UNIVERSITÀ DEGLI STUDI DI FIRENZE

Scuola di Ingegneria
Dipartimento di Ingegneria dell'Informazione
Laurea Triennale in Ingegneria Informatica

Tesi di Laurea Triennale

Sviluppo di sistemi per compressione video semantica (Development of semantic video compression system)

Relatore:
Prof.Bertini Marco

Candidato:
Matteo Tintori

Anno Accademico 2019-2020

Alla mia famiglia che ha garantito economicamente in tutto il percorso di studi.

Ai compagni e agli amici che mi hanno accompagnato e in particolare a quelli che mi hanno sostenuto fino agli ultimi esami.

Al *Prof. Bertini Marco* per la professionalità e risolutezza nel fornire ogni tipo di materiale e aiuto.

La pazienza è amara, ma dolce è il suo frutto.
(Jean-Jacques Rousseau)

Indice

Elenco delle figure

Capitolo 1

Introduzione

1.1 Intento

L'obbiettivo di questo elaborato è la realizzazione di un Codec a partire da formato **H.265** destinato a video di calcio HD, che performi una compressione altrettanto efficiente ma variabile sulla base del contenuto interessante di ogni singolo frame. In particolare, si vuole codificare una partita di calcio utilizzando reti neurali convoluzionali per il riconoscimento e la conseguente segmentazione dei giocatori sul campo, e quando possibile, della palla. Le posizioni delle suddette entità sono le regioni di interesse che saranno lasciate senza perdita di qualità, mentre sulle altre zone verrà applicato un filtro che fornirà una compressione migliore "sacrificando" le alte frequenze e quindi una piccola parte di informazione non interessante. Viene poi effettuato un confronto tra il video originale e quello codificato come indice di efficienza, utilizzando metriche di qualità oggettiva. Infine, come indice di miglior codifica, viene effettuato un confronto tra i risultati di questo prodotto e un altro prodotto basato su Codec x264 e salienza, intesa come punto centrale dove l'osservatore è abituato a guardare spontaneamente, e se ne deduce il migliore in termini di dimensioni del prodotto compresso risultante.

1.2 Ambito del progetto

Il software si colloca nel campo della computer vision e si vuole affermare come Codec **H.265** che punta a comprimere più informazioni delle librerie già usate, valido particolarmente nei casi in cui ci sia una limitazione di spazio nel device in cui si scarica/guarda il video o nel caso si debba risparmiare per motivi economici sulla quantità di dati scaricati (ad esempio una soglia di (G)byte massima fissata dal gestore telefonico). Il Codec non si propone come versione superiore delle implementazioni già esistenti di **H.265** ma piuttosto come alternativa semantica.

Capitolo 2

Strumentazione

2.1 Piattaforma di sviluppo

La workstation usata per lo sviluppo del software è un server Linux con 2 schede video NVIDIA Titan X ed è collegata alla rete con IP pubblico all'indirizzo solaris.micc.unifi.it ed accedibile con utente mtintori. La modalità di lavoro perseguita è remota utilizzando remote desktop da PC Windows e PC Mac in seguito. L'IDE utilizzato per lo sviluppo è Pycharm con versione di Python 3.6.

2.2 Miniconda

Miniconda è un gestore di pacchetti ed environment-manager. E' una versione di installazione minimale di Anaconda, comprende l'interprete Python e permette di utilizzare un'IDE alternativo a Spider, usato con Anaconda, inoltre non vengono installati gli oltre 250 pacchetti che sono compresi nell'installazione di Anaconda lasciando la libertà allo sviluppatore su quali pacchetti installare.

Ambienti Miniconda

Utilizzeremo il termine Conda, istruzione fondamentale Miniconda, per riferirsi al sunnominato manager. Per la realizzazione del Codec si è resa necessaria la creazione di due ambienti Conda: è stata effettuata una simile scelta per possibilità di debug anche in assenza di rete su una workstation personale che non possiede una GPU.

- *tf_cpu* : ambiente più lento, ma utilizzabile anche da una workstation non avente GPU
- *tf_gpu* : ambiente più veloce, ma utilizzabile solo da una workstation con una GPU con ampia memoria adatta al training di reti neurali

2.3 Schema del sistema

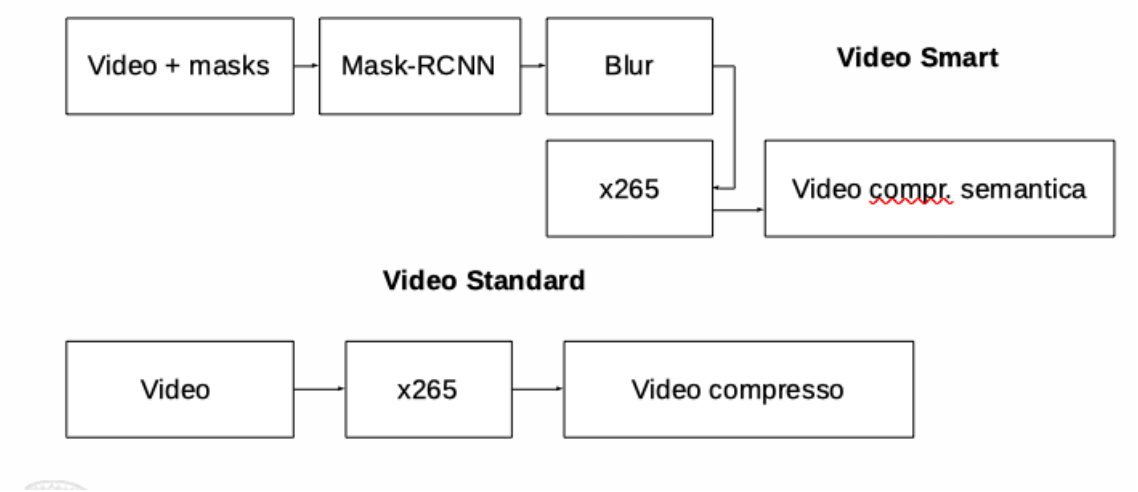

Figura 2.1

2.4 CODEC di partenza

La scelta del codec da utilizzare era contesa tra gli standard candidabili **H.264**, **H.265**, mentre gli altri due standard ottimali **VP9** e **AV1** non sono stati volontariamente presi in considerazione come assunto iniziale, nonostante le performance del secondo qualche volta superiore a **H.265** implementato dall'encoder **x265**.[2.2] (si noti che **VP9** invece è visibilmente inferiore agli altri,come si nota pur dalle info **libvpx**[11] che lo implementa). Sappiamo anche che **x264** implementa lo standard precedente; è quindi sicuramente più scarso. Lo standard dalla quale siamo partiti per una codifica efficiente è allora **H.265** usando **hevc_nvenc** [6.1], ed abbiamo optato per una codifica progressiva frame per frame. Il nostro encoder prevede una quantizzazione "doppia", poiché prima di scrivere ogni frame vengono selezionate le regioni non interessanti e ne vengono tagliate fuori le HF[1]; ciò verrà spiegato in seguito. Quello che invece fa il Codec di partenza che abbiamo scelto è una quantizzazione secondo lo standard, che viene effettuata manovrando il parametro QP cui valore rappresenta la sua intensità, ovvero quanta informazione (sempre HF) viene scartata.

[1] Alte frequenze

Figure-1 summarizes the bit savings (or increase) for VP9, x265, and the AV1-cfg7 cases over x264 across the 3 resolutions considered. It is clear that the average intra coding performance of AV1 is on par or better than x265, while giving a 5-10% average bits reduction over VP9.

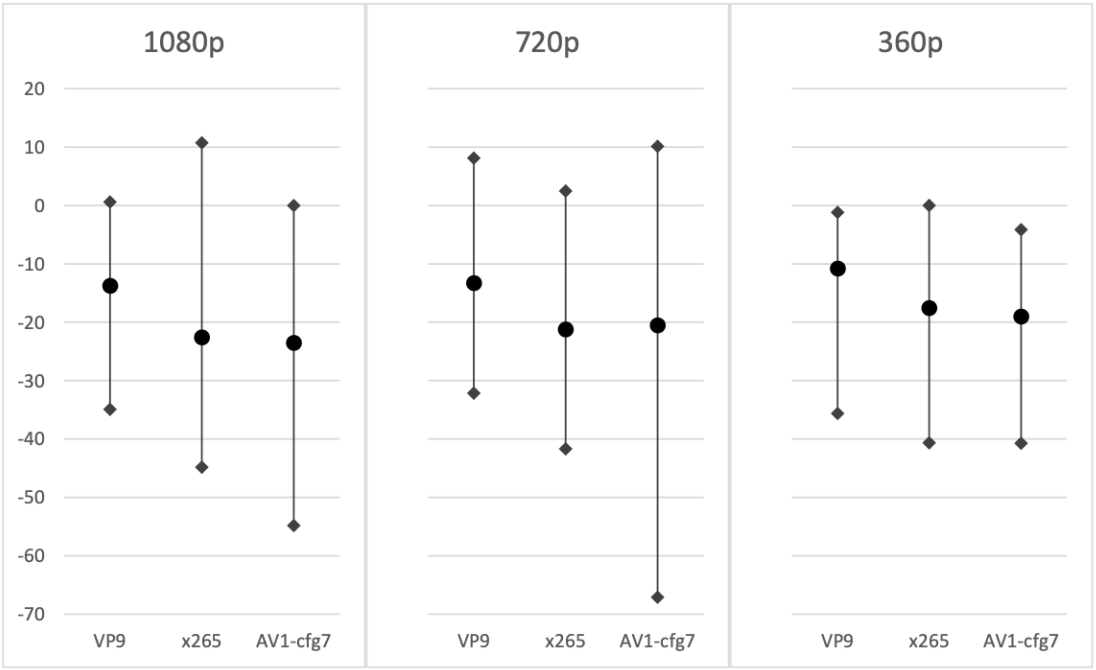

Figura 2.2: Risparmio in bits (segni: - risparmio, + incremento)

CRF

Il Costant Rate Factor[9] è un fattore che influenza la quantizzazione in modo più sofisticato del parametro a basso livello QP previsto dallo standard H.265. Più alto il CRF, più i campioni (pixel o blocchi di pixel) vengono quantizzati in maniera maggiore; questo valore resta fisso per l'intera codifica del video scelto, mentre il valore del parametro QP varia a seconda di quanto si debba quantizzare in ogni frame per mantenere una qualità costante. Nel nostro caso, utilizzando **nvenc** non sarà presente il parametro *crf* ma si deve utilizzare il parametro equivalente *cq* di questa libreria a cui sarà dato un valore che garantirà una compressione visivamente lossless, ovvero con perdita di qualità trascurabile all'occhio umano; essenzialmente questo valore per **h264_nvenc** è circa 19 per cui, preso atto che il valore di default di **x264** è 23 e corrisponde a circa 28 di **x265**, non sbaglieremo nel dire che 19 è sicuramente visually lossless anche per **hevc_nvenc**.

2.5 FFMPEG

Utilizzando la libreria **FFMPEG** è stato possibile scegliere diversi parametri oltre al Codec **hecv_nvenc**. Le modalità tipiche di codifica sono 3: [2]

1. bitrate scelto in 1 passo (sconsigliato)
2. bitrate scelto in 2 passi
3. CRF (fattore di qualità costante)

Dal momento che non è nostro interesse il raggiungimento di una dimensione file particolare, la scelta è caduta sulla terza opzione, che è stata spiegata nella sottosezione precedente. Gli altri parametri che abbiamo utilizzato per la codifica sono i seguenti:

- **-cq**: fattore di qualità che mantiene la suddetta costante in ogni frame manovrando il parametro che definisce la quantizzazione QP
- **-qmin** : fattore minimo di qualità che **cq** può assumere
- **-qmax** : fattore massimo di qualità che **cq** può assumere
- **-b:v** : posto a 0 per causa di un bug nella libreria, che se non venisse settato inibisce il funzionamento di cq
- **-rc** : rate control, sovrascrive la velocità di compressione ("preset",più basso il preset più alta la qualità) e deve essere settato insieme a ***-b:v*** per una qualità costante
- **-r** : frame rate, ovvero quanti frame vengono campionati nell'unità di tempo

2.6 Mask-RCNN

Mask-RCNN è un'implementazione di una rete neurale Faster-RCNN da parte del framework tensorflow di Google. Si differenzia dalle Faster-RCNN per l'introduzione del RoI Align, più preciso rispetto al RoI Pooling precedente. Facciamo un piccolo riepilogo per maggiore chiarezza.

Riepilogo - Dalle CNN alle Faster-RCNN

Le reti neurali convoluzionali **(CNN)** sono state introdotte nel campo dell' IA per vari scopi e più precisamente nella computer vision sono utili per il riconoscimento di qualsiasi entità o pattern all'interno di un immagine. Le potenzialità e i limiti sono elencati di seguito:

[2] vedi rate control modes: https://trac.ffmpeg.org/wiki/Encode/H.265

1. Usano una funzione softmax per classificazione multi-classe e sigmoid per classificazione binaria
2. Usata per image classification e object detection
3. Rileva solo un oggetto alla volta senza sovrapposizioni

Successivamente si sono affermate le CNN basate su regioni **(RCNN)**, che forniscono il miglioramento di riconoscere più oggetti diversi nella stessa immagine e identificarli all'interno di un rettangolo. Questo è possibile con il seguente procedimento:

1. Generazione di regioni propositive con R-CNN per mezzo dell'algoritmo *Selective Search* unito con *Exhaustive Search*
2. Fusione di regioni propositive simili e *Feature extraction* con CNN per ogni regione propositiva
3. Algoritmo *Support vector machine* per la feature estratta per la verifica dell'effettiva presenza dell'oggetto
4. Algoritmo *Non-Max suppression* per lo scarto di regioni con basso punteggio in *Intersection over Union*.

Per ottenere un altro significativo miglioramento siamo arrivati alle reti **Fast-RCNN**, che offrono i seguenti vantaggi:

1. Unica rete neurale (deep ConvNet) che sostituisce quelle migliaia di R-CNN sulle singole regioni
2. Unico modello per la feature extraction, la classificazione e le bounding boxes
3. Introduzione delle Region of Interest (RoI) e del RoI Pooling

Infine, si è voluto perfezionare l'arte del riconoscimento e identificazione delle forme di ogni entità con le reti **Faster-RCNN**, che permettono di:

1. 3-D object detection
2. Part-based detection
3. Instance segmentation
4. Image captioning

Capitolo 3

Analisi

Il codec descritto, non è utiizzabile come versione stand-alone integrata in un software applicativo, ma potrà essere dispiegata come estensione chrome o browser alternativo in modo che sia fruibile sia da PC che da Smartphone e utilizzabile ogni qual volta viene scaricato un video con qualità almeno 720p.

3.1 Analisi Funzionale

Su questo tipo di analisi è necessario essere brevi, perché l'unica funzionalità offerta dall'encoder è appunto su richiesta dell'utente scaricare o visionare il video 720p o 1080p nel nostro formato H.265. Le funzionalità descritte possono essere fruite nei seguenti modi:

- Installando come estensione browser il nostro progetto encoder e selezionando l'opzione "SI" al popup in apertura dopo il click del download.
- Il nostro Codec verrà fornito nella lista delle risoluzioni di un video visionabile elencandolo come 720p e 1080p "ottimizzato", e verrà caricato alla scelta con la codifica in questione.

Di seguito abbiamo previsto un'immagine utilizzata esclusivamente per documentare la metodologia del lavoro svolto di riconoscimento e segmentazione dei giocatori su un particolare frame:

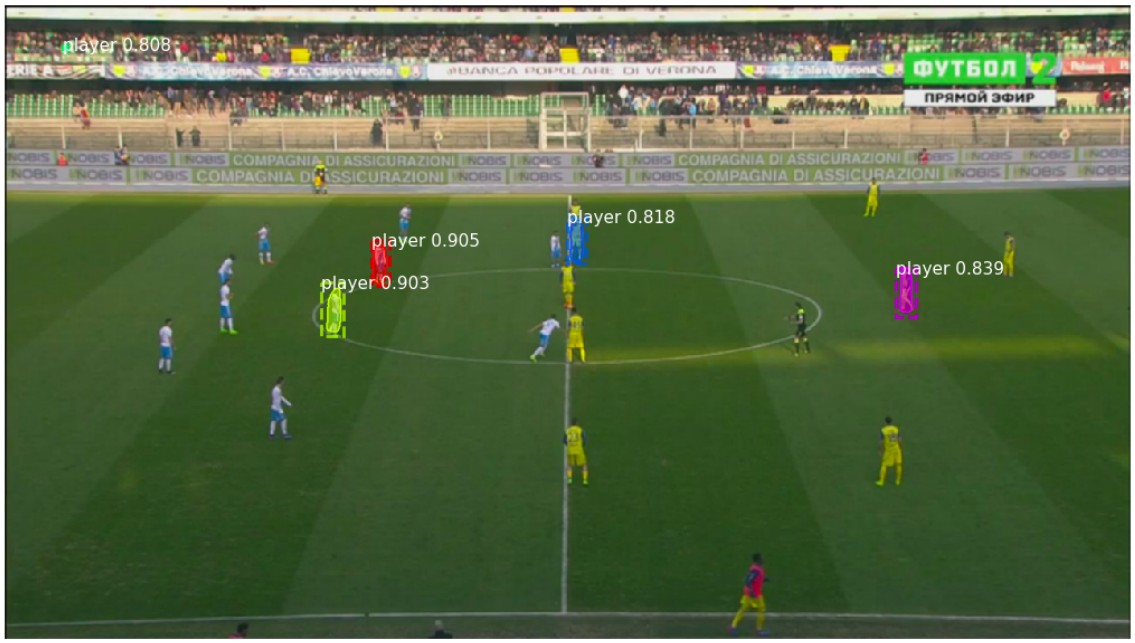

Figura 3.1: dimostrazione di segmentazione giocatori

3.2 Analisi Tecnica

I file scritti personalmente che fanno parte di questo progetto sono 4 e ha anche una particolare importanza il file model.py non implementato personalmente, in quanto contiene la struttura di tutta la rete neurale da cui possiamo trarre i metodi per la creazione dell'oggetto che rappresenta la rete e per effettuare il training di essa.

- fine_tuning.py La classe è stata introdotta con la responsabilità di effettuare tutte le operazioni di training e di fine tuning della nostra rete e instanziare un oggetto soccer_dataset su cui far partire due training: il primo sui top layers chiamati *heads* e il secondo sui livelli *4+* ovvero dallo strato 4 in su di una rete ResNet101. Questo è stato fatto attenendosi ad alcune fonti [2] e all'esempio *balloon.py* poiché potevamo scegliere di fare un fine tuning sbloccando per l'allenamento anche solo i livelli più vicini agli heads: *5+* oppure più lontani dagli heads: *3+* o perfino allenare la rete totalmente.

- soccer_dataset.py Carica un dataset di immagini di annotazioni in formato pascal-VOC

- main.py Si occupa di gestire l'input della barra di comando che deve essere un video HD che mostra una partita di calcio, ne effettua l'instance segmentation, la quantizzazione e codifica per produrre un video Mp4 codificato H.265 con LPF.

- clever_config.py Specifica la configurazione delle impostazioni di fine tuning desiderate che devono essere passate alla classe omonima per almeno i parametri fondamentali quali STEP_PER_EPOCH,GPU_COUNT,NUM_CLASSES oppure BATCH_SIZE nel caso di parametro IMAGES_PER_GPU maggiore di 1.

Alla pagina seguente si riassume la struttura del progetto utilizzando un diagramma UML.

Main

Come si può osservare dalla struttura del file main.py, questo non contiene una classe bensì un'istruzione che si accerta che il file main sia eseguito da riga di comando come primo file, e non sia quindi importato da altri files. L'espressione che rende un booleano per confermare questa condizione è la seguente: **___name___ == "___main___"** Come visibile dal diagramma UML collocato in seguito, il file main offre i seguenti metodi:

- Un metodo *draw_images_with_boxes* che si propone come metodo di "debug" in quanto serve solamente a mostrare il lavoro di riconoscimento e segmentazione svolto quando l'input è un'immagine, cosa che esula dai nostri obbiettivi di codifica **video**.

- Un metodo *is_video_file* che prende in ingresso un estensione e la confronta con tutte le esistenti estensioni video per capire se l'input in ingresso è effettivamente un video o meno.

- Un metodo *quantize_frame* che è il metodo fulcro della nostra codifica per Region of Interest (RoI), infatti utilizza una copia del frame e vi applica un filtro passa-basso per rimuoverne le HF, dopodiché incolla le maschere dei giocatori del frame originale che le contiene ancora in HD sul frame copia, facendo risultare l'effetto LPF solo sulle regioni di non interesse.

- Il metodo *main* che è il metodo contenente tutto il codice utile per arrivare ai seguenti risultati:

    1. Ottenere i parametri inseriti a barra di comando
    2. Controllare se c'è un parametro(booleano) fine tuning attivo e in caso affermativo delegare la responsabilità di effettuare il training alla classe omonima
    3. Controllare se c'è un parametro input e in caso affermativo controllare se l'input è un video: in caso negativo se è un immagine la visualizza a video come instance segmentation
    4. In caso che invece il parametro di input sia un video instanzia un oggetto **FFMpegWriter** con l'encoder **hevc** e con un coefficente di qualità costante adatto, dopodiché ne cicla i frame uno per uno

5. Una volta dentro il ciclo utiizza la funzione *quantize_frame* per applicare un LPF al frame corrente, dopodiché lo scrive nel nuovo video con la funzione *write* di **FFMpegWriter**
6. Alla fine del ciclo, tutti i frame sono stati scritti in modo codificato "2 volte", ovvero una grazie all'oggetto **FFMpegWriter** che ha tolto una quantità di informazioni trascurabile e l'altra grazie al nostro LPF. Infine quindi le risorse vengono rilasciate e il nuovo video viene salvato con estensione contenitore generica .mp4

Classe model

La descrizione di ciò che le classi operano è riassunta nel preambolo di questa sezione, essendo esse strettamente collegate ai rispettivi files in python. Descriviamo ora in particolare la classe non implementata da noi denominata *model* che chiameremo con il suo alias *modellib* e il suo utilizzo, dopodiché ci soffermiamo su attributi e metodi delle classi corrispondenti ai files già descritti in precedenza. Come visibile dal diagramma UML collocato in seguito, la classe modellib all'interno di model.py deve offrire:

- Un attributo accessibile dal model che rappresenta i pesi*(weights)* allenati della rete, accessibile come keras_model.history.history
- Un costruttore statico utilizzabile con l'istruzione modellib.MaskRCNN che restituisce un'istanza del model.
- Un metodo che viene usato per caricare i *weights* escludendo i 4 layers che comprendono la lista di tutte le labels su cui è stata allenata originariamente la rete MaskRCNN, in modo da personalizzare il training su esclusivamente i giocatori e la palla, denominato *load_weights.*
- Un metodo denominato *train* che consente effettivamente di effettuare il training dei *weights* specificando il training set e il test set, i layers bersaglio che devono essere sbloccati per il fine tuning(heads o 3/4/5+),il numero di epochs e il learning rate che vengono ricavati dal file clever_config.py per best practice e le trasformazioni casuali sui frames (*augmentation*) per un allenamento dinamico.
- Un metodo denominato *save_weigths* che viene utilizzato per salvare sul file system i *weights* addestrati dopo il fine tuning e viene richiamato con keras_model.save_weigths.

Classe SoccerDataset

La classe SoccerDataset ridefinisce i metodi della sua superclasse Dataset,eccetto un metodo che merita una menzione particolare chiamato *prepare*, che presa per scontata l'equivalenza tra labels e classi,offre le seguenti operazioni:

- Il metodo suddetto salva tutte le informazioni delle classi come il numero e il nome di esse, compresi gli IDs in forma autoincrementante dalla variabile *self.class_info*
- Mappa IDs interni(class e images) partendo dalle info come *self.class_info* e *self.class_ids*
- Mappa i sorgenti(sources) verso i class_ids che essi supportano.

Il metodo che è sottoposto a ovveride nella classe figlia chiamato *load_dataset*, come si può osservare dal nome autodocumentante, si limita a caricare le classi del dataset personalizzato. Gli altri metodi, che non sono sottoposti a override ma sono totalmente nuovi nella classe figlia SoccerDataset, sono elencati seguentemente con i loro effetti:

- extract_polygons : è il metodo fondamentale per la lettura delle annotazioni JSON di un immagine scelta che contengono le informazioni sui poligoni che descrivono i giocatori e la decodifica dei dati su di essi per poi essere salvati in un array di polygons che sarà restituito al chiamante insieme alla larghezza e all'altezza dell'immagine.
- load_mask : il metodo corrente, data un'ID intera di un'immagine, restituisce la sua altezza,larghezza e l'array di poligoni che sono stati estratti con il metodo *extract_polygons*
- image_reference : ritorna la path del'immagine

Classe FineTuning

La classe sunnomminata si dedica innanzitutto a creare una configurazione personalizzata CleverConfig che sarà utilizzata nella stessa classe come parametro per costruttore del model, nell'unico metodo esistente che è:

- start: questo metodo si dedica alla creazione dei train_set e test_set per l'allenamento della rete, alla creazione dell'oggetto model per il caricamento dei pesi e per il loro allenamento in 2 fasi: prima quello degli *heads* e successivamente quello dei livelli *4+*

Classe CleverConfig

Effettua l'ovverride della classe Config standard che è anonima ma contiene tutti i parametri per effettuare un fine tuning o training da una rete blackbox, ovvero che non ha ricevuto nessun addestramento.
I parametri con cui la classe Config se non ridefinita performerebbe il training della rete sono i seguenti(trascritti direttamente dalla classe config.py):

```
class Config(object):


    NAME = None  # Override in sub-classes

    GPU_COUNT = 1


    IMAGES_PER_GPU = 2

    STEPS_PER_EPOCH = 1000


    VALIDATION_STEPS = 50

    BACKBONE = "resnet101"

    COMPUTE_BACKBONE_SHAPE = None

    # The strides of each layer of the FPN Pyramid. These values
    # are based on a Resnet101 backbone.
    BACKBONE_STRIDES = [4, 8, 16, 32, 64]

    # Size of the fully-connected layers in the classification
        graph
    FPN_CLASSIF_FC_LAYERS_SIZE = 1024

    # Size of the top-down layers used to build the feature pyramid
    TOP_DOWN_PYRAMID_SIZE = 256

    # Number of classification classes (including background)
    NUM_CLASSES = 1  # Override in sub-classes

    # Length of square anchor side in pixels
    RPN_ANCHOR_SCALES = (32, 64, 128, 256, 512)

    # Ratios of anchors at each cell (width/height)
    # A value of 1 represents a square anchor, and 0.5 is a wide
        anchor
    RPN_ANCHOR_RATIOS = [0.5, 1, 2]

    # If 1 then anchors are created for each cell in the backbone
        feature map.
```

```
41      # If 2, then anchors are created for every other cell, and so
            on.
42      RPN_ANCHOR_STRIDE = 1
43
44      # Non-max suppression threshold to filter RPN proposals.
45      # You can increase this during training to generate more
            propsals.
46      RPN_NMS_THRESHOLD = 0.7
47
48      # How many anchors per image to use for RPN training
49      RPN_TRAIN_ANCHORS_PER_IMAGE = 256
50
51      # ROIs kept after tf.nn.top_k and before non-maximum
            suppression
52      PRE_NMS_LIMIT = 6000
53
54      # ROIs kept after non-maximum suppression (training and
            inference)
55      POST_NMS_ROIS_TRAINING = 2000
56      POST_NMS_ROIS_INFERENCE = 1000
57
58      # If enabled, resizes instance masks to a smaller size to
            reduce
59      # memory load. Recommended when using high-resolution images.
60      USE_MINI_MASK = True
61      MINI_MASK_SHAPE = (56, 56)  # (height, width) of the mini-mask
62
63      IMAGE_RESIZE_MODE = "square"
64      IMAGE_MIN_DIM = 800
65      IMAGE_MAX_DIM = 1024
66
67      IMAGE_MIN_SCALE = 0
68          IMAGE_CHANNEL_COUNT = 3
69
70      # Image mean (RGB)
71      MEAN_PIXEL = np.array([123.7, 116.8, 103.9])
72
73          TRAIN_ROIS_PER_IMAGE = 200
74
75      # Percent of positive ROIs used to train classifier/mask heads
76      ROI_POSITIVE_RATIO = 0.33
77
78      # Pooled ROIs
79      POOL_SIZE = 7
80      MASK_POOL_SIZE = 14
```

```
81
82      # Shape of output mask
83      # To change this you also need to change the neural network
            mask branch
84      MASK_SHAPE = [28, 28]
85
86      # Maximum number of ground truth instances to use in one image
87      MAX_GT_INSTANCES = 100
88
89      # Bounding box refinement standard deviation for RPN and final
            detections.
90      RPN_BBOX_STD_DEV = np.array([0.1, 0.1, 0.2, 0.2])
91      BBOX_STD_DEV = np.array([0.1, 0.1, 0.2, 0.2])
92
93      # Max number of final detections
94      DETECTION_MAX_INSTANCES = 100
95
96      DETECTION_MIN_CONFIDENCE = 0.7
97
98      # Non-maximum suppression threshold for detection
99      DETECTION_NMS_THRESHOLD = 0.3
100
101
102     LEARNING_RATE = 0.001
103     LEARNING_MOMENTUM = 0.9
104
105     # Weight decay regularization
106     WEIGHT_DECAY = 0.0001
107
108     LOSS_WEIGHTS = {
109         "rpn_class_loss": 1.,
110         "rpn_bbox_loss": 1.,
111         "mrcnn_class_loss": 1.,
112         "mrcnn_bbox_loss": 1.,
113         "mrcnn_mask_loss": 1.
114     }
115
116     USE_RPN_ROIS = True
117
118     TRAIN_BN = False  # Defaulting to False since batch size is
            often small
119
120     # Gradient norm clipping
121     GRADIENT_CLIP_NORM = 5.0
```

Per effettuare quindi un fine tuning addatto all'evenienza abbiamo ridefinito i seguenti parametri, con il nostro ovveride della classe Config:

```
class CleverConfig(Config):
    # give the configuration a recognizable name
    NAME = "clever_coding_config"

    # set the number of GPUs to use along with the number of images
    # per GPU
    GPU_COUNT = 1
    IMAGES_PER_GPU = 1
    # Skip detections with < 90% confidence
    DETECTION_MIN_CONFIDENCE = 0.8
    # batch_size=2. Samples=50. steps=50/2
    STEPS_PER_EPOCH = 120
    VALIDATION_STEPS = 25
    BATCH_SIZE = 1 #GPU_COUNT*IMAGES_PER_GPU
    # number of classes (we would normally add +1 for the
       background
    # but the background class is *already* included in the class
    # names)
    NUM_CLASSES = 1 + 2
```

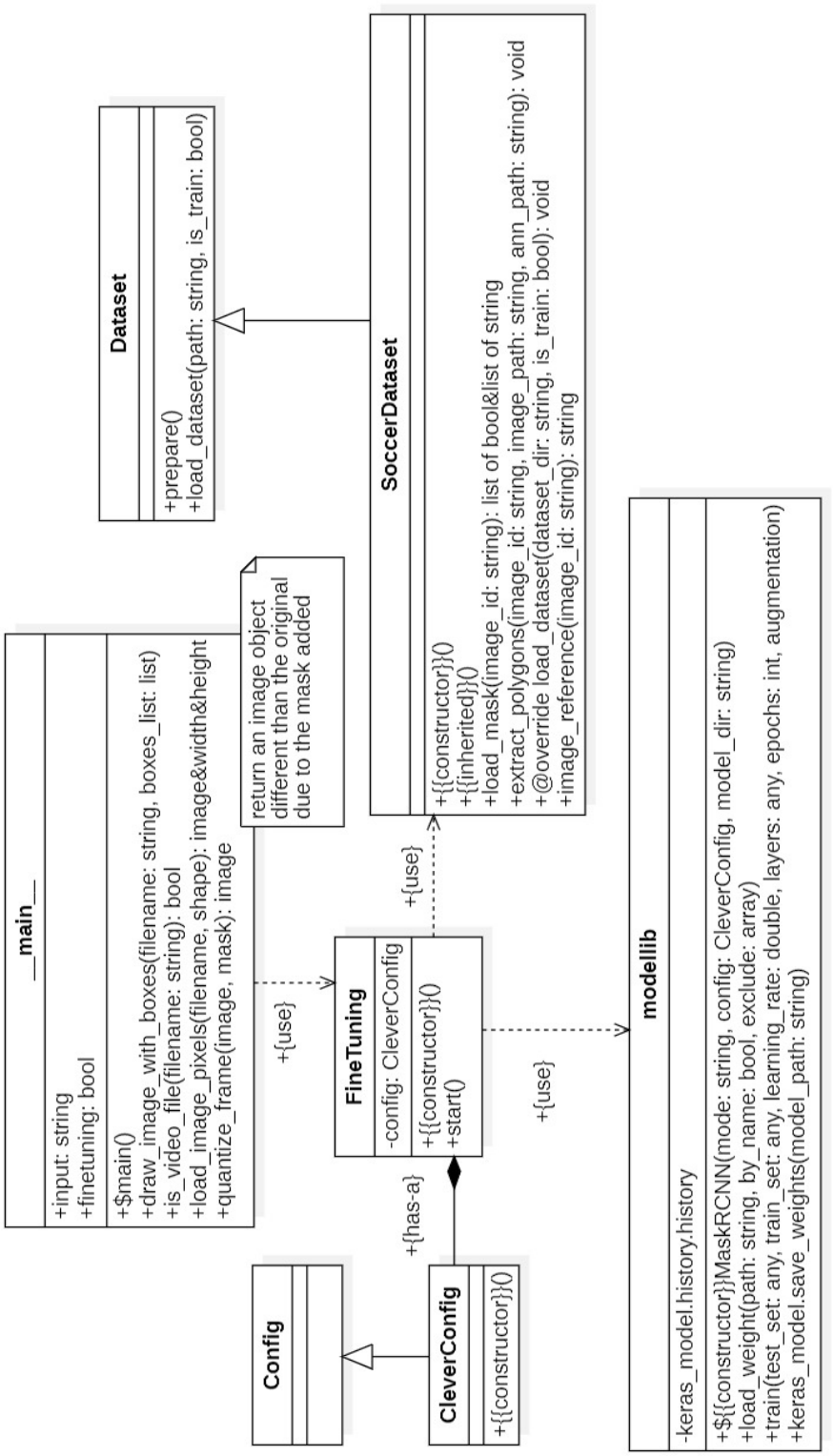

Figura 3.2: Diagramma UML del progetto

Capitolo 4

Sistema di segmentazione oggetti

In questo capitolo, per iniziare, si vuole distinguere la differenza tra un training di una rete blackbox e un fine tuning di una rete preaddestrata.
Essendo **Mask-RCNN** una rete preaddestrata per circa 60 classi, si parlerà innanzitutto di un fine tuning e non di un allenamento di una rete partendo da zero esperienza(blackbox).
La cosa che distingue il nostro agire in questo caso è che faremo un fine tuning, non sulle circa 60 classi sulla quale è stata preaddestrata la rete, ma solamente su 2 classi che rappresenteranno i giocatori e la palla all'interno del campo di calcio.

4.1 Strutturazione Fine-Tuning

Come premessa, bisogna dichiarare il fatto che per un buon fine tuning serve scegliere un software di labeling che fornisce le annotazioni in un certo formato, solitamente XML. Nel nostro caso, siccome sappiamo che PASCALVOC XML viene usato quando c'è da fare object detection mentre noi cerchiamo di fare instance segmentation, abbiamo usato un software chiamato **labelme** che produce annotazioni con estensione JSON in modo da prelevare facilmente i dati dei poligoni racchiusi nei files. Lo strumento **labelme** viene utilizzato per tracciare poligoni intorno ai giocatori nei frames, cioè le immagini singole di un video che è stato frammentato con uno strumento come ad esempio **Avidemux**.
Nel software **labelme** quando viene scelto un tracciamento della curva *poligono*, questi poligoni devono essere tracciati come linea spezzata chiusa con un numero di punti adatto a circondare con la minima tolleranza possibile la forma di un giocatore, e nel caso della scelta sul software **labelme** della curva *cerchio*, esso andrà a circondare la palla. Chiaramente, un giocatore potrà essere all'interno del campo in diverse posizioni, ad esempio in piedi fermo, in piedi in corsa oppure a terra come dopo una parata o una scivolata; tutti questi casi devono essere circondati da punti che formano un poligono per far sì che la rete **MaskRCNN** abbia un allenamento soddisfacente.
Una volta salvato il frame modificato con i poligoni, il software **labelme** genererà un file JSON con all'interno i dati del nome dell'immagine, la sua altezza e larghezza, il numero e i punti dei poligoni in modo da essere letti successivamente da codice. L'operazione di contornamento dei giocatori e della palla con i poligoni di punti, si chiama **labeling**, e si può effettuare un labeling anche massivo specificando all'apertura del software **labelme** una directory dove sono presenti tutti i frames del video obbiettivo e in contempo una directory vuota dove produrre in massa i file JSON delle loro annotazioni; ogni file di annotazione viene prodotto non appena il labeling di un immagine viene ultimato e quindi l'immagine viene salvata.

Capitolo 5

Discussione

5.1 Commenti al Codice

1. Partendo dal principio, analizziamo la riga *get_ipython* e *ipy.run_line_magic("matplotlib,"inline")*: queste righe partono dalla necessità di fare solo un piccolo debug non necessario al reale algoritmo, ma necessario quando l'input è ancora un'immagine singola per mostrare all'utente come visto in 3.1 le operazioni di instance segmentation su un'immagine. Senza queste righe non ci sarebbe il grafico a provare questo tipo di lavoro e quindi non si sarebbe sviluppato con certezza la parte dell'algoritmo che serve realmente e cioè quella dove l'input è un video.

2. L'istruzione *parse.parseargs()* serve chiaramente ad ottenere i parametri di ingresso che sono stati specificati dall'utente per poi controllare se c'è in input un video/immagine oppure c'è una richiesta di fine tuning.

3. Dopo il controllo che sia effettivamente una richiesta di fine tuning, ci sono le istruzioni già descritte: *f_tune = FineTuning()* e *f_tune.start()* per iniziare il training della rete.

4. Dopo il controllo che invece sia un video/immagine, viene creata una variabile *config=CleverConfig()* che viene utilizzata per creare il model e dopo vengono caricati i pesi. Infine, vengono creati un array con la lista delle classi e i colori: *classes = ["BG","player","ball"]* e *COLORS = ["green","red","blue"]*

5. Dopo ciò, viene salvata una path che indirizza verso la cartella del dataset DATASET_PATH e si controlla con *os.path.exists(DATASET_PATH+args.input)* se questo file specificato in args.input esiste in quella path, in caso contrario il programma termina.

6. In caso positivo, si legge il file video con l'istruzione *vread* e se ne leggono i metadati interessanti quali **fps,height,width** per poi creare un nome che è quello del video originale concatenato al timestamp come nome del video codificato.

7. Si crea quindi un oggetto **FFMpegWriter** e si munisce dei parametri ottenuti con i metadati uniti al codec scelto **hecv_nvenc**, e al parametro di qualità costante con la qualità scelta. Da notare che come enunciato in [6] il flag verbosity è attivo.

8. In seguito, si entra nel ciclo madre di tutto l'algoritmo dove si scorrono i frames contenuti nell'oggetto *reader* e se ne controlla se risultano **None**, in caso contrario si effettua prima l'object detection con il metodo *detect*, e infine la quantizzazione.

9. Infine, si rilasciano le risorse dell'oggetto **FFMpegWriter** con l'istruzione *close()*.

10. Per concludere, rileggiamo il file originale e lo confrontiamo con varie metriche di qualità paragonandolo a quello codificato con **FFMpegWriter** e quantizzato per Region of Interest. Le metriche di qualità usate sono: **SSIM,BRISQUE,LPIPS**.

Capitolo 6

Debug

6.1 hevc_nvenc

Il codec sovracitato, adottato e spiegato a inizio documentazione, non è stato adottato casualmente per disponibilità di questo encoder nella lista, ma per essere adottato è stato seguito un procedimento più ostico che merita una descrizione accurata. Innanzitutto la versione di **FFMPEG** adottata non è stata compilata con i flag adatti per una scelta di un encoder H.265, per cui abbiamo dovuto reinstallare la stessa in maniera differente: ciò verrà spiegato in seguito. Come prima cosa, nell'uso di questa libreria grazie all'istruzione **FFmpegWriter** abbiamo potuto constatare che vi era un errore bloccante che non veniva descritto con verbosità tale da capirne l'origine. A video l'errore si limitava a essere descritto come *broken pipe* non appena veniva iniziata la scrittura del primo frame del video di cui si iniziava la codifica, anche se la libreria usata era "libx264". Per causa di ciò, dopo accurate ricerche, si è dovuto impostare un paramentro aggiuntivo nell'istruzione di creazione dell'oggetto **FFmpegWriter** scritto come *verbosity=1.* A questo punto, grazie ad un log dell'errore con verbosità maggiore come segue: *Unrecognized option 'height'* e *Unknown decoder 'libx265'* si è potuto constatare che l'errore dipendeva da questi parametri e abbiamo così preso alcuni provvedimenti. Il parametro height è stato tolto, mentre per l'encoder scelto abbiamo testato se effettivamente questo non fosse disponibile con l'istruzione: *ffmpeg -codecs | grep 26* ed è risultato che il codec **hevc** esiste ma è utilizzabile solo per la decodifica.
Cercando nel canale loopbio abbiamo potuto confermare che si può attivare la codifica H.265 utilizzando un pacchetto del canale stesso, ma ciò è possibile disinstallando **FFMPEG** e **x264,x265** con l'istruzione: *conda uninstall x264 x265 ffmpeg —force* e reinstallandoli dal canale loopbio con l'istruzione: *conda install ffmpeg x264 x265 -c loopbio –force-reinstall*
Un problema aggiuntivo si è verificato in quanto la versione di x264 non è alla versione richiesta di **FFMPEG**, e l'errore se utilizzata l'istruzione *ffmpeg -codecs* risulta: *error*

while loading shared libraries: libx264.so.138: cannot open shared object file: No such file or directory.
Di conseguenza, abbiamo dovuto cercare il package che fornisce la versione di x264 che **FFMPEG** desidera, con l'istruzione: *ls -la /miniconda3/envs/tf_cpu/lib/libx2** e ciò è sfociato con l'istruzione: *conda install -c lightsource2 x264*
Adesso ffmpeg -codecs | grep 26 riporta:
DEV.L. hevc H.265 / HEVC (High Efficiency Video Coding) (encoders: nvenc_hevc hevc_nvenc)
La Lettera "E" nelle iniziali "DEVL" significa che è possibile a questo punto effettuare una codifica, oltre alla decodifica che era già possibile inizialmente.
A questo punto, è bastato sostituire il parametro **-vcodec** da **libx265** a **hevc_nvenc** per far partire la codifica con l'encoder ottimale scelto.

Capitolo 7

Alternativa

In questo capitolo il nostro scopo è differente da quello di proporre un codec efficiente, ma di utilizzare invece un codec alternativo non orientato alle partite di calcio ma general purpose che rende alta la definizioni dell'insieme dei punti dove il visualizzatore del video è abituato a guardare, ovvero nel suo punto centrale. Il codice per utilizzare questo codec, meno efficiente del nostro poiché usa un'implementazione di x264, si può utilizzare da barra di comando /path/to/executable/x264 source.mp4 e con una lista di parametri che viene dettagliata in seguito:

- –output <out>.mp4
- –bitrate <bitrate_totale>
- –saliency <nome_file>
- –saliency-bitrate <bitrate_percentuale_interessante>

Bibliografia

[1] Fangdong Chen et al. «Block-Composed Background Reference for High Efficiency Video Coding». In: *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY* (2017).

[2] François Chollet. *Deep Learning with Python: Chapter 5.3.2: Fine-tuning*. Manning Publications, 2017. URL: https://livebook.manning.com/book/deep-learning-with-python/chapter-5/216l.

[3] L. Hanzo, P. J. Cherriman e J. Streit. *Video Compression and Communications. From Basics to H.261, H.263, H.264, MPEG4 for DVB and HSDPA-Style Adaptive Turbo-Transceivers*. John Wiley&Sons, 2007.

[4] Xuguan Lan et al. «Arbitrary ROI-based wavelet video coding ». In: *Neurocomputing* (2011).

[5] Xiaoqi Liu et al. «Fine-Tuning Pre-trained Convolutional Neural Network for Gastric Precancerous Disease Classification on Magnification Narrow-band Imaging Images». In: *Neurocomputing* (2019). URL: https://www.journals.elsevier.com/neurocomputing.

[6] Jens-Rainer Ohm et al. «Comparison of the Coding Efficiency of Video Coding Standards—Including High Efficiency Video Coding (HEVC) ». In: *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY* (2012).

[7] Diego Rueda Plata, Raul Ramos-Pollan e Fabio A. Gonzalez. «Effective training of convolutional neural networks with small, specialized dataset». In: *Journal of Intelligent & Fuzzy Systems 32* (2017).

[8] Iain E.G. Richardson. *H.264 and MPEG-4*. WILEY, 2003.

[9] Werner Robitza. «CRF Guide (Constant Rate Factor in x264, x265 and libvpx)». In: (2017). URL: https://slhck.info/video/2017/02/24/crf-guide.html.

[10] Hamid Reza Tohidypour et al. «Online-Learning-Based Mode Prediction Method for Quality Scalable Extension of the High Efficiency Video Coding (HEVC) Standard». In: *IEEE TRANSACTIONS ON CIRCUITS AND SYSTEMS FOR VIDEO TECHNOLOGY* (2017).

[11] Wikipedia. «libvpx». In: (). URL: https://en.wikipedia.org/wiki/Libvpx.

[12] Xiaojin Zhu e Andrew B. Goldberg. *Introduction to Semi-Supervised Learning*. Morgan&Claipool, 2009.