UNIVERSITÀ POLITECNICA DELLE MARCHE

FACOLTÀ DI INGEGNERIA

Corso di Laurea Triennale in
Ingegneria Informatica e dell'Automazione

Progettazione e sviluppo di un software per il riconoscimento e il conteggio di olive tramite immagini

Design and implementation of a software to detect and count olives from images

Relatore:
Dott. Mancini Adriano

Laureando:
D'Angelo Valerio

Anno Accademico 2019-2020

Indice

Capitolo 1

Introduzione

La seguente tesi sperimentale nel campo dell'intelligenza artificiale, svolta in stretta collaborazione con il collega Giulio Fischietti, consiste nell'implementazione di un software, basato su una rete neurale convoluzionale, addetto al riconoscimento e conteggio di olive tramite immagini RGB.

Il dataset iniziale utilizzato per il training della rete neurale era composto da immagini prese da internet, ma successivamente è stato arricchito sempre più in modo da aumentare la generalizzazione della performance della rete.

Le **label** delle immagini sono state create tramite la piattaforma ***Labelbox***, mentre i vari **test** e **training** della rete neurale sono stati eseguiti in linguaggio ***Python*** su ***Google Colab***, in modo da arrivare al risultato finale in maniera più efficiente sfruttando schede grafiche più performanti fornite in remoto da Google.

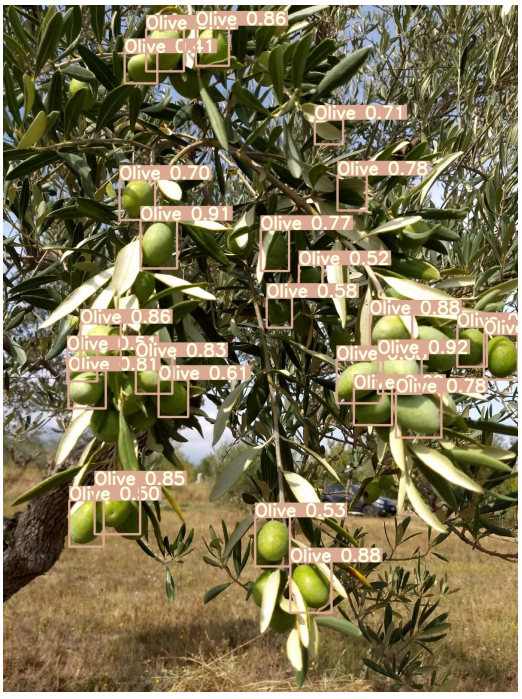

Figura 1.1 – Esempio di olive individuate dalla rete neurale

1.1 Agricoltura 4.0

Per **agricoltura 4.0** si intende l'evoluzione del modello agricolo attuale tramite interventi di precisione attuabili grazie alle nuove tecnologie, con l'intento di migliorare resa, qualità e sostenibilità delle coltivazioni.

Infatti, a seguito di cambiamenti climatici che hanno deteriorato le coltivazioni, e ad un aumento del consumo di cibo a causa dell'incremento demografico, si è rivelata necessaria una rivoluzione in campo agricolo tramite nuovi metodi di coltivazione, tra cui lo ***urban farming***, l'**idrocoltura** e l'utilizzo di sistemi basati sui **big data**.

Nello specifico, in quest'ultimo caso si fa uso di tecnologie che permettono di monitorare la coltivazione, raccogliendo e analizzando dati che si riveleranno fondamentali nella sua gestione, in modo da aumentarne produttività ed efficienza.

In **Italia** l'agricoltura 4.0 costituisce già un mercato che vale oltre 450 milioni di euro, annoverando tra i settori che utilizzano maggiormente i nuovi metodi di coltivazione quello vitivinicolo, lattiero-caseario e cerealicolo.

È stato infatti riscontrato che il 66% delle imprese italiane adotta software gestionali, il 40% sistemi di mappatura di coltivazioni e terreni, e il 39% sistemi di monitoraggio e controllo delle macchine agricole. Una piccola percentuale, infine, fa ricorso a droni o a robot per le attività agricole.

Figura 1.2 – Schema generale dell'agricoltura 4.0 (immagine tratta da `https://www.martignani.com/it/sistema-agricoltura-40`)

1.2 Il progetto

Sulla base di quanto detto, l'intento del progetto è stato quello di implementare un software sempre più addestrato al riconoscimento di olive e al loro conteggio.

Ciò è stato possibile grazie all' ***augmentation*** (e di conseguenza, diversificazione) del ***dataset***, ottenuto aggiungendo più copie "alterate" delle stesse immagini: per ogni immagine del ***dataset*** è stata infatti creata manualmente una versione ***blurred***, due versioni ***flipped*** (sia orizzontalmente che verticalmente), e una versione **ruotata**. È stata infine aggiunta una collezione di immagini prese dal vivo, in collaborazione con il nostro collega Giacomo Pierigè, che ha ulteriormente ingrandito il ***dataset*** a nostra disposizione permettendo di generalizzare la performance finale della rete neurale.

1.3 Struttura della tesi

Il presente lavoro di tesi è strutturato come segue:

- in questo primo capitolo è stata data una visione generale del problema da affrontare, e dei passi necessari per risolverlo.
- nel secondo capitolo verranno elencati i principali strumenti utilizzati per l'implementazione del software finale, con particolare enfasi sulla **rete neurale** utilizzata, e delle tematiche generali riguardanti il progetto in sé.
- nei capitoli 3, 4 e 5 sono descritti i vari passaggi per arrivare al risultato finale. Si è partiti descrivendo la creazione del dataset iniziale mediante gli strumenti a disposizione, per poi passare allo svolgimento dei vari test con conseguente commento dei risultati ottenuti. Sono stati infatti analizzati i vari parametri della rete neurale (*confusion matrix*, *loss function* etc etc), e sulla base di questi ultimi sono state effettuate modifiche al dataset con conseguente retrain della rete fino al raggiungimento di risultati ritenuti soddisfacibili.
- nel sesto ed ultimo capitolo sono stati brevemente discussi alcuni campi di utilizzo del software implementato, facendo riferimento ai problemi tutt'ora rimasti insoluti nel campo dell'agricoltura di precisione.

Capitolo 2

Strumenti e metodi utilizzati

In questo capitolo verrà prima fatta un'introduzione sul ***machine learning*** e sulle **reti neurali**, ambiti di studio del progetto, per poi concentrarsi sui particolari strumenti che ne hanno reso possibile la realizzazione. Particolare enfasi è stata data all'algoritmo della rete neurale utilizzato per il riconoscimento delle immagini, **Yolo v3**.

2.1 Deep learning e reti neurali

L'**apprendimento profondo** o ***deep learning*** è la branca del *machine learning* nella quale il modello di apprendimento segue una struttura a strati, con i concetti di alto livello definiti sulla base di quelli di basso livello.

Tutti questi livelli formano una gerarchia di concetti da apprendere: ogni strato calcola i valori per quello successivo affinché l'informazione venga elaborata in maniera sempre più completa.

Ad oggi le tecniche di ***deep learning*** permettono di:

- identificare oggetti nelle immagini e nei video;
- trascrivere il parlato in testo;
- individuare e interpretare gli interessi degli utenti online, mostrando i risultati più pertinenti per le loro ricerche.

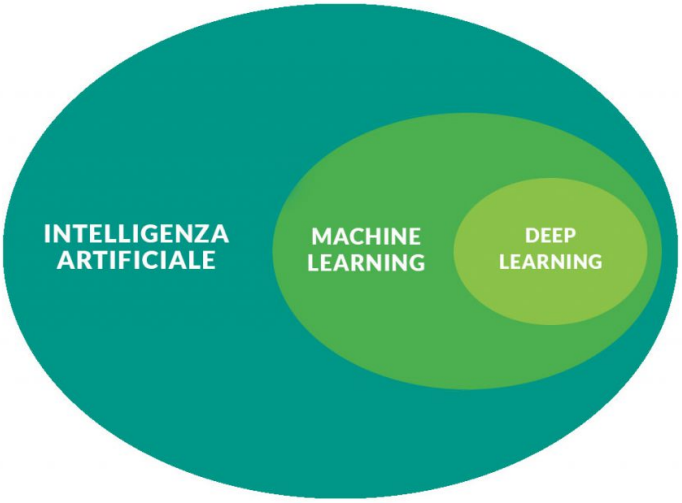

Figura 2.1 – Struttura dell'intelligenza artificiale (immagine tratta da `https://italiancoders.it/deep-learning-svelato-ecco-come-funzionano-le-reti-neurali-artificiali/`)

2.2 Reti neurali artificali

Le architetture di base del *deep learning* sono **reti neurali artificiali (*ANNs*)** organizzate in diversi strati, dove ogni strato calcola i valori per quello successivo per elaborare l'informazione in maniera sempre più completa.

A livello di apprendimento, il prototipo delle *ANNs* sono le **reti neurali biologiche**. Infatti, nonostante le reti artificiali non siano analoghi perfetti della loro controparte biologica (il loro processo di apprendimento è semplificato), ne presentano tuttavia diverse caratteristiche in comune:

- ogni **neurone** può ricevere simultaneamente segnali da più **sinapsi**; quindi misurerà il potenziale elettrico di tali segnali, stabilendo se è stata raggiunta la soglia di attivazione per generare a sua volta un impulso nervoso.

- la **configurazione sinaptica** è dinamica: il numero di sinapsi può incrementare o diminuire a seconda degli stimoli che riceve la rete. Più sono gli stimoli, maggiori sono le connessioni sinaptiche create. In questo modo, la rete neurale ha una buona capacità di adattamento.

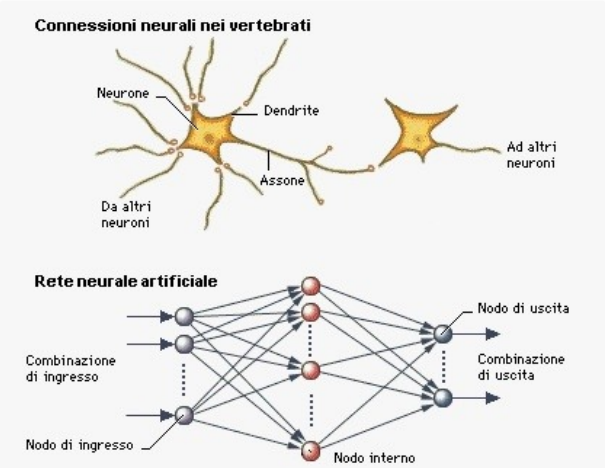

Figura 2.2 – Confronto tra una rete neurale biologica e una artificiale (immagine tratta da `https://it.quora.com/Che-cosa-sono-le-reti-neurali`)

2.2.1 Principio di funzionamento

Nelle *ANNs* i nodi ricevono dati in **input** e, dopo averli processati, sono in grado di inviare l'**output** ad altri neuroni. Attraverso questi **cicli di input-elaborazione-output**, con input che presentano sempre variabili differenti, i nodi diventano in grado di generalizzare e quindi di fornire output corretti associati ad input non facenti parte del training set.

Gli algoritmi di apprendimento utilizzati per istruire le reti neurali sono divisi in 3 categorie. La scelta su quale sia opportuno utilizzare dipende dal campo di applicazione per il quale la rete è stata progettata e dalla sua tipologia (***feedback*** o ***feedforward***):

- l'**apprendimento supervisionato**, che mira a istruire un sistema informatico in modo da consentirgli di elaborare automaticamente previsioni sui valori di uscita di un sistema rispetto ad un input, sulla base di una serie di esempi ideali (coppie di input-output) che gli vengono inizialmente forniti.
- l'**apprendimento non supervisionato**, che consiste nel fornire al sistema informatico una serie di input che esso stesso riclassificherà ed organizzerà sulla base di caratteristiche comuni, per cercare di effettuare ragionamenti e previsioni sugli input successivi.
- l'**apprendimento di rinforzo**, che punta a realizzare agenti autonomi in grado di scegliere azioni da compiere per il conseguimento di determinati obiettivi tramite interazione con l'ambiente in cui sono immersi.

Le tecniche di questo progetto fanno riferimento a quelle dell'apprendimento supervisionato, dove si fornisce alla rete un insieme di input ai quali corrispon-

dono output noti (**training set**). Analizzandoli, la rete apprende il nesso che li unisce, e quindi riesce a generalizzare.

2.2.2 Pesi ed Error Back-Propagation

Man mano che la macchina elabora output, si procede a correggerla per migliorarne le risposta. Questa procedura consiste nel variare i **pesi sinaptici**, ovvero fattori moltiplicativi che "pesano" (a livello di importanza) le connessioni tra un neurone ed un altro. L'obiettivo è aumentare i pesi che determinano gli output corretti e diminuire quelli che generano valori ritenuti non validi.

Daro che variare manualmente i pesi è improponibile a livello operativo, il meccanismo impiegato per migliorare la precisione dell'output è l'***Error Back-Propagation***, che consiste nel variare i pesi associati ad ogni output "all'indietro", ovvero partendo dall'ultimo strato fino ad arrivare al primo, in modo da ridurre sempre di più l'errore totale.

Un altro fattore importante nella correzione della macchina è l'**esperienza del programmatore**, che deve trovare un rapporto adeguato fra le dimensioni del training set, quelle della rete e l'abilità a generalizzare che desidera ottenere.

Infatti, un numero eccessivo di parametri in ingresso e una troppo potente capacità di elaborazione rendono difficile il compito di generalizzazione alla rete neurale, poiché gli input esterni al training set vengono valutati dalla rete come troppo dissimili dai modelli che conosce. Invece, un training set con poche variabili fa sì che la rete non abbia sufficienti parametri per apprendere la generalizzazione. Di conseguenza, è necessario un giusto compromesso, cosa che necessita di molta preparazione ed esperienza.

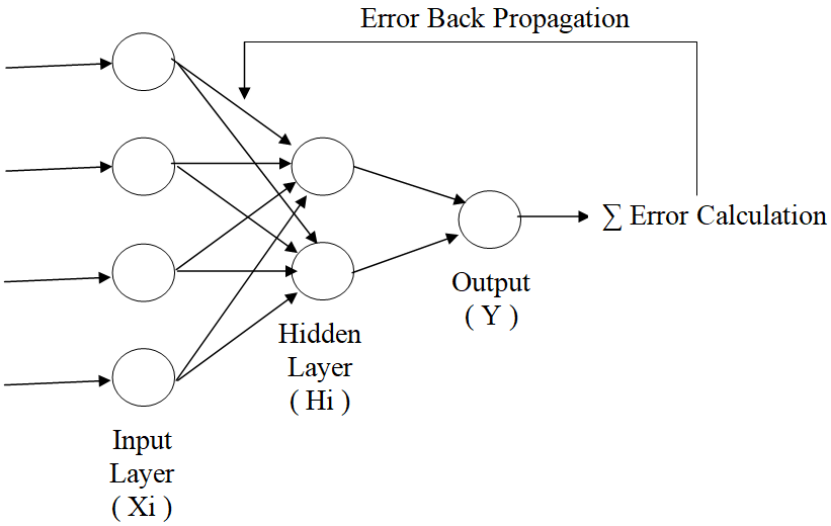

Figura 2.3 – Schema della Error Back-Propagation (immagine tratta da `https://www.researchgate.net/figure/Sample-Error-Back-Propagation_fig4_322332639`)

2.3 Yolo v3

Yolo è un algoritmo che implementa una **rete neurale convoluzionale**, un particolare tipo di rete neurale basata sui processi biologici della corteccia animale (e per questo molto efficiente nel riconoscimento di immagini e video).

La particolare versione di Yolo utilizzata nel progetto è la terza (più efficiente e generalizzata delle precedenti), ottenuta dal **repository github** di ***ultralytics***.

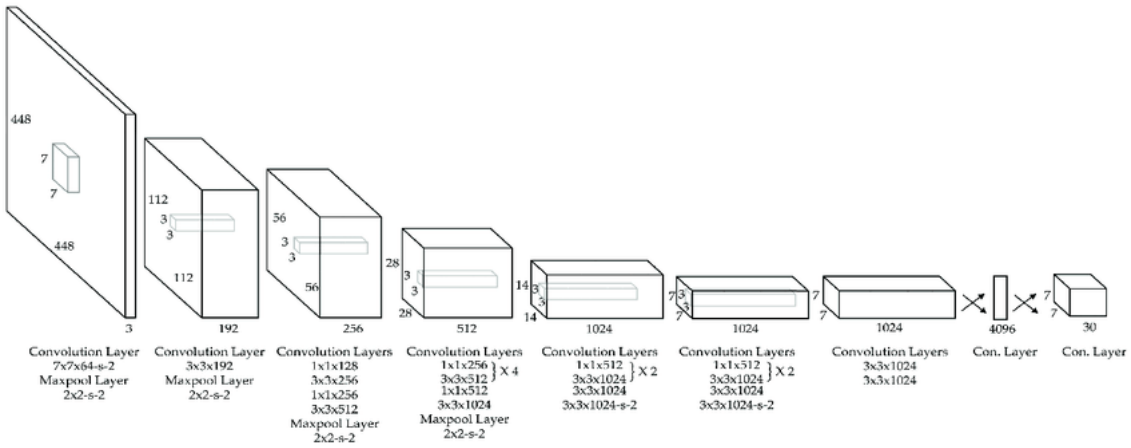

Figura 2.4 – Architettura neurale di Yolo (immagine tratta da `https://medium.com/@jonathan_hui/real-time-object-detection-with-yolo-yolov2-28b1b93e2088`)

2.3.1 Struttura di Yolo

Per il riconoscimento di oggetti, Yolo divide l'immagine in una **griglia nxn**: ogni cella della griglia cerca di predire un singolo oggetto con centro appartenente alla cella stessa mediante un numero finito di ***bounding box*** (o "bbox") .

Ogni **bbox** ha 5 componenti: 4 costituiscono le coordinate della bbox, mentre la quinta rappresenta il l'**indice di confidenza**, un parametro che indica il valore della probabilità che ci sia un oggetto in quella bbox, e la precisione di quest'ultima. Ogni bbox, inoltre, ha un certo numero di **probabilità condizionali di classe** (una per classe), valori che indicano quanto è probabile che l'oggetto in questione appartenga a quella specifica classe.

Dopo aver determinato tutte i bounding box, Yolo scarta quelle con indice di confidenza più basso (minore di 0.25), per poi selezionare la bbox "migliore" (quella con il più alto ***Intersection over Union*** con la label originaria), che sarà il riferimento per il calcolo di un parametro fondamentale per l'efficienza della rete: la "funzione di loss", o ***loss function***.

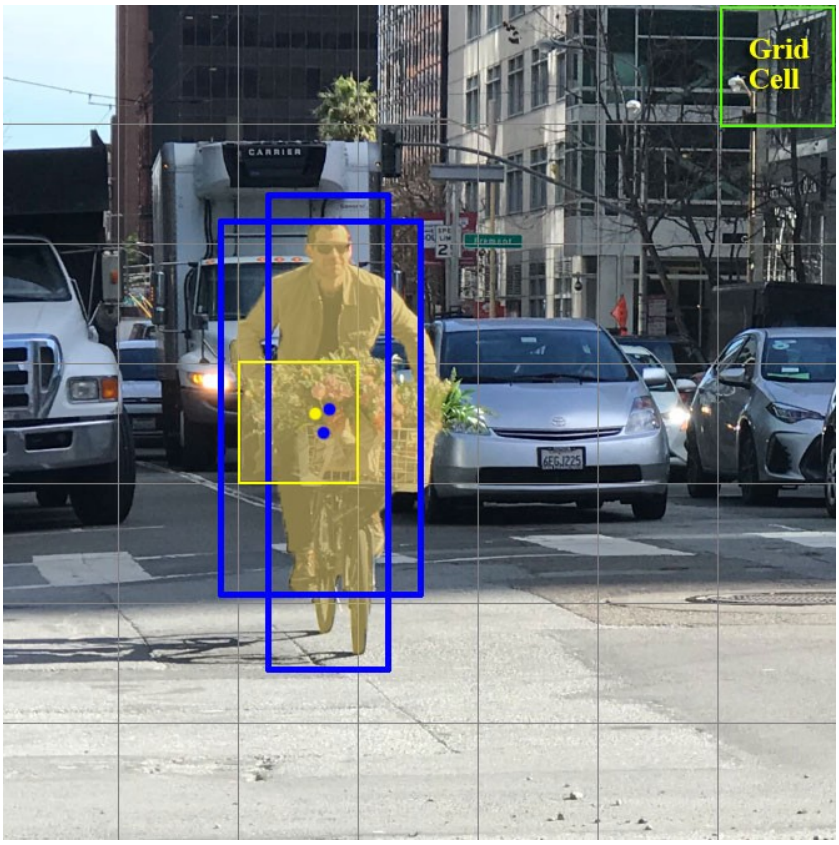

Figura 2.5 – Sistema di griglia nxn di Yolo (immagine tratta da `https://medium.com/@jonathan_hui/real-time-object-detection-with-yolo-yolov2-28b1b93e2088`)

2.3.2 Parametri di Yolo

Yolo ha un'ampia serie di **parametri** per valutare la precisione e l'efficienza delle proprie predizioni.

I principali sono:

- **Loss Function**

  La ***loss function*** è una funzione di costo che rappresenta la somma degli errori commessi nella predizione dell'oggetto. Di conseguenza è un parametro che si punta a minimizzare al fine di ottimizzare la predizione. Questa funzione, calcolata tramite l'errore quadratico medio sulla bbox "migliore", è composta a sua volta da 3 parametri, ognuno rappresentante un particolare tipo di errore nella predizione: ***classification loss***, ***localization loss***, ***confidence loss***.

$$
\begin{aligned}
&\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
&+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right] \\
&+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i \right)^2 \\
&+ \lambda_{\text{noobj}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{noobj}} \left(C_i - \hat{C}_i \right)^2 \\
&+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2
\end{aligned}
$$

Figura 2.6 – Formula per il calcolo della loss function (immagine tratta da `https://medium.com/@jonathan_hui/real-time-object-detection-with-yolo-yolov2-28b1b93e2088`)

Di seguito sono illustrati i parametri che compongono la *loss function*:

1. ***Classification loss***: è l'errore quadratico medio della probabilità condizionale di classe, calcolata per ogni classe.

$$\sum_{i=0}^{S^2} \mathbb{1}_{i}^{\text{obj}} \sum_{c \in \text{classes}} (p_i(c) - \hat{p}_i(c))^2$$

where

$\mathbb{1}_{i}^{obj} = 1$ if an object appears in cell i, otherwise 0.

$\hat{p}_i(c)$ denotes the conditional class probability for class c in cell i.

Figura 2.7 – Formula per il calcolo della classification loss (immagine tratta da `https://medium.com/@jonathan_hui/real-time-object-detection-with-yolo-yolov2-28b1b93e2088`)

2. ***Localization loss***: è l'errore sulla posizione e sulla misura della bbox.

$$
\begin{aligned}
&\lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2 \right] \\
&+ \lambda_{\text{coord}} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left[\left(\sqrt{w_i} - \sqrt{\hat{w}_i} \right)^2 + \left(\sqrt{h_i} - \sqrt{\hat{h}_i} \right)^2 \right]
\end{aligned}
$$

where

$\mathbb{1}_{ij}^{obj} = 1$ if the j th boundary box in cell i is responsible for detecting the object, otherwise 0.

λ_{coord} increase the weight for the loss in the boundary box coordinates.

Figura 2.8 – Formula per il calcolo della localization loss (immagine tratta da `https://medium.com/@jonathan_hui/real-time-object-detection-with-yolo-yolov2-28b1b93e2088`)

3. ***Confidence loss***: è l'errore sulla probabilità che la bbox contenga un oggetto.

$$\sum_{i=0}^{S^2}\sum_{j=0}^{B} \mathbb{1}_{ij}^{\text{obj}} \left(C_i - \hat{C}_i\right)^2$$

where

$\hat{C}_i$ is the box confidence score of the box j in cell i.

$\mathbb{1}_{ij}^{obj} = 1$ if the j th boundary box in cell i is responsible for detecting the object, otherwise 0.

Figura 2.9 – Formula per il calcolo della confidence loss (immagine tratta da `https://medium.com/@jonathan_hui/real-time-object-detection-with-yolo-yolov2-28b1b93e2088`)

- **Intersection Over Union**

  L'***Intersection Over Union***, o ***IoU***, è un parametro che valuta il grado di sovrapposizione di due *bounding box*. Nel caso delle reti neurali le due bbox in questione sono quella originale e quella determinata dalla rete neurale.

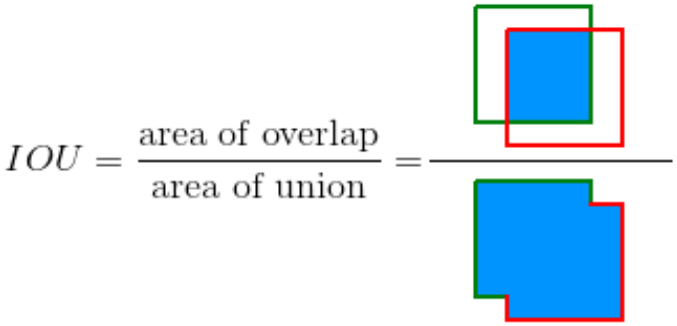

Figura 2.10 – Formula per il calcolo dell'IOU (immagine tratta da `https://supervise.ly/explore/plugins/confusion-matrix-75279/overview`)

Il valore del grado di sovrapposizione (un numero compreso tra 0 e 1) viene confrontato con un valore di soglia e, a seconda della differenza tra i due, si può stimare la correttezza della predizione.

Infatti, si possono individuare 4 casi ben distinti:

- i **veri positivi**, ovvero le individuazioni corrette di oggetti nell'immagine. In questo caso l'IoU è maggiore del valore di soglia.
- i **falsi positivi**, ovvero casi in cui la rete ha individuato un oggetto, ma in realtà esso non è presente nell'immagine, o è molto distante dalla bbox individuata dalla rete. In questo caso l'IoU è minore del valore di soglia.
- i **veri negativi**, ovvero le "non individuazioni" corrette di oggetti nell'immagine. In questo caso l'IoU non si applica.
- i **falsi negativi**, ovvero casi in cui la rete ha mancato di individuare oggetti nell'immagine. Anche qui l'IoU non si applica.

Tutti questi valori verranno poi presentati nei risultati dei test in una struttura chiamata ***confusion matrix***, o "matrice di confusione".

- **Precision, Recall, F1**

  La ***Precision***, la ***Recall*** e l'***F1*** sono 3 metriche che determinano l'efficienza della predizione, ma che si basano, a differenza dei precedenti parametri, sul calcolo dei veri e falsi positivi, e dei veri e falsi negativi.

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

$$Recall = \frac{TP}{TP + FN}$$

$$F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall}$$

TP = True positive

TN = True negative

FP = False positive

FN = False negative

Figura 2.11 – Formule di Precision, Recall ed F1 (immagine tratta da `https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173`)

- **MAP (Mean Average Precision)**

  Il ***MAP***, è l'*AP* (*Average Precsion*, precisione media) calcolata per tutte le classi, ovvero la media dell'*AP* per ogni classe. L'***AP*** è l'area sottesa alla ***curva precision-recall***.

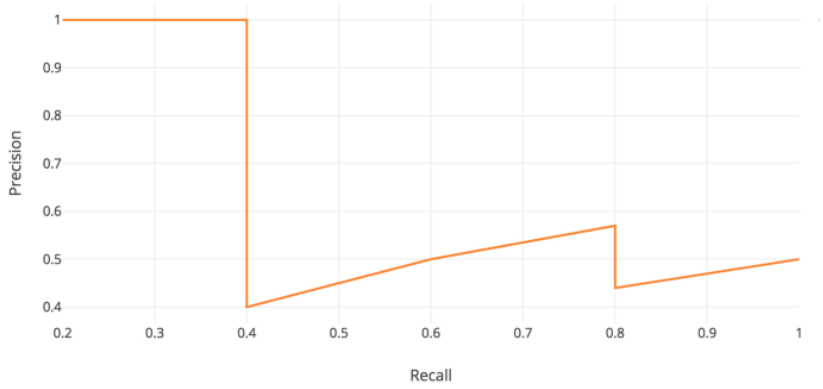

Figura 2.12 – Esempio di curva precision-recall (immagine tratta da `https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173`)

$$\mathrm{AP} = \int_0^1 p(r)dr$$

Figura 2.13 – Formula del calcolo dell'Average Precision (immagine tratta da `https://medium.com/@jonathan_hui/map-mean-average-precision-for-object-detection-45c121a31173`)

2.3.3 Iperparametri di Yolo

Come tutte le reti neurali, Yolo ha i suoi **iperparametri**, ovvero parametri di sistema che governano l'algoritmo di apprendimento. Sono parametri che determinano l'efficienza del training della rete neurale, e una loro buona scelta ha un grande impatto sulle performance dell'algoritmo stesso.

Gli iperparametri di Yolo si dividono principalmente in due categorie: quelli riguardanti la *loss function*, e quelli riguardanti il *data augumentation* (*blurring*,*flipping*, rotazione, colore).

Per quanto riguarda questi ultimi, si è preferito impostarli tutti a zero, in quanto le immagini che compongono il dataset sono state modificate con script progettati ad hoc (tramite ***OpenCV***), poiché con il *tuning* automatico di Yolo sono stati ottenuti risultati non soddisfacenti.

Per quanto riguarda i primi, invece, è stato utilizzato il comando ***evolve.py***, fornito da Yolo, per fare il *tuning* automatico. Questo comando consiste in un **algoritmo evolutivo**, che parte dall'insieme iniziale di iperparametri, per poi modificarli man mano fino a raggiungere dei valori ottimali.

```
# Hyperparameters
hyp = {'giou': 3.54,  # giou loss gain
       'cls': 37.4,  # cls loss gain
       'cls_pw': 1.0,  # cls BCELoss positive_weight
       'obj': 64.3,  # obj loss gain (*=img_size/320 if img_size != 320)
       'obj_pw': 1.0,  # obj BCELoss positive_weight
       'iou_t': 0.20,  # iou training threshold
       'lr0': 0.01,  # initial learning rate (SGD=5E-3, Adam=5E-4)
       'lrf': 0.0005,  # final learning rate (with cos scheduler)
       'momentum': 0.937,  # SGD momentum
       'weight_decay': 0.0005,  # optimizer weight decay
       'fl_gamma': 0.0,  # focal loss gamma (efficientDet default is gamma=1.5)
       'hsv_h': 0.0138,  # image HSV-Hue augmentation (fraction)
       'hsv_s': 0.678,  # image HSV-Saturation augmentation (fraction)
       'hsv_v': 0.36,  # image HSV-Value augmentation (fraction)
       'degrees': 1.98 * 0,  # image rotation (+/- deg)
       'translate': 0.05 * 0,  # image translation (+/- fraction)
       'scale': 0.05 * 0,  # image scale (+/- gain)
       'shear': 0.641 * 0}  # image shear (+/- deg)
```

Figura 2.14 – Lista degli Iperparametri di Yolov3

2.3.4 Label di Yolo

Ogni **label** in formato YOLOv3 è composta da 4 valori, ovvero le 4 coordinate delle bbox, normalizzate rispetto alle dimensioni dell'immagine. Dato che *Labelbox* fornisce le coordinate in base al suo sistema cartesiano, diverso da quello di Yolo, è stato necessario creare un parser, ovvero un programma per ottenere le coordinate corrette e normalizzate.

2.4 Labelbox

Labelbox è un tool di ***data labeling***, che è servito per creare le bbox sulle immagini del dataset, per poi passare tutto a Yolo. Su Labelbox si possono "labellizare" con precisione le immagini in formato png, e poi esportare il tutto in formato json.

Il sistema di "labellizazione" di *Labelbox* è il seguente : i due valori assegnati ad ogni label, una x e una y, seguono un sistema di assi cartesiani che parte da in alto a sinistra; di conseguenza con il parsing si è dovuto poi normalizzare le coordinate per ottenerle nel formato Yolo.

Figura 2.15 – Logo di Labelbox (immagine tratta da `https://roboflow.com/formats/labelbox-json`)

2.5 Python

Python è il linguaggio di programmazione di cui ci si è serviti per creare tutti gli *script* necessari alle modifiche delle immagini e all'esecuzione dei test.

E' un linguaggio di alto livello orientato agli oggetti particolarmente utile per sviluppare applicazioni distribuite, *scripting*, computazione numerica e *system testing*.

Figura 2.16 – Logo di Python (immagine tratta da `https://www.python.org/`)

2.6 Google Colab

Per fare i test ed i train della rete neurale ci si è serviti di ***Google Colab***, una piattaforma per eseguire codice su cloud, con linguaggio utilizzato il *Python*.

Il vantaggio di un tale strumento risiede nel fatto che, eseguendo il codice online, non vi è sfruttamento della potenza di calcolo della propria GPU, poichè quest'ultima viene fornita da Google, tramite remoto. Di conseguenza si possono svolgere agevolmente anche calcoli molto complessi.

Figura 2.17 – Logo di Google Colab (immagine tratta da `https://www.miriade.it/google-colab-il-tool-gratuito-di-google-ha-servizio-dei-data-scientist/`)

2.7 GitHub

E' doveroso infine citare **GitHub** tra gli strumenti utilizzati: questo sito di ***software-hosting***, utilizzato principalmente da sviluppatori, permette l'upload di software in remoto su *repository* appositamente create. Gli utenti sono poi liberi di scaricare il suddetto software per utilizzarlo per i loro scopi. Come detto, in questo progetto è stata scaricata la versione 3 di Yolo dalla repository github **https://github.com/ultralytics/yolov3**.

Figura 2.18 – Logo di Github (immagine tratta da `https://github.com/logos`)

Capitolo 3

Labeling

Come detto nella sezione 2.4, per creare il primo dataset da testare su **Yolo v3** è stato necessario servirsi della piattaforma ***LabelBox***.

Da internet sono state selezionate circa **70 immagini** ritenute valide (sono state evitate immagini troppo sfocate, o con troppe olive), che sono poi state successivamente caricate online sulla piattaforma. *Labelbox* ha consentito di apporre le label manualmente ed esportare localmente i risultati in formato **json**.

L'ultimo step è stato creare manualmente un **algoritmo di *parsing***, che ha avuto una duplice funzione:

- l'estrazione delle coordinate in formato Yolo v3, ovvero nel **formato txt**.
- la **normalizzazione** delle stesse, in quanto, come spiegato precedentemente, il sistema di riferimento cartesiano utilizzato da Labelbox è diverso da quello utilizzato da Yolo.

3.1 Labeling

Il processo di labeling si è rivelato semplice a livello di realizzazione, ma molto dispendioso in termini di tempo.

Una volta creato un nuovo progetto nella sezione ***New Project*** e definita la classe ***Olive*** all'interno del tool di labeling, è stato possibile procedere alla vera e propria fase di "labellizazione".

Labelbox offre la possibilità di creare le bbox in forma poligonale, a mano libera o rettangolare. La scelta è ricaduta sulla forma rettangolare in quanto offre sia un vantaggio in termini di tempo rispetto a quelle a mano libera, sia un vantaggio in termini di semplicità e precisione rispetto a quelle poligonali.

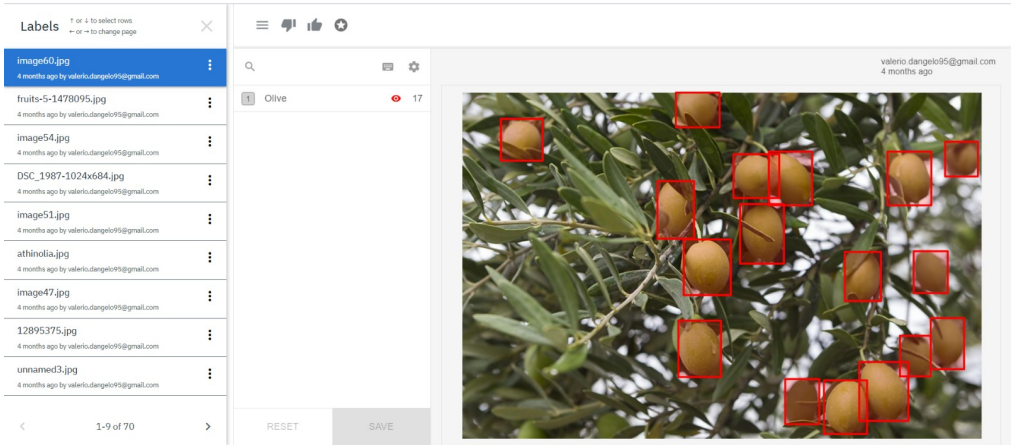

Figura 3.1 – Processo di labelizzazione del dataset su Labelbox

Al termine di questa fase sono state totalizzate, tra le 70 immagini, ben **471** ***bounding box***, con l'accortezza di non aver saltato nemmeno una singola oliva in modo da garantire il massimo dell'affidabilità nella lettura dei dati, evitando così dei "falsi" falsi positivi.

I risultati del processo sono presentati nella seguente immagine.

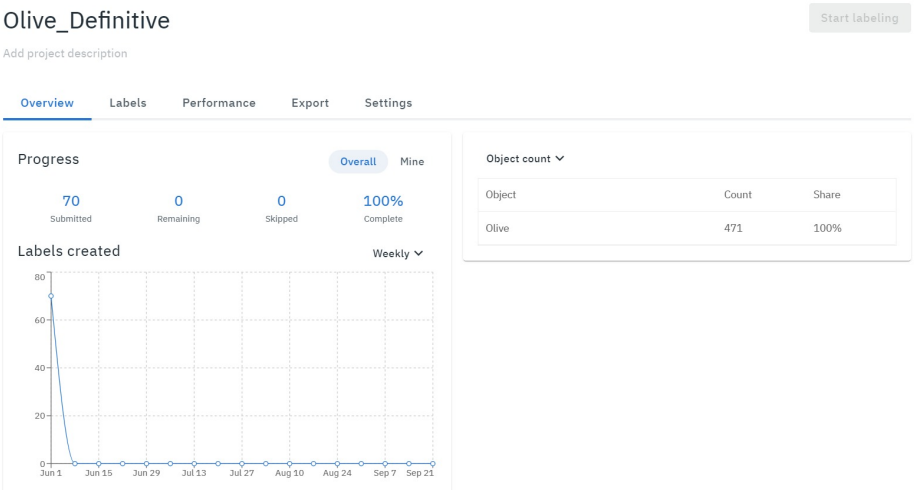

Figura 3.2 – Presentazione del primo dataset.

3.2 Esportazione in formato JSON

Tramite il comando ***export*** di *Labelbox*, è stato effettuato il download in locale del file contenente le label in formato JSON, che presentavano tutte la seguente struttura:

```
[
    {
        "ID": "ckb1081t208cm0755pvv7x5xn",
        "DataRow ID": "ckb0zy9wkxegk0ap87vo090ru",
        "Labeled Data": "https://storage.labelbox.com/ck9y1yply145b0870ok1os5bn%2F1b0b6871-962b-b733-09ef-97fa3e83cc82-Grow-Your-Own-O
        "Label": {
            "Olive": [
                {

                    "geometry": [
                        {
                            "x": 173,
                            "y": 314
                        },
                        {
                            "x": 255,
                            "y": 314
                        },
                        {
                            "x": 255,
                            "y": 385
                        },
                        {
                            "x": 173,
                            "y": 385
                        }
                    ]
                },
                {
                    "geometry": [
                        {
                            "x": 110,
                            "y": 317
                        },
                        {
                            "x": 189,
                            "y": 317
                        },
                        {
                            "x": 528,
                            "y": 324
                        },
                        {
                            "x": 528,
                            "y": 235
                        }
                    ]
                }
            ]
        },
        "Created By": "valerio.dangelo95@gmail.com",
        "Project Name": "Olive_Definitive",
        "Created At": "2020-06-04T16:39:15.000Z",
        "Updated At": "2020-09-22T15:08:54.000Z",
        "Seconds to Label": 140.137,
        "External ID": "Grow-Your-Own-Olives.jpg",
        "Agreement": -1,
        "Benchmark Agreement": -1,
        "Benchmark ID": null,
        "Dataset Name": "Olive_Definitive",
        "Reviews": [],
        "View Label": "https://image-segmentation-v4.labelbox.com?project=ckb0zwps1040h07812td84bk9&label=ckb1081t208cm0755pvv7x5xn"
    },
```


Figura 3.3 – Schema generale di una singola label.

Come si evince dalla figura qui sopra, ogni immagine è contrassegnata da un proprio ID, è memorizzata nel link presente in ***Labeled Data*** e presenta una chiave ***Label***.

Ad ogni chiave corrisponde un array contenente tutte le label assegnate a quell'immagine in oggetti chiamati ***geometry***: tali oggetti contengono le coordinate dei quattro vertici che la individuano nel piano immagine, secondo un sistema di riferimento di tipo assoluto.

Questo sistema, dove ad ogni pixel dell'immagine corrisponde una unità nel sistema di coordinate, ha le seguenti caratteristiche:

- l'origine degli assi è posto in alto a sinistra (*top-left*).
- l'orientamento dell'**asse Y** risulta essere verso il basso, mentre quello dell'**asse X** verso destra.

3.3 Parsing

Dato che il sistema di coordinate non è compatibile con il formato di Yolov3 (quello relativo), è stato necessario effettuare una conversione:

```
<object-class> <x-center> <y-center> <width> <height>
```

Figura 3.4 – Schema coordinate di una singola label

Nello schema in figura 3.4 si possono individuare diverse parole chiave:

- *<object-class>* indica il numero relativo alla classe della label: in questo caso, risulta essere sempre lo stesso, ovvero 0.
- *<x-center>* indica la coordinata X del centro della label: tale valore è relativo alla larghezza dell'intera immagine, risultato dell'operazione ***xabs /(image width)***.
- *<y-center>* indica la coordinata Y del centro della label, risultato dell'operazione ***yabs/(image height)***.
- *<width>* indica la larghezza della label, relativo alla larghezza dell'intera immagine, risultato dell'operazione **(*label width*)/(*image width*)**.
- *<height>* indica l'altezza della label, risultato dell'operazione **(*label height*)/(*image height*)**.

Tale conversione è stata automatizzata tramite il seguente algoritmo, di cui è riportato il codice:

- Viene importato il file JSON e create le apposite cartelle;
- per ogni oggetto nel file JSON, viene scaricata l'immagine presente nella chiave ***Labeled Data*** e le viene assegnato un nome pari al valore del contatore incrementale;
- per ogni immagine, vengono estratte le label;
- per ogni label vengono estratte le coordinate, e distinto il vertice in alto a sinistra da quello in basso a destra;
- vengono calcolate l'altezza e il centro della label (in sistema assoluto);
- tali valori vengono trasformati nel formato Yolov3 (in sistema relativo);
- viene scritta una riga nel file di testo relativo ad ogni immagine contenente le misure delle label sopra calcolate.

```
import json;
import numpy as np;
import requests
import os
import cv2

os.mkdir('Olives')
os.mkdir('Olives/images')

os.mkdir('Olives/labels')

json_file = open('/content/olives_70_with_ripening.json')
JSON = json.load(json_file)



imageCounter = 0
skippedCounter = 0
counter = 0

for row in JSON:
  counter += 1
  if((row['Label'])=='Skip'):
    skippedCounter += 1
  if((row['Label'])!='Skip'):
    imageCounter += 1
    url = row['Labeled Data']
    filename = '/content/Olives/images/'+str(imageCounter) + '.jpg'
    r = requests.get(url, allow_redirects=True)
    open(filename, 'wb').write(r.content)

print('Total images: ' + str(counter))
print('Skipped images: ' + str(skippedCounter))
print('Labeled images: ' + str(imageCounter))



import matplotlib.image as pltim

counter = 0
for row in JSON:
  if((row['Label'])!='Skip'):

    labels = (row['Label']['Olive'])
    counter +=1;
    file = open('/content/Olives/labels/'+str(counter)+'.txt', 'w+')

    for bbox in labels:

      height = 0
      width = 0

      min_x = 100000000
      max_x = 0
      min_y = 1111111110
      max_y = 0
      ripening = bbox['ripening']
      for vertex in bbox['geometry']:

        if(vertex['x']<min_x):
          min_x = vertex['x']
        if(vertex['x']>max_x):
          max_x = vertex['x']
        if(vertex['y']<min_y):
          min_y = vertex['y']
        if(vertex['y']>max_y):
          max_y = vertex['y']

      height = max_y - min_y
      width = max_x - min_x
      center = {
          'x': (max_x + min_x)/2,
          'y': (max_y + min_y)/2
      }

      img = pltim.imread('/content/Olives/images/' + str(counter) + '.jpg')
      abs_image_height = (img.shape[0])
      abs_image_width = (img.shape[1])

      relative_height = height/abs_image_height
      relative_width = width/abs_image_width

      relative_center_x = center['x']/abs_image_width
      relative_center_y = center['y']/abs_image_height
      file.write(str(int(ripening)-1) + ' ' + str(relative_center_x) + ' ' +
      str(relative_center_y) + ' ' + str(relative_width) + ' ' +
      str(relative_height) + '\n')
```

Figura 3.5 – Codice di estrazione delle label dalle singole immagini.

Il risultato finale ottenuto risulta essere il seguente:

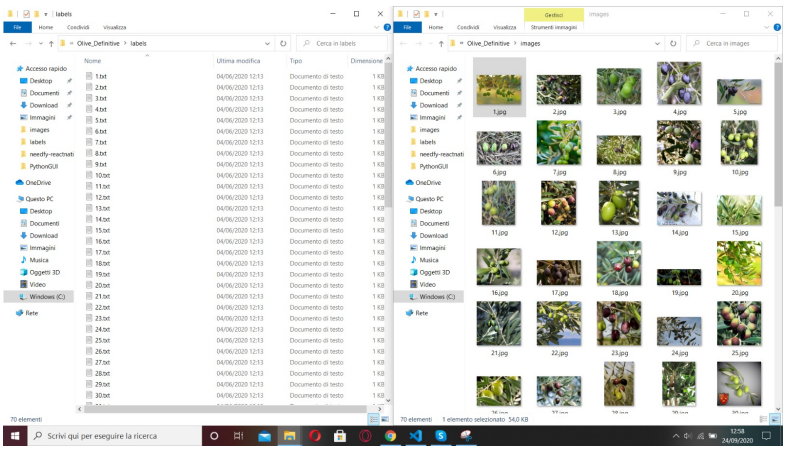

Figura 3.6 – Overview del primo dataset

Il primo dataset è quindi pronto: successivamente avverrà la fase di *augmentation* in modo da consentirne un ampliamento per permettere una maggior generalizzazione della rete.

3.4 Preparazione del dataset per i test

Il primo train della rete è stato effettuato con il dataset sopra presentato, dividendolo in un **rapporto 80:20** tra ***training set*** (l'insieme delle immagini che viene utilizzato per addestrare la rete) e ***validation set*** (un piccolo set di immagini che, periodicamente, è oggetto di un test da parte della rete neurale).

Tale suddivisione è necessaria per evitare il fenomeno dell'***overfitting***, che avviene quando la rete neurale viene addestrata troppo a lungo su un dataset di ridotte dimensioni, causando un apprendimento di tipo mnemonico e privo di qualsivoglia generalizzazione. In questo caso se viene data alla rete un'immagine diversa da quelle presenti nel dataset, questa fallisce nell'*object detection*.

In base ai parametri da generati dai test sul *validation set*, e al confronto con quelli generati sul *training set*, si riesce a capire quanto la rete neurale sia generalizzata.

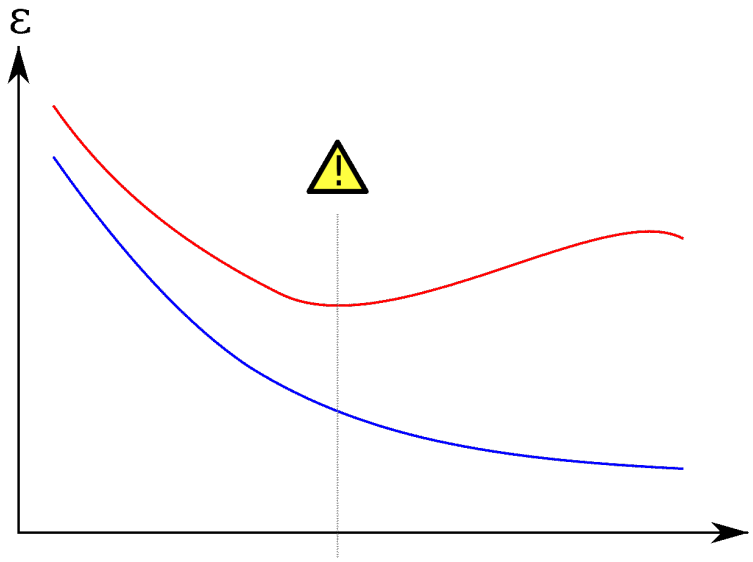

Figura 3.7 – Schema di illustrazione dell'overfitting (immagine tratta da `https://it.wikipedia.org/wiki/Overfitting`)

Come si vede dal grafico, se da un certo punto in poi l'errore nel *train set* si riduce mentre quello del *validation set* aumenta, si sta andando in *overfitting*: la rete non riuscirà a generalizzare l'apprendimento fallendo su dataset diversi da quelli del *train set*.

Per evitare di trovarsi in questa situazione si può procedere in diversi modi:

1. aumentare il dataset (in quantità e in varietà di immagini);
2. fermare il training prima del punto di minimo della funzione di costo del *validation set*;
3. attuare una ***regularization***, un processo che consiste nel semplificare il modello della rete neurale modificando il valore dei pesi o riducendo il numero di archi.

3.5 Train e setup di Yolov3

Per poter usare la rete neurale Yolov3 sono state eseguite una serie di operazioni preparatorie:

1. Modificare il file **yolov3.cfg** nel seguente modo: il numero di filtri è stato settato a 18 per ogni strato della rete neurale, così da farlo coincidere con quanto scritto nella documentazione (***filters=[5 + n] * 3 and classes=n, where n is class count***), mentre il numero di classi è stato settato a 1.
2. Effettuare una ***evolve*** che, come accennato in precedenza, ottimizza i valori degli iperparametri per avere prestazioni ottimali.

3. Settare il ***batch-size*** a 8 e il ***subdivisions*** a 16, valori risultati essere ottimali in termini di memoria ed efficienza.

3.6 Confusion matrix

Per tenere traccia dei veri/falsi positivi e dei falsi negativi è stato modificato il file ***test.py*** e il file ***utils.py***, così che ad ogni test effettuato (e ad ogni epoca) vengano esportati in formato txt i dati che formeranno la *confusion matrix*.

Nel file *utils.py* è stata modificata la funzione ***ap_per_class()*** (richiamata ad ogni test dal file *test.py*): tale funzione in origine restituiva *precision*, *recall*, *average precision*, e *focal loss*. Tuttavia, grazie ai dati che questa analizza è possibile far restituire anche i veri e falsi positivi (indicati nel codice come come tpc e fpc) e falsi negativi (fpc), evidenziati in rosso nell'immagine sottostante.

```python
def ap_per_class(tp, conf, pred_cls, target_cls):
    """ Compute the average precision, given the recall and precision curves.
    Source: https://github.com/rafaelpadilla/Object-Detection-Metrics.
    # Arguments
        tp:    True positives (nparray, nx1 or nx10).
        conf:  Objectness value from 0-1 (nparray).
        pred_cls: Predicted object classes (nparray).
        target_cls: True object classes (nparray).
    # Returns
        The average precision as computed in py-faster-rcnn.
    """

    # Sort by objectness
    i = np.argsort(-conf)
    tp, conf, pred_cls = tp[i], conf[i], pred_cls[i]

    # Find unique classes
    unique_classes = np.unique(target_cls)

    # Create Precision-Recall curve and compute AP for each class
    pr_score = 0.1  # score to evaluate P and R https://github.com/ultralytics/yolov3/issues/898
    s = [unique_classes.shape[0], tp.shape[1]]  # number class, number iou thresholds (i.e. 10 for mAP0.5...0.95)
    ap, p, r = np.zeros(s), np.zeros(s), np.zeros(s)
    for ci, c in enumerate(unique_classes):
        i = pred_cls == c
        n_gt = (target_cls == c).sum()  # Number of ground truth objects
        n_p = i.sum()  # Number of predicted objects

        if n_p == 0 or n_gt == 0:
            continue
        else:
            # Accumulate FPs and TPs
            fpc = (1 - tp[i]).cumsum(0)
            tpc = tp[i].cumsum(0)

            # Recall
            recall = tpc / (n_gt + 1e-16)  # recall curve
            r[ci] = np.interp(-pr_score, -conf[i], recall[:, 0])  # r at pr_score, negative x, xp because xp decreases

            # Precision
            precision = tpc / (tpc + fpc)  # precision curve
            p[ci] = np.interp(-pr_score, -conf[i], precision[:, 0])  # p at pr_score

            # AP from recall-precision curve
            for j in range(tp.shape[1]):
                ap[ci, j] = compute_ap(recall[:, j], precision[:, j])

            # Plot
            # fig, ax = plt.subplots(1, 1, figsize=(5, 5))
            # ax.plot(recall, precision)
            # ax.set_xlabel('Recall')
            # ax.set_ylabel('Precision')
            # ax.set_xlim(0, 1.01)
            # ax.set_ylim(0, 1.01)
            # fig.tight_layout()
            # fig.savefig('PR_curve.png', dpi=300)

    # Compute F1 score (harmonic mean of precision and recall)
    f1 = 2 * p * r / (p + r + 1e-16)
    return p, r, ap, f1, unique_classes.astype('int32'), tpc, fpc, (n_gt-tpc), (n_gt)
```

Figura 3.8 – Modifica del file utils.py per restituire i dati necessari per la confusion matrix

Sono state quindi aggiunte altre righe di codice per esportare i file di testo contenenti i valori aggiornati calcolati ad ogni test, e messi nel file ***confusion_matrix.txt***.

```
if len(stats):
    p, r, ap, f1, ap_class, tpc, fpc, fnc, n_gt = ap_per_class(*stats)
    tpc_txt = open('/content/tpc.txt', 'w')
    tpc_txt.write(str(tpc[len(tpc)-1]))

    fpc_txt = open('/content/fpc.txt', 'w')
    fpc_txt.write(str(fpc))

    fnc_txt = open('/content/fnc.txt', 'w')
    fnc_txt.write(str(fnc))

    ngt_txt = open('/content/ngt.txt', 'w')
    ngt_txt.write(str(n_gt))

    confusion_matrix_txt = open('/content/confusion_matrix.txt', 'w')
    confusion_matrix_txt.write('Veri positivi: ' + str(tpc[len(tpc)-1]) +
     '\n' +'Falsi Positivi: ' + str(fpc[len(fpc)-1]) + '\n' + 'Falsi Negativi: ' +
     str(fnc[len(fnc)-1]) + '\n' + 'Olive Totali: ' + str(n_gt) )
    print(str(type(tpc)))
    if niou > 1:
        p, r, ap, f1 = p[:, 0], r[:, 0], ap.mean(1), ap[:, 0]  # [P, R, AP@0.5:0.95, AP@0.5]
    mp, mr, map, mf1 = p.mean(), r.mean(), ap.mean(), f1.mean()
    nt = np.bincount(stats[3].astype(np.int64), minlength=nc)  # number of targets per class
else:
    nt = torch.zeros(1)

# Print results
pf = '%20s' + '%10.3g' * 6  # print format
print(pf % ('all', seen, nt.sum(), mp, mr, map, mf1))
```

Figura 3.9 – Modifica del file test.py per la scrittura su file dei dati relativi alla confusion matrix

3.7 Risultati

Di seguito vengono i risultati di questo primo train:

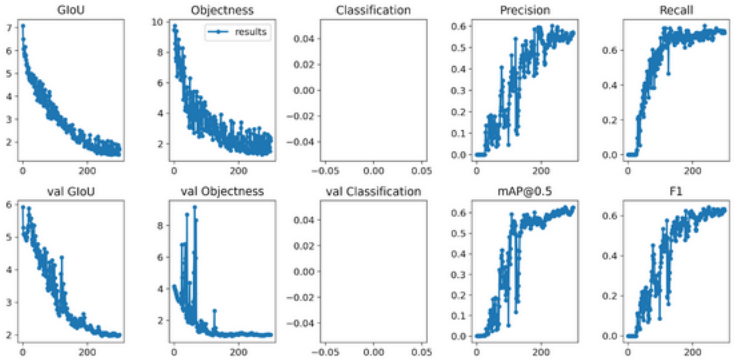

Figura 3.10 – Funzioni risultanti dal primo train effettuato

Tutte le funzioni di costo presentano grandi oscillazioni: esse sono principalmente causate dal ridotto numero di immagini che costituiscono il dataset.

Tuttavia non c'è *overfitting*: il valore delle funzioni di costo *validation GIoU* e *validation Objectness* si riduce nel tempo senza mai incrementare nuovamente, con andamento simile alle funzioni di costo del train set.

Di seguito sono mostrati anche alcuni risultati di *detection* effettuate sul *validation set*:

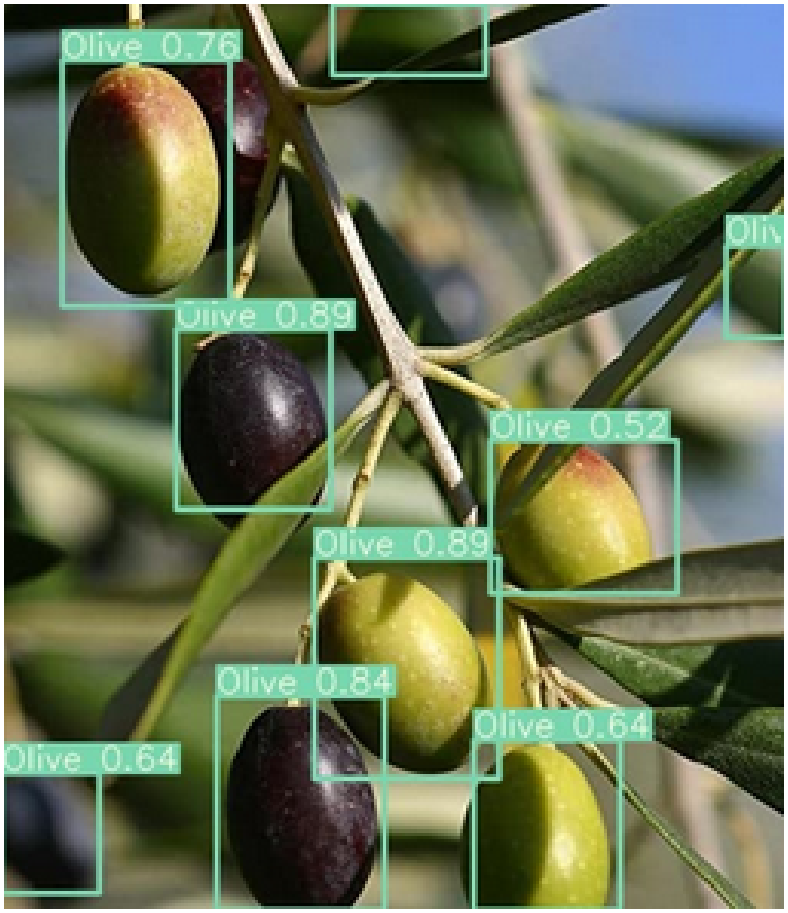

Figura 3.11 – Risultato di esempio della detection

Figura 3.12 – Risultato di esempio della detection

Il rapporto tra veri positivi e falsi negativi è da considerarsi buono, ma i

confidence scores risultano altalenanti: si passa da valori attorno allo 0.34 a valori anche superiori allo 0.90.

Questi valori più bassi sono in corrispondenza di olive non messe a fuoco, orizzontali o inclinate, che la rete non è addestrata a riconoscere. Per aumentare questi valori sarà necessario addestrare la rete su immagini ruotate, capovolte e con effetto *blur*, cosa che, come si vedrà più avanti, porterà grandi miglioramenti in termini di prestazioni.

Di seguito la *confusion matrix*:

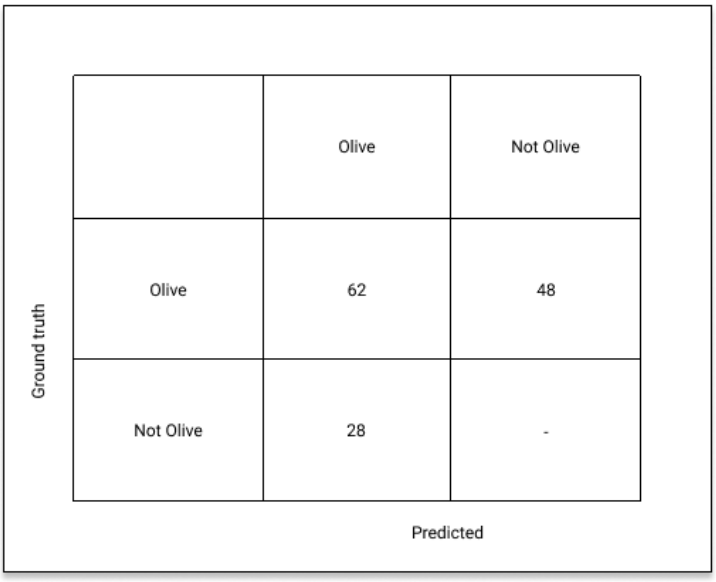

Figura 3.13 – Matrice di confusione relativa al validation set

La rete neurale ha rilevato correttamente 62 olive (i cosiddetti "veri positivi") sulle 90 totali. Inoltre sono presenti 28 falsi negativi e 48 falsi positivi.

Va ricordato che essendo il train effettuato su una singola classe, non ha propriamente senso parlare di “veri negativi”: questi risultano essere, in realtà, un numero indefinito.

Capitolo 4

Data augmentation

Il dataset precedente, sebbene contenente un discreto numero di immagini, non è sufficiente per effettuare una *detection* con risultati soddisfacenti. Le reti neurali convoluzionali ottengono grandi performance quando i dati in fase di test sono molto simili a quelli utilizzati in fase di apprendimento, ma, in caso di rotazioni, traslazioni o trasformazioni, hanno una scarsa capacità di generalizzazione.

Proprio per questo si effettua il cosiddetto **data augmentation** : tale tecnica consiste nel generare delle copie opportunamente modificate delle immagini del proprio dataset, in modo tale da addestrare la rete a riconoscere un maggior numero di oggetti, risolvendo quindi il precedente problema.

Ci si è serviti di quattro differenti tipi di trasformazioni: ***blurring***, ***rotazione***, ***flipping orizzontale*** e ***flipping verticale***. In seguito verranno mostrate nel dettaglio le loro caratteristiche ed implementazione al fine di aumentare il dataset.

4.1 Rotazione

E' stata scelta una rotazione di 30° gradi in senso antiorario, il cui risultato si può vedere nelle immagini qui sotto:

Figura 4.1 – Immagine originale

Figura 4.2 – Immagine ruotata di 30° in senso antiorario

L'implementazione di tale rotazione è composta dai seguenti step:

- **creazione dell'immagine-copia ruotata;**

- **cambio delle coordinate**:

  Dopo aver creato l'immagine-copia ruotata è necessario creare anche un altro file contenente le label, con coordinate X e Y aggiornate in seguito alla rotazione effettuata.

  Le coordinate di un punto che ruota attorno all'origine degli assi sono definite come segue:

$$X_{pr} = X_p\, Cos(\theta) - Y_p Sen(\theta)$$
$$Y_{pr} = X_p Sen(\theta) + Y_p Cos(\theta)$$

Figura 4.3 – Formula delle coordinate

Ciò si può vedere dalle seguenti immagini:

Figura 4.4 – Confronto delle coordinate

Come accennato in precedenza, in Yolo il centro è situato in alto a sinistra, con l'asse X e Y rivolti rispettivamente verso destra e verso il basso.

Ciò ha portato al modificare le equazioni invertendo il segno della Y:

$$X_{pr} = X_p\, Cos(\theta) + Y_p Sen(\theta)$$
$$Y_{pr} = -(X_p Sen(\theta) - Y_p Cos(\theta))$$

Figura 4.5 – Formula del cambio delle coordinate

Tali equazioni sono valide per un sistema di riferimento che ha il centro in (0,0), che corrisponde al centro dell'immagine: per ovviare a questo problema, nel calcolo della rotazione di ogni singolo punto, questo viene prima traslato, in modo da rendere il sistema di riferimento concorde a quello delle equazioni. Vengono così calcolate le coordinate del punto soggetto a rotazione (facendo riferimento alle equazioni precedenti), che poi vengono nuovamente traslate nel sistema di riferimento usato da Yolo.

Ricapitolando:

– primo passo

$$X_p = X_p - (image\ width)/2$$
$$Y_p = Y_p - (image\ height)/2$$

Figura 4.6 – Formula della normalizzazione delle coordinate

– secondo passo

$$X_{pr} = X_p\ Cos(\theta) + Y_p Sen(\theta)$$
$$Y_{pr} = -(X_p Sen(\theta) - Y_p Cos(\theta))$$

Figura 4.7 – Formula della rotazione delle coordinate

– terzo passo

$$X_{pr} = X_{pr} - (image\ width)/2$$
$$Y_{pr} = Y_{pr} - (image\ height)/2$$

Figura 4.8 – Formula del cambio di coordinate

Tutto il processo descritto finora è stato implementato nel seguente script mediante la libreria *OpenCV*:

1. Scarica ogni immagine dal JSON esportato da *Labelbox* e genera una copia ruotata di tale immagine:

```
import cv2
from scipy import ndimage
import math

counter = 0
for row in JSON:
  counter +=1;
  file = open('/content/Olives/labels/'+str(counter)+'_r.txt', 'w+')
  img = cv2.imread('/content/Olives/images/' + str(counter) + '.jpg')
  rows, cols, ht = img.shape

  matrix = cv2.getRotationMatrix2D((cols/2, rows/2), 30, 1)
  new_img = cv2.warpAffine(img, matrix, (cols, rows))
  cv2.imwrite(('/content/Olives/images/' + str(counter) + '_r.jpg'), new_img)
```

Figura 4.9 – Script della creazione delle copie ruotate

2. Per ogni bounding box presente nella label originale calcola il centro delle label originali, e le loro coordinate successive alla rotazione, tenendo conto delle considerazioni fatte in precedenza sul sistema di riferimento:

```
for bbox in labels:
    height = 0
    width = 0

    min_x = 100000
    min_y = 100000
    max_x = 0
    max_y = 0
    ripening = bbox['ripening']
    for vertex in bbox['geometry']:
      if(vertex['x']<min_x):
        min_x = vertex['x']
      if(vertex['x']>max_x):
        max_x = vertex['x']
      if(vertex['y']<min_y):
        min_y = vertex['y']
      if(vertex['y']>max_y):
        max_y = vertex['y']
    height = max_y - min_y
    width = max_x - min_x
    center = {
        'x': (max_x + min_x)/2,
        'y': (max_y + min_y)/2
    }

    appoggio = {
        'x': (max_x + min_x)/2,
        'y': (max_y + min_y)/2
    }

    center['x'] = center['x'] - cols/2
    center['y'] = center['y'] - rows/2

    appoggio['x'] = appoggio['x'] - cols/2
    appoggio['y'] = appoggio['y'] - rows/2

    center['x'] = appoggio['x']*math.cos(math.pi/6) + appoggio['y']*math.sin(math.pi/6)
    center['y'] = -(appoggio['x']*math.sin(math.pi/6) - appoggio['y']*math.cos(math.pi/6))

    center['x'] = center['x'] + cols/2
    center['y'] = center['y'] + rows/2

    relative_height = height/rows
    relative_width = width/cols

    relative_center_x = center['x']/cols
    relative_center_y = center['y']/rows
```

Figura 4.10 – Script del calcolo delle nuove coordinate

3. Controlla che la label ottenuta a seguito della rotazione sia effettivamente ancora contenuta nell'immagine. Se in seguito alla rotazione il centro della

label risulta essere fuori dall'immagine, tale label non è valida e quindi viene ignorata nel processo di scrittura su file:

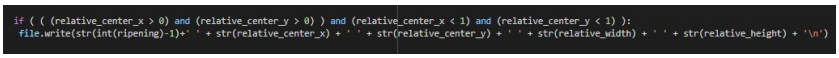

Figura 4.11 – Script del controllo delle label

4.2 Blurring

Il ***blurring*** è un processo attraverso il quale l'immagine viene sfocata, in modo più o meno intenso, a seconda di un parametro detto ***Blur Radius***.

L'effetto ***blur*** non modifica in alcun modo la posizione delle label, verranno quindi generate delle copie delle originali (sempre assegnando loro il nome adeguato).

Di seguito lo script per la creazione di immagini con effetto blur e le label a esse associate:

```
import cv2
from scipy import ndimage
import math


counter = 0

for row in JSON:
  counter +=1;
  file3 = open('/content/Olives/labels/'+str(counter)+'_b.txt', 'w+')
  img = cv2.imread('/content/Olives/images/' + str(counter) + '.jpg')
  blur = cv2.blur(img,(5,5))

  cv2.imwrite(('/content/Olives/images/' + str(counter) + '_b.jpg'), blur)
  labels = (row['Label']['Olive'])


  for bbox in labels:
      [...]
      Stessa procedura del parsing, lasciando invariate
      le variabili relative alle coordinate della bbox
      [...]
      file3.write('0 ' + str(relative_center_x) + ' ' +
str(relative_center_y) + ' ' + str(relative_width) + ' ' +
str(relative_height) + '\n')
```

Figura 4.12 – Codice di implementazione del blurring

Di seguito il risultato finale:

Figura 4.13 – Immagine originale, prima del blurring

Figura 4.14 – Immagine sfocata

4.3 Flipping orizzontale

Il ***flipping orizzontale*** consiste nel modificare l'immagine, assegnando ad ogni pixel in essa contenuto la coordinata simmetrica rispetto ad un asse verticale passante per il centro.

Le label vanno quindi opportunamente copiate e modificate: di esse cambierà solamente la coordinata X, che sarà simmetrica rispetto all'asse passante per il centro, mentre la coordinata Y risulterà invariata.

Figura 4.15 – Confronto coordinate tra immagine originale e flippata orizzontalmente

$$X_{pf} = (image\ width) - X_p$$
$$Y_{pf} = Y_p$$

Figura 4.16 – Formula per il calcolo delle coordinate

L'implementazione dello script è invece la seguente:

```python
import cv2
from scipy import ndimage
import math

counter = 0

for row in JSON:

  counter +=1;



  file2 = open('/content/Olives/labels/'+str(counter)+'_fh.txt', 'w+')
  img = cv2.imread('/content/Olives/images/' + str(counter) + '.jpg')
  flipHorizontal = cv2.flip(img, 1)
  cv2.imwrite(('/content/Olives/images/' + str(counter) + '_fh.jpg'), flipHorizontal)
  labels = (row['Label']['Olive'])


  for bbox in labels:
      height = 0
      width = 0

      min_x = 100000
      min_y = 100000
      max_x = 0
      max_y = 0

      for vertex in bbox['geometry']:
        if(vertex['x']<min_x):
          min_x = vertex['x']
        if(vertex['x']>max_x):
          max_x = vertex['x']
        if(vertex['y']<min_y):
          min_y = vertex['y']
        if(vertex['y']>max_y):
          max_y = vertex['y']
      height = max_y - min_y
      width = max_x - min_x
      center = {
          'x': (max_x + min_x)/2,
          'y': (max_y + min_y)/2
      }

      abs_image_height = (img.shape[0])
      abs_image_width = (img.shape[1])

      center3 = {
          'x': (abs_image_width - center['x']),
          'y': (max_y + min_y)/2
      }



      relative_height = height/abs_image_height
      relative_width = width/abs_image_width

      relative_center3_x = center3['x']/abs_image_width
      relative_center3_y = center3['y']/abs_image_height

      file2.write('0' + ' ' + str(relative_center3_x) + ' ' +
                  str(relative_center3_y) + ' ' + str(relative_width) + ' ' +
                  str(relative_height) + '\n')
```

Figura 4.17 – Implementazione per la creazione di immagini flippate orizzontalmente

Di seguito il risultato:

Figura 4.18 – Immagine originale

Figura 4.19 – Immagine flippata orizzontalmente

4.4 Flipping verticale

Il ***flipping verticale*** consiste nel modificare l'immagine, assegnando ad ogni pixel in essa contenuto la coordinata simmetrica rispetto ad un asse orizzontale passante per il centro dell'immagine.

Figura 4.20 – Immagine originale

Figura 4.21 – Confronto coordinate tra immagine originale e flippata verticale

Le label vanno quindi opportunamente copiate e modificate: di esse cambierà solamente la coordinata Y, la quale sarà simmetrica rispetto all'asse passante per il centro, mentre la coordinata X risulterà invariata.

Figura 4.22 – Formula per il flip verticale

$$X_{pf} = X_p$$
$$Y_{pf} = (image\ height) - Y_p$$

Figura 4.23 – Formula per il cambio di coordinate

L'implementazione dello script è la seguente:

```
import cv2
from scipy import ndimage
import math

counter = 0

for row in JSON:

  counter +=1;


  file = open('/content/Olives/labels/'+str(counter)+'_fv.txt', 'w+')


  img = cv2.imread('/content/Olives/images/' + str(counter) + '.jpg')


  flipVertical = cv2.flip(img, 0)

  cv2.imwrite(('/content/Olives/images/' + str(counter) + '_fv.jpg'), flipVertical)

  labels = (row['Label']['Olive'])


  for bbox in labels:
      height = 0
      width = 0

      min_x = 100000
      min_y = 100000
      max_x = 0
      max_y = 0

      for vertex in bbox['geometry']:
        if(vertex['x']<min_x):
          min_x = vertex['x']
        if(vertex['x']>max_x):
          max_x = vertex['x']
        if(vertex['y']<min_y):
          min_y = vertex['y']
        if(vertex['y']>max_y):
          max_y = vertex['y']
      height = max_y - min_y
      width = max_x - min_x
      center = {
          'x': (max_x + min_x)/2,
          'y': (max_y + min_y)/2
      }

      abs_image_height = (img.shape[0])
      abs_image_width = (img.shape[1])


      center2 = {
          'x': (max_x + min_x)/2,
          'y': (abs_image_height - center['y'])
      }

      relative_center2_x = center2['x']/abs_image_width
      relative_center2_y = center2['y']/abs_image_height

      file.write('0' + ' ' + str(relative_center2_x) + ' ' +
                 str(relative_center2_y) + ' ' + str(relative_width) + ' ' +
                 str(relative height) + '\n')
```

Figura 4.24 – Script per la realizzazione del flip verticale

4.5 Risultati

Sebbene il train sia stato effettuato sul dataset aumentato, è stato scelto lo stesso *validation set* del train precedente, così da avere risultati paragonabili tra loro sia nell'analisi delle funzioni di costo, sia nell'analisi delle *detection*.

Come ci si aspettava la performance della rete neurale è notevolmente aumentata rispetto a prima:

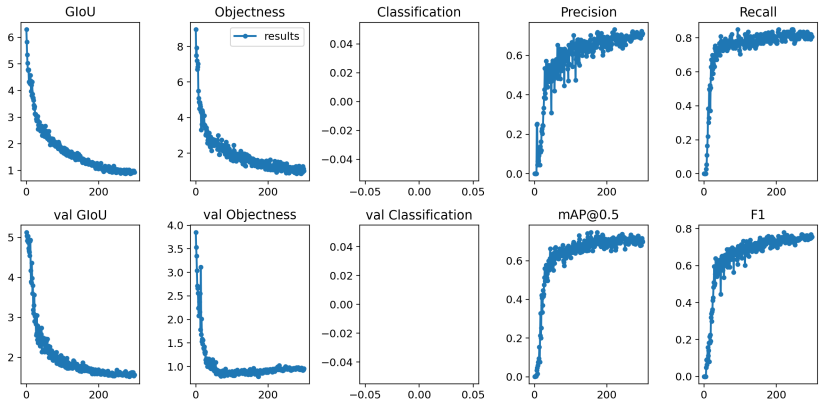

Figura 4.25 – Parametri del test effettuato

Alcune considerazioni:

1. Tutte le funzioni di costo non presentano più oscillazioni grazie all'aumento del dataset.

2. Neanche in questo caso siamo in presenza di *overfitting*: il valore delle funzioni di costo (***Validation GIOU*** e ***Validation Objectness***) si riduce nel tempo senza mai incrementare nuovamente, con andamento simile alle funzioni di costo del *train set*.

3. Le funzioni di costo raggiungono complessivamente valori di gran lunga migliori (0.9 per la *GIOU* e 1.01 per la *objectness loss*) rispetto al dataset senza *augmentation* (rispettivametne 1.69 per la *GIOU* e 2.21 per la *objectness loss*).

4. La funzione *precision* raggiunge il valore finale di 0.71: ciò vuol dire che su 100 oggetti trovati dalla rete neurale, 71 sono veri positivi, mentre 29 falsi positivi.

5. La funzione *recall* raggiunge il valore finale di 0.80: ciò vuol dire che su 100 oggetti rilevati dalla rete neurale, 80 sono veri positivi e 20 falsi negativi.

In basso seguono alcune *detection* effettuate dalla rete neurale, a sostegno delle considerazioni appena fatte.

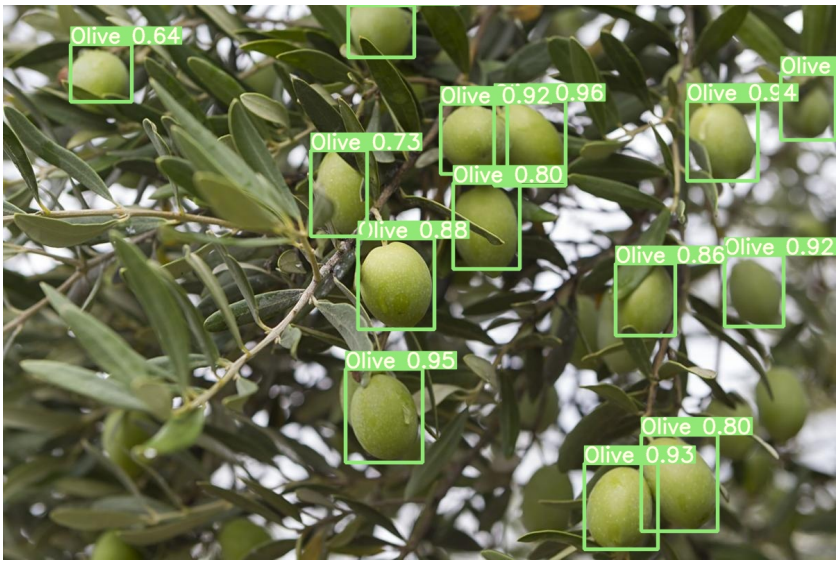

Figura 4.26 – Esempio di detection della rete neurale

Figura 4.27 – Esempio di detection della rete neurale

Il rapporto tra veri positivi e falsi negativi risulta essere migliore del caso precedente, così come i *confidence scores*, che non scendono sotto il 0.60 (tranne che in rari casi), tenendosi in media al di sopra dello 0.90.

Questi risultati confermano l'importanza del *dataset augmentation* per il raggiungimento di performance valide.

Di seguito è mostrata la relativa *confusion matrix*.

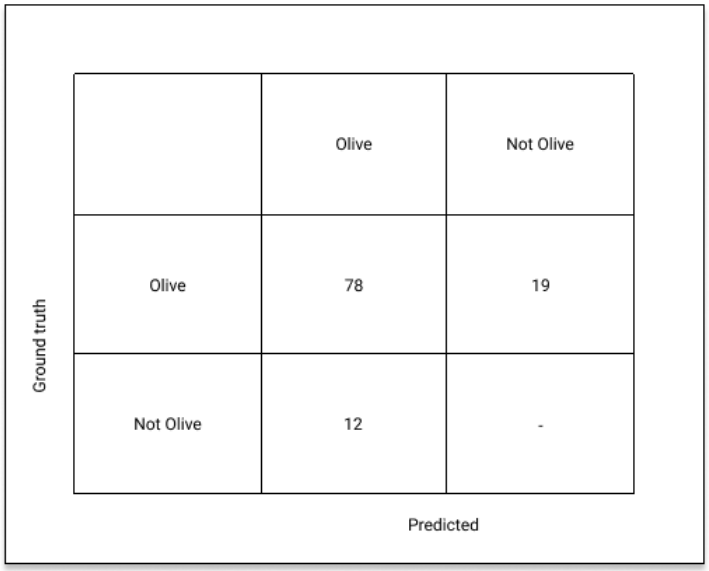

Figura 4.28 – Confusion matrix relativa al validation set

Capitolo 5

Ultimo test con immagini dal vero

5.1 Conclusioni

La rete neurale addestrata finora ha dato ottimi risultati, però il tutto è avvenuto su un dataset contenente immagini campione, o comunque di olive e rami in primo piano.

Con immagini più realistiche l'applicazione pratica del software non risulterebbe altrettanto soddisfacente poiché le immagini dal vero che si andrebbero ad analizzare presentano più disturbi o inquadrature da maggiori distanze. Da qui la necessità di espandere il dataset con immagini più simili a quelle reali, in modo da prevenire scarsi risultati nelle applicazioni future.

5.1.1 Il dataset

Questo dataset aggiuntivo presentava 53 immagini, comprese copie delle stesse ottenute dall'*augmentation*, per un totale di 2234 label.

Figura 5.1 – Esempio di immagine presa dal nuovo dataset

Figura 5.2 – Esempio di immagine presa dal nuovo dataset

Tali copie ottenute dall'*augmentation* sono state rimosse e ignorate, in modo da effettuare l'*augmentation* manualmente arrivando a un totale di 30 immagini, che sono state aggiunte al precedente dataset con un rapporto 50/50 (in modo da generalizzare la rete il più possibile), mentre le restanti immagini hanno formato il *validation set*.

Per effettuare l'*augmentation* manualmente si è partiti dal JSON contenente tutte le label; tuttavia, quest'ultimo risultava essere di una versione differente (e nuova) di *LabelBox*:

Figura 5.3 – Esempio del nuovo formato delle label

Tali label non sono più individuate attraverso i quattro vertici contenuti in "geometry", ma da un oggetto "bbox" che individua il vertice in alto a sinistra con relativa altezza e larghezza.

E' stato quindi necessario creare ed utilizzare un nuovo parser.

5.1.2 il nuovo parser

A fronte dell'aggiornamento del formato delle label è stato necessario effettuare delle piccole modifiche al parser: se prima l'algoritmo individuava il vertice in alto a sinistra confrontandolo con tutti gli altri, ora tale procedimento può essere saltato, rendendo il tutto più veloce.

Di seguito è mostrato il nuovo script del parser:

```
for objectl in objects:
  # print(objectl)
  height = objectl['bbox']['height']
  width = objectl['bbox']['width']
  x = objectl['bbox']['left']
  y = objectl['bbox']['top']

  center = {
      'x': (x + width/2),
      'y': (y + height/2)
  }

  img = pltim.imread('/content/Olives/images/Olives' + str(counter) + '.jpg')
  abs_image_height = (img.shape[0])
  abs_image_width = (img.shape[1])

  relative_height = height/abs_image_height
  relative_width = width/abs_image_width
```

Figura 5.4 – Il nuovo script creato per il parsing

Il centro della *bounding box* viene calcolato a partire dal vertice in alto a sinistra, e poi basta sommare metà altezza e metà larghezza, e infine effettuare l'*augmentation* con *blurring*, rotazione e *flipping* allo stesso modo del vecchio parser.

5.1.3 Risultati e detection

Di seguito sono presentati i risultati delle funzioni di costo del training effettuato, con le relative funzioni di *precision*, *recall*, *mAP*, e *focal loss*.

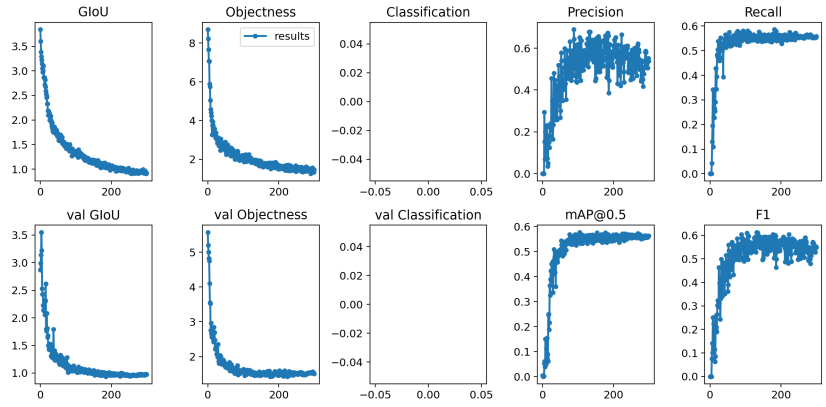

Figura 5.5 – Risultati del train effettuato con il nuovo dataset

Alcune considerazioni:

- Come previsto, non c'è *overfitting*: le funzioni di costo del *validation set* hanno un andamento simile a quelle del *train set*, senza aumentare indefinitamente da un certo punto in poi.
- La *precision* raggiunge il valore finale di 0.539: ciò vuol dire che su 100 oggetti trovati dalla rete neurale 54 sono veri positivi, mentre 46 falsi positivi.

- La *recall* raggiunge il finale di 0.55: ciò vuol dire che su 100 oggetti rilevati dalla rete neurale 55 sono veri positivi, mentre 45 falsi negativi.

I risultati mostrati sono da considerarsi eccellenti, nonostante in questo caso le immagini prese da vivo del *validation set* abbiano messo a dura prova l'algoritmo di *detection*, come si vede nella *confusion matrix* di seguito:

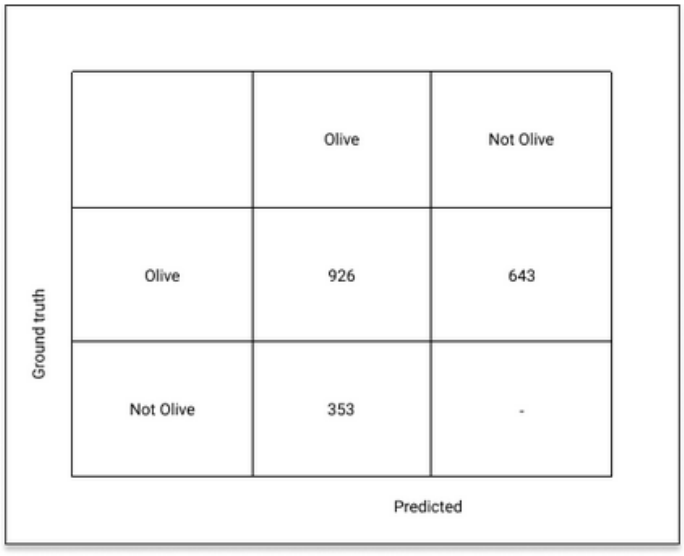

Figura 5.6 – Confusion matrix relativa al training effettuato

Per mostrare i miglioramenti nella generalizzazione della rete è stata effettuata una *detection* su immagini reali, sia con i pesi ottenuti nel training precedente sia con i pesi ottenuti dal nuovo dataset.

Figura 5.7 – Esempio di detection con i pesi del precedente train

Figura 5.8 – Esempio di detection con i pesi aggiornati

Come si evince dal confronto, nelle applicazioni reali (dell'agricoltura 4.0) l'utilizzo di un dataset adeguato si rivela cruciale nelle prestazioni della rete e nei suoi risultati.

Capitolo 6

Sviluppi futuri

In questo capitolo conclusivo verranno presentate due particolari applicazioni del software realizzato nel campo dell'agricoltura 4.0.

6.1 Camera Multispettrale

Una possibile applicazione è l'utilizzo del software su immagini non RGB, ma ottenute da camere multispettrali.

La camera multispettrale è una particolare camera con un sensore sensibile alla luce infrarossa non visibile all'occhio umano (vicino infrarosso 700-1000nm).

Tale camera può acquisire diverse bande spettrali dallo stesso punto di vista eliminando la necessità di dover effettuare operazioni di co-registrazione. La differenza tra le immagini varia a seconda della composizione chimica dell'oggetto stesso: determinati elementi o composti chimici assorbono, infatti, la radiazione infrarossa in modo diverso, e dunque modificheranno lo spettro in modo caratterizzante.

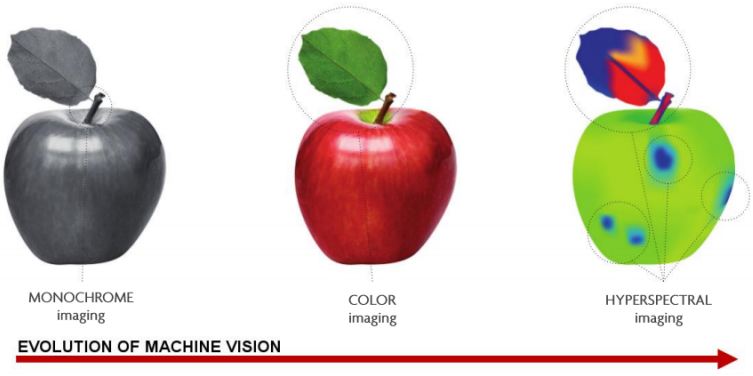

Figura 6.1 – Confronto dell'analisi di una mela con differenti tecniche di visione artificiale (immagine tratta da `2_imec_HSI_technology.pdf`)

In caso di applicazione agricola, tali camere posso essere montate a bordo di un drone aereo o terrestre per effettuare operazioni di conteggio e di stima del

grado di maturazione ottimizzando il processo di raccolta al fine di garantire una uniformità in termini di qualità.

6.2 Farmer-bots

Un altro possibile utilizzo è quello di installare il software su uno dei cosiddetti *Farmer-bots*, dei robot interattivi che fungono da "esperti agricoli" con l'intento di fornire agli agricoltori stessi utili informazioni sul tempo atmosferico, sull'andamento del mercato agricolo, e sui prodotti coltivati.

Per dare informazioni il più possibile corrette e puntuali, questi robot dovranno essere adeguatamente "allenati" e aggiornati su tutto ciò che riguarda il mondo agricolo, quindi un software come quello presentato in questo progetto può essere sicuramente d'aiuto.

Bibliografia

[1] Matthieu De Clercq, Anshu Vats, and Alvaro Biel. Agriculture 4.0: the future of farming technology. https://www.oliverwyman.com/our-expertise/insights/2018/feb/agriculture-4-0--the-future-of-farming-technology.html, 2018.

[2] Giorgio dell'Orefice. Agricoltura 4.0 business da 450 milioni (+22% annuo). *Ilsole24ore*, 2020.

[3] Gullo Di Giuseppe. Deep learning svelato: ecco come funzionano le reti neurali artificiali. *Italiancoders*, 2018.

[4] Jonathan Hui. Real-time object detection with yolo, yolov2 and now yolov3. *Medium*, 2018.

[5] Jonathan Hui. map (mean average precision) for object detection. *Medium*, 2018.

[6] Supervisely. Confusion matrix. https://supervise.ly/explore/plugins/confusion-matrix-75279/overview, 2020.

[7] Wikipedia contributors. Overfitting. https://en.wikipedia.org/wiki/Overfitting, 2020.

[8] Maria Crucitti. Che cosa sono le reti neurali? *Quora*, 2017.

[9] 3Blue1Brown. What is backpropagation really doing? | deep learning, chapter 3. https://www.youtube.com/watch?v=Ilg3gGewQ5U, 2017.

[10] Matthieu De Clercq, Anshu Vats, and Alvaro Biel. Agriculture 4.0: the future of farming technology. https://www.oliverwyman.com/our-expertise/insights/2018/feb/agriculture-4-0--the-future-of-farming-technology.html, 2018.

[11] Luca Maria De Nardo. Iperspettrale, camera. *PackagingWords*, 2019.

[12] Great Learning Snippets. Farmer-bot : An interactive bot for farmers. *Medium*, 2020.

Elenco delle figure

Elenco delle tabelle