Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Informatica

Sperimentazione di tecniche di Deep Learning per l'Object Detection

Relatore:
Chiar.mo Prof.
Andrea Asperti

Presentata da:
Nico Giambi

Sessione II
Anno Accademico 2019-2020

Introduzione

Negli ultimi anni, il **Machine Learning** è diventato uno dei settori dell'Informatica più studiati, e si prevede che in futuro l'interesse verso questo mondo affascinante possa solo aumentare. Ma prima di tutto, cos'è il *Machine Learning*? Concettualmente, si tratta di usare algoritmi specifici per identificare dei pattern che si ripetono spesso in un certo **Dataset**, e in base a questi fare delle predizioni quanto più in linea possibile con lo stesso *Dataset* . Ad esempio, supponiamo di essere grandi fan della musica Folk Irlandese, e quindi avere una grande collezione di musica di questo genere, ma questo non ci basta e la nostra passione verso il Folk Irlandese ci spinge a voler produrre una canzone degna di entrare nella nostra collezione. Sfruttando degli algoritmi di *Machine Learning*, ad esempio, potremmo fare *ascoltare* al nostro PC tutta la nostra collezione, più e più volte, finché il nostro algoritmo riesce a discernere dei pattern ricorrenti, che possono essere sequenze di note, BPM delle canzoni e quant'altro. A questo punto, quando crediamo che il nostro PC sia abbastanza sicuro di aver imparato delle caratteristiche della musica che gli abbiamo fatto ascoltare, gli chiediamo di creare qualcosa di diverso da tutte le canzoni sentite, ma comunque in linea con il genere ascoltato. Se l'algoritmo di *Machine Learning* è corretto, il nostro PC sarebbe a questo punto in grado di creare qualcosa che potremmo apprezzare da amanti del Folk Irlandese quali siamo, o perlomeno una base di idea interessante da sviluppare con i nostri amici amanti del Folk. L'esempio sopracitato, sebbene raccontato in modo informale, rappresenta a grandi linee il concetto del *Machine Learning*, che prevede l'analisi di grandi quantità di dati per

cogliere i tratti fondamentali e in qualche modo sfruttarli a proprio favore. La Produzione Musicale è solo uno dei molti di campi di cui si interessa il *Machine Learning*, per tutti i problemi i cui dati possono essere registrati e formalizzati in grandi quantità, c'è un metodo di approccio alla risoluzione tramite *Machine Learning*. In questa tesi verrà trattato un altro tra i temi più studiati in questo campo, ovvero l'*Object Detection*. In poche parole, si tratta dell'individuazione di oggetti (cose, animali, persone) all'interno di un'immagine, e contemporaneamente riuscire a capire a quale categoria questi ultimi appartengono. Ovvero riuscire a vedere **dove** sono gli oggetti all'interno dell'immagine e allo stesso tempo capire **cosa** sono. Ci sono molti modelli di *Machine Learning* diversi fra loro in grado di svolgere il task dell' **Object Detection** , e per ognuno di questi esistono diversi algoritmi da poter sperimentare, per riuscire a capire quali sono le migliori combinazioni modello/algoritmo. Per la realizzazione di questa tesi è stato usato il modello *MobileNet*, associato all'algoritmo di individuazione YOLO (*You Only Look Once*), entrambi rinomati per essere ottimi per la *Real Time Object Detection* , ovvero l'individuazione di oggetti in tempo reale, che richiede una certa velocità di elaborazione dei dati. Il *Dataset* utilizzato al fine della realizzazione dell'*Object Detector* è **COCO** (*Common Objects in COntext*) che contiene circa 135.000 immagini con le relative annotazioni (informazioni sul contenuto dell'immagine), che contengono oggetti relativi a 80 diverse classi. Lo scopo di questa tesi è stato quello di sperimentare la ricostruzione di un Object Detector a partire da un modello *MobileNet* già allenato su un altro *Dataset*, ovvero Imagenet, che contiene immagini contenenti oggetti relativi a 1000 classi, per il task della *Classificazione*. La maggior parte del lavoro infatti è stato cercare di fare transfer-learning, ovvero utilizzare ciò che il modello già aveva appreso dalle immagini di Imagenet per avere una base di partenza su cui poi costruire l'Object Detector per COCO. Per realizzazione di quest'ultimo sono state usati Python e delle librerie apposite per il *Machine Learning*, quali TensorFlow e Keras.

Nel primo capitolo verranno brevemente introdotti alcuni concetti chiave come *Machine Learning*, *Deep Learning*, *Computer Vision*, e *Training*. Nei capitoli successivi verranno descritte (in ordine) le parti chiave utilizzate per la costruzione dell'*Object Detector*, ovvero il **modello** (*MobileNet*), l'**algoritmo** (YOLO), il **dataset** (COCO) e **la piattaforma d'appoggio per il training** (*Google Colab*). Nel sesto capitolo, verrà spiegato passo a passo il lavoro svolto e le varie tecniche utilizzate per raggiungere l'obiettivo finale. Inoltre nella sezione Risultati si discuterà del funzionamento dell'*Object Detector* e della scelta di alcuni parametri che portano al risultato ottenuto.

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Machine Learning

1.1 Cos'è il Machine Learning

Il *Machine Learning* (in Italiano *Apprendimento Automatico*), è una branca dell'intelligenza artificiale che si occupa principalmente di statistica computazionale, individuazione di pattern, data mining. Utilizzato sia nel campo statistico che in quello informatico, il *Machine Learning* è una delle soluzioni più attuali per trattare problemi che coinvolgono l'analisi di grandi quantità di dati. Il concetto di base del *Machine Learning* è quello di dare in pasto ad un algoritmo una grande quantità di dati dello stesso tipo (immagini, musica, testi) e fare in modo che questo riesca man mano ad apprendere autonomamente le caratteristiche ricorrenti (*Features*) del dataset che gli è stato mostrato. Questo apprendimento può essere categorizzato in due macro categorie, ovvero *supervisionato* e *non-supervisionato*. Per la prima, occorre che il dataset dato in input sia *etichettato*, cioè che per ogni oggetto del dataset si abbiano le informazioni che vorremmo che il nostro algoritmo riesca ad apprendere, in modo da fargliele confrontare con ciò che ha attualmente appreso e vedere quanto sia ancora lontano dalla soluzione corretta. Per la seconda, invece, non c'è bisogno di dare informazioni aggiuntive all'algoritmo, in modo che questo riesca a capire senza essere istruito, le informazioni che contraddistinguono quel preciso dataset. In questa tesi, viene utilizzato

perlopiù il *Machine Learning supervisionato*, con la sola eccezione che poche immagini del dataset sono volutamente lasciate senza informazione. I problemi per la quale soluzione ci si affida al Machine Learning sono suddivisibili in tre macro categorie: *Classificazione, Regressione e Clustering*.

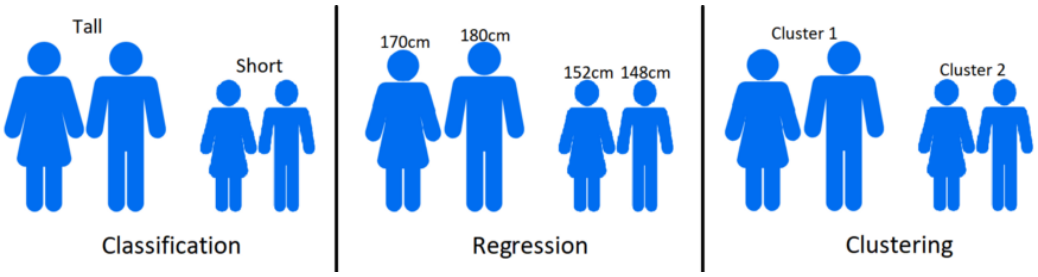

Figura 1.1: Classificazione, Regressione & Clustering

I problemi di *Classificazione* si specializzano nel categorizzare i diversi oggetti del dataset. Ad esempio, un modello per la classificazione di persone potrebbe riconoscere **se** una persona è alta, bassa o di statura media. I problemi di *Regressione* invece puntano a riconoscere informazioni riguardo all'oggetto in input. Sempre rimanendo nell'ambito di riconoscimento di caratteristiche umane, potremmo farci dire dal nostro modello **quanto** è alta una persona. Il terzo tipo di problemi, ovvero quelli di *Clustering* potrebbero servire, restando sullo stesso esempio, a suddividere un gruppo di persone in base alla loro altezza, senza conoscere il concetto di alto o basso né avere informazioni sulla loro statura.

1.2 Deep Learning

Il **Deep Learning** rappresenta una classe di algoritmi di *Machine Learning* utilizzati per l'estrazione di *features* ad alto livello, e i relativi modelli sono caratterizzati generalmente da due o più strati nell'**Hidden Layer** (*Strato Nascosto*). Le **Reti Neurali** per *Machine Learning* sono solitamente caratterizzati da un *layer* di **input**, un *layer* **nascosto** ed uno di **output**

(Figura 1.2) e associano solitamente agli *input* pattern semplici tramite lo strato *nascosto*, per poi produrre un *output* di qualche tipo.

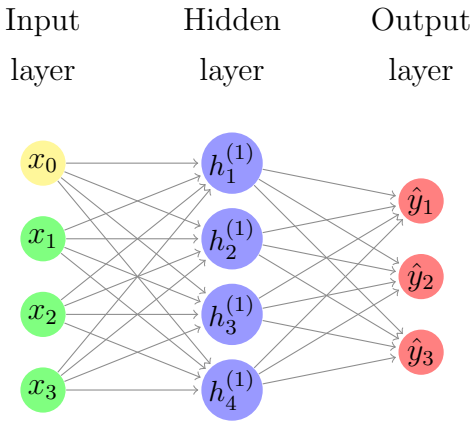

Figura 1.2: Esempio di Rete Neurale

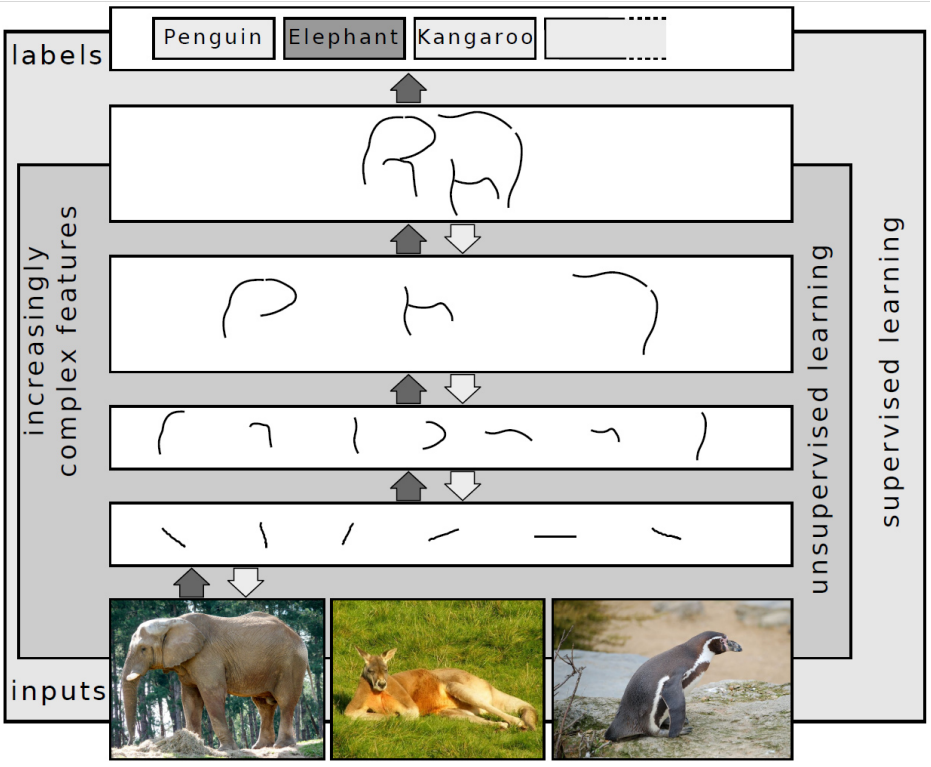

Figura 1.3: Estrazione delle Features a più livelli

La presenza di diversi *layer* nello strato nascosto della **Rete Neurale Profonda** (**Deep Neural Network**), permette invece di estrarre *features* grezze e raffinarle mano a mano che si avanza verso l'*output*. Ad esempio, se vogliamo analizzare un'immagine che raffigura un volto, il primo dei *layer* nascosti di una *Rete Neurale Profonda* potrebbe estrarre come feature i bordi o gli angoli di varie parti del volto, il secondo qualche forma particolare (naso, denti, bocca) e così via fino ad avere una mappa di *features* ad un alto livello di astrazione (Figura 1.3). Le **CNN** (**Convolutional Neural Network**), utilizzate per questa tesi (*MobileNet*, *InceptionV3*) sono un esempio di *Reti Neurali Profonde*.

1.3 Computer Vision

La *Computer Vision* (in Italiano *Visione Artificiale*) è un insieme di meccanismi che cercano di simulare la vista umana, ovvero riuscire a capire informazioni (in modo tridimensionale) a partire da immagini o video (bidimensionali). I compiti principali della branca della *Computer Vision* sono tre:

- ***Image Classification***: passando in input un'immagine con un solo oggetto, si riesce a capire **cosa** è l'oggetto in questione (ad esempio riconoscere la razza di un cane)
- ***Object Recognition***: prendendo in input un'immagine con più oggetti, si riesce a capire **dove** questi oggetti si trovano all'interno dell'immagine (ed eventualmente disegnare una *Bounding Box* attorno ad ognuno di essi).
- ***Object Detection***: L'unione dei due precedenti, si cerca di capire sia **dove** sono gli oggetti nell'immagine, sia **cosa** sono.

L' *Object Detector*, oggetto di questa tesi, come suggerisce il nome, appartiene alla terza categoria sopracitata.

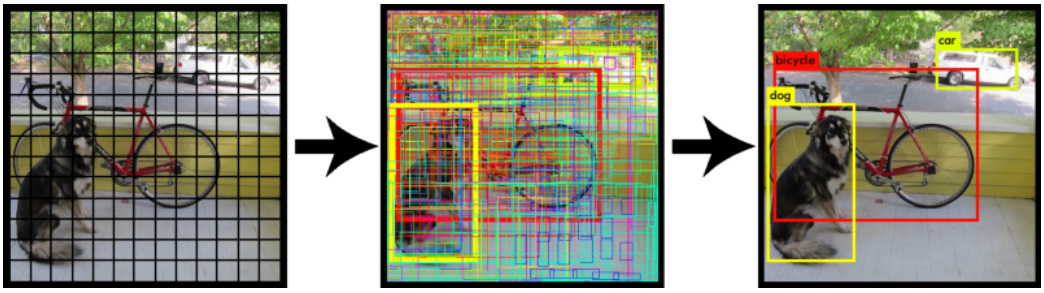

Figura 1.4: Object Detector, YOLO

1.4 Training

Il *Machine Learning* ha un'infinità di utilizzi possibili, sia per la vita di tutti i giorni (ad esempio farsi consigliare cosa guardare in TV), sia in scala globale, ma purtroppo ha anche dei lati negativi. Per poter performare analisi di Dataset di dimensioni notevoli e quindi avere un modello di *Machine Learning (ML)* in grado di svolgere al meglio il suo compito, c'è bisogno di un'enorme potenza di calcolo. In caso questo non sia possibile, un modello di *ML* potrebbe impiegare mesi, anni o anche di più per iniziare ad avere risultati ottimi. Questo perché durante la fase del ***Training*** (ovvero la fase di *apprendimento*), si va incontro a diversi strati di operazioni che possono coinvolgere matrici a molte dimensioni, o comunque elementi molto grandi, e questo deve essere ripetuto per ognuno degli oggetti di un dataset più e più volte. Per questo, avere un acceleratore Hardware (*GPU* o *TPU*) è essenziale per poter approcciarsi al *ML*. Essendo questi abbastanza costosi e quindi non accessibili a tutti, ci sono anche delle piattaforme online che mettono a disposizione i loro acceleratori Hardware (es. *Google Colab, Kaggle*) per poter introdurre nuove persone a questo mondo.

1.5 Transfer Learning

Un altro modo per ridurre il tempo impiegato dalla fase di *Training* di un modello di *ML*, è l'utilizzo del ***Transfer Learning***. Come suggerisce il nome, questa pratica serve per fare in modo che per affrontare dei *task* simili non bisogna allenare un modello da capo ogni volta, ma è possibile far sì che alcune *feature* più comuni apprendibili da un modello vengano riutilizzate per un altro scopo. Per fare un esempio, supponiamo che qualcuno a disposizione di un'elevata potenza di calcolo abbia allenato un modello a distinguere immagini contenenti un cane da quelle contenenti un gatto, e in poco tempo sia riuscito ad ottenere risultati notevolmente positivi. A questo punto, potremmo decidere di ampliare questo modello e provare a distinguere le immagini con un cane dalle immagini con una volpe. Essendo già stato allenato per l'altro *task*, ora il modello è già in grado di riconoscere immagini con dei cani, quello che potremmo fare è fargli capire in cosa si differenzia un cane da una volpe, e questo sarebbe molto più facile e veloce rispetto al prendere un modello che non sa ancora fare niente e fargli imparare tutto da zero. Il *Transfer Learning*, quindi, è un ottima risorsa per riuscire a risolvere dei task simili a quelli più famosi e studiati, risparmiando così molto tempo e ottimizzando il lavoro da compiere. Per questa tesi il *Transfer Learning* è stato fondamentale, in quanto invece di creare un *Object Detector* dal nulla, siamo partiti da un modello (*MobileNet*) già allenato per la classificazione (anche se per un dataset diverso dal nostro), riducendo così di molto i tempi di calcolo necessari ad ottenere risultati quantomeno accettabili.

Capitolo 2

Mobilenet

Le *Reti Neurali Convoluzionali* (*Convolutional Neural Network*, *CNN*), sono una classe di *Reti Neurali Artificiali feed-forward* costruite seguendo lo schema di elaborazione delle immagini da parte dell'occhio degli animali. Le *CNN* sono ottime per task come *Computer Vision*, ma anche in altri settori come la *Bioinformatica*. Generalmente sono composte da decine di *layer* che si alternano tra *Convoluzionali*, *Pooling* e *ReLu*. ***MobileNet*** è un modello particolarmente snello ed efficiente appartenente a questa categoria di Reti Neurali.

2.1 Vantaggi e Svantaggi

MobileNet è nata come *CNN* da essere usata su dispositivi mobili e embedded, entrambi classi di dispositivi che non hanno accesso ad una grande potenza di calcolo. Per ovviare a questo problema *MobileNet* è basata sulla *Depthwise Separable Convolution*, che serve a fattorizzare il lavoro di una normale convoluzione in due parti, una convoluzione in profondità ed un'altra puntuale, riducendo notevolmente il numero di parametri su cui dover lavorare. Senza approfondire troppo la questione che va al di fuori dell'obiettivo di questa tesi, si può dire che è un modello molto veloce rispetto alle sue concorrenti. Questo però ovviamente porta anche a degli svantaggi.

Se si cerca di migliorare in velocità, ovviamente si va a finire per perdere in precisione, infatti *Mobilenet* è leggermente inferiore alle sue concorrenti in quei termini. Nonostante ciò, è molto utile anche per scopi didattici come in questo caso, visto che ci interessa di più sperimentare diverse cose ed ottenere risultati accettabili rispetto a provarne magari solo una ma con risultati eccellenti. Infatti, inizialmente il modello prescelto è stato *InceptionV3*, una delle migliori *CNN* per quanto riguarda la precisione in *Computer Vision*, ma abbiamo deciso in corso d'opera di passare a *MobileNet* per le motivazioni sopracitate.

2.2 Utilizzo dei pesi derivanti da Imagenet

Un grande vantaggio reso possibile dalla grande operabilità di ***Tensorflow***, è stato quello di poter importare il modello *MobileNet*, ma già allenato su un dataset di immagini, ovvero *Imagenet* (che vanta circa 14 milioni di immagini contenenti più di 20.000 classi di oggetti differenti). Avere il modello già allenato su questo dataset ha reso molto più veloce il lavoro, evitando di dover allenare da zero *MobileNet*, che seppur veloce rispetto ad altri, con la potenza di calcolo a nostra disposizione sarebbe stato comunque uno scoglio difficile da superare.

Capitolo 3

YOLO

3.1 Idea Generale

YOLO (*You Only Look Once*) è un algoritmo usato nel campo dell'*Object Detection* che eccelle sia in velocità che in precisione, attributi tali da renderlo un ottimo candidato per la *Real Time Object Detection*. Il nome YOLO indica una delle sue caratteristiche più importanti, ovvero la capacità di ottenere informazioni su tutto l'input in un solo passo di *Forward Propagation*. Le informazioni delle immagini date in input percorrono la rete neurale in una sola direzione, ogni layer della rete neurale viene visitato una ed una sola volta per ogni immagine. Una rete neurale che opera secondo il metodo *YOLO*, prende generalmente in input un'immagine con altezza e larghezza identiche e ogni volta che questa incontra uno dei layer convoluzionali della rete neurale, le dimensioni dell'immagine vengono ridotte, fino ad avere in output una matrice di dimensione (nel nostro caso) *13x13*. La cosa interessante è che ognuna delle celle della matrice possiede in realtà informazioni globali su tutta l'immagine in input. Questo rende possibile affidare ad ognuna delle celle della matrice la predizione di un singolo oggetto, più in particolare, ogni cella della matrice è responsabile di determinare una *Bounding Box* per l'oggetto che ha centro in essa, e contemporaneamente definire la classe dell'oggetto in questione. Oltre a questo, la cella deve anche esprimere un valore di *certezza*

rispetto alla *Bounding Box* predetta. Abbiamo quindi un output della forma *13x13x(4 + 1 + numero di classi)*, dove i primi quattro valori corrispondenti ad ogni cella servono a descrivere la *Bounding Box* predetta (centro espresso in x e y, *larghezza* e *altezza*), il quinto corrisponde al *Confidence Score*, ovvero la certezza che la *Bounding Box* predetta sia corretta, e i restanti valori rappresentano una *distribuzione di probabilità* delle classi predette. Nonostante ciò, prima di avere in output un risultato apprezzabile, è necessario compiere un passo di post-processing, ovvero la *Non Maximum Suppression* (*NMS*). Approfondiremo questo argomento al nella sezione **(6.9)**

3.2 Loss Function

$$\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2] \tag{3.1}$$

$$+ \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2] \tag{3.2}$$

$$+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} (C_i - \hat{C}_i)^2 + \lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{noobj} (C_i - \hat{C}_i)^2 \tag{3.3}$$

$$+ \sum_{i=0}^{S^2} \mathbb{1}_{i}^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2 \tag{3.4}$$

La **Loss Function** di *YOLO* (in questa tesi viene utilizzata principalmente la versione v2), ovvero la funzione utilizzata per calcolare l'errore tra la predizione e i valori di verità degli input durante la fase di *training*, è data da una somma di quattro funzioni, che vanno a pesare in modo differente i parametri di cui si occupano. Ovviamente questa è la funzione che il nostro modello cercherà di minimizzare durante il *training*.
La (3.1) rappresenta la distanza dei valori in x ed y predetti dal centro effettivo dell'oggetto (la distanza viene calcolata come somma della differenza tra le due coppie di valori x - $\hat{x}$ e y - $\hat{y}$, ciascuna elevata al quadrato per avere

in uscita un valore positivo). Questo viene ripetuto per ogni cella responsabile di una predizione, e la somma delle distanze di ogni cella corrisponde a questa parte della *Loss Function*.
La (3.2) è il corrispettivo della (3.1), calcolata però sulla *larghezza* e sulll'*altezza* della *Bounding Box* predetta, invece che sul centro. Prima di calcolare la distanza dei valori reali da quelli predetti, questi vengono passati sotto radice quadrata, per avere ordini di grandezza simili a quelli delle coordinate. Sia la (3.1) che la (3.2) presentano un fattore moltiplicativo (λ_{coord}, utile se vogliamo dare più o meno importanza alla predizione delle *Bounding Box* rispetto alla classificazione.
La (3.3) rappresenta la *Confidence Loss*, ovvero la distanza tra i *Confidence Score* predetti e quelli reali (andremo nel dettaglio nella sezione (6.8)). Questa viene divisa in due parti: la prima indica le *Confidence Loss* per le celle in cui si trova un oggetto, mentre la seconda parte è relativa alle celle vuote. Questa divisione serve a fare in modo da poter pesare in modo diverso gli errori sulla predizione del *Confidence Score* in base alla conformazione dell'input. Solitamente λ_{obj} è cinque volte superiore a λ_{noobj}, questo per penalizzare maggiormente le celle che dovrebbero contenere un oggetto.
L'ultima parte, ovvero la *Classification Loss* (3.4), è calcolata come *Cross Entropy* tra le classi predette e quelle reali. Se il valore massimo nella distribuzione di probabilità predette corrisponde alla classe dell'oggetto in questione per ogni cella contenente un oggetto, la *Classification Loss* tenderà a zero.

3.3 Versioni di YOLO

3.3.1 YOLOv1

E' la prima versione pubblicata di *YOLO*. Essendo ancora alle fasi iniziali, questa presenta diversi problemi. Sebbene l'idea di base è quella spiegata nella sezione precedente, i problemi di performance riscontrati sono molti. Innanzitutto, questa versione viene inizialmente creata per una rete neurale

(*Darknet*) composta solo da *layer convoluzionali*, inoltre, l'algoritmo è stato inizialmente testato su input di dimensione *224x224*, quindi immagini a bassa risoluzione. Questa decisione, però ha fatto sì che la rete neurale al termine del *training* fosse poco efficiente a individuare oggetti in in immagini con dimensioni molto differenti da quelle su cui è stata allenata. Inoltre, inizialmente la matrice in output aveva dimensione minore rispetto a quella delle successive versioni, rendendo così più difficile l'individuazione di oggetti piccoli nelle immagini.

3.3.2 YOLOv2

Il passaggio da *YOLOv1* a *YOLOv2* è stato quello di maggiore importanza nella storia dell'algoritmo. Progettato al fine di colmare le falle scoperte nella prima versione, *YOLOv2* presenta molte innovazioni sia per quanto riguarda la struttura della rete neurale su cui viene fatto eseguire, sia nella descrizione dell'algoritmo in sé. Alcuni dei cambiamenti più importanti sono:

- **Aumento della risoluzione dell'input**: le dimensioni delle immagini in input sono state raddoppiate, ciò per renderle più simili a quelle immagini che si potrebbe trovare davanti una volta finita la fase di training.
- **Batch Normalization**: ai layer convoluzionali viene applicata una funzione di *normalizzazione*, questa serve per scalare tutti i valori in output di ogni layer in modo tale da avere una media uguale a zero ed una deviazione standard di uno. Ciò permette alla rete di stabilizzarsi già da subito durante la fase di training intorno ai valori utili da prevedere.
- **Suddivisione dell'input in celle più piccole**: generando in output una matrice di *13x13* celle, il problema dell'individuazione di oggetti piccoli nelle immagini viene ampiamente risolto. Secondo l'autore, *13x13* è la giusta dimensione per avere un buon compromesso per

un'individuazione corretta sia di oggetti piccoli che di oggetti molto grandi.

- **Introduzione delle Anchor Box**: senza dubbio il cambiamento più sostanziale rispetto al primo modello. La rete neurale ora prevede non una, ma tre o più *Bounding Box* (a discrezione del progettista della rete), ognuna delle quali si omologa ad un certo *aspect ratio*, scelto facendo analisi statistiche sul *dataset*, prendendo gli *aspect ratio* mediamente più utilizzate all'interno di quel *dataset*. Questo permette di generalizzare le informazioni sugli oggetti nelle immagini. Ad esempio, sappiamo che di norma le persone nelle foto appaiono in piedi, quindi le *Bounding Box* che corrispondono a delle persone dovrebbero avere un *aspect ratio* maggiore di uno (dove l'*aspect ratio* è calcolato come *larghezza*/*altezza*). A questo punto, però, la rete neurale sarebbe in difficoltà a prevedere una *Bounding Box* corretta per una persona che appare distesa in un'immagine. Per questo l'introduzione delle *Anchor Box* ha avuto una notevole importanza al fine di migliorare *YOLO*.

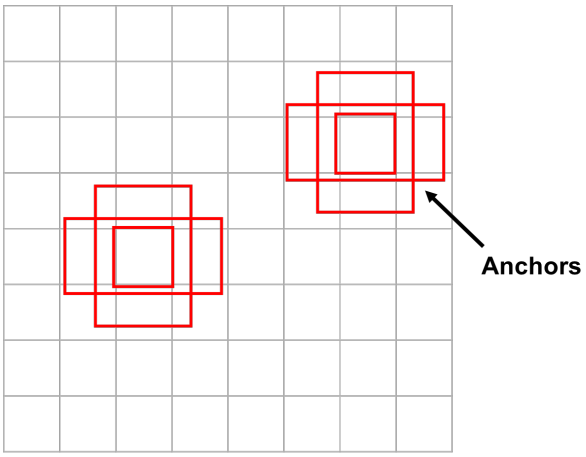

Figura 3.1: Anchor Box

Principalmente, è la versione dell'algoritmo usata per lo scopo di questa tesi.

3.3.3 YOLOv3

YOLOv3 presenta delle novità interessanti, soprattutto nella suddivisione dell'immagine. Vengono introdotte sostanzialmente tre cambiamenti:

- **Multi-Label Classification**: in questa versione dell'algoritmo, la parte relativa alla classificazione non si limita a calcolare una distribuzione della probabilità per ogni cella in output, bensì viene calcolata una *distribuzione logistica delle probabilità*, ovvero si cerca di dare ad ogni classe possibile un valore compreso tra *zero* ed *uno* che sta a indicare quanto l'oggetto in questione si avvicina a una delle classi disponibili. Supponiamo di avere a disposizione più razze di cani nella lista delle classi possibili. A questo punto, mostrando al nostro modello un'immagine contenente un cane che è un incrocio fra due delle classi a nostra disposizione, il nostro classificatore potrebbe etichettare quel cane con entrambe le razze, se questo presenta *features* riconducibili ad ambedue.

- **Objectness Score**: a differenza delle precedenti versioni dell'algoritmo, in *YOLOv3* uno dei campi da prevedere è l' *Objectness Score*, al posto del *Confidence Score*. In questo modo, non si cerca di dare una valutazione alla *Bounding Box* predetta, bensì alla certezza che la cella in questione contenga il centro di un oggetto. L'utilizzo di questo valore, unito a quelli derivati dalla classificazione, serve per scartare le *Bounding Box* che non rientrano sotto ad un certo *threshold*.

- **Feature Pyramid Networks** (**FPN**): *YOLOv3* restituisce in output non una ma ben tre matrici di dimensioni diverse. Questo è possibile grazie ad un *downscaling* dell'input di un fattore di 8, 16 e 32. Questo significa che avendo in entrata un'immagine di dimensione *416x416*, come output avremo in output tre matrici di features di dimensione *52x52*, *26x26* e *13x13*.

Questo concede al modello una maggiore precisione per quanto riguarda le predizioni di oggetti di grandezze diverse all'interno dell'immagine. Con una sola scala, studiata per avere un buon compromesso per quanto riguarda la predizione di oggetti sia piccoli che grandi, potremmo avere comunque problemi ad individuare oggetti *troppo piccoli* o *troppo grandi*.

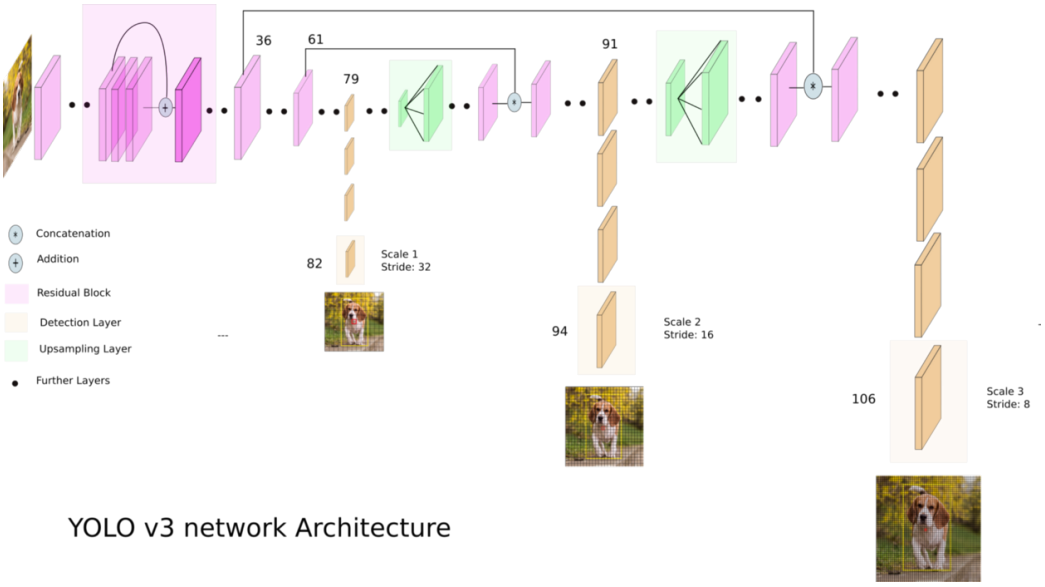

Figura 3.2: Feature Pyramyd Networks

Capitolo 4

COCO

4.1 Struttura del Dataset

Il dataset **COCO** (*Common Objects in COntext*) è principalmente composto da 123.287 immagini *labeled* ripartite in due blocchi: 118.287 immagini che appartengono al *training set*, e le restanti 5.000 appartenenti al *validation set* (ovvero immagini su cui viene testata la performance del modello dopo essere stato allenato sulle immagini del *training set*). Ognuna di queste immagini contiene uno o più oggetti appartenenti a 80 classi distinte. Inoltre, COCO mette a disposizione un file di annotazioni, contenente informazioni sulle immagini e sul loro contenuto, utili sia al fine dell' *Object Detection* che per la *Segmentazione*.

4.2 Struttura delle Annotazioni

I file contenenti le annotazioni di COCO sono leggibili tramite la API fornita dagli stessi autori. Questa permette di caricare, cercare e visualizzare le annotazioni di ognuna delle immagini etichettate del dataset. Grazie ai filtri offerti dalla **COCO API**, si possono ricercare immagini in base al contenuto, o avere una conoscenza su cosa c'è all'interno di ogni immagine. Le annotazioni sono state fondamentali per la costruzione della **Ground**

Truth su cui allenare il nostro modello, di cui parleremo nella prossima sezione.

4.3 Costruzione della Ground Truth

Le *annotazioni*, come già accennato prima, contengono molte informazioni per ogni singolo oggetto. Queste però devono essere rielaborate e trasformate in un formato utilizzabile dalla rete neurale. La creazione della *Ground Truth*, ovvero l'insieme dei valori di verità da confrontare con le predizioni del nostro modello, è un passo importante ai fini di una buona esecuzione. Innanzitutto, dato che a YOLO interessano le posizioni degli oggetti, le loro dimensioni e le loro classi, è importante estrapolare queste informazioni a partire da quelle presenti nelle annotazioni. Nelle annotazioni, tuttavia, i dati riguardanti la localizzazione sono espressi in coordinate del punto in alto a sinistra e coordinate del punto in basso a destra della *Bounding Box* che contiene l'oggetto. Da qui è semplice ricavare centro, altezza e larghezza delle *Bounding Box*, in modo tale da poter utilizzare questi valori senza bisogno di essere processati durante la fase di training, rallentandone ancora l'esecuzione. Questi valori inoltre devono essere scalati per rispecchiare le dimensioni dell'input del modello. Dopo queste elaborazioni, la nostra *Ground Truth* sarà composta da una matrice di 13x13x5 elementi, dove 13x13 corrisponde alla suddivisione in celle dell'input, mentre i restanti cinque canali stanno ad indicare in ordine: coordinata x del centro dell'oggetto, coordinata y del centro dell'oggetto, larghezza della *Bounding Box*, altezza della *Bounding Box*, classe dell'oggetto in questione rappresentata come numero compreso tra uno e il numero delle classi disponibili (per le celle non contenenti oggetti, ognuno dei valori lungo questi cinque canali sarà *zero*).

4.4 Studio statistico sul Dataset

Per un corretto utilizzo di YOLO, come già introdotto nel capitolo 3, c'è bisogno di svolgere un'indagine statistica sul dataset (su cui verrà fatto il *training*) per definire le *Anchor Box*, necessarie per aggiustare le predizioni del modello ed omologarle al dataset su cui viene allenato. Per fare questo, inizialmente sono state calcolate le medie degli *aspect ratio* e le *aree* di tutte le *Bounding box* contenenti gli oggetti del dataset. Vedendo però che i risultati ottenuti erano molto simili a quelli già documentati in altri articoli, abbiamo optato di utilizzare le *Anchor Box* introdotte dagli autori di YOLO in un articolo che parla della combinazione YOLO-COCO. Sono stati anche fatti degli studi sulla frequenza di ogni classe di oggetto appartenente al dataset, per ottenere dei pesi da applicare alla *Loss Function* per dare più importanza a elementi più ricorrenti, ma che per semplicità, abbiamo deciso di non implementare poi nella versione finale dell'algoritmo.

Capitolo 5

Google Colab

5.1 Cos'è Google Colab

Google Colab è una piattaforma messa a disposizione gratuitamente da *Google* in cui gli utenti possono connettersi ad un runtime ospitato dotato di servizi e *Hardware* (*Memoria secondaria*, *RAM* e acceleratori *Hardware* a scelta tra *CPU*, *GPU* e *TPU*) offerti da *Google*, su cui possono eseguire parti codice in forma di **Jupyter Notebook**. Avere un acceleratore *Hardware* è indispensabile ai fini di allenare un modello di *Machine Learning* in modo soddisfacente ed ottenere qualche risultato in tempi ridotti (parliamo comunque di giorni), e questa piattaforma è ottima per scopi didattici e per principianti. Inoltre, colab ha la possibilità di montare un profilo *Drive* (dello stesso utente *Google* connesso a *Colab*) e utilizzare i file lì presenti senza la necessità di doverli scaricare ad ogni sessione (per scaricare un dataset da 25GB circa come COCO impiegheremmo più di un'ora, mentre per importarlo da *Drive* bastano circa dieci minuti).

5.2 Vantaggi e Svantaggi

Nonostante i numerosi vantaggi già descritti nella sezione precedente, tra i quali l'utilizzo completamente gratuito di un servizio che avvicina tutti ciò

che hanno interesse nel campo del *Machine Learning* ai loro obiettivi, *Colab* presenta anche alcuni lati negativi, sebbene sia difficile farne a *Google* una colpa dato che l'offerta di un servizio aperto a tutti e completamente gratuito ha le sue limitazioni. Per garantire a tutti una porzione del servizio, *Colab* limita l'utilizzo di acceleratori *Hardware* a sessioni che possono durare al massimo 12 ore, permettendo così di riservare *GPU* e *TPU* a tutti gli utenti che ne abbiano bisogno. Finito il periodo di 12 ore, bisogna aspettare altrettanto prima di poter avere di nuovo acceso ad un acceleratore *Hardware*. Questo implica che le fasi di training debbano essere spezzate ed è bene salvare spesso i progressi ottenuti, in quanto non vorremmo perdere i progressi di dodici ore di lavoro. Oltre a questo, le *GPU* offerte da *Colab* sono poco sopra il minimo indispensabile per poter pensare di fare *Machine Learning*. Essendo un servizio gratuito, tuttavia, non ci si può aspettare di avere delle *GPU* top di gamma. La *RAM* offerta è di 12GB, anche questa carente per veri progetti di *Machine Learning*, ma abbastanza da supportare progetti di programmatori alle prime armi. Per quanto riguarda la memoria secondaria, tuttavia, ci sono dei problemi più ostici che hanno rallentato il progresso di questa tesi.

5.3 Problemi riscontrati

Come già accennato, il poco spazio disponibile (memoria secondaria) offerto da *Colab* (appena 35GB per una sessione con *GPU* o *TPU*) non è sufficiente per lavorare con dataset grandi come COCO, e questo porta con sé limitazioni considerevoli. Finché abbiamo lavorato in locale per fare piccole prove, accedevamo a COCO tramite una libreria di *TensorFlow* (ovvero **TensorFlow Datasets**), che contiene molti tra i dataset più famosi in un formato molto pratico e veloce da accedere tramite le API offerte appunto da *TensorFlow*. Nel momento del passaggio a *Colab*, l'utilizzo di questo *framework* è diventato impossibile, in quanto *TensorFlow Datasets* richiede il download e l'estrazione dei dataset in un solo passaggio, e la doppia copia

temporanea che si viene a creare supera i limiti di spazio dati da *Colab*, impedendo così l'accesso a COCO in questa modalità. Lo spazio gratuito offerto da *Drive*, inoltre, è di soli 15GB (inferiore alle dimensioni di COCO), quindi anche importarlo attraverso *Drive* è stato impossibile. L'unica soluzione trovata è stata quindi quella di richiedere l'espansione (a pagamento) di *Drive* per poter contenere un file in formato *zip* contenente le immagini di COCO, da estrarre in tempo reale durante l'utilizzo di *Colab*. Anche *Drive*, sebbene con archivio aumentato, ha dei limiti di download giornalieri, che consentono di importare files di dimensioni considerevoli (25GB) come COCO solo una volta al giorno. Se la sessione in cui si riesce a importare il dataset, inoltre, viene terminata per qualche motivo, non si può ovviare al problema se non aspettando circa 24 ore, perdendo così un giorno che potrebbe essere dedicato al *training*.

5.4 Alternative a Colab

Nonostante *Colab* presenti alcune restrizioni, è complessivamente molto utile per chi non ha accesso localmente ad un *Hardware* in grado di sostenere i ritmi necessari al *Machine Learning*. Ovviamente l'alternativa più banale sarebbe avere una *Workstation* con potenza di calcolo superiore a quelle di *Colab*, ma il costo può essere una limitazione troppo importante per chi volesse solamente sperimentare qualche tecnica di *Machine Learning*. In alternativa, è possibile utilizzare **Kaggle**, altro servizio gratuito che opera nella stessa modalità di *Colab*, e ha a disposizione già integrate molte librerie (e dataset). Anche questo però presenta limitazioni per quanto riguarda lo spazio in memoria secondaria, e a differenza di *Colab* è impossibile montarci sopra uno spazio di archiviazione esterno (come *Drive*), per questo non è stato utilizzato ai fini di questa tesi.

Capitolo 6

Lavoro svolto

6.1 Inizializzazione

Lo scopo di questa tesi è quello di ricostruire un *Object Detector* a la YOLO mediante **Keras**, con eventuale sperimentazione di tecniche aggiuntive, a partire da zero (o quasi). Quindi per prima cosa è stato eseguito uno studio su **Keras**, **Tensorflow**, **Numpy** e altri *frameworks* per **Python** adatti allo sviluppo di modelli di *Machine Learning*/*Deep Learning*. Dopo aver colto le basi necessarie ad un primo approccio all'argomento, sono stati svolti piccoli test per prendere confidenza con la terminologia e la metodologia di approccio a questi *framework*, quali l'importazione o la costruzione di un modello per *Machine Learning*, la scelta degli **Optimizer** e il loro **Learning Rate**, la visualizzazione delle diverse fasi necessarie a portare un semplice modello dal non saper fare niente a saper compiere un piccolo *task*. Ciò è stato molto utile per prendere dimestichezza con la manipolazione di strutture dati multidimensionali come **Tensori** e array in formato *Numpy*, comprendendo inoltre quali sono le funzioni delle relative API utili ai fini della tesi. Tra i test svolti possiamo citarne in particolare due: Il riconoscimento delle cifre scritte a mano utilizzando il dataset **MNIST** (*Modified National Institute of Standards and Technology*) ed un primo esercizio di classificazione che prevede la differenziazione di immagini di gatti da immagini di cani (*Dogs VS*

Cats dataset, disponibile anche su *Kaggle*), utile per un'introduzione all'uso del *Transfer Learning*. A questo punto, ottenute le conoscenze e tecnologie di base, è stato possibile procedere verso il reale scopo di questa tesi.

6.2 Setup per il Transfer Learning

Come già accennato nel primo capitolo, il **Transfer Learning** è una pratica che permette di utilizzare i pesi di un modello di *Machine Learning* già in grado di compiere un determinato *task* senza doverli riallenare per svolgere un *task* abbastanza simile, diminuendo radicalmente i tempi della computazione durante il *training*. Avendo scelto come obiettivo della tesi di sviluppare un *Object Detector* a la YOLO (algoritmo che, come già spiegato nel capitolo 3, è molto veloce ed adatto a un uso *Real Time*), abbiamo pensato di partire da un modello anch'esso veloce per poter eventualmente studiare il comportamento di questa coppia modello-algoritmo entrambi ottimizzati per la *Real Time Object Detection* (anche se c'è da menzionare che come primo approccio abbiamo optato per usare come modello **InceptionV3**, che spicca nei test relativi alla precisione ma è abbastanza mediocre per quanto riguarda la velocità di elaborazione). Attraverso le API di *Tensorflow 2*, è possibile importare tutti i modelli raccolti all'interno del loro database, in varie modalità. L'importazione in questa modalità, tuttavia, non si limita solamente alla struttura del modello di *Machine Learning*, ma comprende anche (in maniera opzionale), dei pesi ottenuti da un profondo allenamento per la classificazione per il dataset *Imagenet*. Essendo la classificazione uno dei nostri obiettivi, avere un modello con dei pesi a disposizione ci ha fornito un notevole speed-up, evitandoci così di dover fare imparare anche ai *layer* più profondi del modello ad estrarre features dalle immagini. Dopo aver quindi scelto di utilizzare **MobileNet** per i motivi già spiegati nel capitolo 2, c'è stato bisogno di andare ad operare sugli ultimi *layer* del modello per adattarlo al nostro *task*. Essendo di base un modello di classificazione, tutto ciò che *MobileNet* (la versione importata da *Tensorflow*) restituisce in output

è una distribuzione di probabilità per le classi di *Imagenet*, e per il nostro scopo, questo non è assolutamente sufficiente. C'è stato un primo tentativo di *mappare* questa distribuzione di probabilità per adattarla alle classi di COCO, ma dopo alcuni indagini svolte, abbiamo notato che i due dataset sfortunatamente hanno delle differenze troppo sostanziali per quanto riguarda la suddivisione in classi. Questo tentativo, se possibile, sarebbe stato in grado di risolvere quantomeno parzialmente il *Transfer Learning* da effettuare per la classificazione. Essendo ciò impossibile, però, abbiamo dovuto procedere in maniera canonica. Innanzitutto, è stato necessario eliminare l'ultimo layer del modello, ovvero quello di *softmax*, che distribuisce appunto i valori di probabilità delle classi. poi, abbiamo deciso di aggiungere due *layer convoluzionali* con *kernel* di dimensione 1x1, in modo da non stringere la matrice calcolata dalle *convoluzioni* precedenti e contemporaneamente ridurre il numero degli output lungo il canale delle classi di probabilità (nel primo di questi due *layer* vengono portati a 512 mentre nel secondo ad 81). Questa operazione rende possibile avere in output una matrice quadrata, dove in ogni cella ci sono 81 canali che indicano tramite una distribuzione di valori, maggiori quanto più la classe dell'oggetto contenuto in essa presenta features simili a quelle della classe predetta. E' importante ricordare che ognuna delle celle ha informazioni derivanti da tutta l'immagine iniziale, e non solamente relative alla zona di loro interesse. questo rende possibile l'individuazione di oggetti da parte di una singola cella di oggetti di dimensione molto superiore ad essa. A questo punto, abbiamo quindi un modello già allenato a classificare, che prende in input un'immagine e grazie ai suoi *layer convoluzionali* ne schiaccia progressivamente la dimensione (di un fattore di 32) mantenendone però le informazioni importanti, fino ad avere come output una matrice quadrata di dimensione $Dim/32$ (con Dim indichiamo larghezza e altezza delle immagini che il modello prende in input, nel nostro caso Dim è 416). Avendo ora un modello teoricamente adatto alla multi-classificazione, ci rimane solamente da fare in modo che solo gli ultimi *layer* da noi aggiunti vengano allenati, mentre la parte base del modello con i pesi di *Imagenet*

la vogliamo lasciare così, fidandoci della capacità di estrazione di features già ottenuta dal precedente allenamento. Per fare questo, *Keras* offre delle funzioni per andare ad operare sui singoli layers del modello, rendendo così possibile "*congelare*" tutti i *layer* sottostanti a quelli da noi aggiunti, in modo tale da minimizzare la **Loss Function** operando solo sugli ultimi due *layer*, e riducendo il numero di parametri da allenare da 3,795,217 a 566,353.

Total params	3.795.217
Trainable params	566.353
Non-trainable params	3.228.864

Tabella 6.1: Risultati della funzione summary() sul modello per Transfer Learning

6.3 Preprocessing del Dataset

Siccome l'utilizzo di YOLO richiede per un funzionamento bilanciato la suddivisione in 169 celle (13x13) e sappiamo che il modello ottenuto finora riduce le dimensioni iniziali di un fattore 32, è facile calcolare quanto debbano essere le dimensioni delle immagini da passare in input, ovvero *Dim* = 32 * 13 = 416. Quindi sappiamo che per avere output omogenei dal modello sarebbe indispensabile avere anche input omogenei. Questo ci porta a dover scalare tutte le immagini di COCO alla dimensione richiesta, ovvero 416x416. Fatto ciò, per quanto riguarda la fase di **Preprocessing**, manca da normalizzare i canali dei colori (il formato RGB prevede 3 canali che possono assumere valori da 0 a 255, in modo da descrivere come combinazioni di questi il colore dei singoli *pixel*), ovvero dividere per 255 tutti i valori relativi ai colori per evitare il **biasing**. La mancanza di questo passaggio potrebbe privare la possibilità di riconoscere oggetti illuminati da luci di colori diversi.

Ad esempio, se allenassimo un modello su un dataset contenente solo foto scattate di giorno senza normalizzare i colori, questo difficilmente riuscirà a prevedere correttamente su foto scattate di notte, che hanno colorazioni completamente diverse. La normalizzazione, inoltre, serve ad evitare che il modello si specializzi a pesare maggiormente gli errori che riguardano immagini nella tonalità di colore più frequente nel dataset di allenamento. Dopo aver svolto questi passaggi e aver costruito una **Ground Truth** correttamente (vedi sezione 4.3), abbiamo potuto iniziare ad allenare il nostro modello sulla classificazione di COCO.

6.4 Dalla classificazione all'Object Detection

La ristrutturazione fatta a *MobileNet*, come spiegato nella sezione precedente, ci da la struttura adatta per allenare il nostro modello aggiornato per il *task* della classificazione. Volendo, avremmo potuto direttamente svolgere un *setup* tale da poter allenare in un colpo solo il modello sia per il *task* della classificazione che quello dell'individuazione, ma dato che la tesi si basa sulla sperimentazione di tecniche di *Machine Learning* abbiamo optato per costruire l' *Object Detector* in modo sequenziale, cercando di capire come fosse possibile suddividere le fasi di lavoro in quante più sezioni a sé stanti possibili. L'obiettivo di questa fase era quello di capire in primo luogo se il modello fosse riuscito a svolgere il *Transfer Learning* in tempi considerevoli per adattarsi a COCO, e in secondo luogo se a questo punto fosse facile recuperare i pesi del modello per integrare funzionalità aggiuntive per svolgere l'altro *task*. Dopo aver scritto la *Loss Function* per il task della classificazione (che da ora in poi chiameremo **Classification Loss**, vedi sottosezione 6.7.1), abbiamo provato ad allenare il nostro modello per qualche *epoca* (unità che corrisponde alla visione di una sola volta di tutte le immagini del *training set*), abbiamo iniziato a vedere risultati soddisfacenti, e dopo appena 11 *epoche* di *training*, il nostro modello ha raggiunto una precisione superiore al 50% sul *validation set* di COCO (il *validation set* di COCO è composto da

5000 immagini, diverse da quelle usate per il *training*, etichettate ed utilizzate al fine di valutare le prestazioni del modello dopo l'allenamento). Dalla dodicesima epoca in poi la *Classification Loss* ha iniziato a stabilizzarsi, migliorando di poco le prestazioni del modello sulla classificazione. Abbiamo quindi deciso di fermare il *training* alla dodicesima epoca, ottenendo così i seguenti risultati:

Top_1	Top_2	Top_3	Top_4	Top_5
54,7%	69%	75%	80%	83%

Tabella 6.2: Risultati della classificazione sul Validation Set (Top_x indica il confronto tra le classi annotate e le prime x classi in ordine di probabilità predette dal modello.)

6.5 Rimodellazione e recupero dei pesi

A questo punto, avendo un modello in grado di svolgere la classificazione per COCO, il prossimo step da eseguire è stato quello di cambiare la forma dell'output del modello, aggiungendo più parametri che prevedano dimensione e posizione delle *Bounding Box* e del *Confidence Score*. Inoltre, abbiamo deciso di predire tre *Bounding Box* per ogni oggetto, ognuna delle quali viene modellata secondo le *Anchor Box* calcolate nelle fasi precedenti. In primo momento, questo non è stato così scontato, dato che le API di *TensorFlow* non forniscono mezzi diretti per operazioni di questo tipo, però, dopo una profonda analisi delle strutture date che contengono i pesi dei vari *layer* del modello, siamo riusciti a capire la loro distribuzione ed estrapolarli. Questo ci ha permesso di creare un nuovo modello che si adattasse alle esigenze che avevamo, ovvero l'aggiunta di quei nuovi parametri e la triforcazione della predizione finale. Avendo creato questo nuovo modello con i pesi *randomizzati*, abbiamo potuto a questo punto performare l'operazione inversa svolta per estrarre i pesi dal modello per la classificazione, e sovrascrivere i pesi del nuovo modello (soltanto nei campi corrispondenti ai valori già allenati), con

quelli del vecchio modello. Così facendo, abbiamo ottenuto un nuovo modello adatto all' *Object Detection*, che è già allenato a classificare, ma ancora non sa individuare. Il modello ora da in output un array di questa forma: 13x13x256, dove i 256 (0-255) canali rappresentano i dati in questo modo:

- 0 = x del centro dell'oggetto
- 1 = y del centro dell'oggetto
- 2 = larghezza della Bounding Box
- 3 = altezza della Bounding Box
- 4 = Confidence Score
- 5 ... 85 = Classi di COCO
- 86 ... 177 = 0 ... 85
- 174 ... 255 = 0 ... 85

Data la ripetizione di questi dati lungo lo stesso asse, è stato aggiunto al modello un *layer* di tipo **Reshape**, che non contiene parametri allenabili ma serve semplicemente a rimodellare l'output del modello, rendendolo così di forma **13x13x3x86**.

6.6 Separazione degli Output del modello

Dopo un primo tentativo di *training* non funzionante, abbiamo deciso di separare le due funzioni principali del nostro *Object Detector*. Grazie alle API di *TensorFlow*, è possibile avere dei modelli biforcati, che condividono una parte dei *layer* e da un certo punto in poi si diramano in maniera differente. Così abbiamo deciso di mantenere come corpo condiviso quello originale di *MobileNet*, mentre per i due *task* costruire due blocchi differenti in modo da poter esaminare i problemi di uno e dell'altro. Il ramo della classificazione

è rimasto tale e quale a quello iniziale, non avendo bisogno di prevedere tre volte la classe per ognuna delle celle. Per la parte di individuazione, invece, abbiamo mantenuto la triplice predizione, ma questa volta solo con i cinque parametri utili alla predizione delle *Bounding Box*. Quindi per ogni immagine il modello restituirà due output:

1. Il primo, per la **classificazione**, di forma 13x13x81;
2. Il secondo, per l' **individuazione** di forma 13x13x3x5.

Nella seconda parte, inoltre, è stata aggiunta una funzione di attivazione (**ReLU**), per schiacciare i valori ottenuti fino a quel punto tra zero e uno e spingere il modello a convergere verso i valori giusti più velocemente.

MobileNet (versione Tensorflow senza ultimo layer, output: 13x13x1024)	
Layer Convolutivo, kernel (1x1), output: 13x13x512	Layer Convolutivo, kernel (1x1), output: 13x13x512, funzione di attivazione: ReLU
Layer Convolutivo, kernel (1x1), output: 13x13x81	Layer Convolutivo, kernel (1x1), output: 13x13x256
	Layer Convolutivo, kernel (1x1), output: 13x13x15
	Layer Reshape, output: 13x13x3x5

MobileNet Model Classification Model Detection Model

Tabella 6.3: Modello finale

Questa separazione, inoltre, rende possibile allenare una delle due parti del modello singolarmente, rendendo così possibile concentrarsi sull'individuazione dato che il nostro modello a questo punto era già in grado di classificare in modo soddisfacente.

6.7 Custom Loss Function

Abbiamo già introdotto il concetto di **Loss Function** e in specifico la Loss Function caratteristica di YOLO nella sezione **(3.2)**. Le *Loss Function* in *TensorFlow*, generalmente restituiscono un numero, che indica la magnitudine dell'errore, e che deve essere *minimizzato*. Tramite il grafo delle

operazioni, l' *Optimizer* decide poi quali pesi hanno avuto poca importanza nell'errore e quali invece hanno sbagliato di molto, andando a correggere quindi solamente ciò che è diretto interessato dell'errore. Per questo quando si confrontano due strutture con forme diverse, è necessario utilizzare una serie di funzioni mirate, in modo da non appesantire il calcolo e fare in modo di capire con esattezza da dove deriva l'errore, per poter migliorare la prossima predizione. Per questo è buona regola utilizzare funzioni fornite dalle API di *TensorFlow* o *Keras* in questi casi. Ciò di cui andremo a occuparci in questo capitolo è l'analisi passo a passo delle varie parti di questa funzione e delle differenze presentate tra quella utilizzata e quella proposta in precedenza.

6.7.1 Classification

La **Classification Loss**, come si può intuire, serve per allenare i parametri del modello responsabili del *task* della classificazione. L'idea generale è quella di far prevedere al modello cosa c'è in ognuna delle 13x13 celle della matrice in output, e poi, confrontando questo risultato con la *Ground Truth* precedentemente costruita, vedere quanto siamo vicini ai valori di verità. Per performare questa operazione, bisogna innanzitutto descrivere i due parametri della funzione, che saranno **y-true** (ovvero la matrice dei valori reali dell'immagine in esame estrapolata dalla Ground Truth) e **y-pred** (nientemeno che l'output del modello, che in questo caso ovviamente si riferisce solo al ramo relativo alla classificazione). Come già detto, *y-true* è una matrice 13x13, dove ogni elemento rappresenta una parte dell'immagine, e più precisamente corrisponde a zero se la cella in questione non contiene il centro di un oggetto, mentre sarà un valore nel range $1 \dots 81$ (ricordiamo che le classi annotate in COCO sono 80) che indicherà che tipo di oggetto ha centro in quella cella, se questa ne contiene uno. Invece *y-pred*, sarà una matrice 13x13x81, dove gli 81 canali rappresentano per ogni classe disponibile, un valore che, idealmente, dovrebbe essere massimo per il canale che corrisponde all'indice della classe dell'oggetto e basso per gli altri indici. A questo punto, l'operazione da fare sarebbe quella di confrontare *y-true* con *y-pred*, ma ancora ci sono alcune

altre premesse da fare. Sia *TensoFlow* che *Keras* offrono un'ampia gamma di funzioni che servono in situazioni come questa, come **sparse-categorical-crossentropy** di *Keras* o **sparse-softmax-cross-entropy-with-logits** del backend di *TensorFlow*. Le due funzioni sono molto simili, però quella di *Tensorflow* performa un'operazione addizionale che è molto utile nel nostro caso. Dato che nei *layer* finali del ramo della classificazione non è stata messa nessuna funzione di attivazione, possiamo emulare quest'ultima applicando la *Softmax Cross Entropy* di *TensorFlow*, che per prima cosa, schiaccia i valori degli 81 canali delle predizioni in una distribuzione di probabilità, portandoli tra zero ed uno. Da questo in poi le due funzioni performano in modo simile, ovvero cercano l'indice del canale che contiene il valore massimo tra gli 81 canali per ogni cella, ottenendo così una matrice del tutto omogenea a quella dei valori di verità, e calcolano la distanza tra le due. Questo restituisce una matrice che rappresenta l'errore nelle predizioni di ogni cella (che da ora in poi chiameremo **cross-res**), ma ancora necessita di un ulteriore passo prima di essere confrontata con *y-true*. Dato che *y-true* ha valori diversi da zero soltanto nelle celle contenenti i centri degli oggetti e non su tutta l'area dell'immagine, i valori da confrontare tra *y-true* e *cross-res* sono solamente quelli che in *y-true* sono diversi da zero. Per fare ciò, basta costruire una matrice 13x13 composta soltanto da 1, e passarla ad una funzione di minimo assieme ad *y-true*. Questo in sostanza ci restituisce una sorta di **matrice booleana** 13x13 che indica con **1** le celle contenenti i centri degli elementi e con **0** le celle non interessanti ai fini del *training*. A questo punto ci basta moltiplicare la matrice booleana per *cross-res* elemento per elemento in modo da ottenere la matrice dell'errore relativo solamente alle celle interessanti. Come ultima cosa, Sommiamo tutti i valori della matrice e dividiamo il risultato per il numero degli oggetti annotati nell'immagine (in modo da pesare equamente immagini con un solo oggetto e immagini con molti oggetti), e questo risultato ci darà il valore da ritornare alla fine della *Classification Loss*, ovvero il valore che il nostro modello cercherà di minimizzare durante il training.

6.7.2 Detection

La **Detection Loss**, ovvero la parte della *Loss Function* di YOLO che si occupa della predizione delle *Bounding Box*, è composta da più sottofunzioni, ed ognuna di queste opera su certi parametri responsabili di predizioni diverse, anche in ordini di grandezza. Le due grandi parti in cui si può dividere sono la **Localization Loss**, responsabile della predizione della corretta posizione e dimensione delle *Bounding Box*, e la **Confidence Loss**, responsabile di dare un punteggio ad ogni *Bounding Box* predetta, per capire quale di queste secondo il modello possano essere veramente utilizzate al fine dell'individuazione, ovvero per scegliere solamente le *Box* che hanno un punteggio sopra un certo *threshold* e scartando le altre.

Localization Loss

Questa parte della *Detection Loss* , serve per analizzare le predizioni di centro, larghezza e altezza di ogni singola *Bounding Box* predetta. La versione proposta da YOLO è particolare, in quanto le predizioni sia per le coordinate che per le dimensioni delle *Bounding Box* devono essere processate prima di essere confrontate con la *Ground Truth*. Per quanto riguarda le coordinate, invece di calcolare un centro assoluto per ogni cella dell'immagine, viene utilizzato un **centroide**, ovvero un valore compreso tra 0 ed 1 che indica il punto di interesse.

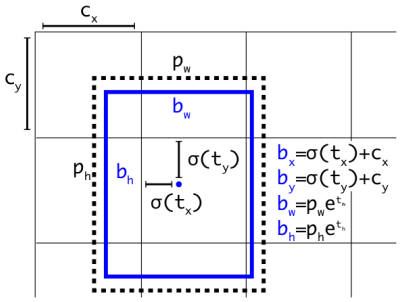

Figura 6.1: Centroidi

Ad esempio, se abbiamo un'immagine di dimensione 416x416, e la matrice di output è della forma 13x13, il centro della prima cella è rappresentato dalle coordinate (0.5,0.5) mentre il centro dell'ultima cella viene rappresentato da (12.5,12.5). Questo fa in modo di ridurre l'errore relativo ad ogni cella per quanto riguarda le coordinate ad un massimo di uno. Per fare ciò, applichiamo una funzione **sigmoid** ai canali responsabili delle predizioni delle coordinate, e per ogni cella vi sommiamo l'intero che corrisponde alla posizione della cella all'interno della matrice. Nella tabella sottostante viene mostrato un esempio della matrice delle coordinate che indica i centri di tutte le celle.

X / Y	X_1	X_2	...	X_{12}	X_{13}
Y_1	(0.5, 0.5)	(1.5, 0.5)	...	(11.5, 0.5)	(12.5, 0.5)
Y_2	(0.5, 1.5)	(1.5, 1.5)	...	(11.5, 1.5)	(12.5, 1.5)
.	.	.	.	.	.
.	.	.	.	.	.
.	.	.	.	.	.
Y_{12}	(0.5, 11.5)	(1.5, 11.5)	...	(11.5, 11.5)	(12.5, 11.5)
Y_{13}	(0.5, 12.5)	(1.5, 12.5)	...	(11.5, 12.5)	(12.5, 12.5)

Tabella 6.4: Centri delle celle della matrice delle coordinate

A questo punto è lecito ricordare che le matrici responsabili delle coordinate dei centri degli oggetti nella *Ground Truth* corrispondono a valori assoluti (quindi compresi tra 0 e 416), mentre la nostra matrice delle predizioni ha valori che spaziano nel range [0 ... 13), per questo prima di confrontare le due è necessario scalare i valori della *Ground Truth* per il fattore di riduzione della dimensione dell'immagine del nostro modello. In altre parole, tutti

i valori della *Ground Truth* vengono divisi per 32 (nel nostro caso) in fase di *training*. Questo sarebbe potuto essere implementato a priori durante la costruzione della *Ground Truth*, ma abbiamo deciso di lasciarla con valori assoluti perché nel caso avessimo portato YOLO dalla versione 2 alla versione 3, avremmo avuto 3 diverse scale su cui calcolare la *Loss*, e quindi la cosa migliore sarebbe stata normalizzare la *Ground Truth* a *run-time* in base alla scala di riduzione che stavamo calcolando. Ora, ci rimane soltanto da calcolare la differenza tra la matrice delle predizioni e quella della *Ground Truth* normalizzata. Fatto ciò, è necessario elevare al quadrato i valori ottenuti in modo da renderli positivi e non commettere errori logici durante la scrittura della funzione. Come nel caso della *Classification Loss*, bisogna creare la matrice booleana degli oggetti e moltiplicarla per la matrice degli errori delle coordinate, dopodichè, sommiamo tutti gli elementi (e poi dividiamo per il numero degli elementi presenti nell'immagine). Questo risultato è la *Loss* relativa al centro degli oggetti (da ora in poi sarà **xy-loss**).
Per quanto riguarda la **wh-loss** (*Loss* relativa a larghezza ed altezza delle *Bounding Box*), invece, operiamo in un modo leggermente diverso. Innanzitutto, invece che schiacciare i valori tra 0 e 1 come per la *xy-loss*, applichiamo un esponenziale alle predizioni, in modo che queste assumano valore positivo (nella realtà non possiamo avere un rettangolo con dimensioni negative). Fatto ciò, intervengono le **Anchor Box** definite in precedenza. Le *Anchor Box* che abbiamo scelto sono 3 ([3.625, 2.8125],[4.875, 6.1875],[11.65625, 10.1875]), che rappresentano le dimensioni più utilizzate in COCO per le *Bounding Box*, scalate anch'esse di un valore di 32, per omologarle a questa suddivisione dell'immagine. Dato che il modello predice valori per 3 *Box* singolarmente, moltiplichiamo quei valori per le nostre *Anchor*, in modo da avere tre predizioni in linea con le *Bounding Box* più frequenti in COCO. La parte restante della *wh-loss* è analoga a quella della *xy-loss*, ad eccezione del fatto che prima di calcolare la differenza tra le matrici dei valori reali e quelle predette ed elaborate, si passano entrambe sotto radice quadrata, in modo da scalare le dimensioni di questi valori con quelli della *xy-loss*, in modo tale da non dare

più peso ad una parte della *Loss Function* rispetto che all'altra. La somma ottenuta dalla *xy-loss* e dalla *wh-loss* ci indicano ciò che stiamo definendo come **Detection Loss**.

Confidence Loss

Infine, l'ultima parte rimanente da aggiungere alla *Loss Function* è la **Confidence Loss**, indispensabile per scegliere come filtrare le *Bounding Box* una volta predette. Ricordiamo che il modello restituisce tre *Bounding Box* per ogni cella, indipendentemente dalla presenza o meno di un oggetto in essa. Sta a noi introdurre questo ulteriore parametro per fare in modo di selezionare solo le *Bounding Box* interessanti e scartare tutte le altre. La *Confidence Loss* si specializza nella predizione di un valore tra 0 ed 1 (il *Confidence Score*, ottenuto tramite *sigmoid* del valore predetto), che indica quanto la *Bounding Box* predetta coincide con quella reale. Durante la fase di *training*, quindi, dobbiamo insegnare al modello a riconoscere quali delle *Bounding Box* da esso stesso predette sono buone e quali invece sono da scartare. Per fare questo, durante la fase di *training* calcoliamo i valori di **IoU** (**Intersection over Union**) tra le *Bounding Box* predette e quelle reali. Così facendo, abbiamo un valore (tra 0 e 1) che indica quanto le *Bounding Box* predette si sovrappongano a quelle reali, ed è proprio questo il valore che andiamo ad utilizzare per ogni cella come *Ground Truth*. In altre parole, idealmente il *Confidence Score* deve essere più vicino possibile ai valori di IoU, o per meglio dire, la loro differenza deve tendere a zero. A differenza degli altri casi, dove andavamo a penalizzare nelle varie *Loss* soltanto i valori per cui avevamo delle annotazioni, questa volta è necessario penalizzare il nostro modello quando predice *Confidence Score* alti per celle che in verità non sono responsabili di nessun oggetto. Quindi possiamo dire che la *Confidence Loss* finale sarà data dalla somma della **obj-loss** (differenza calcolata tra IoU e *Confidence Score* predetti per le celle in cui è presente un oggetto) e **noobj-loss** (stessa cosa ma per celle vuote). La separazione dei due valori di *Loss* ci permette di poter pesare in modo diverso le due parti, e quindi

andare a penalizzare maggiormente il modello se non vede oggetti esistenti rispetto a quando vede oggetti che (almeno secondo che ha redatto le annotazioni del dataset usato per il *training*) in verità non ci sono. A questo punto non ci resta che sommare tutti i valori ottenuti dalle precedenti parti (ovvero *Localization Loss* e *Confidence Loss*) per ottenere la **Detection Loss** voluta inizialmente.

6.8 Confidence vs Objectness

In un primo momento, il *training* sul *task* della *Detection* ha riportato dei problemi per quanto riguarda la predizione del **Confidence Score**. Per questo abbiamo optato di cambiare quella parte dell'algoritmo utilizzando le specifiche di YOLOv3, ovvero provare a predire l' **Objectness Score** invece del **Confidence Score** come parametro per scegliere le *Bounding Box* da scartare. La predizione dell' **Objectness Score** consiste nell'insegnare al modello a riconoscere quali celle contengono il centro di qualche oggetto, e combinando questo risultato con la predizione della classe, riusciamo a mantenere solamente le *Bounding Box* che presentano un **Objectness Score** e una certezza di classe sopra un certo *threshold*. In un primo momento, questa soluzione sembrava funzionare, ma contemporaneamente siamo riusciti a risolvere il problema riguardante il **Confidence Score**, scegliendo così di rimanere fedeli alla struttura di YOLOv2, lasciando aperta come opzione futura un approfondimento sul passaggio a YOLOv3.

6.9 Postprocessing

La fase finale di questa tesi prevede l'elaborazione dell'output del modello per poter recuperare e modellare i risultati della predizione, filtrarli ed utilizzarli per disegnare le *Bounding Box* nelle immagini iniziali. La fase di **Postprocessing** si divide in 4 parti principali, spiegate nelle prossime sezioni.

6.9.1 Selezione iniziale

La prima selezione da fare è quella di scegliere per ogni cella la *Bounding Box* delle tre predette con *Confidence Score* maggiore, avendo così una sola *Bounding Box* per cella della matrice di output (13x13). In questo passaggio, inoltre abbiamo deciso di filtrare anche tramite lo **Score** della **Classificazione** visto che per il tempo ristretto non abbiamo potuto ottenere risultati eccellenti per quanto riguarda l'esattezza del *Confidence Score*, avendo così un filtro addizionale per eliminare le *Bounding Box* sbagliate.

6.9.2 Non Maximum Suppression

A questo punto, l'ultimo filtro rimanente da applicare alle *Bounding Box* rimaste è la **NMS** (**Non Maximum Suppression**). Questa procedura consiste di alcuni passi iterativi che eliminano le *Bounding Box* che presentano un certo valore di sovrapposizione. E' possibile che il modello preveda per due o più celle vicine *Bounding Box* molto simili riferite allo stesso oggetto, per questo è necessario selezionare soltanto la migliore delle due ed eliminare l'altra. Questo algoritmo procede nel modo seguente:

1. Si seleziona la *Bounding Box* con *Confidence Score* più alto;

2. Si calcola la **IoU** tra quella *Bounding Box* e tutte le altre rimaste;

3. Si aggiunge la *Bounding Box* selezionata alla lista delle *Bounding Box* da tenere e si eliminano tutte le quelle con IoU maggiore di un certo *threshold*. Questo perché se la IoU è maggiore della nostra soglia, le *Bounding Box* coinvolte si riferiscono molto probabilmente allo stesso oggetto.

4. Si ripete il processo finché la lista iniziale delle *Bounding Box* non viene svuotata.

6.9.3 Ridimensionamento dell'output della rete

Giunti qui, abbiamo solamente le *Bounding Box* da disegnare, quindi dobbiamo scalarle per la dimensione originale dell'immagine da cui sono state calcolate (ricordiamo che le immagini date in input al modello sono ridimensionate prima di essere rielaborate, questo passo serve a fare il procedimento inverso e riportarle alla dimensione originale). I parametri da scalare sono i 4 responsabili della posizione e dimensione delle *Bounding Box*, ovvero coordinate del centro, larghezza ed altezza. Inoltre, ricordiamo che le predizioni venivano calcolate su una *Ground Truth* con valori corrispondenti a $\frac{1}{32}$ delle dimensioni reali, quindi dobbiamo moltiplicare i risultati delle predizioni per **32** per avere le dimensioni reali delle *Bounding Box*.

6.9.4 Disegno delle Bounding Box

Infine, abbiamo ottenuto le *Bounding Box* corrette da disegnare sull'immagine di partenza. L'ultima cosa rimasta da fare è appunto passare le *Bounding Box*, l'immagine iniziale e i nomi delle classi di COCO ad una funzione che disegnerà le *Bounding Box* nell'immagine e sopra ad ognuna di esse scriverà qual'è la classe dell'oggetto rilevato. Per fare ciò sono state utilizzate delle funzioni della libreria **cv2** (**OpenCV**), che contiene molte funzioni per disegnare poligoni e molto altro ancora sulle immagini.

Risultati

Dopo la costruzione dell'*Object Detector* e le varie prove fatte per verificarne la correttezza, siamo riusciti ad allenare il nostro modello su COCO per **12 epoche** dedicate alla **classificazione** e **33 epoche** dedicate all'**individuazione**, entrambe in modalità **Transfer Learning** (abbiamo allenato quindi solo i *layer* aggiunti da noi). I risultati riguardanti la parte della Classificazione sono già stati esposti nella sezione **6.4** (Tabella **6.2**) e riscontrano una precisione sul *Validation Set* **superiore al 50%**. Per la parte dell'Individuazione non abbiamo effettuato un test vero e proprio sui risultati in quanto il *training* per questioni di tempo è stato interrotto alla 33esima epoca, raggiungendo così risultati che ci portano a credere che continuando ad allenare il modello, questo migliorerà in modo notevole, ma che attualmente sono molto lontani da un funzionamento ideale. Inoltre, dato l'allenamento parziale effettuato, i parametri finali per il filtraggio delle *Bounding Box* (**Confidence Score**, **Classification Score** e **NMS Threshold**) sono molto instabili e basta una minima variazione per avere risultati completamente diversi. Comunque, siamo riusciti a trovare dei valori per i tre parametri che ci danno risultati apprezzabili sul *Validation Set*, valori che però con un maggiore quantità di *training* dovranno essere sicuramente riconfigurati. Riportiamo di seguito alcuni esempi di *Object Detection* positivi e altri fallimentari, per avere un'idea globale sul risultato ottenuto.

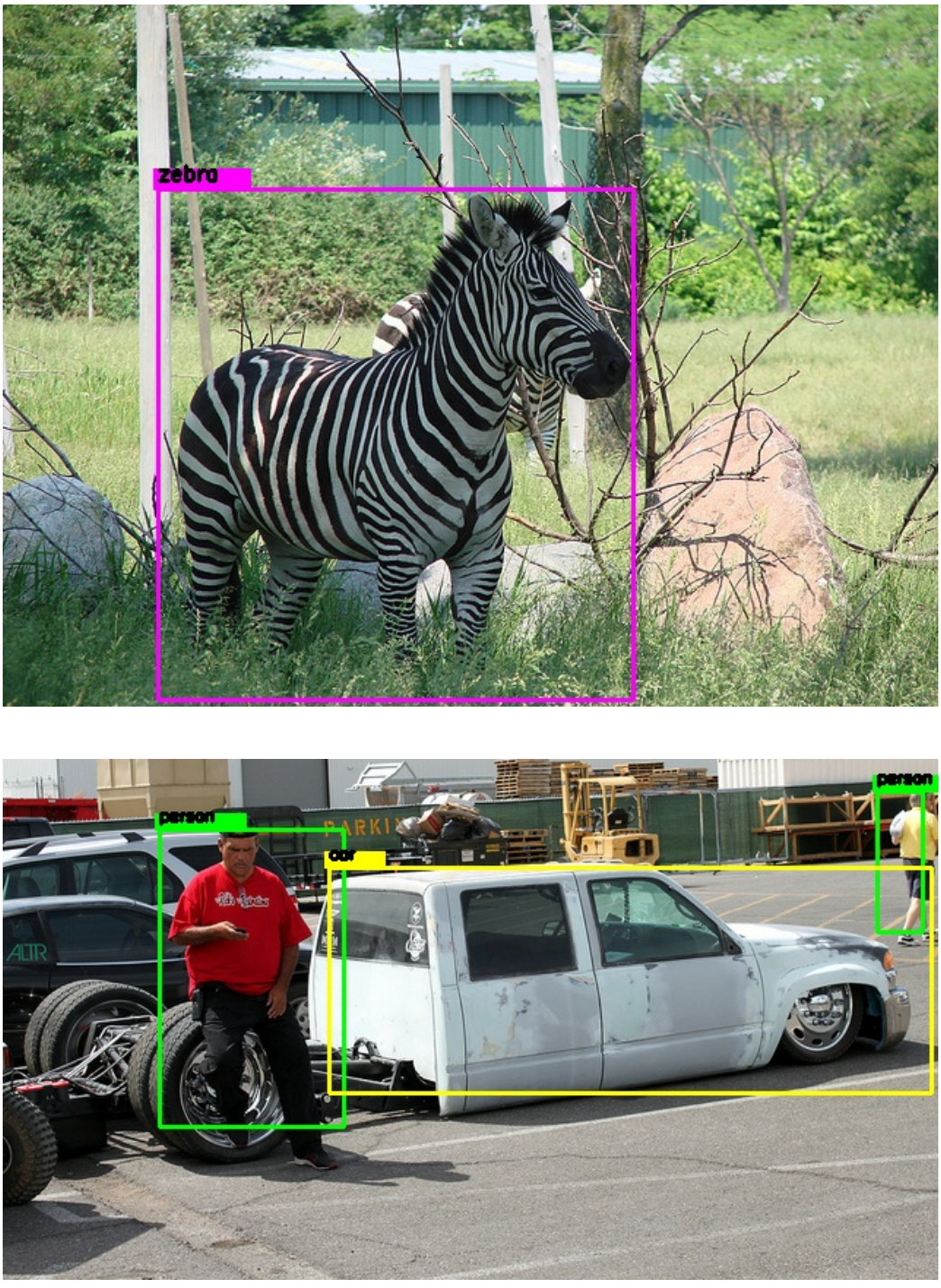

Figura 6.2: Object Detection corretta (Validation Set di COCO)
Classi e Bounding Box esatte

Figura 6.3: Object Detection quasi corretta (Validation Set di COCO)
Classi esatte, Bounding Box sbagliate di poco

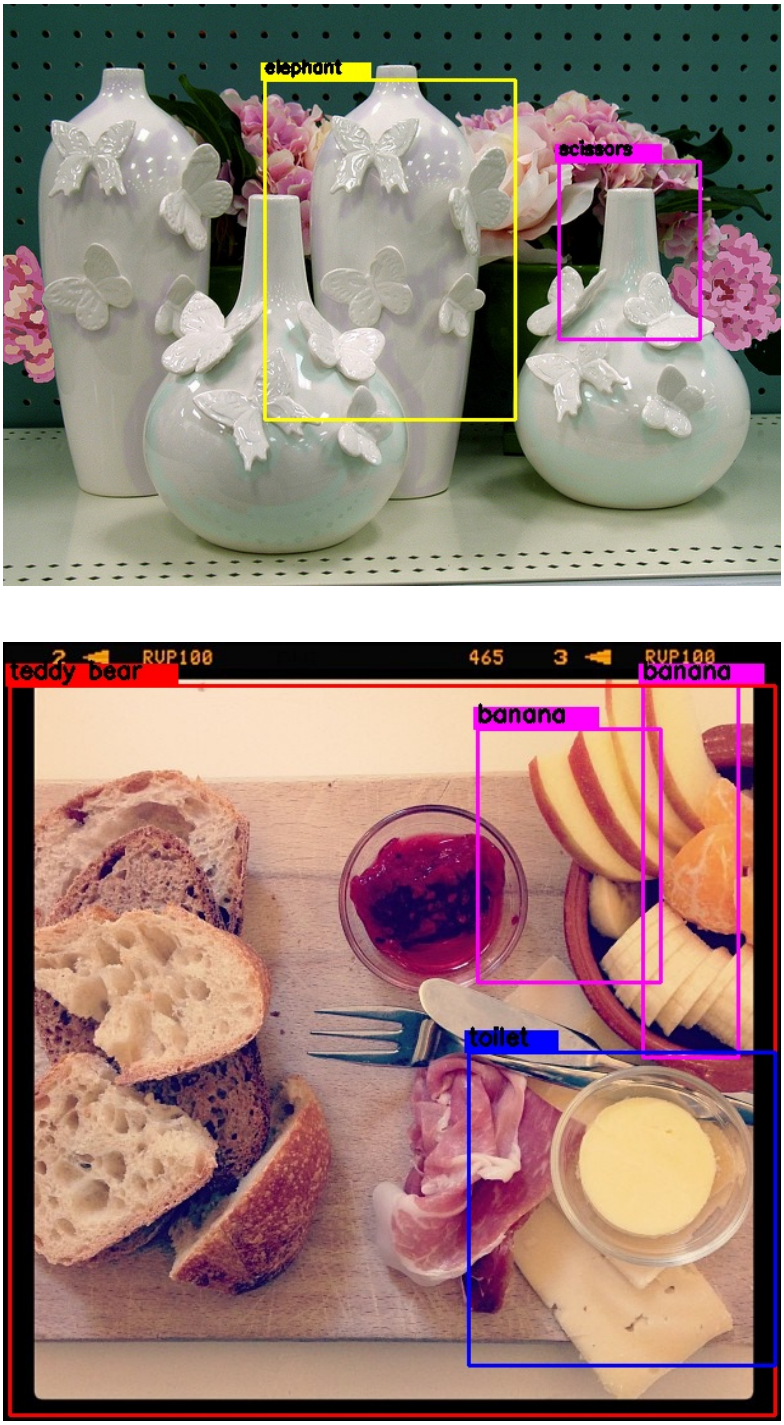

Figura 6.4: Object Detection sbagliata (Validation Set di COCO)
Classi e Bounding Box sbagliate

Conclusioni

La realizzazione di questa tesi ci ha portato a testare delle tecniche di *Object Detection* molto interessanti dal punto di vista didattico. Solitamente, per realizzare un *Object Detector* a partire da zero ci vuole molto tempo, sia per quanto riguarda la struttura generale, sia per i tempi stessi di *training*. Procedendo invece con questa modalità è stato possibile creare qualcosa con prospettive future di utilizzo in tempi relativamente ristretti, sperimentando nei vari *step* del lavoro meccanismi e tecniche alternative che possono essere d'aiuto a chi è interessato all'argomento. Ovviamente, date le restrizioni dovute al tempo e alla potenza di calcolo disponibile (le due cose sono estremamente collegate), i risultati ottenuti possono essere considerati ottimi solo dal punto di vista sperimentale e didattico. La mancanza di tempo ci ha costretto a non poter provare tutte le idee che abbiamo avuto durante lo svolgimento di questa tesi, dato che per un minimo cambiamento alla struttura del modello o dell'algoritmo usato (per la *Loss Function*) sarebbe servito qualche giorno di *training* per poter avere differenze apprezzabili e decidere quale delle due o più modalità di lavoro per ogni sottoparte sarebbe stata migliore. Alcuni esempi di idee che sono rimaste tali sono:

- **Scongelamento** di più/meno *layer* durante la fase di *Transfer Learning* e misura del *tradeoff* ottenuto tra aumento di tempo e precisione;
- Passaggio alla struttura piramidale (**FPN**) di YOLOv3 per misurare concretamente i miglioramenti apportati;

- Utilizzo dell'**Objectness Score** piuttosto che del **Confidence Score** per filtrare le *Bounding Box*;
- Provare a modificare le dimensioni degli **stride** (passi) e le **funzioni di attivazione** che performano gli ultimi *layer* per l'individuazione;
- *Training* con **Optimizer** e **Learning Rate** diversi;
- *Training* puro senza utilizzo del *Transfer Learning*;
- Prove su altri **Dataset**;

Queste sono solo alcune delle prove possibili da effettuare per migliorare il modello, in verità ce ne sono molte altre, magari che non abbiamo neanche pensato, che potrebbero essere cruciali per una svolta radicale del funzionamento dell'*Object Detector*. Credo che questo campo sia veramente interessante e che le possibilità di ampliamento del lavoro svolto in questa tesi siano infinite, e spero che qualcuno che condivide il mio interesse riuscirà a portarlo avanti cercando di provare tutto ciò che la mancanza di tempo mi ha costretto a lasciare in sospeso.

Bibliografia

[1] Martín Abadi et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] Ekaba Bisong. Google colaboratory. pages 59–64, 2019.

[3] Navaneeth Bodla, Bharat Singh, Rama Chellappa, and Larry S. Davis. Soft-nms - improving object detection with one line of code. In *IEEE International Conference on Computer Vision, ICCV 2017, Venice, Italy, October 22-29, 2017*, pages 5562–5570. IEEE Computer Society, 2017.

[4] Serban Carata, Roxana Mihaescu, Eduard Barnoviciu, Mihai Chindea, Marian Ghenescu, and Veta Ghenescu. Complete visualisation, network modeling and training, web based tool, for the yolo deep neural network model in the darknet framework. In Sergiu Nedevschi, Rodica Potolea, and Radu Razvan Slavescu, editors, *15th IEEE International Conference on Intelligent Computer Communication and Processing, ICCP 2019, Cluj-Napoca, Romania, September 5-7, 2019*, pages 517–523. IEEE, 2019.

[5] Francois Chollet et al. Keras, 2015.

[6] Jan Hendrik Hosang, Rodrigo Benenson, and Bernt Schiele. Learning non-maximum suppression, 2017.

[7] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam.

Mobilenets: Efficient convolutional neural networks for mobile vision applications, 2017.

[8] Laura Igual and Santi Seguí. *Introduction to Data Science - A Python Approach to Concepts, Techniques and Applications*. Undergraduate Topics in Computer Science. Springer, 2017.

[9] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. In David J. Fleet, Tomás Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision - ECCV 2014 - 13th European Conference, Zurich, Switzerland, September 6-12, 2014, Proceedings, Part V*, volume 8693 of *Lecture Notes in Computer Science*, pages 740–755. Springer, 2014.

[10] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. *CoRR*, abs/1506.02640, 2015.

[11] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 779–788. IEEE Computer Society, 2016.

[12] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. In *2017 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017*, pages 6517–6525. IEEE Computer Society, 2017.

[13] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018.

[14] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *IEEE Trans. Pattern Anal. Mach. Intell.*, 39(6):1137–1149, 2017.

[15] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 2818–2826. IEEE Computer Society, 2016.

[16] Zhong-Qiu Zhao, Peng Zheng, Shou-tao Xu, and Xindong Wu. Object detection with deep learning: A review. *IEEE Trans. Neural Networks Learn. Syst.*, 30(11):3212–3232, 2019.

Ringraziamenti

Vorrei ringraziare innanzitutto il Prof. Asperti per avermi dato questa possibilità di conoscere il mondo del *Machine Learning/Deep Learning* e lavorarci in prima persona grazie alla sua guida, indispensabile soprattutto nelle fasi iniziali essendo questo un approccio totalmente diverso a tutto ciò che avessi visto e studiato fino ad ora. Ringrazio inoltre gli innumerevoli ricercatori che continuano a portare avanti l'interesse verso questo campo e a diffonderlo a tutti coloro che come me ne rimarranno poi affascinati, segnando un momento importante della propria vita. Ringrazio inoltre Giorgia per il supporto datomi in questi ultimi tre anni. Un grazie a tutta la mia famiglia e a tutti i miei amici sparsi in giro per l'Italia che mi hanno sempre aiutato nel momento del bisogno e portato ad essere ciò che sono, in particolare i membri della Pyrofile con cui ho condiviso molte delle migliori esperienze fatte nel periodo universitario. Grazie a tutti, sono molto fiero di avervi intorno.