DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Implementing a mobile app for object detection

David Drews

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Implementing a mobile app for object detection

Entwicklung einer mobilen App zur Objekterkennung

Author:	David Drews
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Severin Reiz, M.Sc.
Submission Date:	15th of August 2021

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

Munich, 15th of August 2021 David Drews

Abstract

We migrate the entire code base of the Android application TUM-Lens from Java to Kotlin. This facilitates the future development of the app as it makes the code more concise and error-proof. Consequently, we elaborate on further advantages of the Kotlin language over Java and analyse how this migration lowered the lines of the existing code. Moreover, we expand the functionalities of the app by an object detection feature based on Google's open source deep learning framework TensorFlow Lite. The implementation follows in the previous TUM-Lens developer's footsteps and integrates the object detection to entirely work on-device so that no data needs to be exchanged with external servers. On the object detection theory side, we distinguish object detection from other visual machine learning tasks and survey a selection of modern deep learning architectures - both for backbone and detector networks. In addition, we study the mechanics of a specific model in detail, the SSD MobileNet v1, as this is the model applied to the object detection task in TUM-Lens. This thesis expands Maximilian Jokel's previous work *Implementing a TensorFlow-Slim based Android app for image classification* (2020).

Contents

Abstract **vii**

I. Introduction and Background Theory **1**

1. Motivation **2**
1.1. Growing Support for On-Device Machine Learning 2
1.2. Offline Usability 2
1.3. Improved Privacy 3

2. Important Concepts **4**
2.1. Machine Learning 4
2.1.1. Supervised Learning 5
2.1.2. Unsupervised Learning 5
2.1.3. Semi-Supervised Learning 5
2.1.4. Reinforcement Learning 5
2.2. Artificial Neural Networks 6
2.3. Deep Learning 6

3. Object Detection **7**
3.1. How Object Detection Differs From Related Tasks 7
3.1.1. Semantic Segmentation 7
3.1.2. Image Classification 7
3.1.3. Object Detection 8
3.1.4. Instance Segmentation 8
3.2. Object Detection Frameworks 8
3.2.1. Backbones 8
3.2.2. Two-Stage Detectors 9
3.2.3. R-CNN 10
3.2.4. Single-Stage Detectors 10
3.3. SSD MobileNet v1 10
3.3.1. MobileNet Architecture 11
3.3.2. Introduction to SSD 11
3.3.3. The SSD Head 11
3.3.4. Architecture 12
3.3.5. Latest Improvements 12

II. App Development 13

4. Previous State of the Application 14

5. Migration From Java to Kotlin 15
5.1. Advantages of Kotlin . . . 15
5.2. Converting Java Files to Kotlin Files . . . 15
5.3. Measuring The Decrease of The Size of The Code Base . . . 17

6. Implementing Object Detection Using The TensorFlow Lite Framework 19
6.1. Expansion of The Existing App Architecture . . . 19
6.2. DetectionActivity . . . 21
6.2.1. Inheritance . . . 21
6.2.2. Bitmap Conversion . . . 21
6.2.3. Detection and Tracking . . . 21
6.3. Dive Into . . . 22
6.4. Object Detection . . . 22
6.5. Implementation Fun . . . 22
6.6. Next steps in the development of TUM-Lens . . . 22

III. Results 23

7. Evaluation 25
7.1. Performance . . . 25
7.2. Accuracy . . . 25

8. Outlook 26
8.1. Possible Applications . . . 26
8.1.1. Privacy Inspired Use Cases . . . 26
8.1.2. Autonomy and Speed Inspired Use Cases . . . 27
8.2. Future Work . . . 27

A. Screenshots of the Application 29

B. Test Device Specifications 30

Bibliography 33

Acronyms 37

Glossary 38

Part I.

Introduction and Background Theory

1. Motivation

The aim of this work was to further develop the Android app TUM-Lens [17]. Its core function is the analysis of images that are captured by the camera of the Android device and transmitted to the app as a live feed. With the completion of this work, the pre-existing image classification capabilities of the app are now complemented with object detection.

For an optimal user experience, the analysis of the images must take place in near real time. This is the only way to ensure that the analysis results displayed always match the current content of the camera feed, which can change very quickly due to panning of the camera by its user. While in many applications the analysis of image data can take place decentrally in powerful data centres, in the case of TUM-Lens the image analysis runs on the mobile device itself.

1.1. Growing Support for On-Device Machine Learning

Support for the development of machine learning (ml) and also in particular deep learning applications for mobile platforms is growing steadily and from different directions at the same time. Developer-friendly frameworks such as TensorFlow[1], developed by Google Brain, or PyTorch[2], developed by Facebook's AI Research Lab, are among the best-known deep learning frameworks [11]. The releases of TensorFlow Lite[3] 2017 [43] and PyTorch Mobile[4] 2019 [32] show that mobile platforms increasingly come into focus of companies providing machine learning software. In recent years, device manufacturers and operating system developers also started to provide dedicated hardware and software components for mobile machine learning. Examples include Apple's Neural Engine [44], unveiled in 2017, or Android's Neural Networks API (NNAPI) [6]. Apple's Neural Engine is a hardware component optimised for machine learning requirements. Android's NNAPI, on the other hand, is an Android C application programming interface (api) for efficient computation of ml operations and provides a basic set of functions for higher-level ml frameworks. As a result of these developments, it is becoming easier for developers to build ml applications that run efficiently on mobile devices. This support was a major catalyst for the initial and further development of TUM-Lens in the context of two bachelor theses.

1.2. Offline Usability

TUM-Lens is more independent compared to many other machine learning based apps as it does not require an internet connection to use it. Often, apps and services require a

[1] `https://www.tensorflow.org/`
[2] `https://pytorch.org/`
[3] `https://www.tensorflow.org/lite`
[4] `https://pytorch.org/mobile/home/`

connection to the internet because of the nature of the task they are meant to perform. The Amazon voice assistant Alexa can answer simple voice commands to control smart home devices or check the time without having access to the world wide web and thus already uses on-device machine learning. But even if Alexa could analyse and understand all voice commands locally, the request would still have to be forwarded to the Amazon servers in most cases. Due to the large number of possible queries, not all answers can be kept on the device, but must be retrieved from a data centre that has more storage capacity. Such queries include daily topics such as the weather report, traffic or the result of a sporting event. However, an internet connection is not required to use the full range of functions of TUM-Lens. All the information needed for image classification and object detection is stored locally on the device in the form of various already trained artificial neural network (ann). With the integration of the corresponding mobile frameworks, the image analysis can therefore be carried out locally on the device, making the app independent of an internet connection.

1.3. Improved Privacy

The use of on-device ml provides a further mechanism for protecting personal data in the context of machine learning in addition to existing methods such as differential privacy. Due to the growing support for mobile ml applications mentioned above, but also due to the continually increasing power of mobile devices [12], not only the use of pre-trained anns becomes possible, but also the training of new anns on the mobile device itself becomes more and more feasible and relevant [27]. If the training process takes place locally on the device itself, no data needs to be transferred to external instances such as a company's servers. This makes it possible to develop applications that adapt more and more individually to the user as they are used, while guaranteeing a maximum level of data protection. An example of the development of such an application is DeepType [46]. DeepType attempts to predict the next word used when the user enters the keyboard. While every user initially starts with the same pre-trained version of the ann used by DeepType, the application continues to train this ann with each input and thus adapts more and more to the characteristic input behaviour of the user - all without the text inputs ever leaving the device.

2. Important Concepts

Everyone is talking about machine learning. It is already impossible to imagine our everyday life without the use of the term. Due to the multitude of contexts in which machine learning is spoken of, some justifiably and some unjustifiably, it is important to create a common understanding of some machine learning-related concepts in order to ensure a common understanding of the theoretical contents of this work.

2.1. Machine Learning

A popular definition of ml is attributed to Arthur Samuel describing it as the "field of study that gives computers the ability to learn without being explicitly programmed"[1]. machine learning algorithms circumvent this need for explicit programming by improving an internal model through data. This process is called training and the data used to train the model is often regarded to as the model's experience [29]. As depicted in figure Figure 2.1, ml can be divided into the subfields supervised learning, unsupervised learning, semi-supervised learning and reinforcement learning.

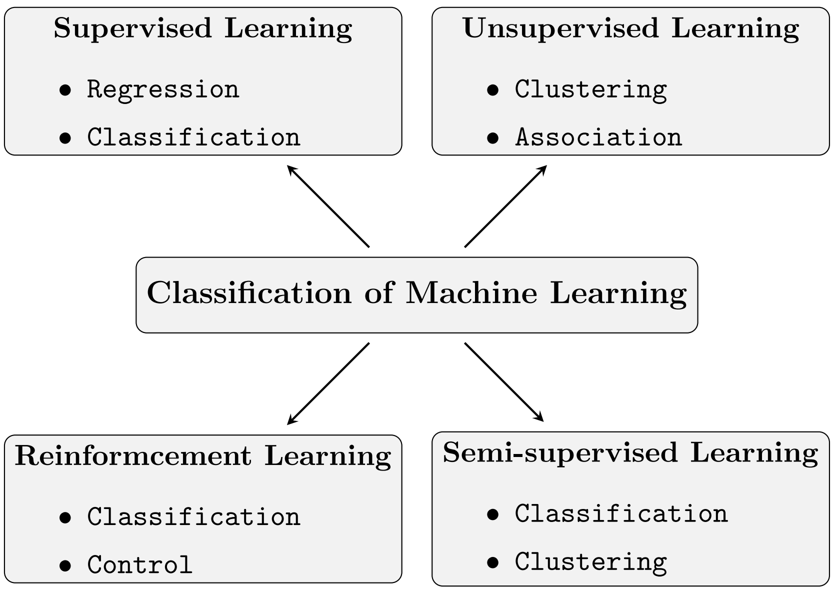


Figure 2.1.: The field of machine learning divided into subfields by the characteristics of the underlying learning process. Also indicates the learning problems that are typically tried to be solved by applying the respective learning process.

[1] Althought cited in popular machine learning material like Andrew Ng's ml course at Stanford [30] the quote appears neither in Samuel's 1959 [35] nor his 1967 paper [36].

2.1.1. Supervised Learning

In supervised learning, the learning machine is provided with input data as well as the output that is expected for the given input [21]. In the classical case of spam filtering, the input can be a collection of emails and the expected output is a label attached to each email that either classifies it as spam or as non-spam. The learning machine is then fed all e-mails as input data and learns to recognise which information in the input is important to produce the correct classification. As the system knows the correct answer for each training input, it can process an email, predict whether or not it is spam, and then use the known answer to change its weights in a way that will make it more likely to lead to a correct prediction and less likely to lead to a false prediction the next time it is presented with similar input.

2.1.2. Unsupervised Learning

Detecting hidden patterns and structuring data is where unsupervised learning comes into play. Learners of this type don't need to be provided with an expected output while being trained [37]. A scenario for the application of unsupervised learning is the problem of dividing a customer base into subgroups in order to treat every subgroup according to their specific needs. An employee might help the machine learning system by providing the number of subgroups she wants the system to generate. The learner then builds up its representation of the internal structure of the entire data set with every input it processes. After having processed enough customers, it will most likely have identified the key metrics that distinguish customers into the different groups.

2.1.3. Semi-Supervised Learning

One use for semi-supervised learning is cluster analysis, which was already used as an example in the previous paragraph. In the case of semi-supervised learning, the system no longer has to work out the different groups (also known as *clusters*) from the unlabelled data alone. Instead, it can use a small set of already labelled customers as a reference and build its internal representation of the entire dataset (labelled and unlabelled) around the clusters indicated by the pre-labelled data. This is especially useful because in many domains collecting or creating labelled data is difficult, expensive, or both [48].

2.1.4. Reinforcement Learning

Reinforcement learning is "learning what to do - how to map situations to actions - so as to maximize a numerical reward signal" [41]. Systems are trained via reinforcement learning to learn how to behave in dynamic environments. The tasks in these environments can stretch from playing a video game [45] to driving an autonomous car [34]. These exemplary tasks show two characteristics that distinguish reinforcement learning from the other subfields of ml: The reward signal is often delayed and attribution to single actions is difficult. Only once a game is won or the car has arrived safely at its destination the system knows if all the decisions it made along the way lead to a positive outcome. *Trial-and-error* is therefore a term that summarises this learning paradigm quite precisely.

2.2. Artificial Neural Networks

An artificial neural network - often just referred to as neural network - is a data processing concept that is inspired by biological neurons and their interconnectivity. As figures Figure 2.2 and Figure 2.3 show the artificial neurons (also called *nodes*) in an ann are grouped in *layers*. There are three important types of layers: The *input layer*[2], the *output layer* and an arbitrary number of *hidden layers* in between the input and output layer. Similar to neurons in human brains, nodes of different layers can be connected. In anns, the nodes exchange signals in the form of numbers. Each node outputs a number that is computed by applying a non-linear function to its inputs. The output signal can then be a new input for other nodes or it can be part of the result returned by the output layer. The connections between nodes are also known as *edges* and typically carry a weight. In the case of anns, the training process that is typical for all machine learning systems is the adjustment of these connection weights. The weights and other variables of the ann are grouped under the term *parameters*. In summary, an ann transforms an input vector into an output vector through a series of non-linear functions, where both the calculation of the output and the training process are characterised by the specific structure of the ann and its parameters.

2.3. Deep Learning

Deep learning is a subarea of machine learning. Deep learning is characterised by the use of anns with many hidden layers. The more hidden layers a network has, the deeper it is. The deeper a network is and the more nodes the network has per layer, the more complex the computations that the ann can successfully perform [10]. As the number of layers and nodes grows, so does the number of parameters. Their large number is the reason deep learning requires extensive amounts of data to provide adequate results compared to other sub-disciplines of machine learning. Networks of this genus have been given the ability to perform extraordinarily complex computations at the expense of a resource-intensive training process.

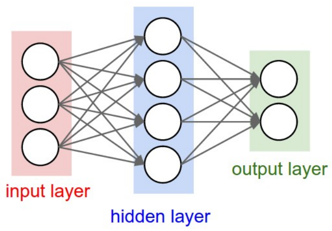


Figure 2.2.: 2-layered ann. It is called fully connected as every node from the previous layer is connected to every node in the next layer.
Source: [22]

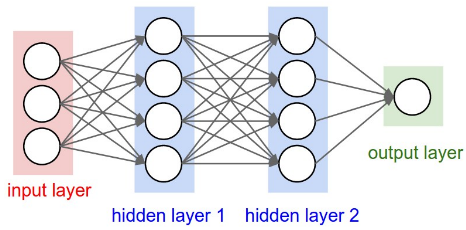


Figure 2.3.: 3-layered ann. In anns, nodes in one layer are connected to nodes in other layers but not to other nodes in the same layer.
Source: [22]

[2]Note that the input layer is not counted towards the total number of layers in an ann.

3. Object Detection

3.1. How Object Detection Differs From Related Tasks

The field of computer vision (cv) encompasses numerous distinct problems and an even larger number of potential solutions. In the following, object detection as a typical task in the cv context is distinguished from other cv tasks that are closest to it in terms of learning objectives.

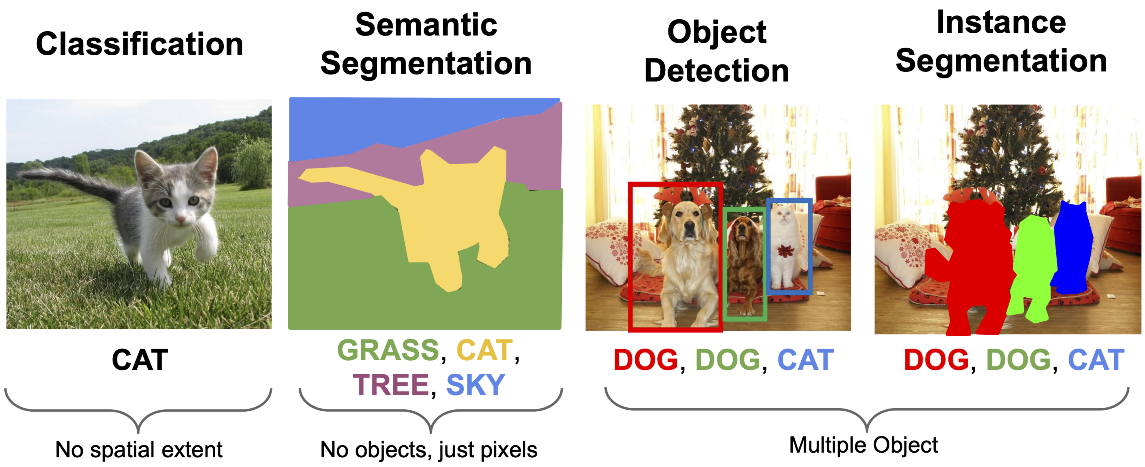


Figure 3.1.: Object detection differs conceptually from other related cv tasks regarding spatial information, the concept of objects and the number of detections in a scene.
Source: [23]

3.1.1. Semantic Segmentation

In semantic segmentation, each pixel of an image is assigned a class. However, there are no objects. This means that if there are several objects of the same class in the image, all the associated pixels receive the same class label. Therefore, the different objects cannot be differentiated based on the result of the semantic segmentation.

3.1.2. Image Classification

In image classification, the result of the detection is a single class. In rare application variants, a bounding box for one object of the detected class is also returned - as a rule, however, no spatial extent is associated with the task of classification.

3.1.3. Object Detection

Object detection deals with the identification of any number of objects within an image. For each object, a class label and its position are returned as the coordinates of a rectangle enclosing the object. It is important here that, as depicted in Figure Figure 3.1, several objects of the same class can also be recognised. In contrast to the task of semantic segmentation, the different objects of the same class can be distinguished.

3.1.4. Instance Segmentation

The instance segmentation essentially fulfils a better variant of the object detection. Again, several objects of different classes are detected and the positions of the classes are also part of the output. However, the positions are not marked by bounding boxes as in object detection, but each pixel belonging to an object receives a label of this object. The objects are therefore even more sharply separated from the image areas that do not contain an object and, in contrast to semantic segmentation, individual objects of the same class remain distinguishable.

3.2. Object Detection Frameworks

State of the art object detection frameworks run predominantly on deep learning architectures [2]. The bachelor's thesis preceding this work, *Implementing a TensorFlow-Slim based Android app for image classification* [18], already explains how convolutional neural networks (cnns) work and why they are the most important building blocks for deep learning frameworks solving perceptual tasks. This explanation will therefore not be pursued here once more. Instead, we survey a range of modern object detection frameworks. Each framework is defined by the combination of a backbone and a detector. The selection was not put together at random but is based on popularity and performance in standard object detection benchmarks [4].

3.2.1. Backbones

The term *backbone* in the context of anns might be used differently depending on the task pursued. In the case of visual tasks and this thesis, the backbone is the part of an object detection framework that extracts features from the input data. This encapsulation of the feature extraction task in its own set of cnns allows the designer of the object detection pipeline to swap and test different backbones for the task at hand [4].

backbone	first publication	detectors
AlexNet	2012 [20]	HyperNet [19]
VGG-16	2014 [39]	PFPNet-R512 [K]
GoogLeNet	2014 [42]	YOLOv1 [33]
ResNets	2015 [13]	BlitzNet512 [8], CoupleNet [49], RetinaNet [26], Faster R-CNN [A], R-FCN ?
Inception-ResNet-V2	tbd	Faster R-CNN G-RMI [23], Faster R-CNN with TDM [24]
DarkNet-19	tbd	YOLOv2 [J]
MobileNet	2017 [16]	SSDv2

Table 3.1.: Overview of modern cnn backbone networks and detectors that build upon them.

TO-DO: further sources:
https://scholar.google.com/scholar?q=DaiJ: http://arxiv.org/abs/1612.08242

AlexNet

ResNet

While the baseline ann described in Section 2.2 only connects adjacent layers, residual neural networks (ResNets) are a special subclass of ann that introduces *shortcut connections* [13]. Shortcut connection are edges in the network that skip one or more layers.

MobileNet

As the name suggests, MobileNets were developed especially for mobile or embedded computer vision applications. The key innovation of the MobileNet architecture is the combination of two elements:

1. utilisation of depth-wise separable filters [38]
2. a network structure that enables the developer to scale the model size down by adjusting only two hyper-parameters

As a MobileNet is a building block of the detection framework implemented in TUM-Lens, a more detailed description can be found in Subsection 3.3.1.

3.2.2. Two-Stage Detectors

This class of detectors separates the object detection task into two distinct stages [40]. The network of the first stage (named Region Proposal Network (RPN) in the context of the R-CNN detector family) uses the image data to generate region proposals. The second stage is a separate network that takes these region proposals (or ROI - short for regions of interest), potentially decreases the final number of ROI using mechanics that depend on the specific detector, and then performs the classification on each of the final regions. The classified regions are then returned as the detected objects. Two-stage detectors tend to have a higher localization and object recognition accuracy than single-stage detectors but can only achieve that at the cost of considerably slower inference speed [14]. Table Table 3.1 shows a selection of popular detectors including R-CNN and some of its many successors, Fast-CNN and Faster-CNN.

3.2.3. R-CNN

R-CNN was Fast, Faster R-CNN, and other advancements like Mask-R-CNN 2014 [9]

3.2.4. Single-Stage Detectors

Detection frameworks are considered to have only a single stage when they consist of one deep neural network only and compute the objects (bounding box coordinates and category) in a single pass through that network. By eliminating the explicit generation of region proposals, single-stage detectors outperform their two-stage competitors with respect to speed while sacrificing a bit of detection accuracy, if at all [28]. This speed improvement makes single-stage detectors the preferred choice for applications running on mobile or embedded devices or in applications that require real-time image detection. Since the object detector implemented in TUM-Lens is described thoroughly in Section 3.3, the following overview of selected single-state detectors will help us to understand how the app's object detector differs from other possible options.

RetinaNet

2018

YOLO

YOLO (You Only Look Once) ; [3]

3.3. SSD MobileNet v1

The SSD MobileNet v1 is the TensorFlow-Lite model used for the new object detection functionality integrated into TUM Lens. The model is pre-trained on the COCO dataset[1] and available on TensorFlow Hub[2]. Following the data flow through the object detection framework, we will first have a closer look at the architecture of the MobileNet used as the backbone network. Secondly, we introduce the core concepts of the Single Shot MultiBox Detector (SSD) followed by an detailed description of the SSD architecture. Finally, we conclude with an outlook on the latest improvements to MobileNets, SSDs and their combined applications.

[1] https://cocodataset.org/

[2] https://tfhub.dev/tensorflow/lite-model/ssd_mobilenet_v1/1/metadata/2

3.3.1. MobileNet Architecture

TBD More Intro
Firstly, using depth-wise separable filters leads to a significantly lower number of parameters the model needs to learn during the training phase. Secondly, the easy shrinking of the model by adjusting the two hyper-parameters (named *width multiplier* and *resolution multiplier*) allows the model-builder to scale the model down to exactly the size that is appropriate for solving the problem the model is applied to [16].

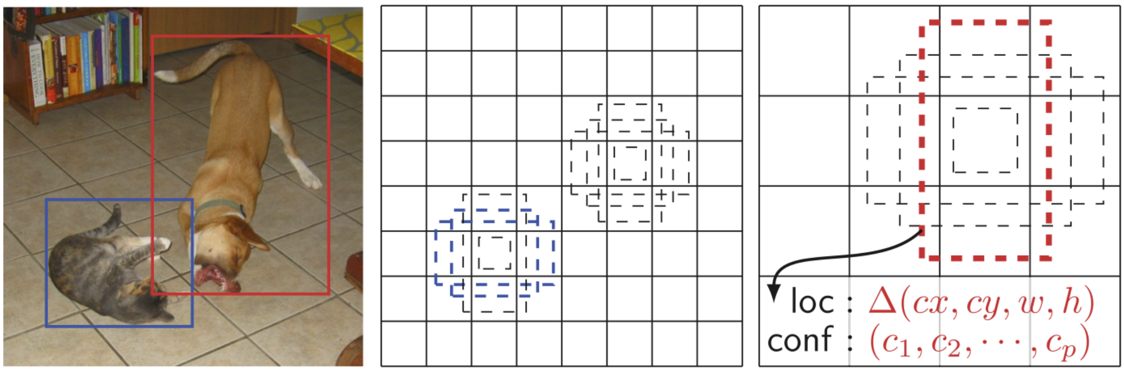
3.3.2. Introduction to SSD

The SSD consists of a backbone and the *SSD head*. In theory, the backbone can be any of the networks mentioned in Subsection 3.2.1 although the choice is not limited to this selection. When SSD was first introduced in 2015, the authors used the VGG-16 network as the backbone [28]. In the case of the specific SSD variant implemented in TUM-Lens, MobileNet is used as a backbone. The reasons for the particular applicability of MobileNet in our Android app use case are described in Subsection 3.2.1. Independent of the specific type of backbone, it is first trained on publicly available image data, e.g. from a database like ImageNet[3]. Once the network is pre-trained the final layers that originally handled the classification are then replaced by the SSD head.

3.3.3. The SSD Head

The SSD head is the part of the detector that computes category scores and box locations. It is a characteristic of SSD to work with a fixed set of default bounding boxes. To account for the different sizes in which objects might appear in an image, SSD applies a sequence of comparatively small convulational filters to the feature map returned by the backbone network. Each of the feature maps obtained by this process represents a different receptive field and allows the network to detect objects at different scales. Consequently, SSD computes predictions for all of these feature maps and for each of its predefined aspect ratios. It then fuses the predections of the different aspect ratios and scales in order to improve detection accuracy. The range of aspect ratios and scales, although predeterminded, enables the SSD to detect object in various shapes and sizes. Its multi-scale feature maps are a core differentiator from other single-stage detectors like YOLO [28, 33] .

[3] `https://image-net.org/`



(a) Image with GT boxes (b) 8×8 feature map (c) 4×4 feature map

Figure 3.2.: The combination of bounding boxes in varying aspect ratios and multi-scale feature maps allows the SSD to detect objects in different sizes and orientations. Source: [28]

3.3.4. Architecture

3.3.5. Latest Improvements

Part II.

App Development

4. Previous State of the Application

The practical part of this work was built in top of the already existing Android application TUM-Lens in its version 1.0. The app is already cable of classifying images received from the live camera stream in real-time. It can also classify images that are loaded from the disk of the Android as an alternative operational mode to the camera stream classification. Moreover, the user can choose between different TensorFlow-Slim[1] models to classify the images. This enables the user to do two things: she can compare the speed of similar detectors and she can also change the type of objects that can be detected as some of the available networks were trained on different data [18] and with different class labels. The most impactful design decision of the former developer of TUM-Lens was to run the entire classification locally on the Android device.

[1] `https://github.com/google-research/tf-slim`

5. Migration From Java to Kotlin

During the Google I/O conference 2017 Google announced that they are making the programming language Kotlin a first-class citizen in Android [7]. Two years later, Google refined this statement annunciating that Android development will "be increasingly Kotlin-first" and, at the present day, the Google Developers website recommends developers to choose Kotlin when they start building a new Android app [5]. This alone can be reason enough to migrate an application to Kotlin - especially as long as its code base is still as manageable as the code base of TUM-Lens. To provide further explanations for the soundness of such a step we will first look at the advantages of Kotlin over Java - with particular emphasis on the context of TUM-Lens being developed by students as part of their final theses. After that, we look at the code conversion process from Java to Kotlin which is backed by powerful functionality integrated into Android Studio. We will also see an example for a pitfall that can be easily overlooked in such a semi-automated conversion process. Finally, we analyse how the number of lines of code changed from the Java files in app version 1.0 compare to their current state being fully migrated to Kotlin.

5.1. Advantages of Kotlin

Continuing the development of TUM-Lens in Kotlin has several advantages. As the is app currently exclusively developed by students, Kotlin's enhanced expressiveness and conciseness over Java makes it easier for fellow students to understand the entire application within the scope of a bachelor's or master's thesis. Moreover, students can learn more in the given amount of time they spend working on the app because they can focus more strongly on the implementation of new features instead of having to write boilerplate Java code. As students are more error prone than professional developers they surpassingly benefit from the safety features integrated into the Kotlin language. The most common cause for crashes of Java applications is the null pointer exception [15, 47]. With Kotlin being a strongly typed language a lot of these null pointer exceptions will be avoided because Kotlin's safety features help the developer to identify potential sources of such errors very easily. Google itself claims that Android apps are 20% less likely to crash when they contain Kotlin code [5].

5.2. Converting Java Files to Kotlin Files

Android Studio offers a convenient way to convert files from Java to Kotlin. On macOS, we first have to open the Java file we want to migrate in Android Studio. Next, we select *Code* from the menu bar and then the menu item *Convert Java File to Kotlin File*. If we have not yet configured Kotlin in your project, Android Studio will prompt us to ether do so or abort the migration process. Once Kotlin is set up Android Studio will immediately start the conversion. At the end of the process we are prompted with a dialogue asking "Some code in the rest of your project may require corrections after performing this conversion. Do you want to find such code and correct it too?" which we confirm by clicking on the yes-button. They file has now been converted from Java to Kotlin. In some cases the resulting Kotlin code works right away but in the context of TUM-Lens manual adjustments were mandatory in every file because of several reasons.

For a start, Android Studio could not always infer all types automatically. With Kotlin being strongly typed it was necessary to restructure the code so that the compiler could know each variable's type or safely cast a variable to the appropriate type (called *Smart Cast* in Kotlin). We solved this situation using two different approaches distinguished by the importance of the successful execution of the respective code block. In the case of non-critical functionality, using Kotlin's safe call operator `?.` to access properties that might be null was sufficient. When it came to core functions, wrapping such functionality in an additional safety check was the better option. This not only capacitates the compiler to be certain that a mutable property had not accidentally become null before being accessed. It also enables us to react appropriately if the safety check fails.

```
var <propertyName >[: <PropertyType >] [= <property_initializer >]
  [<getter >]
  [<setter >]
```

Listing 5.1: Kotlin's declaration syntax for a mutable property. Getter and setter are optional. The property type is only optional when the compiler can infer it from the context (meaning either from the initializer or from the return type of the getter).

Apart from type inference, there was one part to Android Studio's built-in code migration that actually tricked us into introducing a bug: the conversion of getters and setters. Listing 5.1 shows the syntax for the declaration of a property in Kotlin. Every property has default public getters and setters if not explicitly defined otherwise. Java, on the other hand, conventionally defines methods like `public variableType getVariableName()` and `public void setVariableName(variableType newValue)` for every variable of a private class that shall be retrieved or overwritten from outside that class.

Listing 5.2 contains the original Java code and listing Listing 5.3 shows the final solution that implemented in addition to Kotlin's property accessors. The default Kotlin getter is not enough here as it would omit the crucial conversion task. However, this is exactly what happened during the auto-conversion. As a result, other parts of the logic broke because the `rgbBytes` object could not be properly processed which lead to the application not working as expected. In this special case some accessors of the property were in need of the image conversion and others were not. Therefore, we did not put the image conversion into a custom accessor because we did not want to call `imageConverter!!.run()` every time the property was used. Placing it in the separate method `getConvertedRgbBytes()` has therefore been a good decision for us.

```java
protected int[] getRgbBytes() {
  imageConverter.run();
  return rgbBytes;
}
```

Listing 5.2: Java code that lead to the introduction of a bug after using Android Studio's built-in Java to Kotlin converter.

```kotlin
protected fun getConvertedRgbBytes()
    : IntArray? {
  imageConverter!!.run()
  return rgbBytes
}
```

Listing 5.3: Substituting the wrongly placed default Kotlin accessors with this method solved the problem.

5.3. Measuring The Decrease of The Size of The Code Base

Using the npm distribution of the tool Count Lines of Code (cloc)[1] we can easily count lines of Java and Kotlin code while ignoring blank lines and comment lines. This enables us to monitor how the shift from Java to Kotlin influenced the number of lines of code. With Perl, Node and npm installed on our machine, we run the following command which prints its output to stdout:

```
(base) david@DDs-MBP lens-david % npx cloc --by-file --git-diff-rel --
    match-d='main' --include-lang=Java,Kotlin -csv f6a18e0f 031a26a7
```

Listing 5.4: npx cloc command with its options and arguments. Prints the lines of code analysis comapring files before and after the Java to Kotlin conversion. The output was also saved as cloc.csv and can be found in lens-david/thesis/raw_data.

The `--by-file` option enables us to directly compare each Java file with Kotlin equivalent as it changes the output so that the results are returned for every source file encountered. `--git-diff-rel` interprets the command line arguments as git targets and compares only files which have changed in either commit which is exactly what we need to measure the change in the number of lines of code. Only files in the main directory are of our concern so we let cloc only search this directory using the option `--match-d='main'`. This excludes e.g. the test directory from being searched. With `--include-lang=Java,Kotlin` we limit the output to files written in the languages we seek to collate. Other files like Android's XML layout files are not of interest for this analysis. Finally, we provide two git commits. `f6a18e0f` is the last commit pushed by the previous developer of TUM-Lens and `bf0dbf7f` is the more recent commit that does not contain Java files anymore but replaced them with their respective Kotlin substitute.

[1]Original cloc project by Al Danial: `https://github.com/AlDanial/cloc`
cloc npm distribution by Kent C. Dodds: `https://www.npmjs.com/package/cloc`

Files in TUM-Lens v1.0 migrated from Java to Kotlin	Delta in lines of code
CameraRoll	-34
Classifier	4
ListSingleton	-9
PermissionDenied	-8
StartScreen	-12
ViewFinder	-7
fragments/CameraRollPredictionsFragment	9
fragments/CameraSettingsFragment	-15
fragments/ModelSelectorFragment	5
fragments/PredictionsFragment	4
fragments/ProcessingUnitSelectorFragment	-7
fragments/SmoothedPredictionsFragment	6
fragments/ThreadNumberFragment	-6
helpers/App	1
helpers/CameraEvents	0
helpers/FreezeAnalyzer	-6
helpers/FreezeCallback	0
helpers/ImageUtils	43
helpers/Logger	6
helpers/ModelConfig	-2
helpers/ProcessingUnit	-2
helpers/Recognition	-26
helpers/ResultItem	-21
helpers/ResultItemComparator	-2
cumulative delta over all relevant files	**-79**

Table 5.1.: This table shows the results from the command line prompt in Listing 5.4. Packages have been indicated as a prefix to the file name to resemble the original project structure of TUM-Lens v1.0 and file extension have been omitted. Overall, the total size of the codebase shrank because of the migration from Java to Kotlin. This is particularly impressive as understandably further logic was added to existing classes in order to account for the new object detection functionality.

6. Implementing Object Detection Using The TensorFlow Lite Framework

As described in Chapter 4 the core feature of TUM-Lens v1.0 is image classification. Most notably, the classification task is not outsourced to some remote server but performed offline on the device itself. We adhere to this decision and built our logic on top of Google's TensorFlow Lite example app for object detection [24, 25]. The detection task is therefore carried out locally using the same TensorFlow-Lite API as the image classification. We also integrate our detection results into the images received from the camera live-stream so that the user of the app can experience the object detection in real-time. This being said, there is some discrepancy to the existing set of functionalities built around the image classification. Only one model for object detection is currently available within the app and the detection logic cannot be applied to images that have been loaded from the storage of the device.

6.1. Expansion of The Existing App Architecture

In order to make the structure of the app more accessible to new developers we put the core classes responsible for image classification and object detection into two distinct packages. The project now includes the four packages `classification`, `detection`, `fragments` and `helpers`. Only the two base activities `StartScreenActivity` and `PermissionDeniedActivity` remained on the top-level.

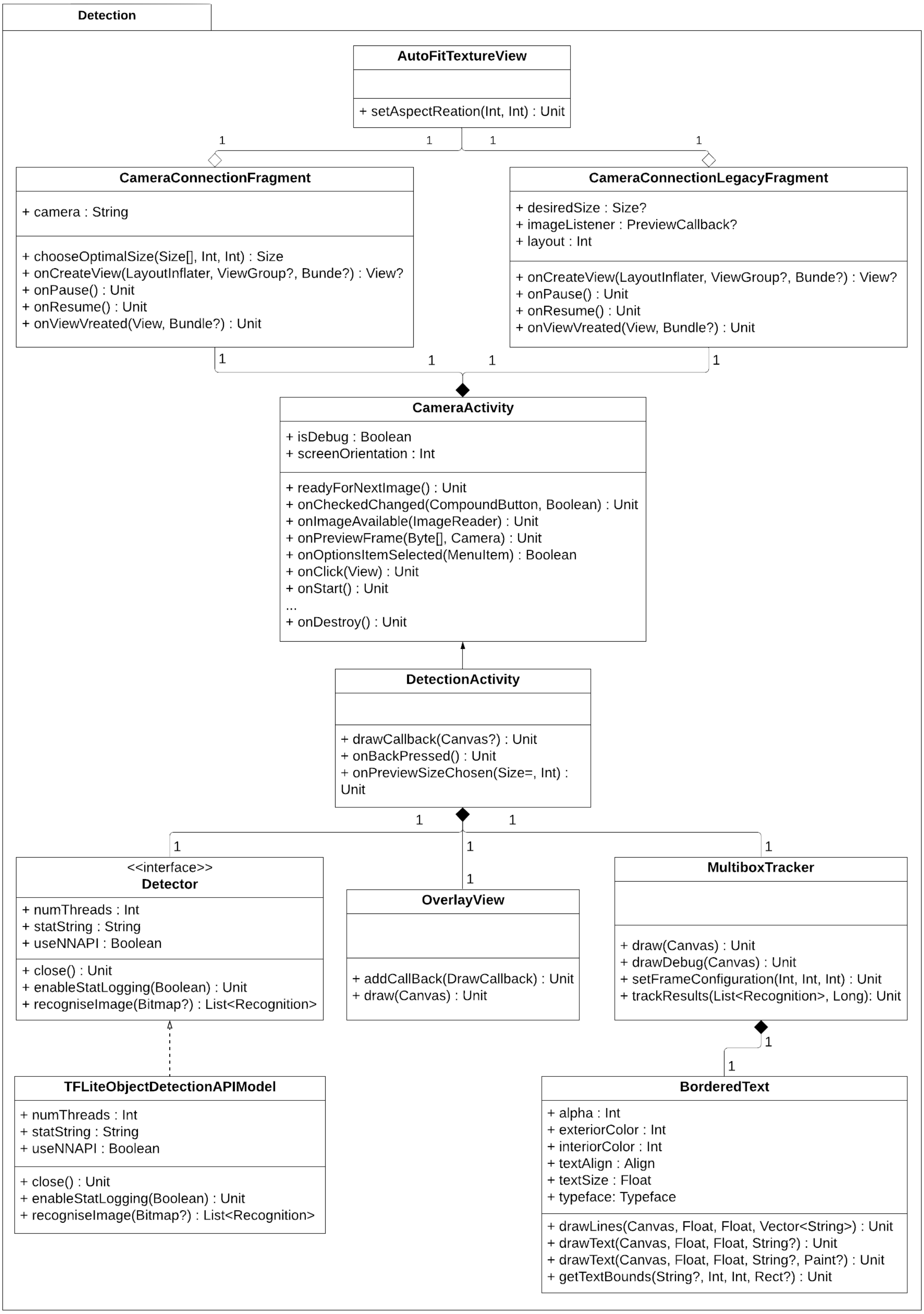


Figure 6.1.: UML class diagram of the new detection package. Only public properties and methods are shown.

6.2. DetectionActivity

Just as `ClassificationActivity` is the core activity that handles the image classification of the camera feed, the new `DetectionActivity` is the central activity for object detection. The user can easily switch between the different mode of image analysis by using a toggle button on top of the screen. We decided to use a toggle switch here since there are exactly two operating modes (classification and detection) and they are working mutually exclusive in order to best utilise the computation power of the device. This design decision may be revised later when the scope of the app is expanded to cover more visual tasks (see Section 3.1 for examples of such tasks). Another reason for a rework can be the fact that mobile devices might have enough computing power so that solving multiple tasks in parallel does not make a noticeable difference for the app user, meaning images are still processed in real-time.

6.2.1. Inheritance

`DetectionActivity` inherits from the abstract class `CameraActivity` and implements the `OverlayView.DrawCallback` interface. The `CameraActivity` handles all communication with Androids camera API and implements the proper setup and behaviour of the elements in the bottom sheet. The `DrawCallback` provides a single function definition that the `DetectionAcitivty` overrides in order to render the detected objects as rectangles in a layer on top of the preview of the camera image stream.

6.2.2. Bitmap Conversion

Since TUM-Lens can run on a multitude of devices with varying combinations of camera and display resolution, the `DetectionActivity` sets parameters such as font sizes and matrices to convert the image data back and forth between the different formats the data is needed in. This is important as in contemporary Android devices, the resolution of the camera is usually above the resolution of the display. This is also the case for the Xiaomi Mi 9 on which we tested the application during its development (see Appendix B for its technical specifications). Sticking to the example of Xiaomi Mi 9, the images captured by the camera have 20 (selfie camera) or 45 (main camera) megapixels and are scaled down to 2.5 megapixel to be displayed on the device screen. After that, they are again scaled down to a bitmap of 300x300 pixels[1] as this is the appropriate input size for the implemented detection framework described in Section 3.3. Once the detector returns bounding boxes for the objects detected in its input image the coordinates of these rectangles in turn are scaled back up to mark the corresponding areas in the higher resolution image shown on the device display.

6.2.3. Detection and Tracking

The `DetectionActivity` also orchestrates the two most fundamental tasks of the object detection process: object detection and, since the images are derived from a continuos camera stream, object tracking. It instantiates the object `detector` of the class

[1] The image received by the detector might indeed be even smaller than 300x300 pixels as the camera input is usually not square and the aspect ratio is kept throughout the image conversion pipeline. Therefore, the 300 pixels denote the larger dimension (width or height) of the input bitmap.

`TFLiteObjectDetectionAPIModel` to handle the interactions with the TensorFlow Lite api and a `MultiBoxTracker` object `tracker` that tries to assign each rectangle drawn on the screen to its related recognition result across successive input images. This is important because the bounding boxes that are drawn as an overlay on the camera view use different colours for different objects. If no tracking would happen, the colour that is assigned to an object will change randomly with every new image the detector processes. This would be very confusing for the user. Therefore, the recognition results returned by the detector are not directly rendered on the screen but first processed by the `tracker`.

The `tracker` guarantees two properties: Firstly, empty or degenerated recognition results are skipped and not tried to be displayed. Secondly, the 15 colours the `tracker` can assign to the remaining objects are allocated in the same order every time the old drawings are cleared from the canvas and new ones are drawn. This process of detecting objects, clearing old rectangles from the screen and drawing new ones happens with every image the detector processes. While the tracking is far from perfect, the order in which recognitions are returned by the detector stays roughly the same. This happens because they are ordered by the coordinates of their bounding boxes. Depending on the camera settings, the camera can usually stream 30 to 60 frames per second. If the user doesn't move the camera surpassingly fast, the coordinates of the recognition results stay close enough together so that they get assigned the same colour across the changing camera images. New objects appearing in the scene (or existing ones becoming larger as the user gets closer) are currently the major cause for inconsistent object colouring. However, as long as no new objects are recognised, the existing ones tend to keep their colours. This is already a great achievement given the tracking is purely based on properties of the TensorFlow Lite api and does therefore add effectively non-existent overhead to the object detection pipeline.

The detection threshold is set to 0.5 in order to display the most relevant objects. Hence, only recognition results received from the `detector` that have a confidence score of 0.5 or higher are handed to the `tracker`. This further improves the user experience supplementary to the object tracking.

6.3. Dive Into

6.4. Object Detection

6.5. Implementation Fun

6.6. Next steps in the development of TUM-Lens

Part III.

Results

Kotlin made the code definitely more concise

7. Evaluation

7.1. Performance

7.2. Accuracy

8. Outlook

8.1. Possible Applications

As outlined in Chapter 1, there are at least two major drivers that will fuel future development efforts. Protecting the sphere of privacy will definitively be a major motivation to build apps running machine learning frameworks locally. We've already touched upon this by alleging the example of DeepType in Section 1.3. The second driver for an increase in on-device machine learning applications has been introduced in Section 1.2. It is the need for machine learners that can act independently of an internet connection. This is important as it is simply impossibly to guarantee such a connection in some circumstances. We will expand this line of thought by adding the requirement for increased speed that some problems have. We will use the following paragraphs to introduce a wide range of potential applications centred around the idea of on-device machine learning partitioned by their major drivers.

8.1.1. Privacy Inspired Use Cases

Some company like Apple and Snap[1] are leading by example - at least in specific areas of their business practices. Apple's Face ID is a well-known security feature of Apple's smartphone division. It relies purely on on-device machine learning to pick up all the important details that make a face unique thereby successfully securing the device without transferring sensitive user data to the cloud. Snapchat uses on-device machine learning to power its manifold filters so that user data only gets transmitted once the user actively shares a photo or video via the app.

Since smartphones are just a subcategory of mobile devices, different hardware opens the door for new types of applications. Health care is one industry that can greatly benefit from on-device machine learning as health data is among the data categories that are most worthy of protection. Today, there's already an astonishing variety of devices that capture health data - each of which also comes in a "smart" version that not only collects this data but also sends it to decentralised servers for storage and processing: smart watches, scales, portable electrocardiogram (ecg) and electroencephalography (eeg) devices, blood pressure monitors, glucometers, thermometers and many more. Future developments might combine more and more of such tools into personal health companion devices that can capture and track every aspect of our health condition. Combining these various inputs to a coherent assessment of one's health status is an ideal use case for the application of machine learning. Since the data is sensitive and can easily tell a story people might want to keep private, it is a reasonable assumption that there will be a market for health companions that perform

[1] Snap Inc. is the developer of Snapchat, a mobile app known for its various and peculiar filters that users can apply to manipulate the images captured by the smartphone camera.

their analysis offline without sharing any data with a third party.

Listing all application domains of on-device machine learning would exceed the scope of this work. It can be observed, however, that there's not only the dimension of machine learning task along which different applications can be found (image classification, object detection, natural language processing, ...) but also other dimensions such as hardware on which the machine learning algorithm is deployed (smartphones, gadgets, robots, ...) or industries benefiting from customers being less scared about privacy breaches (entertainment, media, public sector, ...).

8.1.2. Autonomy and Speed Inspired Use Cases

to latencyDie Anwendungsbereiche von Computer Vision sind zahlreich. Zu den regelmäßigen Aufgaben im Bereich

Autonome Autos werden unahängig von einer Verbindung zum Internet (5G, shared medium, bleibt das Auto im Tunnel dann stehen?)

8.2. Future Work

Replace depracted functionalities

Add more models to chose from for object detection

Apple object detection to files loaded from storage

Change the apps infrastructure so that models (don't forget the labels though) can be loaded from a server (suitable as they tend to be rather small). This would make it easier to add new models on the fly.

Use real tracking to track objects across consecutive input images

Part IV.

Appendix

A. Screenshots of the Application

B. Test Device Specifications

The development of TUM-Lens v2.0 was tested on a Xiaomi Mi 9. Development and testing occurred April to August 2021. The most relevant specifications are as follows.

Brand and name	Xiaomi Mi 9
CPU	Octa-core: 8 Kryo 485 cores with respective clock speeds of 1x2.84 GHz, 3x2.42 GHz and 4x1.78 GHz
GPU	Adreno 640
RAM	6.00 GB
Screen resolution	1080 x 2340 pixels ($\approx$ 2.5 MP)
Screen aspect ratio	19.5:9
Screen size (diagonal)	6.39 inches
Main camera (triple)	48 MP, f/1.8, 27mm (wide), 1/2.0", 0.8µm, PDAF 12 MP, f/2.2, 54mm (telephoto), 1/3.6", 1.0µm, PDAF 16 MP, f/2.2, 13mm (ultrawide), 1/3.0", 1.0µm, PDAF
Selfie camera (single)	20 MP, f/2.0, (wide), 1/3", 0.9µm
Model identifier	M1902F1G
MIUI version	MIUI Global 12.0.5 (QFAEUXM)
Android version	10 QKQ1.190825.002
Release Date	25th of March 2019

Table B.1.: Test Device Specifications

List of Figures

2.1. Classification of Machine Learning 4
2.2. 2-Layered ANN 6
2.3. 3-Layered ANN 6

3.1. Typical Computer Vision Tasks 7
3.2. Detecting Objects at Different Scales Through SSD's Feature Maps 12

6.1. UML Class Diagram of The New Detection Package 20

List of Tables

3.1. Modern CNN Backbones and Detectors 9

5.1. Java Files in TUM-Lens v1.0 And Change in Lines of Code After Their Conversion to Kotlin 18

B.1. Test Device Specifications 30

Bibliography

[1] *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (Las Vegas, NV, USA). IEEE Computer Society, Dec. 2016. ISBN: 978-1-4673-8851-1.

[2] Shivang Agarwal, Jean Ogier Du Terrail, and Frédéric Jurie. "Recent Advances in Object Detection in the Age of Deep Convolutional Neural Networks". In: *arXiv preprint arXiv:1809.03193* (Sept. 2018).

[3] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. "YOLOv4: Optimal Speed and Accuracy of Object Detection". In: (Apr. 2020). URL: `https://arxiv.org/abs/2004.10934v1`.

[4] "Convolutional Neural Networks Backbones for Object Detection". In: *Image and Signal Processing. ICISP 2020. Lecture Notes in Computer Science* 12119 (June 2020), pp. 282–289. DOI: `10.1007/978-3-030-51935-3_30`.

[5] Android Developers. *Android's Kotlin-first approach.* 2021. URL: `https://developer.android.com/kotlin/first` (visited on 07/31/2021).

[6] Android Developers. *Neural Networks API.* 2021. URL: `https://developer.android.com/ndk/guides/neuralnetworks` (visited on 07/31/2021).

[7] Google Developers. *Google I/O Keynote (Google I/O '17).* May 2017. URL: `https://youtu.be/Y2VF8tmLFHw?t=5245` (visited on 07/31/2021).

[8] Nikita Dvornik et al. "BlitzNet: A Real-Time Deep Network for Scene Understanding". In: *Proceedings of the IEEE International Conference on Computer Vision* (Venice, Italy). Vol. 2017-October. Institute of Electrical and Electronics Engineers Inc., Oct. 2017, pp. 4174–4182. URL: `https://arxiv.org/abs/1708.02813v1`.

[9] Ross Girshick et al. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition* (Nov. 2014), pp. 580–587.

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning.* MIT Press, 2016. URL: `http://www.deeplearningbook.org`.

[11] Jeff Hale. *Deep Learning Framework Power Scores 2018.* Sept. 2018. URL: `https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a` (visited on 07/31/2021).

[12] Matthew Halpern, Yuhao Zhu, and Vijay Janapa Reddi. "Mobile CPU's rise to power: Quantifying the impact of generational mobile CPU design trends on performance, energy, and user satisfaction". In: *Proceedings - International Symposium on High-Performance Computer Architecture* (Apr. 2016), pp. 64–76. DOI: `10.1109/HPCA.2016.7446054`.

[13] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (Las Vegas, NV, USA). IEEE Computer Society, Dec. 2016, pp. 770–778. ISBN: 978-1-4673-8851-1. DOI: 10.1109/CVPR.2016.90.

[14] Wang Hechun and Zheng Xiaohong. "A Survey of Deep Learning-Based Object Detection". In: *2nd International Conference on Big Data Technologies* (Jinan, China). Association for Computing Machinery, Aug. 2019, pp. 149–153. DOI: 10.1145/3358528.3358574.

[15] Andras Horvath. *Which Java exceptions are the most frequent?* June 2018. URL: https://blog.samebug.io/which-java-exceptions-are-the-most-frequent-f830b113c37f (visited on 07/31/2021).

[16] Andrew G. Howard et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: (Apr. 2017). URL: https://arxiv.org/abs/1704.04861v1.

[17] Max Jokel. *TUM-Lens*. Version 1.0. July 31, 2021. URL: https://play.google.com/store/apps/details?id=com.maxjokel.lens.

[18] Maximilian Jokel. "Implementing a TensorFlow-Slim based Android app for image classification". Bachelor's thesis. Garching: Lehrstuhl für Wissenschaftliches Rechnen, Technische Universität München, Nov. 2020. URL: https://mediatum.ub.tum.de/1579885.

[19] Tao Kong et al. "HyperNet: Towards accurate region proposal generation and joint object detection". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (Las Vegas, NV, USA). IEEE Computer Society, Dec. 2016, pp. 845–853. ISBN: 978-1-4673-8851-1. DOI: 10.1109/CVPR.2016.98.

[20] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems* 25 (2012), pp. 1097–1105.

[21] Erik G Learned-Miller. "Introduction to supervised learning". In: *I: Department of Computer Science, University of Massachusetts* (Feb. 2014).

[22] Fei-Fei Li, Ranjay Krishna, and Danfei Xu. *CS231n: Convolutional Neural Networks for Visual Recognition*. 2021. URL: https://cs231n.github.io/neural-networks-1/ (visited on 07/31/2021).

[23] Fei-Fei Li, Ranjay Krishna, and Danfei Xu. *Detection and Segmentation*. May 2021. URL: http://cs231n.stanford.edu/slides/2021/lecture_15.pdf (visited on 07/31/2021).

[24] Yuqi Li et al. *TensorFlow Examples*. 2016. URL: https://github.com/tensorflow/examples/tree/master/lite/examples/object_detection/android (visited on 07/31/2021).

[25] Google Ireland Limited. *Object detection — TensorFlow Lite Examples*. URL: https://www.tensorflow.org/lite/examples/object_detection/overview (visited on 07/31/2021).

[26] Tsung Yi Lin et al. "Focal Loss for Dense Object Detection". In: *Proceedings of the IEEE International Conference on Computer Vision* (Venice, Italy). Vol. 2017-October. Institute of Electrical and Electronics Engineers Inc., Oct. 2017, pp. 2999–3007. DOI: 10.1109/ICCV.2017.324. URL: http://arxiv.org/abs/1708.02002.

[27] Jie Liu et al. "Performance Analysis and Characterization of Training Deep Learning Models on Mobile Device". In: *Proceedings of the International Conference on Parallel and Distributed Systems - ICPADS* (Dec. 2019), pp. 506–515. DOI: 10.1109/ICPADS47876.2019.00077.

[28] Wei Liu et al. "SSD: Single Shot MultiBox Detector". In: *Lecture Notes in Computer Science* (2016), pp. 21–37. ISSN: 1611-3349. DOI: 10.1007/978-3-319-46448-0_2. URL: ttp://dx.doi.org/10.1007/978-3-319-46448-0_2.

[29] Tom M. Mitchell. *Machine Learning*. Ed. by Erick M. Munson. McGraw-Hill, Mar. 1997, p. xv. ISBN: 0-07-042807-7.

[30] Andrew Ng. *Machine Learning Course by Stanford University*. 2021. URL: https://www.coursera.org/learn/machine-learning (visited on 07/31/2021).

[31] *Proceedings of the IEEE International Conference on Computer Vision* (Venice, Italy). Vol. 2017-October. Institute of Electrical and Electronics Engineers Inc., Oct. 2017.

[32] Team PyTorch. *PyTorch 1.3 adds mobile, privacy, quantization, and named tensors*. Oct. 2019. URL: https://pytorch.org/blog/pytorch-1-dot-3-adds-mobile-privacy-quantization-and-named-tensors/ (visited on 07/31/2021).

[33] Joseph Redmon et al. "You Only Look Once: Unified, Real-Time Object Detection". In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (Las Vegas, NV, USA). IEEE Computer Society, Dec. 2016, pp. 779–788. ISBN: 978-1-4673-8851-1. DOI: 10.1109/CVPR.2016.91.

[34] Ahmad EL Sallab et al. "Deep reinforcement learning framework for autonomous driving". In: *Electronic Imaging* 2017 (2017), pp. 70–76.

[35] A. L. Samuel. "Some Studies in Machine Learning Using the Game of Checkers". In: *IBM Journal of Research and Development* 3 (3 July 1959), pp. 210–229. DOI: 10.1147/RD.33.0210. URL: http://ieeexplore.ieee.org/document/5392560/.

[36] A. L. Samuel. "Some Studies in Machine Learning Using the Game of Checkers. II - Recent Progress". In: *IBM Journal of Research and Development* 11 (6 Nov. 1967), pp. 601–617. DOI: 10.1147/RD.116.0601.

[37] R. Sathya and Annamma Abraham. "Comparison of Supervised and Unsupervised Learning Algorithms for Pattern Classification". In: *International Journal of Advanced Research in Artificial Intelligence* 2 (2 2013), pp. 34–38. ISSN: 2165-4069.

[38] Laurent Sifre. "Rigid-Motion Scattering For Image Classification". PhD thesis. École Polytechnique, CMAP, 2014. URL: https://www.di.ens.fr/data/publications/papers/phd_sifre.pdf.

[39] Karen Simonyan and Andrew Zisserman. "Very Deep Convolutional Networks for Large-Scale Image Recognition". In: *3rd International Conference on Learning Representations* (San Diego, CA, USA). ICLR, Sept. 2014. URL: https://arxiv.org/abs/1409.1556v6.

[40] Petru Soviany and Radu Tudor Ionescu. “Optimizing the Trade-off between Single-Stage and Two-Stage Object Detectors using Image Difficulty Prediction”. In: *20th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing* (Timișoara, Romania). Institute of Electrical and Electronics Engineers Inc., Sept. 2018, pp. 209–214. URL: `https://arxiv.org/abs/1803.08707v3`.

[41] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Ed. by Frances Bach. second edition. The MIT Press, 2018. ISBN: 9780262039246.

[42] Christian Szegedy et al. “Going deeper with convolutions”. In: *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (Boston, MA, USA). IEEE Computer Society, June 2015, pp. 1–9. ISBN: 978-1-4673-6964-0. DOI: `10.1109/CVPR.2015.7298594`.

[43] James Vincent. *Google’s new machine learning framework is going to put more AI on your phone*. May 2017. URL: `https://www.theverge.com/2017/5/17/15645908/google-ai-tensorflowlite-machine-learning-announcement-io-2017` (visited on 07/31/2021).

[44] James Vincent. *The iPhone X’s new neural engine exemplifies Apple’s approach to AI*. Sept. 2017. URL: `https://www.theverge.com/2017/9/13/16300464/apple-iphone-x-ai-neural-engine` (visited on 07/31/2021).

[45] Oriol Vinyals et al. “Grandmaster level in StarCraft II using multi-agent reinforcement learning”. In: *Nature* 575 (2019), pp. 350–354.

[46] Mengwei Xu et al. “DeepType: On-Device Deep Learning for Input Personalization Service with Minimal Privacy Concern”. In: *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies* 2 (4 Dec. 2018), pp. 1–26. DOI: `10.1145/3287075`. URL: `https://dl.acm.org/doi/abs/10.1145/3287075`.

[47] Alex Zhitnitsky. *The Top 10 Exception Types in Production Java Applications - Based on 1B Events*. June 2016. URL: `https://www.overops.com/blog/the-top-10-exceptions-types-in-production-java-applications-based-on-1b-events/` (visited on 07/31/2021).

[48] Xiaojin Zhu and Andrew B. Goldberg. “Introduction to Semi-Supervised Learning”. In: *Synthesis Lectures on Artificial Intelligence and Machine Learning* 3 (June 2009), pp. 1–130. DOI: `10.2200/S00196ED1V01Y200906AIM006`. URL: `https://doi.org/10.2200/S00196ED1V01Y200906AIM006`.

[49] Yousong Zhu et al. “CoupleNet: Coupling Global Structure with Local Parts for Object Detection”. In: *Proceedings of the IEEE International Conference on Computer Vision* (Venice, Italy). Vol. 2017-October. Institute of Electrical and Electronics Engineers Inc., Oct. 2017, pp. 4146–4154. DOI: `10.1109/ICCV.2017.444`. URL: `https://doi.org/10.1109/ICCV.2017.444`.

Acronyms

ann artificial neural network.

api application programming interface.

cnn convolutional neural network.

cv computer vision.

ecg electrocardiogram.

ml machine learning.

NNAPI Neural Networks API.

ResNet residual neural network.

Glossary

application programming interface set of functions and procedures allowing the creation of applications that access the features or data of an operating system, application, or other service (Oxford Dictionary).

differential privacy property of a learnign algorithm that uses personal data for training while guarateeing that an individual's data cannot be reverse engineered by analysing the algorithm while potentially even having access to the data of all the other individuals. In other words, the differentially private algorithm behaves very similar (if not entirely identical) regardless of whether a particular personal information was used during training..