

CARMINE PALMESE

Hand Tracking e Gesture Recognition con YOLOv5

AIchools Unimore

Introduzione

Lo scopo di questo progetto è quello di voler realizzare un sistema di Hand Tracking e Gesture Recognition basato su rete neurale per implementare una procedura di controllo interattivo real-time da webcam.

I potenziali utilizzi di questo sistema possono ritrovarsi nell'industria tessile o in quella metallurgica. Sono infatti molto diffuse macchine a controllo numerico per il taglio automatico di pezzi su basi di diversi materiali, il cui processo di taglio prevede sempre il piazzamento di un certo numero di pezzi su un canvas in modo da indicare alla macchina quali sono le posizioni dei tagli da effettuare. Questa è una procedura che può essere anche pensata in automatico con algoritmi di nesting [2] ma rendere il processo di piazzamento completamente automatico diventa difficile quando si considerano materiali di qualità non uniforme, con difetti o semplicemente con delle forme particolari (e quindi non poligoni riconoscibili).

È quindi spesso prevista la presenza di un operatore che si occupa di piazzare i pezzi da tagliare in modo da ottimizzare meglio i materiali utilizzati e di programmare i diversi batch di pezzi tagliati. Questa procedura è attualmente implementata con un monitor su cui è rappresentato il canvas di taglio e con un mouse si procede quindi a piazzare i pezzi desiderati. A volte è anche presente un sistema di proiezione sul materiale da tagliare per avere una visione realistica di dove si stanno piazzando i vari pezzi.

L'idea di questo progetto è quella di mantenere il sistema di proiezione su pelle ma andando a sostituire il sistema mouse + monitor con un sistema di computer vision che rileva la posizione e la gestione delle mani tramite una telecamera, e in base a queste realizzare la procedura di piazzamento grazie alle sole mani dell'utente.

Descrizione del problema e soluzioni

Il problema del tracking è riconducibile al problema di Object Detection in real-time. Effettuando varie ricerche, questo problema è affrontato ormai da svariati tipi di reti neurali. Ne andiamo ad esaminare alcuni.

R-CNN

Il problema principale nell'ambito dell'Object Detection è quello di identificare un numero non costante di oggetti in un'immagine. Potrebbero infatti essere presenti uno svariato numero di oggetti da identificare e in posizioni spaziali molto diverse tra loro, oltre al fatto che possono avere dimensioni diverse tra loro.

R-CNN [x] è un metodo proposto per l'Object Detection che prevede una fase iniziale di estrazione di un numero fissato di regioni dell'immagine (2000) da far analizzare a una CNN. In questo modo si riducono molto il numero di regioni da provare a identificare in un'immagine e la CNN si occupa di fare solo feature extraction utile a una classificazione.

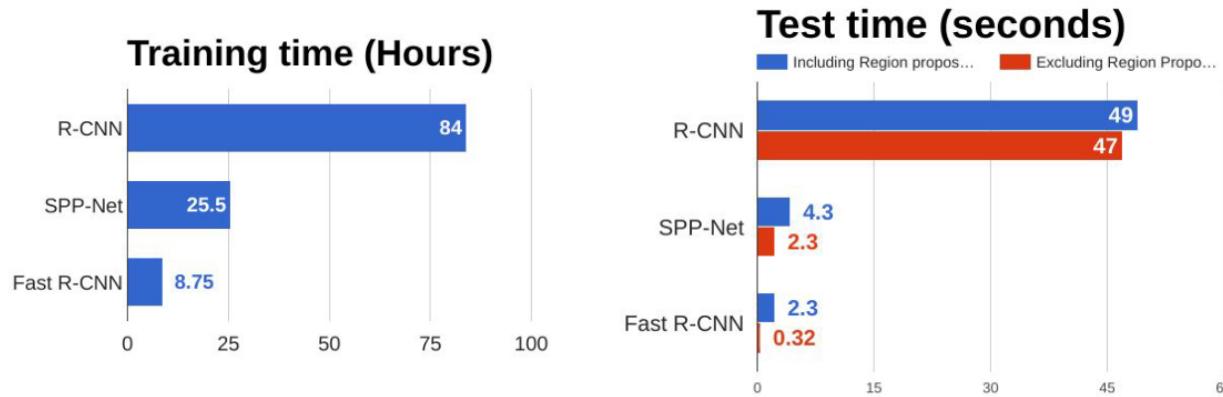
Il metodo è molto utile per ridurre la ricerca di aree da analizzare, ma il problema principale è che è molto lento (deve comunque analizzare 2000 regioni), il ché lo rende inadatto per utilizzi in ambito real-time.

Inoltre un altro problema è che l'algoritmo usato per selezionare le regioni è un algoritmo statico, non è soggetto a learning e quindi la generazione di regioni candidate alla classificazione potrebbe essere non ottimale.

Fast R-CNN

È un miglioramento della R-CNN. Invece di generare le regioni da analizzare con l'algoritmo statico, si fa passare l'intera immagine per una CNN che crea una feature map dell'immagine. Da questa si generano quindi le regioni attraverso un algoritmo, e vengono fatte passare per vari livelli di pooling e fully-connected. A valle del processo, si classifica l'oggetto tramite softmax e si ottiene la bounding box da un regressore.

Il vantaggio principale è chiaramente la generazione delle regioni, che è velocizzata semplicemente addestrando una rete convolutiva a generare feature map per la generazione ottimale di regioni.



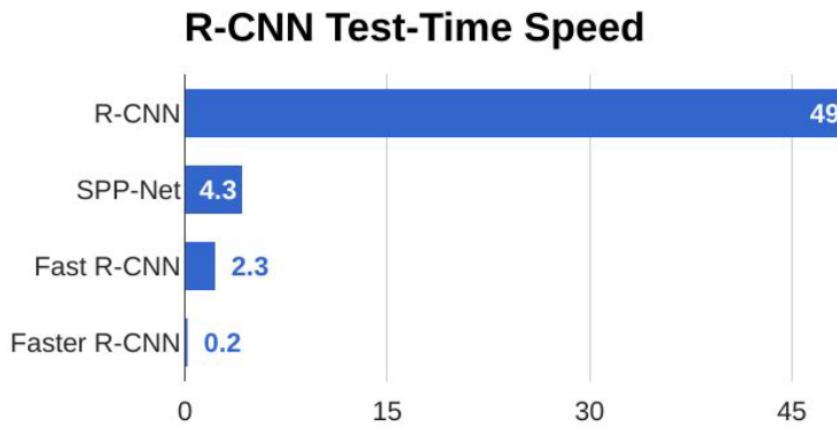
Benchmark di confronto tra R-CNN, Fast R-CNN e SPP-Net [x]

Dai benchmark si nota che comunque la maggior parte del tempo di esecuzione è occupato dalla generazione di regioni, e soprattutto che anche se si è avuto uno speed-up non indifferente, ancora non è adatto a eseguire in ambito real-time.

Faster R-CNN

Per la generazione delle regioni dalla feature map si introduce un'ulteriore rete per predire le regioni (eliminando così l'algoritmo di ricerca precedentemente utilizzato).

Questo metodo è chiaramente il più veloce, grazie all'eliminazione dell'overhead introdotto dall'algoritmo usato per la generazione delle regioni di interesse. Questo lo rende più adatto ad un utilizzo in ambiente real-time, anche se guardando i benchmark abbiamo comunque un tempo di inferenza di 0.2 secondi. È comunque un buon risultato, ma è capace di analizzare solo 5 frame al secondo.



Benchmark di confronto [x]

YOLO e SSD

Le reti **single shot** come YOLO e SSD rimuovono completamente il concetto delle regioni di interesse e si concentrano più sull'analisi dell'intera immagine.

In particolare, dividono l'immagine in una griglia con blocchi quadrati di dimensione fissata. Si genera quindi, per ogni blocchetto, un set di bounding box di base centrate nel quadrato. Si effettua quindi una regressione delle bounding box a una definitiva che terrà conto delle sue dimensioni, della sua posizione e della confidenza della regressione. Parallelamente si calcola quindi lo score delle classi da riconoscere (includendo anche lo sfondo). Vengono quindi mostrate in output solo le bounding box con un certo livello di confidenza.

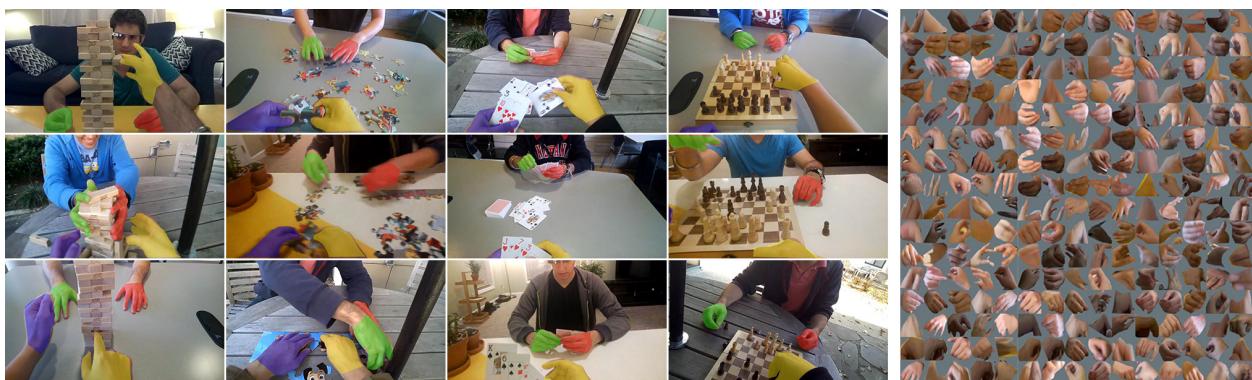
Prendendo come esempio YOLO, questa è molto più veloce degli algoritmi di Object Detection che abbiamo analizzato, permettendo l'analisi di svariate decine di frame al secondo. La sua più grande limitazione è dovuta ai constraint spaziali delle bounding box, che a volte la rendono incapace di identificare oggetti troppo grandi o troppo piccoli.

YOLOv5 [5] è la rete che è stata presa in considerazione per l'implementazione dell'Hand Tracking discusso in questo progetto. Non è ancora presente un paper ufficiale di YOLOv5, ma per una struttura dettagliata del suo funzionamento si rimanda al paper della sua versione precedente: YOLOv4 [7].

Dataset

Sono stati esaminati e testati principalmente due dataset: il dataset *EgoHands* [3] e il dataset *HANDS* [4].

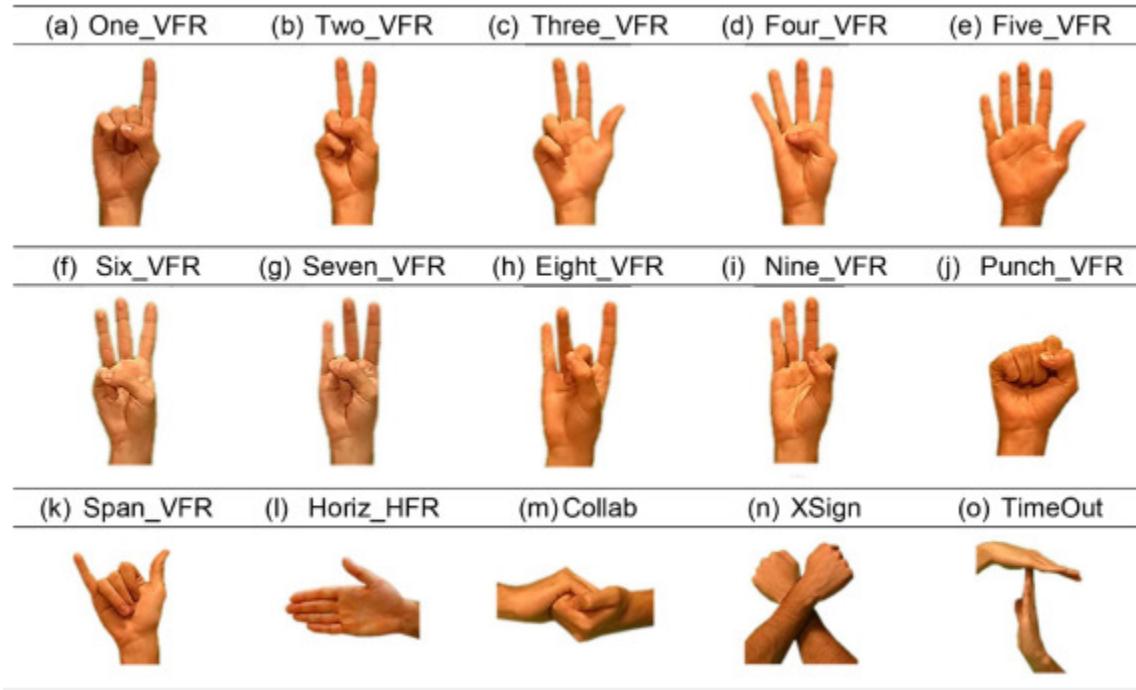
Il dataset *EgoHands* è stato usato inizialmente per effettuare dei test con YOLO ma ha portato con sé molte limitazioni. Questo dataset è stato ottenuto registrando dei filmati di due persone che giocano a diversi giochi da tavolo in vari ambienti, dal punto di vista di una delle due. Le classi con cui il dataset è stato etichettato quindi sono quattro: mano dal punto di vista di "chi registra", sinistra o destra, oppure mano dell'avversario, sempre sinistra o destra. In fase di training, questo dataset è riuscito a dare qualche risultato decente, ma era tutto molto dipendente dalla prospettiva con cui si posizionavano le mani, e soprattutto si sarebbe dovuto aggiungere un passo in più di classificazione delle gesture.



EgoHands Dataset

Il dataset *HANDS* è stata la scelta finale, in quanto non solo contiene molte più immagini del precedente (sempre frame di video), ma contiene anche la classificazione di ben 28 gesture diverse. Questo mi ha consentito di effettuare con una sola rete sia detection che gesture recognition.

Un'altra idea (non implementata) sarebbe stata quella di effettuare un pre-training con il dataset *EgoHands* per avere una base di partenza più robusta, e poi effettuare un secondo training definitivo con il dataset *HANDS*.



HANDS Dataset

Il dataset *HANDS* è stato diviso in questo modo:

- **Training Set:** 9600 immagini
- **Validation Set:** 1200 immagini
- **Test Set:** 1200 immagini

La divisione è effettuata da uno script (*test_valid_test_splitter.py*) che aleatoriamente assegna le immagini del dataset in una di queste tre categorie.

Training

Preprocessing

Una cosa che si è tenuta in considerazione durante lo sviluppo del progetto è stata quella di effettuare del preprocessing sulle immagini del dataset. Questo deriva prima di tutto da un fatto di robustezza del tracking su webcam, e quest'ultima spesso non è di ottima qualità e quindi soggetta a vari disturbi sull'immagine. Inoltre sono presenti spesso situazioni di luce diverse che possono portare quindi a immagini bruciate o poco illuminate. Inoltre, essendo il dataset composto da tanti fotogrammi di video, le immagini per il training sono molto simili e la rete potrebbe essere prona a overfitting (cosa che dopo si verificherà).

Il preprocessing effettuato consiste in un randomizzatore di luminosità nella foto e in un addizionatore di rumore gaussiano sulle immagini in ingresso. Questo per rendere le immagini visivamente più eterogenee e rendere la rete capace di convergere a una feature extraction in modo più robusto.

Per il pre-processing è stato messo a disposizione uno script che richiama automaticamente e in ordine gli algoritmi Python per le modifiche citate. Questo script inoltre si occupa anche automaticamente di scaricare il dataset, ed è parametrizzato in base al dataset da scaricare (*EgoHands* o *HANDS*).

Per ulteriori dettagli su questi algoritmi si faccia riferimento al codice sorgente su GitHub [1].

Processo

Il training è stato effettuato tramite le istruzioni indicate dagli stessi sviluppatori di YOLOv5 [6].

Attraverso gli script di training messi a disposizione dagli sviluppatori è stato possibile effettuare vari test di training con impostazioni diverse, come batch size, training parallelo su più GPU e caching in RAM delle immagini.

Per agevolare la procedura è stato realizzato uno script con parametri di ingresso per effettuare in automatico il comando di training con le varie opzioni, includendo la tipologia della rete e se effettuare il training in parallelo o no.

Al termine di ogni processo di training inoltre vengono memorizzati vari dati per la valutazione complessiva della procedura:

- **matrice di confusione** di tutte le classi + il background
- **score F1**: media armonica tra precision e recall
- **precision**
- **recall**

Una cosa importante di cui tenere conto è che ogni processo di training è stato effettuato su un modello pre-trained della YOLOv5 con un dataset COCO. Questa scelta è stata presa per avere una rete con dei pesi già calcolati, per agevolare il riconoscimento in fase di training.

Ogni sessione di training è stata effettuata su 2 GPU nVidia GeForce RTX 3090, sfruttando l'opzione messa a disposizione dallo script di training della YOLOv5 per il training in parallelo. Questo ha consentito dei training molto più veloci. È stata valutata anche l'opzione di effettuare test sulla piattaforma Google Colab con l'ambiente GPU messo a disposizione, ma da vari test si è notato che per modelli più grandi della YOLO il tempo era estremamente alto.

Alla fine dei training sono stati eseguiti vari test di detection in real-time con una webcam, per cui sono disponibili dei video (il link è disponibile nella sezione di *Risultati e Benchmark*).

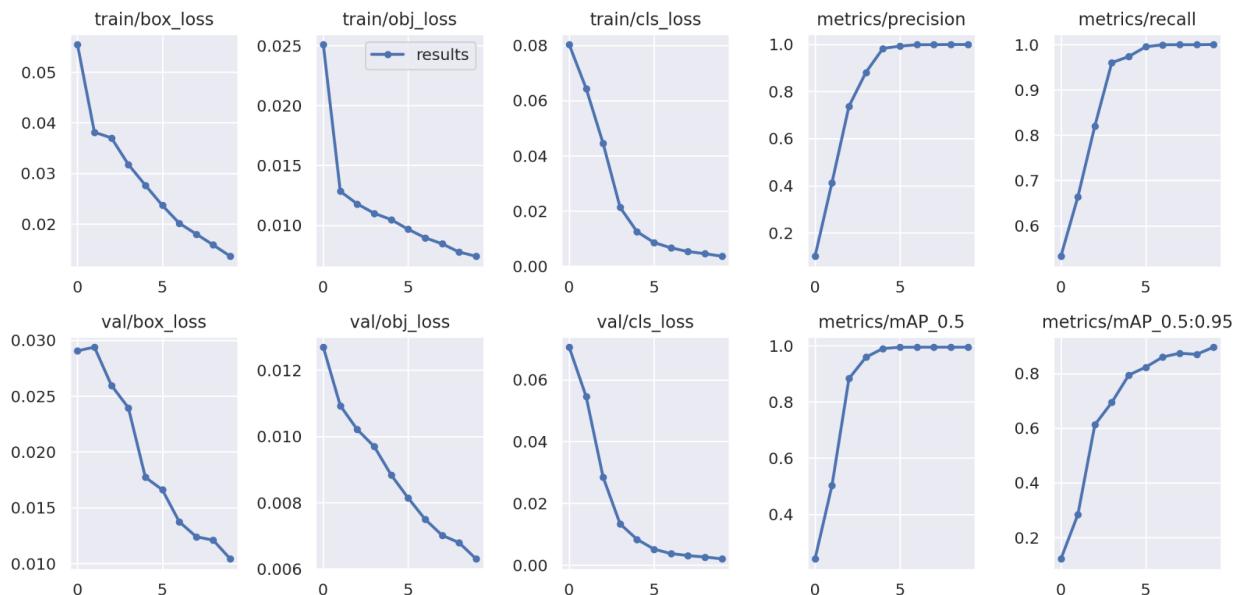
Risultati e Benchmark

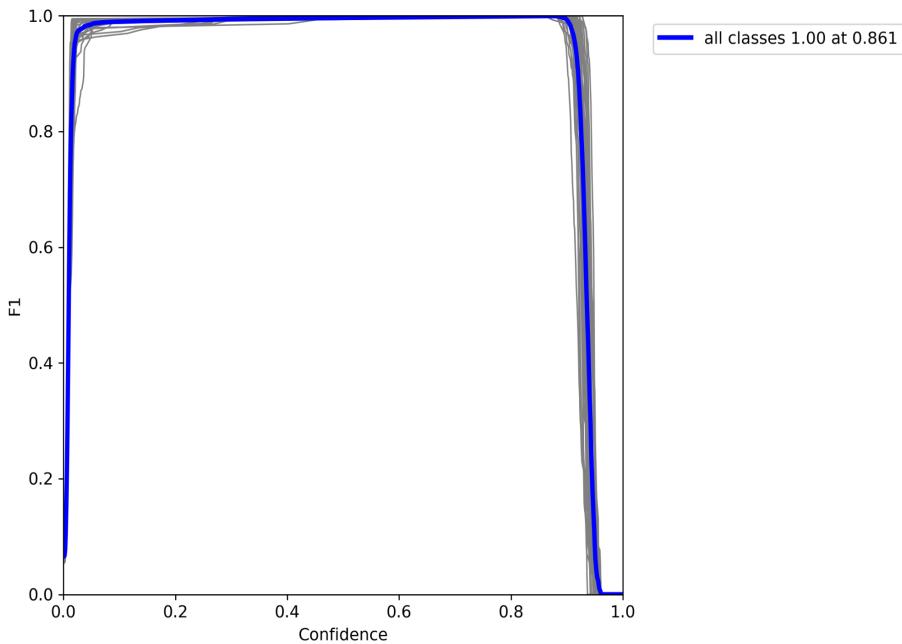
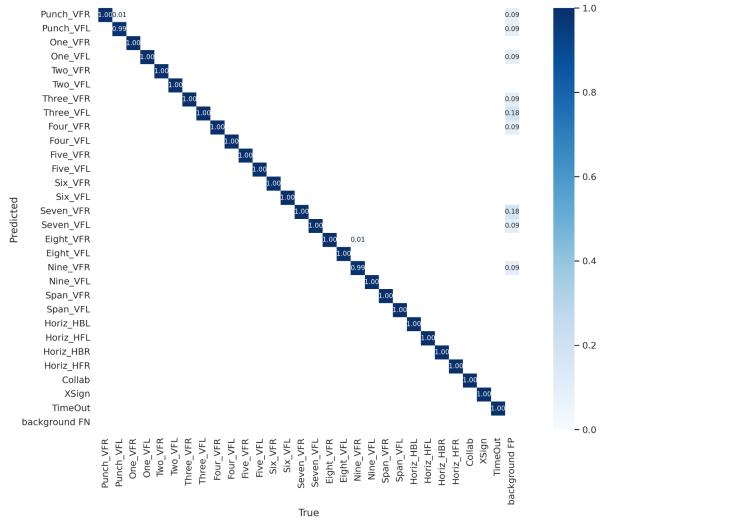
Si prenderanno in analisi alcuni dei risultati ottenuti dai vari training per fare qualche confronto. I confronti fatti evidenzieranno sia la differenza tra le varie versioni di YOLO (S, M, L) sia quella tra il training fatto con preprocessing e senza.

Per i training senza preprocessing si è notata una tendenza del modello ad andare in overfitting dopo le 10 epoch. Questo non è stato possibile valutarlo tramite il Validation Set perché i frame presenti al suo interno sono molto simili alle immagini di training, di conseguenza non si notano anomalie (del tipo che la loss sul training scende mentre quella del validation sale).

Nel codice presente su GitHub [1] è presente uno script chiamato *webcam_detector.py* con cui testare la detection tramite webcam.

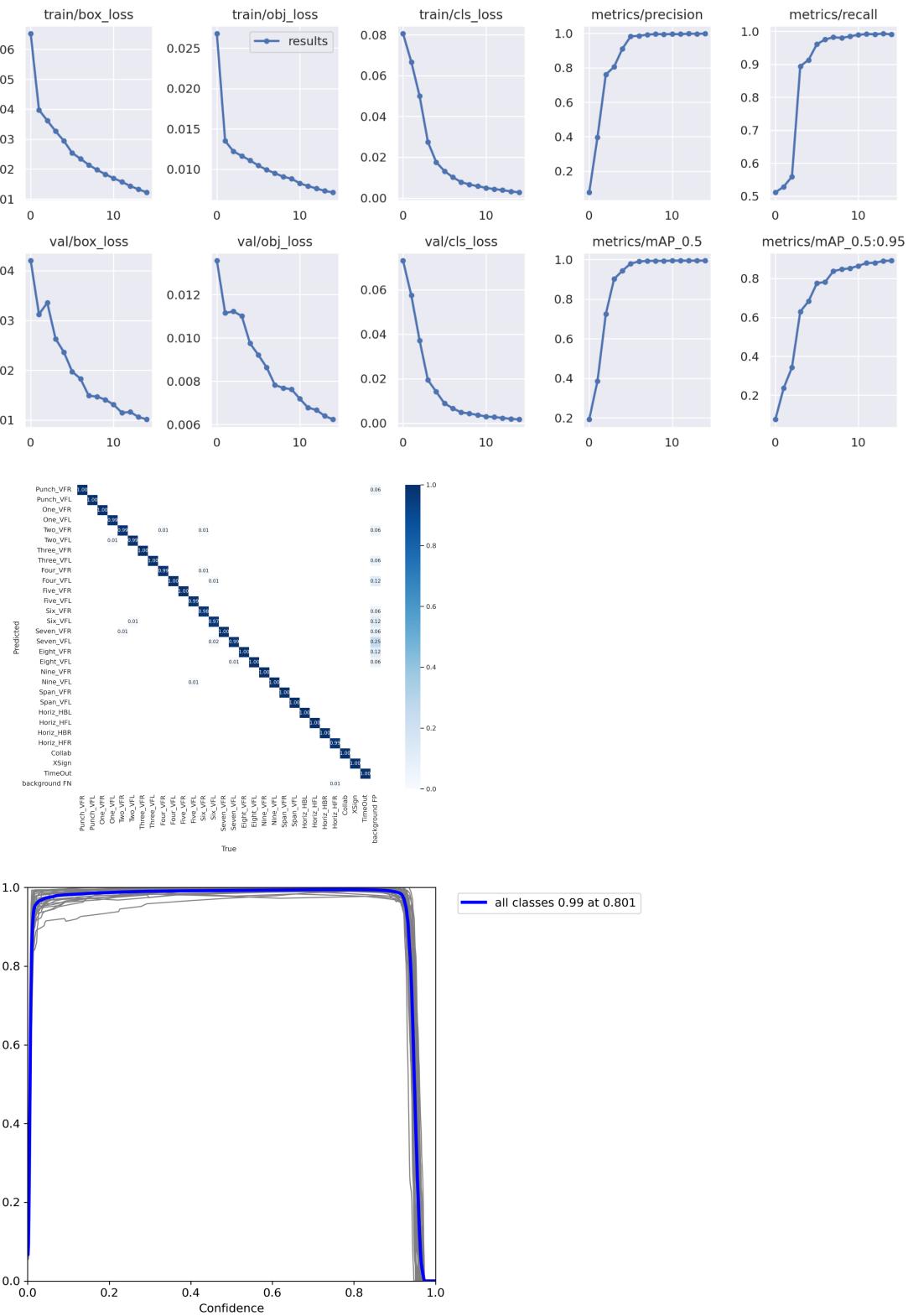
YOLO M (10 epoch)





In questo training si nota la tendenza della rete a raggiungere un valore di loss basse in relativamente poche epocha. Questo fenomeno è da considerarsi positivo in quanto il training non ha bisogno di molto tempo per convergere a una soluzione accettabile, se non fosse per il fatto che sperimentalmente si è notato che andando avanti (nonostante si ottengano valori di loss migliori) il modello tende a overfittare e quindi a dare risultati peggiori durante la detection. La Matrice di Confusione mostra una buona separazione tra le classi e otteniamo un indice F1 intorno al 0.861 di confidenza.

YOLO M + Preprocessing (25 epoches)

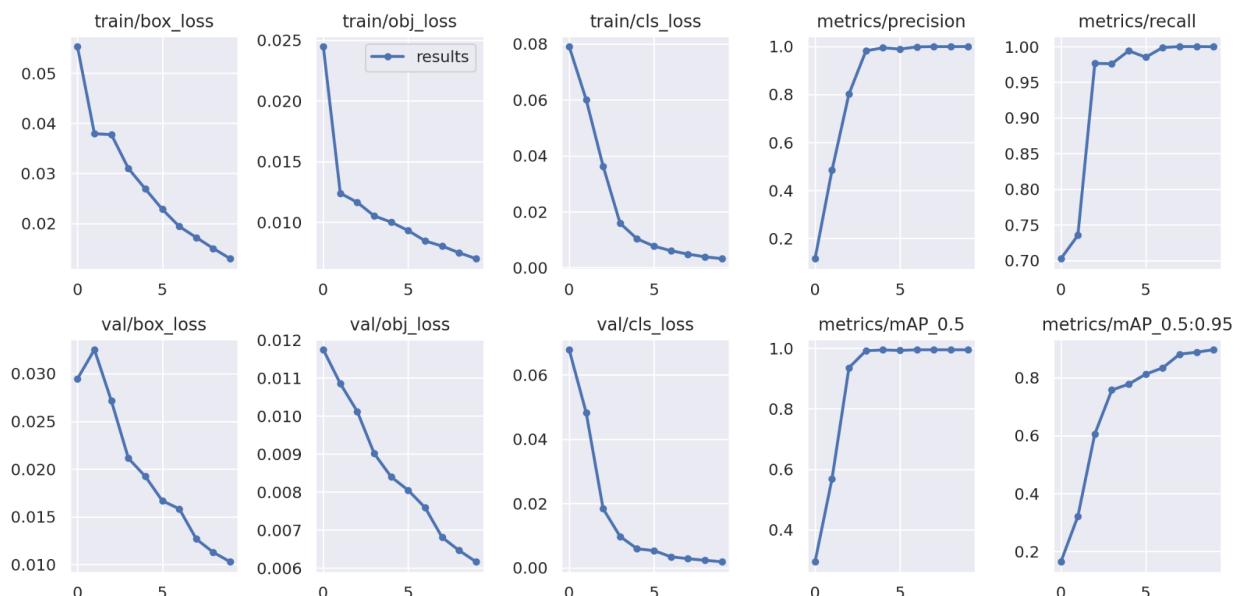


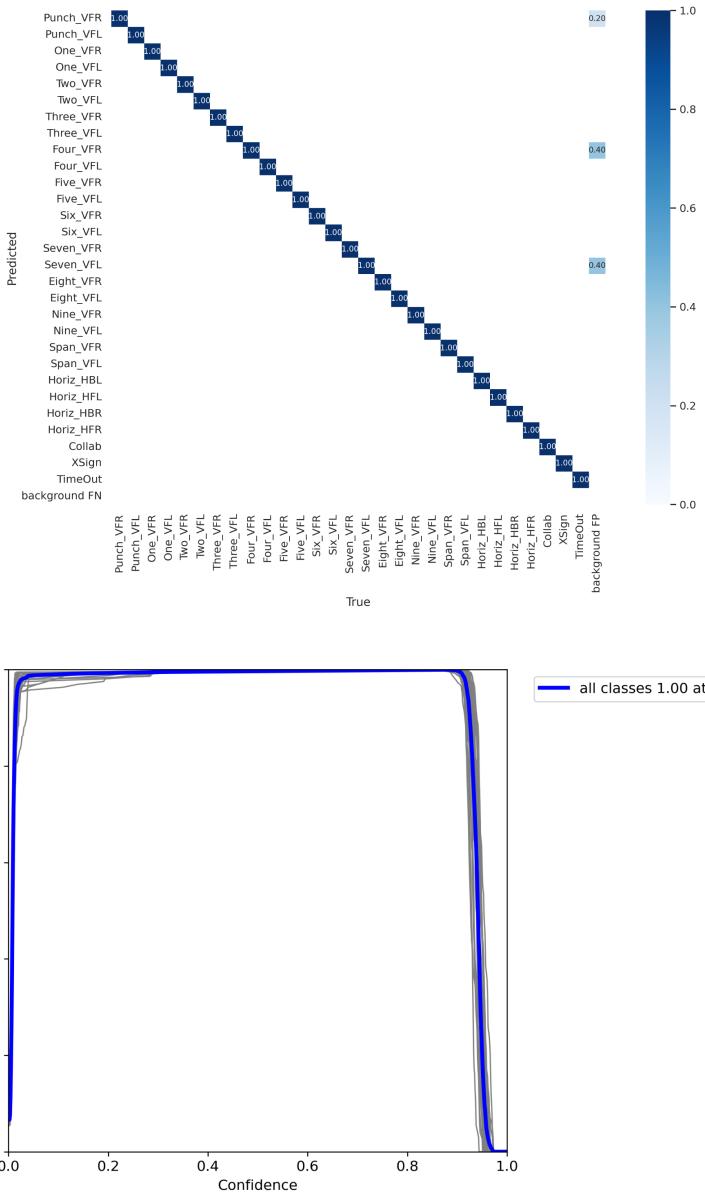
Con il training effettuato quando le immagini vengono preprocessate, il discorso cambia leggermente. Si nota innanzitutto che le loss sul validation set non vadano di pari passo con quelle del training set, un primo segnale che in effetti si hanno immagini diverse tra i due insiemi.

Per questo training si è deciso di andare avanti con 25 epochhe avendo arginato il problema dell'overfitting. Tuttavia è possibile che ce ne vogliano ancora di più per rendere la rete robusta e efficiente, in quanto si nota dalla Matrice di Confusione che ci sono varie classi con ambiguità e soprattutto incertezze sulla classificazione dello sfondo.

Questo si nota molto nel video dimostrativo correlato [11] in cui la detection real-time di questo caso non è molto precisa.

YOLO L (10 epochhe)

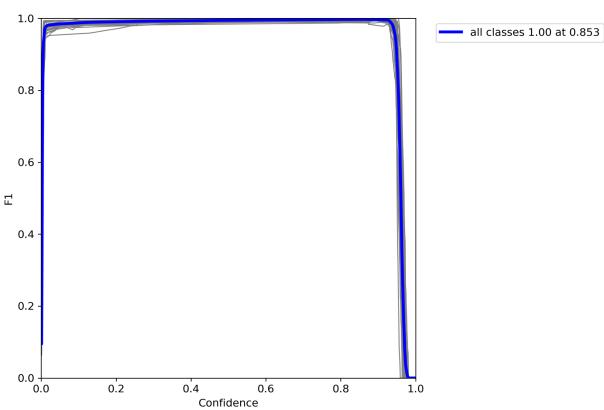
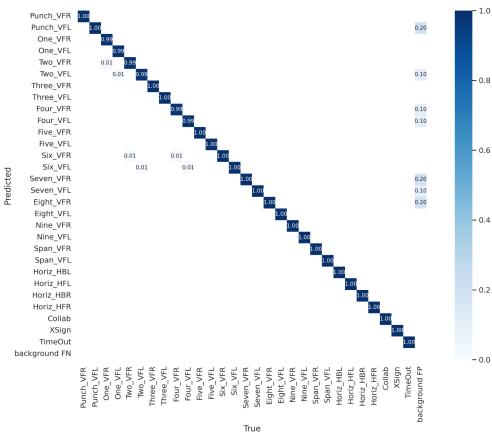
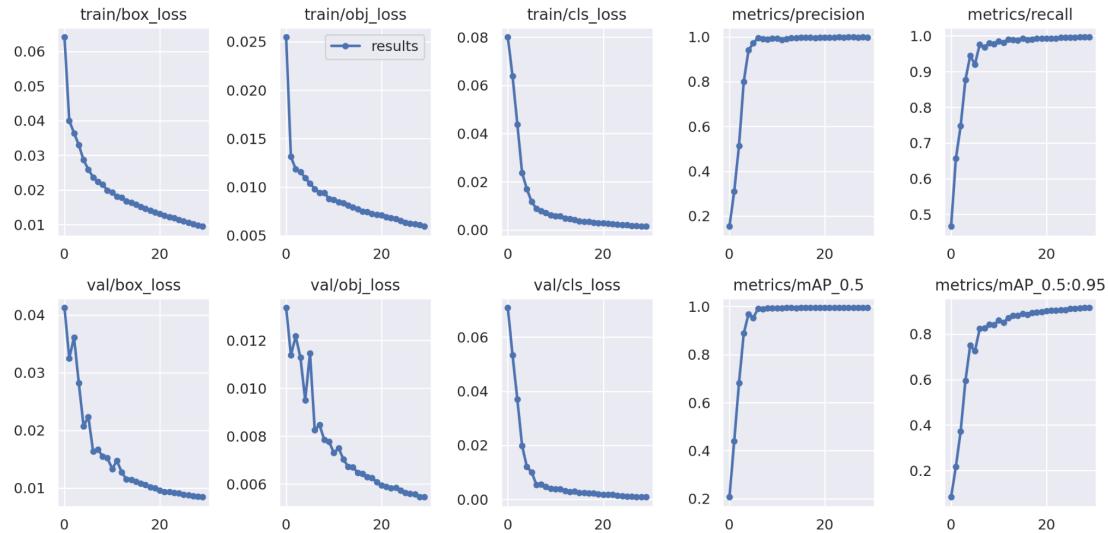




Per questo training non si notano molte differenze rispetto al corrispettivo effettuato sulla versione M, se non che la matrice di confusione evidenzia distinzioni più marcate sulla parte del riconoscimento dello sfondo dal resto delle classi.

Nonostante la rete sembri leggermente meglio della versione M per quest'ultimo dettaglio, è molto più pesante a livello computazionale, e di conseguenza si otterrà uno streaming video con più lag.

YOLO L + Preprocessing (30 epoches)



In quest'ultimo benchmark possiamo notare, come anche nella versione M del training con preprocessing, le oscillazioni sulle loss del validation set e una distinzione meno netta nella Matrice di Confusione.

Ancora una volta è possibile che si debba procedere ad effettuare un training con più epoch per ottenere un risultato migliore.

Tutti i risultati dei benchmark qui sopra offrono sicuramente informazioni quantitative in merito alla detection e alla classificazione, ma per un'analisi qualitativa dei risultati ottenuti sono stati realizzati dei video di confronto per esaminare il risultato dei training anche in maniera visiva.

I video sono disponibili su YouTube [11].

Inoltre tutti i pesi ottenuti dai training, come i grafici, i log e le altre informazioni di training sono disponibili su un file caricato su Drive [10].

Conclusioni

In conclusione, dai test e dai benchmark effettuati si notano comunque delle performance accettabili per il riconoscimento della mano, della gesture e del suo tracking in un video real-time.

Si è notato che la versione M della rete è molto più performante in termini di peso computazionale, mentre la versione L ha dei piccoli dettagli di miglioramento che però non sono apprezzabili in quanto la sua applicazione nell'ambito real-time è lento, almeno nell'ambito di un sistema operativo ad uso domestico, il discorso potrebbe cambiare nel caso in cui si effettuino queste operazioni in sistemi real-time con thread dedicati alla cattura dello stream video e con hardware di computazione (come schede video) capaci di processare più velocemente l'operazione di classificazione tramite la rete YOLO.

Si è visto inoltre che gli algoritmi di preprocessing non hanno migliorato molto la situazione, ma probabilmente ciò è dovuto o a un preprocessing troppo estremo (e quindi difficoltà della rete a trovare un punto di generalizzazione) o a un training con poche epoche. In generale aiutano comunque nella gestione dell'overfitting, ma le performance real-time risultano migliori con la rete addestrata con il dataset puro.

Altre migliorie che potevano essere fatte potevano essere quella di preprocessare anche i frame in input per la detection (magari con dei filtri passa-basso per ridurre il rumore in ingresso) o quella di fare un training di base con il dataset *Egohands* e poi effettuare il training finale con il dataset *HANDS*.

Riferimenti

- [1] Codice Github - <https://github.com/Boo-ray/Handy>
- [2] EgoHands Dataset - <http://vision.soic.indiana.edu/projects/egohands/>
- [3] HANS Dataset - <https://data.mendeley.com/datasets/ndrczc35bt/1>
- [4] Lezione Stanford University - Fei-Fei Li & Justin Johnson & Serena Yeung - http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture11.pdf
- [5] YOLOv5 - <https://github.com/ultralytics/yolov5>
- [6] YOLOv5 Custom Train - <https://github.com/ultralytics/yolov5/wiki/Train-Custom-Data>
- [7] YOLOv4 Paper - <https://arxiv.org/pdf/2004.10934v1.pdf>
- [8] Download pesi -
<https://drive.google.com/file/d/1zDABYj889kYYUN2gqmcZ95gfHlrcM0w8/view?usp=sharing>
- [9] Video dimostrativo - https://www.youtube.com/watch?v=xL_LVF_cwWA
- [10] Nesting Algorithms - <https://isr.umd.edu/Labs/CIM/projects/nesting/sheetmetal.pdf>