Report Progetto Sistemi Digitali M

ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INFORMATICA (DISI)

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

Sviluppato da:

Loris Giannatempo 0001031947

Davide Guidetti 0001028494

Supervisionato da:

Prof. Matteo Poggi

INDICE

ABSTRACT

Il seguente progetto consiste nello sviluppo di una applicazione Android in grado di rilevare la presenza o meno di mascherine protettive contro la diffusione del COVID 19.

L’app farà quindi utilizzo di una rete neurale (da noi opportunamente addestrata) in grado di effettuare Object Detection di due classi di individui: con mascherina e senza mascherina.

I dati rilevati verranno mostrati sullo schermo del dispositivo. In particolare, verranno mostrate le posizioni dei volti correntemente individuati, e il numero complessivo di volti attualmente con mascherina e senza mascherina. Sulla base di questi due valori verrà anche mostrata una stima percentuale del numero di persone con mascherina.

Inoltre, la parte relativa all'estensione per il riconoscimento dell'attività progettuale, ha come obiettivo l'utilizzo dei dati estratti per effettuare Object Tracking degli individui.
Lo scopo è quello di poter utilizzare l'applicazione come un "conta persone" che mostri il conteggio delle persone entrate con e senza mascherina.

TRAINING DEL MODELLO

DATASET E PREPROCESSING

Allo scopo di addestrare una rete neurale, in grado di fare Object Detection degli individui, si è scelto di utilizzare la TensorFlow 2 Object Detection API [1].

Per l'addestramento della rete è quindi stato scaricato un dataset [2], con il quale si sono però ottenuti risultati finali non soddisfacenti. Tale dataset si compone di circa 800 immagini con box appartenenti a 3 diverse classi: soggetti con mascherina, soggetti senza mascherina e soggetti con mascherina indossata in maniera incorretta. Le performance della rete si sono rivelate essere particolarmente scarse nella rilevazione di quest'ultima classe.

Si è poi optato per l'utilizzo di un altro dataset [3] contenente un numero molto più elevato di immagini (circa 4000), ma con label appartenenti a ben 20 classi diverse.

Per avere una migliore comprensione delle label e di come queste effettivamente vengano impiegate all'interno del dataset, si è fatto uso di un apposito script (*filter_images.py*) in grado di mostrare le stesse immagini con le relative box appartenenti alle classi specificate. In questo modo si è potuto constatare che esistono delle label "generali" e altre più specifiche che nel nostro caso possono essere ignorate. Lo script si occupa anche di copiare nella cartella di training le sole immagini per le quali è presente nel file *train.csv* almeno una delle label alle quali siamo interessati.

Le 4 label che sono state usate per questo progetto sono:

- *face_with_mask*: persone con mascherina
- *face_no_mask*: persone senza mascherina
- *face_other_covering*: persone che hanno il volto coperto da qualcosa, il più delle volte sprovvisti di mascherina (o comunque non è possibile determinarlo con certezza)
- *face_with_mask_incorrect*: persone che hanno la mascherina ma la indossano incorrettamente

Si è scelto di rimappare queste 4 categorie in 2 sole classi (con o senza mascherina). In particolare, si considerano volti senza mascherina quelli indicati dalle classi *face_no_mask* e *face_other_covering*. Viceversa, sono considerati con mascherina gli individui etichettati con *face_with_mask* e *face_with_mask_incorrect*.

Il passo successivo è stato quello di far uso di un ulteriore script Python in grado di generare i file TFRecord, uno per il training e uno per il test (*generate_tfrecord.py*).

Questi file contengono tutti i dati di addestramento e di test di cui Tensorflow avrà bisogno per addestrare il modello, quindi sia le label che le immagini sono codificate e inserite all'interno del file TFRecord. La guida mette a disposizione un esempio completo su come creare questi file a partire da immagini annotate con file xml (uno per ogni immagine). Tuttavia, nel nostro caso si è reso necessario modificare opportunamente lo script non solo per far sì che venga utilizzato un unico file .csv al posto di file .xml, ma anche per tradurre le classi nella maniera precedentemente esplicitata.

Inoltre, nella creazione dei file TFRecord, le immagini vengono inserite già scalate nella dimensione richiesta dal modello (che è di 320x320 pixel). Per mantenere l'aspect ratio di ogni immagine viene aggiunto un bordo di opportuno spessore, e si fa uso di API messe a disposizione da Tensorflow per ricollocare correttamente le box sulle immagini.

Figura 1: le immagini sul quale viene effettuato il training hanno una dimensione di 320x320px. Per adattarle senza perdere il fattore di forma originale vengono aggiunti bordi.

Questo passaggio di ridimensionamento potrebbe in teoria essere gestito automaticamente da Tensorflow con l'opzione *image_resizer* e altre opzioni di data augmentation che è possibile specificare all'interno del file di configurazione *pipeline.config*. Nella pratica si è però constatato che le opzioni da impiegare nel file di configurazione sono diverse dalle opzioni che si passano alle API Python, e ad oggi non esiste una documentazione su queste opzioni (si è infatti reso necessario analizzare il repository ufficiale GitHub di Tensorflow, e in particolare visitare i vari file contenenti le definizioni di questi parametri). Nonostante i numerosi tentativi non è stato possibile trovare delle opzioni equivalenti da mettere nel file di configurazione, quindi il preprocessing è stato effettuato direttamente nello script che genera i file TFRecord.

Inoltre, il resizing effettuato dall'opzione del file di configurazione è diverso dal resizing effettuato dalle API Python.

Nelle seguenti immagini è possibile vedere a sinistra il risultato del resizing effettuato dalle opzioni del file di configurazione, mentre a destra quello effettuato dalle API Python (in entrambi i casi è stata specificata un'interpolazione di tipo "area").

Figura 2: a sinistra possiamo vedere l'immagine scalata dal supporto con i parametri di pipeline.config, a destra vediamo la medesima immagine scalata con le API Python di Tensorflow. L'immagine più a sinistra sembra essere più sgranata, sebbene il numero di pixel sia il medesimo.

L'immagine di sinistra sembra avere una qualità peggiore. I risultati di un training con il resizing effettuato dal file di configurazione risultano essere molto più scadenti, mentre il problema non si pone nel caso in cui le immagini siano già della dimensione giusta (l'opzione in *pipeline.config* è sempre presente, ma in questo caso non effettua alcuna operazione).

Generati i file TFRecord, è poi stato scelto un pretrained model sul quale Tensorflow lavorerà per renderlo adatto al problema che si vuole risolvere.

ADDESTRAMENTO MODELLO

Per questo progetto si è scelto di utilizzare, tra le varie opzioni possibili [4], la *SSD MobileNet V2 FPNLite 320x320*. Si tratta di una rete che è stata pre-addestrata facendo uso del dataset *COCO* e che, essendo una *SSD*, fa uso di un'architettura che permette di elaborare i risultati molto più velocemente rispetto ad altre soluzioni basate su Region Proposal. Inoltre, tale rete potrà essere facilmente utilizzata grazie alla "*Task Library ObjectDetector API*" fornita da Tensorflow, che si occuperà delle operazioni di manipolazione dei dati di input e di output, fornendo interfacce di più facile utilizzo. La rete prende comunque in ingresso un tensore che rappresenta l'immagine a colori, e restituisce in output il numero di detections valide insieme ad alcuni array per specificare bounding box, classe di appartenenza e la confidenza di ogni detection.

Come specificato sulla pagina di tutorial della Object Detection API, è necessario scaricare la directory nella quale predisporre il training, porre i TFRecords e il modello che si decide di utilizzare nelle giuste directory. Successivamente si modificano opportunamente i parametri del file *pipeline.config* incluso nel modello scaricato.

In particolare, i parametri di *pipeline.config* che sono stati modificati rispetto alla versione di default, sono:

- *batch_size*: nel nostro caso è stato possibile incrementarlo a 64 (disponendo di un PC con 16GB di RAM)
- *num_steps* e *total_steps*: numero di steps dopo il quale il processo di training termina (è comunque possibile terminare prima il training, ed è anche possibile utilizzare uno qualunque dei checkpoint che vengono automaticamente salvati ogni 1000 steps).
- I path relativi alle opzioni *train_input_reader* e *eval_input_read*er che contengono sia i path dei TFRecords, sia il path ad un file che contiene i nomi e i corrispondenti indici numerici delle label (*label_map.pbtxt*). Il file verrà poi usato in fase di conversione del modello in TFLite per generare un altro file di metadati, che possa poi essere incluso nella applicazione Android, che potrà in questo modo mostrare le relative stringhe sopra ogni detection.
- Sono state disabilitate le opzioni di data augmentation che modifichino le dimensioni delle immagini (per gli stessi motivi riguardanti lo scaling).

A questo punto è possibile avviare il training, utilizzando ancora una volta lo script e le procedure descritte nella guida. Si è poi fatto uso delle COCO API per valutare il modello durante la fase di training ogni 1000 steps (esiste uno script in grado di monitorare la directory dove sono salvati i checkpoint, ad ogni nuovo checkpoint viene fatta una misura).

Tali metriche sono visibili, insieme a tanti altri dati, in Tensorboard.

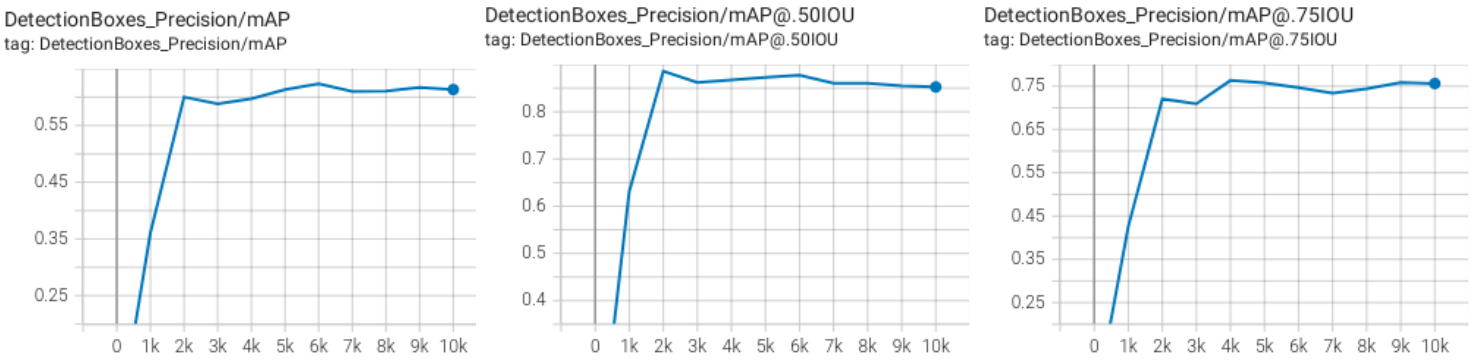

Figura 3: vengono qui mostrati alcuni dei grafici visibili da Tensorboard per quanto riguarda la Mean Average Precision

Come si può osservare l'average precision si attesta intorno a 0.6, ma questo non vuol dire che la rete non rilevi gli oggetti (o li rilevi con la classe sbagliata) nel restante 40% delle volte.

La Mean Average Precision "standard" di *COCO* si basa invece sul valore di IoU (Intersection Over Union) ossia sul "quanto bene" si sovrappone la box rilevata con la ground truth box.

Se ci accontentiamo di contare come valide le box che hanno un livello di sovrapposizione del 75% otteniamo una mAP di circa 0.75, mentre con una sovrapposizione del 50% siamo sopra a 0.85. Come prevedibile, la rete reagisce molto meglio in presenza di oggetti di dimensioni grandi e medie, mentre fa molta più fatica per oggetti piccoli (oggetti con area minore di 32*32 pixel hanno una mAP di 0.4).

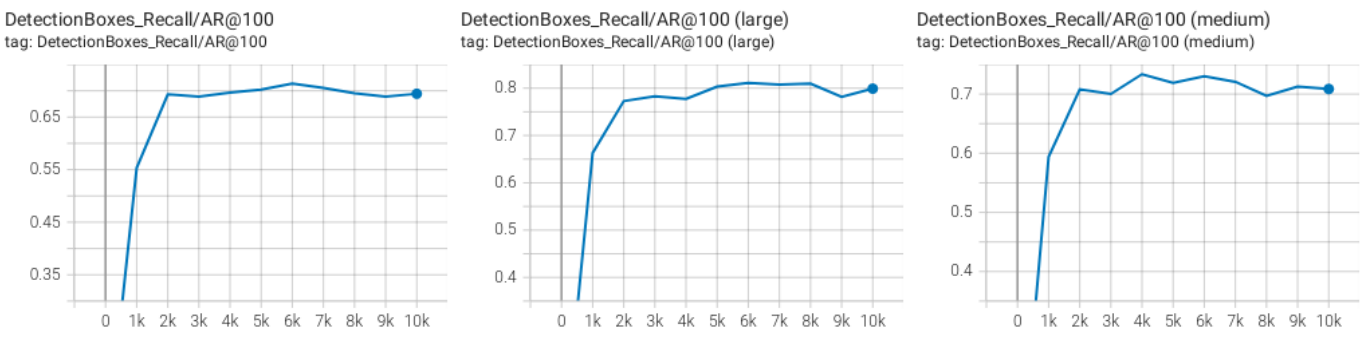

Figura 4: vengono qui mostrati alcuni dei grafici visibili da Tensorboard per quanto riguarda l'Average Recall

Riguardo l'Average Recall possono essere fatte considerazioni simili, con valori che si attestano intorno allo 0.7, e che risultano essere più elevati nel caso di oggetti di dimensioni più grandi.

Come è possibile vedere anche dai grafici, la rete tende a non migliorare più di tanto dopo un certo numero di steps, però la loss ha perlopiù continuato a migliorare (soprattutto per quanto riguarda la localization loss).

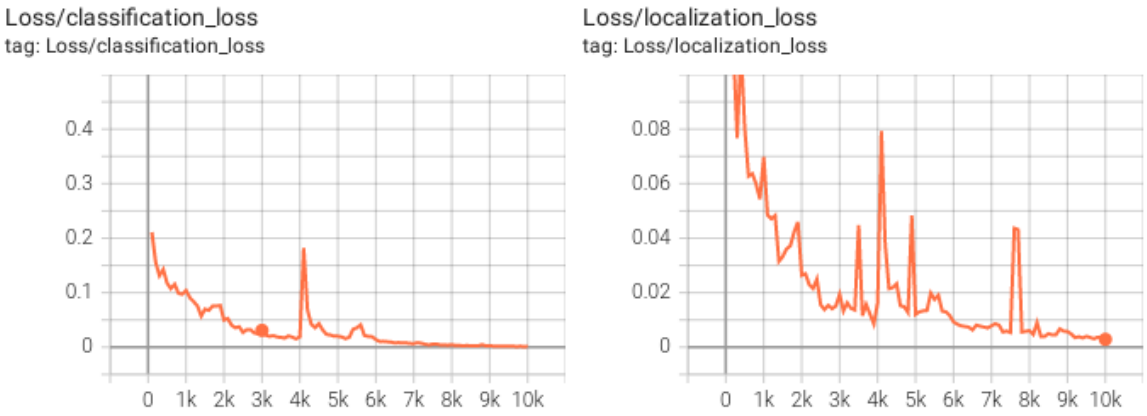

Figura 5: vengono qui mostrati alcuni dei grafici visibili da Tensorboard per quanto riguarda la loss

ESTRAZIONE E CONSIDERAZIONI SUL MODELLO

Tensorflow Object Detection API effettua il training salvando ogni 1000 steps un checkpoint dai quali è poi possibile estrarre il modello in formato "*saved model*", che include anche il file ".pb". Per farlo basta anche qui seguire le indicazioni della guida, che indica come eseguire lo script che si occuperà di tale conversione. Lo script prende in considerazione il checkpoint indicato nel file "*checkpoint*" dal parametro "*model_checkpoint_path*".

Si è dapprima proceduto ad utilizzare il modello con 10000 steps di training, ma ci si è presto resi conto che venivano rilevati un numero considerevole di falsi positivi, tutti con probabilità prossima ad 1.

È poi stato sviluppando uno script in grado di fare inferenza su un video in ingresso (*video_loader.py*). Grazie a questo script si è potuto constatare come il modello con 5000 steps di training risultasse dare ancora un certo numero di falsi positivi, ma con probabilità più basse e senza impattare sulle detection degli oggetti effettivamente presenti (i cui valori di confidenza rimangono pressoché ad 1). Si è quindi convenuto di utilizzare il modello con 5000 steps, e di alzare a livelli molto elevati la confidenza per filtrare i falsi positivi.

CONVERSIONE IN TENSORFLOW LITE

Il modello estratto nel formato *"saved model"* non è ancora adatto ad essere utilizzato su dispositivi mobili, per farlo è necessario convertirlo nel formato TFLite. Questo può essere fatto usando le apposite API messe a disposizione da Tensorflow.

In particolare si fa uso di *tf.lite.TFLiteConverter.from_saved_model()* e del metodo *.convert()* da invocare sull'oggetto così creato. Si è sperimentato facendo anche uso di tecniche di quantizzazione, ma la full integer quantization (che permetterebbe un miglioramento dal punto di vista della CPU, e quindi dei tempi di inferenza) si è rilevata essere decisamente troppo penalizzante in termini di accuratezza dei risultati. Non si osservano invece differenze di rilievo nel caso in cui i pesi vengano salvati come interi ma elaborati come float (post-training dynamic range quantization).

```
69  #tesorflow lite conversion
70  converter = tf.lite.TFLiteConverter.from_saved_model(_SAVED_MODEL_PATH)
71  converter.optimizations = [tf.lite.Optimize.DEFAULT]
72  converter.allow_custom_ops = True
73  converter.experimental_new_converter = True
74  converter.target_spec.supported_ops = [
75    tf.lite.OpsSet.TFLITE_BUILTINS_INT8, # Ensure that if any ops can't be quantized, the converter throws an error
76    tf.lite.OpsSet.TFLITE_BUILTINS, # enable TensorFlow Lite ops.
77    tf.lite.OpsSet.SELECT_TF_OPS # enable TensorFlow ops.
78  ]
79  converter.representative_dataset = representative_data_gen
80  tflite_model = converter.convert()
81  with open(_TFLITE_MODEL_PATH, 'wb') as f:
82      f.write(tflite_model)
```

Figura 6: funzioni di conversione da "saved model" a TFLite "base"
Il frammento di codice mostra come è possibile convertire il modello usando full integer quantization
(in particolare le righe da 71 a 79 comprese settano i parametri per la quantizzazione).

Questo genererà un file senza metadati, utili nel caso di Object Detection poiché contengono parametri che potranno poi essere utilizzati dalla "*Task Library ObjectDetector API*" per manipolare correttamente i dati (ad esempio viene inserito il nome del file contenente le stringe da associare ad ogni classe). Si fa quindi uso di una ulteriore API, come si vede nel seguente frammento.

```
104 from tflite_support.metadata_writers import object_detector
105 from tflite_support.metadata_writers import writer_utils
106 writer = object_detector.MetadataWriter.create_for_inference(
107     writer_utils.load_file(_TFLITE_MODEL_PATH), input_norm_mean=[127.5],
108     input_norm_std=[127.5], label_file_paths=[_TFLITE_LABEL_PATH])
109 writer_utils.save_file(writer.populate(), _TFLITE_MODEL_WITH_METADATA_PATH)
```

Figura 7: funzioni che permettono di aggiungere metadati al modello TFLite

Sono stati creati due appostiti script per svolgere queste catene di conversioni, in particolare *tflite_no_quant.py* esporta il modello senza alcuna ottimizzazione, mentre *tflite_quant.py* usa full integer quantization.

ANDROID

Per fare uso della rete addestrata è stato scelto di impiegare una applicazione su sistema Android, poiché presenta una notevole base installata di dispositivi con a disposizione una fotocamera.

Inoltre, è stato possibile implementare l'applicazione utilizzando linguaggi di programmazione a noi familiari. Infine, il sistema Android presenta numerose librerie in grado di gestire il modello precedentemente generato.

Per riuscire a sviluppare la nostra applicazione siamo partiti da un esempio messo a disposizione da Tensorflow [5].

STRUTTURA

L'applicazione risulta formata da due activity, una che presenta la fotocamera con le box rilevate ed i valori misurati, mentre la seconda gestisce le impostazioni del programma.

Figura 9: Camera Activity

Figura 8: Settings Activity

CAMERA ACTIVITY

La seguente activity risulta essere la schermata principale dell'applicazione.

Essa presenta nella parte superiore una visualizzazione della fotocamera, con le immagini già processate dal modello precedentemente creato, mentre nella parte inferiore si visualizzano i valori, in tempo reale, rilevati a schermo.

In particolare, sono mostrati il numero di persone con e senza mascherina e la percentuale di elementi con la mascherina.

Inoltre, se opportunamente abilitata nella schermata di impostazioni, è possibile visualizzare il numero di persone con e senza mascherina che transitano.

Attraverso la classe Java *DetectorActivity* viene gestito il processamento delle immagini tramite l'utilizzo del modello sviluppato in Tersorflow lite.

La classe gestisce i frame provenienti dalla fotocamera e li elabora tramite l'oggetto *TFLiteObjectDetectionAPIModel* fino ad ottenere una lista di *Detector.Recognition*, ovvero la lista degli elementi rilevati a schermo, che verranno poi utilizzati all'interno della applicazione.

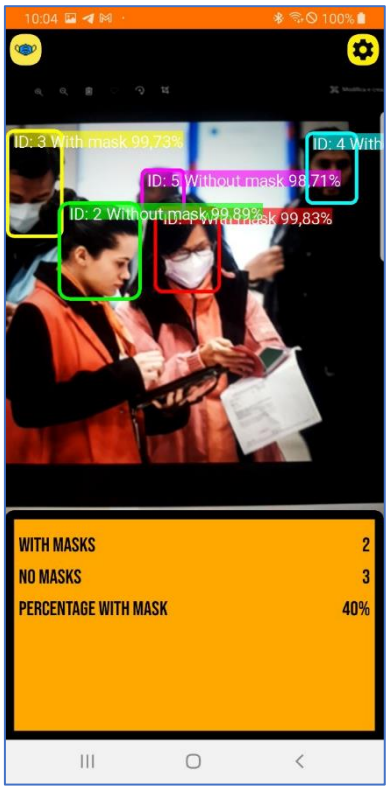

Figura 10: funzionamento CameraActivity

SETTINGS ACTIVITY

La seguente activity ha il compito di gestire le impostazioni della applicazione.

È possibile selezionare, tramite apposite checkbox, quali classi di elementi si vogliono gestire (con o senza mascherina).

Inoltre, è possibile abilitare la "Tracking mode", ovvero la possibilità di usare l'applicazione come conta persone.

L'attivazione del tracking mode da', inoltre, la possibilità di selezionare due ulteriori checkbox, una relativa alla visualizzazione a schermo della linea atta al conteggio delle persone e l'altra relativa a raffigurare il percorso fatto dal punto centrale degli elementi a schermo.

La gestione delle checkbox è stata effettuata dalla classe Java singleton *CheckDetect* che tiene traccia dello stato dei vari elementi.

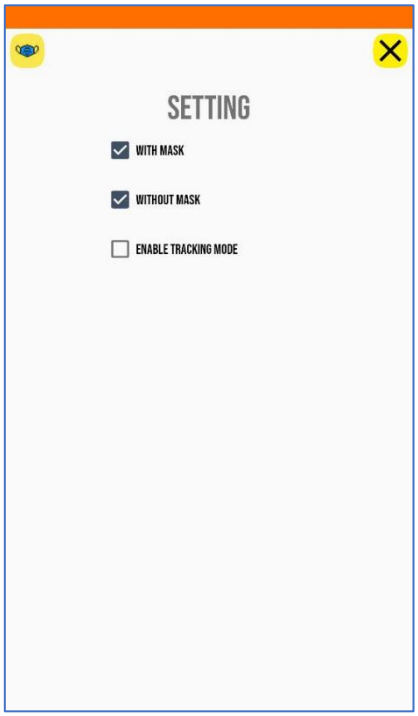

Figura 12: Settings activity

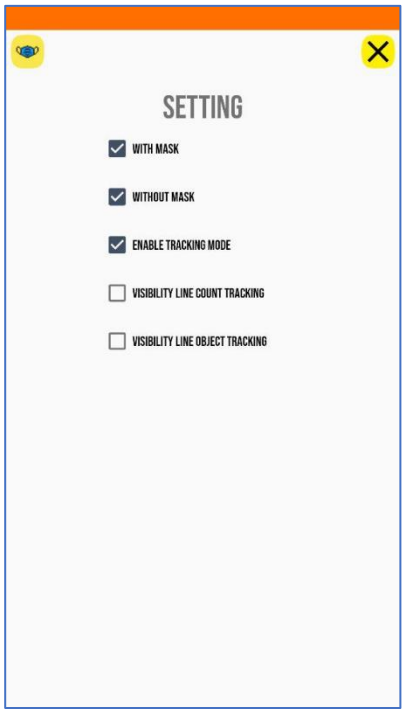

Figura 11: Settings activity

TRACKING

Il tracking è stato progettato e implementato tramite un algoritmo in grado di tracciare gli oggetti basandosi, volta per volta, sulla posizione dei vari elementi nei nuovi frame.

La logica si basa sul rilevamento degli oggetti ed il loro successivo salvataggio dei punti centrali, ovvero il punto dato dalla intersezione delle due diagonali della box. L'utilizzo dei punti centrali rappresenta, nel nostro caso, il miglior compromesso tra costo ed efficienza.

È stato sviluppato un algoritmo che ad ogni nuovo punto rilevato stabilisce se è opportuno associarlo con l'ultimo punto di un oggetto già esistente, oppure considerarlo come un nuovo elemento.

Inoltre, tale codice è stato reso robusto - nei limiti del possibile - alla perdita temporanea di rilevazione degli oggetti da parte della rete: gli oggetti tracciati rimangono in memoria per alcuni frame prima di essere eliminati.

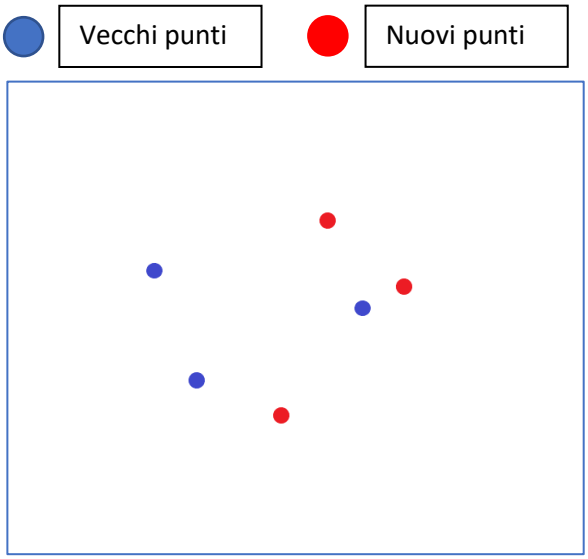

Figura 13: Center points prima del tracking

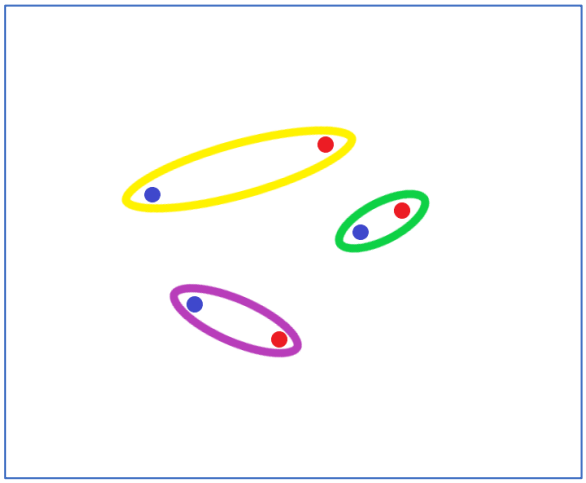

Figura 14: Center points dopo il tracking

LOGICA DELL'ALGORITMO

La logica dell'algoritmo prevede i seguenti passaggi:

1. Per ogni nuovo punto N_i, si trova il punto vecchio più vicino V_j
2. Si calcola il nuovo punto N_k più vicino a V_j, e si verifica se questo sia lo stesso punto di partenza N_i (ossia se $N_k == N_i$):
   2.1. Se $N_k \neq N_i$, si ricomincia dal punto 1 con il successivo punto N_{i+1}.
   2.2. Se $N_k = N_i$, i punti N_i e V_j sono reciprocamente i più vicini l'un l'altro. Si assegna al vecchio oggetto il nuovo punto N_i.
   Si ricomincia dal punto 1 con il successivo punto N_{i+1} (N_i e V_j non potranno essere più presi in considerazione per nuove associazioni).
3. Si ripete l'intero processo (dal punto 1) fino a che non rimangono o solo punti nuovi, o solo punti vecchi. La ripetizione è necessaria perchè il caso *2.1* fa sì che vi possano essere ancora coppie non associate.

CASISTICHE DEI PUNTI CENTRALI

Vecchi punti Nuovi punti

L'algoritmo descritto permette di associare coppie di punti, senza il rischio di collegare immediatamente un punto vecchio con uno nuovo che potrebbe avere un altro punto vecchio più vicino, e viceversa.

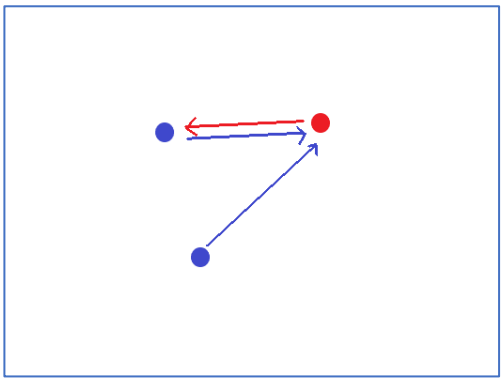

Figura 18:
Caso 1: Viene associato un solo punto vecchio con quello nuovo, quello più vicino. L'altro oggetto vecchio viene comunque mantenuto in memoria per un certo numero di frame.

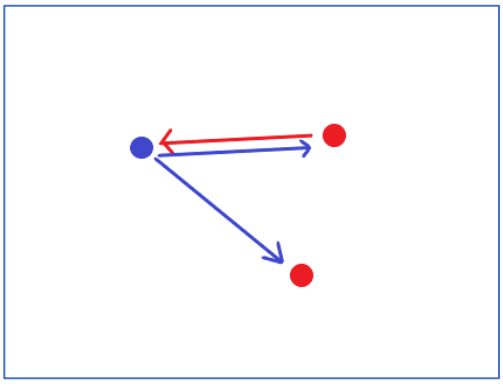

Figura 17:
Caso 2: Viene associato un solo punto nuovo con quello vecchio. Il restante punto nuovo identificherà un nuovo oggetto.

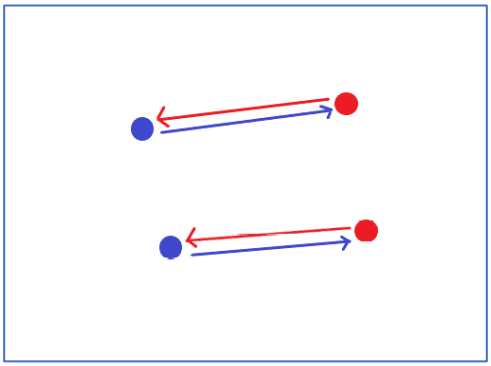

Figura 16:
Caso 3: Vengono associati tuti i punti nuovi con i rispettivi punti vecchi.

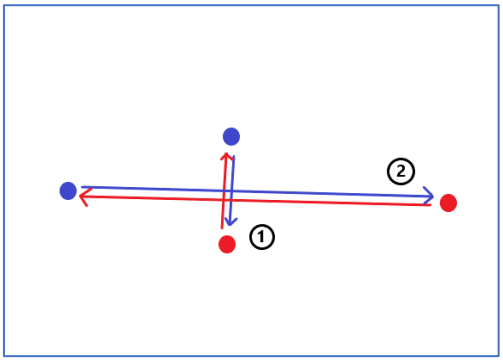

Figura 15:
Caso 4: L'associazione dei punti è contestuale. Come è possibile vedere nell'esempio vengono prima associati i punti della coppia 1 e successivamente vengono associati i punti della coppia 2.

IMPLEMENTAZIONE DELL'ALGORITMO

L'algoritmo è stato sviluppato all'interno della classe Java *Tracking*.

Essa presenta una lista di *TrackedObject* che rappresenta le informazioni relative ad ogni singola box. Ogni *TrackedObject* presenta le informazioni dell'oggetto e la storia passata dei punti centrali di esso.

Viene poi implementato l'algoritmo vero e proprio mediante la funzione *update* il cui compito, data una lista di *Detector.Recognition* che sono gli oggetti rilevati nel frame corrente, è di confrontare la storia passata degli elementi.

Una volta effettuata l'associazione tra i punti possono essere collegati o ad oggetti esistenti o a nuovi oggetti creati appositamente. In particolare l'oggetto viene creato solo nel caso in cui questo presenti una confidenza molto elevata.

Infine, vengono eliminati gli oggetti non rilevati per un certo numero di frame.

Per il collegamento tra i punti viene effettuata una associazione contestuale, ovvero vengono associate le coppie di punti che sono, nell'ambiente, più vicini tra di loro.

Nel caso il punto fosse molto lontano verrebbe considerato come un nuovo elemento.

Per fare ciò è stata utilizzata la libreria *Guava* di Google che fornisce una interfaccia per la gestione di tabelle, affine alle nostre esigenze potendo associare ad ogni riga ed ogni colonna un punto centrale e, nelle caselle corrispondenti, le loro distanze.

In questo modo viene ricreata una lista di *Detector.Recognition* che poi viene passata alla classe *DetectorActivity* che ha poi il compito mostrare le immagini comprensive di detections.

CONTA PERSONE

Grazie all'algoritmo implementato siamo poi in grado di verificare se un determinato punto supera o meno una linea, aggiornando in tal caso l'opportuno contatore.

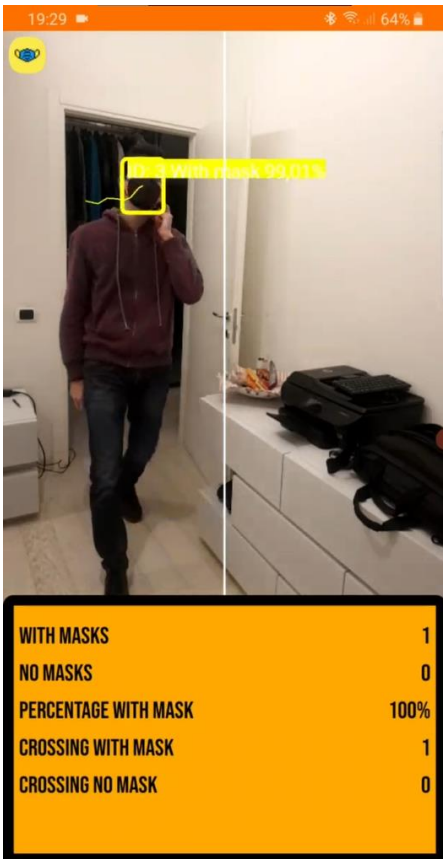

Figura 19: Rappresentazione del funzionamento del conta persone

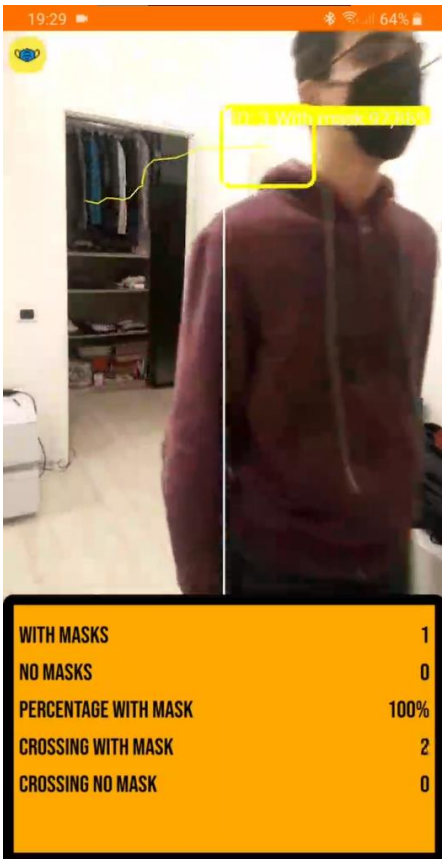

Figura 20: Rappresentazione del funzionamento del conta persone

Questo è stato possibile grazie al mantenimento della storia passata degli oggetti e della loro posizione.

Confrontando infatti la posizione del frame attuale e quella del frame precedentemente rilevato è possibile rilevare l'eventuale attraversamento della linea, incrementando o decrementando quindi il conteggio in base alla direzione. È stato inoltre implementato all'interno della applicazione la possibilità di visualizzare, tramite opportune checkbox, la linea utilizzata per il conta persone ed il percorso effettuato dai vari punti centrali a schermo.

CONCLUSIONI

Gli obiettivi preposti per lo sviluppo del progetto riteniamo essere stati raggiunti con risultati soddisfacenti.

Il modello della rete, da noi addestrato, risulta essere sufficientemente affidabile. Risultati migliori potrebbero comunque essere ottenuti con un dataset di immagini più ampio e di migliore qualità. In particolare il modello attuale rileva efficacemente i volti disposti frontalmente, tuttavia la rete continua a rilevare anche le persone voltate di spalle, non potendo inevitabilmente definire correttamente la classe di appartenenza.

L'applicazione risulta essere sufficientemente fluida e di facile utilizzo e mostra a schermo tutte le informazioni che riteniamo utili all'utente.

L'algoritmo da noi implementato per la funzionalità del "Conta persone", sebbene basato sulla sola posizione degli oggetti, risulta essere piuttosto efficace.

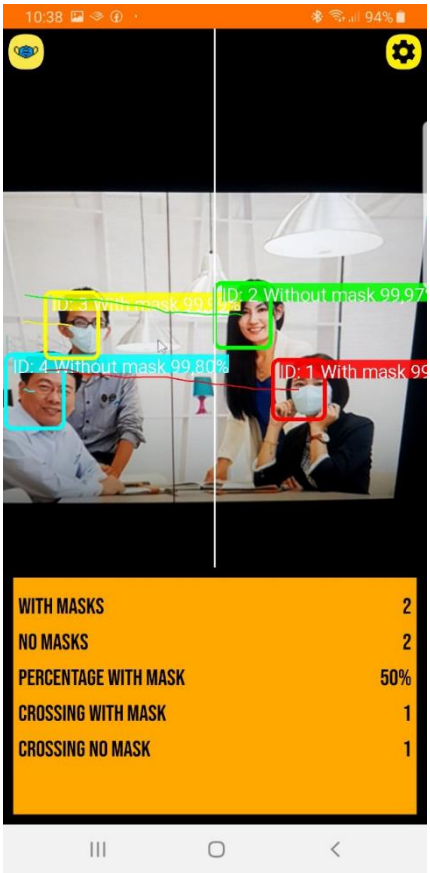

Figura 22: Rappresentazione del funzionamento del tracking

Figura 21: Rappresentazione del funzionamento del conta persone

BIBLIOGRAFIA

[1] «TensorFlow 2 Object Detection API tutorial,» [Online]. Available: https://tensorflow-object-detection-api-tutorial.readthedocs.io/en/latest/.

[2] «Kaggle 1o dataset,» [Online]. Available: https://www.kaggle.com/andrewmvd/face-mask-detection.

[3] «Kaggle 2o dataset,» [Online]. Available: https://www.kaggle.com/wobotintelligence/face-mask-detection-dataset.

[4] «Model Zoo TensorFlow 2 Detection,» [Online]. Available: https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md.

[5] «GitHub applicazione esempio,» [Online]. Available: https://www.tensorflow.org/lite/examples/object_detection/overview.