Università di Pisa

Intelligent Systems
Project: Violence Detection in CCTV footage

Fabio Piras, Giacomo Volpi, Guillaume Quint

Academic Year: 2022/2023

Contents

Chapter 1

Introduction

The goal of this project is to develop a CNN capable of recognizing the presence of violence in the everyday CCTV footage in a binary detection system (violence/non violence). In the world of today it is fundamental for police and other forces to have a quick response in case of violent or criminal behavior especially when the people involved cannot dial the emergency lines due to immediate danger. The main idea would be for these kind of AI algorithms to process enormous amount of videos and signal them in case of foul play in real time. This project presented itself as a very difficult one due to the "*dirtiness*" of the dataset used which, due to the fact that it was composed of real life CCTV footage, was not standardized, contained a lot of noise and many more other problems that will be explained in the next section.

1 Dataset presentation

The data-set was obtained by publicly available sources:

- Smart-City CCTV Violence Detection Dataset[1]: 1Gb
- seymanurakti/fight-detection-surv-dataset[2]: 100Mb

The first one was extracted from Kaggle and the dataset was divided into three sub-category (Non-Violence/Violence/Weapon-Violence)[3], the second one was taken from a GitHub repository and the videos were divided into two sub folder (fight/noFight). The whole data-set was then divided by us into two main categories: *violence* and *non violence*.

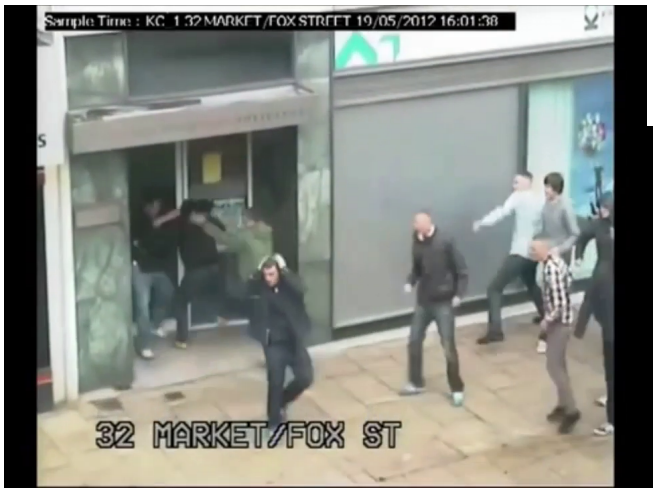
As previously said the problem, which is by itself quite difficult, was made harder due to the poor usability of the dataset which did not give any information on the videos aside form the category: i.e. no action frame nor bounding boxes. Moreover the first set of videos received a usability score of 6.88 on Kaggle, which represents a community rating of the dataset as a whole, for example some videos contained writing over the screen, others were not CCTV footage, but live news ones and some were recording through a phone of a screen with the footage. In addition the dataset was not standardized at all with videos with variable length and some greyscaled, some full color. Some example with the relative category are shown in Fig. 1.1.

What these images prove is the dirtiness of the dataset and the difficulty of the task at hand. For example Fig. 1.1c not only has violence in the distance, but also has writing on the screen, Fig. 1.1d is a phone recoding of a desktop screen with the footage like Fig. 1.1g. Fig. 1.1f is a

[1] `https://www.kaggle.com/datasets/toluwaniaremu/smartcity-cctv-violence-detection-dataset-scvd/data`

[2] `https://github.com/seymanurakti/fight-detection-surv-dataset`

[3] Taken from version 2, now version 3 is available



(a) Image with non standardized size

(b) Image with blur and noise

(c) Violence in the distance

(d) Violence close to camera

(e) writing on the screen and brownscaled

(f) Violent frame with no violence

(g) Video taken with a phone camera

Figure 1.1: Some *violent* frames from the data-set

frame with no violence at all, but it was classified as a violent one because it is part of a video that contains violence, but removing all frames that are not violent would have been too much of a hassle (more than 7k images to be process manually). In addition many videos like Fig. 1.1b and 1.1d are greyscaled. Others like Fig. 1.1e and 1.1f contain black bars on the side. This means that the data-set used was not standardized at all and there was no easy way to automate the cleaning process.

2 Goals and evaluation

The goal of this project is to develop a CNN capable of recognizing the presence of violence in the everyday CCTV footage in a binary detection system. We wanted to evaluate our models with a real life approach, meaning that, although an high accuracy is desirable, it is not the only metric we are interested in, due to the differences in error cost between mislabelling violence as non violence and vice versa. In fact we are also interested in the precision and recall of the model, because we want to minimize the number of false negatives, meaning the case of violence not recognized as such, because it could result in damage of property or even a loss of life. So it is preferable to have a low false negative rate at the cost of a higher false positive rate without making it too high, causing the police to waste time and resources on false alarms.

Chapter 2

2D CNN approach

The very fist approach we tried was to develop a 2D CNN from scratch to recognize the frames of a video containing a scene of violence, to do that we needed to extract the frames from the videos and feed them to the AI model since a 2D CNN processes images.

1 Frame extraction

The idea initially was to extract all frames with the help of *FFmpeg*. FFmpeg is a comprehensive software suite for recording, converting, and playing audio and video, it relies on libavcodec, a library for audio/video encoding. Following there is an example of usage.

```
#!/bin/bash
# To force the frame rate of the output file to 24 fps:
ffmpeg -i input.avi -r 24 output.avi
```

Listing 2.1: FFmpeg example

Following there is the code used to extract the frames from the videos. For brevity we will show only the code used to extract the frames from the "fight" directory and the actual paths are removed.

```
for f in $(ls $path_to_video); do
  ffmpeg -i $path_to_video/$f -vf fps=5 $path_to_frame/${f%.*}-%03d.png;
done;
```

Listing 2.2: Frame extraction

This command uses FFmpeg to extract frames from a video file and save them as individual images, it iterates over each file in the specified directory and applies the FFmpeg command to convert the video to frames. The frames are saved with the same name as the original video file, followed by a three-digit number to indicate the frame sequence, the frames are saved in the specified output directory. However, this method was found to not be flexible enough, in the light of a new approach to the problem like a 3D CNN, discussed in chapter 3, so we decided to use a different approach. We decided to extract frames at runtime depending on various configuration parameters and feed them to the AI model. However performing the frame extraction each time we wanted to train a network would have been too much of a workload. To speed up the process, we decided instead to separate the extraction phase from the training phase and save a serialized copy of the preprocessed dataset on local files, through the help of the *pickle* library in Python. In addition, this approach was more efficient in terms of memory usage. The following pseudo-code shows the basic idea of the frame extraction process:

```
    # Function to load a dataset of videos with corresponding labels
   def create_video_dataset(dataset_path, dataset_config):
        train_data = []
```

```
        train_labels = []
        test_data = []
        test_labels = []

        try:
            # Attempt to load data and labels from pickle files
            # if they already exist
            load_data_from_pickles([pickle_file_paths...])

        except FileNotFoundError:
            samples = collect_all_dataset_samples(dataset_path)

            data, labels = get_labelled_samples(samples, dataset_config)

            # Shuffle the list of labeled videos
            # Avoid deleting always the same videos at balancing time
            shuffle(data, labels)

            # Load and preprocess videos,balancing
            # the number of samples for each class
            data, labels = balance(data, labels)

            train_data, train_labels, test_data, test_labels = split_data(
    data, labels, testing_split)

            # Save the data and labels as pickle files
            save_pickles([pickle_file_paths...])


        except Exception as e:
            print(f"An unexpected error occurred: {e}")

        # Return the loaded or newly created data and labels
        return train_data, train_labels, test_data, test_labels
```

Listing 2.3: Pseudocode of pickles generation logic

The main idea is to create 4 pickle files for each configuration (train_data, train_label, test_data, test_label). The pickle files are created only if they are not present, this way we can load them at runtime and avoid the frame extraction process. In case of creation needed, the videos are shuffled before the extraction phase. In this way, when balancing the dataset, we avoid always removing the last loaded videos. This process is flexible enough to allow us to create pickles for both 2D CNNs and 3D CNNs. In the 3D case, the samples consist in a burst of consecutive frames. The *pkl_ config* dictionary contains the following information:

- frame size
- number of frames
- train split
- fps
- crop

The *frame size* specifies the height and width of the frame. The *number of frames* is the number of frames to be extracted from the video for each *burst*. The *train split* is the fractional split between training and testing. The *fps* is the frame per second of the video, for example if we want to train a 2D CNN both the *number of frames* and the *fps* will be set to 1, if we want

to train a 3D CNN with burst of 2 seconds we might set the *number of frames* to 10 and the *fps* to 5. The *crop* is a boolean value that indicates if the frame should be cropped or not to remove the black bars, as explained in section 2 .

2 Dataset cleaning

The dataset was very dirty. This fact resulted in many difficulties throughout the project development: the dataset was composed of videos of different lengths, different resolutions and different frame rates. Furthermore, many videos contained black bars on the sides, making many AI models learn from them instead of the actual features. To solve this problem we decided to crop the videos removing black bars, this was done with the following code:

```
def crop(image, y_nonzero, x_nonzero):
    # If y_nonzero and x_nonzero are not provided, calculate them from the grayscale version of the image
    if y_nonzero is None or x_nonzero is None:
        gray_image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
        y_nonzero, x_nonzero = np.where(gray_image > 1)

    # Crop the image based on the calculated non-zero values
    cropped_image = image[np.min(y_nonzero):np.max(y_nonzero), np.min(x_nonzero):np.max(x_nonzero)]

    # Return the cropped image along with updated y_nonzero and x_nonzero values
    return cropped_image, y_nonzero, x_nonzero
```

Listing 2.4: Image Cropper

However this proved to be not enough, the dataset was still very noisy: elements like writings over the videos and some mislabeled frames increased the difficulty for models to learn the correct features. An initial approach for frame cropping involved a simple cut-out of the largest square centered within the frame to prevent distortion, but this resulted in the exclusion of many areas of action for all those videos containing violence on the edge of the screen. The final approach introduces a light distortion with respect to the original source, but we preserve all relevant information, discarding all rows and columns of pixels containing pure black values. For future works on the subject, it would be better to aggregate a much more comprehensive dataset, which would include bounding boxes to better identify the actual areas of action.

3 First model

For the first model we decided to use a very simple 2D CNN. The model was composed of 2 convolutional layers, 2 max pooling layers and 1 dense layer. The model was trained for 100 epochs with a batch size of 32 samples. To avoid overfitting, we used the Early Stopping technique with patience set to 15, after an initial warm-up period of 5 epochs; we also set the *restore best weight* flag to save the best model before the validation loss increases. The optimizer used was Adam, a popular optimization algorithm which estimates the first and second moments of the gradients. These parameters were found to perform better from early experimentation runs. The model was trained on the dataset with the black bars removed, the dataset was composed of 2 classes, violence and no-violence. The model was trained on 80% of the dataset and tested on the remaining 20%, the validation set was 30% of the training set with a hold out methodology.

The first comment to be made is that the confusion matrix is heavily biased towards the *Non violence* class. This is due to the fact that the dataset can be quite chaotic and the model

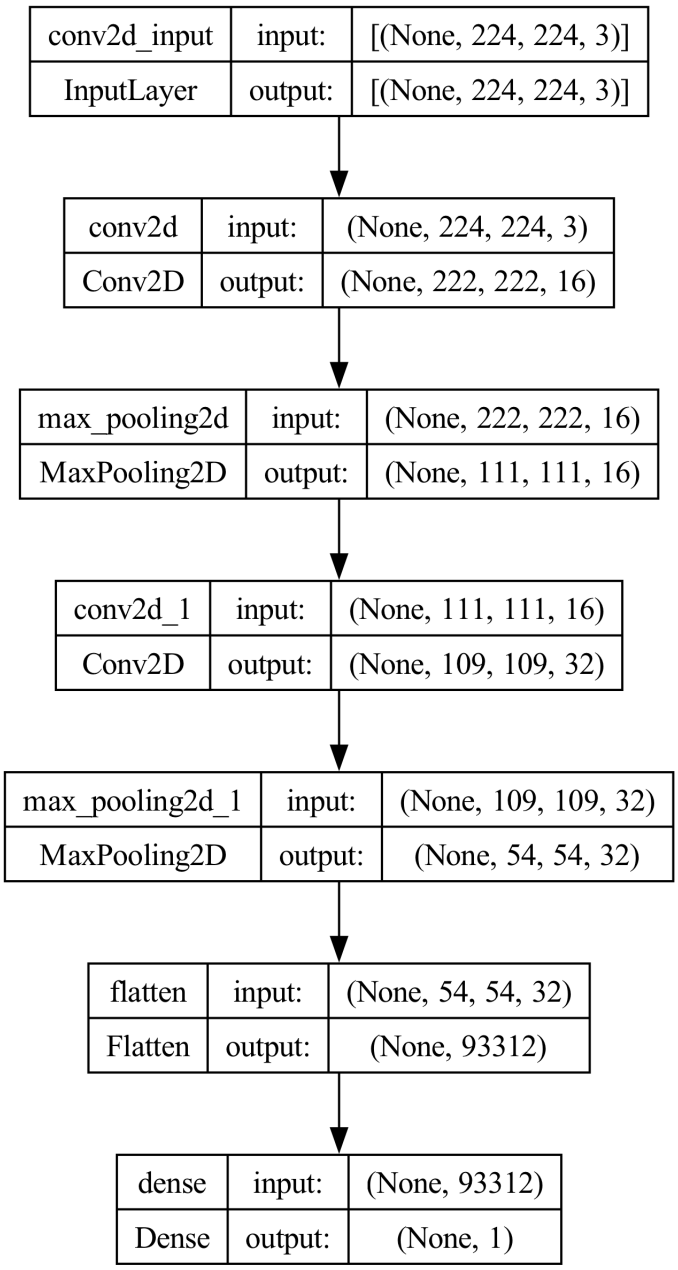


Figure 2.1: Initial 2D CNN model

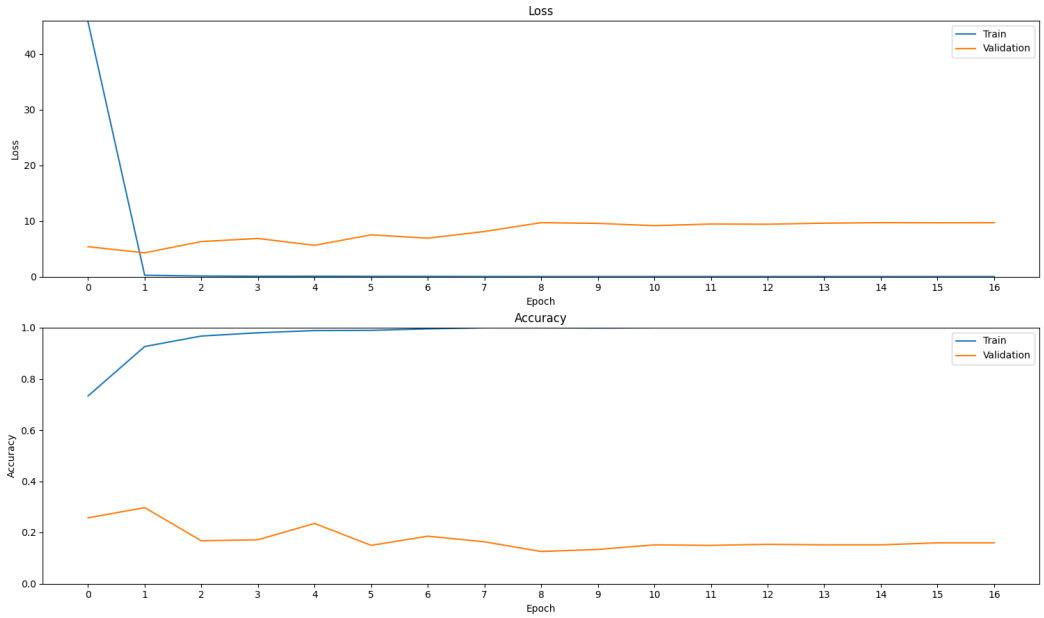


Figure 2.2: Training history of the first model

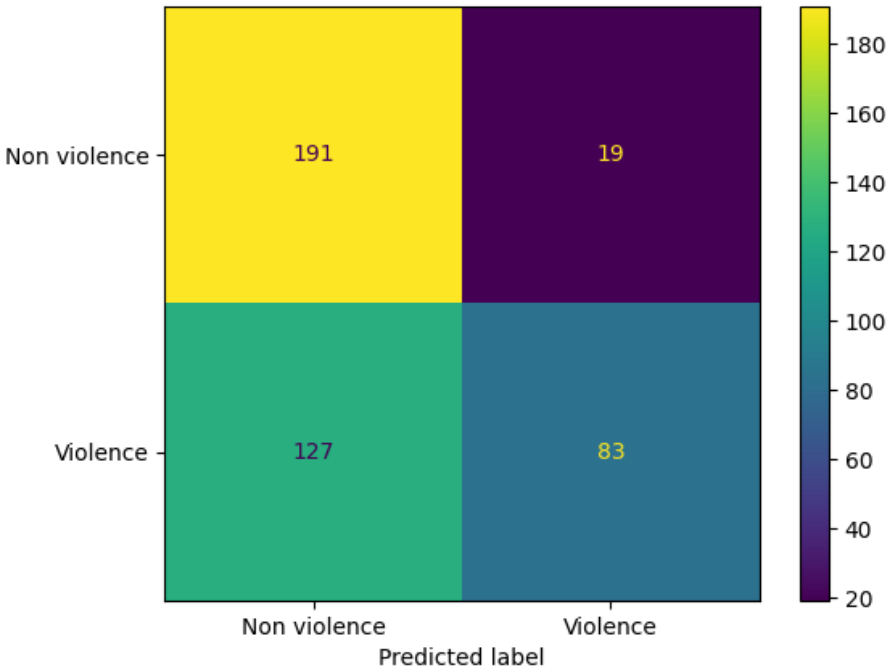


Figure 2.3: Confusion matrix of the first model

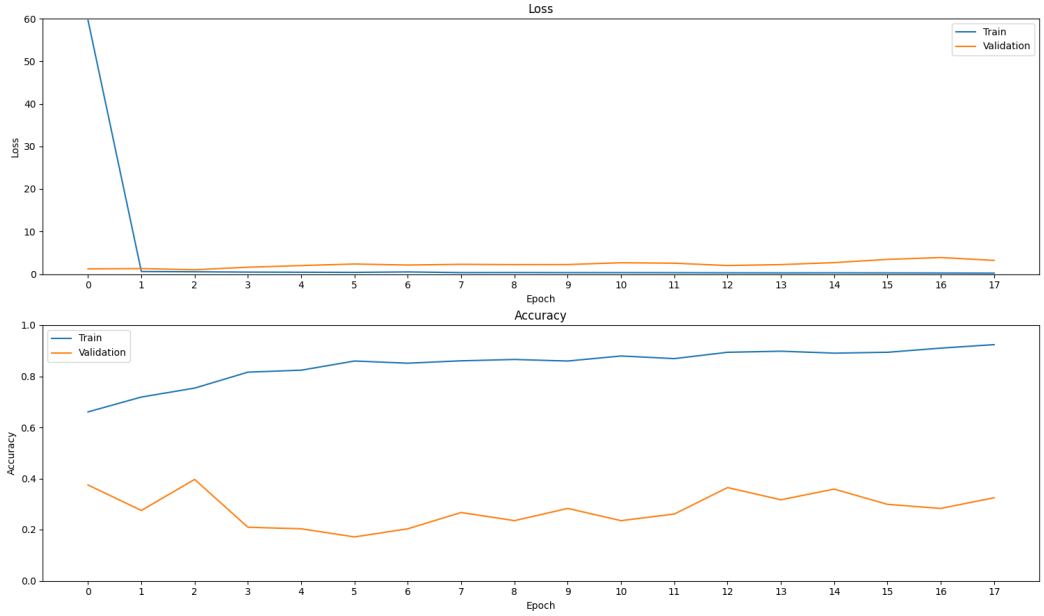


Figure 2.4: Training history of the second model

is not able to learn all relevant features from the videos. For example the model may learn form the Non violence part of the dataset that gatherings of people are not to be flagged as violence, however this could easily lead to many false negatives (Violence images classified as Non violence ones).

Another comment is to be made regarding the validation loss and accuracy, which are unsatisfactory in both cases, this could signal overfitting and a general failing in extrapolating the main features of the videos. Since the training accuracy is almost immediately saturated we can assume that the dataset is too small and the model starts to overfit in the first epochs.

4 Second model - data augmentation

For the second model we decided to keep the convolution layers to 2, add a dense layer and, most importantly, perform data augmentation by adding random_flip(horizontal) and random_rotation(0.1) as shown in Fig. 2.5 to make it harder for the model to overfit immediately by memorizing the entire training dataset. All other parameters remained unchanged: this was done to see if the model would have been able to produce better results. We decided to use a cautious approach since the problem presented itself as a very complicated one and we did not want to bring many changes in a single *pass* to avoid a time consuming trial and error process.

Speaking of results, the model no longer immediately saturates the training accuracy: this could be a sign of a better generalization capability, however the model still has a very low validation accuracy, meaning that it is still not able to solve the problem very well. The confusion matrix is shown in Fig. 2.6, the model is still heavily biased towards the *Non violence* class.

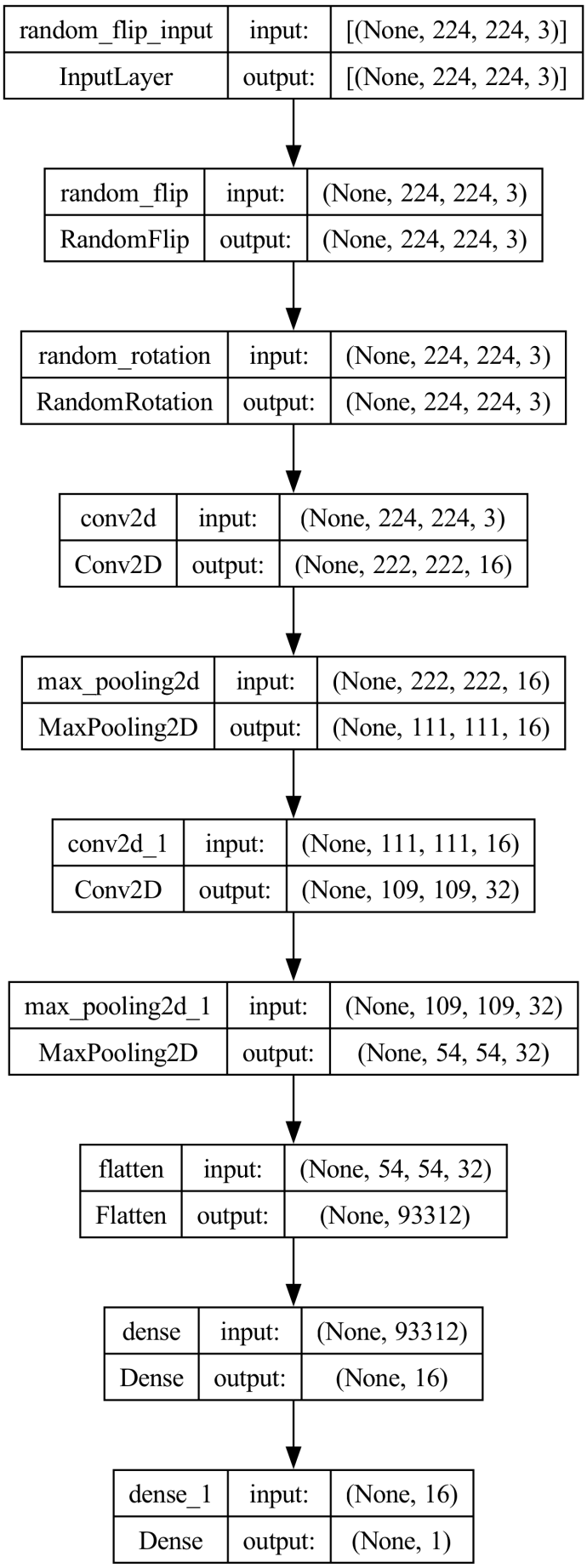


Figure 2.5: Second 2D CNN model

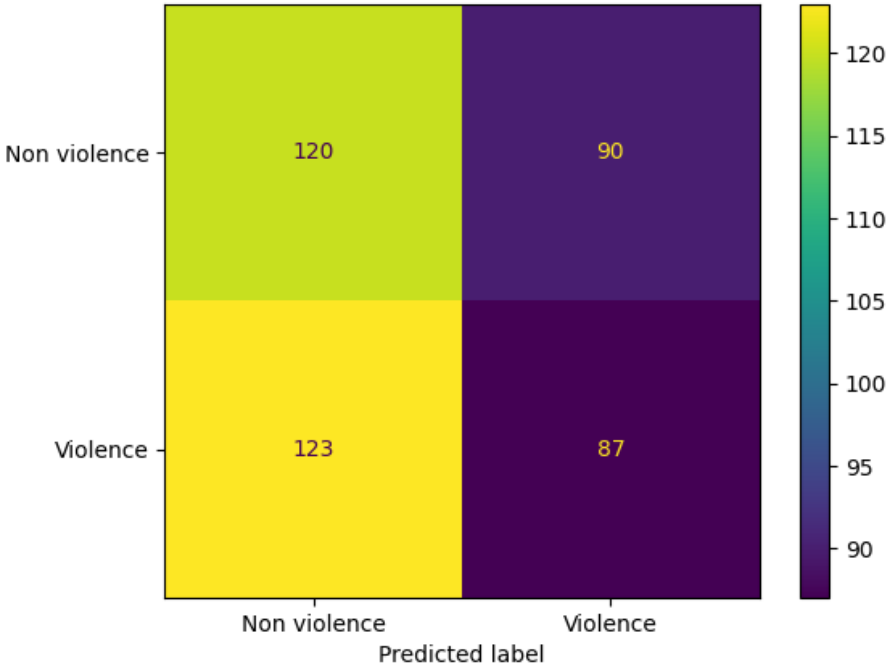


Figure 2.6: Confusion matrix of the second model

5 Third model - *upgrading* the model

Based on the results of the previous models, we decided to trying to *upgrade* the model, this was done for two main reasons. The first one was to test the behavior of a more capable model to tackle this difficult problem. The second one was that, since the last results were not satisfactory, especially in the validation accuracy, we thought that the model was possibly getting stuck on a single result, regardless of the actual features extracted from the frames. Therefore we decide to amp the model capabilities as show in Fig. 2.7.

Moving to the results of the model we can see in Fig. 2.9 that the confusion matrix is now more balanced than the previous ones, with an accuracy rate of almost 60%. However the recall value are still not good enough, especially the violence one with a score of 57%. Another thing to notice is the training history graph in Fig. 2.8, where the validation loss and accuracy are very unstable. This led us to the conclusion that the model is now overfitting, so the next logical step is to try to fight it with dropout layers.

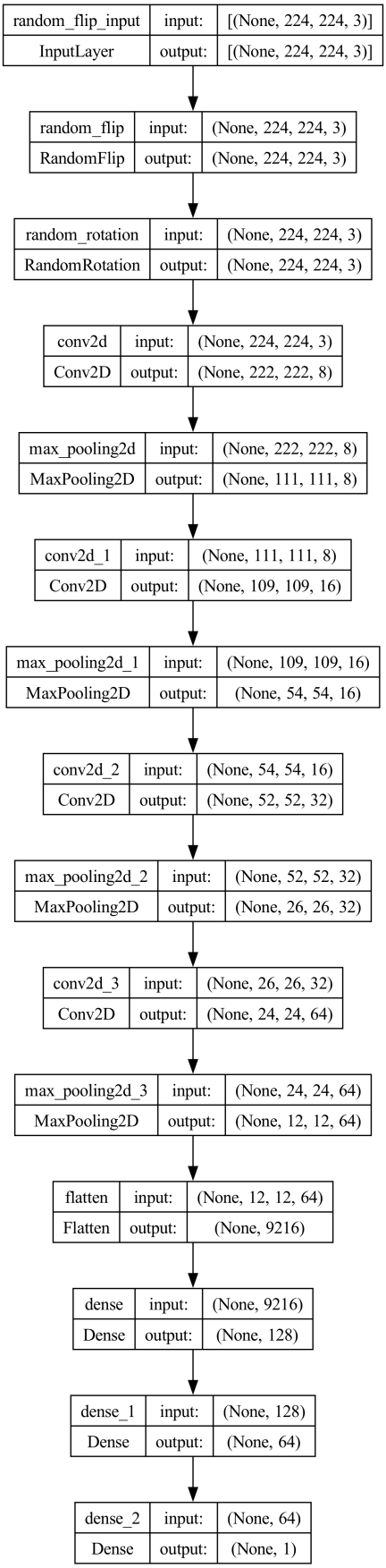


Figure 2.7: Third 2D CNN model

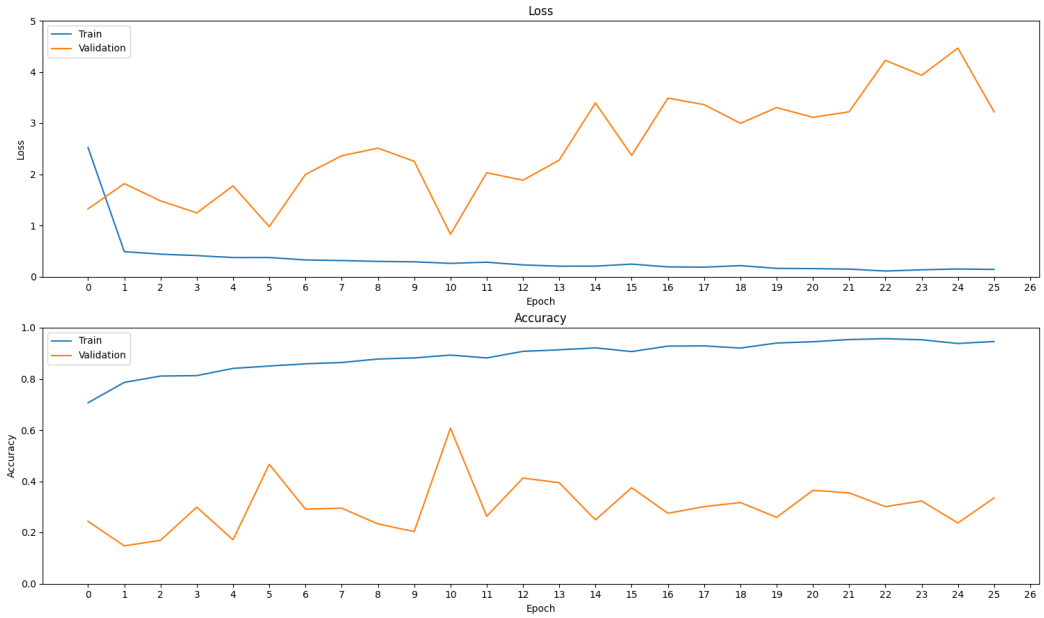


Figure 2.8: Training history of the third model

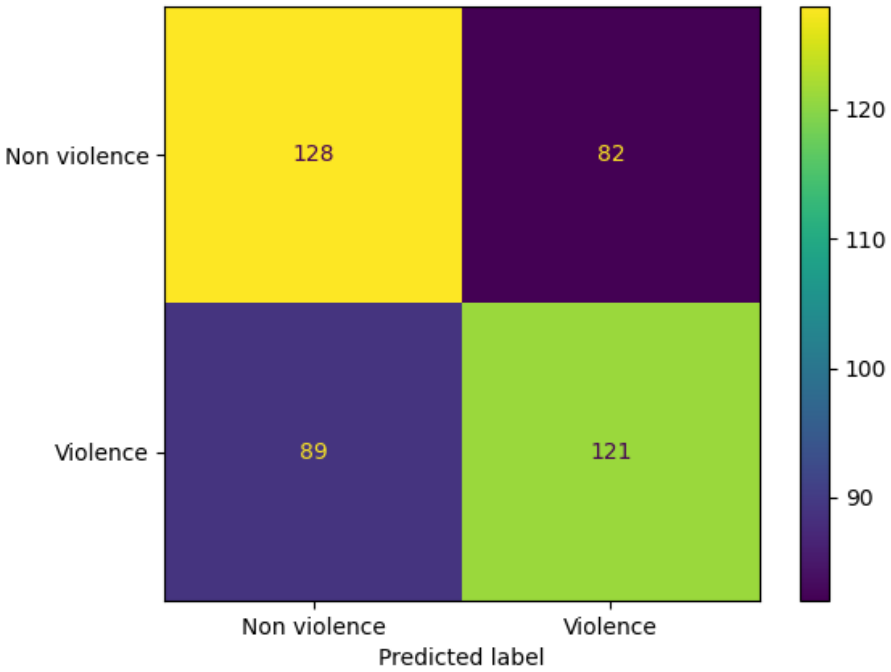


Figure 2.9: Confusion matrix of the third model

6 Fourth model

After various test we produced the model shown in Fig. 2.10 and Fig. 2.11.

The model presents itself with many dense layers and a single dropout layer. The convolutional layers are mostly unchanged. The model was trained for 100 epochs with a batch size of 32 samples as the previous ones until now. The results obtained show a 62% accuracy which is the highest one so far excluding the first model which was heavily biased towards the *Non violence* class and also was clearly overfitting. The main issue of this model remains the violence recall which is still too low to satisfy our needs, in fact a score of 49% is clearly unsatisfactory. The confusion matrix is shown in Fig. 2.13. Regarding the training history shown in Fig. 2.12 we can see that the model quickly starts to overfit, but an increase in dropout layer or weight decay did not improve the results.

To recap all the experiments we ran, we present a table with the accuracy and recall of the models in Tab. 2.1. The first model is the best one apparently, but considering the size of the original dataset this cannot be considered a trustworthy result, the second model is the worst one, this is due to the fact that the data augmentation make the dataset more noisy and the model was not powerful enough to learn the correct features. The third model improves on the results of the previous one, this is due to the fact that the model is able to learn the correct features, but it is not able to generalize well. The fourth model is the best one so far, but it is still not able to learn the correct features of the videos. So a different approach is needed to solve the problem. The main issue is the dataset, it is too small so data augmentation is a must, but then we need a more powerful model to learn the correct features of the videos, so instead of trying to *brute force* the problem we decided to rely on pretrained models to better understand the situation.

	Accuracy	**Violence recall**	**Non violence recall**
First Model	0,65238	0,39524	0,90952
Second Model	0,49285	0,41429	0,57143
Third Model	0,59285	0,57619	0,60952
Fourth model	0,62619	0,49048	0,7619

Table 2.1: 2D models accuracy and recall

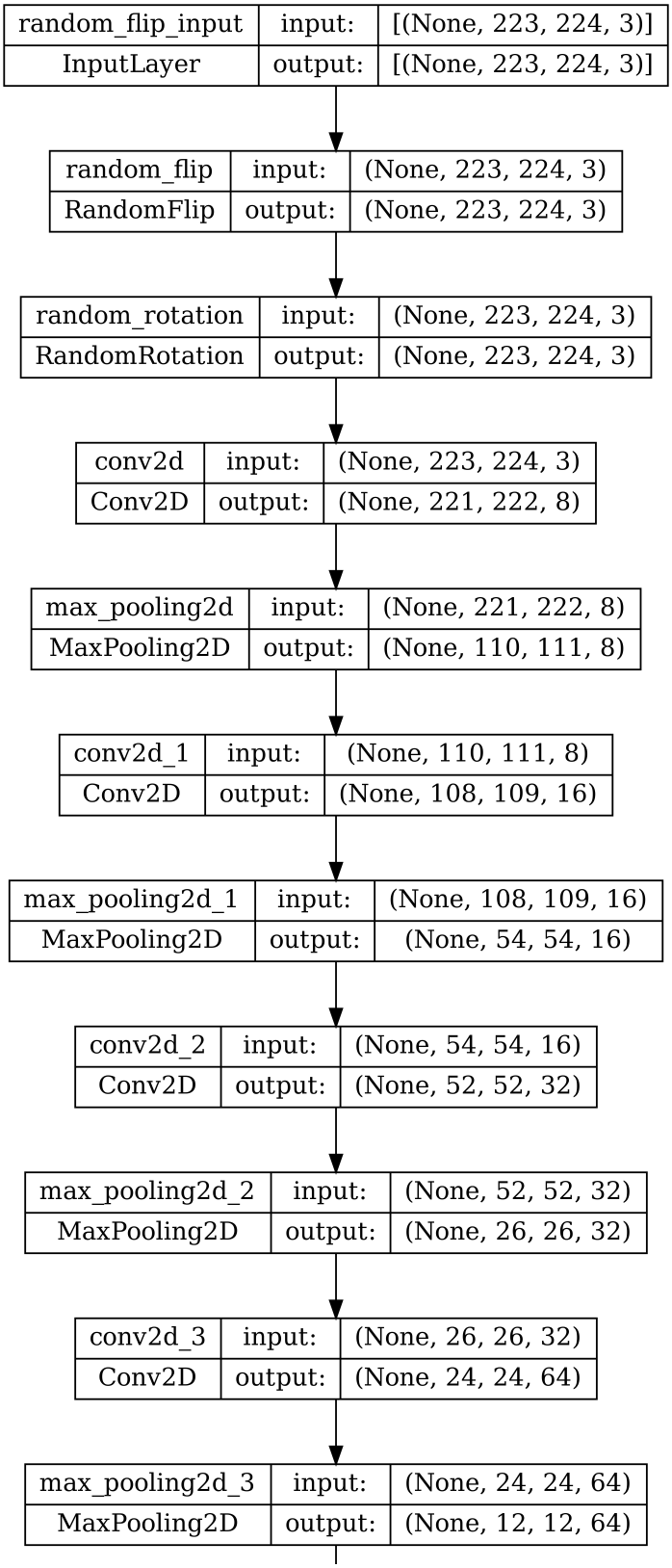


Figure 2.10: Fourth 2D CNN model

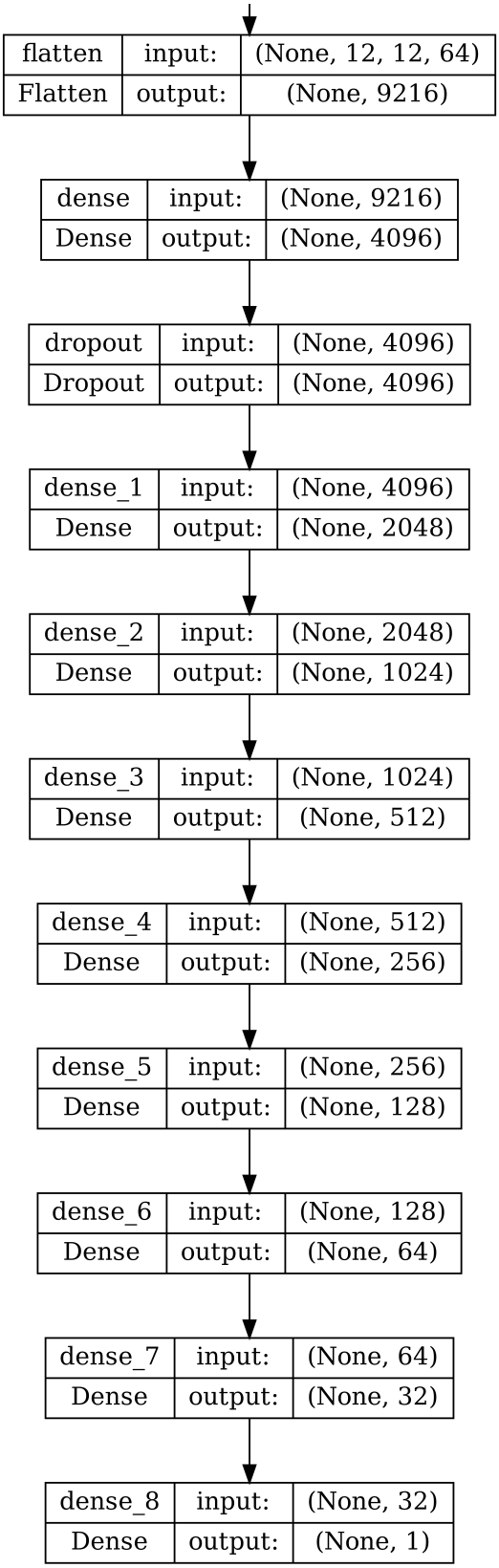


Figure 2.11: Fourth 2D CNN model

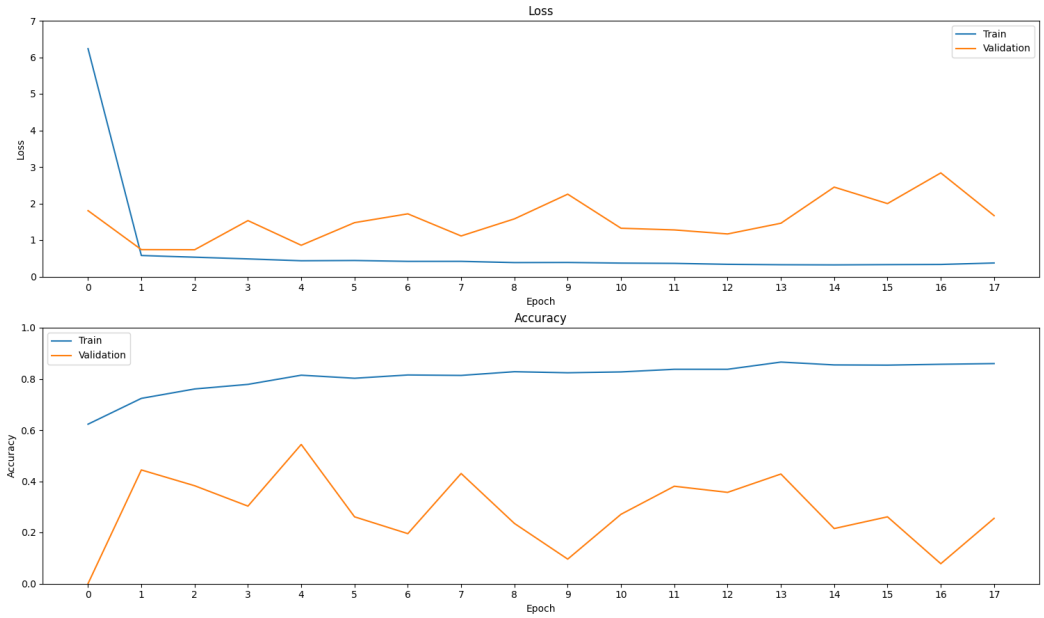


Figure 2.12: Training history of the fourth model

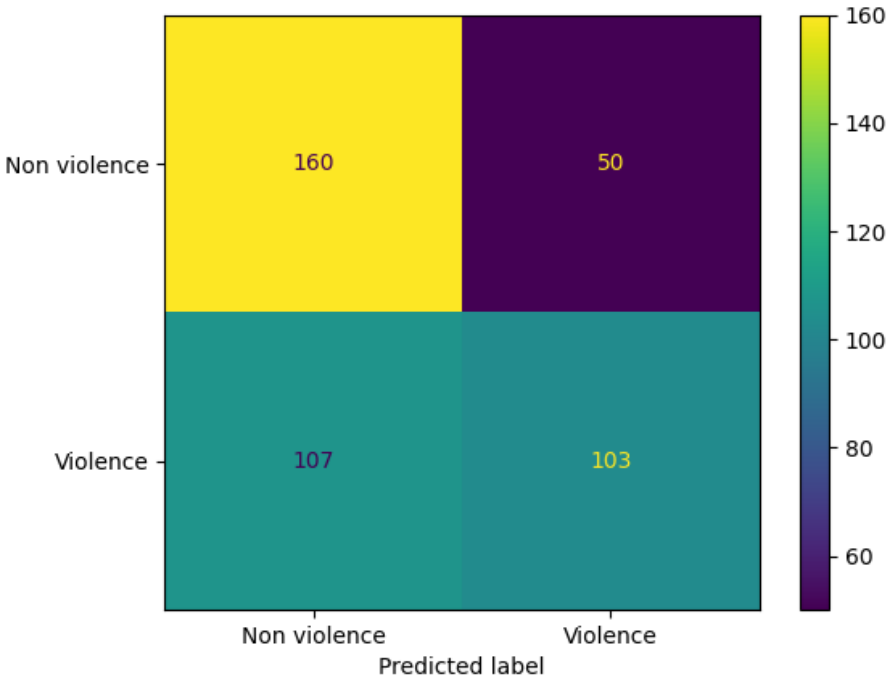


Figure 2.13: Confusion matrix of the fourth model

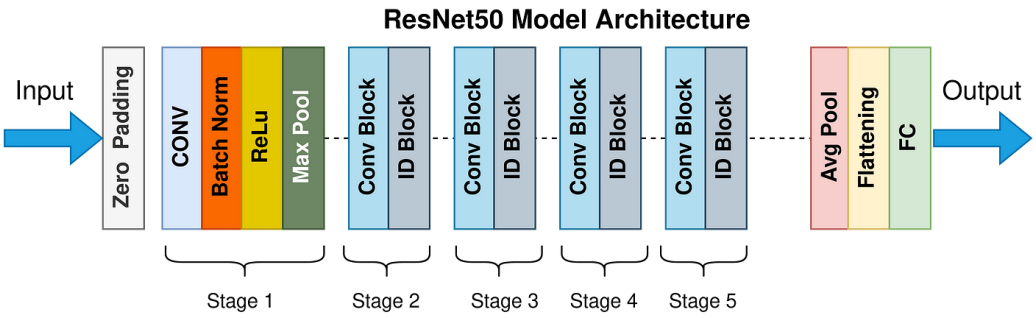


Figure 2.14: ResNet50 architecture

7 Pretrained models: ResNet50

Since the 2D CNN *from scratch* approach was not working we decided to try a different method, we decided to use pretrained models. This was done in order to have a better understanding of the problem by comparing the results of them with the ones we presented previously. We decided to use Resnet50 as the first pretrained model.

ResNet50 [1] was developed by Microsoft Research in 2015, it features a deep structure with 50 layers, utilizing residual blocks that allow the network to learn residual functions to ease the optimization process. The core innovation lies in skip connections, where the input from one layer is added to the output of another, facilitating the flow of gradients during backpropagation. ResNet50 won the ImageNet Large Scale Visual Recognition Challenge in 2015. It includes bottleneck building blocks to improve computational efficiency by reducing the number of parameters in the intermediate layers. We used ResNet50 with the imagenet weights, and we choose to do *fine tuning*. We started by adding our dense layers for classifications and we only train the last convolutional block.

```
def ResnetFirstModel(input_shape):
  model = Sequential()
  resnet = ResNet50(weights='imagenet', include_top=False, input_shape=
   input_shape)
  set_trainable = False
  for layer in resnet.layers:
      if layer.name == 'conv5_block1_1_conv':
          set_trainable = True
      if set_trainable:
          layer.trainable = True
      else:
          layer.trainable = False
  model.add(resnet)
  model.add(GlobalAveragePooling2D())
  model.add(Dense(256, activation='relu'))
  model.add(Dense(128, activation='relu'))
  model.add(Dense(1, activation='sigmoid'))
  return model
```

Listing 2.5: ResNet50, first model

The model shows promising results, with a good confusion matrix (Fig. 2.15) and also with a 76% recall on violence.

[1] https://arxiv.org/abs/1512.03385

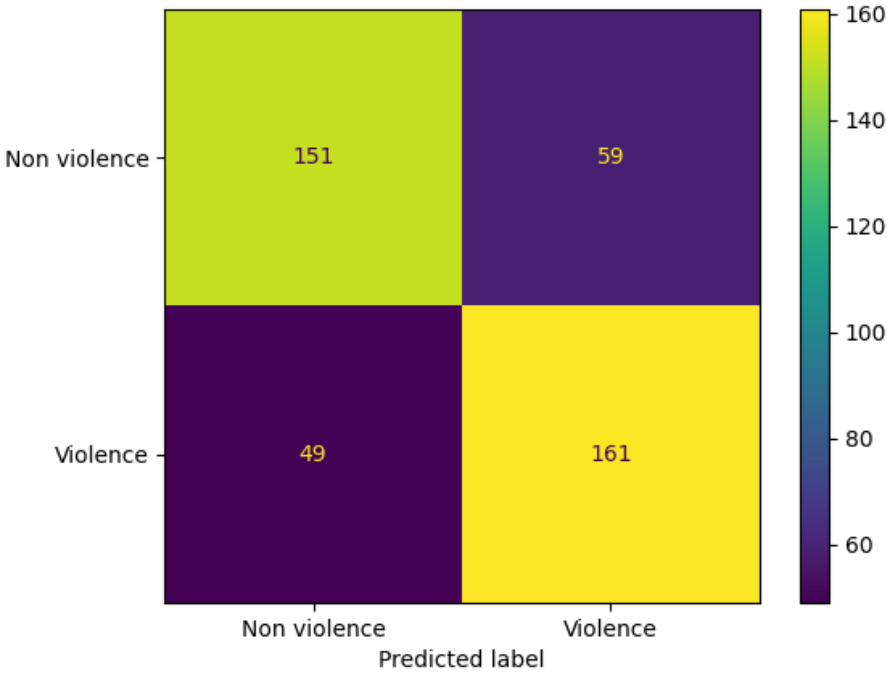


Figure 2.15: ResNet50 first model

We continued to improve the model adding the layers for data augmentation, however we noticed worse performances compared to the first model and signals of overfitting.

```
def ResnetSecondModel(input_shape):
  model = Sequential()
  resnet = ResNet50(weights='imagenet', include_top=False, input_shape=
   input_shape)
  set_trainable = False
  for layer in resnet.layers:
      if layer.name == 'conv5_block1_1_conv':
          set_trainable = True
      if set_trainable:
          layer.trainable = True
      else:
          layer.trainable = False
  model.add(RandomFlip('horizontal'))
  model.add(RandomRotation(0.1))
  model.add(resnet)
  model.add(GlobalAveragePooling2D())
  model.add(Dense(256, activation='relu'))
  model.add(Dense(128, activation='relu'))
  model.add(Dense(1, activation='sigmoid'))
  return model
```

Listing 2.6: ResNet50, second model code

In order to fight overfitting we introduced a dropout layer and the overall accuracy improved, and we obtained better results compared to the original model (Fig. 2.16).

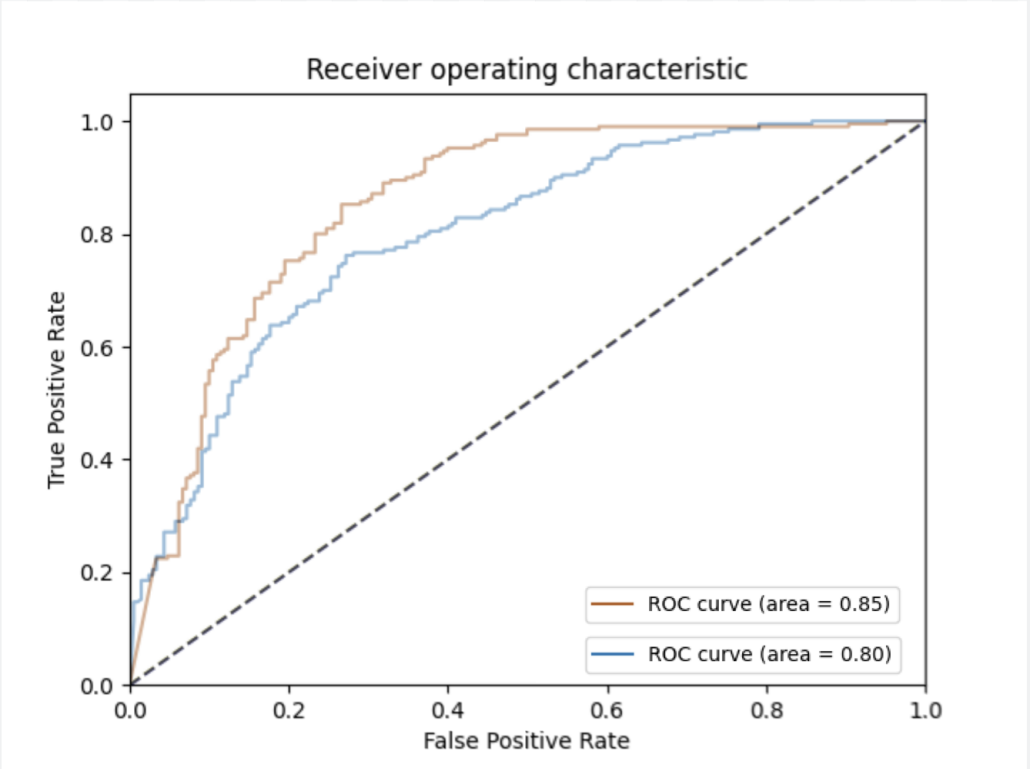


Figure 2.16: ROC of the first Resnet50 vs third Resnet50 model

```
def ResnetThirdModel(input_shape):
  model = Sequential()
  resnet = ResNet50(weights='imagenet', include_top=False, input_shape=
   input_shape)
  set_trainable = False
  for layer in resnet.layers:
      if layer.name == 'conv5_block1_1_conv':
          set_trainable = True
      if set_trainable:
          layer.trainable = True
      else:
          layer.trainable = False
  model.add(RandomFlip('horizontal'))
  model.add(RandomRotation(0.1))
  model.add(resnet)
  model.add(GlobalAveragePooling2D())
  model.add(Dense(256, activation='relu'))
  model.add(Dropout(0.2))
  model.add(Dense(128, activation='relu'))
  model.add(Dense(1, activation='sigmoid'))
  return model
```

Listing 2.7: ResNet50, third model code

We also run a fourth test applying a small dropout also to the second dense layer but we not obtain better results compared to the previous model. We assume that our network needs the 128 neurons of the second layer for a correct classification of the problem.

```
def ResnetFourthModel(input_shape):
  model = Sequential()
  resnet = ResNet50(weights='imagenet', include_top=False, input_shape=
   input_shape)
  set_trainable = False
  for layer in resnet.layers:
```

```
        if layer.name == 'conv5_block1_1_conv':
            set_trainable = True
        if set_trainable:
            layer.trainable = True
        else:
            layer.trainable = False
    model.add(RandomFlip('horizontal'))
    model.add(RandomRotation(0.1))
    model.add(resnet)
    model.add(GlobalAveragePooling2D())
    model.add(Dense(256, activation='relu'))
    model.add(Dropout(0.2))
    model.add(Dense(128, activation='relu'))
    model.add(Dropout(0.1))
    model.add(Dense(1, activation='sigmoid'))
    return model
```

Listing 2.8: ResNet50, fourth model code

Following a recap of the accuracy of the models and the recall:

	Accuracy	**Violence recall**	**Non Violence recall**
ResNet 1st Model	0,7429	0,7667	0,7191
ResNet 2nd Model	0,6357	0,3619	0,9095
ResNet 3rd Model	0,7643	0,6952	0,8333
ResNet 4th Model	0,6809	0,6571	0,7048

Table 2.2: ResNet50 accuracy and recall

As can be seen in Tab. 2.2 the best model is the third one, this is due to the fact that the dataset is very small, so the data augmentation is a must. The fourth model is not as good as the third one, because of the dropout layers are too many and the model is not able to learn features properly. The first model is the second best and functions as a base line. The second one is the worst of the collection, this is due to the fact that the data augmentation alone is not enough and leads to memorizing non important features such as background or irrelevant details in the videos and therefore overfitting.

8 Pretrained models: EfficientNetB0

EfficientNetB0[2] is part of the EfficientNet family, designed to achieve superior performance with fewer parameters compared to traditional models. It introduces a compound scaling method that uniformly scales the depth, width, and resolution of the network, leading to improved efficiency across all dimensions. The architecture includes mobile inverted bottleneck blocks and squeeze-and-excitation blocks to enhance feature extraction and model expressiveness.

EfficientNetB0 is also suitable for resource-constrained environments like mobile devices, being computationally efficient.

The reason we choose EfficientNetB0 is because it can be used in IoT devices, like cameras, for a first filter on images for police operators. The net was used with the imagenet weights like ResNet50. Instead of going directly to fine tuning we start with a *features extraction* approach as seen in listing 2.9.

[2] https://arxiv.org/abs/1905.11946

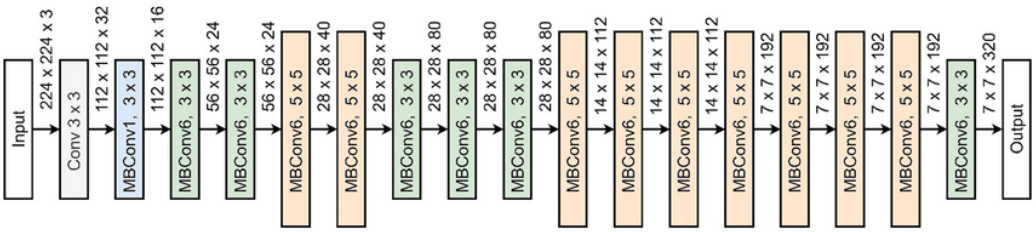


Figure 2.17: EfficientNetB0 architecture

```
def EfficientnetFirstModel(input_shape):
  model = Sequential()
  EfficientNetB0 = EfficientNetB0(weights='imagenet', include_top=False,
   input_shape=input_shape)
  model.add(EfficientNetB0)
  model.add(GlobalAveragePooling2D())
  model.add(Dense(256, activation='relu'))
  model.add(Dense(128, activation='relu'))
  model.add(Dense(1, activation='sigmoid'))
  return model
```

Listing 2.9: EfficientNetB0 first model

The first model starts from the simplest implementation as possible, with only a final dense layer with one neuron with a sigmoid activation function. We obtained a 62% overall accuracy, comparable to the final model of the *scratch* 2D CNN, but below both ResNet models. In order to improve the performance of the network we add data augmentation layers and also more dense layers as seen in listing 2.10.

```
def EfficientnetFirstModel(input_shape):
  model = Sequential()
  EfficientNetB0 = EfficientNetB0(weights='imagenet', include_top=False,
   input_shape=input_shape)
  model.add(EfficientNetB0)
  model.add(GlobalAveragePooling2D())
  model.add(Dense(256, activation='relu'))
  model.add(Dense(128, activation='relu'))
  model.add(Dense(1, activation='sigmoid'))
  return model
```

Listing 2.10: EfficientNetB0 second model

The accuracy of the second model increased from 62% to 69% and accordingly the violence recall (48% compared to 38%), however we are still far away from an acceptable level. This leads us to the conclusion that the best approach to get good results is moving to fine tuning. We decided to train only the two last convolutional blocks, since going at the first layers would need a huge dataset, we leave the same dense layers for classification as used in fine tuning, as seen in listing 2.11.

```
def EfficientnetThirdModel(input_shape):
  model = Sequential()
  model.add(RandomFlip('horizontal'))
  model.add(RandomRotation(0.1))
  EfficientNetB0 = EfficientNetB0(weights='imagenet', include_top=False,
   input_shape=input_shape)
  set_trainable = False
  for layer in EfficientNetB0.layers:
    if layer.name == 'block6a_expand_conv':
        set_trainable = True
    if set_trainable:
        layer.trainable = True
```

```
    else:
        layer.trainable = False
  model.add(EfficientNetB0)
  model.add(GlobalAveragePooling2D())
  model.add(Dense(256, activation='relu'))
  model.add(Dense(128, activation='relu'))
  model.add(Dense(1, activation='sigmoid'))
  return model
```

Listing 2.11: EfficientNetB0 third model

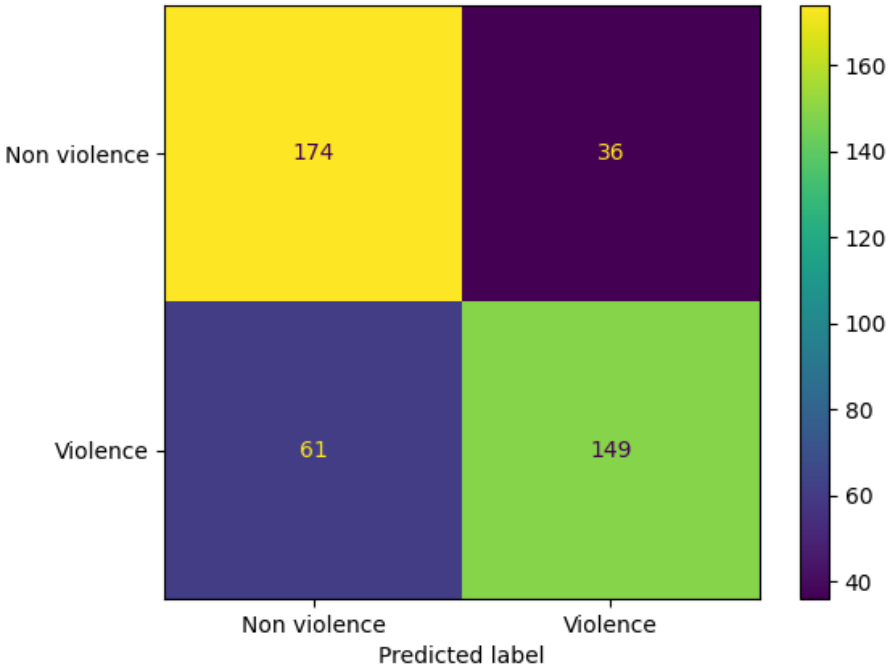


Figure 2.18: EfficientNetB0: third model

The results obtained (Fig. 2.18) are better than the ones in fine tuning, this models also obtains obtains similar results in terms of accuracy and violence recall, to the third ResNet model. We follow this approach unfreezing another half of the fifth convolutional block as seen in listing 2.12.

```
def EfficientnetFourthModel(input_shape):
  model = Sequential()
  model.add(RandomFlip('horizontal'))
  model.add(RandomRotation(0.1))
  EfficientNetB0 = EfficientNetB0(weights='imagenet', include_top=False,
   input_shape=input_shape)
  set_trainable = False
  for layer in EfficientNetB0.layers:
    if layer.name == 'block5c_expand_conv':
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False
  model.add(EfficientNetB0)
  model.add(GlobalAveragePooling2D())
  model.add(Dense(512, activation='relu'))
```

```
  model.add(Dropout(0.2))
  model.add(Dense(256, activation='relu'))
  model.add(Dense(128, activation='relu'))
  model.add(Dense(1, activation='sigmoid'))
  return model
```

Listing 2.12: EfficientNetB0 fourth model

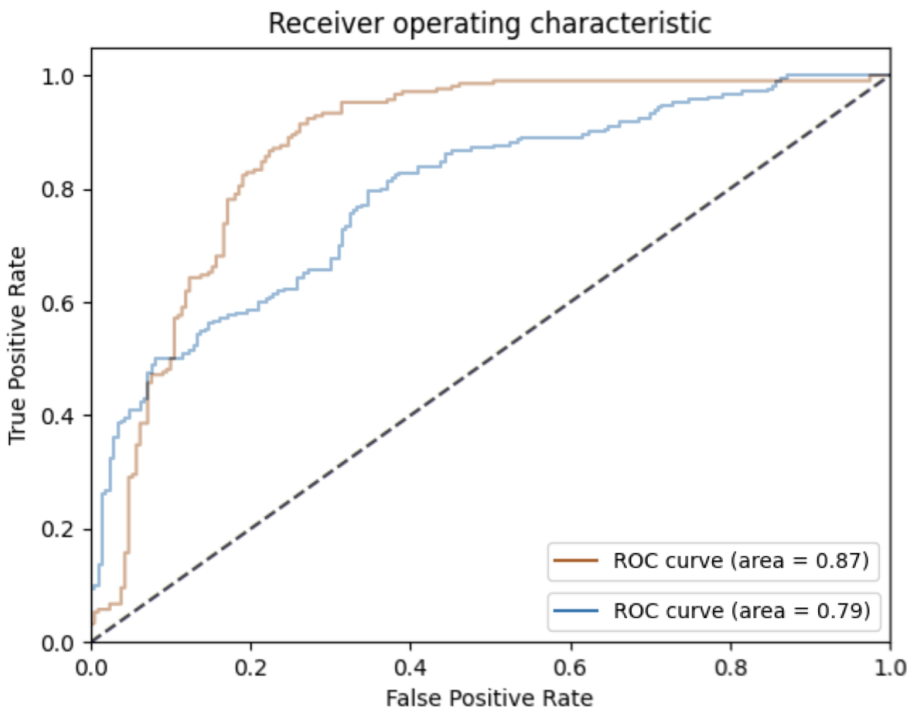


Figure 2.19: ROC of the EfficientNetB0: fine tuning 1 vs fine tuning 2

Looking at the ROC curves (Fig. 2.19) we cannot draw conclusions on the winner but the AUC is better and also between 0.1 and 0.8 false positive rate the fourth model performs better than the third.

However we try to push the model to the limit adding another dense layer in order to improve the classifier performance and adding also dropout to avoid overfitting as seen in listing 2.13.

```
def EfficientnetFifthModel(input_shape):
  model = Sequential()
  model.add(RandomFlip('horizontal'))
  model.add(RandomRotation(0.1))
  EfficientNetB0 = EfficientNetB0(weights='imagenet', include_top=False,
   input_shape=input_shape)
  set_trainable = False
  for layer in EfficientNetB0.layers:
    if layer.name == 'block5c_expand_conv':
        set_trainable = True
    if set_trainable:
        layer.trainable = True
    else:
        layer.trainable = False
  model.add(EfficientNetB0)
  model.add(GlobalAveragePooling2D())
  model.add(Dense(1024, activation='relu'))
  model.add(Dropout(0.3))
  model.add(Dense(512, activation='relu'))
```

```
    model.add(Dropout(0.2))
    model.add(Dense(256, activation='relu'))
    model.add(Dropout(0.1))
    model.add(Dense(128, activation='relu'))
    model.add(Dense(1, activation='sigmoid'))
    return model
```

Listing 2.13: EfficientNetB0 fifth model

We obtain a good model (Fig. 2.20), with a 78% accuracy with a strong recall on violence, that we used as a second choice parameter.

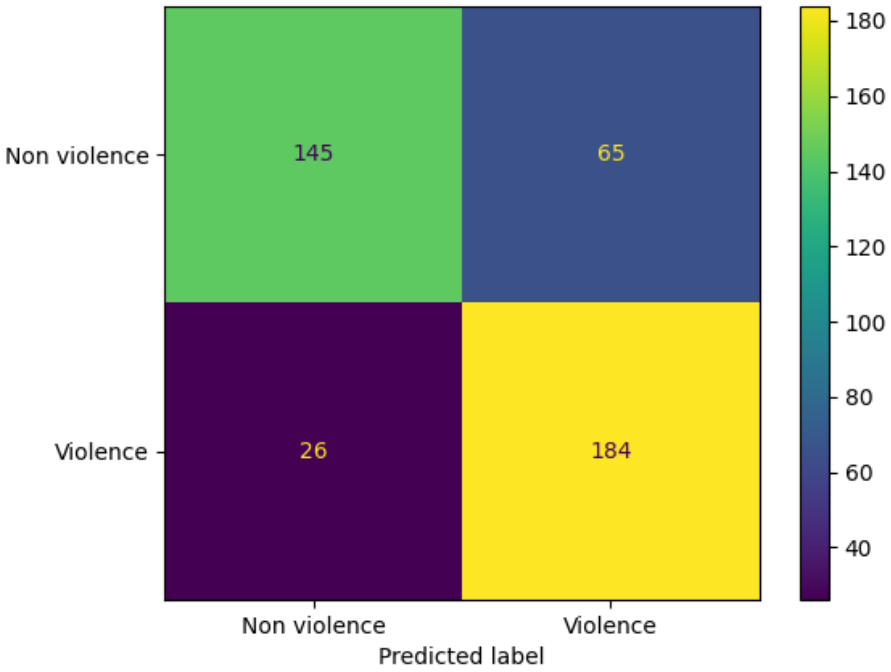


Figure 2.20: EfficientNetB0: fifth model

	Accuracy	Violence recall	Non Violence recall	Approach
EfficientNetB0 1st model	0,6214	0,3857	0,85714	Feature extraction
EfficientNetB0 2nd model	0,6928	0,4809	0,90476	Feature extraction
EfficientNetB0 3rd model	0,7690	0,70952	0,82857	Fine tuning
EfficientNetB0 4th model	0,7762	0,7191	0,8333	Fine tuning
EfficientNetB0 5th model	0,7833	0,8762	0,6905	Fine tuning

Table 2.3: EfficientNetB0 accuracy and recall

9 Reamining problems and solutions

The ResNet50 and EfficientNetB0 models brought some improvements to the accuracy of the model in comparison to the ones we developed, however we thought we could have done better, the main issue, as said before, is the dataset, it is too dirty and it has no action frames or bounding boxes to help the model learn the features of the videos. This leads, during the frame extract phase, to images before the violent acts or after them to be fed to the model with a violence label, this makes the model learns the wrong features. To solve this problem we could

have discarded the *bad* frames, but for this we would need to manually remove them, which would have been too much of a workload and would have been outside of the scope of the project.

What the 2D models lack is context, if the model could evaluate more correlated frames before generating an output it would, in our minds, have been able to learn the features of the videos better, this is the main reason why the 2D models proved unsatisfactory. So we decided to try a different approach, the 3D CNN one.

Chapter 3

3D CNN approach

1 Introduction

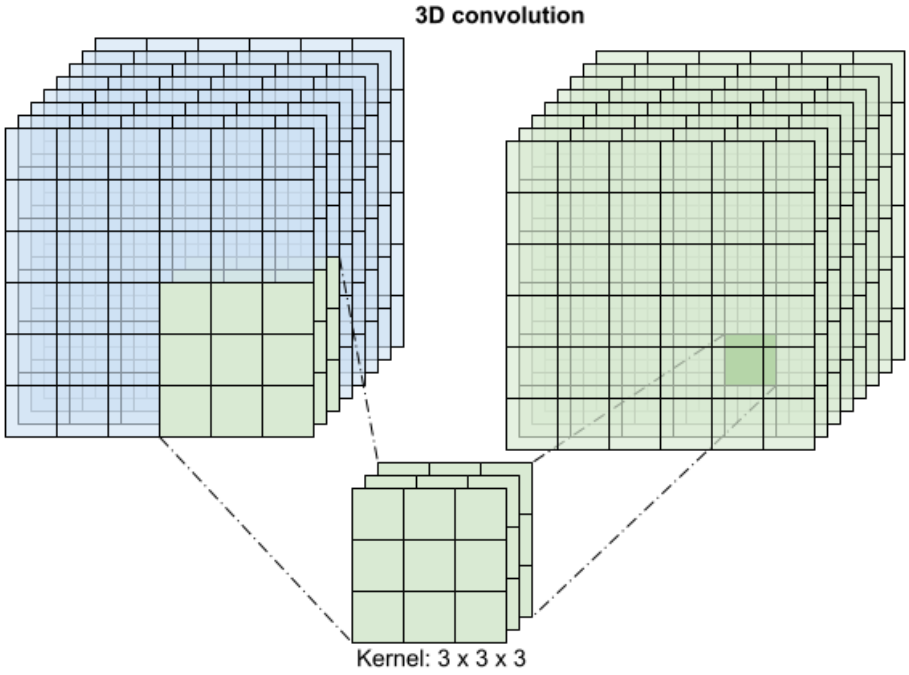


Figure 3.1: How a 3D CNN works

2D CNN networks operate by applying convolutions spatially, traversing both the horizontal and vertical dimensions of the input data. The convolutional operation employs 2D kernels, usually specified in terms of height and width. Channels, representing different aspects of the input data (e.g., Red, Green, Blue in color images), are a common feature of 2D CNNs.

On the other hand, 3D CNNs extend the convolutional approach to three-dimensional data, a structure commonly found in video or volumetric datasets. The input to a 3D CNN includes not only width and height but also a third dimension, often representing depth or frames in the temporal domain. Consequently, the kernels used in 3D CNNs are three-dimensional, incorporating depth, height, and width. This extension allows the model to capture spatial features across multiple frames, introducing a temporal aspect to the convolutional operation. This makes them particularly suitable for tasks involving video analysis and scenarios where temporal information is crucial like our *violence recognition* problem.

While 2D CNNs have proven effective for traditional image-related tasks like classification, object detection, and segmentation, 3D CNNs are specifically designed for applications where

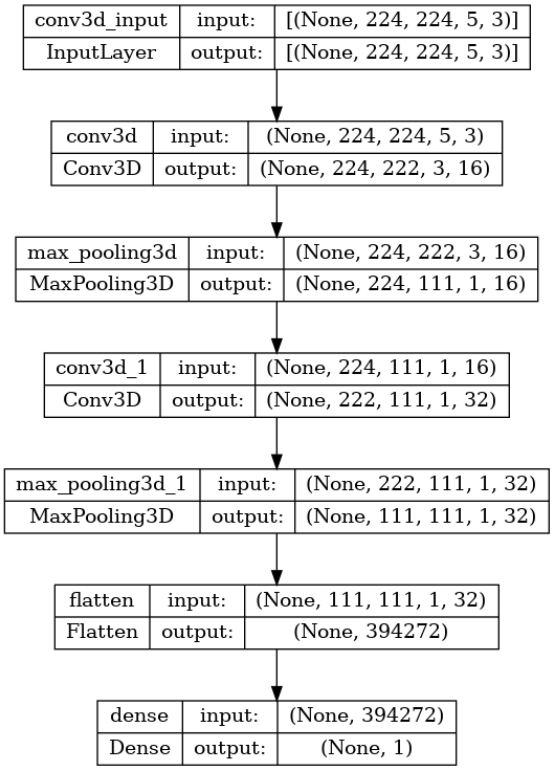


Figure 3.2: Schema of the first 3D CNN

temporal information is essential. However, the use of 3D CNNs comes at a higher computational cost due to the increased complexity introduced by the additional dimension. This would have proved to be a problem for us due to the fact that we did not have a GPU at our disposal and the ones provided by Google Colab were not capable enough to handle big models and big data-sets.

2 Implementation

As explained in section 1 , the pickles approach allowed us to generate files containing the data needed to train and test the model. The *number of frames* and the *fps* values for each pkl_config were used to test the same model on different input data like burst of 1, 2 or 3 seconds. As shown in Fig. 3.2, the model is a very simple one, with 2 convolutional layers, 2 max pooling layers and 1 fully connected. The idea is not to dwell on the model and how to improve it, but to show that the addition of "context" to the input data, in this case the temporal dimension, allows the model to achieve better results independently from the complexity of the model.

The very first test was done with a number of frames (NoF) equal to 5 and fps equal to 5, meaning that the model would receive bursts of 1 second. Fig. 3.3 shows an interesting result, it is very similar to the best 2D scratch one with a slightly better recall score in general.

The second test was done with NoF equal to 10 and fps equal to 5, meaning that the model would receive bursts of 2 seconds. This was done to see if the model would improve with more context, since 1 second of video does not carry much information.

Finally the third test was done with NoF equal to 15 and fps equal to 5, meaning that the model would receive bursts of 3 seconds. Keep in mind, however, that since some of the original dataset is made of videos shorter than 3 seconds both training and test sets are reduced. This means that results are to be taken lightly since they are not representative of the original dataset. In addition the previous model had to be simplified to avoid making it too complex for the smaller dataset, the new schema can be seen in Fig. 3.5. The results shown in Fig. 3.6

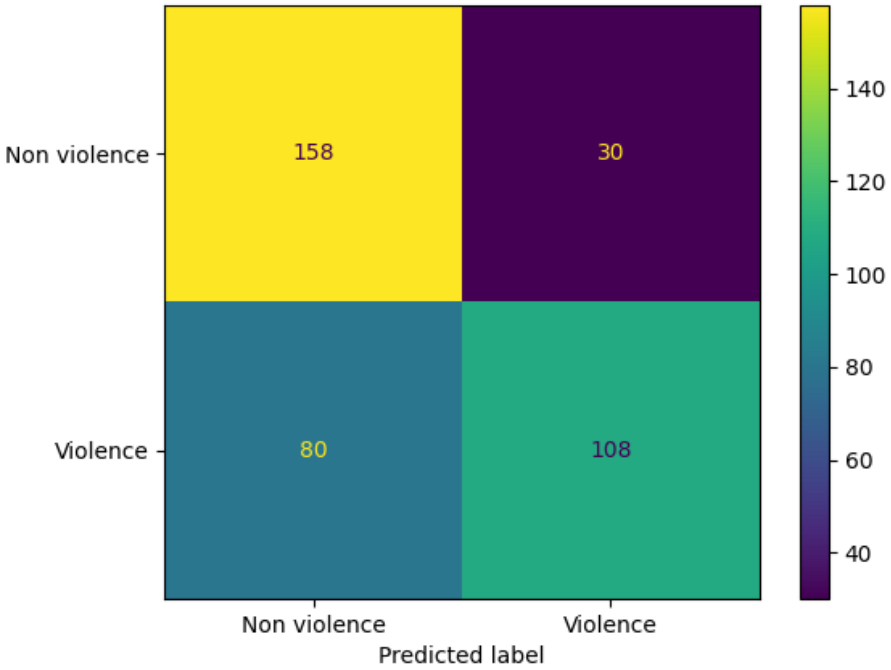


Figure 3.3: Confusion matrix of the first 3D model

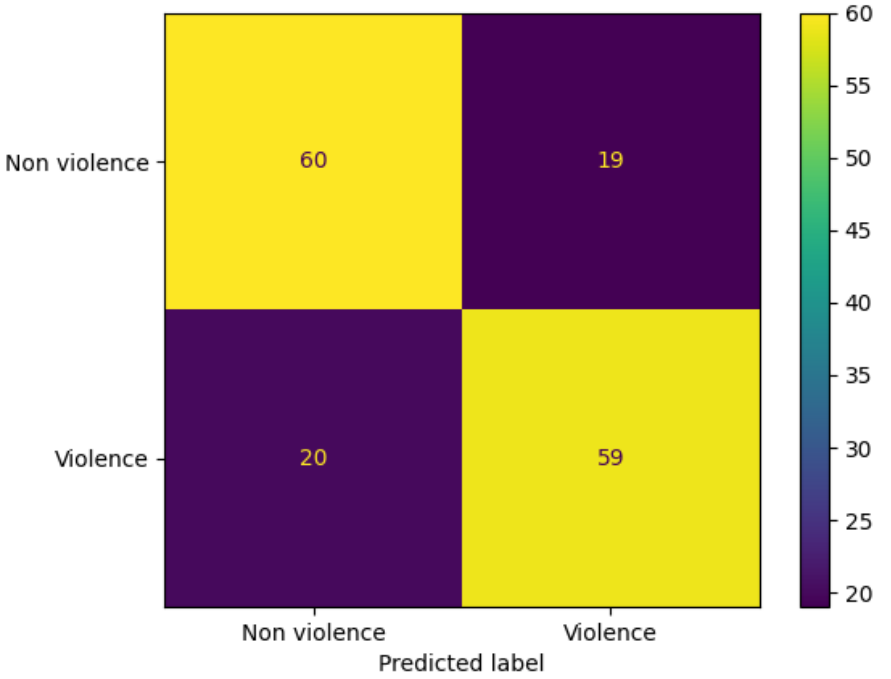


Figure 3.4: Confusion matrix of the second 3D model with 2 seconds bursts

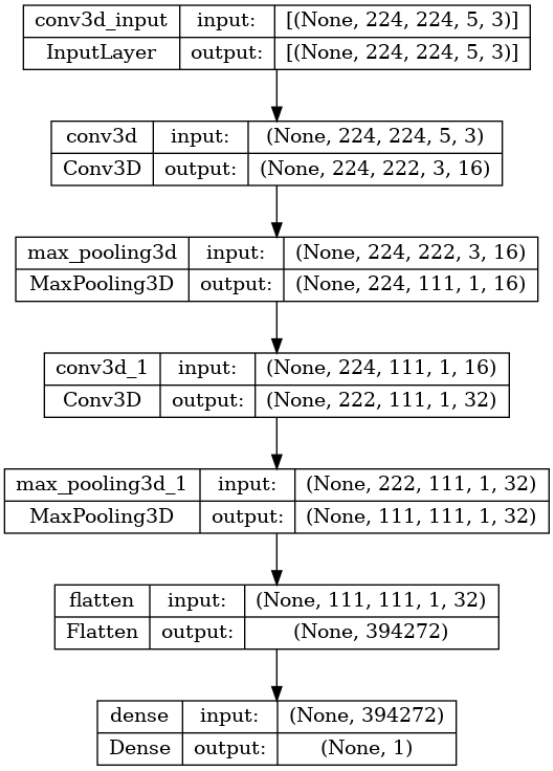


Figure 3.5: Schema of the second 3D CNN

are the best one so far, with a very good recall score for both classes. This is due to the fact that the model has more context to work with and can better understand the situation.

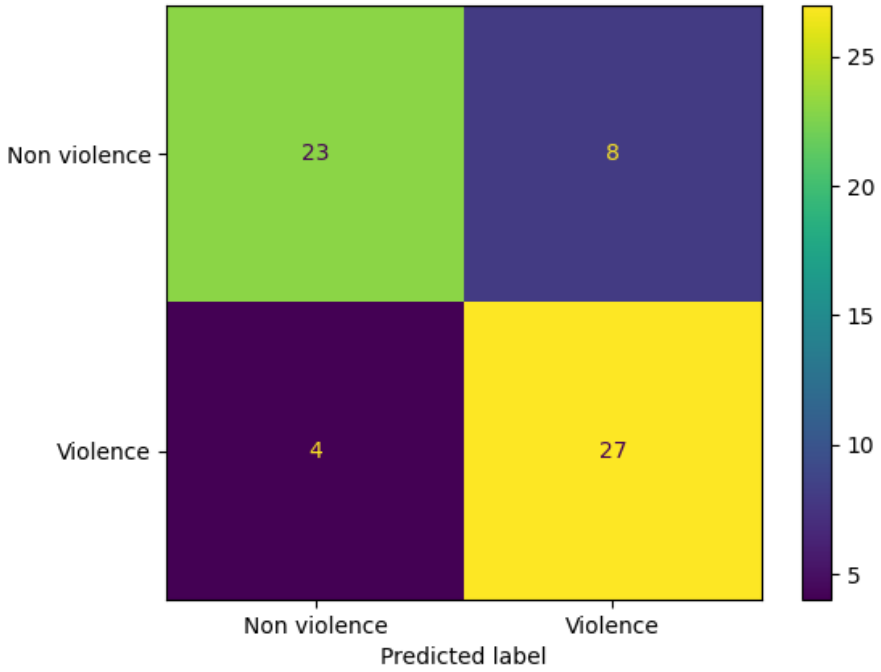


Figure 3.6: Confusion matrix of the second 3D model with 3 seconds burst

Chapter 4

Conclusion

Finally a table with the accuracy of all the models can be seen in Tab. 4.1. This proves that the 3D CNN approach is valid and that it can be used as an alternative to the 2D one. However, the 3D approach is more computationally expensive and requires more time to train and test, but it can achieve better results. Another thing to keep in mind is that the 3D approach seems to gain from longer bursts of frames, giving the model more "context". This means that the 3D method is more suitable for real time analysis of CCTV footage, where the model receives bursts of frames and is able to decide if there is violence or not by analyzing the evolution of the features over time. This is not the case for the 2D CNN approach that is more suitable for static images, where the model is able to decide if there is violence or not by analyzing the features of a single frame, but it is not able to understand the evolution of the action over time.

	Frame burst	**Accuracy**	**Violence recall**	**Non violence recall**
Best 2D scratch	Single frame	0,62619	0,49048	0,7619
Best ResNet50	Single frame	0,7643	0,6952	0,8333
Best EfficientNetB0	Single frame	0,7833	0,8762	0,6905
First 3D model	1 second burst	0,7074	0,5744	0,8404
Second 3D model	2 seconds burst	0,7532	0,7468	0,7595
Second 3D model	3 seconds burst	0,8065	0,8710	0,7419

Table 4.1: Models accuracy