Detection and Segmentation of Interactive Elements in Mobile Applications using Deep Learning Trained on an Automatically Generated and Labeled Dataset

Bachelor's Thesis

of

Benjamin Frank Meyjohann

Degree Course: Industrial Engineering and Management B.S.

Matriculation Number: 2228939

Institute of Applied Informatics and Formal Description Methods (AIFB)

KIT Department of Economics and Management

Reviewer:	Prof. Dr. Andreas Oberweis
Second Reviewer:	Prof. Dr. J. Marius Zöllner
Supervisor:	M. Sc. Demian Frister
Submitted:	9. December 2022

Abstract

Testing of graphical user interfaces of mobile applications is important to ensure their functionality and usability. However, manual testing is time-consuming and expensive. This could be solved by automating the testing process. Previous frameworks for automation rely on random actions or pattern recognition. Both types are not efficient, because the former is missing a structured approach at testing and the latter must often be revised if the graphical user interface is modified. In this thesis, we utilize deep learning to make recognition robust to changes in the graphical user interface. For training, we use an automatically generated and labeled dataset. This allows for fast and accurate labeling resulting in improved data quality compared to manual labeling. Additionally, recognition is performed on images of the mobile application captured by camera and not on screenshots. That way, testing can be performed end-to-end. The results are hard to compare to previous works because of the different ways of capturing the mobile application. While the results look promising and our approach recognizes many elements of graphical user interfaces, the benefit for automated testing is limited because many complex elements are not yet recognized well enough.

Table of Contents

Table of Contents ..... **I**
List of Abbreviations ..... **III**
List of Figures ..... **V**
List of Tables ..... **X**
1. Introduction **1**
1.1. Motivation 1
1.2. Objective 2
1.3. Structure 3
2. Theoretical Foundations **4**
2.1. Mobile Applications 4
2.2. Material Design 5
2.3. Software Testing 6
2.4. Computer Vision 8
2.4.1. Image Recognition 8
2.4.2. Selective Search 10
2.5. Machine Learning 10
2.5.1. Support Vector Machine 14
2.6. Deep Learning 16
2.6.1. Multilayer Perceptrons 16
2.6.2. Optimization 19
2.6.3. Convolutional Neural Networks 25
2.6.4. Metrics 29
2.7. Deep Neural Networks for Object Detection and Instance Segmentation 31
2.7.1. R-CNN Family 32
2.7.2. Mask R-CNN 37
2.7.3. YOLO Family 39
3. Related Work **45**
3.1. “Construction of GUI Elements Recognition Model for AI Testing based on Deep Learning” 45
3.2. “Detection and Segmentation of Graphical Elements on GUIs for Mobile Apps Based on Deep Learning” 46
3.3. “Object detection for graphical user interface: old fashioned or deep learning or a combination?” 46
4. Methodology **48**
5. Implementation **51**
5.1. Android Application 52

5.2. Dataset Collection Scripts using Python 60
5.3. Mask R-CNN 62
5.4. Workflow for Dataset Collection and Training 66
5.5. Problems 69

6. Evaluation 72
7. Summary 76
7.1. Limitations 76
7.2. Outlook 77

8. Appendices 79
References 81
Assertion 87

List of Abbreviations

AIFB	Institute of Applied Informatics and Formal Description Methods
AP	Average Precision
CNN	Convolutional Neural Network
COCO	Common Objects in Context
CPU	Central Processing Unit
CV	Computer Vision
DNN	Deep Neural Network
FN	False Negative
FP	False Positive
FPN	Feature Pyramid Network
FPS	Frames per Second
GPU	Graphical Processing Unit
GUI	Graphical User Interface
IDE	Integrated Development Environment
IE	Interactive Element
IoU	Intersect over Union
KIT	Karlsruhe Institute of Technology
mAP	Mean Average Precision
ML	Machine Learning
MLP	Multilayer Perceptron
NN	Neural Network
OS	Operating System

PCA	Principal Component Analysis
R-CNN	Regions with Convolutional Neural Network Features
RoI	Region of Interest
RPN	Region Proposal Network
SGD	Stochastic Gradient Descent
SVM	Support Vector Machine
TN	True Negative
TP	True Positive
UI	User Interface
YOLO	You Only Look Once

List of Figures

Figure 1. Structure of thesis visualized as mind map. Starting in the upper left, chapters and sections are arranged clockwise. 3

Figure 2. Examples for components provided by Material Design. From left to right, top to bottom: Top app bar, button, floating action button, text fields, bottom navigation, checkboxes, switches, radio buttons and sliders. 5

Figure 3. Different computer vision tasks applied to an image: Classification assigns a class to the entire image (upper left). Object detection outputs bounding-boxes for every object in the image as well as a corresponding class (lower left). Semantic segmentation assigns every pixel belonging to a certain class to a mask for that class (upper right. Instance segmentation outputs a binary mask and class for every object found in the image (lower right). 9

Figure 4. Segmentation masks at various stages in the hierarchical grouping process starting with many small regions and ending in few big regions. Underneath, the corresponding images with bounding-box proposals at each stage. Bounding-boxes of actual objects are highlighted. (Uijlings et al. 2013) 10

Figure 5. Three graphs with a line fitted to training data. Green points are part of the training dataset and blue points part of the test dataset. In the left graph, the line is overfitting. It describes the green points very well but not the blue ones. In the graph in the middle, the line is overfitting. It describes neither green nor blue points appropriately. In the right graph, the line describes both green and blue points decently indicating good generalization..... 12

Figure 6. Supervised learning takes training inputs and training labels as input. It outputs a mode with learned parameters, that can be used to predict outputs for new, to the model unknown inputs. 13

Figure 7. Example of regression: Fitting line to dataset (left), and example of classification: Line dividing dataset into two distinct classes (right). 13

Figure 8. Interaction of agent and environment through observation, action, and reward. Based on Zhang et al. (2021, p. 31)..... 14

Figure 9. SVG with hard margin separating linearly separable data into two distinct classes (left) and SVM with soft margin for data that is not linearly separable (right). 15

Figure 10. Multiclass SVM approaches: One-versus-all separating each one class from all other classes (left) and one-versus-one separating pairs of classes one by one (right). 15

Figure 11. Internals of an artificial neuron: Input is weighted and summed up with a bias. After applying an activation function, the output is obtained. 17

Figure 12. Widely used activation functions: Rectified linear unit (ReLU) is a linear function for positive values and equal to zero for negative values. Leaky rectified linear unit (Leaky ReLU) is the same for positive values but uses a fraction of the value for negative values. Hyperbolic tangent (tanh)

maps values to a range of negative and positive one. The sigmoid function is similar but maps to values between zero and one. 18
Figure 13. Basic example of an MLP with two hidden layers as graph with neurons as vertices/nodes and connections as edges. Every input in the input layer is connected to every neuron of the first hidden layer, each of which is in turn connected to every neuron of the second hidden layer. All those neurons are interconnected with each neuron of the output layer resulting in the final output of the MLP. 19
Figure 14. An example of an MLP trained on images of digits. First, forward propagation is performed. Then, actual outputs are compared with desired outputs and parameters (weights) are adjusted to get closer to the desired result. 20
Figure 15. Graph showing gradient descent (red line) for a function with two inputs using an entire batch. Black lines indicate points where output level of the function is the same. Therefore, the shown example ca be thought of like an elongated valley. Because of this, gradients (black arrows) do not point to the center of the valley. This results in alternating gradients after every step of gradient descent making optimization take much longer compared to the shortest path downhill. 22
Figure 16. Three two dimensional plots with datapoints of two classes (marked in yellow and blue), where blue points are in the center and encircled by a ring of yellow points. The background is colored according to what class a created MLP would assign to a point at that position. Shades in between yellow and blue indicate uncertainty. In the left plot, background has no clear colors. In the middle, a blue polygon is surrounded by yellow background. In the right plot, a blue oval is encircled by yellow background. 24
Figure 17. Shown above is an example of two positions of the sliding window multiplied with the kernel resulting in a single output value per position. 26
Figure 18. Each cell represents the learned weights of a kernel for an image with three color channels. The various kernels picked up specific features of the input like edges, textures, or color combinations. 26
Figure 19. Max pooling with a window size of 2 x 2 applied to a sample input. The maximum of all values in the window is computed. Then, the sliding window is moved to the next position and the process is repeated. 27
Figure 20. Example of simple CNN architecture with one convolutional layer, one pooling layer and two fully connected layers. (O'Shea and Nash 2015, p. 4) 28
Figure 21. Example of matrix of possible combinations with TP, FP, TN and FN. (Szeliski 2022, p. 442) 29
Figure 22. Schematic example of IoU (a) and example using a real image (b). (Szeliski 2022, p. 380) ... 30
Figure 23. Precision-recall-curve for a single class and IoU threshold. For different confidence scores, precision and recall are calculated and points are plotted and connected. (Szeliski 2022, p. 381) 31

Figure 24. Original R-CNN architecture without bounding-box regression: First stage: Region proposal extraction, second stage: Convolutional feature computation and third stage: Classification of region proposals (left). 32
Figure 25. Original version (left) and modified version (right) of figure from Bharati and Pramanik (2020, p. 660) in comparison. They both visualized the architecture of R-CNN with a focus on feature extraction, classification via SVMs and bounding-box regression for each region proposal. The original shows arrows from feature extraction to classification and bounding-box regression while the modified version has an additional arrow pointing from classification to bounding-box regression. 33
Figure 26. Fast R-CNN architecture: 1. CNN and RoI projection on convolutional feature map. 2. For each RoI: RoI pooling layer, fully connected layers, split into branches for classification using fully connected layer and softmax activation, and bounding-box regression. 34
Figure 27. Simplified illustration of working principle of RoI pooling layer: Division of input region with size of 4 × 8 into 2 × 2 sub-regions with size of 2 × 4 and selection of the highest value as output value. 35
Figure 28. RPN utilizing convolutional feature map for region proposals. Thus, flow of information from the feature map onwards is split going into the RPN and the RoI pooling layer. Proposals generated by the RPN are also fed into the RoI pooling layer. 36
Figure 29. Window sliding over convolutional feature map producing offsets (left, right, top and bottom) and class scores (object or no object) based on anchor boxes as base areas. (Ren et al. 2017)... 37
Figure 30. Mask R-CNN architecture divided in Faster R-CNN with RoI pooling layer replaced by RoIAlign layer, and additional mask branch. 38
Figure 31. Detection pipeline of YOLOv1: 1. Division of input into grid, 2. For each cell: Simultaneous class prediction and bounding-box prediction, 3. Merging of both predictions into final detection results. 40
Figure 32. Cutout of hierarchical tree structure assigning low-level classes to higher-level classes. (Redmon and Farhadi 2017, p. 6524) 42
Figure 33. Comparison of version five, six and seven of YOLO and additional model architectures. The graph on the left plots AP for the COCO dataset as y-axis and latency as x-axis. The graph on the right uses the same y-axis but FPS as x-axis. In summary, YOLOv7 outperforms other model architectures for some cases. Except for YOLOv6 which constantly outperforms all other model architectures including YOLOv7. 44
Figure 34. Example of results using YOLOv3 (left) and Mask R-CNN (right). (C. Zhang et al. 2021)... 45
Figure 35. Four examples of non-text GUI detection using the new approach suggested by the authors of the paper. Edges of bounding-boxes accurately match edges of elements. Some elements are merged together while others are split into multiple bounding-boxes. 47

Figure 36. Individual components and their interaction with each other. A mobile application provides random GUIs and information about the interactive elements for labeling. The GUIs are captured by a script that also collects the information about elements. From this, the script creates a dataset. This can be used for training and evaluation of a neural network. 48
Figure 37. Outline of communications between components in order of execution. The dataset collection script serves as client accessing the mobile application acting as server. The script orchestrates collection of each datapoint and saves all information in a directory. For a single datapoint, it starts by requesting a random GUI. Then, it asks for information about classes of elements. For every element in the GUI, it requests, captures, and saved the mask. Afterwards, the DNN uses this directory as dataset for training. 52
Figure 38. Most relevant classes and files of the android application for randomized GUIs: MainActivity, State, Registry, Utils, Connection, Random Elements, and Probabilities. 53
Figure 39. Contents of file MainActivity: Class MainActivity and composable function RandomApp. 53
Figure 40. Content of class State. Shows variables(top) and methods(bottom). 55
Figure 41. Variables and methods of class Registry 56
Figure 42. All composable functions of file RandomElements. 56
Figure 43. Variables and methods of class Connection. 57
Figure 44. Possible requests and their responses with header and body. 58
Figure 45. Functions and methods of file Utils. 59
Figure 46. Three examples of GUIs generated using the application described above. 60
Figure 47. Variables and methods of class Dataset. 62
Figure 48. Methods of class IEDataset. 65
Figure 49. First window shows image of mobile device with green screen and sliders for thresholds. Second window shows the resulting binary mask after selecting the screen color by clicking it and adjusting the sliders. 66
Figure 50. Window before and after cropping. Most pixels that are not part of the screen were removed. 67
Figure 51. Window with threshold chosen based on the mask displayed (left). The other images show the processing applied to the raw image to obtain a binary mask. From left to right, the following steps are performed: Painting everything black except for the screen. Applying threshold to generate a binary output which is then inverted and shown in the image on the right. 68
Figure 52. Each row represents one datapoint. The first image in a row shows the GUI captured by camera, which is the input image for training. The following images in each row display masks of all objects of the four classes appearing most often in the GUI. Each mask instance is colored in a different shade. 68
Figure 53. The two images show GUIs captured overlayed with all their corresponding masks. In the first image, all masks appear to be scaled down, which makes them smaller and move towards the center of the image. In the second image, the opposite seems to be the case. Because each

individual mask is taken one after another, this implies that focus changed only once after taking the GUI image and before taking all mask images. 70
Figure 54. Plots of loss function for training dataset (top) and test dataset (bottom) for four runs. Marked in blue is a first test run. Yellow and golden are part one and two of the first full run with a bug. Grey is a run with bug using only one common class for detection. Red shows the final run after fixing the bug, which performs visibly better than the other runs. All runs start with fast improvement that later slows down and eventually plateaus. 72
Figure 55. Visual examples of detection results for the test dataset. 74
Figure 56. Top row shows screenshots of GUIs. Bottom row shows the corresponding detection results. 75

List of Tables

Table 1. Testing types and corresponding information about code opacity, person conducting the test, and scope of test. Based on Nidhra (2012, pp. 30–31). 7

Table 2. Common Convolutional Neural Network Architectures. Based on (Bharati and Pramanik 2020, pp. 659–660). 28

Table 3. Differences between implementation by Abdulla and official paper by He et al. according to Abdulla (2017) 63

1. Introduction

For both physical products as well as software, testing is an important part of the development process since it has the potential to ensure and improve usability, functionality, and possibly even safety of a product. All of which can lead to more customers and thus additional revenue.

However, software development differs from the development of physical goods. While the latter results in one final product, development of software usually happens iteratively, and software updates are published repeatedly. Hence, all testing of software is generally not only done once but must be repeated continuously while adapting to the changes that have been made. This makes manual testing a task that is time-consuming and inefficient.

Since smartphones are very popular nowadays (Statista 2022a), there exist many mobile applications and a lot of people use them for hours on a daily basis (Statista 2022b). Therefore, testing of mobile applications can improve satisfaction for many customers (Kong et al. 2021). Interaction between the human and a mobile application mainly relies on graphical user interfaces (GUIs). Because of this, testing of GUIs is an important part of the testing process.

Apart from this, the area of artificial intelligence (AI) has seen huge improvements over the last years. Nowadays, AI is capable of performing various visual tasks (Szeliski 2022, p. 237). This raises the question if testing of GUIs could benefit from the employment of artificial intelligence.

1.1. Motivation

Because of the shortfalls of manual testing, automated testing has the potential of decreasing the human effort and costs associated with testing (Nass et al. 2021) while increasing test coverage and reusability (Rafi et al. 2012). Some existing frameworks for automated testing of GUIs rely on random emulated actions, which is inefficient because many actions do not have any effect and because they are missing a structured approach at testing the application. Other frameworks use pattern recognition (Yeh et al. 2009) or access to the application's source code to identify GUI elements. However, according to Coppola et al. (2016), both ways of testing GUI elements in mobile applications are fragile. Changes in the GUI or different devices require most automated tests to be revised defeating the purpose of automation.

Recognition of GUIs independently of changes in the application could thus prove to be a crucial building block in achieving efficient automated testing. With the discovery of robust deep learning models for computer vision (CV), it becomes possible to reliably detect GUIs of mobile applications based solely on what is displayed on the screen. This makes adaptation to changes easier.

Previous work on this topic uses screenshots and mainly relies on object detection to achieve good accuracy when evaluated on its own dataset. Nonetheless, human error is introduced because of manual labeling of

datasets used for training. The fact that manual labeling is a time-consuming and repetitive task may amplify inaccuracies of bounding-boxes in the dataset. Since neural networks can only learn from information provided by the dataset, this inaccuracy is carried on to detection after training. As additional result, it is questionable in how far metrics derived from a manually labeled dataset are trustworthy for evaluation because data used for validation suffers from the same problem. Furthermore, object detection creates rectangular bounding-boxes as output. It cannot accurately describe the shape of interactive elements that do not consist of a rectangle or are captured at an angle.

When using screenshots for detection and emulated events for user actions, a part of the chain between human and mobile application is left out during testing. Everything happening between screen and human cannot be accurately reproduced. E.g., a human might be affected by Parkinson's disease resulting in imprecise actions or impaired vision causing difficulties in recognition of elements and texts below a certain size. Also, environmental conditions can impact usability (e.g., a wet screen). All of this could be included into testing by conducting tests end-to-end via a camera performing detection and a robot performing actions on a real device by using its touchscreen. To automate this, reinforcement learning could be used. This thesis investigates the task of detection.

1.2. Objective

The goal of this thesis is to determine whether a neural network (NN) trained on an automatically generated and labeled dataset offers advantages compared to previous approaches of detecting interactive elements in mobile applications.

Discussed issues with accuracy are addressed by utilizing instance segmentation, which creates masks instead of rectangular bounding-boxes. To obtain accurate labeling of data, information about position, size and type of interactive elements are taken directly from a mobile application. Since this information is usually not accessible in existing applications (source code is not available), a mobile application providing random GUIs with interactive elements and labeling data is programmed. This provides reliable labeling and fast generation of data once setup is finished. To include external influences, detection is performed on images captured by a camera and not on screenshots.

Ideally, the network provides a confidence score for each interactive element and distinguishes between different interactive elements. The possible ways of interaction can then be deducted based on the element type. E.g., tipping or swiping.

The neural network is mainly supposed to offer high accuracy while still preserving a sufficient speed for testing of mobile applications.

The neural network is evaluated on a part of the automatically generated and labeled dataset in addition to GUIs of real-world mobile applications. Its accuracy is compared to that of previous work with similar objectives.

1.3. Structure

So far, the topic of this thesis has been introduced and motivation and objective have been discussed in chapter 1. From here on, the thesis is structured as follows. Chapter 2 explains all theoretical concepts that are relevant to the work of this thesis. It includes mobile applications, Material Design, and software testing as well as computer vision, machine learning, deep learning and deep neural networks for object detection and instance segmentation. Chapter 3 covers previous work related to the objective of this thesis. Chapter 4 outlines the theoretical approach behind the practical implementation. The implementation itself is presented in chapter 5. It includes the android application for randomized GUIs, scripts for dataset collection, Mask-RCNN as DNN for detection, workflow of dataset collection, and problems that arose during implementation. Chapter 6 evaluates the results. Chapter 7 provides a summary of the entire thesis. Furthermore, it discusses limitations of the approach used, and gives suggestions and ideas for further research in the future. The structure is laid out graphically in Figure 1.

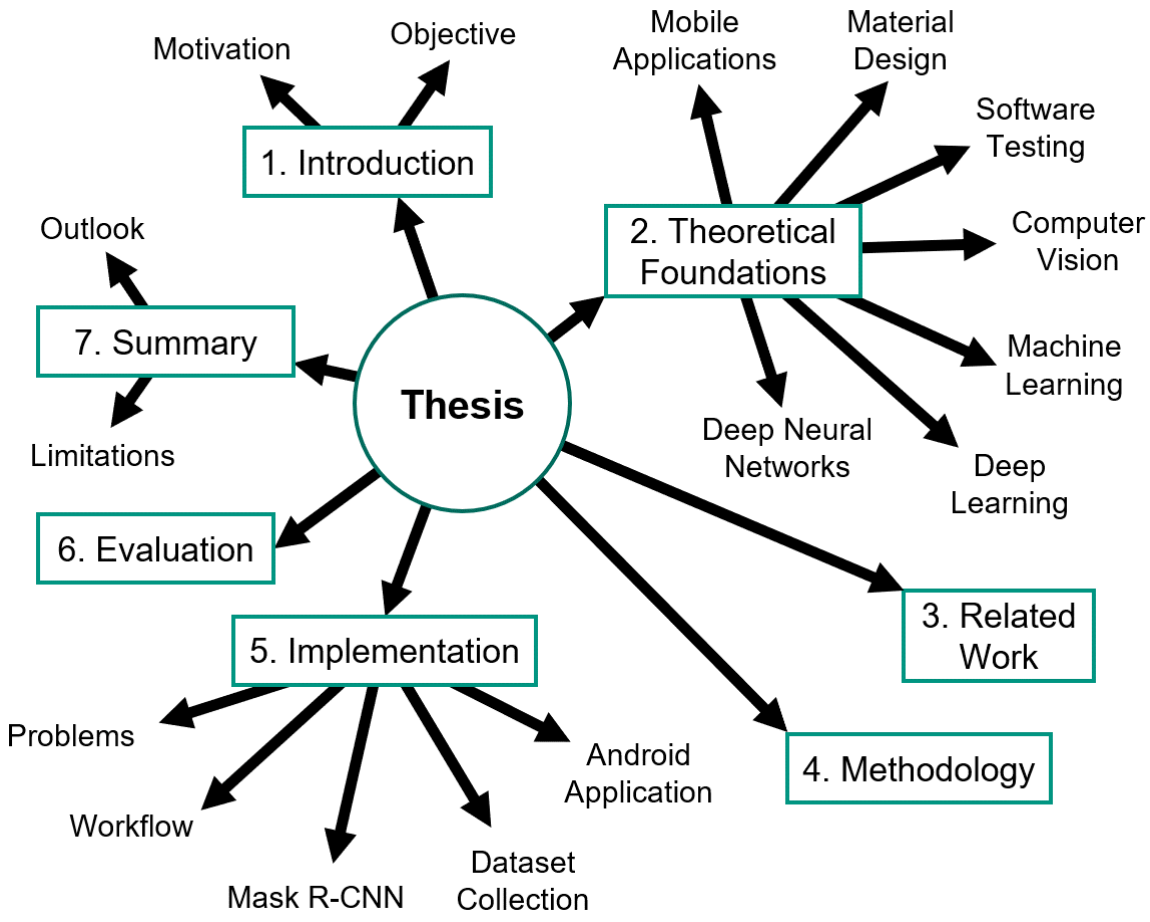


Figure 1. Structure of thesis visualized as mind map. Starting in the upper left, chapters and sections are arranged clockwise.

2. Theoretical Foundations

This chapter provides an overview over subjects relevant for a thorough understanding of this thesis. It covers the following subjects: Mobile applications, Material Design and software testing as well as computer vision, machine learning and deep learning.

2.1. Mobile Applications

Applications are called mobile applications if they are developed specifically for mobile devices. Mobile devices themselves can be classified by their portability, their ability to access the internet via a wireless data connection, their local data storage and their power source that enables usage of the device without a power cable connected. Among others, smartphones and tablets are mobile devices. (Firtman 2010, p. 4; Myers et al. 2012, p. 214; Ross et al. 2020, p. 15)

There are multiple operating systems/platforms for mobile devices. Nonetheless, Android and iOS are almost exclusively used with a respective market share of 70.98% and 28.41% (StatCounter Global Stats 2022).

Mobile applications can be divided into three categories: Native applications, mobile web applications and multi-platform applications. Each of which has its advantages and disadvantages. (Delia et al. 2015; Masi et al. 2013, p. 65)

Native applications use the programming language and tools of a specific platform. This enables applications to use all features of a mobile device (E.g., camera, Global Positioning System (GPS), etc.). Additionally, this offers the best performance. However, support for multiple platforms can only be provided by developing independent applications for each platform. Therefore, the downside of this approach comes down to the programming effort required and costs occurring. (Delia et al. 2015; Masi et al. 2013, p. 65; Zohud and Zein 2021, p. 46)

Mobile web applications run inside a web browser and are programmed using Hypertext Markup Language (HTML), Cascading Style Sheets (CSS) and JavaScript. On the one hand, this makes them highly platform independent. They can be used on all platforms without any platform specific programming overhead. On the other hand, this approach reduces performance. (Delia et al. 2015; Zohud and Zein 2021, p. 47)

Multi-platform applications can be developed using different approaches. All of them have the following goal in common. Reducing or eliminating some of the downsides of mobile web applications and native applications at the same time. I.e., enable hardware access or increase performance in comparison to mobile web applications and still unify application development for all platforms, which is not the case for native applications. One such approach are hybrid applications, where a web application is wrapped in a platform native container. This can potentially give access to more device features than mobile web applications.

Yet, it still reduces performance. Another approach are cross-compiled applications. Here, applications are developed in one environment and compiled into native code for multiple platforms. Lately, the usage of multi-platform approaches in the industry is growing. (Zohud and Zein 2021, p. 47)

2.2. Material Design

Material Design is developed by Google and mainly used for mobile applications. It is a design system for GUIs "[…] inspired by the physical world and its textures […]" (Introduction - Material Design n.d.). Currently, Material Design is available for Android, Flutter, and web. It provides components (examples in Figure 2) and guidelines to build GUIs. GUIs are sometimes only referred to as user interfaces (UIs). The latest version released is Material Design 3. (Introduction - Material Design n.d.)

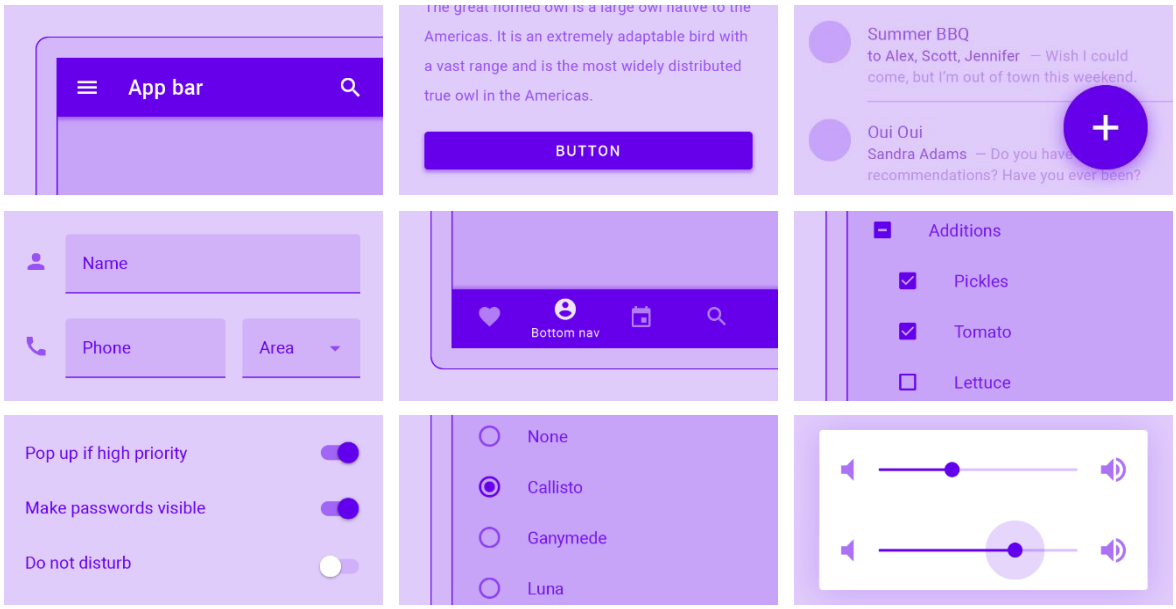


Figure 2. Examples for components provided by Material Design. From left to right, top to bottom: Top app bar, button, floating action button, text fields, bottom navigation, check-boxes, switches, radio buttons and sliders.
(Introduction n.d.)

Components provided with Material Design 3 are grouped into six categories: *Action*, *communication*, *containment*, *navigation*, *selection*, and *text inputs*. *Action* includes various button types with text and/or icons. A special button type are floating action buttons (FABs). There is only one visible at a time and it is typically responsible for the most important action that can be triggered. The category *communication* offers badges, progress indicators and snackbars to display useful information to the user. *Containment* provides wrappers for other components. These wrappers can be cards, bottom sheets, dialogs, dividers, or lists. *Navigation* lets the user navigate between different views. There are tabs, navigation bars/rails/drawers, and app bars. *Selection* gives the user the opportunity to choose between a predefined range of values using date pickers, menus, switches, sliders, radio buttons, time pickers, and chips. *Text input* lets users enter textual information. (Components – Material Design 3 n.d.)

2.3. Software Testing

Software testing is utilized to validate the quality of software. It is the process of assessing whether the software can accomplish the tasks it was designed to do and does not produce unintended results along the way. Single assessments are named tests. If the assessment result is positive, the test is said to have passed. Otherwise, it is said to have failed. For this, software parts are usually executed systematically in a controlled environment. It is a time-consuming task, that makes up a large part of the software development costs. Automated testing can reduce the share of costs, the resources consumed and at the same time further strengthen the reliability of software. (Jamil et al. 2016, p. 179; Luo 2001, p. 1; Myers et al. 2012, p. 2; Shao et al. 2007, p. 137)

Black Box and White Box Testing

Software testing types can be split into two complementary groups: White-box testing and black-box testing. White-box testing refers to structural testing techniques that are designed with knowledge of the source code in mind. They are generally used for verification and answer the following question: "[…] are we building the software right?" (Nidhra 2012, p. 29) In the context of GUI testing, white-box testing would use the source code to find an element and interact with it. Black-box testing spans functional testing techniques. Their design is based solely on the specifications of the software and should not be influenced by the source code. They are generally used for validation and resolve the following question: "[…] are we building the right software?" (Nidhra 2012, p. 29) Here, GUI testing would detect elements based on what is shown on the screen and interact with them, or use random actions. (Liu and Kuan Tan 2009, p. 546; Nidhra 2012, p. 29; O'Regan 2019, pp. 120–127)

Software Testing Types

Unit Testing: The software testing type with the smallest scope is unit testing, which is performed in a white-box fashion. It is usually done by the developer and tests individual units of code. These can be single classes, methods, or functions. (Jorgensen 2018, pp. 229–329; Luo 2001, p. 2)

Integration Testing: This is again a white-box testing technique employed by the developer. It verifies that multiple units work correct in conjunction. For this, it requires properly defined and implemented interfaces. (Jorgensen 2018, pp. 229–329; Luo 2001, p. 2)

System and Acceptance Testing: System testing and acceptance testing are black-box testing approaches. The former assures the quality of the entire system end-to-end and is usually done by independent testers. The latter assesses whether the user/customer is satisfied with the software. A testing framework for mobile applications using a camera for detection and a robot for interaction would fall into the category of system testing. (Jorgensen 2018, pp. 229–329; Luo 2001, p. 2)

An overview over testing types is given in Table 1.

Table 1. Testing types and corresponding information about code opacity, person conducting the test, and scope of test. Based on Nidhra (2012, pp. 30–31).

Testing type	**Opacity**	**Who will do this testing?**	**General scope**
Unit	White-box testing	Generally, programmers who write code they test	For small units of code generally no larger than a class
Integration	White-box testing	Generally, programmers who write code they test	For multiple classes
System	Black-box testing	Independent testers will test	For entire product in representative environment
Acceptance	Black-box testing	Customer Side	For entire product in customer's environment

Code Coverage

Since source code is known for white-box testing, one can determine the percentage of code that is covered by tests. More code covered makes errors less likely. The so-called code coverage can be measured in different ways. *Statement coverage* measures what *percentage of all statements* in the source code is executed at least once. For *decision coverage*, which is also known as branch coverage, the *number of executed decision outcomes* is divided by the *total number of decision outcomes* to calculate the percentage. *A decision outcome* is one possible route taken by the program after the control flow is split up due to some decision that has to be made (i.e., if statements, while loops etc.). (IEEE/ISO/IEC International Standard - Software and systems engineering - Software testing - Part 4: Test techniques, pp. 30–34; O'Regan 2019, pp. 125–126)

Mobile Testing and GUI Testing

Mobile applications are typically required to run on a multitude of devices and varying environmental constrains. Device screens have different resolutions, aspect ratios as well as physical sizes and can often be used in either landscape or portrait mode. There are different types of input (i.e., touch, stylus, mouse, buttons etc.) and the computing power is often inferior to non-mobile devices. Network connectivity can vary greatly or be totally absent. Because of this, extensive testing taking the given constraints into account

is crucial but often quite costly. Mobile testing can be performed on emulators but manual testing with a real device often still takes a necessary part in the process. (Myers et al. 2012, pp. 213–225)

Since mobile applications rely heavily on interaction with the user via GUIs, testing of GUIs is an important part of the testing process. Especially because the complexity of GUIs opens room for errors. Albeit being very important, GUI testing is difficult for various reasons. GUIs often respond to events triggered by the user. Even a single event like a touch interaction is not easy to simulate. At the same time, there is a great number of possible events that can occur and must be tested. Additionally, testing criteria like code coverage are not conclusive for the quality of GUI tests. There are two widely used methods for GUI testing. The first one replays a previously captured user interaction. Because of this, the test can only be created once the GUI has been programmed and must be revised whenever changes in the GUI are made. The second method programmatically simulates interaction. This approach again has its drawbacks because identifying GUI elements and verifying the results is not trivial. (Ruiz and Price 2007, pp. 51–52)

2.4. Computer Vision

The following statements on computer vision are based on Szeliski (2022, pp. 3–9).

For us human-beings, vision is an integral part for our perception of the environment because we can extract a lot of information from it. We are easily able to identify and classify objects, understand their shape, tell their distance from our eyes, and much more. Computer vision is the attempt to transfer this ability to computers. While great progress has been made, computer vision is still vastly inferior to human vision.

This section covers topics of computer vision relevant for the understanding of following chapters in this thesis.

2.4.1. Image Recognition

This section is based on Szeliski (2022, pp. 344–396).

Different types of recognition can be conducted on an image. This section will cover those relevant for this thesis. Further recognition tasks not discussed include *face detection*, *pose estimation* and *panoptic segmentation*. Previously, classical algorithms that do not learn from data were used to tackle recognition tasks. Nowadays, deep learning is becoming more popular since it is well suited to solve such tasks.

Large-scale datasets played an important role in this shift. They are helpful for both training and testing. The first large dataset driving advancements in the field was the PASCAL Visual Object Classes (VOC) challenge (Everingham et al. 2010). However, it provided only 20 labeled classes. This weakness was eliminated in the ImageNet dataset (Deng et al. 2009), which included 1000 classes and over a million images. This enabled end-to-end learning systems to succeed. The Microsoft Common Objects in Context (COCO)

dataset (Lin et al. 2014) allowed further improvements mainly in the area of accurate, per pixel detection of objects.

Figure 24 applies each of the tasks discussed below to the same image to illustrate the difference.

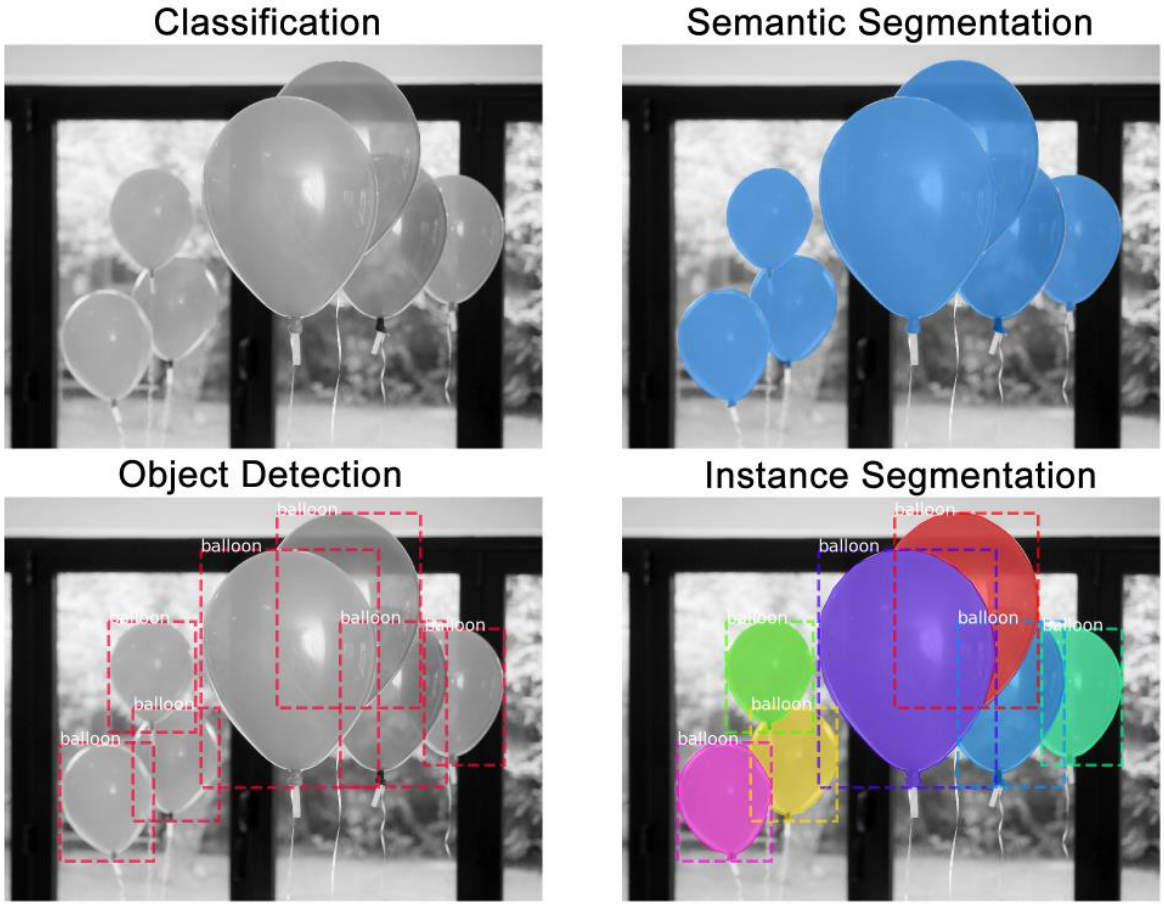


Figure 3. Different computer vision tasks applied to an image: Classification assigns a class to the entire image (upper left). Object detection outputs bounding-boxes for every object in the image as well as a corresponding class (lower left). Semantic segmentation assigns every pixel belonging to a certain class to a mask for that class (upper right. Instance segmentation outputs a binary mask and class for every object found in the image (lower right).
(Abdulla 2018)

The most basic recognition task is *classification*. Here, a specific class is assigned to an entire image.

Next, there is *object detection*. In addition to classification, it localizes objects. The result consists of bounding-boxes, each of which is a rectangular cut-out of the image containing only one object, and the corresponding class. An example would be the detection of faces or pedestrians. Since objects are not necessarily rectangular, the bounding-box usually covers additional pixels that do not belong to the detected object.

Furthermore, there are tasks resulting in pixel-wise, binary masks for the image. This group of tasks is called *segmentation*:

Semantic segmentation produces a common mask for all instances of a class contained in the image. Thus, all pixels of all objects belonging to an individual class are assigned to one mask for that class.

Instance segmentation takes this one step further and produces a mask for each individual object. Hence, if there are multiple objects belonging to the same class, each of them is assigned to the same class but is associated with its own mask covering all pixels of that particular object only.

2.4.2. Selective Search

Selective search is an algorithm in the domain of computer vision designed to propose regions in an image, that are likely to contain an object. It utilizes segmentation to do so. This algorithm is relevant for later parts of this thesis. This section on selective search is based on Uijlings et al. (2013).

At first, selective search divides the image into an initial set of small segmentation regions. Each of the regions should cover at most one actual object in the image. For this, selective search uses a fast version of the algorithm proposed by Felzenszwalb and Huttenlocher (2004).

From this initial set, selective search takes the two most similar regions and merges them. This is repeated until there is a single region left spanning the entire image. Similarity is evaluated by a composition of different criteria: Color, Textures, Size (preferring smaller regions), and Position (preferring regions having

Figure 4. Segmentation masks at various stages in the hierarchical grouping process starting with many small regions and ending in few big regions. Underneath, the corresponding images with bounding-box proposals at each stage. Bounding-boxes of actual objects are highlighted. (Uijlings et al. 2013)

a large common border or regions, where one encloses the other). This results in a hierarchical bottom-up grouping process as can be seen in Figure 4. All the regions produced during this process are regarded as proposals for possible objects in the image.

2.5. Machine Learning

The following statements on machine learning (ML) are mostly based on Zhang et al. (pp. 15–32).

Usually, algorithms solving specific problems are explicitly programmed by human beings. For this, humans require an understanding of the connection between a given information (input) and the desired answer (output). Only then can they convert this knowledge into a computer program. While this approach yields satisfying performance for many tasks, other tasks are hard to solve or cannot be solved at all using this approach. This is because human-beings are not always able to formulate a well-suited algorithm for various reasons.

Some of these tasks like object detection, humans can perform subconsciously with ease. However, at the same time they are not able to consciously understand the connection and formulate a well-working algorithm to solve this task. At other tasks, machine learning approaches exceed the performance of most human-beings. One example is the improved detection of skin cancer based on images (Haenssle et al. 2018). Another example is the more reliable discovery of possibly dangerous polyps, which can develop into cancer, during colonoscopy (Alessandro Repici et al. 2020). In addition, some problems do not have a static answer. The answer can change over time. This requires an algorithm to adapt during execution. (Zhang et al., pp. 15–16)

Machine learning offers an alternative approach to manually programmed algorithms by learning from experience as stated in the definition given in the following well-known quote:

> *"A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E." (Mitchell 1997)*

In machine learning, a *model* digests input and transforms it in some way to generate output. The transformations applied depend on *parameters*, which are tweaked by a *learning algorithm*. Based on experience, the learning algorithm uses an *objective function* to adjust these parameters in a meaningful way that improves the *performance* of the model at the given *task*.

This experience is provided by sample data. A collection of data is called *dataset*. Individual entries are named *data points*. A data point always consists of *features*, which are inputs for the task/problem to be solved. Depending on the type of learning, a data point can also include the corresponding desired output, a so-called *label*. E.g., a data point could be an image where the individual pixel values are input features and a classification like "Cat" or "Dog" is the output label.

Usually, there are two distinct datasets, a *training dataset*, and a *test dataset*. The former is used during learning while the latter is used to judge the performance afterwards. This is needed because good performance on the training dataset does not necessarily translate to good performance on unseen data.

Connected to this is the concept of *generalization*. If the model performs well on the training dataset but not on the test dataset, the model does not generalize well. This is also called *overfitting*. Here, the model found a connecting pattern between features and labels, which is exclusively found in the training dataset and not shared with the data of the test dataset. If the model performs well on both training and test dataset, it generalizes well. However, the model can also happen to perform bad on both datasets, which is then called *underfitting*. When the model is underfitting, this often means that the patterns underlying the data relevant to the task are too complex to be described by the model (E.g., a linear model used to fit a quadratic function.). It is also possible the model does not yet have enough experience to uncover this pattern. All three possible results are illustrated in Figure 5. (Goodfellow et al. 2016, pp. 109–110)

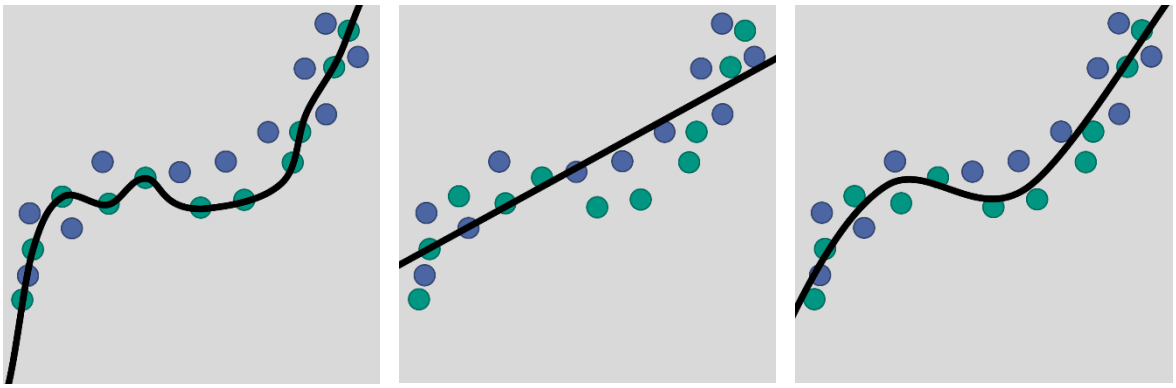


Figure 5. Three graphs with a line fitted to training data. Green points are part of the training dataset and blue points part of the test dataset. In the left graph, the line is overfitting. It describes the green points very well but not the blue ones. In the graph in the middle, the line is overfitting. It describes neither green nor blue points appropriately. In the right graph, the line describes both green and blue points decently indicating good generalization.
Based on Nguyen and Zeigermann (2021, pp. 92–95).

There are broadly three types of machine learning: *Supervised learning*, *unsupervised learning*, and *reinforcement learning*.

Supervised learning takes a training dataset for learning, in which features and labels (inputs and outputs) are present. The learning algorithm then produces a model with fitted parameters, which we can use to acquire a *prediction* of the labels of new, to the model unknown input features as seen in Figure 6. (Szeliski 2022, p. 239)

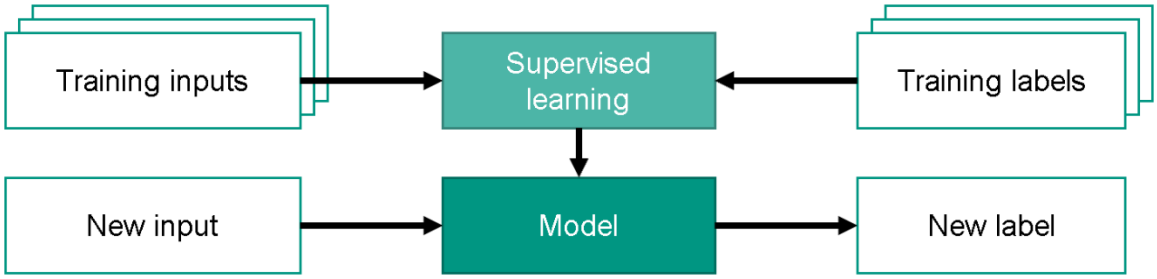


Figure 6. Supervised learning takes training inputs and training labels as input. It outputs a mode with learned parameters, that can be used to predict outputs for new, to the model unknown inputs.
Based on Zhang et al. (2021).

Executing the model with previously unseen data as input to predict unknown labels is sometimes referred to with the term *inference*, even though this term is ambiguous and can thus create confusion (Zhang et al., p. 92).

Common tasks solvable by supervised learning include *regression* and *classification*. For regression, the output/label is a numerical value, and for *classification*, the output is one of multiple categories (classes). An example of both tasks is displayed in Figure 7. (Szeliski 2022, p. 239)

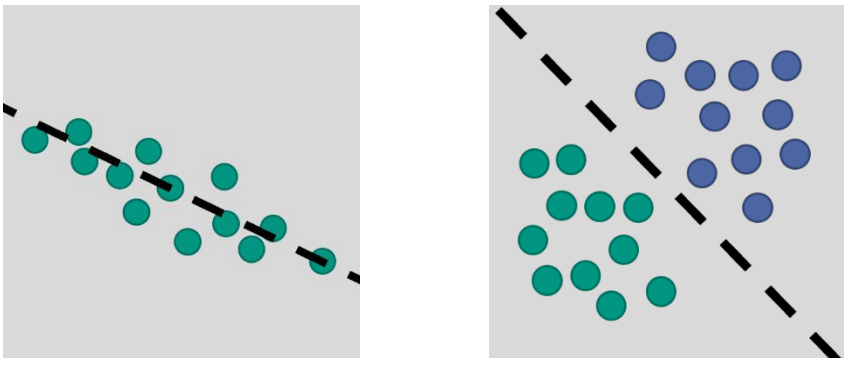


Figure 7. Example of regression: Fitting line to dataset (left), and example of classification: Line dividing dataset into two distinct classes (right).
Based on Soni (2018).

Unsupervised learning takes a training dataset similar to supervised learning. However, the dataset consists only of features, and does not have corresponding labels. Thus, the learning algorithm is supposed to find patterns inside the dataset on its own. (Szeliski 2022, p. 257)

Both supervised and unsupervised learning can be summarized under the term *offline learning* because training takes place before and independently of the environment in which the model is later used. This is problematic if the environment changes, either independently or in response to results of the machine learning model. E.g., a model might initially be able to filter out all spam emails received but the attacker could alter their writing style in an attempt to evade the detection.

Reinforcement learning differs from the previous approaches. Here, a so-called *agent* learns a *policy* during exposure to the environment. I.e., the agent perceives the evolving environment and takes this into account for future actions. For this, the agent can observe the environment and take actions influencing the environment. Furthermore, the agent receives a *reward* for every action taken. This reward is determined by an objective function judging how successful the action of the agent was for achieving the task. Based on the reward, the learning algorithm aims to improve the policy to receive a reward as high as possible. The interaction of the various parts can be seen in Figure 8. (Zhang et al., p. 31)

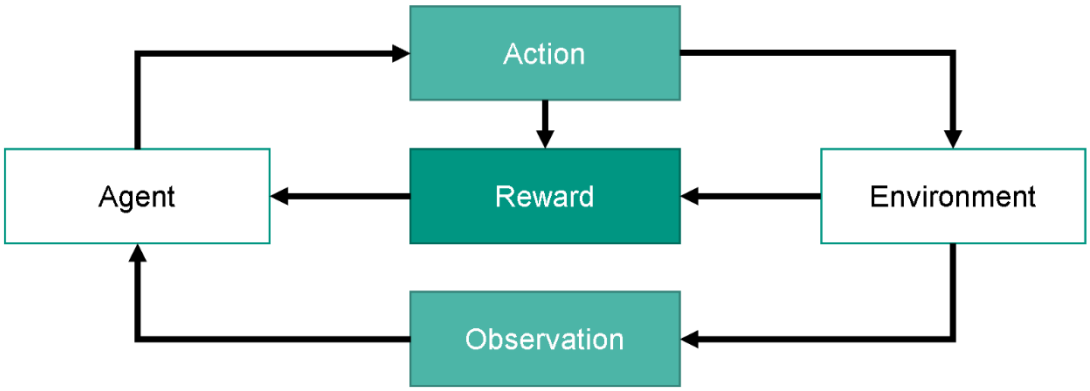


Figure 8. Interaction of agent and environment through observation, action, and reward. Based on Zhang et al. (2021, p. 31).

2.5.1. Support Vector Machine

One algorithm in the domain of supervised machine learning is support vector machines (SVMs). The following statements on SVMs are based on Bishop (2006, pp. 325–339).

Essentially, an SVM divides features of a dataset into two classes. It does this by learning a linear function, which splits the dataset in two. All data points lying on one side of the function are categorized as one class while all data points on the other side are categorized to the other class. The distance between the function and the nearest data point of a class is called margin. To find an appropriate linear function dividing the data, this margin is maximized.

However, this approach only works for a dataset containing two classes for which the data points are linearly separable. If they are not linearly separable, a so-called soft-margin is introduced meaning the margin is allowed to take on negative values for data points that prevent linear separation otherwise. Both variants are shown in Figure 9.

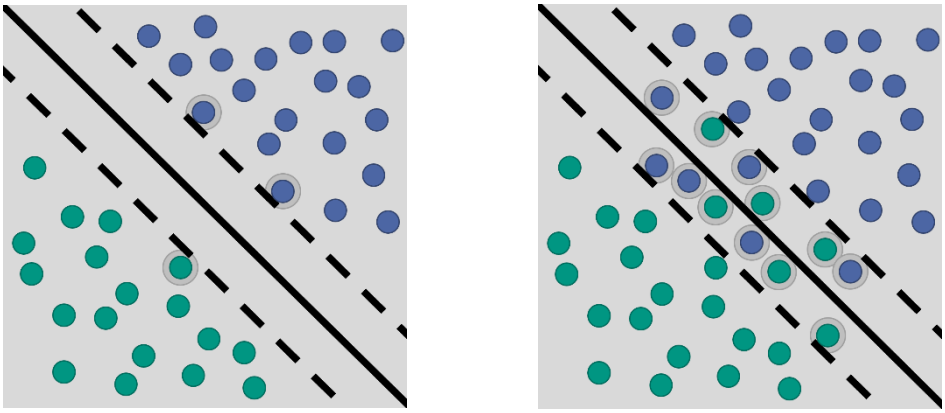


Figure 9. SVG with hard margin separating linearly separable data into two distinct classes (left) and SVM with soft margin for data that is not linearly separable (right). Based on Nguyen and Zeigermann (2021).

For datasets consisting of multiple classes (i.e., more than two classes), there exist two approaches summarized under the term multiclass SVMs.

The first approach trains as many SVMs as there are classes. Each SVM is responsible for one class and trained by treating this exact class as one class while grouping all other classes to a second class. Because of this, it is also named one-versus-the-rest approach. It can be seen in Figure 10 (left).

The second approach goes through all possible pairings of two classes from the entirety of classes. It trains an SVM on each of them. When given a data point, each SVM votes to which of its classes the data point belongs. The class receiving most votes is then chosen as predicted label. This approach is sometimes referred to as one-versus-one and visualized in Figure 10 (right).

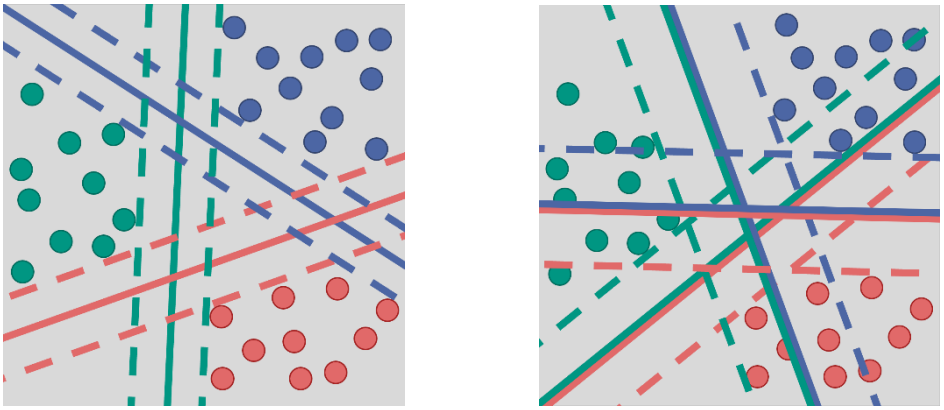


Figure 10. Multiclass SVM approaches: One-versus-all separating each one class from all other classes (left) and one-versus-one separating pairs of classes one by one (right). Based on baeldung (2020).

2.6. Deep Learning

It is different to other machine learning approaches insofar that input values are transformed into output values by performing multiple operations on it one after another (Zhang et al., p. 19).

This chapter starts with an explanation of the most basic model family for deep learning. After this, it shows how learning works in the context of this model family. Finally, it gives an explanation on a different model family especially used for image recognition as well as metrics used in the domain of deep learning.

2.6.1. Multilayer Perceptrons

There are different terminologies used for the most basic type of deep neural networks (DNNs): Deep feedforward network, feedforward neural networks or multilayer perceptron (MLP) (Goodfellow et al. 2016, p. 164; Zhang et al., p. 167). This type was initially derived from what was understood of the human brain and its nervous system at the time of its invention (Da Silva et al. 2016, p. 11). As the different names already imply, this type of DNNs consists of many neurons that are interconnected through a network. More specifically, a network is a structure of multiple layers chained together where each layer in turn is composed of multiple neurons (Zhang et al., p. 167).

A single neuron is typically comprised of the following components: It has a vector of input values x and an output value y. Additionally, it has multiple parameters and an activation function, which can be seen as properties of the neuron. The parameters are a vector of weights w and bias b. (Da Silva et al. 2016, p. 12; Szeliski 2022, p. 270)

The neuron computes its output value y from its input vector *x* in the following manner: At first, it weights each input value x_i, i.e., each input x_i is multiplied with its corresponding weight w_i. Then, all weighted inputs are summed up. This is equivalent to the dot product of vector *x* and *w*. After this, the neuron adds bias *b* to the sum. At last, an activation function is applied to the result yielding the final output *y* of the neuron. The process as a whole is visualized in Figure 11. (Da Silva et al. 2016, pp. 12–13; Szeliski 2022, p. 270)

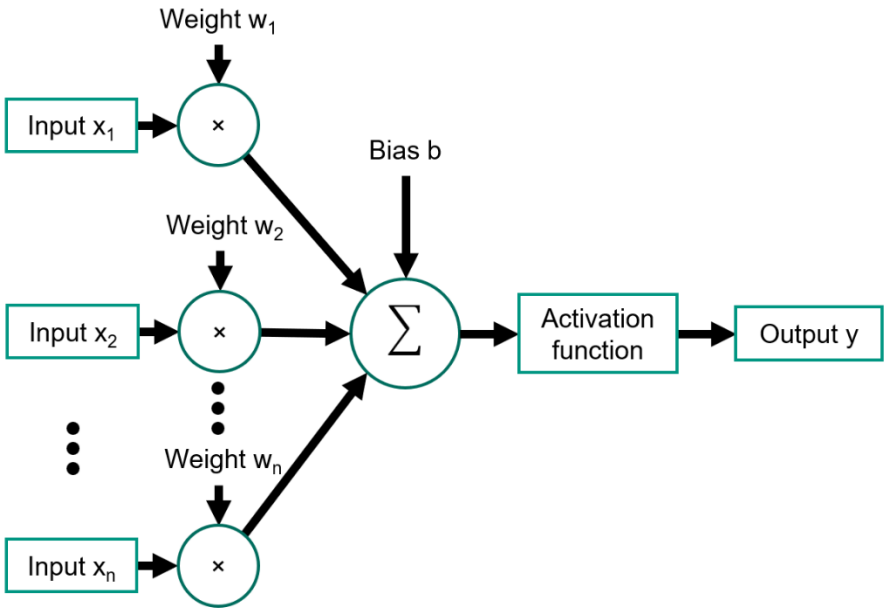


Figure 11. Internals of an artificial neuron: Input is weighted and summed up with a bias. After applying an activation function, the output is obtained. Based on Da Silva et al. (2016, p. 12).

There exist various activation functions as seen in Figure 12, the most popular including rectified linear unit (ReLU) and sigmoid. The ReLU function outputs the maximum of the input and zero bounding the output to values equal to or greater than zero. The sigmoid function maps the input to the range of zero to one. Another important activation function is the softmax function. It is similar to the sigmoid function (Mercioni and Holban 2020, p. 144). However, it takes not one value but a vector as input. It is thus applied to a layer of neurons and not on a neuron individually. It maps each value of the vector to a number between zero and one. The results are then normalized producing a probability distribution. All values of the output vector add up to one (Banerjee et al. 2020, p. 2; Szeliski 2022, p. 274). This probability distribution is often useful in the context of multiclass tasks. Activation functions are needed to give DNNs capabilities beyond linear machine learning models. (Ansari 2020, pp. 146–151; Zhang et al., pp. 169–173)

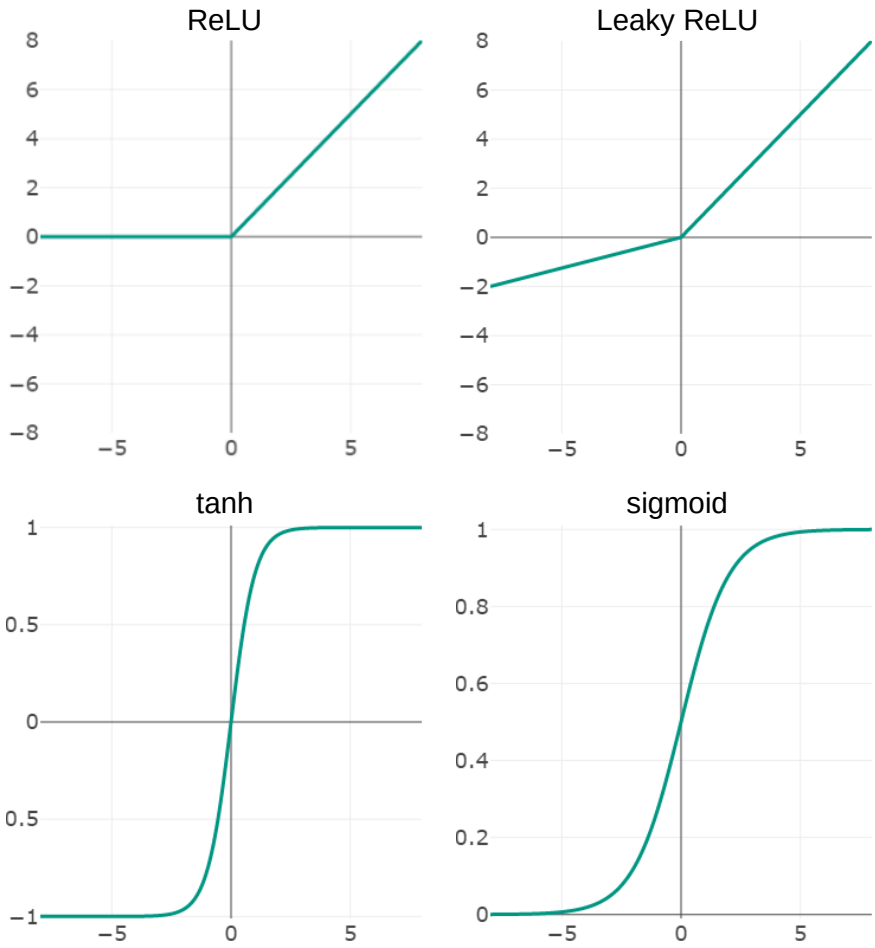


Figure 12. Widely used activation functions: Rectified linear unit (ReLU) is a linear function for positive values and equal to zero for negative values. Leaky rectified linear unit (Leaky ReLU) is the same for positive values but uses a fraction of the value for negative values. Hyperbolic tangent (tanh) maps values to a range of negative and positive one. The sigmoid function is similar but maps to values between zero and one.
Based on Szeliski (2022, p. 273).

In this type of DNN, a neuron in one layer is connected to each neuron of the successive layer. Meaning the output value of this neuron is also an input value for each neuron in the following layer. Because of this, the size of the input vector of these neurons is equal to the number of neurons in the previous layer. This way of connecting the neurons gives this type of layer its name: Fully connected layer (fcl). There are other types of layers. Some of which, namely convolutional layers and pooling layers, will be explained in section 2.6.3 because they are relevant for computer vision tasks. (Szeliski 2022, pp. 271–272; Zhang et al., p. 169)

An MLP consists of one input layer, a varying number of intermediate so-called hidden layers and one output layer. Hidden layers and output layer are all fully connected layers. However, the input layer is different to all other layers. It simply represents the input feature vector and does not include any neurons or other computations whatsoever. The name of the hidden layers stems from the fact that no desired outputs are known for these layers. It must thus be derived during learning, which will be covered in the next section. The output layer provides the final result of the MLP. Since training is supervised, the desired results of

this layer are known when training the MLP because ultimately the output is supposed to predict the label belonging to the input feature vector. Usually, the output layer uses the softmax activation function to acquire a probability distribution over all classes (Szeliski 2022, p. 274). A graphical example of an MLP, where the network is visualized as a graph with neuron as vertices/nodes and connections between neurons as edges, can be seen in Figure 13. (Goodfellow et al. 2016, p. 165; Zhang et al., p. 169)

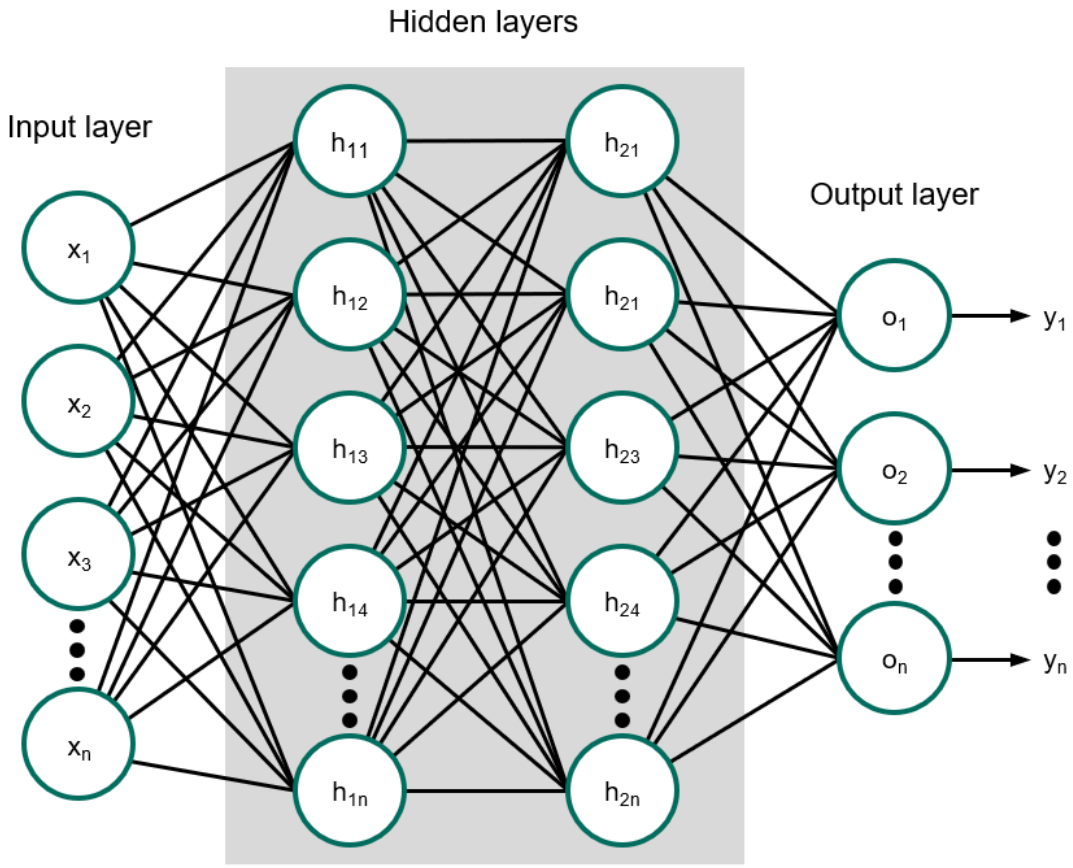


Figure 13. Basic example of an MLP with two hidden layers as graph with neurons as vertices/nodes and connections as edges. Every input in the input layer is connected to every neuron of the first hidden layer, each of which is in turn connected to every neuron of the second hidden layer. All those neurons are interconnected with each neuron of the output layer resulting in the final output of the MLP.
Based on Da Silva et al. (2016, p. 23).

2.6.2. Optimization

MLPs have different types of properties that can be tweaked. On the one hand, they have so-called hyperparameters like the number of layers and the number of neurons in each layer. These are up for a human-being to decide. On the other hand, neurons have internal parameters themselves. Each neuron has a weight and a bias, which can be adjusted. Because of the way neurons are interconnected, the number of parameters is much higher for MLPs than for other machine learning algorithms. One could assign each parameter a random value, provide an input feature vector, and perform all the calculations discussed in the previous

section above to retrieve a result. Performing these calculations is also named forward propagation or forward pass because it starts with the input layer and end with the output layer. However, the likelihood that this forward propagation will yield an in any way meaningful result is vanishingly small. Therefore, a learning algorithm must tweak the parameters for good results. (Zhang et al., p. 180)

There are different ways to look at and explain the working principles of the learning algorithm. Put simply, the learning algorithm uses supervised learning in the following manner. At first, it performs forward propagation with the input vector of a datapoint in the training dataset. Then, it compares the outputs of the MLP with the desired outputs provided by the training sample and slightly adjusts the parameters in a way that moves the actual outputs a little bit closer to what is desired. This process as shown in Figure 14 is repeated many times until the difference between actual and desired outputs does not decrease anymore. However, this is a very simplified view at what is happening during learning. (Moore et al. 2021)

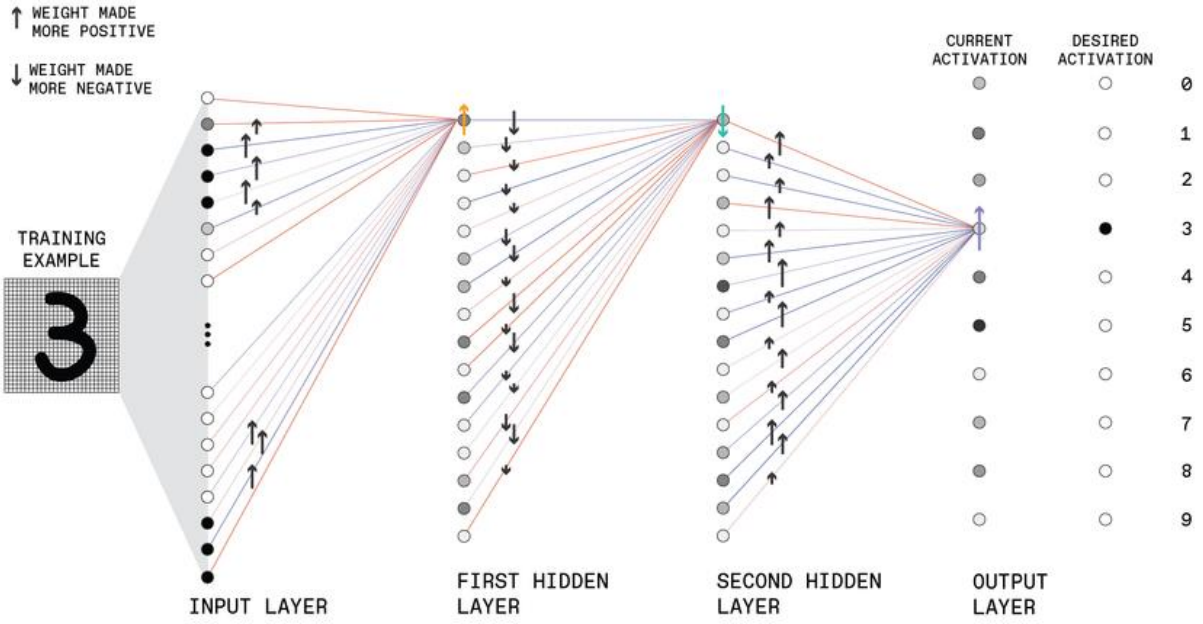


Figure 14. An example of an MLP trained on images of digits. First, forward propagation is performed. Then, actual outputs are compared with desired outputs and parameters (weights) are adjusted to get closer to the desired result.
(Moore et al. 2021)

Loss function

More specifically, this difference between actual output produced by forward propagation and desired output as given by the label is quantified using a loss function. There exist various loss functions. E.g., a very popular loss function for multiple classes is categorical cross-entropy. A high output of the loss function means prediction is way off, an output near zero means good prediction accuracy. Since this loss function measures a difference, it cannot be negative. (Szeliski 2022, p. 280)

Backpropagation

To calculate predictions for different inputs, all internal parameters of the MLP are regarded as fixed and inputs are seen as variable. For learning, this is reversed. The input is fixed but the internal parameters of the MLP can be adjusted. As a result, the loss function is a multivariate function, which has as many variables as there are internal parameters to the MLP.

Now, to improve the accuracy of the MLP, the loss function must be minimized. In other words, the goal is to find the global minimum of the loss function by adjusting the internal parameters. This is an optimization problem that can be solved using derivatives. The chain rule is used to calculate the gradient of the loss function, which contains information about how a small change in each of the parameters changes the final output of the function. I.e., does a slightly higher value for a parameter increase or decrease the output value and how strongly does the output react to that change.

Because the chain rule requires calculations for the gradient to be performed by starting at the last layer and working backwards until the first layer, this process is named backpropagation as opposed to forward propagation. Backpropagation is performed computationally. It requires the computer to keep track of dependencies of individual parts of the MLP in a graph structure. These computations are referred to as automatic differentiation or autograd in short (Zhang et al., p. 68). (Goodfellow et al. 2016, pp. 204–207; Zhang et al., pp. 181–182)

Learning Algorithms

The gradient resulting from backpropagation is multiplied with a hyperparameter named learning rate and subtracted from the current parameters to receive updated parameters that are used for all future calculations. When performing forward propagation with these updated parameters on the same datapoint used for backpropagation, the output of the loss function should now be lower indicating improved prediction of the MLP for that particular datapoint after one step. If the dataset consisted of one datapoint in total, all that is left to do is to repeat this step until improvements stall. This is called gradient descent. However, an MLP trained on one training sample would not be able to learn any general patterns. It would overfit to that one datapoint and not be of any use.

Thus, a larger dataset is needed, and backpropagation and updating of parameters must be repeated for many datapoints in the dataset. Running calculations for all datapoints once is referred to as one epoch of training. Since the optimal configuration of parameters differs for every datapoint, the gradients would vary greatly and output of the loss function over time would be very noisy. This way of learning explained so far, where parameters are updated after calculation of one datapoint, is named stochastic gradient descent (SGD). (Goodfellow et al. 2016, pp. 290–292; Szeliski 2022, pp. 287–288)

Apart from SGD, there is also minibatch stochastic gradient descent and gradient descent using whole batches. For minibatches, not one but multiple datapoints are used and backpropagation is performed for all of them before updating the parameters with the summed up gradients of these datapoints. For whole batches, this goes even further and the entire dataset is used to calculate updated parameters afterwards. All these approaches have pros and cons. E.g., optimization using batches has problems with local minima, which are not the best solution to the optimization problem. Additionally, it requires the most memory of all approaches and can be very inefficient as explained in Figure 15. Sometimes, this can be overcome by adjusting the learning rate. (Goodfellow et al. 2016, pp. 290–292; Szeliski 2022, pp. 287–288)

However, there exist more sophisticated algorithms like ADAM (Kingma and Ba 2014), which make use of additional information to be more efficient. ADAM specifically uses momentum calculated using previous gradients. This is comparable to a marble rolling down a hill. Even if the direction of steepest descent differs from which way the marble is currently rolling, it does not directly move in this direction but slowly turns because of the momentum carried on from before. This would work well to counter the effect discussed in Figure 15. (Goodfellow et al. 2016, pp. 292–293; Szeliski 2022, pp. 289–290)

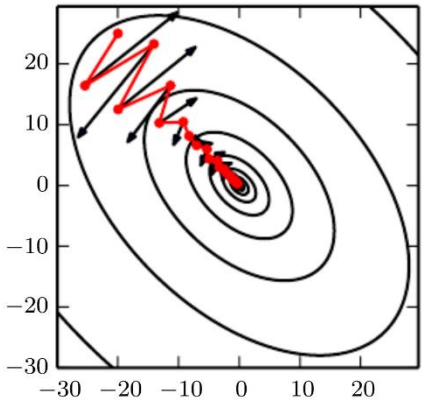


Figure 15. Graph showing gradient descent (red line) for a function with two inputs using an entire batch. Black lines indicate points where output level of the function is the same. Therefore, the shown example ca be thought of like an elongated valley. Because of this, gradients (black arrows) do not point to the center of the valley. This results in alternating gradients after every step of gradient descent making optimization take much longer compared to the shortest path downhill.
(Goodfellow et al. 2016, p. 293)

Hardware

Because of high parameter count and large datasets, the learning process requires quite a lot of computing power and memory especially for deep learning. Since most of the computations are matrix multiplications and graphic processing units (GPUs) happen to be specialized to perform those quickly and in parallel, GPUs are well suited as hardware for training. Apart from GPUs, there exist some hardware architectures

like tensor processing units especially optimized for this task. Eventually, quantum computers could also become useful for this (Biamonte et al. 2017). (Zhang et al., pp. 34–36; Szeliski 2022, p. 964)

Methods for Faster/Better Results

There are some additional measures apart from improved learning algorithms that can be taken to further improve the results.

To reduce training times in practice, transfer learning is a widely adopted method. When using a neural network that someone else already trained on some dataset, one can reuse the resulting parameters and continue training on a different dataset. Because part of what the neural network already learned can often be useful beyond its initial dataset (E.g., for images the first layers often learn more generalized patters like edges, textures and shapes, that are present in other data as well), less training is needed to achieve similar results compared to training of a neural network with randomly initialized parameters. (Zhang et al., p. 606)

When such a pretrained neural network is used as part of a larger network, the rest of the neural network must be trained from scratch. However, the pretrained part is likely to already be somewhat optimized. Thus, its parameters will not change as much as the rest. This case is referred to as fine-tuning. (Zhang et al., p. 606)

Sometimes, it is desired to train parts of a neural network in isolation. For this, all parameters not belonging to that part are frozen meaning they are used for forward propagation, but parameters are not updated and kept fixed during learning.

In general, neural networks have many more parameters than other machine learning algorithms. This makes them more likely to overfit. This means decreased performance for new data. Two ways to counter this are data augmentation and dropout.

Overfitting can be prevented by using larger datasets. However, if the is no more data that can be collected, a workaround is needed. Data augmentation is the process of changing the appearance of already existing datapoints and using this as additional datapoints for training (E.g., rotating, scaling or otherwise altering an input image.). (Zhang et al., pp. 597–602)

Dropout is the process of randomly disabling a part of the connections between neurons. I.e., these connections are not used during calculations at all (neither forward propagation nor backpropagation). Therefore, a neural network cannot rely on a single connection to detect a certain feature and must find other ways to detect it resulting in better generalization. (Zhang et al., pp. 193–194)

Example in Two Dimensions

For the case of binary classification of points in a two-dimensional plane, there is a very neat way to visualize the learning process in a different way by computing the output of the MLP for the entirety of the plane and assigning colors to the classes to get a sense of how learning solves the task. Essentially, the parameters are tweaked in such a way that the entirety of neurons represent a nonlinear boundary separating the classes. An example is given in Figure 16.

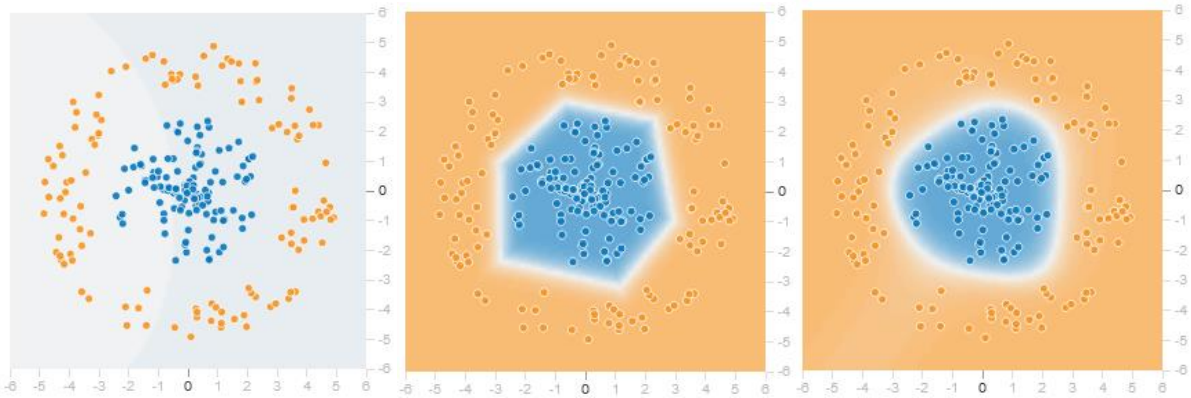


Figure 16. Three two dimensional plots with datapoints of two classes (marked in yellow and blue), where blue points are in the center and encircled by a ring of yellow points. The background is colored according to what class a created MLP would assign to a point at that position. Shades in between yellow and blue indicate uncertainty. In the left plot, background has no clear colors. In the middle, a blue polygon is surrounded by yellow background. In the right plot, a blue oval is encircled by yellow background.
Based on Carter, Daniel Smilkov and Shan (2017).

The left plot shows the output of an MLP where parameters are initialized at random, and no training took place. The colors of the background do not correspond with the color of the datapoints and are in general very neutral. Thus, correct classifications are low, and uncertainty is high.

The plots in the middle and on the right show the output of MLPs after training. In both plots, there is a very clear blue center containing all blue datapoints while the surrounding area is clearly yellow containing all yellow datapoints. This implies that training managed to adjust the parameters very well to solve the given task. In the middle, the boundary between yellow and blue forms a polygon while the boundary on the right forms an oval. This is due to different activation functions utilized in the MLPs. For the results in the middle, ReLU was used as activation function. For the right, sigmoid was used as activation function. This shows how different activation functions can influence the behavior of a MLP for learning.

In a way, the plots are comparable to those obtained by SVMs except there are two differences. Classification is not linear because of activation functions and there is no clear binary border but some degree of uncertainty near the border. In general, MLPs with higher parameter count can solve more complex problems than given in the example shown in Figure 16.

The source for the plots is an interactive website and it is encouraged to visit the source as it provides a very intuitive way of understanding MLPs.

2.6.3. Convolutional Neural Networks

The previous sections discussed the basics of deep learning including the model architecture and learning process of MLPs. While MLPs can be used for computer vision tasks, they are invariant to the structure of their inputs. Changes in the order of the input vector prior to learning do not influence the potential capabilities of the MLP after learning. However, this means that the MLP does not make use of the order even when knowledge of the structure can be advantageous. In images for example, pixels nearby are more likely to contain information that is related to each other. MLPs can gather information about the structure during learning but it would be desirable to retain and utilize this information from the beginning on because it reduces parameter count and thus time needed for computations. A network type doing exactly this are convolutional neural networks (CNNs). They can be beneficial for tasks where inputs have a known matrix-like structure as images, time-series and audio-sequences do (Goodfellow et al. 2016, p. 326; Zhang et al., p. 233).

To take advantage of the input's structure, convolutional neural networks utilize new layer types named convolutional layer and pooling layer which will be explained in the following paragraphs. (O'Shea and Nash 2015, p. 4)

The explanation for convolutional layers uses inputs with two dimensions. An example for such an input would be a greyscale image. Nonetheless, this concept can be applied to inputs of arbitrary numbers of dimensions (including one dimension). Convolutional layers have a matrix of weights called kernel. This kernel is typically much smaller than the entire input matrix. Convolutional layers start with a rectangular cutout the same size as the kernel in one corner of the input matrix. Each input value of the cutout is multiplied by the corresponding weight of the kernel. Afterwards, all results are summed up and an activation function is applied resulting in one output value. As next step, this same cutout is shifted over the input matrix so that it now contains different input values. Again, multiplication, summation and activation are performed on the input values. The same weights and the activation function from the previous cutout are used again. This shifting is repeated for every cutout in the input matrix resulting in a matrix of outputs. Because of this shifting, the cutout is usually referred to as sliding window. The steps happening before the

activation function are illustrated in Figure 17 for two positions of the sliding window. Each convolutional layer has an arbitrary number of kernels. (O'Shea and Nash 2015, pp. 5–6; Zhang et al., pp. 240–242)

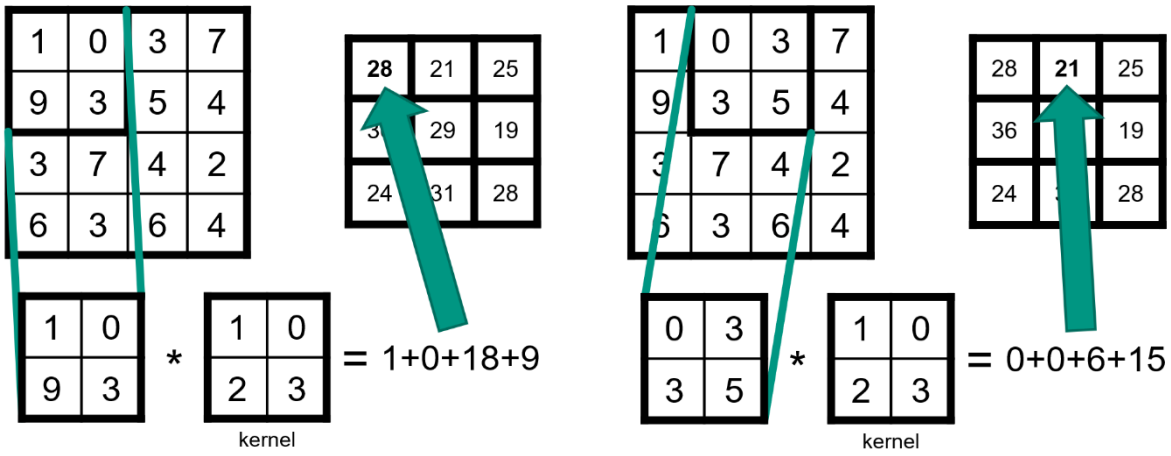


Figure 17. Shown above is an example of two positions of the sliding window multiplied with the kernel resulting in a single output value per position.

The resulting output matrix is sometimes referred to as feature map. The reason for this can be deducted from Figure 18. Because the weights of a kernel are fixed for the entirety of the input, each kernel is responsible for one specific feature in the input image. Whenever a feature is present in the input image, the output value of the kernel responsible for the feature spikes at the position of the feature. Therefore, the output matrix maps features to locations in the input image. (Zhang et al., pp. 244–245)

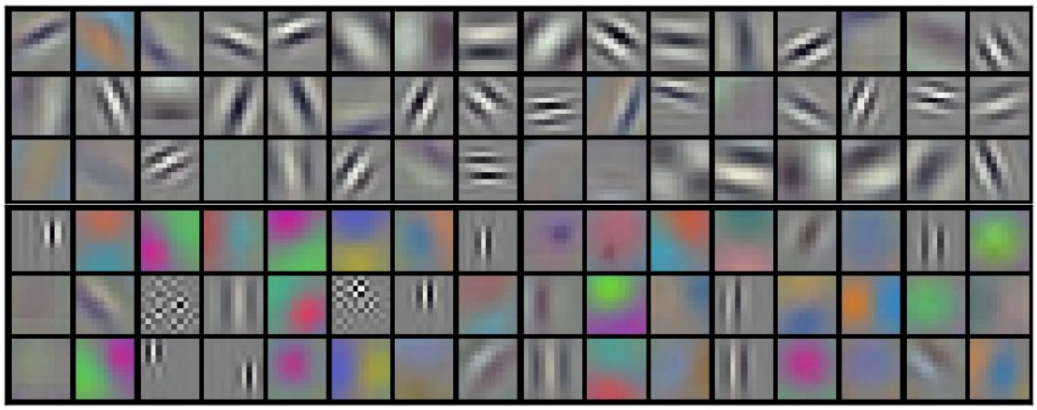


Figure 18. Each cell represents the learned weights of a kernel for an image with three color channels. The various kernels picked up specific features of the input like edges, textures, or color combinations.
(Krizhevsky et al. 2012, p. 6)

The sliding window of the kernel can be shifted by one or more entries in the input matrix. The amount of shifting applied each time is a hyperparameter named stride. Another new hyperparameter is padding. Padding indicates if and how the original input matrix is surrounded with artificially added inputs (possible values for these added inputs are same as real input next to it and all equal to zero). This is done, because

whenever the sliding window is greater than one, the output matrix is smaller than the input matrix, which is not always a desired result. E.g., with a sliding window of 3 × 3 and a stride of 1, an n × n input matrix leads to an n-1 × n-1 output matrix. (O'Shea and Nash 2015, pp. 5–6; Szeliski 2022, p. 294)

In a way, the approach used in a convolutional layer is comparable to a neuron of a fully connected layer. The input is weighted and summed up. Then, an activation function is applied. But instead of connecting the neuron to every input at the same time, the calculations are performed for one cutout at a time while reusing the same neuron with the same weights for all of the cutouts. This greatly reduces parameter count (Lecun et al. 1989, p. 544).

Pooling layer yet again use a sliding window of fixed size. However, they do not perform the same calculations on this window as convolutional layers do. When using average pooling, all inputs values inside the window are averaged resulting in a single output. For max-pooling, only the highest input value is return as output value. An example for the process of max pooling is provided in Figure 19. (Ansari 2020, p. 200; Goodfellow et al. 2016, p. 335)

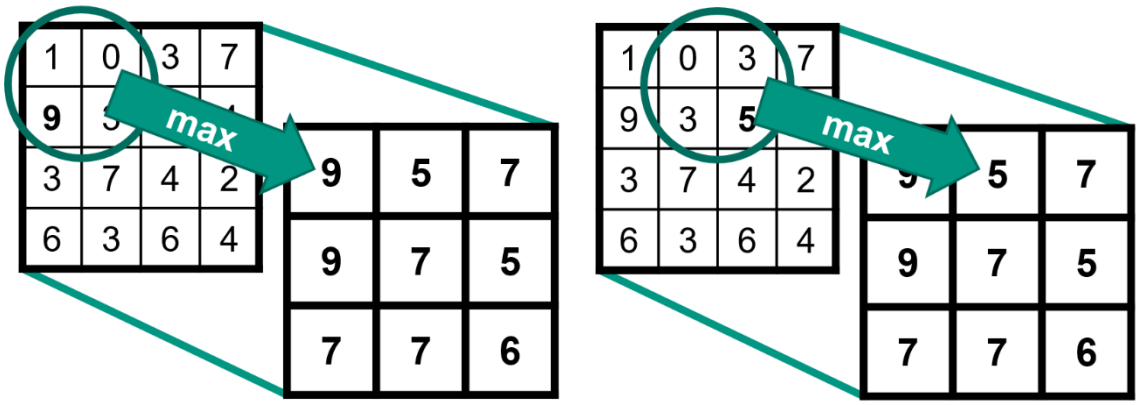


Figure 19. Max pooling with a window size of 2 x 2 applied to a sample input. The maximum of all values in the window is computed. Then, the sliding window is moved to the next position and the process is repeated.

Usually, CNNs use convolutional layers and pooling layers in the beginning followed by one or more fully connected layers as shown in Figure 20. Since the output of those layers is multidimensional, it must be flattened to one dimension to be fed into fully connected layers. Some network architectures only use convolutional layers and pooling layers. These are typically named fully convolutional networks (Szeliski 2022, p. 272). (Ansari 2020, p. 201)

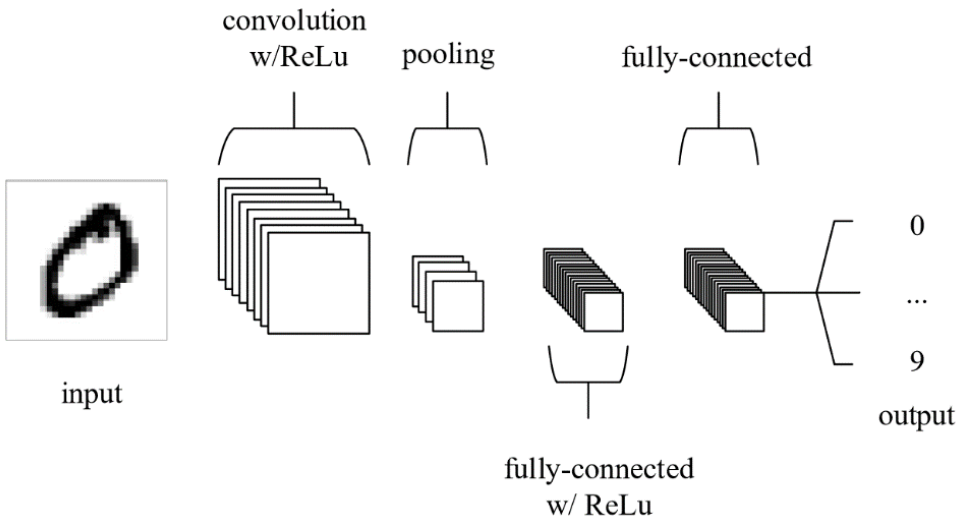


Figure 20. Example of simple CNN architecture with one convolutional layer, one pooling layer and two fully connected layers.
(O'Shea and Nash 2015, p. 4)

More complex model architectures of DNNs often rely on pretrained common CNNs as base network. These base networks are sometimes referred to as backbones. Some of those are presented in Table 2.

Table 2. Common Convolutional Neural Network Architectures. Based on (Bharati and Pramanik 2020, pp. 659–660).

LeNet (Lecun et al. 1998)
AlexNet (Krizhevsky et al. 2012)
ZF Net (Zeiler and Fergus 2014)
GoogLeNet (Szegedy et al. 2015)
VGGNet (Simonyan and Zisserman 2014)
ResNet (He et al. 2016)

2.6.4. Metrics

There exist various metrics to measure performance of DNNs. Some of those are relevant for the understanding of related work and judgement of the performance of their results as well as those of this thesis. Therefore, this section gives a brief explanation for each of them.

An understanding of the possible outcomes of a prediction and their relationship is useful for the explanation of the first two metrics: Precision and recall. Two distinctions can be made resulting in 4 possible combinations when detecting a class. First, there is prediction of the DNN on the one hand and the true reality on the other hand. Second, for a certain class, both can have the value of match and non-match. I.e., a match or a non-match can be predicted and at the same time a match or a non-match can be true. A predicted match that is also a true match is named true positive (TP), a predicted match that is a true non-match is a false positive (FP), a predicted non-match that is a true non-match is a true negative (TN), and a predicted non-match that is a true match is a false negative (FN). This can be displayed as a matrix. An example is given in Figure 21. (Szeliski 2022, p. 442)

	True matches	True non-matches	
Predicted matches	TP = 18	FP = 4	P' = 22
Predicted non-matches	FN = 2	TN = 76	N' = 78
	P = 20	N = 80	Total = 100

Figure 21. Example of matrix of possible combinations with TP, FP, TN and FN. (Szeliski 2022, p. 442)

Precision

In terms of the previously introduced matrix, precision is calculated using this formular:

$$Precision := \frac{TP}{TP + FP} = \frac{TP}{P'}$$

This means that precision is the ratio of true matches that are predicted as being matches to predicted matches. It leaves out predicted matches that are true non-matches. (Szeliski 2022, p. 443)

Recall

In contrast to precision, recall is the ratio of true matches that are predicted as being true out to true matches. This leaves out true matches that are predicted non-matches. (Szeliski 2022, p. 443)

Recall is calculated as follows (Szeliski 2022, p. 443):

$$Recall := \frac{TP}{TP + FN} = \frac{TP}{P}$$

Accuracy

Accuracy is the ratio of true and predicted matches, and true and predicted non-matches to all matches and non-matches (Szeliski 2022, p. 443):

$$Accuracy := \frac{TP + TN}{P + N} = \frac{TP + TN}{Total}$$

Intersect over Union (IoU)

IoU is a common metric used to evaluate the accuracy of bounding-boxes. There is always a ground-truth bounding-box as provided by the dataset and a bounding-box predicted by a DNN. As the name implies, IoU is the ratio of the intersect of both bounding-boxes to the union of both bounding-boxes as visualized in Figure 22. (Szeliski 2022, p. 380)

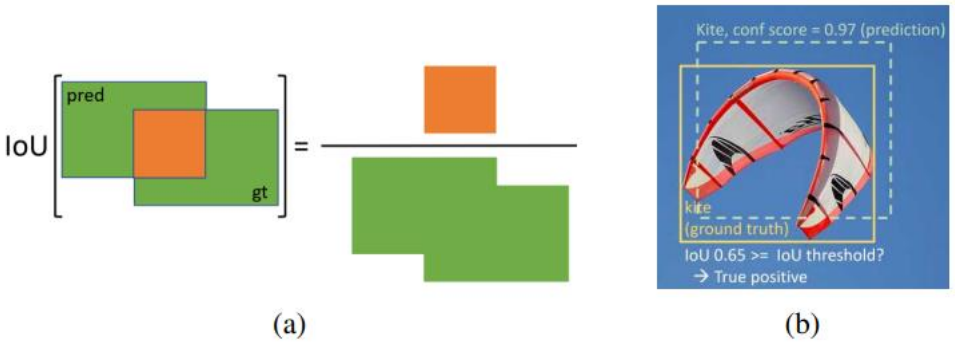


Figure 22. Schematic example of IoU (a) and example using a real image (b). (Szeliski 2022, p. 380)

This is the mathematical formular for calculation (Szeliski 2022, p. 380):

$$IoU := \frac{B_{pr} \cap B_{gt}}{B_{pr} \cup B_{gt}}$$

Precision-Recall-Curve

The precision-recall-curve is always calculated for a single class. It is calculated in the following way: First, a IoU value is chosen as threshold to decide if a bounding-box was properly detected. I.e., if the IoU of ground-truth and prediction is greater than the threshold, it is a true positive. Then, precision and recall are sampled for different confidence score of the predictions of the DNN. These are plotted as points and connected by a line as shown in Figure 23. (Szeliski 2022, p. 381)

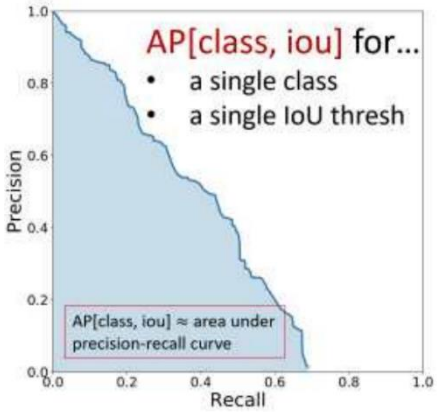


Figure 23. Precision-recall-curve for a single class and IoU threshold. For different confidence scores, precision and recall are calculated and points are plotted and connected. (Szeliski 2022, p. 381)

Average Precision (AP) and Mean Average Precision (mAP)

The average precision is calculated by computing the area under the precision-recall-curve and mAP is simply AP averaged over all classes. Because a IoU threshold must be chosen for the precision-recall-curve, AP and mAP also depend on this IoU threshold. Therefore, they are often provided with the IoU threshold used. E.g., mAP with an IoU threshold of 0.5 is often written as mAP@IoU=0.5. (Szeliski 2022, p. 381)

2.7. Deep Neural Networks for Object Detection and Instance Segmentation

This section covers ideas and versions of R-CNN and YOLO, both of which are popular model architectures for object detection using CNNs as base networks. It has a separate section for Mask R-CNN, which is a version of R-CNN that enables instance segmentation in addition to object detection in contrast to other versions.

2.7.1. R-CNN Family

The following approach at object detection is based on region proposals and convolutional neural networks. Thus, it is given the name R-CNN, which stands for Regions with CNN features (Girshick et al. 2014a). Fast-RCNN and Faster-RCNN are successors building on this method to improve speed of inference and training time (Bharati and Pramanik 2020, pp. 660–662).

R-CNN

The following statements on R-CNN are mainly based on Girshick et al. (2014a).

The architecture of R-CNN consists of three stages, which are displayed in Figure 24 (left): Region proposals, feature extraction and object category classifiers. After analysis revealed inaccurate bounding-boxes, the authors of the paper added bounding-box regression to the third stage of the architecture to increase accuracy.

First Stage: R-CNN starts with many probable candidates for objects and reduces the number later. To extract around 2000 of these candidates, so-called region proposals, from the input image, R-CNN utilizes Selective Search as discussed in section 2.4.2.

Second Stage: Each of the individual proposals is fed into a CNN for feature extraction. This is visualized in Figure 24 (right). The utilized CNN is AlexNet from Table 2 proposed by Krizhevsky et al. (2012). Since CNNs can only be applied to a fixed input vector, the region proposals are warped to a size of 227 × 227 pixels as seen in Figure 24 (left).

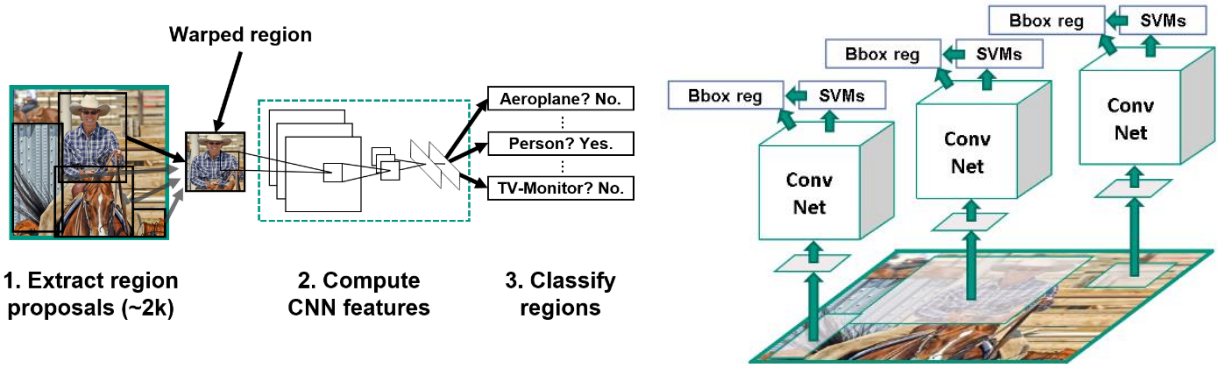


Figure 24. Original R-CNN architecture without bounding-box regression: First stage: Region proposal extraction, second stage: Convolutional feature computation and third stage: Classification of region proposals (left).
Based on Girshick et al. (2014a).
Architecture after error analysis with focus on feature extraction using a CNN, classification via SVMs and after both of those bounding-box regression on every region proposal (right).
Based on Bharati and Pramanik (2020, p. 660).

Third Stage: This stage is responsible for the removal of false region proposals (i.e., proposals that do not contain an object) and for classification and refinement of region proposals. The removal is handled by adding one further class to the set of possible classes, which is considered as background. All region proposals that are later assigned to this class will not be considered as objects and are thus removed from the results. For each possible output class (classes for objects and class for background), a linear SVM trained on that class calculates a score for the extracted features of each region. This equals to the one-versus-the-rest multiclass SVM approach. As result, each region proposal is assigned a class. Afterwards, a linear regression model similar to Felzenszwalb et al. (2010) trained specifically for the detected class of a region predicts an improved bounding-box using information of the feature vector.

Finally, per class but on all proposed regions, a greedy non-maximum suppression rejects all regions that have an IoU overlap above a learned threshold with another region having a higher score for the same class calculated by the SVM.

R-CNN outperforms previous results by 30% (Girshick et al. 2014a, p. 11).

However according to Girshick (2015, p. 1440), R-CNN has several drawbacks: Detection of objects is slow taking 47s per image on a GPU. The single stages have to be trained sequentially one after another and training requires a lot of memory and time.

Discussion of Figure 24 (right) and its Source

Figure 24 (right) is an altered and extended version of a figure provided in Bharati and Pramanik (2020, p. 660). Both figures are visible side-by-side in Figure 25.

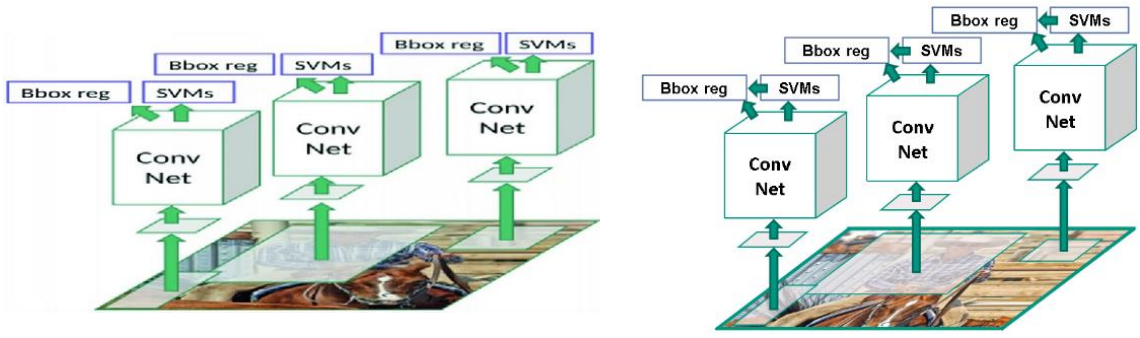


Figure 25. Original version (left) and modified version (right) of figure from Bharati and Pramanik (2020, p. 660) in comparison. They both visualized the architecture of R-CNN with a focus on feature extraction, classification via SVMs and bounding-box regression for each region proposal. The original shows arrows from feature extraction to classification and bounding-box regression while the modified version has an additional arrow pointing from classification to bounding-box regression.

The original figure displays the third stage of the R-CNN architecture with feature extraction performed by CNNs, classification using SVMs and bounding-box regression. Here, an individual arrow points from each

CNN to bounding-box regression and classification indicating that both regression and classification utilize features generated by CNNs. According to Girshick et al. (2014b) however, regression not only relies on features but also classification since it is trained for each individual class and outputs the regression corresponding to the detected class. Thus, not having an arrow point from classification to regression makes the figure incomplete. This is why another arrow has been introduced in the modified version.

Fast R-CNN

This section is based on Girshick (2015).

Fast R-CNN was developed to provide solutions for the drawbacks of R-CNN by modifying the model architecture. The result of the modifications can be seen in Figure 26.

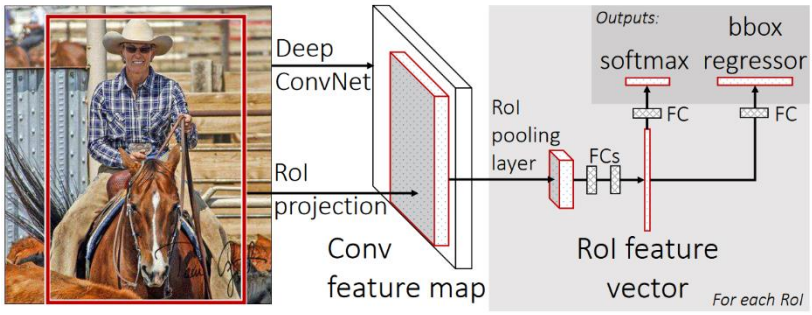


Figure 26. Fast R-CNN architecture: 1. CNN and RoI projection on convolutional feature map. 2. For each RoI: RoI pooling layer, fully connected layers, split into branches for classification using fully connected layer and softmax activation, and bounding-box regression.
(Girshick 2015)

Instead of running a CNN for every region proposal, a CNN processes the whole input image producing a feature map of the entire image. Therefore, the number of times the CNN must be calculated is reduced from around 2000 times to once per image. This is beneficial for speed of training and inference.

The algorithm for identification of region proposals is not adjusted and in the paper on Fast R-CNN considered as external dependency. Each region proposal is projected onto the feature map produced by the CNN resulting in a rectangular cut-out of the map. Such a cut-out is named region of interest (RoI).

On each RoI, a RoI pooling layer is applied. This layer converts the features of a rectangular region of arbitrary size into a feature map with a fixed, reduced size. It does this by dividing the input region into approximately equally sized sub-regions, where the number of sub-regions amounts to the size of the fixed, reduced output feature map. On each sub-region, the RoI pooling layer then applies max-pooling resulting in a singular output value for each of them. A simplified illustration of this process can be found in Figure 27. This layer is necessary to feed the information into the following fully connected layers, which can only process a fixed number of inputs. The result is a RoI feature vector for each RoI.

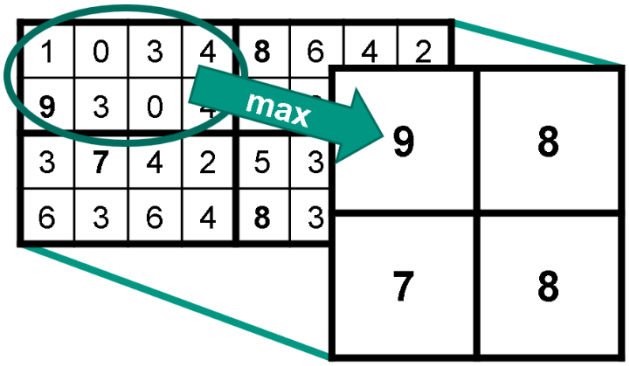


Figure 27. Simplified illustration of working principle of RoI pooling layer: Division of input region with size of 4 × 8 into 2 × 2 sub-regions with size of 2 × 4 and selection of the highest value as output value.

For the third and final stage, classification by an SVM is substituted with a fully connected layer with softmax activation predicting the probabilities for each of the object classes in addition to a background class for RoIs that do not represent an object. The bounding-box regression is done in parallel and calculated for each individual class per RoI. Then, the calculated bounding-box corresponding to the detected class is chosen.

To enable training of the whole network in one stage, a multi-task loss is calculated consisting of the loss function for classification as well as bounding-box regression. This simplifies training. In addition, as shown in the paper, it increases the mAP value. This can be explained by the fact, that both tasks (classification and regression) can have an influence on each other, and while isolated training cannot take advantage of this connection, the multi-loss approach can.

Faster R-CNN

The statements of this section are based on Ren et al. (2017).

While previous model architectures of this family (R-CNN and Fast R-CNN) utilized Selective Search, they are agnostic towards the actual algorithm used for generation of region proposals. This leaves space for optimization of that specific stage. Faster-RCNN introduces a region proposal network (RPN) that replaces the previously used algorithm and jointly makes use of the already existing CNN feature map of Fast

R-CNN. This eliminates the overhead added by selective search and keeps the computational overhead of the RPN itself light (overhead for region proposals is ~10 milliseconds).

The integration of the RPN into the existing model architecture of Fast R-CNN can be inspected in Figure 28. The feature map produced by the CNN was previously only used for regression and classification of region proposals. Now, the RPN uses this map as input. Its output, which are the suggested RoIs, are fed into the RoI pooling layer along with the feature map.

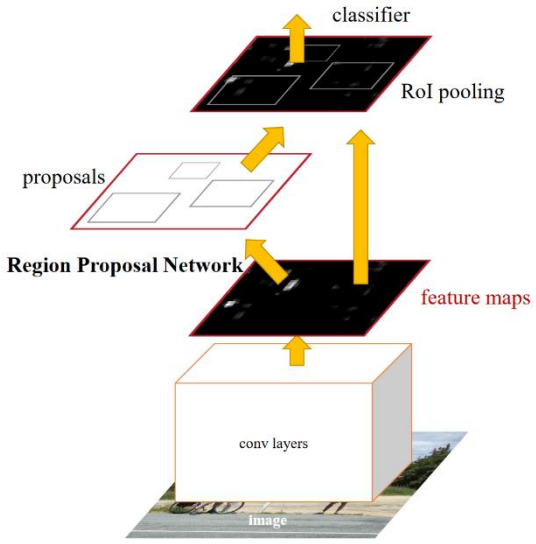


Figure 28. RPN utilizing convolutional feature map for region proposals. Thus, flow of information from the feature map onwards is split going into the RPN and the RoI pooling layer. Proposals generated by the RPN are also fed into the RoI pooling layer. (Ren et al. 2017)

Internally, the RPN produces RoIs by applying a sliding window on the convolutional feature map just like convolutional layers do and inputting each window to a network reducing the number of features. Then, this is fed into two fully connected layers, one for classification and one for bounding-box regression.

However, classification and regression are not done only once but multiple times per window. This is because for each window, there are multiple so-called anchor boxes defined acting as base areas with different scales and aspect ratios (i.e., three scales of 128, 256 and 512, and three aspect ratios of 1:1, 1:2 and 2:1). While these anchor boxes are not actually present in the network, they are used as base areas in the loss function for training of classification and regression. As a result, the layer for classification uses softmax activation to obtain a probability distribution of the two possible outcomes for each anchor box: object or no object. The layer for regression outputs four values for each anchor box acting as offsets to the left, right,

top and bottom of the base area. Therefore, every anchor box receives a score for objectiveness (whether it is an object or background) and bounding-box regression. This process is visualized in Figure 29.

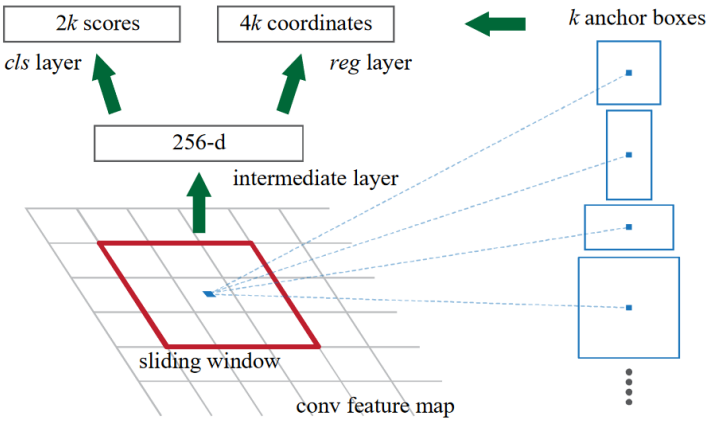


Figure 29. Window sliding over convolutional feature map producing offsets (left, right, top and bottom) and class scores (object or no object) based on anchor boxes as base areas. (Ren et al. 2017)

Training of the whole network including the RPN takes place in the following four steps. At first, the RPN is trained end-to-end using a pre-trained model as CNN with initialized weights. As second step, everything except the RPN is trained using again a pre-trained CNN. Here, the region proposals resulting from step one are used. As third step, the trained CNN from step two replaces the one previously used for the RPN and only the layers belonging to the RPN are fine-tuned. This merges the two networks meaning they now both use a common CNN. As final step, all layers of the CNN and RPN are frozen, and all remaining layers are fine-tuned.

For the steps involved in training of the RPN, the anchor boxes are an important part of the loss function and must have some definition of the desired output of the RPN for each of them. For regression, this is simply the difference between anchor box and bounding-box of the actual object if there is one. For object/background classification itself, the following rules are applied. All anchor boxes with the highest IoU with a bounding-box of an object are considered as objects. Additionally, all anchor boxes with an IoU greater than 0.7 with a bounding-box of an object are labeled as objects. All those where IoU is lower than 0.3 are seen as background. The remaining anchor boxes that do not meet any of the discussed criteria are ignored during training.

2.7.2. Mask R-CNN

The statements on Mask R-CNN are based on He et al. (2017).

Mask R-CNN is based on Faster R-CNN. It is a framework that does instance segmentation in addition to object detection. Thus, it provides bounding-boxes and segmentations masks together with a class label for

every object recognized in an image. Mask R-CNN can also be adapted to other tasks like human pose estimation.

According to the authors, Mask R-CNN outperforms all frameworks that participated in the 2016 COCO challenge. The challenge is a popular competition for image recognition frameworks and uses the Microsoft COCO dataset for training and evaluation (Ready for AI 2018). This challenge took place in the year before the publication of Mask R-CNN, which is why it did not participate in the challenge.

Mask R-CNN runs relatively fast with about 5 frames per second (FPS). Training it is also fast.

Model Architecture

Mask R-CNN is based on Faster R-CNN and does not apply major changes to the internal structure of it. Mask R-CNN mainly adds a new, third branch for outputting masks to the model architecture of Faster R-CNN.

As can be seen in Figure 30, Mask R-CNN has the same first stage as Faster R-CNN with a convolutional backbone producing a feature map and an RPN for region proposals.

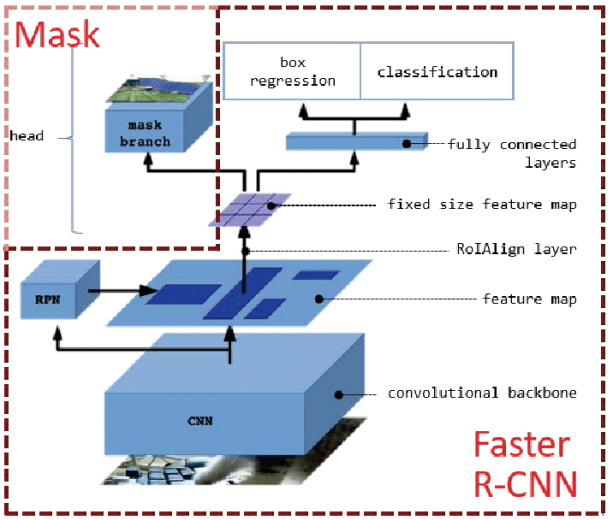


Figure 30. Mask R-CNN architecture divided in Faster R-CNN with RoI pooling layer replaced by RoIAlign layer, and additional mask branch.
(D. Schweitzer and R. Agrawal 2018)

However, the RoI pooling layer is modified. Previously, rounding was used to only get integer values for size and position of sub-sections. This results in misalignments, which are irrelevant for classification and bounding-box regression but have a negative impact on mask generation. For Faster R-CNN, this quantization was removed and replaced by bilinear interpolation for non-integer values. Thus, if a feature is to be sampled in between two integer positions of the feature map, it is not copied from one or the other location but interpolated between both of them best describing the input image at the actual location. The name of this updated layer is RoIAlign.

While Faster R-CNN only predicts class and bounding-box for each RoI in the last stage, Mask R-CNN at the same time independently calculates a binary mask for each class. This is done by a fully convolutional network.

Afterwards, the mask for the actual detected class is chosen as valid. This makes Mask R-CNN different to many other model architectures, for which classification and mask generation is coupled.

Additionally, He et al. experimented with a different backbone. This DNN is named feature pyramid network (FPN). Its structure processes input from large to small features and extracts feature maps at different steps of the process resulting in features of different scales. This FPN was found to perform much better than a conventional CNN as backbone. More on this type of network can be found in Lin et al. (2017).

2.7.3. YOLO Family

For many object detection model architectures, the process of detection is split up into multiple components, that must be trained separately. Model architectures of the YOLO family view the whole process as one single regression problem. They do not require multiple runs of individual components but only have one look at the input image and base their calculations on that. Hence, the name YOLO (You Only Look Once). (Redmon et al. 2016, p. 779)

There exist multiple model architectures building upon the first version of YOLO and each other. The most popular and relevant of them being YOLOv1, YOLOv2, YOLO9000, YOLOv3, YOLOv4, YOLOv5, YOLOv6 and YOLOv7. In general, YOLO is mostly known for its high detection speed allowing for real-time detection. The following sections will discuss the working principles and advantages and disadvantages of each of them in chronological order. (Ansari 2020, pp. 238–247; Jiang et al. 2022, p. 1069)

YOLOv1

Statements on YOLOv1 are mainly based on Redmon et al. (2016). The entire process of detection is illustrated in Figure 31.

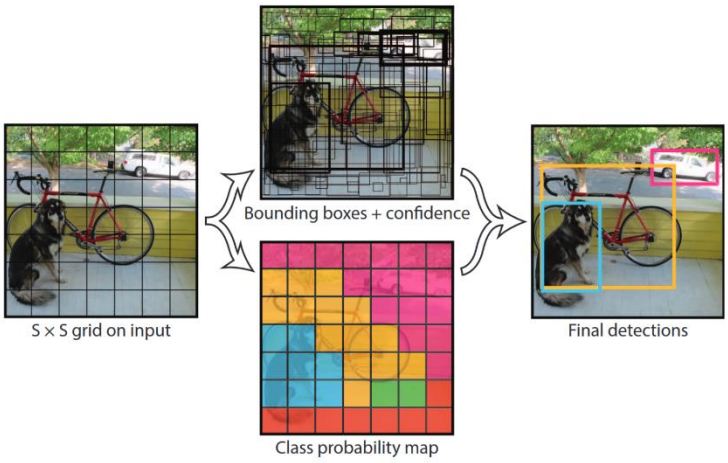


Figure 31. Detection pipeline of YOLOv1: 1. Division of input into grid, 2. For each cell: Simultaneous class prediction and bounding-box prediction, 3. Merging of both predictions into final detection results.
(Redmon et al. 2016, p. 780)

At first, YOLOv1 divides the input image into a grid. Each of the grid's cells is responsible for the detection of objects having their center in that cell. For each cell, multiple bounding-boxes and corresponding confidence scores are predicted (I.e., prediction for one bounding-box consist of a height, a width, a horizontal and a vertical position as well as a confidence score telling the likelihood of this bounding-box containing an object). Independently and at the same time, the neural network calculates the most likely class for each cell. A combination of these predictions is used to determine whether the cell contains a real object and what its bounding-box looks like. All these calculations are performed by a large CNN composed of 24 convolutional layers as well as 2 fully connected layers. The network can detect 20 different classes.

The main limitation of this first version of YOLO is weak detection for larger groups of small objects. This is caused by the grid-based approach only allowing for one class and object to be detected per cell. For the same reason, YOLOv1 cannot detect object of different classes that are close to each other or have overlapping prediction. It is forced to decide for one class. Another issue is inaccurate prediction of bounding-boxes in cases where the shape of a detected object of a certain class deviates from the objects of the same class seen during training.

YOLOv2

The following statements are based on Redmon and Farhadi (2017) in addition to (Ansari 2020, pp. 241–244). The second version of YOLO introduces various improvements to eliminate the shortcoming of the first version.

Most notably, the part of YOLOv2 responsible for classification is fine-tuned standalone using input with higher resolution because this higher resolution is used during detection anyway. Training of the classification network was previously performed using a lower resolution requiring the network to adapt later.

Furthermore, all fully connected layers were removed and anchor boxes like the ones used for Faster R-CNN were introduced. Therefore, multiple objects and classes can now be detected for every single cell (confidence score is not calculated per cell but per anchor box). While this leads to a slight decrease of precision, recall is improved significantly, and many more objects can be detected in a single input image.

To increase dataset size for learning, YOLOv2 uses datasets for object detection and datasets for image classification. This is advantageous because classification datasets are more abundant and offer far more classes as labels. However, YOLOv2 does not make use of the additional classes and only uses datapoints for the same 20 classes offered by the object detection dataset. To be trained on these different types of training data, the following process has been implemented for learning. Whenever a datapoint offers bounding-boxes as information, it is used to train the network end-to-end. However, if a datapoint only offers a class, the network uses this datapoint to further train only the classification part of the network. For such cases, all other parts of the network are frozen.

A speed increase is also achieved by exchanging the convolutional backbone for a smaller one (Darknet-19) requiring less computations per run. At the same time, this gave a small boost to detection accuracy.

YOLO9000

This section about YOLO9000 is based on Redmon and Farhadi (2017).YOLO9000 is tightly coupled with the advancements of YOLOv2 and was proposed in a single paper with YOLOv2. While YOLOv2 still detects only 20 different classes, YOLO9000 is able to detect 9,000 classes.

This is achieved by not only training the network on an object detection dataset and the part of a classification dataset that has the same classes as labels but the whole classification dataset with all its classes. As mentioned in the section about YOLOv2, classification datasets usually offer far more training samples and many more classes than object detection datasets do. When mixing datasets with different classes, some classes are not mutually exclusive or represent a part of another class. To efficiently train YOLO9000 on all these classes, they are structured in a hierarchical tree as shown in Figure 32.

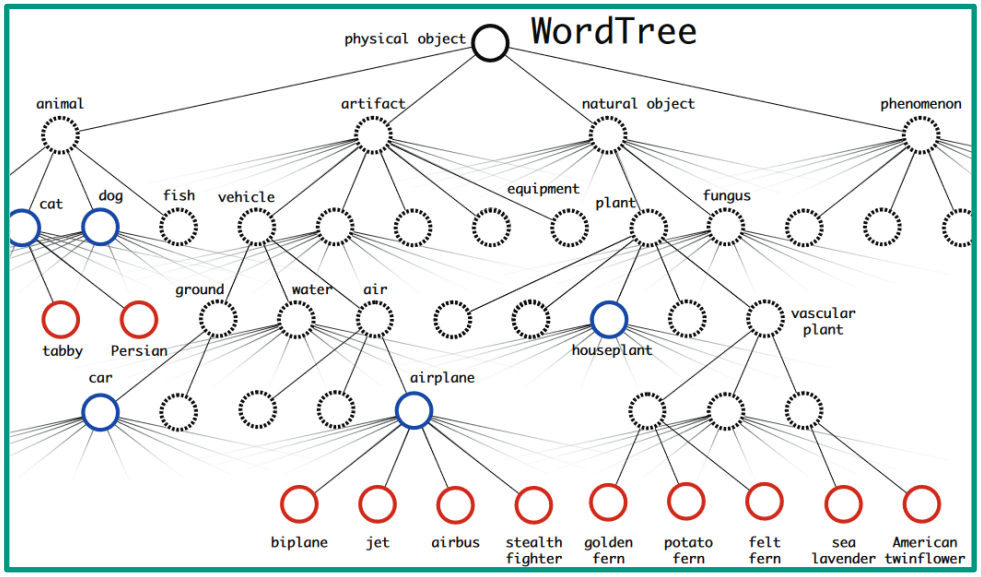


Figure 32. Cutout of hierarchical tree structure assigning low-level classes to higher-level classes.
(Redmon and Farhadi 2017, p. 6524)

As a result of this training process, the network did not have any training data containing bounding-boxes for many of the classes. Nonetheless, while some of these classes have very bad detection results, some of them are being detected quite well. According to the authors, the reason for this may be that some classes have related classes where bounding-boxes are known and can be derived from. However, other classes like clothes are not at all similar to anything seen in the object detection dataset.

YOLOv3

This version only introduces minor changes. The following explanations are based on the paper on YOLOv3 from Redmon and Farhadi (2018).

Previously, the last layer of all YOLO versions used softmax as activation function to get class probabilities. This works well for mutually exclusive classes. Since the last version introduced a hierarchical structure of classes, that condition is not given anymore. Now, it is possible that a detection result fits well in multiple classes (E.g., an object can be a person and a woman at the same time.). Therefore, the activation function for the last layer calculates a logistic score for each class that is independent from other classes.

Another improvement is the prediction of bounding-boxes at different scales. There is not one feature map on which bounding-box detection is performed but 3 of them. They are taken from different positions in the network. Because of the downscaling of the input during multiple convolutional layers, this means that bounding-box map to different sizes in the original input image.

Finally, YOLOv3 uses a new convolutional backbone: Darknet-53. This CNN is larger and has shortcut connections meaning some outputs skip intermediate layers and serve as inputs for layers further down the layer stack.

All of this makes YOLOv3 significantly faster and more accurate than the previous versions of YOLO. When compared to other network architectures, it is still faster but not always as accurate.

YOLOv4

At the time of its release, YOLOv4 was superior to other model architectures in speed and accuracy. This was achieved by implementing many minor changes. First of all, yet again a new backbone was chosen (CSPDarknet53). To improve training, different data augmentation techniques were applied. Activation functions were replaced with a new one named Mish (Misra 2019). There are other changes. However, an explanation of all of the would go beyond the scope of this thesis. (Bochkovskiy et al. 2020)

YOLOv5

No paper was published for the fifth version of YOLO and there is not one final and official implementation. Therefore, the naming is somewhat controversial. Information about YOLOv5 is taken from Jiang et al. (2022, p. 1069).

YOLOv5 is the first version implemented using PyTorch as framework. This offers some usability advancements compared to Darknet. Yet again, it uses a new activation function named Hardswish (Howard et al. 2019, pp. 1317–1318) and applies data augmentation and enhancement (i.e., scaling, color space adjustment etc.).

Accuracy of YOLOv5 is comparable to that of YOLOv4 while performance is moderately better.

YOLOv6 and YOLOv7

The newest versions of the YOLO family are version six and version seven. Both versions are a group of multiple model architectures. Their performance compared to each other in addition to other model architectures is shown in Figure 33. The fact that this figure including both YOLOv6 and YOLOv7 is provided in the paper on YOLOv6 suggests that work on YOLOv6 started first, but the release happened after YOLOv7 was already published. Therefore, explanations will start with YOLOv7 followed by YOLOv6.

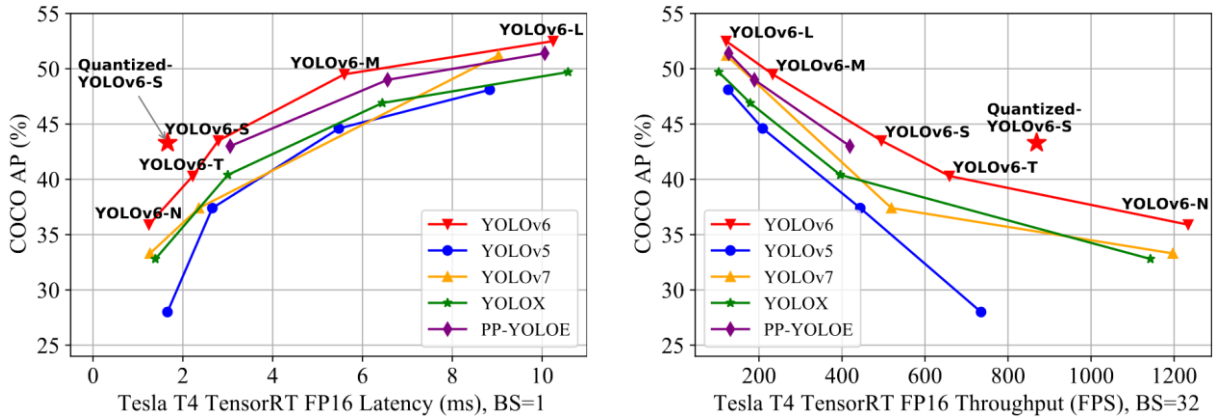


Figure 33. Comparison of version five, six and seven of YOLO and additional model architectures. The graph on the left plots AP for the COCO dataset as y-axis and latency as x-axis. The graph on the right uses the same y-axis but FPS as x-axis. In summary, YOLOv7 outperforms other model architectures for some cases. Except for YOLOv6 which constantly outperforms all other model architectures including YOLOv7.
(Li et al. 2022, p. 1)

YOLOv7 uses a new model architecture based on ELAN (efficient layer aggregation network) and named efficient ELAN. It implements additional minor changes for the training process. However, these improvements were not invented by the authors of YOLOv7 but adopted from other sources. (Wang et al. 2022)

The main thought behind the invention of model architectures grouped together under the name of YOLOv6 were industrial use cases which have diverse requirements when it comes to speed and accuracy. The models use different backbones suited for the varying requirements. Among other improvements, some of the models use quantization. This is the concept of decreasing precision of continuous values by replacing them with some form of discrete values (Rokh et al. 2022, pp. 2–7). This can make models much more suitable for certain hardware used in the industry. (Li et al. 2022)

3. Related Work

This chapter summarizes research papers with similar objectives to this thesis. I.e., papers about the detection of elements of GUIs. Furthermore, it points out similarities and differences.

3.1. "Construction of GUI Elements Recognition Model for AI Testing based on Deep Learning"

C. Zhang et al. (2021) describe a possible way to recognize GUI elements in screenshots taken on a computer running the Windows 10 operating system. The whole dataset consists of 3,000 images, all of which have been labelled manually. GUI elements are assigned to 10 different classes.

After using this dataset for transfer learning on YOLOv3 and Mask R-CNN, the paper compares the mAP achieved using YOLOv3 and Mask R-CNN. Mask R-CNN slightly outperforms YOLOv3 with a mAP of 99.985% compared to 98.513% and yields segmentation masks while YOLOv3 only outputs bounding-boxes and classes. An example can be seen in Figure 34. Judging from the example, bounding-box and mask detection is quite accurate. Additionally, the paper compares the results after training on a third, two thirds and the entire dataset. For most classes, AP increased significantly with the number of images used for training.

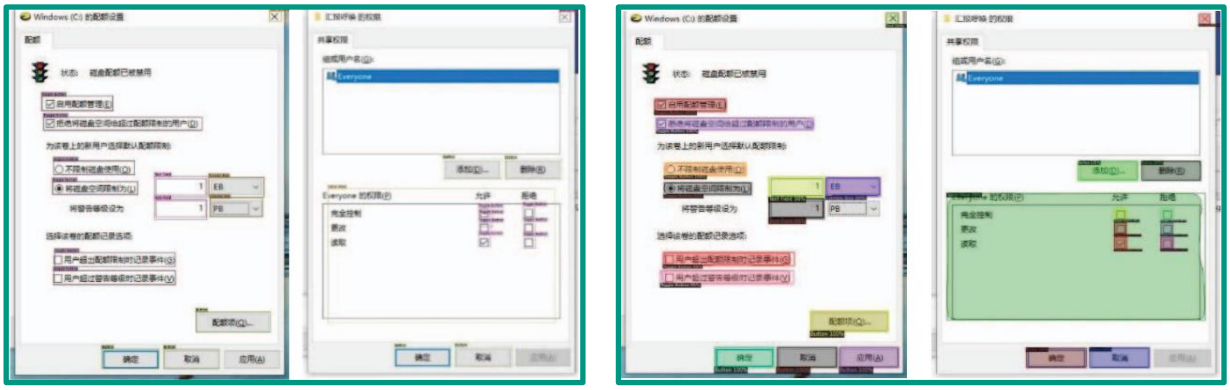


Figure 34. Example of results using YOLOv3 (left) and Mask R-CNN (right).
(C. Zhang et al. 2021)

The objective of this paper is to provide GUI element recognition usable for automated GUI testing. For this, it uses YOLOv3 and Mask-RCNN. Insofar, it is quite similar to the goal of this thesis. However, it uses a desktop operating system (Windows 10) while this thesis uses a mobile operating system (Android). This results in different GUI elements to look for. Additionally, the datasets used in the paper consists of screenshots. This thesis uses camera images taken from a screen.

While the objective and the way of labelling the dataset differs, the methodology of using deep learning and in particular Mask R-CNN to detect and segment elements is a similarity to the approach of this thesis.

3.2. “Detection and Segmentation of Graphical Elements on GUIs for Mobile Apps Based on Deep Learning”

Hu et al. (2020) use deep learning for the detection and segmentation of GUI elements in mobile applications. The motivation behind their research is yet again automated testing of GUIs.

The dataset of this paper differs from the one of the previously discussed paper. It is composed of screenshots from Google Play, Huawei AppGallery and part of the Rico dataset (Deka et al. 2017), which is a collection of 72,219 screenshots of mobile applications. The dataset consists of 2,100 screenshots and each GUI element is assigned to one of 8 classes. The labeling was done manually. In total, 42,156 GUI elements have been labeled. As model architecture, this paper uses Mask R-CNN. The achieved mAP is 98%. Appendix A shows visual examples of the results. Here, some bounding-boxes and masks are very accurate while others are shifted or distorted and do not match the true position of the element. Masks are always rectangular suggesting that only boxes were used for labeling of the dataset independently of the actual shape of a GUI element.

In comparison to the previous paper, the paper of Hu et al. (2020) has more similarities with this thesis, which is trained on android applications and uses the Mask R-CNN framework. Up to this point, there is no difference between them. This thesis uses a camera image of the mobile device’s screen and automates dataset generation and labeling. This distinguishes it from the paper of Hu et al. (2020), where screenshots are used and labels are acquired by manual annotation.

3.3. “Object detection for graphical user interface: old fashioned or deep learning or a combination?”

In their paper, Chen et al. (2020) study existing attempts at GUI detection. In addition, they propose a new approach themselves. They do not have a specific goal building upon GUI detection but describe GUI testing and automation as possible use cases among others.

The dataset used for all their learning and testing consists of 50,524 screenshots of 8,018 android applications. In total, these contain 923,404 GUI elements.

Chen et al. compare deep learning approaches and old-fashioned approaches using classical CV algorithms like edge detection for GUI detection. They conclude that old-fashioned approaches are inferior to deep learning approaches. Old-fashioned approaches work acceptable for simplistic GUIs but get worse with increased complexity. Of all deep learning approaches, Faster R-CNN performed the best. Still, all of these have some problems with accurate bounding-box prediction.

The newly proposed approach by Chen et al. differentiates between GUI elements with and without text and provides separate detection pipelines for them. Text elements are detected using EAST, which stands for efficient and accurate scene text detection pipeline (Zhou et al. 2017, p. 5553). It is used for text detection in natural images (Zhou et al. 2017, p. 5551). For GUI elements without text, the authors use a hybrid approach with both old-fashioned and deep learning parts. Region detection is performed by a top-down classical algorithm. Region classification is done by a pretrained ResNet fine-tuned for GUI elements. This approach yields better performance than Fast R-CNN. Examples of detection for non-text GUI elements are given in Figure 35. In the examples, bounding-boxes appear to be very accurate. However, the approach

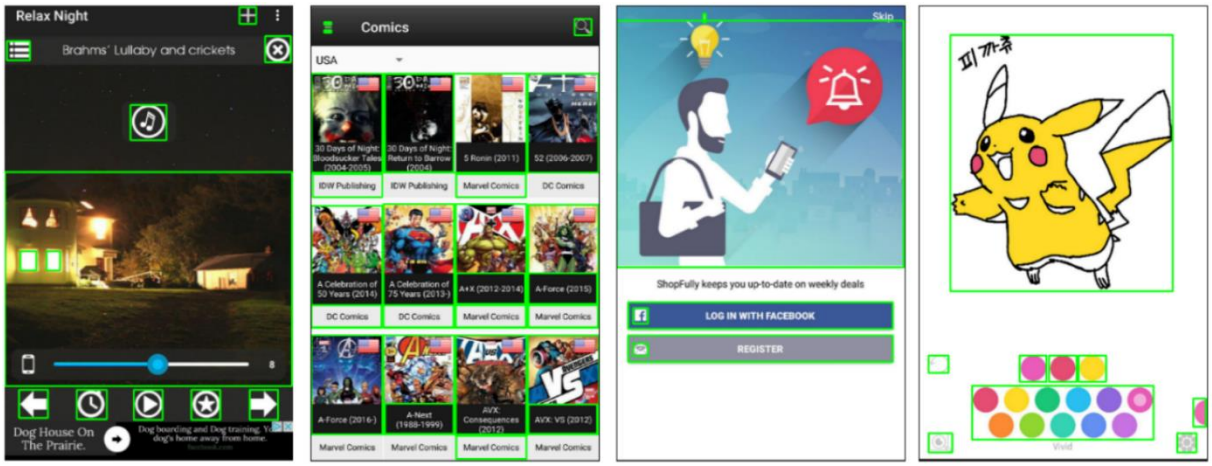


Figure 35. Four examples of non-text GUI detection using the new approach suggested by the authors of the paper. Edges of bounding-boxes accurately match edges of elements. Some elements are merged together while others are split into multiple bounding-boxes.
(Chen et al. 2020, p. 1211)

sometimes detects multiple GUI elements as one or parts of one GUI element as individual units.

The main differences between this thesis and the paper from Chen et al. are as follows. This thesis uses an automatically generated and labeled dataset and masks for detection whereas the paper uses manual labeling and bounding-boxes only. Additionally, this thesis uses a purely deep learning based approach.

4. Methodology

Chapter 1.2 Objective declares the research question of this thesis as follows: It shall be determined whether a neural network trained on an automatically generated and labeled dataset captured by camera provides any advancements in detection of interactive elements of GUIs compared to previous approaches. This chapter describes the theoretical approach taken to answer this question empirically.

For this, an experimental setup as visualized in Figure 36 is created and later evaluated. The setup consists of multiple components interacting with each other. A modular design based on client-server architecture offers the possibility to modify the system and fit it to desired use cases in the future.

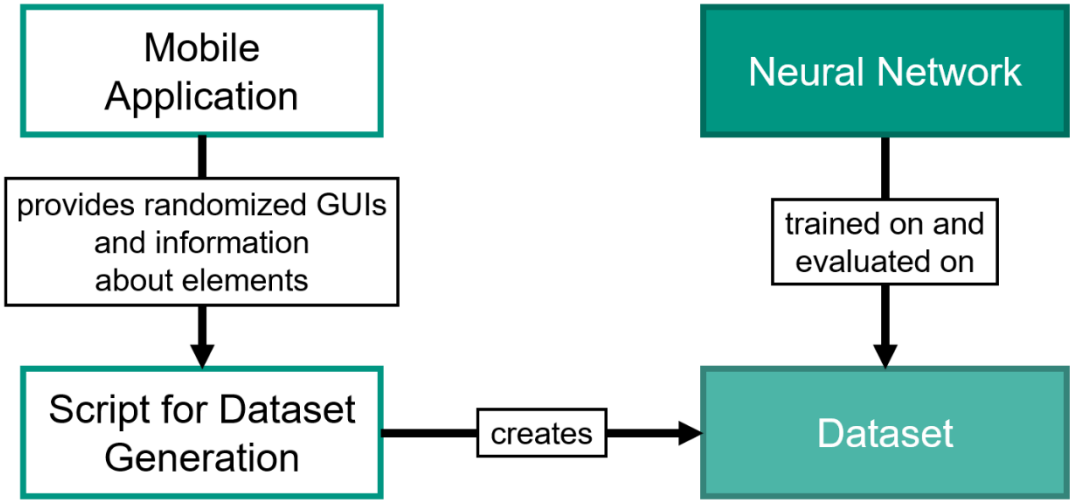


Figure 36. Individual components and their interaction with each other. A mobile application provides random GUIs and information about the interactive elements for labeling. The GUIs are captured by a script that also collects the information about elements. From this, the script creates a dataset. This can be used for training and evaluation of a neural network.

To train a neural network, a dataset is needed. As discussed in the Chapter 1.1 Motivation, manual labeling may have some shortcomings that can be addressed by automatic labeling. Automatic labeling is only possible if the source code of the application is accessible. Therefore, a mobile application is programmed. This makes it possible to access all data relevant for the generation and labeling of the dataset. The mobile application displays randomized GUIs and has some way to provide information about interactive elements visible in the GUIs.

This is used by a script to generate the dataset. The script creates a datapoint by performing the following tasks: It captures an image of a mobile device's screen displaying the randomized GUI of the mobile application. Additionally, it communicates with the mobile application to gather class labels and information about bounding-boxes and masks of interactive elements contained in the GUI. For bounding-boxes and masks, two possible ways of gathering that information are as follows: Bounding-boxes and masks can be collected by transmitting their position and shape in abstract form as text via a client-server connection, or

by altering the GUI and capturing an image of this. For the second approach, the mobile application paints the entire screen in black except for a single interactive element painted in white. The script for dataset collection captures an image of this view and converts it into a binary mask. This is repeated for every interactive element present. After the collection of all relevant information for a datapoint, the mobile application creates a newly randomized GUI, and the process is repeated until the whole dataset is populated with datapoints.

Finally, a neural network is trained on this dataset. The dataset is split into two parts: A training dataset and a test dataset. The test dataset is kept from the neural network during training. After training, it is used to evaluate the performance.

There are certain constraints and requirements for the individual components of the setup, which must be respected during implementation to produce meaningful results:

Since a neural network can only learn patterns present in the given data, the mobile application providing the data must generate GUIs, that are representative for real-world mobile applications. To achieve this goal, the randomized GUIs displayed should include interactive elements with varying appearances similar to those of real-world mobile applications and also as many different styles as possible for every type of element. One GUI should contain many interactive elements because this results in more training data using less storage space and memory. Additionally, it must offer an interface to allow access to the following data: Classes of interactive elements and accurate information about their shape and position. It must also offer an interface on which the mobile application listens to commands of the dataset collection script. A possible command would be to demand a new randomized GUI.

The script creating the dataset must capture GUIs and the provided information about the interactive elements and save all of it in a format that can be understood by the learning algorithm of the neural network.

The dataset must have an appropriate size. It seems reasonable to use a dataset size that is similar to the one used in related papers to make them comparable and since they prove that this size can yield useful results. Since the third related paper uses a huge dataset requiring much more processing power, the aim for dataset size in this paper is to come close the size for the first and second related paper. This requires about 2000-3000 images. Furthermore, the format of the dataset must be capable to store information about position and pixel-accurate shape of interactive elements. This could either be the geometrical shape or a binary pixel mask of the GUI image.

The neural network trained on the resulting dataset should be capable of providing good accuracy on a per-pixel basis while still running decently fast (I.e., perform detection in well under a second). Otherwise, it would not be suitable as part for an efficient (robotic) testing framework.

For this experiment to answer the research question, it is evaluated and compared to related work as a final step. The testing part of the dataset is used to deduce numerical metrics as well as provide visual examples to judge accuracy. To gather more intel about its accuracy on real-world mobile applications, the neural network is also tested on GUI images of real-world mobile applications. This can only lead to a visual inspection of a small number of visual examples, since an appropriately-sized dataset for numerical metrics with per-pixel segmentation does not exist and creating one is not scope of this thesis. Apart from that, labeling of this dataset would have to be done manually. As discussed, this may produce inaccuracies which possibly invalidate numerical metrics derived from evaluation on such a dataset.

5. Implementation

This chapter explains implementation and practical parts of this thesis and their underlying thoughts and decisions. It follows the order of Chapter 4 for explanation of the components.

Before diving into the implementation of each of the components, it is useful to know about the communications between the individual components. After this, the mobile application for randomized GUIs and the scripts for dataset collection are explained. Then, implementation details of the DNN are discussed and finally the entire practical workflow for dataset collection and training of the DNN is shown.

Apart from the training and test implementation, the DNN needs an interface that lets other applications access the inference results produced. This has been implemented but is not described as part of this thesis because it is not required to evaluate the performance of the DNN and therefore not part of the research objectives.

For the understanding of communications and the components architecture, it must be explained how position and shape of interactive elements are supposed to be transmitted to the dataset collection script. In chapter 4, two ways of transmitting position and shape were presented. It can be done either by sending position and shape as textual information or by painting an individual element of the GUI white and everything else black, capturing an image of this, and processing this image to acquire a binary mask. The former approach can only provide positional information in relation to the screen. Because GUIs are not captured as screenshots but via a camera, this would require access and further processing of the information in addition to knowledge about the position of the screen in the captured image. These issues can be evaded by using the latter approach. If processed properly, the resulting binary mask after processing should accurately represent position and shape of an element in relation to the captured image of the GUI. There is another advantage to this approach. If an element is occluded by another one, the application itself would calculate the part of the element that is still visible and only display this part in white. Otherwise, this would also require additional computations. For this approach, images of the GUI itself and of additional screens each highlighting an individual element are taken one after another.

Back to the communication. The mobile application provides a server to be accessed by the dataset collection script. This connection allows the script to get information about classes and request state changes of the mobile application (I.e., new random GUI or black and white mask for an element to be captured by camera.). The script outputs a dataset in the form of a directory. This directory is used for training of the DNN. Each datapoint consists of a GUI image, multiple mask images and a data file with class names for each element. An outline of all communications in order of their execution is illustrated in Figure 37.

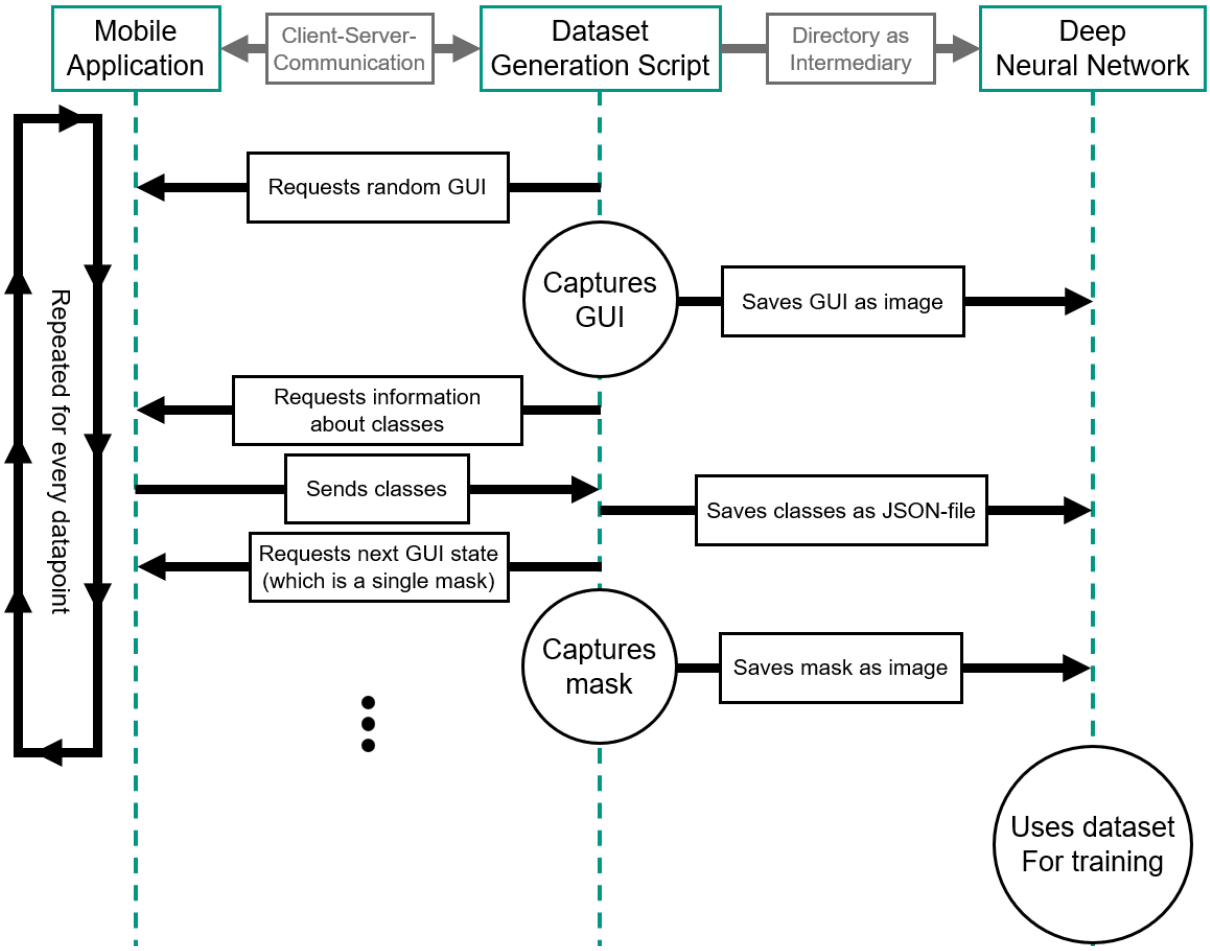


Figure 37. Outline of communications between components in order of execution. The dataset collection script serves as client accessing the mobile application acting as server. The script orchestrates collection of each datapoint and saves all information in a directory. For a single datapoint, it starts by requesting a random GUI. Then, it asks for information about classes of elements. For every element in the GUI, it requests, captures, and saved the mask. Afterwards, the DNN uses this directory as dataset for training.

5.1. Android Application

As discussed, the main goal for the mobile application is to obtain randomized GUIs where interactive elements are representative for those of most real-world mobile applications. Material Design is developed by Google, used as standard for android applications and adopted by many other developers. Because android itself has the highest market share for mobile operating systems, using Material Design 3 as base for GUI design seems logical. One could use multiple design frameworks, but Material Design 3 alone provides a good compromise between the effort required for implementation and requirements of representative interactive elements.

With Material Design as design framework, the mobile application can be implemented as a native android application, flutter application or web application. Because of the author's previous knowledge and familiarity with native android applications and no obvious downsides of using native android as compared to

flutter or web for the task at hand (cross-platform capabilities are not needed for generation of a dataset), a native android application is chosen for implementation. As result, the code of the mobile application is written using Kotlin as programming language. GUIs are programmed using Android Compose where elements of a GUI are expressed as functions annotated with "@composable" in source code.

The architecture of the android application is as follows. A file named *MainActivity* servers as entry point and makes use of other files and classes to orchestrate GUI creation, data management and communication. Class *State* keeps track of information required for labeling, for rendering of the GUI itself, and for black and white masks for every element. Class *Registry* saves class names of all elements. A file named *RandomElements* contains all functions for generation of random elements. File *Probabilities* only consists of probability values used for the generation of random elements. File *Utils* provides helper functions. All these classes and files are shown in Figure 38. There are some additional files that mainly serve a single purpose for files and classes already discussed. Because of this, they are excluded in the figure and will be explained as part of sections about other files and classes.

Figure 38. Most relevant classes and files of the android application for randomized GUIs: MainActivity, State, Registry, Utils, Connection, Random Elements, and Probabilities.

File MainActivity

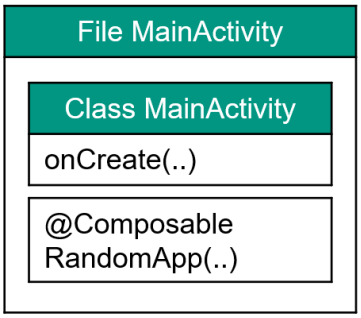


Figure 39. Contents of file MainActivity: Class MainActivity and composable function RandomApp.

The contents of this file are shown in Figure 39. When the android application is started, it executes the method *onCreate(..)* of the class *MainActivity*. Since all we care about is the application itself and we cannot make use of a displayed notification bar at the top or a navigation menu at the bottom of the screen (I.e., there is no way to produce masks for these and therefore they cannot be used for learning.), this function hides them. Additionally, it makes sure that the screen never turns off after a certain time of inactivity

because this would cause the screen to go black during collection of the dataset. Finally, the method calls the composable function *RandomApp(..)* that is responsible for displaying the GUI.

This function delegates the creation of most of the randomized elements to composable functions in file *RandomComponents*. However, there are some things this function takes care of by itself.

The function creates an instance of class *State* to keep track of the current state. It overrides a function of this instance that replaces the instance with a copy of it whenever a change in state occurs. This is necessary because changing the variable triggers recalculation and rerendering of the GUI. Only then are all GUI elements automatically updated to represent the new state. More explanation on this will follow in the section about this class.

Based on some probability defined in file *Probabilities*, the function sets the theme of the GUI as light or dark mode. In combination with this, the function uses function *randomColorSchemes()* defined in an additional file named *Color* to apply a more or less random color schemes to the GUI. This function randomly selects one out of 10 color schemes, which were created using the Material Theme Builder (Material Theme Builder n.d.). Here, primary colors were chosen, and all other colors generated by the builder. The colors of the exported themes were then added to the file *Color*.

Furthermore, the function *RandomApp(..)* adds a basic layout to the GUI in which randomized components can be placed. This layout offers the option for a *TopAppBar*, a *FloatingActionButton*, a *BottomBar* for navigation, and content filling the center of the screen. The content is populated with two nested columns that add 13 random elements to the GUI using the composable function *RandomIE(..)*. The number 13 was chosen because it ensures that most of the screen space is used for random elements, but no overflow is occurring. Additionally, the columns can display some buttons used for debug purposes if a variable *debug* is set. Together, they randomize the look by switching between left, right and centered alignment of content. Because the contained elements have different widths, this results in diverse positioning of elements. This is supposed to help make the GUI representative for real-world applications.

Finally, function *RandomApp(..)* creates an instance of class *Connection* to open a channel for communication with the dataset collection script after everything is set up. If there already is an active instance, no new instance is created but its reference to the application's state is updated. This is because for every datapoint a new instance of class *State* is created and used.

Class State

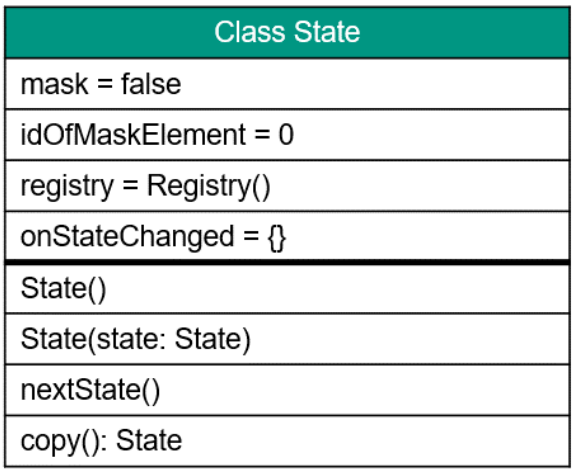


Figure 40. Content of class State. Shows variables(top) and methods(bottom).

Class *State* keeps track of the current state of the application and enables state changes. For this, it uses various variables and functions as displayed in Figure 40. GUI elements change their appearance based on the information provided by this class.

Variable *mask* decides whether the GUI itself is shown or the mask for a one of the elements (I.e., this element is painted white and everything else black.). Variable *idOfMaskElement* contains the ID of the element whose mask is currently shown. Variable *Registry* is used to assign IDs to all elements and store their corresponding classes. More on this in the section about the Class *Registry*. Variable *onStateChanged* has already been touched on in the section about file *MainActivity*. It stores a function to be executed as callback whenever a state change occurs.

The first constructor *State()* is used to create new instances of the class. The second constructor State(..) serves as copy constructor. I.e., it is used to create a new instance of class *State* with identical variables to the instance provided as argument.

Method *nextState()* contains logic to switch to the state that is supposed to follow the current state. First, it switches from GUI to masks. Then, it increments the ID of the element whose mask is displayed. In addition, it triggers the callback function saved in *onStateChanged()* that replaces the current instance with an identical one newly created. This is necessary because only then can GUI elements detect this change and adjust their appearance according to the new state.

Method *copy()* simply returns an identical instance to the one the function is called on. For this, it uses the copy constructor.

Class Registry

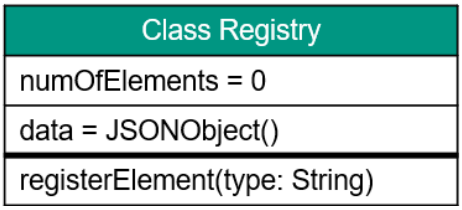


Figure 41. Variables and methods of class Registry.

Contents of class *Registry* are shown in Figure 41. Variable *numOfElements* tracks the total number of interactive elements already created. Variable *data* uses JSON format to store IDs of elements together with their class.

Method *registerElement(..)* adds ID and class to variable *data* and returns the ID, which is also stored inside each element for identification during state changes.

File RandomElements

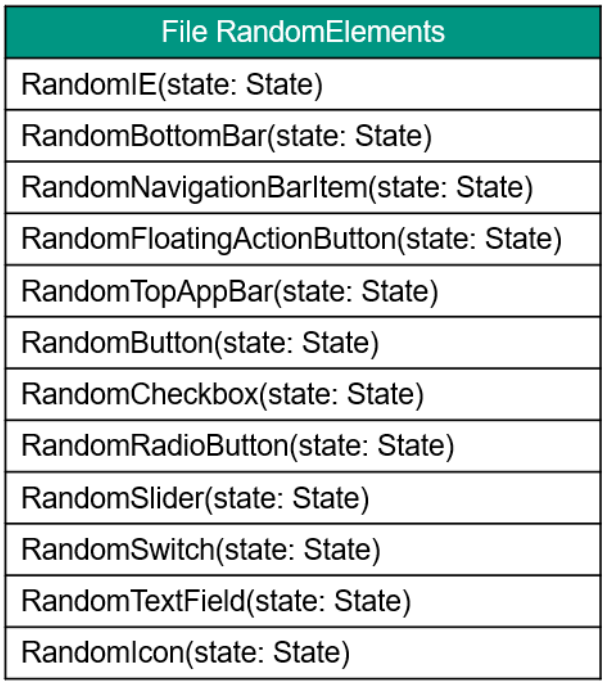


Figure 42. All composable functions of file RandomElements.

As visible in Figure 42, file *RandomElements* provides composable functions for all interactive elements to be displayed. All of them take the applications state as argument because this way they can listen to state changes and automatically adapt their appearance accordingly.

The first function *RandomIE(..)* serves as a wrapper for other interactive elements. It randomly selects one of the possible elements and displays it. It does this according to a probability distribution provided in file *Probabilities*.

All other composable functions mostly work in the same manner. If the application's state requires the GUI to be displayed, they show their corresponding element. This is the first possibility. Usually, the appearance is somewhat randomized. E.g., a button is either displayed as simple text, in a filled box or an outlined box. If the state requires the mask of an element to be displayed, there are two more possible cases that can occur. Whenever the ID saved as part of the element matches the ID of the active mask element saved in the application's state, the shape of the element is filled with white. Otherwise, it is filled with black. Some of the functions require text that can be displayed. This functionality is provided by the function *randomText(..)* of file *Utils* and will be discussed further in the section of that file.

Class Connection

Class Connection
port = 1280
server = null
client = null
input = null
output = null
state = null
activity = null
Connection(state: State, activity: Activity)
listen()
sendResponse(header: Any, body: Any?)
establishConnection()
closeClientConnection()
close()
isClosed(): Boolean
setState(state: State)

Figure 43. Variables and methods of class Connection.

Class *Connection* is responsible for communication with the dataset collection script. As discussed before, it does so by providing a server the script can connect to as client. It propagates requests to other parts of the application and responds with requested information. Internals of class Connection are displayed in Figure 43.

The variables *port*, *server*, *client*, *input*, and *output* are used for operation of the server. Variables *state* and *activity* are needed for the propagation of requests to their destination.

The constructor starts a server on the specified port. Then, it launches the method *listen()*. This method contains most of the logic. In an infinite loop, it does the following. At first, it calls the method *establishConnection()* that waits for a client to connect and establishes communication channels once a client has connected. Then, it reads the incoming request and processes it. JSON is used as format for all communications. Requests and Responses always have a header and a body part. For responses, the helper method *sendResponse(..)*, converts header and potentially body into the required format. The possible requests and their responses are shown in Figure 44.

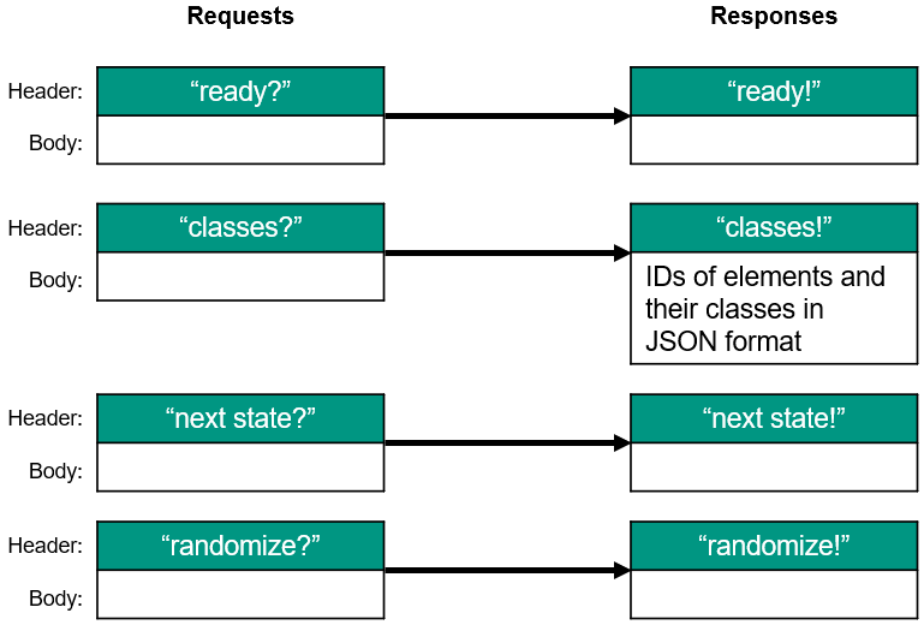


Figure 44. Possible requests and their responses with header and body.

If the request is “ready?”, the response tell the dataset collection script that the GUI has finished rendering and is ready to be captured by camera. If the request is “classes?”, the IDs of all elements and their classes are sent back as body of the response. If the request is “next state?”, the method *nextState()* of the variable state is called before answering with an acknowledgment. As discussed, this method changes the state and triggers rerendering of the GUI according to the new state. If the request is “randomize?”, an acknowledgement is sent, client connection and server are closed using methods *closeClientConnection()* and *close()*, and the method *recreate()* is called. The last method causes the instance of class *MainActivity* used for the GUI to be destroyed. Then, a new instance is created and its method *onCreate(..)* is executed effectively restarting all calculations performed so far resulting in a newly randomized GUI.

The setter method *setState(..)* and the method *isClosed()* are used externally to set and get properties of this class. The former allows the state instance to be replaced in cases where the connection is reused for a new

randomized GUI. This can happen if the connection is lost, or randomization is triggered from somewhere else like the button available in debug mode. The latter returns whether the server is still running or not.

File Utils

File Utils
probToBool(probability: Float): Int
probsToIndex(vararg probabilities: Float): Int
randomText(numMaxWord: Int, numMaxCharsPerWord: Int, allowEmptyString: Boolean): String
onCondition(condition: Boolean, modifier: Modifier): Modifier
clipIntInclusive(value: Int, lowerBound: Int, upperBound: Int): Int
randomHorizontalAlignment(): Alignment.Horizontal

Figure 45. Functions and methods of file Utils.

The internal of file *Utils* as seen in Figure 45 all provide some support functionality. The first two functions *probToBool(..)* and *probsToIndex(..)* take in probabilities and returns a truth value or a number based on those probabilities.

Function *randomText(..)* returns a text that conforms to the restrictions received as parameters. The characters of all words are chosen at random. The first character is always capitalized. The idea behind this approach is, that interactive elements usually adhere to these criteria. A more sophisticated solution would be a dictionary of words that are likely to be used but this would require extensive research and many considerations to provide a more useful text to be displayed in differing types of interactive elements. However, this approach cannot produce representative text for any language that uses non-Latin characters or are written from right to left. Also, there might be some hidden underlying pattern to text of interactive elements that is not present in the dataset when using this approach.

Method *onCondition(..)* makes the use of conditional modifiers for GUI elements easier because this allows them to be applied in one line of code.

Function *clipIntInclusive(..)* should be self-explanatory.

Function *randomHorizonalAlignment()* return one of the possible alignment types (left, right and center) at random.

File Probabilities

This file only contains variables storing probabilities for various decisions made during GUI creation. Apart from being chosen as values that seem like a good fit to the probabilities in real-world applications, there

is no deeper underlying concept to them. E.g., apps are more likely to be used in light mode than in dark mode, or buttons are more often used than text fields.

Visual Results

Some examples of GUIs generated by the application are shown in Figure 46.

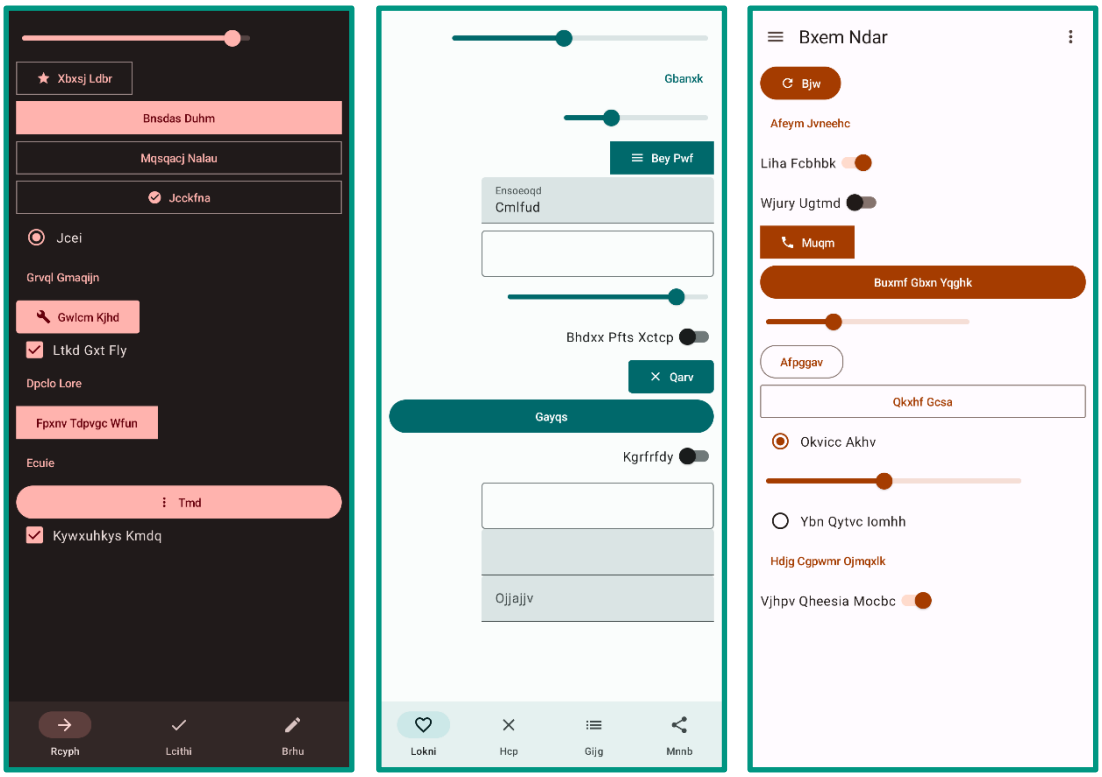


Figure 46. Three examples of GUIs generated using the application described above.

5.2. Dataset Collection Scripts using Python

The component for dataset collection must be able to communicate over the network, access a camera and process the images taken. It must also be able to work with files. All of this can be done using Python with the OpenCV library (OpenCV 2022) among other libraries. Since the DNN component is also written in Python (at least the part of it that needs modification), the reusability of some of the code is an additional advantage when using Python.

To be adaptable to different setups used for dataset collection, the management of collection itself and the capturing of the mobile application's screen with the android application is split up into two Python scripts. Again, the communication works by having the camera act as a service that the managing script can access. Thus, the camera script starts a server to which the managing script can connect as client and request an image whenever needed.

Image Capture Script

This script starts with the initialization of some variables. The IP address and port of the server, and the ID of the camera to be used are specified.

Because so far, the images for the dataset are not taken in the environment the DNN will be deployed in, the background surrounding the mobile device does not provide any useful information for the dataset. Furthermore, the background can interfere with the generation of masks out of the black and white screens provided by the application. Therefore, the image capture script is responsible for removing the background.

To do this, the capture script displays the view of the camera and lets the user select a color and thresholds to filter the camera image for that color. This is saved as binary mask that paints everything black except for the part selected by the user. This part is left as provided by the camera. Ideally, this removes everything but the screen content from the images.

After this, another view of the camera is displayed. Here, the user can crop the image to remove black areas. Because masks must be provided as binary image of black and white, this view also lets the user select another threshold. This threshold is used to paint everything underneath that value black and everything above it white.

Now, all data necessary is collected and the server can be started. The server listens for connections and handles them in the following manner. If the request asks for an image, it provides the cropped image of the screen with black background. If asked for a raw image, it responds with an unedited image of the camera. If asked for a mask, it answers with a cropped binary mask of the screen by using the threshold determined earlier to decide what to paint black and what white. To send the images, they are encoded in the base64 format.

Managing Script

Like the previous script, the managing script starts with the initialization of some variables. These variables determine dataset size and directory, and IP address and port of the android application as well as the camera script.

Next, it defines a class named *Dataset* to handle most operations used in the following main loop. The class and its variables and methods are shown in Figure 47. The constructor saves size and path of the dataset and sets the current size to zero. The next four methods handle the connection to the server of the image capture script and requesting and saving of images and masks. Because of network connectivity issues, they use quite a small value as timeout to retry connecting frequently. Otherwise, dataset collection would take much longer when there is a problem with the connection.

Method *saveAnnotations(..)* takes in relayed data provided by the android application and saves IDs and classes of elements as JSON file. Method *makeRequest(..)* handles connection to the android application.

Finally, an instance of class *Dataset* is created, and the main loop started. This loop is repeated until the current dataset size is equal to the desired dataset size.

Class Dataset
Dataset(total_size, current_size, path)
getImageFromServer(): image
getMaskFromServer(): mask
getFromServerAndSaveImage(path)
getFromServerAndSaveMask(path)
saveAnnotations(path, data)
makeRequest(header): response

Figure 47. Variables and methods of class Dataset.

The loop follows the protocol of Figure 44 shown in the section about the android application. It starts by asking if the android application is ready. After a short delay, it requests an image of the screen from the image capture script and saves it. Then, is asks for information about the classes and saves this. For every element, it asks the application for a mask, waits for a small period, requests an image of the mask from the image capture script and saves this image. Finally, it requests a new randomized GUI from the android application and continues from its beginning for the next datapoint.

The small delays were added because without them, the camera sometimes captures a blurry image. We suspect this to be caused by two problems. First, the GUI may have finished rendering but the camera either has an internal delay when providing images or simply a quite long exposure time and this results in the image being a mix of both the current and the previous GUI displayed. Second, big changes of the camera image trigger refocusing of the camera which also blurs and distorts the image.

5.3. Mask R-CNN

The third component of the implementation is the DNN itself. Different model architectures have been explained in section 2.7. The YOLO family is very popular and widely used. However, its main advantage is the detection speed. While detection should not be much slower than real-time to be used in a robotic testing framework, performances far better than real-time detection do not offer any benefits. The main bottleneck is most likely going to be the movement speed of the robot.

Because of this, accuracy is much more important as long as detection speed is not unreasonably slow. The newer model architectures of R-CNN meet this requirement running detection at about 5 FPS. They also provide good accuracy. But ultimately, detection using bounding-boxes only has two mayor flaws.

First, it cannot properly describe the shape of elements that are not composed of a rectangle. But many elements consist of different shapes. E.g., buttons can have rounded corners, sliders consist of a line with a circle indicating the current position, and switches are also more complex.

Second, if the screen captured is tilted or rotated, even rectangular elements are distorted and cannot be described well using bounding-boxes whose edges are aligned horizontally and vertically. With increasing rotation of an element, bounding-boxes contain more and more pixels that do not belong to the element.

To fix both issues and be able to detect arbitrary shapes accurately, a binary mask is needed. This is possible using instance segmentation.

For all reasons stated above, Mask R-CNN was chosen as model architecture. It is fast enough for the use case at hand, accurate, and performs object detection and instance segmentation.

The specific implementation of Mask R-CNN used in this thesis is provided as GitHub repository by Abdulla (2017). It is implemented in Python using Tensorflow. Its backbone is a feature pyramid network based on ResNet101 as base. According to the author, the implementation deviates from the official paper on Mask-RCNN in the following points: Image resizing, bounding-boxes, and learning rate. The points are explained in more detail in Table 3.

Table 3. Differences between implementation by Abdulla and official paper by He et al. according to Abdulla (2017) .

	Implementation by Abdulla	**Official paper by He et al.**
Image Resizing:	Aspect ratio is kept, and images are resized to fit inside a square of fixed size. If an image is not square, remaining pixels are filled with black.	All images are resized so that the smaller side is 800px long. Then, the larger side is trimmed to be not larger than 1000px.
Bounding-Boxes:	Generates bounding-boxes by calculating the smallest box encapsulating the entire	Uses bounding-boxes as provided in datasets.

	masks provided by the dataset for an object. This allows image augmentation like rotation to be applied.	
Learning Rate:	Uses learning rate of 0.001 to prevent weights from getting too large.	Uses learning rate of 0.02.

To train this implementation of Mask R-CNN on the dataset created by the dataset collection script, a sample script for the detection of geometrical shapes provided with the implementation was modified and a docker image created that contains all necessary libraries and has access to the GPU. Most modifications required were made in the classes *IEConfig* and *IEDataset* and are covered in the next two sections. Afterwards, some additional implementation details are discussed. IE stands for interactive elements.

IEConfig

The class *IEConfig* extends the class *Config* provided by the Mask R-CNN implementation and mainly contains multiple variables defining parameters used for training. The most important ones are explained in the following paragraphs.

Variable *CLASSES* contains all eleven classes occurring in the dataset. Namely, these classes are “Menu”, “BackButton”, “MoreOptions”, “FloatingActionButton”, “NavigationBarItem”, “Button”, “Checkbox”, “RadioButton”, “Switch”, “Slider” and “TextField”.

Variable *AUGMENTATION* contains a class that defines a series of image augmentations to be applied to the images in the dataset before using them for training. In order, these augmentations are the following. Rotation and scaling of the input image, changes in contrast, and changes in brightness. Each of them has a probability of 50% to be applied. It is important that the order is preserved because if contrast or brightness were to be changed before applying rotation or scaling, the outline of the original image would be visible (black in the original image would not be black anymore, but black filling missing pixels would be resulting in visible color difference at image edges).

Variable *INIT_WITH* is initialized with value “coco”. This tells Mask R-CNN to load pretrained weights before starting the training. The weights are result of previous training of Mask R-CNN on the COCO dataset.

IEDataset

Class IEDataset extends utils.Dataset
train_test_split(percentage)
add_classes()
load_ie(dataset_dir)
get_index_of_class(class_of_element)
load_image(image_id)
load_masks(image_id)

Figure 48. Methods of class IEDataset.

Class *IEDataset* is responsible for loading and management of the dataset provided by the dataset collection script. For this, it extends the class *Dataset* defined in file *utils.py* of the Mask-RCNN implementation. The overwritten and added methods are shown in Figure 48 and discussed in the next paragraphs.

Method *train_test_split(..)* divides a single dataset and returns two distinct portions of it as new datasets that can be used for training and testing of Mask R-CNN.

Method *add_classes()* saves all classes given in class *Config* to the dataset and assigns each of them an ID.

Method *load_ie(..)* starts by executing method *add_classes()*. Then, it uses the JSON files to go through every datapoint. For each of them, it saves a datapoint ID, the common part of the path for all files belonging to that datapoint, width and height of the input image, and a list of class IDs of all elements present in the datapoint. For the list of class IDs, it uses method *get_index_of_class(..)* which returns the ID for a given class name.

Method *load_image(..)* uses the ID of an image to load and return the actual image.

Method *load_mask(..)* loads all masks for a datapoint. They are returned as a three-dimensional array where the first two dimensions are height and width of the mask of one single element and the third dimension is the number of masks for all elements of that datapoint.

Additional Implementation Details

Apart from everything discussed so far, the script uses an additional tool named wandb (Weights & Biases – Developer tools for ML 2022) for tracking of the training. It provides graphs with metrics and summaries about all runs, which are very helpful. It is used by adding wandb as library and executing some code before and after training to start and finish tracking.

5.4. Workflow for Dataset Collection and Training

After having looked into the implementation details of all components from a coding perspective, this section provides a different point of view showing the workflow of dataset collection and training of Mask R-CNN in practice.

For the dataset collection, some setup steps are required. At first, the variables IP, port and debug mode for the android application are set in the integrated development environment (IDE) and the application is compile and transferred to a mobile device The device used is a Samsung Galaxy S21 FE 5G running on version 13 of the android operating system. IP must match the IP of the mobile device in the network used for communication since it is used to start a server. Debug mode should be enabled for a following step. Because the mobile device and the camera capturing it should not be moved after starting this process, both are placed in a secure spot where there is no interference to be expected.

After this, the image capture script is configured. The variables for IP, port and camera ID must be set. Since this is a server, the IP must match the IP of the computer running the script in the network used for communication. To create a mask that filters out everything except the screen, the screen of the mobile device is filled with a distinct color (E.g., opening an image in fullscreen mode that contains only green pixels.). Now, the image capture script can be started. It will display two windows. One of them can be used to select the color on the screen by clicking on it and provides sliders to adjust threshold of hue, saturation, and value. The other window gives feedback by showing the current result of applying the mask for removal of background surrounding the screen. This is shown in Figure 49.

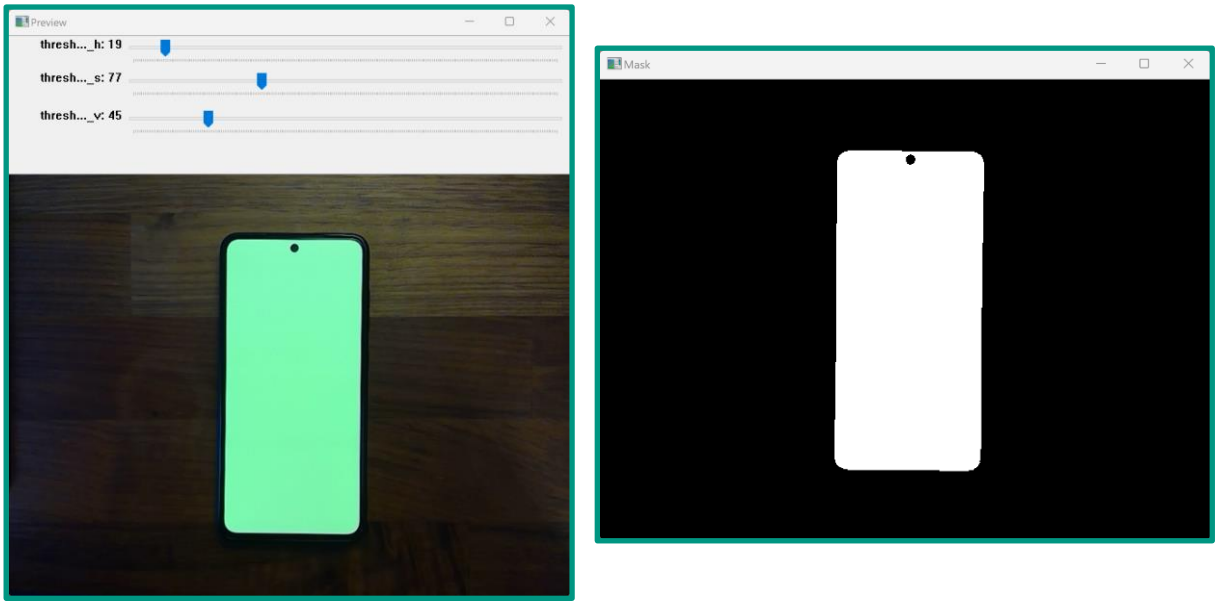


Figure 49. First window shows image of mobile device with green screen and sliders for thresholds. Second window shows the resulting binary mask after selecting the screen color by clicking it and adjusting the sliders.

When pressing “q”, the currently shown windows are closed and the script continues by opening another window. This one is used to crop the image to remove unneeded background and to select a threshold value that will later decide whether to paint a mask pixel black or white. The cropping is done by pressing the left mouse button when pointing at the upper left corner of the desired rectangle and then pressing the right mouse button for the lower right corner. This reduces the storage space required for the dataset. The window before and after cropping is shown in Figure 50.

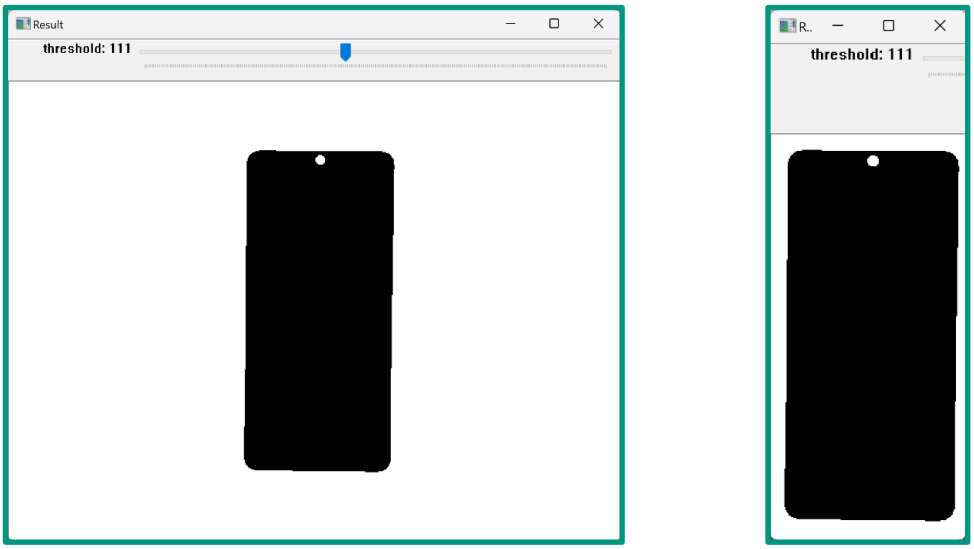


Figure 50. Window before and after cropping. Most pixels that are not part of the screen were removed.

To decide for a threshold value, the android application was started in debug mode. This way, the application’s state can be advanced to display a mask. Now, a threshold is chosen using the image shown in the window as feedback. Usually, a value in the middle around 124 is a good fit. Again, pressing “q” closes the window. Finally, the last part of the script is executed starting the server. Figure 51 shows the window with an appropriately chosen threshold to filter for masks. It also shows the process of converting the raw image into a binary mask.

Next, the button for turning off debug mode in the android application is pressed. At this points, android application and image capture script are ready. After entering the required variables IP and port of the android application and image capture script, and desired dataset size and path, the dataset collection script is executed. This automatically creates the entire dataset and saves it to the specified path. We used a dataset

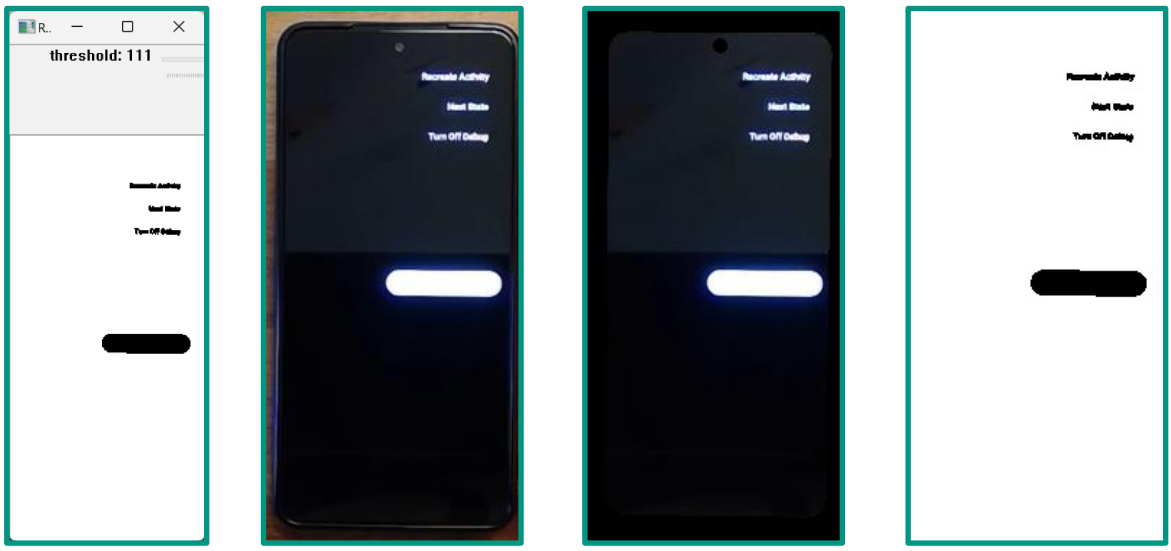


Figure 51. Window with threshold chosen based on the mask displayed (left). The other images show the processing applied to the raw image to obtain a binary mask. From left to right, the following steps are performed: Painting everything black except for the screen. Applying threshold to generate a binary output which is then inverted and shown in the image on the right.

size of 2000 datapoints. The collection of these datapoints took about 4 hours and 50 minutes. Therefore, collection speed was roughly 6.9 datapoints per minute. Some examples for datapoints with GUI and masks of the four classes appearing most often are given in Figure 52.

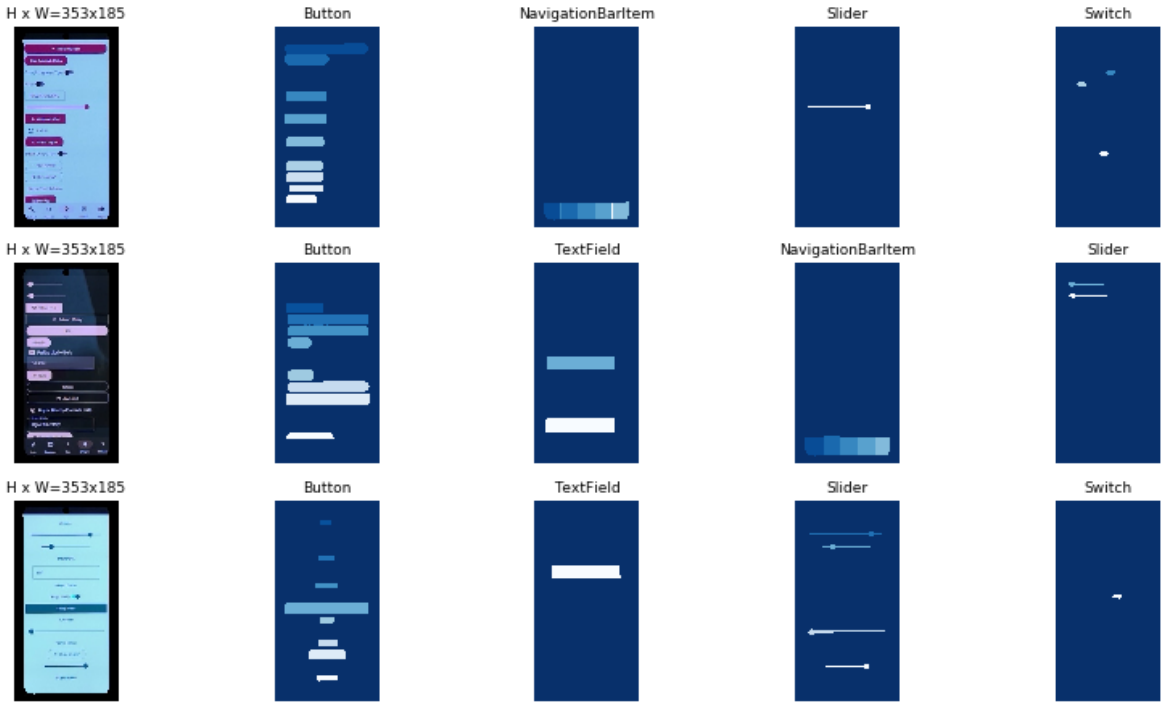


Figure 52. Each row represents one datapoint. The first image in a row shows the GUI captured by camera, which is the input image for training. The following images in each row display masks of all objects of the four classes appearing most often in the GUI. Each mask instance is colored in a different shade.

The script for training of Mask R-CNN is executed next. The dataset is loaded and split into training and test data. Training data is made up of 80% of the original dataset, which is a common percentage for splitting training and test data. As result, the training dataset contains 1600 datapoints and the test dataset contains 400 datapoints.

Finally, the training is started. As suggested by the authors of the Mask R-CNN implementation, training is started with the backbone fixed. Only the randomly initialized layers not part of the CNN are trained using a learning rate of 0.001. After this, all layers are fine-tuned using a reduced learning rate of 0.0001. Because the backbone was trained on images of real-life scenes, we suspect the learned weights to be useful but not quite ideal for the detection of GUI elements shown on a screen. Therefore, we believe that fixing these layers can quickly become a limiting factor for learning. To prevent this, training with fixed backbone is only run for 20 epochs and after this all layers are adjusted during fine-tuning until epoch 150. As shown later, the graph of the loss functions confirms this hypothesis. After unfreezing all weights, the rate of improvement increases.

5.5. Problems

This section points out some issues and difficulties that occurred during implementation. While most problems appeared early in the process and were resolved, some appeared quite late or are still present. Most notably, those problems are the following:

The quality of the camera available is not ideal. Its resolution is quite low, which is not the main issue because usually computer vision tasks can be performed well on low resolution data. It is even beneficial in some way as it increases training speed. However, the camera used also introduces some blur/glare in images. This is probably caused by the high contrast especially present in the black and white masks.

Another issue cause by the camera was only detected after the collection of the dataset. It seems that even though there is a delay between the capture of images, the camera sometimes refocuses. This produces changes in size of the captured scene which in turn resulted in some masks not perfectly fitting the position and shape of the element inside the GUI. Some examples are shown in Figure 53.

Figure 53. The two images show GUIs captured overlayed with all their corresponding masks. In the first image, all masks appear to be scaled down, which makes them smaller and move towards the center of the image. In the second image, the opposite seems to be the case. Because each individual mask is taken one after another, this implies that focus changed only once after taking the GUI image and before taking all mask images.

According to accuracy measures for different datapoint, this issue occurred randomly for some of the datapoints. It did not occur consistently, and the accuracy did not deteriorate over time. It also did not occur only for light mode or dark mode used in the GUI. This suggests that the issue cannot be fixed by increasing the delay because then the issue would be expected to show up in all datapoints. Furthermore, it cannot be caused by changes of lightning conditions during the dataset collection or accidental changes in the physical setup (E.g., bumping into the camera or mobile device or a slow movement of the camera because of physical instability of the setup.). This would have resulted in a trend over time. The best guess is that this refocusing is happening at random and can only be removed by using a different camera.

Another problem occurred when converting masks saved as images back to binary masks. This was discovered after training was finished and therefore training had to be repeated. We expected the images to contain white and black pixels only. However, compression changed some pixels to grey shades although we expected the image format (.png) to apply compression in a way that does not do this. Either the format itself did this or some conversion during processing produced this problem. Because of this, the way of reading in masks was flawed and had to be adapted. It changes some pixels that are part of a mask to non-mask pixels.

Because of the way the elements were implemented in Material Design, some elements could not be simply painted in black to get a mask of their shape. E.g., the radio button either displayed a hollow circle or painted a bigger rectangle around its shape. Therefore, some tricks were needed to receive a valid and

accurate mask. In this example, a rounded white border was painted around the black rectangle to get a circle of the size of the radio button. However, this has the disadvantage that masks of elements behind the radio button would not be displayed properly because of the white border overlaying the mask. Since this dataset rarely has interactive elements overlapping each other and since the items occluding other are always either the FAB or the BottomBar, this is no problem now. However, it must be handled if the feature of occluded masks was to be used more heavily in a future dataset.

Finally, dataset collection took a long time as mentioned before. Theoretically, this could be improved a lot by minimizing delays. However, problems with the camera used were the limiting factor. A different camera should allow for a speedup of the process. This in turn would make the generation of large datasets much more convenient.

6. Evaluation

To judge the performance of the trained DNN, numerical metrics and visual samples are used. At first, different runs are compared. Afterwards, the DNN is evaluated on the test dataset of this thesis and GUIs of real-world android applications.

Training was conducted using an Intel i7-4790k as CPU and an Nvidia GeForce RTX 2060 SUPER as GPU on a machine running Kubuntu 20.04 as operating system. As discussed before, learning rate was 0.001 and the backbone was frozen for the first 20 epochs. For all other epochs, learning rate was 0.0001 and all layers were trained. All runs whose loss functions are plotted in Figure 54 show a faster improvement after epoch 20.

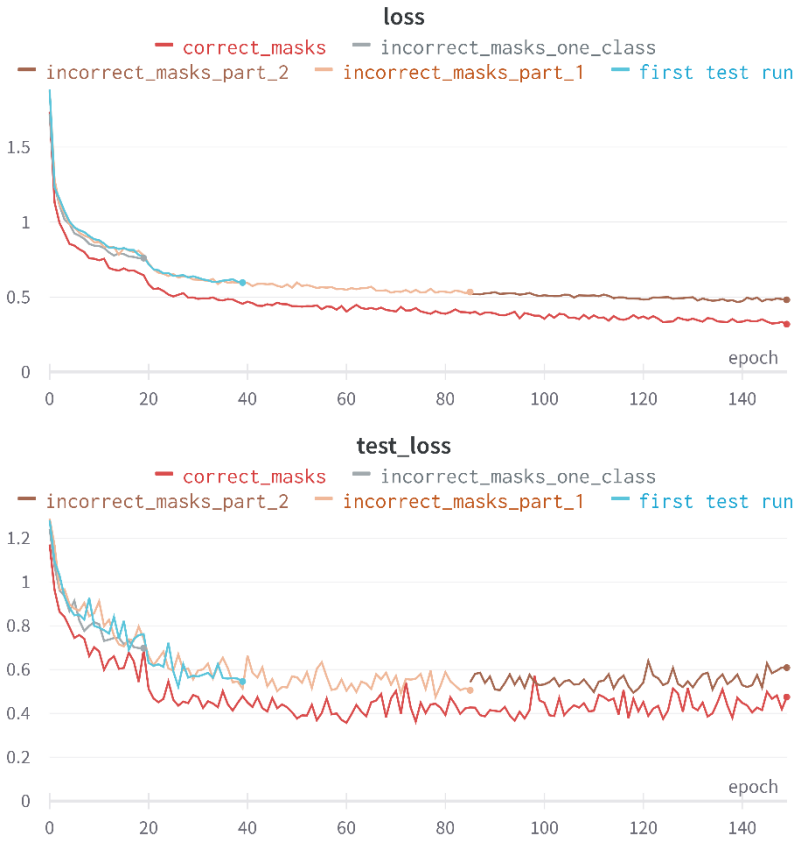


Figure 54. Plots of loss function for training dataset (top) and test dataset (bottom) for four runs. Marked in blue is a first test run. Yellow and golden are part one and two of the first full run with a bug. Grey is a run with bug using only one common class for detection. Red shows the final run after fixing the bug, which performs visibly better than the other runs. All runs start with fast improvement that later slows down and eventually plateaus.

This confirms the hypothesis that the backbone trained on real-world sceneries quickly becomes a limiting factor for GUI recognition if weights are not adjusted. Unfreezing of the backbone could be done even earlier in the training process.

Mask Fix and Overfitting

As already mentioned, one issue concerning the loading of masks was fixed after training runs was already conducted. These previous runs include one first test run with 39 epochs, a full run with 150 epochs (this run is split in two parts because the program stopped prematurely due to missing disk space) and a run with 20 epochs that used one common class for detection and will be discussed later.

As can be seen in Figure 54, this fix significantly improved the error as measured by the loss functions for training and test dataset. Furthermore, Figure 54 gives us intel about the ability of the trained DNN to generalize. For most of the epochs, training and test loss is roughly the same. This indicates good generalization. However, training and test loss slowly diverge for the second half of the epochs. Training loss increases a little bit while test loss gets worse again. At this point, the DNN is starting to slightly overfit on the training data. This suggests, that the DNN could benefit from a larger dataset. However, the benefit is probably going to be quite small.

Common Class

Also visible in Figure 54 is the run with one common class. The idea behind the ablation of the DNN was as follows. Some interactive elements look quite similar. This could cause the DNN to make a decision for the wrong class which in turn would influence its estimate of bounding-box and mask. If this was the case, a DNN having only one class would perform better. However, the losses are on par with those of other runs that were conducted with the mask bug present. This likely means that there is no advantage of using a common class and that the reasoning was wrong.

Evaluation on Training Dataset of this Thesis

Epoch 60 had the best test loss with a value of 0.36. Therefore, the parameters of that epoch were used for all further evaluations.

The mAP@IoU=0.5 for the training dataset is 0.97. This is slightly below the value for most related work. However, it must be noted that they are not directly comparable for multiple reasons. The test dataset is not the same and our approach uses camera images instead of screenshots. Furthermore, the mAP score is only a baseline for what is achievable with this approach because of the quality issues caused by the camera.

Figure 55 shows three visual examples of detection performed on the test dataset. The DNN recognizes all elements and assigns the correct class. Most bounding-boxes and masks look very accurate. However, some of them slightly deviate from the shape of the element. This especially seems to be the case for elements where the ratio of width to height is far above one (I.e., elements that are very wide but not tall.). This could be addressed by adding additional anchor boxes whose shape is closer to that of objects with such an aspect ratio because currently the aspect ratios for anchor boxes are 1:2, 1:1, and 2:1.

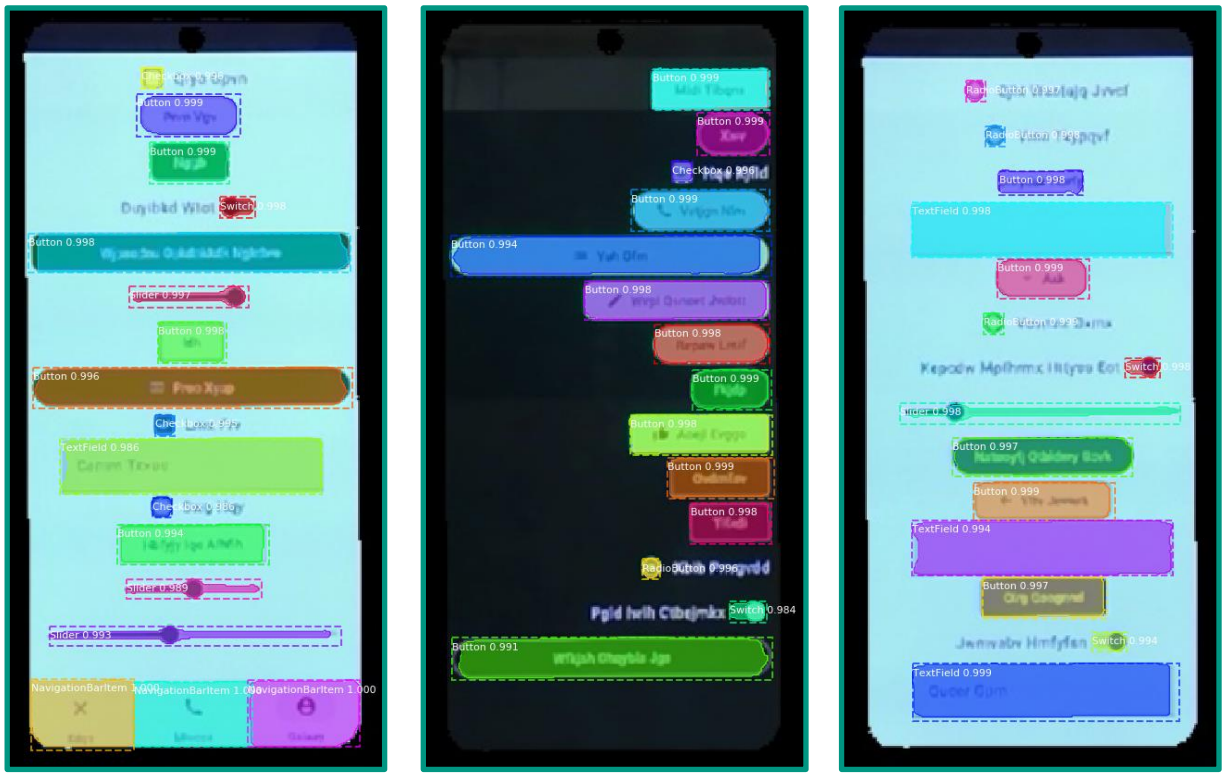


Figure 55. Visual examples of detection results for the test dataset.

Evaluation on Real-World Android Applications

Using GUIs from the self-written android application for evaluation provides information about how well the training itself worked. However, it does not give any hints about how well the dataset and with it the DNN generalizes for real-world applications. A DNN can only discover patterns that are present in the data it is being fed for training. If the patterns underlying the training data are not representative of real-world applications, detection result for those can still be much worse than for the test dataset.

Because there is no suitable dataset available to evaluate the performance of the DNN for real-world applications, we picked sample GUIs from android applications that are popular in the Google Play Store or frequently used by ourselves. This selection may be biased as we only picked GUIs that do not reveal sensitive information.

Figure 56 shows some of the samples. We picked a range of good and bad detection samples to display. The sample on the left is being detected quite well. Almost all interactive elements are recognized even though the switches were detected as buttons. In the middle, some elements are detected but many are missing. On the right, recognition of the elements of the main content works well. Button and text field have been identified. But recognition also returned additional results for the main content which are no interactive elements. For the other areas, multiple elements were thrown together and detected as single button. Further samples are given in Appendix B. Again, some samples are recognized very well while others are recognized very poorly. The DNN seems to do well for simple elements. However, it struggles as elements get more complex. All in all, recognition is definitely working worse than for the test dataset.

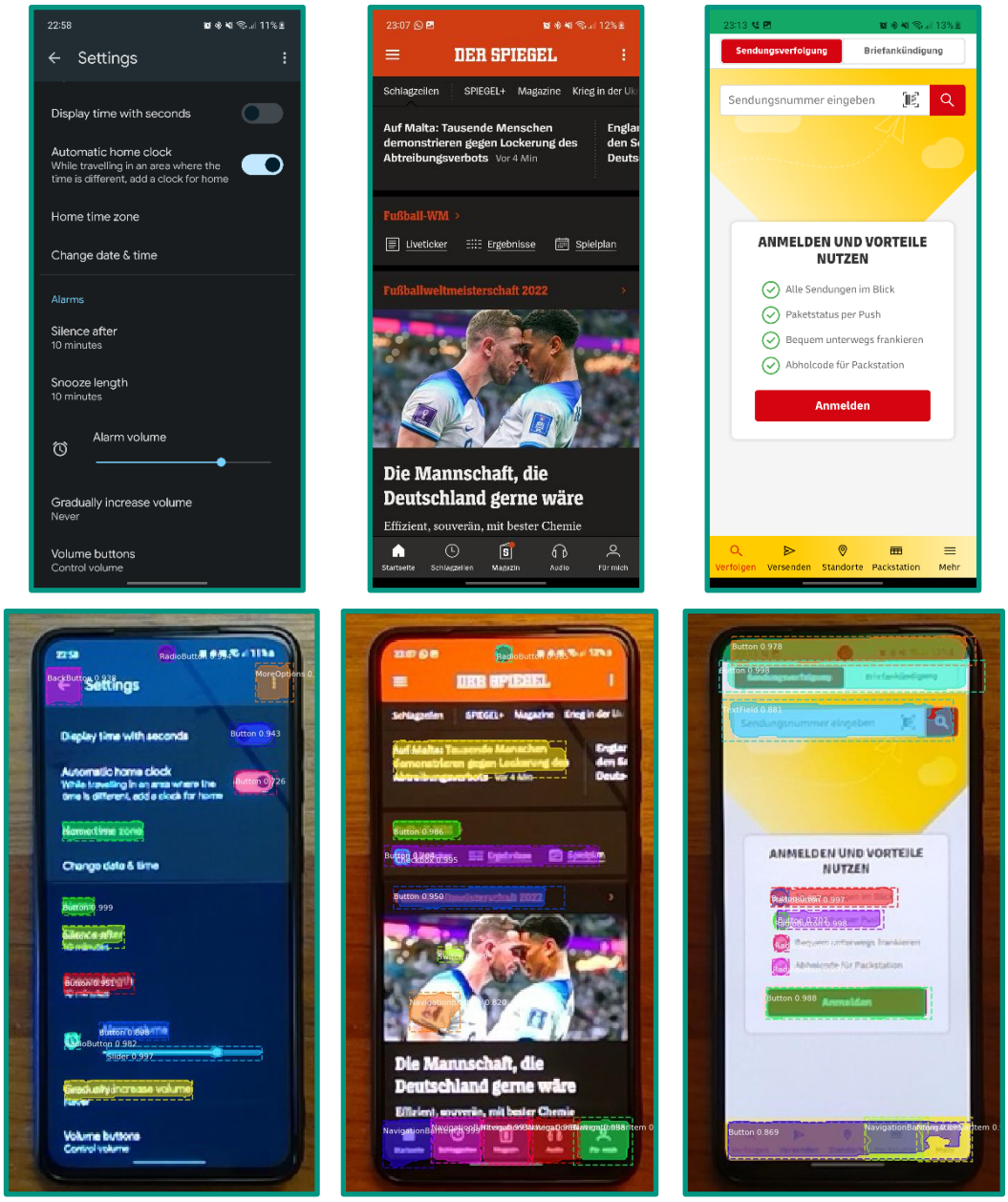


Figure 56. Top row shows screenshots of GUIs. Bottom row shows the corresponding detection results.

7. Summary

This thesis investigated the approach of detecting interactive elements in GUIs of mobile applications using a DNN trained on an automatically generated and labeled dataset. The underlying idea behind this approach was that manual labeling is time-consuming and inaccurate. Automatic labeling could solve both issues at once. To evaluate this approach, it was implemented using three components: An android application that provides randomized GUIs and masks of the interactive elements present in them. Python scripts for capturing of images of the application and collection of a dataset. Mask R-CNN trained on the resulting dataset.

After collecting the training and test dataset and training the DNN, the performance was evaluated on the test dataset and real-world android applications. Numerical metrics and visual examples produced with the test dataset were good (mAP@IoU=0.5 of 0.97). However, the approach did not translate equally well to real-world applications. Often, simple elements were detected well and accurately but more complex ones were usually missed. Because of this, the approach is so far only of limited use for automatic testing frameworks.

7.1. Limitations

The results are hard to compare to related papers because of the different ways of capturing the GUI. In fact, using a camera image of the screen is most likely more challenging. The approach used in this thesis works good for many interactive elements. Still, detection results are not quite good enough for reliable detection of many real-world GUIs. There are too many false positive and false negative results.

For the desired use case of robotic testing, false positives may not pose a big problem. Clicking at something that is not actually clickable does not have any negative effects. Meanwhile, false negatives are a great problem because every element not detected reduces test coverage. It potentially not only means that this element is not tested but that elements in GUIs only accessible by interacting with this element are also left out as they are never shown.

As stated above, more classical interactive elements like a simple button are detected well but many more complicated interactive elements like multiple lines of text or texts with images are not detected as the dataset does not contain data on these. This is a problem as many modern applications rely heavily on interactive elements that vary greatly in shape, size, look, and content.

In part, this may be a problem that cannot really be solved well because even for humans it is often not directly clear if an element is interactive when only looking at GUIs. Sometimes, a human also needs trial and error to figure this out.

7.2. Outlook

In this last section, we would like to present some thoughts and ideas for further research that came up during the writing and implementation of this thesis and that could solve some of the issues discussed above.

First of all, there may be some optimization potential concerning the Mask R-CNN model used. Parameter tweaking could improve results. As noted before, some masks and bounding-boxes for elements with a great difference of width and height were not that accurate. This will potentially be solved by adding further aspect ratios for anchor boxes created to better account for such elements. Next, the DNN detected some false positives and some false negatives during evaluation. Because detecting false positives is less harmful and may actually resemble the way a human would approach this task (trial and error with many potential interactive elements to sort out false positives), the DNN's confidence thresholds could be reduced. This way, detection would shift away from detecting many false negatives towards more false positive results. These could later be filtered out and coverage of true positives would probably increase along the way. Additionally, we came across a newer model version we did not know of before by Huang et al. (2019) named Mask Scoring R-CNN, which could replace Mask R-CNN and boost performance.

As second measure, the dataset could be improved in various ways: Either by making it larger or by improving quality of the interactive elements in the GUIs. It could be complemented by a manually labeled dataset or by implementing additional design frameworks apart from Material Design. The elements already implemented could also be used to create more complex elements. As touched on in the implementation, the text generation is biased and could be improved. To get data to closer resemble real-world GUIs, open-source applications could be adapted for the use in automatic dataset collection and labeling because their source-code is accessible and enables adaptation.

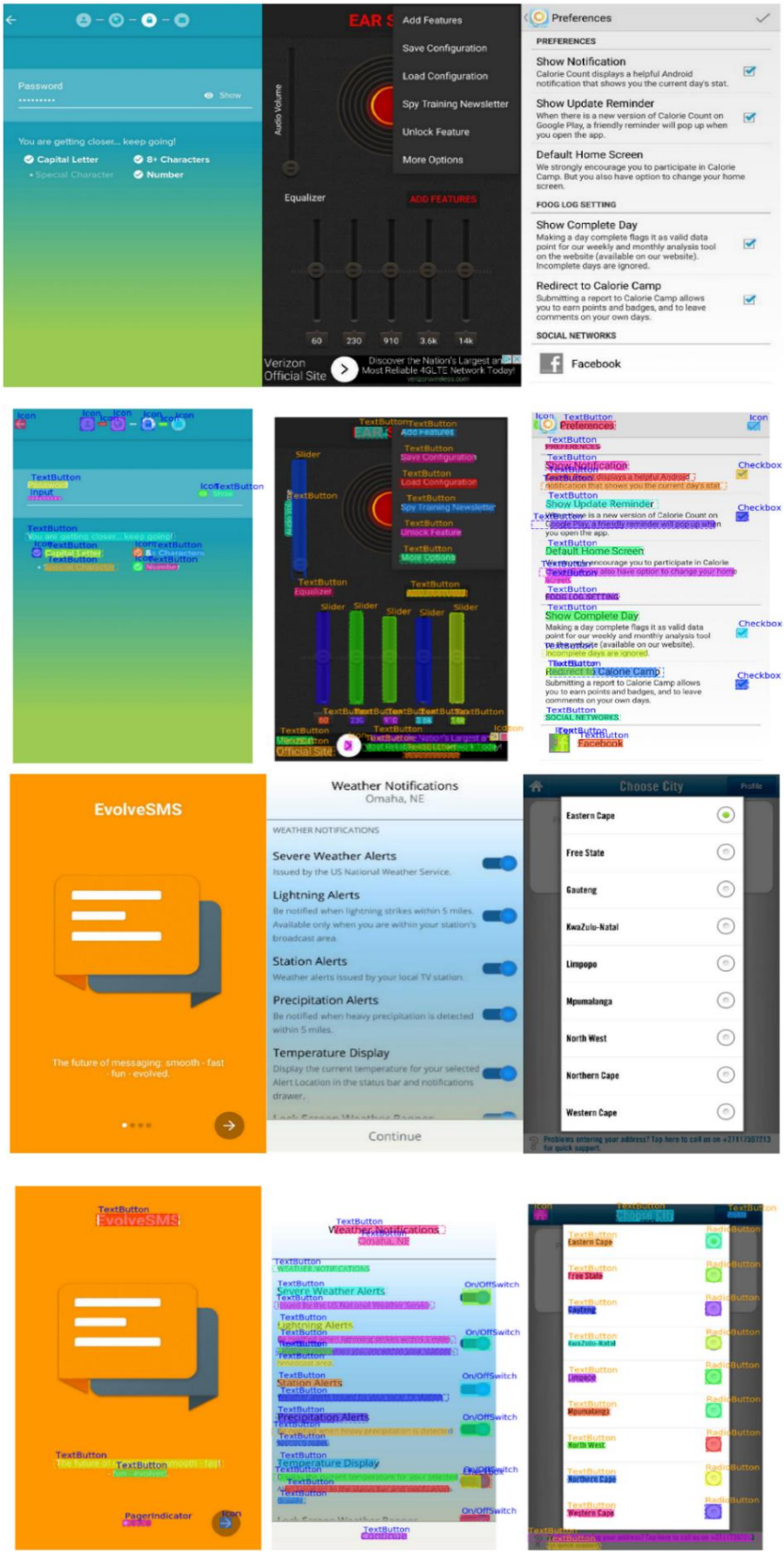
Next, a camera capturing images with less blur and more consistency could improve accuracy. As a side effect, images could be captures in faster succession making the approach a better solution for fast dataset collection and training.

Additionally, we noticed the following: Automatically generated and labeled datasets provide more accurate bounding-boxes and masks but manually labeled datasets are more representative when it comes to the detection of classes and whether or not there is an interactive element present. This happens due to the fact that the appearance of interactive elements in real-world applications varies greatly. This variety can hardly be reproduced in a self-written application. However, manually labeled dataset have access to all these applications as data. Furthermore, accuracy is not as important for classification and region proposals as for masks and bounding-boxes. These observations gave rise to the following idea: One could use both types of datasets for training but only for the parts where the datasets are best at. The manually labeled datasets would be used for training of classification and region proposal generation by freezing mask and

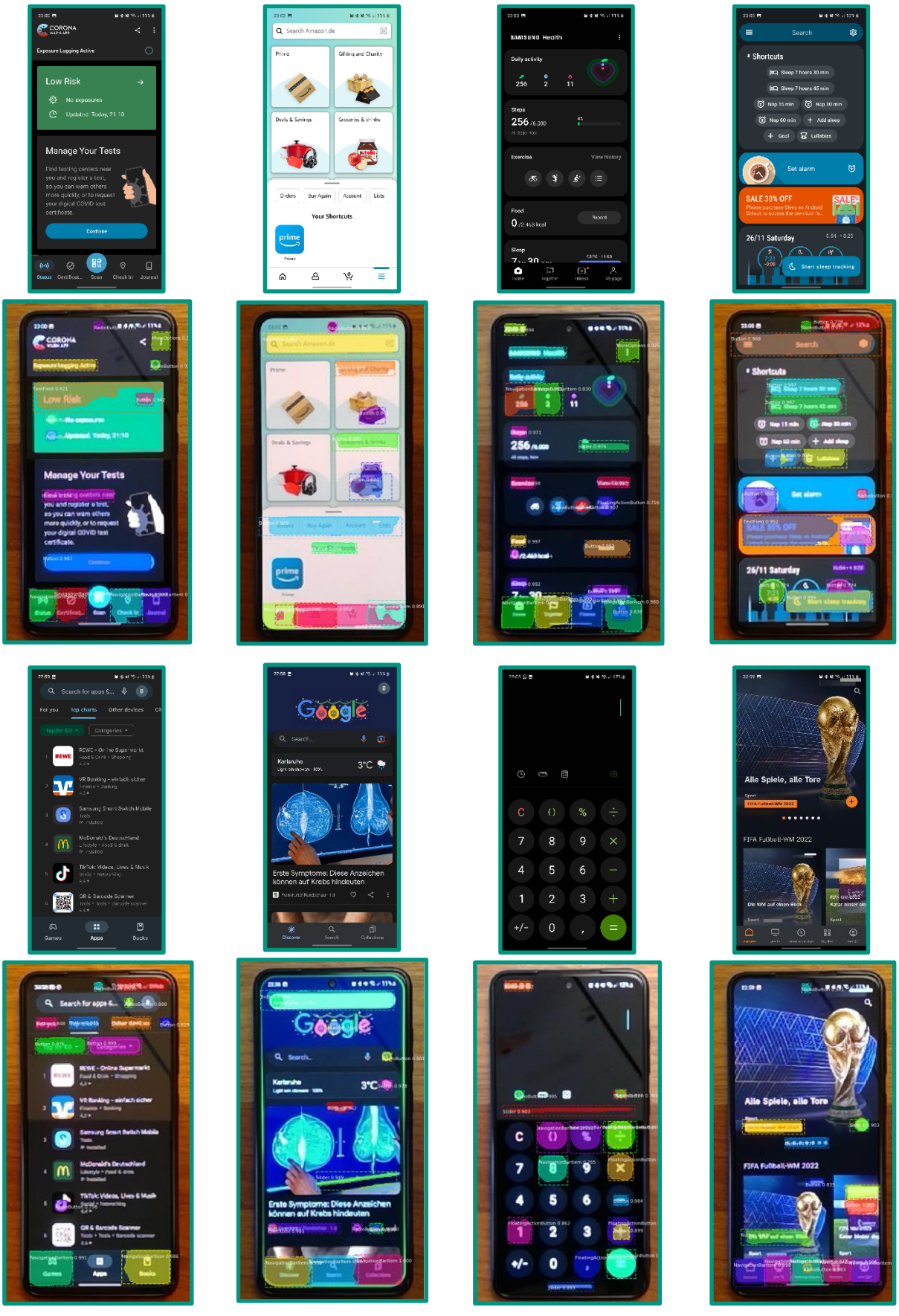
bounding-box branches. The automatically generated and labeled datasets would be used for the exact opposite. They would only train bounding-box and mask branch by freezing the convolutional backbone and region proposal network. This hybrid training could bring out the strengths of each type of data and improve overall performance.

Finally, if this DNN is going to be used for detection in a pipeline with other DNNs (E.g., a robotic testing framework that uses reinforcement learning.), the convolutional backbone could be shared to reduce parameter count for faster training and inference.

8. Appendices



Appendix A. Example of detection and segmentation results. (Hu et al. 2020)



Appendix B. Further samples of detection on real-world applications.

References

Abdulla, Waleed. 2017. Mask R-CNN for object detection and instance segmentation on Keras and TensorFlow. https://github.com/matterport/Mask_RCNN. Accessed 5 December 2022.

Alessandro Repici, Matteo Badalamenti, Roberta Maselli, Loredana Correale, Franco Radaelli, Emanuele Rondonotti, Elisa Ferrara, Marco Spadaccini, Asma Alkandari, Alessandro Fugazza, Andrea Anderloni, Piera Alessia Galtieri, Gaia Pellegatta, Silvia Carrara, Milena Di Leo, Vincenzo Craviotto, Laura Lamonaca, Roberto Lorenzetti, Alida Andrealli, Giulio Antonelli, Michael Wallace, Prateek Sharma, Thomas Rosch, and Cesare Hassan. 2020. Efficacy of Real-Time Computer-Aided Detection of Colorectal Neoplasia in a Randomized Trial. *Gastroenterology* 159 (2): 512-520.e7. doi: 10.1053/j.gastro.2020.04.062.

Ansari, Shamshad. 2020. *Building computer vision applications using artificial neural networks: With step-by-step Eeamples in OpenCV and TensorFlow with Python / Shamshad Ansari.* Berkeley, CA: Apress.

Bharati, Puja, and Ankita Pramanik. 2020. Deep Learning Techniques—R-CNN to Mask R-CNN: A Survey. In *Computational Intelligence in Pattern Recognition*, 657–668. Springer, Singapore.

Biamonte, Jacob, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. 2017. Quantum machine learning. *Nature* 549 (7671): 195–202. doi: 10.1038/nature23474.

Bishop, Christopher M. 2006. *Pattern Recognition and Machine Learning.* New York, NY: Springer New York.

C. Zhang, T. Shi, J. Ai, and W. Tian. 2021. Construction of GUI Elements Recognition Model for AI Testing based on Deep Learning. In *2021 8th International Conference on Dependable Systems and Their Applications (DSA)*, 508–515. 2021 8th International Conference on Dependable Systems and Their Applications (DSA). doi: 10.1109/DSA52907.2021.00075.

Chen, Jieshan, Mulong Xie, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, and Guoqiang Li. 2020. Object detection for graphical user interface: old fashioned or deep learning or a combination? In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 1202–1214, New York, NY, USA. New York, NY, USA: ACM. doi: 10.1145/3368089.3409691.

Components – Material Design 3. n.d. https://m3.material.io/components. Accessed 5 December 2022.

Coppola, Riccardo, Emanuele Raffero, and Marco Torchiano. 2016. Automated mobile UI test fragility: an exploratory assessment study on Android. Proceedings of the 2nd International Workshop on User Interface Test Automation. 2016/07/21. ACM. doi: 10.1145/2945404.2945406.

Da Silva, Ivan Nunes, Danilo Hernane Spatti, Rogerio Andrade Flauzino, Luisa Helena Bartocci Liboni, and Silas Franco dos Reis Alves. 2016. *Artificial neural networks: A practical course / Ivan Nunes da Silva, Danilo Hernane Spatti, Rogerio Andrade Flauzino, Luisa Helena Bartocci Liboni, Silas Franco dos Reis Alves.* Switzerland: Springer.

Deka, Biplab, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In *Proceedings of the 30th Annual Symposium on User Interface Software and Technology*, 845–854.

Delia, Lisandro, Nicolas Galdamez, Pablo Thomas, Leonardo Corbalan, and Patricia Pesado. 2015. Multi-platform mobile application development analysis. In *2015 IEEE 9th International Conference on Research Challenges in Information Science (RCIS 2015): Athens, Greece, 13-15 May 2015*, 181–186. 2015 IEEE 9th International Conference on Research Challenges in Information Science (RCIS), Athens, Greece. 5/13/2015 - 5/15/2015. Piscataway, NJ: IEEE. doi: 10.1109/RCIS.2015.7128878.

Deng, Jia, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE. doi: 10.1109/cvpr.2009.5206848.

Everingham, Mark, Luc van Gool, Christopher K. I. Williams, John Winn, and Andrew Zisserman. 2010. The Pascal Visual Object Classes (VOC) Challenge. *International Journal of Computer Vision* 88 (2): 303–338. doi: 10.1007/s11263-009-0275-4.

Felzenszwalb, Pedro F., and Daniel P. Huttenlocher. 2004. Efficient Graph-Based Image Segmentation. *International Journal of Computer Vision* 59 (2): 167–181. doi: 10.1023/B:VISI.0000022288.19776.77.

Felzenszwalb, Pedro F., Ross B. Girshick, David McAllester, and Deva Ramanan. 2010. Object detection with discriminatively trained part-based models. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 32 (9): 1627–1645. doi: 10.1109/TPAMI.2009.167.

Firtman, Maximiliano. 2010. *Programming the Mobile Web.* Sebastopol: O'Reilly Media, Inc.

Girshick, Ross. 2015. Fast R-CNN. 2015 IEEE International Conference on Computer Vision (ICCV). 2015/12. IEEE. doi: 10.1109/ICCV.2015.169.

Girshick, Ross, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014a. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation. In *2014 IEEE Conference on Computer Vision and Pattern Recognition*. IEEE. doi: 10.1109/cvpr.2014.81.

Girshick, Ross, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2014b. Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation Supplementary Material. https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.667.797&rep=rep1&type=pdf.

Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning.* MIT Press.

Haenssle, H. A., C. Fink, R. Schneiderbauer, F. Toberer, T. Buhl, A. Blum, A. Kalloo, A. Ben Hadj Hassen, L. Thomas, A. Enk, L. Uhlmann, Christina Alt, Monika Arenbergerova, Renato Bakos, Anne Baltzer, Ines Bertlich, Andreas Blum, Therezia Bokor-Billmann, Jonathan Bowling, Naira Braghiroli, Ralph Braun, Kristina Buder-Bakhaya, Timo Buhl, Horacio Cabo, Leo Cabrijan, Naciye Cevic, Anna Classen, David Deltgen, Christine Fink, Ivelina Georgieva, Lara-Elena Hakim-Meibodi, Susanne Hanner, Franziska Hartmann, Julia Hartmann, Georg Haus, Elti Hoxha, Raimonds Karls, Hiroshi Koga, Jürgen Kreusch, Aimilios Lallas, Pawel Majenka, Ash Marghoob, Cesare Massone, Lali Mekokishvili, Dominik Mestel, Volker Meyer, Anna Neuberger, Kari Nielsen, Margaret Oliviero, Riccardo Pampena, John Paoli, Erika Pawlik, Barbar Rao, Adriana Rendon, Teresa Russo, Ahmed Sadek, Kinga Samhaber, Roland Schneiderbauer, Anissa Schweizer, Ferdinand Toberer, Lukas Trennheuser, Lyobomira Vlahova, Alexander Wald, Julia Winkler, Priscila Wölbing, and Iris Zalaudek. 2018. Man against machine: diagnostic performance of a deep learning convolutional neural network for dermoscopic melanoma recognition in comparison to 58 dermatologists. *Annals of oncology : official journal of the European Society for Medical Oncology* 29 (8): 1836–1842. doi: 10.1093/annonc/mdy166.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 2016/06. IEEE. doi: 10.1109/CVPR.2016.90.

He, Kaiming, Georgia Gkioxari, Piotr Dollar, and Ross Girshick. 2017. Mask R-CNN. 2017 IEEE International Conference on Computer Vision (ICCV). 2017/10. IEEE. doi: 10.1109/ICCV.2017.322.

Howard, Andrew, Mark Sandler, Bo Chen, Weijun Wang, Liang-Chieh Chen, Mingxing Tan, Grace Chu, Vijay Vasudevan, Yukun Zhu, Ruoming Pang, Hartwig Adam, and Quoc Le. 2019. Searching for MobileNetV3. In *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*. IEEE. doi: 10.1109/iccv.2019.00140.

Hu, Rui, Mingang Chen, Lizhi Cai, and Wenjie Chen. 2020. Detection and Segmentation of Graphical Elements on GUIs for Mobile Apps Based on Deep Learning, 187–197. International Conference on Mobile Computing, Applications, and Services. Springer, Cham. doi: 10.1007/978-3-030-64214-3_13.

IEEE/ISO/IEC International Standard - Software and systems engineering-Software testing-Part 4: Test techniques. IEEE: 1–286.

Introduction - Material Design. n.d. https://www.material.io/design/introduction#principles. Accessed 5 December 2022.

Jamil, Muhammad Abid, Muhammad Arif, Normi Sham Awang Abubakar, and Akhlaq Ahmad. 2016. Software Testing Techniques: A Literature Review. In *6th International Conference on Information and Communication Technology for the Muslim World: ICT4M 2016 : proceedings : 22-24 November 2016, Jakarta, Indonesia*, 177–182. 2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M), Jakarta, Indonesia. 11/22/2016 - 11/24/2016. Los Alamitos, CA: Conference Publishing Services, IEEE Computer Society. doi: 10.1109/ICT4M.2016.045.

Jiang, Peiyuan, Daji Ergu, Fangyao Liu, Ying Cai, and Bo Ma. 2022. A Review of Yolo Algorithm Developments. *Procedia Computer Science* 199:1066–1073. doi: 10.1016/j.procs.2022.01.135.

Jorgensen, Paul C. 2018. *Software Testing: A Craftsman's Approach.* CRC Press.

Kong, Pingfan, Li Li, Jun Gao, Timothée Riom, Yanjie Zhao, Tegawendé F. Bissyandé, and Jacques Klein. 2021. ANCHOR: locating android framework-specific crashing faults. *Automated Software Engineering* 28 (2): 1–31. doi: 10.1007/s10515-021-00290-1.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton. 2012. ImageNet classification with deep convolutional neural networks. *Communications of the ACM* 60 (6). doi: 10.1145/3065386.

Lecun, Y., B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. 1989. Backpropagation Applied to Handwritten Zip Code Recognition. *Neural Computation* 1 (4): 541–551. doi: 10.1162/neco.1989.1.4.541.

Lecun, Y., L. Bottou, Y. Bengio, and P. Haffner. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86 (11): 2278–2324. doi: 10.1109/5.726791.

Li, Chuyi, Lulu Li, Hongliang Jiang, Kaiheng Weng, Yifei Geng, Liang Li, Zaidan Ke, Qingyuan Li, Meng Cheng, Weiqiang Nie, Yiduo Li, Bo Zhang, Yufei Liang, Linyuan Zhou, Xiaoming Xu, Xiangxiang Chu, Xiaoming Wei, and Xiaolin Wei. 2022. *YOLOv6: A Single-Stage Object Detection Framework for Industrial Applications.* arXiv.

Lin, Tsung-Yi, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. 2014. Microsoft COCO: Common Objects in Context, 740–755. European Conference on Computer Vision. Springer, Cham. doi: 10.1007/978-3-319-10602-1_48.

Lin, Tsung-Yi, Piotr Dollar, Ross Girshick, Kaiming He, Bharath Hariharan, and Serge Belongie. 2017. Feature Pyramid Networks for Object Detection. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. doi: 10.1109/cvpr.2017.106.

Liu, Hui, and Hee Beng Kuan Tan. 2009. Covering code behavior on input validation in functional testing. *Information and Software Technology* 51 (2): 546–553. doi: 10.1016/j.infsof.2008.07.001.

Luo, Lu. 2001. *Software testing techniques: Technology Maturation and Research Strategy*.

Masi, Emiliano, Giovanni Cantone, Manuel Mastrofini, Giuseppe Calavaro, and Paolo Subiaco. 2013. Mobile Apps Development: A Framework for Technology Decision Making, 64–79. International Conference on Mobile Computing, Applications, and Services. Springer, Berlin, Heidelberg. doi: 10.1007/978-3-642-36632-1_4.

Material Theme Builder. n.d. https://m3.material.io/theme-builder#/custom. Accessed 5 December 2022.

Mercioni, Marina Adriana, and Stefan Holban. 2020. The Most Used Activation Functions: Classic Versus Current. In *2020 International Conference on Development and Application Systems (DAS)*, 141–145. 2020 International Conference on Development and Application Systems (DAS), Suceava, Romania. 5/21/2020 - 5/23/2020. [S.l.]: IEEE. doi: 10.1109/DAS49615.2020.9108942.

Mitchell, Tom M. 1997. *Machine Learning.* New York, London: McGraw-Hill.

Moore, Samuel K., David Schneider, and Eliza Strickland. 2021. How Deep Learning Works. *IEEE Spectrum*.

Myers, Glenford J., Corey Sandler, and Tom Badgett (eds.). 2012. *The art of software testing*, 3rd edn. Hoboken, N.J.: John Wiley & Sons.

Nass, Michel, Emil Alégroth, and Robert Feldt. 2021. Why many challenges with GUI test automation (will) remain. *Information and Software Technology* 138:106625. doi: 10.1016/j.infsof.2021.106625.

Nidhra, Srinivas. 2012. Black Box and White Box Testing Techniques - A Literature Review. *International Journal of Embedded Systems and Applications* 2 (2): 29–50. doi: 10.5121/ijesa.2012.2204.

OpenCV. 2022. Home - OpenCV. https://opencv.org/. Accessed 5 December 2022.

O'Regan, Gerard. 2019. *Concise guide to software testing.* Cham, Switzerland: Springer.

O'Shea, Keiron, and Ryan Nash. 2015. *An Introduction to Convolutional Neural Networks.* arXiv.

Rafi, Dudekula Mohammad, Katam Reddy Kiran Moses, Kai Petersen, and Mika V. Mantyla. 2012. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *2012 7th International Workshop on Automation of Software Test (AST): Proceedings : June 2-3, 2012, Zurich, Switzerland*, 36–42. 2012 7th International Workshop on Automation of Software Test (AST), Zurich, Switzerland. 6/2/2012 - 6/3/2012. Piscataway, N.J.: IEEE. doi: 10.1109/IWAST.2012.6228988.

Ready for AI. 2018. The secret of the Chinese team winning the Microsoft COCO Challenge. https://readyforai.com/article/the-secret-of-the-chinese-team-winning-the-microsoft-coco-challenge/. Accessed 5 December 2022.

Redmon, Joseph, and Ali Farhadi. 2017. YOLO9000: Better, Faster, Stronger. 2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 2017/07. IEEE. doi: 10.1109/CVPR.2017.690.

Redmon, Joseph, Santosh Divvala, Ross Girshick, and Ali Farhadi. 2016. You Only Look Once: Unified, Real-Time Object Detection. 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 2016/06. IEEE. doi: 10.1109/CVPR.2016.91.

Ren, Shaoqing, Kaiming He, Ross Girshick, and Jian Sun. 2017. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39 (6): 1137–1149. doi: 10.1109/tpami.2016.2577031.

Ruiz, A., and Y. W. Price. 2007. Test-Driven GUI Development with TestNG and Abbot. *IEEE Software* 24 (3): 51–57. doi: 10.1109/MS.2007.92.

Shao, Danhua, Sarfraz Khurshid, and Dewayne E. Perry. 2007. A Case for White-box Testing Using Declarative Specifications Poster Abstract. In *Testing Academic and Industrial Conference--Practice And Research Techniques: (TAIC PART 2007) co-located with Mutation 2007 proceedings 10th-14th September, 2007, Cumberland Lodge, Windsor, United Kingdom*, 137. Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007), Windsor, UK. 9/10/2007 - 9/14/2007. Los Alamitos Calif.: IEEE Computer Society. doi: 10.1109/TAIC.PART.2007.36.

StatCounter Global Stats. 2022. Mobile Operating System Market Share Worldwide | Statcounter Global Stats. https://gs.statcounter.com/os-market-share/mobile/worldwide. Accessed 5 December 2022.

Statista. 2022a. Smartphone users 2026 | Statista. https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/. Accessed 5 December 2022.

Statista. 2022b. Time spent on average on a smartphone in the U.S. 2021 | Statista. https://www.statista.com/statistics/1224510/time-spent-per-day-on-smartphone-us/. Accessed 5 December 2022.

Szegedy, Christian, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. 2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 2015/06. IEEE. doi: 10.1109/CVPR.2015.7298594.

Szeliski, Richard. 2022. *Computer Vision: Algorithms and Applications*, 2nd edn. Cham: Springer International Publishing; Springer.

Uijlings, J. R. R., K. E. A. van de Sande, T. Gevers, and A. W. M. Smeulders. 2013. Selective Search for Object Recognition. *International Journal of Computer Vision* 104 (2): 154–171. doi: 10.1007/s11263-013-0620-5.

Wang, Chien-Yao, Alexey Bochkovskiy, and Hong-Yuan Mark Liao. 2022. *YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors.* arXiv.

Weights & Biases – Developer tools for ML. 2022. https://wandb.ai/site. Accessed 5 December 2022.

Yeh, Tom, Tsung-Hsiang Chang, and Robert C. Miller. 2009. Sikuli. In *Proceedings of the 22nd annual ACM symposium on User interface software and technology - UIST '09*, 183. the 22nd annual ACM symposium, Victoria, BC, Canada. 04/10/2009 - 07/10/2009. New York, New York, USA: ACM Press. doi: 10.1145/1622176.1622213.

Zeiler, Matthew D., and Rob Fergus. 2014. Visualizing and Understanding Convolutional Networks, 818–833. European Conference on Computer Vision. Springer, Cham. doi: 10.1007/978-3-319-10590-1_53.

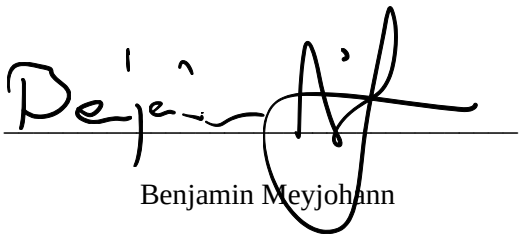
Zhang, Aston, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*.

Zhou, Xinyu, Cong Yao, He Wen, Yuzhi Wang, Shuchang Zhou, Weiran He, and Jiajun Liang. 2017. EAST: An Efficient and Accurate Scene Text Detector. In *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. IEEE. doi: 10.1109/cvpr.2017.283.

Zohud, Tasnim, and Samer Zein. 2021. Cross-Platform Mobile App Development in Industry: A Multiple Case-Study. *1727-6209* 46–54. doi: 10.47839/ijc.20.1.2091.

Assertion

Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.



Karlsruhe, December 9, 2022 Benjamin Meyjohann