POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

A CNN-based detector for video frame-rate interpolation

Tesi di Laurea Magistrale in
Music and Acoustic Engineering - Ingegneria Musicale e Acustica

Simone Mariani, 939469

Advisor:
Prof. Paolo Bestagini

Academic year:
2021-2022

Abstract: Nowadays, thanks to the diffusion of hand-held video capturing devices and the widespread use of social networks and messaging apps, videos are commonly shared and have become part of our daily life. However, video manipulation is at everyone's hand thanks to the huge availability of easy-to-use video editing software suites. This has raised new social concerns, as the distribution of maliciously manipulated videos can lead to severe consequences (e.g., people defamation, fake-news spreading, mass opinion formation, etc.). For this reason, the multimedia forensics community has started developing a series of techniques to assess video authenticity and integrity. The goal of this thesis is to enrich the panorama of video forensic technique, by proposing a video frame-rate interpolation detector. Given a video under analysis, our goal is to detect whether the video has undergone some frame-rate upsampling operations that are often applied when multiple videos are spliced together, or to hide some part of a video. The proposed technique is based on an ensemble of Convolutional Neural Networks (CNNs) working on three different video domains (i.e., pixels, optical flow, and frame residuals), and on a Support Vector Machine (SVM) for a final classification. Results show that the proposed method outperforms state-of-the-art video frame-rate interpolation detectors, and can also be used to localize the spatial regions in which a video has been interpolated.

Key-words: video forensics, interpolation, frame-rate, deep learning, convolutional neural networks

1. Introduction

In the last twenty years, due to the increasing popularity of social media networks and to the technological advancements in video capturing devices, videos are spread everywhere. Anyone can capture videos with extreme facility, but unfortunately, anyone can also tamper with them. In fact, to the wide circulation of videos corresponds a great progress in the video manipulating software suites. These software suites are also straightforward to use, making video tampering a very simple task. Since videos are so popular nowadays, their information can impact on many people's lives. Maliciously manipulated videos can lead to several consequences like people defamation, fake-news spreading or mass opinion formation. For this reason, it is important to guarantee the integrity and the authenticity of the video contents, and this is exactly the aim of our work.

Among the many video manipulation operations that are available, video frame-rate interpolation is one of the most common ones. In principle, interpolating a video is not always related to malicious tampering of the

content, as it can also be used to simply adjust a video speed. However, video frame-rate interpolation operation is often applied when splicing together different videos, since it is required to find a balance between the frame-rates of the videos in question. This operation consists in creating new frames (upsampling) in between already existing consecutive frames, or even in dropping (downsampling) some frames, resulting in a frame-rate variation. If new frames are created, they can be a repetition of the original ones, or they can be built by interpolating between the original frames pixels. The first approach, gives rise to some artifacts in the video, especially if there is a lot of motion in the original content. The interpolating approach, instead is more difficult to detect, especially if the motion is taken into account during the interpolation, for example using Motion Compensated Interpolation (MCI) [1].
Indeed, MCI approaches perform first a motion estimation step, and then an interpolation one. The idea is to estimate the trajectories that each pixel follow from one frame to another. Then, the pixels of the newly generated frames are placed on the trajectory indicated by the estimation. In this way the interpolated frames are computed and inserted in the middle of the original frames, resulting in a very fluid sequence.
Since this video frame-rate interpolation can lead to very realistic results, it is required to develop some tools capable of identify manipulations introduced by interpolation.

For all of these reasons, in the past few years, the multimedia forensics community has started to develop some techniques in order to identify traces of video frame-rate interpolation.
For example, in [2], the authors propose a detector for motion compensated interpolation. In particular, this method relies on the analysis of the correlation introduced by the filter adopted in the interpolation process. This approach was initially implemented for the image spatial resampling problem, as explained in [3].
Another work in the video frame-rate interpolation field shows how interpolation footprints can be found analyzing the size of each encoded frame [4].
Also, since video interpolation can be used to speed up (dropping frames) or slow down (adding frames) a video, in this work [5], the authors use a CNN-based approach to discriminate between normal speed videos and sped up videos.

With this thesis, we want to create a new tool that can be helpful in detecting frame-rate interpolation traces left in videos. The goal of our work, is to be able to classify a given video, as frame-rate interpolated or as original (untouched frame-rate).
With this purpose, we propose a detector capable to identify traces of frame-rate interpolation inside a video. Our detector is composed by a processing stage (a Convolutional Neural Network (CNN)) and by a classifier (a Support Vector Machine (SVM)).
The idea, is to extract a batch of consecutive frames from a video under analysis. We produce different video domain versions of the batch of frames and we process (through the CNN) them in order to produce a feature vector containing some sort of information about the presence of interpolation in the segment. Finally we rely on this feature vector to classify (by the SVM) the video as interpolated or not.

The rest of the work is organized in the following way.
In Section 2 we formally define the goal of our work.
In Section 3 we fully describe all the details of our detector, and we also explain how to extend the approach to localize smaller interpolated video regions.
Section 4 is devoted to discuss the implementation settings of the experiments to validate our detector. Section 5 reports the results of the experiments, presenting a comparison against two state-of-the-art approaches.
In the final Section 6, we draw the conclusions of our work, and we indicate some possible future developments.

2. Problem Formulation

Videos can be temporally interpolated to increase or decrease their frame-rate for different applications. Changing a video frame-rate typically involves the use of some pixel interpolation technique over time. This is necessary to create new frames previously missing in the video stream. As interpolation leaves peculiar footprints on the newly generated frames, it is possible to study these traces in a forensic manner in order to detect if the video has been interpolated in time or not.

The goal of this work is to detect if a video under analysis has undergone frame-rate interpolation or not. With reference to Figure 1, this is a binary classification problem, with one video as input, and one binary label as output.

Formally, let us consider a video sequence defined as

$$\mathbf{V} = [\mathbf{F}_0, ..., \mathbf{F}_{L-1}], \tag{1}$$

where $\mathbf{F}_0$ denotes the first frame in $\mathbf{V}$ and L represents the number of frames contained in $\mathbf{V}$. Our goal is to learn how to associate a label p to the video $\mathbf{V}$ where $p = 1$ means that the video has been temporally interpolated, whereas $p = 0$ means that the video frame-rate is untouched.

In the next section, we provide all the details of the propose solution to this problem.

3. Proposed Method

In this section we introduce our proposed method for video frame-rate interpolation detection. First, we give a general overview of the entire pipeline. Then, we discuss the details of each processing block that compose our pipeline. We start with the preprocessing operations required by each video before passing through our model. After that, we analyze the structure of our detector. Finally, we explore a particular application of our work related to the problem of localizing interpolation that may happen in a small temporal and spatial region of the video.

3.1. Proposed Pipeline

In the current section we give a high level overview of the structure of our frame-rate interpolation detector. As shown in Figure 1, the pipeline can be summarized in three main blocks:

1. Preprocessing
2. CNN
3. Classifier

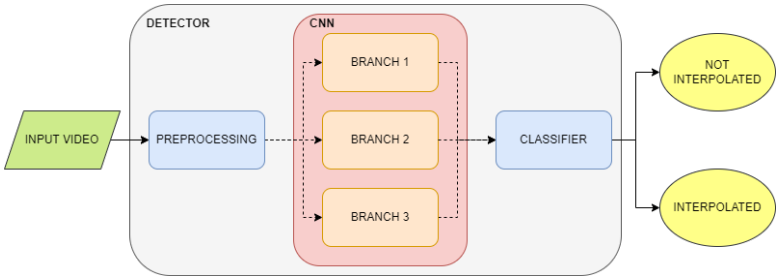


Figure 1: A quick overview of our detector. We analyze the input video in order to classify it as interpolated or not interpolated.

In the preprocessing stage, we start from an input video and we select a video segment from it by extracting only some consecutive frames. Then, we produce different versions of the video segment and we feed them to the CNN branches. The CNN processes our segments and produces an output score for each segment version. This scores are packed together before the final classifier block, which then tells if the video segment is classified as original or as interpolated. In the following, we dive into the details of each block.

3.2. Video Preprocessing

Before feeding video segments into our detector, some video preprocessing is required in order to get accurate and coherent predictions. In this section we focus on these operations. A detailed view of the preprocessing stage is presented in Figure 2.

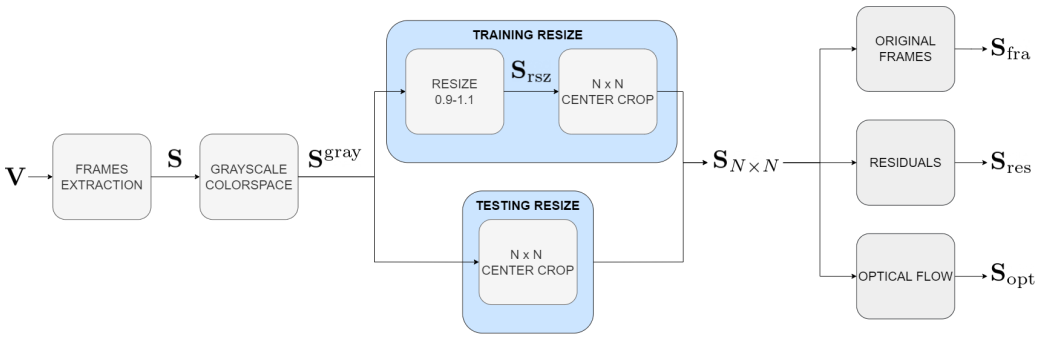


Figure 2: Details of preprocessing block. We start from the input video $\mathbf{V}$, we extract a segment $\mathbf{S}$ of frames. The segment is converted into a grayscale segment $\mathbf{S}^{\text{gray}}$. Then the frames of the segment are resized to obtain the $\mathbf{S}_{N\times N}$ segment. Finally the three input segments are produced from $\mathbf{S}_{N\times N}$.

3.3. Frames Selection

Considering an input video $\mathbf{V}$, the first thing that we do is to extract a video segment of $T+1$ consecutive frames from $\mathbf{V}$ producing $\mathbf{S}$. Formally, let us define the video under analysis as

$$\mathbf{V} = [\mathbf{F}_0, \mathbf{F}_1, ..., \mathbf{F}_{L-1}], \tag{2}$$

where L is the number of frames in $\mathbf{V}$ and $\mathbf{F}_i$ with $i \in [0,\ L-1]$ indicates the i-th frame of the video $\mathbf{V}$. In this way:

$$\mathbf{S} = [\mathbf{F}_i, ..., \mathbf{F}_{i+T}], \tag{3}$$

where $\mathbf{F}_i$ stands for the first frame (in the RGB colorspace [6]) of the new video segment and is composed by:

$$\mathbf{F}_i = [\mathbf{F}_i^{\text{R}}, \mathbf{F}_i^{\text{G}}, \mathbf{F}_i^{\text{B}}], \tag{4}$$

where $\mathbf{F}_i^{\text{R}}$, $\mathbf{F}_i^{\text{G}}$ and $\mathbf{F}_i^{\text{B}}$ represent the red, green and blue color components of the frame $\mathbf{F}_i$, respectively. Then, the next operation that we apply is a change of colorspace. This is essential to lower the global computational complexity, as this reduces the memory requirements to store each frame. In particular, we convert $\mathbf{S}$ into the grayscale video segment $\mathbf{S}^{\text{gray}}$. Formally, each frame of the new segment is produced in the following way:

$$\mathbf{F}_i^{\text{gray}} = \mathbf{F}_i^{\text{R}} \cdot 0.299 + \mathbf{F}_i^{\text{G}} \cdot 0.587 + \mathbf{F}_i^{\text{B}} \cdot 0.114. \tag{5}$$

At this point we have obtained a video segment $\mathbf{S}^{\text{gray}}$ composed by $T+1$ frames in the grayscale colorspace [6].

3.3.1 Frames Resizing

We now want to resize the video segment frames to let them fit in our CNN. This is done to both reduce the computational complexity, and enhance the CNN performance with some data augmentation during training. As a matter of fact, we make use of two different resizing policies: one is used during training; one is used during the testing phases.

Training In the training phase we apply a very light resizing opereation that acts as data augmentation factor. If each frame $\mathbf{F}_i^{\text{gray}}$ of our segment $\mathbf{S}^{\text{gray}}$ has a height H and a width W, the new resized frames have height and width defined as

$$\begin{aligned} H_{\text{rsz}} &= H \cdot r, \\ W_{\text{rsz}} &= W \cdot r, \end{aligned} \tag{6}$$

where H_{rsz} is the resized height, W_{rsz} is the resized width, and r is a random resize factor picked in the range between 0.9 and 1.1. We name this new segment as $\mathbf{S}_{\text{rsz}}$. We do this in order to introduce more variability in the training set. Then we just extract a center crop of $N \times N$ pixels from each frame of $\mathbf{S}_{\text{rsz}}$ obtaining the $\mathbf{S}_{N\times N}$ video segment. At this point, our video segment $\mathbf{S}_{N\times N}$ is composed by $T+1$ grayscale frames with height N and width N.

Testing In the testing stage of our CNN, we just apply a center crop of $N \times N$ pixels as we did at last in training, obtaining $\mathbf{S}_{N\times N}$. This is intended to preserve the original quality of the input video to evaluate, still adapting the frame resolution to that required by the CNN. So the goal of the preprocessing phase is to produce the grayscale video segment $\mathbf{S}_{N\times N}$ composed by $T+1$ grayscale frames of height N and width N.

3.3.2 Input Type (Frames, Residuals or Optical Flow)

In order to obtain a robust detector, our approach is to create three different versions of the preprocessed video segment $\mathbf{S}_{N\times N}$ that we define as

- Original Frames
- Residuals
- Optical Flow

We do this because we want to be able to analyze different inputs exposing different properties of the same video segment, in order to capture more information about the interpolation traces. We want each one of these new sequences to be T frames long. These three segments are then evaluated by our CNN, producing three output scores, which are then aggregated in order to produce the final prediction related to the original input video segment $\mathbf{S}$. In the three following paragraphs we discuss how to obtain these new segments.

Original Frames The first input type is the original video in the pixel domain. This is the most simple and basic input we can think of, and it surely contains all the information to describe the video. $\mathbf{S}_{N\times N}$ already represents the original (preprocessed) frames. The only thing to do is to discard the last frame to have a total length of T frames, since the length of $\mathbf{S}_{NxN}$ was of $T+1$. We refer to this new segment as $\mathbf{S}_{\text{fra}}$.

Residuals The second input type that we consider is the frames residual. This is obtained as the difference between adjacent frames in the time domain. This choice is motivated by two main reasons: it is well known that frames residuals may contain forensic traces in several multimedia forensics applications; the frame difference somehow deletes part of the semantic content of the frames (i.e., the depicted scene), thus helping the CNN to focus on processing traces.
To compute the residuals segment, which we call $\mathbf{S}_{\text{res}}$, all we need to do is one simple operation. Each frame in the residuals sequence is obtained by:

$$\mathbf{R}_i = |\mathbf{N}_{i+1} - \mathbf{N}_i|, \quad \forall i \in [0, T], \tag{7}$$

where $\mathbf{N}_i$ represents a frame of the preprocessed segment $\mathbf{S}_{N\times N}$. This frames are then concatenated to obtain the residuals sequence

$$\mathbf{S}_{\text{res}} = [\mathbf{R}_0, \mathbf{R}_1, ..., \mathbf{R}_{T-1}], \tag{8}$$

where $\mathbf{R}_i$ indicates a frame in the new residuals sequence. In this way the length of $\mathbf{S}_{\text{res}}$ is of T frames exactly.

Optical Flow The third input type we use is the optical flow. This is a representation of the motion of each pixel from one frame to the next one. This choice is motivated by the fact that frame-rate interpolation can leave visible traces if we inspect video motion. It is therefore expected that feeding the optical flow to a CNN can be a fruitful choice to capture motion artifacts, thus detecting frame-rate interpolation.
In order to produce the optical flow segment $\mathbf{S}_{\text{opt}}$, we follow the method proposed by Gunnar Farneback [7]. This approach produces a dense optical flow. The motion estimation is computed between every two adjacent frames of the segment $\mathbf{S}_{N\times N}$ and the flow is calculated for every point in the frames. Let us suppose that $\mathbf{F}_i$ and $\mathbf{F}_j$ are two consecutive grayscale frames of $\mathbf{S}_{N\times N}$. We first define a mask $\mathbf{M}_{\text{opt}}$ with three channels and with the same height and width as each one of the frames of $\mathbf{S}_{N\times N}$. The three channels of $\mathbf{M}_{\text{opt}}$ are related to hue ($\mathbf{M}^h_{\text{opt}}$), saturation ($\mathbf{M}^s_{\text{opt}}$) and value ($\mathbf{M}^v_{\text{opt}}$), since in this approach we want to work in the HSV colorspace [6]. Then, the dense optical flow $\mathbf{O}_{ij}$ is computed from $\mathbf{F}_i$ and $\mathbf{F}_j$ according to Farneback method. From the obtained flow we calculate the magnitude ρ_{ij} and the angle $\mathbf{\Phi}_{ij}$ of the 2D vectors, converting the first two channels of $\mathbf{O}_{ij}$ from cartesian to polar coordinates. We use then the angle $\mathbf{\Phi}_{ij}$ to set the mask $\mathbf{M}_{\text{opt}}$ hue channel $\mathbf{M}^h_{\text{opt}}$ according to the optical flow direction:

$$\mathbf{M}^h_{opt} = \frac{\mathbf{\Phi}_{ij} \cdot 180}{\pi/2} \tag{9}$$

Being $\mathbf{M}^h_{\text{opt}}$ the hue channel (first channel) of $\mathbf{M}_{\text{opt}}$. After that we use the magnitude ρ_{ij} to set the mask $\mathbf{M}_{\text{opt}}$ value channel:

$$\mathbf{M}^v_{\text{opt}} = \rho_{ij} \tag{10}$$

Finally we convert the $\mathbf{M}_{\text{opt}}$ mask back to the grayscale colorspace. We repeat the procedure for all of the consecutive frames in $\mathbf{S}_{N\times N}$ and we concatenate the results ($\mathbf{M}_{\text{opt}}$) obtaining $\mathbf{S}_{\text{opt}}$. In this way we get a total length of T frames.
Figure 3 shows three frames from the three types of input segments.

(a) original frame (b) residual (c) optical flow

Figure 3: Samples of the same frame from the three input segment types.

3.4. CNN Single Branch Structure

In this section we analyze the structure of the core of our detector: the CNN.
As depicted in Figure 4 , the proposed CNN is composed by three branches. Each of these branches is devoted to the processing of one of the three different input segment types that we described in Section 3.3.2: $\mathbf{S}_{\text{fra}}$, $\mathbf{S}_{\text{res}}$, $\mathbf{S}_{\text{opt}}$. The three branches which we call $C_{B1}(\cdot)$, $C_{B2}(\cdot)$, $C_{B3}(\cdot)$ respectively, have the exact same layers, so they are structurally identical. The only difference between them stands in their weights, since each branch needs to adjust its weights in the training phase in order to be optimized for the processing of a specific type of input. The training details can be found in the next section.

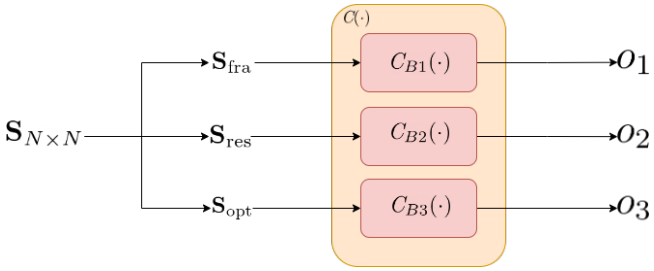


Figure 4: CNN high level structure. From the preprocessed segment $\mathbf{S}_{N\times N}$ we obtained the three input segments $\mathbf{S}_{\text{fra}}$, $\mathbf{S}_{\text{res}}$ and $\mathbf{S}_{\text{opt}}$. We fed each one of them into the correspondent branch. Three output scores are produced from each branch analysis (o_1, o_2 and o_3).

A graphical view of the structure of a single branch is shown in Figure 5. Table 1 reports a branch structure layer by layer. Aside from the common layers as *Convolutional* and *Pooling* for example, a particular mention must be made for the *Mixed* layers (indicated in green both in Table 1 and in Figure 5). They are not properly layers, but they are an aggregation of simpler layers. Figure 6 represents the structure of one of these particular aggregation of layers. All of the *Mixed* layers contain the same set of layers, they only differ in the layers parameters. This is evident from Table 2, in which the output shapes from each layer composing a *Mixed* are shown.
One last remark about the architecture of the individual layers is that it is very similar to the one of the S3D-G network [8], with a few variations:

- Temporal strides for all the Max Pooling layers (except the last one) are set to 1, in order not to alter the temporal dimension.
- We perform max spatial pooling and adaptive average temporal pooling before the last 3D convolution, as opposed to only average pooling in S3D-G, this in order to compress spatial and temporal dimensions.

Going back to the pipeline in Figure 4, our idea is to start from the three segments $\mathbf{S}_{\text{fra}}$, $\mathbf{S}_{\text{res}}$, $\mathbf{S}_{\text{opt}}$ and to feed each one of them respectively to $C_{B1}(\cdot)$, $C_{B2}(\cdot)$, $C_{B3}(\cdot)$. Each branch accepts an input of dimension $T \times N \times N \times 1$, since each segment is composed by T frames of height N, width N and single channel (grayscale colorspace). After processing the segments through $C(\cdot)$, each branch produces as output a value:

$$o_1 = C_{B1}(\mathbf{S}_{\text{fra}}) \tag{11}$$

$$o_2 = C_{B2}(\mathbf{S}_{\text{res}}) \tag{12}$$

$$o_3 = C_{B3}(\mathbf{S}_{\text{opt}}) \tag{13}$$

LAYER	OUTPUT SHAPE
Conv3d	[-1, 64, T, N/2, N/2]
BatchNorm3	[-1, 64, T, N/2, N/2]
ReLU	[-1, 64, T, N/2, N/2]
Conv3d	[-1, 64, T, N/2, N/2]
BatchNorm3d	[-1, 64, T, N/2, N/2]
ReLU	[-1, 64, T, N/2, N/2]
MaxPool3d	[-1, 64, T, N/4, N/4]
Conv3d	[-1, 64, T, N/4, N/4]
BatchNorm3d	[-1, 64, T, N/4, N/4]
ReLU	[-1, 64, T, N/4, N/4]
Conv3d	[-1, 192, T, N/4, N/4]
BatchNorm3d	[-1, 192, T, N/4, N/4]
ReLU	[-1, 192, T, N/4, N/4]
Conv3d	[-1, 192, T, N/4, N/4]
BatchNorm3d	[-1, 192, T, N/4, N/4]
ReLU	[-1, 192, 16, N/4, N/4]
MaxPool3d	[-1, 192, T, N/8, N/8]
Mixed_3b	[-1, 256, T, N/8, N/8]
Mixed_3c	[-1, 480, T, N/8, N/8]
MaxPool3d	[-1, 480, T, N/16, N/16]
Miced_4b	[-1, 512, T, N/16, N/16]
Mixed_4c	[-1, 512, T, N/16, N/16]
Mixed_4d	[-1, 512, T, N/16, N/16]
Mixed_4e	[-1, 528, T, N/16, N/16]
Mixed_4f	[-1, 832, T, N/16, N/16]
MaxPool3d	[-1, 832, T, N/32, N/32]
Mixed_5b	[-1, 832, 16, N/32, N/32]
Mixed_5c	[-1, 1024, 16, N/32, N/32]
MaxPool3d	[-1, 1024, 1, N/32, N/32]
AdaptiveAvgPool3d	[-1, 1024, 1, 1, 1]
Conv3d	[-1, 512, 1, 1, 1]
Dropout	[-1, 512]
Linear	[-1, 1]

Table 1: Layers with output shapes of CNN considering an input shape of [-1, 1, T, N, N]. In the input shape, the first dimension (-1) stands for the batch size, which can be set freely. The second dimension (1) is the number of input channels of each frame. The third dimension (T) is the number of frames in the segment. The last two dimensions (N, N) are the spatial dimensions (height, width) of each frame.

LAYER	MIXED LAYERS OUTPUT SHAPES									
	Mixed_3b	*Mixed_3c*	*Mixed_4b*	*Mixed_4c*	*Mixed_4d*	*Mixed_4e*	*Mixed_4f*	*Mixed_5b*	*Mixed_5c*	
Conv3d_0_0	[-1, 64, T, N/8, N/8]	[-1, 128, T, N/8, N/8]	[-1, 192, T, N/16, N/16]	[-1, 160, T, N/16, N/16]	[-1, 128, T, N/16, N/16]	[-1, 112, T, N/16, N/16]	[-1, 256, T, N/16, N/16]	[-1, 256, T, N/32, N/32]	[-1, 384, T, N/32, N/32]	
BatchNorm3d_0_0	[-1, 64, T, N/8, N/8]	[-1, 128, T, N/8, N/8]	[-1, 192, T, N/16, N/16]	[-1, 160, T, N/16, N/16]	[-1, 128, T, N/16, N/16]	[-1, 112, T, N/16, N/16]	[-1, 256, T, N/16, N/16]	[-1, 256, T, N/32, N/32]	[-1, 384, T, N/32, N/32]	
ReLU_0_0	[-1, 64, T, N/8, N/8]	[-1, 128, T, N/8, N/8]	[-1, 192, T, N/16, N/16]	[-1, 160, T, N/16, N/16]	[-1, 128, T, N/16, N/16]	[-1, 112, T, N/16, N/16]	[-1, 256, T, N/16, N/16]	[-1, 256, T, N/32, N/32]	[-1, 384, T, N/32, N/32]	
Conv3d_1_0	[-1, 64, T, N/8, N/8]	[-1, 128, T, N/8, N/8]	[-1, 96, T, N/16, N/16]	[-1, 112, T, N/16, N/16]	[-1, 128, T, N/16, N/16]	[-1, 144, T, N/16, N/16]	[-1, 160, T, N/16, N/16]	[-1, 160, T, N/32, N/32]	[-1, 192, T, N/32, N/32]	
BatchNorm3d_1_0	[-1, 64, T, N/8, N/8]	[-1, 128, T, N/8, N/8]	[-1, 96, T, N/16, N/16]	[-1, 112, T, N/16, N/16]	[-1, 128, T, N/16, N/16]	[-1, 144, T, N/16, N/16]	[-1, 160, T, N/16, N/16]	[-1, 256, T, N/32, N/32]	[-1, 192, T, N/32, N/32]	
ReLU_1_0	[-1, 64, T, N/8, N/8]	[-1, 128, T, N/8, N/8]	[-1, 96, T, N/16, N/16]	[-1, 112, T, N/16, N/16]	[-1, 128, T, N/16, N/16]	[-1, 144, T, N/16, N/16]	[-1, 160, T, N/16, N/16]	[-1, 256, T, N/32, N/32]	[-1, 192, T, N/32, N/32]	
Conv3d_1_1	[-1, 128, T, N/8, N/8]	[-1, 192, T, N/8, N/8]	[-1, 208, T, N/16, N/16]	[-1, 224, T, N/16, N/16]	[-1, 256, T, N/16, N/16]	[-1, 288, T, N/16, N/16]	[-1, 320, T, N/16, N/16]	[-1, 320, T, N/32, N/32]	[-1, 384, T, N/32, N/32]	
BatchNorm3d_1_1	[-1, 128, T, N/8, N/8]	[-1, 192, T, N/8, N/8]	[-1, 208, T, N/16, N/16]	[-1, 224, T, N/16, N/16]	[-1, 256, T, N/16, N/16]	[-1, 288, T, N/16, N/16]	[-1, 320, T, N/16, N/16]	[-1, 320, T, N/32, N/32]	[-1, 384, T, N/32, N/32]	
ReLU_1_1	[-1, 128, T, N/8, N/8]	[-1, 192, T, N/8, N/8]	[-1, 208, T, N/16, N/16]	[-1, 224, T, N/16, N/16]	[-1, 256, T, N/16, N/16]	[-1, 288, T, N/16, N/16]	[-1, 320, T, N/16, N/16]	[-1, 320, T, N/32, N/32]	[-1, 384, T, N/32, N/32]	
Conv3d_1_2	[-1, 128, T, N/8, N/8]	[-1, 192, T, N/8, N/8]	[-1, 208, T, N/16, N/16]	[-1, 224, T, N/16, N/16]	[-1, 256, T, N/16, N/16]	[-1, 288, T, N/16, N/16]	[-1, 320, T, N/16, N/16]	[-1, 320, T, N/32, N/32]	[-1, 384, T, N/32, N/32]	
BatchNorm3d_1_2	[-1, 128, T, N/8, N/8]	[-1, 192, T, N/8, N/8]	[-1, 208, T, N/16, N/16]	[-1, 224, T, N/16, N/16]	[-1, 256, T, N/16, N/16]	[-1, 288, T, N/16, N/16]	[-1, 320, T, N/16, N/16]	[-1, 320, T, N/32, N/32]	[-1, 384, T, N/32, N/32]	
ReLU_1_2	[-1, 128, T, N/8, N/8]	[-1, 192, T, N/8, N/8]	[-1, 208, T, N/16, N/16]	[-1, 224, T, N/16, N/16]	[-1, 256, T, N/16, N/16]	[-1, 288, T, N/16, N/16]	[-1, 320, T, N/16, N/16]	[-1, 320, T, N/32, N/32]	[-1, 384, T, N/32, N/32]	
Conv3d_2_0	[-1, 16, T, N/8, N/8]	[-1, 32, T, N/8, N/8]	[-1, 16, T, N/16, N/16]	[-1, 24, T, N/16, N/16]	[-1, 24, T, N/16, N/16]	[-1, 32, T, N/16, N/16]	[-1, 32, T, N/16, N/16]	[-1, 32, T, N/32, N/32]	[-1, 48, T, N/32, N/32]	
BatchNorm3d_2_0	[-1, 16, T, N/8, N/8]	[-1, 32, T, N/8, N/8]	[-1, 16, T, N/16, N/16]	[-1, 24, T, N/16, N/16]	[-1, 24, T, N/16, N/16]	[-1, 32, T, N/16, N/16]	[-1, 32, T, N/16, N/16]	[-1, 32, T, N/32, N/32]	[-1, 48, T, N/32, N/32]	
ReLU_2_0	[-1, 16, T, N/8, N/8]	[-1, 32, T, N/8, N/8]	[-1, 16, T, N/16, N/16]	[-1, 24, T, N/16, N/16]	[-1, 24, T, N/16, N/16]	[-1, 32, T, N/16, N/16]	[-1, 32, T, N/16, N/16]	[-1, 32, T, N/32, N/32]	[-1, 48, T, N/32, N/32]	
Conv3d_2_1	[-1, 32, T, N/8, N/8]	[-1, 96, T, N/8, N/8]	[-1, 48, T, N/16, N/16]	[-1, 64, T, N/16, N/16]	[-1, 64, T, N/16, N/16]	[-1, 64, T, N/16, N/16]	[-1, 128, T, N/16, N/16]	[-1, 128, T, N/32, N/32]	[-1, 128, T, N/32, N/32]	
BatchNorm3d_2_1	[-1, 32, T, N/8, N/8]	[-1, 96, T, N/8, N/8]	[-1, 48, T, N/16, N/16]	[-1, 64, T, N/16, N/16]	[-1, 64, T, N/16, N/16]	[-1, 64, T, N/16, N/16]	[-1, 128, T, N/16, N/16]	[-1, 128, T, N/32, N/32]	[-1, 128, T, N/32, N/32]	
ReLU_2_1	[-1, 32, T, N/8, N/8]	[-1, 96, T, N/8, N/8]	[-1, 48, T, N/16, N/16]	[-1, 64, T, N/16, N/16]	[-1, 64, T, N/16, N/16]	[-1, 64, T, N/16, N/16]	[-1, 128, T, N/16, N/16]	[-1, 128, T, N/32, N/32]	[-1, 128, T, N/32, N/32]	
Conv3d_2_2	[-1, 32, T, N/8, N/8]	[-1, 96, T, N/8, N/8]	[-1, 48, T, N/16, N/16]	[-1, 64, T, N/16, N/16]	[-1, 64, T, N/16, N/16]	[-1, 64, T, N/16, N/16]	[-1, 128, T, N/16, N/16]	[-1, 128, T, N/32, N/32]	[-1, 128, T, N/32, N/32]	
BatchNorm3d_2_2	[-1, 32, T, N/8, N/8]	[-1, 96, T, N/8, N/8]	[-1, 48, T, N/16, N/16]	[-1, 64, T, N/16, N/16]	[-1, 64, T, N/16, N/16]	[-1, 64, T, N/16, N/16]	[-1, 128, T, N/16, N/16]	[-1, 128, T, N/32, N/32]	[-1, 128, T, N/32, N/32]	
ReLU_2_2	[-1, 32, T, N/8, N/8]	[-1, 96, T, N/8, N/8]	[-1, 48, T, N/16, N/16]	[-1, 64, T, N/16, N/16]	[-1, 64, T, N/16, N/16]	[-1, 64, T, N/16, N/16]	[-1, 128, T, N/16, N/16]	[-1, 128, T, N/32, N/32]	[-1, 128, T, N/32, N/32]	
MaxPool3d_3	[-1, 32, T, N/8, N/8]	[-1, 256, T, N/8, N/8]	[-1, 480, T, N/16, N/16]	[-1, 512, T, N/16, N/16]	[-1, 512, T, N/16, N/16]	[-1, 512, T, N/16, N/16]	[-1, 528, T, N/16, N/16]	[-1, 832, T, N/32, N/32]	[-1, 832, T, N/32, N/32]	
Conv3d_3_0	[-1, 32, T, N/8, N/8]	[-1, 64, T, N/8, N/8]	[-1, 64, T, N/16, N/16]	[-1, 64, T, N/16, N/16]	[-1, 64, T, N/16, N/16]	[-1, 64, T, N/16, N/16]	[-1, 128, T, N/16, N/16]	[-1, 128, T, N/32, N/32]	[-1, 128, T, N/32, N/32]	
BatchNorm3d_3_0	[-1, 32, T, N/8, N/8]	[-1, 64, T, N/8, N/8]	[-1, 64, T, N/16, N/16]	[-1, 64, T, N/16, N/16]	[-1, 64, T, N/16, N/16]	[-1, 64, T, N/16, N/16]	[-1, 128, T, N/16, N/16]	[-1, 128, T, N/32, N/32]	[-1, 128, T, N/32, N/32]	
ReLU_3_0	[-1, 32, T, N/8, N/8]	[-1, 64, T, N/8, N/8]	[-1, 64, T, N/16, N/16]	[-1, 64, T, N/16, N/16]	[-1, 64, T, N/16, N/16]	[-1, 64, T, N/16, N/16]	[-1, 128, T, N/16, N/16]	[-1, 128, T, N/32, N/32]	[-1, 128, T, N/32, N/32]	
AdaptiveAvgPool3d	[-1, 256, 1, 1, 1]	[-1, 480, 1, 1, 1]	[-1, 512, 1, 1, 1]	[-1, 512, 1, 1, 1]	[-1, 512, 1, 1, 1]	[-1, 528, 1, 1, 1]	[-1, 832, 1, 1, 1]	[-1, 832, 1, 1, 1]	[-1, 1024, 1, 1, 1]	
Conv3d	[-1, 1, 256, 1, 1]	[-1, 1, 480, 1, 1]	[-1, 1, 512, 1, 1]	[-1, 1, 512, 1, 1]	[-1, 1, 512, 1, 1]	[-1, 1, 528, 1, 1]	[-1, 1, 832, 1, 1]	[-1, 1, 832, 1, 1]	[-1, 1, 1024, 1, 1]	
Sigmoid	[-1, 1, 256, 1, 1]	[-1, 1, 480, 1, 1]	[-1, 1, 512, 1, 1]	[-1, 1, 512, 1, 1]	[-1, 1, 512, 1, 1]	[-1, 1, 528, 1, 1]	[-1, 1, 832, 1, 1]	[-1, 1, 832, 1, 1]	[-1, 1, 1024, 1, 1]	

Table 2: Output shapes for all the layers composing the various *Mixed* layers. The notation is the same used in Table 1.

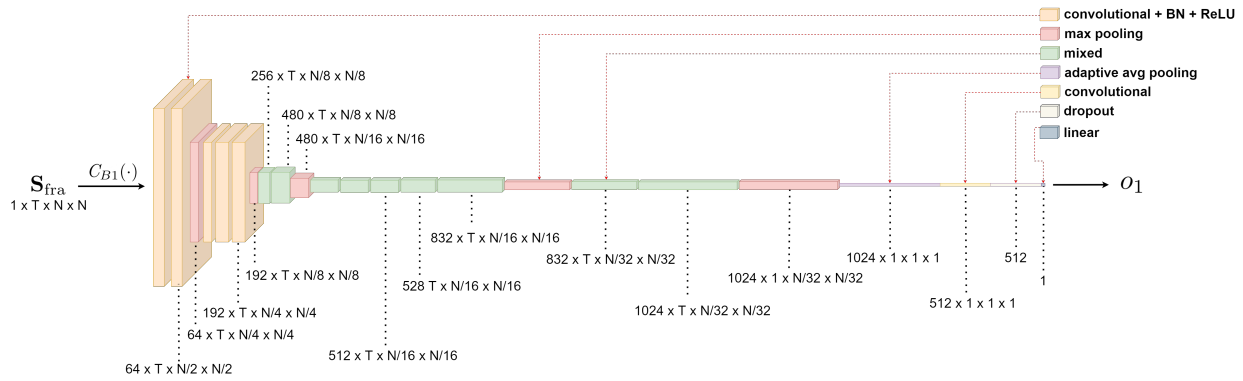


Figure 5: CNN single branch structure. In the figure is reported the structure of the first branch ($C_{B1}(\cdot)$) devoted to the processing of the frames segment ($\mathbf{S}_{\text{fra}}$). Also, the output shapes and the type of layers are indicated.

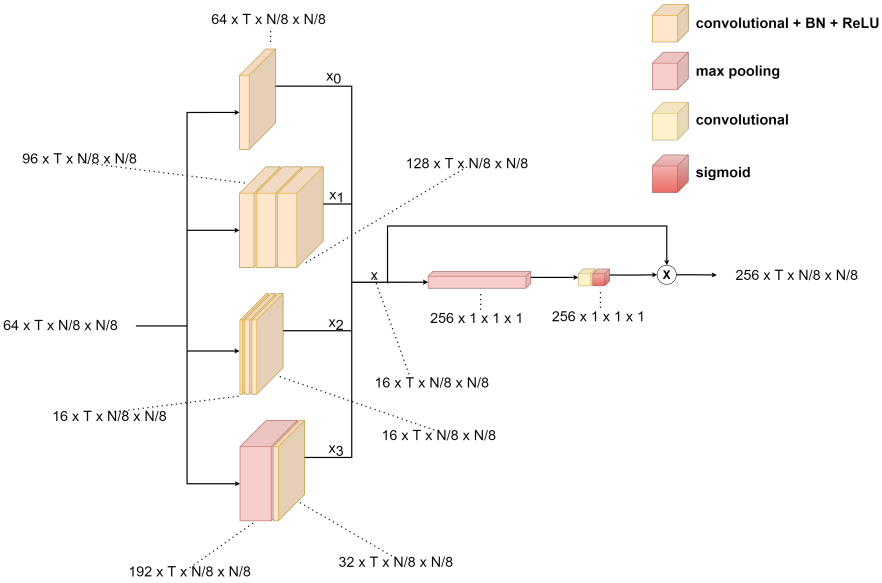


Figure 6: First Mixed layer structure. Sub layers types and output shapes are indicated in the figure.

These values represents a sort of score related to the level of interpolation traces that have been found inside each segment. In particular, o_1, o_2 and o_3 scores span the range $(-\infty, +\infty)$. The higher the score, the higher the probability that the video has been interpolated.

3.5. CNN Branches Aggregation

The final part of our detector consists in putting together the scores that we collected from the single branches and to produce from them a global prediction about the input video. In Figure 7 a simple pipeline representing this stage is presented.

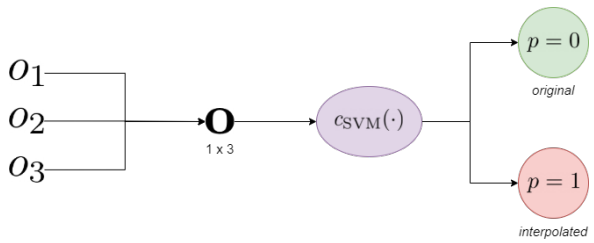


Figure 7: Pipeline of the final classification stage. The three branches outcomes (o_1, o_2 and o_3) are aggregated into the feature vector O. This vector is fed to the classifier ($c_{\text{SVM}}(\cdot)$) which predicts if the video is interpolated ($p = 1$) or original ($p = 0$).

First of all, we concatenate the scores together in a final feature vector:

$$\mathbf{o} = [o_1, o_2, o_3]. \tag{14}$$

Then, we need a tool to analyze this feature vector $\mathbf{o}$ and to predict if the original video segment $\mathbf{S}$ has interpolation traces in it or not. What we propose is to use a SVM.
We build and train a SVM classifier $c_{\text{SVM}}(\cdot)$ in order to discriminate between original video segment (class 0) and interpolated video segment (class 1).
The feature vector $\mathbf{o}$ is then fed to the classifier $c_{\text{SVM}}(\cdot)$ producing the final prediction:

$$p = c_{\text{SVM}}(\mathbf{o}) = \begin{cases} 0, & \text{if } \mathbf{S}_{N\times N} \text{ is predicted as not interpolated,} \\ 1, & \text{if } \mathbf{S}_{N\times N} \text{ is predicted as interpolated,} \end{cases} \tag{15}$$

where p is the final prediction related to the input video preprocessed segment $\mathbf{S}_{N\times N}$ and $\mathbf{o}$ is the feature vector containing the branches scores. In this way, p represents a prediction made about the input video segment $\mathbf{S}$, stating if it contains traces of interpolation.
Details about the training and the costruction of $c_{\text{SVM}}(\cdot)$ are discussed in the next chapter along with some alternatives to this classification procedure.

3.6. Localization Application

In this section we show an extension to a particular application of our detector. Until now, we have shown how to detect if a video segment has interpolation traces in it. However, as we analyze a video segment-by-segment, we can also localize where the interpolation is located inside a video. In order to do this, we designed the pipeline presented in Figure 8.

Figure 8: High level pipeline of localization application. From the input video $\mathbf{V}_{\text{in}}$ to the output video $\mathbf{V}_{\text{out}}$ in which the interpolated sub blocks are highlighted.

Our idea is to divide an input video into many segments, then to process these segments individually to localize the interpolation zones inside of them. For each segment we produce a mask indicating where this interpolated regions are placed. In the end we use this masks to create an output video which consists in a copy of the input video in which the interpolated sectors in each segment are highlighted.
In the following, we explain the main blocks of our localization method.

3.6.1 Video Segmentation

Let us start from an input video $\mathbf{V}_{\text{in}}$. First we want to divide it into $\mathbf{B}$ video segments, each of length $T+1$ frames.
In this way:

$$\mathbf{S}_s = [\mathbf{S}_0, \mathbf{S}_1, ..., \mathbf{S}_{B-1}], \quad B = \left\lfloor \frac{L}{T+1} \right\rfloor, \tag{16}$$

where L is the number of frames in $\mathbf{V}_{\text{in}}$ and $\mathbf{S}_s$ is the succession of frame segments.
In particular each segment is composed by:

$$\mathbf{S}_i = [\mathbf{F}_{i\cdot T}, ..., \mathbf{F}_{i\cdot(T+1)}], \quad i \in [0, B-1], \tag{17}$$

where $\mathbf{F_i}$ is the i-th frame contained in $\mathbf{V}_{\text{in}}$. As we process each segment in the same way, let us focus on one single segment $\mathbf{S}_i$ for the sake of simplicity. The individual processing of each one of the segments is shown in Figure 9.
From the segment $\mathbf{S}_i$, we generate the grayscale version $\mathbf{S}_i^{\text{gray}}$ by converting every frame $\mathbf{F}_i$ in $\mathbf{S}_i$ to the grayscale colorspace with equation (5). After that, we decompose our video segment $\mathbf{S}_i^{\text{gray}}$ from a spatial point of view, into multiple sub segments. In particular we want each sub segment to be composed by the $T+1$ frames of $\mathbf{S}_i^{\text{gray}}$ but for each different subsegment, we crop those frames in different $N \times N$ non overlapping regions. In this way, each sub segment represents the $T+1$ frames of $\mathbf{S}_i^{\text{gray}}$ but evaluated in a specific spatial region. An example of sub segment is reported in Figure 10.

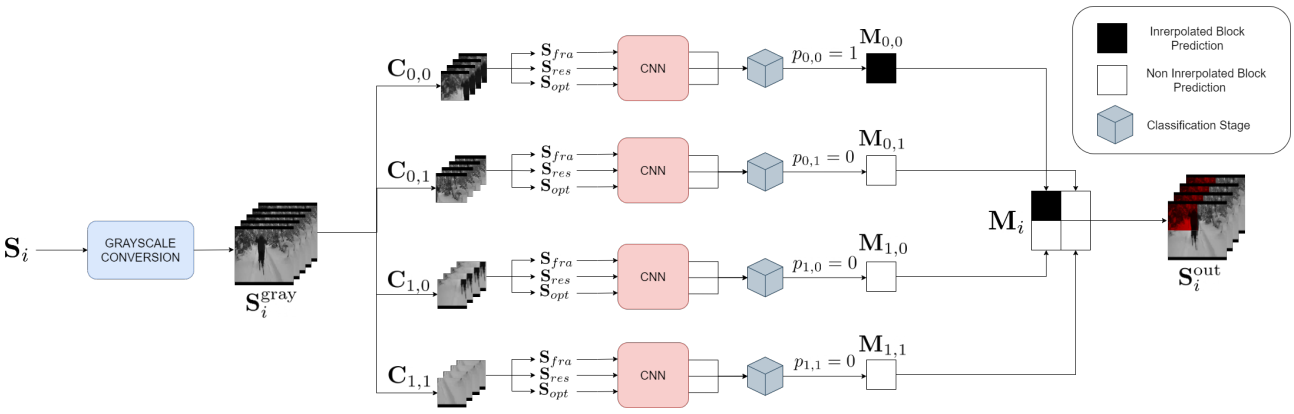


Figure 9: Localization processing chain for individual video segments, in this example the input segment has been divided into four $N \times N$ sub segments. Each subsegment is then processed as a standalone segment producing a prediction. The predictions for each subsegment are then used to compose the mask $\mathbf{M}_i$ related to the input segment.

Figure 10: A sub segment of 5 frames 200 x 200 extracted from a segment of 5 frames 400 x 400.

At this point we can individually analyze different spatial portions of $\mathbf{S}_i^{\text{gray}}$. Supposing that each frame of $\mathbf{S}_i^{\text{gray}}$ has a height of H and a width of W we have:

$$J = \left\lfloor \frac{N}{H} \right\rfloor, \qquad K = \left\lfloor \frac{N}{W} \right\rfloor, \tag{18}$$

where J and K are the number of $N \times N$ non-overlapping blocks that we can get to cover a frame of height H and width W. Therefore, we can define the sub segment subdivision of $\mathbf{S}_i^{gray}$ in the following way:

$$\mathbf{C} = \begin{bmatrix} \mathbf{C}_{0,0} & \mathbf{C}_{0,1} & \mathbf{C}_{0,2} & \dots & \mathbf{C}_{0,W-1} \\ \mathbf{C}_{1,0} & \mathbf{C}_{1,1} & \mathbf{C}_{1,2} & \dots & \mathbf{C}_{1,W-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{C}_{J-1,0} & \mathbf{C}_{J-1,1} & \mathbf{C}_{J-1,2} & \dots & \mathbf{C}_{J-1,W-1} \end{bmatrix}, \tag{19}$$

where $\mathbf{C}$ represents the global spatial subdivision of the segment $\mathbf{S}_i^{gray}$, and it is composed by:

$$\mathbf{C}_{j,k} = [\mathbf{F}_{i \cdot T}[j : j+N, k : k+N], ..., \mathbf{F}_{i \cdot (T+1)}[j : j+N, k : k+N]], \tag{20}$$

where $j \in (0, J-1)$, $k \in (0, K-1)$ and i is related to the index of the current segment ($\mathbf{S}_i$). In practice, $\mathbf{C}$ represents the segment $\mathbf{S}_i$ in the grayscale colorspace and subdivided in many spatial regions. Now, from each one of the $\mathbf{C}_{j,k}$ sub segments we create the three input segments ($\mathbf{S}_{\text{fra}}$, $\mathbf{S}_{\text{res}}$, $\mathbf{S}_{\text{opt}}$) in the same way as we explained in Section 3.3.2. Then the three of them are sent into the CNN producing three scores which are then packed together before the final classification block. The final prediction value $p_{j,k}$ indicates if the sub segment $\mathbf{C}_{j,k}$ is seen as interpolated or not.
In particular:

$$p_{j,k} = \begin{cases} 0, & \text{if } \mathbf{C}_{j,k} \text{ is predicted not interpolated} \\ 1, & \text{if } \mathbf{C}_{j,k} \text{ is predicted interpolated} \end{cases}$$

3.6.2 Mask Creation

Our goal now is to assemble the information obtained from the sub segments predictions $p_{j,k}$ processing into one single element representing the localization of interpolation in the main segment $\mathbf{S}_i$. To do so, we create a 2D binary [6] sub mask $\mathbf{M}_{j,k}$ for each sub segment $\mathbf{C}_{j,k}$, which has a heigth of N and a width of N (the same dimensions as a sub segment cropped frame).
We fill these sub masks in the following way:

$$\mathbf{M}_{j,k} = \begin{bmatrix} 1-p_{j,k} & 1-p_{j,k} & 1-p_{j,k} & \dots & 1-p_{j,k} \\ 1-p_{j,k} & 1-p_{j,k} & 1-p_{j,k} & \dots & 1-p_{j,k} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1-p_{j,k} & 1-p_{j,k} & 1-p_{j,k} & \dots & 1-p_{j,k} \end{bmatrix} \tag{21}$$

So, if a sub segment $\mathbf{C}_{j,k}$ produces a prediction $p_{j,k} = 0$ it generates a $N \times N$ sub mask entirely composed by ones, otherwise by zeros. Now, we just need to assemble the mask related to the video segment $\mathbf{S}_i$ starting from the sub masks $\mathbf{M}_{j,k}$. The segment mask $\mathbf{M}_i$ is composed simply by concatenating the sub masks:

$$\mathbf{M}_i = \begin{bmatrix} \mathbf{M}_{0,0} & \mathbf{M}_{0,1} & \mathbf{M}_{0,2} & \dots & \mathbf{M}_{0,W-1} \\ \mathbf{M}_{1,0} & \mathbf{M}_{1,1} & \mathbf{M}_{1,2} & \dots & \mathbf{M}_{1,W-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \mathbf{M}_{J-1,0} & \mathbf{M}_{J-1,1} & \mathbf{M}_{J-1,2} & \dots & \mathbf{M}_{J-1,W-1} \end{bmatrix} \tag{22}$$

This results in a segment mask which has a total height of $N \cdot J$ and a total width of $N \cdot W$. $\mathbf{M}_i$ is composed by $N \times N$ black or white regions depending on the various sub segments predictions $p_{j,k}$.
The idea is to repeat this set of operations for every segment contained in $\mathbf{S}$, producing a mask for each one of them:

$$\mathbf{M} = [\mathbf{M}_0, \mathbf{M}_1, ..., \mathbf{M}_{B-1}], \quad B = \left\lfloor \frac{L}{T+1} \right\rfloor \tag{23}$$

In this way, $\mathbf{M}$ represents a marker of the (predicted) interpolated regions inside the whole video $\mathbf{V}_{\text{in}}$.

3.6.3 Localization Output Visualization

Finally, in order to realize an output to give a visible feedback, we use the produced mask $\mathbf{M}$ as a highlighter. In particular, for every segment $\mathbf{S}_i \in \mathbf{S}$, we multiply (element wise multiplication) the corresponding mask $\mathbf{M}_i \in \mathbf{M}$ for the Green and Blue components of each one of the frames $\mathbf{F}_j \in \mathbf{S}_i$.
Each frame of the segment becomes:

$$\mathbf{F}_j = [\mathbf{F}_j^R, \mathbf{F}_j^G \cdot M_i, \mathbf{F}_j^B \cdot M_i], \quad \forall j \in [0, T-1] \tag{24}$$

In this way, we leave the original frames unchanged in non interpolated regions (white portions of the mask), instead we suppress the Green and Blue components of the frames interpolated regions (black portions of the mask). This results in a video with the same number of frames as the original video, but for each video segment we have the interpolated regions highlighted through the Red channel. An example of this approach applied to one individual frame is presented in Figure 11.

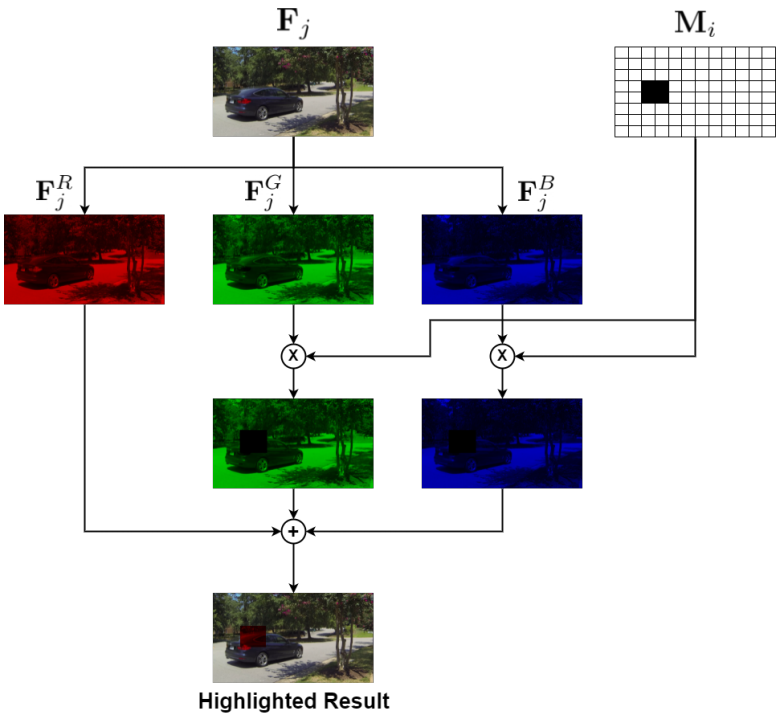


Figure 11: An example of highlighted result for a single frame $\mathbf{F}_j$ belonging to the segment $\mathbf{S}_i$, the mask associated to the segment is $\mathbf{M}_i$.

4. Experimental Setup

This section is devoted to discuss the experimental setup and the implementation details of the detector proposed in Section 3. First, we explain how we built the dataset to conduct our experiments. Then, we fully describe the training and testing procedures of the CNN and the SVM used by our detector. After that, we provide some details on the localization application introduced in Section 3.6.

4.1. Dataset

Our goal is to detect video frame-rate interpolation, so we need a dataset which is composed by a mixture of original and interpolated videos. In this way our detector can learn how to discriminate between them.
In this section we explore every aspect related to the composition of our dataset.
The creation procedure of the dataset is pictured in Figure 12.
In the following, we explain this procedure step by step.

4.1.1 Original Dataset

To build our dataset, we start from an already existing one, the Kinetics400 [9], which is available to download at [10]. This dataset which we call $\mathcal{K}_{400}$ is composed by a total of 298,945 videos downloaded from YouTube in the mp4 format, each of around 10s length and of variable frame-rate. It is worth noting that the original version of this dataset was composed by, 306,245 videos, but today some of the videos cannot be downloaded from YouTube anymore. $\mathcal{K}_{400}$ is widely used in the action recognition field since it contains 400 human action classes, with at least 400 video clips for each action. However, we are not interested in the video action class in our experiment, so the videos are all equivalent for us. The $\mathcal{K}_{400}$ is divided into three partitions: training ($\mathcal{K}_{400}^{\text{train}}$), testing ($\mathcal{K}_{400}^{\text{test}}$) and validation ($\mathcal{K}_{400}^{\text{val}}$). The number of videos contained in each partition is shown in Table 3.

	$\mathcal{K}_{400}$	$\mathcal{K}_{400}^{\text{train}}$	$\mathcal{K}_{400}^{\text{test}}$	$\mathcal{K}_{400}^{\text{val}}$
VIDEOS	298,945	240,258	38,684	19,881

Table 3: Train, test and validation partitions sizes of the original dataset $\mathcal{K}_{400}$.

This amount of videos is too high to have an acceptable computational time during the training phase of our

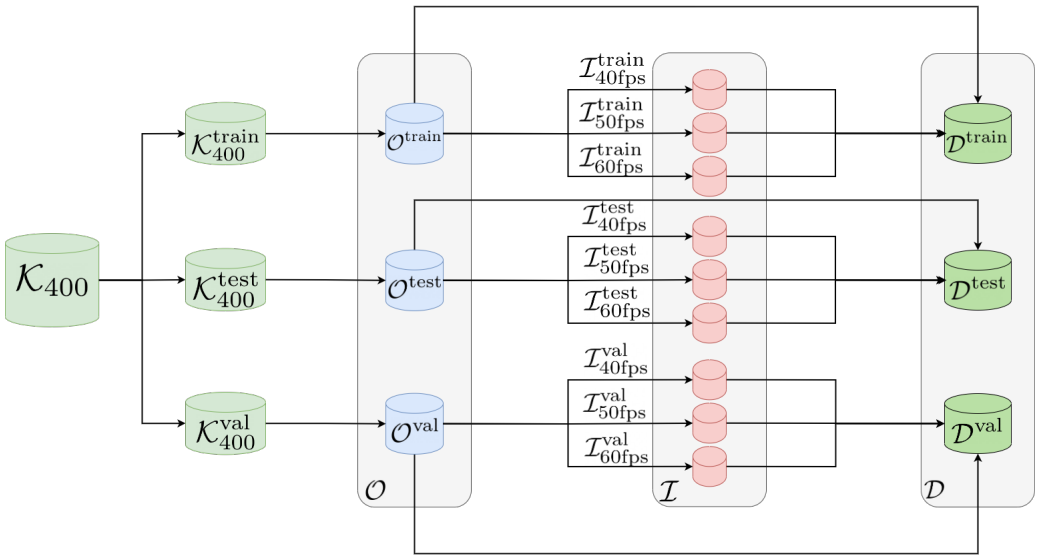


Figure 12: Dataset creation pipeline. We started from the Kinetics400 ($\mathcal{K}_{400}$) dataset partitions ($\mathcal{K}_{400}^{\text{train}}$, $\mathcal{K}_{400}^{\text{test}}$ and $\mathcal{K}_{400}^{\text{val}}$). We selected a part of the $\mathcal{K}_{400}$ videos to create the $\mathcal{O}$ dataset, composed by original videos. We interpolated those videos at different rates to build the dataset $\mathcal{I}$ populated by interpolated videos. Finally the experimental dataset $\mathcal{D}$ is obtained by merging $\mathcal{O}$ and $\mathcal{I}$ partitions.

	$\mathcal{O}$	$\mathcal{O}^{\text{train}}$	$\mathcal{O}^{\text{test}}$	$\mathcal{O}^{\text{val}}$
VIDEOS	42,947	35,224	3,791	3,932

Table 4: Train, test and validation partitions of the original videos dataset $\mathcal{O}$ after the split.

detector. Therefore, we decide to use just a subset of $\mathcal{K}_{400}$. From each partition of $\mathcal{K}_{400}$ we select a set of videos in order to compose a new dataset $\mathcal{O}$. This dataset is composed by the three training, testing and validation partitions called $\mathcal{O}^{\text{train}}$, $\mathcal{O}^{\text{test}}$ and $\mathcal{O}^{\text{val}}$. In particular, we randomly extract 17916 videos from $\mathcal{K}_{400}^{\text{train}}$ into $\mathcal{O}^{\text{train}}$, 1999 videos from $\mathcal{K}_{400}^{\text{test}}$ into $\mathcal{O}^{\text{test}}$ and 2000 videos from $\mathcal{K}_{400}^{\text{test}}$ into $\mathcal{O}^{\text{test}}$. The video extraction is made in a way that in the dataset $\mathcal{O}$ we have a balance of circa 80%, 10%, 10% respectively between the partitions $\mathcal{O}^{\text{train}}$, $\mathcal{O}^{\text{test}}$ and $\mathcal{O}^{\text{val}}$. Also, since in $\mathcal{K}_{400}$ some corrupted videos are present, we make sure that none of those are included in $\mathcal{O}$. All of the videos in the partition of $\mathcal{O}$ are obviously still of 10s length. In order to have a faster processing for each video, we decided to split them into smaller videos of duration length of 3s.
From the practical point of view, video trimming has been implemented using FFMPEG [11]. Specificaly, we executed the following command from on each video of the partitions of $\mathcal{O}$:

```
ffmpeg -i input.mp4 -c copy -map 0 -segment_time 00:00:03 -f segment -reset_timestamps 1
output%03d.mp4
```

where

- `"input.mp4"` is the input video we want to split
- `"-c copy"`, means that the videos resulting from the split are not re-encoded, basically they are just trimmed.
- `"-map 0"` indicates that we are interested in the video stream only.
- `"-segment_time"` 00:00:03 represents the minimum output video length that we accept.
- `"-f segment"` is the actual filter that segments the input video.
- `"reset_timestamps 1"` indicates that the video timestamps are recomputed for each one of the output videos.
- `"output%03d.mp4"` is the output videos name format, used to produce multiple videos from the input one.

With this command the video split is made in a way that each new video starts with a key frame. In practice we cut the input video in slices of 3s minimum always starting from a key frame.
The composition (after the splitting) of the partitions of $\mathcal{O}$ is showed in Table 4.
With these steps we have created the dataset $\mathcal{O}$ composed by its three partitions $\mathcal{O}^{\text{train}}$, $\mathcal{O}^{\text{test}}$ and $\mathcal{O}^{\text{val}}$, as shown in Figure 12.
As this is a dataset composed by original videos, our next step is to create interpolated videos from the videos in $\mathcal{O}$.

	$\mathcal{I}$	$\mathcal{I}^{\text{train}}_{\text{40fps}}$	$\mathcal{I}^{\text{train}}_{\text{50fps}}$	$\mathcal{I}^{\text{train}}_{\text{60fps}}$	$\mathcal{I}^{\text{test}}_{\text{40fps}}$	$\mathcal{I}^{\text{test}}_{\text{50fps}}$	$\mathcal{I}^{\text{test}}_{\text{60fps}}$	$\mathcal{I}^{\text{val}}_{\text{40fps}}$	$\mathcal{I}^{\text{val}}_{\text{50fps}}$	$\mathcal{I}^{\text{val}}_{\text{60fps}}$
VIDEOS	42,947	11,742	11,741	11,741	1,265	1,263	1,263	1,312	1,310	1,310

Table 5: Composition of the various partition of the dataset $\mathcal{I}$ composed by videos interpolated at different frame-rates (40fps, 50fps and 60fps).

	$\mathcal{D}$	$\mathcal{D}^{\text{train}}$	$\mathcal{D}^{\text{test}}$	$\mathcal{D}^{\text{val}}$
VIDEOS	85,948	70,448	7,942	7,864

Table 6: Train, test and validation partitions of the final experimental dataset $\mathcal{D}$.

4.1.2 Interpolated Videos Creation

The next step in the dataset creation is to produce some interpolated videos from the dataset $\mathcal{O}$. To this purpose, let us first explain what do we mean with interpolated video.
Suppose to have an original video $\mathbf{V}_O$ with frame-rate f and duration t. To interpolate this video and create the interpolated version $\mathbf{V}_I$ by a factor x means to produce x frames between every two adjacent frames of $\mathbf{V}_O$ by interpolating pixels among those adjacent frames. Then, if we want $\mathbf{V}_O$ and $\mathbf{V}_I$ to have both duration t we need to set the frame-rate of $\mathbf{V}_I$ to $f \cdot (x+1)$. In our scenario we consider just cases in which the frame-rate of the interpolated video is higher than the one of the original one (i.e., when $x \geq 1$, meaning that we are adding frames). In this way there is an actual interpolation, otherwise, instead of creating frames we would just drop them.
To create our interpolated videos we proceed in this way. We want our detector to be as robust as possible, so we select three frame rates named: $r_1 = 40fps$, $r_2 = 50fps$ and $r_3 = 60fps$. This because the majority of the videos inside $\mathcal{K}_{400}$ have a frame-rate between 25 and 30 fps, so with these three rates we are sure to increase the frame-rate with respect to the original video.
Our idea is to take the dataset $\mathcal{O}$ and create an equivalent dataset $\mathcal{I}$, in which each video from $\mathcal{O}$ is interpolated at fps r_1 or r_2 or r_3. This procedure is graphically represented in Figure 12. In particular, we start from each partition of $\mathcal{O}$, and we interpolate one third of the videos at r_1, and the other two thirds respectively at r_2 and r_3.
In this way we produce three partitions of $\mathcal{I}$ from each partition of $\mathcal{O}$. For example, from $\mathcal{O}^{\text{train}}$ we produce $\mathcal{I}^{\text{train}}_{\text{40fps}}$, $\mathcal{I}^{\text{train}}_{\text{50fps}}$ and $\mathcal{I}^{\text{train}}_{\text{60fps}}$.
The same holds for $\mathcal{O}^{\text{test}}$ producing $\mathcal{I}^{\text{test}}_{\text{40fps}}$, $\mathcal{I}^{\text{test}}_{\text{50fps}}$, $\mathcal{I}^{\text{test}}_{\text{60fps}}$, and also for $\mathcal{O}^{\text{val}}$ producing $\mathcal{I}^{\text{val}}_{\text{40fps}}$, $\mathcal{I}^{\text{val}}_{\text{50fps}}$ and $\mathcal{I}^{\text{val}}_{\text{60fps}}$.
The composition of the dataset $\mathcal{I}$ is sum up in Table 5. All of these videos are interpolated through FFMPEG, by the minterpolate filter [12]. This filter allows to convert a video to a specified frame-rate using MCI. Here is an example of how we interpolate videos with the following command:

```
ffmpeg -i input.mp4 -filter "minterpolate='fps=60':mi_mode=mci" output.mp4
```

In this command:

- `"input.mp4"` is the input video we want to interpolate
- `"-filter minterpolate"`, is the application of the minterpolate filter.
- `"fps=60"` indicates the output video fps.
- `"mi_mode=mci"` represents the MCI which is applied to produce the interpolated frames (all of our videos are generated with this option).
- `"output.mp4"` is the output (interpolated) video.

Finally, merging dataset $\mathcal{O}$ and dataset $\mathcal{I}$, we create the dataset $\mathcal{D}$ which we use to conduct our experiments. $\mathcal{D}$ is also composed by the three train ($\mathcal{D}^{\text{train}}$), test ($\mathcal{D}^{\text{test}}$) and validation ($\mathcal{D}^{\text{val}}$) partitions. Each of these partition is obtained by merging the relative partitions of $\mathcal{O}$ and $\mathcal{I}$. So, for example, $\mathcal{D}^{\text{train}}$ is composed by the videos inside: $\mathcal{O}^{\text{train}}$, $\mathcal{I}^{\text{train}}_{40}$, $\mathcal{I}^{\text{train}}_{50}$, and $\mathcal{I}^{\text{train}}_{60}$. The same holds for $\mathcal{D}^{\text{test}}$ and $\mathcal{D}^{\text{val}}$ respectively, as pictured in Figure 12. $\mathcal{D}$ composition is summarized in Table 6.

4.2. Experimental Setup

This section is mainly devoted to discuss about the details and the tools behind the implementation of the model proposed in Section 3. First, we give an accurate description of the preprocessing operations seen in Section 3.2, specifying every parameter. Then we take a close look at the training and testing phases regarding the CNN and the SVM inside our detector. In the end we also analyze the principal features regarding the interpolation localization experiment proposed in Section 3.6.
To conduct all of our experiments we used the Python [13] programming language. This is because it is very

simple and efficient when dealing with images and videos. In the following sections we also give a description about the Python libraries and functions that we adopted during the implementation of our method.

4.2.1 Preprocessing

As seen in Section 3.2, the first operation to do on every video from our dataset $\mathcal{D}$ is to extract some consecutive frames. In particular, from each video $\mathbf{V} \in \mathcal{D}$, we extract a segment $\mathbf{S}$ of $T+1$ frames, with $T = 16$ (to be precise we extract the first 17 frames and obviously, we discard videos with less than 17 frames). To do so, we rely on a Python library named OpenCV [14]. OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library, which is very useful in video processing applications. In particular we make use of the `VideoCapture` class and the `VideoCapture.read()` function (OpenCV documentation can be found at [15]), with these we can easily open a video and extract its frames. In our application, a segment of frames is handled through a numpy array (from the Numpy library [16]). Also, each frame in the segment is contained in a numpy array itself with the pixel values ranging from 0 (minimum intensity) to 1 (maximum intensity).
After the frames extraction, still using OpenCV, we convert all the frames belonging to $\mathbf{S}$ from the RGB colorspace to the grayscale colorspace producing the segment $\mathbf{S}_{\mathbf{gray}}$. In this operation we utilize the function `cvtColor()` which consents to change the colorspace of a specified input image.
Now, according to what we described in Section 3.2, we need to resize the frames of the $\mathbf{S}_{gray}$ segment. To do that, we proceed in two different ways, depending if we are in the training or in the testing (and validation) phases of our detector. To perform the resizing operations we rely on a Python library called Albumentations [17]. Albumentations is a library to implement fast and flexible image augmentations, more details can be found in this paper [18].
Let us look at the differences in the resizing policies between the training and testing phases.

Training Resizing Obviously, the training resizing operations just involve every grayscale video segment $\mathbf{S}_{gray}$ produced from each input video $\mathbf{V} \in \mathcal{D}^{\text{train}} \in \mathcal{D}$. In other words, we focus on the training partition of $\mathcal{D}$. As explained in Section 3.3.1, the first step is to apply a light resizement. All the frames of the $\mathbf{S}_{gray}$ segment are resized to a height of $r \cdot H$ and to a width $r \cdot W$, where r is a random resize factor picked in the range between 0.9 and 1.1. In this way we produce the resized segment $\mathbf{S}_{rsz}$. This operation is executed through the `Resize` class of the Albumentations library.
Then, from all the frames in $\mathbf{S}_{rsz}$ we extract a $N \times N$ center crop, with $N = 224$ pixels, producing the segment $\mathbf{S}_{N\times N}$. Also this operation is realized through Albumentations, in particular using the `CenterCrop` class.

Testing and Validation Resizing For the segments belonging to the videos $\mathbf{V} \in (\mathcal{D}^{\text{test}}, \mathcal{D}^{\text{val}}) \in \mathcal{D}$, the resizing procedure is a little different as discussed in Section 3.3.1. From the frames of $\mathbf{S}_{gray}$ we just extract a $N \times N$ center crop with $N = 224$ pixels, in the exact same way as we do in training, producing the segment $\mathbf{S}_{N\times N}$.

The last thing to do in the preprocessing stage, is to create the three input segments of $T = 16$ frames length starting from $\mathbf{S}_{N\times N}$, as introduced in Section 3.3.2. This procedure is equivalent for all the videos of all the partitions of $\mathcal{D}$.

Original Frames Segment This segment $\mathbf{S}_{\text{fra}}$ is already equivalent to the segment $\mathbf{S}_{N\times N}$, so no further actions are required.

Residuals Segment In order to obtain the residual segment $\mathbf{S}_{\text{res}}$, we saw in Section 3.3.2 that a simple frame by frame difference is enough. Our idea is so to calculate the difference between two numpy arrays. The first one contains the frames of $\mathbf{S}_{N\times N}$ from the second to the last. The second one contains the frames of $\mathbf{S}_{N\times N}$ from the first to the second last. Subtracting element by element the second array to the first one and then taking the absolute value of the result leads us to the segment $\mathbf{S}_{\text{res}}$.
In Figure 13 this approach is summarized.

Optical Flow Segment As discussed in Section 3.3.2, to get to the optical flow segment $\mathbf{S}_{\text{opt}}$, we follow the Farneback method. The idea is to calculate the flow between every two consecutive frames in the segment $\mathbf{S}_{N\times N}$. In this specific optical flow computation, it is required to convert each frame pixel value into the range from 0 to 255 (8-bit image), after the flow calculation we convert back to the range between 0 and 1. First, we define the mask $\mathbf{M}_{\text{opt}}$ composed by the three channels each of $N \times N$ dimension, in the HSV colorspace: $\mathbf{M}^h_{\text{opt}}$, $\mathbf{M}^s_{\text{opt}}$ and $\mathbf{M}^v_{\text{opt}}$. We initialize the whole hue and value channels (first and last) with all zeros, instead the whole saturation channel with the maximum saturation (255). Now, considering two generic consecutive frames of

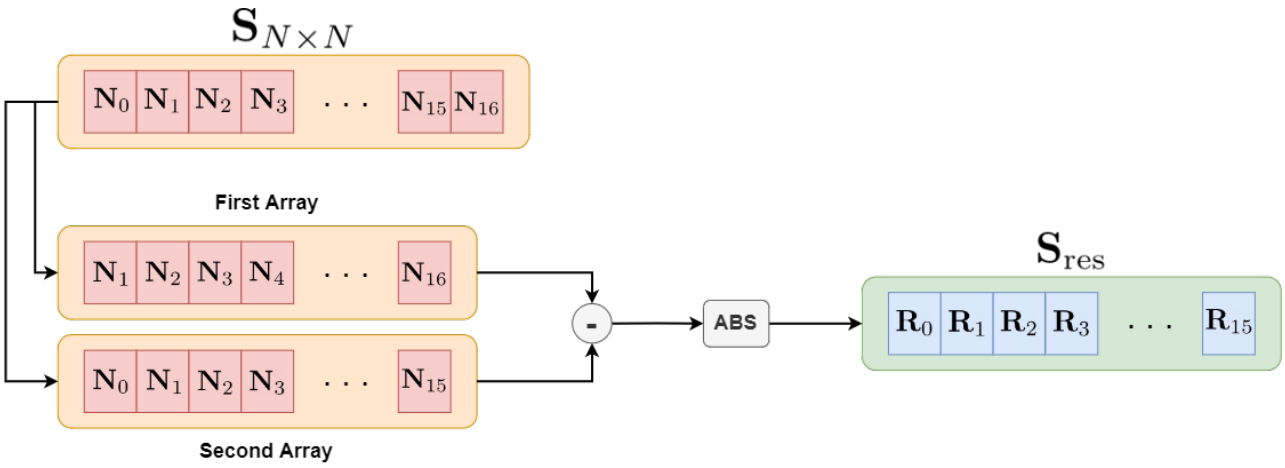


Figure 13: Residuals segment creation approach. The residuals segment $\mathbf{S}_{\text{res}}$ is produced by the subtraction between First arrray (from second to last element of the preprocessed segment $\mathbf{S}_{N\times N}$) and Second array (from first to second last element of the preprocessed segment $\mathbf{S}_{N\times N}$). Then the absolute value of the pixels value is taken.

$\mathbf{S}_{N\times N}$ (named $\mathbf{F}_i$ and $\mathbf{F}_j$), the flow $\mathbf{O}_{ij}$ consists in a two channel array containing the 2D optical flow vectors. This flow is produced through the OpenCV library (a sample of code is available here [19]), using the function `calcOpticalFlowFarneback()`, which receives as input:

- `prev`: previous grayscale frame ($\mathbf{F}_i$)
- `next`: actual grayscale frame ($\mathbf{F}_j$)
- `flow`: None
- `pyr_scale`:0.5
- `levels`:3
- `winsize`:15
- `iterations`:3
- `poly_n`:5
- `poly_sigma`:1.2
- `flags`:0

For a detailed description of all the parameters, please refer to [20].
We then calculate the magnitude ρ_{ij} and direction $\mathbf{\Phi}_{ij}$ of the 2D vectors by the OpenCV function `cartToPolar()`, which receives as input the two channels of the computed flow $\mathbf{O}_{ij}$. After that, we use $\mathbf{\Phi}_{ij}$ and ρ_{ij} (normalized in the range between 0 and 255 through the OpenCV function `normalize()`) respectively to set the hue and value channels of the mask $\mathbf{M}_{\text{opt}}$ according to Equation 9 and Equation 10. Now, the mask $\mathbf{M}_{\text{opt}}$ is a representation of the dense optical flow between $\mathbf{F}_i$ and $\mathbf{F}_j$. We convert the mask back to the grayscale colorspace (again using the OpenCV function `cvtColor()`) and also we convert the pixel values back to the range between 0 and 1.
Repeating this set of operations for all the consecutive frames in $\mathbf{S}_{N\times N}$ and collecting the results $\mathbf{M}_{\text{opt}}$ we get our optical flow segment $\mathbf{S}_{\text{opt}}$.

4.2.2 Training Settings

In this section, we explore the training phase of our detector, differentiating between the two principal components: CNN and SVM.
For the training and the construction of the CNN, we choose to use PyTorch [21]. PyTorch is an open source machine learning framework that accelerates the path from research prototyping to production deployment.
Instead, for the training of the SVM, we used Sci-kit learn [22]. Scikit-learn is an open source machine learning library that supports supervised and unsupervised learning. It also provides various tools for model fitting, data preprocessing, model selection, model evaluation, and many other utilities.
Now, let us analyze the main fratures related to the training of these two elements.

CNN training settings Since our CNN is composed by three distinct branches (Figure 4), and each one of them has to process a different kind of input, we decide to train the branches individually. This decision is taken in order to evaluate the prediction accuracy of the single branches which we compare in the next section. Once trained, the three branches are then aggregated.
We realize an individual PyTorch model for each branch (the structure is equivalent for each one of them

according to Table 1). Each branch model is trained for $e = 15$ epochs on the videos from the $\mathcal{D}^{\text{train}}$ partition of the dataset $\mathcal{D}$. At the end of each epoch, we validate the branch model through the videos from the $\mathcal{D}^{\text{val}}$ partition of the dataset $\mathcal{D}$. We train our branches in a supervised training fashion, meaning that each video is labeled (0 for original videos from $\mathcal{O}$, 1 for interpolated videos from $\mathcal{I}$). The weights of the layers composing a branch model are updated through an Adam optimizer [23] with an initial learning rate $lr = 0.01$ and through a Binary Cross Entropy loss (with logits) [24]. Also, we use a learning rate scheduler in order to adjust the learning rate of our branch models based on the average loss calculated during the validation phase. The scheduler reduces the learning rate if the validation loss does not decrease for three epochs straight.
During each epoch, the videos from $\mathcal{D}^{\text{train}}$ are extracted in batches of 32 videos each. This extraction is performed in a way that the 32 videos are composed by 16 original videos plus the 16 interpolated versions of those videos. These videos are then preprocessed as we described in Section 4.2.1, producing the video segment related to the specific branch input type. So, if we are training the frames branch, from the 32 videos in each batch we produce 32 original frames segments ($\mathbf{S}_{\text{fra}}$). For each batch, we then create two tensors:

- $\mathbf{X}$, containing the 32 video segments related to the branch input type. Its size is [32, 1, 16, 224, 224]. Where 32 is the batch size (number of total segments), 1 is the number of channels of each frame (grayscale), 16 is the number of frames in each segment and 224, 224 is the spatial dimension of the segments frames
- $\mathbf{Y}$, containing the labels associated to the 32 segments.

About the segments tensor:

$$\mathbf{X} = [\mathbf{x}_0, \mathbf{x}_1, ..., \mathbf{x}_{31}] \tag{25}$$

Where the generic $\mathbf{x}_i$, with $i \in [0, 31]$ is the i-th preprocessed video segment in the batch. For example, if we are training the original frames branch model, in $\mathbf{X}$ we have 32 video segment composed by the preprocessed original frames.
About the labels tensor:

$$\mathbf{Y} = [y_0, y_1, ..., y_{31}] \tag{26}$$

More precisely:

$$\begin{cases} y_i = 0, & \text{if } i\text{-th segment in } \mathbf{X} \ (\mathbf{x}_i) \text{ is from an original video} \\ y_i = 1, & \text{if } i\text{-th segment in } \mathbf{X} \ (\mathbf{x}_i) \text{ is from an interpolated video} \end{cases} \tag{27}$$

Where y_i is a generic label related to the i-th segment contained in $\mathbf{X}$ ($\mathbf{x}_i$), with $i \in [0, 31]$.
We feed $\mathbf{X}$ to the branch model, which after processing the data through its layers, produces the actual logits **lgt** as output.
The logits contain the output score of the branch model for all of the 32 segments in the batch:

$$\mathbf{lgt} = [lgt_0, lgt_1, ..., lgt_{31}] \tag{28}$$

Where:

$$\begin{cases} lgt_i < 0, & \text{if } i\text{-th segment in } \mathbf{X} \ (\mathbf{x}_i) \text{ is predicted not interpolated} \\ lgt_i > 0, & \text{if } i\text{-th segment in } \mathbf{X} \ (\mathbf{x}_i) \text{ is predicted interpolated} \end{cases} \tag{29}$$

With $i \in [0, 31]$.
The logits tensor **lgt** along with the labels tensor $\mathbf{Y}$ are then used to compute the loss function. As said before, we use a Binary cross entropy loss. The loss computation is performed through the class `BCEWithLogitsLoss` from the PyTorch `torch.nn` module (more details at [25]).
The approach of the `BCEWithLogitsLoss` class is to calculate the individual losses for each segment in the batch:

$$l_i = -\omega_n \cdot [y_i \cdot \log(\sigma(lgt_i)) + (1 - y_i) \cdot \log(1 - \sigma(lgt_i))], \quad i \in [0, 31] \tag{30}$$

Where ω_n is set to 1, and σ means the application of a Sigmoid layer [26] to lgt_i:

$$\sigma(lgt_i) = \frac{1}{1 + \exp^{-lgt_i}} \tag{31}$$

After the computation of the individual losses we build the total loss (related to the current batch) $\mathbf{L_{BCE}}$ as:

$$\mathbf{L}_{BCE} = [l_0, l_1, ..., l_{31}] \tag{32}$$

Finally, we get the loss l by averaging between the individual losses:

$$l = \frac{\sum_{i=0}^{31} l_i}{32} \tag{33}$$

We backpropagate l to update the branch model weights. This procedure is obviously repeated for each batch. At the end of each epoch, we validate our branch model by extracting videos from $\mathcal{D}^{\text{val}}$ in batches of 32 videos each. Basically we repeat the loss calculation procedure as done in training, but the resulting loss is used as input for the learning rate scheduler. In this way the learning rate can be changed according to the validation loss value.
The same operations are repeated for each epoch.
This process is applied to train and validate each individual branch model.

SVM training settings Once the three branch models are all trained, we connect them before the classifier stage (as in Figure 1). This classifier has the goal of taking the three output scores from the CNN branches and generate a prediction for the relative input segment.
We choose to rely on a SVM able to discriminate between two classes (0 for original video segments, 1 for interpolated video segments) based on the analysis of the aggregated CNN output scores. In particular we implement a Support Vector Classifier (SVC) through the `SVC` class from the module `svm` of the Python library Sci-kit learn. We use a SVC with a rbf kernel [27].
The training procedure is very simple.
For each video from $\mathcal{D}^{\text{train}}$ (after preprocessing it according to Section 4.2.1), we compute the CNN output scores o_1, o_2 and o_3 as seen in Section 3.4. These scores are then packed together in the list $\mathbf{o}$ (see Equation 14). We also store the video segment label (remember that is 0 for a segment from an original video, 1 otherwise). So, for all the videos in $\mathcal{D}^{\text{train}}$, we collect all the labels into a list lbl_{tot} and all the packed scores $\mathbf{o}$ into a list sco_{tot}. Finally we feed lbl_{tot} and sco_{tot} to the `svm.SVC.train()` function which finds a boundary between the two classes based on the received labels and scores.
In this way, we are able to predict the belonging class of a video segment by analyzing the three branch scores produced from an input video segment.

4.2.3 Testing Settings

In this section we describe the main settings related to the testing phase of the detector. As done for the training phase, we test the CNN and the SVM in separate ways. First, we test the trained individual branch models (composing our CNN), to have a clear view on how they perform in identifying interpolation traces. Then we also test the SVM, evaluating the global performance of our detector.
Comparisons between the branch models and the global model performances are discussed in the next section. Obviously, the testing phase involves the videos from the dataset $\mathcal{D}$ test partition $\mathcal{D}^{\text{test}}$.

CNN testing settings The test for each branch model is carried on in the following way. We extract videos from $\mathcal{D}^{\text{test}}$ in batches of 32 videos each. We preprocess those videos according to Section 4.2.1 (remember that training and testing preprocessing have a different resizing policy). Then, we create the segments and labels tensors $\mathbf{X}$ and $\mathbf{Y}$ in the exact same way as in the training phase (see Section 4.2.2, in particular Equation 25 and Equation 26). We feed $\mathbf{X}$ to the branch model producing the logits $\mathbf{lgt}$. After that, we apply a sigmoid layer onto the logits (as in Equation 31), in this way we transform each logit $lgt_i \in \mathbf{lgt}$ in the probability $\sigma(lgt_i)$. This is the probability that the segment corresponding to the i-th ($\mathbf{x}_i \in \mathbf{X}$) segment from the batch belongs to class 1 (interpolated segment). From these probabilities we compute the segments predictions:

$$\begin{cases} p_i = 0, & \text{if } \sigma(lgt_i) < 0.5 \\ p_i = 1, & \text{if } \sigma(lgt_i) >= 0.5 \end{cases} \tag{34}$$

With $i \in [0, 31]$ and with p_i representing the prediction related to the i-th video segment in the batch. Now we just compare the predictions p_i to the respective labels $y_i \in \mathbf{Y}$ (labels values is indicated in Equation 27) for each segment in the batch to find how many predictions are correct.
We repeat this procedure for each batch and compute the final testing accuracy dividing the total number of correct predictions (among the various batches) for the total number of videos in $\mathcal{D}^{\text{test}}$.

SVM testing settings With the testing of the SVM we intend to evaluate the whole detector performances. For each video from $\mathcal{D}^{\text{test}}$ we follow this procedure. First, we preprocess the video (according to Section 4.2.1). Then, we feed the three video segments created in the preprocessing stage ($\mathbf{S}_{\text{fra}}$, $\mathbf{S}_{\text{res}}$ and $\mathbf{S}_{\text{opt}}$) to the three CNN branches (obviously already trained) obtaining the three scores o_1, o_2 and o_3. We collect these scores in $\mathbf{o}$ (as in Equation 14). After that, $\mathbf{o}$ is fed into the trained SVC which predicts the belonging class of the video segment extracted from the video. We repeat this procedure for each video, confronting the video labels with the obtained predictions. We compute the number of correct predictions and then we divide it for the total number of videos from $\mathcal{D}^{\text{test}}$ obtaining the final testing accuracy.

4.2.4 Localization

For what concerns the application of the detector to the localization problem (exposed in Section 3.6), here we discuss the details of our implementation.
To conduct this experiment, we need to have already trained the detector, from the individual branches to the classifier.
Since our goal here is to localize interpolation inside a video, the idea is to create a new dataset containing videos that are locally interpolated, in a way that we can localize the interpolation in them.
So far, we dealt with videos in which the interpolated frames were fully interpolated (the videos from $\mathcal{I}$). Now, we create this new dataset $\mathcal{I}_{loc}^{\text{test}}$. This dataset contains all of the videos from the test partitions of $\mathcal{I}$ (so from $\mathcal{I}_{40\text{fps}}^{\text{test}}$, $\mathcal{I}_{50\text{fps}}^{\text{test}}$ and $\mathcal{I}_{60\text{fps}}^{\text{test}}$) but in a locally interpolated fashion. To locally interpolate a video we proceed in the following way.
For each video from the test partitions of $\mathcal{I}$, we take the original version of it (residing in $\mathcal{O}^{\text{test}}$). We select a $N \times N$ spatial region, with $N = 224$, and for each frame of the original video, we substitute the selected region with the exact same region but taken from the interpolated version of the video. For simplicity, but without loss of generality, the selected region is the same for each video and corresponds to the first 224×224 block in the upper left corner of the frames. Despite this may seem not realistic in a real-world splicing application, this choice makes sense in the testing scenario. An actual malicious user would likely splice a region from an alien video into the forged video (e.g., to insert some objects). However, this operation may introduce additional artifacts, and not just interpolation ones. With our strategy, we create the most challenging forgery for our localization technique, as the interpolated region cannot be distinguished by visual inspection, and only frame-rate interpolation traces are present.
The videos created with this procedure are collected in the $\mathcal{I}_{loc}^{\text{test}}$ dataset.
In Figure 14 an example of the creation of the locally interpolated video frames is represented. In the example, the level of motion in the interpolated area is low, so the original and the locally interpolated frames are really hard to distinguish.
Now we can conduct our localization experiment on the videos from the new dataset $\mathcal{I}_{loc}^{\text{test}}$.
The goal of this experiment is to start from locally interpolated videos and to produce output videos in which the interpolated regions are highlighted.
So, the idea is to process each video from $\mathcal{I}_{loc}^{\text{test}}$ in the exact same way as explained in Section 3.6.
Let us consider this procedure for a single video from $\mathcal{I}_{loc}^{\text{test}}$.
We divide the video in segments of $T + 1$ frames, with $T = 16$ (discarding the video if it has less than $T + 1$ frames). Focusing on a single segment, we subdivide it into subsegments of the same $T+1$ frames as the segment, but with a spatial dimension of 22×224 pixels. Each of this subsegments represents the $T + 1$ frames of the segment but taken in a different $N \times N$ spatial region. In practice, these subsegments are spatial subdivision in $N \times N$ non overlapping blocks of the $T+1$ frames in the segment. One by one, we preprocess these subsegments (applying the testing resizing policy as in Section 3.3.1) generating the three input segments($\mathbf{S}_{\text{fra}}$, $\mathbf{S}_{\text{res}}$ and $\mathbf{S}_{\text{opt}}$). We feed these three segments to our detector which computes a prediction related to the subsegment. As explained in Section 3.6 we create the subsegment submask starting from the subsegment prediction (see Equation 21). We repeat the procedure for each subsegment created from the segment, and we concatenate the resulting masks to compose the segment mask (see Equation 22). This one indicates which regions are interpolated inside the segment. Then, we can do the same for all the segments inside the video, collecting all the segment masks (see Equation 23). Finally, following the procedure indicated in Section 3.6.3, we can realize the output video relying on the generated segment masks.
So, we follow the previous steps for each video from $\mathcal{I}_{loc}^{\text{test}}$ producing the highlighted output videos.
Also, during this experiment, we want to measure the subsegments prediction accuracy of our detector. In particular this accuracy is obtained by dividing the number of correct subsegment predictions for the total number of analyzed subsegments (among all segments of all videos from $\mathcal{I}_{loc}^{\text{test}}$).
In Section 5.5 we show some results in terms of output videos and localization accuracy.

5. Results

This section is devoted to the discuss the results of our work. More specifically, we analyze the results of our detector, not just globally but also in terms of single branches. We investigate on the impact of different resizing policies that can be applied to the video segments. Also we compare our results against some previous works in this very same field. Finally, we present some localization results.

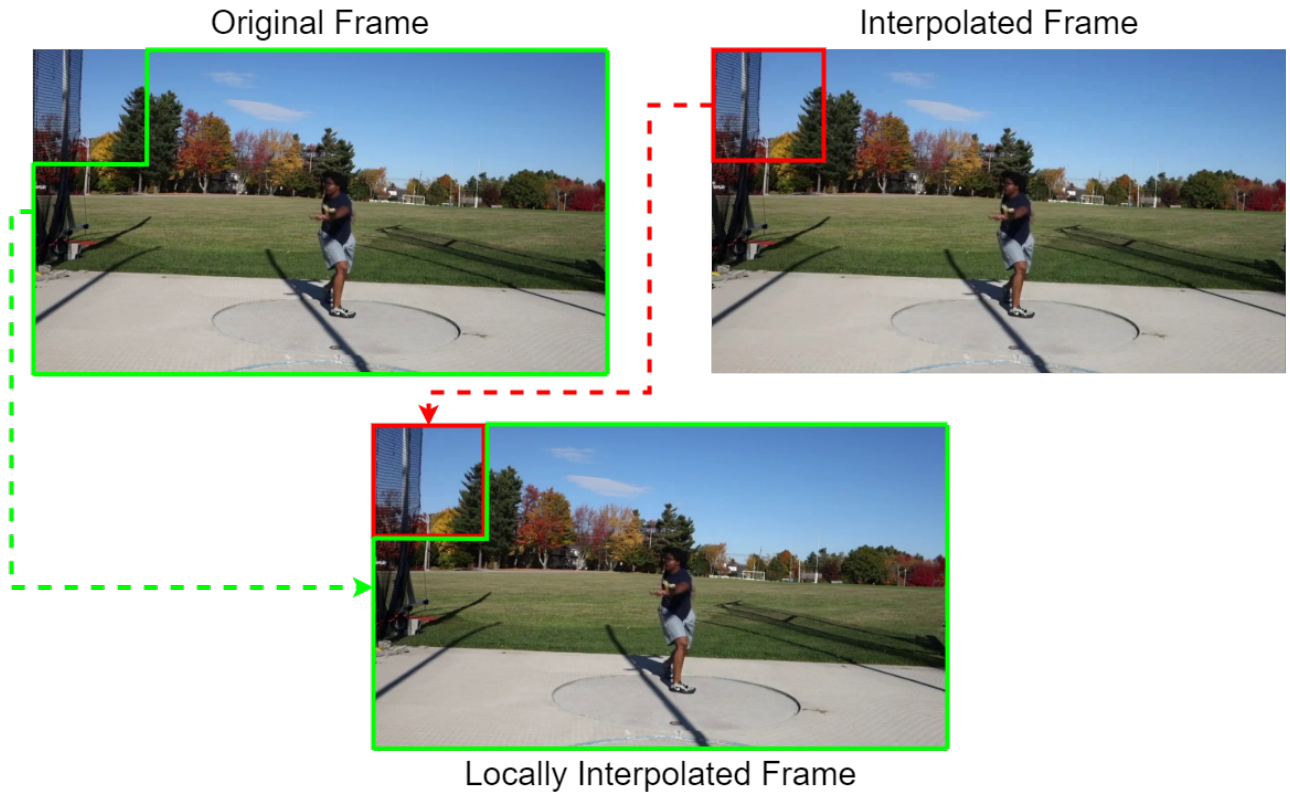


Figure 14: An example of the construction of a locally interpolated video frame. The upper left 224 x 224 block of the original frame is replaced with the same block from an equivalent but interpolated frame.

BRANCH MODEL	**RESIZING POLICY**	
	SpeedNet	*Our*
Original Frames	60.511%	94.503%
Residuals	98.224%	99.205%
Optical Flow	/	96.435%

Table 7: Comparison between the branch models testing accuracy values in the two alternative resizing policy approaches.

5.1. Frames Resizing Comparison

We explained the adopted resizing policy in Section 3.3.1. Here we show the comparison against another resizing approach. This alternative approach still differentiate between the training and testing phases of our detector.

Training alternative resizing In the training phase, the frames of a video segment are resized to a height and a width of N, where N is an integer value between 64 and 336. So, here the dimension of the frames is different from the 224 x 224 that we use in the other policy. This is possible because our CNN branch models are fully connected, and independently from the input segment size (in terms of spatial dimension and number of frames), it packs everything to one single output value.

Testing alternative resizing In the testing phase, the frames of a video segment are resized to a height of 224 pixels but maintaining the aspect ratio of the original frames. Then a 224 x 224 center crop is extracted.

This resizing policy is also used in the Google SpeedNet [5] implementation. Since SpeedNet shares a consistent part of its network architecture with our detector, we decided to try its resizing method. In Table 7 a comparison between the resizing policies is made. In the table are shown the accuracy levels for the individual branch models depending on the applied resizing policy. Since it is evident that our resizing policy is the best in the original frames and residuals cases, we have not tested it for the optical flow model.

		TRAINING		
		Original Frames	*Residuals*	*Optical Flow*
TESTING	*Original Frames*	94.503%	49.986%	50.071%
	Residuals	50.426%	99.205%	55.312%
	Optical Flow	59.801%	50.298%	96.435%

Table 8: Testing accuracies of the cross-test between the three branch models.

OTHER FPS (SINGLE BRANCH)		
Original Frames	*Residuals*	*Optical Flow*
99.565%	99.565%	98.625%

Table 9: Evaluation of the branch models on fps unseen during training. Each model has been tested on 300 randomly selected videos from $\mathcal{O}$ interpolated at 70, 80 and 90 fps. In the table are indicated the testing accuracies.

5.2. Single Branches Scores and Comparison

A first glance of the testing accuracy obtained by each branch model is shown in Table 7. We can see that the model trained on the residuals segments is the one which performs better (independently from the resizing policy).
Another comparison between the branch models is represented in Table 8. In this table, we show how each branch model performs when trained on a segment type and tested on another. Obviously, training and testing on the same segment type gives the best results. In Table 9 is represented another test that we made. In particular, we selected 300 videos from the $\mathcal{K}_{400}$ and we interpolated 100 of them at 70, 100 of them 80 and 100 of them 90 fps, so at rates not considered in the training phase.
In Table 9 the testing accuracies are presented.

5.3. Global Network Performance

To evaluate the global performances of our detector, we need to focus on the classification stage. As discussed in Section 4.2.2, we implemented a SVC to classify the aggregated scores of the three branches. Two alternatives have also been tested. In Table 10, the testing accuracies related to the three approaches are shown.
In particular, the linear layer has the purpose of generating one single score starting from the three branch outcomes for each video segment, then a sigmoid is applied to transform this score into a probability (which is then rounded to the nearest integer to get the final class, 0 or 1). The simple average consists in averaging together the three branch outcomes, and applying a sigmoid layer to transform this average into a probability (which is then rounded to the nearest integer to get the final class, 0 or 1).
Another test has been conducted similarly to what we did in the single branch evaluation. This is related to the testing of interpolated videos at rates unseen during the training phase (on the exact same 300 videos as the single branch cases). In Table 11 the testing accuracies are shown.
Table 9 and Table 11 show that our model is very efficient in detecting interpolation in rates not considered during training.

5.4. Comparison Against the State of the Art

So far we showed the results of our experiments, now let us compare our detector against some of the already existing works in the video interpolation field.
In this section we focus on two recent works:

CLASSIFIER TYPE		
SVC	*LINEAR LAYER*	*SIMPLE AVERAGE*
99.630%	99.516%	99.544%

Table 10: Comparison between the testing accuracies obtained from the different types of classification approaches.

OTHER FPS (GLOBAL)		
SVC	*LINEAR LAYER*	*SIMPLE AVERAGE*
100%	100%	100%

Table 11: Testing accuracies of each classification alternative on 300 randomly selected videos from $\mathcal{O}$ interpolated at 70, 80 and 90 fps.

METHODS COMPARISON		
Our	*SpeedNet*	*S&H*
99.630%	75.600%	91.700%

Table 12: Comparison between our detector against 'SpeedNet' and 'Stamm and Hosler' detector (S&H).

- SpeedNet
- Stamm and Hosler speed manipulation detector

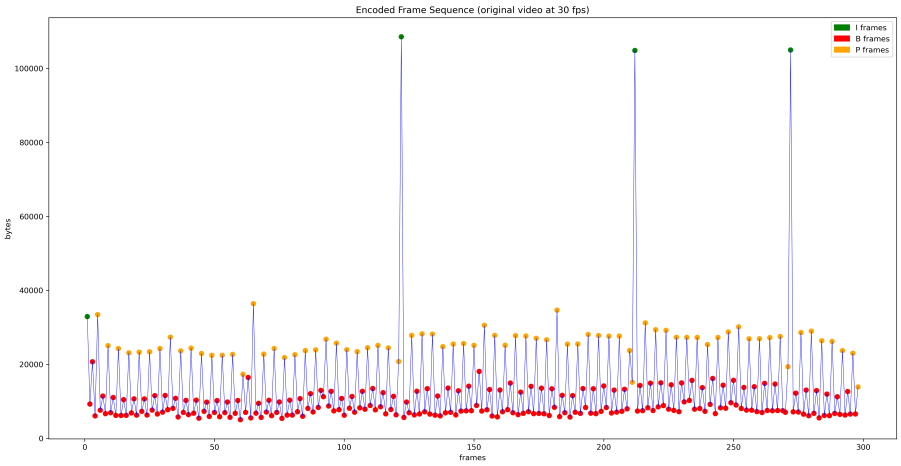
Below, give a brief introduction on these two methods.

SpeedNet We already mentioned SpeedNet because its network structure is very similar to our detector branch models. The details about this work can be found at [5].
In a few words, the goal of SpeedNet is to automatically predict the "speediness" of moving objects in a video. As in our case, it is a binary classification problem.
A video segment is extracted from an input video, it is then fed to the SpeedNet CNN and then it is predicted as 'normal speed' or 'sped up'.
So, SpeedNet does not directly deal with interpolated videos, but with sped up videos. Anyway it is almost the same concept, because speeding up a video means upsampling a video inserting new frames between already existing ones (these new frames can either be interpolated or just copies of adjacent frames). These frames are then red with the same frame rate as for the original video, resulting in a reduced duration.
Another thing in common with our experiment, is that SpeedNet is also trained on videos from the Kinetics dataset, the only difference is that they use a lot more videos than us (246,000 for training and 50,000 for testing).

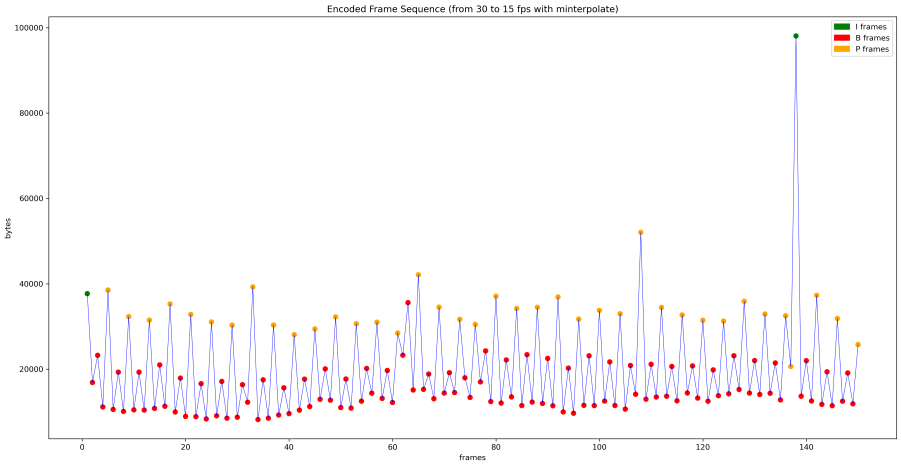
Stamm and Hosler speed manipulation detector Like SpeedNet, this work developed by Brian C. Hosler and Matthew C. Stamm is also related to video speed manipulation detection.
The problem is still a binary classification one, given an input video, the goal is to predict if it has undergone speed manipulation or not. In this paper [4] the implementation of the detector is discussed.
The aim of this work is to detect video speed manipulation in a video and to estimate the rate by which the video's speed has been modified. In particular, they identify a trace left by video speed manipulation inside the encoded frame size sequence (EFS).
The EFS sequence is composed by the number of bytes used to encode each frame in the video. In Figure 15 we show an example of three different EFS sequences: for an original video at 30fps and for the same video but interpolated at 15fps and 60fps. For example, the version at 60fps of the original video consists in inserting a new (interpolated frame) between two every consecutive frames in the original video. In their work, the authors demonstrate that the encoding size in bytes associated to the P (red in Figure 15) and B (orange in Figure 15) frames decreases for interpolated videos. This because, the information in the interpolated frames can be obtained from adjacent frames, so the encoding size of the interpolated frames decreases.
The EFS sequence is then processed and fed into a SVM to state if a video has been altered. The SVM is trained with the processed EFS sequences of 372 videos from the DFDC dataset [28], plus the 372 altered versions for each one of the speed manipulation rates that they used (for a total of 6334 videos).
Now, let us look at a comparison between our detector and the above mentioned methods.
In Table 12

5.5. Interpolation Localization Results

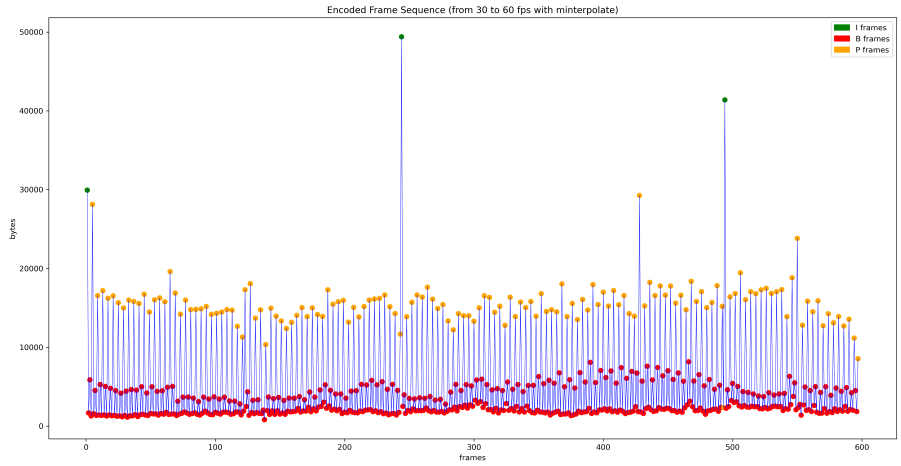
In this section we report some results concerning the localization experiment. As said in Section 4.2.4, the goal of the localization experiment was to produce output videos in which the interpolation zones were highlighted for each segment inside an input video.
Also the secondary goal was to measure the subsegments prediction accuracy.



(a) 30fps (original) video EFS sequence



(b) 15fps (interpolated) video EFS sequence



(c) 60fps (interpolated) video EFS sequence

Figure 15: Three EFS sequences produced from three versions of the same video. For each diagram, on the X-axis we represent the frames, on the Y-axis we indicate the encoded frame byte size. I-frames, B-frames and P-frames are indicated respectively in green, red and orange.

We can state that during the experiment, the 98.942% of the subsegments were correctly detected either as original or interpolated.
In Figure 14 a representation of the creation of the locally interpolated video frames is shown. Now let us look at the output video structure. In Figure 16 the first three frames of three different segments from an output video are shown.
We remind that the interpolation zone for every input video is located in the first 224x224 block in the upper left corner of each frame.
So, for the frames pictured in Figure 16 the localization is perfect.

(a) segment 1, first frame (b) segment 2, first frame (c) segment 3, first frame

Figure 16: Samples of frames from three random video segments from an output video. Analysis performed with subsegments of dimension 224x224 and 16 frames segments. The predicted interpolated sectors are marked in red.

Remember that this analysis is performed with subsegment of dimension 224x224 and with segments of 16 frames each, so we are in the exact same condition as the one adopted in the training phase.
To vary things a little bit, we also show some examples in which the analysis parameters are different from the training ones.
In Figure 17 the same three frames from the same three segments (from Figure 16) are shown, but this time the analysis has been performed with subsegments of dimension 128x128 and on 16 frames segments.

(a) segment 1, first frame (b) segment 2, first frame (c) segment 3, first frame

Figure 17: Samples of frames from three random video segments from an output video. Analysis performed with subsegments of dimension 224x224 and 16 frames segments. The predicted interpolated sectors are marked in red.

As we can see from Figure 17 the localization is not so precise especially for the second and third segments.
Anyway, if we keep the same subsegment dimension (224x224) but we increase the number of frames for each segment (32 for example) we have a better result. This last approach is pictured in Figure 18.

(a) segment 1, first frame (b) segment 2, first frame (c) segment 3, first frame

Figure 18: Samples of frames from three random video segments from an output video. Analysis performed with subsegments of dimension 128x128 and 32 frames segments. The predicted interpolated sectors are marked in red.

This results show that conducting the localization experiment on the same conditions as the training ones, leads

to better results.
Neverthless, a trade off between the subsegments size and the number of frames contained in each segment can be found.

6. Conclusions

In this work, we considered the problem of identifying traces of frame-rate interpolation in video sequences. To this purpose we proposed a detector which is trained in a supervised fashion to understand if a video has been frame-rate interpolated. The proposed methodology exploits the idea of applying different kinds of preprocessing to the video under analysis in order to better expose frame-rate interpolation traces. Preprocessed videos are then passed to a CNN to extract salient features, and an SVM that performs the final classification.

Our method has been validated through a series of experiments. To conduct our experiments, we built a brand new dataset composed by original videos and the correspondent interpolated (at different frame-rates) versions, starting from the Kinetics400 dataset. We have demonstrated the precision of our model, evaluating it from multiple perspectives and proposing different alternatives in the implementation process. Moreover, we have compared against two recently-proposed state-of-the-art techniques, showing that the proposed method is able to outperform both.

Finally, we also made a few steps into the localization of the frame-rate interpolated zones inside a video, rather than just the simple detection, demonstrating a new particular use of the detector.

Despite the promising achieved results, this work has still room for improvement. As a possible continuation or future development of our work, we would like to focus on:

- investigating the effect of training and testing on interpolated videos at frame rates which are different from the adopted ones, in order to increase the variability in the experiment.
- trying to repeat the detection and localization experiments with different parameters, for example extracting video segments of different length or different spatial dimension, to see how this affects the final detection/localization.
- evaluate different kind of video formats with respect to MP4, especially focusing on the effect of possible video coding techniques.
- interpolate videos with different types of interpolation methods with respect to the motion compensated ones, also using other tools rather than just FFMPEG.

References

[1] John W. Woods. Chapter 11 - digital video processing. In John W. Woods, editor, *Multidimensional Signal, Image, and Video Processing and Coding (Second Edition)*, pages 415–466. Academic Press, second edition edition, 2012.

[2] P. Bestagini, S. Battaglia, S. Milani, M. Tagliasacchi, and S. Tubaro. Detection of temporal interpolation in video sequences. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3033–3037, 2013.

[3] Chang Liu and Matthias Kirchner. Cnn-based rescaling factor estimation. pages 119–124, 07 2019.

[4] Brian C. Hosler and Matthew C. Stamm. Detecting video speed manipulation. *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 2860–2869, 2020.

[5] Sagie Benaim, Ariel Ephrat, Oran Lang, Inbar Mosseri, William T. Freeman, Michael Rubinstein, Michal Irani, and Tali Dekel. Speednet: Learning the speediness in videos. `https://arxiv.org/abs/2004.06130`, 2020.

[6] Rafael C. Gonzalez and Richard E. Woods. *Digital image processing.* Prentice Hall, 2008.

[7] Gunnar Farneback. Two-frame motion estimation based on polynomial expansion. volume 2749, pages 363–370, 06 2003.

[8] Saining Xie, Chen Sun, Jonathan Huang, Zhuowen Tu, and Kevin P. Murphy. Rethinking spatiotemporal feature learning: Speed-accuracy trade-offs in video classification. In *ECCV*, 2018.

[9] Will Kay, João Carreira, Karen Simonyan, Brian Zhang, Chloe Hillier, Sudheendra Vijayanarasimhan, Fabio Viola, Tim Green, Trevor Back, Paul Natsev, Mustafa Suleyman, and Andrew Zisserman. The kinetics human action video dataset. *CoRR*, abs/1705.06950, 2017.

[10] Kinetics400 download. `http://deepmind.com/kinetics`.

[11] Ffmpeg. `https://ffmpeg.org/`.

[12] Minterpolate filter. `http://underpop.online.fr/f/ffmpeg/help/minterpolate.htm.gz`.

[13] Python programming language. `https://www.python.org/`.

[14] Opencv python library. `https://opencv.org/`.

[15] Opencv documentation. `https://docs.opencv.org/4.x/d6/d00/tutorial_py_root.html`.

[16] Numpy python library. `https://numpy.org/`.

[17] Albumentations python library. `https://albumentations.ai/`.

[18] Alexander Buslaev, Vladimir I. Iglovikov, Eugene Khvedchenya, Alex Parinov, Mikhail Druzhinin, and Alexandr A. Kalinin. Albumentations: Fast and flexible image augmentations. *Information*, 11(2), 2020.

[19] Optical flow with opencv. `https://docs.opencv.org/3.4/d4/dee/tutorial_optical_flow.html`.

[20] Opencv farneback optical flow parameters. `https://docs.opencv.org/3.4/dc/d6b/group__video_track.html#ga5d10ebbd59fe09c5f650289ec0ece5af`.

[21] Pytorch. `https://pytorch.org/`.

[22] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[23] Adam optimizer. `https://pytorch.org/docs/stable/generated/torch.optim.Adam.html`.

[24] Binary cross entropy loss with logits. `https://pytorch.org/docs/stable/generated/torch.nn.BCEWithLogitsLoss.html`.

[25] Pytorch torch.nn module. `https://pytorch.org/docs/stable/nn.html`.

[26] Sigmoid layer. `https://pytorch.org/docs/stable/generated/torch.nn.Sigmoid.html`.

[27] Radial basis function kernel. `https://scikit-learn.org/stable/modules/generated/sklearn.gaussian_process.kernels.RBF.html`.

[28] Brian Dolhansky, Joanna Bitton, Ben Pflaum, Jikuo Lu, Russ Howes, Menglin Wang, and Cristian Canton Ferrer. The deepfake detection challenge (dfdc) dataset. `https://arxiv.org/abs/2006.07397`, 2020.

Abstract in lingua italiana

Al giorno d'oggi, grazie alla diffusione di dispositivi di acquisizione video a portata di mano e all'uso molto diffuso di social network e app di messaggistica, i video vengono condivisi facilmente e sono diventati parte della nostra vita quotidiana. Tuttavia, la manipolazione dei video è diventata alla portata di tutti grazie alla grande disponibilità di software, semplici da utilizzare, per la modifica video. Ciò ha sollevato nuove preoccupazioni sociali, poiché la distribuzione di video manipolati in modo dannoso può portare a gravi conseguenze (ad esempio, diffamazione delle persone, diffusione di notizie false, formazione di opinioni di massa, ecc.). Per questo motivo, la comunità di forensica multimediale ha iniziato a sviluppare una serie di tecniche per stabilire l'autenticità e l'integrità dei video. L'obiettivo di questa tesi è di arricchire il parnorama delle tecniche di forensica video, proponendo un detector per identificare l'interpolazione frame-rate di video. Dato un video da analizzare, il nostro obiettivo è di identificare se il video ha subito qualche operazione di sovra campionamento frame-rate che sono solitamente applicate quando molteplici video vengono uniti, o per nascondere alcune parti di un video. La tecnica proposta è basata su un insieme di Convolutional Neural Network (CNN) che lavorano su tre diversi domini video (ovvero, pixels, optical flow e residui di frame), e su una Support Vector Machine (SVM) per una classificazione finale. I risultati mostrano che il metodo proposto supera in prestazioni i più recenti detector di interpolazione frame rate di video, e può anche essere usato per localizzare le regioni di spazio in cui un video è stato interpolato.

Parole chiave: forensica video, interpolazione, frame-rate, deep learning, convolutional neural network

Acknowledgements

This thesis represents the conclusion of two years at MAE (and a little more) in which I learned a lot of new things and met a lot of new people.

I really want to thank each one of those people, colleagues and professors, that I met along the way.

Among them, a special mention goes to Daniele Ugo Leonzio with whom I shared a lot of moments.

A huge thank to my family and friends for always supporting me.

Finally, a special thank to prof. Paolo Bestagini for the help and the support during the months in which this work was developed.

Simone Mariani