Project

Team members

- Akshay Sarode
- Ashray Malhotra
- Rohan Prinja

Paper

The paper we have set out to implement is 'Computationally Efficient Face Spoofing Detection with Motion Magnification'. It describes a method to determine if a video presented to a biometric identification system is of a person actually standing in front of the system, or is a spoof video.

This paper relies heavily on a previous paper 'Eulerian Video Magnification For Revealing Subtle Changes in the World', which describes a way to exaggerate small-scale motions in a video, like the rise and fall of a baby's chest as it breathes while asleep, or the swaying of branches of a tree under a light breeze. The algorithm takes as input a video and outputs another video in which these small motions are enhanced and made more noticeable to the human eye.

Project Abstract

The aim of this project is to develop a classifier which takes as input a video and tells us whether the video is of a real person or is a video made by someone trying to trick the system.

In this project, we concentrate on replay attacks and print attacks. In a replay attack, a video of a person is used as a spoof of the actual person. In a print attack, a snapshot of the person is presented to the identification system. The key insight in this paper is that print attacks lack liveness information (for example, rapid eye movements, twitching of facial muscles in particular cheeks, blinking) and replay attacks lack texture information (for example, the difference in lighting of a video, which is displayed on a flat screen, vs the illumination of a real person). The method proposed by this paper is to use a combination of SVMs to classify an input video as 'spoof' or 'real' by looking at two feature vectors that respectively capture liveness and texture information.

Results

We implemented the video magnification paper entirely, and part of the anti-spoofing paper before running out of time. Some of our results can be seen in our project's Dropbox folder, in the 'output/' folder. You can also run our code, as described below, for other inputs.

Running Our Code

To run our code, please run the shell script `magnify-video.sh` as follows:

```
./magnify-video.sh <filter_type> <video_name> <magnification_factor>
```

The last parameter is optional. If left out, it defaults to 2. `filter_type` is either 'iir' (Infinite Impulse Response) or 'butter' (Butterworth). `video_name` is any one of the videos in the 'input' folder. For example, a run of our code might look like this:

```
./magnify-video.sh iir baby 4
```

This produces an output in the folder 'output/iir/' named 'baby-4.mp4'.

Technical Details

Pre-processing: motion magnification

Before forming the feature vector, we motion-magnify the video so as to exaggerate small temporal differences like eye blinking, or small smiles. This is not strictly necessary but having exaggerated video motions will make the job of the SVMs much simpler.

Eulerian motion magnification

The idea behind motion magnification is to extract out subtle changes in a time-dependent data set and amplify them. To begin with, the data set can be considered as videos (2D data at each temporal resolution) but scope of the project can be modified to deal with different dimensional datasets at each time instant. We also plan to build custom views for Video magnification in VTK. We extract out temporal and spatial frequencies from the given data and amplify specific frequencies according to our use case.

Below we have explained the significant steps of the algorithm. * We start by considering each of the frame of the video independently for analysis * We choose a suitable colour space in which we want to work, This could depend on the specific application that we are dealing with, though in the paper, authors have used NTSC color space for further operations.

```
frame = rgb2ntsc(rgbframe);
```

- For each of the color level(or spectrum level for hyper spectral images), we build a Laplacian pyramid. Note the Laplacian Pyramid is built of the NTSC image, not the RGB image.

```
[pyr,pind] = buildLaplacianPyramid(frame(:,:,1))
```

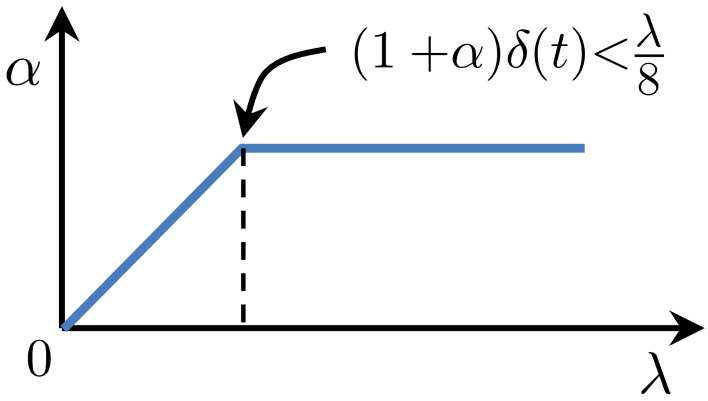
- We initialise the lowpass filter to have the Laplacian Pyramid values. Later, we will change the values of the filter limits to perform temporal filtering of the signal.
- We consider the next frame, and perform the similar laplacian pyramid calculation on it(after converting in NTSC colour space).
- The value of laplacian pyramids of subsequent frames is used to perform the temporal filtering of the signal. The exact method of temporal filtering could vary with application, for ex. we could use a butterworth filter, an IIR filter etc. For an IIR filter, we perform a simple multiplicative update of both the filter threshold

```
lowpass = (1-const_factor)*lowpass + const_factor*pyramid;
```

- The difference of the computed thresholds gives us the range of frequencies to work with(magnify or suppress, based on the laplacian pyramid level)

```
filtered = (cutoff1 - cutoff2);
```

- Now we have performed the temporal selection of the signal. We will selectively perform the spatial magnification of the signal. Note that the equation that we will use to magnify the spatial frequencies(amount of magnification of a specific spatial frequency), can vary across different use cases, but in this paper, the authors have used linearly increasing magnification with spatial wavelengths with a specific threshold after which the magnification remains constant.



- The above figure gives us the magnification value(Y axis) at each spatial wavelength(X axis) level(which is given by the pyramid index). We multiply the filtered signal above by the appropriate multiplication(or magnification) factor to get the modified filter values. Note that we will have to do this for all the images in the pyramid.
- Using these final filtered values, we again recreate the image frame from the new image pyramid.
- We can also consider adding some chromatic aberration if we either want to mix the motion magnified signal and the original signal better(more homogeneously without weird colour artifacts) or we want to show a clear motion in the subsequent frames hence clearly separate the motion magnification and the frame(we can have a contrast between them). Which of the two cases happens will depend on the exact method(and value) of chromatic aberration.

```
output(:,:,1) = output(:,:,1)*chromaticAttenuation1;
output(:,:,2) = output(:,:,2)*chromaticAttenuation2;
output(:,:,3) = output(:,:,3)*chromaticAttenuation3;
```

- So we finally have the magnified motion. But we need the magnified motion on the image. Hence we add this magnified motion to the original input frame to get the output frame which we will write back to the output video.

```
output = frame + output;
```

Practically observing outputs of the above algorithm

To understand the algorithm completely and to verify if the outputs are intuitive, we ran the code and we am summarising the results below.

The baby sample video (which can be found in the "input/" folder of our dataset folder) was used to generate the images below.

Below is the second frame of the video. This is what goes as an input to our algorithm.

Figure 1: Frame 2 of baby video

The algorithm find out the difference between frame 1 and frame 2 and gives us the following output

Note that the above difference image had to be scaled to normalise it to cover the entire intensity levels otherwise its very difficult to see the difference and we only observe a black image. But one thing is really interesting in this image, that amidst the random squares that we observe, there is one straight red line, nearly in the centre of the image. If we try to match this to out input image, we will observe that this line belongs to the chain of the child's clothes. But as it would be obvious here, if we add this motion to the original frame, the resultant frame will be really bad since we see lots of random areas where there was ideally zero motion.

To solve this problem, we use chromatic aberration factors. We can see that most of the useful information is in red here, so we suppress the other color channels(by multiplying them by 0.1, so reducing them to 10% of original value). After performing the above calculation, the frame becomes

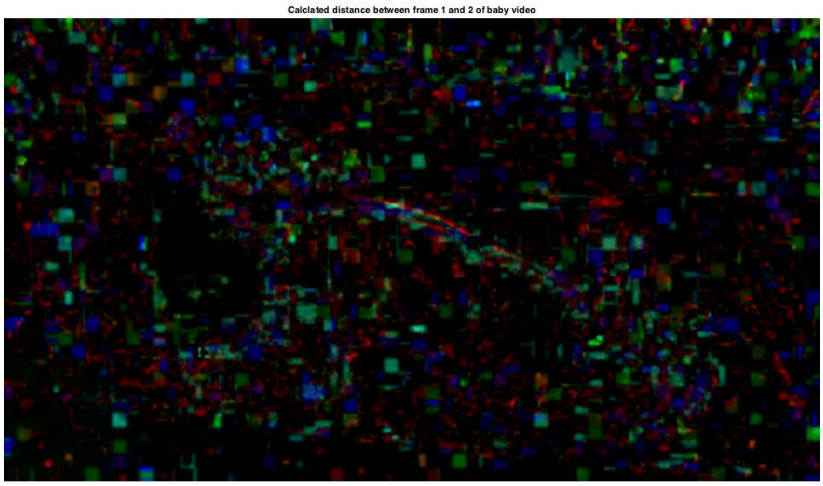


Figure 2: Frame 1-2 difference

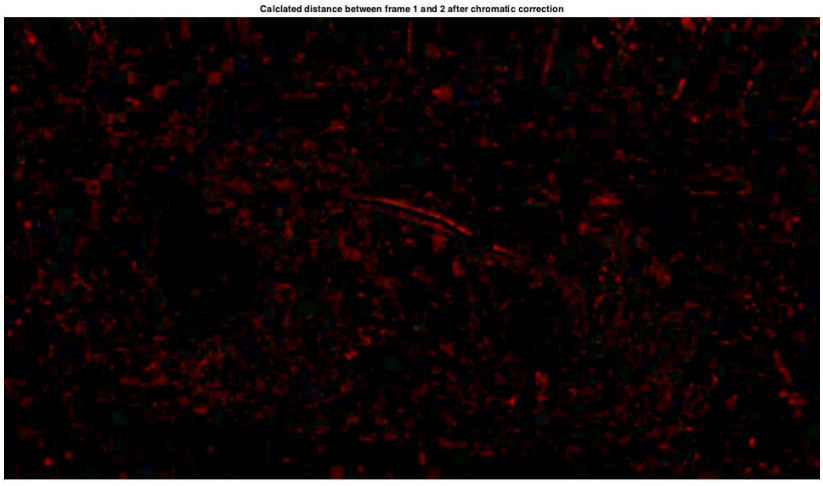


Figure 3: Frame 1-2 difference chromatic corrected

It is very clear now that the major motion is the one of the child's zip. This should be expected because other areas are more continuous, hence have a lesser frequency components, whereas a zip is like a discontinuity which contributes multiple frequencies and hence has a higher contribution to the difference frame.

We add the above frame to our original frame(in the RGB domain) and we get the final frame (for the output video) below.

Figure 4: Final Processed 2nd frame

If we closely notice, some of the bad areas in the image(patches in smooth areas) are exactly the same as what were observed in the final difference image(after chromatic aberrations).

Now, we would expect that the difference image that we get for nearby frames should be similar(since we aren't allowing for infinite frequency changes) and should be more different for frames which are further apart.

This claim is verified by the 3 images below.

We also see much lesser artefacts in frame 29-30 difference because after some time, the error caused by the initial condition nullifies.

Clearly we can see that the above technique is able to identify the motion in the frame(we can say this because it identifies the child's body motion, or the motion of his zip and neglects the other bodies in the frame). Also as we have seen the new created frames aren't that big a problem to the video quality(artefacts).

The final processed video was created by running the complete code. Output video can be found here.

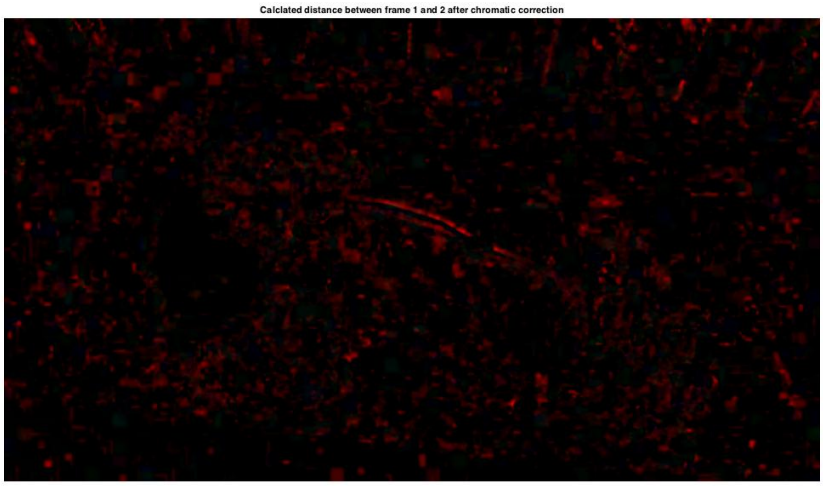


Figure 5: Frame 1-2 difference

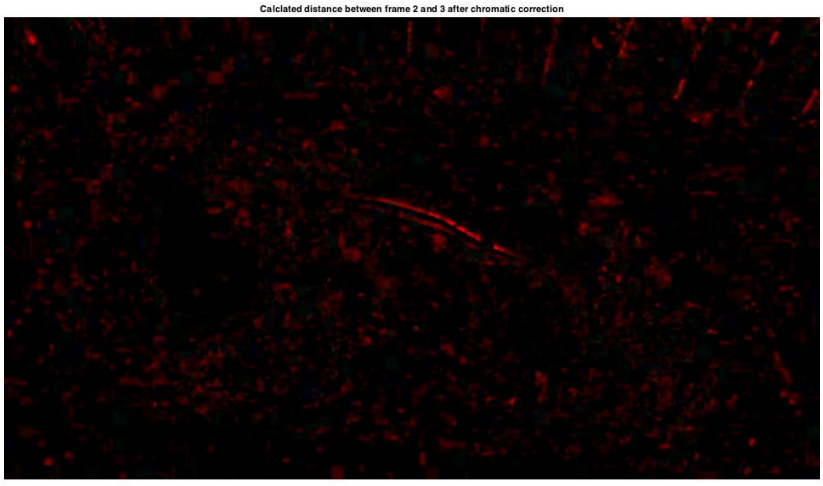


Figure 6: Frame 2-3 difference

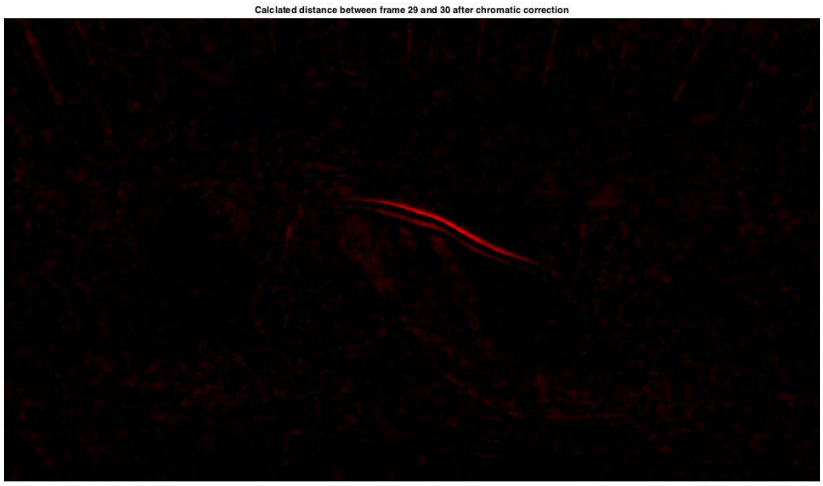


Figure 7: Frame 29-30 difference

Face Anti-spoofing

In this section, we briefly some aspects of the face anti-spoofing algorithm and then present the algorithm.

Local Binary Patterns

Feature vector The first set of features is obtained by calculating the local binary pattern for each pixel for a given frame, then obtaining a histogram of the LBP frequency values for the entire image. Due to the way the LBP is calculated, we will get exactly `P*(P-1) + 3` entries in the histogram, where `P` is the number of points we are using to sample.

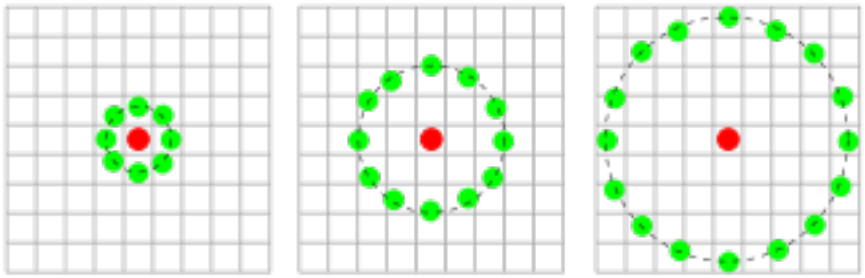


Figure 8: Local Binary Patterns for different P, R

Local Binary Pattern for a single pixel At a single pixel, the local binary pattern is calculated as follows: sample the image at `P` points on a circle of radius `R` around the centre of the pixel (we use an interpolation scheme like bilinear interpolation to calculate the intensity values at an arbitrary point inside the pixel). The figure above, taken from Wikipedia, shows some examples of how to circularly sample pixels around the centre of the current pixel.

Write down a binary number in which the kth digit is 1 if the kth sample point has intensity less than the centre pixel, and 0 otherwise. Count the number of 0-1 and 1-0 transitions in this binary number, and output 255 if there are more than 2 transitions, else output the binary number itself.

This is the LBP value for this pixel. Using a combinatoric argument, we can prove that there are at most `P*(P-1) + 3` possible values that can be output.

Local Binary Pattern for the entire image Collect the LBP values for each pixel of the image into a histogram. In the paper, the values (P,R) = (8,1), (8,2) and (16,1) have been used to obtain three histograms. These histograms have been concatenated into a single histogram array which is used as a feature vector for the SVM.

Final Algorithm

The overall algorithm for classifying an input video as spoofed or not is given below in Figure 9, taken from the paper. Intuitively, what it is doing is summing up the prediction scores from two SVMs. One is trained on features obtained from Local Binary Patterns, described above. The other is trained on features obtained from Histograms of Oriented Optical Flows. LBPs capture texture information. HOOFs capture liveness information. Together, these two metrics are summed to produce a net prediction score for each sliding window (the paper calls these windows videolets) in the input video. The net prediction scores over all windows are summed up to obtain a final prediction score. If this final score is above a certain threshold, we classify the video as spoofed.

Conclusions

In summary, we completely implemented the Video Magnification paper and got results quite close to the original code. Our code runs in near-real-time. We also partially implemented the Face Anti-spoofing paper.

Algorithm 1 Spoofing Detection in video V

input: A video V=$\{F_1, F_2, \ldots F_N\}$, trained models: SVM^{hoof} and SVM^{lbp}, videolet size w, interval k, and decision threshold T.
V^{mag} = Motion magnification of input video V
$F^{Mag-LBP} = lbp_{F^{mag}}$ (as in Eq. 10)
$F^{Mag-hoof} = hoof_{F^{mag},k}$ (as in Eq. 13)
$\eta = (\frac{2N}{w} - 1)$ (number of videolets)
$s = 1$
iterate: $i = 1$ **to** η **do**
$videolet_{Mag-LBP}$= $\{F_j^{Mag-LBP} | s \leq j \leq (s + w)\}$
$videolet_{Mag-hoof}$= $\{F_j^{Mag-hoof} | s \leq j \leq (s + w)\}$
$s = s + (\frac{w}{2})$
$P_{Mag-LBP}$= $SVM^{lbp}(videolet_{Mag-LBP})$
$P_{Mag-hoof}$= $SVM^{hoof}(videolet_{Mag-hoof})$
P_t^i= $P_{Mag-LBP}$ + $P_{Mag-hoof}$ (Evidence fusion)
end iterate.
F= $\frac{1}{\eta}\sum_{j=1}^{\eta}(P_t^j)$ (Videolet aggregation)
Output: report **if** $(F > T)$ "spoof" **else** "non-spoof"

Figure 9: Spoof detection algorithm