
42. *The Chromium Projects* [online] [visited on 2019-09-22]. Available from: https://www.chromium.org.
43. *Node.js* [online] [visited on 2019-09-22]. Available from: https://nodejs.org/en/.
44. *Electron WebView documentation* [online] [visited on 2019-07-05]. Available from: https://electronjs.org/docs/api/webview-tag.
45. *Cloud Firestore* [online] [visited on 2019-09-22]. Available from: https://firebase.google.com/docs/firestore.
46. *Chrome Extensions* [online] [visited on 2019-07-09]. Available from: https://developer.chrome.com/extensions.
47. *Google Chrome: The New Chrome & Most Secure Web Browser* [online] [visited on 2019-09-22]. Available from: https://www.google.com/intl/en/chrome/.
48. *webpack* [online] [visited on 2019-09-22]. Available from: https://webpack.js.org.
49. *React - A JavaScript library for building user interfaces* [online] [visited on 2019-09-22]. Available from: https://reactjs.org.
50. *Labelbox: The leading training data solution* [online] [visited on 2019-09-22]. Available from: https://labelbox.com.
51. *Tiny Go program for exporting/reading data from Firestore database and Labelbox.* [online] [visited on 2019-10-22]. Available from: https://github.com/martin-galajda/firestore-go-utilities.

52. *Python leads the 11 top Data Science, Machine Learning platforms: Trends and Analysis* [online] [visited on 2019-09-13]. Available from: https://www.kdnuggets.com/2019/05/poll-top-data-science-machine-learning-platforms.html.

53. *TensorFlow* [online] [visited on 2019-09-22]. Available from: https://www.tensorflow.org.

54. *Keras: The Python Deep Learning library* [online] [visited on 2019-09-22]. Available from: https://keras.io/.

55. *Tensorflow detection model zoo* [online] [visited on 2019-09-22]. Available from: https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md.

56. *Protocol Buffers* [online] [visited on 2019-09-22]. Available from: https://developers.google.com/protocol-buffers/.

57. REDMON, Joseph. *Darknet: Open Source Neural Networks in C* [http://pjreddie.com/darknet/]. 2013–2016.

58. *A Keras implementation of YOLOv3* [online] [visited on 2019-09-22]. Available from: https://github.com/qqwweee/keras-yolo3.

59. *Project Jupyter* [online] [visited on 2019-09-22]. Available from: https://jupyter.org.

60. *Google Colaboratory* [online] [visited on 2019-09-22]. Available from: https://colab.research.google.com.

61. *YOLOv3 on Open Images* [online] [visited on 2019-09-22]. Available from: https://github.com/radekosmulski/yolo_open_images/.

62. *Tensorflow detection model zoo* [online] [visited on 2019-09-22]. Available from: https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md/.

63. *Metrics for object detection* [online] [visited on 2019-09-22]. Available from: https://github.com/rafaelpadilla/Object-Detection-Metrics.

64. EVERINGHAM, M.; VAN GOOL, L.; WILLIAMS, C. K. I.; WINN, J.; ZISSERMAN, A. *The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Results*. Available also from: http://www.pascal-network.org/challenges/VOC/voc2012/workshop/index.html.

65. *Object detection on Open Images.* [online] [visited on 2019-10-22]. Available from: https://github.com/martin-galajda/object-detection.

66. *Chrome extension for marking content related images in websites.* [online] [visited on 2019-10-22]. Available from: https://github.com/martin-galajda/mark-content-related-images.

67. *Program with a command-line interface for importing data to Firestore.* [online] [visited on 2019-10-22]. Available from: https://github.com/martin-galajda/firestore-importer.