ANTI-SPOOFING SYSTEM

A project made by:
Brugnera Matteo
Depedri Kevin

Introduction

On this project we are going to analyze and create a possible anti-spoofing system that can be used in many applications.
There are multiple types of **Presentations Attacks** (PA) that can be used to spoof a **Facial Recognition system** and they are all based on the same idea: *showing a representation of the original face to the camera, which will spoof and give access to the system if the representation quality is high enough*.

The various PAs for facial recognition system all belong to one of the following categories:

Printed presentation attack which can be a simple print or something more elaborated like a printed cheap face mask

Replay presentation attack which is based on showing a video played by another device to the PAD system

Silicon masks presentation attack (we will exclude this since it can be successfully performed only in a collaborative environment where the target of the spoof will need to provide its facial information to manufacture the silicon mask realistically).

Given the previous observations we need a system that is able to detect printed and replay presentation attacks, and classify them as such. The best way to do so is by exploiting their common weaknesses such as the *limited color gamut* and the *double acquisition*, which derives from the fact that it is a re-captured image.

Both these presentation attacks are based on showing a media where the spoofing object is present to the camera, but both ways (that can be done through a printed photo or an external device) have a limited *color gamut* with respect to the visible/true colors.

Furthermore, these media will be subjected to a *double acquisition* (the first one when they are created, and the second one when they will be shown to the system) and this will lead to a *greater quantity of noise* and to a *lower general quality* of the image and of the colors, which we aim to detect through the use of different operators.

In order to obtain such system, we will implement different methods which are able to deal with both printed and replay presentation attacks. In the first case we will check whether the subject in front of the camera blinks or not, in this way the system recognizes if it's dealing with a static image or a real person. As we will see, this method doesn't distinguish between a real person and a video of the very same person. So, in order to solve such problem, the system will capture a single frame of what it has in front and will apply some transformations that, together with an SVM, will allow the system to detect whether it's a real person or a video.

Eye blinking

Many methods have been proposed to automatically detect eye blinks in a video sequence. The existing methods can be either *active* or *passive*. In the first case the system is more robust and reliable but there is the need to use special hardware (often expensive and intrusive) in order to obtain the expected result.

The passive systems, on the other hand, rely only on a standard remote camera.
The traditional processing method used for detecting blinks typically involves:

- Eye localization
- Thresholding to find the white part of the eye
- Calculating whether the white region of the eye disappears for a certain period of time, which indicates that a possible blink has occured

Real-Time Eye Blink Detection using Facial Landmarks

Instead of using the previously cited method, we built the blink detection system by computing a metric called *eye aspect ratio* (also known as **EAR**), a process introduced by *Soukupová and Čech*.

This solution is much more elegant and involves simple calculation based on the ratio of distances between facial landmarks located around the eye. It is also *fast*, *easy* to implement and very *efficient*.

This metric is a single scalar quantity that tells us whether the eye is open or not.

A major drawback of the this approach is that it imposes *too strong requirements* on the setup, e.g. image resolution, illumination, motion dynamics and partial head pose sensi in order to track correctly the eye.

However nowadays we can usee robust real-time facial landmark detectors that capture most of the characteristic point on a human face image.

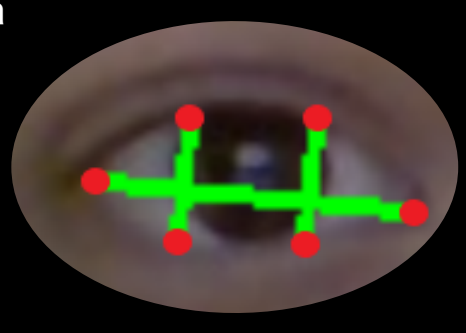
Understanding the EAR

An eye blink can be seen as a fast closing and re-opening of the human eye. Even if we all do the same exact action 15-20 times per minute, the patterns can differ in the speed of closing and opening, in the degree of the squeezing and in the blink duration. In fact, the eye blink lasts approximately 100-400 ms. Since this movement is extremely quick, we adopted the following process in order to detect it:

01

Step 01

Each eye is represented by 6 points starting from the left corner of the eye and then going around it clockwise. Each one of them is described by a pair of coordinated *(x,y)*.



02

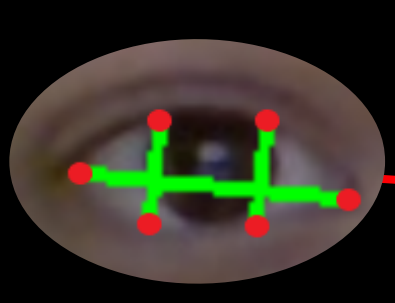
Step 02

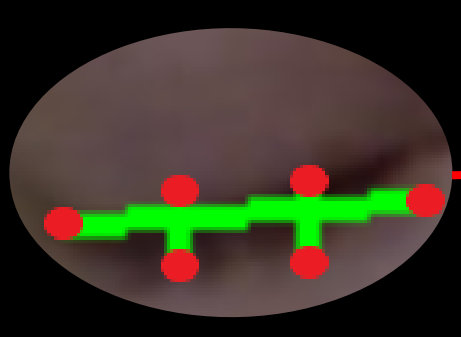
From the landmarks detected in the image, we derive the eye aspect ratio that is used as an estimate of the eye opening.

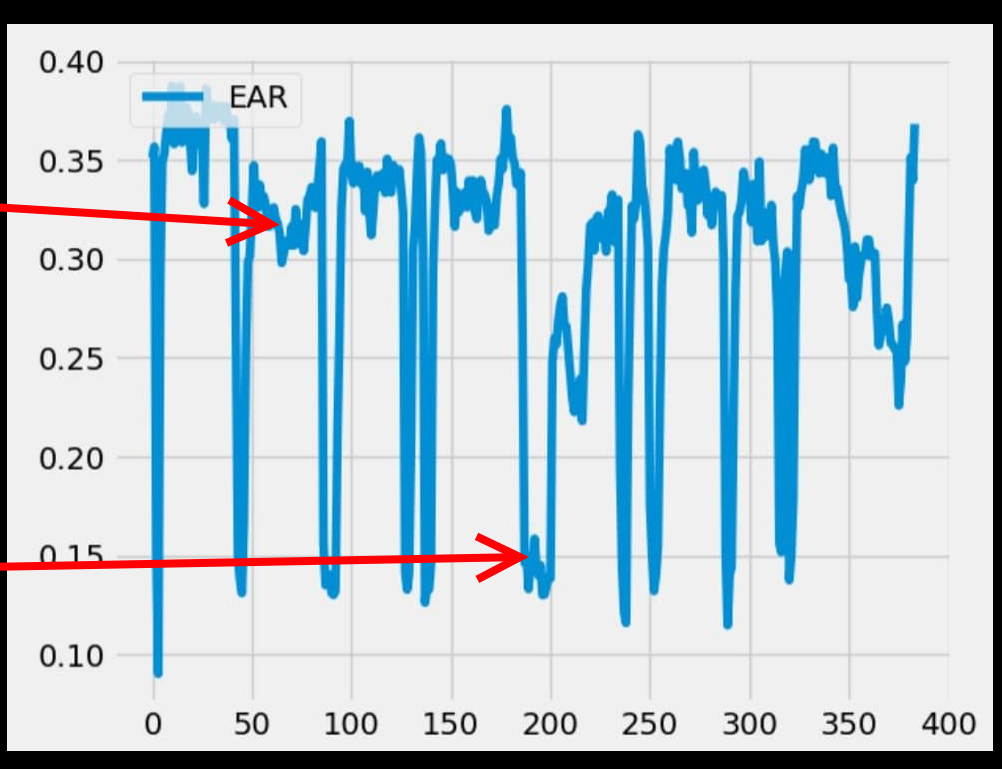
$$\mathrm{EAR} = \frac{\|p_2 - p_6\| + \|p_3 - p_5\|}{2\|p_1 - p_4\|}$$

where p1,...,p6 are the landmarks locations

The numerator computes the distance between the vertical eye landmarks while the denominator compues the distance between the horizontal eye landmarks, weighting it since there is only one horizontal line and two vertical lines.
The value EAR is almost constant when the eye is open and it goes to zero when the eye closes. Since eye blinking is performed by both eyes simultaneously, the EAR of both eyes is averaged.







Code

The following sections represent the most important parts of the code for observing the blinks:

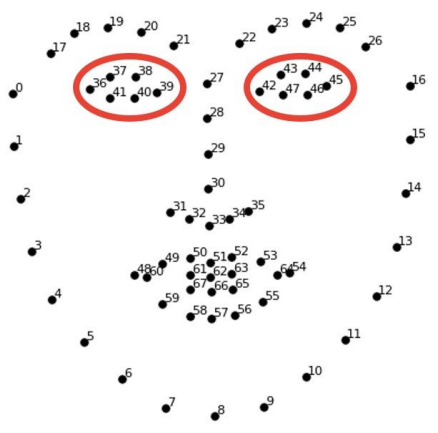
1

```
left_point = (facial_landmarks.part(eye_points[0]).x, facial_landmarks.part(eye_points[0]).y)
right_point = (facial_landmarks.part(eye_points[3]).x, facial_landmarks.part(eye_points[3]).y)
hor_distance = dist.euclidean(left_point, right_point)

top_left = (facial_landmarks.part(eye_points[1]).x, facial_landmarks.part(eye_points[1]).y)
bottom_left = (facial_landmarks.part(eye_points[5]).x, facial_landmarks.part(eye_points[5]).y)
ver_distance_left = dist.euclidean(top_left, bottom_left)

top_right = (facial_landmarks.part(eye_points[2]).x, facial_landmarks.part(eye_points[2]).y)
bottom_right = (facial_landmarks.part(eye_points[4]).x, facial_landmarks.part(eye_points[4]).y)
ver_distance_right = dist.euclidean(top_right, bottom_right)
```

In the first portion of the code we define all the necessary elements to compute the EAR value.
We use *facial_landmarks* (a pre-trained face detector) in order to locate all the useful points around the eyes.
By using these indexes we'll be able to extract eye regiorns effortlessly



2

Here we initialize four important variables. They represent the conditions that have to be met in order to exit the infinite loop where the blinking detection is executed (see next section). The first two are boolean variables that indicate whether a closed/opened eye has been detected. The last two define the maximum period of time in which the detection is performed (time limit of 30 seconds).

```
eye_open = False
eye_closed = False

#time limit of 30 seconds
start_time = int(time.time())
finish_time = int(time.time()) + 30
```

3

```
#either the 30 seconds pass or both eye_open and eye_closed have been detected
while (start_time != finish_time) and not(eye_open and eye_closed):
```

In this section we take a look at the most important part of the code where we execute the procedure to detect the blink of the eye.
We do this until both the previously mentioned boolean variables are *true* or the time limit of 30 seconds occurs.

```
for face in faces:
    landmarks = predictor(gray, face)

    left_eye_ratio = get_blinking_ear([36, 37, 38, 39, 40, 41], landmarks, frame)
    right_eye_ratio = get_blinking_ear([42, 43, 44, 45, 46, 47], landmarks, frame)
    blinking_ratio = (left_eye_ratio + right_eye_ratio) / 2
    x_value = blinking_ratio

    if blinking_ratio > eye_threshold:
        eye_open = True
```

For every face that we detected so far, we compute the EAR.
If this value is greater then a predetermined threshold, then it means that the eye is open, so we put the variable *eye_open* to true.

Whenever we detect that the EAR value is below the threshold, we know for sure that the eye is closed, so we change the value of the parameter *eye_closed* to true.

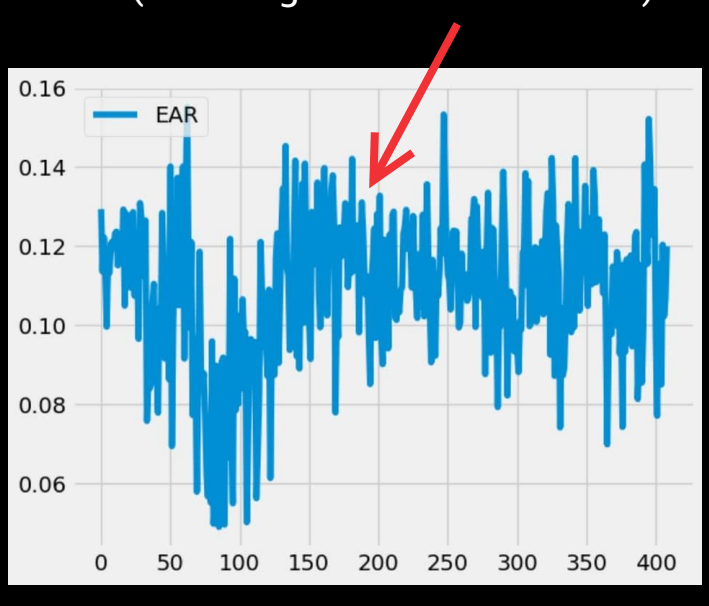
```
if blinking_ratio < eye_threshold:
    cv2.putText(frame, "BLINKING", (50, 50), font, 2, (0, 0, 255))
    eye_closed = True
```

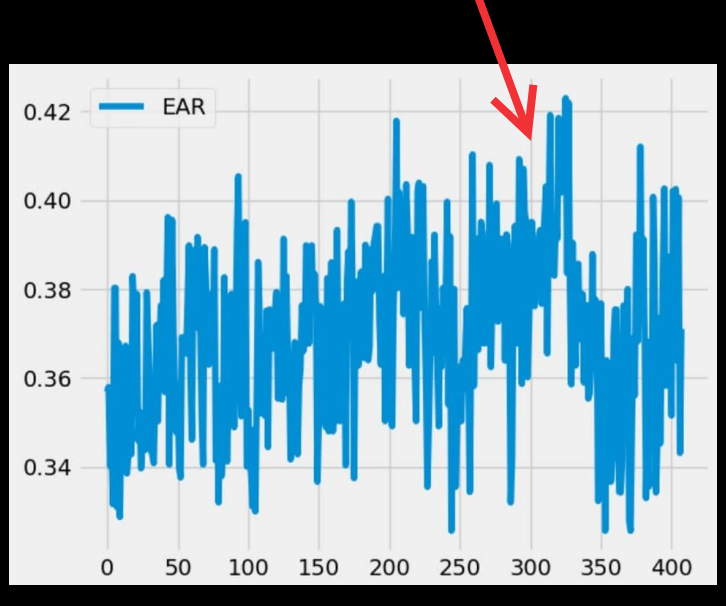
To sum up, the possible situations that we can encounter are:

1 both boolean parameters are *true*. This means that a blink has been detected and thus we exit from the while loop.

2 after 30 seconds only the variable *eye_open* is true. In this case all the EAR values obtained in this period of time will not go below the threshold value 0.2 (in average they are between 0.35 and 0.40). This allows us to tell that a fake face has been detected.

3 the same goes when only the variable *eye_closed* is true. All the EAR values will be below the 0.2 threshold (in average between 0 and 0.1).





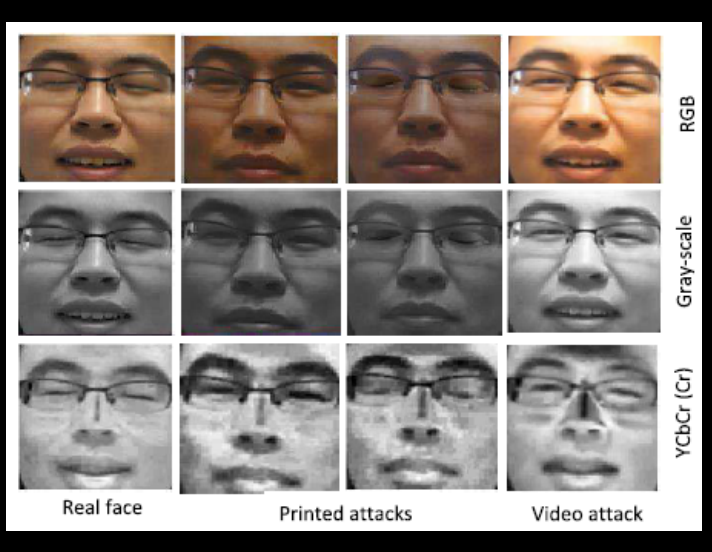
Feature Extraction

After that a face has been framed and after receiving a positive feedback from the eye blinking detection system, we proceed with the feature extraction phase. We will adopt different techniques and operators with the goal of extracting data from the picture of a face. Later these data will be used to determine if the person in front of the camera is a real person or someone else who is trying to spoof the system.

Since our Presentation Attack Detection (PAD) system is completely based on cameras, the only data that we can exploit are pictures. Thus, we adopt three operators to perform feature extraction, which allow us to obtain three different texture descriptors. These are:

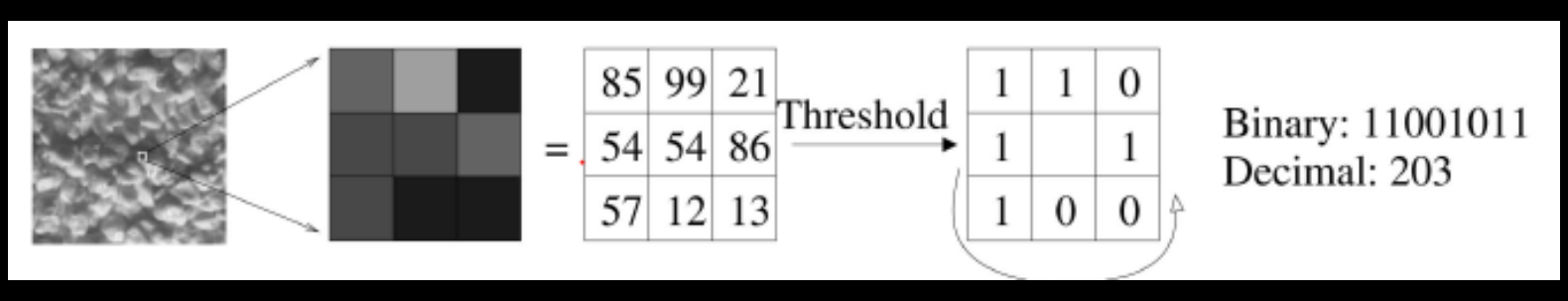
- **LBP** (Local Binary Patterns)
- **CoALBP** (Co-occurrence of Adjacency Local Binary Patterns)
- **LPQ** (Local Phase Quantization).

These three operators are applied on all the channels of the YCrCb and HSV color spaces, which have been chosen since they allow us to work with chrominance information.



Chrominance is the key component to differentiate a PA from a real access attempt. Infact, the color reproduction obtained on different media (such as on printed paper or on digital display) focuses on preserving the luminance component (which is more important for the human eye) at the expenses of the chrominance, which is subjected to imperfections with respect to the original photo. Thanks to this, we will be able to spot and define the presentation attack.

The first operator to be applied is LBP (Local Binary Patterns) which is a highly discriminative texture descriptor. It produces a binary code for each pixel of an image, which is computed by thresholding a circularly symmetric neighborhood with the value of the central pixel. Then the binary codes are converted to decimal and represented through a histogram.
Since LBP considers only the magnitude relation between the center and neighboring pixel intensities, LBP is invariant to uniform changes of image intensity over the entire image, making it robust against changes in illumination.



In applying this operator, we adopt a special procedure, which consists in dividing the image in regions through a grid, computing the LBP for each region separately, creating the separated histograms and then concatenating them all in a unique spatially enhanced histogram. In this way we are computing several local descriptors of different face zones, which are then combined to engender a global descriptor, which leads us to two major advantages:

- more **robustness** against variations in pose or illuminance.
- more **specificity**. Since texture descriptors tend to average over the image area, working on smaller regions allow us to retain the information about spatial relations.

This approach leads us to work on three different level of locality: the LBP labels are computed on a pixel level information, then the LBP labels are summed up in each region histogram creating a local level information, and in the end the different histograms are concatenated composing a global level information and giving us our first texture descriptor.

In our system we adopted an LBP with a radius of 1 which generates 8bit binary sequences. This gave us single histograms with 256 bins for each region on which the operator has been applied. Here, we decided to use a grid 3 x 3, which led to a total of 9 regions. Concatenating the histograms of the nine regions led to a final histogram with 2304 bins, which is our first texture descriptor.

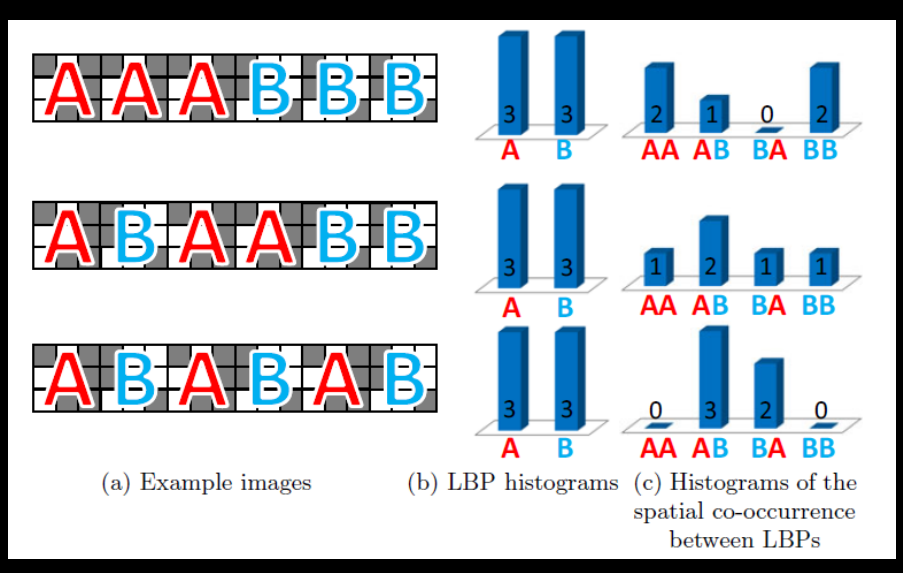
CoALBP (Co-occurrence of Adjacency Local Binary Patterns)

The second operator to be applied is CoALBP(Co-occurrence of Adjacency Local Binary Patterns), which is a variant of LBP that has been created to exploit also the spatial relations between LBPs.

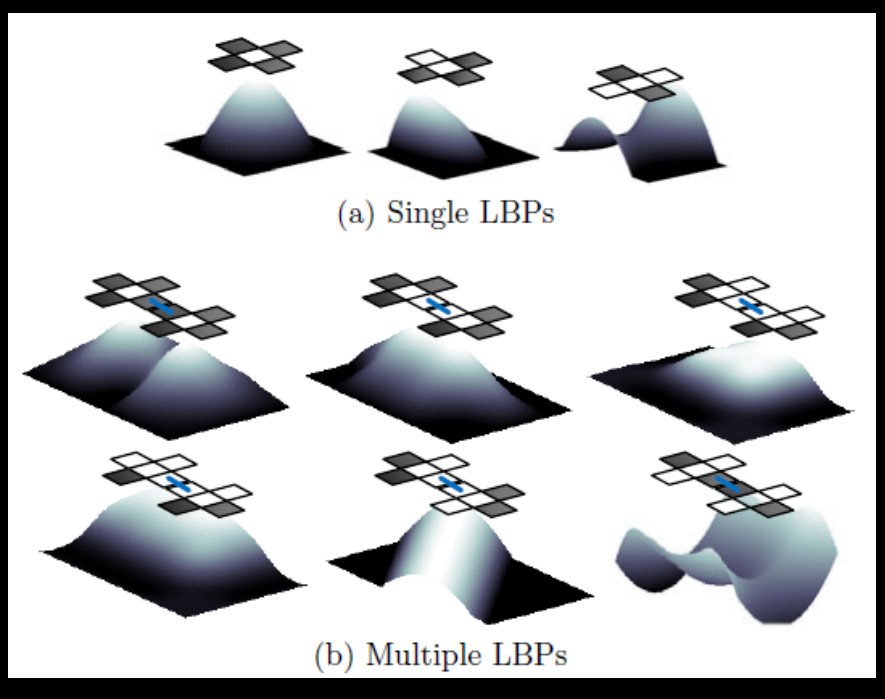
As we know, LBP was originally designed as a texture descriptor for a local region, called micropattern, and it wasn't supposed to gather also the spatial information (which are mostly discarded during the LBP histogram generation process since the LBPs are forcedly packed into a single histogram, resulting in the loss of global image information.

To consider the spatial relation among LBPs, we introduce the concept of **co-occurrence**. A sophisticated way to get the co-occurrences of all combinations of LBPs is by using **auto-correlation matrices** calculated from two considered LBPs.

From the picture on the right, we can see that the expression ability of the original LBP is insufficient, and the spatial co-occurrence of LBPs is a valid requirement for realizing a higher expression ability.



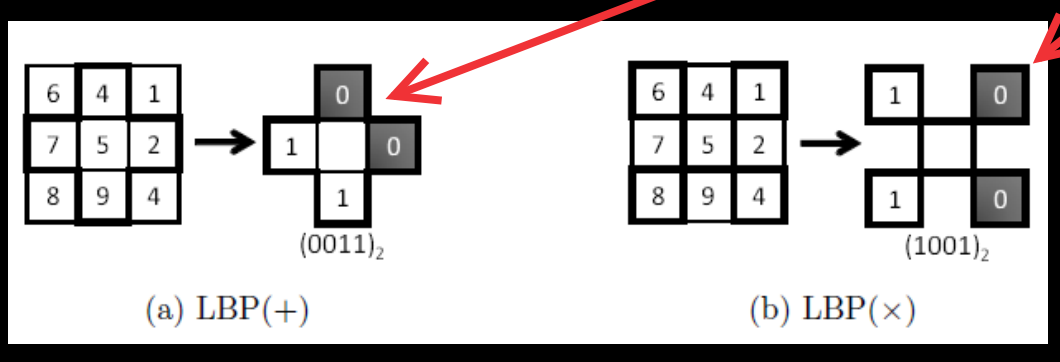
(a) Example images (b) LBP histograms (c) Histograms of the spatial co-occurrence between LBPs


(a) Single LBPs

(b) Multiple LBPs

The co-occurrence of adjacent LBPs is defined as an index of how often their combination occurs in the whole image. The original LBP can represent only a simple image pattern, while the combination of multiple LBPs can represent various image patterns derived from more complicated surfaces.

We introduce an auto-correlation matrix as an effective method of calculating the co-occurrence of LBPs. First, although the original LBP uses eight neighbor pixels of a given center pixel, we modify the LBP configuration to consider two sparser configurations, thereby reducing computational cost. One configuration is LBP (+), which considers only two horizontal and two vertical pixels, the other configuration is the LBP (x), which considers the four diagonal pixels.



(a) LBP(+) (b) LBP(×)

The process flow for the proposed feature is the following:

01

Step 01

Firstly, LBPs are extracted from the input image.

02

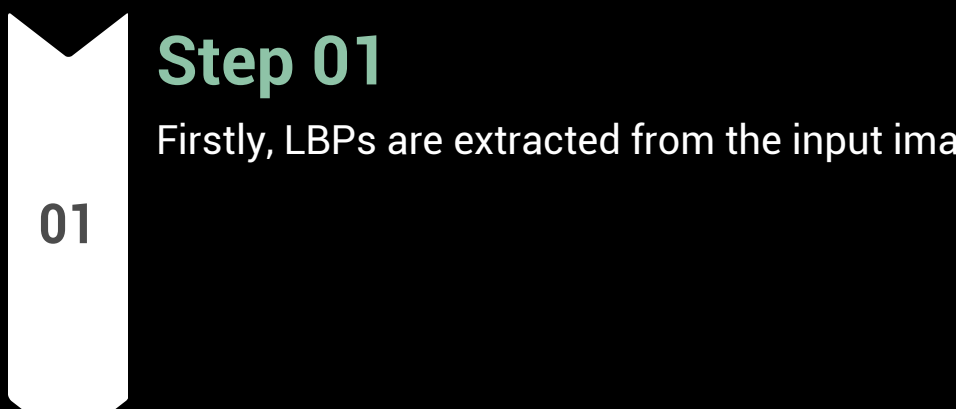
Step 02

Next, we compute four Np×Np (where Np = 2^Nn represents all the possible LBPs, and Nn = 4 in the new sparser LBPs, which are LBP+ and LBPx) auto-correlation matrices of spatial co-occurrences of adjacent LBPs, which represents respectively the four possible spatial relations: (Δr, 0), (Δr, Δr), (0, Δr),(−Δr, Δr) present in this model.

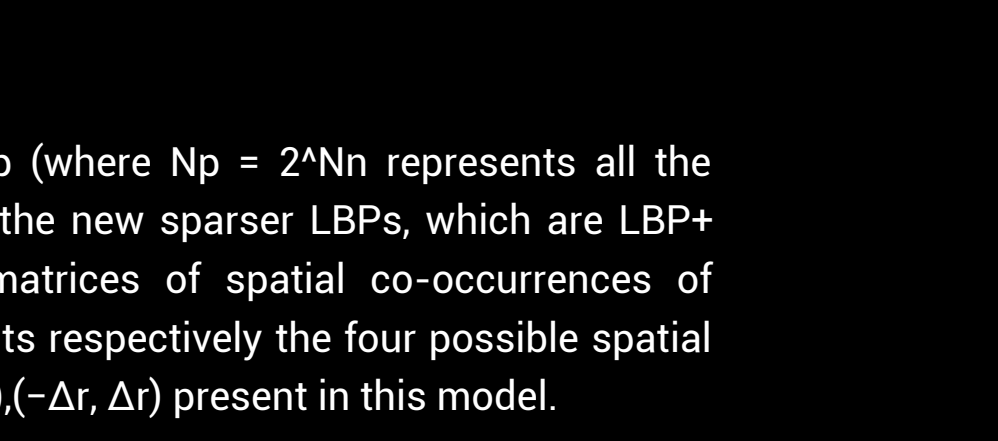
03

Step 03

Finally, these matrices are vectorized and combined to a 4*(Np^2) dimensional feature vector (or 1024bins histogram).



(a) Example image (b) Auto-correlation matrices of co-occurrence of LBPs, $H(a)$



(a) Extracting LBPs from the image (b) Auto-correlation matrices of co-occurrence of LBPs, $H(a)$ (c) Combined feature z

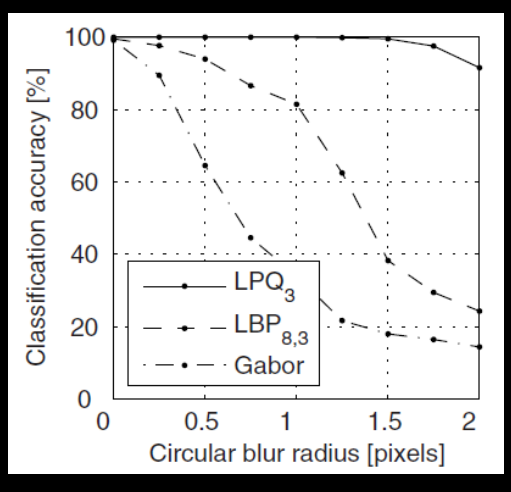
In our system we decided to compute this operator three times for each color channel (using a different input configuration for each iteration, such that the radius for adjacent LBP and the radius for co-occurrence of the pattern (Δr) were respectively (1,2), (2,4) and (4,8) in the 3 runs) to achieve a higher level of accuracy. Each computation leads to an independent histogram, then the three histograms are concatenated between them to create our second texture descriptor which will be represented by a final histogram with 3072 bins.

LPQ (Local Phase Quantization)

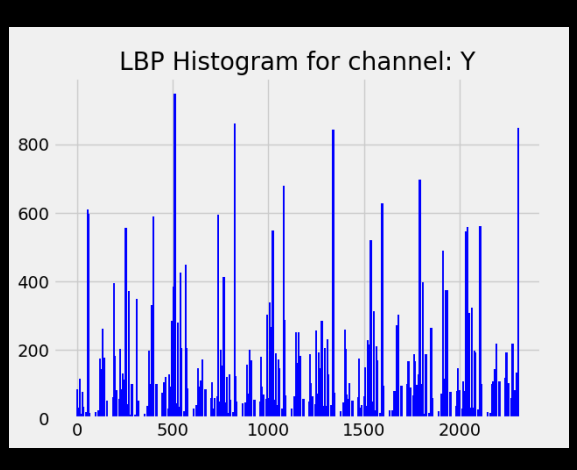
The third operator to be applied is LPQ(Local Phase Quantization), which is a blur insensitive texture descriptor. We know that in the real world, image degradations may limit the applicability of the texture information. One important class of degradation is blur due to motion, out of focus, or atmospheric turbulence. But because image deblurring is very difficult and introduces new artifacts, it is desirable to be able to analyze texture in a way that is insensitive to blur.

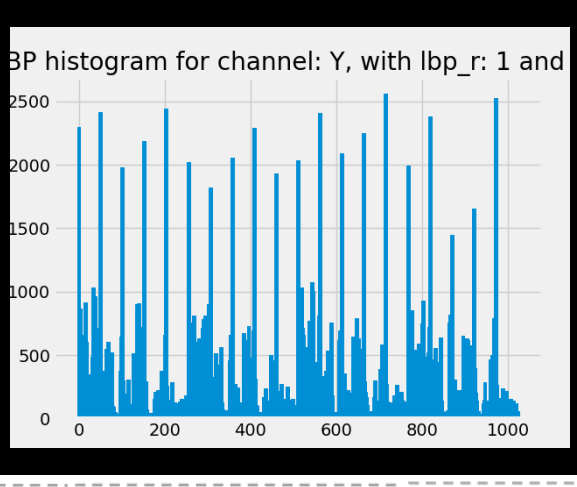
This can be done through LPQ operator, which is based on the quantized phase of the discrete Fourier transform (DFT), which is computed in local image windows. The codes produced by the LPQ operator are insensitive to centrally symmetric blur, which includes motion, out of focus, and atmospheric turbulence blur. The LPQ operator is applied to texture identification by computing it locally at every pixel location and presenting the resulting codes as a histogram. Generation of the codes and their histograms is similar to the LBP method, but it is based on more complex concept, such as the blur invariance property of the Fourier phase spectrum. For that property we can create the pixel codes using the local phase information extracted from the 2-D DFT (or, more precisely, using a short-term Fourier transform, STFT), that is computed over a rectangular neighborhood at each pixel position x of the image.

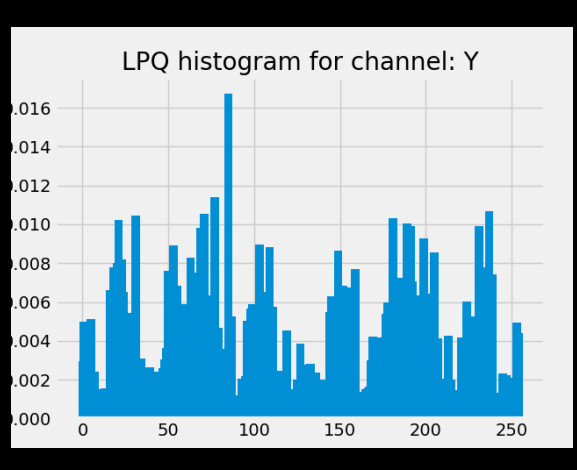
Finally, a histogram of these integer values from all image positions is composed. The resulting integers are invariant to centrally symmetric blur provided that the window used is infinitely large. However, this condition cannot be fulfilled in practice, and therefore, complete invariance is not achieved, but as shown in the experiments, even a relatively small neighborhood is enough for robustness to reasonable extents of blur.

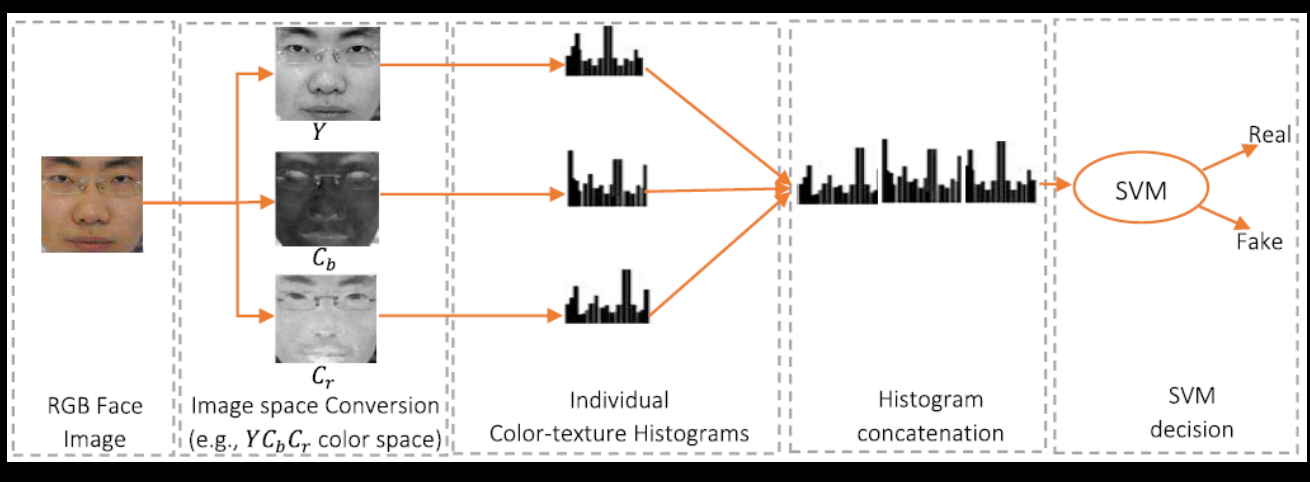


In our PAD this operator is computed once for each color channel leading to a histogram with 256bins which represents our third texture descriptor.
The three texture descriptors that have been introduced previously has been computed and concatenated in sequence for each channel of the YCrCb and HSV color spaces, leading to a total of six individual color-texture histograms (for the six color channels: Y, Cr, Cb, H, S, V) with 5632 bins. Then these six histograms have been concatenated between them following the ordering: Y, Cr, Cb, H, S, V, leading to our final descriptor for the input image, which can be represented through a histogram with 33792 bins.









The final descriptor that we obtain will then been given as input into a SVM to compare it with the registered data of the user, and to determine if it is a real face, or a fake face (or PA).

SVM MODELING AND PREDICTION: CODE

Once we know how to obtain the final descriptor for a given image, we can setup and train a SVM such that the final system is able to detect whether an input image is the result of a presentation attack or it's a real person.
We can divide this whole process in 2 main steps:

1

```
dir = 'C:\\Users\\matte\\OneDrive\\Desktop\\SIV_Project\\images dataset'

categories = ['True', 'Fake']
dataset = []

for category in categories:
    path = os.path.join(dir, category)
    label = categories.index(category) # I'll get the indexes of the categories -> 0 for true and 1 for fake

    for img in os.listdir(path):
        imgpath = os.path.join(path, img)
        print(imgpath)
        face_image = cv2.imread(imgpath, 0)
        try:
            face_image = cv2.resize(face_image, (30, 30))
            image = full_histogram.final_function(imgpath)
            dataset.append([image, label])
        except Exception as e:
            pass

pick_in = open('data1.pickle', 'wb')
pickle.dump(dataset, pick_in)
pick_in.close()
```

In this first step, we are going to compute the full histogram for every single image that we have in our database.
After that we associate every image with its corresponding label that can be either 0 (True) or 1 (Fake). At the end we use pickle in order to save the dataset that we obtained such that we don't need to run this portion of code to use the dataset in the following steps.

2

After that, the first thing that we do is load the dataset by using the specific pickle function.
Then we divide these informations into input samples and their labels.
Then we train the model by using a linear support vector classifier.
At the end we save the obtained model using pickle as we did before. Thanks to this we are able to use the model to make predictions without having to train it every single time.

```
#open the dataset from the pickle file
dataset = []
pick_in = open('data1.pickle', 'rb')
dataset = pickle.load(pick_in)
pick_in.close()

#create an array composed by all the input samples and one with all the corresponding labels
input_sample = []
labels = []

for sample, label in dataset:
    input_sample.append(sample)
    labels.append(label)

#train the model with a linear SVC
X_train, X_test, y_train, y_test = train_test_split(input_sample, labels, test_size=1)

model = LinearSVC(C=100.0, random_state=42)
model.fit(X_train, y_train)

#save the trained model in order to use it without training it every single time
pick = open('model.sav', 'wb')
pickle.dump(model, pick)
pick.close()
```

Conclusions

Up to this point we saw the main technical features related to the project and the steps that the algorithm performs from the eye-blinking to the final computation of the features and classification.
But it is also really important to analyze other aspects that can determine the validity, correctness and robustness of the algorithm.

A really important aspect that we have to take into consideration is the **accuracy** of the overall system.
Before commenting the results that we can get from the anti-spoofing set-up, we have to make some considerations related to the limitations that affect this system.

First of all the system is subject to the variations in brightness. As a matter of fact, it is really hard to both detect the face (in particular the eyes) for the eye-blinking part and to identify the face through the use of the SVM in tough light conditions.

This problem can be partially solved by increasing the size of the dataset. By doing this the classification part of the system will become more robust.
On the other hand, the computation of the histograms and the training of a single image takes around 3-4 minutes. So if we have a lot of images on our dataset, the overall training part requires a huge period time (for instance, our dataset is composed by 42 images and it took around 3 hours to train).

The best solution in this case is to implement the system in a static environment, where there are no variations of the brightness and there is no light coming from the outside that could hinder the whole system.
In fact, we decided to use this set-up and the results that we got are pretty satisfying:

	Training_Test split	Accuracy obtained
Case 1	0.8	90%
Case 2	0.5	75%
Case 3	0.3	66%

Another big problem is that the time needed to recognize whether the face is fake or not is around 3-4 minutes. Due to this problem, the system cannot be used in a real-time application.

To solve this problem (and also all the previous ones) the SVM can be substituted with a neural network. Since the dimensionality of the feature space is too high (around 34.000), the usage of a NN will increase the speed of the overall system.