UNIVERSITÀ
POLITECNICA
DELLE MARCHE

Facoltà di Ingegneria
Corso di Laurea in Ingegneria Informatica e dell'automazione

Valutazione comparata di reti neurali profonde per il rilevamento automatico di scene di violenza all'interno di video

Comparative evaluation of deep neural networks for automatic detection of violence scenes within videos

Candidato:
Matteo Giri

Relatore:
Prof. Aldo Franco Dragoni

Correlatore:
Dott. Paolo Sernani

Anno Accademico 2020-2021

A Sofia, alla mia famiglia ed ai miei amici.

Ringraziamenti

Grazie a Sofia per essere sempre stata al mio fianco e per avermi sempre ascoltato e incoraggiato.

Grazie alla mia famiglia per avermi accompagnato durante il mio percorso universitario, supportandomi e sostenendo tutte le mie scelte.

Grazie al professor Aldo Franco Dragoni per la sua disponibilità e per avermi dato la possibilità di lavorare ad un argomento così affascinante.

Grazie al professor Paolo Sernani per essere sempre stato disponibile e per avermi guidato ed aiutato durante tutto il percorso di tirocinio e tesi.

Grazie ai miei amici e compagni di corso per esserci sempre stati nel momento del bisogno.

Ancona, Luglio 2021

Matteo Giri

Abstract

Al giorno d'oggi le occorrenze di attività violente in luoghi pubblici sono molteplici e in continuo aumento. La maggior parte dei sistemi di videosorveglianza ad oggi utilizzati, tuttavia, non è in grado di riconoscere autonomamente e prevenire queste attività. Il campo dell'activity recognition, di recente molto studiato e in continuo sviluppo, può offrirci uno strumento molto valido per il riconoscimento automatico di scene di violenza e quindi permetterne la prevenzione. In questo lavoro vengono testate diverse soluzioni basate sull'addestramento di reti neurali, al fine di documentare le performance e dimostrare l'efficacia di ciascuna nel differenziare scene violente da scene non violente. Si utilizzano inoltre strumenti per rappresentare in maniera grafica il comportamento di tali soluzioni, per giustificare e confermare la validità dei risultati ottenuti.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

Con il termine "***violenza***" si indica la caratteristica di essere violento, soprattutto come tendenza abituale a usare la forza fisica in modo brutale o irrazionale, facendo anche ricorso a mezzi di offesa, al fine di imporre la propria volontà e di costringere alla sottomissione, coartando la volontà altrui sia di azione sia di pensiero e di espressione, o anche soltanto come modo incontrollato di sfogare i propri moti istintivi e passionali. Questa è la definizione data alla violenza dal vocabolario della lingua italiana Treccani [1], ed è soltanto una delle molteplici descrizioni che nel corso degli anni si è provato a dare a questo termine. Non esiste infatti una definizione unica e globalmente accettata di questa parola, in quanto essa stessa assume diversi significati in base al contesto in cui viene posta. La World Health Organization[1], nel suo rapporto "*World report on violence and health*" [2], descrive la violenza come l'uso intenzionale della forza fisica, sotto forma di minaccia o come atto fisico, contro sé stessi, contro un'altra persona, o contro un gruppo o comunità, che risulta o che ha un'alta possibilità di risultare in ferite, morte, danno psicologico, o privazione.
É quindi chiaro che la violenza si manifesta in diverse forme e tipologie. La stessa World Health Organization scompone il termine in 3 diverse macrotipologie: violenza autodiretta, violenza interpersonale e violenza collettiva [3]. Quella che meglio descrive l'argomento generale trattato in questo elaborato è la violenza interpersonale, che è divisa in due sottocategorie: violenza familiare e da partner intimo, ovvero violenza in gran parte tra membri della famiglia e partner intimi, che di solito, anche se non esclusivamente, si svolge in casa; violenza di comunità, ovvero violenza tra individui che non sono imparentati e che possono o meno conoscersi, generalmente avvenuta fuori casa. Il primo gruppo comprende forme di violenza come abusi sui minori, violenza da parte del partner e abusi sugli anziani. Il secondo include violenza giovanile, atti di violenza casuali, stupro o aggressione sessuale da parte di estranei e violenza in contesti istituzionali come scuole, luoghi di lavoro, carceri e case di cura.

La violenza nelle sue varie forme può essere prevenuta, tuttavia è innegabile che scene di violenza si verifichino costantemente e continuano a verificarsi. In questo

[1] L'Organizzazione mondiale della sanità (OMS; in inglese World Health Organization, WHO) è un istituto specializzato dell'ONU per la salute, fondata il 22 luglio 1946 ed entrata in vigore il 7 aprile 1948 con sede in Svizzera, Ginevra.

contesto, è utile avere uno strumento efficace che ci permetta di riconoscere atti di violenza laddove la prevenzione fallisce.

La disciplina della *Violence Detection* si pone come obbiettivo quello di individuare automaticamente aggressioni di diversi tipi tra due o più persone. Lo scopo di questo lavoro è testare tecniche per il riconoscimento automatico di scene di violenza all'interno di video. In particolare, ci concentreremo su quelle tecniche basate sul *Deep Learning*, dato che stanno emergendo come più "promettenti". Le architetture testate in questo lavoro consistono quindi di diversi tipi di reti neurali profonde, principalmente basate su reti neurali convoluzionali 3D (3D CNN), architetture ConvLSTM e Bi-ConvLSTM.

1.1 Motivazioni

La necessità di creare sistemi che riescano a identificare ed etichettare diversi tipi di azioni svolte da esseri umani è cresciuta esponenzialmente negli ultimi tempi, in quanto questi sistemi possono fornire una soluzione adeguata al problema dell'identificazione di comportamenti inappropriati.

Con lo sviluppo della tecnologia, il metodo che più degli altri si è affermato con lo scopo di riconoscere scene di violenza è indubbiamente quello dell'utilizzo di sistemi di videosorveglianza. Questi sistemi, seppur molto efficaci, sono soggetti all'errore umano. Infatti, i filmati provenienti dalle telecamere di sorveglianza vengono esaminati manualmente da operatori umani, e gli esseri umani possono commettere errori e magari non accorgersi di un evento significativo. Nel mondo ci sono milioni di videocamere per la sorveglianza, e anche se le percentuali di errore sono basse c'è sempre pericolo per migliaia di persone. É per questo che la comunità di ricercatori è in continuo lavoro per trovare ed approfondire metodi di rilevamento di violenza automatici, che non abbiano bisogno dell'aiuto di un essere umano.

La *violence detection* è un campo in continuo sviluppo e stanno emergendo molti lavori promettenti a riguardo che utilizzano diversi sistemi e tecnologie. le tecniche che utilizzano il *Machine learning* (l'apprendimento automatico), in particolare, hanno dimostrato risultati eccellenti nella classificazione di immagini e video e come tali sono stati riadattati per il riconoscimento di azioni umane. Questo lavoro nasce dalla volontà di testare architetture sviluppate secondo quest'ultima tecnica, in particolare sistemi basati sull'addestramento di reti neurali profonde, con lo scopo di dimostrarne l'efficacia e di mettere a confronto diversi modelli proposti negli ultimi anni. Il progetto ha anche il secondo fine di analizzare il comportamento di queste architetture, attraverso l'utilizzo delle *Class Activation Maps* (CAM), per capire come le reti fanno uso delle informazioni che ricevono.

1.2 Obiettivi

Gli obiettivi alla base di questo lavoro di tesi sono i seguenti:

- Testare l'accuratezza di tre diverse architetture basate sull'addestramento di reti neurali profonde sul dataset "Airtlab Dataset", sviluppato dall'AIRTLab[2], confrontando per ognuna di esse i diversi tipi di ottimizzatori messi a disposizione dalla Keras API.
- Testare l'accuratezza delle tre architetture sui dataset "Hockey fights" e "Crowd Violence", utilizzando gli ottimizzatori che hanno ottenuto i risultati migliori sul dataset "Airtlab Dataset".
- Implementare, con Keras, un'architettura basata su reti convoluzionali bidirezionali in una forma completa e una semplificata, e testare l'accuratezza di entrambe sui tre diversi dataset sopra citati.
- Utilizzare le Class Activation Maps per generare delle heatmaps sui video dei dataset utilizzati, in modo tale da costruire una visualizzazione grafica delle decisioni prese da un particolare modello di rete neurale nel riconoscere scene di violenza, e verificarne quindi la correttezza implementativa.

1.3 Struttura della tesi

Il presente elaborato è organizzato nel modo seguente:

- Nel prossimo capitolo verranno introdotti i lavori sulla *violence detection* presenti nello stato dell'arte.
- Nel terzo capitolo verranno descritti gli aspetti fondamentali di una rete neurale convoluzionale, base di tutte le architetture testate in questo elaborato.
- Nel quarto capitolo verranno introdotti tutti gli strumenti utilizzati e verranno descritti i test effettuati sui modelli e dataset presi in considerazione.
- Nel quinto capitolo si parlerà dell'implementazione di un modello basato su rete neurale convoluzionale bidirezionale LSTM e dei test per verificarne le performance nei vari dataset proposti.
- Nel sesto capitolo si introdurrà il concetto delle *Class Activation Maps* e verranno utilizzate per capire meglio su quali caratteristiche di un video si focalizzano le reti neurali.
- Nel capitolo conclusivo si analizzeranno i risultati ottenuti complessivamente e si proporranno alcuni sviluppi futuri.

[2] Artificial Intelligence and Real Time Systems Laboratory, Dipartimento di Ingegneria dell'Informazione, Università Politecnica delle Marche

Capitolo 2

Lo Stato dell'Arte

In questo capitolo verrà approfondito l'argomento della *violence detection*, partendo da una breve introduzione sul tema e su come questa derivi dal più ampio campo della *human action recognition*. Si introdurranno infine le varie tecniche che nel corso dell'ultimo decennio si sono sviluppate con lo scopo di riconoscere violenza nei filmati.

2.1 Violence Detection e action recognition

Come anticipato nella sezione introduttiva, la *violence detection* è un problema che può essere inserito all'interno di un problema più grande che è quello della *action recognition*, e più in particolare della *human action recognition*.
Esistono molteplici tecniche di *action recognition*, quella di interesse per questo lavoro è l'utilizzo della scienza della computer vision[1] per tenere traccia e capire il comportamento di vari agenti tramite video ottenuti da videocamere.
Le attività svolte da esseri umani possono essere raggruppate in quattro categorie in base alle parti del corpo che sono impiegate nell'azione e nella sua complessità [4].

- Gesto: É un'azione corporea visibile che rappresenta un messaggio specifico. Non è una questione di comunicazione verbale o vocale, ma piuttosto un movimento fatto con le mani, il viso o altre parti del corpo come gesti di "okay" e pollici in su.
- Azione: É un insieme di movimenti fisici condotti da una singola persona come correre o camminare.
- Interazioni: É un insieme di azioni eseguite da al massimo due attori. Almeno un soggetto è una persona mentre l'altro può essere un umano o un oggetto (stretta di mani, conversare, ecc.).
- Attività di gruppo: è un misto di gesti, azioni, o interazioni. Il numero di attori è almeno di due più uno o più oggetti interattivi (giocare a basket, corse ad ostacoli, ecc.).

[1] La visione artificiale (nota anche come computer vision) è l'insieme dei processi che mirano a creare un modello approssimato del mondo reale partendo da immagini bidimensionali. Lo scopo principale della visione artificiale è quello di riprodurre la vista umana.

In questo contesto la *human action recognition* (HAR) [5] mira ad esaminare e riconoscere automaticamente la natura di varie azioni svolte da esseri umani. La ricerca sulla *vision-based HAR*[2] è la base di moltissime applicazioni come la videosorveglianza, l'assistenza sanitaria, e l'interazione umano-computer (HCI).

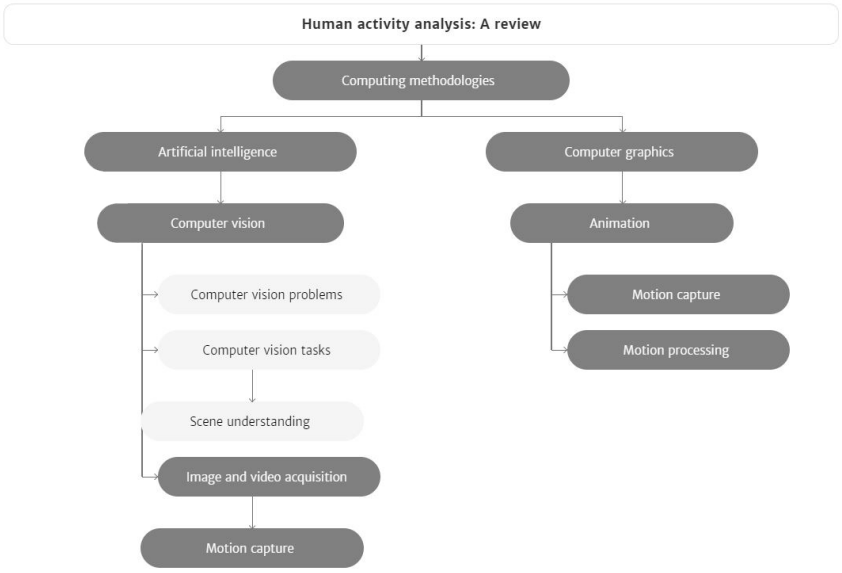

Figura 2.1: Tecniche per la human action recognition.

La *violence detection* si presenta come un caso particolare della *human action recognition*, poiché si concentra sull'analisi di flussi video in cerca di interazioni violente tra due o più persone. Per identificare un atto violento necessitiamo di tre informazioni fondamentali [6]:

- L'informazione spaziale: l'aggressione ha una particolare rappresentazione.
- L'informazione temporale: l'aggressione si sviluppa per un periodo temporale.
- L'informazione del movimento come l'accelerazione: l'aggressione è caratterizzata da brusche variazioni di velocità (i colpi).

L'insieme di queste prende il nome di *informazioni spazio-temporali*.

2.2 Tecniche per la Violence Detection

In questa sezione verranno presentate brevemente le diverse tecniche sviluppate negli ultimi anni per il riconoscimento di scene di violenza nei video.
Nel processo di riconoscimento di scene violente, il primo step è quello di dividere l'intero video che si vuole analizzare in segmenti e frames. Come seconda cosa, bisogna rilevare l'oggetto dai frames del video. Il terzo step è quello di estrarre le

[2]human action recognition basata sulla computer vision

features (caratteristiche) del video in accordo alla tecnica utilizzata. infine, si rileva l'attività anomala dai frames. L'implementazione degli step varia in base alla tecnica utilizzata per la classificazione. Questi passi sono riassunti nella figura 2.2.

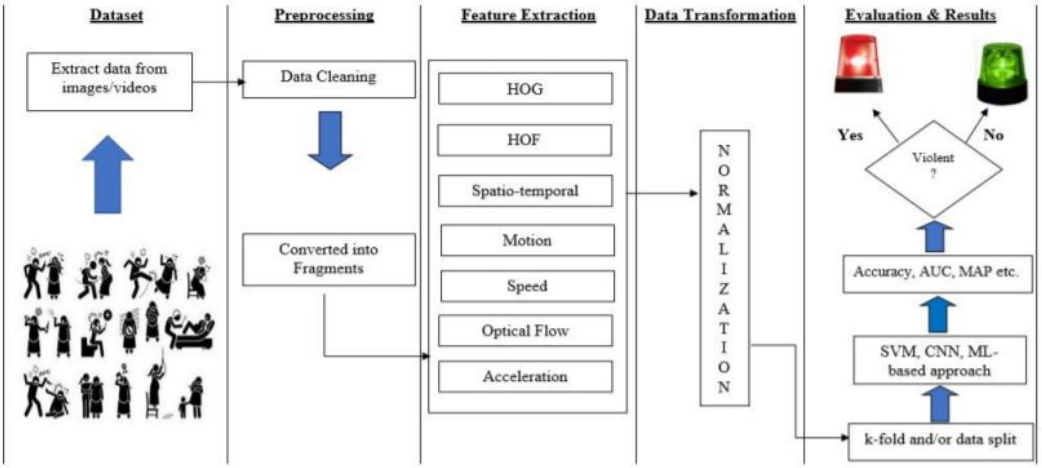

Figura 2.2: Passi principali di una tipica tecnica di violence detection.

I metodi utilizzati per la *violence detection* possono essere suddivisi in tre grandi categorie, basate sulle tecniche di classificazione utilizzate, come riportato nel report *"A Review on State-of-the-Art Violence Detection Techniques"*[7]:

- Violence detection utilizzando il *machine learning* tradizionale
- Violence detection utilizzando la *Support Vector Machine* (SVM)
- Violence detection utilizzando il *deep learning*

2.2.1 Feature Descriptors tradizionali

Prima di illustrare i vari approcci alla *violence detection* suddividendoli per tecnica di classificazione utilizzata, è utile elencare alcuni dei metodi tradizionali più utilizzati riguaranti gli step precedenti a quello della classificazione, in particolare quello dell'estrazione delle features.

Analizzeremo velocemente alcuni algoritmi utilizzati relativamente a questo step, chiamati *feature descriptors*. Un *feature descriptor* è un algoritmo che prende come input un immagine e ne estrae i vettori di features. L'idea è quella di codificare informazioni interessanti presenti in un'immagine all'interno di una serie di numeri che fungono da una sorta di impronta numerica che può essere usata per differenziare una caratteristica da un'altra.

Scale-invariant Feature Transform (SIFT)

Scale-invariant feature transform (o SIFT) è un algoritmo utilizzato in *computer vision* che permette di rilevare e descrivere caratteristiche, o features, locali in immagini[8].
L'idea alla base di questo metodo è l'estrazione di punti interessanti (keypoints) di un oggetto in un'immagine, per fornire una "feature description" dell'oggetto. Questa descrizione, estratta da un'immagine di addestramento (training image), può quindi essere utilizzata per identificare l'oggetto quando si tenta di individuare l'oggetto in un'immagine di prova (test image) contenente molti altri oggetti. Per eseguire un riconoscimento affidabile, è importante che le caratteristiche estratte dall'immagine di addestramento siano rilevabili anche in caso di cambiamenti nella scala dell'immagine, nel rumore e nell'illuminazione. Tali punti di solito si trovano su regioni dell'immagine ad alto contrasto, come i bordi degli oggetti.

Figura 2.3: Keypoints di un oggetto ottenuti tramite l'algoritmo SIFT.

SIFT viene utilizzato nell'ambito della *violence detection* soprattutto per rilevare movimenti. Il *Motion Scale-invariant feature transform* (o MoSIFT)[9] è un algoritmo basato sul SIFT creato con questo scopo. MoSIFT genera dei descrittori ottenuti da punti salienti in due parti: La prima parte è un istogramma aggregato di gradienti (HoG) che descrive l'aspetto spaziale. La seconda parte è un istogramma aggregato di flusso ottico (HoF) che indica il movimento del punto caratteristico. Quindi il MoSIFT è un descrittore che codifica sia il movimento che informazioni sull'aspetto.
Un altro algoritmo derivato dal SIFT è il *Lagrangian Scale Invariant Feature Transform* (LaSIFT)[10]. LaSIFT è un descrittore correlato al MoSIFT ma basato sui campi direzionali lagrangiani. SIFT e derivati vengono utilizzati come *feature descrip-*

tors in diverse ricerche sulla *violence detection* effettuate negli ultimi anni. Alcune di esse sono delineate nei lavori [10] [11] [12] [13].

Space-Time Interest Points (STIP)

Space-time interest points (o STIP)[14] sono punti in un frame di un video dove i valori dell'immagine hanno una variazione significativa sia nello spazio che nel tempo. L'idea alla base di STIP è utilizzare gli operatori dei punti di interesse di Harris e Förstner per rilevare i punti di interesse spazio-temporali [15] [16]. I punti di interesse rilevati sono caratterizzati da un'elevata variazione dell'intensità nello spazio e da un non costante movimento nel tempo. Questi punti salienti vengono rilevati su più scale spaziali e temporali. Dopo, HOG (Histograms of Oriented Gradients), HOF (Histograms of Optical flow) e una combinazione di HOG e HOF denominati "vettori di features HNF" vengono estratti nelle vicinanze degli STIP rilevati. Queste features possono essere utilizzate per riconoscere movimenti con performance molto elevate e sono inoltre robuste nello scalaggio e nelle variazioni di frequenza e velocità del pattern.

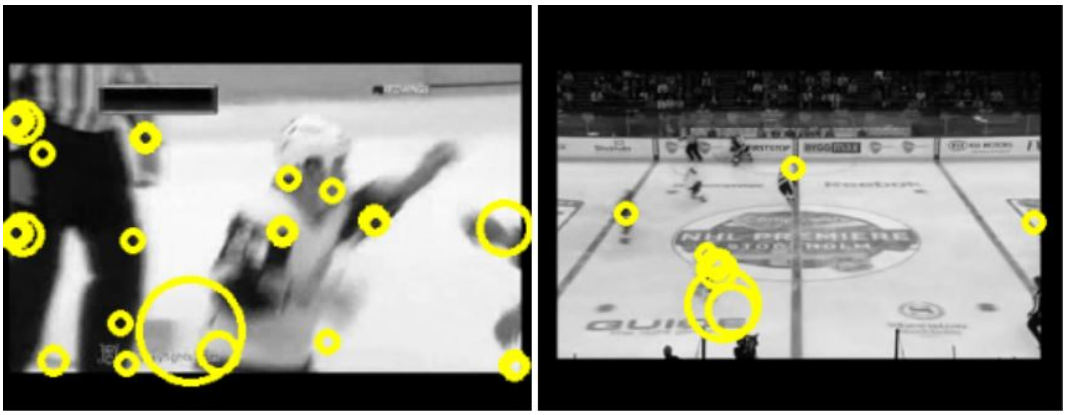

Figura 2.4: STIP in video di violenza e non.

Un'implementazione di STIP per la *violence detection* è consultabile in [12].

Violent Flows Descriptor (ViF)

Il *Violent Flows Descriptor* (o ViF) è un descrittore creato appositamente per il problema della *violence detection*[17]. L'idea di ViF è quella di considerare come le grandezze del vettore di flusso cambiano nel tempo.
Più in particolare, si parte dalla stima del flusso ottico tra coppie di frames consecutivi. Questo dà come risultato un vettore di flusso per ogni pixel dei frames considerati. Di questi vettori si considera solamente il modulo e di come questo varia nel tempo. Il modulo varia in base alla risoluzione del frame, ai diversi movimenti in diverse posizioni spazio-temporali, e ad altri parametri. Mettendo a confronto questi moduli si riescono ad ottenere misure importanti del significato del movimento osservato in ogni

frame. In pratica per ogni pixel di ogni frame si va ad ottenere un indicatore binario che riflette il significato del cambiamento di modulo tra i frames. Fare ciò porta alla creazione di una mappa binaria che descrive i cambiamenti di modulo. Nella sua forma più semplice, quindi, un descrittore ViF è un vettore di questi indicatori binari. Nella pratica però, regioni spaziali differenti hanno comportamenti caratteristici diversi. Quindi il descrittore ViF viene prodotto partizionando questi indicatori in celle che non si sovrappongono, in base alle regioni spaziali che compongono i frame. La distribuzione delle variazioni di modulo in ognuna di queste celle è rappresentata da un istogramma. Questi istogrammi vengono infine concatenati in un unico vettore che caratterizza il ViF descriptor.

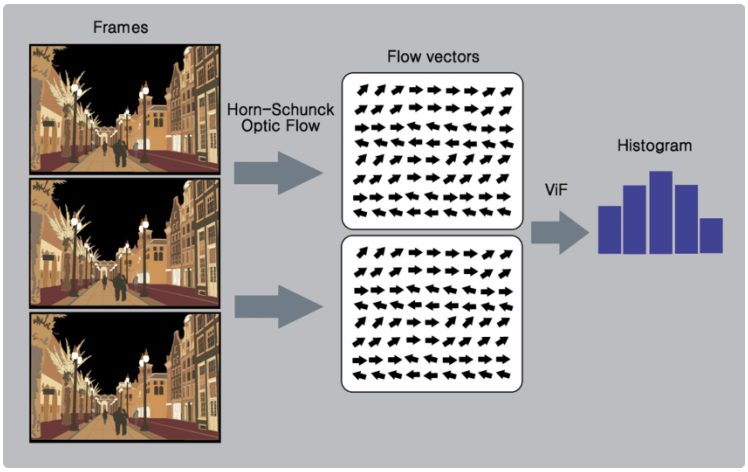

Figura 2.5: Creazione di un ViF decriptor.

Una variante di ViF è l' *Oriented Violent Flows Descriptor* (o OViF), che come suggerisce il nome differisce dal ViF per il fatto che le informazioni che vengono utilizzate coinvolgono sia il modulo del vettore di movimento sia il suo orientamento[18]. Il suo funzionamento è riassunto molto brevemente nella figura 2.6.

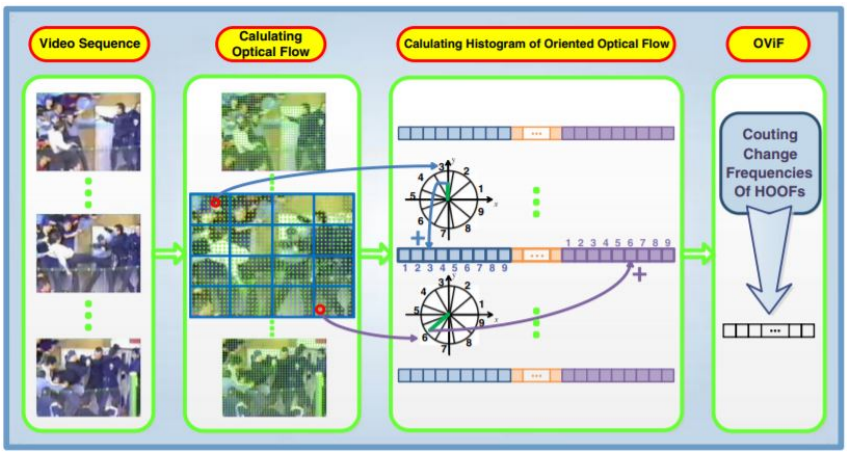

Figura 2.6: Funzionamento di OViF.

Alcuni lavori che utilizzano ViF e varianti come descrittore sono consultabili in [18] [19] [20] [21].

2.2.2 Violence Detection con Machine Learning tradizionale

In questa sezione verranno elencati alcuni esempi di approcci alla *Violence Detection* che rientrano nella categoria del *Machine Learning* più tradizionale.

Fast Fight Detection

Fast Fight Detection è un lavoro proposto in [22] in cui si assume che nelle scene di violenza, i motion blob abbiano una specifica forma e posizione, che permette quindi di caratterizzarli. Un "blob" (che in italiano potremmo tradurre con "agglomerato" o "macchia") è una regione di un'immagine in cui alcune proprietà sono costanti o approssimatamente costanti. Tutti i punti in un blob possono essere considerati in un certo senso simili tra di loro.
La tecnica utilizzata parte dal calcolo della differenza tra frames consecutivi di un determinato video. Successivamente l'immagine risultante viene binarizzata, portando alla creazione dei diversi motion blobs. Solamente i k motion blob più grandi vengono selezionati per i processamenti successivi. Per categorizzare i vari blobs, vengono calcolati diversi parametri come il centroide, l'area, il perimetro e anche la distanza tra di essi. Infine, i blobs vengono caratterizzati come violenti o non.

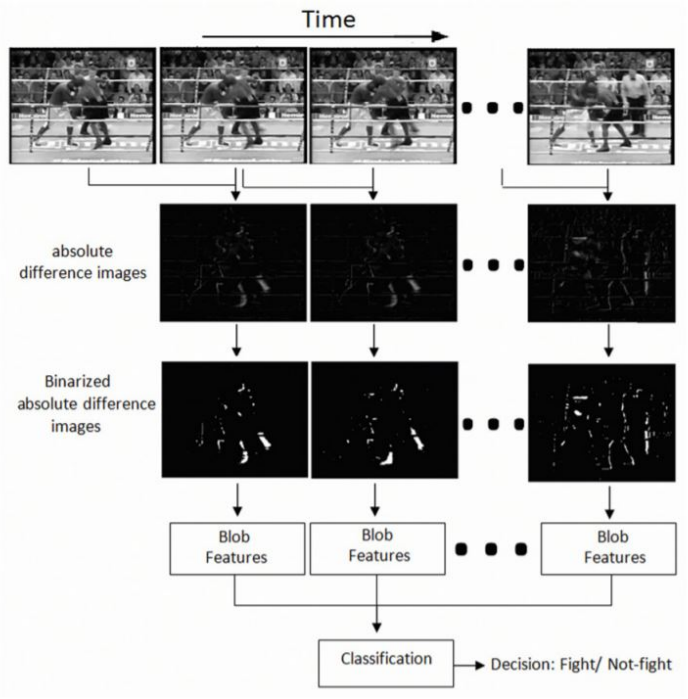

Figura 2.7: Diagramma dei passi utilizzati in *Fast Fight Detection*.

Il metodo proposto risulta meno preciso di altri presenti in letteratura ma computazionalmente molto più veloce, caratteristica che ne rende l'utilizzo molto conveniente nelle applicazioni real-time.

Rotation-Invariant Feature Modeling Motion Coherence

Rotation-Invariant Feature Modeling Motion Coherence (o RIMOC) è un metodo per la *violence detection* introdotto in [23] in cui si cerca di distinguere le scene violente da quelle non violente codificando movimenti strutturati e discriminandoli da movimenti irregolari e/o scatti, che si assume siano presenti in scene violente.
Il metodo si basa sugli autovalori ottenuti dalle statistiche di secondo ordine degli istogrammi dei vettori di flusso ottico di istanti temporali consecutivi, calcolati localmente e densamente, e successivamente incorporati in una varietà riemanniana sferica[3]. In particolare la tecnica descritta propone un nuovo metodo per modellare la struttura dei movimenti per mezzo di un nuovo spazio di features di basso livello basato sugli autovalori degli istogrammi di flusso ottico (HOF). Il processo aggiunge l'invarianza alle rotazioni nello spazio dell'immagine ed è discriminante al massimo tra movimenti di interesse e l'ambiente. Lo spazio di features viene successivamente incorporato in una metodologia statistica di rilevamento di anomalie, come mostrato nella figura 2.8.

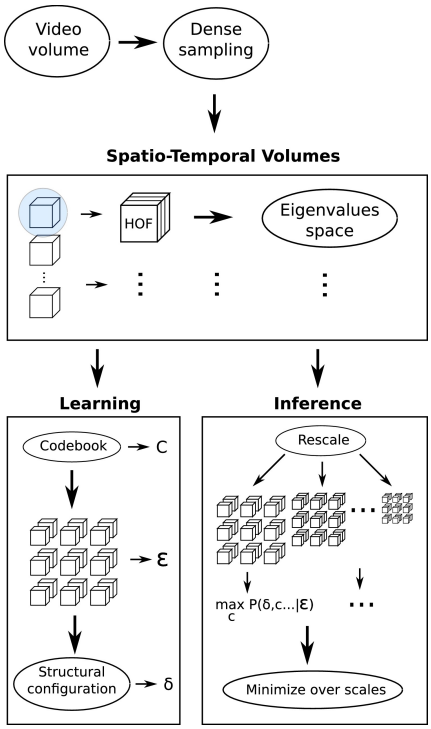

Figura 2.8: panoramica schematica del RIMOC.

[3]Una varietà riemanniana è una varietà differenziabile su cui sono definite le nozioni di distanza, lunghezza, geodetica, area (o volume), curvatura.

Nell'elaborato partono facendo un dense sampling del video in input e costruendo gli Spatio-Temporal Volumes (STV)[24] attorno a ogni punto campionato. Ogni frame degli STV è codificato come un HOF, di cui vengono calcolati gli autovalori. Durante il training si apprende un codebook[4] di dati che comprendono situazioni normali, e si utilizza il notebook per apprendere successivamente le configurazioni spazio-temporali delle parole. Le configurazioni spazio-temporali sono apprese come distribuzioni probabilistiche sulle posizioni relative, delle parole su regioni più larghe.

Durante l'inference si valuta se la query è stata generata utilizzando i modelli stimati, massimizzando la probabilità sulle possibili configurazioni su scale diverse. Il processo si basa sull'assunzione chiave che le istanze di dati normali occorrono in regioni ad alta probabilità dei modelli stocastici appresi, mentre le anomalie occorrono in regioni a bassa probabilità. La formulazione dell'inferenza è basata sul metodo proposto in [25], con l'aggiunta di uno step di fusione multiscala prima di impostare la soglia del valore di somiglianza. Infatti, la probabilità che la configurazione del movimento osservata sia espressa dal modello è ridotta al minimo sulle scale per rilevare anomalie a qualsiasi scala.

Multiple Anomalous Activity Detection

"*Multiple Anomalous Activity Detection in Videos*" è un lavoro proposto in [26] in cui il framework costruito è capace di riconoscere attività multiple in un singolo video e anche di eseguire una "behaviour understanding" (comprensione del comportamento). Il framework si compone di tre passi principali: rilevamento di oggetti in movimento, tracciamento di oggetti e comprensione del comportamento per il riconoscimento dell'attività.

Nella prima fase di preprocessing, viene effettuata la segmentazione del video in input, viene costruito un modello del background, vengono rilevati gli oggetti in movimento e viene rimosso il rumore. Per riconoscere lo sfondo in un video viene utilizzato il *Gaussian Mixture Model* (GMM) [27]. Viene poi eseguita una fase di feature extraction per identificare le caratteristiche chiave grazie al tracciamento di oggetti nei frames del video. Per il riconoscimento di attività normali o anomale si considerano parametri come misurazioni, velocità, direzione, movimento dell'oggetto e centroide. Nell'ultima fase, il metodo della classificazione rule-based viene utilizzato per categorizzare le azioni presenti nel video e se alcune attività sospettose vengono rilevate viene generato un allarme per evidenziarne la presenza, colorandone il rettangolo che le contiene di rosso.

[4] Un codebook è un tipo di documento utilizzato per raccogliere e immagazzinare codici crittografici. originariamente i codebooks erano spesso libri, oggi un codebook è un sinonimo di registrazione completa di una serie di codici, indipendentemente dal formato fisico.

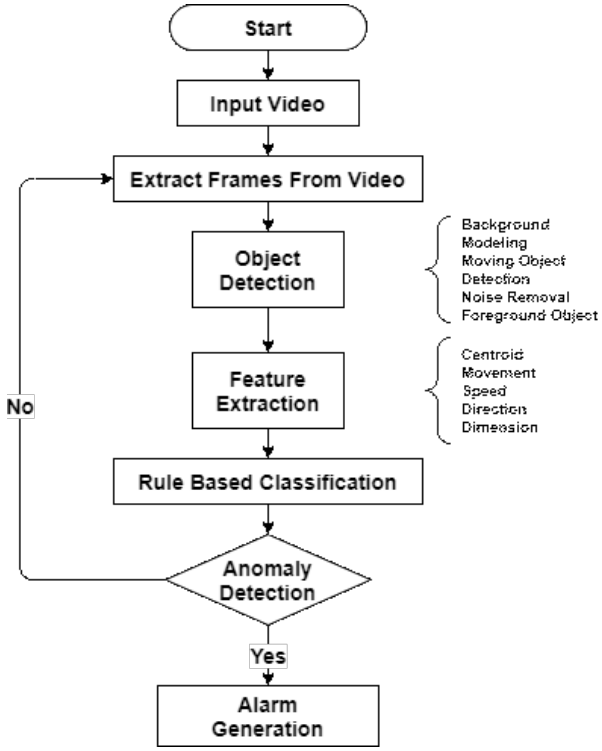

Figura 2.9: framework del modello proposto in "Multiple Anomalous Activity Detection in Videos".

2.2.3 Violence Detection con Support Vector Matchine

Le *Support Vector Machines* (o SVM)[28] sono modelli di apprendimento supervisionato[5] con algoritmi di apprendimento associati, che vengono utilizzati nell'ambito del *machine learning* come tecniche per la classificazione o regressione (anche se vengono per la maggior parte utilizzate come algoritmi di classificazione).
Dati un set di esempi per il training, ognuno marcato come appartenente a una tra due categorie, un algoritmo SVM crea un modello che assegna nuovi esempi a una categoria o all'altra, rendendolo un classificatore binario lineare e non probabilistico (anche se in realtà esistono metodi per rendere una SVM un classificatore probabilistico).
L'obbiettivo di un algoritmo SVM è quello di trovare un iperpiano in uno spazio N-dimensionale (dove N è il numero delle features) che classifichi distintamente i data points. Gli iperpiani, in questo contesto, sono confini decisionali che aiutano a classificare i data points. I punti che andranno a trovarsi in una certa zona dell'iperpiano possono essere attribuiti a diverse classi. Inoltre, la dimensione dell'iperpiano dipende dal numero di features. Se il numero di features in input è per esempio 2, allora l'iperpiano è una semplice retta. Se il numero di features è 3, allora l'iperpiano diventa un piano bidimensionale. Per separare due classi di data points, ci sono molti possibili iperpiani che possono essere scelti. L'idea è quella di trovare l'iperpiano che

[5] L'apprendimento supervisionato è quel compito del machine learning di apprendere una funzione che mappa un input in un output basandosi su coppie di input-output di esempio.

abbia il margine massimo, ovvero la massima distanza tra data points di entrambe le classi. Massimizzare il margine fornisce una specie di rinforzo per i futuri punti, in modo da poterli classificare con più confidenza.

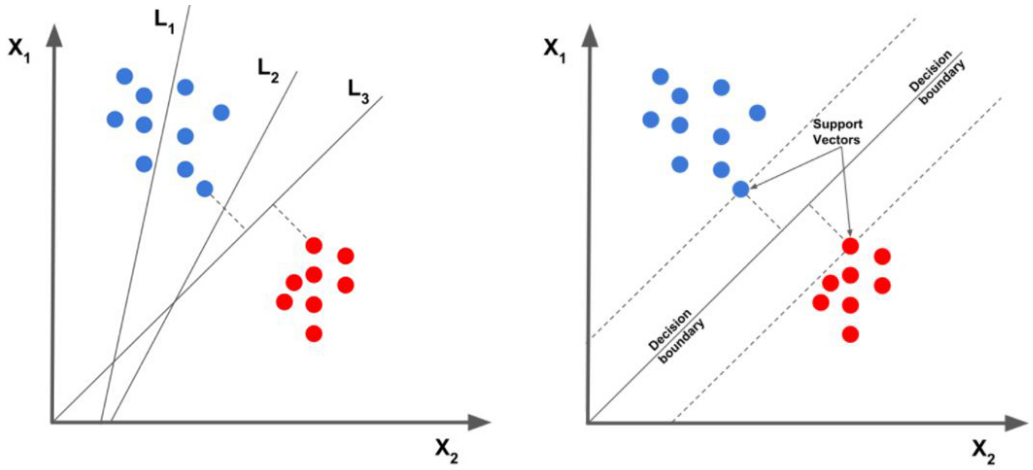

Figura 2.10: Esempio del funzionamento di una SVM.

Nella figura 2.10 è esemplificato il funzionamento di una SVM. Nella figura a sinistra abbiamo due classi di features, rappresentate nel piano come pallini blu e rossi, che identificano i vari data points. Vengono proposti tre iperpiani da utilizzare come confine decisionale: L_1, L_2 e L_3. É chiaro che L_1 non sia una buona scelta perchè non separa le due classi. L_2 e L_3 separano entrambi le classi, ma intuitivamente si vede come L_3 sia una scelta migliore poiché massimizza il margine dalle due classi. Vediamo infatti che nella figura a destra è proprio L_3 il confine decisionale utilizzato per la classificazione in questo esempio. I vettori di supporto (support vectors) sono i data points più vicini all'iperpiano e influenzano la posizione e l'orientamento dello stesso. Usando i vettori di supporto si riesce a massimizzare il margine del classificatore, infatti il metodo che utilizza una SVM per definire il confine decisionale è proprio quello di massimizzare la sua distanza dai vettori di supporto.

Le SVM possono anche eseguire efficientemente classificazioni non lineari utilizzando il metodo del "*kernel trick*", che consiste nel mappare implicitamente gli input in spazi di features a più dimensioni, come mostrato in figura 2.11.

Quando i dati non sono etichettati, l'apprendimento supervisionato non è possibile, ed è quindi necessario un approccio di apprendimento non supervisionato, che cerca di trovare un naturale raggruppamento (clustering) dei dati in gruppi, e poi mappare nuovi dati in questi gruppi formati. L'algoritmo *Support-vector Clustering*, introdotto in [29], applica le statistiche dei vettori di supporto, sviluppate nell'algoritmo della SVM, per categorizzare dati non etichettati, ed è uno degli algoritmi di clustering più utilizzati nelle applicazioni industriali.

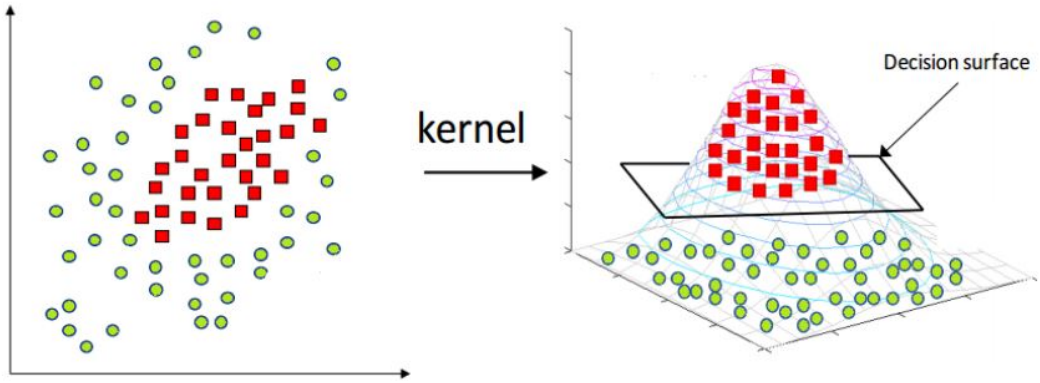

Figura 2.11: Esempio di kernel trick per effettuare classificazioni non lineari con SVM.

Le SVM sono ampiamente utilizzate nel campo della *computer vision* e negli ultimi anni sono comparsi moltissimi lavori che ne fanno uso come classificatore per la *violence detection*. Alcuni esempi possono essere trovati in [17] [18] [30] [31] [32].

2.2.4 Violence Detection con Deep Learning

Il *Deep Learning* (o Apprendimento Profondo) è un sottocampo del machine learning che fa uso delle *reti neurali*[6] per apprendere diversi livelli di rappresentazioni (da questo il nome "profondo"), che corrispondono a una gerarchia di caratteristiche, fattori o concetti, dove i concetti a più alto livello sono definiti partendo da concetti a livelli più bassi, e gli stessi concetti a basso livello possono aiutare a definire molti concetti a livello più alto.
Per esempio, guardando la figura 2.12, potremmo pensare che il livello più basso (quelli colorato di rosso) identifichi le informazioni di base di un'immagine, come spigoli e angoli, mentre andando più in alto con i livelli si abbiano rappresentazioni più significative per un umano, come lettere, numeri o parti del corpo, ottenuti partendo dai concetti identificati dai livelli più bassi.
Il *Deep Learning* fa parte di una più ampia famiglia di metodi del machine learning basata sul *representation learning*, ovvero sull'utilizzo di tecniche che permettono a un sistema di trovare automaticamente le rappresentazioni richieste per il rilevamento di features. Un'immagine può essere rappresentata in molte maniere (per esempio dei vettori di pixels), ma alcune rappresentazioni rendono più facile apprendere i task di interesse (*p.es.* questa immagine rappresenta un volto umano?) da esempi. I campi di applicazione del *Deep learning* non si limitano alla *Computer Vision*, ma sono ampiamente utilizzate anche nei campi del riconoscimento vocale, dell'elaborazione

[6] Nel campo dell'apprendimento automatico, una rete neurale artificiale (in inglese artificial neural network, abbreviato in ANN o anche come NN) è un modello computazionale composto di "neuroni" artificiali, ispirato vagamente dalla semplificazione di una rete neurale biologica.

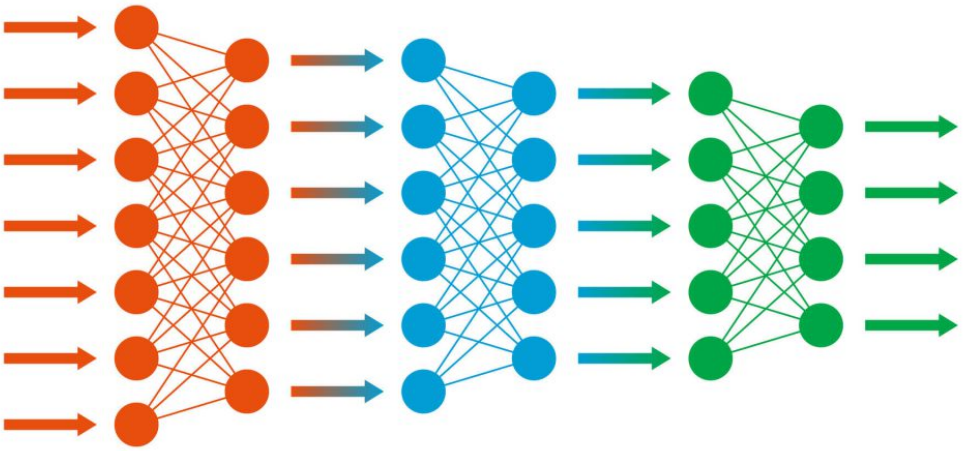

Figura 2.12: Esempio di un'architettura che utilizza il concetto di *Deep Learning*.

del linguaggio naturale, della traduzione automatica, della bioinformatica, dello sviluppo di farmaci e dell'analisi di immagini mediche.

Le architetture del *Deep Learning* ad oggi più utilizzate sono le *deep neural networks, deep belief networks, graph neural networks, recurrent neural networks e convolutional neural networks.*
Una ***Deep Neural Network*** (Rete neurale profonda o DNN) è una rete neurale che presenta layers multipli tra i layers di input e di output. Le DNN sono tipicamente reti "feedforward", nelle quali i dati viaggiano dal layer di input a quello di output senza tornare indietro. Per prima cosa la DNN crea una mappa di neuroni virtuali e assegna un valore numerico random, o "peso", alle connessioni tra di essi. I pesi e gli input vengono moltiplicati e ritornano un output tra 0 e 1. Se la rete non riesce a riconoscere accuratamente un particolare pattern, un algoritmo "calibra" i pesi. In questo modo l'algoritmo può rendere alcuni paramentri più influenti, fino a quando riesce a determinare la manipolazione matematica corretta per processare completamente i dati.
Una ***Deep Belief Network*** (o DBN) è un modello grafico generativo, o alternativamente una classe di DNN, composta da layers multipli di variabili latenti (unità nascoste) con connessioni tra i layers ma non tra le unità di ciascun layer. Se addestrata in un set di esempi senza supervisione, una DBN riesce ad imparare a ricostruire probabilisticamente i suoi input. I layers poi si comportano come rilevatori di features. Dopo questo step di apprendimento, una DBN può essere successivamente addestrata con supervisione per eseguire la classificazione.
Una ***Graph Neural Network*** (o GNN) è un tipo di rete neurale che opera direttamente in una struttura a grafo. Una tipica applicazione delle GNN è quella della

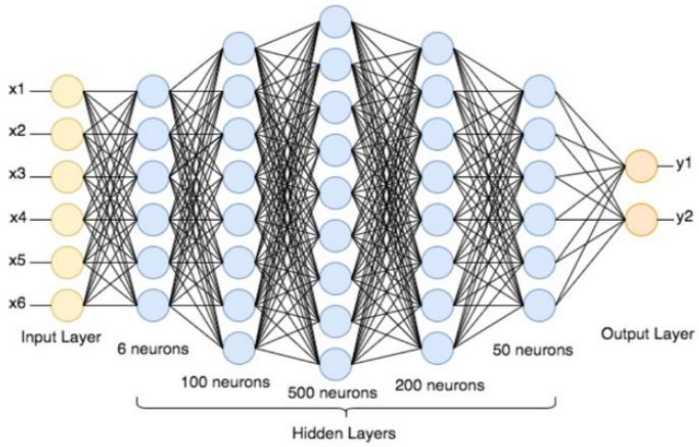

Figura 2.13: Esempio di *Deep Neural Network*.

"node classification". Essenzialmente, ogni nodo nel grafo è associato a un'etichetta, e si vuole prevedere l'etichetta dei nodi senza averne prova a priori.
Una ***Recurrent Neural Network*** (rete neurale ricorrente o RNN) è una classe di DNN in cui le connessioni tra i nodi formano un grafo orientato lungo una sequenza temporale. Questo le permette di esibire un comportamento temporale dinamico. Le RNN possono utilizzare il loro stato interno (la loro memoria) per processare sequenze di input a lunghezza variabile. Questo le rende applicabili in compiti come il riconoscimento della grafia o il riconoscimento vocale.
Una ***Convolutional Neural Network*** (rete neurale convoluzionale o RNN) è una classe di DNN che utilizza dei layers detti "convoluzionali" che estraggono attraverso l'uso di filtri delle caratteristiche dalle immagini di cui si vuole analizzare il contenuto. Parleremo più approfonditamente delle reti neurali convoluzionali nel capitolo successivo, in quanto oggetto di questo lavoro.

Il *Deep Learning* è l'approccio alla *violence detection* recentemente più utilizzato e moltissimi lavori vengono tutt'oggi presentati a riguardo. Alcuni di questi sono consultabili in [33] [34] [35] [36] [37].

Capitolo 3

Le Reti Neurali Convoluzionali

Come accennato precedentemente, in questo capitolo si parlerà delle *Reti Neurali Convoluzionali*, che sono le architetture con le quali i modelli implementati e testati in questo elaborato sono costruiti.
Per prima cosa verranno introdotte le reti neurali convoluzionali standard, definendone architettura e layers utilizzati. Si parlerà poi delle reti convoluzionali 3D e della loro differenza da quelle standard. In un'ultima sezione verrà introdotta una particolare tipologia di *rete neurale ricorrente*, detta LSTM, e di come questa possa essere utilizzata in accordo al modello convoluzionale per catturare caratteristiche spazio-temporali nei video.

3.1 Architettura di una Rete Neurale Convoluzionale

Le *Reti Neurali Convoluzionali* (Convolutional Neural Networks o CNN) sono una particolare classe di *Deep Neural Networks* che riesce a riconoscere e classificare caratteristiche da immagini attraverso l'uso di kernel (filtri) ed è per questo ampiamente utilizzata per analizzare video ed immagini.
Le CNN devono il loro nome all'operazione matematica della "convoluzione" di cui fanno uso per estrarre le caratteristiche dalle immagini. La convoluzione è un'operazione tra due funzioni che produce una terza funzione che esprime come la forma di una viene modificata dall'altra. É definita come l'integrale del prodotto delle due funzioni dopo che una viene invertita e traslata. L'integrale viene valutato per tutti i valori di traslazione, producendo la funzione di convoluzione:

$$(f * g)(t) := \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau = \int_{-\infty}^{\infty} f(t-\tau)g(\tau)d\tau \tag{3.1}$$

Nel campo del processamento delle immagini, l'operazione di convoluzione può essere effettuata tra due immagini (che possono essere rappresentate come matrici) per ottenere un output che viene utilizzato per estrarre caratteristiche dalle immagini.
Le CNN sono varianti delle reti neurali profonde standard, anche dette *Multilayer perceptron* (MLP). Una MLP è una rete neurale feedforward, composta da almeno un layer nascosto oltre a quelli di input e di output. Le MLP sono reti neurali "fully-connected", ovvero ogni neurone di un layer è connesso a tutti i neuroni del

layer successivo. Questa caratteristica la rende un'architettura molto pesante, in quanto il numero di connessioni può diventare enorme se la quantità di neuroni è elevata. Questa pesantezza rende le MLP inutilizzabili in alcune discipline, come quelle del riconoscimento e classificazione di immagini. Un'immagine di 1000x1000 pixel con canali RGB corrisponde a 3 milioni di pesi in un'architettura fully-connected (1000*1000*3), numero troppo alto per processare efficientemente l'immagine. Inoltre, questi tipi di reti neurali non tengono conto della struttura spaziale dei dati, trattando quindi pixel che sono distanti tra loro nello stesso modo con cui trattano pixel vicini tra loro. Questo comportamento ignora il principio di località in dati con una topologia a griglia (come i pixel delle immagini). Si è quindi cercato un sostituto a questa architettura che non avesse questi problemi e lo si è trovato nelle reti neurali convoluzionali. Le CNN differiscono dalle MLP per tre caratteristiche fondamentali:

- *Volumi 3D di neuroni*: le reti neurali convoluzionali sfruttano il fatto che l'input è costituito da immagini e vincolano l'architettura in modo più sensato. In particolare, i layer di una CNN hanno neuroni organizzati in 3 dimensioni: largezza, altezza e profondità.

- *Connettività locale* (Local connectivity): ogni neurone di un layer è connesso solo a una piccola regione del layer precedente. Questa regione è detta *receptive field*. Questa caratteristica garantisce che i "filtri" appresi producano la risposta più forte a un pattern di input spazialmente locale. L'accatastamento di molti di questi layers porta a filtri non lineari che diventano sempre più globali (cioè reattivi a una regione spaziale più ampia di pixel) in modo che la rete crei prima rappresentazioni di piccole parti dell'input, e poi da esse assembli rappresentazioni di aree più grandi.

- *Pesi condivisi* (Shared Weights): nelle CNN, ogni filtro viene replicato sull'intero campo visivo. Queste unità replicate condividono gli stessi parametri (stessi pesi e bias). Questo significa che i pesi vengono condivisi a gruppi, e quindi il numero di pesi totale è ridotto.

Insieme, queste proprietà permettono alle CNN di raggiungere generalizzazioni migliori in problemi visivi. La condivisione dei pesi riduce drasticamente il numero di parametri liberi appresi, riducendo quindi i requisiti di memoria per eseguire le reti neurali e permettendo quindi l'addestramento di reti più grandi e potenti.
Un'architettura standard di rete neurale convoluzionale è suddivisa in due parti principali:

- Uno strumento convoluzionale che identifica e separa le varie caratteristiche di un immagine nel processo di *Feature Extraction*.

- Un layer fully-connnected che utilizza l'output del processo convoluzionale e predice la classe dell'immagine basata sulle features estratte negli stadi precedenti.

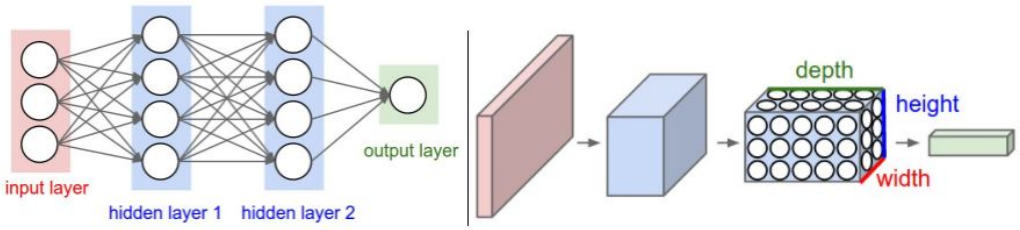

Figura 3.1: A sinistra un esempio di semplice MLP con 2 layers nascosti. A destra l'architettura semplificata di una CNN.

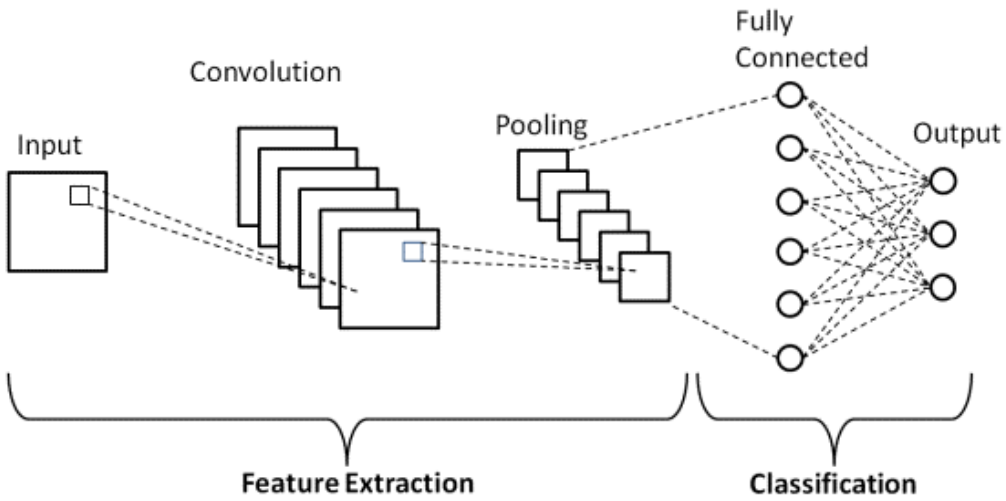

Figura 3.2: Suddivisione di una CNN in due processi, uno di feature extraction e l'altro di classificazione.

Una rete neurale convoluzionale è composta da diversi tipi di layers:

- Layer di input
- Layer convoluzionale
- Layer di pooling
- Layer fully-connected
- Layer Softmax
- Layer di dropout

3.1.1 Layer di Input

Il layer di input è il layer di ingresso della CNN e contiene i dati di un'immagine. Pensando a un'immagine, sappiamo che ha una sua altezza e una sua larghezza, quindi avrebbe senso rappresentare l'informazione contenuta in essa in una matrice

bidimensionale. Le immagini però hanno anche i colori, e per rappresentare l'informazione dei colori abbiamo bisogno di un'altra dimensione. Le immagini sono codificate in canali di colori (i più comuni sono RGB: Red Green Blue), e i dati vengono rappresentati come l'intensità che il colore di un certo canale ha in un dato punto.

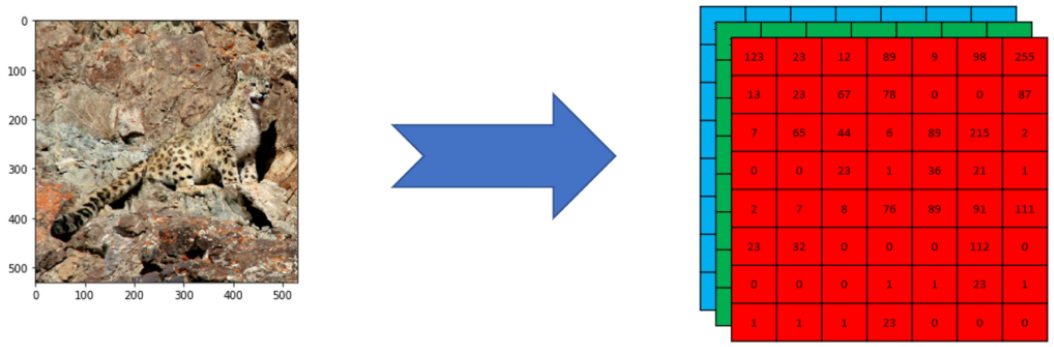

Figura 3.3: Rappresentazione RGB di un'immagine

Quindi il layer di input conterrà un'immagine rappresentata come array tridimensionale in cui ogni cella dell'array rappresenta l'intensità di colore di un canale nel pixel corrispondente.

3.1.2 Layer Convoluzionale

Il layer convoluzionale è il principale layer che compone una *rete neurale convoluzionale*.
Una convoluzione è una semplice applicazione di un filtro in un input che risulta in un'attivazione. Applicazioni ripetute dello stesso filtro su un input risultano in una mappa di attivazioni chiamata *Feature Map* (mappa di feature), che indica la posizione e la robustezza di una feature rilevata in un input, che è spesso un'immagine. Nel contesto di una *convolutional neural network*, una convoluzione è un'operazione lineare che coinvolge la moltiplicazione di un set di pesi posti come array bidimensionale, chiamato filtro o nucleo (kernel), con l'input. Il filtro è più piccolo dell'input e il tipo di moltiplicazione applicata tra il filtro e la parte di immagini di dimensioni del filtro è un prodotto scalare. Il risultato di questa operazione è quindi sempre un valore singolo.
L'utilizzo di un filtro più piccolo dell'input è intenzionale in quanto permette allo stesso filtro (e quindi allo stesso set di pesi) di essere moltiplicato per l'array di input più volte in punti diversi dell'input. In particolare, il filtro viene applicato sistematicamente a ogni "quadro" delle dimensioni del filtro presenti nell'input, da sinistra a destra, da sopra a sotto.
L'appplicazione sistematica dello stesso filtro è molto utile per evidenziare specifiche caratteristiche in un'immagine. Infatti, se il filtro è progettato per rilevare uno specifico tipo di feature in un input, allora l'applicazione di quel filtro sistematica-

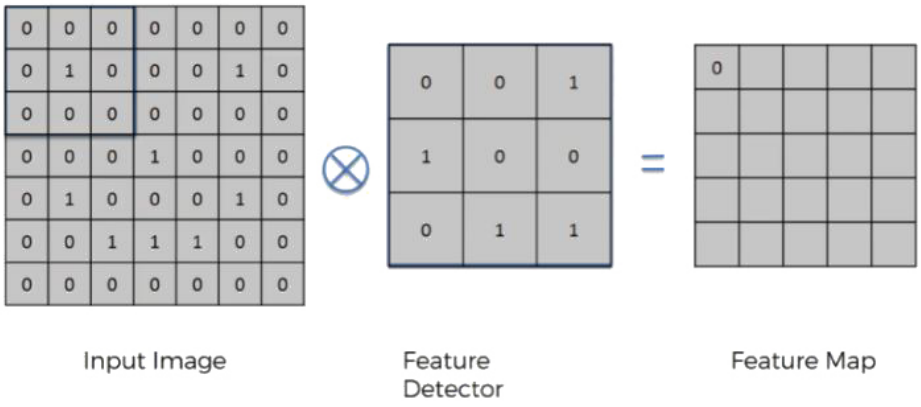

Figura 3.4: Singola applicazione di un filtro 3x3

mente su tutto l'input dà l'opportunità al filtro di scoprire quella feature ovunque sia nell'immagine. Questa capacità è comunemente indicata come *Translation Invariance* (invarianza alle traslazioni). L'invarianza alle traslazioni locali può essere una proprietà molto utile in quanto evidenzia la presenza di una feature indipendentemente dalla sua posizione. Per esempio, quando vogliamo determinare se un'immagine contiene una faccia, non ci serve sapere la posizione degli occhi con precisione assoluta, ci serve solo sapere che c'è un occhio nella parte sinistra della faccia e un occhio nella parte destra della faccia.

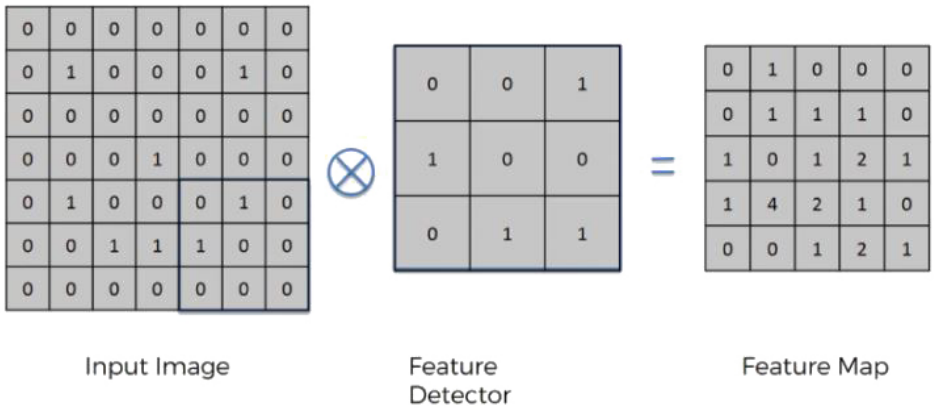

Figura 3.5: Applicazione completa di un filtro 3x3

Ricapitolando, l'output ottenuto moltiplicando il filtro con l'array di input una volta è un singolo valore. Quando il filtro viene applicato più volte al vettore di input, il risultato è un array bidimensionale di valori di output che rappresenta un filtraggio dell'input chiamato *Feature Map*.

Alcuni parametri interessanti di un layer convoluzionale sono: *Depth, Stride* e *Padding*. Il *Depth* è la profondità del layer convoluzionale. Il risultato dell'applicazione di un

filtro su un'immagine in input è una *feature map*, e lo stesso filtro produce sempre la stessa *feature map*. Normalmente siamo interessati ad estrarre più features dalla stessa immagine, e questo è ottenibile applicando più filtri diversi allo stesso input. Questo parametro rappresenta il numero di filtri applicati e quindi il numero di *feature map* ottenute.

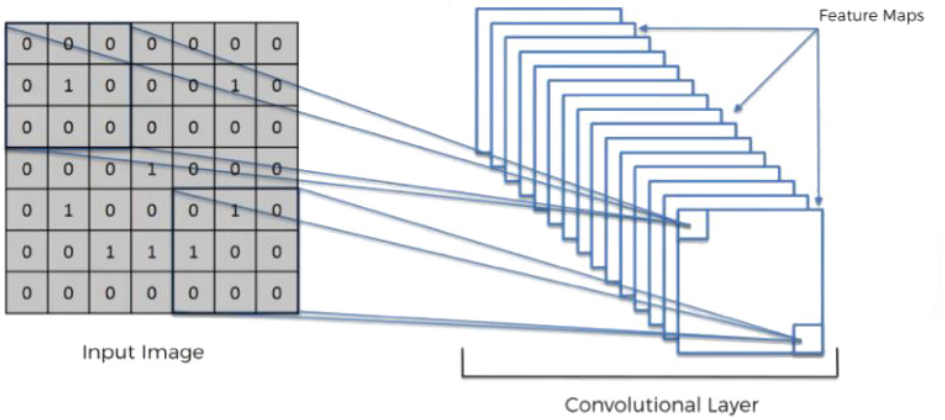

Figura 3.6: Il parametro *Depth* indica il numero di feature map estratte da uno stesso input.

Lo *Stride* è il parametro che indica di quanti pixel si sposta il filtro dopo essere stato applicato su una regione dell'input. Più lo spostamento è grande e più la *feature map* in output sarà piccola. Se per esempio abbiamo in input un'immagine 7x7 e un filtro 3x3, l'applicazione del filtro con *stride* pari a un pixel produrrà una feature map 5x5, mentre un filtro con *stride* pari a due pixel produrrà una feature map 3x3.

Il *Padding* è utilizzato per evitare di perdere informazioni da un'immagine. Quando si applica un filtro convoluzionale su un input si tende a perdere alcuni pixel del perimetro dell'immagine. Tipicamente si utilizzano filtri piccoli, e quindi per ogni layer convoluzionale si tende a perdere solo pochi pixels, però questa perdita si va poi a sommare a quelle dei layer convoluzionali successivi. Una soluzione a questo problema è l'aggiunta di pixels extra di riempimento intorno al contorno dell'immagine, incrementando di fatto la dimensione dell'immagine. Solitamente, si settano i valori dei pixels extra a zero.

Solitamente dopo l'applicazione di un filtro convoluzionale si passa l'input a una funzione ReLU. Una *Rectified Linear Unit* (o ReLU) è una funzione di rettificazione che serve per incrementare la non-linearità delle immagini. Il motivo per cui vogliamo fare ciò è che le immagini sono di natura non lineari. Quando facciamo passare un'immagine attraverso un filtro convoluzionale, gli imponiamo una certa linearità. La funzione di rettificazione serve per "rompere" questa linearità non voluta.

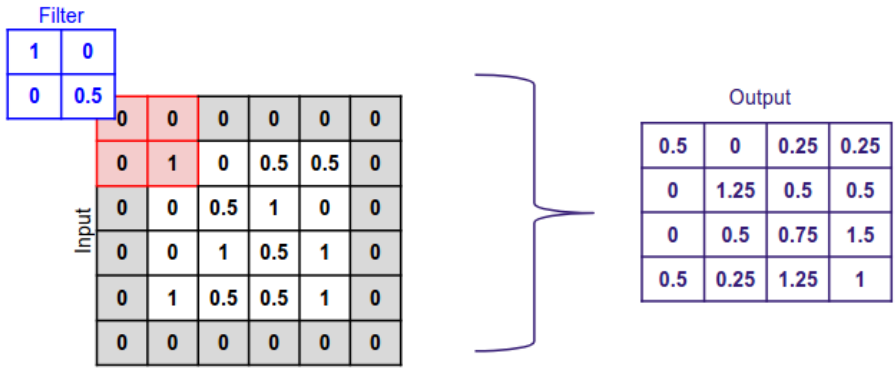

Figura 3.7: Esempio di *Padding* applicato a un input 4x4.

3.1.3 Layer di Pooling

Una limitazione delle *feature maps* ottenute come output dei layers convoluzionali è che registrano la precisa posizione delle features nell'input. Questo significa che piccoli movimenti nella posizione della feature nell'immagine in input risulteranno in una *feature map* differente. Questo può accadere per rotazioni, traslazioni e altri piccoli cambiamenti nell'immagine in input.
Un approccio comune per risolvere questo problema è quello di effettuare un "*down sampling*" delle feature maps (ovvero un sottocampionamento). Questa tecnica deriva dalla teoria dei segnali, e sarebbe la pratica di creare una versione a minore risoluzione del segnale in input mantenendo tuttavia gli elementi strutturali più importanti, liberandosi solo di piccoli dettagli che potrebbero non essere utili allo scopo. Il *down sampling* può essere eseguito tramite i layers convoluzionali aumentando lo stride della convoluzione. Un approccio più robusto è però quello di usare un layer di pooling. Un layer di pooling è un nuovo layer che viene aggiunto dopo il layer convoluzionale e più specificamente dopo aver applicato una non linearità (solitamente ReLu) al layer convoluzionale. L'aggiunta del layer di pooling dopo il layer convoluzionale è un pattern comunemente usato per organizzare i layers in una *rete neurale convoluzionale* che viene ripetuta nello stesso modello. Per esempio, i layers potrebbero essere disposti in questo modo:

- Layer di input
- Layer convoluzionale
- Layer di pooling
- Layer convoluzionale
- Layer di pooling
- Layer convoluzionale
- Layer di pooling
- Ecc..

Un layer di pooling opera su ogni *feature map* separatamente per creare un nuovo set di *feature maps* ridimensionate.
L'operazione di pooling riguarda la scelta di un filtro che verrà applicato alle *feature maps*. La dimensione del filtro è minore della dimensione delle *feature maps*; quasi sempre è una 2x2 pixels applicata con uno stride di 2 pixels. Questo significa che il layer di pooling ridurrà sempre la dimensione di ogni *feature map* di un fattore di 2, ovvero ogni dimensione sarà dimezzata, riducendo il numero di pixels in ogni *feature map* a un quarto. Per esempio, un layer di pooling applicato a una *feature map* 6x6 (36 pixels) risulterà in una *feature map* in output 3x3 (9 pixels).
Le due funzioni più comuni utilizzate in un'operazione di pooling sono l'*Average Pooling* e il *Max Pooling*.
Il *Max Pooling* consiste nel calcolare il valore massimo contenuto nelle regioni della *feature map* di dimensioni del filtro utilizzato per l'operazione. Il risultato dell'applicazione di questa tecnica è una *feature map* che evidenzia le features più presenti nelle regioni dell'immagine.

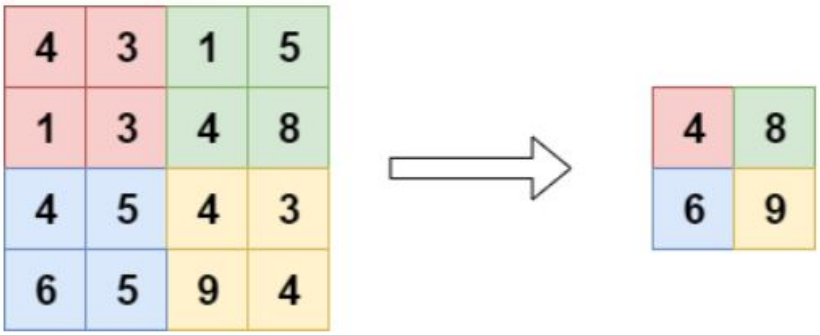

Figura 3.8: Esempio dell'applicazione del *Max Pooling* su un input 4x4.

L'*Average Pooling* consiste nel calcolare la media tra i valori di ogni regione della *feature map* di dimensione del filtro utilizzato nell'operazione. Questo significa che ogni regione di una *feature map* viene "riassunto" come la media dei valori contenuti al suo interno.

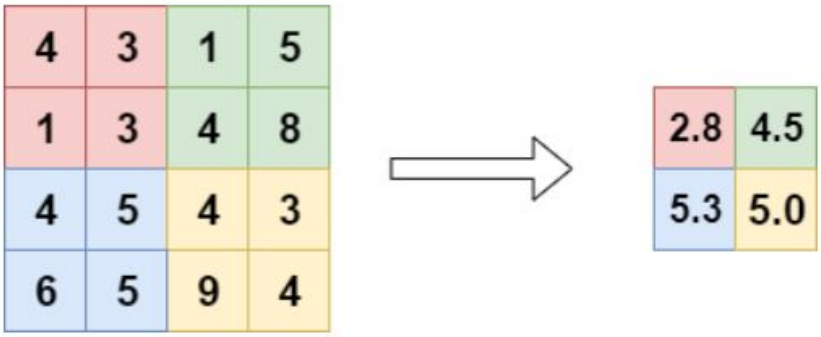

Figura 3.9: Esempio dell'applicazione dell'*Average Pooling* su un input 4x4.

L'*Average Pooling* è diverso dal *Max Pooling* in quanto tira fuori molte più informazioni riguardanti gli elementi meno importanti di una regione. Mentre il *Max Pooling* semplicemente "butta via" le informazioni meno importanti, l'*Average Pooling* le mescola con quelle più importanti.

Questa caratteristica può essere utile in alcune applicazioni, ma è facile immaginare che la tecnica più utilizzata sia la prima, in quanto le features tendono a codificare la presenza spaziale di un certo pattern sulle diverse regioni di una *feature map*, ed è quindi più informativo osservare la massima presenza di diverse features piuttosto che la loro presenza media [38].

Il risultato dell'utilizzo di un layer di pooling e della creazione di *feature maps* sottocampionate è una versione "riassunta" delle features rilevate nell'input. Queste versioni ridimensionate sono molto utili in quando piccoli cambiamenti nella posizione di una feature nell'immagine di input rilevata dal layer convoluzionale risulteranno in una *feature map* sottocampionata con la feature nella stessa posizione. Questa proprietà è chiamata *invarianza su traslazioni locali*.

Inoltre, l'operazione di pooling ha l'ulteriore vantaggio di ridurre il numero di parametri (in quanto la dimensione dell'immagine viene ridotta), e questo serve anche a prevenire il problema dell'*overfitting*. In statistica e in informatica, si parla di overfitting (in italiano: adattamento eccessivo, sovradattamento) quando un modello statistico molto complesso si adatta ai dati osservati (il campione) perché ha un numero eccessivo di parametri rispetto al numero di osservazioni. Un modello assurdo e sbagliato può adattarsi perfettamente se è abbastanza complesso rispetto alla quantità di dati disponibili [39].

3.1.4 Layer Fully-connected e Softmax

Il layer fully-connected e il layer di Softmax sono i layer che si occupano della classificazione. Sono quindi posti dopo la sequenza di layers convoluzionali e di pooling. Il layer fully-connected è costituito da una semplice MLP feed-forward che prende in input le *feature maps* ottenute attraverso i processi di convoluzione e pooling, appiattite attraverso un'operazione di *flattening*. L'output degli ultimi layer convoluzionali e di pooling è infatti una matrice tridimensionale, l'operazione di *flattening* sarebbe lo "srotolamento" della matrice in un vettore, che può essere utilizzato come input della feed-forward neural network.

Dopo essere passati attraverso i vari layers nascosti della MLP, il layer finale utilizza una funzione di attivazione chiamata *Softmax*, che serve per ottenere la probabilità che l'input appartenega a una particolare classe. In matematica, una funzione *softmax*, o funzione esponenziale normalizzata è una generalizzazione di una funzione logistica che comprime un vettore k-dimensionale z di valori reali arbitrari in un vettore k-dimensionale $\sigma(z)$ di valori compresi in un intervallo (0,1) la cui somma è 1

[40]:

$$\sigma(z)_j = \frac{e^{z_j}}{\sum_{k=1}^{K} e^{z_k}} \qquad perj = 1, ..., K \tag{3.2}$$

L'utilizzo della funzione *softmax* è importante poiché se non venisse utilizzata le probabilità che otterremmo dall'ultimo layer fully-connected sarebbero valori reali che però sommati non danno 1. Questo significa che per esempio, in una CNN che ha lo scopo di capire se l'immagine è un cane o un gatto, potremmo ottenere probabilità del tipo: 90% gatto e 55% cane. Questi risultati sono virtualmente inutili.

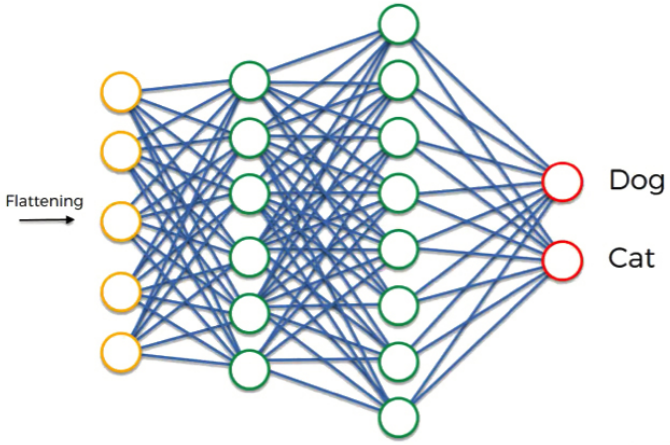

Figura 3.10: Esempio di layer fully-connected di una CNN che ha lo scopo di distinguere cani e gatti.

In conclusione il risultato di queste operazioni sono le varie probabilità che ha un oggetto di un immagine di appartenere alle diverse classi utilizzate nella classificazione.

3.1.5 Layer di Dropout

Reti neurali molto grandi e complesse che vengono addestrate su datasets relativamente piccoli possono cadere nel problema dell'*overfitting*. Il layer di dropout ha lo scopo di ridurre questo problema. Può essere inserito tra i vari layers di una *rete neurale convoluzionale*[41].
Durante il training, un numero di outputs (nodi) scelti a caso del layer precedente a quello di dropout vengono ignorati o "abbandonati". Questo ha l'effetto di far sembrare il layer come un layer con un numero diverso di nodi dal precedente. Con "abbandonare" un nodo si intende rimuoverlo temporaneamente dalla rete, insieme alle sue connessioni agli altri nodi. Il dropout ha l'effetto di rendere il processo di training rumoroso, costringendo i nodi all'interno di uno strato ad assumersi probabilisticamente più o meno responsabilità per gli inputs. Questa concettualizzazione suggerisce che forse il dropout interrompe quelle situazioni in cui i layers della rete si adattano insieme per correggere gli errori dei livelli precedenti,

rendendo quindi il modello più robusto. Infatti, questi meccanismi di adattamento collaborativo possono risultare in overfitting, in quanto non si generalizzano a dati mai visti.

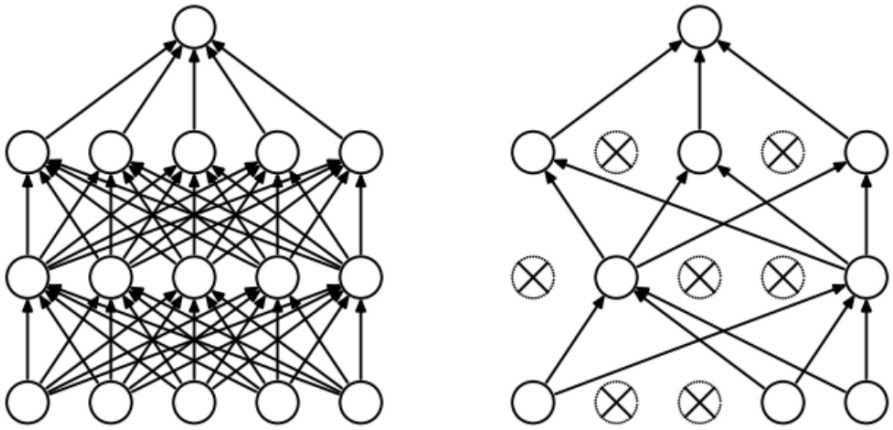

Figura 3.11: Esempio di applicazione del dropout.

3.1.6 Training

Per concludere la trattazione delle reti neurali convoluzionali tradizionali, in questa sezione verrà introdotto brevemente il metodo che viene utilizzato per l'addestramento delle tali.
Il processo di training utilizzato dalle rete neurali è chiamato *backpropagation*, ed è separato in 4 passi principali:

1. *forward propagation*: l'input viene processato dalla rete basandosi sul valore dei pesi e viene prodotta in output la predizione sull'appartenenza dell'input alle varie classi. Il training è costituito di un ciclo di operazioni che si ripetono chiamato *epoca*. Nella prima epoca i pesi della rete vengono inizializzati in maniera casuale, quindi l'output ottenuto in questo caso sarà con molta probabilità errato.

2. *Loss Function*: l'addestramento di una rete utilizza immagini etichettate, con l'etichetta che ne definisce la classe di appartenenza. Il risultato ottenuto dopo il primo ciclo sarà probabilmente errato, quindi differente dall'etichetta. Si vuole minimizzare l'errore che c'è tra l'output ottenuto e l'output desiderato, ovvero la reale classe di appartenenza di un'immagine. Questo è ottenuto attraverso funzioni dette *loss functions*, e la più popolare è denominata *Mean Squared Error* (errore quadratico medio):

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(Y_i - \hat{Y}_i)^2 \tag{3.3}$$

dove n è il numero di predizioni, Y_i il vettore dei valori di target e $\hat{Y}_i$ il vettore delle predizioni.

3. *Backward Pass*: si determina quali pesi hanno contribuito di più nell'errore e si cercano modi per calibrare il loro valore in modo da ridurre il *loss*. Questo è effettuato calcolando il gradiente dell'errore rispetto ai pesi dei vari filtri della rete.

4. *Weight Update*: In questo passaggio si aggiornano i pesi della rete, utilizzando la seguente espressione:

$$w = w_i - \eta \frac{dL}{dW} \tag{3.4}$$

dove w è il valore del peso aggiornato, w_i è il valore del peso prima dell'aggiornamento, L è l'errore, W i pesi del filtro, e η è il *learning rate*. Il *learning rate* è un parametro scelto dal programmatore. Più grande è il suo valore più lunghi saranno i passi verso l'ottimizzazione e più lunghi saranno i tempi che portano a un peso ottimizzato.

3.2 Reti Convoluzionali 3D

Nella sezione 2.1 abbiamo detto che per identificare atti violenti abbiamo bisogno sia dell'informazione spaziale che dell'informazione temporale (informazioni spazio-temporali). Le reti convoluzionali tradizionali utilizzano kernel di due dimensioni, che non rendono possibile l'acquisizione di entrambi le informazioni contemporaneamente. Come descritto in [42], infatti, le *reti convoluzionali 2D* perdono le informazioni temporali del segnale d'ingresso dopo ogni operazione di convoluzione, mantenendo solo l'informazione spaziale. Abbiamo quindi bisogno di un'architettura che sia capace di mantenere entrambi i tipi di informazioni se vogliamo utilizzare una CNN nel campo della *Violence Detection*.
Una *rete convoluzionale 3D* è una rete convoluzionale che fa uso di operazioni di convoluzione e pooling 3D [43]. Utilizza quindi kernel tridimensionali che permetteno di tenere traccia sia delle informazioni spaziali che di quelle temporali, inglobandole entrambi nelle feature maps ottenute come output delle operazioni 3D.

3.2.1 Operazioni di Convoluzione e Pooling 3D

Le operazioni di Convoluzione e Pooling 3D sono simili ai loro rispettivi 2D, solo che utilizzano filtri tridimensionali al posto di quelli bidimensionali utilizzati dalle CNN standard.
La convoluzione 3D è ottenuta effettuando un'operazione di convoluzione di un kernel 3D sul cubo formato impilando più frames contigui. In questo modo, le feature maps nel layer convoluzionale sono connesse a più frames contigui nel layer precedente, catturando così le informazioni sul movimento, ovvero le informazioni temporali.

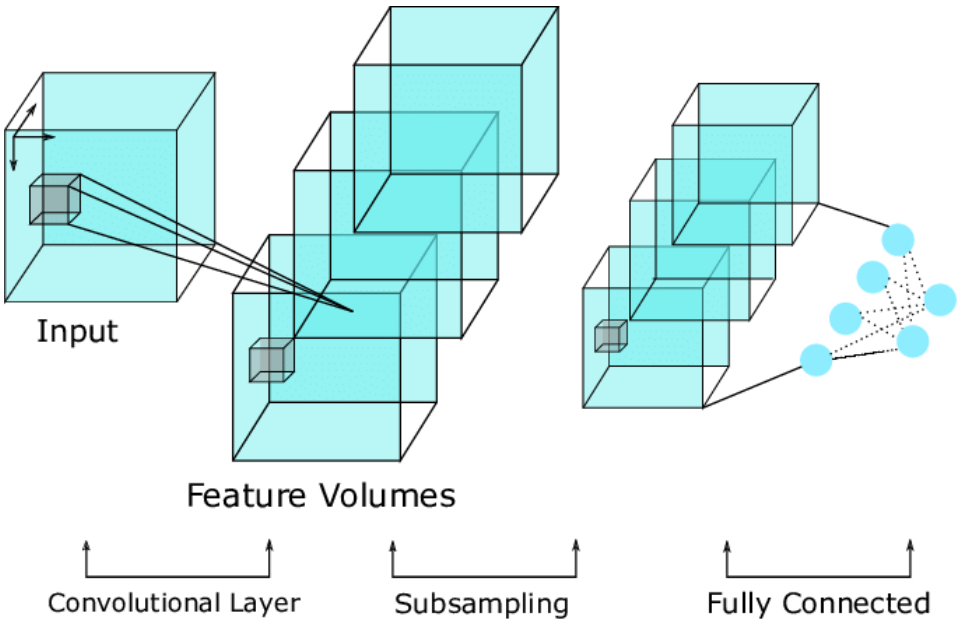

Figura 3.12: Architettura di una *rete convoluzionale 3D*.

Le feature maps ottenute con questa operazione saranno quindi tridimensionali, e vengono perciò chiamate *Feature Volumes*.

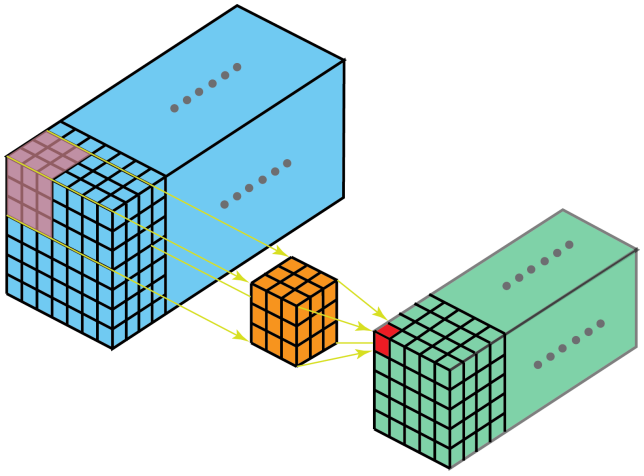

Figura 3.13: Funzionamento di un'operazione di convoluzione 3D.

Applicare un'operazione di convoluzione 3D a un'insieme di frames è diverso dall'applicare un'operazione di convoluzione 2D su frames multipli, trattando ogni frame come canale diverso. La figura 3.14 ci mostra la differenza: la convoluzione 2D applicata ad un'immagine genera in output un'immagine (a); la convoluzione 2D applicata a immagini multiple su diversi canali genera anch'essa un'immagine (b); la convoluzione 3D applicata a un volume (insieme di frames consecutivi) genera un volume 3D; la convoluzione 3D applicata a più volumi su diversi canali genera anch'essa un volume 3D. Dato che un'operazione di convoluzione 2D genera comunque

un'immagine 2D, non mantiene le informazioni sul movimento anche se applicata a più frames.

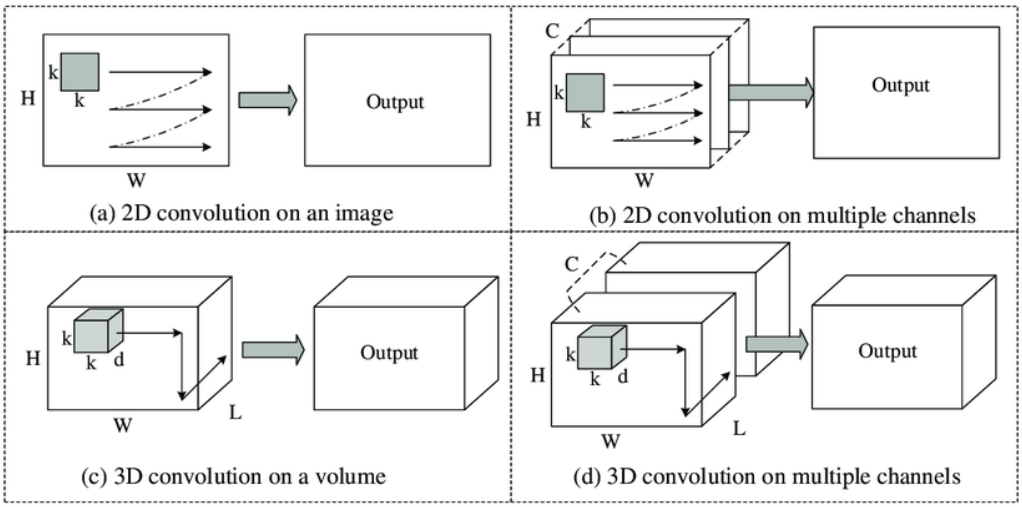

Figura 3.14: Differenza tra la convoluzione 2D e la convoluzione 3D.

L'operazione di pooling 3D è molto simile alla sua controparte 2D, si utilizzano infatti le stesse funzioni usate nell'approccio bidimensionale, ma applicate a volumi tridimensionali. Abbiamo quindi funzioni come *Average Pooling 3D* e *Max Pooling 3D*, che effettuano rispettivamente la media e il massimo tra gli elementi contenuti in un volume $NxNxN$, generando in output un *downsampling* del *feature volume* che continua ad essere tridimensionale.

3.3 Long Short-Term Memory Networks

Una *Long Short-Term Memory* (LSTM) è una particolare tipologia di *Rete Neurale Ricorrente* (RNN) introdotta nel 1997 in [44].
Una *Rete Neurale Ricorrente*, come accennato nel capitolo precedente, è un particolare tipo di rete neurale che contiene dei cicli, i quali permettono all'informazione di persistere, fornendo all'architettura una specie di memoria nella quale salvare le informazioni. Anche se detto così possono sembrare misteriose, le RNN in realtà possono essere viste come più copie della stessa rete neurale, ognuna che passa un messaggio alla successiva. Questo può essere visualizzato nella figura 3.15, dove x_i è l'input della rete neurale, h_t è l'output e A è la rete neurale.

3.3.1 Struttura di una LSTM

Un problema delle *Recurrent Neural Networks* tradizionali è quello delle "*long-term dependencies*" (dipendenze a lungo termine). Le RNN sono molto efficaci quando le informazioni da analizzare sono recenti, mentre non riescono ad adattarsi alle

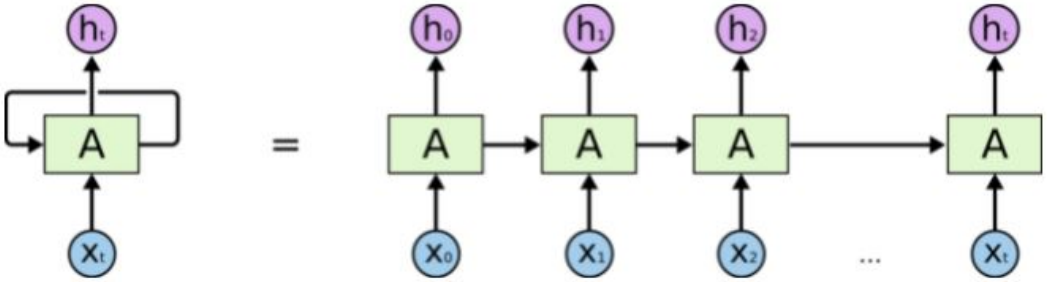

Figura 3.15: Esempio di una RNN "srotolata".

situazioni in cui potrebbero servire informazioni meno recenti (le dipendenze a lungo termine).
É per questo che sono state introdotte le *Long Short-Term Memories*. Le LSTM sono particolari RNN che riescono ad apprendere dipendenze a lungo termine. La loro caratteristica principale è quindi quella di ricordarsi di informazioni per lunghi periodi di tempo. Anche le LSTM possono essere viste come una concatenazione di blocchi ripetuti, solo che a differenza delle normali RNN (che solitamente hanno al loro interno un singolo layer "tanh") ogni blocco ripetuto ha una struttura più complessa, descritta nella figura 3.16.

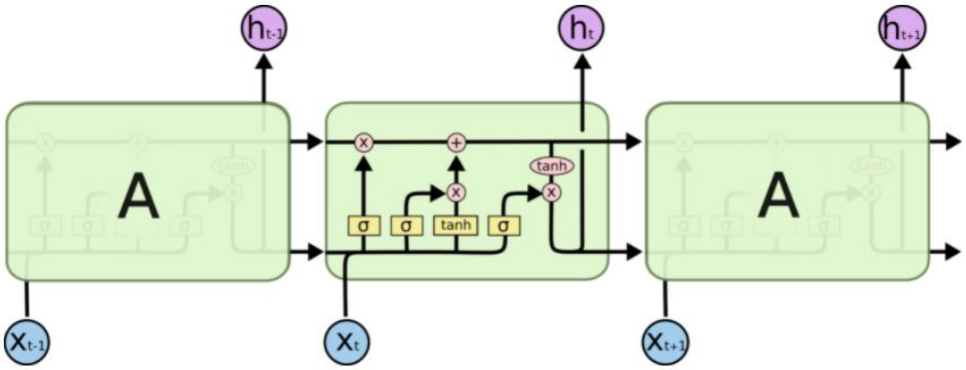

Figura 3.16: Diagramma della struttura di una LSTM.

Ogni linea all'interno del blocco trasporta un intero vettore, i cerchi viola rappresentano operazioni di somma o prodotto tra gli elementi dei vettori, mentre i rettangoli gialli sono reti nuerali. Il componente principale della LSTM è la linea orizzontale posta nella parte alta del diagramma, chiamata "*cell state*". La *cell state* è una specie di nastro trasportatore che connette ingresso e uscita del blocco, con all'interno alcune interazioni lineari con l'output delle reti neurali contenute al suo interno. É facile che le informazioni percorrano l'intero blocco senza subire cambiamenti. La LSTM ha l'abilità di rimuovere o aggiungere informazione alla *cell state*, regolata da strutture chiamate "gates" (cancelli). I gates sono composti da una rete neurale sigmoidale e da operazioni di moltiplicazione. Il layer sigmoidale genera in output numeri tra zero e uno, descrivendo quanto di ogni componente lasciar passare. Un

valore di zero significa che non deve passare nulla, mentre un valore di uno significa che tutta l'informazione deve passare. Una LSTM ha tre di questi cancelli, che controllano il *cell state*.
É quindi grazie a questa struttura più complessa che le LSTM riescono ad apprendere informazioni meno recenti in maniera più efficace delle normali CNN.

3.3.2 Convolutional LSTM

Nell'ambito del riconoscimento delle immagini il tipo di rete neurale più efficace è la *Convolutional Neural Network*. Un modo per congiungere le CNN e le caratteristiche che rendono le LSTMs efficaci è quello di utilizzare una *Convolutioal LSTM*.
Le *Convolutional LSTM* (o ConvLSTM), sono delle reti neurali introdotte in [45], che sono strutturate come una normale LSTM, tranne per il fatto che le operazioni di prodotto al suo interno sono sostituite da operazioni di convoluzione. Come risultato, i dati che scorrono all'interno della ConvLSTM mantengono la loro dimensione di input al posto di essere trasformati in vettori 1D. Questo è molto importante quando vogliamo applicare le LSTM nel campo del riconoscimento delle immagini, perchè le immagini in input sono array tridimensionali, e non vettori con una singola dimensione.

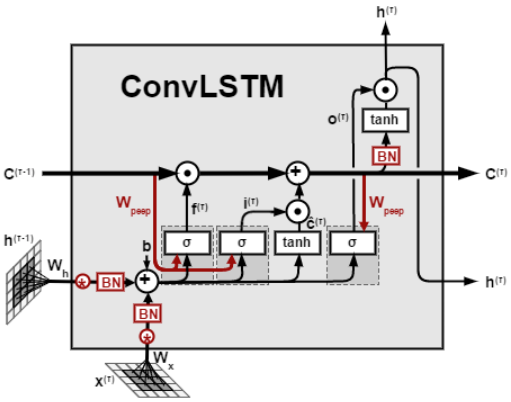

Figura 3.17: Diagramma della struttura di una ConvLSTM.

L'ordine delle immagini che vengono passate in input a una rete neurale che ha lo scopo di riconoscere azioni nei video è molto importante, in quanto passiamo alla rete sequenze di frame successivi tra loro. Le reti LSTM sfruttano e danno importanza all'ordine dell'input. L'idea alla base delle ConvLSTM è quindi quella di sfruttare questa caratteristica delle LSTM, insieme alla loro capacità di memorizzare informazioni, per lo scopo della *Video Classification*, nelle quali le CNN performano meglio.

Capitolo 4

Valutazione di Architetture basate su CNN

In questo capitolo verranno testate tre architetture basate su *reti neurali convoluzionali* che hanno lo scopo di riconoscere scene di violenza all'interno di video. Il fine è quello di verificare l'accuratezza delle architetture, utilizzando ogni volta diversi ottimizzatori per trovare quelli che, in questo contesto, ottengo i risultati migliori sui diversi dataset presi in considerazione.
Con ottimizzatori si intendono quegli algoritmi o metodi utilizzati all'interno di una rete neurale per cambiare gli attributi della rete stessa (per esempio i pesi) in modo tale da ridurre le perdite e aumentare l'accuratezza.
Verranno quindi testati per prima cosa i diversi modelli sul dataset dell'*AIRTLab*, utilizzando uno ad uno i diversi ottimizzatori forniti dalle librerie impiegate. Successivamente verranno testati i vari modelli su altri dataset molto famosi in letteratura, facendo uso degli ottimizzatori che hanno ottenuto i migliori risultati nei test precedenti.
Prima di testare questi modelli verrà fatta una panoramica sulle diverse tecnologie utilizzate in questo lavoro, insieme ai vari dataset presi in considerazione e a una descrizione dei tre modelli proposti.

4.1 Tecnologie Utilizzate

Tutte le architetture testate in questo capitolo e nei successivi sono state implementate nella piattaforma "*Google Colaboratory*", utilizzando il linguaggio di programmazione *Python* e le funzionalità messe a disposizione dalle librerie "*Tensorflow*", "*Keras*", "*Scikit-learn*" e "*OpenCV*".
Google Colaboratory [46], spesso abbreviata in "*Colab*", è una piattaforma cloud sviluppata da Google che permette di scrivere ed eseguire codice in *Python* attraverso l'utilizzo di blocchi note. I blocchi note di Colab eseguono il codice sui server cloud di Google, il che significa che l'utente può utilizzare la potenza dell'hardware Google, tra cui GPU e TPU, a prescindere dalla potenza della sua macchina. Nella sua versione gratuita, *Colab* fornisce all'utente circa 70GB di spazio su disco e fino a 32GB di RAM. I vantaggi che *Colab* offre sono numerosissimi, e il suo utilizzo si è rilevato

indispensabile per questo elaborato. Infatti, addestrare e testare modelli di reti neurali è un processo che solitamente occupa un elevato numero di risorse hardware, quali CPUs, GPUs e RAM. Per esempio, l'addestramento di alcune architetture ha occupato fino a 30GB di RAM e utilizzi prolungati delle risorse (fino a 8/10 ore di seguito). É facile notare come la maggior parte dei PC e laptop disponibili all'utenza non abbia le caratteristiche necessarie per completare addestramenti e test di architetture così pesanti, quindi senza l'utilizzo di strumenti come *Colab* questo lavoro non sarebbe possibile.
I blocchi note di *Colab* sono blocchi note Jupyter, o *Jupyter Notebooks*[47]. Un *Jupyter notebook* è una web application open-source che permette all'utente di creare e condividere documenti che contengono non solo codice eseguibile, ma anche testo, equazioni e visualizzazioni. Questi tipi di documenti sono molto popolari in quanto permettono di accompagnare il codice interattivo con descrizioni e osservazioni e sono molto facili da utilizzare.
Il linguaggio di programmazione utilizzato da *Colab* è il *Python*. Questa scelta è dovuta al fatto che *Python* è uno dei linguaggi di programmazione più popolari, soprattutto nella comunità del *machine learning*. Questa popolarità è dovuta al fatto che il *Python* è un linguaggio molto intuitivo e con una sintassi leggibile, inoltre ha una comunità di sviluppatori molto estesa ed ha un ampia gamma di frameworks e librerie molto utili, come:

- Keras, TensorFlow, e Scikit-learn per il machine learning
- NumPy per l'analisi dei dati e calcoli a performance elevate
- SciPy per calcoli avanzati
- Pandas per analisi di dati general-purpose
- Seaborn per la visualizzazione di dati
- ecc..

TensorFlow è una piattaforma open source per il *machine learning* fondata da Google [48] per rendere più facile ed efficiente lo sviluppo e l'addestramento di reti neurali. Ha un ecosistema molto flessibile di strumenti, librerie e risorse fornite dalla comunità che permette a ricercatori e programmatori di costruire ed addestrare reti neurali molto potenti. Anche se il linguaggio utilizzato da *TensorFlow* è il *Python*, tutte le operazioni matematiche vengono scritte come file binari *C++* ad alta efficienza.
Uno degli strumenti più utili forniti da *TensorFlow* è la *Keras API*. *Keras* è una API per il *deep learning* scritta in *Python* con l'obbiettivo di permettere sperimentazioni rapide ed efficienti nel campo del *machine learning*: *"avere la possibilità di andare da un'idea a un risultato il più velocemente possibile è la chiave di una buona ricerca"*[49]. *Keras* fornisce astrazioni essenziali e blocchi per sviluppare e fornire applicazioni di *machine learning* con alta velocità di iterazione. Le strutture dati principali di *Keras* sono modelli e layers. Il tipo di modello più semplice è quello sequenziale,

ovvero una pila lineare di layers. Per architetture più complesse si utilizzano i modelli funzionali, che permettono di costruire grafi di layers arbitrari, oppure di scrivere modelli interamente da zero tramite le sottoclassi.
Un'altra libreria molto utile è *Scikit-learn*[50], che contiene numerosi algoritmi utili per l'apprendimento automatico sviluppati in *Python*. Contiene, ad esempio, algoritmi per la classificazione, regressione, clustering, dimensionality reduction, model selection e preprocessamento.
Un'ultima libreria utilizzata è *OpenCV* (acronimo di "Open Source Computer Vision Library")[51], sviluppata in *C++* per la visione artificiale in tempo reale. Anche se scritta in *C++*, è possibile interfacciarcisi attraverso il *Python*.

4.2 Dataset Utilizzati

I vari modelli trattati in questo elaborato sono stati testati su tre diversi dataset creati appositamente per il problema della *violence detection*. Sono composti quindi da una serie di video, alcuni contenenti scene di violenza e altri invece contenenti situazioni normali o comunque non violente. Due di questi dataset, l'*Hockey Fight Dataset* e il *Crowd Violence Dataset* sono molto conosciuti in letteratura e sono stati utilizzati per testare la maggior parte delle soluzioni alla *violence detection* presenti nello stato dell'arte. Il terzo, l'*AIRTLab Violece Dataset*, è stato invece costruito dall'*AIRTLab* dell'*Università Politecnica delle Marche*.

4.2.1 Hockey Fights Dataset

L'*Hockey Fights Dataset* è stato introdotto in [12] ed è uno dei dataset più utilizzati nella disciplina della *violence detection*. É composto da 1000 video di azioni provenienti da partite di hockey su ghiaccio della National Hockey League (NHL). Il dataset è molto bilanciato, infatti 500 delle 1000 clips rappresentano scene violente mentre i restanti 500 raffigurano scene di gioco non violente. Ogni video consiste di un numero di frames che spazia tra 40 e 49 (la maggior parte dei video consistono di 41 frames) di 720x576 pixels (anche se spesso si usa una versione del dataset in cui i video hanno dimensione 320x240) ed è stato manualmente etichettato come violento (fight) o come non violento (no_fight) per identificarne la classe di appartenenza. Questo dataset offre uno strumento molto robusto per misurare le performance di una varietà di approcci per il riconoscimento della violenza, grazie alla grande quantità di video e a come sia le attività normali che quelle violente si verifichino entrambi in un contesto simile e dinamico.

4.2.2 Crowd Violence Dataset

Il *Crowd Violence Dataset*, in letteratura chiamato anche *Violent-Flows Dataset*, è stato introdotto in [17] e come l'*Hockey Fights* è uno dei dataset più utilizzati per addestrare e testare soluzioni alla *violence detection*.

É composto da un insieme di 246 video che rappresentano situazioni di affollamento, tipicamente folle di tifosi alle partite o comunque gruppi numerosi di persone ad eventi. Anche questo dataset è perfettamente bilanciato, con 123 video etichettati come violenti (fights) e i restanti 123 etichettati come non violenti (no_fights). Le clips hanno una durata variabile, con la più corta di 1.04 secondi e la più lunga di 6.52 secondi, e i frame di dimensione 320x240 pixels.
Tutti i video presenti nel dataset sono stati scaricati da YouTube e hanno una qualità piuttosto bassa, scelti apposta per simulare situazioni reali e difficili catturate da videocamere di sorveglianza.
Il *Crowd Violence Dataset* è stato quindi costruito con lo scopo di testare modelli di *machine learning* in situazioni affollate, che a differenza dell'*Hockey Fights Dataset* in cui i video rappresentano azioni svolte principalmente da due persone, queste sono molto più difficili da interpretare in quanto presentano azioni umane costanti, monotone e non vincolate spazialmente. Situazioni di questo tipo possono riempire i sistemi di *Computer vision* con grandissime quantità di informazioni sul movimento, rendendo i metodi che si basano sui punti di interesse troppo inefficienti, ed è quindi importante testare i vari modelli anche in dataset di questo tipo.

4.2.3 AIRTLab Violence Dataset

L'*AIRTLab Violence Dataset* è stato costruito dall'*AIRTLab* [52] dell'*UNIVPM* appositamente per il problema della *violence detection*.
Contiente 350 video etichettati come violenti (violent) o non violenti (non-violent), che rappresentano situazioni interpretate da attori che includono volutamente comportamenti (come abbracci, battiti di mani, ecc.) che possono causare falsi positivi nel compito di riconoscere azioni violente, per via dei movimenti molto veloci e di altre similarità con comportamenti violenti.
Le 350 clips sono dei video MP4 (codec H.264) con una lunghezza media di 5.63 secondi. Tutti i video sono composti da frames di risoluzione 1920x1080 pixels e un frame rate di 30 fps. A differenza degli altri due dataset presentati, l'*AIRTLab Dataset* non è perfettamente bilanciato, in quanto contiene 230 clips rappresentanti scene violente e 120 clips con comportamenti non violenti. Tutti i video sono inoltre stati registrati con due videocamere posizionate in due posizioni diverse (la fotocamera principale dell'Asus Zenfone Selfie ZD551KL e l'Action Cam TOPOP OD009B), costruendo così un dataset con 2 punti di vista.

4.3 Modelli Testati

In questo capitolo verranno testati tre diversi tipi di *reti neurali profonde*, basate su *3D Convolutional Neural Networks* e architetture *ConvLSTM*, precedentemente approfondite. In particolare:

1. Il primo modello utilizza C3D, un modello pre-addestrato di *3D CNN*, come estrattore di features e una *SVM* come classificatore.

2. Il secondo modello utilizza anch'esso C3D fino al primo layer fully-connected, aggiungendo due layers fully-connected e ottenendo così una rete end-to-end.

3. Il terzo modello è basato sull'architettura *ConvLSTM*, e è seguito da due layers fully-connected, ottenendo anche in questo caso una rete end-to-end.

4.3.1 Modello C3D + SVM

C3D è una *3D Convolutional Neural Network* [42] pre-addestrata sul dataset *Sports-1M*. Il dataset *Sports-1M* [53] è composto da più di un milione di video sportivi presi da YouTube. Le numerosissime clips sono state automaticamente etichettate con 487 diverse classi di sport, utilizzando le *YouTube Topics API*.
La caratteristica principale di C3D è l'utilizzo di filtri convoluzionali 3x3x3, seguiti da layers di pooling con filtri 2x2x2. La scelta è dovuta al fatto che in più casi è stato suggerito che l'utilizzo di kernel piccoli, come per esempio kernel 3x3, migliorino notevolmente le performance delle reti [54].
Più in particolare, la rete contiene 8 layers convoluzionali, 5 layers di pooling e 2 layers fully-connected, come mostrato nella figura 4.1

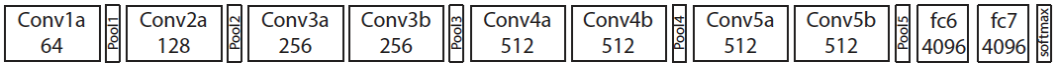

Figura 4.1: Struttura del modello C3D.

l'input della rete è di dimensione 3x16x112x112, dove 3 sono i canali dei colori, 16 la profondità temporale (ovvero il numero di frames presi in considerazione) e 112x112 sono le dimensioni dei frames del video in pixels. Il primo layer di pooling è di dimensione 1x2x2, a differenza degli altri che sono tutti 2x2x2, per preservare l'informazione temporale nel primo layer e costruire rappresentazioni a livelli più alti di queste informazioni nei layers successivi della rete. I layers fully-connected hanno dimensioni pari a 4096 e il layer softmax di output riflette le 487 classi del dataset *Sports-1M*. L'implementazione della rete C3D in *Keras* è mostrata nella figura 4.2.
In questo caso il modello C3D viene utilizzato solo come estrattore di features, quindi vengono presi in considerazione solo i layers fino a "*fc6*", il quale viene usato come descrittore dei 16 frames in input, mentre il layer fully-connected e il layer di softmax che si occupano della classificazione non vengono considerati in quanto verrà poi utilizzato un SVM come classificatore (figura 4.3). Vengono comunque riutilizzati i pesi originali di C3D, in quanto solamente il classificatore viene addestrato da zero con i tre dataset introdotti precedentemente.
Il classificatore utilizzato è preso dalla libreria *sklearn* ed è implementato nel seguente modo:

```
# C3D definition
from keras.models import Sequential, Model
from keras.layers import Input, Dense, Dropout, Flatten
from keras.layers.convolutional import Conv3D, MaxPooling3D, ZeroPadding3D

def create_C3D_model(summary = False):
    model = Sequential()
    input_shape = (16, 112, 112, 3)

    model.add(Conv3D(64, (3, 3, 3), activation='relu',
                     padding='same', name='conv1',
                     input_shape=input_shape))
    model.add(MaxPooling3D(pool_size=(1, 2, 2), strides=(1, 2, 2),
                           padding='valid', name='pool1'))
    # 2nd layer group
    model.add(Conv3D(128, (3, 3, 3), activation='relu',
                     padding='same', name='conv2'))
    model.add(MaxPooling3D(pool_size=(2, 2, 2), strides=(2, 2, 2),
                           padding='valid', name='pool2'))
    # 3rd layer group
    model.add(Conv3D(256, (3, 3, 3), activation='relu',
                     padding='same', name='conv3a'))
    model.add(Conv3D(256, (3, 3, 3), activation='relu',
                     padding='same', name='conv3b'))
    model.add(MaxPooling3D(pool_size=(2, 2, 2), strides=(2, 2, 2),
                           padding='valid', name='pool3'))
    # 4th layer group
    model.add(Conv3D(512, (3, 3, 3), activation='relu',
                     padding='same', name='conv4a'))
    model.add(Conv3D(512, (3, 3, 3), activation='relu',
                     padding='same', name='conv4b'))
    model.add(MaxPooling3D(pool_size=(2, 2, 2), strides=(2, 2, 2),
                           padding='valid', name='pool4'))
    # 5th layer group
    model.add(Conv3D(512, (3, 3, 3), activation='relu',
                     padding='same', name='conv5a'))
    model.add(Conv3D(512, (3, 3, 3), activation='relu',
                     padding='same', name='conv5b'))
    model.add(ZeroPadding3D(padding=((0, 0), (0, 1), (0, 1)), name='zeropad5'))
    model.add(MaxPooling3D(pool_size=(2, 2, 2), strides=(2, 2, 2),
                           padding='valid', name='pool5'))
    model.add(Flatten())
    # FC layers group
    model.add(Dense(4096, activation='relu', name='fc6'))
    model.add(Dropout(.5))
    model.add(Dense(4096, activation='relu', name='fc7'))
    model.add(Dropout(.5))
    model.add(Dense(487, activation='softmax', name='fc8'))

    if summary:
      print(model.summary())

    return model
```

Figura 4.2: Definizione di C3D in Keras.

```
def getFeatureExtractor(weigthsPath, layer = "fc6", verbose = False):
    model = create_C3D_model(verbose)
    model.load_weights(weigthsPath)
    model.compile(loss='mean_squared_error', optimizer='sgd')
    return Model(inputs=model.input,outputs=model.get_layer(layer).output)
```

Figura 4.3: Funzione per recuperare parte di C3D per l'estrazione delle features.

```
clf = svm.SVC(kernel='linear', C=1, probability=True)
```

La funzione "SVC" sta per *Support Vector Classification* e serve per istanziare un classificatore dalla famiglia delle *Support Vector Machines*. il classificatore utilizzato ha un kernel lineare. Il kernel di una SVM è un set di funzioni matematiche che ha lo scopo di prendere dei dati in input e trasformarli nella forma richiesta. Diversi algortimi SVM utilizzano diversi tipi di kernel. Queste funzioni possono esssere di diversi tipi, per esempio lineari, non lineari, polinomiali, sigmoidali, e altri ancora. Il kernel lineare è il tipo più semplice di kernel, e si è dimostrato la funzione migliore quando si ha un gran numero di features. I kernel lineari sono più veloci di altri tipi di kernel e la formula che utilizzano è del tipo:

$$F(x, x_j) = sum(x, x_j) \tag{4.1}$$

dove x e x_j rappresentano i dati che si sta cercando di classificare.

4.3.2 Modello C3D End-To-End

Con modello *End-To-End* si intende un modello adibito sia all'estrazione delle features che alla classificazione. Questo modello è molto simile al primo per il fatto che utilizza anch'esso C3D come base, lasciando fuori l'ultimo layer fully-connected che funge da classificatore. In questo caso, al posto di utilizzare una SVM per la classificazione vengono aggiunti al modello sprovvisto degli ultimi due layers, due nuovi layers fully-connected. Questa sostituzione viene fatta per trasformare il classificatore a 487 classi della C3D in un classificatore binario.

Come per il primo modello, anche in questo caso vengono riutilizzati i pesi originali di C3D già addestrati. Solamente i due layers fully-connected aggiunti vengono addestrati da zero sui dataset utilizzati.

Nella figura 4.4 è mostrata la definizione del modello, implementato attraverso la functional API di Keras.

Nella figura si nota come al modello di C3D, preso fino al primo layer fully-connected (fc6), vengono aggiunti i due layers FC di dimensioni 512 e 1 e che hanno come funzioni di attivazione rispettivamente *ReLU* e *Sigmoid*.

La funzione di attivazione ha lo scopo di definire l'output di un layer in base all'input che riceve. La funzione di attivazione *ReLU* (Rectified Linear Unit), per esempio, è

```
def getC3DCNNModel():
    pretrainedModel = getFeatureExtractor('drive/MyDrive/VDT/weights/weights.h5', 'fc6', False)
    for layer in pretrainedModel.layers:
        layer.trainable = False

    dropout1 = Dropout(.5)(pretrainedModel.output)
    fc7Alt = Dense(512, activation='relu', name='fc7-alt')(dropout1)
    dropout2 = Dropout(.5)(fc7Alt)
    output = Dense(1, activation='sigmoid')(dropout2)
    model = Model(inputs=pretrainedModel.inputs, outputs=output)
    model.summary()
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

    del pretrainedModel
    return model
```

Figura 4.4: Definizione del modello C3D End-To-End in Keras.

definita come la parte positiva del suo argomento:

$$f(x) = x^+ = max(0, x) \tag{4.2}$$

dove x è l'input del layer. La funzione *Sigmoid* è invece definita come:

$$S(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1} = 1 - S(-x) \tag{4.3}$$

e ha la caratteristica principale di trasformare l'input in un output compreso tra 0 e 1. Questo tipo di funzione è molto utile nel nostro caso in quanto in un problema di classificazione binario è utile ottenere un output che sia tra 0 e 1. Se l'output è più vicino all'uno allora sarà considerato di una certa classe, mentre se è più vicino allo zero all'altra. Praticamente la funzione *Sigmoid* è equivalente a un *softmax* tra due classi, dove il secondo elemento si assume che sia 0.

4.3.3 Modello ConvLSTM End-To-End

Il terzo modello testato è basato su un'architettura *ConvLSTM*, introdotta nel capitolo precedente. É composto da un layer *ConvLSTM*, seguito da due layers fully-connected, ottenendo come risultato una rete *End-To-End*. A differenza degli altri, l'intero modello viene addestrato da zero sui dataset utilizzati.
La definizione della rete è mostrata in figura 4.5, implementato attraverso la Sequential API di Keras.
Come si vede dalla figura, il layer *ConvLSTM* è composto da 64 filtri e utilizza un kernel 3x3. I due layers fully-connected sono molto simili a quelli utilizzati nel modello C3D *End-To-End*, con l'unica differenza che il primo ha dimensione 256 al posto di 512.
Di questo modello verrà testata anche una sua versione modificata, chiamata *BiConvLSTM* (Bidirectional ConvLSTM), nei vari dataset proposti. Nel capitolo precedente si è parlato di come le reti LSTM tengano conto non solo del presente, ma anche

```
def getLSTMModel():
    model = Sequential()
    model.add(ConvLSTM2D(filters=64, kernel_size=(3,3), input_shape=(16,112,112,3)))
    model.add(Dropout(0.5))
    model.add(Flatten())
    model.add(Dense(256, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(1, activation='sigmoid'))
    model.summary()
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model
```

Figura 4.5: Definizione del modello ConvLSTM End-To-End in Keras.

delle azioni passate. Una LSTM bidirezionale è una rete che oltre a tenere memoria del passato, tiene conto anche delle azioni future. *Keras* mette a disposizione un wrapper layer bidirezionale con il quale possiamo implementare questa architettura. La sua definizione è mostrata nella figura 4.6.

```
def getBiConvLSTMModel(optimizer='adam'):
    model = Sequential()
    model.add(Input(shape=(16,112,112,3)))
    model.add(Bidirectional(ConvLSTM2D(filters=64, kernel_size=(3,3))))
    model.add(Dropout(0.5))
    model.add(Flatten())
    model.add(Dense(256, activation='relu'))
    model.add(Dropout(0.5))
    model.add(Dense(1, activation='sigmoid'))
    model.summary()
    model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])

    return model
```

Figura 4.6: Definizione del modello BiConvLSTM End-To-End in Keras.

4.4 Test di Diversi Ottimizzatori sull'AIRTLab Dataset

In questa sezione verranno mostrati i risultati dei test sulle performance dei vari ottimizzatori che *Keras* mette a disposizione, per i tre modelli introdotti nella sezione precedente, sul dataset dell'*AIRTLab*. In particolare, vengono presi in considerazione gli ottimizzatori: *SGD*, *RMSprop*, *Adagrad*, *Adadelta*, *Adam*, *Adamax*, *Nadam*, *Ftrl*.

4.4.1 Setting Sperimentale

Prima di essere processati, i video passano attraverso una fase di preprocessing, nella quale vengono preparati ad essere passati come input alle varie reti neurali.
Per prima cosa i frames di ogni clip vengono ridimensionati in 112x112 pixels, in modo tale da rispettare le dimensioni richieste in input dai modelli. In seguito le varie clip vengono suddivise in gruppi (chunks) da 16 frames. Si è scelto di lavorare con

16 frames consecutivi perché considerati un numero rappresentativo per identificare un movimento. Questi chunks identificano la profondità temporale dell'input dei modelli. Infine ogni chunk viene etichettato come violento o non violento.

La modalità di test utilizzata è quella dello "*Stratified Shuffle Split Cross-Validation Scheme*". Nell'ambito del *Machine Learning*, uno schema di *cross-validation* è un metodo utilizzato nell'addestramento di una rete neurale che ha l'obbiettivo di limitare l'overfitting. Questo lo fa dividendo i dati disponibili in 2 set, uno per l'addestramento e l'altro per il test. Infatti far apprendere ad una rete i parametri di una funzione di predizione e poi testarli sugli stessi dati utilizzati per il learning è metodologicamente sbagliato: un modello che per esempio ripete semplicemente le etichette dei campioni che ha appena imparato otterrebbe un punteggio perfetto, quando in realtà fallirebbe nel predire qualsiasi dato che non ha mai visto.
Lo *Stratified Shuffle Split* è uno schema di *Cross-Validation* messo a disposizione dalla libreria *Sklearn*. Quello che fa è predisporre n suddivisioni del dataset in set per l'addestramento e set per il testing, scegliendo quali video vanno in quale set in maniera randomica. In questo modo i test effettuati saranno in realtà n, ognuno con un training set e un test set differente dal precedente. La funzione è stata impementata nel seguente modo:

```
cv = StratifiedShuffleSplit(n_splits=nsplits, train_size=0.8)
```

dove "n_splits" è il numero di suddivisoni del dataset in train e test, mentre "train_size" è la percentuale di clip utilizzate per l'addestramento. Un train_size di 0.8, ad esempio, significa che l'80% dei video presenti nel dataset verranno utilizzati per l'addestramento, mentre il restante 20% verrà utilizzato per il testing.

I parametri utilizzati per testare le performance dei modelli con i diversi ottimizzatori sono *Accuracy*, *Sensitivity*, *Specificity* e *F1-score*:

- L'*Accuracy* (accuratezza) è una metrica utilizzata per valutare modelli di classificazione che identifica, in maniera informale, la frazione delle predizioni che il modello ha indovinato. É definita come:

$$Accuracy = \frac{Numero\ di\ predizioni\ corrette}{Numero\ totale\ di\ predizioni} \tag{4.4}$$

  Nell'ambito della classificazione binaria l'accuratezza può essere calcolata anche in termini di positivi e negativi. Con positivi e negativi si intendono le due classi considerate (in questo elaborato i positivi saranno i chunks da 16 frames violenti, i negativi quelli non violenti). La formula in questo caso diventa:

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{4.5}$$

dove TP sta per *True Positive* (veri positivi), TN per *True Negative* (veri negativi), FP per *False Positive* (falsi positivi), FN per *False Negative* (falsi negativi). Un *True Positive* si ha quando il campione appartiene alla classe dei positivi e il modello predice correttamente la sua classe, mentre un *False Positive* si ha quando il campione appartiene alla classe dei negativi ma la predizione del modello è che il campione appartenesse ai positivi. Analogamente si definiscono I *True Negative* e i *False Negative*.

- La *Sensitivity* o "*Recall*" (sensibilità) misura la proporzione di positivi che sono stati correttamente identificati dal modello rispetto al numero totale di campioni che sono etichettati come positivi. Per questo motivo, la sensibilità viene anche chiamata *True Positive Rate*, ed è definita matematicamente come:

$$Sensitivity = \frac{TP}{TP + FN} \tag{4.6}$$

  La sensibilità è una misura di quanto bene un modello riesce ad identificare i campioni positivi.

- La *Specificity* (specificità) misura la proporzione di negativi che sono stati correttamente identificati dal modello rispetto al numero totale di campioni che sono etichettati come negativi. Per questo motivo, la specificità viene anche chiamata *True Negative Rate*, ed è definita matematicamente come:

$$Specificity = \frac{TN}{TN + FP} \tag{4.7}$$

  La specificità è una misura di quanto bene un modello riesce ad identificare i i campioni negativi.

- Nella classificazione binaria, l'*F1-score* è definito come la media armonica tra la *Precision* (precisione) e la *Sensitivity*:

$$F_1 = 2 \cdot \frac{precision \cdot sensitivity}{precision + sensitivity} \tag{4.8}$$

  La *Precision* (precisione) è il numero dei veri positivi ottenuti diviso per il numero di tutti i risultati positivi, inclusi quelli non identificati correttamente:

$$Precision = \frac{TP}{TP + FP} \tag{4.9}$$

  Il valore più alto possibile dell'*F1-Score* è 1.0, che indica una precisione e sensibilità perfetta, e il valore più basso possibile è 0, ottenuto se la precisione o la sensibilità sono uguali a zero. L'*F1-Score* può anche essere formulato come:

$$F_1 = \frac{TP}{TP + \frac{1}{2}(FP + FN)} \tag{4.10}$$

Quando si parla di testare metodi di classificazione è importante non basare le performance di un modello soltanto sull'accuratezza, perché in alcuni casi potrebbe non rappresentare la realtà, soprattutto in situazioni in cui i dataset non sono bilanciati.

I test mandati in esecuzione sono caratterizzati da un ciclo ripetuto un numero di volte pari al numero di ottimizzatori, all'interno del quale il modello viene addestrato e testato con uno specifico ottimizzatore un numero di volte pari al numero di splits dello *Stratified Shuffle Split* e infine i risultati vengono stampati a schermo come media dei risultati ottenuti dai vari splits, compresi della loro deviazione standard. Quando tutti i cicli sono terminati vengono stampati a schermo gli ottimizzatori che hanno ottenuto i migliori risultati per ogni parametro utilizzato.

4.4.2 Test sul Modello C3D + SVM

Per testare il primo modello sono stati eseguiti 5 splits dello *Stratified Shuffle Split*, questo significa che il test per ogni ottimizzatore è stato eseguito 5 volte.
I risultati dei test sono mostrati nella figura 4.7.
Come possiamo vedere dalla figura, la migliore accuratezza è stata raggiunta dall'ottimizzatore *RMSprop* (96%). Lo stesso ottimizzatore ha ottenuto i risultati migliori anche per quanto riguarda la specificità (94%) e l'f1-score (97%). L'ottimizzatore *Adam* ha invece ottenuto il risultato migliore per quanto riguarda la sensibilità (98%).
Possiamo quindi concludere che per quanto riguarda il modello C3D + SVM l'ottimizzatore che ha ottenuto i migliori risultati è l'*RMSprop*, anche se in merito alla capacità di riconoscere i positivi (i video violenti) viene superato, anche se di poco, da *Adam*.
Una considerazione da fare è che il modello, indipendentemente dall'ottimizzatore utilizzato, ha avuto più difficoltà nell'identificare correttamente i video non violenti, in quanto la specificità ha una percentuale più bassa rispetto agli altri parametri.

4.4.3 Test sul Modello C3D End-To-End

Per testare il secondo modello sono stati eseguiti 2 splits dello *Stratified Shuffle Split*, questo significa che il test per ogni ottimizzatore è stato eseguito 2 volte. In questo caso non sono stati eseguiti 5 splits come per il primo modello per problemi hardware, in quanto effettuare i test con 5 splits produceva sempre errori di OOM (Out Of Memory), dovuti al fatto che l'addestramento era troppo pesante per le risorse a disposizione per questo lavoro. Questo problema non si presenta nel primo modello perché in quel caso solo il classificatore viene addestrato da zero, e l'estrazione delle features viene eseguita in una sessione differente rispetto a quella di training e testing. Nei modelli *End-To-End* le due fasi vengono eseguite contemporaneamente e questo appesantisce il testing.
I risultati dei test sono mostrati nella figura 4.8.

```
Using optimizer:  SGD
Avg accuracy: 0.955367231638418 +/- 0.007969907333144562
Avg sensitivity: 0.9697478991596638 +/- 0.005247897477645694
Avg specificity: 0.9267241379310345 +/- 0.01766715649303382
Avg f1-score: 0.9671177739150558 +/- 0.006060211506545517
+------------------------------------------------------------------+
Using optimizer:  RMSprop
Avg accuracy: 0.9607344632768362 +/- 0.005747737273834983
Avg sensitivity: 0.9697478991596638 +/- 0.005413486859968521
Avg specificity: 0.9422413793103448 +/- 0.012669774531637151
Avg f1-score: 0.9707694552061668 +/- 0.004371664370264479
+------------------------------------------------------------------+
Using optimizer:  Adagrad
Avg accuracy: 0.9564971751412429 +/- 0.0073879643110859
Avg sensitivity: 0.9697478991596638 +/- 0.009070181993665053
Avg specificity: 0.9275862068965518 +/- 0.016219730795023087
Avg f1-score: 0.9673029224143963 +/- 0.005148230841801835
+------------------------------------------------------------------+
Using optimizer:  Adadelta
Avg accuracy: 0.9564971751412429 +/- 0.005463016837800507
Avg sensitivity: 0.9739495798319326 +/- 0.008038288432650807
Avg specificity: 0.925 +/- 0.01779290296599162
Avg f1-score: 0.9688622821463972 +/- 0.0051878161524767695
+------------------------------------------------------------------+
Using optimizer:  Adam
Avg accuracy: 0.9567796610169491 +/- 0.008688449999564788
Avg sensitivity: 0.9768907563025211 +/- 0.009672995322034738
Avg specificity: 0.9189655172413792 +/- 0.008791412954470328
Avg f1-score: 0.9689329296189018 +/- 0.006357579181632983
+------------------------------------------------------------------+
Using optimizer:  Adamax
Avg accuracy: 0.9596045197740114 +/- 0.006953973799486516
Avg sensitivity: 0.9714285714285715 +/- 0.007469070938920678
Avg specificity: 0.935344827586207 +/- 0.009829098492233953
Avg f1-score: 0.9699937503505376 +/- 0.005722834599145593
+------------------------------------------------------------------+
Using optimizer:  Nadam
Avg accuracy: 0.9533898305084746 +/- 0.003459731275117501
Avg sensitivity: 0.9697478991596638 +/- 0.006030546258154317
Avg specificity: 0.9172413793103449 +/- 0.016219730795023094
Avg f1-score: 0.9648879507090875 +/- 0.0029252694963940755
+------------------------------------------------------------------+
Using optimizer:  Ftrl
Avg accuracy: 0.9581920903954803 +/- 0.007398757544468221
Avg sensitivity: 0.9701680672268906 +/- 0.005852263982010114
Avg specificity: 0.9344827586206896 +/- 0.01689303270884951
Avg f1-score: 0.9691577142110382 +/- 0.005188933019631607
+------------------------------------------------------------------+
Final results:
Best accuracy was 0.9607344632768362 , from optimizer: RMSprop
Best sensitivity was 0.9768907563025211 , from optimizer: Adam
Best specificity was 0.9422413793103448 , from optimizer: RMSprop
Best f1-score was 0.9707694552061668 , from optimizer: RMSprop
```

Figura 4.7: Risultati del test degli ottimizzatori sul modello C3D + SVM.

Per quanto riguarda i modelli *End-To-End* viene preso in considerazione anche il parametro del *Loss* (errore).

Dalla figura possiamo vedere come l'ottimizzatore *Nadam* si sia comportato meglio degli altri per quanto riguarda accuratezza (96%), specificità (94%), f1-score (97%) e errore (0.10). La migliore sensibilità è stata invece ottenuta dall'ottimizzatore *RMSprop* (98%).

Possiamo quindi concludere che per quanto riguarda il modello C3D *End-To-End* l'ottimizzatore che ha ottenuto i risultati migliori è *Nadam*. Nella capacità di

```
Using optimizer:  RMSprop
Avg loss: 0.19271165132522583 +/- 0.022540763020515442
Avg accuracy: 0.9427966177463531 +/- 0.006355911493301392
Avg sensitivity: 0.9821428571428572 +/- 0.005252100840336171
Avg specificity: 0.8620689655172413 +/- 0.03017241379310348
Avg f1-score: 0.9585238200334205 +/- 0.0042090992212378
+--------------------------------------------------------------------+
Using optimizer:  SGD
Avg loss: 0.12710463628172874 +/- 0.009988967329263687
Avg accuracy: 0.9491525292396545 +/- 0.008474588394165039
Avg sensitivity: 0.9695378151260504 +/- 0.009453781512605008
Avg specificity: 0.9073275862068966 +/- 0.006465517241379337
Avg f1-score: 0.9624409572108317 +/- 0.0063740116041370865
+--------------------------------------------------------------------+
Using optimizer:  Adagrad
Avg loss: 0.17204611003398895 +/- 0.001992523670196533
Avg accuracy: 0.9385593235492706 +/- 0.004943519830703735
Avg sensitivity: 0.9590336134453781 +/- 0.009453781512605064
Avg specificity: 0.896551724137931 +/- 0.004310344827586188
Avg f1-score: 0.9544991485054577 +/- 0.003920809914500778
+--------------------------------------------------------------------+
Using optimizer:  Adadelta
Avg loss: 0.553386464715004 +/- 0.15223251283168793
Avg accuracy: 0.7422316372394562 +/- 0.07415255904197693
Avg sensitivity: 0.921218487394958 +/- 0.022058823529411742
Avg specificity: 0.375 +/- 0.27155172413793105
Avg f1-score: 0.8303701064684936 +/- 0.037784051746516556
+--------------------------------------------------------------------+
Using optimizer:  Adam
Avg loss: 0.12323061749339104 +/- 0.01506837084889412
Avg accuracy: 0.9512711763381958 +/- 0.016242921352386475
Avg sensitivity: 0.9653361344537815 +/- 0.024159663865546244
Avg specificity: 0.9224137931034483 +/- 0.0
Avg f1-score: 0.9636667656733001 +/- 0.012499037435508176
+--------------------------------------------------------------------+
Using optimizer:  Adamax
Avg loss: 0.14867732673883438 +/- 0.002149827778339386
Avg accuracy: 0.9413841664791107 +/- 0.0007062256336212158
Avg sensitivity: 0.9789915966386554 +/- 0.0
Avg specificity: 0.8642241379310345 +/- 0.0021551724137931494
Avg f1-score: 0.9573705658529791 +/- 0.0004917157503097069
+--------------------------------------------------------------------+
Using optimizer:  Nadam
Avg loss: 0.10271023958921432 +/- 0.0031403228640556335
Avg accuracy: 0.9597457647323608 +/- 0.0007061958312988281
Avg sensitivity: 0.9695378151260504 +/- 0.009453781512605008
Avg specificity: 0.9396551724137931 +/- 0.017241379310344807
Avg f1-score: 0.9700402085542594 +/- 0.0007931247790739504
+--------------------------------------------------------------------+
Using optimizer:  Ftrl
Avg loss: 0.14134794101119041 +/- 0.01738225296139717
Avg accuracy: 0.9484463334083557 +/- 0.009180784225463867
Avg sensitivity: 0.9779411764705883 +/- 0.009453781512605064
Avg specificity: 0.8879310344827587 +/- 0.008620689655172431
Avg f1-score: 0.9622562897281128 +/- 0.006815875220340828
+--------------------------------------------------------------------+
Final results:
Best losses was 0.10271023958921432 , from optimizer: Nadam
Best accuracy was 0.9597457647323608 , from optimizer: Nadam
Best sensitivity was 0.9821428571428572 , from optimizer: RMSprop
Best specificity was 0.9396551724137931 , from optimizer: Nadam
Best f1-score was 0.9700402085542594 , from optimizer: Nadam
```

Figura 4.8: Risultati del test degli ottimizzatori sul modello C3D End-To-End.

riconoscere i positivi viene però superato da *RMSprop*. In questo caso però, *RMSprop* si comporta decisamente peggio di *Nadam* nella capacità di riconoscere i negativi, in quanto ha ottenuto una specificità del 86%, contro il 94% ottenuto da *Nadam*.

Anche in questo caso, indipendentemente dall'ottimizzatore utilizzato, il modello ha avuto più difficoltà nell'identificare i video non violenti.

4.4.4 Test sul Modello ConvLSTM End-To-End

Per il terzo modello, come per il secondo, sono stati eseguiti solamente 2 splits dello *Stratified Shuffle Split*. Essendo però quest'ultimo modello il più pesante da addestrare, per via dell'architettura LSTM, si sono aggiunte ulteriori complicanze: indipendentemente dal numero di split effettuati, per gli ottimizzatori *adadelta*, *ftrl* e *nadam* non è stato possibile ottenere risultati poiché davano in ogni caso problemi di OOM.
I risultati dei test sono mostrati nella figura 4.9.

```
Using optimizer:  SGD
Avg loss: 0.043273004703223705 +/- 0.015765254385769367
Avg accuracy: 0.9823446571826935 +/- 0.0035310685634613037
Avg sensitivity: 0.9978991596638656 +/- 0.0
Avg specificity: 0.9504310344827587 +/- 0.01077586206896547
Avg f1-score: 0.9870196459412781 +/- 0.002563687392055203
+------------------------------------------------------------------+
Using optimizer:  RMSprop
Avg loss: 0.1560632735490799 +/- 0.016601547598838806
Avg accuracy: 0.9576271176338196 +/- 0.009886980056762695
Avg sensitivity: 0.9800420168067226 +/- 0.0031512605042016695
Avg specificity: 0.9116379310344828 +/- 0.023706896551724088
Avg f1-score: 0.9688770080001898 +/- 0.007139676045691212
+------------------------------------------------------------------+
Using optimizer:  Adagrad
Avg loss: 0.030245683155953884 +/- 0.0045871371403336525
Avg accuracy: 0.9922316372394562 +/- 0.002118617296218872
Avg sensitivity: 0.9978991596638656 +/- 0.0021008403361344463
Avg specificity: 0.9806034482758621 +/- 0.002155172413793094
Avg f1-score: 0.9942430283248264 +/- 0.0015728712567636416
+------------------------------------------------------------------+
Using optimizer:  Adam
Avg loss: 0.10411583259701729 +/- 0.03089570626616478
Avg accuracy: 0.9646892845630646 +/- 0.005649715662002563 5
Avg sensitivity: 0.9821428571428572 +/- 0.005252100840336171
Avg specificity: 0.9288793103448276 +/- 0.02801724137931033
Avg f1-score: 0.9739951102473199 +/- 0.003922870825235347
+------------------------------------------------------------------+
Using optimizer:  Adamax
Avg loss: 0.04047234542667866 +/- 0.01009221188724041
Avg accuracy: 0.9844632744789124 +/- 0.0014124512672424316
Avg sensitivity: 0.9884453781512605 +/- 0.0010504201680672232
Avg specificity: 0.9762931034482758 +/- 0.002155172413793094
Avg f1-score: 0.9884453781512605 +/- 0.0010504201680672232
+------------------------------------------------------------------+
Final results:
Best losses was 0.030245683155953884 , from optimizer: Adagrad
Best accuracy was 0.9922316372394562 , from optimizer: Adagrad
Best sensitivity was 0.9978991596638656 , from optimizer: Adagrad
Best specificity was 0.9806034482758621 , from optimizer: Adagrad
Best f1-score was 0.9942430283248264 , from optimizer: Adagrad
```

Figura 4.9: Risultati del test degli ottimizzatori sul modello ConvLSTM End-To-End.

Dalla figura si vede che l'ottimizzatore *Adagrad* ha ottenuto i risultati migliori su tutti i parametri utilizzati, con un'accuratezza del 99%, una sensibilità prossima al 100%, una specificità del 98%, un f1-score del 99% e un errore di 0.03.

Dai risultati ottenuti possiamo notare che il modello, con l'ottimizzatore *Adagrad* ha avuto delle performance eccellenti, molto vicine al 100%. Questo risultato potrebbe essere sia un bene che un male, perché solitamente quando un modello ottiene performance prossime al 100% significa che è vittima di *overfitting*, dovuto all'eccessivo adattamento della rete al dataset utilizzato.
Anche gli altri ottimizzatori hanno ottenuto comunque risultati ottimi, e come per gli altri modelli anche in questo caso c'è stata difficoltà nell'identificare correttamente i video non violenti.

4.4.5 Considerazioni Finali

In conclusione possiamo dire che non c'è un ottimizzatore specifico che si comporta meglio degli altri in ogni modello, per quanto riguarda il riconoscimento di violenza sull'*AIRTLab Violence Dataset*. Ogni modello ha ottenuto i migliori risultati con ottimizzatori diversi: *RMSprop* per il modello C3D + SVM; *Nadam* per il modello C3D *End-To-End*; *Adagrad* per il modello ConvLSTM *End-To-End*. Di seguito viene mostrata la tabella di comparazione delle performance dei vari modelli sul dataset dell'*AIRTLab*, utilizzando per ognuno l'ottimizzatore che ha ottenuto i migliori risultati:

Tabella 4.1: Migliori performance dei vari modelli sull'AIRTLab Violnece Dataset.

	Accuracy	Sensitivity	Specificity	F1-Score
C3D+SVM	96.07 ± 0.59%	96.97 ± 0.56%	94.22 ± 1.34%	97.07 ± 0.45%
C3D E2E	95.97 ± 0.07%	96.95 ± 0.97%	93.97 ± 1.83%	97 ± 0.08%
LSTM E2E	99.22 ± 0.21%	99.79 ± 0.21%	98.06 ± 0.22%	99.42 ± 0.16%

Come è stato possibile osservare nei vari test, tutti i modelli hanno ottenuto valori della specificità più bassi rispetto agli altri parametri. Questo significa che tutti i modelli hanno avuto più difficoltà nell'identificare correttamente i video non violenti rispetto ai video violenti. Questo risultato è giustificabile dal fatto che l'*AIRTLab Violence Dataset* è stato costruito proprio con lo scopo di ingannare i modelli con azioni ambigue come abbracci, battiti di mani e azioni rapide che possono essere facilmente scambiate per atti violenti. I risultati confermano semplicemente che questi tipi di azioni possono facilmente ingannare i modelli di *violence detection*.

4.5 Test dei Modelli sui Diversi Datasets

In questa sezione verranno mostrati i risultati delle performance dei vari modelli sugli altri dataset proposti (*Hockey Fights Dataset* e *Crowd Violence Dataset*), utilizzando per ognuno gli ottimizzatori che hanno ottenuto i migliori risultati nell'*AIRTLab Violence Dataset*. In particolare verranno testati: per il modello C3D + SVM gli ottimizzatori *SGD* e *RMSProp*; per il modello C3D *End-To-End* gli ottimizzatori

Adam e *Nadam*; per il modello ConvLSTM *End-To-End* gli ottimizzatori *Adam*, *Adagrad* e *Adamax*.
Verrà inclusa nel test anche la variante bidirezionale del modello ConvLSTM, valutato con gli stessi ottimizzatori che utilizza quest'ultimo.

4.5.1 Setting Sperimentale

I test sui vari dataset sono predisposti esattamente come quelli della sezione precedente. I video attraversano una fase di preprocessamento in cui vengono ridimensionati e suddivisi in chunks da 16 frames consecutivi. La modalità di test utilizzata è ancora quella dello *Stratified Shuffle Split* e i parametri utilizzati per la valutazione dei modelli sono *Accuracy*, *Sensitivity*, *Specificity* e *F1-Score*.

4.5.2 Test sul Modello C3D + SVM

Nei test effettuati sul primo modello sono stati eseguiti 5 splits dello *Stratified Shuffle Split*. Gli ottimizzatori utilizzati per la valutazione sono *SGD* e RMSProp.
I risultati dei test sull'*Hockey Fights Dataset* sono riportati in figura 4.10.

```
Using optimizer:  SGD
Avg accuracy: 0.9781094527363183 +/- 0.010122880571828745
Avg sensitivity: 0.9772277227722771 +/- 0.013135147684575852
Avg specificity: 0.9789999999999999 +/- 0.014966629547095779
Avg f1-score: 0.9782056702112095 +/- 0.010064807333650965

+------------------------------------------------------------------+

Using optimizer:  RMSprop
Avg accuracy: 0.9791044776119403 +/- 0.004613740545022741
Avg sensitivity: 0.9762376237623762 +/- 0.012286805590089968
Avg specificity: 0.9799999999999999 +/- 0.006324555320336764
Avg f1-score: 0.9781451004972354 +/- 0.005178996739645791

+------------------------------------------------------------------+

Final results:
Best accuracy was 0.9791044776119403 , from optimizer: RMSprop
Best sensitivity was 0.9772277227722771 , from optimizer: SGD
Best specificity was 0.9799999999999999 , from optimizer: RMSprop
Best f1-score was 0.9782056702112095 , from optimizer: SGD
```

Figura 4.10: Risultati del test del modello C3D + SVM sull'Hockey Fights Dataset.

Come possiamo vedere dalla figura l'ottimizzatore *RMSprop* ha ottenuto i migliori risultati sull'accuratezza (98%) e sulla specificità (98%), mentre l'ottimizzatore *SGD* ha ottenuto risultati migliori su sensibilità (98%) e f1-score(98%).
I due ottimizzatori si sono comportati in maniera praticamente identica, cono differenze tra i due minime e percentuali consistenti in tutti parametri.

I risultati dei test sul *Crowd Violence Dataset* sono riportati in figura 4.11.
L'ottimizzatore *SGD* ha ottenuto i migliori risultati in accuratezza (99%), specificità (99%) e f1-score (99%), mentre è stato superato da *RMSprop* per quanto riguarda la sensibilità (100%).

```
Using optimizer:  SGD
Avg accuracy: 0.9951020408163265 +/- 0.004759960730485967
Avg sensitivity: 0.9957746478873238 +/- 0.0034499855532157393
Avg specificity: 0.9922330097087378 +/- 0.011322236689019995
Avg f1-score: 0.9950876152549949 +/- 0.004737819470278075

+------------------------------------------------------------------+

Using optimizer:  RMSprop
Avg accuracy: 0.9942857142857143 +/- 0.007570300812649558
Avg sensitivity: 0.9985915492957746 +/- 0.0028169014084507005
Avg specificity: 0.9864077669902912 +/- 0.018007026204846018
Avg f1-score: 0.9944295698346947 +/- 0.006021403939521694

+------------------------------------------------------------------+

Final results:
Best accuracy was 0.9951020408163265 , from optimizer: SGD
Best sensitivity was 0.9985915492957746 , from optimizer: RMSprop
Best specificity was 0.9922330097087378 , from optimizer: SGD
Best f1-score was 0.9950876152549949 , from optimizer: SGD
```

Figura 4.11: Risultati del test del modello C3D + SVM sul Crowd Violence Dataset.

Anche in questo caso il modello ha ottenuto risultati molto consistenti e simili per entrambi gli ottimizzatori, con differenze tra i due minime.

4.5.3 Test sul Modello C3D End-To-End

Nei test effettuati sul secondo modello sono stati eseguiti 2 splits dello *Stratified Shuffle Split*. Gli ottimizzatori utilizzati per la valutazione sono *Adam* e *Nadam*.
I risultati dei test sull'*Hockey Fights Dataset* sono riportati in figura 4.12.

```
Using optimizer:  Adam
Avg loss: 0.13919412344694138 +/- 0.01230929046869278
Avg accuracy: 0.9689054787158966 +/- 0.003731340169906616
Avg sensitivity: 0.965 +/- 0.0050000000000000044
Avg specificity: 0.9727722772277227 +/- 0.012376237623762387
Avg f1-score: 0.9686630140437055 +/- 0.0034888846904717052

+------------------------------------------------------------------+

Using optimizer:  Nadam
Avg loss: 0.18552862852811813 +/- 0.04947080463171005
Avg accuracy: 0.9490049779415131 +/- 0.011194020509719849
Avg sensitivity: 0.98 +/- 0.0050000000000000044
Avg specificity: 0.9183168316831684 +/- 0.017326732673267342
Avg f1-score: 0.9503673229503379 +/- 0.010608286805759537

+------------------------------------------------------------------+

Final results:
Best losses was 0.13919412344694138 , from optimizer: Adam
Best accuracy was 0.9689054787158966 , from optimizer: Adam
Best sensitivity was 0.98 , from optimizer: Nadam
Best specificity was 0.9727722772277227 , from optimizer: Adam
Best f1-score was 0.9686630140437055 , from optimizer: Adam
```

Figura 4.12: Risultati del test del modello C3D End-To-End sul Hockey Fights Dataset.

L'ottimizzatore *Adam* ha ottenuto i migliori risultati riguardo errore (0.14), accu-

ratezza (97%), specificità (97%) e f1-score (97%), mentre l'ottimizzatore *Adam* ha ottenuto il risultato migliore sulla sensibilità (98%).
Il modello ha quindi performato meglio con l'ottimizzatore *Adam*, mentre con l'ottimizzatore *Nadam* ha avuto maggiori problemi soprattutto nell'identificare correttamente video non violenti.

I risultati dei test sull'*Crowd Violence Dataset* sono riportati in figura 4.13.

```
Using optimizer:  Adam
Avg loss: 0.10546950250864029 +/- 0.021902434527873993
Avg accuracy: 0.9716598987579346 +/- 0.004048585891723633
Avg sensitivity: 0.9711538461538461 +/- 0.009615384615384581
Avg specificity: 0.9720279720279721 +/- 0.013986013986014012
Avg f1-score: 0.9665689686755816 +/- 0.004304817732185384

+-----------------------------------------------------------------+

Using optimizer:  Nadam
Avg loss: 0.056731438264250755 +/- 0.015580693259835243
Avg accuracy: 0.9838056564331055 +/- 0.008097171783447266
Avg sensitivity: 0.9855769230769231 +/- 0.004807692307692346
Avg specificity: 0.9825174825174825 +/- 0.010489510489510523
Avg f1-score: 0.9809065934065934 +/- 0.009478021978022055

+-----------------------------------------------------------------+

Final results:
Best losses was 0.056731438264250755 , from optimizer: Nadam
Best accuracy was 0.9838056564331055 , from optimizer: Nadam
Best sensitivity was 0.9855769230769231 , from optimizer: Nadam
Best specificity was 0.9825174825174825 , from optimizer: Nadam
Best f1-score was 0.9809065934065934 , from optimizer: Nadam
```

Figura 4.13: Risultati del test del modello C3D End-To-End sul Crowd Violence Dataset.

Anche in questo caso l'ottimizzatore *Nadam* ha ottenuto risultati leggermente migliori di *Adam*, con un errore di 0.06, accuratezza del 98%, sensibilità del 98%, specificità del 98% e f1-score del 98%.

4.5.4 Test sul Modello ConvLSTM End-To-End

Nei test effettuati sul terzo modello è stato eseguito un unico split dello *Stratified Shuffle Split*, a causa della pesantezza della rete e dei problemi di *Out Of Memory* che l'addestramento del modello generava. Gli ottimizzatori utilizzati per la valutazione sono *Adam*, *Adagrad* e *Adamax*.
I risultati dei test sull'*Hockey Fights Dataset* sono riportati in figura 4.14.
Dalla figura si nota che l'ottimizzatore *Adagrad* è risultato migliore in accuratezza (98%), f1-score (98%), sensibilità (98.5%) e errore (0.09), mentre l'ottimizzatore *Adamax* ha ottenuto i migliori risultati in sensibilità (98.5%) e l'ottimizzatore *Adam* in specificità (98%).
L'ottimizzatore che si è comportato meglio in questo caso è quindi *Adagrad*, che viene superato solo in specificità da *Adam* di un piccolo margine.

```
Using optimizer:  Adam
Avg loss: 0.1164359301328659 +/- 0.0
Avg accuracy: 0.9701492786407471 +/- 0.0
Avg sensitivity: 0.96 +/- 0.0
Avg specificity: 0.9801980198019802 +/- 0.0
Avg f1-score: 0.9696969696969697 +/- 0.0
+------------------------------------------------------------------+
Using optimizer:  Adagrad
Avg loss: 0.08648539334535599 +/- 0.0
Avg accuracy: 0.9800994992256165 +/- 0.0
Avg sensitivity: 0.985 +/- 0.0
Avg specificity: 0.9752475247524752 +/- 0.0
Avg f1-score: 0.9800995024875623 +/- 0.0
+------------------------------------------------------------------+
Using optimizer:  Adamax
Avg loss: 0.10671838372945786 +/- 0.0
Avg accuracy: 0.9726368188858032 +/- 0.0
Avg sensitivity: 0.985 +/- 0.0
Avg specificity: 0.9603960396039604 +/- 0.0
Avg f1-score: 0.9728395061728395 +/- 0.0
+------------------------------------------------------------------+
Final results:
Best losses was 0.08648539334535599 , from optimizer: Adagrad
Best accuracy was 0.9800994992256165 , from optimizer: Adagrad
Best sensitivity was 0.985 , from optimizer: Adamax
Best specificity was 0.9801980198019802 , from optimizer: Adam
Best f1-score was 0.9800995024875623 , from optimizer: Adagrad
```

Figura 4.14: Risultati del test del modello ConvLSTM End-To-End sul Hockey Fights Dataset.

I risultati dei test sul *Crowd Violence Dataset* sono riportati in figura 4.15.

```
Using optimizer:  Adam
Avg loss: 0.26746034622192383 +/- 0.0
Avg accuracy: 0.8663967847824097 +/- 0.0
Avg sensitivity: 0.7307692307692307 +/- 0.0
Avg specificity: 0.965034965034965 +/- 0.0
Avg f1-score: 0.8216216216216217 +/- 0.0
+------------------------------------------------------------------+
Using optimizer:  Adagrad
Avg loss: 0.1568225473165512 +/- 0.0
Avg accuracy: 0.9433198571205139 +/- 0.0
Avg sensitivity: 0.9326923076923077 +/- 0.0
Avg specificity: 0.951048951048951 +/- 0.0
Avg f1-score: 0.9326923076923077 +/- 0.0
+------------------------------------------------------------------+
Using optimizer:  Adamax
Avg loss: 0.30162760615348816 +/- 0.0
Avg accuracy: 0.9473684430122375 +/- 0.0
Avg sensitivity: 0.9423076923076923 +/- 0.0
Avg specificity: 0.951048951048951 +/- 0.0
Avg f1-score: 0.937799043062201 +/- 0.0
+------------------------------------------------------------------+
Final results:
Best losses was 0.1568225473165512 , from optimizer: Adagrad
Best accuracy was 0.9473684430122375 , from optimizer: Adamax
Best sensitivity was 0.9423076923076923 , from optimizer: Adamax
Best specificity was 0.965034965034965 , from optimizer: Adam
Best f1-score was 0.937799043062201 , from optimizer: Adamax
```

Figura 4.15: Risultati del test del modello ConvLSTM End-To-End sul Crowd Violence Dataset.

In figura si nota che l'ottimizzatore *Adagrad* ha ottenuto il risultato migliore per quanto riguarda l'errore (0.16), l'ottimizzatore *Adamax* ha ottenuto i migliori risultati

in accuratezza (95%), sensibilità (94%) e f1-score (94%), e l'ottimizzatore Adam ha ottenuto i migliori risultati in specificità (96.5%).
I risultati ottenuti da questo modello nel dataset *Crowd Violence* sono piuttosto peggiori rispetto ai risultati ottenuti dagli altri modelli. Questo è quasi sicuramente dovuto ad un problema di overfitting, di cui abbiamo conferma osservando i risultati ottenuti dal modello in fase di addestramento, riportati in figura 4.16, che ne mostra le ultime 5 epoche:

```
Epoch 11/50
108/108 [==============================] - 80s 745ms/step - loss: 0.0100 - accuracy: 0.9997 - val_loss: 0.1294 - val_accuracy: 0.9512
Epoch 12/50
108/108 [==============================] - 79s 736ms/step - loss: 0.0068 - accuracy: 1.0000 - val_loss: 0.1494 - val_accuracy: 0.9512
Epoch 13/50
108/108 [==============================] - 79s 736ms/step - loss: 0.0075 - accuracy: 1.0000 - val_loss: 0.1436 - val_accuracy: 0.9512
Epoch 14/50
108/108 [==============================] - 79s 736ms/step - loss: 0.0054 - accuracy: 1.0000 - val_loss: 0.1463 - val_accuracy: 0.9512
Epoch 15/50
108/108 [==============================] - 79s 736ms/step - loss: 0.0050 - accuracy: 1.0000 - val_loss: 0.1541 - val_accuracy: 0.9512
Epoch 16/50
108/108 [==============================] - 79s 736ms/step - loss: 0.0068 - accuracy: 1.0000 - val_loss: 0.1495 - val_accuracy: 0.9593
```

Figura 4.16: Fase di addestramento del modello ConvLSTM End-To-End sul Crowd Violence Dataset.

Dalla figura si nota come nelle ultime epoche il modello riesca ad ottenere un'accuratezza del 100% e un errore prossimo allo 0, ma comunque ottiene risultati peggiori per quanto riguarda *val_loss* e *val_accuracy*. Questi ultimi due parametri identificano rispettivamente l'errore e l'accuratezza di validazione. Durante la fase di addestramento, solitamente parte del dataset viene separato dal dataset totale e viene utilizzato esclusivamente per la validazione del modello (nel caso di questi test il 12.5% del dataset è stato utilizzato per la validazione). Questo viene fatto appunto per evidenziare eventuali problemi di overfitting, infatti il nostro modello riesce ad ottenere risultati perfetti in fase di addestramento solamente perché si è adattato troppo ai campioni presi in input, mentre quando viene valutato su dati che non ha ancora visto perde in accuratezza.
Il fatto che questo modello sia stato vittima di overfitting mentre gli altri testati no è probabilmente dovuto alla quantità di parametri che contiene. Il modello *ConvLSTM End-To-End* ha un numero totale di parametri di 198,401,537, tutti addestrabili, mentre per esempio la rete *C3D End-To-End* ne contiene solo un terzo (63,312,641), di cui solamente 2,098,177 sono addestrabili.

4.5.5 Test sul Modello BiConvLSTM

Come per il modello *ConvLSTM End-To-End*, per questo modello è stato eseguito un unico split dello *Stratified Shuffle Split*. Inoltre, per via della pesantezza della rete (che con 396,802,561 parametri è il doppio più pesante della rete *ConvLSTM*), non è stato possibile effettuare test utilizzando 16 chunks come input alla rete, per via dei continui crash dovuti all'utilizzo di tutta la memoria. Sono quindi stati utilizzati chunks da 4 frames per i test sull' *Hockey Fights Dataset* e chunks da 8 frames per il *Crowd Violence Dataset*, in modo da ottenere comunque dei risultati. Il modello è stato anche testato sull'*AIRTLab Violence Dataset*, con 4 frames per chunk.

Ovviamente per questo motivo i risultati ottenuti da questi ultimi test non saranno confrontabili con quelli ottenuti dai test precedenti. Gli ottimizzatori utilizzati per la valutazione sono *Adam*, *Adagrad* e *Adamax*.
I risultati dei test sull' *Hockey Fights Dataset* sono riportati in figura 4.17.

```
Using optimizer:  Adam
Avg loss: 0.06637014448642731 +/- 0.0
Avg accuracy: 0.9700449109077454 +/- 0.0
Avg sensitivity: 0.979 +/- 0.0
Avg specificity: 0.9611166500498505 +/- 0.0
Avg f1-score: 0.9702675916749257 +/- 0.0

+------------------------------------------------------------------+

Using optimizer:  Adagrad
Avg loss: 0.014290853403508663 +/- 0.0
Avg accuracy: 0.9975037574768066 +/- 0.0
Avg sensitivity: 0.995 +/- 0.0
Avg specificity: 1.0 +/- 0.0
Avg f1-score: 0.9974937343358395 +/- 0.0

+------------------------------------------------------------------+

Using optimizer:  Adamax
Avg loss: 0.010318044573068619 +/- 0.0
Avg accuracy: 0.9965052604675293 +/- 0.0
Avg sensitivity: 0.999 +/- 0.0
Avg specificity: 0.9940179461615155 +/- 0.0
Avg f1-score: 0.9965087281795512 +/- 0.0

+------------------------------------------------------------------+

Final results:
Best losses was 0.010318044573068619 , from optimizer: Adamax
Best accuracy was 0.9975037574768066 , from optimizer: Adagrad
Best sensitivity was 0.999 , from optimizer: Adamax
Best specificity was 1.0 , from optimizer: Adagrad
Best f1-score was 0.9974937343358395 , from optimizer: Adagrad
```

Figura 4.17: Risultati del test del modello BiConvLSTM End-To-End sull' Hockey Fights Dataset.

L'ottimizzatore *Adamax* ha ottenuto i rsiultati migliori su sensibilità (100%) e errore (0.01), mentre l'ottimizzatore *Adagrad* ha ottenuto i risultati migliori su accuratezza (100%), specificità (100%), e f1-score (100%).
Si nota come il modello abbia ottenuto risultati praticamente perfetti su questo dataset. Questi risultati sono tuttavia probabilmente conseguenza di overfitting, in quanto il modello è eccessivamente complesso rispetto ai dati utilizzati, che sono stati anche ridotti a un quarto nella dimensione temporale. Infatti ridurre il numero di frames per chunk a 4 potrebbe essere un problema in quanto per esempio in un video violento potrebbero esserci sequenze di frames che comunque non contengano violenza. É piuttosto probabile che utilizzando chunks da 4 frames, alcuni di questi chunks non contenenti violenza siano comunque stati etichettati come violenti in fase di preprocessing, perché contenuti in un video violento. Il fatto che il modello sia comunque riuscito ad identificare correttamente questi chunks particolari dimostra che il modello si è adattato fortemente ai dati utilizzati in fase di addestramento.

I risultati dei test sull' *Hockey Fights Dataset* sono riportati in figura 4.18.

```
Using optimizer:  Adamax
Avg loss: 0.0614724550396204 +/- 0.0
Avg accuracy: 0.9825581312179565 +/- 0.0
Avg sensitivity: 0.9659090909090909 +/- 0.0
Avg specificity: 0.9949324324324325 +/- 0.0
Avg f1-score: 0.9792519444665658 +/- 0.0

+------------------------------------------------------------------+

Using optimizer:  Adam
Avg loss: 0.355296790599823 +/- 0.0
Avg accuracy: 0.8410852551460266 +/- 0.0
Avg sensitivity: 0.6636363636363636 +/- 0.0
Avg specificity: 0.972972972972973 +/- 0.0
Avg f1-score: 0.7806663806663806 +/- 0.0

+------------------------------------------------------------------+

Using optimizer:  Adagrad
Avg loss: 0.04650311544537544 +/- 0.0
Avg accuracy: 0.9825581312179565 +/- 0.0
Avg sensitivity: 0.9590909090909091 +/- 0.0
Avg specificity: 1.0 +/- 0.0
Avg f1-score: 0.9791183294663574 +/- 0.0

+------------------------------------------------------------------+

Final results:
Best losses was 0.04650311544537544 , from optimizer: Adagrad
Best accuracy was 0.9825581312179565 , from optimizer: Adagrad
Best sensitivity was 0.9659090909090909 , from optimizer: Adamax
Best specificity was 1.0 , from optimizer: Adagrad
Best f1-score was 0.9792519444665658 , from optimizer: Adamax
```

Figura 4.18: Risultati del test del modello BiConvLSTM End-To-End sul Crowd Violence Dataset.

L'ottimizzatore *Adagrad* ha ottenuto i risultati migliori su accuratezza (98%), specificità (100%), e errore (0.05), mentre l'ottimizzatore *Adamax* su sensibilità (96.5%) e f1-score (89%).
In questo caso i risultati sono più attendibili rispetto ai precedenti in quanto i chunks utilizzati sono composti ognuno da 8 frames.

I risultati dei test sull' *AIRTLab Violence Dataset* sono riportati in figura 4.19.
Come si può osservare dalla figura l'ottimizzatore *Adagrad* ha ottenuto risultati perfetti su tutti i parametri utilizzati per il test. Anche in questo caso siamo sicuramente di fronte ad un problema di overfitting, anche perchè come per il dataset *Hockey Fights* i chunks utilizzati sono composti da 4 frames.

4.5.6 Considerazioni Finali

Di seguito sono riportate le tabelle di comparazione delle performance dei vari modelli sui dataset presi in considerazione, utilizzando per ognuno l'ottimizzatore che ha ottenuto i migliori risultati.
Per l'*Hockey Fights Dataset* i risultati sono evidenziati nella tabella 4.2.

```
Using optimizer:  Adamax
Avg loss: 0.002502481685951352 +/- 0.0
Avg accuracy: 0.9993174076080322 +/- 0.0
Avg sensitivity: 0.998960498960499 +/- 0.0
Avg specificity: 0.9994918699186992 +/- 0.0
Avg f1-score: 0.998960498960499 +/- 0.0

+------------------------------------------------------------------+

Using optimizer:  Adam
Avg loss: 0.6330668330192566 +/- 0.0
Avg accuracy: 0.6716723442077637 +/- 0.0
Avg sensitivity: 0.0 +/- 0.0
Avg specificity: 1.0 +/- 0.0
Avg f1-score: 0.0 +/- 0.0

+------------------------------------------------------------------+

Using optimizer:  Adagrad
Avg loss: 0.0009006300242617726 +/- 0.0
Avg accuracy: 1.0 +/- 0.0
Avg sensitivity: 1.0 +/- 0.0
Avg specificity: 1.0 +/- 0.0
Avg f1-score: 1.0 +/- 0.0

+------------------------------------------------------------------+

Final results:
Best losses was 0.0009006300242617726 , from optimizer: Adagrad
Best accuracy was 1.0 , from optimizer: Adagrad
Best sensitivity was 1.0 , from optimizer: Adagrad
Best specificity was 1.0 , from optimizer: Adagrad
Best f1-score was 1.0 , from optimizer: Adagrad
```

Figura 4.19: Risultati del test del modello BiConvLSTM End-To-End sull'AIRTLAB Violence Dataset.

Tabella 4.2: Tabella di comparazione delle performance ottenute dai modelli nell'Hockey Fights Dataset.

	Accuracy	Sensitivity	Specificity	F1-Score
C3D+SVM	97.91 ± 0.47%	97.62 ± 1.26%	98 ± 0.64%	97.81 ± 0.53%
C3D E2E	96.89 ± 0.38%	96.50 ± 0.52%	97.28 ± 1.27%	96.87 ± 0.36%
LSTM E2E	98.01 ± 0.00%	98.50 ± 0.00%	97.52 ± 0.00%	98.01 ± 0.00%
BILSTM	99.75 ± 0.00%	99.50 ± 0.00%	100 ± 0.00%	99.75 ± 0.00%

Come possiamo vedere dalla tabella il modello che ha ottenuto i risultati complessivamente migliori è il modello *BiConvLSTM End-To-End*. Come analizzato prima però, i risultati ottenuti da questo modello non sono comparabili a quelli ottenuti dagli altri. Escludendo quest'ultimo, il modello *LSTM End-To-End* ha avuto le migliori performance.

Per il *Crowd Violence Dataset* i risultati sono evidenziati nella tabella 4.3.
In questo caso il modello che ha ottenuto i migliori risultati è *C3D + SVM*.

Per l'*AIRTLab Violence Dataset* i risultati finali contenenti anche le performance del modello *BiConvLSTM End-To-End* sono evidenziati nella tabella 4.4.

Tabella 4.3: Tabella di comparazione delle performance ottenute dai modelli nel Crowd Violence Dataset.

	Accuracy	Sensitivity	Specificity	F1-Score
C3D+SVM	99.51 ± 0.48%	99.58 ± 0.34%	99.22 ± 1.14%	99.51 ± 0.48%
C3D E2E	98.38 ± 0.82%	98.56 ± 0.49%	98.25 ± 1.07%	98.09 ± 0.97%
LSTM E2E	94.33 ± 0.00%	93.27 ± 0.00%	95.10 ± 0.00%	93.27 ± 0.00%
BILSTM	98.26 ± 0.00%	96.59 ± 0.00%	99.49 ± 0.00%	97.93 ± 0.00%

Tabella 4.4: Tabella di comparazione delle performance ottenute dai modelli nell'AIRTLab Violence Dataset.

	Accuracy	Sensitivity	Specificity	F1-Score
C3D+SVM	96.07 ± 0.59%	96.97 ± 0.56%	94.22 ± 1.34%	97.07 ± 0.45%
C3D E2E	95.97 ± 0.07%	96.95 ± 0.97%	93.97 ± 1.83%	97 ± 0.08%
LSTM E2E	99.22 ± 0.21%	99.79 ± 0.21%	98.06 ± 0.22%	99.42 ± 0.16%
BILSTM	100 ± 0.00%	100 ± 0.00%	100 ± 0.00%	100 ± 0.00%

Escludendo l'ultimo modello, il modello *LSTM End-To-End* ha ottenuto i risultati migliori.

Infine nella tabella 4.5 sono riportati i risultati sull'accuratezza ottenuta da alcuni modelli presenti nello stato dell'arte sui dataset *Hockey Fights* e *Crowd Violence*, insieme a quelli ottenuti dai modelli testati in questo elaborato:

Tabella 4.5: Tabella di comparazione delle performance ottenute da vari modelli dello stato dell'arte sui dataset Hockey Fights e Crowd Violence.

Metodo	Hockey	Crowd
MoSIFT+HIK[12]	90.9%	89.5%
ViF[17] & 82.9 ± 0.14%	-	
MoSIFT+KDE+Sparse Coding[13]	94.3 ± 1.68%	-
Deniz et al.[31]	90.1 ± 0%	98.0 ± 0.22%
Gracia et al.[22]	82.4 ± 0.4%	97.8 ± 0.4%
Substantial Derivative et al.[55]	-	96.89 ± 0.21%
Bilinski et al.[32]	93.4%	99%
MoIWLD[56]	96.8 ± 1.04%	-
ViF+OViF[18]	87.5 ± 1.7%	-
Three streams + LSTM[57]	93.9%	-
C3D+SVM	**97.91 ± 0.47%**	**99.51 ± 0.48%**
C3D E2E	**96.89 ± 0.38%**	**98.38 ± 0.82%**
LSTM E2E	**99.01 ± 0.00%**	**94.33 ± 0.00%**
BILSTM	**99.75 ± 0.00%**	**98.26 ± 0.00%**

Capitolo 5

Implementazione di un Architettura Basata su BiConvLSTM

In questo capitolo verrà mostrata l'implementazione in *Keras* e la valutazione di un architettura per la *Violence Detection* presente nello stato dell'arte e basata su *Reti Neurali Convoluzionali Bidirezionali LSTM*.
In una prima sezione introduttiva si parlerà dell'architettura implementata, descrivendo le particolari tecnologie che la compongono e introducendo le due diverse realizzazioni della stessa (una completa e una semplificata). Successivamente verrà mostrata l'implementazione delle due varianti dell'architettura in *keras* su *Google Colab*, descrivendo l'approccio utilizzato. Infine l'architettura verrà testata sui tre dataset introdotti nel capitolo precedente per valutarne le performance e confrontarle con quelle ottenute dai creatori del modello.

5.1 Descrizione dell'Architettura

Il modello preso in considerazione è stato introdotto da Alex Hanson, Koutilya PNVR, Sanjukta Krishnagopal, e Larry Davis nell'articolo [58].
L'idea generale è quella di riuscire a classificare in modo corretto scene di violenza generando una codifica robusta dei video da passare a un classificatore fully-connected. Questa rappresentazione viene prodotta tramite un encoder spazio-temporale che estrae features da un video che corrispondono a dettagli sia spaziali che temporali. Il punto forte di questa architettura è che la codifica viene fatta in entrambi le direzioni temporali, in modo tale da avere anche accesso a informazioni future dallo stato corrente.
Nell'articolo viene studiata anche una versione semplificata dell'architettura che codifica soltanto le informazioni spaziali tramite un encoder semplificato.

5.1.1 Architettura dell'Encoder Spazio-Temporale

Una rappresentazione dell'architettura dell'encoder spazio-temporale è mostrata nella figura 5.1.
L'encoder è composto da tre parti principali: un encoder spaziale che estrae le caratteristiche spaziali per ogni frame del video; un encoder temporale che permette

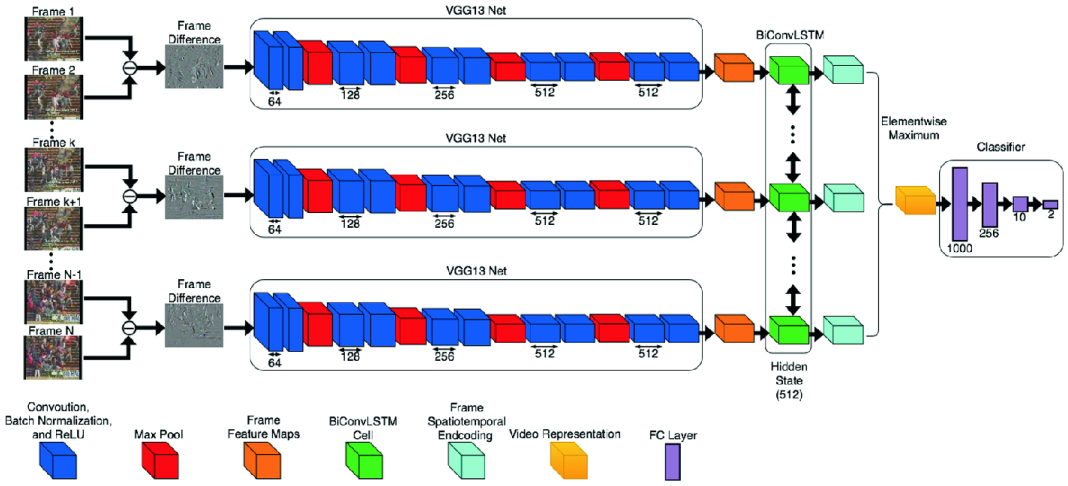

Figura 5.1: Architettura dell'encoder spazio-temporale

di "mescolare" temporalmente le caratteristiche spaziali estratte precedentemente producendo così una codifica spazio-temporale per ogni step temporale; un classificatore fully-connected.

L'encoder spaziale è stato realizzato tramite la rete neurale convoluzionale *VGG-13*. Questa rete fa parte della famiglia di *CNNs* chiamata *VGG*, sviluppata e addestrata dal *Visual Geometry Group* (VGG) di Oxford[1] per il task dell'*object recognition.*. Il "13" sta ad indicare la versione di *VGG* a 13 layers. L'architettura della rete *VGG-13* è rappresentata nella figura 5.2

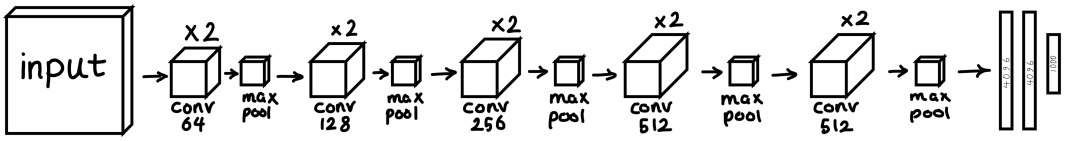

Figura 5.2: Architettura di VGG-13

L'ultimo layer di max pooling e i layers fully-connected della rete sono stati rimossi, producendo quindi in output delle feature maps spaziali per ogni frame di dimensioni 14 x 14 x 512. Al posto di passare alla rete i frames del video direttamente, nell'articolo i frames adiacenti vengono prima sottratti e la loro differenza usata come input dell'encoder spaziale. Utilizzare la differenza tra frames adiacenti come input forza la rete neurale a codificare i cambiamenti che occorrono nel video [59].

L'encoder temporale è stato realizzato tramite una *Bidirectional Convolutional LSTM* (*BiConvLSTM*). L'input della rete sono le feature maps ottenute dall'encoder spaziale. La *BiConvLSTM* è stata costruita in modo che l'output di ogni cella sia anch'esso di dimensione 14 x 14 x 512. Agli output dell'encoder temporale viene poi applicata un'operazione di *elementwise maximum*, che produce come risultato

[1] Visual Geometry Group è un gruppo accademico dell'università di Oxford incentrato nella computer vision focalizzato nella computer vision

una rappresentazione finale del video completo di dimensioni 14 x 14 x 512. Questa operazione prende in input una lista di tensori[2] e gli applica un'operazione di massimo, producendo un output con le stesse dimensioni dei tensori in input.
Il classificatore fully-connected utilizzato è composto da 4 layers. Il numero di nodi per layer è di 1000, 256, 10 e 2. Ogni layer utilizza la funzione della tangente iperbolica come non linearità (tanh), la cui formula è la seguente:

$$tanh(z) = \frac{sinh(z)}{cosh(z)} = \frac{e^z - e^{-z}}{e^z + e^- z} \tag{5.1}$$

Il classificatore prende come input il risultato dell'operazione di massimo e produce in output una predizione binaria della classe di appartenenza del video (violento o non).

5.1.2 Architettura dell'Encoder Spaziale

Una rappresentazione dell'architettura dell'encoder spaziale è mostrata nella figura 5.3.

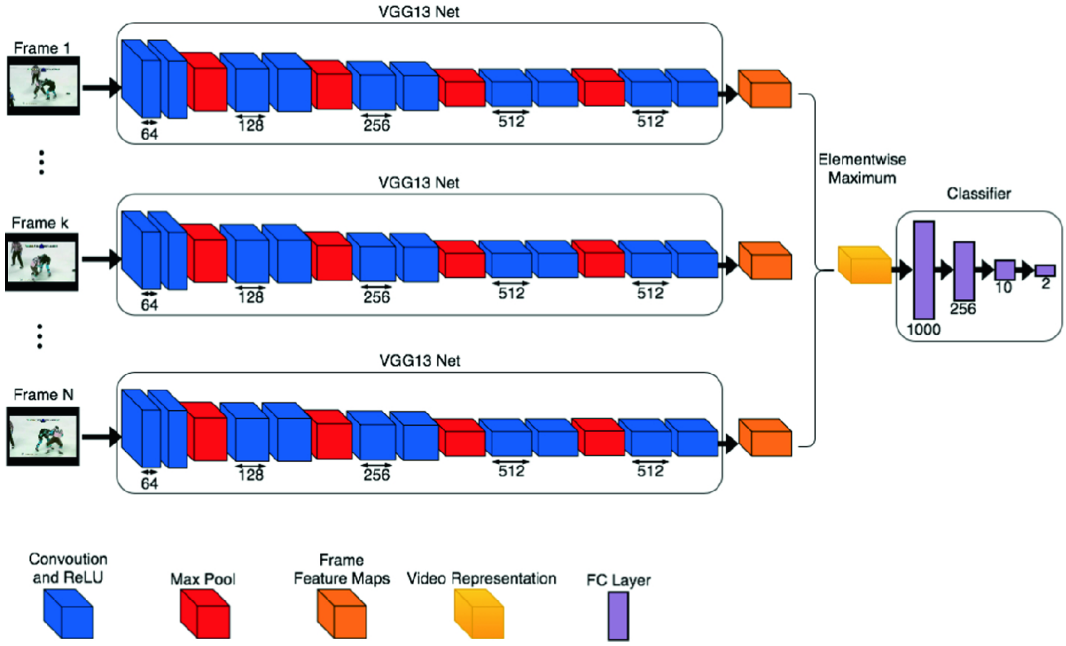

Figura 5.3: Architettura dell'encoder spaziale

L'encoder spaziale è una versione semplificata dell'encoder spazio-temporale. Come si vede dalla figura, l'encoder temporale realizzato tramite *BiConvLSTM* è stato rimosso e l'operazione di *elementwise maximum* viene applicata direttamente alle features spaziali ottenute come output della rete *VGG-13*. Inoltre, siccome questa versione semplificata dell'architettura si interessa solamente delle features spaziali, la

[2] Un tensore è la generalizzazione di vettori e matrici e può essere interpretato come array multidimensionale.

differenza tra frames adiacenti non viene effettuata e i singoli frames vengono passati direttamente come input dell'encoder spaziale.

5.1.3 Metodologia di Addestramento

Per quanto riguarda l'encoder spaziale costruito tramite il modello *VGG-13*, sono stati riutilizzati i pesi pre-addestrati di *VGG-13* nel dataset *ImageNet*. *ImageNet* è un dataset proposto in [60] composto da circa 3.2 milioni di immagini appartenenti a diverse categorie, quali: mammifero, uccello, pesce, rettile, anfibio, veicolo, arredamento, strumento musicale, formazione geologica, utensile, fiore, frutta. I pesi dell'encoder temporale e del classificatore sono invece stati inizializzati in maniera randomica, e addestrati direttamente nei dataset presi in considerazione, ovvero: *Hockey Fights Dataset*, *Violent Flows Dataset* e *Movies Fights Dataset*. Il *Movies Fights Dataset* [12] è composto di una serie di sequenze violente prese da vari film. Le scene non violente sono invece prese da datasets di action recognition disponibili pubblicamente. Il dataset è composto da un totale di 200 video, di cui 100 violenti e 100 non violenti. A differenza dell'*Hockey Fights Dataset*, i video del *Movies Fights Dataset* sono sostanzialmente differenti nel loro contenuto.
Per l'encoder spazio-temporale sono state prese come input le differenze tra frames adiacenti e sono state normalizzate per ricadere nel range tra 0 e 1. Per entrambi le architetture, il *learning rate* utilizzato è di 10^{-6}, il *batch size* è di 8 video e il *weight decay* è settato a 0.1. L'ottimizzatore utilizzato è *Adam*. I frames sono stati selezionati ad intervalli regolari e ridimensionati a 224 x 224 pixels. Inoltre sono state applicate tecniche di *data augmentation* ai dataset, quali *random cropping* (RC) e *random horizontal flipping* (RHF). Con *data augmentation* si intendono quelle tecniche utilizzate per incrementare il numero di dati di un dataset aggiungendo copie leggermente modificate di dati già esistenti. Per esempio, il *random cropping* è una tecnica che consiste nel creare un "subset" (una porzione) randomico di un'immagine. Questo aiuta il modello a generalizzare meglio poiché l'oggetto di interesse che vogliamo far imparare ai nostri modelli non sempre è completamente visibile nell'immagine analizzata. Il *random horizontal flipping* invece è una tecnica che capovolge un'immagine orizzontalmente data una certa probabilità.
La funzione di errore utilizzata è la *cross entropy* (entropia incrociata), la cui formula è la seguente:

$$H(p, q) = -\sum_{x \in X} p(x) \log q(x) \tag{5.2}$$

dove q è la distribuzione di probabilità stimata, mentre p è la distribuzione di probabilità attesa.
Infine è stata utilizzata la tecnica di validazione *K-Fold Cross-Validation* con 5 splits. Questo approccio consiste nel "mescolare" il dataset in maniera randomica e poi suddividerlo in K gruppi. Successivamente per ogni gruppo creato il modello verrà

addestrato sui restanti $K-1$ gruppi(che verranno utilizzati come training set) e verrà poi valutato su di esso (che verrà utilizzato come test set).

5.2 Implementazione in Keras

In questa sezione verrà mostrata l'implementazione dell'architettura nelle sue due varianti in *Keras*. Una considerazione da fare è che gli autori dell'articolo [58] sono stati un poò vaghi su alcuni aspetti descrittivi del modello, trascurando i dettagli implementativi e incentrandosi di più sui risultati ottenuti. Per questo alcuni dei metodi utilizzati per l'implementazione in *Keras* potrebbero non essere gli stessi utilizzati dagli autori.
Entrambi le varianti dell'architettura sono state implementate utilizzando la *Keras Functional API*, che permette di costruire modelli più flessibili rispetto alla *Sequential API* e soprattutto di lavorare con input multipli e modelli innestati più facilmente.

5.2.1 Implementazione dell'Encoder Spazio-Temporale

Per quanto riguarda l'implementazione dell'encoder spaziale, non è stato possibile utilizzare il modello *VGG-13* poichè in rete non erano reperibili i pesi pre-addestrati nel dataset *ImageNet*. É stato invece utilizzato il modello *VGG-16* come sostituto, variante a 16 layers di *VGG-13*, i cui pesi pre-addestrati erano facilmente reperibili. L'implementazione in Keras del modello VGG-16 è stata effettuata come mostrato in figura 5.4.

```
def get_VGG16_model():
  #prepare model for TPU acceleration
  resolver = tf.distribute.cluster_resolver.TPUClusterResolver(tpu='grpc://' + os.environ['COLAB_TPU_ADDR'])
  tf.config.experimental_connect_to_cluster(resolver)
  tf.tpu.experimental.initialize_tpu_system(resolver)
  strategy = tf.distribute.TPUStrategy(resolver)
  with strategy.scope():
    inputs = keras.Input(shape = (224,224,3))
    x = layers.Conv2D(filters=64,kernel_size=(3,3),padding="same", activation="relu", trainable = False)(inputs)
    x = layers.Conv2D(filters=64,kernel_size=(3,3),padding="same", activation="relu", trainable = False)(x)
    x = layers.MaxPool2D(pool_size=(2,2),strides=(2,2))(x)
    x = layers.Conv2D(filters=128, kernel_size=(3,3), padding="same", activation="relu", trainable = False)(x)
    x = layers.Conv2D(filters=128, kernel_size=(3,3), padding="same", activation="relu", trainable = False)(x)
    x = layers.MaxPool2D(pool_size=(2,2),strides=(2,2))(x)
    x = layers.Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu", trainable = False)(x)
    x = layers.Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu", trainable = False)(x)
    x = layers.Conv2D(filters=256, kernel_size=(3,3), padding="same", activation="relu", trainable = False)(x)
    x = layers.MaxPool2D(pool_size=(2,2),strides=(2,2))(x)
    x = layers.Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu", trainable = False)(x)
    x = layers.Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu", trainable = False)(x)
    x = layers.Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu", trainable = False)(x)
    x = layers.MaxPool2D(pool_size=(2,2),strides=(2,2))(x)
    x = layers.Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu", trainable = False)(x)
    x = layers.Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu", trainable = False)(x)
    outputs = layers.Conv2D(filters=512, kernel_size=(3,3), padding="same", activation="relu", trainable = False)(x)
    model = keras.Model(inputs=inputs, outputs=outputs)
    model.summary()
    model.load_weights("drive/MyDrive/VDT/weights/vgg16_weights.h5") #loads ImageNet vgg16 pre-trained weights
    model.compile(loss='mean_squared_error', optimizer='Adam')
  return model
```

Figura 5.4: Implementazione di VGG-16 in Keras.

Come anche descritto nell'articolo, l'ultimo layer di max pooling e i layers fully-connected sono stati rimossi e i layers resi non addestrabili. Prima della costruzione, si prepara il modello ad essere eseguito tramite la TPU (Tensor Processing Unit). La TPU è un'unità elaborativa creata da Google appositamente per applicazioni specifiche nel campo del *Machine Learning* e velocizza drasticamente il processo di addestramento ed esecuzione delle reti neurali che ne fanno uso.

La rete convoluzionale bidirezionale LSTM che realizza l'encoder temporale è stata invece implementata come descritto nella figura 5.5

```
def get_BiConvLSTM_model():
  inputs = keras.Input(shape = (1,14,14,512))
  outputs = keras.layers.Bidirectional(keras.layers.ConvLSTM2D(filters=256, kernel_size=(1,1), activity_regularizer=l2(0.001)))(inputs)
  model = keras.Model(inputs=inputs, outputs=outputs)
  optimizer = keras.optimizers.Adam(lr=0.000001)
  model.summary()
  #model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
  return model
```

Figura 5.5: Implementazione della rete BiConvLSTM in Keras.

Il modello è stato definito tramite un unico layer "ConvLSTM2D" racchiuso in un layer "Bidirectional". Al modello è stato aggiunto un "activity regularizer", che tramite l'aggiunta di "sanzioni" all'output tende a ridurre l'overfitting.

Il classificatore fully-connected è stato definito come mostrato in figura 5.6.

```
def get_classifier():
    inputs = keras.Input(shape=(14,14,512))
    x = keras.layers.Flatten()(inputs)
    x = layers.Dense(1000, activation = 'tanh', activity_regularizer=l2(0.001))(x)
    x = keras.layers.Dropout(0.4)(x)
    x = layers.Dense(256, activation = 'tanh', activity_regularizer=l2(0.001))(x)
    x = keras.layers.Dropout(0.4)(x)
    x = layers.Dense(10, activation = 'tanh', activity_regularizer=l2(0.001))(x)
    x = keras.layers.Dropout(0.4)(x)
    outputs = keras.layers.Dense(1, activation = 'sigmoid')(x)
    optimizer = keras.optimizers.Adam(lr=0.000001)
    model = keras.Model(inputs=inputs, outputs=outputs, name="Classifier")
    model.summary()
    #model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    return model
```

Figura 5.6: Implementazione del classificatore in Keras.

Tutti i layers del classificatore utilizzano la funzione "tanh" per la non linearità, tranne il layer di output che utilizza la funzione "sigmoid". Anche il classificatore utilizza degli activity regularizers, insieme a layers di dropout, per cercare di ridurre l'overfitting.

Per mettere insieme le diverse parti dell'architettura e creare un modello unico da utilizzare per i test è stato costruito un "ensemble model", che funge da contenitore per i vari modelli creati precedentemente e permette di connetterli per realizzare il modello finale. La figura 5.7 mostra l'ensemble model costruito.

```
def get_ensemble_model():

  #prepare model for TPU acceleration
  resolver = tf.distribute.cluster_resolver.TPUClusterResolver(tpu='grpc://' + os.environ['COLAB_TPU_ADDR'])
  tf.config.experimental_connect_to_cluster(resolver)
  tf.tpu.experimental.initialize_tpu_system(resolver)
  strategy = tf.distribute.TPUStrategy(resolver)
  with strategy.scope():

    biconv1 = get_BiConvLSTM_model()
    classifier = get_classifier()

    input1 = keras.Input(shape=(1,14,14,512))
    input2 = keras.Input(shape=(1,14,14,512))
                      .
                      .
                      .
    input16 = keras.Input(shape=(1,14,14,512))

    y1 = biconv1(input1)
    y2 = biconv1(input2)
              .
              .
              .
    y16 = biconv1(input16)

    z = layers.maximum([y1,y2,y3,y4,y5,y6,y7,y8,y9,y10,y11,y12,y13,y14,y15,y16])

    outputs = classifier(z)

    model = keras.Model(inputs=[input1,input2, ... , input16], outputs=outputs, name="ensemble")
    optimizer = keras.optimizers.Adagrad(lr=0.001)
    model.summary()
    model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
  return model
```

Figura 5.7: Implementazione dell'ensemble model in Keras.

Come si può vedere dalla figura, anche l'ensemble model viene preparato all'esecuzione tramite TPU. Il primo passo per la costruzione del modello completo è l'istanziazione dei modelli dell'encoder temporale e del classificatore. L'encoder spaziale realizzato con VGG-16 non viene istanziato all'interno dell'ensemble model poiché viene utilizzato separatamente per estrarre le features spaziali dai video. Dato che il modello utilizza pesi pre-addestrati, è molto più efficiente estrarre le features spaziali separatamente dall'addestramento dell'ensemble model, in questo modo si riesce ad alleggerire il modello, che comunque resta molto pesante anche senza l'encoder spaziale. Successivamente si definiscono gli input multipli della rete, che si è deciso essere un numero pari a 16 (ogni chunk passato in input alla rete sarà di 16 differenze di frames). Questi input non sono i frames originali del video, ma piuttosto le features spaziali estratte dalle clips tramite la rete *VGG-16*. In seguito i vari inputs vengono processati dalla rete *BiConvLSTM* che ne estrae le features spazio-temporali e tramite la funzione "`maximum`" messa a disposizione da *Keras* viene effettuato il massimo tra le features prodotte e si ottiene così una rappresentazione finale del chunk passato in ingresso. Questa rappresentazione viene infine passata al classificatore che predice la classe di appartenenza del chunk. Il learning rate è stato settato a 10^{-3} e non a 10^{-6} come descritto nell'articolo perchè

con un learning rate di 10^{-3} il modello riesce ad ottenere risultati leggermente migliori.

Per quel che riguarda la fase di preprocessing degli inputs e l'estrazione delle features spaziali, il procedimento seguito è il seguente: ogni video presente nel dataset viene suddiviso in chunks da 17 frames, e di questi chunks solo la metà vengono presi in considerazione per i processamenti successivi (questo è stato fatto per riprodurre la selezione di frames ad intervalli regolari effettuata nell'articolo); tutti i frames dei chunks presi in considerazione vengono ridimensionati in 224 x 224 pixels, viene effettuata la differenza tra frames adiacenti (il primo con il secondo, il secondo con il terzo e ecc.) e normalizzata per essere nel range tra 0 e 1; la differenza viene passata in input all'encoder spaziale che ne estrapola le features; le varie features ottenute dalle 16 differenze (con 17 frames si ottengono 16 differenze perché l'ultimo fa la differenza solo con il precedente) vengono messe in 16 arrays diversi (uno per ogni differenza), che costituiranno poi i chunks da 16 che verranno passati in input all'ensemble model; dopo aver terminato il processamento dei chunks di tutti i video, gli arrays riempiti delle features ottenute vengono salvati su file, per essere poi utilizzati come inputs dell'encoder temporale.

Le tecniche di data augmentation descritte nell'articolo non sono state utilizzate in quanto non essendoci sicurezza su dove e come effettuarle si è preferito lasciare i datasets non modificati. Inoltre il batch size è stato scelto di 64 e non di 8 perché la TPU divide automaticamente il batch negli 8 cores che mette a disposizione e quindi è possibile moltiplicare il batch size per il numero di core in modo da rendere il modello molto più veloce.

5.2.2 Implementazione dell'Encoder Spaziale

L'implementazione dell'architettura semplificata segue le stesse modalità utilizzate nell'architettura completa. L'encoder spaziale viene realizzato con *VGG-16* allo stesso modo dell'architettura completa. Il classificatore cambia da quello precedente solamente per il fatto che gli activity regularizers sono stati rimossi, in quanto davano problemi in fase di training e il modello non riusciva ad ottenere risultati affidabili. Il classificatore è mostrato in figura 5.8.
Ciò che differisce di più dal modello completo è l'ensemble model che si occupa di connettere le varie parti del modello, in quanto in questo caso la rete *BiConvLSTM* non viene utilizzata. L'implementazione dell'ensemble model è raffigurata in figura 5.9.
Come si vede dalla figura i 16 input del modello (che si ricordano essere le features precedentemente estratte tramite encoder spaziale) vengono direttamente passate alla funzione di elementwise maximum, senza passare per l'encoder temporale come accadeva nell'architettura completa.

```
def get_classifier():
    inputs = keras.Input(shape=(14,14,512))
    x = keras.layers.Flatten()(inputs)
    x = layers.Dense(1000, activation = 'tanh')(x)
    x = keras.layers.Dropout(0.4)(x)
    x = layers.Dense(256, activation = 'tanh')(x)
    x = keras.layers.Dropout(0.4)(x)
    x = layers.Dense(10, activation = 'tanh')(x)
    x = keras.layers.Dropout(0.4)(x)
    outputs = keras.layers.Dense(1, activation = 'sigmoid')(x)
    optimizer = keras.optimizers.Adam(lr=0.000001)
    model = keras.Model(inputs=inputs, outputs=outputs, name="Classifier")
    model.summary()
    #model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
    return model
```

Figura 5.8: Implementazione del classificatore del modello semplificato in Keras.

```
def get_simplified_ensemble_model():

  #prepare model for TPU acceleration
  resolver = tf.distribute.cluster_resolver.TPUClusterResolver(tpu='grpc://' + os.environ['COLAB_TPU_ADDR'])
  tf.config.experimental_connect_to_cluster(resolver)
  tf.tpu.experimental.initialize_tpu_system(resolver)
  strategy = tf.distribute.TPUStrategy(resolver)

  with strategy.scope():

    classifier = get_classifier()

    input1 = keras.Input(shape=(1,14,14,512))
    input2 = keras.Input(shape=(1,14,14,512))
                    .
                    .
                    .
    input16 = keras.Input(shape=(1,14,14,512))

    z = layers.maximum([input1,input2, ... ,input16])
    z = layers.Reshape((14,14,512))(z)
    outputs = classifier(z)

    model = keras.Model(inputs=[input1,input2, ... ,input16], outputs=outputs, name="ensemble_sinplified")
    optimizer = keras.optimizers.Adagrad(lr=0.001)
    model.summary()
    model.compile(loss='binary_crossentropy', optimizer=optimizer, metrics=['accuracy'])
  return model
```

Figura 5.9: Implementazione dell'ensemble model semplificato in Keras.

La fase di preprocessing non subisce grosse modifiche rispetto a quella del modello completo. L'unica differenza è infatti la rimozione del calcolo della differenza tra frames adiacenti, come riportato nell'articolo, che comporta quindi la riduzione dei chunks iniziali da 17 a 16 frames. Per il resto i frames dopo essere stati ridimensionati vengono comunque normalizzati e passati poi in input all'encoder spaziale che ne estrapola le features.

5.3 Test dell'Architettura

In questa sezione verranno mostrati i risultati dei test dell'architettura nelle sue due varianti. I due modelli sono stati valutati sui tre dataset già descritti in precedenza: *Hockey Fights*, *Crowd Violence* e *AIRTLab Violence*. Come nell'articolo, l'ottimizzatore utilizzato per il testing è *Adam*, ma i modelli verranno testati anche con l'ottimizzatore *Adagrad* per valutarne le differenze e vedere come l'architettura si comporta con ottimizzatori differenti.

La tecnica di valutazione utilizzata non è propriamente una *K-Fold Cross Validation* con $K = 5$ come fatto nell'articolo, poiché non è stato trovato un metodo semplice fornito da *Keras* che permettesse di effettuare un K-Fold lavorando con input multipli, in modo da suddividerli in vari splits. Si è optato quindi per la ripetizione di un ciclo n volte (dove n è il numero di splits) all'interno del quale il dataset viene mescolato in maniera randomica e poi suddiviso interamente in training e test sets, tramite la funzione "train_test_split" che permette di suddividere i 16 inputs e le labels in sets per il training e per il test. La dimensione del test set è del 20% dell'intero dataset, in modo da coincidere con la dimensione che il test set avrebbe avuto in una *5-Fold Cross Validation*. All'interno del ciclo successivamente il modello viene addestrato e valutato. Infine viene effettuata la media tra i risultati ottenuti nei vari splits e viene disegnata la *ROC Curve* delle performance del modello.

Una *ROC curve* (Receiver Operating Charateristic curve) è un grafo che mostra le performance di un classificatore binario. Lungo i due assi vengono rappresentati il *True Positive Rate* (TPR) e il *False Positive Rate* (FPR), introdotti nei capitoli precedenti. La *ROC curve* mostra il rapporto tra TPR e FPR a diverse soglie di classificazione. Per calcolare i punti nella curva viene utilizzata la *AUC*. L'*AUC* (Area Under the ROC Curve) è l'area bidimensionale che si trova sotto l'intera *ROC Curve* e misura la performance attraverso tutte le possibili soglie di classificazione. L'*AUC* varia tra 0 e 1 e un modello le cui predizioni sono al 100% sbagliate avrà un'*AUC* di 0, mentre un modello le cui predizioni sono al 100% corrette avrà un'*AUC* di 1. Questa misura è utile perché è invariante rispetto alla scala utilizzata e perché non dipende dalla soglia di classificazione scelta.

Nei test effettuati in questo lavoro è stato scelto un numero di 3 splits per la valutazione dei modelli nell'*Hockey Fights Dataset* e nel *Crowd Violence Dataset*, mentre è stato utilizzato un unico split nell'*AIRTLab Violence Dataset*, per via della pesantezza degli inputs che tendevano a saturare la memoria a disposizione.

5.3.1 Test dell'Architettura Completa

Test sul Crowd Violence Dataset

I risultati dei test effettuati nel *Crowd Violence Dataset* sono mostrati nella figura 5.10.

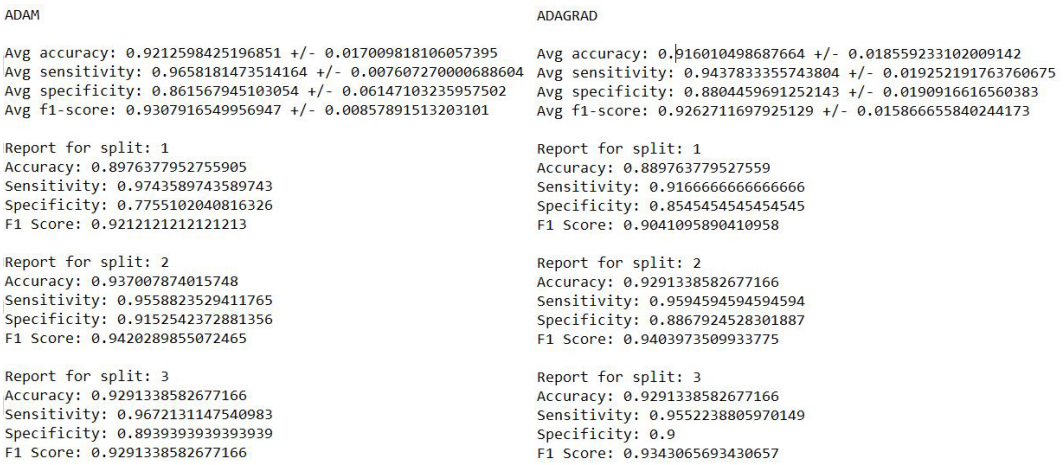

Figura 5.10: Risultati dei test dell'architettura completa BiConvLSTM sul Crowd Violence Dataset.

A sinistra sono mostrati i risultati ottenuti con l'ottimizzatore *Adam*, mentre a destra quelli ottenuti con l'ottimizzatore *Adagrad*. In alto è riportata la media delle performance ottenute nei vari splits, e sotto sono riportati i risultati ottenuti per ogni singolo split effettuato. Come possiamo vedere dalla figura il modello ha ottenuto risultati discreti per entrambi gli ottimizzatori, anche se l'ottimizzatore *Adam* ha avuto nel complesso una performance leggermente migliore di quella dell'ottimizzatore *Adagrad*, che lo supera solamente nella specificità.
Una considerazione da fare è che il modello ha avuto in entrambi i casi più difficoltà nell'identificare correttamente le clips non violente, in quanto la specificità ottenuta è sempre relativamente bassa rispetto agli altri paramentri in tutti gli splits effettuati.

Di seguito sono riportate le ultime 10 epoche di addestramento del modello con ottimizzatore *Adam*.

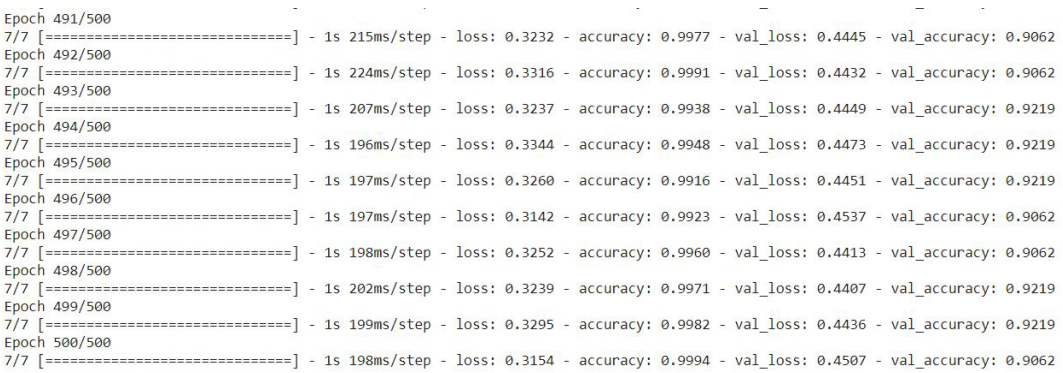

Figura 5.11: Fase di addestramento dell'architettura completa BiConvLSTM sul Crowd Violence Dataset con ottimizzatore Adam.

Come succedeva anche per altri modelli testati precedentemente, anche qui il modello risulta vittima dell'overfitting, in quanto l'accuratezza in fase di addestramento

risulta prossima al 100%, mentre poi la validation accuracy risulta essere più bassa. É probabilmente per questo motivo che l'architettura non riesce ad ottenere risultati migliori. Un comportamento diverso si ottiene invece nell'addestramento con ottimizzatore *Adagrad*, riportato in figura 5.12.

```
Epoch 491/500
7/7 [==============================] - 1s 195ms/step - loss: 0.3590 - accuracy: 0.9743 - val_loss: 0.3556 - val_accuracy: 0.9688
Epoch 492/500
7/7 [==============================] - 1s 199ms/step - loss: 0.3625 - accuracy: 0.9794 - val_loss: 0.3523 - val_accuracy: 0.9688
Epoch 493/500
7/7 [==============================] - 1s 189ms/step - loss: 0.3630 - accuracy: 0.9828 - val_loss: 0.3512 - val_accuracy: 0.9531
Epoch 494/500
7/7 [==============================] - 1s 184ms/step - loss: 0.3439 - accuracy: 0.9814 - val_loss: 0.3585 - val_accuracy: 0.9688
Epoch 495/500
7/7 [==============================] - 1s 186ms/step - loss: 0.3515 - accuracy: 0.9867 - val_loss: 0.3517 - val_accuracy: 0.9688
Epoch 496/500
7/7 [==============================] - 1s 189ms/step - loss: 0.3467 - accuracy: 0.9880 - val_loss: 0.3498 - val_accuracy: 0.9531
Epoch 497/500
7/7 [==============================] - 1s 188ms/step - loss: 0.3335 - accuracy: 0.9747 - val_loss: 0.3507 - val_accuracy: 0.9531
Epoch 498/500
7/7 [==============================] - 1s 195ms/step - loss: 0.3458 - accuracy: 0.9840 - val_loss: 0.3506 - val_accuracy: 0.9531
Epoch 499/500
7/7 [==============================] - 1s 191ms/step - loss: 0.3339 - accuracy: 0.9792 - val_loss: 0.3479 - val_accuracy: 0.9844
Epoch 500/500
7/7 [==============================] - 1s 181ms/step - loss: 0.3313 - accuracy: 0.9839 - val_loss: 0.3479 - val_accuracy: 0.9688
4/4 [==============================] - 31s 55ms/step - loss: 0.4351 - accuracy: 0.9291
```

Figura 5.12: Fase di addestramento dell'architettura completa BiConvLSTM sul Crowd Violence Dataset con ottimizzatore Adagrad.

In questo caso la differenza tra accuratezza di addestramento e accuratezza di validazione è molto meno accentuata, tuttavia il modello in fase di valutazione ottiene comunque risultati leggermente peggiori rispetto a quelli ottenuti durante l'addestramento. Questo potrebbe comunque essere dato da un problema di overfitting.

Nella figura 5.13 sono invece riportate le curve ROC ottenute dall'architettura (a sinistra quella dell'architettura con l'ottimizzatore *Adam* e a destra quella con *Adagrad*).

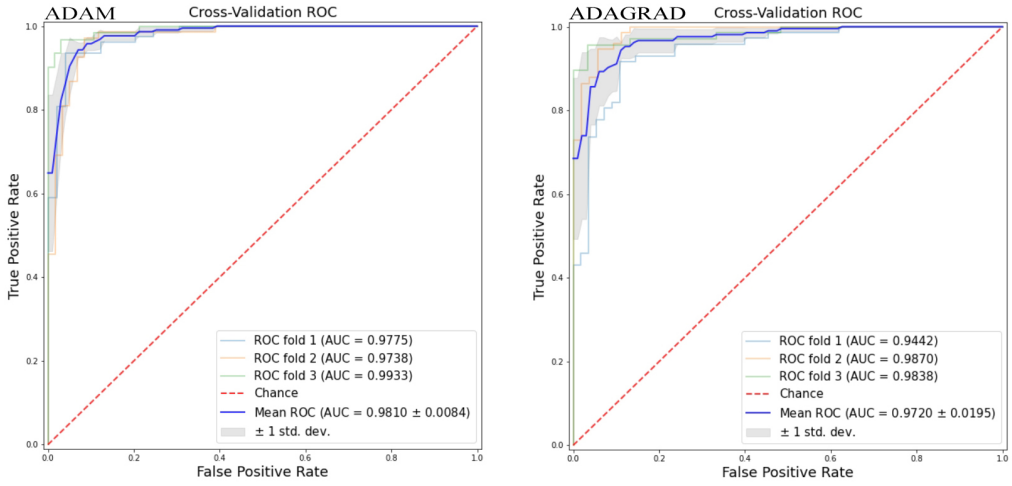

Figura 5.13: ROC curve dell'architettura completa BiConvLSTM sul Crowd Violence Dataset.

Con l'ottimizzatore *Adam* l'architettura ha ottenuto una *AUC* di 0.9810, che va a confermare le migliori performance ottenute rispetto all'ottimizzatore *Adagrad*, con

il quale il modello ha ottenuto una *AUC* di 0.9720.

Test sull'Hockey Fights Dataset

I risultati dei test effettuati nell'*Hockey Fights Dataset* sono mostrati nella figura 5.14.

```
ADAM

Avg accuracy: 0.9066666666666667 +/- 0.009428090415820642
Avg sensitivity: 0.8722410021873136 +/- 0.03967203248055956
Avg specificity: 0.9367349699059986 +/- 0.019809796809661412
Avg f1-score: 0.9005690571111131 +/- 0.017507427393210734

Report for split: 1
Accuracy: 0.9
Sensitivity: 0.8297872340425532
Specificity: 0.9622641509433962
F1 Score: 0.8863636363636364

Report for split: 2
Accuracy: 0.92
Sensitivity: 0.9252336448598131
Specificity: 0.9139784946236559
F1 Score: 0.9252336448598131

Report for split: 3
Accuracy: 0.9
Sensitivity: 0.8617021276595744
Specificity: 0.9339622641509434
F1 Score: 0.8901098901098901
```

```
ADAGRAD

Avg accuracy: 0.9233333333333333 +/- 0.012472191289246433
Avg sensitivity: 0.938156288156288 +/- 0.018458736083343693
Avg specificity: 0.9081908900692097 +/- 0.014825932401260344
Avg f1-score: 0.9226334121819838 +/- 0.01683030560154095

Report for split: 1
Accuracy: 0.92
Sensitivity: 0.95
Specificity: 0.89
F1 Score: 0.9223300970873787

Report for split: 2
Accuracy: 0.94
Sensitivity: 0.9523809523809523
Specificity: 0.9263157894736842
F1 Score: 0.9433962264150944

Report for split: 3
Accuracy: 0.91
Sensitivity: 0.9120879120879121
Specificity: 0.9082568807339449
F1 Score: 0.9021739130434783
```

Figura 5.14: Risultati dei test dell'architettura completa BiConvLSTM sull'Hockey Fights Dataset.

Anche in questo caso il modello ha ottenuto risultati discreti per entrambi gli ottimizzatori. Questa volta però è l'ottimizzatore *Adagrad* ad aver ottenuto dei risultati leggermente migliori nel complesso, superato da *Adam* solo per quanto riguarda la specificità.

Una nota interessante da fare è che guardando i vari splits si nota che con l'ottimizzatore *Adam* il modello sia riuscito ad identificare meglio le scene non violente rispetto a quelle violente, mentre con l'ottimizzatore *Adagrad* il modello ha avuto più confidenza nell'identificare correttamente le scene violente rispetto a quelle non violente.

Nella figura 5.15 sono riportate le ultime 10 epoche di addestramento del modello con ottimizzatore *Adam*.

Anche qui come prima il modello sembra essere vittima di overfitting e numeri molto simili si ottengono nell'addestramento del modello con ottimizzatore *Adagrad*.

Nella figura 5.16 sono invece riportate le curve ROC ottenute dall'architettura.

Con l'ottimizzatore *Adam* l'architettura ha ottenuto una *AUC* di 0.9640, mentre con l'ottimizzatore *Adagrad* il modello ha ottenuto una *AUC* di 0.9783, a conferma dei risultati leggermente migliori ottenuti dal modello con *Adagrad*.

```
Epoch 134/500
11/11 [==============================] - 2s 187ms/step - loss: 0.0161 - accuracy: 1.0000 - val_loss: 0.2727 - val_accuracy: 0.9200
Epoch 135/500
11/11 [==============================] - 2s 189ms/step - loss: 0.0170 - accuracy: 1.0000 - val_loss: 0.2736 - val_accuracy: 0.9200
Epoch 136/500
11/11 [==============================] - 2s 190ms/step - loss: 0.0158 - accuracy: 1.0000 - val_loss: 0.2965 - val_accuracy: 0.9100
Epoch 137/500
11/11 [==============================] - 2s 190ms/step - loss: 0.0173 - accuracy: 1.0000 - val_loss: 0.3410 - val_accuracy: 0.8900
Epoch 138/500
11/11 [==============================] - 3s 321ms/step - loss: 0.0185 - accuracy: 1.0000 - val_loss: 0.3154 - val_accuracy: 0.8900
Epoch 139/500
11/11 [==============================] - 2s 185ms/step - loss: 0.0165 - accuracy: 1.0000 - val_loss: 0.3463 - val_accuracy: 0.8900
Epoch 140/500
11/11 [==============================] - 2s 185ms/step - loss: 0.0160 - accuracy: 1.0000 - val_loss: 0.3346 - val_accuracy: 0.8900
Epoch 141/500
11/11 [==============================] - 2s 188ms/step - loss: 0.0177 - accuracy: 1.0000 - val_loss: 0.2969 - val_accuracy: 0.9100
Epoch 142/500
11/11 [==============================] - 2s 192ms/step - loss: 0.0178 - accuracy: 0.9998 - val_loss: 0.3025 - val_accuracy: 0.9200
Epoch 143/500
11/11 [==============================] - 2s 192ms/step - loss: 0.0155 - accuracy: 1.0000 - val_loss: 0.3033 - val_accuracy: 0.9100
```

Figura 5.15: Fase di addestramento dell'architettura completa BiConvLSTM sull'Hockey Fights Dataset.

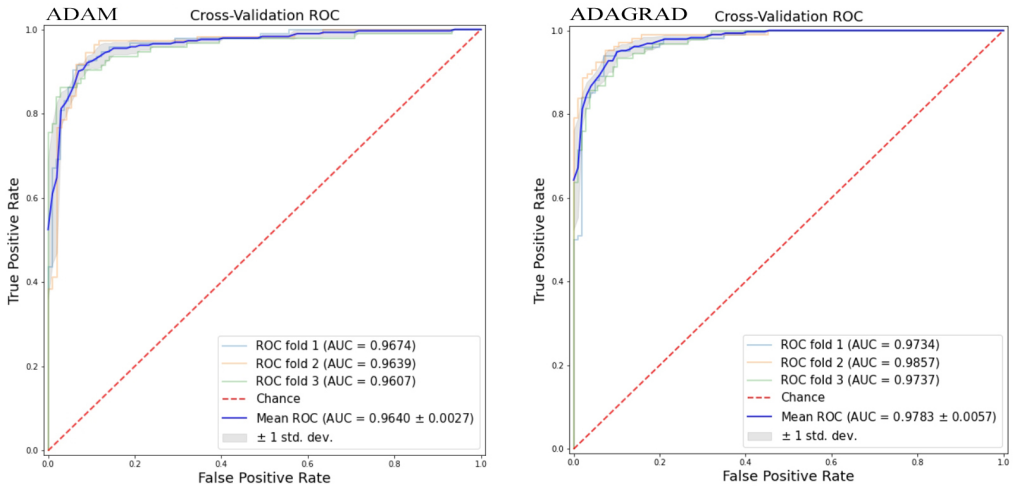

Figura 5.16: ROC curve dell'architettura completa BiConvLSTM sull'Hockey Fights Dataset.

Test sull'AIRTLab Violence Dataset

I risultati dei test effettuati nell'*AIRTLab Violence Dataset* sono mostrati nella figura 5.17

Dalla figura si nota come questa volta il modello non sia riuscito ad ottenere buoni risultati, sia con l'ottimizzatore *Adam* che con *Adagrad*. Questi risultati sono piuttosto inaspettati, in quanto negli altri dataset testati il modello ha ottenuto comunque risultati discreti. É anche vero però che l'*AIRTLab Violence Dataset* è strutturalmente molto diverso dagli altri due. Inoltre quest'ultimo dataset contiene molte scene ambigue che possono fuorviare il modello, ed infatti con entrambi gli ottimizzatori il modello ha avuto parecchi problemi nel riconoscere correttamente le clips non violente, e questo ha sicuramente influito sulle non ottime prestazioni ottenute.

Nella figura 5.18 sono riportate le ultime 10 epoche di addestramento del modello

```
Adam
Avg accuracy: 0.8342857142857143 +/- 0.0
Avg sensitivity: 0.9461883408071748 +/- 0.0
Avg specificity: 0.6377952755905512 +/- 0.0
Avg f1-score: 0.8791666666666665 +/- 0.0

Adagrad
Avg accuracy: 0.8057142857142857 +/- 0.0
Avg sensitivity: 0.8508771929824561 +/- 0.0
Avg specificity: 0.7213114754098361 +/- 0.0
Avg f1-score: 0.850877192982456 +/- 0.0
```

Figura 5.17: Risultati dei test dell'architettura completa BiConvLSTM sull'AIRTLab Violence Dataset.

con ottimizzatore *Adam*.

```
Epoch 121/500
20/20 [==============================] - 3s 160ms/step - loss: 0.0373 - accuracy: 1.0000 - val_loss: 0.4925 - val_accuracy: 0.8571
Epoch 122/500
20/20 [==============================] - 3s 160ms/step - loss: 0.0319 - accuracy: 1.0000 - val_loss: 0.5001 - val_accuracy: 0.8514
Epoch 123/500
20/20 [==============================] - 3s 156ms/step - loss: 0.0309 - accuracy: 1.0000 - val_loss: 0.4791 - val_accuracy: 0.8514
Epoch 124/500
20/20 [==============================] - 3s 157ms/step - loss: 0.0311 - accuracy: 1.0000 - val_loss: 0.4861 - val_accuracy: 0.8629
Epoch 125/500
20/20 [==============================] - 3s 156ms/step - loss: 0.0327 - accuracy: 0.9973 - val_loss: 0.4976 - val_accuracy: 0.8686
Epoch 126/500
20/20 [==============================] - 3s 158ms/step - loss: 0.0279 - accuracy: 1.0000 - val_loss: 0.4958 - val_accuracy: 0.8629
Epoch 127/500
20/20 [==============================] - 3s 157ms/step - loss: 0.0285 - accuracy: 1.0000 - val_loss: 0.5120 - val_accuracy: 0.8457
Epoch 128/500
20/20 [==============================] - 3s 160ms/step - loss: 0.0329 - accuracy: 0.9982 - val_loss: 0.6670 - val_accuracy: 0.8171
Epoch 129/500
20/20 [==============================] - 3s 156ms/step - loss: 0.0773 - accuracy: 0.9871 - val_loss: 0.6196 - val_accuracy: 0.8229
Epoch 130/500
20/20 [==============================] - 3s 158ms/step - loss: 0.0831 - accuracy: 0.9808 - val_loss: 0.6349 - val_accuracy: 0.8057
Epoch 131/500
20/20 [==============================] - 3s 157ms/step - loss: 0.1620 - accuracy: 0.9610 - val_loss: 0.7178 - val_accuracy: 0.8343
Epoch 132/500
20/20 [==============================] - 4s 224ms/step - loss: 0.1050 - accuracy: 0.9746 - val_loss: 0.6925 - val_accuracy: 0.8057
```

Figura 5.18: Fase di addestramento dell'architettura completa BiConvLSTM sull'AIRTLab Violence Dataset.

Anche nell'*AIRTLab Dataset* il modello risulta vittima di overfitting, con un'accuratezza di addestramento prossima al 100% ma accuratezza di validazione decisamente più bassa. La stessa cosa accade con l'ottimizzatore *Adagrad*.

Nella figura 5.19 sono invece riportate le curve ROC ottenute dall'architettura.

Con l'ottimizzatore *Adam* l'architettura ha ottenuto una *AUC* di 0.8967, mentre con l'ottimizzatore *Adagrad* il modello ha ottenuto una *AUC* di 0.9010, e la forma delle curve rispecchia i risultati non ottimi ottenuti.

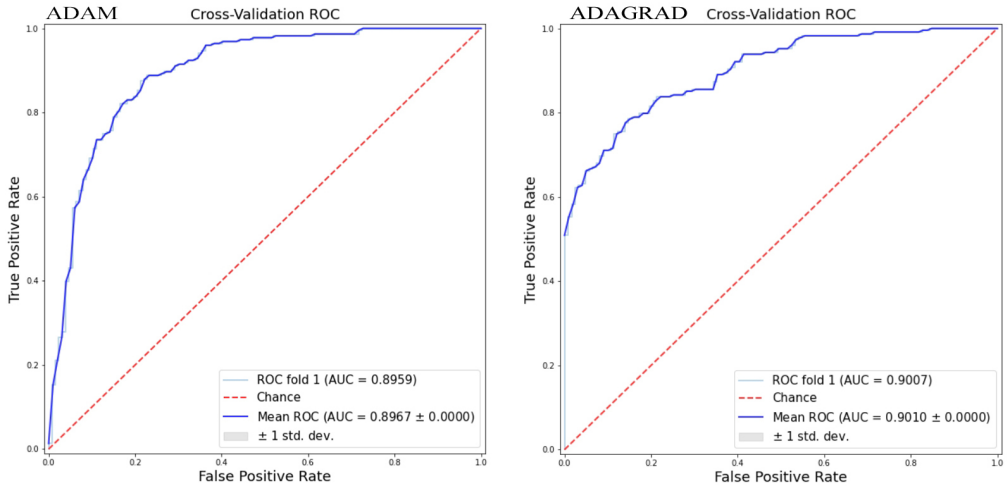

Figura 5.19: ROC curve dell'architettura completa BiConvLSTM sull'AIRTLAB Violence Dataset.

5.3.2 Test dell'Architettura Semplificata

Test sul Crowd Violence Dataset

I risultati dei test effettuati nel *Crowd Violence Dataset* sono mostrati nella figura 5.20.

```
ADAM

Avg accuracy: 0.945679012345679 +/- 0.01944199475064644
Avg sensitivity: 0.9672409488139827 +/- 0.00861593281445783
Avg specificity: 0.9188405797101448 +/- 0.04875040055522946
Avg f1-score: 0.9542442712867061 +/- 0.01719296702396789

Report for split: 1
Accuracy: 0.9185185185185185
Sensitivity: 0.9733333333333334
Specificity: 0.85
F1 Score: 0.9299363057324841

Report for split: 2
Accuracy: 0.9555555555555556
Sensitivity: 0.9550561797752809
Specificity: 0.9565217391304348
F1 Score: 0.9659090909090908

Report for split: 3
Accuracy: 0.9629629629629629
Sensitivity: 0.9733333333333334
Specificity: 0.95
F1 Score: 0.9668874172185431
```

```
ADAGRAD

Avg accuracy: 0.9654320987654321 +/- 0.009238660214256619
Avg sensitivity: 0.9750387596899225 +/- 0.00888017119532645
Avg specificity: 0.9530612244897959 +/- 0.027558717885536564
Avg f1-score: 0.9705549889230567 +/- 0.007687383391103202

Report for split: 1
Accuracy: 0.9555555555555556
Sensitivity: 0.9866666666666667
Specificity: 0.9166666666666666
F1 Score: 0.961038961038961

Report for split: 2
Accuracy: 0.9629629629629629
Sensitivity: 0.9651162790697675
Specificity: 0.9591836734693877
F1 Score: 0.9707602339181286

Report for split: 3
Accuracy: 0.9777777777777777
Sensitivity: 0.9733333333333334
Specificity: 0.9833333333333333
F1 Score: 0.9798657718120806
```

Figura 5.20: Risultati dei test dell'architettura semplificata BiConvLSTM sul Crowd Violence Dataset.

Il modello è riuscito ad ottenere risultati abbastanza buoni con entrambi gli ottimizzatori, soprattutto con l'ottimizzatore *Adagrad*. Andando a vedere i singoli splits effettuati si nota che in alcuni di essi il modello raggiunge risultati ottimi, in particolare nell'ultimo split con *Adagrad*, mentre in altri l'architettura ottiene risultati peggiori per via della difficoltà nel riconoscere scene non violente. Come accadeva molto spesso anche nell'architettura completa, anche qui in generale l'architettura raggiunge risultati tendenzialmente peggiori nella specificità rispetto agli

altri parametri presi in considerazione.

Di seguito sono riportate le ultime 10 epoche di addestramento del modello con ottimizzatore *Adam*.

```
Epoch 432/500
8/8 [==============================] - 1s 191ms/step - loss: 0.1272 - accuracy: 0.9596 - val_loss: 0.1121 - val_accuracy: 0.9706
Epoch 433/500
8/8 [==============================] - 2s 276ms/step - loss: 0.1400 - accuracy: 0.9626 - val_loss: 0.0804 - val_accuracy: 0.9853
Epoch 434/500
8/8 [==============================] - 1s 194ms/step - loss: 0.0790 - accuracy: 0.9728 - val_loss: 0.1051 - val_accuracy: 0.9706
Epoch 435/500
8/8 [==============================] - 1s 192ms/step - loss: 0.1150 - accuracy: 0.9707 - val_loss: 0.0741 - val_accuracy: 0.9853
Epoch 436/500
8/8 [==============================] - 1s 192ms/step - loss: 0.1594 - accuracy: 0.9599 - val_loss: 0.0895 - val_accuracy: 0.9706
Epoch 437/500
8/8 [==============================] - 1s 188ms/step - loss: 0.1191 - accuracy: 0.9645 - val_loss: 0.1172 - val_accuracy: 0.9706
Epoch 438/500
8/8 [==============================] - 1s 189ms/step - loss: 0.1188 - accuracy: 0.9636 - val_loss: 0.0556 - val_accuracy: 0.9853
Epoch 439/500
8/8 [==============================] - 1s 189ms/step - loss: 0.1377 - accuracy: 0.9507 - val_loss: 0.0958 - val_accuracy: 0.9706
Epoch 440/500
8/8 [==============================] - 1s 192ms/step - loss: 0.1397 - accuracy: 0.9659 - val_loss: 0.0640 - val_accuracy: 0.9853
Epoch 441/500
8/8 [==============================] - 1s 192ms/step - loss: 0.1205 - accuracy: 0.9697 - val_loss: 0.0740 - val_accuracy: 0.9853
```

Figura 5.21: Fase di addestramento dell'architettura semplificata BiConvLSTM sul Crowd Violence Dataset con ottimizzatore Adam.

Dai dati mostrati nell'addestramento del modello sembrerebbe che a differenza dell'architettura completa il modello semplificato non cade nel problema dell'overfitting e di conseguenza ottiene risultati migliori. Probabilmente il numero ridotto di parametri presente nell'architettura semplificata aiuta il modello a non adattarsi eccessivamente al dataset.

Nella figura 5.22 sono invece riportate le curve ROC ottenute dall'architettura.

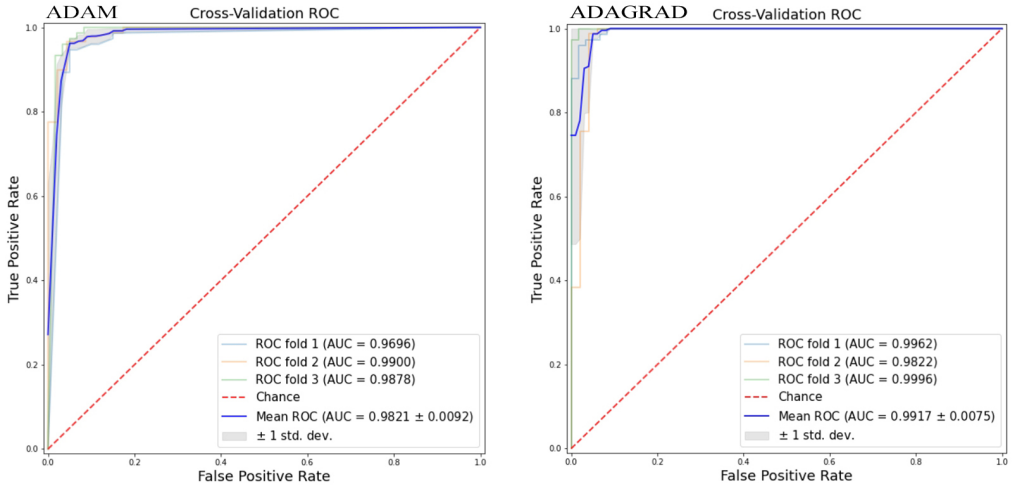

Figura 5.22: ROC curve dell'architettura semplificata BiConvLSTM sul Crowd Violence Dataset.

Con l'ottimizzatore *Adam* l'architettura ha ottenuto una *AUC* di 0.9821, mentre con l'ottimizzatore *Adagrad* il modello ha ottenuto una *AUC* di 0.9917. Le curve confermano i buoni risultati ottenuti dal modello.

Test sull'Hockey Fights Dataset

I risultati dei test effettuati nell'*Hockey Fights Dataset* sono mostrati nella figura 5.23.

```
ADAM

Avg accuracy: 0.9438943894389439 +/- 0.014195256216241925
Avg sensitivity: 0.9564625850340135 +/- 0.009139475403091435
Avg specificity: 0.9316867772750127 +/- 0.02147243783830611
Avg f1-score: 0.9438996199098127 +/- 0.013412182182153617

Report for split: 1
Accuracy: 0.9455445544554455
Sensitivity: 0.95
Specificity: 0.9411764705882353
F1 Score: 0.9452736318407959

Report for split: 2
Accuracy: 0.9257425742574258
Sensitivity: 0.95
Specificity: 0.9019607843137255
F1 Score: 0.9268292682926829

Report for split: 3
Accuracy: 0.9603960396039604
Sensitivity: 0.9693877551020408
Specificity: 0.9519230769230769
F1 Score: 0.9595959595959594
```

```
ADAGRAD

Avg accuracy: 0.9504950495049505 +/- 0.010694291581531132
Avg sensitivity: 0.9492269392033542 +/- 0.010152287216094923
Avg specificity: 0.95020964360587 +/- 0.026383046013948065
Avg f1-score: 0.9499385947902527 +/- 0.00924705421276142

Report for split: 1
Accuracy: 0.9455445544554455
Sensitivity: 0.9375
Specificity: 0.9528301886792453
F1 Score: 0.9424083769633508

Report for split: 2
Accuracy: 0.9405940594059405
Sensitivity: 0.9622641509433962
Specificity: 0.9166666666666666
F1 Score: 0.9444444444444444

Report for split: 3
Accuracy: 0.9653465346534653
Sensitivity: 0.9479166666666666
Specificity: 0.9811320754716981
F1 Score: 0.962962962962963
```

Figura 5.23: Risultati dei test dell'architettura semplificata BiConvLSTM sull'Hockey Fights Dataset.

Anche in questo caso il modello ha ottenuto buoni risultati per entrambi gli ottimizzatori. L'ottimizzatore *Adagrad* ha ottenuto dei risultati leggermente migliori nel complesso rispetto all'ottimizzatore *Adam*, che supera *Adagrad* solo per quanto riguarda la sensitività.
Al contrario di quanto accaduto con l'architettura completa nello stesso dataset, guardando i vari splits si nota che con l'ottimizzatore *Adam* il modello è riuscito ad identificare meglio le scene violente rispetto a quelle non violente, mentre con l'ottimizzatore *Adagrad* il modello ha avuto complessivamente più confidenza nell'identificare correttamente le scene non violente rispetto a quelle violente.

Nella figura 5.24 sono riportate le ultime 10 epoche di addestramento del modello con ottimizzatore *Adam*.

```
Epoch 226/500
11/11 [==============================] - 2s 150ms/step - loss: 0.1367 - accuracy: 0.9678 - val_loss: 0.3001 - val_accuracy: 0.9208
Epoch 227/500
11/11 [==============================] - 2s 157ms/step - loss: 0.1401 - accuracy: 0.9656 - val_loss: 0.2501 - val_accuracy: 0.9208
Epoch 228/500
11/11 [==============================] - 2s 156ms/step - loss: 0.1769 - accuracy: 0.9494 - val_loss: 0.2730 - val_accuracy: 0.9208
Epoch 229/500
11/11 [==============================] - 2s 154ms/step - loss: 0.1520 - accuracy: 0.9534 - val_loss: 0.2723 - val_accuracy: 0.9208
Epoch 230/500
11/11 [==============================] - 2s 153ms/step - loss: 0.1554 - accuracy: 0.9564 - val_loss: 0.2762 - val_accuracy: 0.9208
Epoch 231/500
11/11 [==============================] - 2s 151ms/step - loss: 0.1504 - accuracy: 0.9615 - val_loss: 0.3223 - val_accuracy: 0.9208
Epoch 232/500
11/11 [==============================] - 2s 157ms/step - loss: 0.1485 - accuracy: 0.9581 - val_loss: 0.2887 - val_accuracy: 0.9208
Epoch 233/500
11/11 [==============================] - 2s 153ms/step - loss: 0.1754 - accuracy: 0.9405 - val_loss: 0.2397 - val_accuracy: 0.9307
Epoch 234/500
11/11 [==============================] - 2s 156ms/step - loss: 0.1647 - accuracy: 0.9565 - val_loss: 0.2282 - val_accuracy: 0.9307
Epoch 235/500
11/11 [==============================] - 2s 152ms/step - loss: 0.1995 - accuracy: 0.9455 - val_loss: 0.2743 - val_accuracy: 0.9208
```

Figura 5.24: Fase di addestramento dell'architettura semplificata BiConvLSTM sull'Hockey Fights Dataset con ottimizzatore Adam.

L'architettura ottiene un'accuratezza migliore in fase di training rispetto a quella di validazione, ciò potrebbe significare un adattamento leggermente eccessivo al dataset, rispetto a quello che invece accade con il *Crowd Violence Dataset*.
L'overfitting del modello risulta invece più accentuato nell'addestramento del modello con ottimizzatore *Adagrad*, riportato in figura 5.25, nella quale si vede come l'accuratezza di addestramento sia esattamente del 100%, decisamente più elevata rispetto a quella di validazione.

```
Epoch 189/500
11/11 [==============================] - 2s 193ms/step - loss: 0.0276 - accuracy: 1.0000 - val_loss: 0.2562 - val_accuracy: 0.9406
Epoch 190/500
11/11 [==============================] - 2s 154ms/step - loss: 0.0287 - accuracy: 1.0000 - val_loss: 0.2597 - val_accuracy: 0.9307
Epoch 191/500
11/11 [==============================] - 2s 150ms/step - loss: 0.0280 - accuracy: 1.0000 - val_loss: 0.2619 - val_accuracy: 0.9307
Epoch 192/500
11/11 [==============================] - 2s 154ms/step - loss: 0.0301 - accuracy: 1.0000 - val_loss: 0.2602 - val_accuracy: 0.9307
Epoch 193/500
11/11 [==============================] - 2s 151ms/step - loss: 0.0258 - accuracy: 1.0000 - val_loss: 0.2596 - val_accuracy: 0.9307
Epoch 194/500
11/11 [==============================] - 2s 155ms/step - loss: 0.0248 - accuracy: 1.0000 - val_loss: 0.2598 - val_accuracy: 0.9307
Epoch 195/500
11/11 [==============================] - 2s 154ms/step - loss: 0.0274 - accuracy: 1.0000 - val_loss: 0.2586 - val_accuracy: 0.9307
Epoch 196/500
11/11 [==============================] - 2s 155ms/step - loss: 0.0264 - accuracy: 1.0000 - val_loss: 0.2586 - val_accuracy: 0.9307
Epoch 197/500
11/11 [==============================] - 2s 150ms/step - loss: 0.0295 - accuracy: 1.0000 - val_loss: 0.2585 - val_accuracy: 0.9307
Epoch 198/500
11/11 [==============================] - 2s 157ms/step - loss: 0.0287 - accuracy: 1.0000 - val_loss: 0.2596 - val_accuracy: 0.9307
```

Figura 5.25: Fase di addestramento dell'architettura semplificata BiConvLSTM sull'Hockey Fights Dataset con ottimizzatore Adagrad.

Nella figura 5.26 sono riportate le curve ROC ottenute dall'architettura.

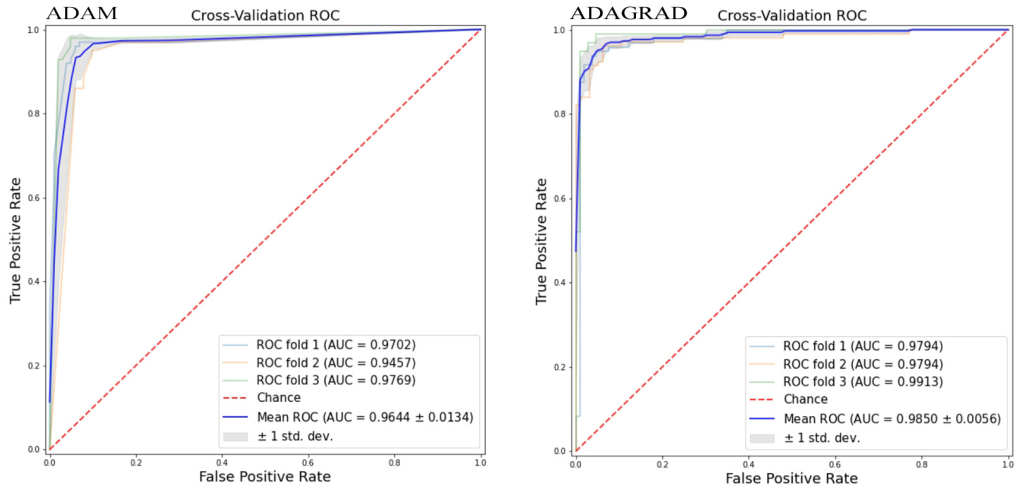

Figura 5.26: ROC curve dell'architettura semplificata BiConvLSTM sull'Hockey Fights Dataset.

Con l'ottimizzatore *Adam* l'architettura ha ottenuto una *AUC* di 0.9644, mentre con l'ottimizzatore *Adagrad* il modello ha ottenuto una *AUC* di 0.9850.

Test sull'AIRTLab Violence Dataset

I risultati dei test effettuati nell'*AIRTLab Violence Dataset* sono mostrati nella figura 5.27

```
AIRTLAB DATASET

ADAGRAD

Avg accuracy: 0.967741935483871 +/- 0.0
Avg sensitivity: 0.975609756097561 +/- 0.0
Avg specificity: 0.9523809523809523 +/- 0.0
Avg f1-score: 0.975609756097561 +/- 0.0

ADAM

Avg accuracy: 0.9301075268817204 +/- 0.0
Avg sensitivity: 0.9841269841269841 +/- 0.0
Avg specificity: 0.8166666666666667 +/- 0.0
Avg f1-score: 0.950191570881226 +/- 0.0
```

Figura 5.27: Risultati dei test dell'architettura semplificata BiConvLSTM sull'AIRTLab Violence Dataset.

Dalla figura si nota che il modello questa volta è riuscito ad ottenere risultati molto buoni, a differenza di ciò che accade invece con l'architettura completa nello stesso dataset. I risultati sono buoni sia con l'ottimizzatore *Adam* che con *Adagrad*, con il quale l'architettura ottiene i risultati migliori, anche se viene superato in sensitività da *Adam* ma solamente perché con *Adam* il modello tende a classificare come violenti la maggior parte dei video e infatti ha una specificità molto bassa (81.5%).

Nella figura 5.28 sono riportate le ultime 10 epoche di addestramento del modello con ottimizzatore *Adam*.

```
Epoch 353/500
21/21 [==============================] - 3s 128ms/step - loss: 0.4813 - accuracy: 0.7581 - val_loss: 0.3599 - val_accuracy: 0.9355
Epoch 354/500
21/21 [==============================] - 3s 130ms/step - loss: 0.5126 - accuracy: 0.7676 - val_loss: 0.2997 - val_accuracy: 0.9462
Epoch 355/500
21/21 [==============================] - 3s 129ms/step - loss: 0.4687 - accuracy: 0.7901 - val_loss: 0.2620 - val_accuracy: 0.9516
Epoch 356/500
21/21 [==============================] - 3s 131ms/step - loss: 0.4537 - accuracy: 0.7990 - val_loss: 0.2507 - val_accuracy: 0.9355
Epoch 357/500
21/21 [==============================] - 3s 154ms/step - loss: 0.3501 - accuracy: 0.8658 - val_loss: 0.2238 - val_accuracy: 0.9409
Epoch 358/500
21/21 [==============================] - 3s 130ms/step - loss: 0.2992 - accuracy: 0.8872 - val_loss: 0.2516 - val_accuracy: 0.9247
Epoch 359/500
21/21 [==============================] - 3s 131ms/step - loss: 0.4368 - accuracy: 0.8293 - val_loss: 0.2700 - val_accuracy: 0.9247
Epoch 360/500
21/21 [==============================] - 3s 132ms/step - loss: 0.4097 - accuracy: 0.8141 - val_loss: 0.2526 - val_accuracy: 0.9247
Epoch 361/500
21/21 [==============================] - 3s 129ms/step - loss: 0.3427 - accuracy: 0.8692 - val_loss: 0.2651 - val_accuracy: 0.9247
Epoch 362/500
21/21 [==============================] - 3s 129ms/step - loss: 0.3688 - accuracy: 0.8403 - val_loss: 0.2434 - val_accuracy: 0.9247
```

Figura 5.28: Fase di addestramento dell'architettura semplificata BiConvLSTM sull'AIRTLab Violence Dataset con l'ottimizzatore Adam.

La fase di addestramento con *Adam* è piuttosto curiosa: l'accuratezza ottenuta in fase di addestramento è decisamente bassa quando invece quella di validazione è molto più elevata. Dai risultati osservati nella figura 5.27 una giustificazione che si

può dare è che dato che il modello con l'ottimizzatore *Adam* è incline a classificare la maggior parte dei video come violenti, nella fase di addestramento è più probabile che si siano presentati più video etichettati come non violenti rispetto invece alla fase di validazione nella quale i video sono di meno (la fase di validazione utilizza il 12.5% del dataset completo) e dove quindi il modello ha avuto meno probabilità di incontrare clips non violente (si ricorda che il dataset è sbilanciato e contiene più video etichettati come violenti che non violenti). La figura 5.29 mostra invece la fase di addestramento del modello con ottimizzatore *Adagrad*.

```
Epoch 491/500
21/21 [==============================] - 3s 133ms/step - loss: 0.0569 - accuracy: 1.0000 - val_loss: 0.1681 - val_accuracy: 0.9570
Epoch 492/500
21/21 [==============================] - 3s 131ms/step - loss: 0.0555 - accuracy: 1.0000 - val_loss: 0.1682 - val_accuracy: 0.9570
Epoch 493/500
21/21 [==============================] - 3s 131ms/step - loss: 0.0537 - accuracy: 1.0000 - val_loss: 0.1695 - val_accuracy: 0.9516
Epoch 494/500
21/21 [==============================] - 3s 129ms/step - loss: 0.0522 - accuracy: 1.0000 - val_loss: 0.1682 - val_accuracy: 0.9570
Epoch 495/500
21/21 [==============================] - 3s 131ms/step - loss: 0.0585 - accuracy: 1.0000 - val_loss: 0.1677 - val_accuracy: 0.9570
Epoch 496/500
21/21 [==============================] - 3s 131ms/step - loss: 0.0528 - accuracy: 1.0000 - val_loss: 0.1676 - val_accuracy: 0.9570
Epoch 497/500
21/21 [==============================] - 3s 133ms/step - loss: 0.0526 - accuracy: 1.0000 - val_loss: 0.1681 - val_accuracy: 0.9570
Epoch 498/500
21/21 [==============================] - 2s 121ms/step - loss: 0.0565 - accuracy: 1.0000 - val_loss: 0.1677 - val_accuracy: 0.9570
Epoch 499/500
21/21 [==============================] - 2s 115ms/step - loss: 0.0538 - accuracy: 1.0000 - val_loss: 0.1677 - val_accuracy: 0.9570
Epoch 500/500
21/21 [==============================] - 2s 121ms/step - loss: 0.0597 - accuracy: 1.0000 - val_loss: 0.1686 - val_accuracy: 0.9516
12/12 [==============================] - 3s 103ms/step - loss: 0.1222 - accuracy: 0.9677
```

Figura 5.29: Fase di addestramento dell'architettura semplificata BiConvLSTM sull'AIRTLab Violence Dataset con l'ottimizzatore Adagrad.

A differenza dell'addestramento con *Adam* qui l'architettura non si comporta in modo anomalo ma tende ad adattarsi troppo ai dati utilizzati, come accade in molti casi visti in precedenza.

Nella figura 5.30 sono riportate le curve ROC ottenute dall'architettura.

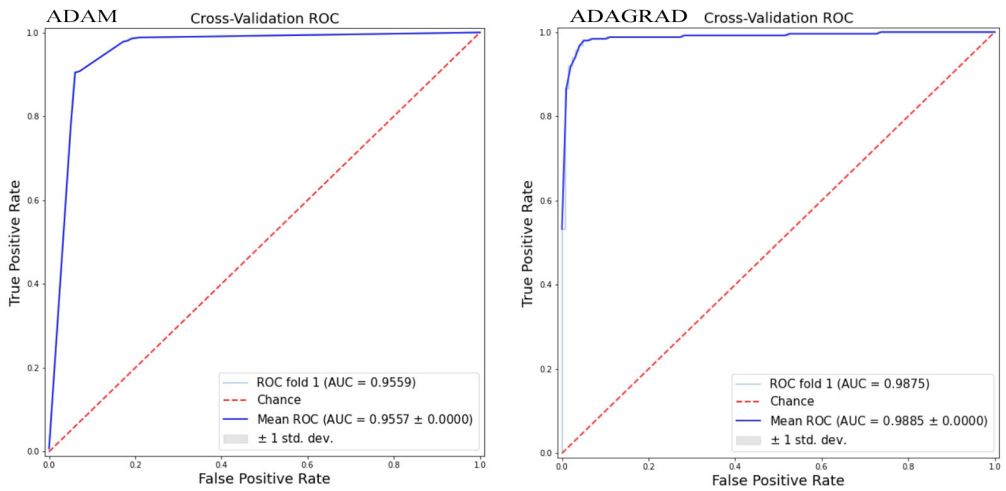

Figura 5.30: ROC curve dell'architettura semplificata BiConvLSTM sull'AIRTLAB Violence Dataset.

Con l'ottimizzatore *Adam* l'architettura ha ottenuto una *AUC* di 0.9557, mentre con l'ottimizzatore *Adagrad* il modello ha ottenuto una *AUC* di 0.9885.

5.3.3 Considerazioni Finali

Di seguito sono riportate le tabelle di comparazione delle performance delle varianti dell'architettura, una per ogni dataset.
Per il *Crowd Violence Dataset* i risultati sono evidenziati nella tabella 5.1.

Tabella 5.1: Tabella di comparazione delle performance ottenute dalle due varianti dell'architettura BiConvLSTM nel Crowd Violence Dataset.

	Accuracy	Sensitivity	Specificity	F1-Score
Arch.compl.	92.13 ± 1.86%	96.58 ± 0.79%	86.16 ± 7.13%	93.08 ± 0.92%
Arch.sempl.	96.54 ± 0.96%	97.50 ± 0.91%	95.31 ± 2.89%	97.06 ± 0.79%

Dalla tabella si nota che l'architettura semplificata ha ottenuto risultati decisamente migliori rispetto all'architettura completa. Come già accennato prima, il motivo di questi risultati potrebbe essere il numero ridotto di parametri dell'architettura semplificata, che rende il modello più leggero e meno incline all'overfitting.

Per l'*Hockey Fights Dataset* i risultati sono evidenziati nella tabella 5.2.

Tabella 5.2: Tabella di comparazione delle performance ottenute dalle due varianti dell'architettura BiConvLSTM nell'Hockey Fights Dataset.

	Accuracy	Sensitivity	Specificity	F1-Score
Arch.compl.	92.33 ± 1.35%	93.82 ± 1.97%	90.82 ± 1.63%	92.26 ± 1.82%
Arch.sempl.	95.05 ± 1.13%	94.92 ± 1.07%	95.02 ± 2.78%	94.99 ± 0.97%

Anche in questo caso l'architettura semplificata ha ottenuto risultati migliori rispetto a quella completa.

Per l'*AIRTLab Violence Dataset* i risultati sono evidenziati nella tabella 5.3.

Tabella 5.3: Tabella di comparazione delle performance ottenute dalle due varianti dell'architettura BiConvLSTM nell'AIRTLab Violence Dataset.

	Accuracy	Sensitivity	Specificity	F1-Score
Arch.compl.	83.43 ± 0.00%	94.62 ± 0.00%	63.78 ± 0.00%	87.92 ± 0.00%
Arch.sempl.	96.77 ± 0.00%	97.56 ± 0.00%	95.24 ± 0.00%	97.56 ± 0.00%

L'architettura semplificata nell'*AIRTLab Violence Dataset* si comporta decisamente meglio di quella completa, che invece non riesce ad ottenere buoni risultati nel dataset.

Nella tabella 5.4 sono infine riportati i risultati sull'accuratezza ottenuta da alcuni modelli presenti nello stato dell'arte sui dataset *Hockey Fights* e *Crowd Violence*, insieme a quelli ottenuti dall'architettura implementata in questo elaborato e a quelli ottenuti dall'architettura descritta nell'articolo [58].

Tabella 5.4: Tabella di comparazione delle performance ottenute da vari modelli dello stato dell'arte sui dataset Hockey Fights e Crowd Violence.

Metodo	Hockey	Crowd
MoSIFT+HIK[12]	90.9%	89.5%
ViF[17] & 82.9 ± 0.14%	-	
MoSIFT+KDE+Sparse Coding[13]	94.3 ± 1.68%	-
Deniz et al.[31]	90.1 ± 0%	98.0 ± 0.22%
Gracia et al.[22]	82.4 ± 0.4%	97.8 ± 0.4%
Substantial Derivative et al.[55]	-	96.89 ± 0.21%
Bilinski et al.[32]	93.4%	99%
MoIWLD[56]	96.8 ± 1.04%	-
ViF+OViF[18]	87.5 ± 1.7%	-
Three streams + LSTM[57]	93.9%	-
C3D+SVM	97.91 ± 0.47%	99.51 ± 0.48%
C3D E2E	96.89 ± 0.38%	98.38 ± 0.82%
LSTM E2E	99.01 ± 0.00%	94.33 ± 0.00%
BILSTM	99.75 ± 0.00%	98.26 ± 0.00%
Impl: Spatiotemporal Encoder	**92.33 ± 1.35%**	**92.13 ± 1.86%**
Impl: Spatial Encoder	**95.05 ± 1.13%**	**96.54 ± 0.96%**
Spatiotemporal Encoder[58]	**96.54 ± 1.01%**	**92.18 ± 3.29%**
Spatial Encoder[58]	**96.96 ± 1.08%**	**90.63 ± 2.82%**

Facendo un confronto tra i risultati ottenuti dall'implementazione in *Keras* dell'architettura completa (encoder spazio-temporale) e i risultati ottenuti dagli autori dell'articolo, si può concludere che nel *Crowd Violence Dataset* i due modelli dell'architettura si sono comportati in maniera molto simile, con un'accuratezza del 92.13% per l'implementazione e un'accuratezza del 92.18% per il modello originale, mentre i risultati differiscono di più nell'*Hockey Fights Dataset*, nel quale l'implementazione ha ottenuto un'accuratezza del 92.33% e il modello originale un'accuratezza del 96.54%. Queste differenze nascono dal fatto che l'implementazione dell'architettura in keras non è una "trasposizione" perfetta del modello descritto nell'articolo, nel quale gli autori tralasciano alcuni dettagli di implementazione e utilizzano tecniche di data augmentation che non vengono invece inserite nell'implementazione in *Keras* e che sicuramente influiscono sui risultati.
Confrontando invece i risultati ottenuti dall'implementazione dell'architettura semplificata (encoder spaziale) e i risultati ottenuti dagli autori dell'articolo, si può concludere che nell'*Hockey Fights Dataset* i modelli si sono comportati similmente, con un'accuratezza ottenuta dal modello originale del 96.96% che supera quella del 95.05% ottenuta dall'implementazione, mentre nell'*Crowd Violence Dataset* l'ar-

chitettura implementata in *Keras* ottiene risultati decisamente migliori (96.54%) dell'architettura originale, che si comporta discretamente (90.63%).

Una conclusione che si può trarre dai risultati ottenuti è che in molti casi può essere sufficiente estrarre delle informazioni spaziali robuste per ottenere buoni risultati nel campo della *Violence Detection*.

Capitolo 6

Generazione di Class Activation Maps

In questo capitolo verrà mostrato il processo dietro la generazione delle *Class Activation Maps* per i tre dataset già utilizzati precedentemente in questo elaborato. In una prima sezione verranno introdotte le *Class Activation Maps* e verrà spiegato il motivo della loro importanza nel campo del *Machine Learning*. Successivamente verrà mostrato il processo attraverso il quale queste *Class Activation Maps* sono state generate utilizzando la piattaforma *Google Colab* e un toolkit per la visualizzazione e il debugging di modelli creati tramite la libreria *Keras*. Infine verranno mostrati i risultati dell'elaborazione.

6.1 Class Activation Maps

Il *Machine Learning* sta guadagnando popolarità in maniera esponenziale e le sue applicazioni stanno aumentando in ogni dominio.

Con l'incremento delle performance dei sistemi di *Machine Learning*, l'interpretabilità di questi sistemi sta gradualmente diminuendo. Questo trend si è visto di più negli algoritmi di *Deep Learning*, i quali sono composti da milioni di parametri e centinaia di layers che li rendono estremamente difficili da interpretare.

Con l'aumento della ricerca nel campo del *Machine Learning* e particolarmente in quello del *Deep Learning* vengono fatti molti sforzi per cercare di risolvere il problema dell'interpretabilità dei modelli. Nel caso delle *Reti Neurali Convoluzionali* sono state scoperte varie tecnice di visualizzazione per cercare di raggiungere lo stadio di AI interpretabile, e uno di questi sono le *Class Activation Maps*.

Le *Class Activation Maps* (o CAM) sono una tecnica molto potente introdotta nell'articolo [61] e utilizzata in *Computer Vision* per problemi di classificazione. Permettono a chi ne fa uso di ispezionare l'immagine (o anche video) che deve essere categorizzata e capire quali regioni/pixels di quell'immagine hanno contribuito di più all'output finale del modello. Utilizzare questo tipo di tecniche è molto importante in quanto permette di capire su cosa un algoritmo di classificazione si focalizza quando elabora un input e produce una predizione. In questo modo è possibile per esempio "debuggare" un modello e trovare il motivo del perché le sue predizioni siano errate cercando di capire le regioni dell'immagine di input più significative per l'output del

modello, e modificare di conseguenza i layers che provocano inaccuratezza o cambiare l'approccio di preprocessing delle immagini.

Si prenda per esempio una rete costruita con il compito di riconoscere la presenza di persone all'interno di un'immagine, e si utilizzino le *Class Activation Maps* per capire in quali regioni dell'immagine il modello si focalizza per prendere una decisione. La figura 6.1 mostra un'immagine in cui il modello ha effettuato una predizione corretta.

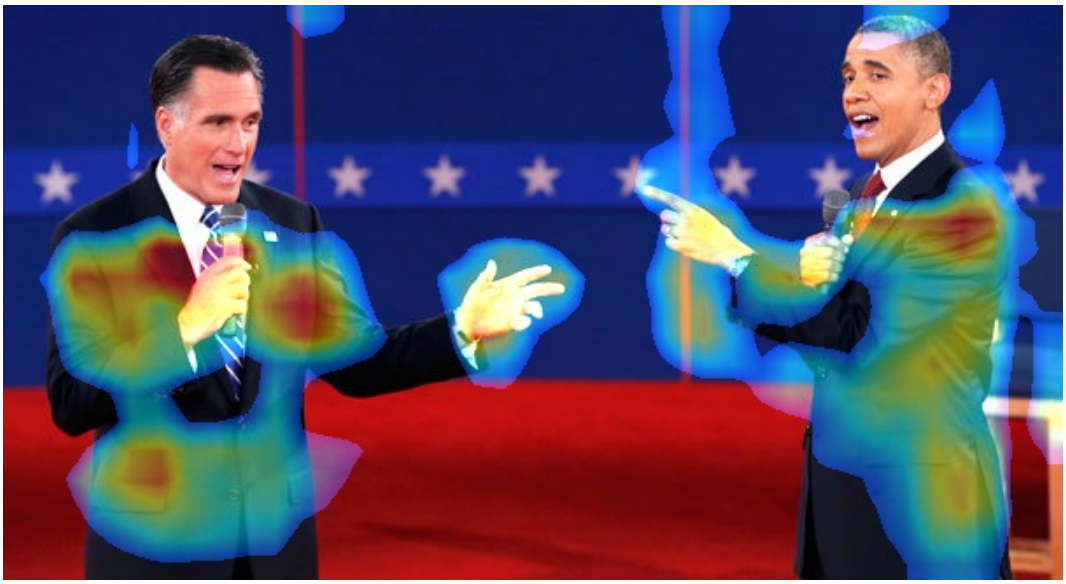

Figura 6.1: Esempio dell'applicazione delle CAM.

Anche se la rete ha preso la decisione corretta e ha riconosciuto la presenza di persone nell'immagine, è un pò deludente scoprire come per effettuare la decisione non abbia per niente utilizzato le regioni conteneti le facce delle persone, come invece ci si aspetterebbe. Magari si potrebbe pensare di addestrare il modello su più immagini in cui i volti sono esposti più chiaramente, così da abituarlo ad utilizzare le facce per future classificazioni più corrette. Le *Class Activation Maps* risultano molto utili in problemi simili.

Nell'articolo, gli autori propongono una tecnica per la generazione delle *Class Activation Maps* basata sull'utilizzo del *Global Average Pooling* nelle *Reti Neurali Convoluzionali*.

La procedura per generare queste mappe è illustrata nella figura 6.2.

L'architettura è molto simile a quella di una *CNN*. É composta da diversi layers convoluzionali, con il layer posto prima dell'ultimo layer di output che performa un *Global Average Pooling*. Le features ottenute vengono poi immesse nel layer fully-connected della rete governato dalla funzione di attivazione softmax che produce le probabilità richieste. L'importanza dei pesi rispetto a una categoria può essere rilevata proiettando indietro i pesi sulla feature map dell'ultimo layer convoluzionale.

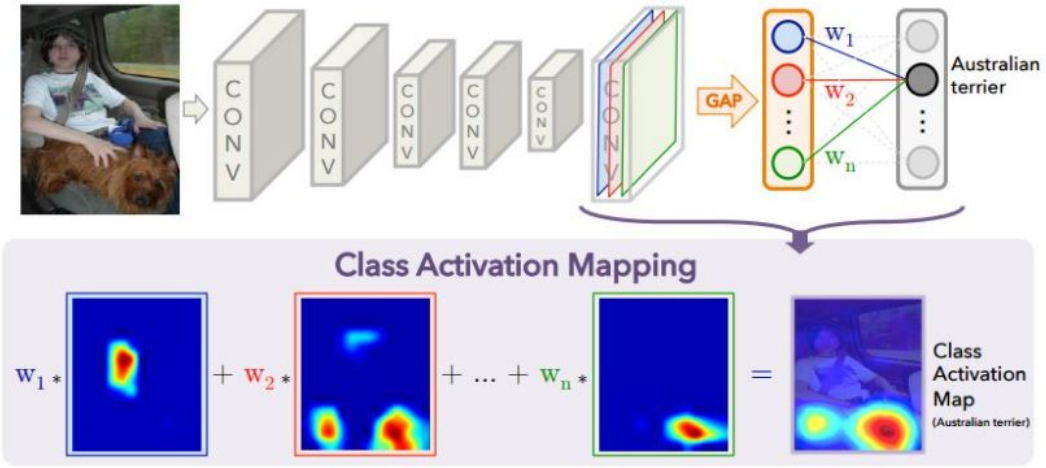

Figura 6.2: Processo di generazione delle Class Activation Maps

Un'operazione di *Global Average Pooling* (GAP) è un'operazione di pooling attraverso la quale si genera una feature map per ogni categoria corrispondente al problema di classificazione nell'ultimo layer convoluzionale. Al posto di aggiungere layers fully-connected dopo le feature maps, si prende la media di ogni feature map e il vettore risultante viene direttamente immesso nel layer di softmax.
Viene utilizzato il *Global Average Pooling* piuttosto che il *Global Max Pooling* (GMP) percé il *GAP* aiuta ad identificare l'intera estensione dell'oggetto mentre invece il *GMP* identifica solamente una porzione discriminante dell'oggetto. Infatti nel *Global Average Pooling* viene presa una media tra tutte le mappe di attivazione che permette di trovare tutte le possibili regioni discriminanti presenti in esse. Al contrario, il metodo del *Global Max Pooling* considera solamente la regione più discriminante. É per questo che il *GAP* ottiene risultati migliori del *GMP*.

Un problema nel generare *Class Activation Maps* con questo approccio è che la *CNN* sulla quale vogliamo generare le mappe deve avere la stessa struttura descritta qui sopra. Un approccio più versatile rispetto a quello appena discritto è quello del *Gradient-weighted Class Activation Mapping* (Grad-CAM). *Grad-CAM* è una tecnica introdotta in [62] che utilizza i gradienti di un qualsiasi "target concept" (per esempio "cane" in una rete di classificazione) che sfocia nell'ultimo layer convoluzionale per produrre una mappa di localizzazione che evidenzia le regioni importanti in un'immagine per la predizione del concept stesso. A differenza di altri approcci, *Grad-CAM* è applicabile a una grande varietà di *Reti Neurali Convoluzionali*, comprese *CNNs* con layers fully-connected (come *VGG*).
Più formalmente, per ottenere la *Grad-CAM* di un'immagine di interesse per prima cosa viene calcolato il gradiente dei logits (valori rappresentanti probabilità) della classe di interesse rispetto alle mappe di attivazione sull'ultimo layer convoluzionale e poi i gradienti vengono mediati su ciascuna feature map per ottenere un punteggio

di importanza:

$$\alpha_k^c = \overbrace{\frac{1}{Z}\sum_i\sum_j}^{global\ average\ pooling} \underbrace{\frac{\partial y^c}{\partial A_{ij}^k}}_{gradients\ via\ backprop} \tag{6.1}$$

Dove k è l'indice della mappa di attivazione nell'ultimo layer convoluzionale e c è la classe di interesse. Alpha rappresenta l'importanza della feature map k per la classe di interesse c. Infine, si moltiplica ogni mappa di attivazione per il suo punteggio di importanza (i.e. alpha) e si sommano i valori. Per considerare solo i pixels che hanno un'influenza positiva sul punteggio della classe di interesse, una non linearità $ReLU$ viene applicata alla sommatoria:

$$L_{Grad-CAM}^c = ReLU\Big(\underbrace{\sum_k \alpha_k^c A^k}_{linear\ combination}\Big) \tag{6.2}$$

Dove L è la *Grad-CAM* generata per la classe di interesse c.

L'immagine 6.3 mostra una visualizzazione prodotta con *Grad-CAM*.

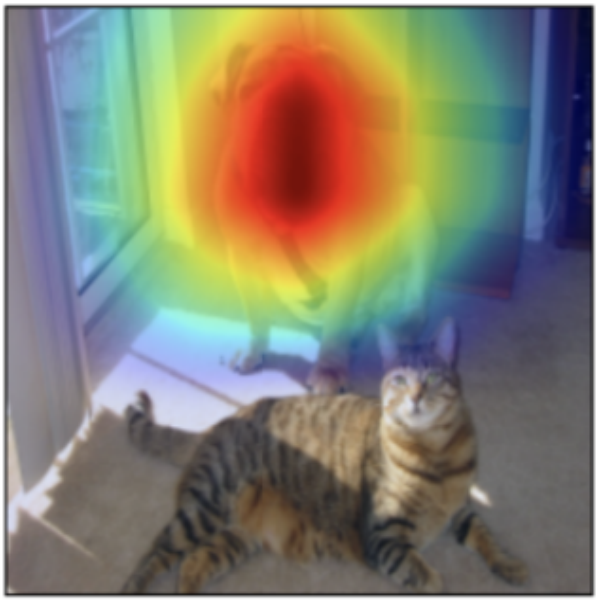

Figura 6.3: Visualizzazione con Grad-CAM di un'immagine d'esempio per la classe "Cane".

6.2 Implementazione e Risultati

6.2.1 Implementazione

Il principale strumento utilizzato in questo elaborato per la generazione delle *Class Activation Maps* è *Keras-vis*.
Keras-vis [63] è un toolkit per la visualizzazione e il debugging di modelli di *Machine Learning* costruiti con la libreria *Keras*. Le visualizzazioni correntemente supportate includono:

- Activation maximization
- Saliency maps
- Class activation maps

Keras-vis permette la generazione di *Class Activation Maps* tramite l'utilizzo della tecnica *Grad-CAM*.
Le tre funzioni che *Keras-vis* mette a disposizione per la generazione delle *CAMs* e che sono state utilizzate in questo elaborato sono **`_find_penultimate_layer`**, **`visualize_cam`** e **`visualize_cam_with_losses`**.

La funzione **`_find_penultimate_layer`** ricerca il penultimo layer convoluzionale o di pooling più vicino del modello passato come parametro. La funzione prende come parametri il modello preso in considerazione, l'indice del layer i cui filtri vogliono essere visualizzati (solitamente l'ultimo layer convoluzionale), e l'indice del layer precedente al layer i cui filtri vogliono essere visualizzati (solitamente il penultimo layer o convoluzionale o di pooling).
Nella figura 6.4 è mostrata la definizione della funzione.

```
def _find_penultimate_layer(model, layer_idx, penultimate_layer_idx):
    if penultimate_layer_idx is None:
        for idx, layer in utils.reverse_enumerate(model.layers[:layer_idx - 1]):
            if isinstance(layer, (Conv2D, Conv3D, MaxPooling1D, MaxPooling2D, MaxPooling3D)):
                penultimate_layer_idx = idx
                break
    if penultimate_layer_idx is None:
        raise ValueError('Unable to determine penultimate `Conv` or `Pooling` '
                         'layer for layer_idx: {}'.format(layer_idx))

    # Handle negative indexing otherwise the next check can fail.
    if layer_idx < 0:
        layer_idx = len(model.layers) + layer_idx
    if penultimate_layer_idx > layer_idx:
        raise ValueError('`penultimate_layer_idx` needs to be before `layer_idx`')

    return model.layers[penultimate_layer_idx]
```

Figura 6.4: Definizione della funzione _find_penultimate_layer.

Se l'indice del layer precedente al layer i cui filtri vogliono essere visualizzati è nullo la funzione prende come penultimate_layer_index l'indice del penultimo layer con-

voluzionale o di pooling più vicino al layer_idx. La funzione ritorna semplicemente l'oggetto del penultimo layer.

La funzione **`visualize_cam_with_losses`** genera una *Gradient Based Class Activation Map* usando i gradienti positivi del tensore passato in input rispetto agli errori pesati. Questa funzione va richiamata esplicitamente soltanto per particolari casi d'uso in cui vengono utilizzati degli errori pesati personalizzati. La funzione prende come parametri un tensore con forma "(samples, channels, image_dims...)" se il formato dell'immagine è "channels_first" oppure "(samples, image_dims..., channels)" se il formato dell'immagine è "channels_last", la lista delle tuple degli errori con i loro pesi, l'input del modello per il quale le mappe di attivazione devono essere visualizzate, il layer precedente a quello di cui le feature maps dovrebbero essere usate per calcolare i gradienti rispetto agli output del filtro, e un "gradient modifier" opzionale. Se il gradient modifier non viene elencato i gradienti rimarranno immodificati.
Nella figura 6.5 è mostrata la definizione della funzione.
Per prima cosa la funzione performa il metodo del gradiente nell'immagine passata in input rispetto gli errori definiti e normalizza i gradienti per una questione di stabilità numerica. Successivamente viene effettuata l'operazione di *Global Average Pooling* tra tutte le feature maps e poi viene generata la mappa moltiplicando gli output delle feature maps per i rispettivi pesi.

La funzione **`visualize_cam`** è la funzione generica che viene utilizzata per generale le *Class Activation Maps*. La funzione prende come parametri il modello preso in considerazione, l'indice del layer del modello di cui vogliamo visualizzare la mappa (solitamente l'ultimo layer convoluzionale), gli indici dei filtri all'interno del layer che vogliamo vengano massimizzati (se non viene specificato nessun valore tutti i filtri del layer verranno visualizzati), l'immagine per cui vogliamo visualizzare le mappe d'attivazione, l'indice del layer precedente al layer di cui vogliamo visualizzare la mappa, un "backprop_modifier" opzionale usato per modificare il processo di back propagation, e un "grad_modifier" opzionale. Se il gradient modifier non viene elencato i gradienti rimarranno immodificati.
Nella figura 6.6 è mostrata la definizione della funzione.
La funzione applica il backprop_modifier se settato, poi richiama la funzione **`_find_penultimate_layer`**. Successivamente viene richiamata la funzione "ActivationMaximization" che serve per massimizzare l'attivazione di un set di filtri all'interno di un particolare layer. Questo permette di capire quali tipi di pattern di input attivano un particolare filtro. Per esempio, potrebbe esserci un filtro per occhi che si attiva quando c'è una presenza di occhi all'interno dell'immagine in input. Usando la funzione per un layer convoluzionale per esempio, si può visualizzare quale pattern attiva un particolare filtro. Questo potrebbe essere utile per scoprire cosa i filtri calcolano. Infine viene richiamata la funzione **`visualize_cam_with_losses`**

```python
def visualize_cam_with_losses(input_tensor, losses,seed_input, penultimate_layer,grad_modifier=None):
    penultimate_output = penultimate_layer.output
    opt = Optimizer(input_tensor, losses, wrt_tensor=penultimate_output, norm_grads=False)
    _, grads, penultimate_output_value = opt.minimize(seed_input, max_iter=1, grad_modifier=grad_modifier, verbose=False)

    # For numerical stability. Very small grad values along with small penultimate_output_value can cause
    # w * penultimate_output_value to zero out, even for reasonable fp precision of float32.
    grads = grads / (np.max(grads) + K.epsilon())

    # Average pooling across all feature maps.
    # This captures the importance of feature map (channel) idx to the output.
    channel_idx = 1 if K.image_data_format() == 'channels_first' else -1
    other_axis = np.delete(np.arange(len(grads.shape)), channel_idx)
    weights = np.mean(grads, axis=tuple(other_axis))

    # Generate heatmap by computing weight * output over feature maps
    output_dims = utils.get_img_shape(penultimate_output)[2:]
    heatmap = np.zeros(shape=output_dims, dtype=K.floatx())
    for i, w in enumerate(weights):
        if channel_idx == -1:
            heatmap += w * penultimate_output_value[0, ..., i]
        else:
            heatmap += w * penultimate_output_value[0, i, ...]

    # ReLU thresholding to exclude pattern mismatch information (negative gradients).
    heatmap = np.maximum(heatmap, 0)

    # The penultimate feature map size is definitely smaller than input image.
    input_dims = utils.get_img_shape(input_tensor)[2:]
    #heatmap = imresize(heatmap, input_dims, interp='bicubic', mode='F')

    # Normalize and create heatmap.
    heatmap = utils.normalize(heatmap)

    target_size = input_dims

    mapa = heatmap

    mapa_ex = np.zeros((target_size[0], target_size[1], mapa.shape[2]))
    for i in range(mapa.shape[2]):
        mapa_ex[:, :, i] = cv2.resize(mapa[:, :, i],(target_size[1], target_size[0]), interpolation = cv2.INTER_CUBIC)

    mapa_ex2 = np.zeros(target_size)
    for i in range(len(mapa_ex)):
        mapa_ex2[i, :, :] = cv2.resize(mapa_ex[i, :, :],(target_size[2], target_size[1]), interpolation = cv2.INTER_CUBIC)

    return mapa_ex2
```

Figura 6.5: Definizione della funzione visualize_cam_with_losses.

```python
def visualize_cam(model, layer_idx, filter_indices,
                  seed_input, penultimate_layer_idx=None,
                  backprop_modifier=None, grad_modifier=None):

    if backprop_modifier is not None:
        modifier_fn = get(backprop_modifier)
        model = modifier_fn(model)

    penultimate_layer = _find_penultimate_layer(model, layer_idx, penultimate_layer_idx)

    # `ActivationMaximization` outputs negative gradient values for increase in activations. Multiply with -1
    # so that positive gradients indicate increase instead.
    losses = [
        (ActivationMaximization(model.layers[layer_idx], filter_indices), -1)
    ]
    return visualize_cam_with_losses(model.input, losses, seed_input, penultimate_layer, grad_modifier)
```

Figura 6.6: Definizione della funzione visualize_cam.

che produce la mappa con i parametri calcolati.

Un'altra funzione utilizzata che non fa però parte del toolkit *Keras-vis* è **`get_overlap`**.

Questa funzione è stata definita riadattando la funzione "`save_and_display_gradcam`" fornita direttamente dalla documentazione di *Keras* per visualizzare mappe di attivazione [64]. La funzione serve per creare una sovrapposizione tra un frame e la *CAM* generata. Prende come parametri l'immagine in input, la mappa precedentemente generata e un valore di trasparenza.
Nella figura 6.7 è mostrata la definizione della funzione.

```
def get_overlap(img, heatmap, alpha=0.4):
    # Rescale heatmap to a range 0-255
    heatmap = np.uint8(255 * heatmap)

    # Use jet colormap to colorize heatmap
    jet = cm.get_cmap("jet")

    # Use RGB values of the colormap
    jet_colors = jet(np.arange(256))[:, :3]
    jet_heatmap = jet_colors[heatmap]

    # Create an image with RGB colorized heatmap
    jet_heatmap = preprocessing.image.array_to_img(jet_heatmap)
    jet_heatmap = jet_heatmap.resize((img.shape[1], img.shape[0]))
    jet_heatmap = preprocessing.image.img_to_array(jet_heatmap)

    # Superimpose the heatmap on original image
    superimposed_img = jet_heatmap * alpha + img
    superimposed_img = preprocessing.image.array_to_img(superimposed_img)

    return np.array(superimposed_img)
```

Figura 6.7: Definizione della funzione get_overlap.

Per prima cosa la funzione riscala la mappa in un range tra 0 e 255, poi la mappa d'attivazione viene colorata utilizzando le jet colormaps della libreria *matplotlib.cm*. Infine la mappa viene sovrapposta all'immagine con il valore di trasparenza passato come parametro.

Il processo completo di generazione delle *Class Activation Maps* utilizzato in questo elaborato per i video dei dataset *Hockey Fights*, *Crowd Violence* e *AIRTLab Violence* è il seguente:

1. Viene definito il modello che si vuole utilizzare per la generazione delle mappe: in questo elaborato il modello utilizzato è la rete *C3D End-To-End* introdotta nel capitolo 4.

2. I video del dataset vengono preprocessati per poter essere passati in input alla rete che viene successivamente addestrata sul dataset considerato e i pesi salvati

per essere poi utilizzati nelle predizioni dei singoli video di cui si vorranno generare le *CAMs*.

3. Vengono generate le *CAMs*: per prima cosa viene istanziato il modello con i pesi relativi al dataset considerato salvati precedentemente; il video di cui si vuole ottenere la mappa viene preprocessato e passato in input al modello che ne effettua la predizione; viene generata la mappa di attivazione per il video considerato utilizzando le funzioni di *Keras-vis* introdotte precedentemente; viene creata la sovrapposizione tra ogni frame del video e la relativa mappa tramite la funzione `get_overlap`; il video viene ricomposto di ogni suo frame per ottenere una clip finale del video iniziale con sovrapposte le *Class Activation Maps* che mostrano le zone in cui il modello *C3D End-To-End* si è incentrato per la predizione della classe di appartenenza del video.

Il processo completo della creazione delle mappe utilizzato in questo lavoro è mostrato nella figura 6.8.
La figura mostra la creazione delle mappe per ogni singolo video dell'*AIRTLab Violence Dataset*, ma lo stesso procedimento vale anche per i dataset *Hockey Fights* e *Crowd Violence*. Alcune considerazioni: la funzione `getLayerIndexByName` serve per recuperare l'indice del layer il cui nome viene passato come parametro; ogni video viene diviso in chunks da 16 frames e ogni chunk viene processato singolarmente per la generazione delle mappe; il layer scelto per essere passato alla funzione `visualize_cam` e di cui vogliamo visualizzare i filtri è l'ultimo layer convoluzionale del modello *C3D End-To-End*, che in particolare sarebbe il layer con nome "conv5b" della rete *C3D*; vengono inoltre passati alla funzione tutti i filtri del layer e tutti i frames del chunk che deve essere processato, infatti il risultato di `visualize_cam` non è una singola mappa d'attivazione, ma è un insieme di 16 mappe, una per ogni frame del chunk processato; il video viene ricomposto dei suoi frames sovrapposti alle heatmaps generate tramite la libreria *OpenCV*, utilizzando la classe *VideoWriter*.

6.2.2 Risultati e Considerazioni

Nella figura 6.8 viene mostrato un esempio di *Class Activation Map* generata per un frame di una clip del dataset *Hockey Fights*.
L'immagine contiene 3 visualizzazioni: la prima è la visualizzazione della *Class Activation Map* generata per il frame; la seconda è il frame preso in considerazione; la terza è la sovrapposizione tra il frame e la mappa colorata tramite la jet colormap per renderla migliore esteticamente.
Grazie a questa visualizzazione possiamo capire su cosa si concentra il modello *C3D End-To-End* per fare la sua predizione. In questo caso il modello ha predetto correttamente la classe di appartenenza del chunk di cui questo frame fa parte riconoscendo la presenza di violenza all'interno della clip. Guardando la sovrapposizione finale nella figura si nota come il modello si incentra sulle zone in cui sono presenti i soggetti

```python
#function the get the index of a layer passed by parameter
def getLayerIndexByName(model, layername):
    for idx, layer in enumerate(model.layers):
        if layer.name == layername:
            return idx

datasetBasePath = 'drive/MyDrive/VDT/datarepo/airtlab_dataset'
heatmapsBasePath = 'drive/MyDrive/VDT/Airtlab_Maps'
mmapDatasetBasePath = 'video'
folders = ['fights', 'no_fights']
cams = ['cam1', 'cam2']


for folder in folders:
  for cam in cams:
    heatmapsPath = os.path.join(heatmapsBasePath, folder,cam) #path where the maps will be saved
    path = os.path.join(datasetBasePath, folder,cam) #path of the videos in the dataset
    videofiles = os.listdir(path)
    for videofile in videofiles:
      model = getC3DCNNModel('Adagrad')
      model.load_weights('drive/MyDrive/VDT/weights/C3DAirtWeights')
      heatPath = os.path.join(heatmapsPath, videofile) #path of the video's heatmap
      filePath = os.path.join(path, videofile) #path of the video
      filename = os.path.splitext(videofile)[0]

      #preprocess the video
      preprocessVideo(filePath, mmapDatasetBasePath,folder, False)

      #get preprocessed video
      chunk_number = count_video_chunks(filePath)
      chunk_size = 16
      X = np.memmap(os.path.join(mmapDatasetBasePath, 'samples.mmap'), mode='r', dtype=np.float32, shape=(chunk_number, chunk_size, 112, 112, 3)
      y = np.memmap(os.path.join(mmapDatasetBasePath, 'labels.mmap'), mode='r', dtype=np.int8, shape=(chunk_number))

      #prediction
      preds = model.predict(X).ravel()
      preds = np.round(preds).astype(int)
      print(X.shape)
      print(y)
      for i in range(chunk_number):
        if preds[i] == 1:
          print("prediction: fight")
        else:
          print("prediction: no_fight")

        if y[i] == 1:
          print("was: fight")
        else:
          print("was: no_fight")
        print("------------------")

      #get frames of the video
      video = cv2.VideoCapture(filePath)
      videoFrames = []
      fps = int(video.get(cv2.CAP_PROP_FPS))
      codec = int(video.get(cv2.CAP_PROP_FOURCC))
      width = int(video.get(cv2.CAP_PROP_FRAME_WIDTH))
      height = int(video.get(cv2.CAP_PROP_FRAME_HEIGHT))

      while True:
          ret, img = video.read()
          if not ret:
              break
          videoFrames.append(cv2.cvtColor(img, cv2.COLOR_RGB2BGR))

      #grad maps
      overlays = []
      for j in range(chunk_number):
        layer_idx = getLayerIndexByName(model, 'conv5b') #index of last conv layer
        filter_indices = list(range(0,512)) #indices of the conv layer filters
        heatmap = visualize_cam(model, layer_idx, filter_indices,X[j])

        for i in range(chunk_size):
          fore = cv2.resize(heatmap[i], (width, height))
          back = videoFrames[i + 16*j]
          overlay = get_overlap(back, fore, 0.7)
          overlays.append(overlay)

      #write the video
      size = (width,height)
      video = cv2.VideoWriter(heatPath,cv2.VideoWriter_fourcc(*"mp4v"), fps, size)
      for frame in overlays:
        BGR_frame = cv2.cvtColor(frame, cv2.COLOR_RGB2BGR)
        video.write(BGR_frame)
      video.release()
      print("video: " + filename + "," + " from folder: " + folder + "/" + cam +  " saved")
```

Figura 6.8: Processo di creazione delle Class Activation Maps.

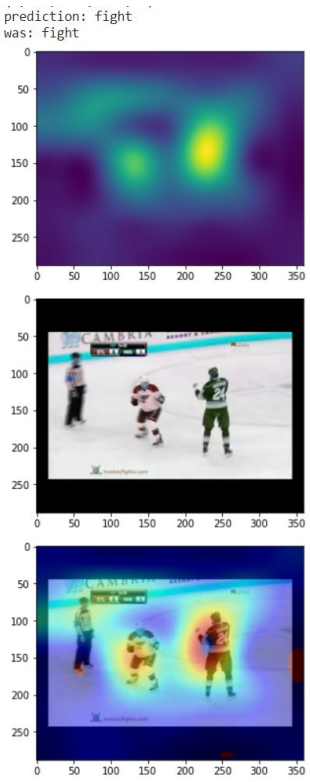

Figura 6.9: Class Activation Map di un frame di un video del dataset Hockey Fights.

del video e in particolare sulle zone del corpo che nella clip subiscono variazioni brusche nel movimento, come le braccia. Il modello ha posto la sua attenzione anche nell'arbitro presente nel frame, ma ha saputo correttamente assegnargli meno importanza rispetto ai due soggetti che sono i protagonisti della scena violenta. Dall'analisi di questa rappresentazione sembrerebbe quindi che il modello effettua le sue predizioni in maniera corretta e si incentra sulle zone giuste.

Nella figura 6.10 sono rappresentati degli esempi di frames estratti da due video dell'*Hockey Fights Dataset*. I tre frames a sinistra sono estratti da un video violento, mentre i tre frames a destra sono estratti da un video non violento.
Guardando i frames del video violento si nota che anche in questo caso il modello *C3D End-To-End* si concentra soprattutto nei soggetti della lotta e in particolare sulle zone del corpo che subiscono variazioni brusche nei movimenti, come braccia e gambe.
Nei frames del video non violento si vede come il modello tenda comunque a dare importanza alle zone contenenti nel complesso i giocatori presenti nella clip, soprattutto quelli che si muovono più velocemente (infatti il modello non si interessa di quei giocatori più statici nella clip).

Nella figura 6.11 sono rappresentati degli esempi di frames estratti da due video del

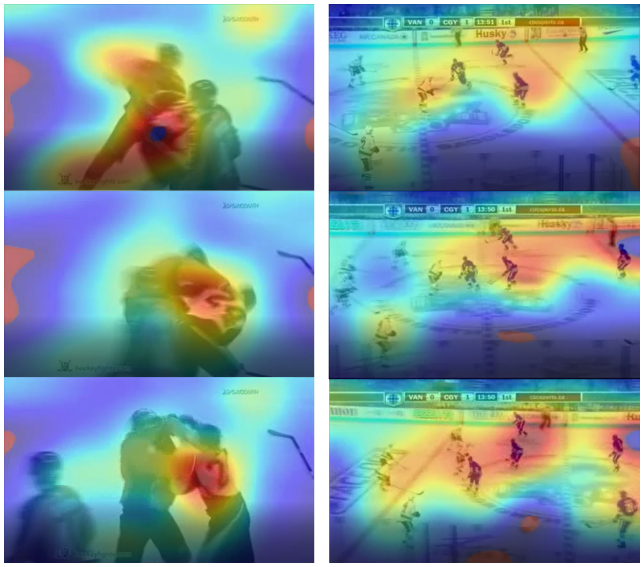

Figura 6.10: Frames sovrapposti alle mappe d'attivazione di due video del dataset Hockey Fights.

Crowd Violence Dataset. I tre frames a sinistra sono estratti da un video violento, mentre i tre frames a destra sono estratti da un video non violento.

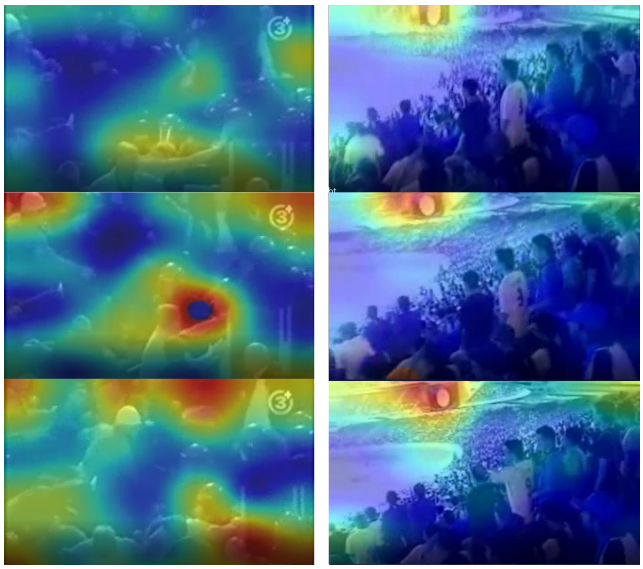

Figura 6.11: Frames sovrapposti alle mappe d'attivazione di due video del dataset Crowd Violence.

Sfortunatamente il *Crowd Violence Dataset* è molto più difficile da visualizzare dell'*Hockey Fights Dataset* in quanto le clip contenute al suo interno sono tutte molto più caotiche e di bassa qualità. Guardando i tre frames del video violento non si riesce infatti a capire molto, anche se comunque sembrerebbe che il modello si concentri sempre su azioni brusche provenienti dalla folla.

Un risultato interessante è invece quello dei frames del video non violento. In questo video il modello non pone per niente la sua attenzione nella folla presente, ma piuttosto tende a dare importanza a una zona che sembrerebbe non avere nulla a che fare con la ricerca di violenza all'interno del video. Il modello riesce comunque a etichettare in maniera corretta il video, quindi si potrebbe concludere che non trovando alcun movimento brusco all'interno della clip che possa essere assimilato a un'azione violenta, la rete si disinteressa completamente dei soggetti del video.

Nella figura 6.12 sono rappresentati degli esempi di frames estratti da due video dell'*AIRTLab Violence Dataset*. I tre frames a sinistra sono estratti da un video violento, mentre i tre frames a destra sono estratti da un video non violento.

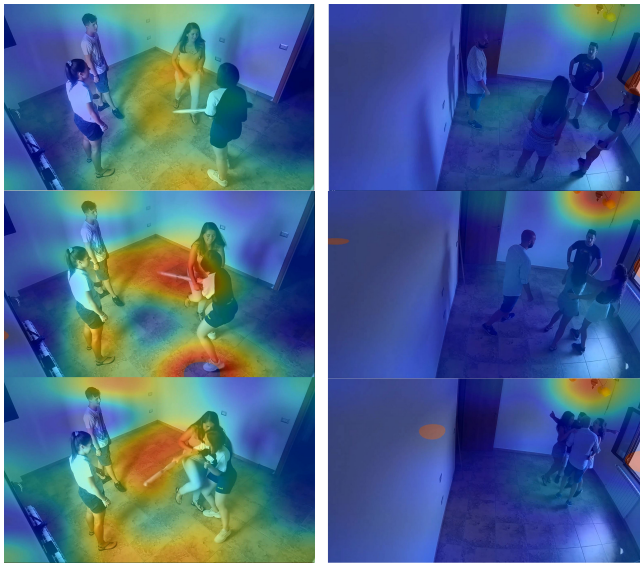

Figura 6.12: Frames sovrapposti alle mappe d'attivazione di due video del dataset AIRTLab Violence.

Nei frames del video violento si nota come anche questa volta il modello ha posto la sua attenzione sulle zone del video che subiscono variazioni brusche nel movimento, in particolare in questo caso la rete dà importanza alla mazza che uno dei soggetti tiene in mano.
Nei frames del video non violento si vede che anche in questo caso il modello trascura completamente i soggetti del video, anche se alcuni dei loro movimenti possono essere scambiati per azioni violente, e si incentra su una zona contenente un lampadario. Una considerazione da fare è che però questa volta il modello tende a porre l'attenzione in quella zona in quasi tutte le clips registrate in questa angolazione, sia violente che non. Questo comportamento è molto strano e sembrerebbe che il modello si concentri su zone errate del video. Un'ipotesi che si può trarre è che il modello si sia adattato in maniera sbagliata ai video durante la fase di addestramento, ponendo importanza solamente alla zona contenente il lampadario in tutte le clips che hanno quell'angolazione.

Capitolo 7

Conclusioni e Sviluppi Futuri

Questo lavoro ha riguardato la valutazione e comparazione di diverse reti neurali profonde per il rilevamento automatico di scene di violenza all'interno di video.

In primo luogo sono state testate tre architetture basate su reti neurali convoluzionali con lo scopo di valutarne le performance nei dataset più utilizzati nello stato dell'arte e nell'AIRTLab Dataset, e per scoprire quali ottimizzatori ottengono i migliori risultati per ognuna di queste architetture nell'AIRTLab Dataset.
I test delle architetture hanno riportato risultati molto interessanti: tutte le architetture hanno ottenuto i loro migliori risultati con ottimizzatori diversi, questo significa che non esiste un'ottimizzatore universalmente migliore degli altri quando si tratta di addestrare reti per il riconoscimento di violenza, ed è quindi importante testare le proprie architetture con più ottimizzatori per capire quale meglio si adatta al modello testato; tutte le architetture testate sull'AIRTLab Dataset hanno avuto più difficoltà nel riconoscere correttamente le scene non violente, risultato che conferma la capacità del dataset di fuorviare i modelli di Violence Detection con scene ambigue e movimenti bruschi che possono essere scambiati per scene violente; tutti i modelli testati hanno ottenuto buoni risultati nei dataset utilizzati, in particolare il modello LSTM End-To-End è risultato il migliore nei dataset Hockey Fights e AIRTLab Violence, mentre il modello C3D+SVM ha avuto le migliori performance nel Crowd Violence Dataset.

Successivamente il lavoro ha trattato l'implementazione in Keras e il test di un'architettura presente nello stato dell'arte e basata su VGG-13 e su una rete BiConvLSTM (encoder spazio-temporale), e di una sua versione semplificata che non fa uso della rete BiConvLSTM (encoder spaziale), per testarne le performance e confrontare i risultati con quelli ottenuti dai creatori della rete.
Con grande sorpresa, l'implementazione dell'architettura completa ha ottenuto risultati complessivamente peggiori dell'implementazione dell'architettura semplificata su tutti e tre i dataset utilizzati per il test. Questi risultati potrebbero star a significare che in molti casi può essere sufficiente estrarre delle informazioni spaziali robuste per ottenere buoni risultati. Le due architetture implementate hanno ottenuto risultati in alcuni casi simili e in altri o leggermente peggiori o leggermente migliori di quelle

ottenute dagli autori dell'articolo, concludendo che l'implementazione dell'architettura non è stata perfetta ma comunque una buona approssimazione dell'originale.

In un'ultima fase il lavoro ha riportato un'introduzione alle Class Activation Maps e alla loro importanza, e un'implementazione in keras di un metodo per generarle tramite l'utilizzo del Grad-CAM, per il modello C3D End-To-End.
La generazione delle mappe ha reso possibile capire come il modello si comporta e su quali zone dei video la rete si incentra per effettuare la predizione della sua classe di appartenenza. Un risultato curioso è stato osservare come la rete si comporta quando predice video non violenti, nei quali in molti casi tende a trascurare completamente o quasi i soggetti principali dei video.

A partire dai risultati ottenuti in questo elaborato, sono diversi gli sviluppi futuri che potrebbero essere presi in considerazione, alcuni dei quali sono di seguito elencati:

- Provare ad addestrare la rete C3D End-To-End in modo tale da evitare che il modello dia importanza ad elementi dei video che non riguardano l'azione principale svolta, come accade per il lampadario in alcune clips dell'AIRTLab Violence Dataset.
- Implementare e testare nuovi modelli per la Violence Detection presenti nello stato dell'arte e confrontarne le performance con quelle ottenute dai modelli testati in questo elaborato.
- Convertire i modelli da classificatori binari a classificatori multi-classe in modo da creare architetture che riescano a distinguere tra diverse categorie di violenza.
- Provare ad aggiungere meccanismi di data augmentation al preprocessing dei dataset per il modello VGG-13 + BiConvLSTM, per ottenere un'implementazione più esatta dell'architettura e per osservare eventuali miglioramenti delle performance.

Bibliografia

[1] Vocabolario treccani: definizione di "violenza". https://www.treccani.it/vocabolario/violenza/. [Consultata il 21/04/2021].

[2] Krug et al. *"World report on violence and health"*. World Health Organization, 2002.

[3] Wikipedia. https://en.wikipedia.org/wiki/Violence/. [Consultata il 22/04/2021].

[4] M.S. Ryoo J.K. Aggarwal. *"Human activity analysis: a review"*, page 15. ACM Comput. Surv., 2011.

[5] Shugang Zhang, Z. Wei, Jie Nie, L. Huang, S. Wang, and Z. Li. "a review on human activity recognition using vision-based method". *Journal of Healthcare Engineering*, 2017, 2017.

[6] S. Accattoli. *"Riconoscimento in tempo reale di scene di violenza utilizzando il Deep Learning"*. UNIVPM, A.A. 2017/2108.

[7] Muhammad Ramzan, Adnan Abid, Hikmat Ullah Khan, Shahid Mahmood Awan, Amina Ismail, Muzamil Ahmed, Mahwish Ilyas, and Ahsan Mahmood. "a review on state-of-the-art violence detection techniques". *IEEE Access*, 7:107560–107575, 2019.

[8] David G. Lowe. "object recognition from local scale-invariant features". *IEEE Computer Society*, 1999.

[9] Mingyu Chen and A. Hauptmann. "mosift: Recognizing human actions in surveillance videos". *InformediaTRECVID*, 2009.

[10] Tobias Senst, Volker Eiselein, Alexander Kuhn, and Thomas Sikora. "crowd violence detection using global motion-compensated lagrangian features and scale-sensitive video-level representation". *IEEE Transactions on Information Forensics and Security*, 12(12):2945–2956, 2017.

[11] Febin, I.P., Jayasree, K. & Joy, and P.T. "violence detection in videos for an intelligent surveillance system using mobsift and movement filtering algorithm". *Pattern Anal Applic*, 2019.

[12] Enrique Bermejo Nievas, Oscar Deniz Suarez, Gloria Bueno García, and Rahul Sukthankar. Violence detection in video using computer vision techniques. *Springer Berlin Heidelberg*, pages 332–339, 2011.

[13] L. Xu, Chen Gong, Jie Yang, Q. Wu, and Lixiu Yao. "violent video detection based on mosift feature and sparse coding". *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3538–3542, 2014.

[14] Laptev and Lindeberg. "space-time interest points". *Computational Vision and Active Perception Laboratory (CVAP)*, pages 432–439 vol.1, 2003.

[15] C. G. Harris and M. Stephens. "a combined corner and edge detector". *Alvey Vision Conference*, 1988.

[16] W. A. F ¨orstner and E. G ¨ulch. " a fast operator for detection and precise location of distinct points, corners and centers of circular features". *ISPRS*, 1988.

[17] Tal Hassner, Yossi Itcher, and Orit Kliper-Gross. "violent flows: Real-time detection of violent crowd behavior". *2012 IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, pages 1–6, 2012.

[18] Yuan Gao, Hong Liu, Xiaohu Sun, Can Wang, and Yi Liu. "violence detection using oriented violent flows". *Image and Vision Computing*, 48-49:37–41, 2016.

[19] Vicente Machaca Arceda, Karla Fernández Fabián, and Juan Carlos Gutíerrez. "real time violence detection in video". In *International Conference on Pattern Recognition Systems (ICPRS-16)*, pages 1–7, 2016.

[20] Piyush Vashistha, Charul Bhatnagar, and Mohd Aamir Khan. "an architecture to identify violence in video surveillance system using vif and lbp". In *2018 4th International Conference on Recent Advances in Information Technology (RAIT)*, pages 1–6, 2018.

[21] V.E. Machaca Arceda, K.M. Fernández Fabián, P.C. Laguna Laura, J.J. Rivera Tito, and J.C. Gutiérrez Cáceres. "fast face detection in violent video scenes". *Electronic Notes in Theoretical Computer Science*, 329:5–26, 2016. CLEI 2016 - The Latin American Computing Conference.

[22] Ismael Serrano Gracia, Oscar Deniz Suarez, Gloria Bueno Garcia, and Tae-Kyun Kim. "fast fight detection". *PLoS ONE*, 10, 2015.

[23] Pedro Canotilho Ribeiro, Romaric Audigier, and Quoc Cuong Pham. "rimoc, a feature to discriminate unstructured motions: Application to violence detection for video-surveillance". *Computer Vision and Image Understanding*, 144:121–143, 2016. Individual and Group Activities in Video Event Analysis.

[24] Jing Wang, Zhijie Xu, and Y. Liu. "spatio-temporal volume-based shape modelling for video event detection". *2013 19th International Conference on Automation and Computing*, pages 1–6, 2013.

[25] Mehrsan Javan Roshtkhari and Martin D. Levine. "an on-line, real-time learning method for detecting anomalies in videos using spatio-temporal compositions". *Computer Vision and Image Understanding*, 117(10):1436–1452, 2013.

[26] Sarita Chaudhary, Mohd Aamir Khan, and Charul Bhatnagar. "multiple anomalous activity detection in videos". *Procedia Computer Science*, 125:336–345, 2018. The 6th International Conference on Smart Computing and Communications.

[27] Adi Nurhadiyatna, Wisnu Jatmiko, Benny Hardjono, Ari Wibisono, Ibnu Sina, and Petrus Mursanto. "background subtraction using gaussian mixture model enhanced by hole filling algorithm (gmmhf)". *2013 IEEE International Conference on Systems, Man, and Cybernetics*, 125:4006–4011, 2013.

[28] Corinna Cortes and V. Vapnik. "support-vector networks". *Machine Learning*, 20:273–297, 2004.

[29] Asa Ben-Hur, David Horn, Hava T. Siegelmann, and Vladimir Vapnik. "support vector clustering". *J. Mach. Learn. Res.*, 2:125–137, 2002.

[30] Tao Zhang, Zhijie Yang, Wenjing Jia, Baoqing Yang, Jie Yang, and Xiangjian He. "a new method for violence detection in surveillance scenes". *Multimedia Tools Appl.*, 75(12):7327–7349, 2016.

[31] Oscar Deniz, Ismael Serrano, Gloria Bueno, and Tae-Kyun Kim. Fast violence detection in video. *"2014 International Conference on Computer Vision Theory and Applications (VISAPP)"*, 2:478–485, 2014.

[32] Piotr Bilinski and Francois Bremond. "human violence recognition and detection in surveillance videos". *2016 13th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pages 30–36, 2016.

[33] Chunhui Ding, Shouke Fan, Ming Zhu, Weiguo Feng, and Baozhi Jia. Violence detection in video by using 3d convolutional neural networks. *Springer International Publishing*, pages 551–558, 2014.

[34] Swathikiran Sudhakaran and Oswald Lanz. "learning to detect violent videos using convolutional long short-term memory". *2017 14th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pages 1–6, 2017.

[35] F. U. M. Ullah, Amin Ullah, Khan Muhammad, I. Haq, and S. Baik. "violence detection using spatiotemporal features with 3d convolutional neural network". *Sensors (Basel, Switzerland)*, 19, 2019.

[36] Al-Maamoon R. Abdali and Rana F. Al-Tuma. "robust real-time violence detection in video using cnn and lstm". *2019 2nd Scientific Conference of Computer Sciences (SCCS)*, pages 104–108, 2019.

[37] Rohit Halder and Rajdeep Chatterjee. "cnn-bilstm model for violence detection in smart surveillance". *SN Computer Science*, 1, 06 2020.

[38] francois Chollet. *"Deep Learning with Python"*. Manning Pubns Co, 2017.

[39] Wikipedia. https://it.wikipedia.org/wiki/Overfitting. [Consultata il 08/05/2021].

[40] Wikipedia. https://it.wikipedia.org/wiki/Funzione_softmax. [Consultata il 08/05/2021].

[41] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "dropout: A simple way to prevent neural networks from overfitting". *J. Mach. Learn. Res.*, 15(1), 2014.

[42] Du Tran, Lubomir Bourdev, Rob Fergus, Lorenzo Torresani, and Manohar Paluri. "learning spatiotemporal features with 3d convolutional networks". *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 4489–4497, 2015.

[43] Shuiwang Ji, Wei Xu, Ming Yang, and Kai Yu. "3d convolutional neural networks for human action recognition". *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(1):221–231, 2013.

[44] Sepp Hochreiter and Jürgen Schmidhuber. "long short-term memory". *Neural computation*, 9:1735–80, 12 1997.

[45] Xingjian Shi, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai-kin Wong, and Wang-chun Woo. "convolutional lstm network: A machine learning approach for precipitation nowcasting". *MIT Press*, 2015.

[46] [online] google colaboratory. https://colab.research.google.com/notebooks/intro.ipynb.

[47] [online] jupyter notebook. https://jupyter.org/.

[48] [online] tensorflow. https://www.tensorflow.org/.

[49] [online] keras. https://keras.io/.

[50] [online] scikit-learn. https://scikit-learn.org/.

[51] [online] opencv. https://opencv.org/.

[52] [online] airtlab violence dataset. https://github.com/airtlab/A-Dataset-for-Automatic-Violence-Detection-in-Videos.

[53] [online] sports-1m dataset. https://github.com/gtoderici/sports-1m-dataset/blob/wiki/ProjectHome.md.

[54] Karen Simonyan and Andrew Zisserman. "very deep convolutional networks for large-scale image recognition". *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[55] Sadegh Mohammadi, Hamed Kiani, Alessandro Perina, and Vittorio Murino. "violence detection in crowded scenes using substantial derivative". *2015 12th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pages 1–6, 2015.

[56] Tao Zhang, Wenjing Jia, Xiangjian He, and Jie Yang. "discriminative dictionary learning with motion weber local descriptor for violence detection". *IEEE Transactions on Circuits and Systems for Video Technology*, 27(3):696–709, 2017.

[57] Zhihong Dong, Jie Qin, and Yunhong Wang. Multi-stream deep networks for person to person violence detection in videos. *Springer Singapore*, pages 517–531, 2016.

[58] Alex Hanson, Koutilya PNVR, Sanjukta Krishnagopal, and Larry Davis. "bidirectional convolutional lstm for the detection of violence in videos". *Springer International Publishing*, pages 280–295, 2019.

[59] Swathikiran Sudhakaran and Oswald Lanz. "learning to detect violent videos using convolutional long short-term memory". *2017 14th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pages 1–6, 2017.

[60] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. "imagenet: A large-scale hierarchical image database". *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pages 248–255, 2009.

[61] Bolei Zhou, Aditya Khosla, Àgata Lapedriza, Aude Oliva, and Antonio Torralba. "learning deep features for discriminative localization". *CoRR*, abs/1512.04150, 2015.

[62] Ramprasaath R. Selvaraju, Michael Cogswell, Abhishek Das, Ramakrishna Vedantam, Devi Parikh, and Dhruv Batra. "grad-cam: Visual explanations from deep networks via gradient-based localization". *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 618–626, 2017.

[63] [online] keras-vis. https://github.com/raghakot/keras-vis.

[64] [online] keras-vis. https://keras.io/examples/vision/grad_cam/.