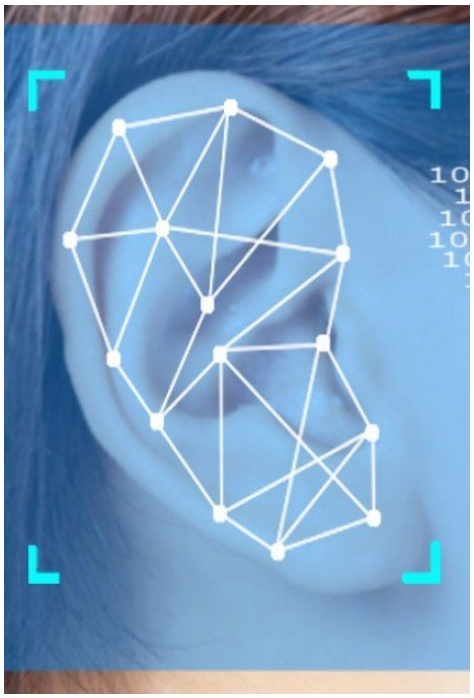


Ear Recognition

BIOMETRIC SYSTEMS

D'Orazio Antonio, 1967788
Minut Robert Adrian, 1942806

Summary

Introduction

Biometric Systems make use of human features for identification purposes. One such feature is the ear, which greatly varies across individuals, it's non-invasive to capture because it only requires a photo, and it doesn't change much after the first 8 years of life and before 70. All these characteristics make the ear very appealing to biometric experts and it's growing in popularity in the recent years.

Our objective is to build a system which performs ear recognition using Neural Networks. The pipeline of the system is:

1. Ear Detection
2. Feature Extraction
3. Matching
4. Decision

We used Python and various libraries to build, train and evaluate our models. To improve the performance and to fairly evaluate the models, we used various datasets which are described in the following sections as they were needed for each part of the system.

Libraries and References

YOLOV5 AND PYTORCH

YOLOv5 is a powerful deep neural network designed for object detection. It comes with pre-trained weights based on the COCO128 dataset, and it supports training on custom datasets starting from these weights.

There are many models available according to the different number of parameters and different sizes of input image. After many tests, we found that the using the Nano model (i.e., the least complex one, YOLOv5n), had almost zero drawbacks in its scoring, while significantly improving the framerate in the mobile app.

The image below pictures a detailed comparison of the many models available.

Model	size (pixels)	mAP^{val} 0.5:0.95	mAP^{val} 0.5	Speed CPU b1 (ms)	Speed V100 b1 (ms)	Speed V100 b32 (ms)	params (M)	FLOPs @640 (B)
YOLOv5n	640	28.4	46.0	**45**	**6.3**	**0.6**	**1.9**	**4.5**
YOLOv5s	640	37.2	56.0	98	6.4	0.9	7.2	16.5
YOLOv5m	640	45.2	63.9	224	8.2	1.7	21.2	49.0
YOLOv5l	640	48.8	67.2	430	10.1	2.7	46.5	109.1
YOLOv5x	640	50.7	68.9	766	12.1	4.8	86.7	205.7
YOLOv5n6	1280	34.0	50.7	153	8.1	2.1	3.2	4.6
YOLOv5s6	1280	44.5	63.0	385	8.2	3.6	12.6	16.8
YOLOv5m6	1280	51.0	69.0	887	11.1	6.8	35.7	50.0
YOLOv5l6	1280	53.6	71.6	1784	15.8	10.5	76.7	111.4
YOLOv5x6	1280	54.7	**72.4**	3136	26.2	19.4	140.7	209.8
+ TTA	1536	**55.4**	72.3	-	-	-	-	-

Source

WANDB.AI

Yolo includes an evaluation framework which was useful to test the model and keep track of how the training was going, and therefore to tune the hyperparameters accordingly.

MOBILENET V3 AND TENSORFLOW

MobileNet is a convolutional neural network that is designed with the limited computational power of mobile phones in mind. There are two available models, Large and Small, but both offer very good performance in our case, since we only perform verification when the ear is detected in the image. Also, we need a higher accuracy in the verification step, so we preferred to use the more complex MobileNetV3-Large model.

For training and finetuning of this model we used the TensorFlow library, which is very intuitive and offers very good performance.

PYEER

For evaluation of the verification model, we needed a library that could calculate and plot the metrics of various models. PyEER offers this functionality through some simple functions that only require a little setup for the input data.

REFERENCES

Training Convolutional Neural Networks with Limited Training Data for Ear Recognition in the Wild

This paper allowed us to build the foundation of our project, since it contains a lot of information about how to use pre-trained models on a new task, by finetuning them on a new dataset. It also describes how important it is to perform data augmentation for delaying overfitting on the training set and the achievable performance improvements.

Handcrafted versus CNN Features for Ear Recognition

Thanks to this paper we were able to get a better idea of the concepts mentioned before and also compare the CNN features to the handcrafted features mentioned during the lectures.

Keras Pipeline

For our training pipeline we used this code as reference but preferred to use the data augmentation tools included in TensorFlow because the imgaug library seemed to not generate augmented images properly.

YOLOv5 Reference

This article was very useful to understand how Yolo outputs its predictions. The code in the article was used as a reference to write the functions which handle the outputs of the model in Java (for the Android app) and in Python.

Ear Detection

DATASET

UBEAR

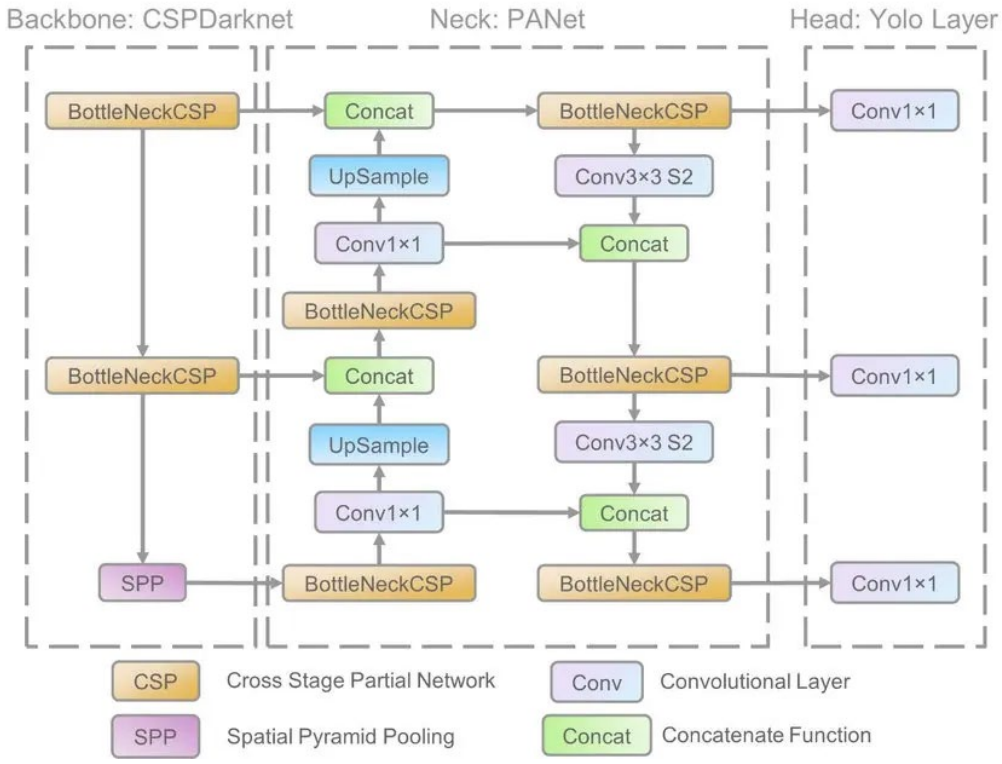
The UBEAR dataset contains more than 4430 images captured in uncontrolled conditions and under-constrained protocols, so that images appear to be captured in real-world conditions. They are equally distributed between genders, and it contains for each subject many photos of the left and right ears.

The images are black and white, and they may or may not present different noise factors:

- Good quality image (i.e., no noise)
- Occlusion: earrings, hair
- Strong occlusion: hair
- Over illumination
- Partially captured ear

MODEL TRAINING

The detection of the ear was implemented using the YOLOv5 neural network, which is designed for object detection in images.

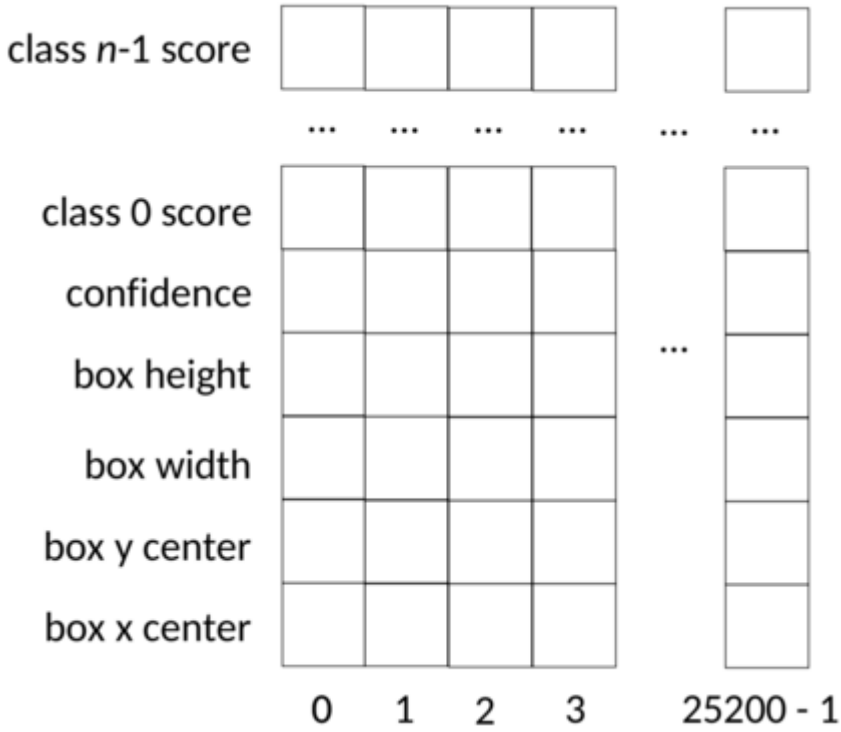


Architecture of YOLOv5

Source.

The detections can be found in the *output* layer.

This layer has 25,200 positions where each one has an array of length 85. Each array contains the data of a single detection, for a total of 25,200 possible detections.



Structure of the output.

Source.

For each detection, the first 4 elements contain the coordinates of the center of the bounding box along with its width and height. The fifth element contains the confidence of the prediction and the 6^{th} to 85^{th} elements contain the score for each class. Given that we only needed two classes, LeftEar and RightEar, we needed to access only from the first to the 7^{th} element.

The confidence for the prediction is a useful measuring for discarding bad detections even if a single class has a very high score versus all the other ones.

The dataset didn't provide any labels, so we had to label the ears ourselves, using the YOLOv5 format.

We used transfer learning from the pre-trained weights (trained on COCO128) and finetuned the model on our dataset and labels to speed up the learning of the model. We separated the labels between left ear and right ear, and we ran the training for 30 epochs on the untouched dataset.

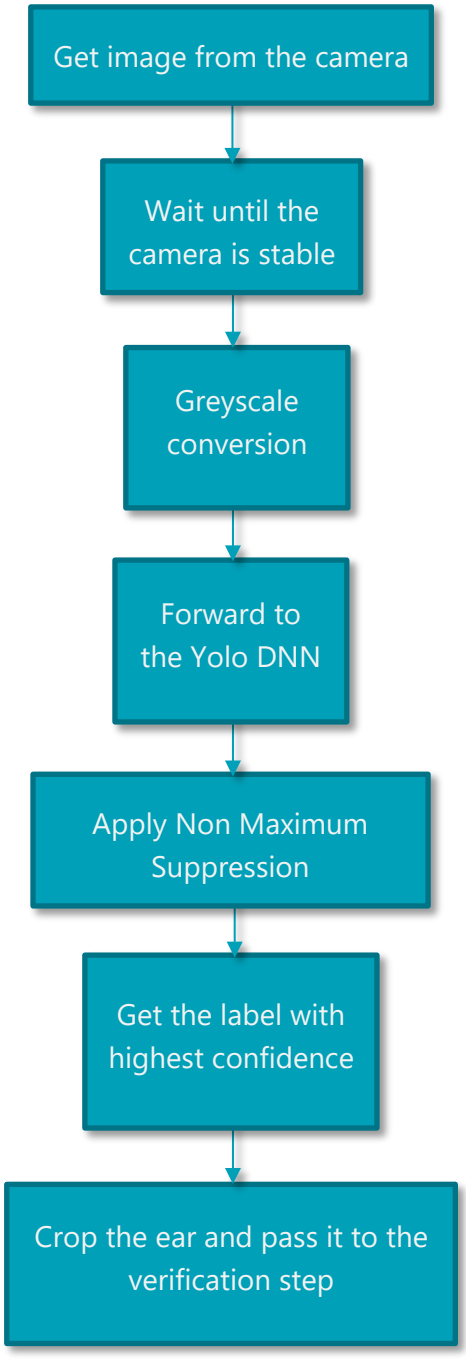
The performance was already good, but since all the images in the dataset had the entire face profile, the network developed a bias which made impossible for it to detect an ear if the face wasn't fully present in the image.

Instead of discarding this training, we decided to implement a function which randomly crops parts of the images leaving the ear in a more neutral background, and we trained other 30 epochs with this new version of the dataset.

We made sure that the test images were never in the train/validation phase, even in this training.

The results were accurate, and the bias was removed.

IMPLEMENTATION



To import the model in Android Studio, we used the Yolo library to export it as an onnx file, since it's supported by OpenCV's DNN module.

The main steps for the implementation are shown in the diagram, with a detailed explanation written below.

- **Wait until the camera is stable:**
  We didn't want the neural network to forward at each frame, because it would make the camera stutter and it may also capture bad frames. To solve this problem, we implemented a simple software motion detector with OpenCV working in the following way:
  - Capture a frame and make it grayscale
  - Threshold its colors to black and white only
  - Get another frame and do the same preprocessing
  - Subtract the images and compute the area of the result
  - If the area is below a threshold, the camera has a stable position.
- **Forward to the Yolo DNN**:
  At this step the app runs the detection on the captured, gray-scaled image frame. If the result isn't empty, the app goes through the next phase.
- **Apply non maximum suppression:**
  The Yolo DNN outputs a raw prediction in a tensor, so we applied the NMS algorithm included in OpenCV to filter out the irrelevant ones.
- **Last steps:**
  If there is at least one detection after the NMS, and if it's above a confidence threshold, the app crops the ear according to the box contour it detected, and finally passes it to the verification phase.

Ear Recognition

DATASETS

AWE

AWE contains 1.000 images of 100 subjects (10 images/subject), which were collected from the web. It's a very challenging dataset because it contains a lot of variation by combining different illumination, poses, genders, ages, and ethnicities.

EarVN

EarVN1.0 contains around 164 Asian people, with 28.412 color images collected by unconstrained environment with different cameras and light conditions. There is a lot of variety in the poses and scale.

This dataset was very helpful, since the big number of images and the data augmentation prevented our model from easily overfitting the training set.

AMI

The AMI ear dataset contains 700 images from 100 subjects (7 images/subject), which were taken in controlled conditions using a single camera, under the same lighting conditions, scale and poses.

Since this dataset contains pictures of ears but also captures part of the side of the face, we used it to evaluate the performance of both the detection model and the recognition ability of the features' extraction model combined. We first cropped the images using the localization predictions and then performed feature extraction on them, the features were then passed to the models performing verification.

FEATURE EXTRACTION

We implemented a Convolutional Neural Network which takes the segmented ear image as input and outputs a vector with 1280 components.

To simplify training and not spend too much time on the design of the architecture we decided to use a pre-trained model named MobileNet V3, which seemed the best choice for an Android app that could run on devices with fairly limited computing power. So, we removed the fully connected layer at the top and added one for our specific dataset. After that, we followed a pretty common approach by first freezing the base layer, which was already trained on ImageNet, and then after training the fully connected layer (keeping track of the cross-entropy loss and accuracy on a validation set), we proceeded to train the whole model with a small learning rate until convergence.

We then removed the classification part of the model and used it as a feature extractor to perform evaluation on both AWE and the AMI dataset, which was cropped using the detection model.

The architecture of the model can be described as follows:

Layer (type)	Output Shape	Param #
input_4 (InputLayer)	[(None, 224, 224, 3)]	0
MobilenetV3large (Functional)	(None, 1, 1, 1280)	4226432
flatten_1 (Flatten)	(None, 1280)	0
dropout_1 (Dropout)	(None, 1280)	0
dense_1 (Dense)	(None, 80)	102480

The first model was trained using the AWE dataset, but the scores weren't good enough, so we implemented data augmentation as described in the papers we used as reference, but the results were still far from our expectations. We also trained a second model using the EarVN dataset, but we observed that the scores on the AMI dataset were lower than expected (since we are considering a controlled environment), and we guessed that the issue could be that the EarVN dataset contains low resolution images, while in AMI the resolution was much higher. That's why we trained a third model by using a part of the cropped AMI dataset (497 images) to finetune the already previously trained model (on EarVN), and then re-tested again the models on the remaining images.

We obtained higher scores on AMI while still keeping good scores on the AWE dataset.

For the training of the models, we used a few utility functions which are called after each epoch:

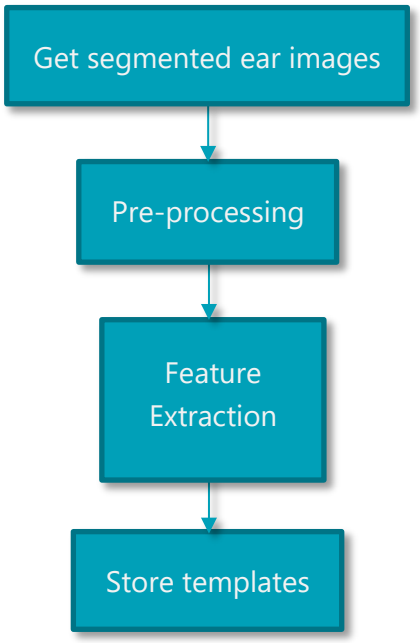
- Reduce learning rate when the validation loss doesn't improve
- Checkpoint the model when the loss improves
- Stop training if the validation loss doesn't improve after a few epochs

We also evaluated the scores obtained by testing the two most common similarity measures, correlation and cosine similarity, and chose correlation since it gave the best results.

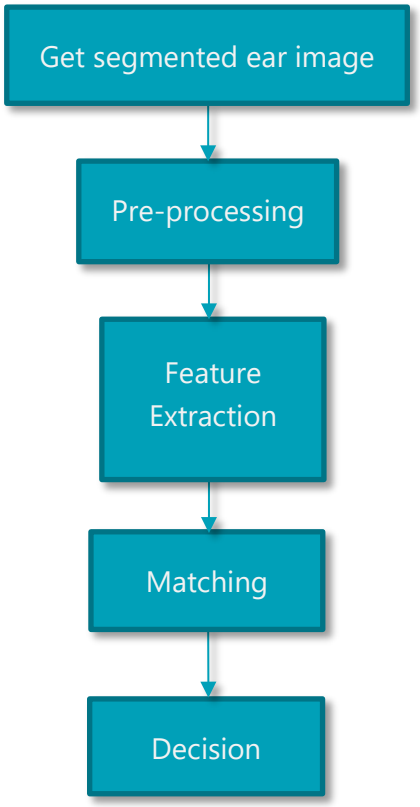
After building our model we added it to our Android app by using the TensorFlow Lite library. We wrote some code to:

- Pre-process the inputs
- Extract the features using the TensorFlow Lite model
- Save the templates
- Retrieve the templates at recognition time
- Perform matching using a similarity measure between the probe and the saved templates
- Considering the maximum similarity decide whether the identity is the same or not

Enrollment flow



Verification flow

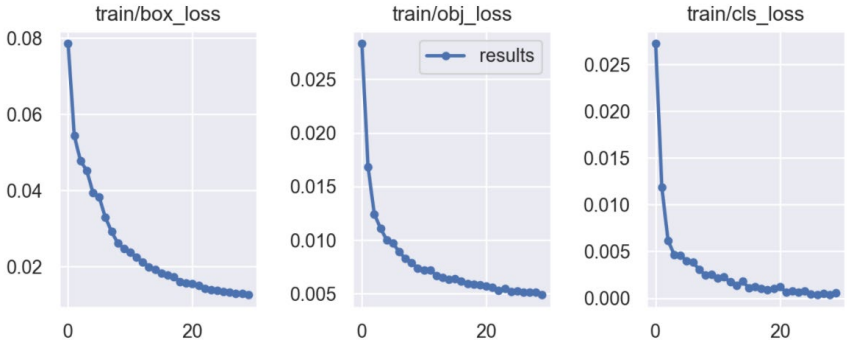


Evaluation

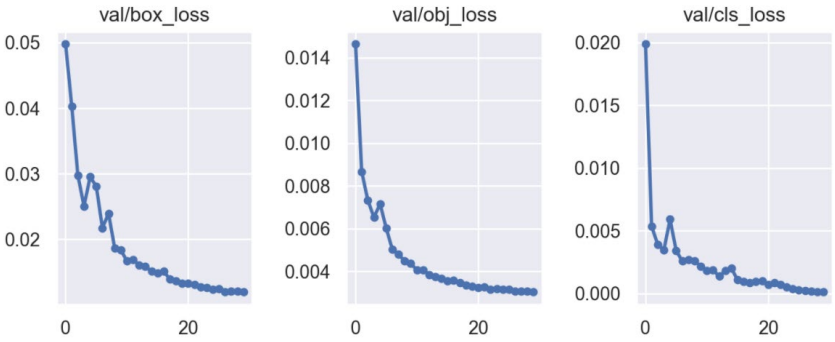
EAR DETECTION

Type of weights: Yolov5-nano

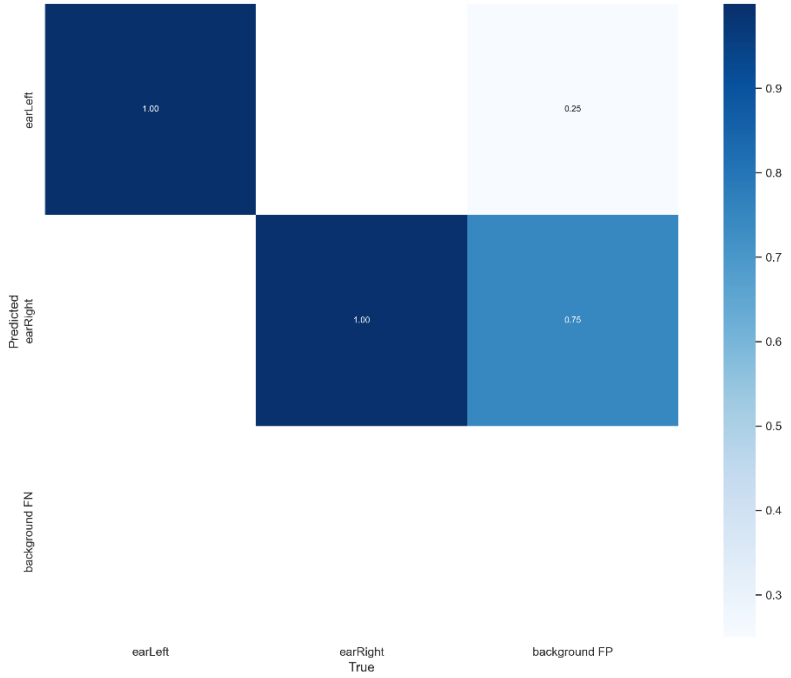
Epochs 0-30



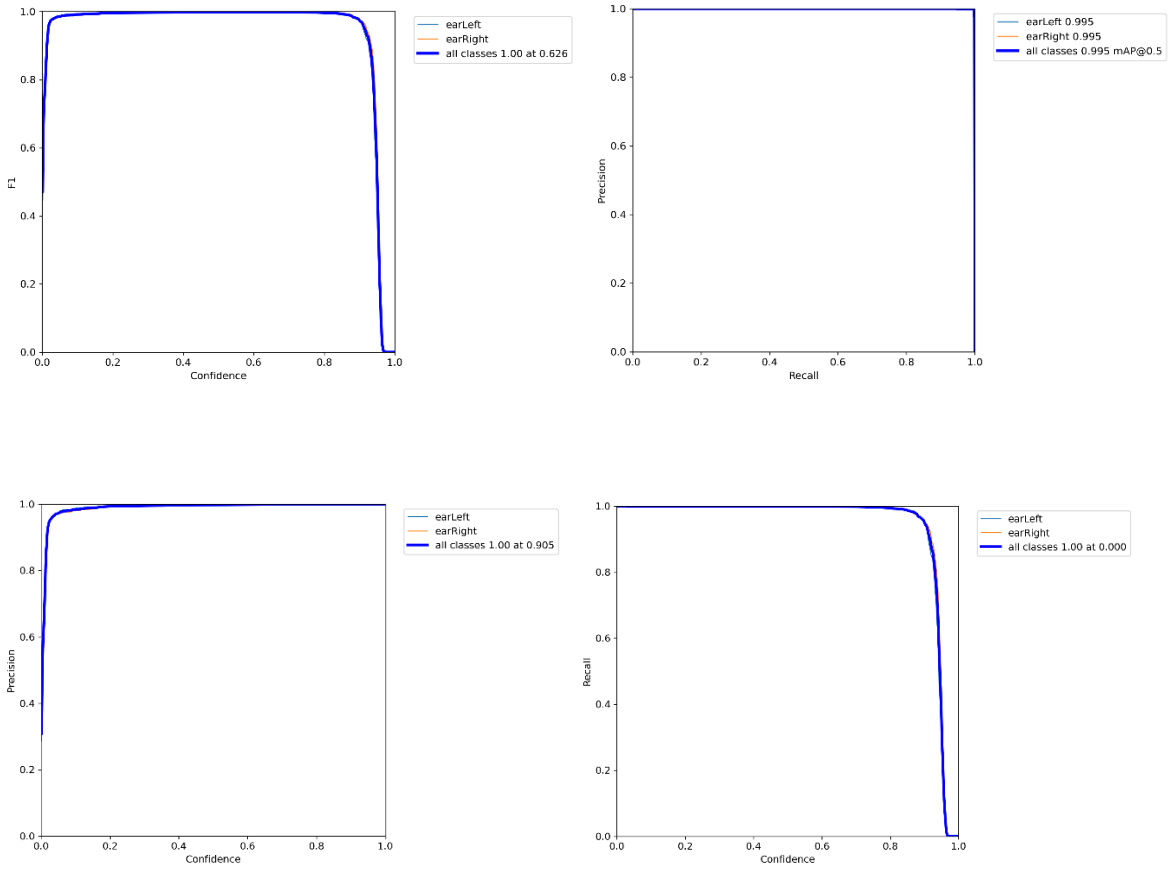
Box, object, and class loss at training time



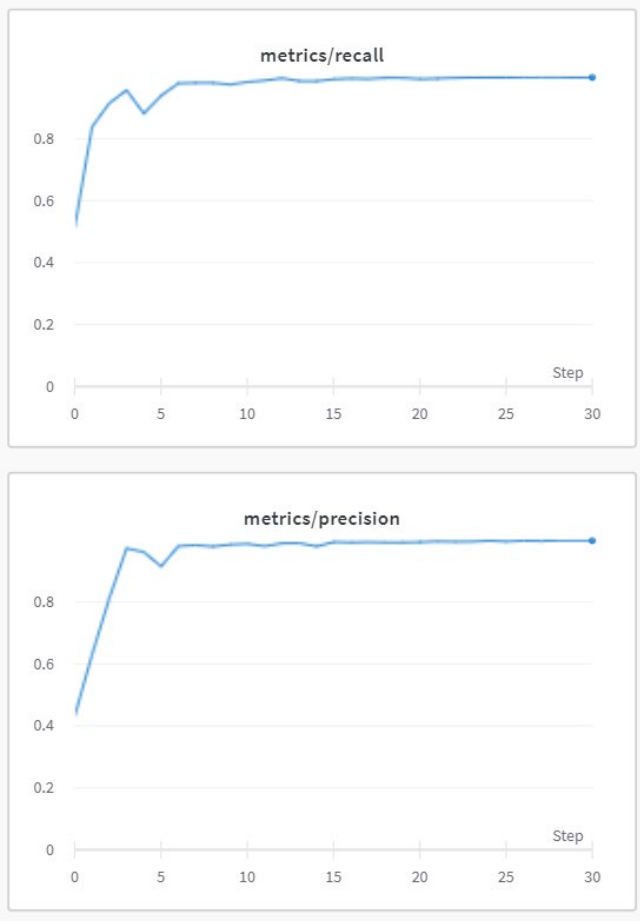
Box, object, and class loss at validation time



Final confusion matrix

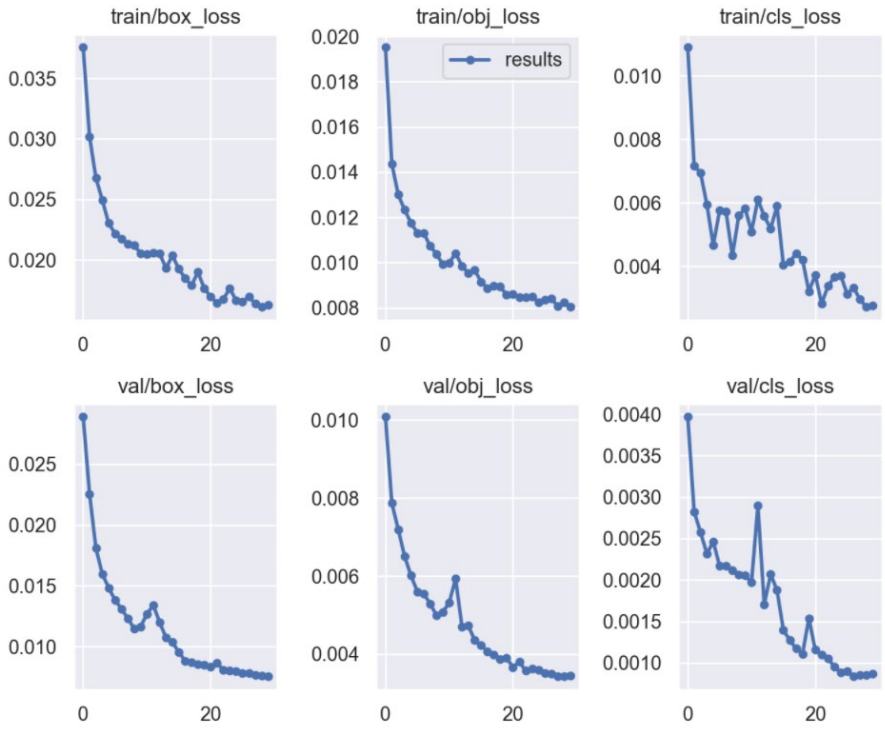


F1, ROC, Precision/Confidence, Recall/Confidence curves.



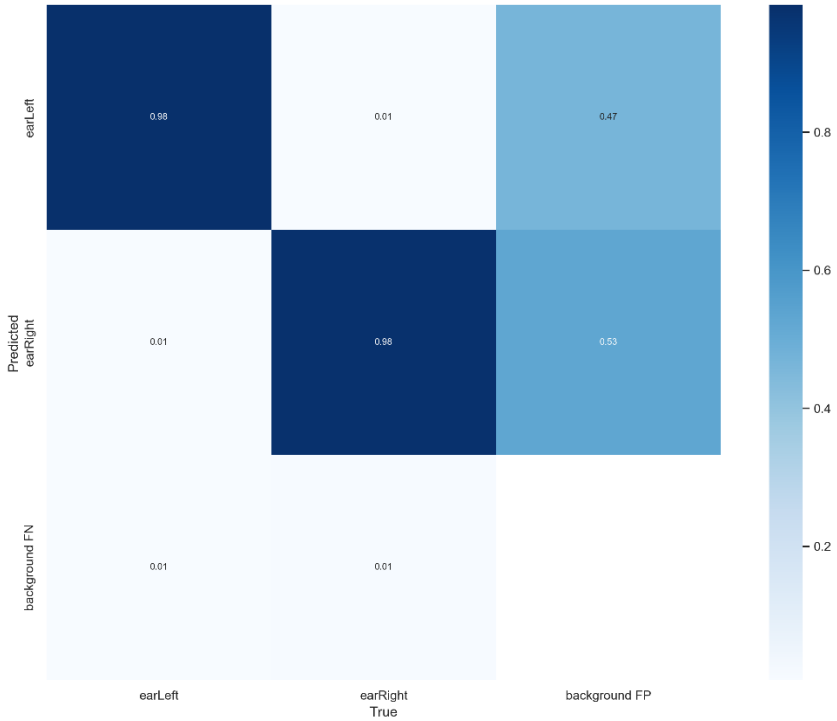
Precision and recall with respect to the epochs.

Epochs 31-60 – Random crops on the dataset

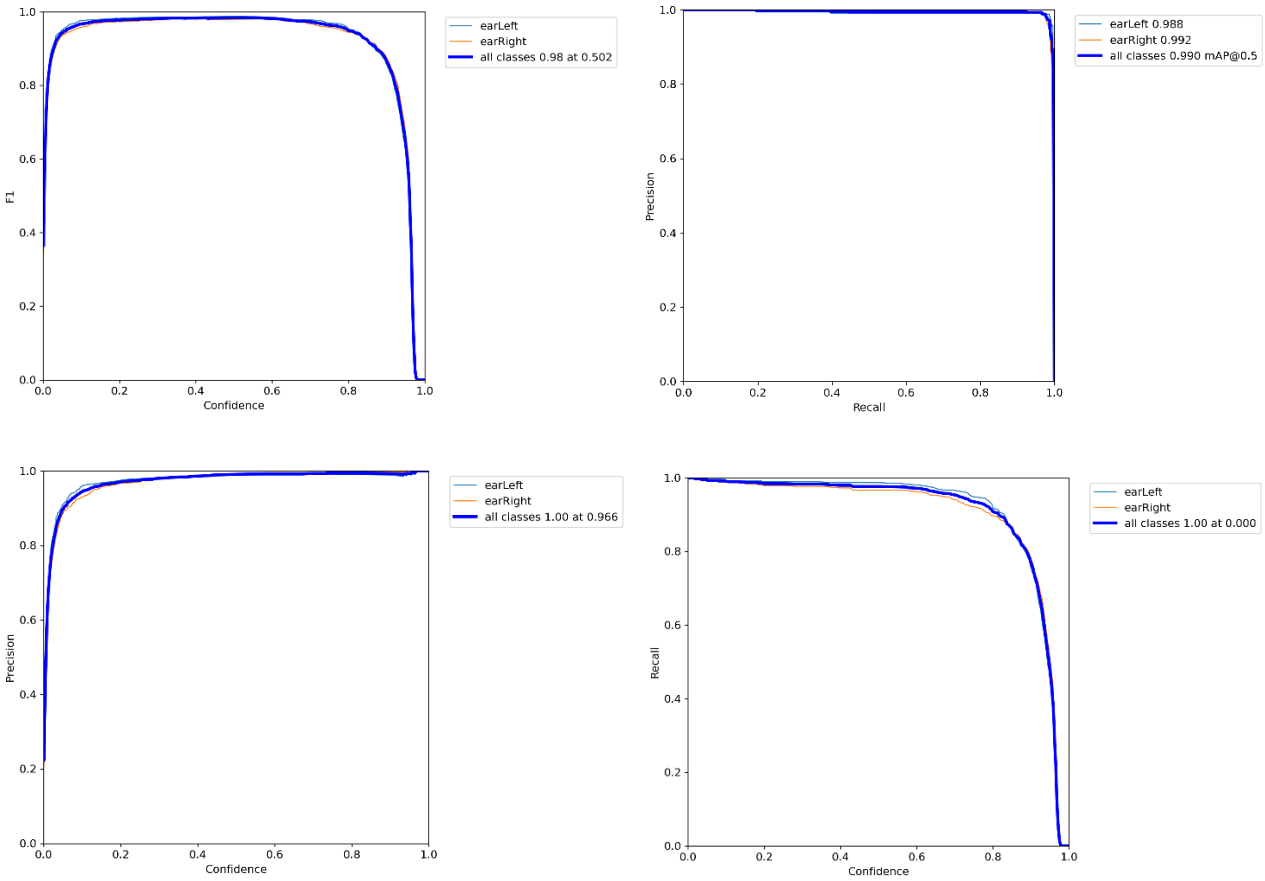


Box, object, and class loss at training time

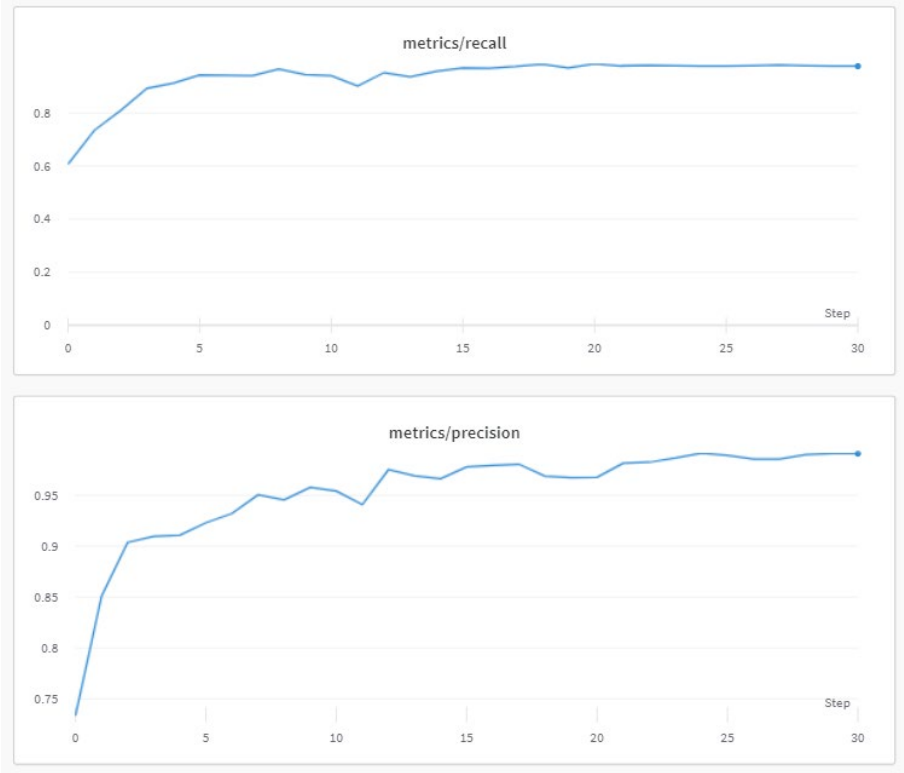
Box, object, and class loss at validation time



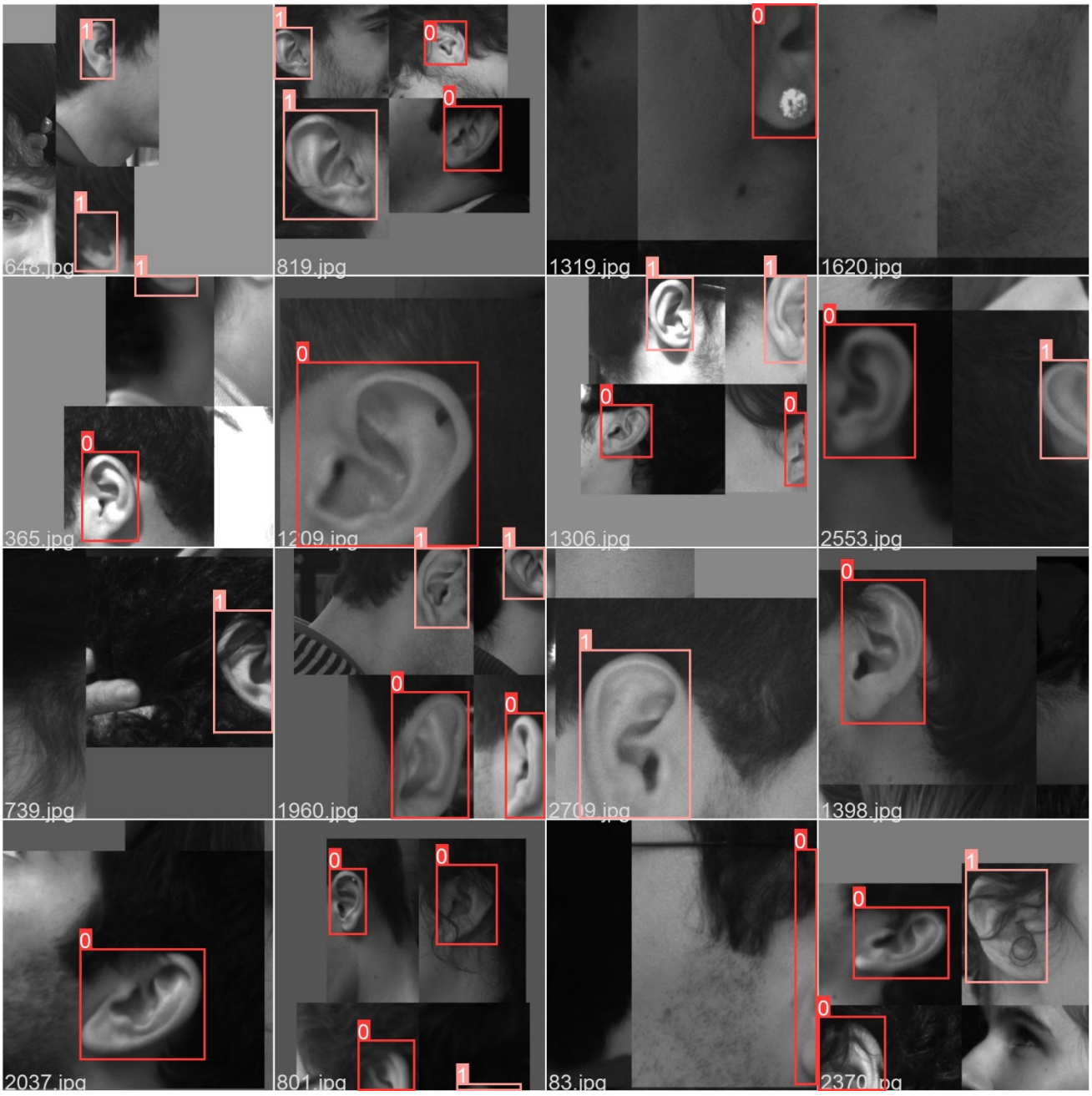
Final confusion matrix



F1, ROC, Precision/Confidence, Recall/Confidence curves.



Precision and recall with respect to the epochs



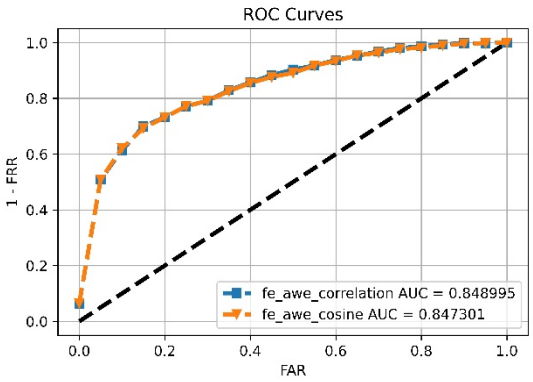
Examples of cropped batch

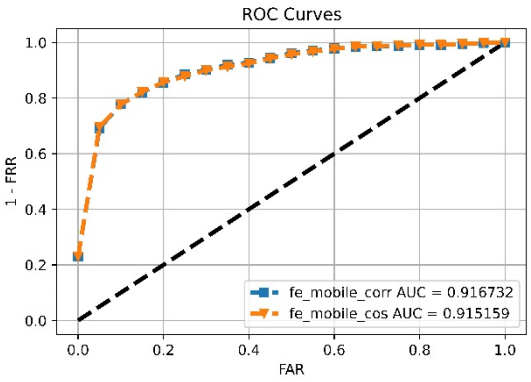
EAR RECOGNITION

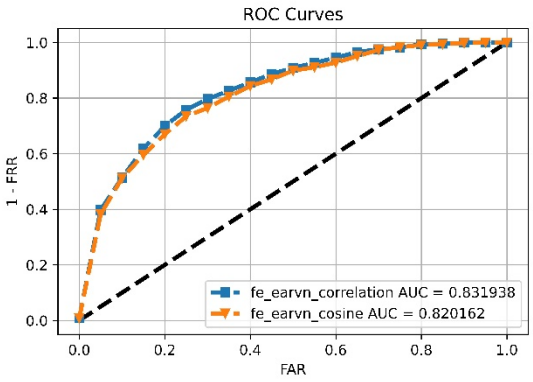
Similarity Measure

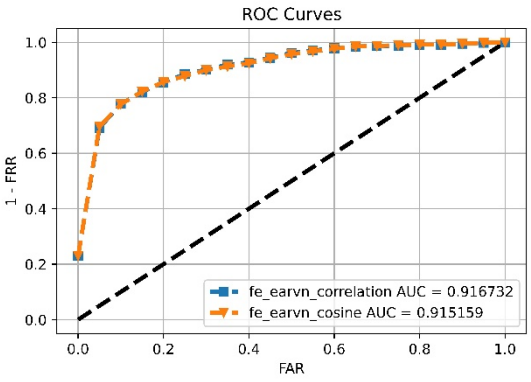
We tested the two most common similarity measures (correlation and cosine similarity) and compared them using the ROC and DET curves. For testing we used two models, one trained on AWE and the other on EarVN. By observing the results, we chose correlation because it consistently performed better on the tests performed on the two datasets.

The first column are the results from the two models using the AWE testing set, and the second column are ones using the cropped AMI testing set.









Models' Comparison

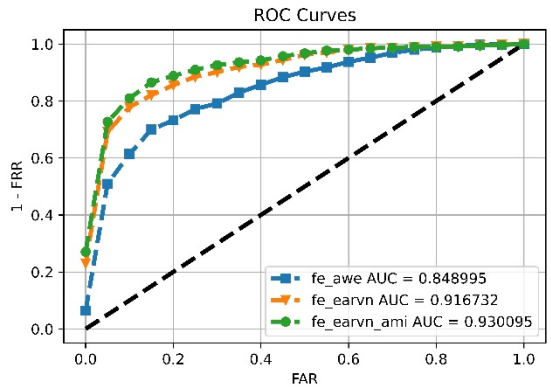
Here we present the performance of three models trained and finetuned on different datasets.

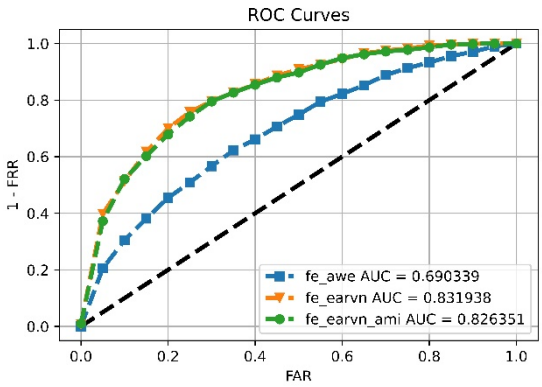
The first model was trained and finetuned on AWE, but the results weren't as good as expected because, as already mentioned, it was overfitting the training set very rapidly, and data augmentation didn't help very much.

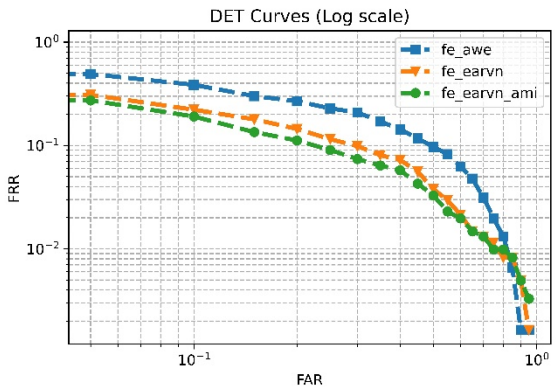
The second model was trained and finetuned on EarVN, which gave better results but didn't perform as expected on the cropped AMI dataset.

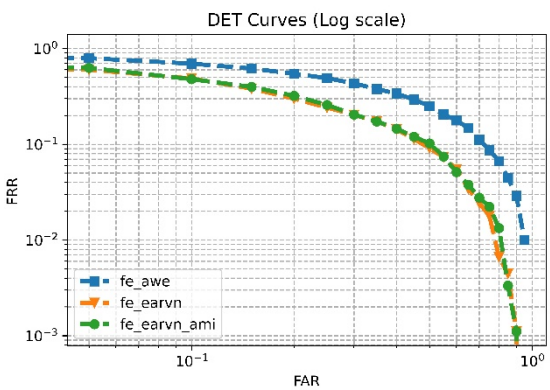
The third model was trained and finetuned on both EarVN and part of the cropped AMI dataset.

First, we take a look at the ROC and DET curves. On the left column we have the results from the cropped AMI dataset, and on the right we have the results from the AWE dataset.

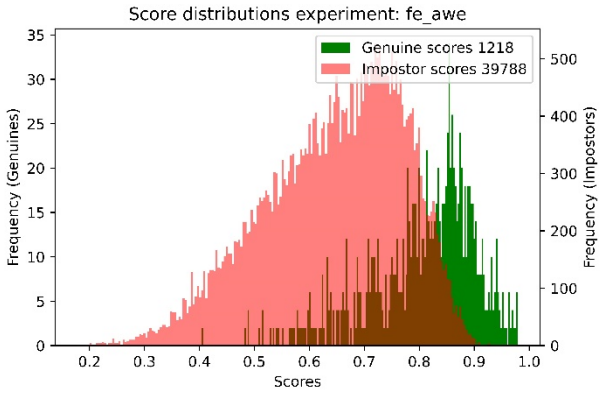


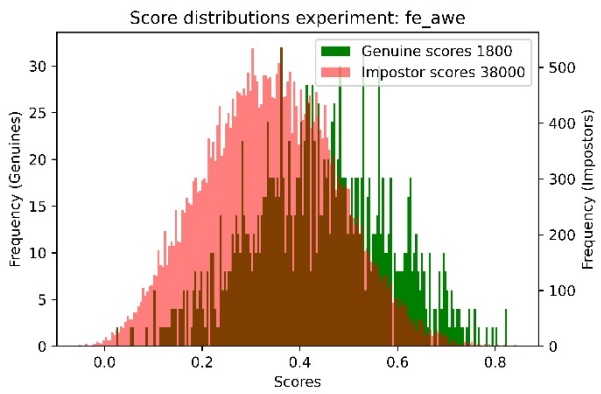


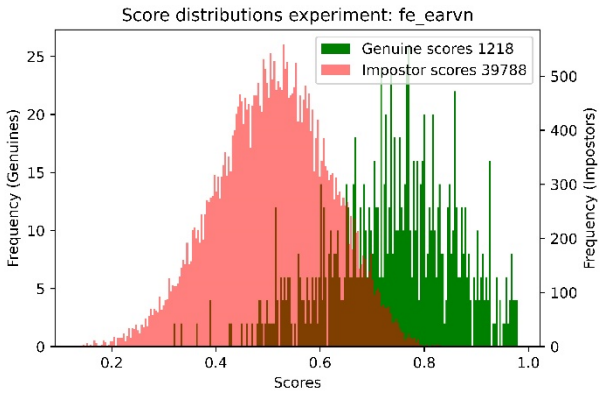


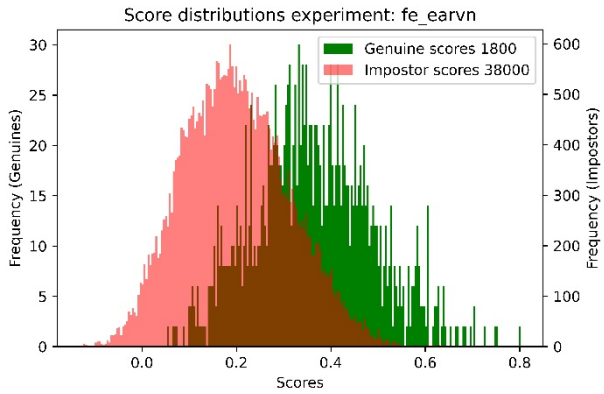


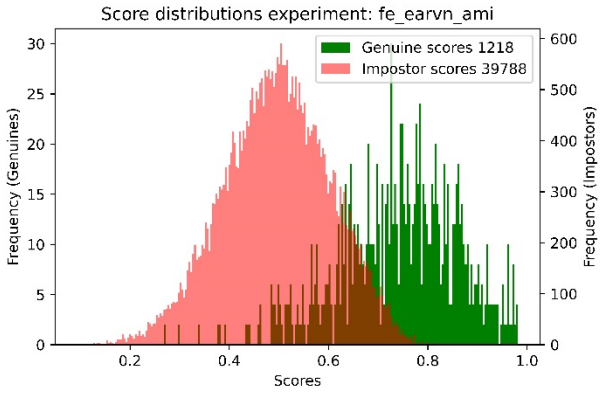
Then, we can see how the imposter and genuine scores are distributed, the columns still represent the same datasets.

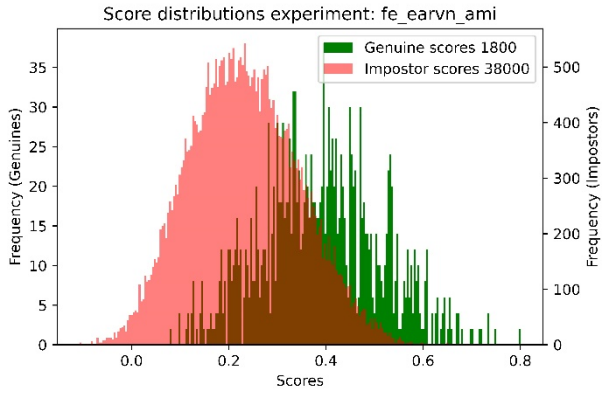












Final thoughts

For the detection part, we can see that YoloV5 is a very powerful neural network even when trained for a low number of epochs, and even on its tiniest implementation. We decided to use the nano version after many 'trial and error' trainings, going down from the medium-sized, which was too heavy for a phone to be handled without stuttering.

From the evaluation, we can see that training only on the untouched UBEAR dataset led to overfitting issues. That was hard to see on the charts because the test images had the same bias as the train images, but the perfect 100% score after too little training was already suspicious. That was also experienced from the app since it could detect the ear only if the whole head profile was present in the picture.

After other 30 epochs with random crops on the dataset, it learned to better generalize, and the real-world results were near perfect this time, it was able to capture the ear at different distances.

We can also see that the model which performs better overall is the one that uses the EarVN dataset and part of AMI, but introducing some samples from AMI didn't influence much the performance. This is why we used this model in our Android app, but the real performance we expected wasn't reached by our system. We set a pretty high threshold (0.98) to improve security as much as possible, but it still wasn't reliable enough for use in a real-world application. Our objective though was to build an entire biometric system and implement it for a mobile device, and for this reason it couldn't easily reach the higher performance of the current state of the art models.

Talking about the performance of the app, adding the detection for the motion led to a nice optimization, because it heavily reduced both lag issues and blurred captures. We are satisfied about the experience of the app because it requires the user only to tap on the option and to put the device in front of the ear, simulating the behavior of a "Face ID" unlock but for the ear.

Starting from pre-trained models was a great choice, even though it wasn't as easy to implement as initially thought, since it still required great attention on data augmentation and finetuning of the models. We are overall satisfied with the system we were able to build, and the experience acquired during its design and implementation.