JOHANNES KEPLER
UNIVERSITY LINZ

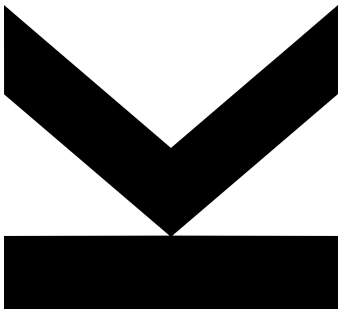
Author / Submitted by
Katharina Sternbauer
k11812499

Submitted at
Institute of Computational Perception

Supervisor and First Examiner
Josef Scharinger, a.Univ.-Prof, Dr.

June 23, 2023

Ear spoof detection by Convolutional Neural Networks



Implementation, improvements and evaluation of a proposed CNN for distinguishing between real and spoof images of ear biometrics.

JOHANNES KEPLER UNIVERSITY LINZ
Altenbergerstraße 69
4040 Linz, Austria
www.jku.at
DVR 0093696

Kurzfassung

Einzigartige biometrische Merkmale von Menschen, wie die Ohren, Finger, Handfläche, Stimme oder das Gesicht, sind bereits weit verbreitet in der Authentifizierung von Personen. Eines dieser Merkmale, welche seitens eine:r Benutzer:in kaum Interaktion erfordert, über die Zeit stabil bleibt, von Mensch zu Mensch eine große Unterschiedlichkeit aufweist sowie reich an Datenpunkten ist, ist das Ohr. Biometrische Systeme müssen nicht nur sicherstellen, dass ein präsentiertes Ohr auch wirklich zu der erwarteten Person gehört, sondern gleichzeitig auch, dass es sich um keine Replika/Fälschung handelt. Für Letzteres kommen einfache Ausdrucke infrage, jedoch auch 3D-Masken oder gar kosmetische Operationen. Um zwischen echten und gefälschten Merkmalen unterscheiden zu können, wurden in den letzten zehn Jahren Methoden des maschinellen Lernens populär. Neben konventionellen Ansätzen wie Support Vector Maines (SVM) kommen auch tiefe neuronale Netze, etwa Convolutional Neural Networks (CNNs) zum Einsatz. Der in dieser Arbeit besprochene Ansatz nutzt ein CNN zur Unterscheidung zwischen echten und mittels Druck gefälschten Bildern eines Ohres. Dafür wurden über 2000 Bilder händisch gefälscht. Die Arbeit beginnt mit einem Überblick zu CNNs und den verwendeten Datensätzen und legt folgend den Fokus auf die Implementierung. Zudem werden die gewonnenen Ergebnisse besprochen, am Ende folgt eine Zusammenfassung sowie ein Ausblick auf zukünftige Forschung und weitere Anwendungsmöglichkeiten.

Abstract

Unique biometric features on humans, such as the ear, fingerprint, palmprint, speech or face, are widely used to authenticate users against identification and verification systems. One feature which requires hardly any interaction by the user, is relatively stable over time, different between individual humans and rich in features to tell them apart, is the human ear. Biometric identification / authentication systems need not only to verify that the presented ear is actually from an anticipated user, such a system also needs to confirm that the provided feature is really from said human and not a fake replica (spoof). The latter involve, among other things, simple fotografs, 3D printed masks and, in the most extreme cases, plastic surgery. To distinguish between live and spoof biometrics, Machine Learning approaches have become increasingly popular in the last ten years. These range from "conventional" methodologies like Support Vector Machines (SVM) to deep neural networks like Convolutional Neural Networks (CNNs). The following work shows one such approach utilizing a CNN to detect whether a presented image is genuine, therefore a live capture, or spoofed via a printed photo attack, simulated by generating over 2000 spoofed images by hand. It starts with an overview of CNNs and the used datasets, then puts the focus on the implementation. Furthermore, this work discusses the gathered results and in the end gives a conclusion in addition to an outlook on further possible work and research.

Contents

List of Figures

List of Tables

List of Abbreviations

AMI Mathematical Analysis of Images

BN Batch Normalization

CNN Convolutional Neural Network

FFR False Fake Rate $\widehat{=}$ FNR

FGR False Genuine Rate $\widehat{=}$ FPR

FNR False Negative Rate

FPR False Positive Rate

HTER Half Total Error Rate

IITD Indian Institute of Technology Delhi

ReLU Rectified Linear Unit

USTB University of Science and Technology Beijing

1 Introduction

1.1 Motivation

Each human being has biometric features that are unique among all other 9 billion people living on earth or floating in a space station above. However, said features are vulnerable to attacks. These include fingerprints with their patterns of shallow and raised ridges, which may be faked by using doubles made of silicone or gel. Systems relying on the unique heights and depths of human faces can be tricked by simple presentation attacks, or more sophisticated 3D masks [1]. Palmprints suffer from the same problem with presentation attacks, where a printed image of a hand is enough to trick an authentication system into generating a false-positive response [2]. Even the main topic of this project, ear biometrics, suffer from the same problems with both simpler presentation attacks and more sophisticated 3D replicas, plastic surgery. [3]. If this topic is taken one step further, the whole system in use to authenticate users is vulnerable to attacks like sensor tampering, database manipulation or attacks on communication channels between individual parts [4]. This, however, is out of the scope for machine learning in spoof detection and the assumption is made that the connections and systems are not being hijacked and manipulated.

1.2 History of machine learning in spoof detection

To counter the threat posed by newer commercially available technologies such as the various forms of 3D-printing or simpler, household items like Play-Dog or wood glue, countermeasures have been developed and implemented into a sizeable number of systems [5]. Earlier methods, before the current strong area of machine learning, focussed on semi-automated, genetic algorithms and holistic approaches [6]. Since the late 2000s until

the early 2010s, conventional machine learning starting their victory streak in terms of performance and accuracy over what came before. Furthermore, the first partial fingerprint matching algorithms were published during that period of computer science history. From 2012 and onwards, deep learning neural networks started achieving significantly improved accuracy results, up in the area of 97 to 99 percent already in 2018, however not on a standardized dataset which would enable competitions and fair comparisons [7]. This still ongoing area is also the first time liveness detection techniques were considered for evaluation and gained importance with the now renowned LiveDet datasets (2009, 2011, 2013, 2015) [6].

Ears, the main focus of this work, have, like many other biometric features used for identification and verification, unique features between human beings. These have only recently been used in biometrics, but provide many benefits such as being time invariant, reliable and feature rich [8]. Furthermore, they most often do not need any direct cooperation from or contact with the user, which is a sharp contrast to other methods like voice or fingerprint, where a human must actively interact [9]. Imren Toprak and Önsen Toygar focus their work, which serves as the basis for this project, on image presentation attacks and the detection of such via the usage of fusion techniques [3]. They combined feature, score and decision level features together with a Convolutional Neural Network, where the result of both systems is merged through the help of an OR-gate. The neural network is trained on publicly available datasets, namely AMI, UBEAR, IITD and USTB, and achieves desirable results with Half Total Error Rates (HTER) down to 0.5 percent. This is why it was chosen to re-implement the given CNN for this project and test the results whether they hold to what is claimed.

1.3 Research topics

This project splits its main focus on various points, all centered around the implementation and testing of a CNN. First, it aims to answer on how to acquire or generate spoof data for biometrics, in this case presentation attack images of ears. Second, the concrete way of implementing a network in PyTorch [1], a Python-based toolbox for machine / deep learning, developed my Meta. Further on, hyperparameter search with RayTune [2] is utilized, and

[1] `https://pytorch.org/`
[2] `https://docs.ray.io/en/latest/tune/index.html`

the results mirrored back into the evaluation and testing phase of the developed CNN. Data loading, preparation and augmentation is another topic. Finally, this work will also show one way among many on how to avoid overfitting via an early stop during the training-validation cycle.

1.4 Structure

The first part discusses Convolutional Neural Networks, with a focus on the implementation chosen by Imren Toprak and Önsen Toygar. The implementation goes into detail on the different tools provided by PyTorch and on how to evaluate the results. The datasets are presented as well, along with the spoof generation. Further on, a conclusion is given, alongside an outlook on further research topics.

- Chapter 2 discusses the theoretical concepts behind Convolutional Neural Networks, the parts of which they consist, followed by a quick overview of the chosen evaluation measurements.
- Chapter 3 shows the used dataset in the base paper, discusses which sets were obtainable and how fake image data was generated and what data augmentation is.
- Chapter 4 dives into the implementation details and the parts of PyTorch and RayTune which were picked. It features the most relevant code parts for generating a CNN, loading data, training, evaluating and storing a model, as well as the hyperparameter search. It will not show every detail, as this would break the perimeter of this work without any doubts, but provide a link to the source code.
- Chapter 5 explains the results of the hyperparameter search, model verification during training and an eventual early training stop. The highlight of this section however are the results achieved on the trained model with the test dataset and the takeaways from these numbers.
- Chapter 6 will sum up the content of this project, give an overview of the measured results, followed by a short discussion on further topics which have to be looked into more deeply or are of interest in biometric identification.

2 Methodology

This chapter covers the basics of Convolutional Neural Networks (CNNs) and the measurement metrics utilized to analyse the effectiveness of the trained model. First, it will discuss the basic building blocks of CNNs, then show the example architecture used for this project and explain it. Second, different metrics are presented that will help to discuss and evaluate the effectiveness of the implemented network.

2.1 Convolutional Neural Network

2.1.1 General

Convolutional Neural Networks are known for their power in image classification tasks due to their network structure with multiple convolutional layers [10]. They first showed their power in the 2012 ImageNet competition, where a CNN (AlexNet) outperformed other methods and showed unprecedented precision in its classification ability [11]. This skill is very well useable for spoof detection, the task at hand. The decision is between one of two classes, real and fake images.

Quite generally speaking, each Neural Network is built up by a system of neurons, placed in all layers (input, hidden, output) and interconnected with each other in very specific ways for communication purposes. Every neuron has, in addition, weights attached to it. A neuron in the input layer takes the 'signal' from the outside environment (i.e. whatever is presented as an input), apply their weights to the values, takes the sum of the products and outputs it to the next layer. From there on, other neurons take this signal as an input, apply their weights and pass the result on to the next layer, until the output layer is reached [12].

2.1.2 Building blocks

A typical convolutional neural network is built of three main parts: The convolutional layer, the pooling (i.e. subsampling) layer, and the fully connected layer [3]. Each of these may and will be used multiple times to form a CNN. A typical architecture can be seen in image 2.1.

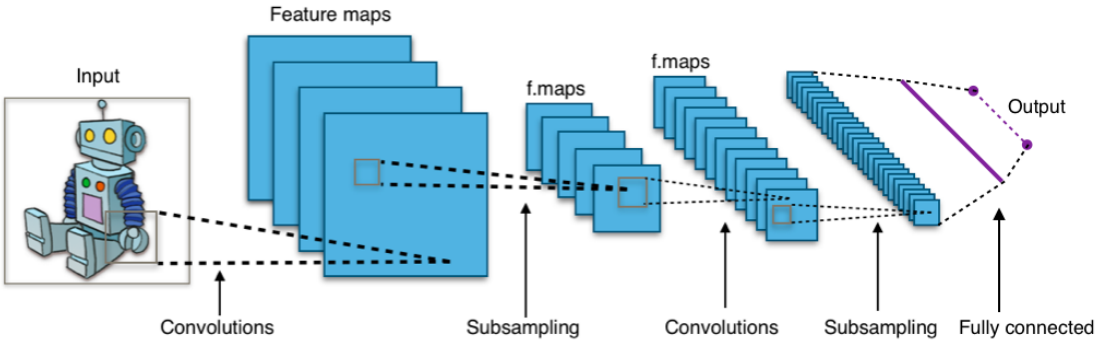


Figure 2.1: A typical CNN architecture [1]

Convolutional Layers extract features by using filters and applying them with the convolution operation to the input. For this, the filter (called 'kernel') is moved element by element over the entire input, so the output forms a feature map. A kernel has a certain size (example: 3x3x6), where the first two elements denote the 2D size of the kernel and the last one has to match the number of input feature maps. For an RGB image, there are 3 feature maps (red, green, blue), so any filter operating on them has to be of size 'LxHx3'. To obtain more output feature maps, simply applying additional filters is sufficient.

Before sending the output to the next layer, a further item called an activation function is applied to the values. These are non-linear functions. Most typical nowadays is 'Rectified Linear Unit' (ReLU), which offers considerable performance advantages in training time compared to other choices like $\tan(x)$, where x is the output map value [13]. The working principle of ReLU are rather simple, as it will let any value through that is greater than zero, and sets everything else to zero. Therefore, no negative values will be passed to the next layer. The mathematical definition typically looks as followed:

[1] https://en.wikipedia.org/wiki/Convolutional_neural_network#/media/File:Typical_cnn.png

$$f(x) = \begin{cases} x & \text{x} > 0 \\ 0 & \text{x} \leq 0 \end{cases} = max(x, 0)$$

Pooling Layers reduces the size of data and aims to select only the most important features [14]. It performs subsampling on the feature map and aims with this to reduce complexity. The most popular choice is Max Pooling, which selects the highest value in each section of the feature map of a certain size (example: 2x2). An example is given in image 2.2.

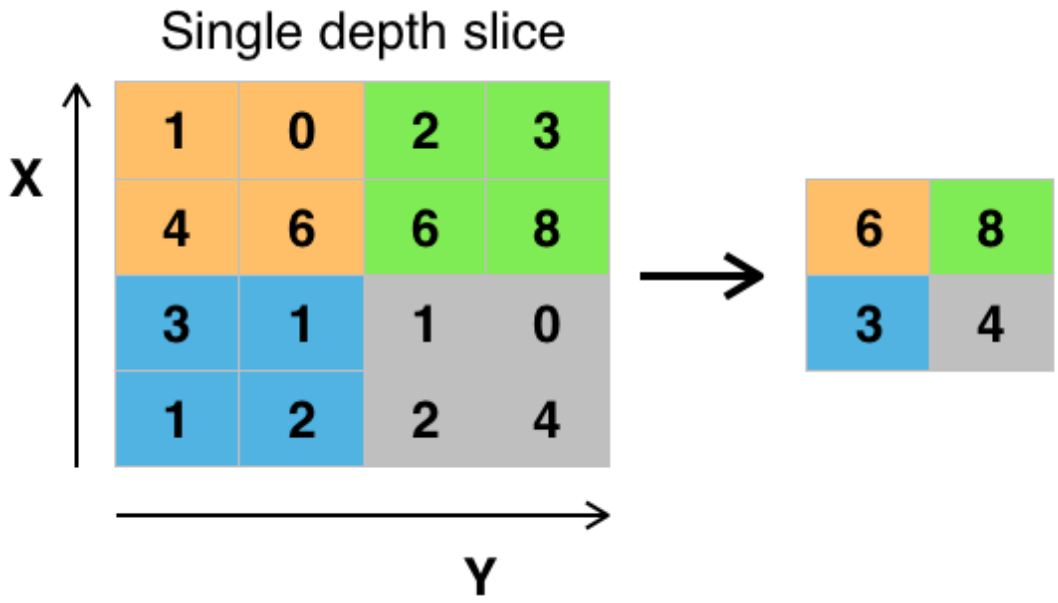


Figure 2.2: Max pooling with a 2x2 filter and stride=2 [2]

The last layers of a CNN typically consists of **Fully Connected Output Layer**s, where each element (neuron) on the input is connected to every element at the output, hence the name. This is not the case at convolutional layers, where only a part of the input has an influence on a single output neuron. For the activation function, which most often acts as the classifier, several possibilities exist. Prominent representatives include Softmax for multi-class problems and Sigmoid for binary decisions.

[2] https://en.wikipedia.org/wiki/Convolutional_neural_network#/media/File:Max_pooling.png

Adoption of weights during traing is achieved by applying the back-propagation algorithm. A network is typically trained for multiple epochs and the progress determined via the training error/loss [15]. This includes stopping the training process when enough progress is made. To determine such, one strategy could be to stop after the value drops below a certain predefined number, or when it does not drop any longer.

Overfitting is a problem where the network adapts too close to the training data and looses the ability to generalize on unknown inputs, resulting in a high test error. There are techniques to prevent this from happening, such as Dropout and Batch Normalization. The latter normalizes the output of a layer and helps to omit the so-called covariant shift [16]. With dropout, some of the neurons are dropped from the network each iteration to avoid the model becoming too focused on single neurons by attaching a high weight to it [17] Furthermore, stopping training at the right time also prevents overfitting, done through model validation during the training process.

2.1.3 CNN structure by Toprak and Toygar

The following structure is the reference implementation from the base paper [3]. It consists of five convolutional layers with a 3x3 kernel and uses ReLU as the activation function. The number of output feature maps from the first to the last convolutional layers are 32, 64, 64, 96 and 32. Following each convolutional part, max pooling with a size of 2x2 is applied, effectively shrinking the input to one quarter of the original size. Batch Normalization comes next. One layer in the network can be described to be processed in the following order:

1. Convolution
2. ReLU activation
3. Max Pooling
4. Batch Normalization

After the five iterations of the items described and before flattening the output feature maps for the fully connected layer, dropout with a 20 percent probability is applied. This aims to combat eventual overfitting problems. After the single fully connected layer,

Softmax is used to do the binary classification task between real and spoof images. The last three steps after the layers therefore are:

1. Dropout
2. Fully Connected Layer
3. Softmax

The number of training epochs is set to 250. A graphical representation of the described network can be seen in image 2.3.

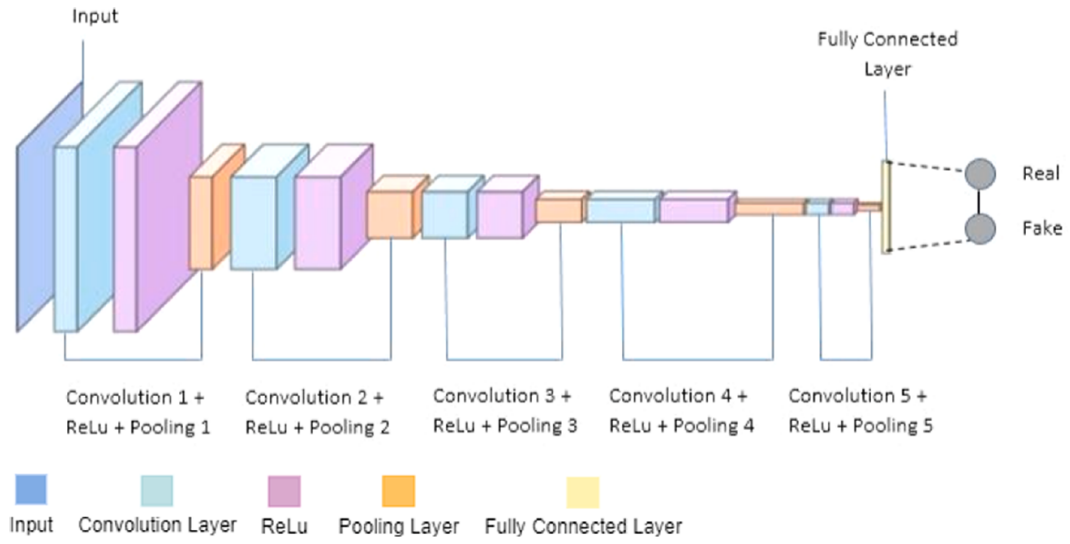


Figure 2.3: Convolutional Neural Network for real/fake decision in spoof detection of ear biometrics [3]

2.2 Performance metrics

For evaluating the trained neural network, some metrics need to be defined that produce understandable numbers, interpretable graphical representations. Originally, only three scores were used (False Fake Rate, False Genuine Rate, Half Total Error Rate) [3], which focus on the incorrectly classified images. In terms of security, which is one of the main reasons behind the usage of biometrics, this is a good idea. False Fake Rate is also known

as False Negative Rate (FNR), False Genuine Rate as False Positive Rate (FPR) and the Half Total Error Rate is the arithmetic mean between the other two rates. For this work, the number of performance parameters is extended for a more precise overview of the models' performance.

First off, the four elementary values gathered are True Positives (TP), True Negatives (TN), False Positives (FP), False Negatives (FN). The first describes those elements which were predicted as genuine/real and actually are. The second one is the same, but for negatives (spoof). False Positives is the term for elements which were predicted as genuine, but are in reality spoofs. False Negatives are predicted as fake, but are truly real images. Obviously, TP and TN should be as high as possible, and FP/FN low. However, for biometrics, having False Positives can be more of an issue than False Negatives, as the first means that some attacker successfully tricked the system and possibly got access to sensitive data or services. With these four values, the confusion matrix is built, that displays these values in a more understandable way by aligning them in a 2x2 configuration.

From here one, metrics can be calculated, among them the already mentioned False Negative Rate and False Positive Rate. Others include the Sensitivity (True Positive Rate, Recall) and Specificity (True Negative Rate), Accuracy within each class and over all classes (real/spoof), Precision and F1-Score. The following listing gives an overview of the different terms, what they mean and how they are calculated [18].

- **Accuracy**: The correct predictions over all classes.
- **Precision**: Another term is Positive Predictive value; the number of correct positive (i.e. genuine) classifications among all items classified positive.
- **Recall**: Also known as True Positive Rate, Sensitivity; answers how many actual positive values were identified as such.
- **Specificity**: can also be called True Negative Rate; tells how many actual negative samples have been identified as such.
- **F1-Score**: It is a measure of the accuracy of the testing and tells, with a value between 0 and 1, how precise the tests were (higher is better).
- **False Positive Rate**: The opposite of the True Negative Rate; shows the fraction of elements incorrectly classified as positive among actual positive elements.

- **False Negative Rate**: The opposite of the True Positive Rate; shows the fraction of elements incorrectly classified as negative (therefore actually positive) among all negative elements.

$$
\begin{aligned}
Accuracy &= \frac{TP + TN}{TP + TN + FP + FN} \\
Precision &= \frac{TP}{TP + FP} \\
Recall &= \frac{TP}{TP + FN} \\
Specificity &= \frac{TN}{TN + FP} \\
\text{F1-Score} &= \frac{2 \cdot Precision * Recall}{Precision + Recall} \\
FPR &= \frac{FP}{FP + TN} = 1 - TNR \\
FNR &= \frac{FN}{FN + TP} = 1 - TPR
\end{aligned}
$$

3 Dateset

The work of Toprak and Toygar uses a total of five datasets, namely AMI [19], UBEAR [20], IITB [21], USTB set 1 and set 2 [22]. These all features images of ears, captured from humans in different angles, both the left and right ear, various image sizes, grayscale or color.

3.1 Used datasets

Even though the authors used five different datasets, not all were not available to download or there was no response to access requests. Overall, only AMI and IITD images were available and only after explicitly applying for either passwords or a login information. This leaves this project with only two out of five. The images are therefore fused together into a single dataset and the CNN can train on all of them. This is in contrast to the original paper, where only one set was tested (and probably trained) on at a time and the networks could better adapt to the single given type of images, at least in theory.

In total, 2070 coloured images picturing left and right ears (even though there are many more right ears) are in the combined dataset. 700 images are from AMI and 1370 from IITD. AMI image size is 492x702 pixels, IITD features 227x227 pixels. Two example images from AMI are visible in figure 3.1, another two for IITD in figure 3.2.

(a) Right (female) ear with earing and longer black hair

(b) Left (male) ear without jewellery and white/gray hair

Figure 3.1: Two exemplary images from the AMI ear database

(a) Right (male) ear with short white hair

(b) Left (female) ear with visible earholes without jewellery and redish-brown hair

Figure 3.2: Two exemplary images from the IITD ear database

3.2 Spoof images and generation

As with genuine datasets, there do exist spoofed ones for different kinds of biometric features, most prominent among them are fingerprints [23]. In contrast, there are no datasets for spoofed ear data publicly available and the authors of the original paper, Toprak and Toygar, did not reply to an email about access to their spoof data. This is why the fake data had to be generated manually.

To do so, a total of 2070 images, separated into a 3x3 grid on each A4 sheet of paper, were printed double-sided. A total of 115 sheets were needed. 50 were printed using a "Canon MG7150" inkjet, the remaining 65 pages by an office-grade "Ricoh MP C6503 PCL 6" device. This represents the same method an attacker would use for an analogue presentation attack. To re-digitalize the images, the main camera of a "Nothing Phone (1)" captured everything using the "Microsoft Lens" application, which also allowed to directly crop larger white borders away. The images were taken with indirect natural lighting, however not all at the same time of day, which can result in some variation of the final image quality. After cropping, the images were resized to the longer side (in this case the vertical axis) having 1000 pixels, the shorter one around 750 to 760 pixels. Additional resizing/cropping is only done later on with the help of image transformations along with more data augmentation, using PyTorch.

Due to an error in data capture, one sheet with a total of 18 images was captured twice. Even though manual search and semi-automated image compare tools were utilized, the duplicates could not be found. As a file content search did not reveal any duplicates, the assumption can be made that the images were indeed captured twice, not just digitally duplicated. This means that some ears now have twins in the spoof data, but they are not the exact same pictures. This problem is, also with the addition of randomized augmentation, not of significance, but is mentioned for transparency and clarity reasons.

(a) Spoofed image example 1 **(b)** Spoofed image example 2

Figure 3.3: Two exemplary images from the generated spoofed data

3.3 Data augmentation

Deep learning neural networks, like CNNs, need a lot of data to be trained properly [24]. The availability of the required amounts of data however is often lacking at best, which leaves the question on how to work around this problem. Fortunately, data augmentation offers a viable solution. While techniques such as dropout, batch normalization, transfer learning etc. can help with the situation, modifying the limited amount of input data and generating various versions of it addresses the root cause of the problem. Data augmentation comes in various forms, ranging from simpler geometric transformations (flipping, cropping, rotation, translation, colour) to more advanced photometric operations (Kernel filters, image mixing, random erasing) up to augmentation with the help of deep learning. An overview of the different approaches is shown in image 3.4.

This project focuses mainly on the usage of geometric transformations, including some colour work, such as randomly applied sharpening, greyscale conversion and randomized jitter. More on this in the implementation section 4.1.2 Loading images and data augmentation. Following, some transformation methods are presented [24].

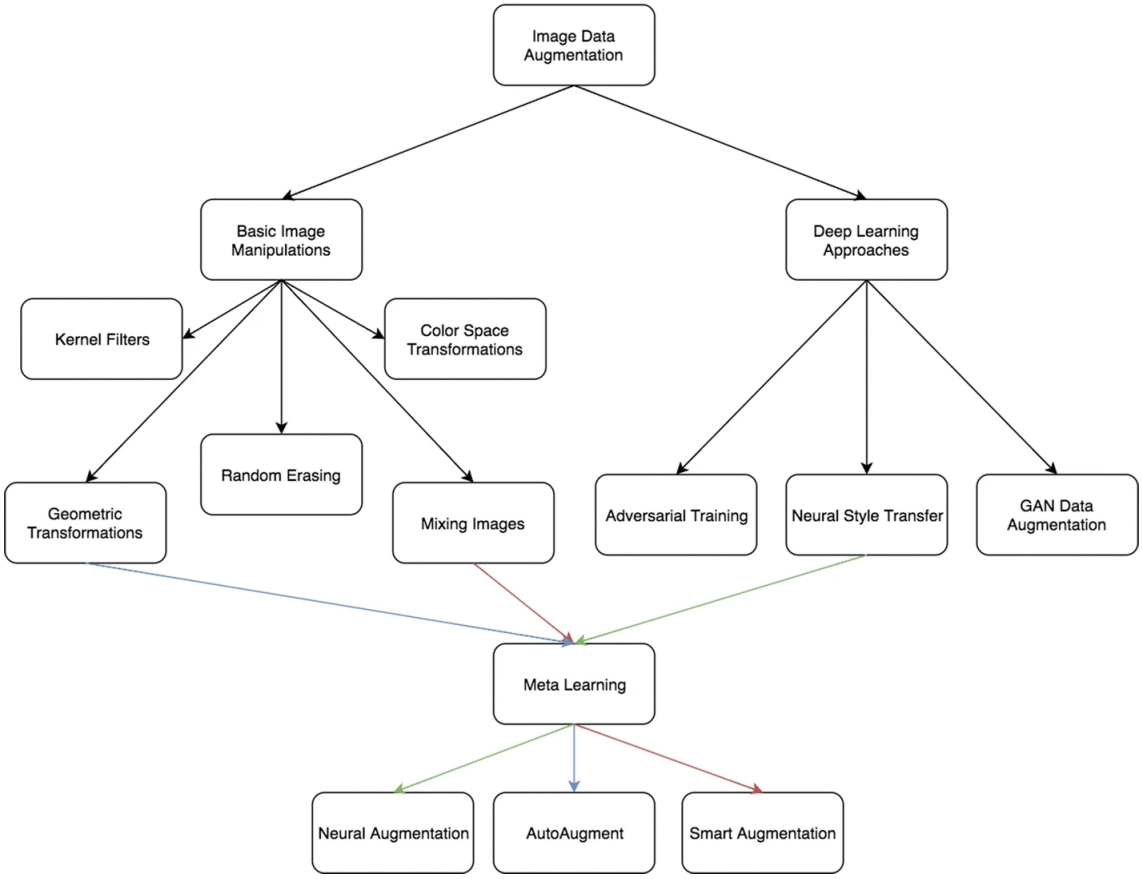
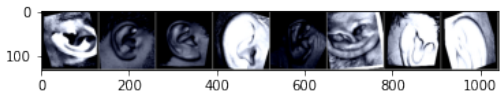


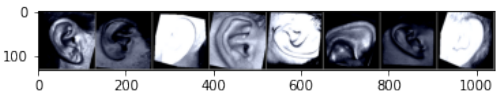
Figure 3.4: Overview of different approaches to image augmentation techniques [24]

- **Flipping**: Flip the image along the horizontal or vertical axis, where the first method is more prevalent. However, it cannot be applied to any dataset, especially those where the data orientation is critical (text recognition for example).
- **Cropping**: Reduces the size of an image and only leaves the central area of an image, usually to get uniform width/height on input data. Depending on how much is cropped out, this operation can also destroy relevant information.
- **Rotation**: An image can be rotated by 1 to 359 degrees along an axis. Usually, smaller values have proved to be useful. Higher values can mess with image data where the data orientation is relevant, as with flipping.

- **Translation**: Shifts the image along an axis and is used to avoid positional bias in images. An example is facial image data, where all faces are centred, which in the real world is often not the case. With translation, the images are placed off-centre. Translation does not reduce the image size, unlike cropping.
- **Noise injection**: Places random values in an image and therefore generating noise, which may disturb features in an image and force a network to learn more generalized concepts instead of overly exact details.
- **Colour space transformations**: Is mostly used to quickly manipulate the colour values in images, such as the brightness level, min/max RGB values, colour shifts, image sharpening, white balance, etc.



(a) Grid with eight augmented images 1



(b) Grid with eight augmented images 2

Figure 3.5: Two exemplary images from the generated spoofed data

4 Implementation

The project is implemented in Python, using a machine learning framework named PyTorch. The code execution is done on Kaggle [1], which provides free, though limited access to CUDA-capable GPUs which can accelerate CNN learning dramatically compared to systems without such graphics units. Jupyter Notebooks were used, which offer benefits such as cell-based execution of code and the possibility to structure by adding markdown elements between code. The outputs, texts, images etc. are additionally preserved and can be viewed again at another time/day and on another system.

This chapter only deals with the basic structure on how to implement a Convolutional Neural Network and the fundamental code elements. The snippets presented will not enable anyone automatically to implement their own CNN, but can help with the underlying structure and flow of data through a CNN. The complete source code is freely available at `https://github.com/Stern1710/ear-spoof-networks` and may be copied, modified, but ships without any warranty.

The project is subdivided into three separate projects, where each of them is responsible for one task, either training, hyperparameter optimization or testing. Once the best set of hyperparameters is determined, the models are trained and the best result saved to a file. This enables the evaluation to be done without re-training the model, risking finding a different (and perhaps worse) model each time.

[1] `https://www.kaggle.com/`

4.1 Model training

4.1.1 Parameters

There are several parameters that are defined at the beginning, including the batch size, weights for class distribution, maximum number of epochs to be trained, parameters for early stopping and the computing device (which is critical for performance).

```
batch_size = 8
fcl1 = 32
fcl2 = 16
learning_rate = 0.001
num_epochs = 50

# for early stopper
es_patience=4
es_min_delta=0.005

real = 2070
spoof = 2088
w_spoof = 1 - (spoof/(spoof+real))
w_real = 1-w_spoof

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
```

4.1.2 Loading images and data augmentation

To apply data augmentation, PyTorch offers a variety of transformers, which are easy to both define and apply. They are applied in the order they are written down. In this case, first a random sharpness adjustment is done and the image converted to grayscale, with three output channels. A random rotation and perspective distortion is laid on top next, then the image is resized to 132x132 and finally cropped to 128x128. The last step is done in order to omit any white borders on the spoof data. Finally, the images are transformed into tensors and a normalization on the values is applied, which should help to smoothen the images a little bit. For all random adjustments, where no probability is explicitly defined (p=...), it is 50:50.

```
all_transforms = transforms.Compose([
# Color randomizers
transforms.RandomAdjustSharpness(sharpness_factor=2),
# Convert to grayscale
transforms.Grayscale(num_output_channels=3),
# Randomized whole image manipulation
transforms.RandomRotation(20),
transforms.RandomPerspective(distortion_scale=0.2, p=0.4),
# Resize, to tensor and normalize the image.
transforms.Resize((132,132)),
transforms.CenterCrop(128),
transforms.ToTensor(),
transforms.Normalize(mean=[0.4914, 0.4822, 0.4465],std=[0.2023, 0.1994,
    0.2010])
])
```

A whole dataset can be loaded by one command.

```
dataset = datasets.ImageFolder(img_path, transform=all_transforms)
```

The dataset then needs to be split into three parts. The largest one is the training data, which in this application takes 64 percent of all images, validation with 16 percent and test data with 20 percent. The split is done by a random split. For reproducible results, the generator value in this case is fixed.

```
train_size = int(0.8 * len(dataset))
test_size = len(dataset) - train_size
train_dataset, test_dataset = torch.utils.data.random_split(dataset, [
    train_size, test_size], generator=torch.Generator().manual_seed(42))

train_orig = train_size
train_size = int(train_size * 0.8)
val_size = train_orig - train_size
train_dataset, val_dataset = torch.utils.data.random_split(train_dataset, [
    train_size, val_size], generator=torch.Generator().manual_seed(42))
```

Finally, the data is passed to the data loaders, which will on each iteration give out data (images) in the amount specified by the batch size. This is done for all three parts of the data.

```
train_loader = torch.utils.data.DataLoader(dataset = train_dataset,
    batch_size = batch_size, shuffle = True)
```

```
test_loader = torch.utils.data.DataLoader(dataset = test_dataset,
    batch_size = batch_size, shuffle = True)
val_loader = torch.utils.data.DataLoader(dataset=val_dataset, batch_size =
    batch_size, shuffle = True)
```

4.1.3 Defining a CNN in PyTorch

The actually implemented model differs in some areas from the one presented by Toprak and Toygar. It does still feature the same number of convolutional layers, but the number of fully connected layers has increased to three, as well as the classification function changed from Softmax to Sigmoid.
The definition of a model is split into two parts. First, the initialization of its parts, second the connection between the different items, also called the forward pass. The back propagation of error is done at a later point, again with PyTorch provided functions.

```
# Creating a CNN class
class SpoofDetectionNetwork(nn.Module):
  #  Determine what layers and their order in CNN object
    def __init__(self, l1=64, l2=16):
        ...

    # Progresses data across layers
    def forward(self, x):
        ...
```

For the initialization, a typical layer configuration looks like the following example (in this case for the very first layer). The number of input/output channels, the batch normalization layers etc. have to be adapted to the according values, but other than that the elements are basically similar.

```
# Layer 1
self.conv1 = nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3)
self.relu1 = nn.ReLU()
self.max_pool1 = nn.MaxPool2d(kernel_size = 2, stride = 2)
self.batchNorm1 = nn.BatchNorm2d(32)
```

After all convolutional layers have been created, dropout is applied and the three fully connected layers, each again with ReLU as the activation functions, come next. The

number of layers for the output of layer 1 and layer 2 are determined by hyperparameter search, therefore in the parameters section, and only passed into the network as parameters. As the last step, the sigmoid activation function is defined, which maps the value between 0 and 1 for the respective class, with 0.5 meaning that the network is inconclusive about the class assignment. This threshold value can be shifted according to the needs of the designer (for security: Safety vs. Convenience) of use. In case there were more than two classes, a change back to Softmax is necessary, as Sigmoid only works well for binary tasks.

```
# After convolution layers
self.dropout = nn.Dropout(0.2)
self.fc1 = nn.Linear(128, l1)
self.fc_relu1 = nn.ReLU()
self.fc2 = nn.Linear(l1, l2)
self.fc_relu2 = nn.ReLU()
self.fc3 = nn.Linear(l2, 1)
self.fc_sigmoid = nn.Sigmoid()
```

The forward pass defines how the input data is passed through the network. An example for the convolutional layers, again for number 1, is given below, where x defines the input to the network.

```
out = self.conv1(x)
out = self.relu1(out)
out = self.max_pool1(out)
out = self.batchNorm1(out)
```

The fully connected and output layers are wired in similar way as before, only the reshape is the sole thing not defined in the init section. It's there to flatten all the data into a one-dimensional array, which is then passed into the fully connected layers. Last of all, the output is returned.

```
out = self.dropout(out)
out = out.reshape(out.size(0), -1)

out = self.fc_relu1(self.fc1(out))
out = self.fc_relu2(self.fc2(out))
out = self.fc_sigmoid(self.fc3(out))

return out
```

4.1.4 Early training stop

Avoiding overfitting, where a model adapts too closely to the training data and looses the ability to generalize, is very essential for training an efficient neural network. Countermeasures such as dropout and normalization have already been taken care of in the network itself. Another phenomenon is, that with too many training epochs, the network will overfit as well. Therefore, stopping as soon as the generalization ability becomes worse is important.

For this task at hand, the idea is as followed, based on an answer to the question on designing and early stopper on the webpage StackOverflow [2]:

- Whenever a new iteration of the model performs better than the old one, save this version
- To avoid a too early stop, a patience parameter is introduced. It defines how many times the accuracy value during the validation phase can go up, before the training is stopped. Finding a new best model resets the internal patience counter.
- The model performance can vary from run to run, sometimes very minimal only. In order to avoid increasing the patience counter every time with only a very small difference between the validation accuracy, a delta parameter is introduced. Only when the current performance is worse than the current best plus the delta, the patience counter is increased. Otherwise it is not affected, so not reset either.

```
1  class EarlyStopper:
2      def __init__(self, patience=1, min_delta=0):
3          self.patience = patience
4          self.min_delta = min_delta
5          self.counter = 0
6          self.min_validation_loss = np.inf
7          self.improved = False
8
9      def early_stop(self, validation_loss):
10         if validation_loss < self.min_validation_loss:
11             print(f"New best loss {validation_loss:.10f}, was: {self.
       min_validation_loss:.10f}")
12             self.min_validation_loss = validation_loss
```

[2] https://stackoverflow.com/questions/71998978/early-stopping-in-pytorch

```
            self.counter = 0
            self.improved = True
        elif validation_loss > (self.min_validation_loss + self.min_delta):
            print(f"Val_Loss: {validation_loss}; Threshold: {self.
    min_validation_loss + self.min_delta}")
            self.counter += 1
            self.improved = False
            if self.counter >= self.patience:
                return True
        else:
            print("Within delta")
            self.improved = False
        return False
# based one: https://stackoverflow.com/questions/71998978/early-stopping-in
    -pytorch
```

4.1.5 Setup for training

With the data loaded, network defined, early stopper in place, only the backward pass (loss function) is still open for implementation and the different parts have to be connected, before training can begin.

```
model = SpoofDetectionNetwork()
if torch.cuda.device_count() > 1:
    model = nn.DataParallel(model)
model = model.to(device)
best_model = None

# store losses to plot later
val_losses = []
train_losses = []

# Set Loss function with criterion
criterion = nn.BCELoss().to(device)
# Set optimizer with optimizer
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate,
    weight_decay = 0.005, momentum = 0.9)
total_step = len(train_loader)

# Init the early stopper
early_stopper = EarlyStopper(patience=es_patience, min_delta=es_min_delta)
```

```
actual_epochs = 0
```

4.1.6 Training

This part, again, may be broken into a number of parts, that are connected with each other, but looking at them individually gives a better understanding of the mode of operation. The network is trained up to the maximum number of epochs, yet this number is set quite high on purpose to avoid underfitting, and the early stopper should typically kick in first to avoid overfitting.
As a first step in each epoch, the network iterates over all images from the test set. Per batch, it calculates the loss, then performs the back propagation and optimization. The loss value for each batch is recorded and later calculates the loss for the entire epoch. The second step calculates the validation loss on the according dataset, which is a separate one from training. The evaluation is quite comparable to a single training step, but does not perform back propagation of error. The validation loss for each batch is again counted and stored for the entire epoch. In the last step, the early stopper checks if the criteria for ending the training are fulfilled. If so, the best model is set as the final model, otherwise it is check whether an improvement has taken place. When it does, a deep copy of the model is created, to avoid being overwritten by the next epoch, and stored as the current best model for comparison purposes.
Throughout training, some text output is created, informing the user on the current epoch, the train and evaluation loss, improvements and training stop.

```
for epoch in range(num_epochs):
    epoch_train_loss = []
    for i, (images, labels) in enumerate(train_loader):
        # Move tensors to the configured device
        images = images.to(device)
        labels = labels.to(device)

        # Forward pass
        outputs = model(images)
        labels = labels.unsqueeze(1).float()

        criterion.weight = labels * w_spoof + (1-labels)*w_real
        loss = criterion(outputs, labels)

```

```
        # Backward and optimize
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        epoch_train_loss.append(loss.item())

    train_losses.append(sum(epoch_train_loss)/len(epoch_train_loss))

     # Validation loss
    epoch_val_loss = []
    with torch.no_grad():
        for inputs, labels in val_loader:
            inputs, labels = inputs.to(device), labels.to(device).int()

            outputs = model(inputs)
            predicted = torch.flatten(torch.round(outputs)).int()
            labels = labels.unsqueeze(1).float()

            criterion.weight = labels * w_spoof + (1-labels)*w_real
            loss = criterion(outputs, labels)

            epoch_val_loss.append(loss.item())

    val_losses.append(sum(epoch_val_loss)/len(epoch_val_loss))

    print('Epoch [{}/{}], Train-Loss: {:.4f}, Val-Loss: {:.4f}'.format(
    epoch+1, num_epochs, sum(epoch_train_loss)/len(epoch_train_loss), sum(
    epoch_val_loss)/len(epoch_val_loss)))
    actual_epochs = epoch+1
    # Run early stopper and check whether to break
    if early_stopper.early_stop(sum(epoch_val_loss)/len(epoch_val_loss)):
        print(f"Early stop of training in epoch {epoch+1}, values did not
    decrease sufficiently anymore")
        model = best_model
        break

    if early_stopper.improved:
        print("Copy new best model")
        best_model = deepcopy(model)
```

Once training has finished, a quick evaluation is done which gives the accuracy values for the network and the loss for training and validation per epoch is displayed. For the former, it is discussed in more detail in section 4.3 and therefore will not be explained here. The latter however is only possible when training, as it needs the collected loss data. The code for the line graph plotting both losses and the epoch numbers is listed below.

```
plt.figure(figsize=(20,6))
plt.title("Training and Validation Loss")
plt.plot(range(1,actual_epochs+1), val_losses,label="val", linestyle = '
    solid')
plt.plot(range(1,actual_epochs+1), train_losses,label="train", linestyle =
    'dashed')
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.xticks(range(1,actual_epochs+1))
plt.legend();
```

4.1.7 Exporting a model

Training a model is a very resource intensive task, both for the hardware and on the users' time budget. Additionally, due to shuffling of training data, random splits in the training/test data, and so on, no two trainings produce the exact same results and neither are two best models. It therefore makes sense to save the parameters of a model, and later load it again, perhaps also in a separate program, to do the evaluation. PyTorch supports saving the model by just using a simple command, where `output_path` is an actual path on the file system where one wants to save the model file too. The file format is `.h5` by convention.

```
torch.save(model.state_dict(), 'output_path/filename.h5')
```

4.2 Hyperparameter search

The term hyperparameter has been called out a couple of times already, but how are these values being found? One possibility is to get them from papers and other source material, or to search for them via hyperparameter search. In large parts, the principles

of finding these are already familiar from training the network itself. The same neural network is used, with the same image data loaders, an only slightly modified train function with validation loss and the accuracy test, practically the same as in Model evaluation. A few additional parts are added which handle the search for hyperparameters itself. These are from the Ray Tune library [3] and provide things such as reporting, efficient early termination of misleading search processes, a maximum number of combinations to search on, among many more

Imports, parameter definition, model definition, data augmentation and data loading will be skipped in this section, as they have been done already and would be quite similar or carbon copies. Please check the source code for further information.

4.2.1 Checkpoints

The training and validation process is complemented by so-called checkpoints. These save the model throughout training multiple times and can be used for various reasons, but here simply to avoid loosing progress in case something goes south. Checkpoints can restore the latest version of the model and start training from the last save onwards. Once per epoch, the validation loss and accuracy is reported to raytune and used to abort training when no progress over the current best configuration is found. The three dots (. . .) symbolize left out code in these areas.

```
def train_model(config, checkpoint_dir=None, data_dir=None):
    net = Net(config["l1"], config["l2"])
    ...

    if checkpoint_dir:
        model_state, optimizer_state = torch.load(os.path.join(
    checkpoint_dir, "checkpoint"))
        net.load_state_dict(model_state)
        optimizer.load_state_dict(optimizer_state)

    for epoch in range(config['epochs']):  # loop over the dataset multiple
     times
        ...

        # Validation
```

[3] https://docs.ray.io/en/latest/tune/index.html

```
        ...
        with tune.checkpoint_dir(epoch) as checkpoint_dir:
            path = os.path.join(checkpoint_dir, "checkpoint")
            torch.save((net.state_dict(), optimizer.state_dict()), path)

        tune.report(loss=(val_loss / val_steps), accuracy=correct / total)
```

4.2.2 Evaluated hyperparameters

Generally, five different hyperparameters were tested. As the early stopper for network training was only implemented after hyperparameter search, the epochs search result was ignored in the end.

- **l1**: The number of outputs in the first fully connected layer, values between 16 and 256
- **l2**: The number of outputs in the second fully connected layer, values between 4 and 64
- **lr**: The learning rate of the network, values between 0.1 and 0.0001
- **batch_size**: The number of items presented to the network in each learning step, values are 2, 4, 8, 16
- **epochs**: The number of epochs the network is trained, values are 10, 15, 20, 25. Due to the implementation of the early stopper, the result of these values did not matter in the end, but nonetheless interesting.

```
config = {
    "l1": tune.sample_from(lambda _: 2 ** np.random.randint(4, 8)),
    "l2": tune.sample_from(lambda _: 2 ** np.random.randint(2, 6)),
    "lr": tune.loguniform(1e-4, 1e-1),
    "batch_size": tune.choice([2, 4, 8, 16]),
    "epochs": tune.choice([10,15,20,25])
}
```

4.2.3 Reporting, scheduling and execution

Schedulers are Ray Tune's way of terminating ill-fated searches early on, as well as pause trials, or clone them etc. if needed [4]. For this project, the early termination proofed rather useful. Ray advocates to use the ASHA scheduler, so it was. The reported validation loss from the training phase functions as the metric for the scheduler, which has to be minimized. A maximum of 25 epochs is trained.

```
num_samples=15
max_num_epochs=25
gpus_per_trial=2

scheduler = ASHAScheduler(
    metric="loss",
    mode="min",
    max_t=max_num_epochs,
    grace_period=2,
    reduction_factor=2)
```

Ray tune does automatically write a lot of information to the output, but not all of it is useful. Therefore, a self-defined CLIReporter was configured to only report every two minutes and tell the user the latest loss, accuracy and iteration count.

```
reporter = CLIReporter(
    # parameter_columns=["l1", "l2", "lr", "batch_size"],
    metric_columns=["loss", "accuracy", "training_iteration"],
    max_report_frequency = 120 # only report every 2 minutes
)
```

To kick off the actual parameter search, the configured elements have only to be plugged together into at `tune.run` method. The data, configuration, scheduler, reporter are passed, along with the number of different combinations (15) the tuning algorithm should try.

```
result = tune.run(
    partial(train_model, data_dir=data_dir),
    resources_per_trial={"cpu": 1, "gpu": gpus_per_trial},
    config=config,
    num_samples=num_samples,
    scheduler=scheduler,
    progress_reporter=reporter,
```

[4] https://docs.ray.io/en/latest/tune/api/schedulers.html

```
8     verbose=1)
```

The last execution cell gets the winner configuration and prints the values to the CLI. These are the values, with the aforementioned exception of epochs, that are plugged into the parameters in 4.1.1.

```
1 best_trial = result.get_best_trial("loss", "min", "last")
2 print("Best trial config: {}".format(best_trial.config))
```

4.3 Model evaluation

As with testing and hyperparameter optimization, evaluating the trained CNN also has some shared code with the other two. These include, once more, data loading, augmentation and neural network definition, which are therefore not listed here. This section will however have a look at how to evaluate the network and how to get the statistics (described in 2.2 Performance metrics),

4.3.1 Import a model

First, the exported neural network has to be imported, which can be done via a simple PyTorch command to load the state dictionary. Another one evaluates the correctness of the loaded information.

```
1 model = SpoofDetectionNetwork()
2 model.load_state_dict(torch.load("/kaggle/input/personal-best-model-for-ear
    -spoof/best_model_32_16_8_0.001_4_0.005_0.498.h5"))
3 model.eval()
```

4.3.2 Testing

Testing works quite similar to validation, but with the difference that no loss is calculated this time, but the number of (in-)correct classifications. Furthermore, testing is done on a larger subset of data and typically more information about the predictions is stored. For this project, the number of correct classifications and overall samples per class (i.e.

genuine / spoof) is taken, along the predicted and actual labels of a given image. The whole procedure works again with batches at a size of eight. The data is sent into the model, the predictions gathered from it and stored with the actual (true) labels. Next on, to get the number of proper classifications per class, a one-on-one comparison between each pair of labels is done and the overall and correct number calculated.

```
# Testing
with torch.no_grad():
    n_class_correct = [0 for i in range(num_classes)]
    n_class_samples = [0 for i in range(num_classes)]

    pred_labels = np.array([])
    act_labels = np.array([])

    for images, labels in test_loader:
        images = images.to(device)
        labels = labels.to(device).int()
        outputs = model(images)
        predicted = torch.flatten(torch.round(outputs)).int()

        pred_labels = np.append(pred_labels, predicted.cpu())
        act_labels = np.append(act_labels, labels.cpu())

        for i in range(labels.size(dim=-1)):
            label = labels[i]
            n_class_samples[label] += 1
            if (label == predicted[i]):
                n_class_correct[label] += 1
```

4.3.3 Reporting values

The first value of interest is the accuracy of the whole network and per class. To get these, the number of correct classifications is divided by the number of overall samples. For a nicer presentation on the CLI, the values are then plugged into an `PrettyTable`, that does some output formatting before printing.

```
# Accuracy values
t = PrettyTable(['Class', 'Accuracy'])
acc = 100.0 * np.array(n_class_correct).sum() / np.array(n_class_samples).
    sum()
```

```
t.add_row(['Overall', f"{acc:.2f}%"])

for i in range(num_classes):
    acc = 100.0 * n_class_correct[i] / n_class_samples[i]
    t.add_row([dataset.classes[i], f"{acc:.2f}%"])

print(t)
```

The rest of the listed evaluation criteria can be nicely computed by a confusion matrix. The `sklearn` library provides a class for this purpose, which needs only to be fed by the true and predicted labels of the network. Additionally, there is also the option to display the matrix for a given data object.

```
conf_mtx = confusion_matrix(act_labels, pred_labels)
disp = ConfusionMatrixDisplay(confusion_matrix=conf_mtx, display_labels=['
    real', 'spoof'])
disp.plot();
```

The confusion matrix also lets the user retrieve the values for True Positives, True Negatives, False Positives and False Negatives. These are used to calculate the statistics such as the precision, recall, specificity, false rates and the F1-Score.

```
TP, FN, FP, TN = conf_mtx.ravel()

rec = ((TP) / (TP+FN)) * 100
spec = ((TN) / (TN+FP)) * 100
prec = ((TP) / (TP+FP)) * 100
fpr = ((FP) / (FP+TN)) * 100
fnr = ((FN) / (TP+FN)) * 100
f1s = ((2 * prec * rec) / (prec + rec))

t = PrettyTable(['Precision', 'Sensitivity/TPR/Recall', 'Specificity/TNR',
     'FPR', 'FNR', 'F1-Score'])
t.add_row([f"{prec:.2f}%", f"{rec:.2f}%", f"{spec:.2f}%", f"{fpr:.2f}%", f"
    {fnr:.2f}%", f"{f1s:.2f}%"])
print(t)
```

5 Testing and Results

This section is dedicated to the results generated by the implemented code. It is split into three sections, the first one presents the selected hyperparameters which were themselves used to train the model. The second part has a look at the achieved training and validation loss and its progression over time, which can also explain why or when the early stop was triggered. The last part, which is also the longest one, dedicates itself to the evaluation of various calculated rates and scores from section 2.2 Performance metrics.

5.1 Hyperparemeter search

The search over 15 different combinations concludes with the result listed below. They seem reasonable, as they are neither on the extreme ends of a spectrum and could achieve very low loss and high accuracy values on the validation data. Additionally, they are not extreme values for any of the searched parameters, which could hint that the results were a lucky one-shot only and could cause problems when actually training the model.

- **l1**: 32
- **l2**: 16
- **lr**: 0.0007888482098784208, which is rounded up to 0.001
- **batch_size**: 8
- **epochs**: 15, but this value is ignored and set to 50.

5.2 Training and validation loss

The model was trained for 36 epochs, during which each epoch both the training and the validation loss were recorded. The latter value was then used to determine whether training has already gone far enough and the model begins to overfit, or further tuning is necessary. This is done via the early stopper. At the 32 epoch, the best model was found. After this, the validation loss increased and stayed steady above the limit, which caused the patience counter to rise and cancel training in the 36th epoch. During the last couple of rounds, the training loss further decreased, which seemed to indicate that the model started to overfit slightly, as the validation loss did not drop again within the delta range of the best evaluation. The minimum validation loss value was found to be 0.0094807.

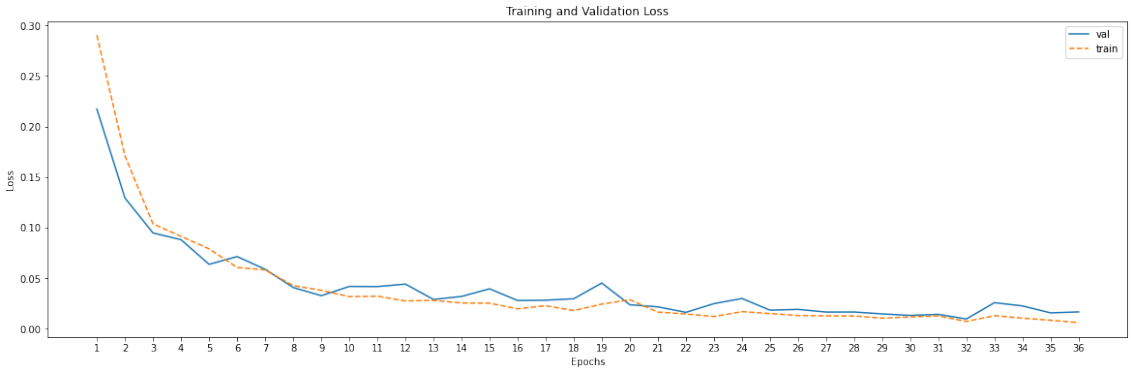


Figure 5.1: Loss curve of the model over training and validation data set for all 36 trained epochs.

5.3 Trained model evaluation

5.3.1 Confusion matrix

The confusion matrix displays the True Positives (TP), True Negatives (TN), False Positives (FP) and False Negatives (FN) in a 2x2 grid. On the horizontal axis are the true (actual) labels, the verticals hold the prediction from the neural network. The upper left square is for TP, upper right for FN, lower left for FP and lower right for TN.

As seen in image 5.2, the model performs exceptionally well. Only one image is classified as False Positive or False Negative, respectively. For an application securing a facility,

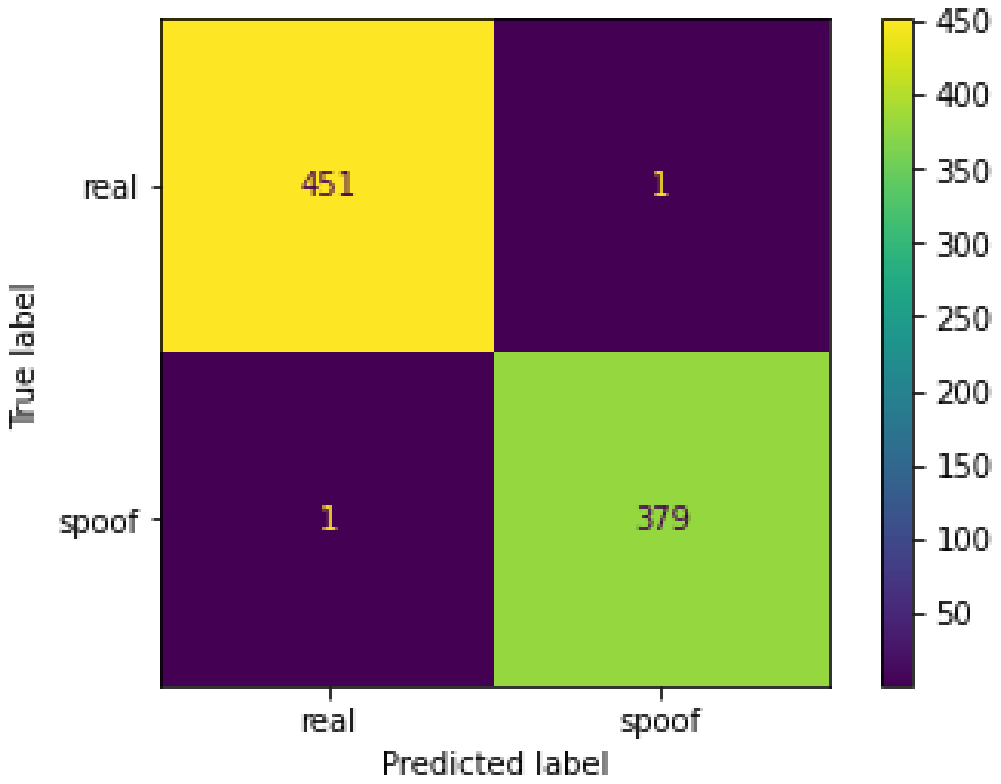


Figure 5.2: Confusion matrix of the trained model on the test dataset.

this would mean that of 832 scans performed by the ear detection system, one fraudulent person got access and one legitimate person was left out. Overall, this is still a very good result. In case no False Positives are allowed, due to the very high costs associated with it, the threshold value for the classification may be shifted. Decisions are more likely to be negative and omit False Positives, although this for sure will increase the number of False Negatives along.

5.3.2 Accuracy, Precision, Recall, Specificity, F1-Score

The accuracy score is the percentage of correct classifications over all decisions made. With an excellent 99.76 percent, it shows that the network configuration and training was very successful, and the application can quite accurately decide between the two classes. For real images, the percentage is 0.02 percent higher. As there are more images in this class

the one False Negative does not have as big of an impact as for the fake data, as it is 0.02 percent below the total average. The data is listed in table 5.1.

Class	Accuracy
Overall	99.76%
Real	99.78%
Spoof	99.74%

Table 5.1: Accuracy values of the CNN

Precision, otherwise also called the Positive Predictive Value, is the number of correctly identified genuine items among all items classified as genuine. It shows high values with 99.78 percent, meaning that nearly all items classified as positive are actually positive. The Recall (also labelled as the Sensitivity) shows the rate of how many actually genuine samples were identified as such. A low score here, while maintaining high accuracy, could indicate that the network may not identify any fakes as real, but at the same time miss a lot of actual positive data. Luckily, this is not the case either, with another 99.78 percent, the recall is excellent well. Specificity could be named the precision for the fake class, as it measures how many items among all classified fakes were actually fake. With a 99.74 percent rate, the network performs incredibly well here as well.

The False Positive Rate and False Negative Rate are the opposites of Specificity (also called True Negative Rate) and Recall (also called True Positive Rate) and should both be 0 ideally. With 0.26 percent and 0.22 percent, they are rather close to that goal.

The data for these measurements is listed in table 5.2.

Precision	Recall	Specificity	FPR	FNR
99.78%	99.78%	99.74%	0.26%	0.22%

Table 5.2: Precision, Recall, Specificity of the CNN

5.3.3 F1-Score

When the class data is imbalanced, which for the test set it actually slightly is towards the real images, even though overall there are marginally more spoof images, the accuracy score can become non-telling [25]. The higher the imbalance, the less intuitive accuracy becomes and may mislead to a faulty conclusion. Therefore, the F1-Score can be used,

which is the harmonic mean of accuracy and recall. It is also meant as an addition to the other measurements, not a replacement. Lucky enough, that F1-Score tops at 99.78 percent, showing the capabilities of the neural network. The value is also listed in table 5.3.

F1-Score	99.78%

Table 5.3: F1-Score of the CNN

5.4 Comparison to Toprak and Toygar

Hyperparameter search and validation loss are not given in the original paper, therefore they cannot be compared [3]. For the results on the trained model, this is quite different, although direct comparisons are not always possible either. For this project, not all the originally listed data sources were accessible, which lead to the unification of datasets on what was gathered. Furthermore, the spoof data had to be created manually, like Toprak and Toygar had to.

The authors did use False Fake Rate (i.e. False Negative Rate), False Genuine Rate (i.e. False Positive Rate) and HTER, which is a combination of the first two. For more details, see section 2.2. The authors test the CNN on each database they had access to individually, the results are shown in table 5.4. As it can be seen, for three out of five sets it performs pretty decent, with error rates not above one percent. For USTB Set 1, which this project has no access to, six percent for FGR and 4.5 percent HTER is reported. UBEAR is the weakpoint of the original CNN, as it shows an FFR of 69 percent, but to be fair with no fake images classified as spoofs. This is obviously far from being perfect.

Dataset	FFR/FNR	FGR/FPR	HTER
AMI	0.0	1.0	0.5
UBEAR	69.0	0.0	34.5
IITD	1.0	1.0	1.0
USTB 1	3.0	6.0	4.5
USTB 2	1.0	0.0	0.5

Table 5.4: Caption

The CNN in this project has improved on this. By combining the two databases, a far bigger test set is created and also more universal, as different data sources were combined.

On top, different printers and slightly different lighting conditions for re-capturing the images added additional variety to the dataset, even without applying extensive image augmentations. There is no record that this was done in the original paper as well. With an FPR (FGR) an FNR (FFN) of 0.26 percent and 0.22 percent, respectively, the results show a clear advantage for the implemented CNN over the original design.
Toprak and Toygar combine the CNN with image quality measurements, which finally bring the error rates down to zero percent, i.e. a perfect result, indicating that the system is not making any mistakes. This is impressive, but also showing that potential was missed when designing and implementing the network, as it still could have been better, especially when looking at the performance on UBEAR and USTB set 1. While this project could, sadly, not take have a look UBEAR and USTB, the already improved performance, with possibly more variation in the data on top, on other datasets shows the potential.

6 Conclusion and outlook

This final chapter of the project transcript will summarize the main points regarding spoof detection in ear biometrics, the implementation of a convolutional neural network and the results of the implementation. Additionally, some words will be spent aimed towards further research topics that are relevant to improve and/or adapt the current procedure to other areas of application and alternative approaches.

6.1 Conclusion on the CNN

Overall, Convolutional Neural Networks have shown their ability for image classification, in this case extending their capabilities to distinguish between genuine and spoof images which have all fairly similar shapes. The attack scenario was focused on presentation/print attacks, with the real images collected from the AMI and IITD ear datasets. The spoofing had to be done by hand, utilizing two different types of printers and a smartphone camera for re-digitalization. This was primarily described and discussed in chapter 3, sections 3.1 and 3.1.

Another larger point of research was on how to implement, optimize and test a neural network. For this work, PyTorch proved to be a very good solution for bringing the CNN into code-form with ease, with Ray Tune for hyperparameter optimization over said network and finally model import/export to make testing independent of training. A number of measurement options were introduced, including accuracy, precision, recall, specificity and the F1-score, to take unequal class distribution into account. Further on, training and validation accuracy scores were used to stop training and the right point in time to avoid overfitting. More actions to counter this problem include batch normalization and dropout during the training phase. Additionally, image augmentation to create more

variety in the dataset was also applied. Overall, implementation details are discussed in chapter 4, the metrics in section 2.2.

The results for all parts of search, training, validation and testing were pretty satisfying. The hyperparameter search concluded with a reasonable set of values for the learning rate, batch size, and the number of two layers in the fully connected output layers. Training stopped in Epoch 36 (of a maximum of 50), as the validation error did not drop below it all-time minimum value of just 0.9 percent for another four training cycles. Finally, testing also proved the effectiveness of the implemented CNN, concluding in an overall accuracy value of 99.76 percent, precision, recall and F1-score of 99.74 percent and therefore indicating that the model works fine, even with slightly unbalanced test dataset. Also, in comparison to the original paper [3], the values were at minimum on-par or better for the CNN-only part and only marginally worse (close to 0.3 percent) compared to fusion techniques. The training and testing of this work was further done on a dataset containing images from different sources, therefore was more diversified and therefore to be believed that it can better generalize. The detailed discussion is done in chapter 5.

6.2 Outlook

Further fields of application could be to extend the angle of attacks from image presentation attacks to video, or combine face and ear detection into a single system, improving the overall safety of a system [3]. It could, however, also bring with it more chances to successfully attack a system by enlarging the attack surface.

So far, the authentication and spoof detection have been handled as two very separate parts from each other, but it does not have to be that way [5]. Fusing two different CNNs together, where one part does the user authentication and one the spoof detection, both remarkable performance in terms of speed and detection rates have been achieved. The authors were among the selected view who gave any insight to the computational demands of their work. This cannot be said for many other projects, and could also be considered to be done for this network if brought up again. A spoof detection is close to being unpractical when the computational demands are too high to handle in a reasonable amount of time for embedded computer systems at places of authentication.

Apart from including precise time measurements to accurately estimating system demands, an improved number of datasets would help with broadening the chances to detect spoofs. This would involve more diverse data, for example in terms of race, genders, jewellery and maybe also partly covered ears (hair, caps, hoods). Further on, this model may also be part of a fusion decision. More standardized datasets are also a part to make the ear spoof detectors much more comparable with each other, especially in regard to fake image data, where nothing is currently available publicly. Competing against other solutions would be fun to see how efficient the model trained in here actually is and where room for improvement still exists.

Bibliography

[1] Nesli Erdogmus and Sébastien Marcel. "Spoofing Face Recognition With 3D Masks". In: *IEEE Transactions on Information Forensics and Security* 9.7 (2014), pp. 1084–1097. DOI: 10.1109/TIFS.2014.2322255. URL: https://ieeexplore.ieee.org/document/6810829 (cit. on p. 1).

[2] Mina Farmanbar and Önsen Toygar. "Spoof detection on face and palmprint biometrics". In: *Signal Image Video Process.* 11.7 (2017), pp. 1253–1260. DOI: 10.1007/s11760-017-1082-y. URL: https://doi.org/10.1007/s11760-017-1082-y (cit. on p. 1).

[3] Imren Toprak and Önsen Toygar. "Detection of spoofing attacks for ear biometrics through image quality assessment and deep learning". In: *Expert Syst. Appl.* 172 (2021), p. 114600. DOI: 10.1016/j.eswa.2021.114600. URL: https://doi.org/10.1016/j.eswa.2021.114600 (cit. on pp. 1, 2, 5, 7, 8, 37, 40).

[4] Nalini K. Ratha, Jonathan H. Connell, and Ruud M. Bolle. "An analysis of minutiae matching strength". In: *Audio- and Video-Based Biometric Person Authentication, Proceedings of 3rd AVBPA ed.* 2091 (2001), pp. 223–228 (cit. on p. 1).

[5] Ho Yub Jung, Yong Seok Heo, and Soochahn Lee. "Fingerprint Liveness Detection by a Template-Probe Convolutional Neural Network". In: *IEEE Access* 7 (2019), pp. 118986–118993. DOI: 10.1109/ACCESS.2019.2936890. URL: https://doi.org/10.1109/ACCESS.2019.2936890 (cit. on pp. 1, 40).

[6] Syed Farooq Ali, Muhammad Aamir Khan, and Ahmed Sohail Aslam. "Fingerprint matching, spoof and liveness detection: classification and literature review". In: *Frontiers Comput. Sci.* 15.1 (2021), p. 151310. DOI: 10.1007/s11704-020-9236-4. URL: https://doi.org/10.1007/s11704-020-9236-4 (cit. on pp. 1, 2).

[7] Tarang Chugh, Kai Cao, and Anil K. Jain. "Fingerprint Spoof Buster: Use of Minutiae-Centered Patches". In: *IEEE Trans. Inf. Forensics Secur.* 13.9 (2018), pp. 2190–2202. DOI: 10.1109/TIFS.2018.2812193. URL: https://doi.org/10.1109/TIFS.2018.2812193 (cit. on p. 2).

[8] M. Hassaballah, Hammam A. Alshazly, and Abdelmgeid A. Ali. "Ear recognition using local binary patterns: A comparative experimental study". In: *Expert Systems with Applications* 118 (2019), pp. 182–200. ISSN: 0957-4174. DOI: https://doi.org/10.1016/j.eswa.2018.10.007. URL: https://www.sciencedirect.com/science/article/pii/S0957417418306493 (cit. on p. 2).

[9] Önsen Toygar, Esraa Alqaralleh, and Ayman Afaneh. "Symmetric ear and profile face fusion for identical twins and non-twins recognition". In: *Signal Image Video Process.* 12.6 (2018), pp. 1157–1164. DOI: 10.1007/s11760-018-1263-3. URL: https://doi.org/10.1007/s11760-018-1263-3 (cit. on p. 2).

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet classification with deep convolutional neural networks". In: *Commun. ACM* 60.6 (2017), pp. 84–90. DOI: 10.1145/3065386. URL: https://doi.org/10.1145/3065386 (cit. on p. 4).

[11] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25: 26th Annual Conference on Neural Information Processing Systems 2012. Proceedings of a meeting held December 3-6, 2012, Lake Tahoe, Nevada, United States*. Ed. by Peter L. Bartlett et al. 2012, pp. 1106–1114. URL: https://proceedings.neurips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html (cit. on p. 4).

[12] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. "Deep learning". In: *Nat.* 521.7553 (2015), pp. 436–444. DOI: 10.1038/nature14539. URL: https://doi.org/10.1038/nature14539 (cit. on p. 4).

[13] Shuihua Wang and Yi Chen. "Fruit category classification via an eight-layer convolutional neural network with parametric rectified linear unit and dropout technique". In: *Multim. Tools Appl.* 79.21-22 (2020), pp. 15117–15133. DOI: 10.1007/s11042-018-6661-6. URL: https://doi.org/10.1007/s11042-018-6661-6 (cit. on p. 5).

[14] Peng Zhou et al. "Text Classification Improved by Integrating Bidirectional LSTM with Two-dimensional Max Pooling". In: *COLING 2016, 26th International Conference on Computational Linguistics, Proceedings of the Conference: Technical Papers, December*

11-16, 2016, Osaka, Japan. Ed. by Nicoletta Calzolari, Yuji Matsumoto, and Rashmi Prasad. ACL, 2016, pp. 3485–3495. URL: https://aclanthology.org/C16-1329/ (cit. on p. 6).

[15] Jiuxiang Gu et al. "Recent advances in convolutional neural networks". In: *Pattern Recognition* 77 (2018), pp. 354–377. ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2017.10.013. URL: https://www.sciencedirect.com/science/article/pii/S0031320317304120 (cit. on p. 7).

[16] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015*. Ed. by Francis R. Bach and David M. Blei. Vol. 37. JMLR Workshop and Conference Proceedings. JMLR.org, 2015, pp. 448–456. URL: http://proceedings.mlr.press/v37/ioffe15.html (cit. on p. 7).

[17] Haibing Wu and Xiaodong Gu. "Towards dropout training for convolutional neural networks". In: *Neural Networks* 71 (Nov. 2015), pp. 1–10. DOI: 10.1016/j.neunet.2015.07.007. URL: https://doi.org/10.1016%5C%2Fj.neunet.2015.07.007 (cit. on p. 7).

[18] *Understanding the Confusion Matrix and How to Implement it in Python*. May 1, 2020. URL: https://towardsdatascience.com/understanding-the-confusion-matrix-and-how-to-implement-it-in-python-319202e0fe4d (visited on 06/02/2023) (cit. on p. 9).

[19] *AMI Ear Database*. URL: http://www.ctim.es/research_works/ami_ear_database (visited on 05/25/2023) (cit. on p. 11).

[20] *UBEAR: A Dataset of Ear Images Captured On-the-move in Uncontrolled Conditions*. Jan. 18, 2011. URL: http://ubear.di.ubi.pt/ (visited on 05/25/2023) (cit. on p. 11).

[21] *IIT Delhi Eat Database*. URL: https://www4.comp.polyu.edu.hk/~csajaykr/IITD/Database_Ear.htm (visited on 05/25/2023) (cit. on p. 11).

[22] Rui Raposo et al. "UBEAR: A dataset of ear images captured on-the-move in uncontrolled conditions". In: *2011 IEEE Workshop on Computational Intelligence in Biometrics and Identity Management (CIBIM)*. 2011, pp. 84–90. DOI: 10.1109/CIBIM.2011.5949208 (cit. on p. 11).

[23] Luca Ghiani et al. "Review of the Fingerprint Liveness Detection (LivDet) competition series: 2009 to 2015". In: *Image Vis. Comput.* 58 (2017), pp. 110–128. DOI: 10.1016/j.imavis.2016.07.002. URL: https://doi.org/10.1016/j.imavis.2016.07.002 (cit. on p. 13).

[24] Connor Shorten and Taghi M. Khoshgoftaar. "A survey on Image Data Augmentation for Deep Learning". In: *J. Big Data* 6 (2019), p. 60. DOI: 10.1186/s40537-019-0197-0. URL: https://doi.org/10.1186/s40537-019-0197-0 (cit. on pp. 14, 15).

[25] *The F1 score*. Aug. 31, 2021. URL: https://towardsdatascience.com/the-f1-score-bec2bbc38aa6 (visited on 06/04/2023) (cit. on p. 36).