# University of Salerno


## Machine Learning project activity report

Gruppo 10


| | | |
|---|---|---|
| Membro 1: | Battipaglia Valerio | mat.0622702091 |
| Membro 2: | Caso Antonio | mat.0622702068 |
| Membro 3: | Dell'Orto Giuseppe Maria | mat.0622701935 |


Academic Year 2022-2023

# Contents

# 1 Introduction

In the field of safety and surveillance, fire detection plays a crucial role in preventing incidents, protecting people and resources, and responding promptly to emergency situations.

Strictly related to this reality, the ***Onfire*** competition aims to bring forth efficient and reliable systems of swift *fire detection*; in order to partecipate, we built a net starting from the lightweight net **SqueezeNet** and used it as a feature extractor for individual video frames, integrating it with a Long-term Recurrent Convolutional Network (LRCN) architecture for video analysis.

In particular, the chosen approach was based both on Convolutional Neural Networks and LSTM (Long Short-Term Memory) Networks, given that they proved themselves effective in recognising complex patterns and in modelling of temporal relations in provided data.

To train and test the network, a wide range of differentiated videos was used, both containing and not containing fire hazards.

In this report, we will comprehensively address the entire process of conceptualization, creation, and utilization of the network, elucidating the pre-processing, model architecture design, training, and result evaluation. Additionally, we will discuss the model's performance, the challenges encountered during development, and potential future directions for enhancing the system.

# 2 Requirements

A set of requirements has been defined in order to guide the ordeal and have a preferred kind of network to be used in particular conditions (such as microcontrollers) and in order to build networks sufficiently sophisticated to be actually reliable.

## 2.1 Description

- **Real-Time Fire Detection**: the algorithm should be able to detect fires (flames and/or smoke) in real-time from video sequences acquired by fixed CCTV cameras.
- **Lightweight**: the algorithm should be designed to run on "intelligent cameras" or embedded systems with limited computational resources.
- **Processing Frame Rate**: the methods' processing speed is to be measured in terms of the average number of frames processed in one second.
- **Memory Usage**: the memory occupied by the methods on the target GPU is to be measured to evaluate the computational resources needed for real-time processing.

- **Fire Detection Accuracy**: the performance of the methods will be evaluated in terms of fire detection capabilities, including both detection errors and notification speed:

  **Detection errors**: an indicator of the network's efficiency would be the number of false positives during a validation conducted on videos with no presence of fire, and the number of false negatives during a validation conducted on videos with presence of fire.

  **Promptness in Notification**: the methods should generate immediate notifications for the alarm centers of competent authorities as soon as the fire is detected (with respect to the time manually annotated by human hands). The average delay between fire start and notification will be evaluated. Detection delays longer than five seconds are set to be marked as errors, that is incapability of detecting the fire.

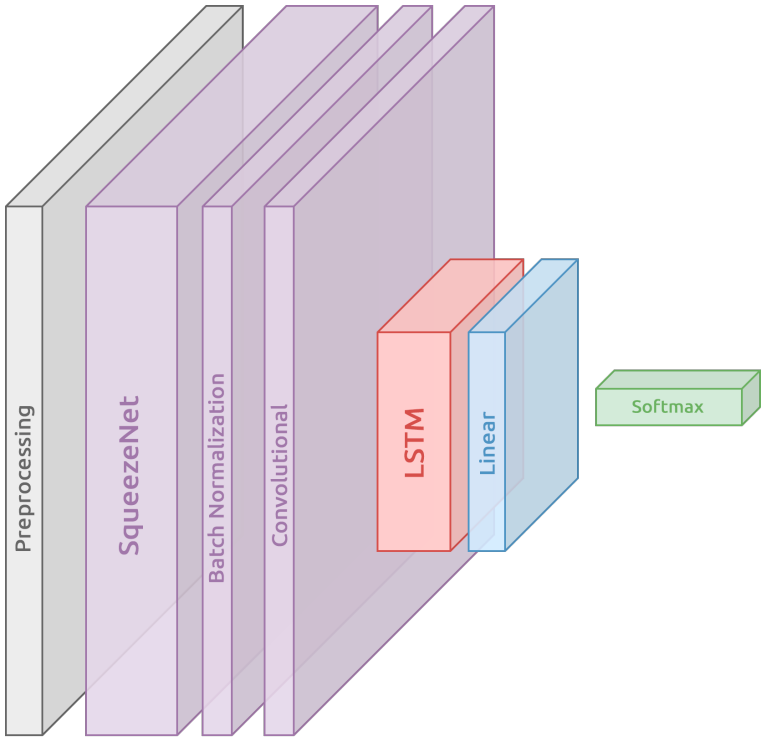
## 2.2 Metrics for evaluating results

- **Precision**: precision measures the capability of the methods to avoid false positives. It is the ratio of true positive detections to the total number of detections. The higher the precision, the higher the specificity of the method.
- **Recall**: recall evaluates the sensitivity of the methods to detect fires. It is the ratio of true positive detections to the total number of actual fires present in the positive videos. The higher the recall, the higher the sensitivity of the method.
- **Normalized Average Notification Delay**: the normalized average notification delay is a normalized version of the average notification delay. It ranges from 0 to 1, with 0 indicating the fastest notification and 1 indicating the slowest notification.
- **Normalized Processing Frame Rate**: the normalized processing frame rate is a normalized version of the processing frame rate. It is computed relative to the minimum processing frame rate needed to achieve real-time performance.
- **Normalized Memory Usage**: the normalized memory usage is a normalized version of the memory usage. It is computed relative to the maximum GPU memory available on the target processing device.
- **Fire Detection Score**: the Fire Detection Score is the overall ranking metric that takes into account the trade-off between fire detection accuracy, notification promptness, and required processing resources. It is computed based on Precision, Recall, Normalized Average Notification Delay, Normalized Processing Frame Rate, and Normalized Memory Usage.

# 3 Design

In this section, we will discuss the design choices made during the process of creating the network.

## 3.1 Architecture




A *Long-term Recurrent Convolutional Network* (LRCN) was used in this project for real-time fire detection from video sequences given its ability to effectively capture both spatial and temporal features in video data.

LRCN is a combination of *Convolutional Neural Network* (CNN) and L*ong Short-Term Memory* (LSTM) layers, and thus combines the power of Convolutional Neural Networks (CNNs) and LSTMs. The initial CNN backbone is responsible for extracting spatial features from individual frames. These features are then fed into the LSTM layers, which capture temporal patterns across multiple frames. This combination of spatial and temporal feature extraction enables the network to capture complex spatiotemporal patterns associated with fire events.

The CNN backbone is obtained using **SqueezeNet**, which is a lightweight and efficient Convolutional Neural Network (CNN) architecture designed to reduce

the number of parameters and model size while maintaining accuracy. This is in accordance to the desired *lightweigth* aspect of the network.

In particular, the pre-trained *SqueezeNet1.0* model was employed, which had been trained on a large-scale dataset and achieved reasonable accuracy on various computer vision tasks. This allowed us to leverage knowledge from a large dataset and generalize well to detect features relevant to fire detection. In fact, SqueezeNet's intermediate layers act as feature extractors from individual frames in the video. These features capture spatial information, which is then fed into the LSTM layers to model temporal dependencies across consecutive frames.

## 3.2 Layers

We used only one layer of CNN and two layers of LSTM.

As previously said , our model uses a pre-trained SqueezeNet as the CNN backbone to extract spatial features from individual frames. However, the last fully connected layers of the SqueezeNet are modified to adapt it for the LRCN task. The first fully connected layer is replaced with a 1x1 convolutional layer to provide a feature vector for the LSTM. The main reason for using only one CNN layer is to keep the network lightweight and efficient, which aligns with the requirements mentioned in the contest description.

Regarding the two layers of LSTM, on the other hand, we chose such a number in order to both accomodate efficiency of the network and avoid computational burden while also avoiding potentially *overfitting the data*, which, as a matter of fact, proved to be a critical matter to be solved in the context of providing an efficient network.

Using a single LSTM layer might not be sufficient to capture long-term dependencies and complex temporal patterns, so having two represented a good choice that allows the network to capture more complex temporal dependencies and patterns across consecutive frames in the video sequences.

## 3.3 Modified CNN Layers

SqueezeNet has been integrated with a **normalization layer**, specifically Batch Normalization, to decouple the different layers of the network. Batch Normalization helps stabilize and accelerate the training process by normalizing the activations between layers. This modification was introduced to improve the convergence and overall performance of the network.

SqueezeNet has been integrated with an **additional convolutional layer** to provide the LRCN with a feature vector. This new convolutional layer has 512 input channels (corresponding to the output channels of the last Fire module in SqueezeNet) and the desired number of output channels ($feature\_dim$). By adding this layer, the network can extract relevant features from the input images

and generate a feature vector that is subsequently fed into the LSTM layers to capture temporal dependencies in the video sequence.

### 3.4 Regarding output channels and hidden units

The original SqueezeNet model outputs 512 channels after the last Fire module. Instead of using this output directly, a custom convolutional layer with 512 input channels and feature_dim (which is set to 256) output channels is added. This reduction in the number of output channels to 256 serves as a dimensionality reduction step. It helps to control the number of parameters and memory usage while still preserving useful information for the subsequent LSTM layers.

The reason for feature_dim to be equal to 256 is to avoid overfitting, while also being efficient in memory usage and partially based on empirical performance improvement.

On the other hand, the choice of 256 hidden units in the LSTM layers is a design choice that balances model complexity and learning capacity. Having a smaller number of hidden units reduces the risk of overfitting, especially when the dataset is not very large.

### 3.5 Dropout

Commonly used as a regularization tecnique to mitigate overfitting, it works by choosing random neurons (60% of LSTM neurons in this case) and setting their output to zero, making the network more robust and preventing it from relying too heavily on specific neurons.

In our model, the Dropout layer has been used in the LSTM layer. Specifically, the line Dropout(0.6) in the code sets a dropout rate of 0.6; a choice based both on common practises (values between .5 and .7 are used) and empirical performance.

### 3.6 Classification Head

It is the final part of the model, the one intended to give predictions based on extracted features from previous layers. In our LCRN, it is obtained by combination of LSTM layers and one fully connected linear layer.

The LSTM layer is used to model sequential data and capture dependencies, so it can process the sequence of features extracted by the CNN. Then, the *fully connected layer* takes the final hidden state of the LSTM and maps it to the number of output classes (2 in this case). This result is then fed to the **Softmax** activation function which provides us with an estimate of confidence in the predictions.

### 3.7 Softmax Activation Function

It was used for a wide variety of reasons:

- It is useful to distribute the raw scores into a probability distribution, so that values are non negative and sum up to 1; this is made so that the outputs are interpreted as class probabilities.
- With the aim of Multi-Class classification, the softmax function can handle multiple classes.
- It normalizes the values so that they all fall in a valid range; this allows for quicker convergence.
- It is worth reminding that, aside from design choices, the softmax is differentiable, which is essential for performing backpropagation during training. Backpropagation allows the model to update its parameters using gradient-based optimization algorithms like stochastic gradient descent to minimize the classification error and improve the model's performance.

### 3.8 Pre-processing and augmentations

We applied a series of commonly used tecniques:

- The images are resized to a fixed size of 256x256 pixels and then center-cropped to a smaller size of 224x224 pixels (standard input size for SqueezeNet). This resizing and cropping were applied both to ensures that all input images had the same dimensions and to make sure that the network would not learn patterns connected to dates and timestamps, situated on the corners of the frame.
- The pixel values of the images are normalized using mean and standard deviation values commonly used for pre-trained models. Normalization helps to bring the pixel values within a specific range so to improve convergence and stability. In this case, the mean and standard deviation values used are [0.485, 0.456, 0.406] and [0.229, 0.224, 0.225], respectively.
- We introduiced horizontal flipping, random rain, random snow, and random fog in the images without changing their semantic content. This way, the model learns to be more robust to different environmental conditions and improves its performance on unseen data.

### 3.9 Hyper-Parameters

The first hyperparameter to be discussed is *feature_dim*: it defines the dimensionality of the output feature from the CNN. It is set to 512, which means the CNN output will have a feature vector of size 512. This value was chosen based on the original architecture of SqueezeNet, where the output of the last layer before the classification head has 512 channels.

After that, *hidden_size*, which is the size of the LSTM hidden state and is set to 256. We opted for a relatively moderate-sized hidden state to balance model capacity and computational efficiency.

Regarding the **learning rate** to be used during training, a lot of time and computational efftor was profused in order to device the best value, that is the value that allowed the model to (we would have liked so) converge the quickest and to oscillate the least in the metrics of accuracy and loss. In the end, given that no significant change could be accomplished, a quite standard value of 1e-4 was chosen.

Last but not least is the **regularization term**. Regularization is used to prevent overfitting and improve generalization to unseen data but in this case we found that regularization was not necessary due to the fact little to no significant change could be accomplished by addind such element.

# 4 Implementation

Here follows a description and analysis of the implementation and code that makes up the network.

## 4.1 train.ipynb

### 4.1.1 Pip Install

The *pip* command is used to install both the *striprtf* package (used to convert RTF text into plain text removing any formatting information) and the *albumentations* package, which is an essential component of the pre-processing phase.

### 4.1.2 Dataset population

For the objectives of this research project, we were granted authorization to employ the MIVIA Fire detection dataset, comprising 330 videos, with 246 of them containing fire and smoke video clips. The designated dataset class utilized in this investigation was VideoFrameDataset, as delineated in the following description.

Nevertheless, preliminary to leveraging the class, it was essential to undertake video preprocessing to extract discrete frames from each video. This preprocessing task was performed before-hand using the FFMPEG tool, enforcing a frame rate of 1 FPS. First of all, we import the gdown library in order to download files from a Google Drive directory, where both the **frames** and the **labels** (GT) will be located.

The function `download_google_file` that takes two arguments: `shader_url` (the URL of the file to be downloaded) and `output_name` (the name with which the downloaded file will be saved).

```
import gdown
def download_google_file(shader_url, output_name):
  id_url = "https://drive.google.com/uc?id=" + shader_url.split("/")[5]
  gdown.download(id_url, output_name)

download_google_file("https://drive.google.com/file/d/15S-MYaon0TCgk6UxkpObwjwrm_CdsMBL/view?usp=sharing", "FRAMES.tar.xz")
!tar -xvf FRAMES.tar.xz

download_google_file("https://drive.google.com/file/d/123AcAQCldRNE6iKpXuCaVtsaR3uHIOeN/view?usp=sharing", "GT.zip")
!unzip GT

!mkdir -p GT/VIDEOS/TRAINING_SET
!mv GT_TRAINING_SET_CL0 GT/VIDEOS/TRAINING_SET/0
!mv GT_TRAINING_SET_CL1 GT/VIDEOS/TRAINING_SET/1
```

In the function, we first extract the file ID from the Google Drive URL. The URL is split using / as a delimiter, and the 6th element (index 5) is taken as the file ID. Then, we construct a new URL by appending the file ID to the base Google Drive URL.

Finally, the `gdown.download` function is used to download the file from the constructed URL and save it with the specified `output_name`.

Then, we both download, rename and unzip, the .zip archive containing the previosuly extracted frames. Subsequently, we do the same thing with the archive containing the labels. At this point, we create a directory named `GT/VIDEOS/TRAINING_SET` and move to it the folders that were previously extracted. In this case, we move both the labels for videos containg and not containing fire.

Having done this, we move to define the dataset class to be used and its helper classes.

```
import os
import os.path
import numpy as np
from PIL import Image
from torchvision import transforms
import torch
from typing import List, Union, Tuple, Any
from striprtf.striprtf import rtf_to_text
import albumentations
```

**imports** A series of imports is made in order to have all the proper tools to process labels, to set and manage paths, to work with tensors and images, and to apply trasformations and augmentations.

**class VideoRecord** Here we define this helper class, extending the `object` class, which is used to encapsulate information about a video record, such as the video file path, start frame, end frame, and label. It provides easy access to these attributes through the defined property methods, allowing other parts of the code to interact with video records easily and efficiently.

```
class VideoRecord(object):

    def __init__(self, row, root_datapath):
        self._data = row
        self._path = os.path.join(root_datapath, row[0])

    @property
    def path(self) -> str:
        return self._path

    @property
    def num_frames(self) -> int:
        return self.end_frame - self.start_frame + 1

    @property
    def start_frame(self) -> int:
        return int(self._data[1])

    @property
    def end_frame(self) -> int:
        return int(self._data[2])

    @property
    def label(self) -> Union[int, List[int]]:
        if len(self._data) == 4:
            return int(self._data[3])
        else:
            return [int(label_id) for label_id in self._data[3:]]
```

The constructor of the class takes two input parameters, `root_datapath` which is the system path to the root folder of the video, and `row` which is simply a

list of elements that represent characteristics about the video. We store both of these values in two private attributes; in particular, joining the path of the video (row[0]) with the root_datapath gives us the full path of the video.

After this, a series of property decorators are defined. For instance, the first one allows the path method to be accessed as an attribute without the need for parentheses. The path method returns the video file path.

`num_frames()` calculates and returns the number of frames in the video. It does this by subtracting the start_frame from the end_frame and adding 1 (since the end frame is inclusive).

`start_frame()` returns the start frame of the video. The start frame is the second element in the row list.

`end_frame()` returns the end frame of the video. The end frame is the third element in the row list.

`label()` returns the label associated with the video. The method checks the length of the row list to determine whether there is a single label or multiple labels. At this point, if the row list has more than four elements, then the video in question has surely more than one label, so the method returns a list containing all the labels (converted from strings) from the fourth element onwards in the row list. Otherwise, if the elements are exactly four, only one label is applied to the video.

**class VideoFrameDataset** The main purpose of this dataset class is to load video frames with sparse temporal sampling, meaning it doesn't load every frame of a video but instead samples a fixed number of frames (specified by `frames_per_segment`) from evenly chosen locations across the video. The sampling is divided into segments, and each segment contributes a set of consecutive frames to the dataset. It inherits from *torch.utils.data.Dataset*, making it compatible with PyTorch's DataLoader for seamless integration into the training pipeline.

The constructor method takes several arguments to configure the instance of the dataset:

- root_path: The root path where video folders lie. This is the directory containing video data.
- num_segments: The number of segments the video should be divided into. For each segment, we will take a total of five frames.
- frames_per_segment: The number of frames that should be loaded per segment.
- imagefile_template: The image filename template that video frame files have inside their video folders.

```
class VideoFrameDataset(torch.utils.data.Dataset):

    def __init__(self,
                 root_path: str,
                 num_segments: int = 1,
                 frames_per_segment: int = 3,
                 imagefile_template: str='{:05d}.jpg',
                 transform=None,
                 totensor=True,
                 test_mode: bool = False):
        super(VideoFrameDataset, self).__init__()

        self.root_path = root_path
        self.num_segments = num_segments
        self.frames_per_segment = frames_per_segment
        self.imagefile_template = imagefile_template
        self.test_mode = test_mode

        if transform is None:
            self.transform = None
        else:
            additional_targets = {}
            for i in range(self.num_segments * self.frames_per_segment - 1):
                additional_targets["image%d" % i] = "image"
            self.transform = albumentations.Compose([transform],
                                                    additional_targets=additional_targets,
                                                    p=1)
        self.totensor = totensor
        self.totensor_transform = ImglistOrdictToTensor()

        self._parse_annotationfile()
        self._sanity_check_samples()

    def _load_image(self, directory: str, idx: int) -> Image.Image:
        return np.asarray(Image.open(os.path.join(directory, self.imagefile_template.format(idx))).convert('RGB'))
```

- transform: Transform pipeline that receives a list of numpy images/frames. This pipeline will be applied to the frames to perform data augmentation or other transformations.
- totensor: If True, the loaded images will be converted to tensors (if not already in tensor format).
- test_mode: If True, frames are taken from the center of each segment instead of a random location in each segment.

In the end of it, the constructor of the parent class (torch.utils.data.Dataset) is called to initialize the dataset, given that the class at hand inherits from it. Then, the attributes are set using the arguments passed to the constructor. While still in the costrunctor, more operations are done.

The first part of the code checks whether any transformation (transform) has been provided to the dataset during initialization. If no transformation is provided (e.g transform is set to None), no augmentation has been specified. If a transformation is provided, the code creates an empty dictionary additional_targets.

The code then iterates `self.num_segments * self.frames_per_segment - 1` (given 1 segment and five frames, it would mean we iterate four times) times to create keys for the additional_targets dictionary. These keys are in the form of "image0", "image1", ..., up to "image(n-1)", where n is the total number of frames that will be sampled from the video. The purpose of these keys is to

provide distinct names for each image frame when applying the transformation. This is essential because each frame sampled from different segments needs to be treated independently by the augmentation process.

Then the transformation pipeline is created using albumentations.Compose. It includes the provided transformation (transform) as the main operation and assigns the additional_targets dictionary to it. This means that the same transformation will be applied to each image frame independently (for each segment). The parameter p=1 means that the transformation has a probability of 100% to be applied to each image frame.

The transformation pipeline is stored in the self.transform attribute. This attribute will later be used to apply the transformation to the loaded video frames before they are fed into the model.

The `self.totensor` attribute is set based on the *totensor* parameter passed during initialization. If `totensor` is True, it indicates that the data needs to be converted to PyTorch tensors after applying the augmentation. The `self.totensor_transform` attribute is initialized with `ImglistOrdictToTensor()`. This is a custom transformation from the dataset that converts the augmented images into tensors to make them suitable for processing with PyTorch models.

After setting up the transformation pipeline, the `_parse_annotationfile()` method is called. This method reads the annotation files for each video sample to get information about the start frame, end frame, and label. This information will be used to assign the correct label to each video sample during training.

The `_sanity_check_samples()` method is then called. This method performs a sanity check on the loaded video samples. It checks for potential issues, such as videos with zero frames or videos where the number of frames is insufficient for the specified segments and frames per segment. If any issues are found, warning messages are printed to alert the user about these problematic samples.

The `_load_image(self, directory: str, idx: int)` is an helper method used to load an image from a directory given its index idx. It opens the image using the provided imagefile_template and converts it to an RGB image (3 channels), returning the converted image.

```
def _parse_annotationfile(self):
    self.video_list = []
    for class_name in os.listdir(self.root_path):
        for video_name in os.listdir(os.path.join(self.root_path, class_name)):
            frames_dir = os.path.join(self.root_path, class_name, video_name)
            if os.path.isdir(frames_dir):
                frame_path = os.path.join(class_name, video_name)
                end_frame = len(os.listdir(frames_dir))

                annotation_path = frames_dir\
                    .replace("\\", "/") \
                    .replace("FRAMES/", "GT/") \
                    .replace(".mp4", ".rtf")

                with open(annotation_path, 'r') as file:
                    text = rtf_to_text(file.read())
                if len(text):
                    label = 1
                    start_frame = int(text.split(",")[0])
                    if start_frame == 0:
                        start_frame = 1
                else:
                    label = 0
                    start_frame = 1

                self.video_list.append(VideoRecord(
                    [frame_path, start_frame, end_frame, label],
                    self.root_path))
```

The `_parse_annotationfile()` method handles parsing the annotation file and populating the *video_list*. It iterates through the video directories, reads the annotation file, and extracts the label and frame information for each video sample. It creates a VideoRecord object for each video sample and adds it to the.

For each video sample, it follows these steps:

- It constructs the path of the video frames directory.
- Checks if the directory exists.
- Constructs the path for the video frames, combining the class name and video name.
- Determines the total number of frames in the video, which helps in setting the end frame.
- Constructs the path for the annotation file associated with the video.
- Reads the content of the annotation file and processes it using the rtf_to_text function.

- If the annotation contains valid information, it sets the label to 1 and extracts the start frame from the annotation. If the start frame is 0, it adjusts it to 1 to account for 1-based indexing.
- If the annotation is empty or doesn't contain valid information, it sets the label to 0 and the start frame to 1.
- Creates a new object called VideoRecord to store the gathered information, which includes the frame path, start frame, end frame, and label. This VideoRecord object is then added to the dataset's video list.

```
def _sanity_check_samples(self):
    for record in self.video_list:
        if record.num_frames <= 0 or record.start_frame == record.end_frame:
            print(f"\nDataset Warning: video {record.path} seems to have zero RGB frames on disk!\n")

        elif record.num_frames < (self.num_segments * self.frames_per_segment):
            print(f"\nDataset Warning: video {record.path} has {record.num_frames} frames "
                  f"but the dataloader is set up to load "
                  f"(num_segments={self.num_segments})*(frames_per_segment={self.frames_per_segment})"
                  f"={self.num_segments * self.frames_per_segment} frames. Dataloader will throw an "
                  f"error when trying to load this video.\n")
```

This method is responsible for checking the integrity of the video samples in the dataset. It ensures that each video has a valid number of frames and is compatible with the specified settings for the number of segments and frames per segment.

```
def _get_start_indices(self, record: VideoRecord) -> 'np.ndarray[int]':

    if self.test_mode:
        distance_between_indices = (record.num_frames - self.frames_per_segment + 1) / float(self.num_segments)

        start_indices = np.array([int(distance_between_indices / 2.0 + distance_between_indices * x)
                                  for x in range(self.num_segments)])

    else:
        max_valid_start_index = (record.num_frames - self.frames_per_segment + 1) // self.num_segments

        if max_valid_start_index <= 0:
            start_indices = np.multiply(list(range(self.num_segments)), max_valid_start_index)
        else:
            start_indices = np.multiply(list(range(self.num_segments)), max_valid_start_index) + \
                np.random.randint(max_valid_start_index,
                                  size=self.num_segments)

    return start_indices
```

This method returns the array of start indices, which will be used to load frames from each segment of the video during the data loading process. The choice of start indices is essential for effective and efficient sampling of frames from the video for training or evaluation purposes.

The method takes as input the VideoRecord object of which to determine the start indices, from where frames will be loaded for each segment of the video. The choice of start indices depends on whether the method is being used in test mode or not.

If the method is in test mode, the `distance_between_indices` is calculated as the difference between the total number of frames in the video (`record.num_frames`) and the number of frames required for loading a segment (`self.frames_per_segment`), divided by the number of segments (`self.num_segments`). This value represents the average distance between consecutive start indices for each segment.

An array of start indices (`start_indices`) is generated using a list comprehension, where each index x in the range of 0 to the number of segments is calculated. These start indices are evenly spread across the video frames, ensuring a regular sampling of frames for each segment.

On the contrary, if the method is not in test mode, we calculate `max_valid_start_index`, and this value ensures enough frames are available for loading a segment. Then, is the number is less than zero, it means there are not enough frames to sample from, and the method returns a set of start indices with a single value repeated for all segments.

If instead it is indeed greater than zero, then an array of start indices is generated where the indices are randomized: this results in start indices that are approximately evenly spread across the available frames, with some randomization to provide temporal diversity in sampling.

```
def __getitem__(self, idx: int) -> Union[
    Tuple[List[Image.Image], Union[int, List[int]]],
    Tuple['torch.Tensor[num_frames, channels, height, width]', Union[int, List[int]]],
    Tuple[Any, Union[int, List[int]]],
    ]:

    record: VideoRecord = self.video_list[idx]

    frame_start_indices: 'np.ndarray[int]' = self._get_start_indices(record)

    return self._get(record, frame_start_indices)
```

As a magic method, this function defines the behavior when an item from the dataset is accessed using square brackets. It allows to retrieve the VideoRecord corresponding to the given index (idx) from the `video_list`, which contains all the video samples in the dataset. Not only that, allows also to access the `start_indices` of the record and return both the former and the latter.

This method allows to obatain the frames of a video at the corresponding indices. Both the VideoRecord and the start indices are taken as parameters, so that the method can cycle through the indices (after having initialized a list to contain the frames).

The function then iterates over each start_index in frame_start_indices. A frame_index variable is set to the current start_index. It then loads self.frames_per_segment consecutive frames from the video starting at frame_index. For each frame, it

```
def _get(self, record: VideoRecord, frame_start_indices: 'np.ndarray[int]') -> Union[
    Tuple[List[Image.Image], Union[int, List[int]]],
    Tuple['torch.Tensor[num_frames, channels, height, width]', Union[int, List[int]]],
    Tuple[Any, Union[int, List[int]]],
    ]:


    frame_start_indices = frame_start_indices + record.start_frame
    images = list()

    for start_index in frame_start_indices:
        frame_index = int(start_index)

        for _ in range(self.frames_per_segment):
            image = self._load_image(record.path, frame_index)
            images.append(image)

            if frame_index < record.end_frame:
                frame_index += 1

    if self.transform is not None:
        transform_input = {"image": images[0]}
        for i, image in enumerate(images[1:]):
            transform_input["image%d" % i] = image
        images = self.transform(**transform_input)

    if self.totensor:
        images = self.totensor_transform(images)
    return images, record.label
```

calls the _load_image function to read the image from the specified path and adds it to the images list. The frame_index is incremented until it reaches the end of the segment, ensuring that self.frames_per_segment frames are loaded from each segment.

After loading the frames, the function checks if self.transform is not None, meaning there are image transformations to apply. If transformations are present, the frames are combined into a dictionary transform_input with keys like "image0," "image1," etc., and corresponding frame images as values. The transformations are applied to the frames using the Albumentations library, specified by self.transform.

If self.totensor is True, the function applies the transformation self.totensor_transform to convert the list of frames (images) into a tensor format. The self.totenso_transform function converts the list of frames into a 4D tensor with the shape (NUM_IMAGES x CHANNELS x HEIGHT x WIDTH).

The function returns a tuple containing the loaded frames (either as a list of PIL images or as a tensor) and the label of the video sample (record.label).

```
class ImglistOrdictToTensor(torch.nn.Module):

    @staticmethod
    def forward(img_list_or_dict):
        if isinstance(img_list_or_dict, list):
            return torch.stack([transforms.functional.to_tensor(img)
                                for img in img_list_or_dict])
        else:
            return torch.stack([transforms.functional.to_tensor(img_list_or_dict[k])
                                for k in img_list_or_dict.keys()])
```

**class ImglistOrdictToTensor** This class is a custom PyTorch module that converts a list or a dictionary of numpy images to a single torch.FloatTensor. It is designed to be used as the first custom transformation for VideoFrameDataset. The forward method is a static method of the class and acts as the forward pass of the module. It takes `img_list_or_dict` as input, which can be either a list or a dictionary of numpy images.

If `img_list_or_dict` is a list, it means that multiple numpy images are provided as a list. The forward method uses a list comprehension to iterate over each numpy image in the list and for each numpy image it calls `transforms.functional.to_tensor` to convert the image to a torch tensor. The resulting torch tensors are stacked together using torch.stack to form a single tensor of shape (NUM_IMAGES x CHANNELS x HEIGHT x WIDTH).

If it is not a list, then it is a dictionary: it means that the images are provided as a dictionary where each key represents an image. The forward method uses a list comprehension to iterate over each key in the dictionary. For each key, it retrieves the corresponding numpy image using `img_list_or_dict[k]`. Similar to the list case, it converts each image to a torch tensor using `transforms.functional.to_tensor`. The resulting torch tensors are stacked together using torch.stack to form a single tensor of shape (NUM_IMAGES x CHANNELS x HEIGHT x WIDTH).

### 4.1.3 Model

```
from torch.nn import Module, LSTM, Softmax, Linear, Conv2d, BatchNorm2d, Dropout
from torch import nn
import torch
from torchvision import models
```

**imports** We imported the necessary classes and modules from PyTorch and torchvision to have these libraries available for use in the rest of the script.

```
class CNN(Module):

    def __init__(self, feature_dim) -> None:
        super().__init__()
        self.pretrained = models.squeezenet1_0(weights=models.SqueezeNet1_0_Weights.DEFAULT)

        self.pretrained.classifier[0] = BatchNorm2d(num_features=512)

        self.pretrained.classifier[1] = Conv2d(in_channels=512, out_channels=feature_dim, kernel_size=(1,1), stride=(1,1))

    def forward(self, x):
        return self.pretrained(x)
```

**class CNN** We define the CNN class by firstly, of course, extending the Module class which is the basis for all NNs. The constructor takes as input only the `feature_dim` value. We use the Module class' constructor, and then we set new properties:

- pretrained: we implement SqueezeNet in our model, and more importantly we also import the weights *as is*.
- self.pretrained.classifie[0]: one of the last two layers gets converted to a Normalization layer that normalizes the input along the batch dimension.
- self.pretrained.classifier[1]: the last layer is converted into a 2D convolutional layer, it takes an input with 512 channels (in_channels) and produces an output with feature_dim channels (out_channels). It uses a kernel size of (1,1) and a stride of (1,1), reducing the number of channels in the output to feature_dim.

The `forward(x)` method then manages the forward pass, passing the input tensor x which contains a batch of video sequences, where each video has multiple frames represented as a 3D tensor (channels, width, height).

It is a 5-dimensional PyTorch tensor, and the shape of the tensor is (batch_size, num_frames, channels, width, height) where:

- batch_size: This dimension represents the number of samples (videos) processed in each forward pass. It indicates how many videos are processed together as a batch.
- num_frames: This dimension represents the number of frames in each video sample. It indicates the length of the video, or in other words, how many frames are used to represent the video sequence.

- channels: This dimension represents the number of channels in each frame of the video. In the context of RGB videos, it is typically set to 3, as there are three color channels (red, green, and blue).
- width: This dimension represents the width of each frame in the video.
- height: This dimension represents the height of each frame in the video.

```
class LRCN(Module):

    NUM_CLASSES = 2

    def __init__(self, feature_dim=512, hidden_size=256) -> None:
        super().__init__()

        self.cnn = CNN(feature_dim)

        self.lstm = LSTM(
            input_size=feature_dim,
            hidden_size=hidden_size,
            num_layers=2,
            dropout=0.6,
            batch_first=True
            )

        self.fc = Linear(hidden_size, self.NUM_CLASSES)

        self.softmax = Softmax(dim=-1)
```

**class LRCN** First we define the number of classes with which our network has to work, which is two, then we define the constructor.

The constructor takes both feature_dim and hidden_size as input parameters. After calling the Module class' constructur, we istantiate the CNN providing 512 as the feature_dim. We then define the *LSTM*:

- input_size: it represents the number of features in the output of the CNN that is fed into the LSTM as input.
- hidden_size: represents the number of units (or hidden states) in the LSTM layer. It defines the dimensionality of the LSTM's hidden state, which captures information from previous time steps and helps in learning temporal patterns in the video data.
- num_layers: his sets the number of LSTM layers stacked on top of each other. In this case, two LSTM layers are used.
- batch_first=True: this parameter sets the order of the input tensor.

The we created a fully connected (linear) layer with an input size of hidden_size and an output size of self.NUM_CLASSES (which is 2 for binary classification) and assigned it to the attribute self.fc. This layer is used for the final classification.

Finally, we created an instance of the Softmax class with dim=-1 (along the last dimension) and assigns it to the attribute self.softmax.

**class LRCN** In the forward method, we defined the forward pass of the model. It takes the input tensor x and an optional argument verbose (default is False) to control whether to print the shape of intermediate tensors.

First, we unpack the shape of the input tensor x into individual variables for easier access. Then, we reshape the input tensor x to have the batch size

```
def forward(self, x, verbose=False):
    if verbose:
        print("Input ", x.shape)

    batch_size, num_frames, channels, width, height = x.shape
    conv_input = x.view(batch_size*num_frames, channels, width, height)

    if verbose:
        print("Conv input ",conv_input.shape)

    conv_output = self.cnn(conv_input)

    if verbose:
        print("Conv output ",conv_output.shape)

    lstm_input = conv_output.view(batch_size, num_frames, -1)

    if verbose:
        print("LSTM input ",lstm_input.shape)

    self.lstm.flatten_parameters()
    lstm_output, _ = self.lstm(lstm_input, None)

    if verbose:
        print("LSTM output ", lstm_output.shape)

    predictions = self.fc(lstm_output[:,-1,:])

    if verbose:
        print("Prediction ", predictions.shape)

    return self.softmax(predictions)
```

multiplied by the number of frames as the first dimension. It prepares the tensor for the convolutional part of the model.

After that we can apply the convolutional neural network (self.cnn) to the conv_input tensor and get the convolutional output.

At that point we can reshape again the convolutional output tensor back to the original batch size, number of frames, and flatten the other dimensions. It prepares the tensor for the LSTM part of the model.

We then apply the LSTM layer (self.lstm) to the lstm_input tensor to obtain the LSTM output and the hidden state (which is not used in this case).

At the end, we first apply the fully connected layer to the last time step of the LSTM output to obtain the final predictions and then call the softmax function to obtain class probabilities, and the result is returned as the output tensor.

### 4.1.4 Build the model

```
def build_model():

    model = LRCN(feature_dim=256, hidden_size=256)

    if torch.cuda.is_available():
        model = model.cuda()

    model = model.train()

    n = 0

    for p in model.parameters():
        p.requires_grad = False

    trainable_parameters = []
    for p in model.cnn.pretrained.classifier.parameters():
        trainable_parameters.append(p)
        n += len(p)

    for p in model.lstm.parameters():
        trainable_parameters.append(p)
        n += len(p)

    for p in model.fc.parameters():
        trainable_parameters.append(p)
        n += len(p)

    for p in trainable_parameters:
        p.requires_grad = True

    return model, trainable_parameters
```

Having done all the preparation, we can actually build our model. The first step is of course to instantiate the architecture of the network and provide the hyperparameters discussed above.

At this point, given the chance to explot CUDA's processing, we make use of such instrument and then launch the **training** of the model and set the model in training mode. This is essential because it activates certain operations like dropout, which is only applied during training to prevent overfitting.

We then loop through all parameters in the model and set requires_grad to False. This effectively freezes all parameters in the network, preventing them from being updated during training. This is done to ensure that the pretrained CNN weights remain unchanged while training the other parts of the network.

After that, the method creates a list trainable_parameters that will store all the parameters in the model that are trainable. It iterates through the parameters of the CNN classifier, the LSTM layer, and the fully connected layer, appending them to the trainable_parameters list. It also calculates the total number of trainable parameters by summing the length of each parameter.

After identifying the trainable parameters, the loop sets requires_grad to True for each parameter in the trainable_parameters list. This makes all these parameters trainable and subject to update during training.

Finally, the function returns the configured LRCN model and the list of trainable parameters. The model is now ready to be trained, with the CNN feature extractor frozen and only the LSTM and fc layers being updated during the training process.

### 4.1.5 Training

```
from torch.utils.tensorboard import SummaryWriter
import torch
import os
from tqdm import tqdm
from torch.utils.data import Subset, DataLoader
from torch.nn import CrossEntropyLoss
from datetime import datetime
from tensorboard import notebook
```

We import tools fundamental to the training phase such as DataLoader, CrossEntropyLoss library and the tensor board library. We then define the `start_tensorboard()` function, to be used to actually start the TensorBoard in order to give us insight on the evolution of the model.

```
def cross_validation_dataloaders(train_dataset, batch_size, K = 10):
    if os.path.exists("cross-val-indexes.pt"):
        indexes = torch.load("cross-val-indexes.pt")
    else:
        indexes = torch.randperm(len(train_dataset)) % K
        torch.save(indexes, "cross-val-indexes.pt")

    dataloader_params = {"batch_size": batch_size, "num_workers": 4, "pin_memory": True}

    train_folds, val_folds = [], []
    for k in range(K):

        val_fold   = Subset(train_dataset,   (indexes==k).nonzero().squeeze())
        train_fold = Subset(train_dataset, (indexes!=k).nonzero().squeeze())

        val_fold   = DataLoader(val_fold,   shuffle=False, **dataloader_params)
        train_fold = DataLoader(train_fold, shuffle=True,  **dataloader_params)

        val_folds.append(val_fold)
        train_folds.append(train_fold)

    return train_folds, val_folds
```

This function generates cross-validation data loaders for training and validation sets using the K-fold cross-validation technique. It aims to split the provided train_dataset into K folds, where K is typically set to 10.

After that, the method either loads precomputed cross-validation indexes from the file "cross-val-indexes.pt" if it exists, or generates new random cross-validation

indexes if the file does not exist. These indexes are used to split the dataset into K folds, ensuring that the data in each fold is unique and randomly distributed.

We then create a dictionary dataloader_params that contains parameters for the DataLoader objects, such as the batch size, the number of workers used for data loading, and pinning memory for faster transfer to the GPU if available. After that, we instantiate two lists that will contain the fold both during training and during validation.

The loop iterates K times, representing each iteration of the cross-validation process. It generates training and validation subsets (folds) based on the cross-validation indexes. For each iteration, it creates two DataLoader objects: one for the validation data and one for the training data. The Subset function from PyTorch is used to select the appropriate data based on the cross-validation indexes. The shuffle argument is set to True for the training DataLoader to ensure that the data is shuffled during each epoch, and False for the validation DataLoader to keep the data order consistent for evaluation purposes.

```
def one_epoch(model, lossFunction, optimizer, train_loader, val_loader, writer, epoch_num, fold):
    i_start = epoch_num * len(train_loader)
```

**open_epoch method** The `one_epoch(model, lossFunction, optimizer, train_loader, val_loader, writer, epoch_num, fold)` method is used to perform one epoch of training and evaluation for a given machine learning model. Its purpose is to update the model's parameters based on the training data and calculate performance metrics on the validation data for monitoring the model's progress.

The first line calculates the starting index for tracking training progress. It helps in recording the training metrics at the correct position in the tensorboard or writer.

```
for i, (X, y) in tqdm(enumerate(train_loader), desc="epoch {} - train".format(epoch_num)):

    if torch.cuda.is_available():
    X = X.cuda()
    y = y.cuda()

    optimizer.zero_grad()

    o = model(X)
    l = lossFunction(o, y)

    l.backward()
    optimizer.step()

    acc = (o.detach().argmax(-1) == y.detach()).float().mean()

    writer.add_scalars(f'train/loss', {
                                f'fold_{fold}': l.detach().item(),
                                }, i_start+i)
    writer.add_scalars(f'train/acc', {
                                f'fold_{fold}': acc,
                                }, i_start+i)
```

The loop's purpose is to iterate through the training data in batches using the train_loader. It goes through the entire dataset once during this epoch.

At that point, two lines move the input data and labels to the GPU if it is available, making use of GPU acceleration for faster computation.

Then the loop performs a single training iteration. It passes the input data X through the model, calculates the loss l between the model's predictions o and the target labels y, computes the gradients of the loss with respect to the model's parameters using backward(), and updates the model's parameters using the optimizer optimizer through step().

We then calculate the accuracy of the model's predictions on the current batch by comparing the predicted class labels to the ground truth labels. It uses the .argmax(-1) function to get the predicted class index with the highest probability, and detach() is used to prevent gradients from being computed for this operation.

At that point, we log the training loss and accuracy for each batch using the writer. It records the metrics for the specific fold identified by fold.

```
with torch.no_grad():
    val_loss = []
    val_corr_pred = []
    for X, y in tqdm(val_loader, desc="epoch {} - validation".format(epoch_num)):
    if torch.cuda.is_available():
        X = X.cuda()
        y = y.cuda()

    o = model(X)
    val_loss.append(lossFunction(o, y))
    val_corr_pred.append(o.argmax(-1) == y)

    val_loss = torch.stack(val_loss).mean().item()
    val_accuracy = torch.concatenate(val_corr_pred).float().mean().item()

    print("Validation loss and accuracy : {:.7f}\t{:.4f}".format(val_loss, val_accuracy))
    writer.add_scalars(f'val/loss', {
                                    f'fold_{fold}': val_loss,
                                    }, i_start+i)
    writer.add_scalars(f'val/acc', {
                                    f'fold_{fold}': val_accuracy,
                                    }, i_start+i)
return val_loss, val_accuracy
```

The block of code shown in the figure performs the validation step by iterating through the validation data in batches using the val_loader. During validation, no gradients are computed since we don't want to update the model parameters; hence, we use torch.no_grad() to disable gradient calculations.

Subsequently, we calculate the validation loss and accuracy for each batch of validation data. The predictions o are obtained by passing the input data X through the model, and the loss between the predictions and target labels y is calculated using the lossFunction. The correct predictions are also recorded in val_corr_pred for later computation of the overall validation accuracy. In the context of the one_epoch function, "correct predictions" refer to the instances where the model's predictions match the ground truth labels in the validation set.

We then compute the overall validation loss and accuracy for the epoch by averaging the loss values in val_loss and calculating the mean accuracy from the boolean tensors in val_corr_pred.

Again, we log the validation loss and accuracy using the writer, recording the metrics for the specific fold identified by fold.

Finally, the function returns the calculated validation loss and accuracy for the epoch. These metrics can be used to monitor the performance of the model during training.

```
def train(train_dataset, config, batch_size=32, epochs=200, K_cross_val=10, early_stopping_patience=5):
  lossFunction = CrossEntropyLoss()
  train_folds, val_folds = cross_validation_dataloaders(train_dataset, batch_size, K_cross_val)

  val_losses = torch.zeros(epochs, K_cross_val)
  val_accuracies = torch.zeros(epochs, K_cross_val)

  experiment_name = "lrcn_{}".format(datetime.now().strftime("%Y-%m-%d-%H:%M:%S"))
  log_dir = os.path.join("models", experiment_name)
  os.makedirs(log_dir)
  writer = SummaryWriter(os.path.join(log_dir, "runs"))
```

**train method** The train function is responsible for training a machine learning model using cross-validation and monitoring its performance over multiple epochs.

The first line initializes the *CrossEntropyLoss* function, which will be used to calculate the loss between the model's predictions and the target labels. CrossEntropyLoss is commonly used for multi-class classification problems.

The subsequent line generates cross-validation data loaders for the training and validation sets. It splits the train_dataset into K_cross_val folds and creates data loaders for each fold with the specified batch_size.

We then initialize tensors to store the validation losses and accuracies for each epoch and fold during training. The tensor has dimensions (epochs, K_cross_val).

We creat, at this point, a unique experiment name by combining the current date and time. It will be used for creating an output directory and saving the training logs.

The next line creates a directory path where the logs and model checkpoints will be saved. The models folder will contain all experiments, and we create the output directory if it doesn not exist.

We then initialize a SummaryWriter to write logs or summaries during the training process. It will be used for logging various metrics and visualizing them in TensorBoard. The following loop iterates over each fold (subset) and performs the following steps for each fold:

- Assigns data loaders, creates the model and optimizer, and trains the model for each epoch.
- Validates the model, applies early stopping based on the validation loss, and saves the best model and validation metrics.

In the loop, we create the model and get a list of trainable parameters. The build_model() function constructs the LRCN model with the specified feature dimension and hidden size.

The optimizer initializes the Adam optimizer with the specified learning rate (lr) and regularization term (lambda_reg). It will update the model's trainable parameters based on the calculated loss.

It is worth remembering that training and validation steps are performed for

```
for k in range(K_cross_val):
  train_loader, val_loader = train_folds[k], val_folds[k]

  model, trainable_parameters = build_model()

  optimizer = torch.optim.Adam(
                                trainable_parameters,
                                lr=config['lr'],
                                weight_decay=config['lambda_reg'])

  # early stopping and best model saving
  early_stopping_counter = early_stopping_patience
  min_val_loss = 1e10

  # training and validation
  for e in range(epochs):
    print("FOLD {} - EPOCH {}".format(k, e))
    val_loss, val_accuracy = one_epoch(model, lossFunction, optimizer, train_loader, val_loader, writer, e, k)

    # store the validation metrics
    val_losses[e, k] = val_loss
    val_accuracies[e, k] = val_accuracy
    torch.save(val_losses, os.path.join(log_dir, 'val_losses.pth'))
    torch.save(val_accuracies, os.path.join(log_dir,'val_accuracies.pth'))

    if e % 50 == 0:
      torch.save(model.state_dict(), os.path.join(log_dir,'fold_{}_{}_model.pth'.format(k,e)))
      print("- saved checkpoint model with val_loss =", val_loss, "and val_accuracy =", val_accuracy)

    # save the best model and check the early stopping criteria
    if val_loss < min_val_loss: # save the best model
      min_val_loss = val_loss
      early_stopping_counter = early_stopping_patience # reset early stopping counter
      torch.save(model.state_dict(), os.path.join(log_dir,'fold_{}_best_model.pth'.format(k)))
      print("- saved best model with val_loss =", val_loss, "and val_accuracy =", val_accuracy)

    if e>0: # early stopping counter update
      if val_losses[e, k] > val_losses[e-1, k]:
          early_stopping_counter -= 1 # update early stopping counter
      else:
          early_stopping_counter = early_stopping_patience # reset early stopping counter
    if early_stopping_counter == 0: # early stopping
        break
```

each epoch, iterating over the training data using the one_epoch function, and that the validation metrics (loss and accuracy) for each epoch and fold are stored in the val_losses and val_accuracies tensors. The model's state and validation metrics are saved periodically based on the epoch number.

Early stopping criteria are checked by comparing the current validation loss with the previous validation loss. If there is no improvement in validation loss for early_stopping_patience consecutive epochs, the training process stops for the current fold.

Finally, the function returns the val_losses and val_accuracies tensors, which contain the recorded validation metrics for each epoch and fold during training.

```
def build_dataset(path):
    preprocessing = albumentations.Sequential([
        albumentations.Resize(height=256, width=256, always_apply=True),
        albumentations.CenterCrop(height=224, width=224),
        albumentations.Normalize(mean=[0.485, 0.456, 0.406],
                                 std=[0.229, 0.224, 0.225],
                                 max_pixel_value=255.,
                                 always_apply=True),
    ])

    augmentation = albumentations.OneOf([
            albumentations.HorizontalFlip(p=1.),
            albumentations.RandomRain(p=.1),
            albumentations.RandomSnow(p=.1),
            albumentations.RandomFog(p=.1),
            ], p=.3)

    return VideoFrameDataset(path,
                             imagefile_template='Frame{:05d}.jpg',
                             num_segments=1,
                             frames_per_segment=5,
                             transform=albumentations.Compose([
                                 preprocessing,
                                 augmentation]))
```

To build the dataset, we can finally apply the pre-processing and augmentation tools imported through the albumentations library.

We first create an instance of sequential transformations using the Albumentations library. Sequential transformations mean that the transformations listed inside will be applied in sequence to each frame. The transformations specified here are resizing the image to 256x256, center cropping it to 224x224, and normalizing the pixel values to have a mean of [0.485, 0.456, 0.406] and standard deviation of [0.229, 0.224, 0.225]. These are common preprocessing steps used in many computer vision tasks.

The, we create an instance of "OneOf" transformations using the Albumentations library. "OneOf" means that only one of the transformations listed inside will be randomly applied to each frame. The transformations specified here are horizontal flip, random rain, random snow, and random fog. The probability p of applying these augmentations is set to 0.3.

Finally we returns an object of the VideoFrameDataset class with the specified parameters:

- path: The path where the video frames are located.
- imagefile_template: The template to construct the name of the frames, where : $05d$ will be replaced with the frame number. For example, "Frame00001.jpg".
- num_segments: themselveshe number of "blocks" of 5 frames for each video. In this case, it is set to 1, indicating that only one segment of 5 frames will be used for each video.
- frames_per_segment: the number of frames in each segment. In this case, it is set to five, meaning each segment contains five frames.

- transform: a single transformation pipeline created by composing the preprocessing and augmentation transformations defined earlier. This pipeline will be applied to each frame in the dataset, first the preprocessing and then one of the augmentations.

```
path = './FRAMES/VIDEOS/TRAINING_SET'
dataset = build_dataset(path)


lr = 1e-4
config = {
    'lr': lr  ,
    'lambda_reg': 0
}

train(dataset, config, batch_size=32, epochs=300)
```

**launching training** The first line sets the path to the directory where the video frames for the training set are located.

After that, we create the training dataset using the build_dataset function defined above. The function applies preprocessing and data augmentation to the video frames and constructs a VideoFrameDataset object.

The hyperparameters are:

- lr = 1e-4: The learning rate, which determines the step size for updating model weights during training.
- config: A dictionary that contains configuration parameters for training. It includes hyperparameters such as the learning rate and the regularization term.

We then set the configuration dictionary with the specified learning rate and regularization term. In this case, the regularization term is set to zero to disable regularization during training.

In the end, we start the training:

- dataset: The training dataset, which was created using the build_dataset function earlier.
- config: The configuration dictionary containing hyperparameters for training.
- batch_size=32: The batch size, which determines the number of samples processed in each iteration during training. In this case, it is set to 32.
- The number of epochs, which indicates the number of complete passes through the training data. In this case, it is set to 300.

Last thing, we start tensorBoard.

## 4.2 test.py

Here we discuss the test.py document.

### 4.2.1 init_parameter()

```
def init_parameter():
    parser = argparse.ArgumentParser(description='Test')
    parser.add_argument("--videos", type=str, default='foo_videos/', help="Dataset folder")
    parser.add_argument("--results", type=str, default='foo_results/', help="Results folder")
    args = parser.parse_args()
    return args
```

We define a function named that initializes a command-line argument parser for testing purposes. The argument parser allows users to provide values for specific command-line arguments when running the script.

First, we create an ArgumentParser object named parser with the description set to 'Test'. The ArgumentParser class allows you to define command-line arguments and their corresponding types, default values, and help messages.

We then add a command-line argument named –videos. It specifies that the argument should be of type string (type=str). If the user does not provide a value for this argument, the default value 'foo_videos/' will be used. Additionally, the help parameter provides a description of the argument, which will be displayed when the user asks for help with the script.

Following up, we add another command-line argument named –results. It follows the same pattern as the previous argument but is used to specify a folder path for the results. The default value is 'foo_results/'.

At that point, we include a line that parses the command-line arguments provided when running the script. It takes the arguments from the command-line and stores them in the args variable as a Namespace object, which is similar to a dictionary. The parsed values can then be accessed using dot notation, such as args.videos or args.results.

Finally, the function returns the parsed command-line arguments as the args Namespace object.

With this function, when werun the script from the command line, we can pass values for the –videos and –results arguments.

### 4.2.2 get_preprocess()

```
NUM_FRAMES = 5

def build_model():
    model = LRCN(feature_dim=256, hidden_size=256)
    model.load_state_dict(torch.load('model.pth', map_location=torch.device('cpu')))
    model = model.eval()
    return model


args = init_parameter()

model = build_model()
preprocess = get_preprocess()
to_tensor = ImglistOrdictToTensor()

true_pos = 0
```

The get_preprocess() function returns an image preprocessing transform using the Albumentations library for image augmentation. The transformation includes resizing the image to 256x256 pixels, center cropping it to 224x224 pixels, and normalizing the pixel values. Additionally, the function creates five additional targets for image augmentation. The transformation is applied with a probability of 1 to all images. This preprocessing transform can be used to augment images for training or any other task that requires image preprocessing.

#### 4.2.3 build_model() & print results

Here we processe a collection of test videos to detect a certain event (e.g., fire). We do so using a pre-trained model to classify image frames.

- build_model(): Builds a pre-trained model and loads its weights from a file called 'model.pth'. The model is set to evaluation mode (model.eval()), and the function returns the loaded model.
- args = init_parameter(): Initializes command-line argument parser for testing and retrieves the parsed arguments, such as the input and output folders for videos and results.
- model = build_model(): Loads the pre-trained model using the 'build_model()' function.
- preprocess = get_preprocess(): Gets the image preprocessing transform using Albumentations library.
- to_tensor = ImglistOrdictToTensor(): Creates a transform to convert a list or dictionary of numpy images to a torch.FloatTensor.
- We then process each test video file in the specified folder (args.videos). We read the frames, apply preprocessing, and convert them to a torch tensor.
- The preprocessed frames are then passed through the pre-trained model, which predicts the event (e.g., fire) for each sequence of frames.
- If the model predicts a certain event (output = 1) in a video sequence, it writes the frame number (i) in the video where the event is detected to a text file with the same name as the video, inside the specified results folder (args.results).

# 5 Results and discussion

In the following chapter, we will discuss how the training phase was handled.

## 5.1 Training

The first step involved the utilization of cross-validation to partition the data provided to the model due to the absence of a sufficiently large dataset. Consequently, the dataset was divided into 10 folds, with each training session using one fold for validation data and the remaining folds for training. The division of the dataset into folds and the use of separate data loaders for each fold allowed the evaluation of model performance on independent validation data in each iteration of cross-validation. This facilitated obtaining a more accurate estimation of the model's performance and addressed issues such as overfitting.

The training of the network comprised an initial phase of division into folds, as mentioned earlier, followed by the training of each individual fold. During the training of each fold, the "one_epoch" function was employed to execute a single epoch of training and model evaluation. Within each training epoch, the Loss and accuracy were computed for each batch of data, and the model parameters were updated through an optimizer. During the validation phase, model performance was evaluated on the validation dataset, and both the loss and accuracy were calculated.

To mitigate overfitting and select the best model, the technique of "early stopping" was applied. This approach allowed us to monitor the validation loss during training and halt the training process if the validation loss failed to exhibit significant improvement for a predetermined number of consecutive epochs. Additionally, model checkpoints were saved during training to enable resumption from a previous point if necessary. Finally, we recorded the loss and accuracy metrics during training using a "writer" object, facilitating a more in-depth visualization and analysis of the model's performance on each fold and epoch.

Several decisions were made during this phase, including the use of the CrossEntropy loss function for multi-class classification problems, the Adam optimizer with specified parameters in the configuration, and the adoption of early stopping to prevent overfitting. Furthermore, model checkpoints were saved during training, and loss and accuracy values were tracked for each epoch and fold to facilitate subsequent result analysis.

These choices were made to organize and monitor model training, ensuring the recording of results and the ability to resume training from a previous point in case of interruptions or analysis requirements. The implementation of early stopping served to avoid prolonged training when significant performance improvement did not occur.

Regarding the number of epochs, we utilized 300, a sufficiently large value to allow the network to converge, but not excessively low. The number of

epochs, however, was influenced by the early stopping mechanism, which not only addressed overfitting issues but also expedited training by terminating when convergence was achieved.

A learning rate of 0.00001 was employed, representing a comparatively low value, meaning that the weight updates during model training were very small. This approach proves useful when closely controlling the model's learning rate and preventing excessively large updates that may cause oscillations or divergence during training. The use of a very small learning rate can assist in reducing the risk of overfitting during model training. A learning rate that is too large could lead to overfitting on the training data, while a very small learning rate can help regularize the model and mitigate overfitting.

The selected values for the learning rate and the number of epochs were influenced by experiments conducted on the network and appear to yield the best results for this type of network.

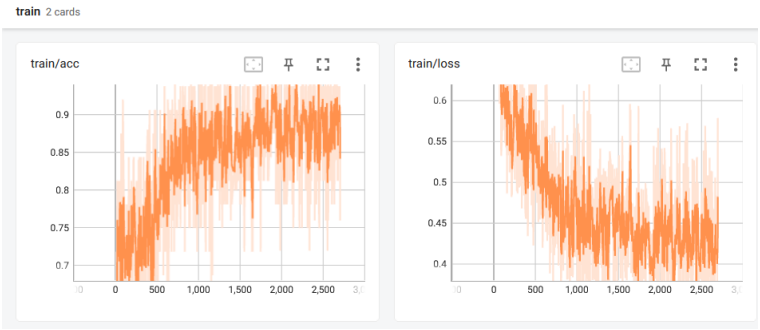
## 5.2 Result analysis

During the training phase, we observed that the values of the loss were quite similar across the various folds. Nevertheless, as the network evolved, excessive fluctuations in the loss influenced our decision-making process, leading us to favor folds that, despite not having lower loss values, exhibited smoother oscillations.

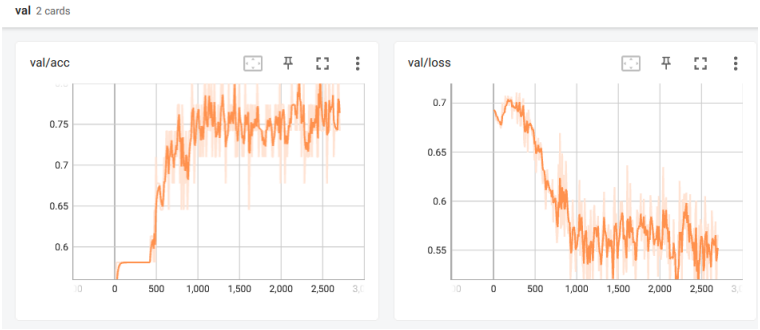
Another factor taken into consideration was the accuracy values; for the final model selection, we opted for the one that demonstrated high accuracy values while avoiding excessive fluctuations.

In addition to the experimental data, we also analyzed empirical data using a specifically designed test set for the model. Each fold was subjected to evaluation on this test set, and it was noted that suboptimal values of loss and accuracy resulted in mediocre performance.

Consequently, the chosen fold for the final model is fold n9, which exhibits favorable values of loss and accuracy:




Also the values on the test set:




Positives: 15 / 15

False positives: 8 / 28

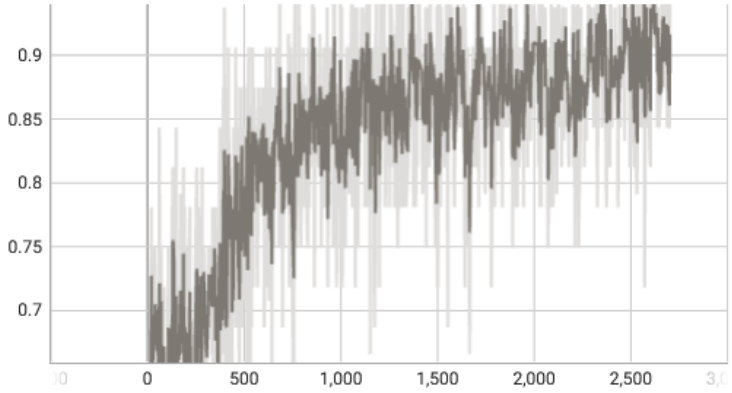
Ultimately, despite having a slightly higher loss value compared to other folds, this one was preferred due to its superior performance on the test set.

## 5.3 Comparison

In the following exposition, we will refrain from commenting on the results of each fold to achieve a more effective presentation; suffice it to highlight that a generic fold among those discarded exhibited different behavior compared to the carefully selected one.

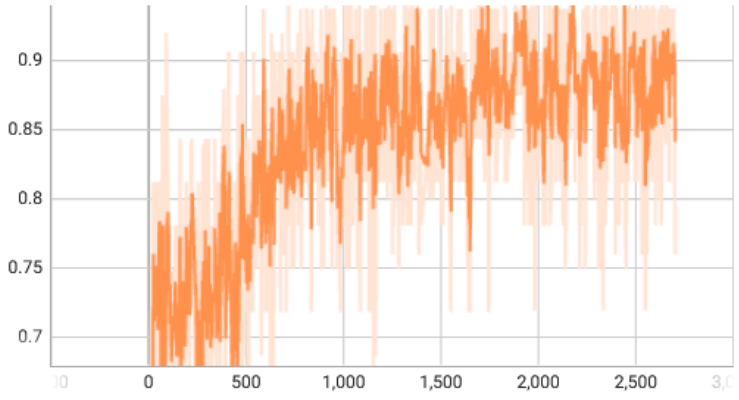
### 5.3.1 Train

**Accuracy** For the first fold taken as an example (the accuracy graph is shown here):

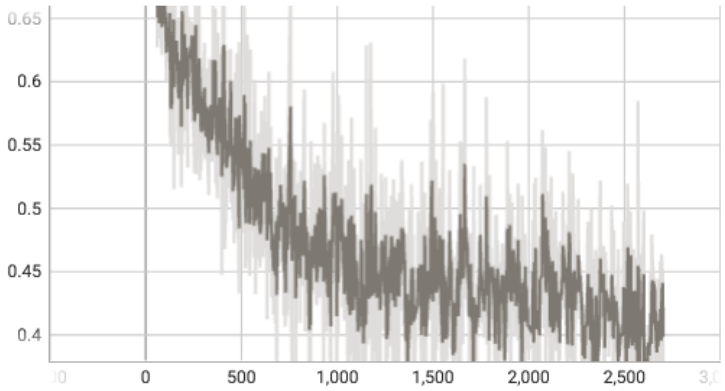



It can be observed that it converges towards a value of approximately 0.9, with fluctuations during its evolution.

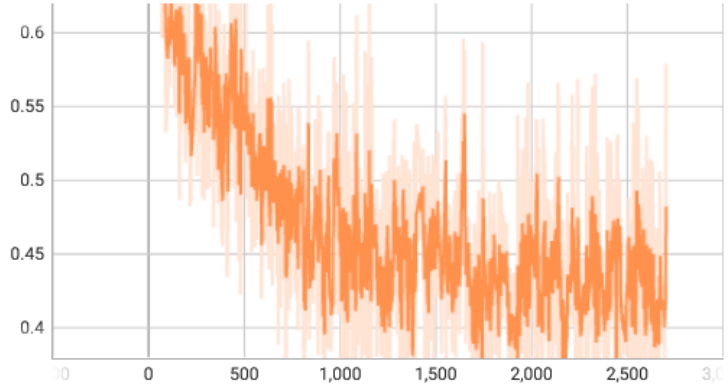
A very similar situation is also evident in the selected fold:




**Loss** For the first fold taken as an example (the loss graph is shown here):



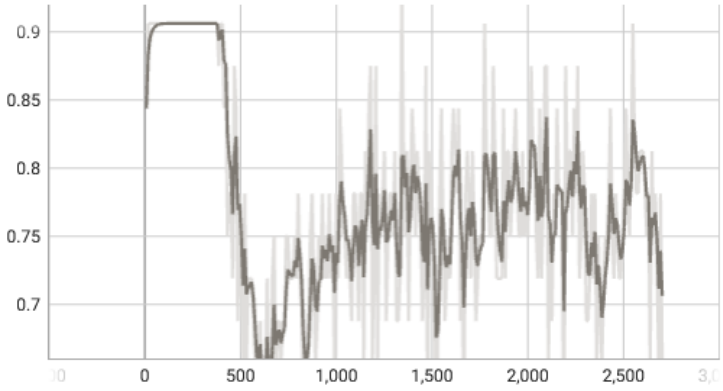

The selected fold:



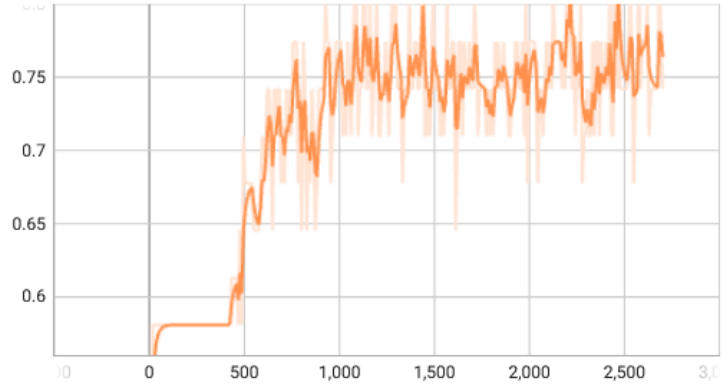

Once again, they are quite similar to each other; furthermore, both exhibit oscillations that stabilize around the value of 0.4.

### 5.3.2 Validation

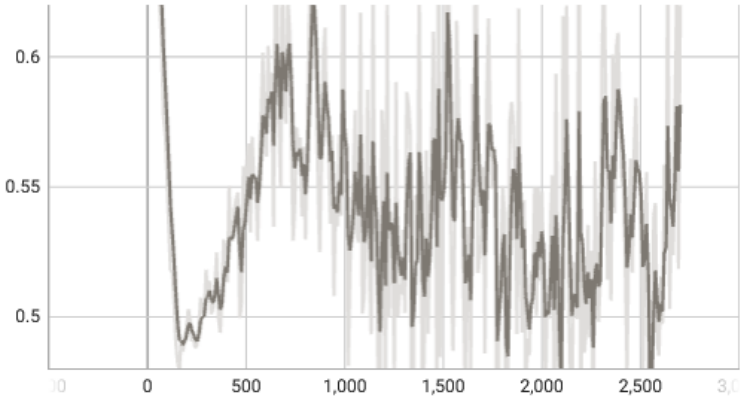
**Accuracy** Once again, the graphs are very similar to each other. Below is the accuracy graph pertaining to the fold taken as an example:



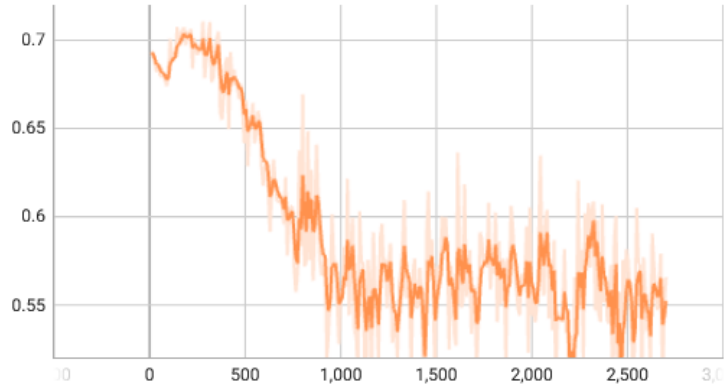

And the fold we have selected:



**Loss** Here there's one of the reasons that led us to favor the selected fold. Below is the loss graph pertaining to the fold taken as an example:




And the fold we have selected:



### 5.3.3 Test

As previously mentioned, in addition to the data illustrated earlier, we also utilized empirical test data by creating a custom test set.

The difference with these latter results is evident; just observe how the fold taken as an example yielded the following outcomes:

Positives: 14 / 15

False positives: 18 / 28

In contrast, the selected fold yielded the following results:

Positives: 15 / 15

False positives: 8 / 28

# 6 Final considerations

The concluding reflections of this fire recognition project through machine learning have been extremely promising and represent a significant step forward in the application of advanced techniques for fire prevention and monitoring. By employing deep learning algorithms and training a model on an extensive dataset of videos, we achieved satisfactory results in fire classification and detection with a high degree of precision and timeliness. However, the project has also highlighted some important challenges. The availability of labeled data proved to be an obstacle, and collecting representative videos of real fires in various environmental conditions was complex. Additionally, the model exhibited sensitivity to false positives in particularly complex and rare situations. Therefore, to enhance the reliability and robustness of the system, it is suggested to expand the dataset and consider implementing a continuous feedback mechanism to further refine the model based on practical experience.

Despite these challenges, the potential of machine learning in fire recognition is undeniable and offers significant opportunities to improve safety and environmental protection. With further efforts and future developments, we are confident that this system can become a valuable tool for fire prevention and control, benefiting communities and the environment.