TECHNISCHE UNIVERSITÄT DRESDEN

Dep. of Computer Science Institute for System Architecture, Database Technology Group

Diploma Thesis

EVALUATION OF WEAK SUPERVISION AND TRANSFER LEARNING FOR GUI ELEMENT DETECTION

Eric Schölzel
Matr.-Nr.: 4051160

Supervised by:
Prof. Dr.-Ing. Wolfgang Lehner

and:
Dr.-Ing. Maik Thiele
Dr. Carsten Neise

Submitted on 30th September 2020

CONFIRMATION

I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, 30th September 2020

ABSTRACT

Deep learning based object detection algorithms evolve rapidly. The application of these object detection models for detecting Graphical User Interface (GUI) elements on screenshots might provide a powerful alternative to the error-prone automated analysis of technology-dependend interfaces such as the *Document Object Model* (DOM). However, the implementation of state-of-the-art object detection models for this task suffers from the lack of sufficient training data which is difficult and costly to annotate.

In this thesis, an object detection model is pre-trained on a huge web page screenshot data set that contains automatically obtained, but erroneous labels. After that, it is trained on a manually refined and considerably smaller high-quality data set. It is shown that this combination of *Weak Supervision* and *Transfer Learning* leads to a significant improvement of the prediction performance compared to naive training on high-quality data. This effectively reduces the amount of manually annotated data needed in order to reach a certain level of prediction quality.

CONTENTS

1 Introduction 11

2 Theoretical Foundations 13

2.1 GUI Element Detection 13

2.2 Machine Learning Paradigms 14

2.2.1 Weak Supervision 15

2.2.2 Transfer Learning 16

2.3 Object Detection Metrics 17

2.3.1 Intersection over Union 18

2.3.2 IoU and Confidence Thesholds 18

2.3.3 Precision and Recall 19

2.3.4 Average Precision 20

2.4 YOLO - You Only Look Once 22

3 Data Set Conception 27

3.1 Graphical User Interfaces 29

3.2 Acquisition of Low-Quality Data via DOM Analysis and Web Crawling 30

3.2.1 Crawling for Data 30

3.2.2 DOM Analysis 30

3.3 High-Quality data set Conception 33
3.3.1 Design Principles 33
3.3.2 Preventing Information Leakage 37
3.3.3 Classes and Sub-classes 39
3.4 Labeling Process 40
3.4.1 Reviews and Feedback 41
3.4.2 Handling real-world data 41
3.4.3 Pre-labeling 43
3.5 Pre- and Post-processing 43
3.6 Data Set Variants 44
3.6.1 LQDOM - Low Quality DOM Data Set 45
3.6.2 HQv1 - High Quality Data Set v1 45
3.6.3 HQv2 - High Quality Data Set v2 45
3.6.4 FINAL-TEST 46

4 Experimental Setup **47**
4.1 Environment 47
4.1.1 Darknet and YOLOv3 47
4.1.2 Hardware 48
4.1.3 Default Training Configuration 48
4.2 Label Variants 49
4.2.1 Binary 49
4.2.2 Multilabel 49
4.3 Multi-Stage Training - Combining Weak Supervision and Transfer Learning 50
4.4 Custom Evaluation Metrics 51
4.4.1 Selective AP 51
4.4.2 Weighted (Selective) AP 52
4.4.3 Total Recall 53

4.4.4 Comparison of Metrics between Multiple Experiments 54
4.5 Model Selection Process 54

5 Experiments **57**
5.1 Notation and Interpretation of Results 57
5.2 Baseline - Naive Training on Hand-labeled Data 58
5.3 COCO Pre-training 59
5.4 Binary Pre-training 60
5.5 Multilabel Pre-training 61
5.6 COCO- and Multilabel Pre-training 62

6 Discussion **65**
6.1 Effectiveness of Pre-trainings 65
6.1.1 Classes not included during Pre-training 66
6.1.2 Sub-classes 67
6.1.3 Summary 69
6.2 Limitations 69
6.2.1 Statistical Error Range 69
6.2.2 Metric Conception 70
6.2.3 Classification Limitations 71
6.2.4 GUI Type Limitations 72
6.2.5 Tree Structures 74

7 Conclusion **77**
7.1 Related Work 77
7.2 Outlook - Future Work 80
7.3 Summary 81

A Glossary **83**

A.1 Abbreviations 84

A.2 Machine Learning 85

A.3 R-CNN 86

B DOM Analysis Filters **89**

C Classes **93**

C.1 DOM-classes 93

C.2 Main-classes 93

C.3 All Classes and Sub-classes 93

C.4 Class Count Overview 94

D Experiment Permutations **95**

D.1 YOLOv3-SPP default preset without COCO weights 96

D.2 YOLOv3-SPP default preset with COCO weights 96

E Experiment Result Overview **97**

F Image Appendix **101**

F.1 Annotation Examples 101

F.2 Model Predictions 104

1 INTRODUCTION

Detecting elements of Graphical User Interfaces (GUIs) is a prerequisite for many software engineering tasks or techniques such as software testing or automated interaction with GUIs. Traditionally, detecting these elements involved accessing the source code or dedicated debugging interfaces that provide rendering information. For example, test automation aims to detect unwanted changes such as Buttons, Textfields or Images that are not functional anymore after the latest code changes. To detect such defects, automated test scripts for web pages may evaluate the *Document Object Model* (DOM) in order to find these elements and interact with them to test their functionality.

However, evaluating interfaces such as the DOM of web pages may give incorrect information about the occurrence of element types. Additionally, this approach strongly depends on the UI technology and availability of such an interface. Furthermore, User Interface Technologies evolve rapidly. Debugging interfaces can become obsolete quickly as well if not updated consistently. For other applications such as native software, there might be no DOM available, so the implementation of automated interaction has to rely on entirely different debugging interfaces.

Therefore, detecting GUI elements based solely on their visual properties provided by a screenshot using computer vision and object detection algorithms can provide an option that is not tied to the platform or UI technology. This can further be utilized in order to perform for example cross-platform application testing [1], automated device interaction without debugging access or GUI code synthesis based on a mocking screenshot [2, 3]. For some use cases, no debugging interface such as the DOM may be accessible at all.

Deep learning based object detection models such as R-CNN [4, 5, 6] or YOLO [7, 8, 9] provide state-of-the-art performance on challenging object detection data sets such as COCO [10, u3], which consists of over 200,000 labeled images. To achieve a similar prediction quality for detecting GUI elements on screenshots, a sufficient amount of labeled data is needed in order to train and evaluate these algorithms.

To approach this problem, White at al. [11] trained an object detection model on synthetic data that consists of randomly generated GUIs. However, the performance of models trained on synthetic data is limited as it is hard to generate synthetic GUIs that sufficiently reflect the properties of

real-world software designed by humans. The ReDraw [12] and Rico [13] data sets include mined GUI information of mobile apps. Annotations obtained through automatic methods may not always reflect the actual visual properties of an element. Additionally, desktop GUIs are often more complex than mobile GUIs as they utilize the larger screen size and more precise mouse input, which often results in a higher density of elements.

This work evaluates another approach. The focus of this work is the detection of elements that frequently occur as part of different types of desktop web pages. For that, the *YOLOv3-SPP* object detection model is used to train an end-to-end object detection pipeline for the detection of GUI elements on web page screenshots.

A huge data set of web pages has been collected and labeled automatically through web crawling and DOM analysis prior to this work. As labels provided by automated DOM analysis may be erroneous and noisy, a portion of this data set has been manually refined to provide the training data. An object detection model is pre-trained on the huge crawled and partly incorrect labeled data set (*Weak Supervision*) and then trained on the manually refined samples afterwards (*Transfer Learning*).

In order to evaluate whether this combination is suitable, three main unknown factors have to be taken into consideration.

First, how to efficiently design a manual refined data set. The manual annotation should follow dedicated principles that aim to normalize the annotation, correct errors and handle edge cases. To fully utilize object detection performance and avoid model confusion, these principles should base solely on visible properties of the elements, implicitly cover a wide range of GUI type functionality and account for the similarity of certain elements.

Second, the influence of noisy classification on the overall model performance. The data used for pre-training is obtained automatically. As web pages may utilize non-standard implementations as well as a variety of different frameworks and web technologies, automatic analysis of elements is error-prone. In the implemented analysis this work bases on, the element type of many elements is unclear, which might harm the overall performance. Additionally, some elements may be missclassified or not found at all. It has to be analyzed if the inclusion of this class information during the pre-training might confuse the model and therefore worsen the prediction quality compared to pre-training without class information.

Finally, the overall influence of the pre-training on the prediction quality. It has to be observed if this dedicated pre-training, considering its limitations, improves the results compared to a naive baseline as well as pre-training on a generic object detection data set.

In order to explain the object detection techniques and the metrics used for evaluation, Chapter 2 introduces the theoretical foundations of this work. Chapter 3 explores the data acquisition methods and pipeline for all involved data sets as well as the the manual refinement process. Chapter 4 provides information about the experimentation environment and its implementation. Chapter 5 introduces the individual experiments. Chapter 6 contains an overarching discussion of these results and the limitations of the approaches used in this work. Finally, Chapter 7 includes a brief reference to related work, sets this work in contrast and provides an outlook for future work.

2 THEORETICAL FOUNDATIONS

The purpose of this chapter is to introduce theoretical concepts, which is complemented by the Glossary in Appendix A. While the Glossary serves as a reference to more general Machine Learning terminology, the focus of this chapter is to introduce specific methods and metrics used in this thesis.

This chapter begins with the definition of the GUI element detection task in Section 2.1. After that, different machine learning paradigms are discussed in order to introduce *Weak Supervision* and *Transfer Learning* in Section 2.2. As this work aims to apply both paradigms for the task of GUI element detection, it is also discussed why the combination of both is suitable. To evaluate the implemented techniques based on these methods, dedicated evaluation metrics are discussed in Section 2.3. Finally, Section 2.4 introduces a brief overview about the *YOLO* object detector family. The *YOLOv3-SPP* [9, 14] architecture is used for evaluation as it is suitable for real-time prediction while maintaining an adequate trade-off between speed and accuracy. It also allows end-to-end training and *multi-label* prediction where each predicted object can have multiple classes assigned.

2.1 GUI ELEMENT DETECTION

The task of detecting object instances of specific classes in images is called *Object Detection* in computer vision. Detected elements consist of the corresponding class prediction as well as the *bounding box*. Each bounding box is a 4-tuple depicting a rectangle enclosing the object (Figure 2.1). Therefore, it includes information about position and size. Common annotation formats to denote bounding boxes are (x_1, y_1, x_2, y_2) or $(x, y, width, height)$. The former consists of just the edge positions, the latter denotes the upper left point coordinate as well as the width and height of the rectangle. In all cases, remaining information can be calculated by the provided annotation.

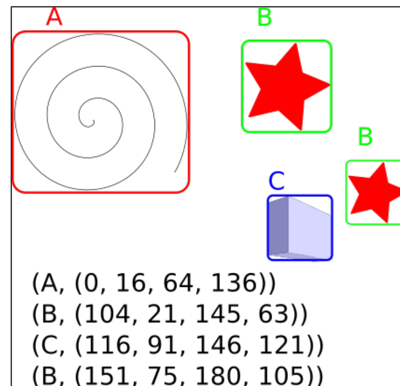


Figure 2.1: Object Detection aims to detect bounding boxes of object instances in images.

GUIs of arbitrary size and complexity are composed of elements like *Buttons* or *Textfields*. These elements are the fundamental building blocks. The GUI element detection task aims to detect these elements as well as their respective position and size, by relying solely on visual information provided by a screenshot of the rendered GUI. Therefore, detecting GUI elements is the application of object detection on GUI screenshots.

The focus of this work is the detection of elements that frequently occur as part of different types of web pages. Some element types are used more frequently in certain categories of GUIs, which are not covered by this work. However, the methods introduced in the following sections may be extended with reference to new element types in future work. This is discussed in Section 6.2.4 and Section 6.2.5.

State-of-the-art object detection algorithms often utilize deep neural networks for object detection tasks. They can be implemented using different machine learning approaches or techniques, which is introduced within the next section.

2.2 MACHINE LEARNING PARADIGMS

Depending on which type of labels are available for training and prediction, different *paradigms* can be applied.

In **Supervised Learning**, input data x as well as corresponding ground truth labels y are present. Tasks where labels are continuous values are named *regression* tasks. If labels consist of discrete values, this is called *classification*. Examples for supervised classification range from recognizing handwritten digits using the well-known *MNIST*[15] data set to classifying brain tumors [16] or noise for hearing aids [17]. In most cases, object detection is also implemented using supervised learning methods.

If a training algorithm operates on input data x without ground truth labels, this is called **Unsupervised Learning**. The objective is to find structures, similar groups or rules which are valid for the majority of the data. A simple example is the extraction of dominant colors or color palettes from images using k-means clustering [u1] or the calculation of *YOLO* anchor boxes which is introduced in Section 2.4.

However, the lack of labeled training data is a huge problem for many supervised learning problems. Due to this reason, there are paradigms and techniques that aim to reduce the amount of high-quality labels needed (Figure 2.2):

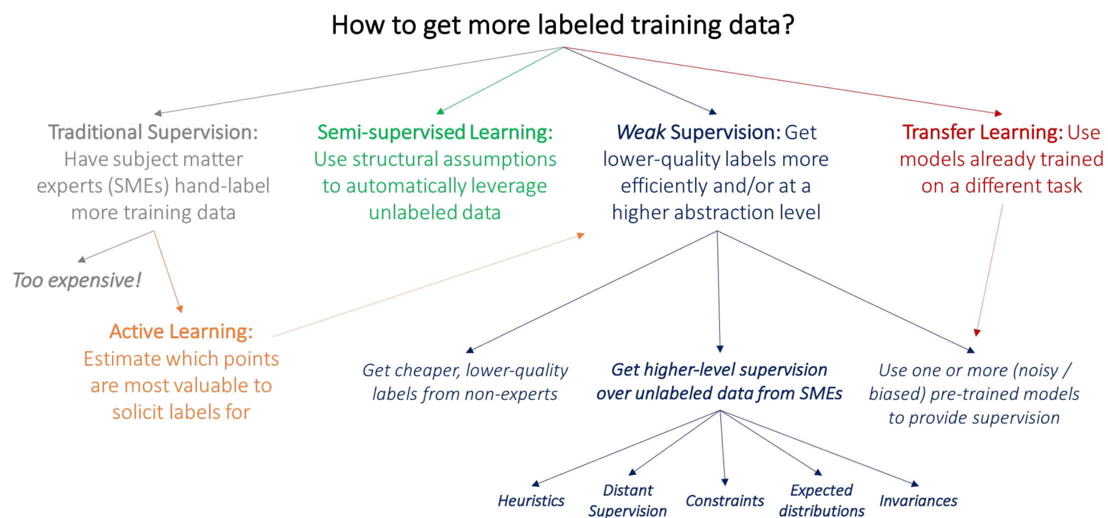


Figure 2.2: This figure summerizes different paradigms that can be applied to reduce the need for manually annotated data. *Weak Supervision* and *Transfer Learning* are implemented and evaluated for the detection of GUI elements in this work. Image Source: [u2]

- **Semi-supervised Learning** is a mixture of supervised and unsupervised learning techniques. In this case, only a portion of the data has been labeled. Unlabeled data can still be utilized in order to improve the model performance.
- **Active Learning** aims to estimate which data has the greatest value for the model to learn from. Therefore, the data which is annotated next is actively selected.
- **Transfer Learning** utilizes models pre-trained on another data set or task, which can significantly vary from the target data.
- **Weak Supervision** leverages low-quality labels which can sometimes be obtained easily. This can be used in combination with Transfer Learning.

Within this work, the combination of *Transfer Learning* and *Weak Supervision* has been implemented for the task of GUI element detection.

For this, an object detection model has been trained on a huge, but noisy data set using *Weak Supervision*. Afterwards, the pre-trained model has been utilized to continue training on a data set containing high-quality labels using *Transfer Learning*. Both paradigms are discussed seperately within the following Sections.

2.2.1 Weak Supervision

Obtaining a suitable amount of labeled training data can be costly. That hinders state-of-the-art supervised learning methods from being used for many real-world problems. Object detection research often aims at increasing an evaluation score at a given, well-known data set like

COCO [u3], which consists of over 200,000 labeled images. Building data sets to achieve similar results for other tasks can take huge efforts and human labeling.

However, there are use cases where inexactly or incompletely labeled data can be obtained easily. Leveraging these noisy, but often huge data sets is called *Weak Supervision* [u2].

There are three common types of Weak Supervision problems [18]. In *Incomplete Supervision*, relevant elements are not always labeled. *Inexact Supervision* is applied if only coarse-grained labels are available, for example per-image labels instead of per-object labels in an object detection task [19]. Finally, in *Inaccurate Supervision*, some labels may be wrong and not depict the real ground truth.

For the task of detecting GUI elements on screenshots, huge amounts of data can be easily obtained fully automatically by web crawling. Labels can be extracted by analyzing the *Document Object Model* (DOM) on-the-fly. This process and its limitations are discussed in Chapter 3. This approach is limited to applications where a DOM is accessible and the analysis is error-prone due to modern dynamic web frameworks and the variety of different implementations. Due to this reason, some labels may be incorrect or noisy and some elements may not be labeled at all. Therefore, learning from this type of data set is both *Incomplete Supervision* and *Inaccurate Supervision*.

Despite these limitations, training on this data can generalize a model that only needs a Screenshot, similar to how a human would perceive a GUI. As shown in this work, naive training on noisy data is quite limited when done seperately due to the reasons mentioned. However, it can have a significant impact on overall prediction quality if combined with transfer learning, which is discussed in the next section.

2.2.2 Transfer Learning

The first layers of neural networks often learn similar features even if the data sets or tasks are different. For example, it has been shown that the first layers of Convolutional Neural Networks often learn basic filters which are relatively independent of the specific data set [20]. This does not only take place when training using different data sets for the same task. It also occurs when considering different tasks like object detection or segmentation.

This phenomenon can be exploited for using features learned on a task using data set D_{source} to improve the results of a training for another task using data set D_{target}. This method is called *Transfer Learning* and has been widely adopted in the literature due to its versality [21][22].

Using Transfer Learning, it is possible to pre-train an object detection model on large-scale data set like *COCO* [u3]. After this, the already learned low-level features can be exploited in order to continue training on D_{target}. This has several advantages. Many weights will not receive major updates during the following training as important features have been learned before any sample of D_{target} has been processed. Therefore, it can greatly improve the actual learning effectiveness on D_{target}. That implies:

- It can greatly reduce the amount of training data needed in D_{target} in order to achieve a certain prediction quality and lead to faster *convergence*.

- Using the same amount of training data in D_{target}, prediction performance may be improved significantly
- Layers of the network can be *frozen* during the training on D_{target}. Freezing layers indicates that they will no longer be optimized during the training. Even though this can slightly reduce the final prediction quality, this can significantly reduce the computational power needed in that step.

Intuitively, the more distinct the tasks D_{source} and D_{target} are, the less effective the usage of transfer learning becomes.

Two variants of transfer learning are considered for the GUI element detection task. First, the usage of weights pre-trained on common object detection data sets. Second, the usage of weights pre-trained on a large amount of automatically annotated, but noisy GUI data, which can be achieved using *Weak Supervision* as introduced in the previous section.

It can be assumed that the difference between D_{source} and D_{target} is significantly smaller using the second option. However, since the implementation is not exclusive and both approaches can be stacked, a combination of both variants is examined as well. The details of the implementation used in this work is introduced in Section 4.3.

2.3 OBJECT DETECTION METRICS

Selecting proper metrics is crucial when evaluating the performance of any machine learning model, as it is the main indicator on how well a model performs. The performance of multiple models is directly compared using selected metrics.

The usage of common object detection metrics and their limitations are discussed in this section. Based on the *Intersection over Union*, the basic machine learning metrics *True Positives*, *False Positives* and *False Negatives* are defined in the context of object detection. After that, these basic metrics are utilized in order to derive the more sophisticated metrics *Precision*, *Recall* and $AveragePrecision$.

2.3.1 Intersection over Union

Also known as *Jaccard Index*, the Intersection over Uniun (IoU) is a statistic for measuring the similarity of two sets. It is used as metric for measuring the quality of bounding box regression in object detection tasks. For two sets A and B, it is calculated by

$$IoU(A, B) = \frac{A \cap B}{A \cup B}$$

The IoU of two bounding boxes can be calculated by dividing the overlapping area by the united area of both rectangles. This is visualized in Figure 2.3.

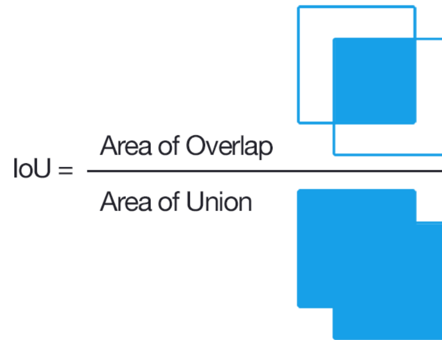


Figure 2.3: Bounding Boxes IoU. Source: [img1]

2.3.2 IoU and Confidence Thesholds

In contrast to classification, where a prediction only consists of the respective class, a prediction from an object detection algorithm consists of both bounding box and class predictions. More specifically, any object prediction A consists of bounding box prediction A^{bb}, a class prediction A^{class} and an optional confidence value A^{conf}. The latter depicts how sure the algorithm is that the predicted location and object class are correct.

By calculating the IoU and testing if it exceeds a threshold value, *True Positives*, *False Positives* and *False Negatives* can be defined in the context of object detection. For the following definitions, PR is the set of all predicted bounding boxes and GT is the set of ground truth bounding boxes. Following this, objects can be counted class-specifically.

- **True Positive** (TP^{class}): Since most predictions are not pixel-perfect, an *IoU threshold* $thresh^{IoU}$ must be chosen for the definition of TP detections. Additionally, a confidence threshold $thresh^{conf}$ can be applied to filter noisy low-confidence predictions. A specific detected object $A \in PR$ is considered where a ground truth object $B \in GT$ exists so that $IoU(A^{bb}, B^{bb}) \geq thresh^{IoU}$, $A^{class} = B^{class}$ and $A^{conf} \geq thresh^{conf}$. This is counted as *True Positive* detection for the respective $A^{class} = B^{class}$.
- **False Positive** (FP^{class}): A predicted object $A \in PR$ is considered where no object $B \in GT$ exists within the ground truth, so that the TP conditions explained above are fulfilled. This detection is counted as *False Positive* for the predicted class A^{class}.

- **False Negatives** (FN^{class}): Conversely, a ground truth object $B \in GT$ is considered where no object $A \in PR$ has been predicted so that the TP conditions are fulfilled. This is counted as *False Negative* for that ground truth class B^{class}.

If those class-specific metrics are available, the corresponding class-independent metrics can be defined by summing the class-specific occurrences (Formula 2.1, 2.2 and 2.3):

$$TP = \sum_{class} TP^{class} \tag{2.1}$$

$$FP = \sum_{class} FP^{class} \tag{2.2}$$

$$FN = \sum_{class} FN^{class} \tag{2.3}$$

For this work, $thresh^{IoU} = 0.5$ is applied.

Note that the *IoU Threshold* does not account for quality differences of predicted bounding boxes besides exceeding the provided threshold, which is a limitation for all metrics based on that.

2.3.3 Precision and Recall

Following the previous definitions, the metrics *Precision* and *Recall* can be calculated. For this, an additional confidence threshold $thres^{conf}$ can be applied where each prediction A is only included if $A^{conf} \geq thresh^{conf}$.

The *Precision* indicates the proportion of correctly predicted elements from all predicted objects. Therefore, a precision value of 1 indicates that all predicted objects are True Positives, independently of how many objects were predicted (Formula 2.4).

$$Precision = \frac{TP}{TP + FP} \tag{2.4}$$

The *Recall* is the proportion of all detected objects from all objects that should have been detected. A recall value of 1 specifies that all ground truth objects have been detected, independently of how many False Positives have been included in the prediction (Formula 2.5).

$$Recall = \frac{TP}{TP + FN} \tag{2.5}$$

Precision and Recall can be calculated *class-specifically* and *class-independently*.

The class-independent recall and precision is calculated as defined in Formulas 2.4 and 2.5, using TP, FP and FN as defined in Formulas 2.1, 2.2 and 2.3.

TP, FP^{class} and FN of Formula 2.4 and Formula 2.5 can be substituted by TP^{class}, FP^{class} and FN^{class} as defined in the previous section in order to calculate the class-specific $Precision^{class}$ and $Recall^{class}$.

2.3.4 Average Precision

The number of predictions as well as their quality often varies significantly for different $thresh^{conf}$ values during the calculation of the metrics defined in Section 2.3.2. To account for this, the *Average Precision* (AP) is a commonly used metric in object detection. The meaning and calculation process of the AP metric is introduced within this section.

Precision-Recall Curve

A weak estimator has to make more predictions with low confidence and precision in order to reach a high level of recall. Conversely, if the precision stays relatively high with higher recall values, this can be seen as an indicator for good model performance. Therefore, plotting a *Precision-Recall Curve* (PR Curve) for each individual class can be utilized for comparing models.

As an example for calculating the PR-curve, consider the predictions depicted in Table 2.3.4, which are ordered by confidence. TP detections are marked by ✓, FP detections are marked by X. Precision and Recall for each individual point are then calculated row-wise starting from the top row, considering the total count of all previous TP/FP until the current row. Therefore, Precision decreases with every FP and Recall increases with every TP.

Pt	Conf	TP?	Precision	Recall
R	0.95	✓	1.00	0.07
Y	0.95	X	0.50	0.07
J	0.91	✓	0.67	0.13
A	0.88	X	0.50	0.13
U	0.84	X	0.40	0.13
C	0.80	X	0.33	0.13
M	0.78	X	0.29	0.13
F	0.74	X	0.25	0.13
D	0.71	X	0.22	0.13
B	0.70	✓	0.27	0.20
G	0.67	X	0.27	0.20
...	...	...	...	...

Table 2.1: PR-curve (values). Source: [23]

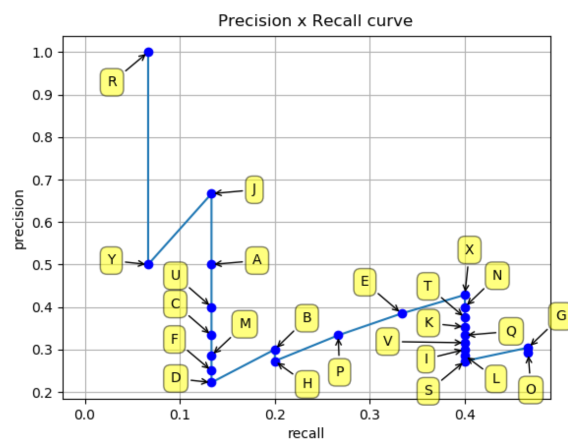


Figure 2.4: Calculation of the PR-curve (plot). Image Source: [23]

Area under PR-curve

As shown in Figure 2.4, plotting the PR curve often depicts a zigzac pattern with the precision going up and down as recall increases. Therefore, comparing different plots visually is not always

suitable. However, calculating the area under the PR-curve while smoothing the zigzac curve by using an interpolated precision of relevant points results in a numeric value that can be used in order to compare the quality of different model predictions. This metric is called *Average Precision* (AP).

Calculating the area under the PR-curve results in the AP for a specific class. The AP is calculated by interpolating through all points n (Formula 2.6 and Formula 2.7):

$$AP = \sum_{n} (r_{n+1} - r_n) p_{inter}(r_{n+1}) \tag{2.6}$$

with

$$p_{inter}(r_{n+1}) = \max_{\tilde{r}:\tilde{r} \geq r_{n+1}} p(\tilde{r}) \tag{2.7}$$

r_n — Recall level of n-th point (cf. n-th row of Table 2.3.4)
$p_{inter}(r_{n+1})$ — Interpolated recall value for recall level r_{n+1}
$p(\tilde{r})$ — (Not interpolated) precision value at given recall $\tilde{r}$

A visualization of this interpolation is depicted in Figure 2.5.

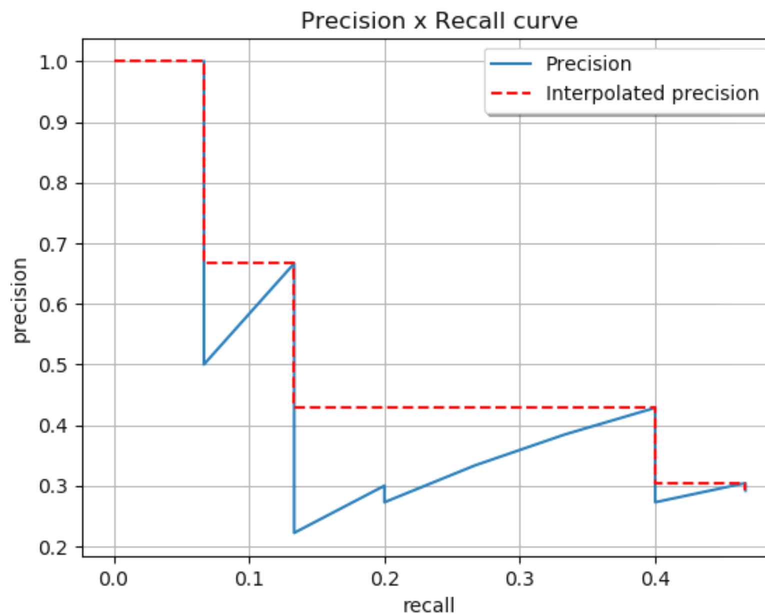


Figure 2.5: PR curve utilizing precision values interpolated for each occuring recall value as defined in Formula 2.6. The interpolation is used in order to smoothen the zigzac pattern of precision values measured for specific recall values. Image Source: [23]

Note that the precision and recall values are class-specific as defined in Section 2.3.2.

Instead of the defined interpolation performed for all points as depicted in Formula 2.6, other variations of this interpolation can be applied as well. Examples are the 11-point interpolated AP used in the Pascal VOC 2008 challenge [u4] and the 101-point interpolation utilized by the COCO evaluation. A more detailled breakdown of those variations can be found in [23]. However, this work uses the AP interpolated for all points as defined.

AP_x denotes that in order to calculate recall and precision, an IoU Threshold of x is used. In this work, $AP_{0.5}$ is utilized. Therefore, any reference to AP implies the usage of the defined $AP_{0.5}$ metric.

Mean Average Precision (mAP)

After calculating each class-specific AP, the arithmetic mean of each class AP is the *mean Average Precision* (mAP) which is defined in Formula 2.8. The mAP is commonly used to evaluate the performance of models on data sets like COCO. Note that COCO utilizes a 101 point-interpolated AP instead of interpolating over all points.

$$mAP = \frac{\sum_{i=1}^{c} AP(i)}{c} \tag{2.8}$$

c — : Number of Classes
$AP(i)$ — : Average Precision of Class i

2.4 YOLO - YOU ONLY LOOK ONCE

Object detection consists of two challenges: localization and classification. *Two-stage object detectors* like the R-CNN architecture [4, 5] seperate these two tasks (cf. Appendix A.3). After generating region proposals for potential objects, each region is classified as second step. However, two-stage approach comes at the cost of performance. *One-stage object detectors* try to directly predict object locations as well as their classes. The difference in speed is why one-stage object prediction is crucial in order to achieve real-time prediction performance.

Predicting UI elements in (near-)real-time allows for the observing GUI operations such as scrolling, sliders or other interactive elements. Additionally, a fast architecture can be considered to have a performance headroom high enough for further increases in resolution. In contrast to object detection problems where many images only consist of a few objects to detect, GUI screenshots often contain a very high amount of smaller objects. A low resolution may result in these objects being barely detectable. Furthermore, predicting more images within the same time frame also implies better scalability, for example running multiple GUI interaction scripts for different applications at the same time where a single GPU is sufficient for image processing.

The YOLOv3-SPP architecture has been used for the majority of evaluations in this work. It allows a sufficient trade-off between speed and accuracy. It further implements features like *multi-scale* training, where the network is being trained on multiple resolutions. This may be particularly useful for detecting elements on GUI screenshots with different resolution. Additionally, it allows *multilabel* prediction, where each detected object can have multiple classes assigned. This is excessively utilized in order to implement hierarchic labeling, which is introduced later in Chapter 3.

Note that the methods introduced in this work are not dependent on this specific architecture, although the specific results or implementation details may vary using another architecture.

You Only Look Once (YOLO) [7, 8, 9] is a real-time object detection architecture family. As the name indicates, YOLO predicts objects from input images within a single pass by framing object detection as regression task. *YOLOv3-SPP* is a variant of *YOLOv3* which is the third major version of the *YOLO* architecture.

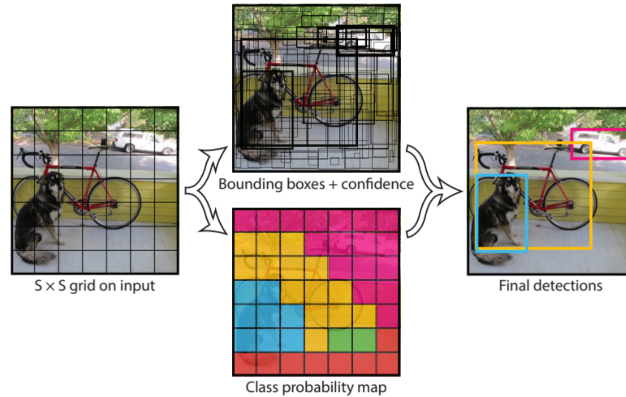


Figure 2.6: Grid cell-based object detection.
Image source: [7]

YOLO divides the input image into an *SxS* grid. A certain grid cell is responsible for all objects which center is located inside the cell. Each cell predicts bounding boxes as well as corresponding *confidence scores* (Figure 2.6.

The *confidence score* as defined in Formula 2.9 indicates how confident the model is about the presence of the predicted object and the assumed accuracy of the bounding box prediction[1].

$$confidence = Probability(object) * IoU(groundtruth, prediction) \tag{2.9}$$

Instead of directly predicting the location of bounding boxes, offsets from *Anchor Boxes* are predicted which serve as bounding box priors. To determine those priors, *k-means* clustering is performed on the bounding boxes of the training data set before the actual model training. To improve the performance for objects of different sizes, *YOLOv3* predicts bounding boxes at three different scales. For each scale, three anchor boxes are used and three bounding boxes are predicted per grid cell.

Each bounding box prediction consists of the five coordinates t_x, t_y, t_w, t_h and t_o. Using these predictions, the actual bounding box location b_x, b_y, b_w, b_h (x, y, width, height) can be calculated (Figure 2.7 and Formula 2.10).

[1] Note that $Probability(object) * IoU(groundtruth, prediction)$ is the *ideal prediction* YOLO is trained to predict for the $confidence$ value. It is not the calculation of $confidence$ itself, since $groundtruth$ is not available during prediction.

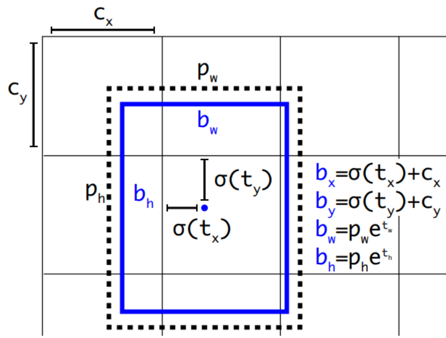


Figure 2.7: Anchor Box predictions. Image Source: [8]

$$
\begin{aligned}
b_x &= \sigma(t_x) + c_x \\
b_y &= \sigma(t_y) + c_y \\
b_w &= p_w e^{t_w} \\
b_h &= p_h e^{t_h} \\
Pr(object) * IoU(b, object) &= \sigma(t_o)
\end{aligned}
\tag{2.10}
$$

t_x, t_y — Offset from *Anchor Box* center
c_x, c_y — Cell position as offset from the top left image corner
p_w, p_h — Bounding Box prior width and height
σ — Logistic activation function which bounds values inside the range (0, 1)

Additionally, *multi-scale training* is performed where the input resolution randomly changes in a certain range. This forces the network to become robust against input images with different resolutions. Every 10 batches, the input size changes and the network is resized to fit the new input resolution[2].

In contrast to earlier versions, *YOLOv3* uses *independent logistic classifiers* instead of *softmax* activations for class predictions as depicted in Figure 2.8. This enables the usage of *multilabel classification*, where one object can have multiple classes assigned. For example, one object can have assigned both *Human* and *Woman* classes.

[2]Intuitively, this technique may be particularly useful for detecting user interface elements as input screenshots may vary in resolution.

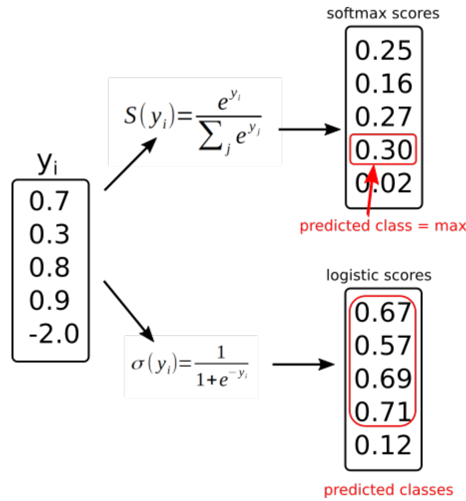


Figure 2.8: Left: Outputs of the last layer without activation.
Top: Softmax scores. As softmax results in a probability distribution, the maximum value is the predicted class, which does not account for multi-label classification.
Bottom: Using independent logistic classifiers, the resulting scores depict the predicted confidence independently for each class. This allows for one object being assigned to multiple classes with high confidence. Classes can be filtered using a confidence threshold.

As stated earlier, *YOLOv3-SPP* is a variant of *YOLOv3* and the main architecture utilized in this work. In *YOLOv3-SPP*, the use of multi-scale features is further improved due to the use of *Spacial Pyramid Pooling* [24, 14] (SPP). As some layers such as classification layers at the end of the network require fixed input dimensions, the networks require an input image of a fixed size (for example 224x224). As it is trained for that specific resolution, this might reduce the prediction performance for arbitrary image sizes. By using a SPP layer as depicted in Figure 2.9, a fixed length representation is calculated from the input to this layer, which allows the use of arbitrary size images without changing and re-training the network architecture.

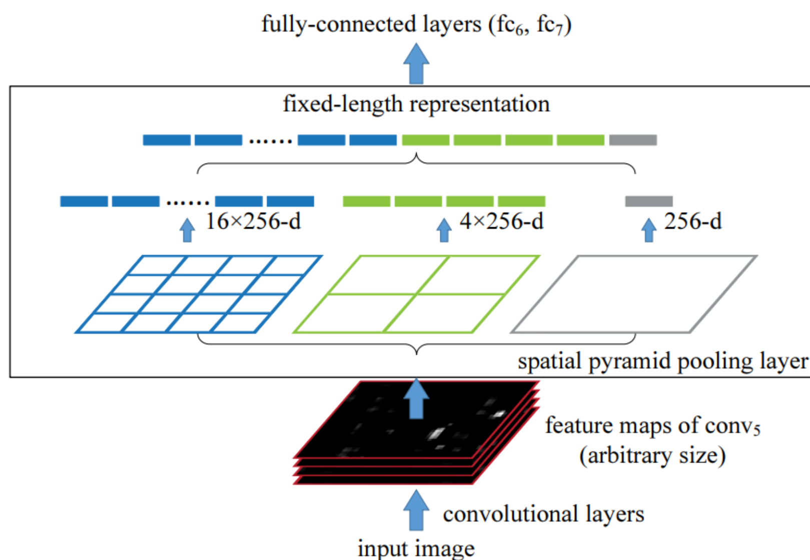


Figure 2.9: Spatial Pyramid Pooling calculates a fixed-length representation of variable input dimensions. Image Source: [24]

3 DATA SET CONCEPTION

When evaluating the performance of certain architectures, metrics, data augmentation or other techniques, data sets such as COCO [u3] are a common choice. They act as the benchmark for object detection algorithms and allow the performance comparison of different methods. However, applying state-of-the-art algorithms to solve real world problems often suffers from the lack of suitable data sets including sufficient labeled data with the required level of quality.

The chapter starts with a brief overview about modern GUI types in Section 3.1. This section discusses why web pages can be seen as representative for a variety of common GUIs and provide an easy-to-access raw data source for detecting GUI elements.

Creating a data set for a specific task may have different requirements, according to the task, type of data and data source. Within this work, two types of data aquisition considered. Both are different, but they are not independent.

The first data aquisition method is the collection of web pages through web crawling, where labels are aquired automatically through DOM analysis. Since this approach has several limitations introducing label noise, it is referred to as *LQDOM* (low-quality DOM-labeled data set). This data set already existed prior to this work and is used as training data for *weakly supervised pre-training*. In order to discuss the limitations as well as their implications, Section 3.2 covers an explanation of the data pipeline and the analysis procedure.

The second data aquisition method manually refines selected pages collected by the first approach. The conception of high quality data sets based on the aquired low quality data is discussed in Section 3.3. Based on that, Section 3.4 covers the human labeling process that resulted in two data set versions *HQv1* and *HQv2* as well as a test set named *FINAL-TEST*.

Pre- and Post-processing in order to preparate the data for object detection is discussed in Section 3.5. Finally, Section 3.6 provides an overview about the data set variants as well as the size of each data set.

Figure 3.1 provides an overview about the data pipeline that is introduced in this section and serves as reference.

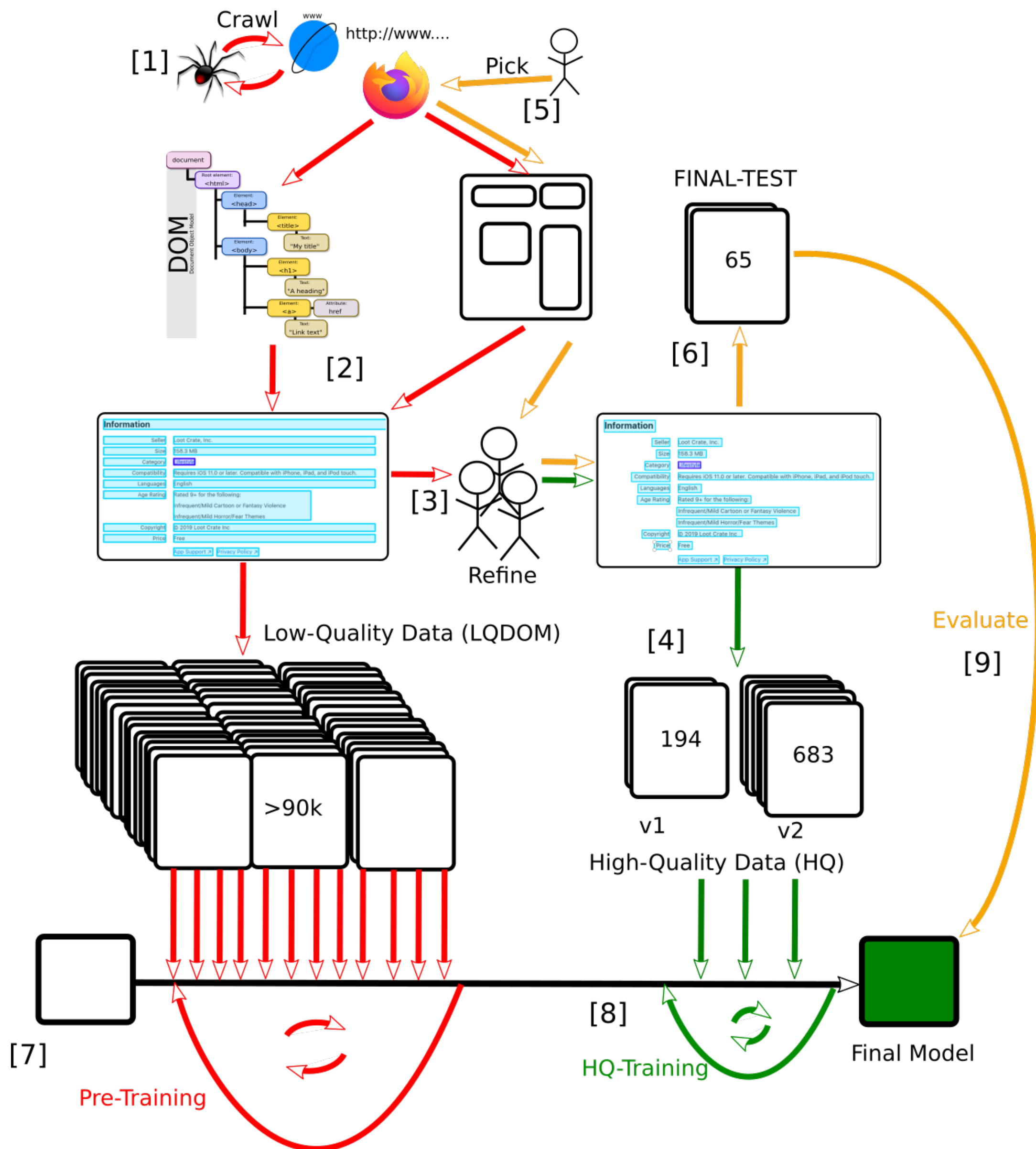


Figure 3.1: Data and training pipeline introduced in this section. Training cycles depict training over multiple epochs. Red: Low-Quality data. Green: High quality data sets. Yellow: Extra test set for final evaluation containing manually selected screenshots.
Included Image Sources: [img2, img3, img4, img5, img6]

References:
[1] - Section 3.2.1
[2] - Section 3.2.2
[3] + [4] - Section 3.4
[5] + [6] - Section 3.3.2
[7] + [8] - Section 4.3
[9] - Chapter 5 and Chapter 6

3.1 GRAPHICAL USER INTERFACES

Human-Computer-Interaction has developed significantly over the years. While terminal usage is still a common tool for IT professionals as well as enthusiasts, interacting via on-screen icons, menus and similar graphical components using a pointing device has become standard for decades.

The design of average Graphical User Interfaces (GUIs) has changed rapidly over time, including a wide variety of appearances. However, due to the wide distribution and advantages of web frameworks, there is a trend of unification. For the purpose of this work, GUIs can be classified into one of the following categories[1]:

- **Web GUIs**: While early web pages mainly consisted of just bare text and hyperlinks, modern HTML5 [u5] pages utilize seamless and responsive front-ends. Besides traditional web pages, modern web frameworks allow the creation of web applications that can be used like a native desktop application, without the installation of a dedicated client. Examples for this type of web apps are Google Docs [u6], Overleaf [u7], Discord [u8] and Slack [u9].
- **Desktop Applications**: Native desktop applications are rendered by the underlying window manager of the operating system. Being not tied to the restrictions web pages have inside a browser, they can traditionally be arbitrary unique and complex. However, implementing a clear, streamlined and straightforward GUI is important in modern software design. This has been further amplified by the demand for cross-platform GUIs as well as touch screen support. Frameworks like *Electron* [u10] allow the development of GUIs for desktop applications using modern HTML technologies. Well-known examples utilizing *Electron* are Visual Studio Code[u11] or the desktop clients of Discord and Slack.
- **Mobile Apps**: Despite differences in operating systems and their respective rendering and detailed design patterns, interaction using mouse and keyboard is still quite similar comparing these platforms. However, the distribution of smartphones (and mobile apps respectively) introduced a necessity for other design choices. While traditional desktop applications often rely on a lot of textual information and small elements, high pixel density on small screens that rely solely on touch inputs require other approaches. Additionally, mobile App GUIs need to be streamlined, focused and intuitively understandable by a wide variety of users. To provide a unified user experience throughout different apps, guidelines such as the *Material Design* for Android[u12] have been created. Additionally, technologies such as *Progressive Web Apps*[u13] or frameworks like *Flutter*[u14] further amplify the unification of GUI appearance as they allow the implementation of HTML-based GUIs for mobile apps.
- **Other native non-standard GUIs**: This category includes for example video games or UEFI GUIs which can greatly differ from the other types of user interfaces. Despite a particular interface can look similar to more traditional GUIs within certain aspects as for example menu buttons, these GUIs can significantly vary. Therefore, they are not a focus of this work.

[1] These categories are not an extensive definition nor a study of GUI types. Rather, they are intended only to be a rough seperation of the most common GUIs for applications a typical user might encounter.

The differences between mobile or native applications and the appearance of web pages tends to vanish in favor of platform-independent technology and unified look. As web pages provide an easy-to-access data source for a wide variety of complex GUIs, the recognition of web page elements is the main focus of this work.

Note that this may not account for some element types primarily found in mobile apps such as sliders or switches. However, the general principles such as the data set design discussed in Section 3.3.1 can still apply with slight modifications such as new element classes.

3.2 ACQUISITION OF LOW-QUALITY DATA VIA DOM ANALYSIS AND WEB CRAWLING

Screenshots of web page GUIs can be obtained automatically via web crawling. To obtain information about occurring elements and their respective element type, location and size, the Document Object Model of the rendered page can be analyzed on-the-fly. This combination of both techniques can be used in order to obtain a large amount of labeled training data. The resulting data set is named *LQDOM* and consists of 93,896 full-size web page screenshots that have been used for training.

The following sections provide an overview about the analysis and crawling process. This includes basic information about the implementation as well as advantages and limitations of this approach.

3.2.1 Crawling for Data

Over 100,000 web pages were crawled around June to August of 2019 prior to this work[2]. To avoid bias towards a few, popular pages, only a small, varying amount of pages per domain were crawled.

Per sample, a stitched high-resolution screenshot of the entire page, annotations obtained from the DOM analysis and the corresponding URL have been saved.

This is the first step of the pipeline depicted in Figure 3.1.

3.2.2 DOM Analysis

Position and size of GUI Elements are often determined by hierarchically arranged tree structures. For HTML or other XML-representable GUIs, this structure order can be represented and manipulated using the *Document Object Model* (DOM) [u15, u16, u17].

[2]Since some screenshots are processed manually by several people during the pipeline, it is sensible to apply some rules such as blacklist-based filtering of NSFW content.

Analyzing the DOM tree provides information about the current state of the respective rendering, including element types and positions. It is utilized by web developers, software testers as well as web crawlers or accessibility frameworks. This is the second step in the pipeline depicted in Figure 3.1.

Advantages

For GUI element detection, automated DOM analysis can be used in combination with a web crawler to extract huge amounts of data into a data set. The data can further be used to train an object detector that detects elements on screenshots without having to rely on the DOM. This approach has several advantages:

- Technology-dependent data collection for training a more technology-independent algorithm: While the DOM analysis relies on the availability of the DOM information, the input of an object detector consists of just a screenshot of the rendered GUI. Therefore, the trained detector may work on GUIs without the availability of the respective DOM, which also accounts for non-web page GUIs and outliers.
- Automation: The analysis can be performed by an algorithm. Therefore, human intervention is not needed when collecting the data.
- Scalability: When combined with a web crawler, data of thousands of different web GUIs can be collected. The runtime of the data collection is $\mathcal{O}(n)$ where n is the number of crawled pages.

Implementation

For analysing the *LQDOM* data, the *Selenium* framework [u18] is utilized. Selenium is a web automation framework that allows remote control of web browsers such as Google Chrome [u19] or Mozilla Firefox [u20]. It allows to access debugging information of the current page, including rendering-based data. For example, it can be checked if a certain element node of the DOM tree is displayed and which position it belongs to.

Since this information is not always correct or sufficient, a few rules can be applied to filter relevant elements. The implemented rule set already existed prior to this work and has been included in Appendix B. It may be refined in future work. Note that the applied rules are always a trade-off between *Recall* and *Precision* of this method, as web elements are implemented in many variations. Too restrictive filters result in too few elements being found. If they are too permissive, this results in False Positives. A trade-off is achieved by applying restrictive filters for labeling an element as a specific class such as *Button* or *Textfield* and more permissive filters for finding elements with the class being unclear (*Interactable* only).

Limitations

Automated DOM analysis is limited. Obvious limitations are that it is only possible when a DOM is available and analyzing elements is only feasable for defined element types like *Buttons*

or *Textfields*. For example, it is not possible to distinct between different types of well-known and established *Icons*, since their functionality often can not be reliably derived from the DOM. However, from the persepective of a user, their functionality is depicted by the corresponding graphic[3].

Apparent properties of DOM elements do not always correspond to their visual properties. Conversely, a certain rendered element like a *Textbox* may not be identified as such by using automated DOM analysis only. Furthermore, elements may appear in the DOM that significantly differ from what is rendered on the screen. This may occur due to the use of dynamic JavaScript that controls the actual behaviour of an element. Figure 3.2 depicts such an example.

However, if it is known which specific visual element to search for, e.g. by a web developer debugging a website, it is possible to retrieve the corresponding element from the DOM. Verification of the chosen DOM element can be achieved by comparing the respective properties to the given visual element such as position and size.

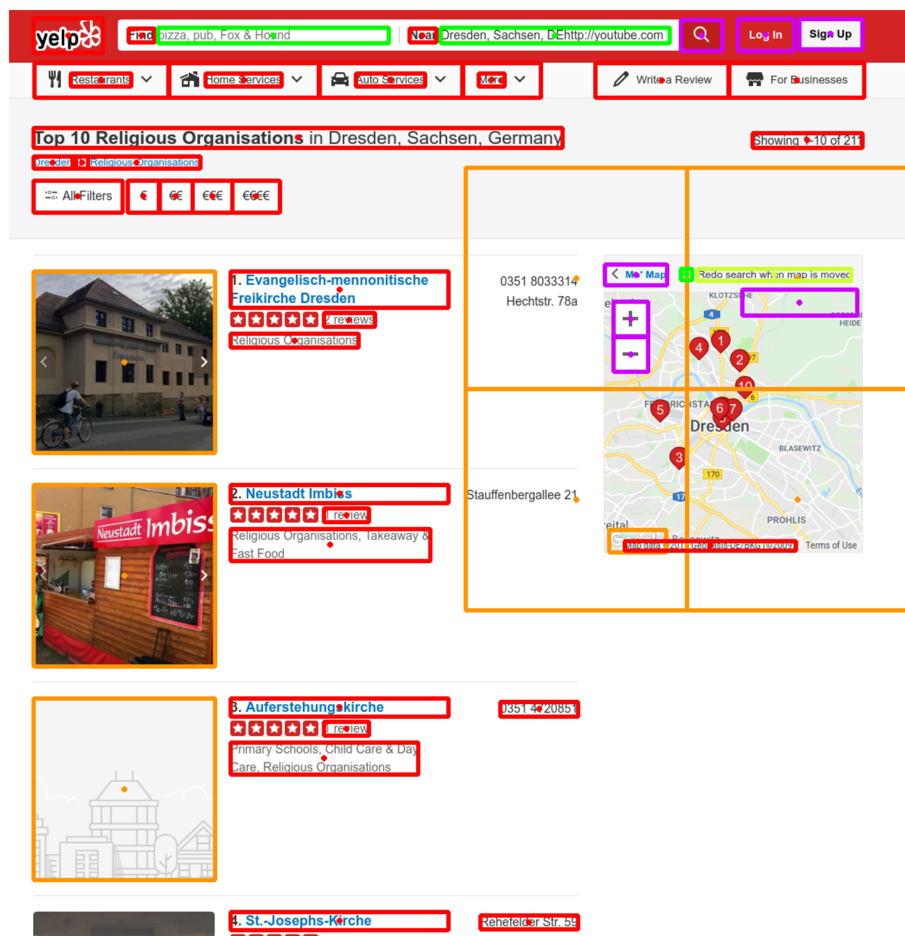


Figure 3.2: Information from the DOM is not always reliable as many edge cases occur. While not-real images appear in the DOM (orange), the rating stars are not recognized as elements by the automated analysis. The text boxes at the top (green) appear to be much smaller in the DOM as they are visually. Image Source: DOM analysis of [img7]

[3]This allows an object detection algorithm to distinct between different icons as well, which can then be used to predict functionality.

.

Despite its limitations, huge amounts of data can be automatically obtained when combining DOM analysis with a web crawler, which is introduced in the next section.

3.3 HIGH-QUALITY DATA SET CONCEPTION

As the automated collection of labeled data is limited, a high-quality data set labeled by humans is introduced, which is then used to perform a multi-stage training with a weak supervision approach. Since the training and evaluation results heavily depend on the quality of this data set, the conception of the data set and labeling process which is introduced in this chapter are key factors.

Using manual refinement, three data sets have been created: *HQv1* (244 web pages) and *HQv2* (844 web pages), which have been used for training and initial evaluation. Additionally, *FINAL-TEST*(65 web pages) has been created as another indicator for real-world performance.

3.3.1 Design Principles

When creating a data set, the label representation - or *what* should be detected and *how* the detected elements are annotated - is an important design choice as it emulates the considered *ideal detection*. This ideal detection needs to be clearly formulated, which can be quite trivial for many object detection tasks. As an example, consider a person bounding box detector. A data set could just have one class (*Person*) and each object that should be detected (every human in a photo) would have a bounding box specified by four points. However, if distinct parts of the human body should be detected seperately - such as *Head* or *Upper Body*, class representation as well as annotation have should be created with that in mind.

Although this may seem obvious for such cases, this is not necessarily the case for GUI element detection. While the general structures of a person or a street sign is similar in most cases, GUIs of different websites/applications may differ significantly and even design principles have changed clearly over time. Trying to distinct between all types of GUI element in modern GUIs[4] is not always an easy task due to different GUI design approaches as well as edge cases that may occur. This is not only the case for the object detector itself. Human labelers that annotate the data need definitions that are as straightforward as possible, which is in direct contrast to complex and sometimes messy real-world GUI designs. Additionally, for some kind of elements there may be only a small amount of samples, which amplifies imbalancing problems.

To anticipate these problems, the following design principles for creating a dedicated data set were deduced:

- **Few, but meaningful classes**: Focus on as few classes as possible while still being able to distinct between the majority of important elements.

[4] As an illustration, there is a non-comprehensive list of different element types at `https://en.wikipedia.org/wiki/List_of_graphical_user_interface_elements`

- **Sub-classes**: *Icon*s are one of the most important element categories. A vast amount of different types of elements like popular menu buttons such as *sandwich* or *gear* symbols or *floppy disk* save buttons are basically a subtype of icon. Subclasses are therefore named *Mainclass.Subclass*. This also allows for class annotations like *Icon.Arrow.Right*.
- **Nested Objects, atomic elements**: More complex or unique types of elements can be split up into smaller parts, while grouping together smaller parts again may be done via post-processing. This implies that for example *Texts* and *Icons* on buttons should be annotated as well. Besides the dividing of more complex elements into essential elements and classes, this atomic annotation allows a more robust detection of elements like menu tabs without further dedicated samples. Another advantage is that even if a button is not detected as such, the model may still correctly detect the text on the button. Additionally, the model is trained to detect for example text in almost any element constellation, which may improve overall robustness. An example is depicted in Figure 3.4.
- **Over-annotation**: During annotation, more information can be annotated than finally used. Since the annotation can be done hierarchically, subclasses such as *Icon.Arrow.Down* and *Icon.Arrow.Right* may be grouped together into just Icon.Arrow later on. This allows further investigation of different labeling detail levels without re-labeling data, while not introducing significant overhead during annotation. Additionally, it allows for later experimentation such as implementing overlay recognition subsequent to this work.
- **Annotating *candidates* as-seen**: Especially on webpages, images such as logos, ads or even screenshots of other GUIs may be included frequently. Such chunks may be detected as images when analyzing the DOM or identified as such by experienced web developers. However, an object detection algorithm performing solely on screenshots does not have access to that information and does not have experience with internal functionality of websites or GUIs in general. As an example, the distinction of embedded ad banners depicting faked buttons as shown in Figure 3.3 may be learned implicitly. However, this could worsen the detection of real buttons in interactive environments if the model falsely assumes it to be an ad. As the final prediction of a deep object detection model is the sum of very complex filters, this might result in unforeseen bad predictions in some cases that are barely explainable. It can be hard to reproduce what *implicit* rules a model may have learned, therefore it is suitable to reduce the need to learn *implicit knowledge* per-design. For this reason, elements are annotated *as-seen*: if an element appears to be a button visually, it is annotated as button, even if it is just part of an included image to prevent confusion. Each detected element is considered a *candidate*. For an implementation using the model trained like that, validation can be done at runtime by interacting with the GUI[5].
- **Minimal Bounding Boxes**: GUI object structures are often built tabular. As an example, menu bars include multiple elements with the same bounding box size that are placed along a horizontal or vertical line. However, it is often hard or not even possible to determine the exact boundaries as given by the GUI-defining code based purely on visual information. To remove noise and make detection easier, minimal bounding boxes are annotated instead. An example is shown in Figure 3.5. This includes for example tabular menu cells where there is no explicit visual border. Minimal bounding boxes are defined as the smallest possible rectangle to contain every pixel of a *visible* element such as text, without any implicit structural information.

[5]This does not necessarily imply clicking on element. A human would for example hover over a potential hyperlink and watch if the cursor changes, which is an interaction as well.

- **Text as blocks**: Hyperlinks are a crucial part of web pages and must be handled by a GUI element detection algorithm as well. However, it is often not obvious if a given text is a hyperlink or just normal text. The well-known hyperlink formatting may naively be assumed as indicator for clickable hyperlinks. However, this is not always true and can be misleading. Even though the *candidate* definition intentionally encourages false positives, the opposite may still be problematic, as many hyperlinks are not formatted that way. Often, links appear just like normal text. In some cases, the context may indicate a hyperlink for a user. Without interaction such as hovering over potential links, distinguishing can be hard even for humans. To circumvent this problem, text is labeled blockwise. A text block is defined as connected text of one paragraph with the same font, size and formatting. The ideal prediction for a potential hyperlink within continuous text is considered a textblock element inside another textblock element. Examples are shown in Figure 3.6.

The focus on essential, atomic main classes additionally allows for easier definition of each class and therefore an easier annotation process (see Section 3.4). Furthermore, it reduces the distance of the web GUI element detection task to other applications such as native desktop applications or mobile apps. Figure 3.7 depicts a web page annotation following the defined principles. Additional examples are provided in Appendix F.1.

Figure 3.3: Element Candidates (extreme example): Guessing the element type based on visual information only can be hard for humans as well. This is exploited in a scamming technique [u21] which uses ads disguised as buttons in order to distribute malicious software. Image Source: [u21]

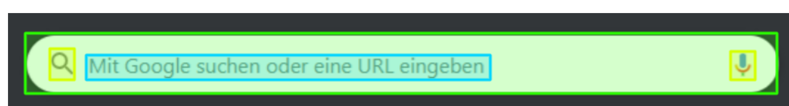
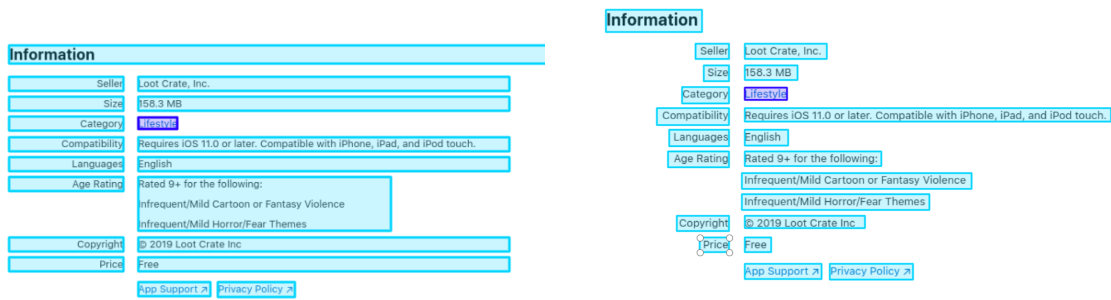


Figure 3.4: Atomic Elements: The microphone icon has its own functionality. Therefore, annotating just a searchbar would not be sufficient. The model is trained to detect the text and the icons even with the textbox as background. Image source: [img8] (cropped).



(a) Before: Boundaries as assumed after DOM analysis. Compared to minimal bounding boxes, the boundaries are barely determinable based on visual information only.

(b) After: Minimal bounding boxes circumvent the problem depicted left. 3.5.

Figure 3.5: Minimal Bounding Boxes. Image Source: DOM analysis/annotation of [img6]

(a) Blocks of text of the same paragraph with the same font, size and formatting. Image Source: [img9] (Crop)

(b) Hyperlinks are annotated as text-inside-text. Image Source: [u11]

Figure 3.6: Text Blocks.

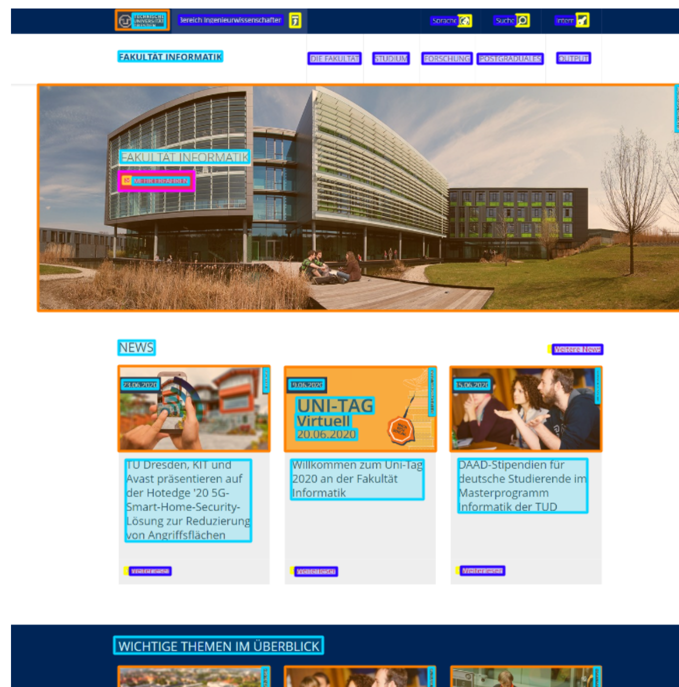


Figure 3.7: Sample excerpt of annotated webpage. Image Source: [img10]

.

3.3.2 Preventing Information Leakage

Since training is evaluated on data not included during training (cf. Appendix A.2), tuning for example hyperparameters in order to achieve a better score on the evaluation data implicitly gives the model feedback based on the evaluation data during this cycle.

Therefore, specific parameters that perform better on the evaluation data do not neccessarily perform better on real-world data afterwards. This phenomenon is known as *information leakage* in machine learning. This section covers two different types of leakage relevant for this work. It also covers approaches that try to minimize their influence in order to measure the model performance as realistic as possible.

Validation and Test Sets

The implicit feedback when tuning training parameters as described above is a common problem in supervised learning. To prevent the type of leakage, it is possible to hold back a certain amount of data during the optimization process. After all hyperparameters and methods are determined and the best performing models are selected, the score on the held back test set is calculated as very last step of the whole researching process. The evaluation data used during hyperparameter optimization is referred to as *validation set (HQv1/HQv2-VAL)*, the hold back data is called *test set (FINAL-TEST)*.

However, this may have some drawbacks depending on the data. If the final test set consists of too few samples, the evaluation will not be meaningful. This further worsens if the data is imbalanced, since some classes may already have few samples without held back test data. Therefore, the final test set has to contain enough data as well. However, labeled data may be expensive to obtain and holding back samples during the main training process further reduces the amount of available data for training and validation.

DOM Analysis Leakage

Another problem arises due to the data aquiration and multi-step training. The entire *LQDOM* data set includes automatically crawled data. A subset of this data is revised by humans in order to create the *HQv1/HQv2* data sets. This implies, that a specific web page screenshot included in *HQv1/HQv2-VAL/FINAL-TEST* may have been already present in *LQDOM* and therefore in training data. Since the subset that is manually refined during labeling is sampled randomly from *LQDOM*, it is possible for information leakage to occur, making validation and testing inconsistent.

This is not limited to the exact pages. During crawling, multiple pages of the same domain may be included. Since pages of the same web domain usually have the same overall appearance, the leakage problem extends to these as well. Therefore, in order to counter leakage during this step, the train/validation subsets for all data sets can be chosen using the algorithm shown in listing 3.1:

Listing 3.1: Extended Train/Val Split based on crawled Web Page Data (Pseudo-Code)

```
def create_data set_splits(LQ_DOM: list ,\
                           dst_len_hq_train: int, \
                           dst_len_hq_val: int):
    HQ_TRAIN = []
    HQ_VAL = []

    for i in range(dst_len_hq_val):
        sample = select_random_sample(LQ_DOM)
        HQ_VAL.append(sample)
        leaks = find_all_pages_from_same_subdomain(LQ_DOM, sample)

        for leak_sample in leaks:
            LQ_DOM.remove(candidate)

    for i in range(dst_len_hq_train):
        HQ_TRAIN.append(select_random_sample(LQ_DOM))

    return LQ_DOM, HQ_TRAIN, HQ_VAL
```

This algorithm assumes that the source URL of each crawled page is available. For each sample selected for *HQ-VAL*, all images of pages from the same subdomain are filtered out of *LQDOM*. After that, *HQ-TRAIN* can be sampled normally as it does not have to account for the leakage. Note that the algorithm depicted only accounts for single *HQ* data set.

Even though this can be assumed to prevent large proportions of the described effect, a few drawbacks and implications have to be taken into consideration:

- The filtering has to be assured for both evaluation on the high quality data set and the final test set. In order to account the described leakage effect, a certain data set used for pre-training always limits potential evaluation data for the high quality data sets.

- Depending on the crawling procedure and filtering according to the algorithm, this reduces the amount of data in the low quality data set. To limit this data drainage, only the subdomain is taken into consideration in the implementation of this work.

- The same may page be reachable under different domains and different pages may use very similar designs, patterns or even templates/themes. Naive domain-based filtering does not account for this, therefore the effect is limited in this case. However, it is assumed that diminishing return in terms of cost-benefit is reached quickly when taking more advanced methods into consideration. To a certain degree, this even is intended behaviour. Learning to perform better on a wider range of typical modern UI design patterns is part of what can make a pre-training effective.

- If the low quality data set should be improved later on by obtaining new data, the new low quality data set *must* be filtered accordingly as well for the results to be comparable. For example, this can be neccessary in order to account for latest web technologies, to include improved DOM analysis or to increase the number of included samples.

Determining the Test Set

So far, all data has originated from the initially crawled data. Since a final test set is used for a better approximation of real-world performance, it is possible to move away from this principle, using a greater variety of GUI screenshots in this case for more realistic results. This is possible due to the convergence of web pages and traditional software appearance as described in Section 3.1. Since random webpage screenshots as in *LQDOM* and the *HQ* data sets may not always model the desired final use cases of such an detector, it is also an opportunity to introduce selected test samples that are not limited to web page screenshots. However, this has some implications:

- In contrast to many other tasks in machine learning, the test data does not always originate from the same source. The domain the models are trained for is only a subset of the domain samples in the test are. Therefore, some characteristics may differ more than previously.
- While the scores might be lower due to this, it is assumed that this models generic GUI element detection better than only using randomly selected webpage screenshots do. Since this handicap is the same for all trained models, comparison can still be considered fair. However, these differences have to be accounted for during evaluation of the test results.
- Samples are selected by hand to model potential use-cases. This may introduce human bias.
- If a sample is a web page screenshot, the leakage described in Section 3.3.2 has to be accounted for.

The creation of the test set is marked as the sixth step in the pipeline of Figure 3.1. The testing process is the last step of the pipeline.

3.3.3 Classes and Sub-classes

Element classes are labeled hierarchically as proposed in Section 3.3.1. There are *main-classes* and *sub-classes*. The former are rough classifications for elements, where the latter are more fine-grained.

During labeling, the following main-classes where defined which may have individual sub-classes:

- **Interactable**: This is considered to be a fallback class for elements not covered be the remaining classes during labeling. Note that for training, the *Interactable* class is used differently, which is introduced in Section 4.2.
- **Image**: Included images with arbitrary complex image content. As a special case, some logos may be labeled image and text at the same time (edge cases).
- **Button**: Any element that looks like a typical button. In most cases, Buttons have a rectangular shape.
- **Textfield**: The bounding box is defined by the visual borders that may include for example an *Icon.Search*.

- **Dropdown**: Only *real* dropdowns are classified as such. Many modern webpages or GUIs include elements with dropdown-like appearance. As an example, menu items that are basically just a combination of text element and an *Icon.Arrow.Down* occur without any closed border. These are labeled as such (*atomic labeling*, see section Section 3.3.1) and are not considered *real dropdowns* to avoid confusion.

- **Checkbox-Radio**: Even though Checkboxes and Radiobuttons have different functionality, both are quite similar in appearance and are relatively rare on modern webpages. To account for this, this main-class includes both and is further divided into *Checkbox-Radio.Checkbox* and *Checkbox-Radio.Radiobutton*

- **Icon**: They consist of a stylistic pictogram of low complexity which is often associated with a certain functionality (for example 'floppy drive' for 'save')[6]. These are considered to be small elements in most cases. Even though there may be a few edge cases, this definition allows the distinction between *Images* and *Icons*. Further, this allows to define sub-classes which account for different types of elements, making element classifications hierarchic. As described in Section 3.3.1, *Icons* inside *Buttons* are also considered independent elements to be detected. A full list of all *Icon* sub-types is available in Appendix C.3.

  - *Icon.IconGroup*: Although most *Icon* subclasses are self-explaining, there is one subclass where elements are *not* labeled atomic. This category insists of symbols that occur as a group, which are labeled as blocks instead. This is mainly due to the reason that once for example rating stars appear on a page, blocks with the same or very similar appearance often are included multiple times[7]. Atomic labeling of those objects would make the labeling process unneccesserily tedious.

- **Text**: Despite modern web pages rely heavily on giving information through images, text is still one of the most crucial parts of many GUIs. Therefore, the labeling of text is an important part in the data set design as well. Labeling text as blocks as defined in Section 3.3.1 accounts for whole text sections as well as menu entries and hyperlinks. Fixed patterns like menu structures can be clustered together again after detection. This also removes the need of defining a variety of classes for different menu structures, making use of the atomic element principle.

The prediction of a main-class can be considered an easier task than the distinction between similar sub-classes. For example, a button and a textfield have great differences in terms of visual characteristics. However, different icon types which are represented by respective sub-classes can be quite similar.

3.4 LABELING PROCESS

Besides the data itself, a labeling process involving human labelers can lead to particular challenges. External labelers cannot be expected to have wide knowledge about machine learning or possible edge cases. As a more advanced example, besides distinguishing humans from bots,

[6]According to cambridge dictionary, an icon is defined as 'a small picture or symbol on a computer screen that you point to and click on (= press) with a mouse to give the computer an instruction'[u22]

[7]As an example, consider product pages where every product has rating stars next to it

Google reCAPTCHA [u23] is designed to obtain and validate training data. Due to its widespread usage, this has to be as intuitive and accessible as possible while still fulfilling the objective of acquiring labeled data. Another strategy involves using *Gamification* in order to motivate labelers by designing the labeling process as a game [25].

Even though this work utilizes a conventional labeling process, instructions being as straightforward as possible becomes especially important the more complex the data or use case is. The data sets used in this work have been labeled by three external human labelers. The labeling has been conducted using the principles described in Section 3.3.1 and the classes in Section 3.3.3 as guidelines. This section provides a brief overview of the process, which is the third and fourth step in the pipeline of Figure 3.1.

3.4.1 Reviews and Feedback

For quality assurance, there are techniques that distribute one sample to multiple labelers, using cross-validation between the results of the participants to measure a level of consensus.

However, this requires to label the same samples multiple times and requires a certain level of confidence of all participants to start with. Early tests indicated that a conventional review process is more appropriate for the complexity of the real-world data as well as the guidelines to assure the required quality.

In order to fine-tune the process, the labeling started with a warm-up phase including an extensive review and feedback loop between the author and participants. This process was further used to determine details such as the non-atomic labeling of the *Icon.IconGroup* as well as fine-tuning the pre-labeling described in Section 3.4.3 to streamline the process.

Due to the initial warm-up phase, feedback and reviews became less required as the labeling proceeded. Using a chat platform, all participants were able to communicate with each other, asking questions and exchanging edge-cases. Examples of common edge cases have been collected as reference. For that, it became apparent that it is suitable not only to provide edge cases and how they should be handled, but also the reasons for that decision process. For example, the *candidates-as-seen* labeling principle was explained based on real-world samples such as the scamming technique depicted in Figure 3.3. During that, labelers showed interest in how the training and data pipeline worked. Explaining basic machine learning principles and keeping them up-to-date about the training results based on their annotated data strongly helped keeping them motivated. This is crucial, as labeling can be quite monotonous and a lack of motivation causes frustration, which may greatly decrease label quality.

3.4.2 Handling real-world data

The manually annotated data set consists of randomly sampled screenshots previously crawled as discussed in Section 3.3.2. Since this data is sampled from in-the-wild online sources, samples are not always valid. They may be corrupted due to connection or server errors or web pages may still not be loaded correctly when taking the screenshot. Hard edge cases such as malformed

designs or overlays may occur as well as pages with hundreds of redundant elements that would be specifically tedious for a human to label.

For this reason, labelers have an option to skip samples and/or set specific flags during the process. Even though flagged or skipped samples were filtered out for the purpose of this work, this provides additional information which may be utilized in future work[8].

The following flags where used during the labeling process:

- **Overlays**: Pages may load overlays such as ads or cookie notifications. Due to the General Data Protection Regulation (GDPR) introduced May 2018[u24], the latter have become a common issue for automated web crawling as well. Browser extensions such as the *I don't care about cookies* plugin [u25] or ad blockers can be used in order to avoid certain overlays or popups. However, overlays may still occur quite frequently. While small banners often do not lead to any issues, *whole-page-overlays* as depicted in Figure 3.8 result in elements partly obscured. Since larger screenshots are split during pre-processing as discussed in Section 3.5, a single split may not even contain the overlay itself, but just blurred background. In certain situations, it may be impossible to implicitly detect whether an element is a background element or not based on a single split. Therefore, these images are not included for the high-quality data sets. This flag also allows the collection of dedicated samples usable for overlay recognition in future work.

- **Bad Sample**: Samples that may not be suited for the data set, for example connection errors or otherwise malformed/not correctly rendered pages. Annotations with this flag are skipped.

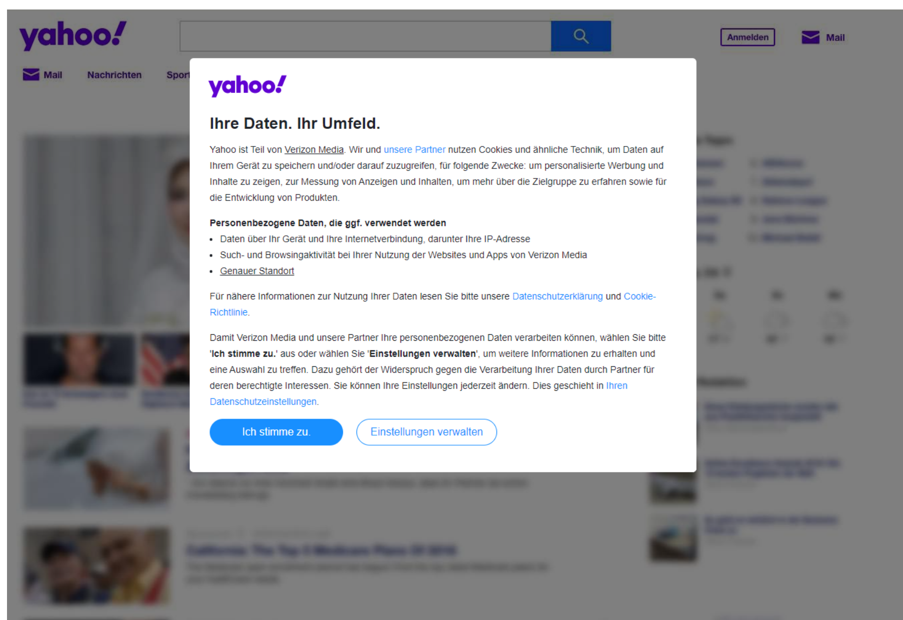


Figure 3.8: This Whole-Page-Overlay blurs all background elements. Image Source: [img11]

[8] Also see *over-annotation* in Section 3.3.1

3.4.3 Pre-labeling

Labeling hundreds of web pages requires the creation of thousands of bounding boxes. In order to speed up the process, *pre-labeling* can be used. Analog to using *weak supervision* for ML models, noisy samples can be utilized as baseline for human labelers instead of creating labels from scratch. Labels given by *DOM analysis* only include few classes which may be incorrect and the locations do often not meet the criteria defined in Section 3.3.1, for example by not being atomic or minimal.

Early tests included additional text segmentations provided by the open source tool tesseract[u26]. However, the segmentations of tesseract on web page screenshots could not meet the quality requirements and included many false positives as well as false negatives. Based on strongly negative labeler feedback, the inclusion of these predictions was counter-productive for the human labeling process. Therefore, tesseract segmentations have not been included.

Later, the early pre-labels were replaced by predictions of models trained on currently available data. These predictions were taken from a model that was not included in the evaluation. This may further be used to implement an automated loop where pre-labels provide more significant help the more the labeling process advances [9].

3.5 PRE- AND POST-PROCESSING

The application of object detection networks for high-resolution screenshots implies problems which are discussed in this Section. As a workaround, additional pre- and post-processing steps are proposed.

Most web pages include more content than a single screen view can reasonable display on a normal zoom setting. Since scrolling downwards is common on web pages, samples consist of high resolution entire-page screenshots and their corresponding labels. Since the amount of content displayed can be different for each page, the height of the screenshots varies significantly.

Even screenshots with a 1080p resolution may be substantially scaled down since state-of-the-art object detection networks utilize a relatively low input resolution. For example, the original *YOLOv3-SPP* configuration from *Darknet* has an input resolution of 608x608px[u27], which has been increased to 704x704px in the *default configuration* described in Section 4.1.3. Increasing the input resolution has significant impact on training performance and memory consumption. Furthermore, since the height of the sample varies, scaling them to a fixed size would cause small elements to be undetectable when scaling down a higher resolution screenshot (see Figure 3.9).

[9] According to labeler feedback, the model predictions provided substantial help.

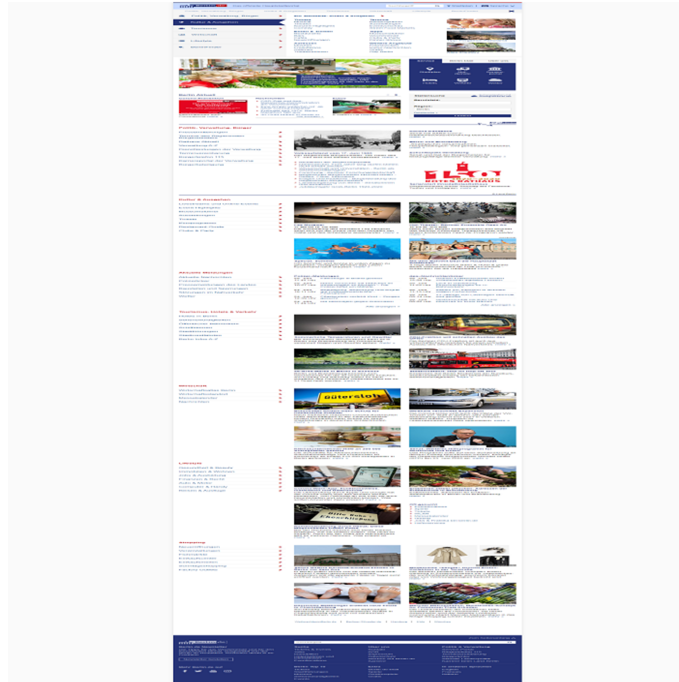


Figure 3.9: Screenshot of `https://www.berlin.de/`. The original has a resolution of 1903x7322px. When downsizing the full-scale image to 704x704px, smaller elements such as icons may not be distinguishable anymore.

For the mentioned reasons, images are divided into *splits* with a maximum height of 1080px. Each split after the first one includes the last 50px of the previous split at the top to avoid horizontal cutting of elements. Note that this may result in some duplicate elements during training and testing due to consecutive splits. This effect is neglected in the metric calculation.

For the labels annotated by DOM analysis, additional pre-processing includes for duplicate removal based on an IoU Threshold[10].

As the Tables in Section 3.6 show, the average whole page screenshot is divided into two or three splits. After that, each individual split is resized to fit the input resolution of the network configuration. This is applied for all images as a *pre-processing* step.

For inference, the prediction results of each split can be merged together again in order to match the original image. However, this process may introduce minor errors for large objects included in multiple splits. Metrics are calculated using the split results in order to measure only the network performance. This may cause some objects to occur twice in different splits due to the overlap defined above. The influence of this behaviour is neglected during evaluation.

3.6 DATA SET VARIANTS

This section gives an overview about the size and names of the resulting data sets. Note that the object count has been calculated from splits. This implies that a few objects may have counted

[10] Additional pre-processing that further reduces errors made by DOM analysis may help to improve results on this data in the future.

twice due to the splitting as explained in Section 3.5[11].

A more detailed overview including class counts for all classes is available in Appendix C.4.

3.6.1 LQDOM - Low Quality DOM Data Set

The Low Quality DOM Data Set *LQDOM* set refers to the noisy data aquired by web crawling and HTML DOM analysis. To avoid data leakage if a web page that is included in a high-quality validation set has been already included during *LQDOM* training, the latter has been filtered to exclude samples from the same subdomain as in the human-labeled validation and test sets. Table 3.1 depicts the size of *LQDOM*.

	Pages	Splits	Objects
LQDOM	93,896	229,755	8,905,452

Table 3.1: Sample and object count of *LQDOM*.

3.6.2 HQv1 - High Quality Data Set v1

HQv1 is the first version of the human-labeled data set. It was used for the majority of evaluations. The size of *HQv1* is shown in Table 3.2.

	Pages	Splits	Objects
HQv1-TRAIN	194	504	22,233
HQv1-VAL	50	127	5,579
$\sum$	244	631	27,812

Table 3.2: Sample and object count of *HQv1*.

3.6.3 HQv2 - High Quality Data Set v2

Since even more data has been labeled over time, a second version of the high quality data set is available, which is named *HQv2*. This can be used to measure the impact of the high-quality data amount. Table 3.3 depicts the size of *HQv2*.

	Pages	Splits	Objects
HQv2-TRAIN	683	1,690	72,552
HQv2-VAL	161	411	18,644
$\sum$	844	2,101	91,196

Table 3.3: Sample and object count of *HQv2*.

[11] These are also seen twice during training and evaluation. The impact of this is mostly neglected during further evaluation in this work as it is consistent.

3.6.4 FINAL-TEST

As the validation sets from the previous data sets are used for model selection, the final evaluation is performed using another seperate data set to prevent information leakage (cf. Section 3.3.2). This data set is named *FINAL-TEST*. Its size is depicted in Table 3.4.

	Pages	Splits	Objects
FINAL-TEST	65	215	9,998

Table 3.4: Sample and object count of *FINAL-TEST*.

4 EXPERIMENTAL SETUP

This chapter covers general conditions for both experimentation environment and evaluation. Section 4.1 includes details about framework and hardware as well as baseline network configurations. Section 4.2 defines labeling variants with a varying level of detail. Implementation details of weak supervision and transfer learning are discussed in Section 4.3. Section 4.4 defines custom metrics that are used for evaluation and discussion within the following chapters. Finally, the model selection process is described in Section 4.5.

4.1 ENVIRONMENT

This section introduces details about the utilized *Darknet* [u28] framework and the system configuration which was used for training. Additionally, it defines a baseline training configuration, which is used in order to allow meaningful comparisons between individual experiments.

4.1.1 Darknet and YOLOv3

Darknet is an open source framework for training neural networks. It has been introduced by Joseph Redmon who also published the original YOLO architectures until *YOLOv3*[7, 8, 9]. Therefore, it includes the original reference implementations for the *YOLO* architecture family up to version 3.

The development by the original author [u29] has been discontinued. A fork with extended functionality is maintained by Alexey Bochkovskiy [u28]. In this work, the term *Darknet* always references this fork.

Darknet implements several architectures and techniques. There are standard configurations which are intended to be used as presets to achieve good results on a variety of object detection data sets. Each configuration file includes the net architecture as well as the specification of

options such as hyperparameters or data augmentation [u30, u31]. Additionally, *Darknet* implements further features like state-of-the-art data augmentation techniques or experimental memory management that leverages CPU memory while training on GPU [u32]. These implementations do not always lead to better results, but allow experimentation. Depending on the data set, further improvements compared to the default presets can be made. Although there is no complete documentation available, most of the features are explained throughout github issues.

4.1.2 Hardware

The following hardware configuration has been used for the training experiments:

- 8x GTX 1080Ti (11 GiB VRAM each)
- Intel Xeon E5-2630 v4 (10 cores plus hyperthreading, 2.2 GHz)
- 226 GB RAM

Darknet can parallelize a single training on up to four GPUs [u28]. Note that the actual GPU utilization varies as experiments were performed inside individual docker containers, using a single GPU only on most cases.

4.1.3 Default Training Configuration

All experiments are variations based on a default preset, which is described in this section. It is designed to model a training using the *Darknet* framework without additional task-specific research. Realistically, only a few possible variations can be tested due to high training times and permutation count. Therefore, the *default preset* used for the experiments models a naive baseline following some of the best practises and presets recommended in [u28]. The latter have been tweaked by the author of the framework with reference to state-of-the-art techniques to perform well on a variety of object detection data sets.

Improvements made by the following experiments can be measured realistically against these baseline results. Note that in order to achieve a fair comparison to implementations in other frameworks, similar prerequisites, for example in terms of data augmentation techniques that are implemented in darknet, should be assured.

The default preset is based on the *YOLOv3-SPP* configuration for the *Darknet* framework, which is available at [u27]. The input resolution was increased to 704x704px, as a higher resolution increases the prediction performance and training a model at this resolution was still possible using the VRAM of the GPUs utilized. Additionally, data augmentation using the *mixup* algorithm [26] has been included.

4.2 LABEL VARIANTS

Since the noise introduced by DOM analysis is different for localization and classification as discussed in Section 3.2.2, it is reasonable to investigate different labeling variations in all stages of training. This allows to measure the effect of those variants seperately. In this section, the naming of these variations is introduced.

4.2.1 Binary

As stated in Section 3.2.2, class labels and locations obtained by DOM analysis are often limited. As both introduce different noise, generalizing to a single class can be considered being an easier task than additionally trying to learn noisy class labels. This class is called *Interactable*. In this single-class detection task, only the localization of potential GUI elements is learned.

4.2.2 Multilabel

Training on noisy labels could negatively impact the overall performance due to *confusion*: If 3 elements are labeled textfields according to DOM analysis, but one should be a button in reality, the model will learn to consider a button each time it encounters a textfield. Depending on how noisy the annotation is, this may result in lower confidences and worsen overall recognition.

Additionally, as stated in Section 3.2.2, many elements are localized as such, but the corresponding class is unclear. While a generic *unknown* class could serve as naive approach, this could further impact the already existing model confusion.

A simple approach which uses the advantages of *binary* labeling, but takes different classes into account, can be created through the introduction of hierarchic classification labels[1]: Each element is considered to be an mandatory *Interactable*, but can additionally be assigned another optional class like *Button* or *Textfield*. Additionally, this approach accounts for further sub-classifications. For example, it allows the introduction of a main *Icon* class, which further splits into sublasses like *Icon.Arrow* or *Icon.Close*. Furthermore, this allows for an investigation of the effect the class count has on the overall results.

Three variants of *MULTILABEL* labeling variants are available:

- **DOM-Classes**: This is the labeling used for training on *LQDOM*. It consists of all classes that were obtained during DOM analysis (see Section 3.2.2 and Appendix C.1). All other elements such as *Text* and *Icons* as well as their respective subclasses are just labeled as *Interactable*.
- **Main-Classes**: Only utilize non-subclasses (see Appendix C.2). Subclasses are just labeled as their respective parent class (e.g. *Icon.Arrow* becomes *Icon*). All classes except *Text* and

[1] A network architecture has to support multi-labels for this approach to work. YOLOv3 uses independent logistic classifiers for class predictions instead of softmax (which is often used as default activation function in the last layer of neural networks for classification) in order to achieve this. See [9].

Icon have been included in the pre-training. For most classes except *Checkbox-Radio* and *Dropdown*, plenty of training and validation/test data is available in the HQ data sets (cf. Appendix C.4).

- **29-Classes**: All classes that the *HQ* data set distinguishes between are used for training. Many classes may have only few samples, therefore, the classification of those can be expected to be noisy, which may negatively impact the overall performance. By including this experiments, this impact can be estimated. This measurement can also be an indicator if including weak/noisy estimations for additional classes during a pre-training might harm the prediction quality of already included classes.

4.3 MULTI-STAGE TRAINING - COMBINING WEAK SUPERVISION AND TRANSFER LEARNING

Deep neural networks used for object detection achieve State-of-the-Art results on large object detection data sets such as the COCO data set. The concept of the training process is to combine the advantages of both *Weak Supervision* and *Transfer Learning* in a pipeline (cf. Figure 3.1) in order to reduce the amount of required high-quality training data.

The key idea is to pre-train an object detection model on a large amount of data, where labels can be obtained easily, but may be noisy or incomplete. After the pre-training is done, the model can be fine-tuned via the use of fewer, but high quality data with manually annotated labels. Instead of mixing both LQ and HQ data, these steps have to be treated seperately, making use of the advantages of transfer learning. This leads to a training pipeline with multiple successive steps.

In the case of a GUI element detection task, huge amounts of noisy data can be obtained easily with the use of web crawling and DOM analysis (cf. Section 3.2.2).

Two types of *transfer learning* are implemented in the training setups.

The first variant is the usage of weights pre-trained on *LQDOM* as described. The weakly supervised training can be directly evaluated using *HQv1*. This is limited due to the DOM analysis labeling not following the same principles as described in Section 3.3.1. However, this handicap applies to all compared weights trained on *LQDOM*. Therefore, a higher score on *HQv1* directly implies a smaller distance between the predictions after the pre-training and the considered ideal predictions. The results of the selected *LQDOM* models are included in Appendix E.

The second variant is the usage of weights pre-trained on the *COCO* data set. For the network architectures used, these are publicly available at [u28].

Both variants can be combined as the training on *LQDOM* can be performed using weights already pre-trained on *COCO*.

For the *YOLOv3-SPP configuration*, the first 81 layers can be used for *transfer learning*. These layers perform general feature extraction and include the basic filters pre-trained (cf. Section 2.2.2). The extracted weights are called *partial weights*. Later layers are used to perform classification, therefore the number of weights directly depend on the number of classes. If the number of

loaded pre-trained weights does not match the current model size, *Darknet* does not load the weights and reinitializes the whole network [u33].

As a workaround, a pre-training using *Fake Classes* can be exploited to prevent the class number from being changed. By pre-training a model with weights initialized for more classes that are actually annotated, these classes have zero samples during the pre-training. However, since this can be exploited for the classes of both pre-training and training on *HQv1/v2* to match exactly, weights from all layers can be preserved. This may impact model performance if too many fake classes are included. Another drawback of this approach is that for each class/setup combination, a dedicated pre-training has to be performed.

The multi-stage training process is the seventh and eighth step of the pipeline depicted in Figure 3.1.

4.4 CUSTOM EVALUATION METRICS

With the usage of mAP (see Section 2.3.4) as evaluation metric and the calculation procedure *Darknet* utilizes, several problems occur. The first problem is the mAP metric itself: Calculating the mean of the AP of all classes implies that all classes are treated as equally important for evaluation. Due to the hierarchic classification of GUI elements, using the mAP would for example over-weight specific *Icon* subclasses: mAP would treat the AP *Icon.Arrow.Down* equally with *Text*, even though the first one is a more specific classification with less samples both in training as in real world.

Furthermore, the AP is calculated per class as defined in Section 2.3.4. However, in the hierarchic *MULTILABEL* labeling, one object has assigned multiple classes. When calculating the AP, the sample is included in the AP calculation of each involved class. While this behaviour is correct for each individual class AP, it prevents the correct calculation of other metrics such as the overall recall based on class-specific recalls or the comparison of different labeling approaches. This is due to the reason that one occuring object is counted multiple times, depending on how many classes are assigned (cf. class count table in Appendix C.4).

4.4.1 Selective AP

The described behaviour becomes a problem when comparing the different labeling approaches defined in Section 4.2. For this reason, custom scores based on the seperate class results are calculated instead of the default mAP, seperating the AP scores for the main-classes and sub-classes. With this approach, results of different labeling strategies can be directly compared to each other.

Due to the strong imbalance of class count in the training as well as in the validation and test sets, several variants of calculating a custom score can be derived. Each variant includes only the scores of a subset of all classes, allowing a direct comparison of trainings with a different amount of classes as well as measuring the impact of classes with less samples to the score (Formula 4.1).

$$APC(S) = \sum_{c \in S} AP_{50}(c) \tag{4.1}$$

S — : Set of classes to take into consideration

Following this definition, APM can be calculated over the AP scores of the main-classes as defined in Section 3.3.3. With APM, the AP score of *Icon* subclasses is not taken into consideration. However, despite *Interactable* being a main-class, it has been exluded from the APM calculation. This is due to the reason that every element is considered an *Interactable* when using the *multilabel* labeling. Including this class would add up to 0.5 to the score even if no other classifications were predicted. To weight the score towards class recognition instead, the *Interactable* class is not included in the score calculation[2].

Since most of the main classes are included in the pretraining and these classes are considered the most important to distuinguish between, this is the main metric for experiments involving *multilabel* labelings (Formula 4.2).

$$APM = APC(c \in C_{MAIN}) = \sum_{c \in C_{MAIN}} AP_{50}(c) \tag{4.2}$$

C_{MAIN} — : Main Classes as defined in Appendix C.2, *excluding* $Interactable$

For experiments utilizing *binary* labelings, only $AP(Interactable)$ is available. This score is abbreviated as APB and is available for both *binary* and *multilabel* experiments.

Sub-classes of *multilabel-29classes* experiments can be evluated within a seperate metric using the same principle. The API metric only takes the AP scores of $Icon$ sub-classes into account.

$$API = APC(c \in C_{Icon.*}) = \sum_{c \in C_{Icon.*}} AP_{50}(c) \tag{4.3}$$

$C_{Icon.*}$ — : Subclasses of $Icon$ (cf. C.4)

4.4.2 Weighted (Selective) AP

The detection of single samples of classes with only a few samples has greater impact when class AP scores are averaged. Therefore, mAP is biased to strongly overestimate the importance of classes with less occurrences. In some cases, this can be intended if the detection of all classes is considered equally important, despite some having fewer samples. The downside is that classes with a few samples are proned to noisy results due to the even smaller amount of validation/test set samples for that class. For example, in the validation set of *HQv1*, only 12 elements of the class *checkbox_radio* exist. Missing a single *Checkbox_Radio* element during test time would have the same impact as missing over hundred *Text* elements. However, it can be expected that the former does not have as much significance for the performance on real-world data outside of the validation set than the latter has. Therefore, a weighted AP can be taken into consideration to lower the impact of classes with few samples without entirely neglecting them (Formula 4.1).

[2]However, the *recall* still accounts for found elements with unclear classification.

$$APC^{w} = \sum_{c \in C} AP_{50}(c) * w(c) \tag{4.4}$$

C — set of classes used for selective AP (see Section 4.4.1)
w — weighting function

The most simple approach is to directly weight elements based on their total occurrence number in the validation/test set, which biases the score towards classes with many occurrences (Formula 4.5).

$$APC^{wo}(C) = \sum_{c \in C} APC(c) * count(c) \tag{4.5}$$

C — set of classes
$count(c)$ — total number of samples with class c in the val/test set

The definition of Formula 4.5 can be used in order to implement APM^{wo}(Formula 4.6) and API^{wo} (Formula 4.7) that are weighted by the class count.

$$APM^{wo} = APC^{wo}(c \in C_{MAIN}) \tag{4.6}$$

$$API^{wo} = APB^{wo}(c \in C_{Icon.*}) \tag{4.7}$$

C_{MAIN} — : Main Classes (cf. Appendix C.2), *excluding Interactable*
$C_{Icon.*}$ — : Sub Classes of *Icon* (cf. Appendix C.4)

4.4.3 Total Recall

When recognizing GUI elements, the overall recall is an important metric as it depicts how many GUI elements where actually found by the detector. Therefore, it should be observed separately.

For the *MULTILABEL* labeling strategy as defined in Section 4.2.2, each element may have multiple classes assigned. However, The calculation implemented in *Darknet* utilizes TP and FP summed up over all class-specific recalls to calculate the overall recall. The *mean of class-dependent recalls* is being calculated instead of the *class-independent recall* defined in Section 2.3. This implies that the more classes an individual element has assigned, the more the detection of this element impacts the calculated recall. For example, consider a rating star element with the labels *(Interactable, Icon, IconGroup, Stars)*. That element is therefore counted as four objects according to this calculation. This not only distorts the results when including objects with different hierarchy levels, but also prevents this results from being correctly compared with the *BINARY* labeling strategy, which can be used to measure the effect of noisy pretraining (see Section 3.2.2 and chapter 5).

A naive approach is to calculate the overall recall directly from the TP count for the *Interactable* class, since in the *MULTILABEL* (*ML*) labeling strategy, every element is considered an *Interactable*). However, it is possible for some elements having only a single class assigned and there-

fore not being classified as *Interactable*. This behaviour was observed for at least a few elements even for the best performing models[3].

For a correct calculation of the overall recall, the calculation has been reimplemented by counting all TP/FP detections instead of using the per-class *Darknet* implementation. In this implementation, TP/FP counts are based only on the IoU threshold, not on classification. Therefore, the *class-independent* Recall has been implemented seperately. For this, the recall is calculated using the confidence threshold $thresh^{conf} = 0.25$ and the IoU threshold $thresh^{IoU} = 0.5$.

RC denotes the recall value according to this calculation.

4.4.4 Comparison of Metrics between Multiple Experiments

For experiments utilizing *binary* data sets, the calculated score only consists of the *Interactable* AP (APB). For experiments utilizing *multilabel* labelings, the *(weighted) main-class AP* (APM or APM^{wo}, see Section 4.4.1 and Section 4.4.2) is calculated.

This has the following implications for comparing experiments with different labeling strategies:

- APB and APM cannot be compared directly between *binary* and *multilabel* experiments.
- Comparison between APB from *multilabel* experiments and APB from *binary* experiments is allowed.
- Comparison between *recall* scores of *binary* and *multilabel* experiments is allowed.
- APM for *multilabel-29classes* also only accounts for main classes, which is intended. This allows a direct comparison between all *multilabel* experiments and measuring the degradation of prediction performance when introducing more classes. For evaluating the sub-class performance, a dedicated sub-class score can be calculated similarly to the APM. Since no pre-training data is available for those classes and some sub-classes may only have a few samples, this is not the focus of the evaluation. This is further discussed in Section 6.1.2.

4.5 MODEL SELECTION PROCESS

As introduced in Section 3.3.2 and 3.6, the *FINAL-TEST* set is used to measure the final model performance on unseen data. In order to fulfill this purpose, those experiments have to be done at the very end of evaluation, only testing final models. This implies that no feedback loop such as recurring hyperparameter optimization is allowed when testing models on FINAL-TEST. This includes selecting the best performing model, which has to be performed using the HQ validation sets instead.

During training, model checkpoints were saved every 1000 iterations, which equals 64,000 trained images. Each model checkpoint is then evaluated by calculating validation metrics.

[3]This may be amplified when higher confidence thresholds are applied for all classes.

For selecting the pre-trained weigths that are used for transfer learning, the non-weighted mAP is calculated over all classes that are included in the DOM analysis (see Appendix C.1). More detailed results of that process are included in Appendix E.

Models trained on the HQ data sets were selected using APM^{wo}. The results of the best performing models according to this metric are shown for different experiments in the following chapter.

5 EXPERIMENTS

The main aspect of this chapter is the introduction and interpretation of individual training experiments, while the next chapter covers an overarching result discussion.

Due to the focus of this work, these experiments investigate different data-oriented setups rather than optimizing hyperparameters such as the *learning rate*. It is not suitable to test all possible permutations due to combinatorial explosion and long training times. Therefore, reasonable setups have to been created based on assumptions and expectations. For each setup, results are displayed in corresponding tables.

Each section first introduces the general idea behind this type of experiment. After that, the results based on *HQv1/FINAL-TEST* are shown for each individual experiment. A more detailed result overview including *HQv2* results is included in Appendix E. A graphical overview of all experiment permutations is depicted in Appendix D.

5.1 NOTATION AND INTERPRETATION OF RESULTS

For each experimental setting, a result table is provided. Each table includes information about several experiment variations, which consist of training on HQv1/HQv2 (see Section 3.6) and the three labeling variants *binary* (BIN), *multilabel-mainclasses* (MAIN) and *multilabel-29classes* (29C) as introduced in Section 4.2.

For each experiment and variation, different result metrics are provided as summarized in Table 5.1. Observing single metrics is limited. The APM is especially noisy due to class imbalance. For that reason, the APM^{wo} was defined, however this metric is biased towards classes with many occurrences. The recall RC depicts how many relevant elements were found, but this metric does not account for the number of false positives. There is an error range for single metric values, which is discussed in Section 6.2.1. Therefore, multiple metrics have to be considered per experiment for a differentiated result interpretation. An improvement considering multiple metrics is a significant indicator for better overall performance.

Abbreviation	Meaning
APB	*Binary AP* / AP of class *Interactable*, Main metric for binary experiment variants
APM	*(mean/unweighted) main-class AP*. Equal to mAP considering only main-classes. Can be noisy, high error range.
APM^{wo}	*occurrence-weighted main-class AP*, Main metric for evaluating *multilabel* experiments
RC	Class-independent Recall, Calculated as defined in Section 4.4.3
Cls	Labeling variant (cf. Section 4.2)
$Iter$	Iteration at which the best model was selected, 1 Iteration equals batch size (64 images) (cf. Section 4.5)

Table 5.1: Abbreviations used in result tables

Note that in the individual tables below, only *HQv1-VAL* and *FINAL-TEST* results are displayed. Full overview result tables including *HQv2-VAL* results are shown in Appendix E.

As an example, the term APM^{wo} *on FINAL-TEST* refers to the *occurrence-weighted main-class AP* calculated from the predictions of the selected model on the *FINAL-TEST* set.

The APM can be noisy due to the heavy class imbalance, which is why APM^{wo} should be used as main metric for comparing *multilabel* experiments (see Section 4.4.2). Note that for the validation results, information leakage has occurred due to model selection (cf. Section 3.3.2). The *FINAL-TEST* results depict the performance on completely separated data.

For easier comparison, the difference between each experiment and the corresponding baseline result is written *cursive* below the result metrics of each experiment. Positive numbers indicate that the current experiment performs better than the baseline by the given margin.

5.2 BASELINE - NAIVE TRAINING ON HAND-LABELED DATA

Naive training is utilized as baseline. Training is performed on high quality data sets only and utilizes the default settings defined in Section 4.1.3 without further adjustments. Therefore, weakly supervised pre-training was not used.

The baseline can be used for measuring the effect of the examined approaches as well as showing tendencies of different test setups. Since computational resources are limited and large trainings can take several days of training, the latter can be used to pre-filter setups for the multi-stage experiments. This avoids combinatorial explosion for both experimentation and evaluation.

Using *HQv1*, the results of a naive training approach are strongly limited by the few training samples compared to a bigger training data set. Training settings such as *Hyperparameters* that affect the naive training in a positive, negative or neutral way may translate into similar effects

on the multi-stage approaches as well. However, these effects might be smaller due to the larger amount of training data used in *LQDOM*.

The baseline results are depicted in Table 5.2.

TRAIN	Cls	HQv1-VAL				FINAL-TEST			
		APB	APM	APM^{wo}	RC	APB	APM	APM^{wo}	RC
HQv1	BIN	79.90	-	-	83.37	81.70	-	-	81.71
	MAIN	78.82	59.99	75.40	81.21	80.84	51.41	75.57	78.88
	29C	79.88	64.16	76.46	83.17	78.71	60.42	74.54	77.89
HQv2	BIN	83.68	-	-	85.32	87.10	-	-	84.07
	MAIN	83.90	66.10	80.82	85.86	86.79	68.47	82.79	82.89
	29C	82.27	67.88	79.30	85.30	86.31	61.39	81.38	84.45

Table 5.2: Naive baseline training without pre-trained weights.

The results indicate that the higher amount of training data in *HQv2* increase the prediction performance by ~4-6%, depending on the metric considered. Including more classes in the baseline tends lower the prediction performance on *FINAL-TEST*, especially the recall. A higher amount of training data effectively mitigates this effect.

5.3 COCO PRE-TRAINING

The training on high-quality data sets can be performed using weights pre-trained on COCO as they are provided in the *Darknet* repository. Table 5.3 shows the result metrics in relation to the baseline of the previous section.

TRAIN	Cls	HQv1-VAL				FINAL-TEST			
		APB	APM	APM^{wo}	RC	APB	APM	APM^{wo}	RC
HQv1	BIN	77.84 *-2.1*	-	-	81.21 *-2.2*	81.22 *-0.5*	-	-	80.67 *-1.0*
	MAIN	79.05 *+0.2*	60.77 *+0.8*	76.14 *+0.7*	82.00 *+0.8*	82.11 *+1.3*	58.98 *+7.6*	77.45 *+1.9*	81.05 *+2.2*
	29C	79.13 *-0.8*	64.56 *+0.4*	75.78 *-0.7*	81.64 *-1.5*	79.52 *+0.8*	61.59 *+1.2*	75.32 *+0.8*	79.23 *+1.3*
HQv2	BIN	83.54 *-0.1*	-	-	86.52 *+1.2*	86.63 *-0.5*	-	-	85.12 *+1.1*
	MAIN	82.94 *-1.0*	66.02 *-0.1*	79.93 *-0.9*	86.54 *+0.7*	87.94 *+1.1*	71.67 *+3.2*	83.64 *+0.8*	86.48 *+3.6*
	29C	82.06 *-0.2*	72.59 *+4.7*	79.62 *+0.3*	86.04 *+0.7*	88.05 *+1.7*	76.38 *+15.0*	84.45 *+3.1*	86.62 *+2.2*

Table 5.3: Training using *partial COCO-pre-trained* weights.

Considering the prediction performance on *FINAL-TEST*, utilizing weights pre-trained on a generic object detection data set leads to a quite consistent increase in prediction performance.

5.4 BINARY PRE-TRAINING

As introduced in Section 2.2.1, the pre-training on *LQDOM* lies within two types of weak supervision. As some elements may not be found by the DOM analysis, it is *incomplete supervision*. Additionally, class labels as well as bounding boxes of may be incorrect, therefore it is *inaccurate supervision*. As there might be performance differences depending which type of noise is present, it is suitable to measure the effects of a *binary pre-training* where class labels are discarded and all elements are just labeled as *Interactable*. By testing the two-stage pipeline with a binary pre-training, the effect of noisy class labels in *LQDOM* can be measured.

Using *multilabel* labelings in the second stage, the lower noise of the binary pre-training is utilized before introducing distinct classes in the second stage. While the pre-training can be expected being more consistent than a multilabel pre-training, distinction between the different classes had to be learned using only the data provided in the second stage.

TRAIN	Cls	HQv1-VAL				FINAL-TEST			
		APB	APM	APM^{wo}	RC	APB	APM	APM^{wo}	RC
HQv1	BIN	85.99 *+6.1*	-	-	86.86 *+3.5*	85.74 *+4.0*	-	-	84.50 *+2.8*
	MAIN	82.20 *+3.4*	64.56 *+4.6*	79.01 *+3.6*	85.17 *+4.0*	84.25 *+3.4*	58.47 *+7.1*	78.48 *+2.9*	83.66 *+4.8*
	29C	82.39 *+2.5*	67.01 *+2.9*	79.05 *+2.6*	84.60 *+1.4*	81.61 *+2.9*	57.52 *-2.9*	76.91 *+2.4*	81.07 *+3.2*
HQv2	BIN	84.80 *+1.1*	-	-	87.26 *+1.9*	87.12 *+0.0*	-	-	85.89 *+1.8*
	MAIN	84.47 *+0.6*	69.40 *+3.3*	81.25 *+0.4*	87.37 *+1.5*	88.68 *+1.9*	65.87 *-2.6*	83.90 *+1.1*	87.67 *+4.8*
	29C	82.81 *+0.5*	69.16 *+1.3*	79.99 *+0.7*	86.05 *+0.8*	87.17 *+0.9*	71.43 *+10.0*	82.81 *+1.4*	85.42 *+1.0*

Table 5.4: Results of trainings using *partial binary pre-training* weights as described in Section 4.3.

Binary pre-training leads to a significant increase in the overall performance compared to the baseline. It outperforms the COCO pre-training if less training data is available.

The results depict the positive effect of pre-training on a huge amount of data compared to the naive approach, which is visible across the metrics. Possible negative effects of noisy pre-training are not visible in the results. The largest increase is visible for APB and RC of the binary experiments.

In transfer learning experiments with a varying class count in the second stage, only a fraction of the pre-trained weights can be utilized as described in Section 4.3. Since both binary pretraining and binary HQ training use a single *Interactable* class, the class count remains the same. Therefore, when using binary pre-training as well as binary labeling in the HQ training, the full pre-trained weights can be utilized. The impact of using all pre-trained weights over using only a fraction can be measured by testing both variants.

TRAIN	Cls	HQv1-VAL				FINAL-TEST			
		APB	*APM*	APM^{wo}	*RC*	*APB*	*APM*	APM^{wo}	*RC*
HQv1	BIN	87.60 *+7.7*	-	-	88.14 *+4.8*	86.67 *+5.0*	-	-	85.79 *+4.1*
HQv2	BIN	85.09 *+1.4*	-	-	87.37 *+2.1*	88.89 *+1.8*	-	-	87.28 *+3.2*

Table 5.5: BIN-AP on HQv1/v2 and final test set (binary) using *binary* pre-trainings. This variation utilizes all layers of the pre-trained model.

As depicted in Table 5.5, the positive effects on *APB* and *RC* on *FINAL-TEST* are further amplified by utilizing the full pre-trained weights considering the *HQv1* training. However, the results are strongly diminished for the larger *HQv2* training.

5.5 MULTILABEL PRE-TRAINING

In this setup, a pre-training including *MULTILABEL-DOMCLASSES* (cf. Section 4.2.2 and Appendix C.1) is used. Since the *multilabel* format labels elements hierarchically and every element is an *Interactable*, it can be expected that confusion caused by noisy class labels only affects the other classes, not the labeling of *Interactable* itself. This incidates that even if the model is highly unsure whether an object of a specific class is detected, it can still be marked as *Interactable*, which provides the same recall advantages as the binary pretraining. Additionally, distincting between the provided classes is already pre-learned.

The *multilabel* labelings can take advantage of both more-stable *Interactables* as in the *binary* pre-training while still utilizing all class information learned, even if these introduce additional noise. As only some classes were annotated by the DOM analysis, the class count is different. Therefore, *Transfer Learning* has to be performed using extracted partial weights, including the weights of the first 81 layers of the pre-trained model. The results are depicted in Table 5.6.

TRAIN	Cls	HQv1-VAL				FINAL-TEST			
		APB	*APM*	APM^{wo}	*RC*	*APB*	*APM*	APM^{wo}	*RC*
HQv1	BIN	86.33 *+6.4*	-	-	86.05 *+2.7*	**86.93** *+5.2*	-	-	85.34 *+3.6*
	MAIN	84.04 *+5.2*	66.08 *+6.1*	80.79 *+5.4*	85.08 *+3.9*	83.81 *+3.0*	62.12 *+10.7*	79.41 *+3.8*	82.63 *+3.8*
	29C	82.13 *+2.2*	66.2 *+2.0*	78.4 *+1.9*	83.80 *+0.6*	83.77 *+5.1*	66.69 *+6.3*	79.12 *+4.6*	82.17 *+4.3*
HQv2	BIN	84.73 *+1.0*	-	-	87.04 *+1.7*	88.40 *+1.3*	-	-	85.45 *+1.4*
	MAIN	84.68 *+0.8*	72.46 *+6.4*	82.23 *+1.4*	87.62 *+1.8*	88.43 *+1.6*	70.97 *+2.5*	84.39 *+1.6*	87.24 *+4.3*
	29C	84.33 *+2.1*	72.98 *+5.1*	81.45 *+2.2*	86.81 *+1.5*	87.52 *+1.2*	77.06 *+15.7*	84.05 *+2.7*	86.40 *+2.0*

Table 5.6: Training using *partial multilabel pre-trained* weights.

For *HQv1*, this pre-training consistently outperforms the binary and COCO pre-training. As for the binary-pretraining, this advantage diminishes if the amount of training data is increased in *HQv2*.

As introduced in Section 4.3, a pre-training using *fake classes* that do not actually exist in *LQDOM* can be exploited to utilize all weights for transfer learning. However, a dedicated pre-training is required that already includes the final class labeling. In contrast to the binary training of the previous section, the *binary* experiments are limited to partial pre-training weights as the class count is different from the pre-training.

Table 5.7 shows the results of *main-class* trainings using this technique.

TRAIN	Cls	HQv1-VAL				FINAL-TEST			
		APB	APM	APM^{wo}	RC	APB	APM	APM^{wo}	RC
HQv1	MAIN	84.03	68.33	80.99	85.48	84.96	66.62	80.43	83.41
		+5.2	*+8.3*	*+5.6*	*+4.3*	*+4.1*	*+15.2*	*+4.9*	*+4.5*
HQv2	MAIN	85.55	70.86	82.87	87.60	89.87	75.73	86.29	87.60
		+1.6	*+4.8*	*+2.1*	*+1.7*	*+3.1*	*+7.3*	*+3.5*	*+4.7*

Table 5.7: Training utilizing all weights of a *fakemainclasses* pre-training.

A dedicated *fakemainclass* pre-training further increases the performance gain. When all pre-training weights are utilized, this pre-training performs better than other pre-training variants even for *HQv2*.

5.6 COCO- AND MULTILABEL PRE-TRAINING

The *multilabel* pre-training used in the last Section can also be combined with a COCO-pre-training. Therefore, this stacked Transfer Learning:

1. Load partial weights pre-trained on COCO as in the experiments of Section 5.3
2. Perform training on *LQDOM* as in the experiments of Section 5.6, but with COCO weights instead of complete model re-initialization
3. Train on high-quality data *HQv1*/*HQv2*

TRAIN	Cls	HQv1-VAL				FINAL-TEST			
		APB	*APM*	APM^{wo}	*RC*	*APB*	*APM*	APM^{wo}	*RC*
HQv1	BIN	84.86 *+5.0*	-	-	85.77 *+2.4*	85.61 *+3.9*	-	-	83.94 *+2.2*
	MAIN	80.85 *+2.0*	66.18 *+6.2*	77.75 *+2.3*	82.51 *+1.3*	81.13 *+0.3*	61.45 *+10.0*	77.30 *+1.7*	79.98 *+1.1*
	29C	80.56 *+0.7*	63.74 *-0.4*	77.57 *+1.1*	83.14 *+0.0*	82.66 *+4.0*	59.82 *-0.6*	77.79 *+3.2*	80.3 *+2.4*
HQv2	BIN	84.61 *+0.9*	-	-	87.31 *+2.0*	89.25 *+2.2*	-	-	87.20 *+3.1*
	MAIN	83.08 *-0.8*	71.20 *+5.1*	80.62 *-0.2*	85.5 *-0.4*	87.96 *+1.2*	75.21 *+6.7*	84.33 *+1.5*	85.70 *+2.8*
	29C	83.87 *+1.6*	69.46 *+1.6*	80.99 *+1.7*	86.76 *+1.5*	87.42 *+1.1*	71.11 *+9.7*	83.40 *+2.0*	85.28 *+0.8*

Table 5.8: Training utilizing partial COCO-weights + partial *multilabel pre-training*

TRAIN	Cls	HQv1-VAL				FINAL-TEST			
		APB	*APM*	APM^{wo}	*RC*	*APB*	*APM*	APM^{wo}	*RC*
HQv1	MAIN	82.45 *+3.6*	67.00 *+7.0*	79.64 *+4.2*	84.52 *+3.3*	85.03 *+4.2*	68.13 *+16.7*	80.33 *+4.8*	84.11 *+5.2*
HQv2	MAIN	84.29 *+0.4*	69.69 *+3.6*	81.16 *+0.3*	86.99 *+1.1*	88.97 *+2.2*	72.35 *+3.9*	84.73 *+1.9*	87.60 *+4.7*

Table 5.9: Training utilizing partial COCO-weights + *fakemainclass-multilabel pre-training*

The results shown in Table 5.8 and Table 5.8 indicate that stacked pre-training does not further increase prediction quality compared to the normal *multilabel* pre-training experiments of the previous section. While it still improves prediction performance compared to the baseline, the overall performance is worse compared to the non-stacked *multilabel* pre-training.

6 DISCUSSION

This chapter contains an overarching evaluation and interpretations of the experiment results from the previous chapter. In Section 6.1, the effectiveness of pre-training are discussed with focus on *multilabel-mainclass* experiments. Section 6.2 discusses different theoretical and practical limitations of this work.

6.1 EFFECTIVENESS OF PRE-TRAININGS

As shown in Section 5.6, utilizing a *multi-label* pre-training is the most effective pre-training, which outperforms other pre-training variants. The effectiveness can be slightly improved by using *fake main-classes* during the pre-training so that all weights from the pre-trained model can be utilized.

How effective a dedicated pre-training is compared to increasing the amount of high-quality data, can be estimated by comparing the baseline *main-class* results using *HQv2* with the corresponding multilabel pre-training using *HQv1*. Both experiments are marked bold in Table 6.1.

#	Iteration	HQv1-VAL				FINAL-TEST			
		APB	APM	APM^{w}	RC	APB	APM	APM^{w}	RC
1	10,000	78.82	59.99	75.40	81.21	80.84	51.41	75.57	78.88
2	16,000	79.05	60.77	76.14	82.00	82.11	58.98	77.45	81.05
3	**9,000**	**83.90**	**66.10**	**80.82**	**85.86**	**86.79**	**68.47**	**82.79**	**82.89**
4	8,000	82.94	66.02	79.93	86.54	87.94	71.67	83.64	86.48
5	**1,000**	**84.03**	**68.33**	**80.99**	**85.48**	**84.96**	**66.62**	**80.43**	**83.41**
6	2,000	85.55	70.86	82.87	87.60	89.87	75.73	86.29	87.60

Table 6.1: Comparing *multilabel-mainclasses* baseline results with *HQv1* using multilabel-pretraining.
1: Naive HQv1 training
2: HQv1 training using partial weights pre-trained on COCO,
3: **Naive HQv2 training**
4: HQv2 training using partial weights pre-trained on COCO
5: **HQv1 training using *multilabel fakemainclass* pre-training**
6: HQv2 training using *multilabel fakemainclass* pre-training
Additionally, the iteration number of the selected best model is included.

For context, *HQv1-TRAIN* consists of 194 pages/504 training image splits (cf. Table 3.6.2). The *HQv2-TRAIN* set consists of 683 pages/1690 training image splits (cf. Table 3.6.3), which is more than three times this size. Therefore, the pre-training significantly reduces the amount of manually annotated data needed in order to reach a certain level of prediction quality. As a rough estimate based on the results, including the pre-training might correspond to a naive training with approx. ~2 - 2.5x the manually annotated data set size[1].

The *HQv1* baseline results (#1, #2) suffer from the lack of training data compared to *HQv2* (#3, #4), however the pre-training (#5) effectively decreases this gap. On *FINAL-TEST*, it still does not reach the APM^{wo} of the naive HQv2 training (#4), but it even outperforms the latter considering the Recall RC.

Additionally, using a dedicated pre-training results in drastically faster convergence during the high-quality training. This allows for faster fine-tuning of parameters and experimentation in this training stage if a pre-training is already available. For future work, this implies that within the same time frame, more experiments can be completed. This greatly improves the practical viability of techniques such as hyperparameter optimization via random search [27], as they require a large amount of training experiments.

The next section discusses the effectiveness of the pre-trainings on classes that were not included during pre-training.

6.1.1 Classes not included during Pre-training

The pre-training does not only increase the prediction performance of all main-classes, even if they were not included during pre-training (cf. Appendix C.1). The difference is clearly visible

[1]This rough estimate is based on the assumption that there is a diminishing return in terms of the amount of training data leading to better results. Therefore, a further increase of training data leads to less and less improvements. For more precise estimates, more experiments with varying data set sizes across different object detection architectures should be considered.

even with partial COCO-pre-trained weights, as *COCO* does not include any GUI element classes.

As Table 6.2 shows, the pre-training siginficantly improves prediction quality for all classes, with the dedicated pre-training introduced in this work being more effective than using weights pre-trained on COCO.

Class	HQv1			HQv2		
	#1	#2	#3	#1	#2	#3
interactable	80.84	82.11	**84.96**	86.79	87.94	**89.87**
image	66.29	63.74	**72.58**	71.95	70.50	**76.76**
button	60.03	66.50	**68.50**	73.67	74.75	**78.06**
textfield	61.98	70.92	**83.35**	76.07	**85.75**	85.48
dropdown	19.33	25.98	**42.79**	32.14	54.71	**67.50**
checkbox-radio	10.61	38.36	**45.68**	**64.33**	53.25	54.45
text	83.30	84.41	**86.13**	87.64	88.77	**90.86**
icon	58.34	62.94	**67.31**	73.51	73.99	**76.98**
mAP	55.09	61.87	**68.91**	70.76	73.71	**77.50**
APM_{FT}	51.41	58.98	**66.62**	68.47	**75.73**	**75.73**
APM^{w}_{FT}	75.57	77.45	**80.43**	82.79	83.64	**86.29**

Table 6.2: Class AP scores for main-class experiments trained on HQv1/HQv2, evaluated on *FINAL-TEST*. Note that the mAP includes *Interactable*, APM_{FT} excludes this class.
#1: Naive training without pre-trained weights
#3: Using partial weights pre-trained on COCO
#3: Using *multilabel fakemainclass* pre-training

The *multilabel-29classes* experiments include a variety of additional *sub-classes* that provide a significantly worse data imbalance compared to including only *main-classes*. The impact on including those classes is discussed within the next section.

6.1.2 Sub-classes

The effect of different pre-trainings on sub-classes can be estimated by calculating the unweighted API and occurrence-weighted API^{wo} metrics that only include the $Icon$ sub-classes (cf. Section 4.4.1). Table 6.3 depicts the API and API^{wo} results on the validation/test sets considering different pre-trainings. Note that the value fluctuations of API is considered to be comparable to the noise of the APM metric, as both do not account for the heavy class imbalance. Therefore, the weighted API^{wo} has to be observed as a more reliable indicator for prediction performance if API results are close.

TRAIN	Pre-Training	HQv1-VAL		HQv2-VAL		FT-VAL	
		API	API^{wo}	API	API^{wo}	API	API^{wo}
HQv1	-	28.03	**35.35**	-	-	**25.64**	42.18
	partial COCO	28.38	31.51	-	-	24.20	37.68
	partial BIN	24.77	34.89	-	-	23.61	43.13
	partial ML	25.76	32.53	-	-	24.50	43.83
	partial COCO + partial ML	**28.82**	31.35	-	-	25.25	**44.31**
HQv2	-	26.41	37.44	31.76	45.11	28.81	56.09
	partial COCO	**36.03**	**47.54**	37.84	48.48	**36.14**	**62.49**
	partial BIN	28.29	39.77	30.86	41.57	29.93	58.85
	partial ML	32.38	44.92	**38.96**	**49.46**	33.24	57.52
	partial COCO + partial ML	34.88	46.97	37.54	48.63	35.90	59.88

Table 6.3: Icon sub-class AP scores using different pre-trainings.

Considering API^{wo} on $FINAL\text{-}TEST$, the naive training is outperformed by the use of any pre-training. However, the overall results are consistently mixed. This is due to to multiple reasons.

Including sub-classes heavily amplifies the data imbalance problem as shown in Appendix Table C.4. For, $FINAL\text{-}TEST$, 1950 out of 9998 elements are *Icons*, which is 19.5%. Considering these 1950 *Icon* elements, only 1028 have assigned any sub-class, which is approx. ~ 10% of all elements in $FINAL\text{-}TEST$. These fraction further splits into 19 sub-classes, which AP scores are measured using this metric.

Despite sub-class prediction performance not being the focus of this work, the *multilabel-29classes* experiments were included in order to measure the impact on the *main-class* prediction performance if more potentially noisy classes are present. The *multilabel-29classes* labeling includes all sub-classes considered during labeling, intentionally ignoring the *over-annotation* (cf. Section 3.3.1).

As the results indicate, including many additional classes tends to lower the overall recall. This is especially visible when comparing the RC results of the baseline *HQv1* experiments tested on $FINAL\text{-}TEST$ (cf. Table 5.2). However, utilizing any pre-training and a higher amount of high-quality training data as in *HQv2* mitigate this effect. Giving these circumstances, including these sub-classes did not have severe negative impact on the overall performance. Therefore, sub-classes with sufficient training data and prediction performance may be included in future models, while others are excluded.

For that reason, the model selection process was based solely on the APM^{wo} metric for *multilabel* experiments, which does only account for *main-classes*. As API and API^{wo} were completely neglected during model selection, this further increases the noise range for the API and API^{wo} metrics.

Including classification of many different sub-classes within a single model might not lead to reliable results. Therefore, expert classification models for certain sub-classes may be considered in future work. This is discussed in Section 6.2.3 among other limitations related to classification.

6.1.3 Summary

Utilizing a dedicated pre-training with a huge amount of automatically obtained and labeled web page screen shots can significantly improve the performance of GUI element detection even if the pre-training labels are limited by various factors. This can significantly reduce the amount of high-quality annotations needed in order to reach a certain level of prediction quality. For the maximum effect, utilizing pre-trained weights for as many layers as possible is recommended.

The pre-training does have a positive impact even on classes that were not explicitly labeled during the pre-training. Additionally, it can speed up the training duration for the training on the high-quality data if the pre-training is already available, which allows for faster experimentation during that stage.

Including many classes within a single model is practically limited by the strong class imbalance of different GUI elements. This is one of the limitations of the implemented approach and the experimental setup, which are discussed within the next section.

6.2 LIMITATIONS

Due to the experimentation setup, metrics and methodology, several limitations have to be considered. This Section discusses three remaining main obstacles that were observed during conception and evaluation of the approach implemented in this work. There is a statistical error range that has to be considered when interpreting the experiment results, which is discussed in Section 6.2.1. This is complemented by a brief discussion of the conceptual metric limitations in Section 6.2.2. Furthermore, the current implementation is still does not account for all types of GUIs, which is explored in Section 6.2.4. The Finally, Section 6.2.5 covers limitations introduced by the flat detection of bounding boxes.

6.2.1 Statistical Error Range

There is a statistical error range that has to be regarded when interpreting results. The purpose of this section is to introduce a rough estimate of the metric-dependent error range as well as the impact on the interpretation of results. Whether a certain result is considered within an error range depends on the metric under consideration as well as the results for other metrics. For this work, there are two reasons for value fluctuations.

First, due to random weight initialization. The more weights have to be initialized in advance of the training, the higher the influence of this random factor is. For the baseline training without pre-trained weights, all weights have to be initialized. When loading *partial* weights into *YOLOv3-SPP*, the first 81 layers are loaded from the pre-trained weights (see Section 4.3). This implies that only the initialization of the remaining layers differs between the experiment variations using specific pre-trained weights. If the class count does not change and therefore the full pre-trained weights can be loaded, no additional weight initialization is performed.

As this work utilizes many different experiments and two different data set versions, correlations that are consistently measurable still imply meaningful results. A possible workaround to further limit this implication is the repetition of experiments, where each set of experiments uses a dedicated seed. However, this greatly increases experimental and computational overhead as each experiment has to be performed multiple times.

The second reason is the model selection process described in Section 4.5. When training a model, checkpoints are saved every 1000 iterations which are then used in order to evaluate the model based on the APM^{wo} introduced in Section 4.4.2. As one iteration consists of 64 images[2], there is a notable gap between each evaluation. To account for this, the evaluation frequency can be set higher. In order to minimize the sum of saved weight file sizes, the calculation of these custom metrics can be directly incorporated into the training process in future experiments. This also increases the chance to consistently find model checkpoints with slightly better performance than previously.

How a certain result should be evaluated also depends on the metric under consideration as well as the results for the other metrics. The unweighted APM equals the mAP taking only main-classes without *Interactable* into consideration. This implies that for APM, every *class* has the same relevance. This metric is especially noisy due to heavy data set imbalance. A single false positive for a specific class AP has greater impact the less samples are available for this class. Therefore, the unweighted APM is significantly more noise-prone than the weighted APM^{wo} as miss-classifying a single *Checkbox* can already have visible impacts.

For APB, APM^{wo} and RC, every *object* has the same relevance. This implies that these metrics are generally more stable, which can be expected to result in less fluctuation.

Based on the observations of multiple experiment results, rough estimates of the error range are $\pm$~5% for APM and $\pm$~1% for APB, APM^{wo} and RC. The high error range of APM is the reason why APM^{wo} should be referenced to as the main metric for *multilabel* experiments. However, as the latter is biased towards element classes with many occurrences, the APM can still be included as as additional metric.

For the comparison of two different experiments, values of a single metrics inside these ranges are not sufficient indication for improvements or deteriorations. However, if these differences are consistent between multiple experiments of comparable types (such as HQv1/HQv2) and/or across different metrics, this has to be interpreted accordingly. For very close and mixed results, this limitation indicates that more experiments are neccessary in future work in order to gain meaningful results.

6.2.2 Metric Conception

There are conceptual differences between metrics that lead to individual limitations. As an example, classes are considered equally important when using APM where elements are treated equally for APM^{wo} (cf. Section 6.2.1). Therefore, a single metric only depicts a fraction of the overall prediction performance.

[2] which is equal to the batch size defined in the network configuration (see Section 4.1.3)

The metrics used in the evaluation are based on the IoU threshold $thresh^{IoU} = 0.5$ for the definition of True Positive detections (cf. Section 2.3.2), which is another limitation to consider. Some use cases of GUI element detection such as GUI code synthesis may require another level of bounding box reliability as for example interacting with applications. Therefore, a single IoU threshold does not account equally for all use cases.

Furthermore, the IoU threshold does not give further information about the quality of the bounding box prediction besides exceeding the threshold (Figure 6.1). Counting True Positives based on the IoU threshold does not acccount for differences in the perceived importance of elements as well. For example, a small *Image* is treated as important as a much larger one.

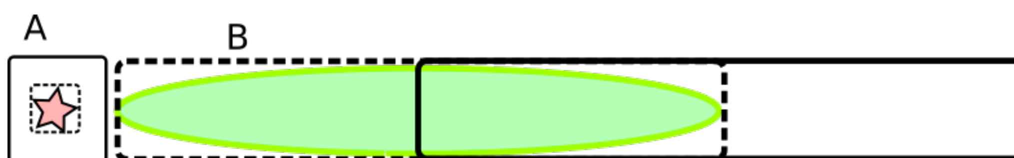


Figure 6.1: IoU might not always reflect the perceived bounding box quality.

To account for these limitations, metrics that use different IoU thresholds and distinct between elements of different sizes should be considered in future work.

6.2.3 Classification Limitations

In order to make experiments comparable, the main focus of the evaluation was the impact on the overall prediction quality considering *main-classes*. However, the annotation process introduced in Section 3.3 accounts for a variety of sub-classes, especially for the *Icon* main-class. These are exclusive to the *29Classes* experiments, not comparable to the other experiments and not included in the pre-trainings as well. Therefore, evaluating and improving these sub-classification results was not the focus of this work.

As the results indicate, the inclusion of these sub-classes might reduce the overall recall for training on a smaller data set such as *HQv1*. Using *HQv2* and a pre-training, the inclusion of these sub-classes do not drastically decrease main-class prediction performance. However, the prediction quality within these sub-classes greatly differs due to the variety of different *Icon* sub-types and the strong data imbalance (cf. Section 6.1.2).

For this reason, an adopted two-staged approach can be considered for future work. For example, an object detection model can be trained to detect main-classes, while seperate, dedicated classification models are fully optimized to distinct between specific sub-classes. While the key idea of this approach is similar to two-staged object detection implemented in algorithms such as *R-CNN*, which is explained in Appendix A.3, this has three major advantages.

In contrast to early *R-CNN* models, the classification model has not to account for hundreds or thousands of object candidates per image that are no actual objects. The model has still to handle for example *Icon* predictions from an object detector that are no real icons. However, the amount of those false positives is greatly reduced in contrast to R-CNN models which lowers the difficulty of the classification task.

The classification would not have to handle all element classes, but only a subset of them such as the *Icon* subclasses. This allows for the dedicated creation of more balanced data sets in order to handle imbalance of real-world data. This also allows for the investigation of other classification algorithms and/or models which may perform better, especially considering the lack of data for some sub-classes. As an example, if an *Icon.Arrow* is detected, a very simple model could be used to distinct between the arrow directions. Distincting between arrow directions is not considered being a hard problem for any modern object detection model. However, the strong class imbalance between all element types may hinder even simple sub-classes from being recognized reliably.

While the classification step still needs additional time for computation and therefore affects overall performance, this impact is greatly reduced compared to *R-CNN* runtimes (cf. Appendix A.3) for two main reasons. First, as already stated, the classification is only needed for a subset of all classes and elements. Second, depending on the use case, the classification can be implemented as optional during runtime. As an example, a web crawler using screenshot-based detection of elements as well as element type predictions for crawling heuristics can be considered. A heuristic based on *Icon* sub-classes may only be triggered in certain situations or states. Therefore, the classification of *Icon* sub-types can performed only when needed, which allows for flexibility and removes most of the overhead of two-stage prediction.

6.2.4 GUI Type Limitations

As explained in Section 3.1, the main focus of this work are web GUIs. More specifically, GUIs that are similar to most modern web pages. This does not yet account for elements that may be occur more commonly in other types of GUIs such as sliders/switches in mobile apps (Figure 6.2).

Figure 6.2: Switch Buttons

Besides additional element types, extending the principles to fit for other GUI types may have to account for other characteristics as well. For example, a model trained to recognize textfields on screenshots of desktop applications might not recognize textfields on mobile applications reliably. This behaviour may not be obvious, as textfields in mobile apps appear very similar to their desktop-counterparts for a human. However, the size of elements in mobile apps are designed according to the screen size. This results in textfields that may have an absolute size similar to desktop-counterparts, but the size ratio compared to the entire screen is different. As stated in Section 3.5, the model input is resized to a fixed resolution. The textbox stretches over the entire width of the input, where this is not the case for most desktop textboxes. In edge cases where this occurs, the models have problems recognizing these text boxes as well. As an example, the Google Chrome Screenshot depicted in Figure 6.4 has two textboxes. The large textbox on the top of the screen is such an edge case and is not recognized by the models either. The textbox at the center of this screenshot is recognized normally. As a result, textboxes in mobile applications are not recognized well.

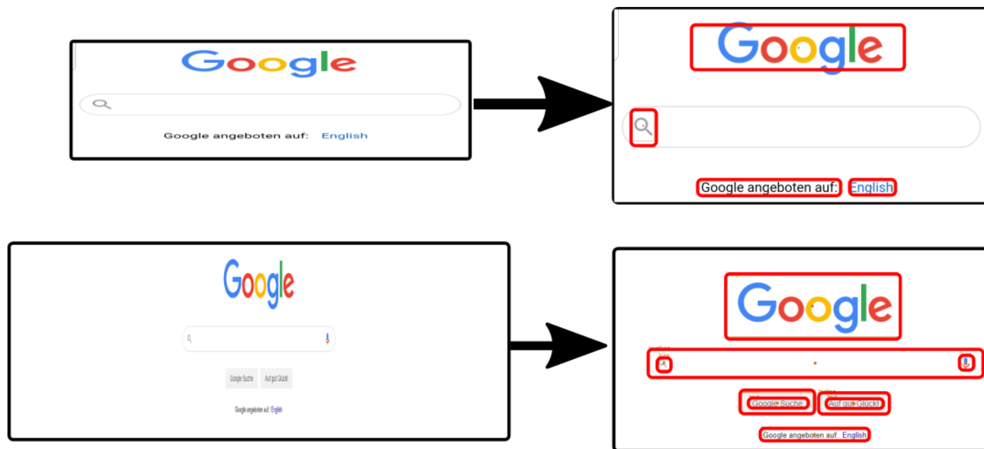


Figure 6.3: Top: Mobile Screenshot stretched. The model is not used to textboxes this broad.
Bottom: Same site, but the screenshot is taken from the desktop variant. Textboxes on desktop do not stretch over the entire screen in most cases.
Image contains arranged Screenshot of [img12]

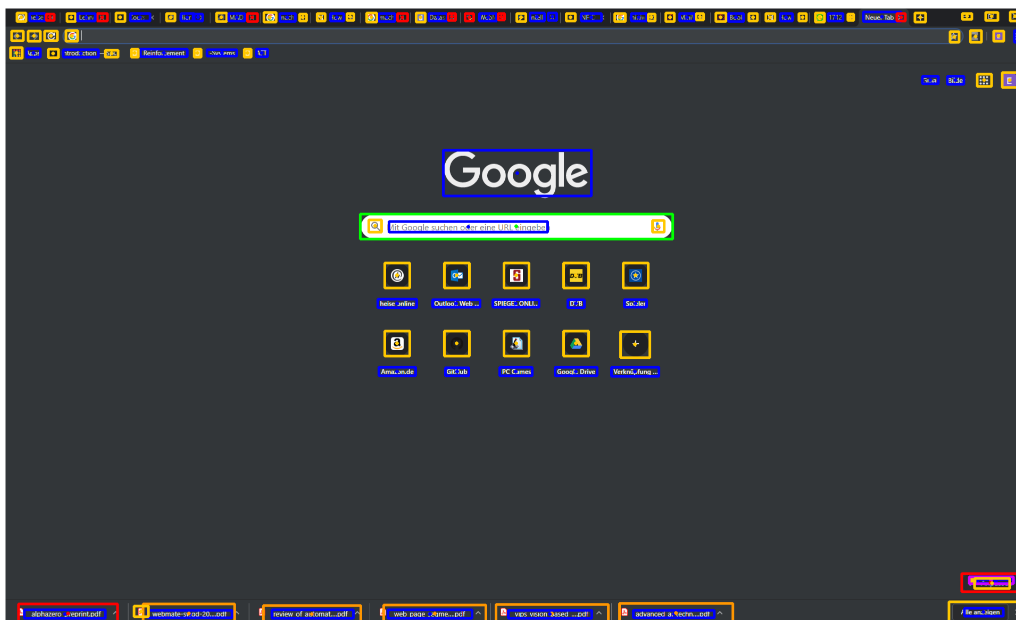


Figure 6.4: Model prediction of [img8]. The large textbox at the top is not recognized.

Another problem occurs when trying to handle large and dense texts. The crawl that was the original data source mainly included pages that are strongly structured. Text mainly occurs in small paragraphs or even just menu entries. Large texts such as long Wikipedia [u34] articles do not occur frequently. The model does have significant problems if very large texts are present.

Figure 6.5: Left: Long dense texts causes significant problems for the detector.
Right: Structured, smaller text blocks on the same page are detected as intended.
Image Source: Model Prediction of [img13] using the best *HQv2*-trained *multilabel-mainclasses* model with *fakemainclass* pre-training

These are just two examples of implications and differences that may occur when testing other GUI types. To account for these differences, specific data sets for the GUI type in consideration have to be created and/or combined with the existing ones in future work. This includes the adaptation of class types as well, depending on the use case. Additionally, dedicated data augmentation of existing data can be considered as well, for example cropping already labeled screenshots so that textboxes full the entire width.

6.2.5 Tree Structures

The GUI element detection implemented in this work does *not* include the explicit detection of visual separators or tree structures of elements as provided by methods like DOM analysis, which is discussed later in Section 3.2.2[3]. Detecting elements directly allows for a more straightforward human-like use of results for a variety of tasks like automated interaction, while allowing a higher independence of underlying technology. Tree structure estimations based on a screenshot of the rendered GUI without access to the underlying data structure can be ambiguous (Figure 6.6). Direct detection of rendered elements circumvents this problem, which facilitates data annotation and generalization for different types of GUI technology as well. The *atomic labeling* principle (cf. Section 3.3.1) slightly accounts for layered structures, which may already be sufficient for a range of applications.

However, if a full tree of rendered elements is required, using at least a combination with a debugging interface such as *Selenium* might be more suitable than purely visual detection, since it does not have to be used exclusively. This also allows to leverage the advantages of both approaches, while bypassing individual limitations such as outlier-sensitivity discussed in Section 6.2.

Detecting structures is also possible based on bounding box results, using algorithms such as *Box Clustering Segmentation*[28], which has been developed for that purpose.

[3]Detection of visual separators will however be learned implicitly by the object detection network.

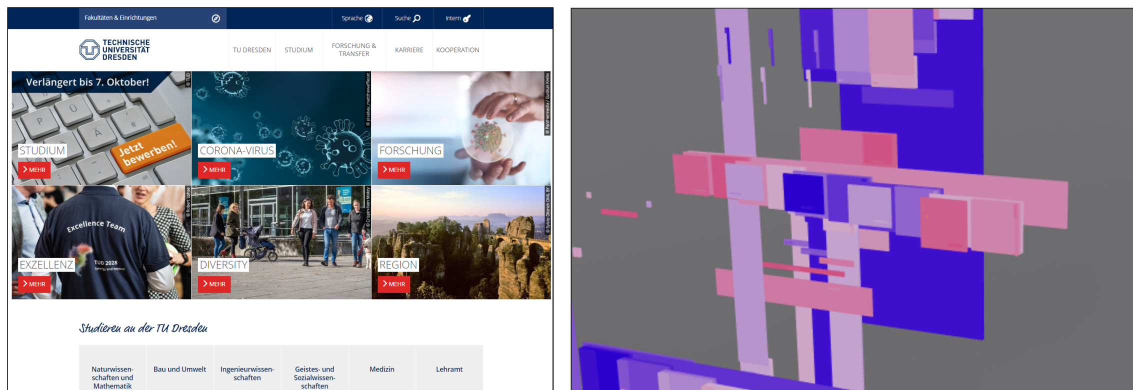


Figure 6.6: The detection of deeply hierarchical structures is especially difficult and in some cases ambiguous based on visual detection only.
Left: Excerpt of TU Dresden Web Page [img14].
Right: TU Dresden web page rendered by Microsoft Edge DevTools 3D DOM View Prototype [u35].

7 CONCLUSION

Considering the results and the limitations discussed in the previous chapter, the final chapter concludes this thesis. For that, Section 7.1 provides a brief overview about related work and sets this thesis in context. Based on that, Section 7.2 gives an outlook for future work. The last Section 7.3 provides as final summary of this thesis.

7.1 RELATED WORK

GUI element detection is the application of object detection for GUI elements. Therefore, a variety of common object detection algorithms and techniques such as the already mentioned R-CNN [4, 5, 6] and YOLO [7, 8, 9] architectures are directly related to this task. As deep learning based models evolve rapidly, newer or improved architectures release regularly such as *YOLOv4* [29], EfficientDet [30] or SpineNet [31].

Due to the amount of research in the area of object detection and deep learning in general, it can be difficult to keep track with the constantly released and updated techniques. Even though they might be incomplete, online sources such as [u36] may provide substantial help for getting an rough overview about state-of-the-art techniques. The COCO data set [10, u3] is the current main benchmark for common object detection techniques.

Other paradigms (cf. Section 2.2) might also be considered in the context of the GUI element detection class imbalance problem. For example, *Active Learning* [32] aims to actively estimate which samples should be included to improve the model the most. The semi-supervised method *Pseudo-Labeling* [33] leverages additional unlabeled data by training a model on its own predictions.

YOLO9000 [8] proposed a mechanism for training hierarchic classifications. The *WordTree* approach calculates a tree of probabilities via multiple *softmax* (cf. Figure 2.8) calculations. To specify the final class, the tree is traversed from the root along the highest confidence path. For the data sets of this thesis, the root node would be *Interactable*. The child notes for *Button*, *Image* and other classes would then represent the conditional probabilities $Pr(Button|Interactable)$ et cetera.

The classification loss is only backpropagated at the level of the provided class and above. For example, the classification loss for an (non-subclass) *Icon* would just be backpropagated for *Icon* and *Interactable*, but not *Icon.Search*. Compared to the *multilabel* approach for hierarchic classification, this ensures that any node of the tree is also classified as its predecessor. This is not always the case when using *multilabel* classification (cf. Section 4.4.3). For example, some elements that should be classified as *(Interactable, Image)* may just have the *Image* class assigned. However, this might be easily fixed in post-processing. In contrast, the *multilabel* approach allows classifications such as *(Interactable, Image, Text)* for logos as well, which is not possible using *softmax* as both *Image*and *Text* would be on the same level of the tree.

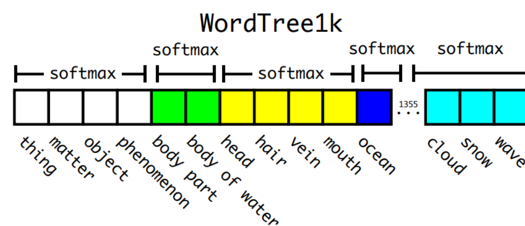


Figure 7.1: WordTree utilizes multiple *softmax* calculations. To determine the predicted class, they traverse along the class hierarchy along the nodes with the highest predicted confidence. The final class probability can then be calculated by multiplying the confidences to conditional probabilities. Image Source: [8]

The comparison of the *multilabel* pre-training experiments of Section 5.5 and the stacked pre-training of Section 5.6 indicates that the latter has a worse overall performance. Zoph et al. [31] found that if a pre-training is utilized, strong data augmentation or a high amount of labeled data in the second stage not only result in diminishing effects of the pre-training. They may even *decrease* the final prediction performance by about 1% *AP*. Since the *LQDOM* pre-training used in this work already includes a huge amount of training data (cf. Section 3.6.1), this effect may be an explanation for the results of the stacked pre-training experiments.

As mentioned in the introduction, detecting GUI elements based on visual detection may support different software engineering tasks. Therefore, a variety of approaches have been explored in the literature that directly cover GUI element detection or related problems.

Box Clustering Segmentation[28] aims to detect structures on web pages based solely on bounding box inputs. Clustering bounding boxes may for example be used to detect headers, menus or other groups of elements that belong together visually.

REMAUI [1] and Pix2Code [2] recreate GUIs based on screenshots. However, they are mostly evaluated on simple and artificial GUIs (Figure 7.2).

(a) Groundtruth GUI 1 (b) Generated GUI 1 (c) Groundtruth GUI 2 (d) Generated GUI 2

Figure 7.2: Example GUIs used in pix2code. Image Source: [2]

A recently published study by Chen et al. [34] covers the application of different approaches on the *Rico* data set [13], which is a data set containing screenshots as well hierarchic GUI information and meta data of over 9,000 mined Android applications.

The study compares the performance of several traditional computer vision approaches for detecting and classifying regions as well as deep learning-based object detectors such as *YOLOv2/v3*. They conclude that deep learning based methods significantly outperform traditional computer vision algorithms for detecting the regions of non-text GUI elements, but often fail to predict very precise bounding boxes and may need lots of training data. The traditional algorithms such as REMAUI [1] had problems regarding more complex GUIs. The detected element regions of the tested traditional algorithms were mostly noise. However, if they detected True Positives, the predictions were quite accurate. The performance of deep learning based algorithms significantly dropped for higher IoU thresholds such as $thresh^{IoU} = 0.9$. Very high bounding box accuracy might be neccessary for some use cases such as GUI code synthesis.

They also observed that the classification of elements is challenging due to GUI elements having large in-class variance and high cross-class similarity. For example, the *ToggleButton* samples depicted in Figure 7.3 can be very similar to the *ImageButton* samples. Distuinguishing between those often requires *implicit knowledge* about the implementation or functionality of these icons. What distinguishes a *ToggleButton* from a regular *ImageButton* is the *toggling* functionality. Based only on a screenshot without the ability to validate the assumption through interacting, this may be hard to determine for humans as well in some cases. In this thesis, this problem was addressed by utilizing a hierarchic classification scheme where element classes were defined by visual properties only (cf. Section 3.3.1) as well as multilabel classification. In that example, the *ToggleButton* may just be implemented as a sub-class of *ImageButton* or *Icon* to allow a more robust and flexible classification.

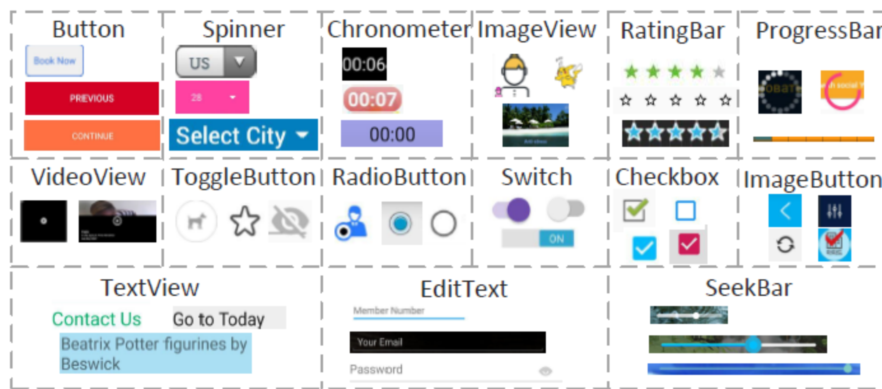


Figure 7.3: Class examples from [34] based on the Rico data set [13]. Samples may significantly vary within a class, while classes can be very similar. Image Source: [34]

Another aspect of the study was the question if text and non-text elements should be detected seperately. They conclude that neither OCR techniques nor object detection models such as *YOLO* can reliably detect GUI text and that scene text recognition algorithms such as EAST [35] are needed.

In this work, significant problems regarding large texts were observed. Text scene recognition algorithms such as EAST may achieve better recognition of *Text* elements and solve the problems with large texts. However, good results were achieved on smaller structured text blocks (cf. Figure 6.3) and therefore the majority of text elements in the utilized data sets (cf. Table 6.2). This indicates that a seperation is not neccessary for all use cases if only small text blocks appear.

They also proposed a combined algorithm that uses a traditional algorithm for object localization using a top-down coarse-to-fine strategy. This greatly improves the bounding box quality compared to object detection models and does not need training data. For the localization text elements, EAST is utilized. All other non-text elements are classified by a neural network for image classification.

The proposed algorithm is shown to outperform other techniques for $thesh^{IoU} = 0.9$ significantly on the Rico data set. However, the performance considering a more tolerant IoU theshold which may be sufficient for other use cases as well as the performance on other types of GUIs remain open. To account for these differences, variations of the *AP* metric can be used that average over different IoU thresholds or treat objects of different sizes seperately [23]. This can be addressed in future work, which is discussed in the final Section.

7.2 OUTLOOK - FUTURE WORK

With the study referenced in the last Section being released very recently, a direct comparison of the proposed algorithm with the model of this thesis may be considered. Using *HQv1-VAL*, *HQv2-VAL* and *FINAL-TEST*, for example recall and speed of both techniques could be directly compared considering different IoU thresholds.

To improve the classification of GUI elements, hierarchic labeling may be further explored. For

distinguishing between certain sub-classes, lightweight expert models can be implemented. This may also help handling the heavy class imbalance problem. For example, it can be considered that a powerful, harder-to-train deep neural network may not be neccessary in order to distuinguish between certain sub-classes such as different *Icon.Arrow* types.

Another possibility of improving the classification considering the class imbalance is the implementation of additional paradigms such as *Active Learning* or *Pseudo-Labeling* that were mentioned in the last section. This may be combined with domain- or technology-specific techniques such as DOM analysis in order to more efficiently leverage crawled GUI data.

As object detection algorithms evolve rapidly, the performance of other architectures mentioned in the previous section might be evaluated using the existing techniques.

To further improve deep learning based models, data from different GUI types should be included. Training and evaluating based on a single data source or GUI type results in GUI type limitations such as the problems discussed in Section 6.2.4. In order to achieve true platform-independent recognition of common GUI elements, combining different data sources and/or data sets as well as metrics is highly recommended.

7.3 SUMMARY

Considering the variety of different GUI designs and technology, the detection of arbitrary GUI elements remains a challenging task.

The application of state-of-the-art deep learning based object detection algorithms requires a huge amount of training data. This thesis contributes by lowering the amount of manually annotated high-quality training data needed to achieve a certain prediction quality.

It is shown that leveraging automatically obtained, non-perfect annotated GUI data from easily accessible sources can lead to notable improvements for deep learning based GUI element detection. A dedicated pre-training using is more effective than pre-training on generic object detection data sets, even if both bounding box and class label quality is limited. This is especially effective if little high quality-labeled data is available.

Additionally, the use of hierarchic element class definitions based solely on visual properties as well as multi-label classification are introduced. This allows for flexible classification which can be easily extended to account for other element types as well as the implementation of sub-type specific expert models in future work.

A GLOSSARY

A.1 ABBREVIATIONS

Abbreviation	Meaning	Section
AI	Artificial Intelligence	A.2
AP	Average Precision, Area under Precision-Recall Curve	2.3.4
APB	AP of class *Interactable*, Main metric for binary experiment variants	4.4.1
APM	(mean/unweighted) main-class AP. Equal to mAP considering only main-classes. Can be noisy, high error range.	4.4.1
APM^{wo}	occurrence-weighted main-class AP, Main metric for evaluating *multilabel* experiments	4.4.2
Cls	Labeling variant	4.2
R-CNN	*Regions with CNN features* (Object Detection Architecture)	A.3
COCO	*Common Objects in Context*, Reference object detection data set[10, u3]	7.1
DOM	Document Object Model	3.2.2
FN	False Negative	2.3.2
FP	False Positive	2.3.2
GUI	Graphical User Interface	3.1
HQv1/v2	High-Quality Data Set v1/v2	3.3
IoU	*Intersection Over Union*	2.3.1
Iter	Iteration at which the best model was selected, 1 Iteration equals batch size (64 images)	4.5
LQDOM	low-quality DOM-labeled data set, automatically obtained by web crawling and DOM analysis	3.2
PR-curve	Precision-Recall curve	2.3.4
RC	Class-independent Recall	4.4.3
TP	True Positive	2.3.2
$thresh^{conf}$	Confidence Threshold	2.3.2
$thresh^{IoU}$	IoU Threshold	2.3.2
YOLO	You Only Look Once (Object Detection Architecture)	2.4

A.2 MACHINE LEARNING

Machine Learning is a part of Artificial Intelligence (AI). The term sums up different approaches, methods and algorithms that address the automated recognition of patterns and regularities in order to automatically evaluate new, unseen samples of data.

Training

For supervised learning, given data set D_{TRAIN}, the objective is to find a function f that approximates $f_{GroundTruth}(x) = y, (x, y) \in D_{TRAIN}$ as accurately as possible.

This is achieved through step-wise minimization of a loss function (see A.2). In the case of neural networks, this is mainly done by adapting weight factors w_i. The step-wize adaptation of f is called *Training*.

The approximation f is also called *model*.

A training cycle over the entire training data set is called *Epoch*. Normally, a training of neural networks consists of many epochs.

If the loss of a model further and further approximates the optimal value as training progresses, this is called *Convergence*. If for example *Hyperparameters* are set badly, it is possible for the model not to converge (or even *diverge*).

Validation and Testing

After training an estimator f trained on D_{TRAIN}, f is evaluated using D_{VAL} by calculating specific metrics. D_{VAL} includes labeled ground truth samples (even when training unsupervised) on input data not used during training. This is called *validation*.

To prevent information leakage (cf. Section 3.3.2) it is also common to hold back extra data that is evaluated last when all experiments are finished and models are selected. The held back data is called *Test Set*.

A common approach to determine the training duration is to monitor a target validation metric through frequent validation and stop the training if the validation score did not increase anymore after a certain duration, which is often a defined amount of epochs. This technique is called *Early Stopping*.

Loss

The loss function, which is sometimes called *cost function*, is a function for estimating the model error. The main objective of model training is the minimization of the loss function[1].

An example is *(Mean) Square Error* (Formula A.1).

$$(M)SE(f) = (\frac{1}{N}) \sum_{i=1}^{N} (y_i - f(x_i))^2 \quad \text{(A.1)}$$

x — : input data/"features"
y — : label corresponding to x
$f(x)$ — : model prediction
N — : number of samples in D (data set size)

Note that object detection networks such as *YOLO* have own sophisticated loss functions, which are not covered by this work.

Hyperparameter

Parameters that directly influence the training algorithms itself like the *Learning Rate* are called *Hyperparameter*.

The step size used in *Gradient Descend* is called *Learning Rate*. It is a scalar the magnitude of the gradient is multiplied with before adapting the weights. If the *Learning Rate* is set too small, steps taken in the process of *Gradient Descend* are too small which leads to slow convergence and not being able to escape from local minima. If it is set too high, it is possible to overshoot global minima. Optimization techniques like *Adam* make use of dynamic *Learning Rate*s which are adapted during the training process for each single weight.

A.3 R-CNN

Detecting arbitrary objects in images depicts a hard challenge in computer vision. Before the release of the R-CNN algorithm in 2014[4] state-of-the-art algorithms consisted of ensembles of models, utilizing low-level features in combination with higher-level context. The usage of CNNs suffered from the fact that the number of objects to recognize in an image is unknown, while the number of output neurons in the final fully connected layers had to be fixed. The R-CNN algorithm solved this by generating a fixed number region proposals and feeding the cropped image of each proposal into a CNN.

[1] In contrast, a *metric* like *AP* is designed to be interpreted by humans, not to have properties that are suited for optimization problems.

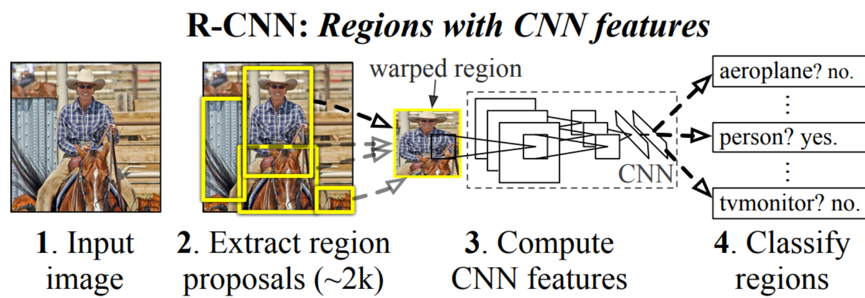


Figure A.1: R-CNN algorithm. Source: [4]

As shown in Figure A.1, the R-CNN algorithms consists of these main steps:

1. **Input Image**: Read image data and perform pre-processing (e.g. rescaling).
2. **Region Proposals**: A total of 2000 region proposals was generated. Note that in contrast to later architectures, R-CNN used a conventional computer vision algorithm (*selective search*) in order to generate region proposals.
3. **Feature Extraction**: Using a Convolutional Neural Network, the features of each proposal are extracted
4. **Classification**: Classify each feature vector by using class-specific linear *Support Vector Machines (SVMs)*.

However, this approach had some disadvantages:

- The region proposals being generated by a fixed algorithm prevents it from using the advantages of machine learning at this stage
- This disadvantage further worsens since the object classification is completely dependend on the localization. Furthermore, no context beyond the provided crop can be utilized.
- High computational cost in order to process one image. Due to the CNN being used on each single region proposal as a crop, an R-CNN pass of one input image equals the runtime of 2000 image predictions of the CNN plus the runtime of the other steps. Therefore, it is not suited for real-time usage. The first version of the R-CNN needed almost 50 seconds runtime on a Nvidia K40 GPU in order to process a single image [5].

B DOM ANALYSIS FILTERS

These filters were applied for labeling elements using Selenium and DOM analysis. The filters are notated in Pseudo-Logic. Functions that access node information were implemented using Selenium. For each element/node provided by the DOM, the $Filter()$ function is evaluated based on the properties of the element. If the function evaluates to *true*, the element is included as *Interactable*. Otherwise it is discarded.

Each function that checks for an element property is applied to the element currently filtered. For example, *IsDisplayed()* should be read as *IsDisplayed(currentElement)*, which has been shortened to improve readability.

The notation $Condition \rightarrow Class$ indicates that the element is additionally labeled as $Class$ if $Condition$ is fulfilled and $Filter()$ becomes *true*.

Note that this implementation existed prior to this work and has been included for better comprehensibility and reproducibility.

Filter():

$IsDisplayed()$

$\wedge ((AreaGreaterThan(200)$

$\wedge WidthInPercentOfScreen < 50$

$\wedge \neg HasTag(option) \wedge \neg HasTag(optgroup)$

$\wedge (labelFilters() \vee anonymousFilters()))$

$\vee IsCheckbox()$

$\vee IsRadioButton())$

AnonymousFilters():

$HasTag(button)$

$\vee HasTag(input)$

$\vee (HasTag(a)$

$\wedge (HasImageChild()$

$\vee HasText()$

$\vee HasCssContent())$

$\wedge \neg HasBiggerMarkedChild())$

$\vee HasBackgroundImage()$

LabelFilters():

$\neg ContainsAttributeValues(class, [wrapper])$

$\wedge (100 < Area() < 15000)$

$\wedge (IsButton() \rightarrow \textit{Button}$

$\vee IsTextfield() \rightarrow \textit{Textfield}$

$\vee IsImage() \rightarrow \textit{Image}$

$\vee IsCheckboxRadio() \rightarrow \textit{CheckboxRadio}$

$\vee IsDropdown() \rightarrow \textit{Dropdown}$

$\vee IsMisc()$

IsButton():

$$((ContainsAttributeValues([class, button, Button, btn, Btn, control])$$
$$\lor HasTag(button)$$
$$\lor (HasTag(input) \land HasAttributeValues(type, [submit]))$$
$$\lor HasTag(svg))$$
$$\land \neg HasTag(div)$$
$$\land \neg HasEnclosingMarkedParent()$$
$$\land \neg IsRadioButton()$$
$$\land \neg ContainsAttributeValues(class, [buttons, Buttons, btns, Btns])$$
$$\land \neg HasAttributeValues(type, [submit, text, password, email, search, username])$$

IsTextfield():

$$HasTag(input)$$
$$\land (ContainsAttributeValues(type, [text, password, email, search, username, form, find])$$
$$\lor ContainsAttributeValues(class, [search, input, form, find]))$$
$$\lor HasAttributeValues(type, [text, password, email])$$

IsImage():

$$HasTag(img) \lor HasTag(svg)$$
$$\land Area() > 1500$$

IsCheckboxRadio():

$$HasTag(input)$$
$$\land (HasAttributeValues(type, [checkbox])$$
$$\lor HasAttributeValues(type, [radio])$$
$$\lor HasAttributeValues(role, [radio])$$

IsDropdown():

$$(ContainsAttributeValues(class, [select])$$
$$\lor ContainsAttributeValues(class, [dropdown])$$
$$\lor HasTag(select)$$
$$\lor HasAttributeValues(type, [radio])$$
$$\lor HasAttributeValues(role, [radio]))$$
$$\land \neg ContainsAttributeValues(class, [selected])$$
$$\land \neg HasMarkedParent()$$

IsMisc():

$$HasOwnText()$$
$$\lor HasTagCheckParentAlso(a)$$
$$\lor HasTagCheckParentAlso(li)$$

The additional functions described in Table B.1 are implemented using Selenium [u18] and the JSoup Java HTML Parser [u37].

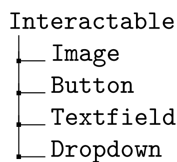
Function	Explanation
IsDisplayed()	Checks if the element is displayed (Selenium)
Area()	Calculates the area of the current element
WidthInPercentOfScreen()	Calculates the element width relative to view port size
HasTag(a)	Checks if the DOM element has the tag <a>
HasTagCheckParentAlso(a)	Checks if the current element or its parent element has the tag <a>
HasImageChild()	Checks if the element has a child node that has an <image> or <svg> tag
HasText()	Checks if the element has text (JSoup)
HasCssContent()	Checks if the element has the CSS value *content*
HasBiggerMarkedChild()	Checks if the element has a child element that fulfills *Filter()* and has a bigger size than the current element
HasBackgroundImage()	Checks if the element has the CSS value *background-image* and if it is neither *None* nor empty
ContainsAttributeValues(a, [b,c, ...])	Checks if the element attribute string *a* contains at least one of the strings **b, c, ...**
HasAttributeValues(a, [b, c, ...])	Checks if the element has the attribute *a* that has one of the values *b, c, ...*
HasMarkedParent()	Checks if the element has a parent element that fulfills *Filter()*
HasEnclosingMarkedParent()	Checks if the element has a parent element that fulfills *Filter()* that visually encloses the current element
HasOwnText()	Checks if the node has a direct successor node that contains text

Table B.1: Additional functions that access element properties. These functions where implemented using the *Selenium* and *JSoup* Frameworks.

C CLASSES

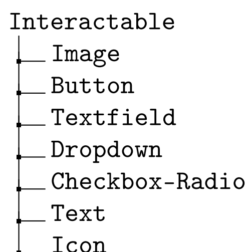
C.1 DOM-CLASSES

These classes were included in the DOM analysis to create LQDOM prior to this work:



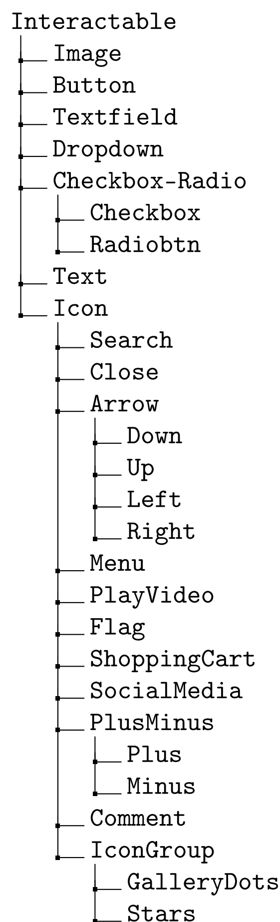
C.2 MAIN-CLASSES

The following classes are defined as *Main Classes*, which have been identified as the most atomic classes most webpages can be segmented into:



Note: Some logos can be both "Image" and "Text".

C.3 ALL CLASSES AND SUB-CLASSES



C.4 CLASS COUNT OVERVIEW

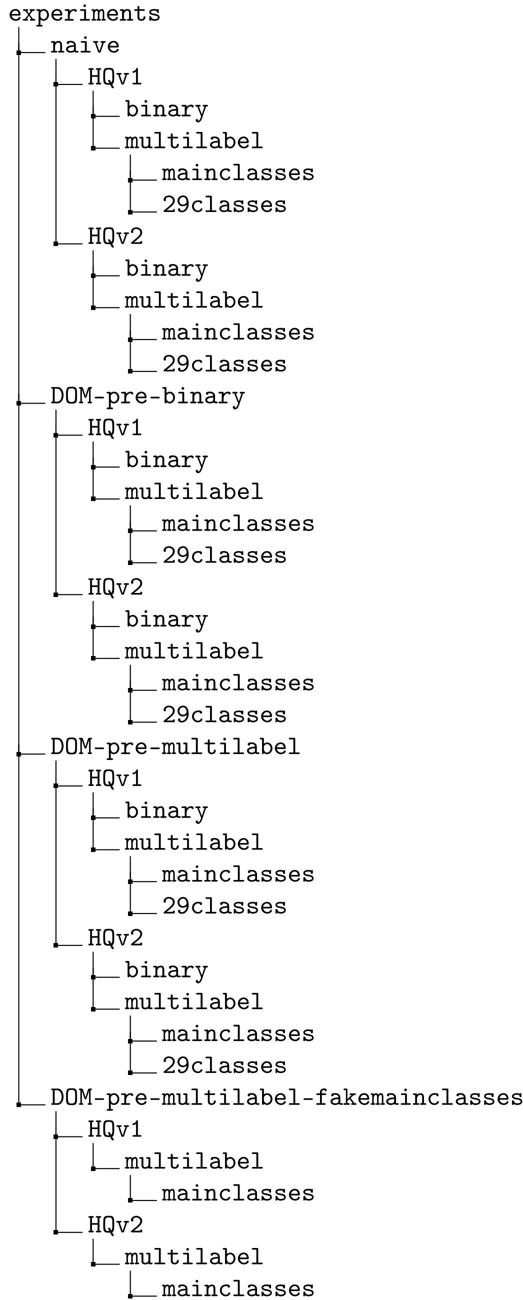
Table C.1 includes detailed size information for all data sets.

Class	LQDOM	HQv1		HQv2		FINAL-TEST
	TRAIN	*TRAIN*	*VAL*	*TRAIN*	*VAL*	*TEST*
interactable	8,905,452	22,233	5,579	72,552	18,644	9,998
image	894,218	2,723	670	9,091	2,181	815
button	351,755	1,081	321	3,607	9,04	329
textfield	102,126	209	69	773	194	75
dropdown	82,904	146	25	243	98	26
checkbox-radio	27,002	142	12	219	103	53
.checkbox	-	103	12	0	0	45
.radio	-	33	0	0	0	8
text	-	15,197	3,746	46,358	12,686	6,716
icon	-	2,614	724	8,864	2,447	1,950
icon.search	-	125	42	390	111	49
icon.close	-	63	26	227	65	25
icon.arrow	-	723	169	2,316	640	568
.down	-	279	61	893	255	147
.up	-	20	10	34	26	36
.left	-	66	12	164	49	32
.right	-	358	86	1,225	310	347
icon.menu	-	24	12	88	22	23
icon.playvideo	-	100	26	350	87	23
icon.flag	-	8	6	21	13	0
icon.shoppingcart	-	30	9	82	30	28
icon.socialmedia	-	401	139	1,586	349	220
icon.plusminus	-	72	13	153	79	41
.plus	-	52	12	81	60	33
.minus	-	20	1	72	19	5
icon.comment	-	28	5	137	25	6
icon.icongroup	-	186	25	431	163	45
.stars	-	130	19	316	120	25
.gallerydots	-	59	6	101	46	20
Whole Page Count	93896	194	50	683	161	65
Split Count	229755	504	127	1690	411	215

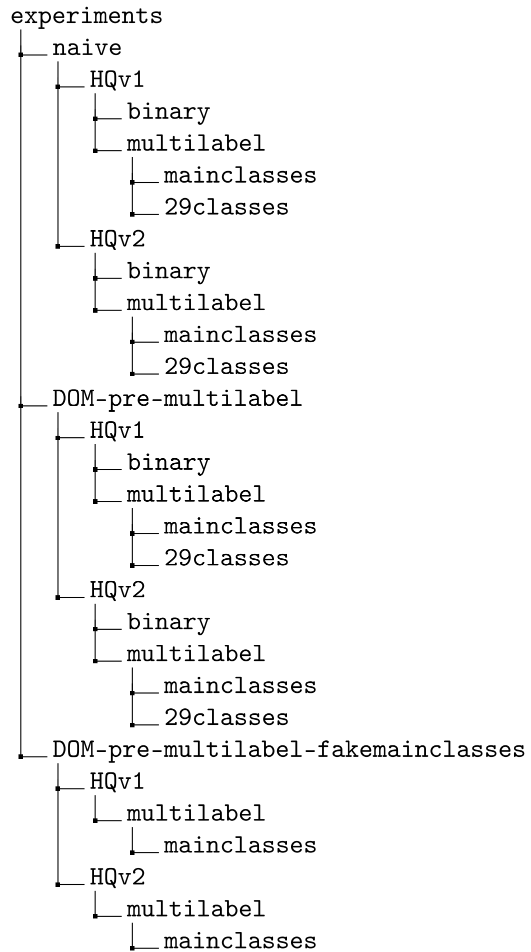
Table C.1: Class Count Overview

D EXPERIMENT PERMUTATIONS

D.1 YOLOV3-SPP DEFAULT PRESET WITHOUT COCO WEIGHTS



D.2 YOLOV3-SPP DEFAULT PRESET WITH COCO WEIGHTS



E EXPERIMENT RESULT OVERVIEW

This Appendix summarizes the results of the experiments introduced in Chapter 5. It additionally includes the $Iteration(Iter)$ at which the best performing model has been selected as well as *HQv2-VAL* results. The results for training on *HQv1* are shown in Table E.1. Table E.2 depicts the results for training on *HQv2*. Table E.3 depicts the weight selection results of selecting the pre-training weights after training on *LQDOM*.

Pre-Training	Cls	Iter	HQv1-VAL				FINAL-TEST			
			APB	APM	APM^{wo}	RC	APB	APM	APM^{wo}	RC
-	BIN	10000	79.90	-	-	83.37	81.70	-	-	81.71
	MAIN	10000	78.82	59.99	75.40	81.21	80.84	51.41	75.57	78.88
	29C	19000	79.88	64.16	76.46	83.17	78.71	60.42	74.54	77.89
partial COCO	BIN	16000	77.84	-	-	81.21	81.22	-	-	80.67
	MAIN	16000	79.05	60.77	76.14	82.00	82.11	58.98	77.45	81.05
	29C	8000	79.13	64.56	75.78	81.64	79.52	61.59	75.32	79.23
partial BIN	BIN	2000	85.99	-	-	86.86	85.74	-	-	84.50
	MAIN	5000	82.20	64.56	79.01	85.17	84.25	58.47	78.48	83.66
	29C	7000	82.39	67.01	79.05	84.6	81.61	57.52	76.91	81.07
full BIN	BIN	1000	**87.60**	-	-	**88.14**	86.67	-	-	**85.79**
partial ML	BIN	2000	86.33	-	-	86.05	**86.93**	-	-	85.34
	MAIN	2000	84.04	66.08	80.79	85.08	83.81	62.12	79.41	82.63
	29C	2000	82.13	66.20	78.4	83.80	83.77	**66.69**	79.12	82.17
***fakemain* ML**	MAIN	1000	84.03	**68.33**	**80.99**	85.48	84.96	66.62	**80.43**	83.41
partial COCO + partial ML	BIN	4000	84.86	-	-	85.77	85.61	-	-	83.94
	MAIN	8000	80.85	66.18	77.75	82.51	81.13	61.45	77.30	79.98
	29C	9000	80.56	63.74	77.57	83.14	82.66	59.82	77.79	80.30
partial COCO + *fakemain* ML	MAIN	1000	82.45	67.00	79.64	84.52	85.03	68.13	80.33	84.11

Table E.1: HQv1 training results

Pre-Training	Cls	Iter	HQv1-VAL				HQv2-VAL				FINAL-TEST			
			APB	APM	APM^{wo}	RC	APB	APM	APM^{wo}	RC	APB	APM	APM^{wo}	RC
-	BIN	11,000	83.68	-	-	85.32	80.27	-	-	82.35	87.10	-	-	84.07
	MAIN	8,000	82.94	66.02	79.93	86.54	79.78	71.18	77.43	83.41	87.94	71.67	83.64	86.48
	29C	8,000	82.27	67.88	79.30	85.3	78.98	70.36	76.76	82.69	86.31	61.39	81.38	84.45
partial COCO	BIN	15,000	83.54	-	-	86.52	79.90	-	-	83.57	86.63	-	-	85.12
	MAIN	8,000	82.94	66.02	79.93	86.54	79.78	71.18	77.43	83.41	87.94	71.67	83.64	86.48
	29C	9,000	82.06	72.59	79.62	86.04	79.51	73.78	77.59	83.66	88.05	76.38	84.45	86.62
partial BIN	BIN	17,000	84.80	-	-	87.26	81.34	-	-	84.42	87.12	-	-	85.89
	MAIN	3,000	84.47	69.40	81.25	87.37	81.09	72.75	78.68	84.46	88.68	65.87	83.90	87.67
	29C	8,000	82.81	69.16	79.99	86.05	79.54	70.10	76.47	82.92	87.17	71.43	82.81	85.42
full BIN	BIN	**1,000**	85.09	-	-	87.37	**82.06**	-	-	**84.62**	88.89	-	-	87.28
partial ML	BIN	9,000	84.73	-	-	87.04	80.96	-	-	84.04	88.40	-	-	85.45
	MAIN	5,000	84.68	72.46	82.23	87.62	80.31	74.06	78.17	83.90	88.43	70.97	84.39	87.24
	29C	6,000	84.33	**72.98**	81.45	86.81	81.02	**75.41**	78.82	84.18	87.52	77.06	84.05	86.40
***fakemain* ML**	MAIN	2,000	**85.55**	70.86	**82.87**	**87.60**	81.82	73.73	**79.64**	84.54	**89.87**	75.73	**86.29**	**87.60**
partial COCO + partial ML	BIN	3,000	84.61	-	-	87.31	80.65	-	-	84.15	89.25	-	-	87.20
	MAIN	4,000	83.08	71.20	80.62	85.50	79.15	70.84	77.12	82.17	87.96	75.21	84.33	85.70
	29C	10,000	83.87	69.46	80.99	86.76	80.44	73.42	78.44	83.78	87.42	71.11	83.40	85.28
partial COCO + *fakemain* ML	MAIN	2000	84.29	69.69	81.16	86.99	80.91	72.99	78.22	83.98	88.97	72.35	84.73	**87.60**

Table E.2: HQv2 training results

Pre-Training	Iter	HQv1-VAL				HQv2-VAL				FINAL-TEST			
		APB	mAP	AP^{wo}	RC	APB	mAP	AP^{wo}	RC	APB	mAP	AP^{wo}	RC
BINARY	33,000	67.06	-	-	64.14	63.40	-	-	60.91	59.76	-	-	56.03
MULTILABEL	18,000	65.61	66.08	67.31	62.23	62.43	70.36	64.17	60.08	57.85	56.44	58.11	57.61
MULTILABEL with COCO weights	25,000	66.15	65.27	67.63	63.74	62.84	69.42	64.43	60.71	59.58	58.74	59.94	57.12

Table E.3: Evaluating LQDOM training using high-quality validation/test sets.

As discussed in Section 4.5, the weights after pre-training on *LQDOM* are selected by evaluating the mAP result on *HQv1-VAL* (Table E.3). As there are many elements for all classes in *LQDOM* (cf. C.4), the weights can be selected using the unweighted mAP as class imbalance is less important.

F IMAGE APPENDIX

This appendix includes several additional example images for visualization. Appendix F.1 includes annotated examples.

F.1 ANNOTATION EXAMPLES

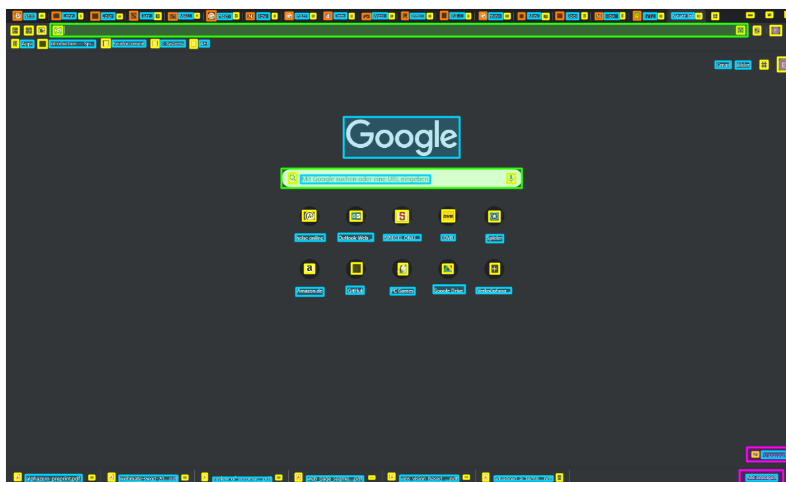


Figure F.1: Sample excerpt of annotated web browser GUI for *FINAL-TEST*. Image Source: [img8]

.

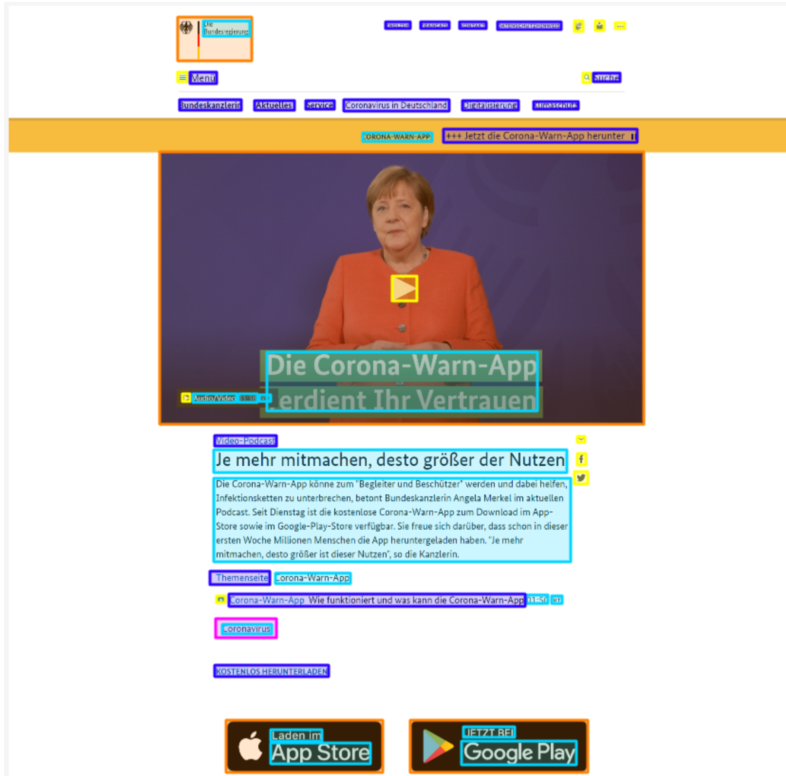


Figure F.2: Sample excerpt of annotated web page. Image Source: [img15]

.

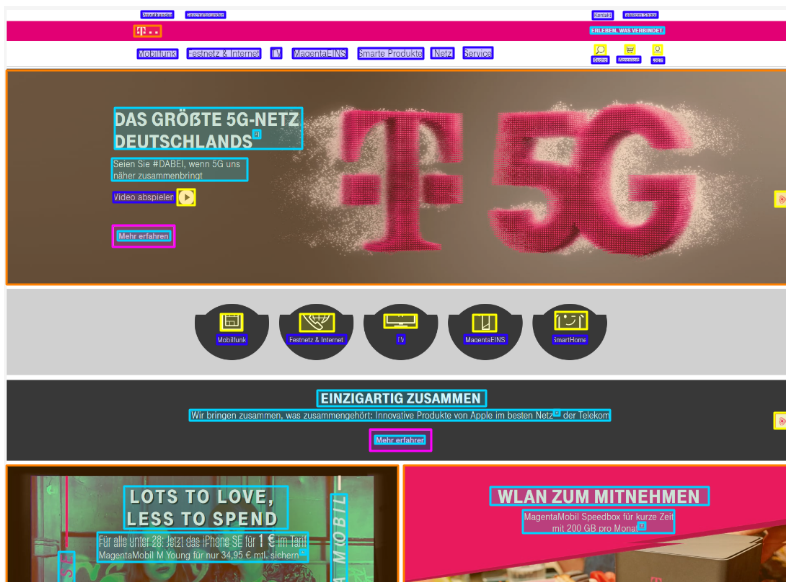


Figure F.3: Sample excerpt of annotated web page. Image Source: [img16]

.

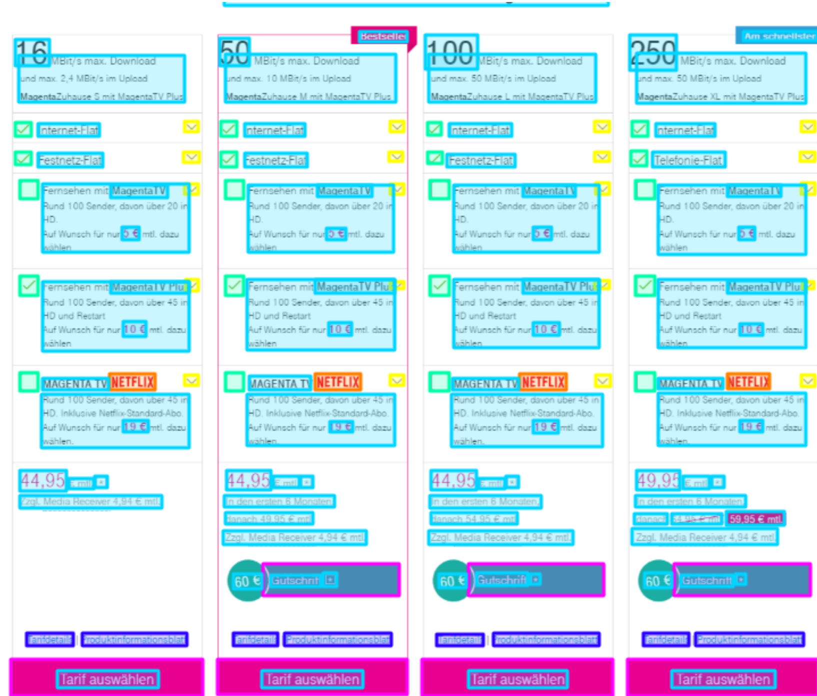


Figure F.4: Sample excerpt of a more complex annotation. Image Source: [img17]

.

F.2 MODEL PREDICTIONS

This section includes several example predictions using the best performing model trained on *HQv2* with *fakemainclass* pre-training. Figures F.5, F.6 and F.7 depict web pages. Figure F.8 includes the web browser itself. Figures F.9 and F.10 depict other desktop applications.

Color codes: Just Interactable (unknown class), Button, Image, Textfield, Dropdown, Icon, Checkbox-Radio, Text.

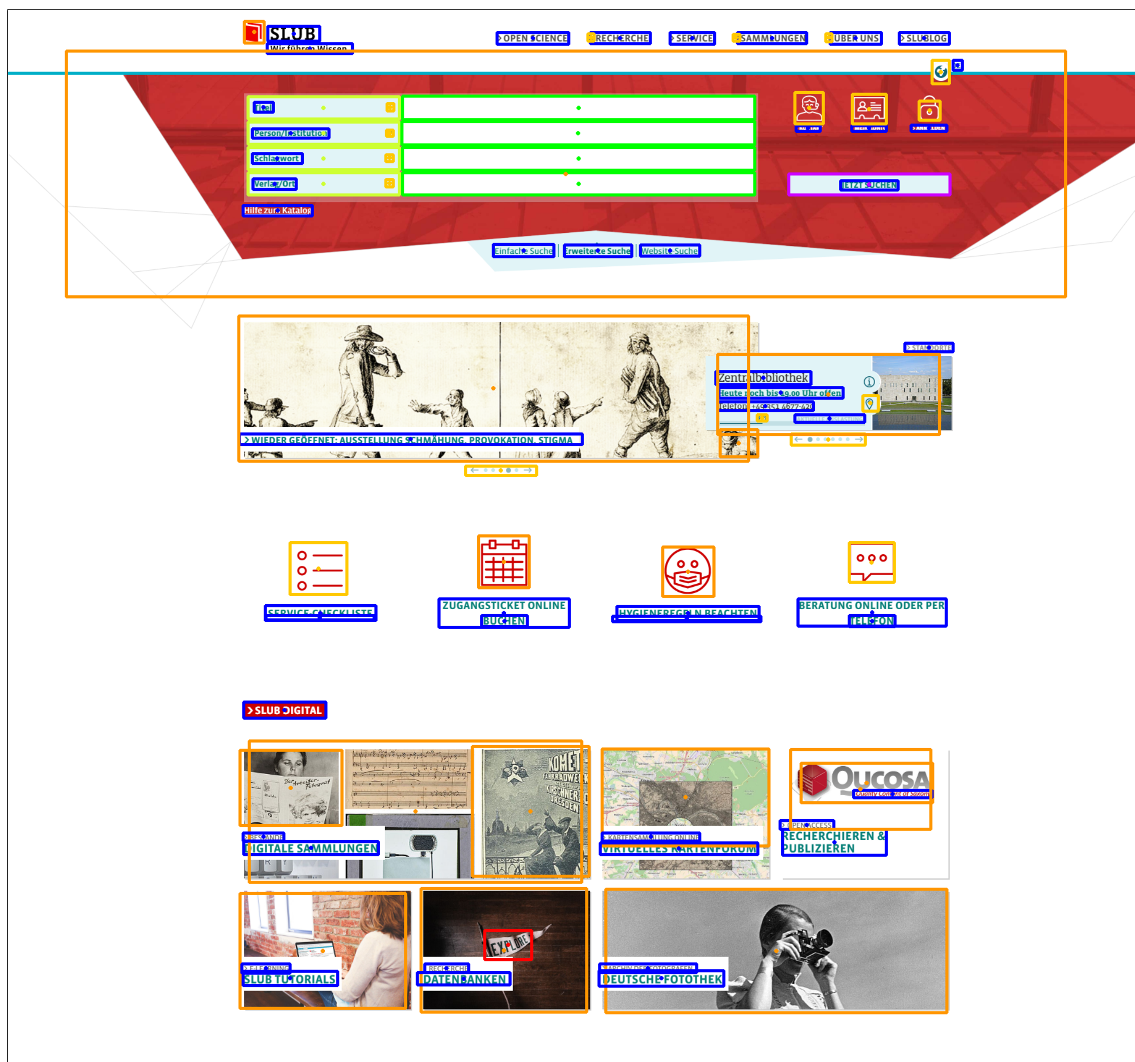


Figure F.5: Model prediction of [img18]

Figure F.6: Model prediction of [img15]

Figure F.7: Model prediction of [img19]

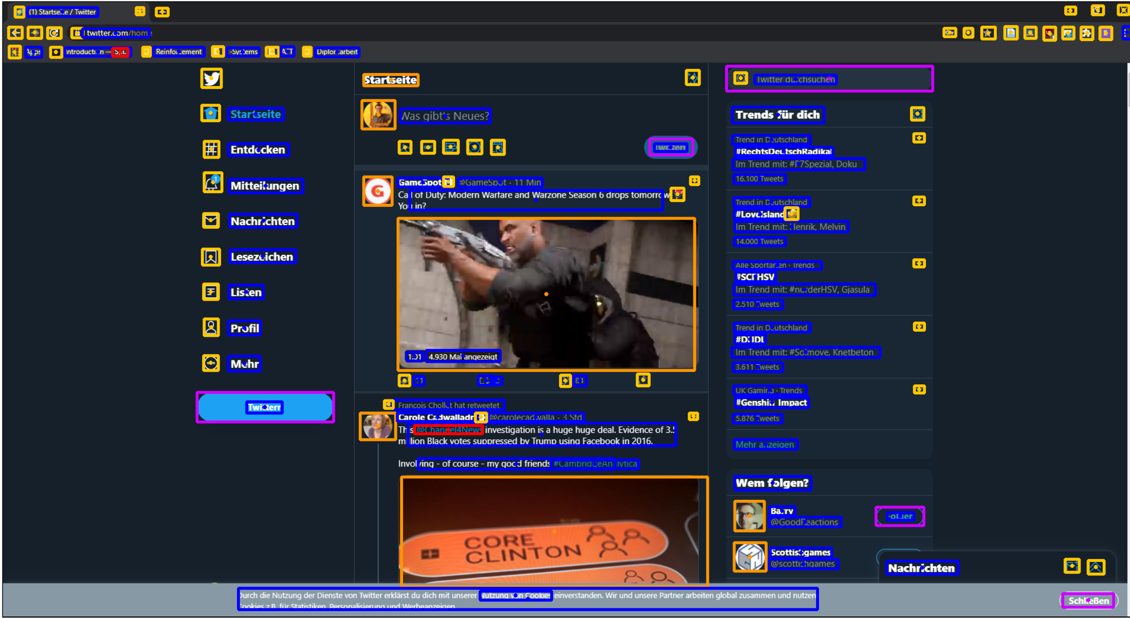


Figure F.8: Model prediction of [img20] including web browser interface

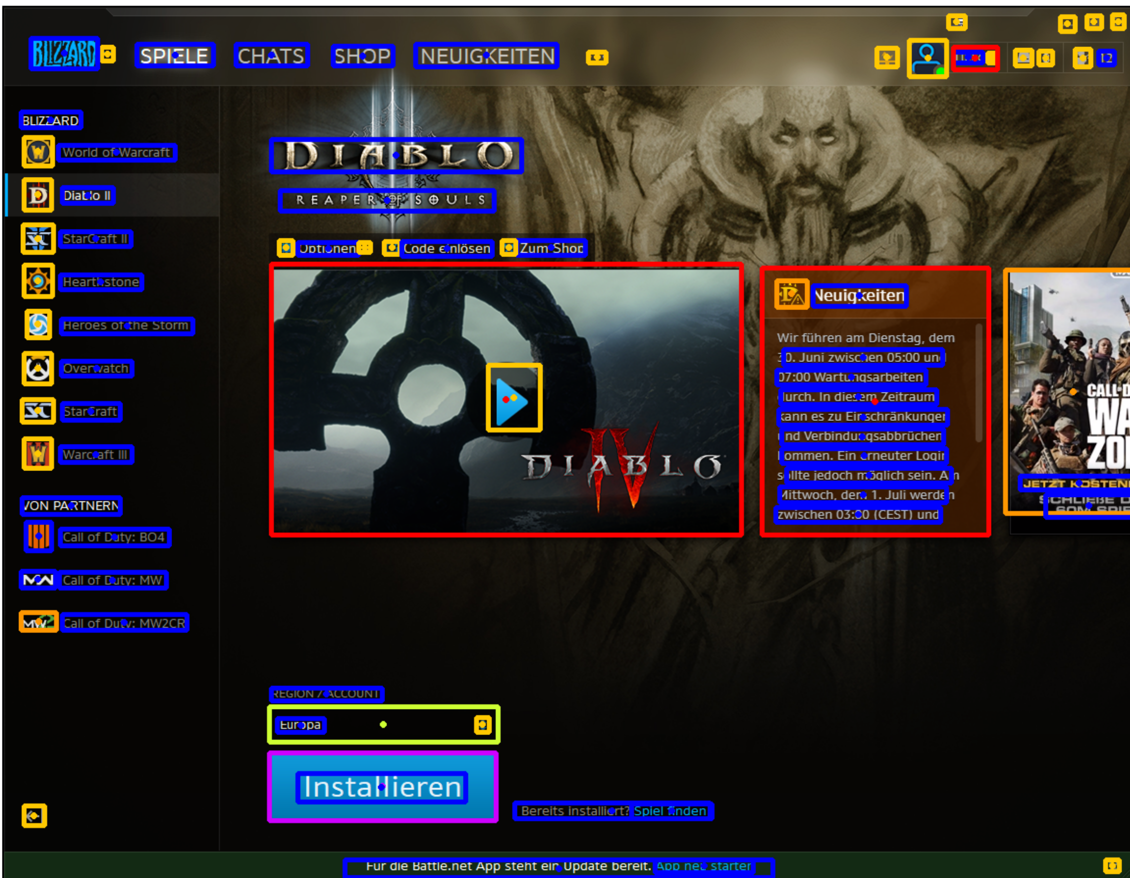


Figure F.9: Model prediction of [img21]

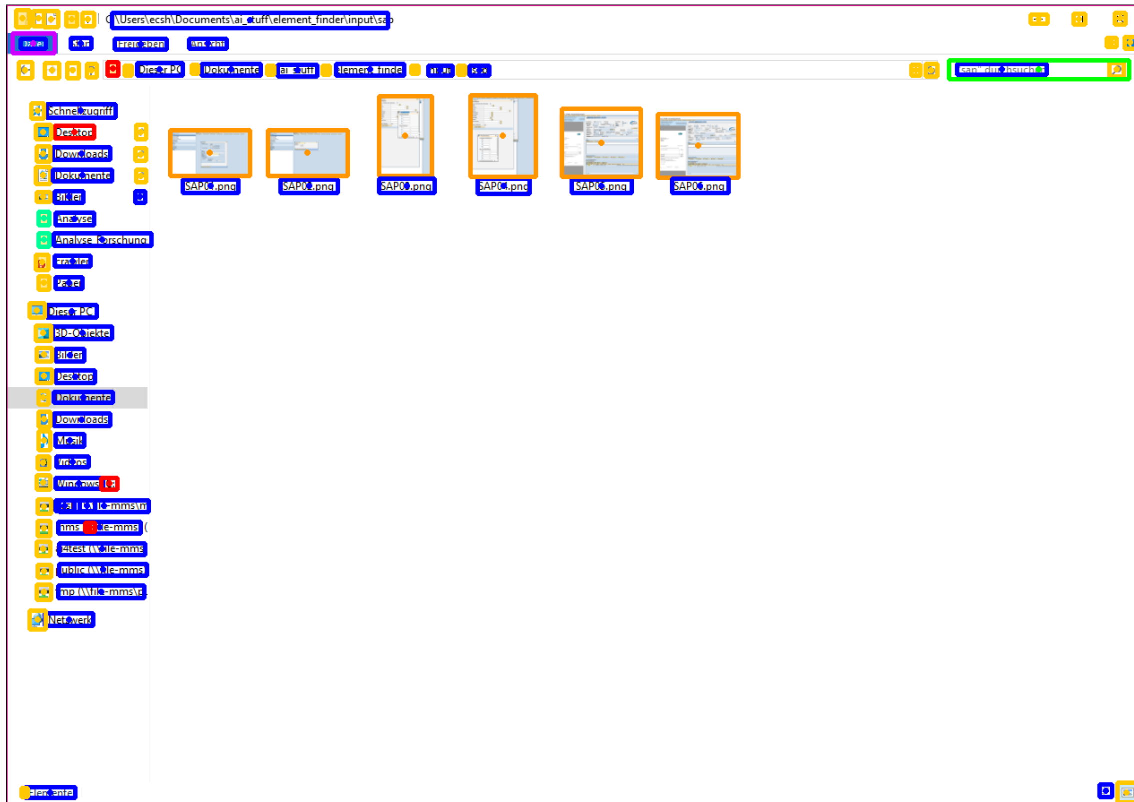


Figure F.10: Model prediction of [img22]

LIST OF FIGURES

2.1 Object Detection Example 14
2.2 Supervised Learning Paradigms 15
2.3 Bounding Boxes IoU 18
2.4 PR-Curve Calculation 20
2.5 AUC of PR-curve 21
2.6 Grid cell-based object detection 23
2.7 Anchor Box predictions 24
2.8 Linear Activations vs Softmax 25
2.9 Spatial Pyramid Pooling 25

3.1 Data and Training Pipeline 28
3.2 DOM analysis edge case 32
3.3 Fake-Button Scamming 35
3.4 Atomic Elements 35
3.5 Minimal Bounding Boxes 36
3.6 Text Blocks 36
3.7 Annotated Web Page Example 36
3.8 Whole-Page Overlay 42

3.9 Downsizing Full-Scale Screenshots 44

6.1 IoU Limitations 71
6.2 Switch Buttons 72
6.3 Mobile GUI differences 73
6.4 Large Textboxes 73
6.5 Text Problems 74
6.6 Hierarchic DOM Structures 75

7.1 WordTree 78
7.2 Pix2Code GUIs 79
7.3 Rico Data Set Class Samples 80

A.1 R-CNN algorithm 87

F.1 Annotated Web Browser GUI 101
F.2 Annotated Web Page 102
F.3 Annotated Web Page 102
F.4 Complex Annotation 103
F.5 Model prediction of [img18] 104
F.6 Model prediction of [img15] 105
F.7 Model prediction of [img19] 106
F.8 Model prediction of [img20] including web browser interface 107
F.9 Model prediction of [img21] 107
F.10 Model prediction of [img22] 108

BIBLIOGRAPHY

[1] T. A. Nguyen and C. Csallner, "Reverse Engineering Mobile Application User Interfaces with REMAUI (T)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Nov. 2015, pp. 248–259.

[2] T. Beltramelli, "Pix2code: Generating Code from a Graphical User Interface Screenshot," *arXiv:1705.07962 [cs]*, Sep. 2017.

[3] V. Jain, P. Agrawal, S. Banga, R. Kapoor, and S. Gulyani, "Sketch2Code: Transformation of Sketches to UI in Real-time Using Deep Neural Network," *arXiv:1910.08930 [cs, eess]*, Oct. 2019.

[4] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," *arXiv:1311.2524 [cs]*, Oct. 2014.

[5] R. Girshick, "Fast R-CNN," *arXiv:1504.08083 [cs]*, Sep. 2015.

[6] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," in *Advances in Neural Information Processing Systems 28*, C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, and R. Garnett, Eds. Curran Associates, Inc., 2015, pp. 91–99.

[7] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," *arXiv:1506.02640 [cs]*, May 2016.

[8] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger," in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Honolulu, HI: IEEE, Jul. 2017, pp. 6517–6525.

[9] J. Redmon and A. Farhadi, "YOLOv3: An Incremental Improvement," *arXiv:1804.02767 [cs]*, Apr. 2018.

[10] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár, "Microsoft COCO: Common Objects in Context," *arXiv:1405.0312 [cs]*, Feb. 2015.

[11] T. D. White, G. Fraser, and G. J. Brown, "Improving random GUI testing with image-based widget detection," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2019. New York, NY, USA: Association for Computing Machinery, Jul. 2019, pp. 307–317.

[12] K. Moran, C. Bernal-Cárdenas, M. Curcio, R. Bonett, and D. Poshyvanyk, "Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps," *arXiv:1802.02312 [cs]*, Jun. 2018.

[13] B. Deka, Z. Huang, C. Franzen, J. Hibschman, D. Afergan, Y. Li, J. Nichols, and R. Kumar, "Rico: A Mobile App Dataset for Building Data-Driven Design Applications," in *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*, ser. UIST '17. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 845–854.

[14] Z. Huang, J. Wang, X. Fu, T. Yu, Y. Guo, and R. Wang, "DC-SPP-YOLO: Dense connection and spatial pyramid pooling based YOLO for object detection," *Information Sciences*, vol. 522, pp. 241–258, Jun. 2020.

[15] Y. LeCun, L. Bottou, G. Orr, and K. Muller, "Efficient BackProp," in *Neural Networks: Tricks of the Trade*, G. Orr and M. K, Eds. Springer, 1998.

[16] A. M. Alqudah, "Brain Tumor Classification Using Deep Learning Technique - A Comparison between Cropped, Uncropped, and Segmented Lesion Images with Different Sizes," *International Journal of Advanced Trends in Computer Science and Engineering*, vol. 8, no. 6, pp. 3684–3691, Dec. 2019.

[17] G. Park and S. Lee, "Environmental Noise Classification Using Convolutional Neural Networks with Input Transform for Hearing Aids," *International Journal of Environmental Research and Public Health*, vol. 17, no. 7, Apr. 2020.

[18] Z.-H. Zhou, "A brief introduction to weakly supervised learning," *National Science Review*, vol. 5, no. 1, pp. 44–53, Jan. 2018.

[19] J. Xu, A. G. Schwing, and R. Urtasun, "Learning to segment under various forms of weak supervision," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. Boston, MA, USA: IEEE, Jun. 2015, pp. 3781–3790.

[20] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?" in *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 3320–3328.

[21] K. Weiss, T. M. Khoshgoftaar, and D. Wang, "A survey of transfer learning," *Journal of Big Data*, vol. 3, no. 1, p. 9, Dec. 2016.

[22] F. Zhuang, Z. Qi, K. Duan, D. Xi, Y. Zhu, H. Zhu, H. Xiong, and Q. He, "A Comprehensive Survey on Transfer Learning," *arXiv:1911.02685 [cs, stat]*, Jun. 2020.

[23] R. Padilla and S. L. Netto, "A Survey on Performance Metrics for Object-Detection Algorithms," p. 6.

[24] K. He, X. Zhang, S. Ren, and J. Sun, "Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition," *arXiv:1406.4729 [cs]*, vol. 8691, pp. 346–361, 2014.

[25] N. Mairittha and S. Inoue, “Gamification for High-Quality Dataset in Mobile Activity Recognition,” in *Mobile Computing, Applications, and Services. 9th International Conference, MobiCASE 2018, Osaka, Japan, February 28 – March 2, 2018, Proceedings*, May 2018.

[26] H. Zhang, M. Cisse, Y. N. Dauphin, and D. Lopez-Paz, “Mixup: Beyond Empirical Risk Minimization,” *arXiv:1710.09412 [cs, stat]*, Apr. 2018.

[27] J. Bergstra and Y. Bengio, “Random Search for Hyper-Parameter Optimization,” p. 25.

[28] J. Zeleny, R. Burget, and J. Zendulka, “Box clustering segmentation: A new method for vision-based web page preprocessing,” *Information Processing & Management*, vol. 53, no. 3, pp. 735–750, May 2017.

[29] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, “YOLOv4: Optimal Speed and Accuracy of Object Detection,” *arXiv:2004.10934 [cs, eess]*, Apr. 2020.

[30] M. Tan, R. Pang, and Q. V. Le, “EfficientDet: Scalable and Efficient Object Detection,” in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Seattle, WA, USA: IEEE, Jun. 2020, pp. 10 778–10 787.

[31] B. Zoph, G. Ghiasi, T.-Y. Lin, Y. Cui, H. Liu, E. D. Cubuk, and Q. V. Le, “Rethinking Pre-training and Self-training,” *arXiv:2006.06882 [cs, stat]*, Jun. 2020.

[32] C.-A. Brust, C. Käding, and J. Denzler, “Active Learning for Deep Object Detection,” *arXiv:1809.09875 [cs]*, Sep. 2018.

[33] D.-H. Lee, “Pseudo-Label : The Simple and Efficient Semi-Supervised Learning Method for Deep Neural Networks,” *ICML 2013 Workshop : Challenges in Representation Learning (WREPL)*, Jul. 2013.

[34] J. Chen, M. Xie, Z. Xing, C. Chen, X. Xu, L. Zhu, and G. Li, “Object Detection for Graphical User Interface: Old Fashioned or Deep Learning or a Combination?” *arXiv:2008.05132 [cs]*, Sep. 2020.

[35] X. Zhou, C. Yao, H. Wen, Y. Wang, S. Zhou, W. He, and J. Liang, “EAST: An Efficient and Accurate Scene Text Detector,” *arXiv:1704.03155 [cs]*, Jul. 2017.

URL AND TOOL REFERENCES

[u1] V. Valkov. Color palette extraction with k-means clustering. [Online]. Available: https://www.curiousily.com/posts/color-palette-extraction-with-k-means-clustering/

[u2] Weak supervision: A new programming paradigm for machine learning. [Online]. Available: http://ai.stanford.edu/blog/weak-supervision/ (Accessed 2020-09-13).

[u3] COCO - common objects in context. [Online]. Available: http://cocodataset.org/#home (Accessed 2020-06-03).

[u4] The PASCAL visual object classes challenge 2012 (VOC2012). [Online]. Available: http://host.robots.ox.ac.uk/pascal/VOC/voc2012/ (Accessed 2020-07-02).

[u5] HTML5. [Online]. Available: https://dev.w3.org/html5/spec-LC/Overview.html (Accessed 2020-07-06).

[u6] Google docs. [Online]. Available: https://docs.google.com/document/u/0/ (Accessed 2020-07-06).

[u7] Overleaf. [Online]. Available: https://www.overleaf.com/ (Accessed 2020-07-06).

[u8] Discord. [Online]. Available: https://discord.com/new (Accessed 2020-07-06).

[u9] Slack. [Online]. Available: https://slack.com/intl/ (Accessed 2020-07-06).

[u10] Electron | plattformübergreifende desktop-anwendungen mit JavaScript, HTML und CSS entwickeln. [Online]. Available: https://www.electronjs.org/ (Accessed 2020-07-06).

[u11] Visual studio code. [Online]. Available: https://github.com/microsoft/vscode (Accessed 2020-07-06).

[u12] Material design. [Online]. Available: https://developer.android.com/design (Accessed 2020-07-06).

[u13] Progressive web apps. [Online]. Available: https://codelabs.developers.google.com/codelabs/your-first-pwapp/#0 (Accessed 2020-07-06).

[u14] Flutter - beautiful native apps in record time. [Online]. Available: https://flutter.dev/ (Accessed 2020-07-06).

[u15] Document object model living standard, url = https://dom.spec.whatwg.org/, urldate=2020-09-25, author = World Wide Web Consortium (W3C).

[u16] W. Data. Javascript html dom. [Online]. Available: https://www.w3schools.com/js/js_htmldom.asp (Accessed 2020-09-17).

[u17] Document object model (dom). [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model (Accessed 2020-09-25).

[u18] Selenium. [Online]. Available: https://www.selenium.dev/ (Accessed 2020-07-06).

[u19] Google chrome. [Online]. Available: https://www.google.com/chrome/ (Accessed 2020-09-15).

[u20] Mozilla firefox. [Online]. Available: https://www.mozilla.org/firefox/ (Accessed 2020-09-15).

[u21] A. Kujawa, "Pick a download, any download," 2016. [Online]. Available: https://blog.malwarebytes.com/cybercrime/2012/10/pick-a-download-any-download/ (Accessed 2020-08-08).

[u22] ICON | definition in cambridge english dictionary. [Online]. Available: https://dictionary.cambridge.org/de/worterbuch/englisch/icon (Accessed 2020-06-22).

[u23] Google. reCAPTCHA. [Online]. Available: https://www.google.com/recaptcha (Accessed 2020-07-01).

[u24] General data protection regulation (GDPR) compliance guidelines. [Online]. Available: https://gdpr.eu/ (Accessed 2020-07-01).

[u25] D. Kladnik. I don't care about cookies. [Online]. Available: https://www.i-dont-care-about-cookies.eu/ (Accessed 2020-07-01).

[u26] "tesseract-ocr/tesseract." [Online]. Available: https://github.com/tesseract-ocr/tesseract (Accessed 2020-07-02).

[u27] Yolov3-spp configuration. [Online]. Available: https://github.com/AlexeyAB/darknet/blob/b8e6e80c6d411d05a9e09f1e3676eb9a7f3ea0e8/cfg/yolov3-spp.cfg (Accessed 2020-07-06).

[u28] Alexey, "AlexeyAB/darknet," original-date: 2016-12-02T11:14:00Z. [Online]. Available: https://github.com/AlexeyAB/darknet (Accessed 2020-06-16).

[u29] J. Redmon, *Darknet: Open Source Neural Networks in C*. [Online]. Available: http://pjreddie.com/darknet/

[u30] A. Bochkovskiy. Darknet cfg parameters in the [net] section. [Online]. Available: https://github.com/AlexeyAB/darknet/wiki/CFG-Parameters-in-the-%5Bnet%5D-section (Accessed 2020-09-25).

[u31] A. Bochkovskiy. Darknet cfg parameters in the different layers. [Online]. Available: https://github.com/AlexeyAB/darknet/wiki/CFG-Parameters-in-the-different-layers (Accessed 2020-09-25).

[u32] A. Bochkovskiy. Beta: Using cpu-ram instead of gpu-vram for large minibatch=32 - 128.

[u33] Darknet partial command. [Online]. Available: https://github.com/AlexeyAB/darknet/issues/6138#issuecomment-654128384 (Accessed 2020-09-17).

[u34] Wikipedia. [Online]. Available: https://www.wikipedia.org/ (Accessed 2020-09-20).

[u35] Microsoft edge devtools 3d view (early prototype). [Online]. Available: https://github.com/MicrosoftEdge/DevToolsSamples/tree/master/3DView (Accessed 2020-09-2020).

[u36] Papers with code: Real-time object detection. [Online]. Available: https://paperswithcode.com/task/real-time-object-detection (Accessed 2020-09-26).

[u37] Jsoup java html parser. [Online]. Available: https://jsoup.org/ (Accessed 2020-09-18).

IMAGE REFERENCES

[img1] W. Commons, "File:intersection over union - visual equation.png — wikimedia commons, the free media repository," 2017. [Online]. Available: https://commons.wikimedia.org/w/index.php?title=File:Intersection_over_Union_-_visual_equation.png&oldid=239782811 (Accessed 2020-08-08).

[img2] Wikimedia Commons: Spider.svg. [Online]. Available: https://commons.wikimedia.org/wiki/File:Spider.svg (Accessed 2020-09-27).

[img3] Wikimedia Commons: DOM-model.svg. [Online]. Available: https://commons.wikimedia.org/wiki/File:DOM-model.svg (Accessed 2020-09-27).

[img4] Wikimedia Commons: Firefox Logo 2019. [Online]. Available: https://commons.wikimedia.org/wiki/File:Firefox_logo,_2019.svg (Accessed 2020-09-27).

[img5] YOLO logo. [Online]. Available: https://pjreddie.com/darknet/yolo/ (Accessed 2020-09-27).

[img6] "Screenshot of apple appstore (excerpt)." [Online]. Available: https://apps.apple.com/us/app/loot-crate/id1207390925 (Accessed 2019-08-07).

[img7] "Screenshot of yelp search." [Online]. Available: https://www.yelp.de/dresden (Accessed 2019-08-07).

[img8] Annotated Google Chrome UI, [Screenshot taken 18-June-2020]. [Online]. Available: https://www.google.com/chrome/ (Accessed 2020-06-18).

[img9] "Screenshot of tinnitus.org.uk." [Online]. Available: https://www.tinnitus.org.uk/ (Accessed 2019-08-07).

[img10] Screenshot of TU Dresden webpage (excerpt, annotated). [Online]. Available: https://tu-dresden.de/ing/informatik (Accessed 2020-06-18).

[img11] Screenshot of Yahoo privacy overlay. [Online]. Available: https://yahoo.com/ (Accessed 2020-09-08).

[img12] Screenshot of Google Start Page. [Online]. Available: https://www.google.de/ (Accessed 2020-09-02).

[img13] Prediction on Wikipedia Article about AI, [Screenshot taken 20-September-2020]. [Online]. Available: https://de.wikipedia.org/wiki/K%C3%BCnstliche_Intelligenz (Accessed 2020-09-20).

[img14] "Tu dresden web page." [Online]. Available: https://tu-dresden.de/ (Accessed 2020-09-18).

[img15] Screenshot of the Web Page of the Federal Government of Germany. [Online]. Available: https://www.bundesregierung.de/ (Accessed 2020-06-23).

[img16] Annotated web page of Telekom. [Online]. Available: https://www.telekom.de/ (Accessed 2020-06-23).

[img17] Annotated web page of Telekom offer. [Online]. Available: https://www.telekom.de/festnetz (Accessed 2020-06-23).

[img18] "Screenshot of slub dresden web page." [Online]. Available: https://www.slub-dresden.de/startseite/ (Accessed 2020-06-30).

[img19] "Screenshot of slub dresden web page." [Online]. Available: https://www.chemnitz.de/ (Accessed 2020-06-30).

[img20] "Screenshot of twitter including google chrome interface." [Online]. Available: https://www.twitter.com/ (Accessed 2020-09-28).

[img21] "Screenshot of blizzard battle.net launcher." [Online]. Available: https://www.blizzard.com/apps/battle.net/desktop (Accessed 2020-06-29).

[img22] "Screenshot of microsoft windows 10 explorer." [Online]. Available: https://www.microsoft.com/windows/ (Accessed 2020-06-29).

This section only covers external image sources. Images citations from publications or referenced online sources refer to the corresponding sources instead.