ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Integrierte Systeme
Integrated Systems Laboratory

DEPARTMENT OF
INFORMATION TECHNOLOGY AND ELECTRICAL ENGINEERING

Spring Semester 2023

Hand-Detection for Sign Language Aid on Android-Based Smart Glasses

Semester Project

Luca Specht
lspecht@student.ethz.ch

03.07.2023

Advisors: Andrea Ronco, andrea.ronco@pbl.ee.ethz.ch
Luca Pascarella, lpascarella@ethz.ch

Professor: Prof. Dr. L. Benini, lbenini@iis.ee.ethz.ch

Abstract

This thesis explores the feasibility of real-time hand detection for sign language translation using YOLOv5 and YOLOv7 models on the Snapdragon 4100 wearable SoC. The project involves training and optimizing various YOLO models, with the best performing model achieving an mAP@50 score of 0.977 and an inference time of 71 ms on the CPU. The results demonstrate reliable real-time hand detection. Limitations such as the tedious model conversion between different ML frameworks and the underwhelming GPU acceleration using NNAPI have been identified.

Acknowledgments

First and foremost I would like to thank my supervisors, Andrea Ronco and Luca Pascarella for their guidance, assistance and feedback throughout this semester thesis. I would also like to acknowledge PBL for funding the material used in this project.

Declaration of Originality

Contents

Chapter 1

Introduction

Sign language is a visual language mostly used by deaf people to communicate. They use hand gestures and facial expression to convey meaning. Technological advancements in AR and machine learning have enabled automated sign language translation with the use of cameras.

One of the essential components of sign language translation is real-time hand detection. This is a demanding task due to challenges such as variance in viewpoints, intra-class variation, self-occlusion of fingers and complex background settings. Recent advancements in computer vision using deep learning have enabled significant improvements in object detection. YOLO (You Only Look Once) is the state-of-the-art object detection algorithm. It is based on a CNN (Convolutional Neural Network) and its efficiency allows for real-time object detection. The context of wearable applications bears several limitations that have to be considered. These include low-resolution cameras, limited performance, and restrictions in power consumption. To facilitate these challenges several optimizations in hardware as well as software may be conducted. The use of hardware accelerators such as Digital Signal Processors (DSPs) and Graphics Processing Units (GPUs) increases inference speed of neural networks. On the other hand, modifications to the neural network such as architecture adjustment and quantization can further improve performance. Lastly, having a well-balanced dataset is a crucial aspect as it directly impacts accuracy of a model.

This semester thesis aims to evaluate the feasibility of hand detection for sign language using a wide variety of recent YOLO models on the TurboX SDW4100 development board which comprises a state-of-the-art system-on-chip (SoC) for wearables. The report is structured in the following way. First of all, Chapter 2 explains the working principle of YOLO and discusses hardware components of the SDW4100. Then the report addresses some related work in Chapter 5. Chapter 3 covers the entire implementation pipeline including, dataset selection, neural network training, optimizations and deployment on the target hardware. Afterwards results are presented and evaluated in Chapter 4. Lastly, Chapter 6 concludes this project and reflects on future work.

Chapter 2

Background

2.1. YOLO

In the year 2016 Joseph Redmon et al. proposed YOLO (You Only Look Once); a real-time object detection algorithm. Unlike other region proposal networks, YOLO makes use of a single neural network which outputs object bounding boxes and class probabilities all in one forward pass. The unified architecture is extremely fast allowing to process up to 45 frames per second. YOLO outperforms detection methods like DPM and fast R-CNN by a wide margin making it one of the most popular algorithms for real-time object detection and classification. YOLO has undergone several improvements across different versions since then, however, it has retained its key characteristics throughout.

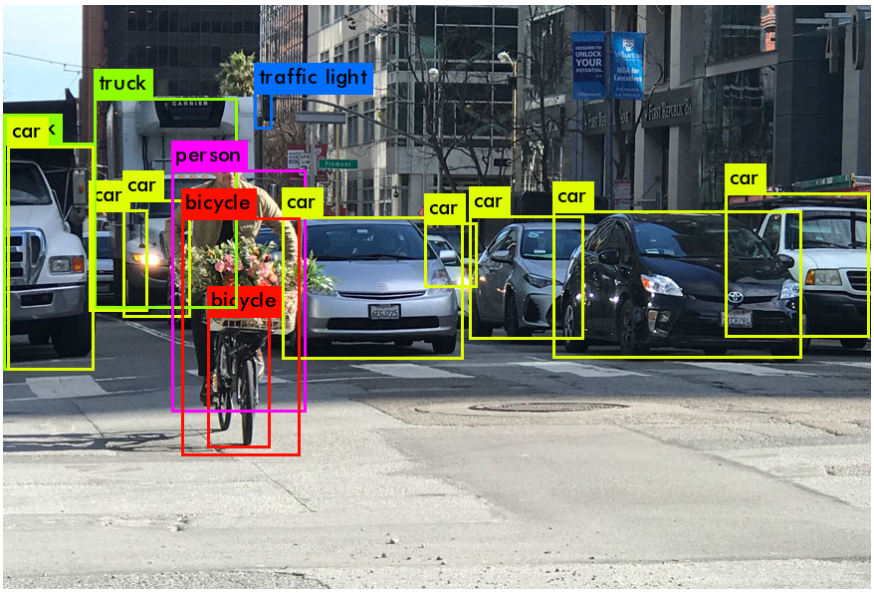


Figure 2.1.: YOLO in action [1].

2.1.1. Working Principle

YOLO divides the input image into a *SxS* grid. If the bounding box center of an object falls into a grid cell, that cell is responsible for the detection of that object. Every cell predicts B bounding boxes. Each bounding box is defined by four parameters: x, y coordinates of its center, height h and width w. Furthermore, a bounding box is associated with a confidence score which reflects the model's certainty that there is an object in the box. This amounts to 5 predictions per bounding box. Lastly, the model generates C class probabilities for every grid cell, the highest of which determines the detected object. Consequently, the output tensor has *S x S x (5*B + C)* prediction values [2].

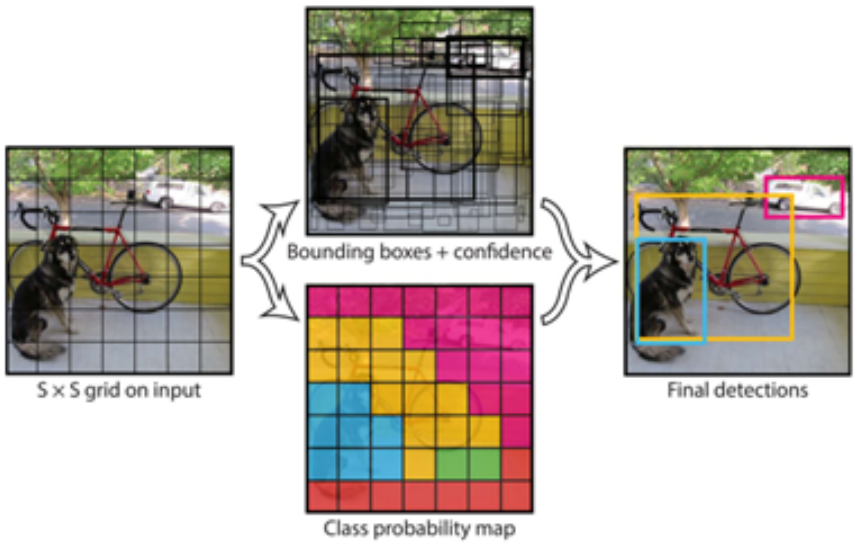


Figure 2.2.: Overview of YOLO [2].

The resulting bounding boxes might overlap or cover the same object multiple times, leading to redundant predictions. In order to eliminate duplicates and select the most relevant boxes following post-processing steps are performed:

- Initially, bounding boxes with a confidence score below a certain threshold are discarded. This process is called **Confidence Thresholding**.
- For the remaining bounding boxes, the **Intersection over Union (IoU)** is calculated to determine the extent of overlap between them. IoU is defined as the ratio of the intersection area between two bounding boxes (BB_1 and BB_2) and their union area.

$$IoU = \frac{BB_1 \cap BB_2}{BB_1 \cup BB_2}$$

- In a last step called **non-maximum suppression (NMS)**, all bounding boxes are sorted based on their confidence scores. Starting with the box having the highest confidence score, NMS compares its IoU with the remaining boxes. If the IoU exceeds a predefined threshold, the box with lower confidence is suppressed. If the IoU is below the threshold, the box is considered a separate detection. This process is repeated iteratively for all boxes until no more boxes remain. The surviving boxes after the suppression process are considered the final detections.

Figure 2.3.: Simple Example of NMS [3].

2.1.2. Network Architecture

YOLO classifies as a convolutional neural network whose architecture can be roughly divided into three components:

- **Backbone:** The backbone network serves as a feature extractor and typically consists of multiple blocks consisting of several convolutional layers followed by a pooling layer. It takes the input image and progressively reduces its spatial dimensions while increasing the depth of feature maps. This component is responsible for learning high-level representations of the input image. Specific choices for the backbone network will be discussed in YOLOv5 and YOLOv7 chapters.
- **Neck:** The neck connects the backbone to the head. The purpose of the neck is to enhance the feature representation by aggregating and concatenating features from multiple levels of the backbone network. Some versions consider the neck as part of the backbone network.
- **Head:** The head predicts bounding boxes and class probabilities based on the features extracted by the neck and backbone network. It is also referred to as the detection head. It usually consists of a few additional convolutional layers followed by fully connected layers. The detection head predicts the final bounding boxes as well as the corresponding class probabilities and confidence scores. It employs techniques like anchor boxes, which are predefined boxes used to match the predicted bounding boxes with ground truth objects.

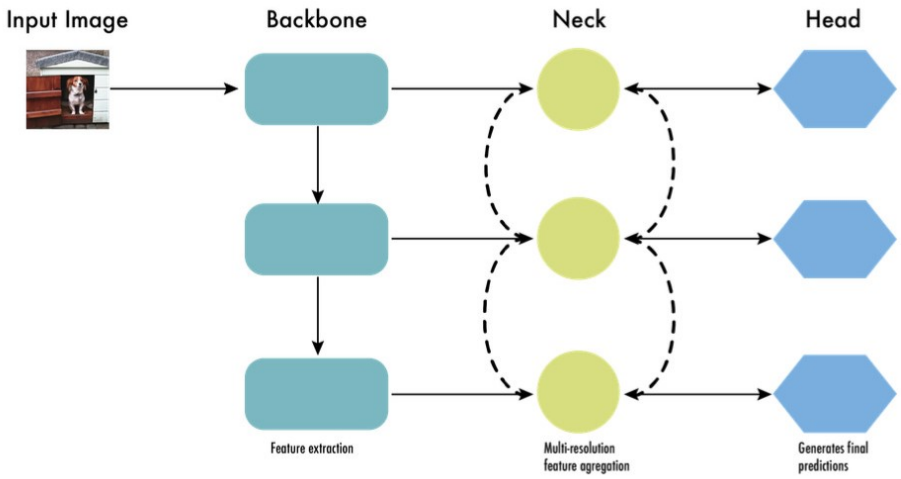
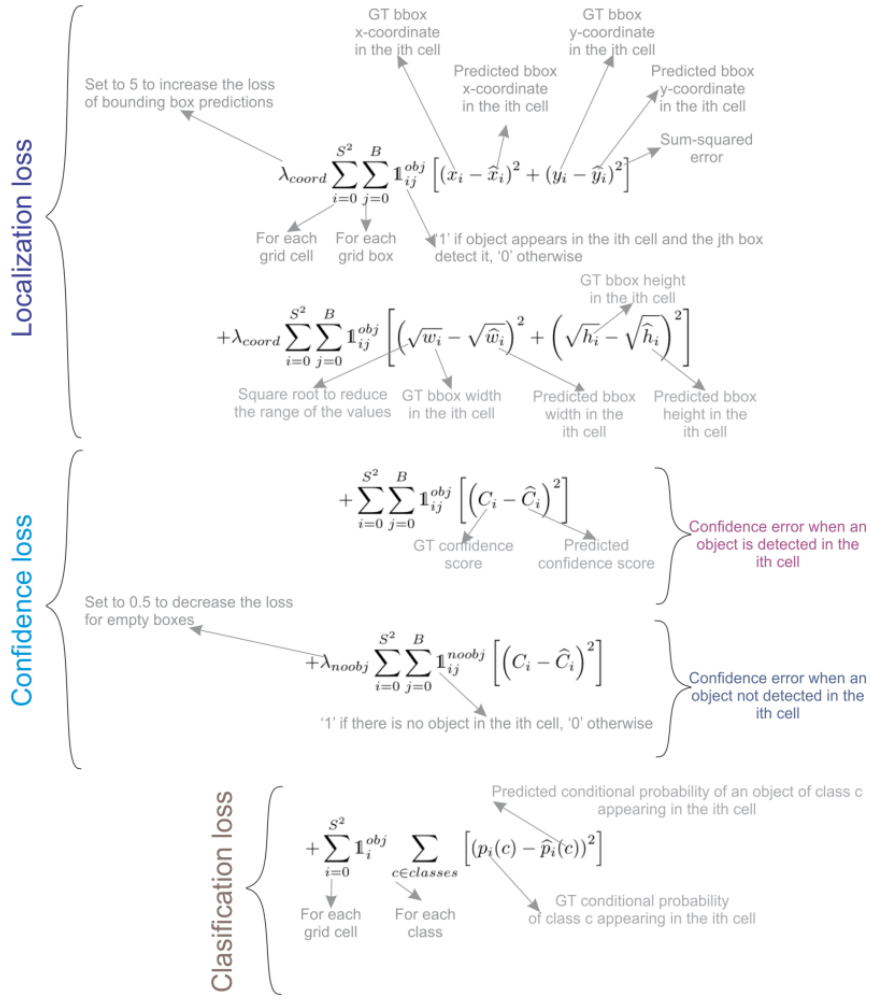


Figure 2.4.: Backbone, neck and head of YOLO [4].

2.1.3. Loss Function

YOLO predicts multiple bounding boxes per grid cell. To compute the loss we only want one of them to be responsible for the object. For this purpose, we select the one with the highest IoU with the ground truth. The YOLO loss function comprises localization, classification and confidence loss [4]. The **localization loss** measures the difference between the predicted bounding box coordinates and the ground truth bounding box coordinates. The **confidence loss** compares the predicted confidence scores with the ground truth labels (1 if the bounding box contains an object, 0 otherwise). The **class loss** measures the difference between the predicted class probabilities and the ground truth class labels. The total loss is the sum of these individual losses, weighted by hyperparameters. The hyperparameters help balance the contribution of each component in the loss function.



2.1.4. Different YOLO versions

Since its initial release in 2015 different versions and variants of YOLO have been proposed, each providing their own improvements in performance and efficiency. YOLOv1 to YOLOv3 were created by the original authors of YOLO - Joseph Redmon and his advisors. Starting from YOLOv4, the legacy of YOLO was continued by other researchers since Redmon had stopped his work in computer vision. Since then, various companies have published their own models as new YOLO versions such as YOLOv6 and YOLOv8, trying to profit from the YOLO hype without providing any significant improvements. A detailed timeline of all YOLO releases can be found in Fig. 2.5. Most notable enhancements were attained through refined network architectures, optimized training methods and tweaked loss functions [4].

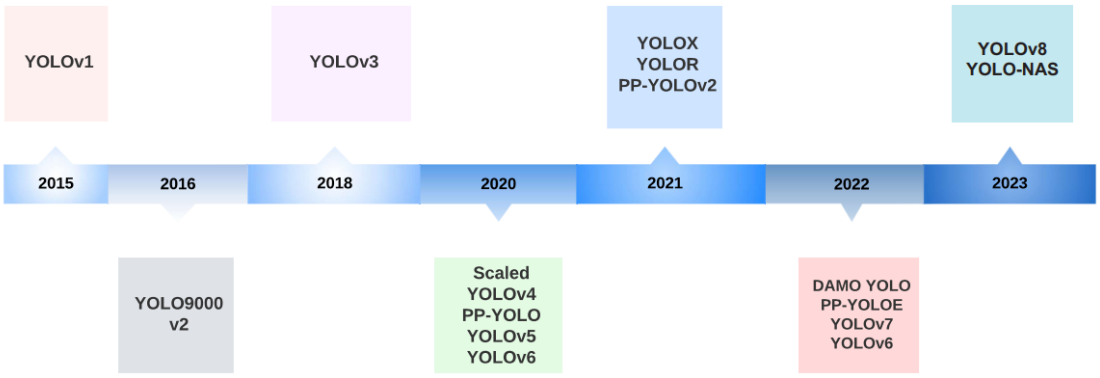


Figure 2.5.: Timeline of YOLO releases [4].

Initial YOLO versions were developed on the Darknet framework, whereas newer releases have been ported to Pytorch. Moreover, from YOLOv5, most releases shifted their focus on fine-tuning the trade-off between speed and accuracy. Following subsections will provide an overview of YOLOv5 and YOLOv7, the two YOLO architectures used in this project.

YOLOv5

YOLOv5 uses many improvements introduced in the YOLOv4 release but was developed in Pytorch instead of Darknet. The use of new data augmentation and regularization techniques - commonly referred to as **"bag of freebies"** - allowed the model to generalize better to new data. Fig. 2.6 displays some of these data augmentation techniques. On the other hand, **"bag of specials"** refers to a set of network modifications to the backbone and neck such as residual connections, path aggregation network (PAN) or spatial pyramid pooling (SPP) [5].

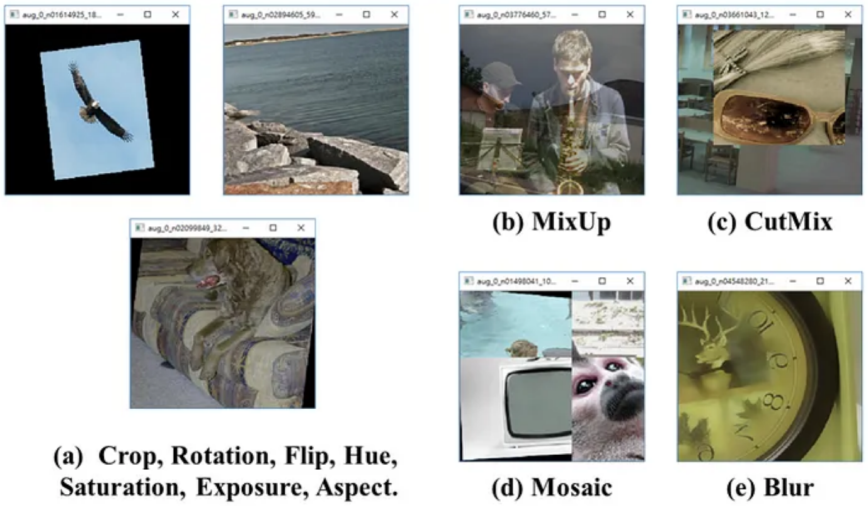


Figure 2.6.: Data augmentation techniques introduced by YOLOv4 and YOLOv5.

The architecture consists of a modified CSPDarknet53 backbone with a Stem (a strided convolution layer with a large window size) followed by convolutional layers that extract image features. A spatial pyramid pooling fast (SPPF) layer is used to accelerate computation by pooling features into a fixed-size map. Each convolution has batch normalization and SiLU activation. The head is based on YOLOv3 and supports **multi-scale predictions** which allows the model to detect objects at different sizes by leveraging multiple grid sizes. Furthermore, YOLOv5 utilizes **anchor boxes** (first introduced in YOLOv2), which are bounding boxes with predefined dimensions to match prototypical object shapes.

YOLOv5 offers five scaled variations: `yolov5n` (nano), `yolov5s` (small), `yolov5m` (medium), `yolov5l` (large), and `yolov5x` (extra large). Width and depth of the convolution modules vary across versions to suit specific applications and hardware requirements. This project targets the use of the two smallest sized networks (`yolov5n`, `yolov5s`) and also introduces an even tinier implementation referred to as `yolov5p` (pico).

YOLOv7

YOLOv7 is structured around YOLOv5 and introduces two new key features: Extended efficient layer aggregation network (E-ELAN) and a novel scaling strategy for concatenation-based models. E-ELAN optimizes learning efficiency by manipulating feature grouping and merging without disrupting the original gradient path. The new scaling strategy ensures the optimal structure of concatenation-based models by scaling the depth and width of the block with an equal factor. Lastly, leaky ReLU instead of SiLU is used as the activation layer.

2.2. SDW4100

The target hardware used for inference of the trained networks is the SDW4100 development kit which boasts a Snapdragon Wear 4100 System-on-Chip that was released in 2020. Table 2.1 compiles its most important hardware features and specifications. Moreover, the development board comes with a rounded 1.2-inch display, a 2MP camera and various sensors. The system is Wear OS compatible but for this project Android 8.1 was used.

CPU	4-Core ARM Cortex-A57
GPU	Adreno 504
DSP	Hexagon v56
Memory	LPDDR3, 750MHz
Process Node	12nm
Coprocessor	Qualcomm QCC1110

Table 2.1.: Snapdragon Wear 4100 specifications [6]

The two standout components are the DSP and GPU as they hold the potential to enable hardware acceleration of the YOLO network.

- **DSP**: DSPs are specialized microprocessors designed for high-performance signal processing applications. They are designed with specific hardware features such as multiply-accumulate (MAC) units and SIMD (Single Instruction Multiple Data) instruction sets. MAC units can perform both multiplication and accumulation in a single operation, enabling efficient execution of matrix operations which form the principal component of a neural network. Furthermore, SIMD allows for parallel processing of multiple data elements simultaneously, which again can significantly accelerate matrix operations. The Snapdragon Wear 4100 embeds a Hexagon v56, a VLIW 32-bit DSP.

- **GPU**: Originally developed for rendering graphics, GPUs have become a central component of modern machine learning systems. They usually consist of hundreds of small processing cores that can work together to process large amounts of data simultaneously. GPUs are well-suited for training and executing neural networks due to their ability to perform matrix operations in parallel. For instance, deep learning frameworks, such as TensorFlow and PyTorch, have leveraged the parallel processing capabilities of GPUs to accelerate training times for complex neural networks significantly. The Snapdragon Wear 4100 comes with an Adreno 504 GPU that has a clock speed of up to 320 MHz.

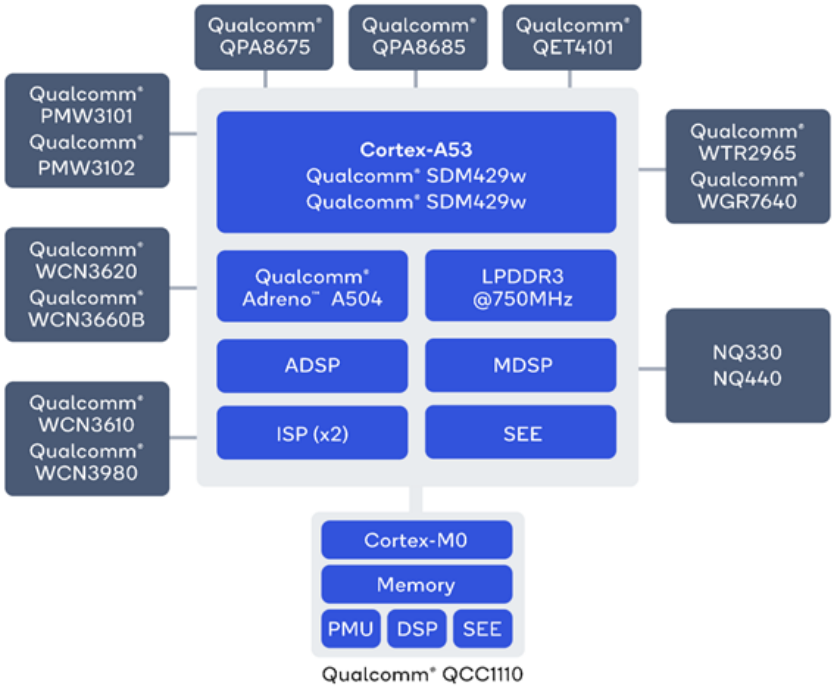


Figure 2.7.: Snapdragon Wear 4100 hardware overview [6].

Chapter 3

Implementation

3.1. Dataset

The dataset used for this thesis is **HaGRID (Hand Gesture Recognition Image Dataset)** [7]. It contains 552'992 Full HD (1920 x 1080) images which amounts to 716 GB of data. The images are originally divided into 18 classes of different gestures, however, for the sake of this project they are all merged into a single "hand" class. The training-test split is by default 92%/8%. The dataset contains over 34'000 different people from various age groups and ethnicities. The scenes range from dark indoor environments to bright outdoor settings. The subjects were instructed to show different gestures, mostly in third person, with a 0.5m to 4m distance to the camera. Fig. 3.1 shows a subset of images from HaGRID.

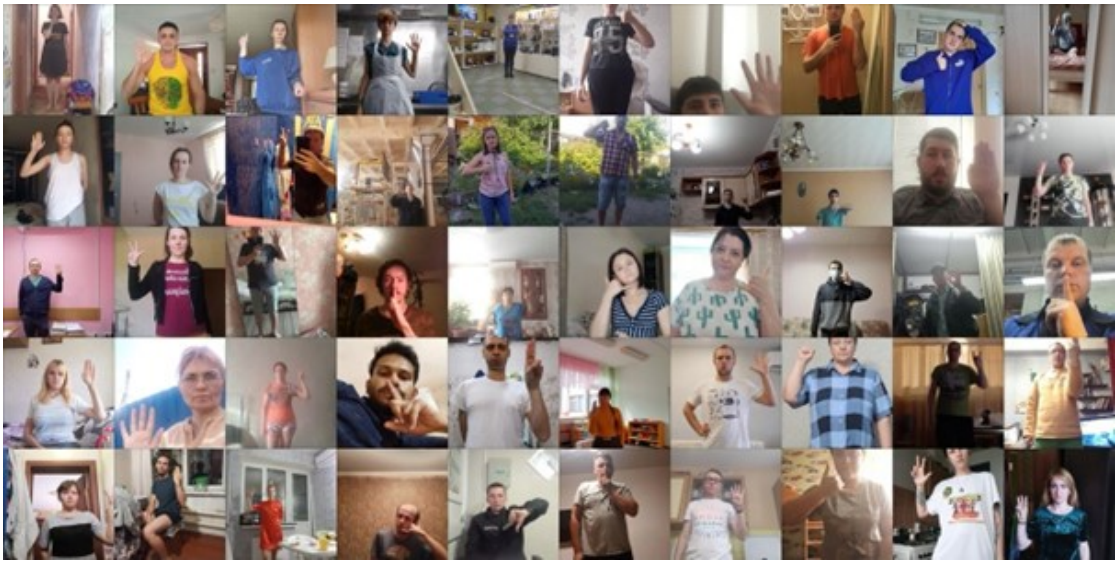


Figure 3.1.: Subset of images from HaGRID [7].

The HaGRID dataset seems to be an optimal choice for the task of hand recognition in the context of sign language detection for multiple reasons. First of all, the dataset is enormous and comes with a great variety of different scenes. This is substantial for the training of a reliable and well-performing model. Second, most of the images are captured in third person. Consequently, the dataset is a good fit for the use case of sign language detection where target subjects are also going to be presented in third person to the camera. Lastly, the variety in gestures will make the trained network more robust to variables such as finger occlusion and hand position as this is also the case for sign language.

Fig. 3.2 shows a sample of an image from HaGRID together with its JSON annotation. The annotation compiles bounding box positions (top left corner coordinates x, y) and dimensions (width w and height h) as well as gesture classification and confidence score of each hand. Bounding box parameters x, y, w and h are given as normalized relative values with origin point in the top left corner. Furthermore, each person is denoted with a unique user ID and there is a distinction between left and right hand.


```
"09ee9bcb-076d-4a3a-97042a": {
    "bboxes": [
      [
        0.42639135,
        0.36274419,
        0.12887326,
        0.09358317
      ],
      [
        0.68150371,
        0.82525372,
        0.06819769,
        0.08949067
      ]
    ],
    "labels": [
      "peace",
      "no_gesture"
    ],
    "leading_hand": "left",
    "leading_conf": 1.0,
    "user_id": "ef8506x2df0a0"
}
```


Figure 3.2.: Sample image with and its annotation [7].

These annotations had to be converted to YOLO labels. Confidence values and bounding box dimensions were directly transferred from the original HaGRID annotations. YOLO uses center box coordinates whereas HaGRID annotations rely on top left corner coordinates to define the bounding box position. Thus YOLO coordinates were simply deduced using the following formulas:

$$x_{YOLO} = x_{HaGRID} + \frac{w}{2}$$

$$y_{YOLO} = y_{HaGRID} + \frac{w}{2}$$

Lastly, all class labels were converted to class "0" which represents the hand class. Remaining labels were omitted. Fig. 3.3 shows the corresponding YOLO labels of the sample annotation shown in Fig. 3.2.

```
0 0.49082798 0.40953577500000005 0.12887326 0.09358317
0 0.715602555 0.869999055 0.06819769 0.08949067
```

Figure 3.3.: YOLO labels derived from Fig. 3.2.

3.2. Training

Table 3.1 shows the different YOLO models that were trained using the HaGRID dataset. The number of parameters refers to the number of distinct weights and biases that can be learned during training. More parameters in a CNN allow the model to capture fine-grained details and intricate patterns at the cost of increased computational complexity. The same applies to the number of layers.

Model	**# Parameters**	**# Layers**	**Size (FP32)**
Yolov7-tiny	6'014'038	255	23.5 MB
Yolov5s	7'022'326	214	28.6 MB
Yolov5n	1'765'270	214	7.3 MB
Yolov5p	318'046	200	1.6 MB

Table 3.1.: YOLO models and their sizes

Yolov7-tiny, Yolov5s and Yolov5n are commonly used YOLO architectures in industry and research, whereas Yolov5p was specifically tailored for this project. With only 318k parameters this model is supposed to minimise inference speed while maintaining reasonable precision.

All models were trained on a Nvidia RTX 3080 GPU using stochastic gradient descent with the following hyperparameters: learning rate = 0.01, momentum = 0.937, weight decay = 0.0005, batch size = 32 and epochs = 1. With the COCO dataset pretrained models were used to initialise weights for training. Since the HaGRID dataset consists of over half a million images one training epoch turned out to be sufficient and also prevented overfitting.

3.3. Optimizations

3.3.1. Input Size

Varying the input size of the image to the neural network has a direct impact on inference speed and performance. Increasing the input size enables the model to capture more fine-grained details, leading to improved accuracy. It allows the network to perceive objects at different scales and enhance spatial information. However, larger input sizes result in longer inference time. Although the number of network parameters doesn't change with increasing input size, the amount of required operations increases as filters used for convolution and pooling layers have to process more pixels. Intuitively, decreasing the input size has the opposite effect. For this project input sizes of 640×640, 320×320 and 224×224 were used.

3.3.2. Quantization

Quantization is a technique that reduces the precision of network parameters, namely weights, biases, and activations. Instead of using 32-bit floating-point numbers, quantization represents them with lower bit precision, such as 8-bit integers. This technique offers two main advantages at the cost of reduced accuracy:

- **Reduced Memory Footprint:** Quantization significantly reduces the memory required to store model parameters. For instance, a model using 32-bit floating-point parameters requires around $4\times$ more memory compared to its quantized 8-bit integer version.
- **Improved Inference Speed:** Quantized models perform computations with lower precision, which can be executed faster on hardware. This acceleration can lead to improved inference speed, enabling real-time applications.

3.4. Export

The trained model were exported using **ONNX (Open Neural Network Exchange)**, a standard format for representing machine learning models that enables interoperability between various machine learning frameworks. The end goal is to deploy the models on TensorFlow Lite, a lightweight machine learning framework designed to run on Android mobile devices. A huge benefit of TensorFlow Lite is that it supports **NNAPI (Neural Network Application Programming Interface)** which allows for easy implementation on the DSP or GPU on the target device [8]. PyTorch Mobile is another alternative for running inference on Android [9]. However, this solution lacks support of hardware acceleration. One common problem that emerges when exporting a model from PyTorch to TensorFlow Lite is their mismatch in input sizes. While PyTorch uses the NCHW (Number of Batches, Channels, Height, Width) format, TensorFlow and TensorFlow Lite use the NHWC format [10]. To enable a seamless conversion without the need of additional transpose layers another framework called **OpenVino** is resorted to. The final export flow looks as follows:

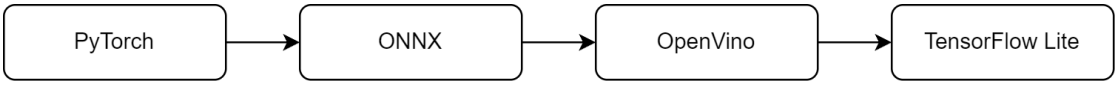


Figure 3.4.: Export flow.

Some of the layers included in the detection head of YOLOv5 aren't supported by NNAPI. For this reason, only the backbone of the architecture was exported. The detection head is implemented separately and run on the CPU as explained in the next subchapter. As for YOLOv7, some operations in the backbone network were identified to not be compatible with TensorFlow Lite. Consequently, YOLOv7 based models were deployed using PyTorch Mobile without the option of hardware acceleration.

Furthermore, it was found that the DSP of the target hardware (SDW4100) wasn't accessible via NNAPI, thus preventing potential acceleration of inference. This leaves the GPU as the sole hardware accelerator besides the CPU.

3.5. Android App

The scope of this project involved the development of an Android app which is capable of running inference on a selected network configuration using the camera or an arbitrary image as input. Test results were evaluated using this app. The GUI of the app is shown in Fig. 3.5. The user chooses model configuration (model, input size, quantization) and post-processing settings (confidence and IoU threshold) along with the desired input feed (image or camera). Besides displaying bounding box predictions and confidence scores the app maintains a log of inference and post-processing times.

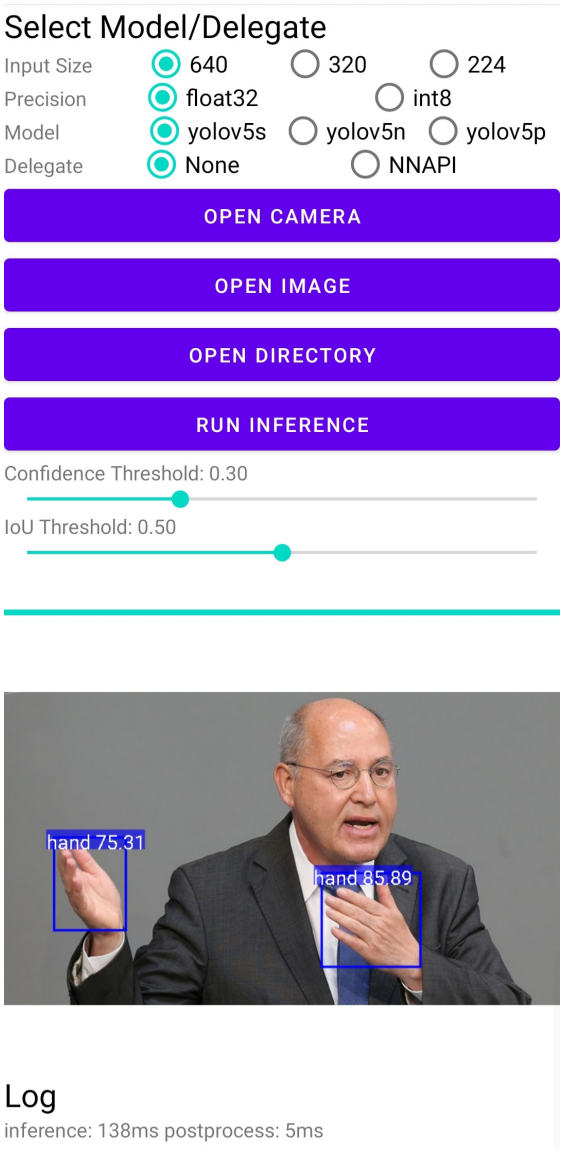


Figure 3.5.: User interface of the app.

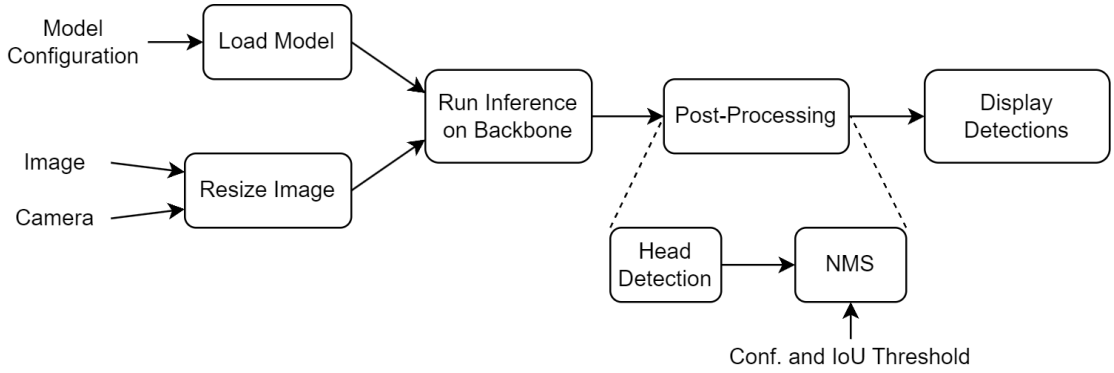


Figure 3.6.: App operation flow.

The source code of the app is based on a prior implementation by Yasuhiro Nitta [11] and has been updated and modified to fit the needs of this project. Fig. 3.6 illustrates the operating principle of the app. The input image is resized and forwarded using the exported TensorFlow Lite model which includes the backbone of the network. Depending on the users choice, the backbone is executed either on the CPU using 4 cores or on the GPU via the NNAPI delegate. The output of the backbone model is then used as an input to the detection head which is implemented as an external native C++ program using the **JNI (Java Native Interface)**. Irrespective of any settings the detection head always runs on the CPU. The resulting bounding boxes are then filtered in a final post-processing step using confidence and IoU thresholds which were set in the GUI.

Chapter 4

Results

The accuracy and inference speed of each network configuration was evaluated on a test set consisting of 43,669 images which were all unseen during training.

4.1. Metrics

Following metrics were used to quantify the accuracy of the trained networks.

- **Precision** measures the proportion of correctly predicted hands (true positives) out of the all predicted hands (true positives and false positives).

$$\text{Precision} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}}$$

  A higher precision value indicates that the models detections are more likely to be actual hands.

- **Recall** measures the proportion of correctly predicted hands (true positives) out of all existing hands specified by the ground truth (true positives and false negatives).

$$\text{Recall} = \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}}$$

  A higher recall value indicates that the model can effectively detect most of the hands present in the image.

To determine whether an object counts as true positive or false positive the IoU of prediction and ground truth has to be computed. If a predefined IoU threshold is exceeded, the prediction is considered a true positive. This IoU threshold is not to be confused with the IoU threshold used for NMS. Both precision and recall values depend on the confidence threshold chosen in post-processing (see Chapter 2). Having a

high confidence threshold will result in better precision as bounding boxes with lower confidence values are discarded, whereas having a low confidence threshold yields better recall as more bounding boxes are considered. To characterise this dynamic a **Precision-Recall (PR) curve** is used [12]. The PR curve is generated by plotting precision against recall at various confidence thresholds (Fig. 4.1). With the help of this graph, the final and most important metric can be defined:

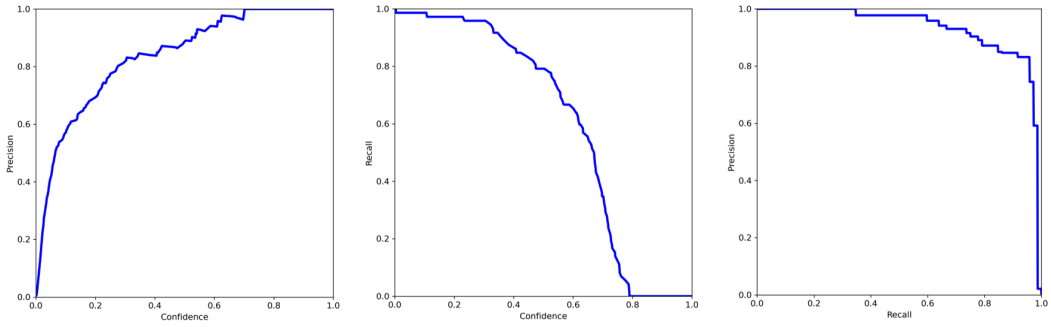


Figure 4.1.: P-curve (left), R-curve (middle) and PR-curve (right).

- The **Average Precision (AP)** is a way to summarize the precision-recall curve into a single value representing the average of all precisions. The AP can be computed using the following equation:

$$\mathrm{AP} = \frac{1}{n} \sum_{r} P(r) \times \Delta r$$

  where: n is the total number of recall values, $P(r)$ is the maximum precision at recall level r, Δr represents the change in recall between adjacent levels.

  In other words, the AP corresponds to the area under the PR curve.

- **Mean Average Precision (mAP)** is the mean AP across all classes. As there is only one class in the hand detection task, the two are equivalent.

Inference time refers to the spent on a single forward pass of a neural network. This is the most crucial performance metric, particularly for real-time applications. Assuming negligibly small pre- and post-processing times, the attainable FPS (frames per second) for real-time hand detection using a camera can be approximated as:

$$FPS = \frac{1}{t_{inference}}$$

4.2. Evaluation

Table 4.1 shows the resulting accuracy and inference speed metrics from every single model configuration which was evaluated. The model configuration comprises model, input size and precision. Its string is of the format `model_input-size_quantization`. The displayed precision and recall values implicate the tuple in the PR curve with the greatest **F1 score**. The F1 score computes the harmonic mean of precision and recall:

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}$$

Furthermore, mAP at an IoU threshold of 50% and 95% were evaluated. Lastly, inference time on the CPU and GPU (using NNAPI) were measured and averaged across 10 inference runs on the same test image. Post-processing time including NMS was neglected as it only accounts for a few milliseconds. The following subchapters go into more details and conclude the most important findings based on the measured results.

Model Config.	Precision	Recall	mAP@50	mAP@95	Inf. time CPU (ms)	Inf. time GPU (ms)
yolov7-tiny_640_fp32	0.992	0.9873	0.9962	0.8438	1590	Na
yolov7-tiny_320_fp32	0.9793	0.9714	0.9931	0.7758	400	Na
yolov5s_320_fp32	0.9884	0.9817	0.9941	0.7808	322	598
yolov5s_320_int8	0.965	0.967	0.991	0.71	170	367
yolov5s_224_fp32	0.9814	0.9734	0.9927	0.7639	172	311
yolov5s_224_int8	0.939	0.944	0.982	0.661	92	172
yolov5n_640_fp32	0.9894	0.9823	0.994	0.7977	420	757
yolov5n_640_int8	0.98	0.97	0.992	0.729	258	651
yolov5n_320_fp32	0.978	0.9584	0.989	0.7247	123	208
yolov5n_320_int8	0.947	0.934	0.98	0.649	81	177
yolov5n_224_fp32	0.9662	0.9336	0.977	0.6859	71	124
yolov5n_224_int8	0.916	0.897	0.956	0.589	54	99
yolov5p_640_fp32	0.9492	0.8323	0.9195	0.6314	223	310
yolov5p_640_int8	0.944	0.819	0.91	0.587	161	350
yolov5p_320_fp32	0.911	0.7808	0.8734	0.5472	63	103
yolov5p_320_int8	0.842	0.731	0.82	0.445	50	108

Table 4.1.: Evaluation results of every model

Remark: Updating from TensorFlow Lite v2.4.0 to v2.11.0 resulted in a 20%-40% reduction in inference speed across all model configurations.

4.3. Accuracy

The mAP score is the most comprehensive accuracy metric as it takes into account both, precision and recall. In Fig. 4.2 the mAPs at 50% and 95% IoU threshold are depicted for every model configuration. These IoU thresholds define the minimum level of overlap required for a predicted bounding box to be considered a true positive. By adjusting the IoU threshold the margin of error can be controlled. The mAP deviations are negligible for yolov7-tiny, yolov5s and yolov5n. However, there is a drastic decline in mAP for yolov5p. This implies that the reduced network size of yolov5p fails to withstand the complexity of the hand detection task. Additional real-life testing concluded that an mAP@50 of around 0.90 is required to guarantee reliable real-time hand detection. Lastly, yolov7-tiny didn't provide any significant improvements compared to its equivalent predecessor yolov5s.

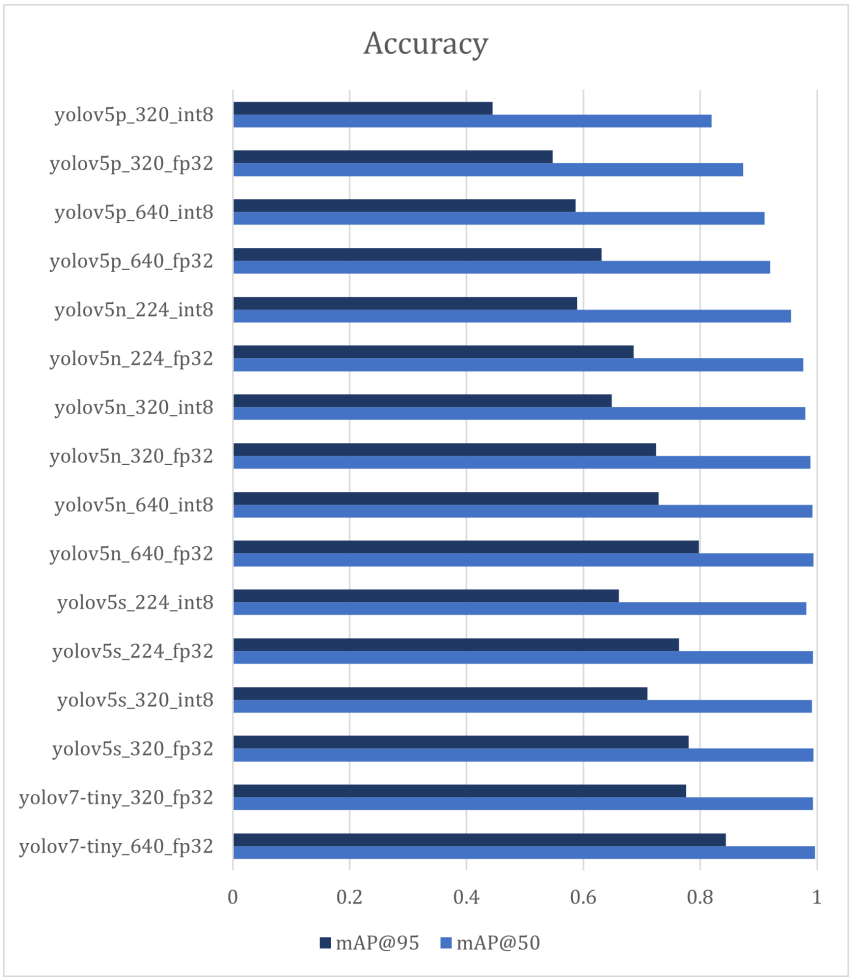


Figure 4.2.: Mean Average Precision of each model.

4.4. GPU vs. CPU

Fig. 4.3 displays a direct comparison of inference time between CPU and GPU for every model configuration. It can be observed that there is around a 2× slowdown caused by the deployment of NNAPI which uses the GPU as hardware accelerator. In conclusion NNAPI doesn't provide any benefit in terms of performance. This isn't surprising considering the Adreno 504 is a lower midclass GPU with a single core clocked at 320 MHz. In contrast, the CPU has 4 cores with a maximum clock speed of 1.7 GHz. Besides that, data transfer with limited VRAM and bandwidth potentially deteriorates performance of the GPU execution. Lastly, flawed implementation of NNAPI cannot be ruled out as an additional reason.

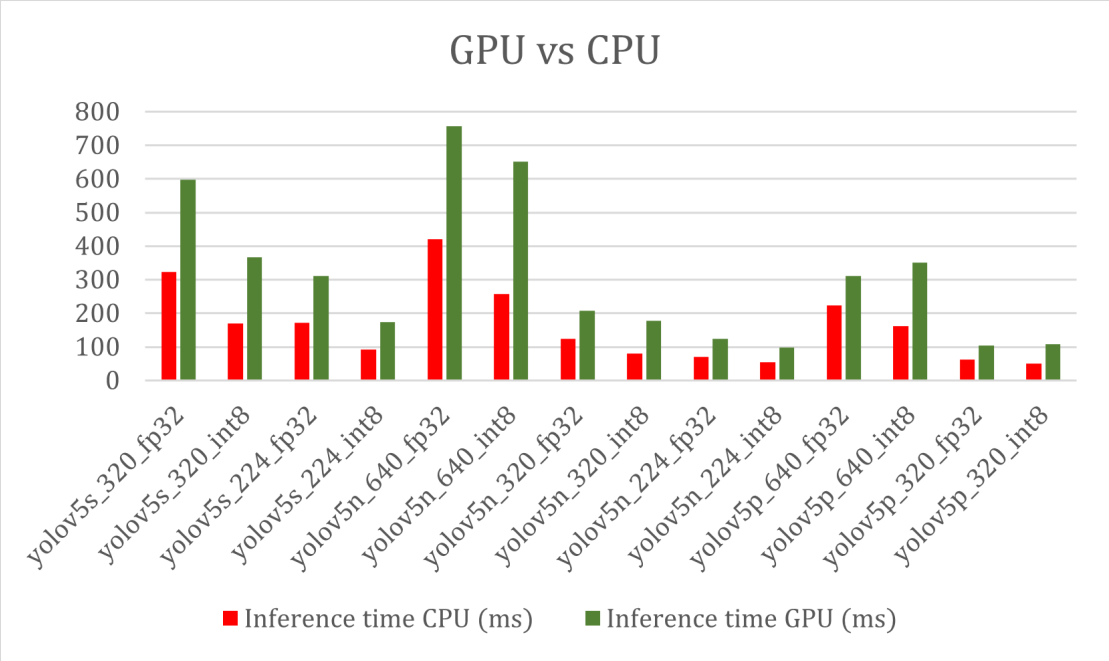


Figure 4.3.: Performance comparison between CPU and GPU.

4.5. Input Size Effect

Inference time scales almost 1:1 proportionally with input size as can be observed in Fig. 4.4. For instance, `yolov5n_640_fp32` has an input size of 640 × 640 (409'600 pixels) and inference time of 420 ms, whereas `yolov5n_320_fp32` has an input size of 320 × 320 (102'400 pixels) and inference time of 123 ms. In this case a 4× decrease in input size correlates to a 3.42× decrease in inference time. This is expected as convolutional and pooling layers make up the fundamental components of YOLO. The number of required operations to compute these layers scales directly with input data size. Consequently, a larger input takes more time to process.

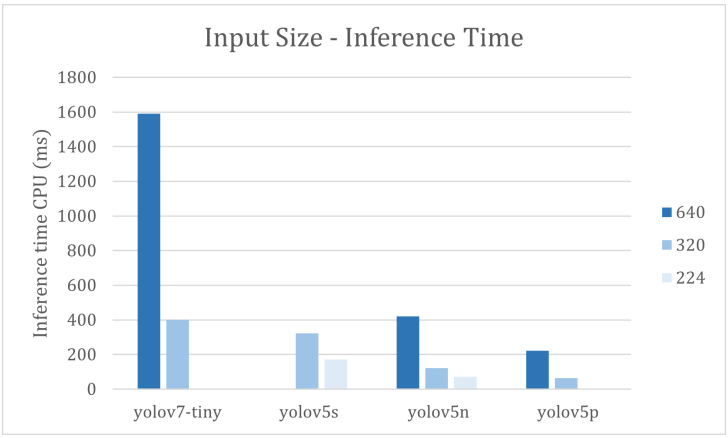


Figure 4.4.: Effect of input size on inference time.

The impact on accuracy is insignificant in comparison to the huge gains in inference speed. This makes input size reduction arguably one of the most viable optimization techniques in the context of this project.

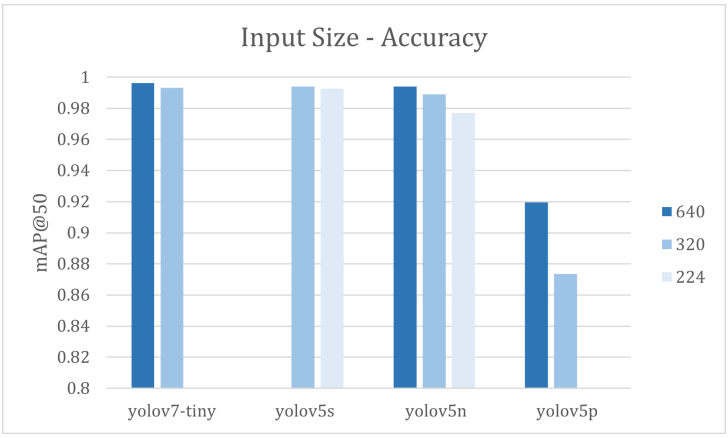


Figure 4.5.: Effect of input size on inference time.

4.6. Quantization Effect

Fig. 4.6 and Fig. 4.7 compare inference time on the CPU and mAP@50 score respectively of between quantized and unquantized models. Quantization yields better performance at the cost of decreased accuracy. However, there are diminishing returns for smaller models and input sizes. For instance, `yolov5s_320` experiences a 47% decrease in inference time when quantized, whereas `yolov5n_224` only sees a 24% decrease.

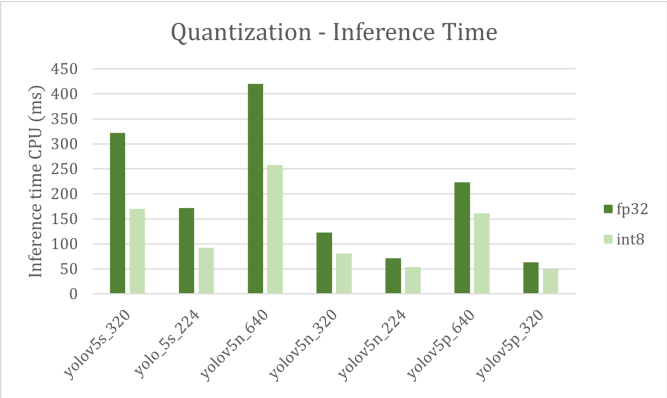


Figure 4.6.: Effect of quantization on inference time.

The reason for this behaviour most likely lies in the cache access patterns. The bigger memory footprint of large models usually results in more data movement between cache and main memory and consequently worse performance. On the other hand, small models such as `yolov5p` may almost fit into the cache at which point quantization hardly affects inference time. Furthermore, the amount of floating point units in the CPU architecture might also play an important role. It is worth noting that inference speed for quantized models could be drastically reduced if an integer-based DSP was used as a hardware accelerator.

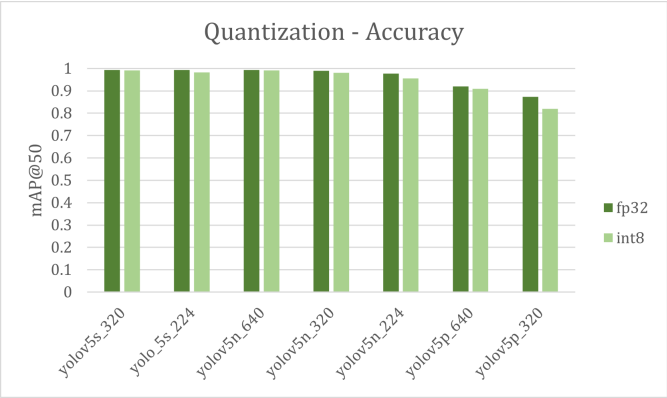


Figure 4.7.: Effect of quantization on accuracy.

4.7. Optimal Configuration

In order to find the optimal trade-off between inference speed and accuracy we plot a Pareto diagram (see Fig. 4.8). Inference time was used as a speed metric on the y-axis. Situated on the x-axis, mAP@50 was chosen as the representative accuracy metric. Each dot represents a specific network configuration. To ensure better visibility, each network architecture is color-coded (yolov7-tiny, yolov5s, yolov5n and yolov5p). Moreover, the degree of saturation correlates to the input size.

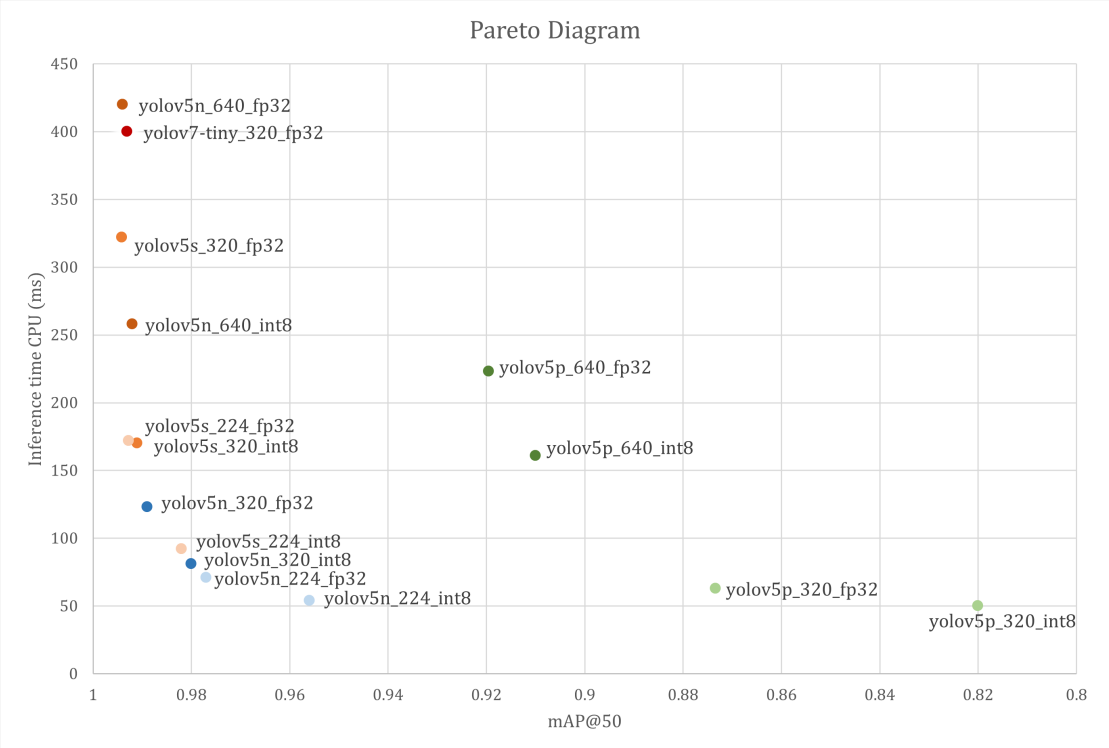


Figure 4.8.: Pareto curve including all network configurations.

Most discoveries discussed in previous subchapters can also be deduced from this Pareto diagram. Another thing that stands out is the fact that `yolov5n_320_fp32` outperforms its equivalent successor `yolov7-tiny_320_fp32` in terms of inference speed. This is due to the different frameworks used for deployment. The Pareto dominant models with a minimal FPS of 10 for real-time hand-detection are `yolov5s_224_int8`, `yolov5n_320_int8`, `yolov5n_224_fp32` and `yolov5n_224_int8`. All things considered, `yolov5n_224_fp32` prevails as the optimal configuration with an inference speed of 71 ms ($\simeq$ 14 FPS) and a mAP@50 score of 0.977.

Chapter 5

Related Work

In a 2022 paper by Ali Raza et al. YOLO was used for for real-time human fall detection and deployed on the edge using a Raspberry Pi 3 together with a Movidius Myriad X hardware accelerator [13]. The work included training of various YOLO versions (YOLOv1 to YOLOv4) on the UR fall detection dataset. The trained models were then converted with OpenVino and deployed using a DepthAI python API which enables acceleration on the Myriad X, a 16-core Visual Processing Unit (VPU) developed by Intel. The implementation pipeline is shown in Fig. 5.1.

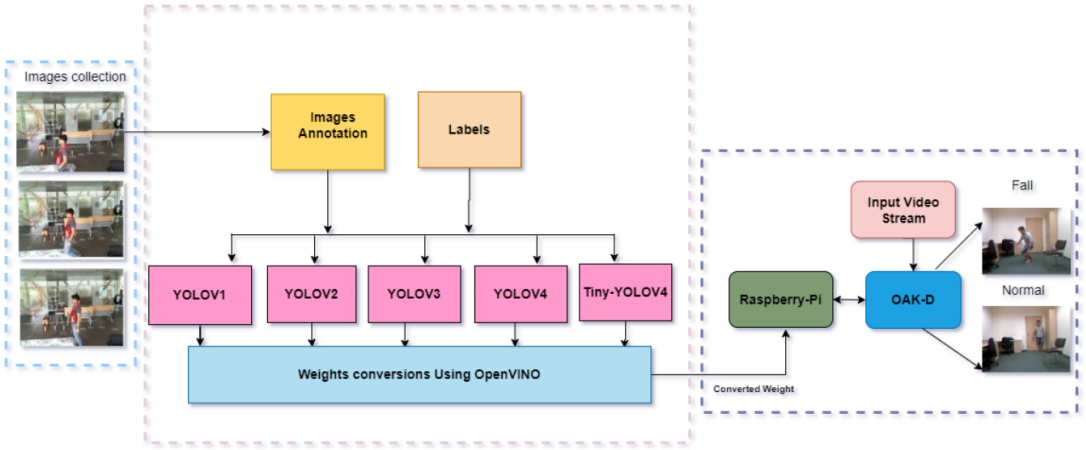


Figure 5.1.: Proposed methodology [13].

The paper concludes that YOLOv4-tiny was the best-performing among all evaluated models with an mAP of 0.95. Furthermore, real-time fall detection has been successfully achieved with an FPS of 26.

Chapter 6

Conclusion and Future Work

This project investigated the feasibility of running YOLO for real-time hand-detection at the edge using a Snapdragon 4100 wearable SoC. For this purpose, numerous YOLO models have been trained, optimized and evaluated. The results conclude that `yolov5_224_fp32` is the optimal model configuration for the target hardware. The network has an mAP@50 score of 0.977 and an inference time of only 67 ms when run on 4 threads of the CPU. This allows for reliable real-time hand-detection which is a principal component for tasks such as sign language detection. The use of the HaGRID dataset significantly contributed to the accuracy of the hand detection system.

Additionally, the thesis has identified certain limitations and challenges that should be acknowledged. NNAPI allows for easy access and utilization of hardware components that can accelerate neural network computations. However, the high-level implementation results in a black box which gives the programmer less control over the targeted accelerators. Other Java API delegates provided by TensorFlow Lite such as the GPU or Hexagon delegate [14][15] suffer from incomplete documentation and are prone to library version conflicts which drastically hinder development. Furthermore, newer but more complex architectures (e.g. YOLOv7) lack TensorFlow Lite support and complicate the conversion process which, most of the time, isn't worth the slight performance gain. Ultimately, having a well-suited and elaborate dataset has the most prominent effect on accuracy.

There are several avenues for future work which could enhance real-time hand-detection using the Snapdragon 4100. Modifying training parameters and experimenting with new network configurations can improve model performance. Furthermore, successful deployment of the Hexagon delegate might enable hardware acceleration using the DSP. Lastly, cloud computing could enable better real-time performance as it leverages the use of more powerful GPUs on a server [16]. In a more sophisticated approach, the current work can be embedded into a complete sign language translation pipeline which requires additional steps such as sign classification and translation.

Appendix A

Task Description

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CENTER FOR PROJECT-BASED LEARNING

Task Description for a Semester Thesis on

Hand-Detection for Sign Language Aid on Android-Based Smart Glasses

at the Department of Information Technology and Electrical Engineering

for

Luca Specht
lspecht@student.ethz.ch

Advisors:	Michele Magno, michele.magno@pbl.ee.ethz.ch Luca Pascarella, luca.pascarella@pbl.ee.ethz.ch Andrea Ronco, andrea.ronco@pbl.ee.ethz.ch
Professor:	Prof. Dr. Luca Benini, lbenini@ethz.ch
Handout Date:	27.03.2023
Due Date:	03.07.2023

Project Goals

Smart glasses have opened up a world of opportunities for augmented reality (AR) applications. The increasing popularity of these devices has led to a surge in the development of AR technology, creating exciting possibilities for enhanced user experiences. The efficiency of computing is essential for AR applications to function seamlessly and deliver the desired level of immersion, making it a critical factor for the success of these devices.

One area that has the potential to benefit significantly from wearables is real-time sign language translation. Smart glasses could be an effective aid to break the existing language barrier between individuals with hearing impairments. Hand detection is a crucial component of sign language translation, and state-of-the-art object detection neural networks are showing promising results for this task.

In the context of your thesis, you will develop a hand-detection application based on state-of-the-art computer vision algorithms. The application will be deployed on the SDW4100 development board, based on Qualcomm's wearable SoC. Finally, you will evaluate your design on the platform.

Tasks

The project will be split into three phases, as described below:

Phase 1 (Week 1-4)

1. Literature Research on hand and object detection [1, 2]. Identification of suitable datasets for the training and evaluation of the neural network.
2. Setup and familiarization with the development environment, both software side (Android Studio) and hardware (SDW4100 platform). Familiarization with the tools to flash and sideload android apps.
3. (Optional) Understanding how to move to WearOS (if possible)

Phase 2 (Week 5-11)

1. Implementation of the ML model offline, using the previously identified dataset(s), and preliminary evaluation for the classification task.
2. Development of the Android/WearOS app backbone (camera interface, controls, layout)
3. Deployment of the network in the Application
4. Optimizations of the inference performance with quantization, pruning, architecture-level techniques, and use of the Hexagon accelerator

Phase 3 (Week 12-14)

1. Polishing of the codebase, documentation
2. Final evaluation of the application in terms of accuracy of the model, inference time, and power consumption with and without optimizations.
3. Report and Presentation

Milestones

The following milestones need to be reached during the thesis:

- Development of the App with appropriate layout, hardware interfaces, and permission management.
- Development of the Machine Learning model and evaluation
- Deployment and integration in the app. Power and latency evaluation
- Final report and presentation.

Project Organization and Grading

During the thesis, students will gain experience in the independent solution of a technical-scientific problem by applying the acquired specialist and social skills.

The grade is based on the following: Student effort; thoroughness and learning curve; achieving qualitative and quantitative results with a scientific approach; supporting practical findings with theoretical background and literature investigations; final presentation and report; documentation and reproducibility. All theses include an oral presentation, a written report and are graded. The report and presentation need to have publication grade quality to achieve a good grade. Students are graded based on the official ITET grading form[1].

For students of IIS (Prof. Benini) a special grading scheme exists, please contact your supervisor for details there. Before starting, the project must be registered in myStudies and all required documents need to be handed in for archiving by PBL.

Laboratory Rules

The students agree to follow the lab rules set by PBL staff, for detail please contact us. The most important points are:

[1] https://ethz.ch/content/dam/ethz/special-interest/itet/department/Studies/Forms/Grading%20Form.xlsx

- All ETH safety regulations need to be followed[2], in addition to ones given by PBL staff
- No device in the lab is used without introduction by your supervisor or PBL staff
- No device leaves the lab without being offically borrowed, this is done by PBL staff and needs your Legi.
- Any damage to devices or tools needs to be reported immediately to PBL staff.
- The Lab-desk is clean and free for others after you finished your task, or when you take longer breaks. All tools are correctly sorted into their drawers/cupboards when you leave

Weekly Report

There will be a weekly report/meeting held between the student and the assistants. The exact time and location of these meetings will be determined within the first week of the project in order to fit the students and the assistants schedule. These meetings will be used to evaluate the status and document the progress of the project (required to be done by the student). Beside these regular meetings, additional meetings can be organized to address urgent issues as well. The weekly report, along with all other relevant documents (source code, datasheets, papers, etc), should be uploaded to a clouding service, such as Polybox and shared with the assistants.

Project Plan

Within the first month of the project, you will be asked to prepare a project plan. This plan should identify the tasks to be performed during the project and sets deadlines for those tasks. The prepared plan will be a topic of discussion of the first week's meeting between you and your assistants. Note that the project plan should be updated constantly depending on the project's status.

Final Report and Paper

PDF copies of the final report written in English are to be turned in. Basic references will be provided by the supervisors by mail and at the meetings during the whole project, but the students are expected to add a considerable amount of their own literature research to the project ("state of the art").

[2] https://ethz.ch/staffnet/en/service/safety-security-health-environment/sicherheit-in-laboren-und-werkstaetten/laborsicherheit.html

Final Presentation

There will be a presentation (15 min presentation and 5 min Q&A for BT/ST and 20 min presentation and 10 min Q&A for MT) at the end of this project in order to present your results to a wider audience. The exact date will be determined towards the end of the work.

References

Will be provided by the supervisors by mail and at the meetings during the whole project.

Place and Date _____ Signature Student _____

Bibliography

[1] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," May 2016, arXiv:1506.02640 [cs]. [Online]. Available: http://arxiv.org/abs/1506.02640

[2] "Sign Language Recognition: A Deep Survey | Elsevier Enhanced Reader." [Online]. Available: https://www.sciencedirect.com/science/article/pii/S095741742030614X

1.

List of Acronyms

AP . . . . . .Average Precision

CNN . . . . . .Convolutional Neural Network

CPU . . . . . .Computing Processing Unit

DSP . . . . . .Digital Signal Processor

FPS . . . . . .Frames per Second

GPU . . . . . .Graphics Processing Unit

GUI . . . . . .Grahpical User Interface

IoU . . . . . .Intersection over Union

JNI . . . . . .Java Native Interface

mAP . . . . . .mean Average Precision

ML . . . . . .Machine Learning

NMS . . . . . .Non-Maximum Suppression

NNAPI . . . . .Neural Network Application Programming Interface

ONNX . . . . .Open Neural Network Exchange

SoC . . . . . .System-on-Chip

YOLO . . . . . .You Only Look Once

List of Figures

List of Tables

Bibliography

[1] K. R. Velasco, "YOLO (You Only Look Once)," accessed June, 2023. [Online]. Available: https://towardsdatascience.com/yolo-you-only-look-once-17f9280a47b0

[2] J. Hui, "Real-time Object Detection with YOLO, YOLOv2 and now YOLOv3," accessed May, 2023. [Online]. Available: https://jonathan-hui.medium.com/real-time-object-detection-with-yolo-yolov2-28b1b93e2088.

[3] S. Du, "Object Detection without Anchors and NMS," accessed June, 2023. [Online]. Available: https://ai.plainenglish.io/object-detection-without-anchors-and-nms-6ca3d56f65ba

[4] D. M. C.-E. Juan R. Terven. (2023, Jun.) A COMPREHENSIVE REVIEW OF YOLO: FROM YOLOV1 AND BEYOND. [Online]. Available: https://arxiv.org/pdf/2304.00501.pdf

[5] S.-H. Tsang, "Brief Review: YOLOv5 for Object Detection," accessed June, 2023. [Online]. Available: https://sh-tsang.medium.com/brief-review-yolov5-for-object-detection-84cc6c6a0e3a

[6] , "SDW4100 Development Kit," accessed April, 2023. [Online]. Available: https://www.thundercomm.com/product/sdw4100-development-kit/#specifications

[7] K. K. Kapitanov Alexander, Makhlyarchuk Andrew. (2022, Jun.) HaGRID — HAnd Gesture Recognition Image Dataset. [Online]. Available: https://arxiv.org/pdf/2206.08219.pdf

[8] , "TensorFlow Lite NNAPI delegate," accessed June, 2023. [Online]. Available: https://www.tensorflow.org/lite/android/delegates/nnapi

[9] , "PYTORCH MOBILE," accessed June, 2023. [Online]. Available: https://pytorch.org/mobile/android/

[10] @PINTO, "Converting PyTorch, ONNX, Caffe, and OpenVINO (NCHW) models to Tensorflow / TensorflowLite (NHWC) in a snap," accessed June, 2023. [Online]. Available: https://qiita.com/PINTO/items/ed06e03eb5c007c2e102

[11] Y. Nitta, "yolov5s_android," github repository. [Online]. Available: https://github.com/lp6m/yolov5s_android

[12] A. F. Gad, "Evaluating Object Detection Models Using Mean Average Precision," accessed June, 2023. [Online]. Available: https://blog.paperspace.com/mean-average-precision/

[13] S. A. V. Ali Raza, Muhammad Haroon Yousaf. (2022, Jun.) Human Fall Detection using YOLO: A Real-Time and AI-on-the-Edge Perspective. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9854070

[14] , "GPU acceleration delegate with Interpreter API," accessed June, 2023. [Online]. Available: https://www.tensorflow.org/lite/android/delegates/gpu

[15] , "TensorFlow Lite Hexagon delegate," accessed June, 2023. [Online]. Available: https://www.tensorflow.org/lite/android/delegates/hexagon

[16] J. K. Haythem Bahri, David Krcmarik. (2020, Jan.) Accurate object detection system on HoloLens using YOLO algorithm. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=9057151

[17] G. Boesch, "YOLOv7: The Most Powerful Object Detection Algorithm," accessed April, 2023. [Online]. Available: https://viso.ai/deep-learning/yolov7-guide/