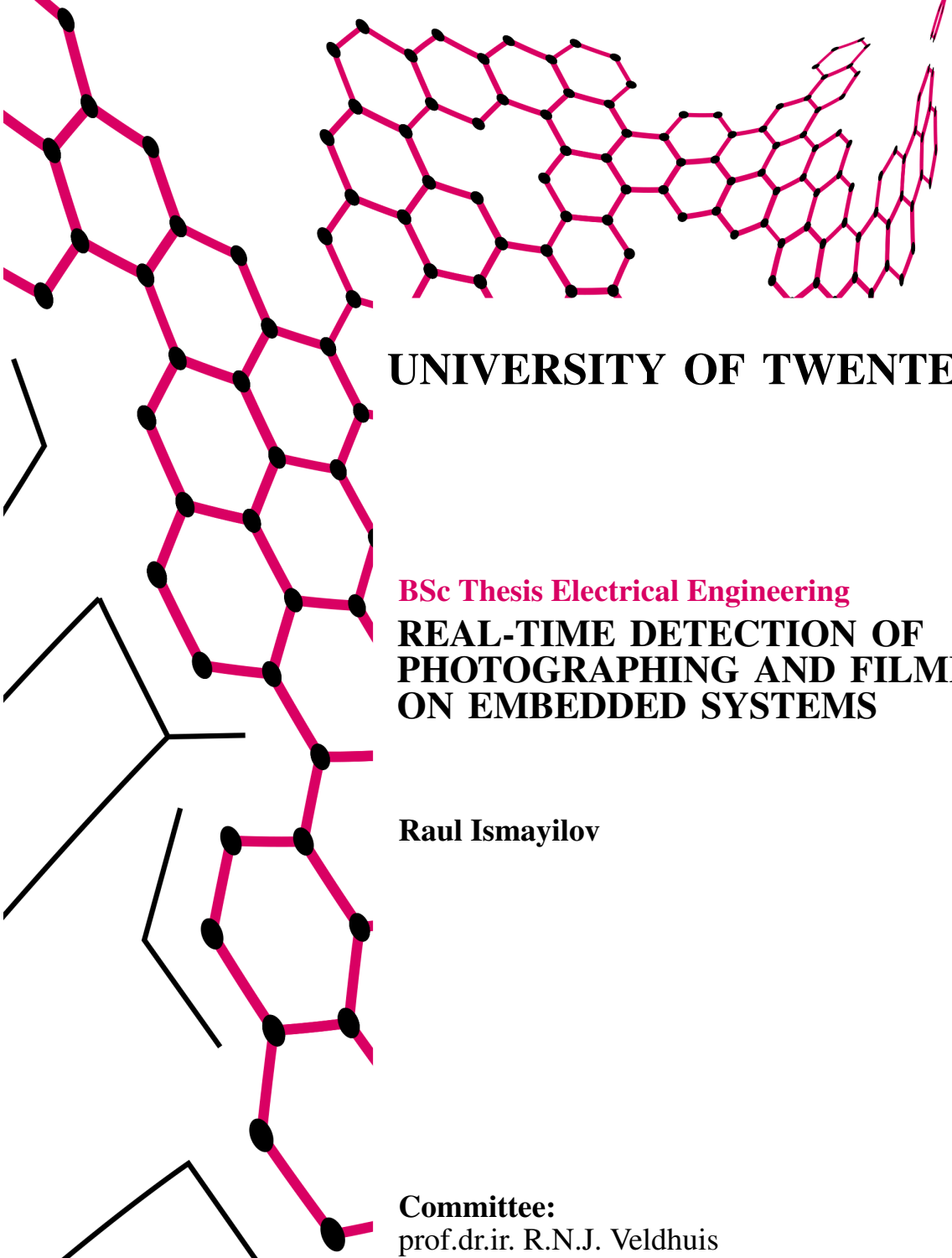


UNIVERSITY OF TWENTE

BSc Thesis Electrical Engineering

REAL-TIME DETECTION OF PHOTOGRAPHING AND FILMING ON EMBEDDED SYSTEMS

Raul Ismayilov

Committee:
prof.dr.ir. R.N.J. Veldhuis
dr. C.G. Zeinstra
dr. J. Zwiers

Faculty of Electrical Engineering,
Mathematics and Computer Science (EEMCS)

July 2022

***Abstract*—Current developments in image and video capturing technologies enable the possibility of non-consensual distribution of one's identity information. Whether it is accidental photography or a deliberate attempt of filming, with emerging privacy concerns, potential methods of preventing a person from being identified by most facial recognition systems are currently being investigated. Several wearable solutions, such as jewelry or glasses, targeted to prevent facial identification exist; however, most require manual control due to their passive nature. This paper solves this problem with an embedded system capable of automatic real-time inhibition of photography and filming. Using a lightweight YoloFastestV2 deep learning model in combination with NCNN and MNN inference frameworks, targeted for optimal performance on embedded devices, an object detection algorithm is trained and deployed on Raspberry Pi devices to identify when the user is being filmed. Glasses with variable lens transparency are used in the system and instructed to turn dark when filming is detected to prevent identity recognition. Precision-recall curves are used as a metric to evaluate the designed models, and differences between NCNN and MNN frameworks are examined. Based on the results, the proposed system achieves an accuracy of 89% when evaluated using images depicting expected filming scenarios. Real-world experiments are also conducted to validate the performance, and results demonstrate that accurate detections with an inference time of 46ms are achievable on Raspberry Pi 4.**

I. Introduction

Ever-evolving image and video capturing technologies make it easier than ever to photograph a person without their consent. While one might assume that the privacy rights can protect an individual from being filmed in a public location, the reality is different: photography is often considered a form of art, and any form of prohibition of filming may be regarded as a violation of the rights of the artistic expression [1]. As a result, privacy concerns arise since the images can be shared, and the location, time, and, most importantly, identity information of the captured individual can become easily accessible without their permission. Thus, there is a great need for the protection of one's privacy when it comes to photography and filming [2].

To address the forementioned issue, several methods of preventing the person from being filmed exist. The most obvious solution, yet impractical in many scenarios, would be to wear a facial disguise. This prevents most facial recognition systems from identifying a person. Furthermore, facial paint and jewelry designs capable of achieving the same results exist. Wearable glasses implanted with retroreflectors are another existing option that can be used to prevent measurements by many LFR systems [3]. In [4], wearable glasses equipped with infrared LEDs are used to prevent a facial recognition system from detecting the user. However, most of the existing literature focuses on methods of passive prevention of filming; for instance, the IR LEDs need to be turned on by the user when the cameras are observed. Instead, this paper investigates the possibility of making this process automated by using deep learning (DL) on embedded systems to detect cameras in real-time and, upon detection, prevent the person from being filmed. For achieving this, a lightweight neural network is selected, trained to detect filming, and deployed on two different inference frameworks designed for optimal speed on mobile and embedded devices. The resulting DL models are then deployed on a Raspberry Pi 4 and Zero W and tested using a live video feed from a webcam. When the user is observed to be filmed, E-TINT® glasses [5] are used to change their lens transparency and turn dark to prevent the capture of the user's identity. The experimental results show that this task can be achieved, and the trade-off between accuracy and inference time is investigated. The results are then compared with a similar model trained on the YOLOv5 model, and conclusions are drawn.

The research and experiments conducted in this paper aim to answer the following research questions:

- How photography and filming can be detected, and which method provides the most accurate results?
- What steps need to be taken to achieve real-time object detection on embedded systems such as Raspberry Pi?
- What impact do inference frameworks have on inference times of models deployed on embedded systems?

II. Related Work

A. Object Detection Models

In the field of computer vision, object detection can be described as the task of identification and localization of objects present in an image or a video. It is realized by collecting images containing objects of interest and extracting the features using backbone architectures. Commonly used backbone architectures include examples such as VGG, ResNet, and EfficientNet, which are explained in more detail in Appendix A. The object detectors are then built on top of the backbone architecture layers and can be separated into two classes: two-stage and one stage. In two-stage object detectors, region proposals are generated first, and classification and detection occur in the second stage. Contrarily, one-stage object detectors have classification and regression performed at once, often resulting in faster and lighter models [6]. In other words, while two-stage object detectors apply the model at different scales and locations on the image and place the bounding boxes at locations with the highest probability scores, one-stage models, such as YOLO family models, run the entire image through the neural network, divide the image into multiple regions where weighted bounding boxes are placed, and use non-max suppression (NMS) algorithm to discard overlapping bounding boxes and keep only one with the highest probability of detection (see Figure 1) [7]. Thus, in the context of real-time object detection, one-stage models such as YOLO and SSD Mobilenet V2 are often preferred.

B. Inference Frameworks

The inference performance of DL models is influenced by many factors. Most of the existing literature focuses on hardware and models to understand the performance, leaving the software side, which plays a critical role in determining the inference time, unexplored [8]. When it comes to software, popular frameworks for DL model deployment are TensorFlow Lite [9], NCNN [10], MNN [11], and PyTorchMobile [12].

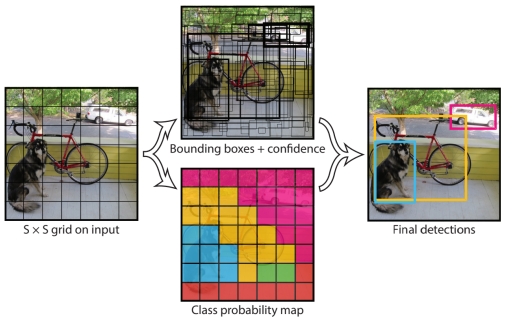


Figure 1: The architecture of YOLO model [7].

Q. Zhang et al. [8] argue that when comparing these frameworks, the results can vary greatly for different models. Yet, it can be observed that benchmarked one-stage YOLOv3 and YoloFastest models perform best on NCNN and MNN frameworks.

Generally, DL models are composed of many layers and parameters. YOLO algorithms utilize Convolutional Neural Networks (CNN) for detection and can have thousands of interconnected units and thousands or even millions of parameters. Thus, the vast majority of embedded devices fail to run these models in real-time due to the challenges related to resources and data [13]. Both NCNN and MNN frameworks, which show the best results on already exceptionally fast YOLO models, include several optimization techniques, which result in a further decrease in inference time.

Tencent's neural network inference framework (NCNN) is highly optimized for the use with ARM CPUs present on Raspberry Pi devices. Fully developed in C++, it targets the NEON accelerator on Raspberry Pi. Moreover, NCNN employs simpler Winograd convolutions in place of 3×3 convolutions. Additionally, quantization (FP32 to INT8) is available to reduce the model size and further lower the latency of computations [10].

Alibaba's MNN framework also offers strong support for ARM CPUs with its vast versatility of integrations [14]. MNN utilizes a pre-inference mechanism that performs runtime optimizations. These optimizations include techniques such as memory pre-allocation and reuse. It also employs algorithms for in-depth kernel optimizations, which further improve the functioning of widely-used DL operations [15]. Similar to NCNN, model quantization is also available on MNN [11].

C. Real-time Object Detection on Raspberry Pi

When it comes to high-performance computing, ARM technology is becoming more widespread. Raspberry Pi uses this technology and integrates a NEON pipeline, which improves its computing power by a factor of four through the support of vectorization. It provides a low-cost, high-performance portable platform that can be used in many scenarios [16]. For instance, authors of [17] use Raspberry Pi for object detection and drawing by using TensorFlow. In [18], Raspberry Pi processes the image data for location and motion tracking of seniors living alone. In [19], it is used for the detection and tracking of human faces.

III. Design and Implementation

This section explains the design choices behind the implementation of the photography and filming detection system. The image data used for training the model and inference framework selection are discussed first. After, the training settings are listed, and an optimal way of photography detection is identified. Finally, three designed models are presented, and their deployment and evaluation methods are described.

A. Dataset

A dataset containing a sufficient number of images for training and validation is needed to achieve satisfactory results. Several large datasets containing various object classes are available for this task: ImageNet [20], Microsoft COCO [21], Open Images [22], and Pascal VOC [23]. In this project, Open Images V4 dataset is used since it contains the highest number of images per class of interest. To only download the images of needed classes, OIDv4 ToolKit [24] is used. However, to achieve better results, a custom dataset was created for the final model. The dataset mainly contains people holding phones or cameras in their hands and taking an image in the mirror. It includes 1480 manually labeled mobile phone images and 1315 camera images from Open Images V4 dataset.

B. Image Data Pre-processing

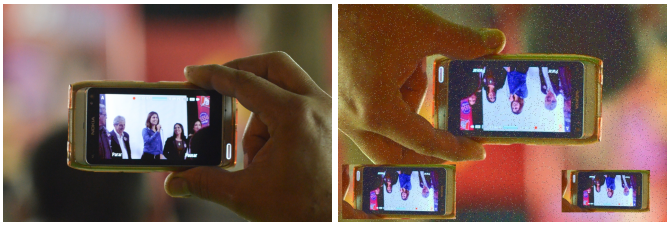
Once training images are available, several pre-processing steps can be carried out before model training commences. In this project, two pre-processing steps are considered: normalization of image inputs and data augmentation.

Normalization of image inputs is an operation performed on an image to meet the input size requirements of the model. For example, if the model with an input size of 300×300 pixels is given an image with a size 1920×1080, the input image needs to be normalized. One way of achieving this is cropping the image, but there are several scenarios where this method will not work, e.g., if the object is contained within the entire image. On the other hand, resizing the image preserves all image pixels, keeping the information about the object intact. In this project, images are resized to meet the model input size requirements.

Data augmentation increases the dataset size by using various augmenting techniques to alter the available images. In addition, it prevents model over-fitting, which is often the result of the use of smaller datasets and leads to models that do not discern well the data obtained from the test and validation sets. Hence, with the integration of data augmentation, the model's validation accuracy can be increased [25]. In this project, seven traditional methods of data augmentation are implemented as described in Appendix B.

However, even with traditional data augmentation methods, experimental results of top submissions in image detection challenges show that the Average Precision (AP) scores for small objects are often 2-3 times lower than for large objects. To improve the AP scores for small object detection, Kisantal et al. [26] propose a method to augment the dataset's images by copying smaller versions of the known objects and pasting

them in random places in the image. The new objects should not intersect with each other or the original objects. The best results are obtained when only 1-2 small objects are added to each image. Therefore, this method for small object data augmentation was implemented in this project and is available for use with any class of choice. In cases when there is no space for the small object to be placed in the image or the original object is smaller than 200×200 pixels in size, this data augmentation is not performed. An example of both traditional and small object data augmentation techniques implemented in one image is given in Figure 2.



(a) Original image (b) Augmented image

Figure 2: Image augmentation using both traditional and small-object data augmentation methods.

C. Model and Framework Selection

As previously mentioned, due to their low inference time, YOLO models are often preferred for real-time object detection. For this reason, one of the YOLO family models is used in this project. Given that the model not only needs to be used for real-time object detection but also needs to be sufficiently light for deployment on embedded devices, several lightweight YOLO models that can be deployed on Raspberry Pi for real-time detection are considered in Table I.

Model	Size	COCO mAP(0.5)	RPi 4 64-OS 1950 MHz
NanoDet	320×320	20.6	13.0 FPS
YoloFastestV2	352×352	24.1	18.8 FPS
YoloV2	416×416	19.2	3.0 FPS
YoloV3	352×352 tiny	16.6	4.4 FPS
YoloV4	416×416 tiny	21.7	3.4 FPS
YoloV4	608×608 full	45.3	0.2 FPS
YoloV5	640×640 small	22.5	1.6 FPS
YoloX	416×416 nano	25.8	7.0 FPS
YoloX	416×416 tiny	32.8	2.8 FPS
YoloX	640×640 small	40.5	0.9 FPS

Table I: Benchmark of various lightweight YOLO models trained on MS COCO dataset [27].

Based on the available models, the fastest model, which also has the 4th highest mAP score, is YoloFastestV2. For this project, the YoloFastestV2 model is trained to detect photographing and filming.

Furthermore, the deployment occurs on NCNN and MNN lightweight frameworks, both of which provide the best benchmark results for the YoloFastest model [8] and are optimized for mobile and embedded devices, making the inference time sufficiently low for real-time object detection on Raspberry Pi devices. An attempt was also made to deploy the model on TensorFlow Lite (TFLite) framework, however, due to a different layout in memory (NHWC instead of NCHW) and an unsupported implementation of the GatherV2 layer, this was not possible. For more information, refer to Appendix C.

C.1 YoloFastestV2 Object Detection Algorithm

In the early days of lightweight object detection, Mobilenet-SSD was a frequent model of choice. However, due to slow inference time, real-time object detection was challenging to achieve on ARM processor devices, examples of which include most smartphones and Raspberry Pi devices. Moreover, in practical applications of real-time object detection, mobile multi-core systems usually do not fully utilize all cores, resulting in high power consumption and CPU usage, leading to overheating and thermal throttling. YoloFastestV2 not only targets to achieve real-time performance on ARM processor devices but also low power consumption by using one or two processor cores at once. The model architecture of YoloFastestV2 consists of shufflenetV2 as a backbone and a modified detection head of YoloX. The anchor matching mechanism is appropriated from YOLOv5 [28].

YoloFastestV2 is a successor of the YoloFastest object detection algorithm, which, compared with MobileNet-SSD, is three times faster, and its model size is 20 times smaller. Using the VOC dataset, the accuracy results of 72.7% and 61.2% are achieved on MobileNet-SSD and YoloFastest, respectively [29]. Thus, to achieve lower inference time on ARM processors, the accuracy had to be sacrificed. Compared to the original version, YoloFastestV2 trades a 0.3% loss of accuracy for a 30% increase in inference speed [28].

D. Model Training

In this project, PyTorch is used for model training, following the choice made by the creator of YoloFastestV2 [30]. For faster training, NVIDIA RTX3080 GPU is used as the primary training device in PyTorch. Additionally, since Open Images V4 dataset contains train and test sets for each class [22], images obtained from the train set are further split with a ratio of 8:2 for train and validation sets, respectively. All input images are resized to YoloFastestV2's 352×352 input size.

D.1 Object Labels

During training, the model needs to have information about the coordinates of the bounding boxes corresponding to the objects to be detected. OIDv4 ToolKit was used to download the CSV file containing bounding boxes of all objects in the Open Images V4 dataset. Labels for each image containing classes used by the model are added to the text files. When data augmentation is performed and locations of bounding boxes are changed, the label files are updated accordingly. Once the labels are generated and updated, the next step is to convert them to the YOLO format, as shown in Figure 3.

OIDv4 ToolKit labels contain the coordinates of the top-left and bottom-right corners of the bounding box, while YOLO labels need to have the center coordinates of the box as well as its dimensions relative to the image size.

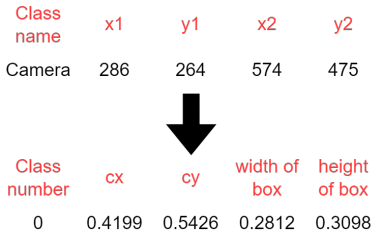


Figure 3: Conversion to YOLO label format.

D.2 Anchor Boxes

Unlike CenterNet or ExtremeNet model architectures, which are anchor-less, YOLO models, including YoloFastestV2, require the generation of anchor boxes, which are used in predicting the coordinates of the bounding boxes [31]. The ratios and scales of the model's anchor boxes are essential hyper-parameters since their shape should match the targets expected to be detected. As creators of the YOLO-face model argue, in most images where a face appears, it is expected to have a larger height than width, making the anchor boxes different for that specific model [32]. Similarly, this idea applies to the camera detection model. In the YoloFastestV2 model, a k-means clustering algorithm is used with the Intersection over Union (IoU) metric to automatically generate anchor boxes based on the bounding boxes given in the label files [30].

D.3 Train Configuration Settings

Besides anchor boxes, multiple hyper-parameters can be set before the model training begins. Examples include optimizer, initial learning rate, scheduler, and batch size. Overall, except for anchor boxes, batch size, and initial learning rate, the adjustable model hyper-parameters had little effect on the accuracy of the detections. Therefore, the focus, instead, was shifted towards the dataset and methods of detecting the photographing and filming. Nevertheless, more details about the selection of model hyper-parameters are given in Appendix D. Table II presents the summary of the choices made.

Anchors	Model specific; generated automatically
Optimizer	SGD (Stochastic Gradient Descent)
Initial Learning Rate	0.001
Training Scheduler	MultiStepLR (Multi-step Learning Rate)
Scheduler Steps	150, 250
Batch Size	128

Table II: Summary of configuration settings for model training.

E. Detection of Photographing and Filming

Detection of cameras on their own can be impractical for the final model implementation on the wearable since every detection will be considered a true positive, which, in turn, would trigger the glasses to turn dark even if the person is not being filmed and the object is in the background.

One approach to solve this problem is to trigger the glasses to turn dark only when the camera is pointing in the user's direction. However, there are several limitations to this implementation. Firstly, the dataset can be manually adjusted by removing the images where the cameras are not directly pointed at the user. However, in the case of large datasets consisting of thousands of images, this task becomes very time-consuming. In addition, the size of the dataset will be reduced drastically, resulting in worse performance since less unique data is available to train the model. Secondly, a custom dataset can be created by labeling the images manually, but once again, this task requires substantial time and data resources. Thirdly, the use of Arbitrary-Oriented Object Detection, an emerging field of object detection tasked with determining the orientation of the objects in the image, can be implemented but requires more complex model architectures and dynamic anchor selection [33], defeating the possibility of current deployment on embedded devices.

Another approach to this problem is to detect the camera and the person taking the picture. If both classes are detected, the glasses can be turned dark. Still, this method is prone to erroneous detections, for example, when the detected person is in the same frame as the camera but does not interact with it. Nevertheless, following this idea, multiple object classes can be identified for the detection of photographing and filming. For detecting cameras, classes "Camera" and "Mobile phone" can be utilized. As for the detection of humans, different parts of the human body can be used. The classes that were considered and tested are: *Human hand*, *Human eye*, *Person*, and *Human face*. Based on findings given in Appendix E, classes "Human face" and "Person" were selected as the final classes for detection of humans filming the user.

F. Custom Trained Models

In total, three different models were created and tested in this project. Each model is given a version number to signify implemented improvements. All of the models are deployed on both NCNN and MNN frameworks.

For traditional data augmentation, an augmentation factor of 3 was used, while for small object data augmentation, only one small object was added per image. For more information regarding this choice, refer to Appendix F. To prevent some of the false positives in all of the models, both the human and the camera need to be detected on multiple images in a row; this parameter can be easily adjusted and was set to 5. When several consecutive detections are made, the glasses are instructed to turn dark for 5 seconds, starting from the last observation of a human with a camera.

F.1 Model V1

Model V1 contains one model with three different classes: "Camera," "Mobile phone," and "Human face." Only traditional data augmentation is used. When both "Human face" and "Camera" or "Mobile phone" classes are detected, the glasses are instructed to turn dark. For training and validation, about 15k camera, 13k mobile phone, and 50k human face images are used (data augmentation is considered).

F.2 Model V2

Model V2 contains two One-Class Classification (OCC) sub-models: the first sub-model detects only cameras (images

of "Camera" and "Mobile phone" classes are combined in one "Camera" class), while the second sub-model detects human faces. Naturally, the inference time of Model V2 is expected to be twice larger than Model V1. However, by having OCC models, the effects of issues discussed in Appendix E can be considerably alleviated. The "Camera" class is augmented using traditional data augmentation, while the "Human face" class was not augmented due to a large number of available images. For training and validation, 28k camera and 50k human face images are used.

Moreover, since the inference time is increased by a factor of 2, a pre-trained Slim-320 face detection model deployed on MNN [34] is available for use instead of the custom-trained YoloFastestV2 model. This model has a decreased inference time due to its simpler architecture.

F.3 Model V3

Model V3 further improves Model V2 by using custom datasets made for the "Camera" and "Mobile phone" classes, which are merged in one "Camera" class of the first sub-model. During training, background images of cars, buildings, people, sunglasses, laptops, and monitors are added to the dataset with no labels to prevent some of the false camera detections. Traditional and small object data augmentations are used. Moreover, a model capable of only detecting mobile phones was trained to observe if the accuracy decreases when digital cameras are included in the dataset.

The second sub-model detecting human faces is replaced with "Person" class detection since the person filming might have their face covered. In addition, to address the problem when both camera and person are detected but the person is not interacting with the camera, a new rule for positive detections was created: if 70% of the bounding box around the camera is inside the bounding box of the human, then that means the human is holding the camera. For training and validation, 7.5k camera and 150k person images are used.

G. Deployment

To deploy the trained model, PyTorch generates a weight file that can be converted to the ONNX model for interoperability between PyTorch and NCNN/MNN frameworks. Both NCNN and MNN frameworks are implemented in the C++ environment and contain tools to convert the model from ONNX to corresponding framework files [10, 11]. An overview of model training and deployment is given in Figure 4.

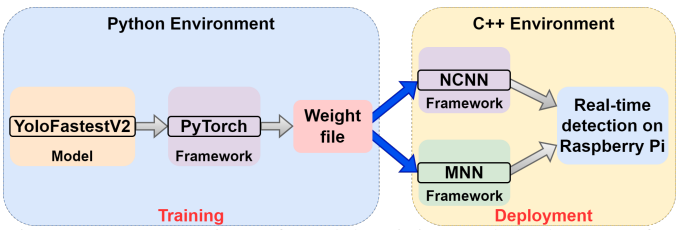


Figure 4: An overview of model training and deployment for the proposed object detection algorithm.

H. Model Evaluation

Since the deployment on NCNN and MNN frameworks occurs on Raspberry Pi, the model evaluation is also implemented on it. The evaluations are done on the test sets by comparing the known bounding boxes with the predicted bounding boxes generated by the model. To quantify this comparison, the model evaluation is done via Precision-Recall (PR) curves. To obtain a PR curve, one needs to calculate precision and recall at detection confidence thresholds ranging from 0 to 1. At each threshold, values for precision and recall are calculated using Equation 1, and a point is plotted on the PR curve. Figure 5 further displays how the required components of Equation 1 can be found in the context of bounding boxes.

$$\begin{aligned} \text{Precision} &= \frac{\text{TP}}{\text{TP}+\text{FP}} = \frac{\text{TP}}{\text{PP}} \\ \text{Recall} &= \frac{\text{TP}}{\text{TP}+\text{FN}} = \frac{\text{TP}}{\text{P}} \end{aligned} \tag{1}$$

Where TP, FP, FN, PP, and P are true positive, false positive, false negative, predicted positive, and actual positive, respectively.

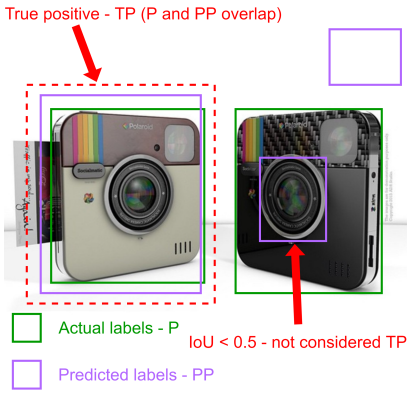


Figure 5: Definitions of TP, PP and P for bounding box detections.

True positives were defined to occur when the IoU of predicted and known bounding boxes is greater than 0.5. Once all the PR points are obtained, a python script is used to plot them and calculate the area under the graph using the Trapezoidal Rule. This area corresponds to the AP score and can be used as a metric to determine the performance of the object detection algorithm. Mean Average Precision (mAP) is calculated by averaging AP scores for each class; thus, in the case of OCC models, mAP = AP.

IV. Experimental Setup and Results

In this section, the hardware and software environments of the photography and filming detection system are introduced first, then all of the proposed models/sub-models are evaluated using PR curves, inference times collected on three different platforms are given, and, finally, a joined experiment with B. Wenhao [35] is introduced and differences between designed Model V3 and a similar YOLOv5 model are discussed.

A. Development Environment

The hardware environment for the proposed system shown in Figure 6 is described as follows:

- Raspberry Pi 4B
  - Architecture: ARM architecture
  - CPU: Cortex-A72 64-bit quad-core
  - Frequency: 2.1GHz (overclocked)
  - Memory: 4GB
- Logitech C920 USB webcam
- Circuit with a button to start/stop classification and a buzzer to signify detections using audio
- Driver circuit to control the transparency of glasses
- E-TINT® glasses

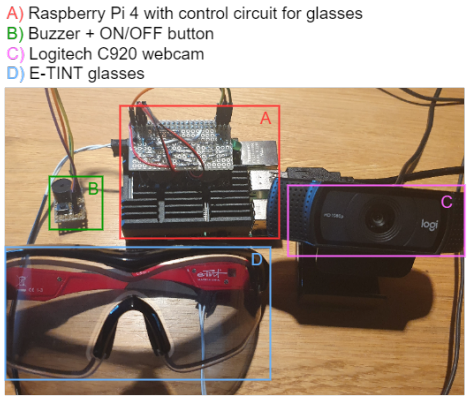


Figure 6: Hardware environment of the proposed system.

The flow diagram of the system is given in Figure 7 and the software environment is described as follows:

- Software platform: Raspberry Pi 64-OS
- Inference frameworks: NCNN and MNN

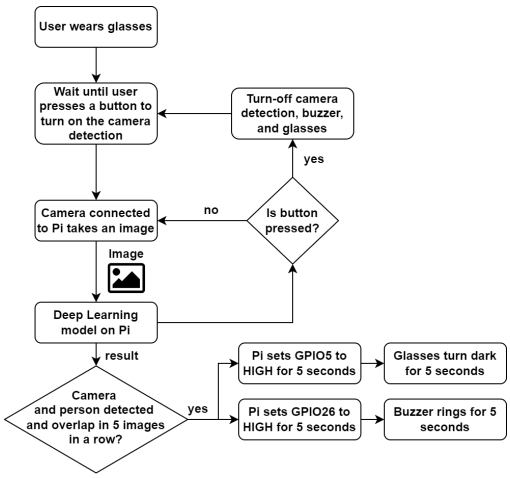


Figure 7: Flow diagram of the proposed system.

Moreover, the models were also deployed on Raspberry Pi Zero W, with a 1GHz single-core ARMv6 32-bit CPU and 512MB of memory, to only measure the inference times.

B. Evaluation of Models

Areas under PR curves, i.e., AP scores, were used as the main indicator of how well a class of a certain model performs. For this, test image sets belonging to each class were used. While classes "Human face" and "Person" contain an abundance of images with different backgrounds and scales, most of the "Camera" and "Mobile phone" class images from the Open Images V4 dataset represent unrealistic scenarios for filming, e.g., cameras taking all of the image space. For this reason, manually labeled images of people holding cameras and unlabeled background images were used in a test set. This dataset will be referred to as Mixed Dataset (MD), while the term Class Dataset (CD) will be used to refer to datasets in which each image contains the class of interest. Table III displays the AP scores of each class in each model/sub-model. MD was not used with human detection sub-models since it was specifically designed for cameras. It was also not used on Model V3, which is trained with a custom-labeled dataset, and, hence, is not expected to detect cameras in unrealistic filming scenarios.

Several findings were made from the AP scores given in Table III. Firstly, quantized MNN models were observed to have much worse performance when compared to non-quantized versions. This can be explained by the YoloFastestV2 model containing a layer not well supported by MNN quantization tools. More information on this is given in Appendix G. Secondly, since Models V1&2 are trained on mobile phone images from Open Images V4 dataset, which contain mostly screens or outdated phones, the AP scores for MD were lower than for CD. In comparison, Model V3, which was created to prevent this problem, performs much better. However, the opposite can be said about digital camera detection; Model V1 performs better than V3 on MD since it uses five times more camera images for training. Thirdly, having separate sub-models for human and camera classes when transitioning from Model V1 to V2 resulted in an average of 13% increase in mAP scores since class interference was eliminated (see Appendix E). Fourthly, using the "Person" class instead of "Human face" resulted in better AP scores. It is suspected this is due to the use of 150k images instead of 50k during training and the "Person" class test dataset containing manually selected images of people facing the camera at reasonable distances, which represent well the expected filming scenarios. Finally, for Model V3, it was found that the model trained only on mobile phone images performed better than one also trained with cameras. This is because cameras add an additional layer of complexity to the model. Overall, the additions in each model version proved to improve the accuracy of detections.

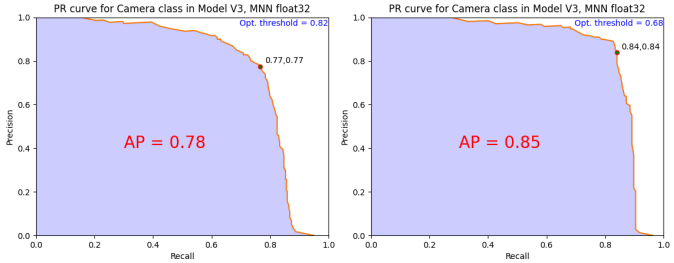
Furthermore, no apparent differences are observed when AP scores of NCNN and MNN models (only FP32 versions) are compared. It can be concluded that when it comes to accuracy, the two frameworks perform identically. When quantization is used on NCNN, an average drop of 3% in AP scores is recorded.

While AP scores can be used as an indicator of how well

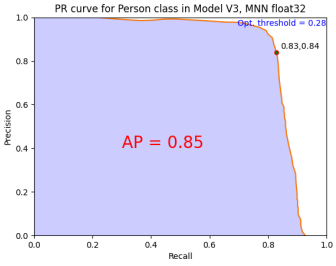
Model V1		NCNN FP32		NCNN INT8		MNN FP32		MNN INT8	
	Class	**CD**	**MD**	**CD**	**MD**	**CD**	**MD**	**CD**	**MD**
	Camera	0.74	0.87	0.69	0.82	0.74	0.88	0.20	0.15
	Mobile Phone	0.87	0.23	0.85	0.15	0.87	0.23	0.42	0.00
	Human Face	0.52		0.54		0.52		0.05	
Model V2									
Sub-model 1	Camera	0.86	0.58	0.84	0.57	0.85	0.57	0.40	0.06
Sub-model 2	Human Face (YoloFastestV2)	0.66		0.67		0.66		0.07	
	Human Face (Slim-320)					0.59		0.58	
Model V3									
Sub-model 1	Camera		0.78		0.77		0.78		0.06
	Camera (Mobile Phone only)		0.85		0.84		0.85		0.01
Sub-model 2	Person	0.85		0.84		0.85		0.07	

Table III: AP scores of each class in all proposed models based on the PR curve evaluation. **CD** - Class Dataset, e.i. test images contain only images in which class instances are present. **MD** - Mixed Dataset, e.i. test images include class instances and images with only background and day-to-day objects. All AP scores are generated for IoU and NMS thresholds of 0.5. Colors are given to AP scores for easier comparison between models. Better scores are represented with green color, while worse with red.

particular models perform, it is still beneficial to analyze the PR curves to find a threshold at which desired precision and recall values are obtained. In this project, a choice was made to select the optimal threshold when a model has equal precision and recall values; however, in some applications, having more emphasis on TPs or TNs might be useful, and for this, Figure 8 includes the PR curves of Model V3.



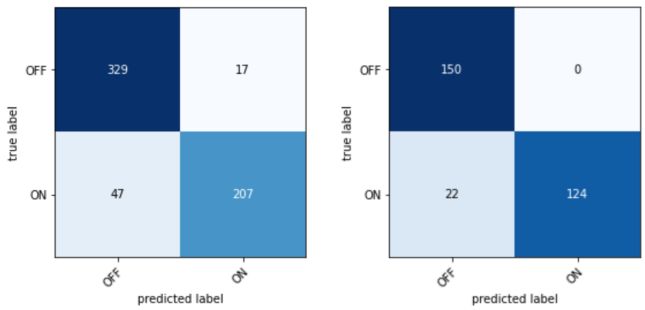
(a) Camera-and-phone sub-model (b) Phone-only sub-model



(c) Person sub-model

Figure 8: PR curves of Model V3. Camera-and-phone and phone-only detection sub-models are presented with the person detection sub-model.

Finally, to test the accuracy of the system implementing the detection rule of Model V3 (detection is only counted when the camera is inside the human bounding box), a binary system of classifications is used. Wearable glasses can have two states: OFF (transparent) and ON (dark). A confusion matrix is used to show the number of TP, TN, FP, and FN detections for the state of the glasses. The confusion matrices were made for two versions of Model V3 (first detects both cameras and phones, while second only phones) and are given in Figure 9. The accuracy of the camera-and-phone sub-model is **89%** while the phone-only sub-model has **93%** accuracy, confirming that the addition of digital cameras to the dataset decreases the accuracy of the model. For an interested reader, Appendix H includes additional plots of total loss during training.



(a) Camera-and-phone sub-model (b) Phone-only sub-model

Figure 9: Confusion matrices of Model V3.

C. *Inference Times*

Three platforms were used to deploy the models: Windows 11 system with Intel i7-11700KF CPU, Raspberry Pi 4, and Raspberry Pi Zero W. The recorded inference times are listed in Table IV. Due to Raspberry Pi Zero's ARMv6 architecture, which is considered outdated and is not supported by both NCNN and MNN frameworks [10, 11], the Cmake files for building the libraries had to be modified. While both libraries were built successfully, the MNN framework would throw an "Illegal Instruction" error when used, suggesting that it cannot support ARMv6 architecture.

During experiments, it was observed that both NCNN and MNN frameworks do not provide reduced inference time when

Figure 10: Examples of detections made by Model V3.

Platform	Model version	NCNN	MNN
RPi 4 64-OS 2.1GHz	V1	35ms (29 FPS)	22ms (45 FPS)
	V2 Slim-320	42ms (24 FPS)	31ms (32 FPS)
	V2&3	67ms (15 FPS)	46ms (22 FPS)
RPi Zero W 32-OS 1.0GHz	V1	2s (0.5 FPS)	Not supported
	V2&3	4s (0.25 FPS)	Not supported
Intel i7-11700KF	V1	5ms (200 FPS)	Not built
	V2&3	11ms (90 FPS)	Not built

Table IV: Inference time values for Models V1-3 deployed on three different platforms.

quantized, and only the model size is reduced. This result was unexpected since quantization decreases the precision of the data type, which reduces computations and, in theory, should accelerate inference. Similar results were also recorded on the pre-trained Slim-320 face detection model. Based on the results recorded on Raspberry Pi 4, the MNN framework, on average, is 46% faster than NCNN. Considering that the results of Table III show that both frameworks have the same accuracy, MNN becomes the clear choice for the deployment of YoloFastestV2 models since it provides reduced inference time with no accuracy loss. Moreover, as expected, inference times of Models V2&3 are twice larger than for Model V1 due to two sub-models running in parallel. Slim-320 face detection model used in Model V2 indeed provided reduced inference time; however, this came at the cost of accuracy (see Table III). Overall, all models deployed on Raspberry Pi 4 are capable of real-time object detection.

Raspberry Pi Zero W was found to be not capable of real-time object detection due to its unsupported and outdated ARM architecture. Raspberry Pi Zero 2 W would have been a much better choice for the deployment since it features the same form factor while using the CPU of Raspberry Pi 3 [36]. However, due to a stockout, no units were available for purchase, and the board was not used in the experiments.

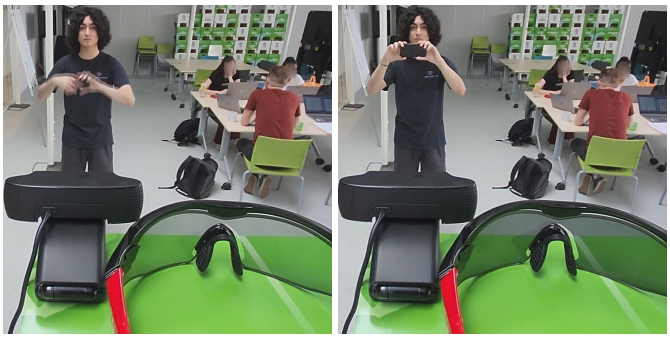
D. Joined Experiment: Comparison Between YoloFastestV2 and YOLOv5 Models

Joined experiments were conducted with B. Wenhao [35] since a similar system for detection of photographing and filming was developed on the YOLOv5 model. The MD and test dataset from [35] were evaluated using both Model V3 (camera-and-phone sub-model) and the YOLOv5 model. The results are given in Table V. It can be observed that YOLOv5 has much better accuracy on both datasets. The reason for this discrepancy is YOLOv5's more complicated architecture, containing 7M parameters [35] compared to 250k of YoloFastestV2 [28], and its 3.3 times larger input image size (see Table I). YOLOv5 provides higher accuracy at the expense of increased inference time; if a single YoloFastestV2 model (Model V1) is compared to a similar YOLOv5 model, the inference time is expected to be 18 times larger on YOLOv5 (based on Tables I and IV). Thus, for deployment on embedded devices, the accuracy has to be sacrificed for lower inference time to enable the possibility of real-time object detection.

YOLO Model	MD	Dataset X
YoloFastestV2	mAP = 0.78	mAP = 0.66
YOLOv5 [35]	mAP = 0.92	mAP = 0.96

Table V: mAP scores of YoloFastestV2 and YOLOv5 for camera and mobile phone detection models. Dataset X represents the test images used by [35].

A real-world experiment was also conducted to observe how two different systems compare against each other. Figure 11 shows the proposed system at work. The results are given in Appendix I. From the experiments, it can be concluded that both systems perform well under various scenarios. While the response of Model V3 was much faster when compared with the YOLOv5 model, it also had lower accuracy and resulted in occasional false positive and false negative detections.



(a) No camera detected (b) Camera detected

Figure 11: Real-world experiment on the proposed photography and filming detection system. Glasses decrease their lens transparency in the presence of cameras pointing at the user.

V. Conclusion

In this paper, the concept behind a system capable of real-time detection of photography and filming on an embedded system was introduced. To answer the research question regarding how photography and filming can be detected most optimally, it was discovered that an unconventional approach to object detection was needed since models trained using a publicly available dataset resulted in poor real-world performance. Since the final solution required the detection of cameras only when the user is being filmed, the available datasets could not suffice because a substantial number of images contained within them did not represent well possible filming scenarios. In addition, due to the lightweight nature of the model, occurrences of false detections were frequent. A method of detecting both humans and cameras was investigated to prevent them. It was found that having one model for this task was not sufficient since the detection accuracy was lower due to class interferences. Instead, two OCC models were created and run in parallel. Moreover, to represent real-world scenarios of filming, a custom-labeled dataset had to be created. The added changes considerably improved the accuracy of detections in real-world scenarios, and the addition of the small object data augmentation technique increased the accuracy of detections at greater distances.

To provide an answer to the two remaining research questions about real-time object detection on embedded systems and the effects of inference frameworks on latency, research and experiments were conducted. Based on the results, it was determined that to achieve real-time object detection on embedded systems, a lightweight neural network paired with an inference framework designed for optimal operation on ARM processor devices is needed for the best combination of accuracy and latency. In addition, it was found that while both MNN and NCNN inference frameworks can considerably decrease the inference time on ARM devices, using the MNN inference framework with the YoloFastestV2 model results in a system that is significantly faster and has no accuracy loss when compared to one deployed on NCNN. Even though all of the models enabled real-time object detection on Raspberry Pi 4, the same cannot be said for Raspberry Pi Zero W due to its outdated ARM architecture. Nevertheless, based on real-world experiments on Raspberry Pi 4, it was found that the proposed system is capable of accurate detections and is 18 times faster than a similar YOLOv5 model.

This paper lays the foundation needed to achieve real-time detection of photography and filming on an embedded device. More insight is needed into how the accuracy and inference time can be improved further. One suggestion would be to have a vast custom-labeled dataset for human and camera detection. Given that the labels are correct and images represent realistic filming scenarios, OCC models are no longer required, making the model twice faster and more accurate than Model V3. Another possibility would be an implementation of a simple algorithm for arbitrary-oriented object detection, which would enable the detection of cameras only in particular orientations, removing the current need for human detection. The effects of using different cameras for object detection can be investigated, and data augmentation techniques can be used to make the models more robust to such variations. Moreover, the system can be further miniaturized by using Raspberry Pi Zero 2 W or other commercial DL-oriented boards. Finally, given that the designed system attempts to protect one's identity information, a method is needed to ensure that the images used by the camera for object detection are never stored on the embedded device.

References

[1] S. Philipp, "Street photography and the right to privacy", pp. 2–3, 2020. [Online]. Available: https://cognitio-zeitschrift.ch/index.php/cognitio/article/download/843/1037.

[2] T. Yamada, S. Gohshi, and I. Echizen, "Privacy visor: Method for preventing face image detection by using differences in human and device sensitivity", in *Communications and Multimedia Security*, 2013, pp. 152–161.

[3] N. Madison and M. Klang, "Recognizing everyday activism: Understanding resistance to facial recognition", *Journal of Resistance Studies*, p. 103,

[4] M. Frearson and K. Nguyen, "Adversarial attack on facial recognition using visible light", 2020.

[5] E-TINT. "Ctrl® one black red / smoke lens". (2021), [Online]. Available: https://e-tintproducts.com/product/ctrl-one-black-red-smoke-lens/.

[6] R. Ismayilov, "Real-time object detection on a tinyml system", 2022.

[7] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection", *CoRR*, vol. abs/1506.02640, 2015. [Online]. Available: http://arxiv.org/abs/1506.02640.

[8] Q. Zhang *et al.*, "A comprehensive benchmark of deep learning libraries on mobile devices", in *Proceedings of the ACM Web Conference 2022*, 2022, 3298–3307. [Online]. Available: https://doi.org/10.1145/3485447.3512148.

[9] "Performance measurement", [Online]. Available: https://www.tensorflow.org/lite/performance/measurement?hl=zh-cn.

[10] Tencent, "Ncnn", 2018. [Online]. Available: https://github.com/Tencent/ncnn.

[11] Alibaba, "Mnn", 2019. [Online]. Available: https://github.com/alibaba/MNN.

[12] "Pytorch mobile", 2019. [Online]. Available: https://github.com/alibaba/MNN.

[13] S. Voghoei, N. Hashemi Tonekaboni, J. G. Wallace, and H. R. Arabnia, "Deep learning at the edge", in *2018 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2018, pp. 895–901. DOI: 10.1109/CSCI46756.2018.00177.

[14] V. Courville and V. P. Nia, "Deep learning inference frameworks for arm cpu", *Journal of Computational Vision and Imaging Systems*, vol. 5, p. 3, 2020. [Online]. Available: https://openjournals.uwaterloo.ca/index.php/vsl/article/view/1645.

[15] X. Jiang *et al.*, "MNN: A universal and efficient inference engine", *CoRR*, vol. abs/2002.12418, 2020. [Online]. Available: https://arxiv.org/abs/2002.12418.

[16] N. Gupta *et al.*, "Deploying a task-based runtime system on raspberry pi clusters", in *2020 IEEE/ACM Fifth International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*, 2020, pp. 11–20. DOI: 10.1109/ESPM251964.2020.00007.

[17] T. Bayrak, V. Marttin, and U. Yüzgeç, "Raspberry pi based object detection and drawing", Jun. 2021.

[18] N. Tabbakha, W. H. Tan, and C. Ooi, "Indoor location and motion tracking system for elderly assisted living home", Nov. 2017, pp. 1–4. DOI: 10.1109/ICORAS.2017.8308073.

[19] R. A. Tripathy and R. N. Daschoudhury, "Real-time face detection and tracking using haar classifier on soc", 2014.

[20] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database", in *2009 IEEE conference on computer vision and pattern recognition*, Ieee, 2009, pp. 248–255.

[21] T. Lin *et al.*, "Microsoft COCO: common objects in context", *CoRR*, vol. abs/1405.0312, 2014. [Online]. Available: http://arxiv.org/abs/1405.0312.

[22] A. Kuznetsova *et al.*, "The open images dataset V4: unified image classification, object detection, and visual relationship detection at scale", *CoRR*, vol. abs/1811.00982, 2018. [Online]. Available: http://arxiv.org/abs/1811.00982.

[23] M. Everingham, S. M. A. Eslami, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The pascal visual object classes challenge: A retrospective", *International Journal of Computer Vision*, vol. 111, no. 1, pp. 98–136, 2015.

[24] M. Vittorio, "OIDv4 toolkit", 2019. [Online]. Available: https://github.com/EscVM/OIDv4_ToolKit.

[25] L. Perez and J. Wang, "The effectiveness of data augmentation in image classification using deep learning", 2017. DOI: 10.48550/ARXIV.1712.04621. [Online]. Available: https://arxiv.org/abs/1712.04621.

[26] M. Kisantal, Z. Wojna, J. Murawski, J. Naruniec, and K. Cho, "Augmentation for small object detection", *CoRR*, vol. abs/1902.07296, 2019. [Online]. Available: http://arxiv.org/abs/1902.07296.

[27] Qengineering, "Yolofastestv2 raspberry pi 4", 2022. [Online]. Available: https://github.com/Qengineering/YoloFastestV2-ncnn-Raspberry-Pi-4.

[28] qiuqiuqiu, "Yolo-fastestv2: Faster and lighter", 2021. [Online]. Available: https://zhuanlan.zhihu.com/p/400474142.

[29] qiuqiuqiu, "Yolo-fastest: Ultra-ultra-fast open source arm real-time object detection algorithm", 2020. [Online]. Available: https://zhuanlan.zhihu.com/p/234506503.

[30] dog-qiuqiu, "Yolo-fastestv2", 2021. [Online]. Available: https://github.com/dog-qiuqiu/Yolo-FastestV2.

[31] Z. Tian, R. Zhan, J. Hu, W. Wang, Z. He, and Z. Zhuang, "Generating anchor boxes based on attention mechanism for object detection in remote sensing images", *Remote Sensing*, vol. 12, no. 15, 2020, ISSN: 2072-4292. DOI: 10.3390/rs12152416. [Online]. Available: https://www.mdpi.com/2072-4292/12/15/2416.

[32] W. Chen, H. Huang, S. Peng, C. Zhou, and C. Zhang, "Yolo-face: A real-time face detector", *The Visual Computer*, vol. 37, no. 4, pp. 805–813, 2021, ISSN: 1432-2315. DOI: 10.1007/s00371-020-01831-7. [Online]. Available: https://doi.org/10.1007/s00371-020-01831-7.

[33] Q. Ming, Z. Zhou, L. Miao, H. Zhang, and L. Li, "Dynamic anchor learning for arbitrary-oriented object detection", *CoRR*, vol. abs/2012.04150, 2020. [Online]. Available: https://arxiv.org/abs/2012.04150.

[34] Qengineering, "Face detection on raspberry pi 32/64 bits", 2021. [Online]. Available: https://github.com/Qengineering/Face-detection-Raspberry-Pi-32-64-bits.

[35] B. Wenhao, "Real-time detection of photographing and filming on mobile devices", 2022.

[36] "Raspberry pi zero 2 w", 2022. [Online]. Available: https://www.raspberrypi.com/products/raspberry-pi-zero-2-w/.

[37] "A survey of modern deep learning based object detection models", *Digital Signal Processing*, vol. 126, p. 103 514, 2022, ISSN: 1051-2004. DOI: https://doi.org/10.1016/j.dsp.2022.103514.

[38] Onnx, "Tensorflow backend for onnx", 2019. [Online]. Available: https://github.com/onnx/onnx-tensorflow.

[39] H. Katsuya, "Openvino2tensorflow", 2020. [Online]. Available: https://github.com/PINTO0309/openvino2tensorflow.

[40] W. Tom, "Issue with gatherv2 conversion", 2021. [Online]. Available: https://github.com/onnx/tensorflow-onnx/issues/1317.

[41] A. Paszke *et al.*, "Pytorch: An imperative style, high-performance deep learning library", in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

APPENDIX A
BACKBONE ARCHITECTURES

Backbone architectures are an essential part of object detectors since they extract the features from the input images of the model. These architectures are constructed by combining various neural network layers: convolution layers, pooling layers, fully connected layers, etc. Each layer can have several tunable parameters and performs a specific operation. For example, while convolution layers apply filters to an image, pooling layers are used to decrease the dimensions of feature maps by, for instance, taking an average or maximum value of multiple cells. The differences between commonly used backbone architectures such as VGG, ResNet, and EfficientNet are, thus, in the number and arrangement of these layers. For example, VGG constructs the network using small convolution filters, reducing the number of network parameters. ResNet architectures are significantly deeper when compared to VGG; however, by using skip connections between stacked convolution layers, they alleviate the performance decay without adding more parameters to the network and achieve higher accuracy results. Creators of EfficientNet had a different approach: they meticulously analyzed how changes in network parameters affect the accuracy and, based on the results, developed a simple yet efficient and accurate architecture [37].

APPENDIX B
TRADITIONAL DATA AUGMENTATION

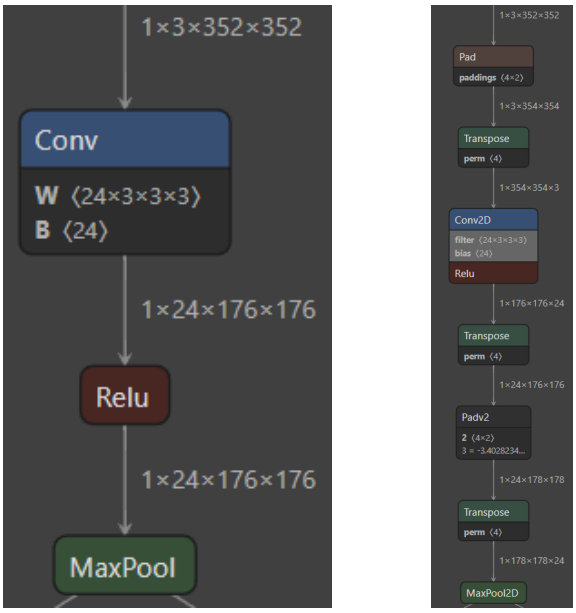
For each class used in the model, an augmentation factor can be set. When the augmentation factor is set to 1, no augmentation is applied to the class; on the other hand, if the augmentation factor is set to 3, for each existing image in the class, two more altered versions of this image are added. In total, seven different augmentation techniques are implemented:

- Horizontal, vertical, and horizontal & vertical flips
- Rotations by +90°, -90°, and 180°
- Brightness level change
- Saturation level change
- Contrast level change
- Addition of Gaussian noise
- Addition of salt-and-pepper noise

Each augmentation setting can be chosen to be used or not during the augmentation procedure. It is also possible to randomly select different augmentation techniques for each image rather than applying all of them at once.

APPENDIX C
ISSUES WITH DEPLOYMENT ON TFLITE FRAMEWORK

Similar to how PyTorch models can be converted to NCNN and MNN, they can also be converted to TFLite. The models first need to be converted to ONNX, and then by using onnx-tensorflow conversion tools [38], they can be converted to TensorFlow models, which are later converted to TFLite. However, using the official tool for conversion results in models which are much larger in size and contain many transpose layers. This is due to PyTorch models being in NCHW (channel second) format, while TensorFlow models are in NHWC (channel last) format. The conversion tool converts the models from ONNX to TFLite, and to solve the problem of TensorFlow's layers being in NHWC format, inserts multiple transpose layers throughout the model. This can be seen from Figure 12, where several ONNX layers and converted TFLite layers are compared. Given that TFLite convolution layers are in NHWC, while input and output need to be in NCHW format, two transpose layers are added. This inefficient conversion results in a significant increase in the model size and inference time.



(a) ONNX model layers (b) TFLite model layers

Figure 12: Comparison between ONNX and TFLite implementations of the same layers in YoloFastestV2 model.

Another tool for this conversion was developed by [39]. It is designed to solve the problem of added layers in TensorFlow models. The conversion process is described in Figure 13. While conversion from ONNX to OpenVINO platform was carried out without problems, OpenVINO to TensorFlow conversion resulted in an error in layer 80, which corresponds to the GatherV2 layer as seen in Figure 14. After further research, it was found that ONNX does not support the correct conversion of the GatherV2 layer and its *batch_dim* property [40]. Due to this error, no efficient method for conversion from PyTorch to TFLite was found for the YoloFastestV2 model.

Figure 13: Conversion of ONNX model to TFLite model without addition of extra transpose layers.

```
ERROR: layer_id    : 80
ERROR: input_layer0 layer_id=79: KerasTensor(type_spec=TensorSpec(shape=(1, 44, 44, 1, 24),
dtype=tf.float32, name=None), name='tf.compat.v1.gather_1/GatherV2:0', description="created
by layer 'tf.compat.v1.gather_1'")
```

Figure 14: Error during conversion from OpenVINO to TensorFlow.

Appendix D
Model Hyper-parameters

The performance of models with varying hyper-parameters was evaluated using the evaluation metrics printed every 10 epochs during training. Due to minor performance differences, the results are only discussed briefly.

By default, YoloFastestV2 uses an SGD (Stochastic Gradient Descent) optimizer, which holds the current state and, based on calculated gradients, optimizes the parameters such as weight decay and learning rate [41]. The PyTorch library contains many optimizer algorithms, a few of which (Adam, Adamax, and RMSprop) were tested during the training procedure. No significant improvements were detected, and, in some cases, worse performance was recorded. Hence, it was decided to keep SGD as the model optimizer. Similar results were observed during the choice of a scheduler. The learning rate scheduler adjusts the learning rate dynamically during training. By default, MultiStepLR scheduler is used, which multiplies the learning rate at predefined epochs with some factor gamma. It was discovered that the original value of gamma $= 0.1$ was excessive since the learning rate reduction resulted in no noticeable improvement as training epochs progressed, especially after the decrease for the second time (100 times less than the initial learning rate). Hence, through several experimental runs, it was found that gamma $= 0.2$ works optimally during the training. When ConstantLR scheduler was used, model over-fitting was observed after epoch 150. ExponentialLR with gamma $= 0.995$ (meaning that every epoch, the learning rate is multiplied by gamma) was found to work quite well and sometimes even converge to best results faster than MultiStepLR. However, for different models gamma parameter had to be varied slightly to achieve the best results, making the scheduler not universally applicable to most models. As a result, MultiStepLR scheduler with gamma $= 0.2$ and steps at epochs 150 and 250 was used during training.

The final value for the initial learning rate was set to be 0.001, as was set by default. A model that over-fits during the first 50 epochs was observed when this value was increased by a factor of 10, and when it was decreased by the same factor, a slightly worse performance was recorded since the model is learning much slower than preferred. Batch size, which represents the number of training samples going through the neural network at once, was set to 128. Using a smaller batch size resulted in slower training and worse performance, while having it larger (256 was a maximum value that could have been used with 10GB of GPU memory) resulted in slightly better results (AP score being higher by 0.02), but due to CPU and GPU overheating when training, it was decided to keep the batch size of 128.

Appendix E
Optimal Training Data Selection

While all four mentioned classes can be used to find a human, some perform better than others. For example, the class "Human hands" was found to be more challenging for the model to detect when compared to the class "Human eye." However, when these classes are added to the camera class in one model, it was discovered that the accuracy of classifications considerably worsens. The reason for that is that most images that contain humans from the camera and mobile phone classes do not have corresponding labels for humans. That creates a problem: the model learns the features of a human, but when images with cameras are analyzed, the unlabeled human in them is found, and the model assumes it made a wrong classification and adjusts the weights wrongly. Furthermore, it was noted that, for example, human eyes could be detected correctly, but as soon as a camera appears in the same frame, the eyes become no longer detectable, as displayed in Figure 15.

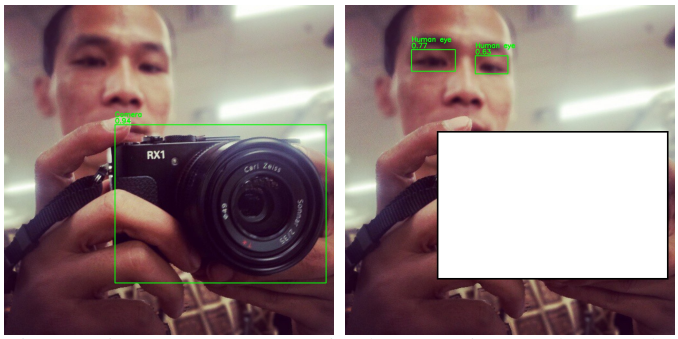
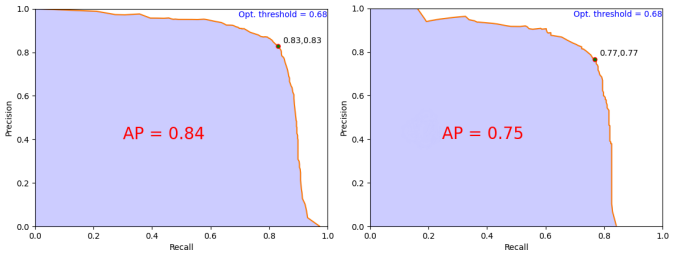


Figure 15: Camera presence in the same image changes the detectability of eyes.

This problem makes it hard to correctly detect when the glasses should turn dark and when not. The issue lies within the dataset, which had to be accurately and thoroughly labeled for all objects of interest to be detected. Nevertheless, it was noted that out of all mentioned classes, class "Human face" performed the best with cameras. There are two reasons for this: this class contains a large number of images for training, thus, not labeled images from the camera class play a lesser role in determining the weights, and though not all, some images containing cameras also have labels for the human face.

Finally, another method that can be used to avoid this problem is to have multiple models trained to detect only one object. Input images will then be used as an input for these separate models, and the outputs of the models will be combined into one image. This, however, decreases the inference time by a factor equal to the number of such models running in parallel. Nevertheless, the images from different classes are no longer affected by each other, and best performing classes for detection of a human ("Human face" and "Person") can be used in parallel with a model detecting only cameras.

Another preliminary experiment was carried out on the dataset to determine the best method to detect photographing and filming. “Camera” and “Mobile phone” classes contain images that are not expected to be detected when the model is deployed, for example, large filming cameras, the display side of cameras and phones, and old phones. Thus, an attempt was made to remove these images from the training and test sets to determine the impact on the accuracy. This resulted in the following changes: “Camera” class images were reduced from 5000 to 4109, and “Mobile phone” class images were reduced from 4312 to 674. The PR curves of models trained with the original and refined dataset are given in Figure 16.



(a) Original dataset (b) Refined dataset

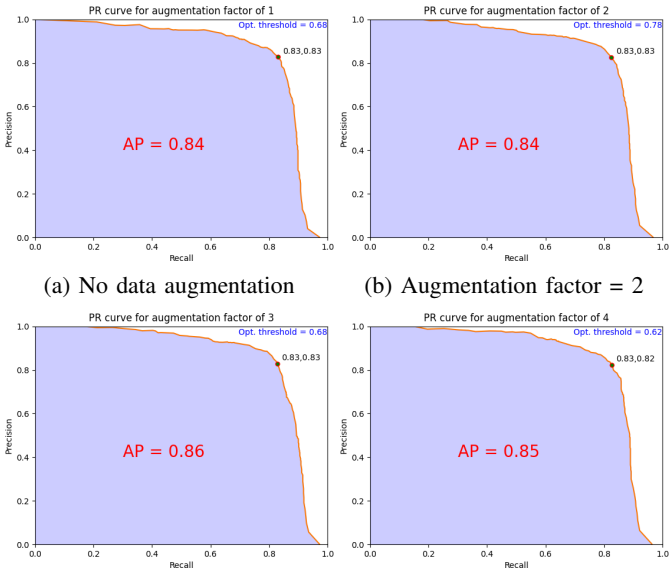
Figure 16: Comparison between PR curves of the original and refined datasets. Evaluation is performed on CD and Model V2 deployed on NCNN FP32.

As can be seen from the areas under the curves, which are equal to AP scores, when tested on the refined test set, the performance of the original model is better. While the removed images did not represent the anticipated reality, they amounted to 51.4% of the entire collection of images, which, in turn, made the model have less available data to be trained with, leading to worse performance. Using data augmentation to bring the classes to about the same number of images as were available initially resulted in no significant improvement. Based on these results, it was decided not to limit the dataset and use all available images for training.

Appendix F
Optimal Data Augmentation Settings

Compared to “Human face” and “Person” classes which contain hundreds of thousands of images, “Camera” and “Mobile phone” classes lack useful training data which can further improve the model. As already mentioned, in such scenarios, data augmentation can be used. To determine the optimal factor by which the dataset of camera images should be increased, an experiment was conducted with datasets of four different sizes. The results are given in Figure 17. Based on the results, augmentation factors of 1 (no data augmentation) and 2 give similar AP scores; an augmentation factor of 3 results in the best AP score, while a factor of 4 brings the AP score down by 0.01. Thus, an augmentation factor of 3 was used for all of the models.

Small object data augmentation was used in this project to enhance the accuracy of small object detections. Results of Kisantal et al. [26] suggest that for most optimal performance,

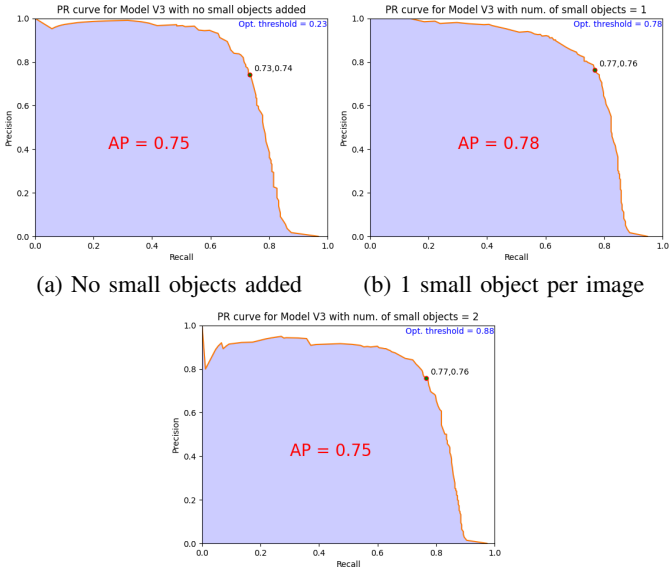


(a) No data augmentation (b) Augmentation factor = 2

(c) Augmentation factor = 3 (d) Augmentation factor = 4

Figure 17: Effects of different augmentation factors on PR curves of a model. Evaluation is performed on CD and Model V2 deployed on NCNN FP32.

only 1-2 small objects need to be added per image. This was tested on the custom dataset made for Model V3, and the results are given in Figure 18. Based on the results, the highest AP score was recorded for a model with one small object added per image. Thus, this factor was used during the training of the Model V3.



(a) No small objects added (b) 1 small object per image

(c) 2 small objects per image

Figure 18: Effects of small object data augmentation on PR curves of a model. Evaluation is performed on MD and Model V3 deployed on NCNN FP32.

Scenario description	YoloFastestV2 Model V3	YOLOv5 model
Samsung smartphone in 3 different poses at 2 meters distance; indoors	Passed	Passed
iPhone smartphone in 3 different poses at 2 meters distance; indoors	Passed	Passed
Huawei smartphone in 3 different poses at 2 meters distance; indoors	Passed	Passed
A smartphone at 2 meters distance in dim lighting conditions; indoors	Passed	Passed
A smartphone at 8 meters distance; indoors	Passed	Passed
Phone-like object at 2 meters distance; indoors	Failed	Passed
A smartphone at 2 meters distance; outdoors	Passed	Passed

Table VI: Real-world experiments conducted on YoloFastestV2 Model V3 and YOLOv5 model [35].

APPENDIX G
ISSUES WITH MODEL QUANTIZATION ON MNN FRAMEWORK

Both NCNN and MNN frameworks require calibration datasets to convert FP32 models to INT8. Additionally, both, by default, use the KL divergence method to quantize the features and the max absolute value of weights to perform symmetrical quantization [10, 11]. While the conversion procedure is similar, a different set of tools is used by MNN. During the debugging of the model quantization, it was observed that *cosine distance* parameter of one of the layers results in NaN value as shown in Figure 19. This parameter error results in incorrect model quantization and poorly performing MNN INT8 models. GatherV2 was found to be the layer that had the corresponding input given by the debugger (see Figure 20). This same layer is also responsible for the error occurring during conversion from ONNX to TFLite (see Appendix C). Thus, due to the unsupported implementation of the GatherV2 layer in ONNX [40], the MNN quantization process fails and results in worse-performing models.

```
Debug info:

input.104 output_tensor_0:  cos distance: 0.536165, overflow ratio: 0.000010
input.108 input_tensor_1:  cos distance: nan, overflow ratio: 1.000000
input.108 output_tensor_0:  cos distance: 0.420387, overflow ratio: 0.000000
input.116 input_tensor_0:  cos distance: 0.420996, overflow ratio: 0.000000
input.116 output_tensor_0:  cos distance: 0.775863, overflow ratio: 0.000000
```

Figure 19: MNN quantization error.

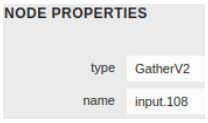
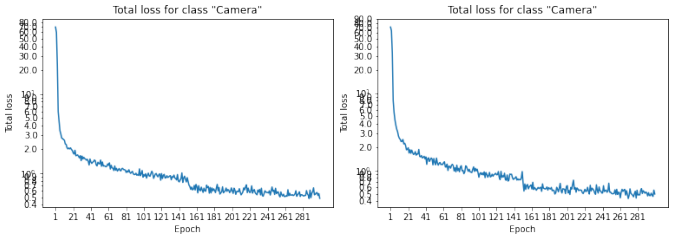


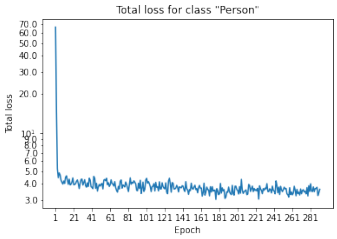
Figure 20: Layer responsible for MNN quantization error.

APPENDIX H
TOTAL LOSS OF MODEL V3

During model training, the total loss function can be used as an indicator of how well the model is learning from the data. This function needs to be minimized to achieve better results. From Figure 21, it can be seen that values for total loss for different sub-models of Model V3 flatten out, meaning that the detector becomes stable over time.



(a) Camera-and-phone sub-model (b) Phone-only sub-model



(c) Person sub-model

Figure 21: Total loss plots of Model V3. Camera-and-phone and phone-only detection sub-models are presented with the person detection sub-model.

APPENDIX I
REAL-WORLD EXPERIMENTS ON YOLOFASTESTV2 AND YOLOV5 SYSTEMS

Experiments were conducted on both the proposed Model V3 and YOLOv5 model designed by [35] to observe the real-world performances. Seven different scenarios were considered, and the results are summarised in Table VI.