ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

Tecniche di deep learning per l'object detection

Scuola di Scienze
Corso di Laurea in Informatica

Presentata da:
Erich KOHMANN

Relatore:
Chiar.mo Prof.
Andrea ASPERTI

III Sessione
Anno Accademico 2018/19

Bologna, 8 dicembre 2019

ALMA MATER STUDIORUM - UNIVERSITÀ DI BOLOGNA

Sommario

Scuola di Scienze
Corso di Laurea in Informatica

Tecniche di deep learning per l'object detection

di Erich KOHMANN

L'*object detection* è uno dei principali problemi nell'ambito della *computer vision*. Negli ultimi anni, con l'avvento delle reti neurali e del *deep learning*, sono stati fatti notevoli progressi nei metodi per affrontare questo problema.

Questa tesi è il risultato di una ricerca fatta sui principali modelli di *object detection* basati su *deep learning*, di cui presenta le caratteristiche fondamentali e gli elementi che li contraddistinguono dai precedenti.

Dopo un infarinatura iniziale sul *deep learning* e sulle reti neurali in genere, vengono presentati i modelli caratterizzati da tecniche innovative che hanno portato ad un miglioramento significativo, sia nella precisione e nell'accuratezza delle predizioni, che in termini di consumo di risorse.

Nella seconda parte l'elaborato si concentra su *YOLO* e sui suoi sviluppi. *YOLO* è un modello basato su reti neurali convoluzionali, con il quale i problemi di localizzazione e classificazione degli oggetti in un'immagine sono stati trattati per la prima volta come un unico problema di regressione. Questo cambio di prospettiva apportato dagli autori di *YOLO* ha aperto la strada verso un nuovo approccio all'*object detection*, facilitando il successivo sviluppo di modelli sempre più precisi e performanti.

Indice

Elenco delle figure

Elenco delle tabelle

A mio fratello Carlo…

Capitolo 1

Introduzione

Negli ultimi anni, con l'affermarsi del *deep learning*, sono stati fatti notevoli progressi nell'ambito dell'interpretazione e dell'elaborazione automatizzata dei dati. Uno dei principali problemi nell'interpretazione di dati, o più precisamente nella *computer vision*, è l'*object detection*. L'*object detection* consiste nell'individuare e classificare gli oggetti presenti in un'immagine. Il problema si può estendere abbastanza facilmente ad altre rappresentazioni della realtà, come ad esempio le matrici tridimensionali di punti ottenute dagli scanner laser. I modelli per la risoluzione di questo problema sono alla base di numerosi software che si occupano di *computer vision*, come applicazioni per il monitoraggio di video di sorveglianza, applicazioni per l'analisi automatica di radiografie, sistemi di guida autonoma, etc..

L'argomento della tesi è stato concordato con il gentilissimo professore *Asperti* - al quale rinnovo i miei ringraziamenti - in virtù del lavoro da me precedentemente svolto per il tirocinio curricolare: la creazione di un software per il campionamento automatico di immagini dal social media *Instagram*, attraverso le relative *API*, e l'implementazione di una rete neurale convoluzionale pre-addestrata per l'*object detection* per la classificazione delle immagini così ottenute. Ho chiamato questo software *IASIC* (acronimo di ***I**nstagram **A**utomatic **S**ampler and **I**magenet **C**lassifier*); *IASIC* è *open-source* con licenza *GPL v3* ed è reperibile a questo indirizzo: `https://github.com/3richK/IASIC`.

Con questa tesi si cerca di fornire un quadro complessivo sul problema dell'*object detection* nell'era del *deep learning*. Introduciamo il lettore all'argomento con un po' di cenni sul *deep learning* e spiegando il problema dell'*object detection*. Esponiamo quindi i modelli basati su *deep learning*, che negli ultimi anni hanno apportato le principali innovazioni nell'ambito dell'*object detection*, concentrandoci sugli aspetti più originali introdotti dagli stessi. Ci soffermiamo infine su *YOLO* e sui suoi sviluppi, un modello che nel 2016 ha apportato un significativo miglioramento prestazionale, se paragonato ai modelli precedentemente in essere.

Questa tesi può essere dunque vista come un'indagine approfondita sull'evoluzione dei modelli basati su *deep learning* per l'*object detection*. Per lo scopo ci serviamo di numerosi riferimenti esterni ad articoli e riviste che trattano l'argomento. Il periodo di riferimento va dalla nascita di *RCNN* (2014), alla pubblicazione di *RetinaNet* (2018).

Iniziamo introducendo l'argomento delle reti neurali con il capitolo 2, facendo un po' di chiarezza sulla terminologia ed esponendo le basi teoriche relative alla singola unità d'apprendimento ed alla composizione di unità per formare reti più complesse. In questo capitolo esponiamo sommariamente anche le principali tecniche di

addestramento di una rete neurale.

Con il capitolo 3 presentiamo il problema dell'*object detection* ed alcuni tra i più celebri modelli risolutivi, con relative peculiarità e le principali novità introdotte; per chiudere il capitolo discutiamo sommariamente quelli che sono gli ambiti d'applicazione più comuni di questi modelli.

Dal capitolo 4 in poi ci concentriamo su *You Only Look Once*, il primo *one-stage detector*, ossia il primo modello a gestire i problemi di individuazione e di classificazione di un oggetto, come un unico problema di regressione. Precedentemente il problema di localizzare e quello di classificare un oggetto erano stati trattati sempre come due problemi separati, da risolvere con due modelli distinti. *YOLO* permette di ottenere, oltre che una maggiore generalizzazione del modello sui dati, ottime prestazioni in termini di tempo d'esecuzione, stravolgendo lo stato dell'arte ed aprendo la strada all'implementazione di questo genere di modelli su dispositivi con risorse limitate.

Con il capitolo 5 introduciamo infine gli aggiornamenti apportati a *YOLO* negli ultimi due anni e *Complex-YOLO*, un modello basato sull'architettura di *YOLO*, ideato come supporto principale per sistemi di guida autonoma. *Complex-YOLO* è in grado di compiere *object detection* in tre dimensioni partendo da dati provenienti da sensori laser.

Capitolo 2

Le Reti Neurali

2.1 Il Deep Learning

Con il termine *deep learning* ci si riferisce ad un particolare campo del *machine learning* che comprende una serie di modelli matematici particolarmente efficienti nella risoluzione di problemi molto complessi da risolvere con modelli tradizionali. Più precisamente, il termine deriva dall'atto di addestrare una rete neurale.

In questa sezione tratteremo sommariamente le caratteristiche principali di questi modelli, partendo dalla singola unità logica per arrivare ad introdurre i diversi approcci per l'apprendimento.

Nella sezione successiva tratteremo invece le principali categorie di reti neurali, per poi soffermarci nei capitoli successivi sui principali modelli per l'*object detection*.

2.1.1 La Regressione Logistica

La regressione logistica è l'algoritmo d'apprendimento base di cui si compongono le reti neurali.

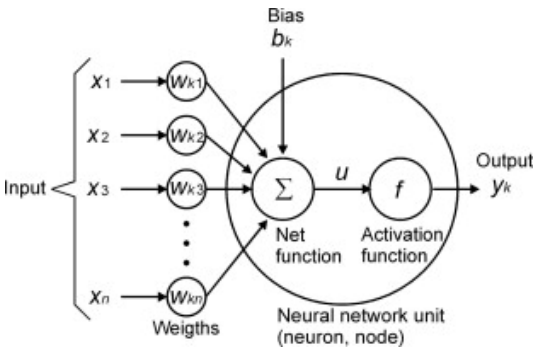

FIGURA 2.1: **La struttura della regressione logistica.** Dato il vettore x come input, le sue *features* vengono moltiplicate per i pesi w ed al risultato viene aggiunto il valore del *bias* b. Infine il valore di output è ottenuto applicando la funzione d'attivazione al risultato.

Serve a risolvere problemi del tipo:
Dato $x \in \mathbb{R}^{n_x}$[1], calcola $\hat{y} = W^T x + b$, dove $\hat{y}$ è un'approssimazione del risultato esatto y e W, b sono parametri del modello.

[1] dove n_x è la cardinalità del vettore x

Per semplicità, assumiamo che il problema che vogliamo risolvere sia un problema di classificazione binaria. Per esempio vogliamo decidere se, data una certa immagine, questa rappresenti o meno un gatto[2].

Rappresentiamo la nostra immagine come un vettore x unidimensionale di byte[3].

Fissiamo $y = 1$ nel caso in cui l'immagine rappresenti un gatto, $y = 0$ nel caso in cui nell'immagine non ci sia un gatto. La nostra unità di regressione logistica dovrà dunque calcolare un'approssimazione di y; più precisamente calcolerà le probabilità che nell'immagine ci sia un gatto.

Dato x, vogliamo dunque calcolare:

$$\hat{y} = P(y = 1|x)$$

Perché ciò avvenga, serve che i parametri W e b siano opportunamente configurati. Ed è qui che entra in gioco il concetto di *apprendimento*. La computazione infatti, durante l'apprendimento[4], si compone di due fasi: una prima fase - detta propagazione in avanti, o *forward propagation* - durante la quale viene calcolata l'approssimazione del risultato $\hat{y}$, ed una seconda - detta propagazione all'indietro, o *backward propagation* - durante la quale viene calcolato l'errore avvenuto durante la computazione e, a partire da questo, sfruttando il concetto di *derivata*, vengono aggiornati i parametri W e b in modo tale da ridurre l'errore in computazioni future.
Va da sé che, in linea generale, maggiore sarà la cardinalità e la qualità[5] dell'insieme degli esempi d'apprendimento, migliore sarà l'approssimazione del risultato.

In linea di massima, una volta che il modello compie un errore trascurabile nell'approssimazione del risultato - *i.e. ha raggiunto un buon livello di apprendimento* - potrà essere utilizzato per analizzare dati nuovi e non precedentemente classificati.

Nota: perché $\hat{y} \in [0, 1]$ è necessario processare l'output di $\hat{y} = W^T x + b$ con una specifica funzione che lo regolarizzi; tale funzione è detta *funzione d'attivazione*. Nel caso della classificazione binaria ad esempio, in genere si usa la funzione *sigmoide* definita come segue:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

con $\sigma(z) \in [0, 1]$ e $z = (\hat{y}) = W^T x + b$.

[2] esempio ricorrente in letteratura.

[3] data un'immagine di 250x250 pixel rappresentata in RGB, avremo che ogni pixel può assumere un valore da 0 a 255 per il rosso, per il verde e per il blu. Avremo quindi $250 * 250[pixel]$, ciascuno dei quali dal peso di 3 byte ($3 * 8 = 32bit$). Costruiamo quindi il nostro vettore x come segue:
il primo elemento sarà la componente rosso del pixel 1x1, la seconda il verde del pixel 1x1, la terza il rosso dell'1x1, la quarta sarà invece il rosso del pixel 2x1, e così via, fino ad ottenere un vettore di $250 * 250 * 3[bytes]$.

[4] in fase di apprendimento, oltre all'input x, in genere, è noto anche il risultato atteso y; la coppia (x, y) compone quindi quello che è un singolo esempio d'apprendimento o *training example*.
L'insieme degli esempi d'apprendimento è detto *training set*.

[5] perché un *training set* sia considerabile di buona qualità, è opportuno che porti in esempio configurazioni quanto più possibile diverse dell'input che diano lo stesso risultato, in modo tale che il modello si comporti al meglio anche in circostanze molto diverse

2.1.2 Le Reti Neurali

Le reti neurali o *neural networks* sono la composizione di più unità logiche.

La loro architettura è divisa in livelli, ciascuno dei quali ha un numero prefissato di unità logiche, una certa funzione di attivazione ed i propri parametri W e b.

Escludendo il livello di input (il vettore x) e quello di output (il vettore $\hat{y}$), restano i così detti livelli nascosti, o *Hidden Layers*.

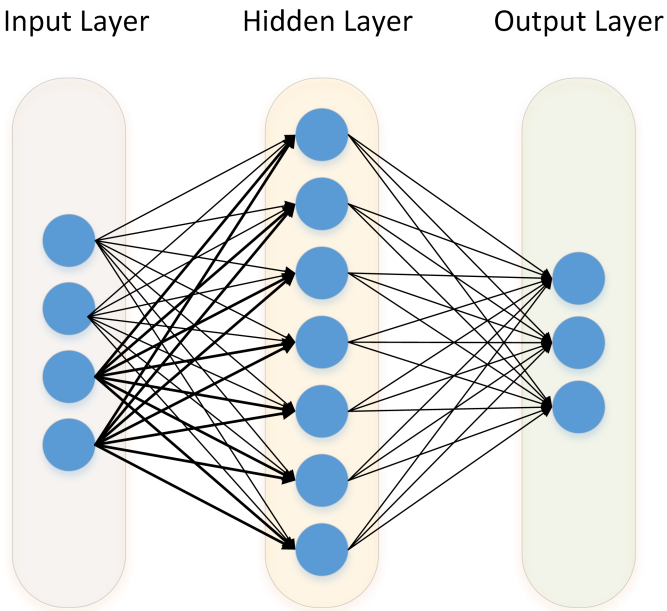

FIGURA 2.2: **L'architettura di una rete neurale generica.** (*Immagine ottenuta da «Deep Learning - What is it and why does it matter?»*)

Ciascun livello calcola il vettore di input per il livello successivo, detto anche *vettore d'attivazione* o *attivazione* per il livello successivo. Esistono reti la cui complessità si sviluppa principalmente nella cardinalità delle unità logiche di ciascun livello, piuttosto che nel numero di livelli, e viceversa. Diverse architetture rispondono a diverse esigenze computazionali.

Talora le reti neurali possono essere a loro volta composizione di altre reti neurali. Questo ci permette di dividere il problema iniziale in diversi sotto-problemi più semplici.

È il caso, ad esempio, di alcune reti neurali per il riconoscimento facciale: i primi livelli si occupano di riconoscere figure geometriche all'interno dell'immagine - come un segmento o un ovale - i successivi di interpretarne il significato (per esempio ad un certo ovale può corrispondere il contorno di un occhio, oppure un segmento può essere uno dei lati del perimetro di uno zigomo) ed eventualmente di decidere se la faccia riconosciuta appartenga o meno ad una certa categoria.

2.1.3 Approcci per l'apprendimento

Come già sottolineato, uno dei punti di forza delle reti neurali è la capacità adeguarsi e, con l'ausilio degli *esempi di apprendimento*, di apprendere gradualmente il modo di calcolare la soluzione per un certo problema.

L'apprendimento avviene in fase di *backward propagation* ed in pratica consiste nell'aggiornamento dei parametri dei diversi livelli, fatto in modo tale da ridurre lo scarto[6] tra il valore ottenuto e quello atteso.

Per far ciò, il modello prima calcola - partendo dal livello di output e, all'indietro, per ciascun livello della rete - il valore delle derivate delle operazioni effettuate durante la *forward propagation* rispetto W e b. Una volta ottenuti questi valori, i parametri di ciascun livello vengono aggiornati di α volte tali valori.
Il parametro α è detto *fattore d'apprendimento* e determina quanto velocemente il modello aggiorna i propri parametri.

Sia a seconda della loro architettura che a seconda dello scopo, le reti possono essere addestrate in diversi modi.

Distinguiamo tre categorie di apprendimento: supervisionato, semi-supervisionato e non supervisionato.

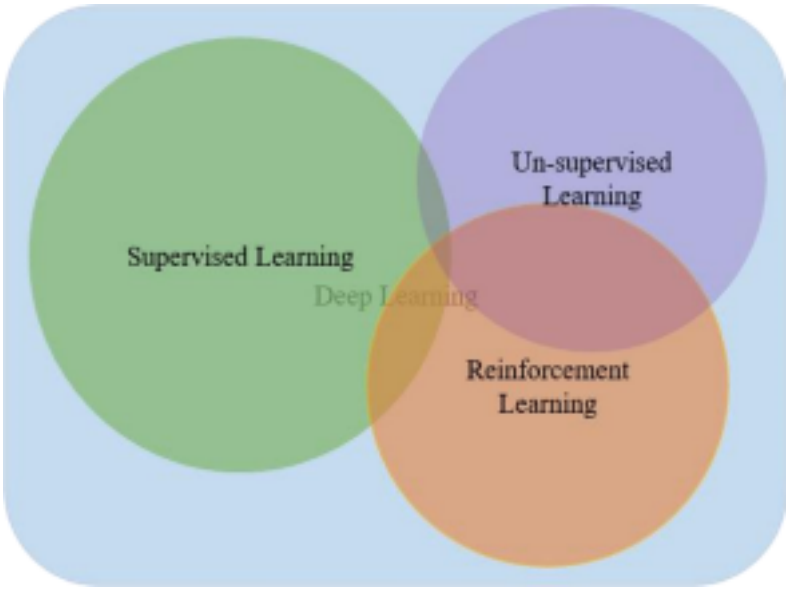

FIGURA 2.3: **Categorie di approcci di apprendimento.** [1]

- Apprendimento Supervisionato

È una tecnica di apprendimento durante la quale al modello vengono dati in pasto esempi d'apprendimento composti dal vettore di input x e dal risultato atteso y. In questo caso l'esempio si dice *labeled*, ossia pre-classificato.

- Apprendimento Semi-Supervisionato

In questo caso l'apprendimento avviene prendendo in input dati solo parzialmente pre-classificati (*semi-labeled*). Usano questa tecnica di apprendimento reti quali quelle generative e quelle ricorsive.

[6] tale scarto/errore viene calcolato usando la così detta *loss function* o, nel caso di un intero *training set* dalla *cost function*, che altro non è che la media aritmetica tra i risultati delle *loss function* calcolate su ciascun *training example*.

- Apprendimento Non-Supervisionato

I sistemi ad apprendimento non supervisionato invece sono in grado di apprendere senza la necessità di dati pre-classificati. Come spiegato in [1], in questo caso il modello impara la rappresentazione interna, o le caratteristiche importanti dell'input per scoprire relazioni o strutture nascoste all'interno dei dati di input.

- Deep Reinforcement Learning

Zahangir Alom et al. [1] individuano anche una quarta categoria di metodi d'apprendimento, ossia il *Deep Reinforcement Learning* (o *DRL*). Il *DRL* nasce nel 2013 con *Google Deep Mind*. Da allora, sono stati proposti molti modelli avanzati basati su *DRL*. *Zahangir Alom et al.* propongono inoltre un esempio di *DRL*: dato in input un vettore x_t, il modello predice $\hat{y}_t = f(x_t)$ e riceve un costo $c_t \sim P(c_t|x_t, \hat{y}_t)$, dove P è una distribuzione di probabilità sconosciuta; a questo punto il modello restituisce un dato, detto *"livello di rumore"*, come output.

2.2 Tipologie ed ambiti di applicazioni delle Reti Neurali

2.2.1 "Standard" Neural Networks

Sono reti neurali che hanno un'architettura semplice; generalmente vengono usate per classificare e/o elaborare dati strutturati.

Per esempio presentiamo un modello che calcola il prezzo di una casa. Il nostro vettore x potrà essere ad esempio composto dalle seguenti componenti:

- x_0 area calpestabile;
- x_1 numero di camere da letto;
- x_2 CAP (*o Zip Code*);
- x_3 numero di scuole nelle vicinanze;
- x_4 aspettativa di vita del quartiere;
- x_5 numero di supermercati in zona;
- x_6 tasso di criminalità.

A questo punto definiamo l'architettura nostra rete: supponiamo di avere un unico *hidden layer* composto da 10 unità logiche (o neuroni) e, per la natura del problema, un livello di output composto da un'unica unità logica. Otteniamo così l'architettura mostrata in figura 2.4.

La computazione in fase di addestramento, dato un singolo *training example* (x, y), si compierà quindi in 5 passi:

1. Per ogni unità del primo (ed unico) livello nascosto, si calcola $a_{0,i} = g_0(W_0^T x + b_0), i \in \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$ (dove g_0 è la funzione d'attivazione per il primo livello) ottenendo così il vettore a_0;

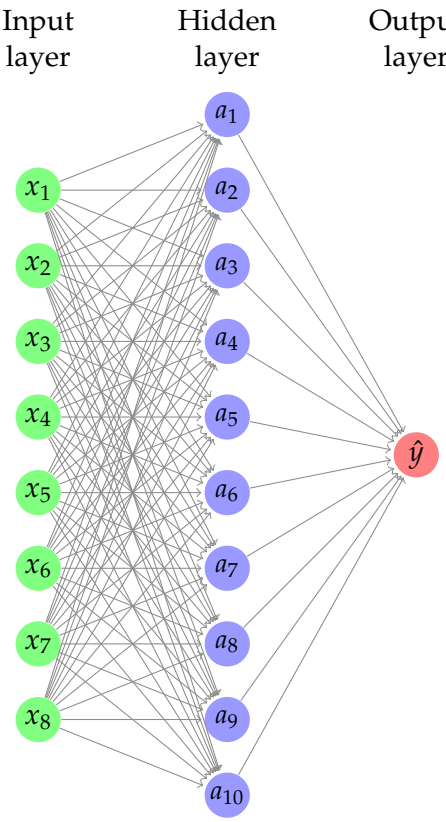

FIGURA 2.4: **Esempio dell'architettura standard di una rete neurale.** In questo esempio l'input è un vettore di 8 *features*; la rete ha un unico livello nascosto composto da 10 unità logiche ed un livello di output composto da un'unica unità logica.

2. Viene calcolato il valore dell'unica unità del livello di output come $\hat{y} = a_1 = g_1(W_1^T a_0 + b_1)$;
3. Si calcola l'errore nell'approssimazione del risultato usando una *loss function*, per esempio, usando l'errore quadratico: $L(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$;
4. Si calcolano le derivate delle funzioni composte applicate per arrivare al calcolo dell'errore:
   (a) $da_1 = \frac{dL}{da_1} = a_1 - y$;
   (b) $dW_1 = da_1 * a_1^T$;
   (c) $db_1 = da_1$;
   (d) $da_0 = \frac{dL}{da_1}\frac{da_1}{da_0}$ (per la regola della catena delle derivate);
   (e) $dW_0 = da_0 * a_0^T$;
   (f) $db_0 = da_0$;
5. Si aggiornano i parametri del modello usando il *learning rate* α:
   - $W_0 = W_0 - \alpha * dW_0$;
   - $b_0 = b_0 - \alpha * db_0$;
   - $W_1 = W_1 - \alpha * dW_1$;
   - $b_1 = b_1 - \alpha * db_1$;

2.2.2 Convolutional Neural Networks

Vengono utilizzate principalmente nella *computer vision*. La loro caratteristica principale è la presenza appunto di uno o più livelli convoluzionali, ossia livelli le cui unità, per combinare input e parametro W, invece di calcolare un semplice prodotto tra vettori, usano una funzione convoluzionale[7].

2.2.3 Recurrent Neural Networks

Sono reti utilizzate principalmente per elaborare dati dipendenti da computazioni precedenti. È il caso dei software *Speech to Text* o per l'elaborazione ed interpretazione del linguaggio naturale.

2.2.4 Custom/Hybrid Neural Networks

Esistono poi reti neurali che si compongono di più architetture e/o di reti più semplici. Vengono usate, ad esempio, per la realizzazione di modelli per la guida autonoma.

[7] "la funzione convoluzionale è definita come l'integrale del prodotto di due funzioni, dopo che una delle due è stata invertita e traslata: $(f * g)(t) = \int_{-\infty}^{\infty} f(t-r)g(r)dr$." *Definizione di* https://en.wikipedia.org/wiki/Convolution

Capitolo 3

L'Object Detection

Presentiamo ora il problema dell'*object detection*; contestualmente ne presentiamo i principali modelli risolutivi e ne discutiamo alcune applicazioni pratiche.

3.1 Il problema dell'Object Detection

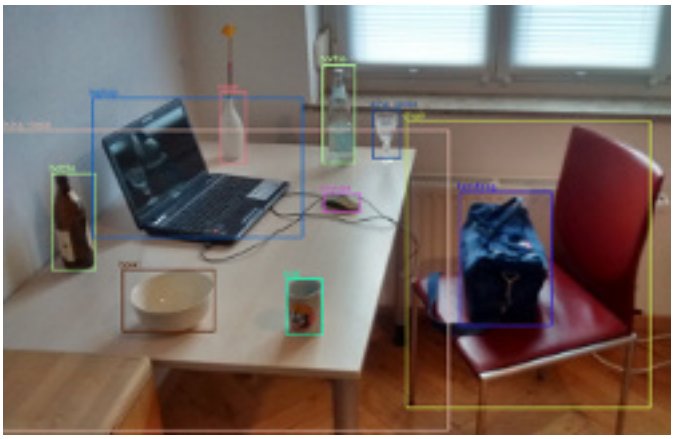

FIGURA 3.1: **Esempio di Object Detection.** *MTheiler@wikipedia.com, "Detected-with-YOLO"* (2019)

Uno dei principali problemi nella *computer vision* è l'*object detection*. Come spiegato da *Zou et al.* [32], l'*object detection* è un'importante problema che consiste nell'individuare istanze di oggetti all'interno di un'immagine e nel classificarle come appartenenti ad una certa classe (come umani, animali o auto).

L'obiettivo è quello di sviluppare tecniche e modelli computazionali che forniscano uno degli elementi basilari necessari per le applicazioni di *computer vision*: la risposta alla domanda "quali oggetti ci sono qui?".
L'*object detection* è alla base di molte applicazioni per la *computer vision*, quali l'*instance segmentation*, l'*image captioning*, l'*object tracking*, etc..
Dal punto di vista applicativo, è possibile raggruppare l'*object detection* in due categorie: *"general object detection"* e *"detection applications"* [1]. Per la prima, l'obiettivo è quello di indagare i metodi per individuare diversi tipi di oggetti usando un unico *framework*, al fine di simulare la visione e la cognizione umana. Nel secondo caso ci si riferisce al riconoscimento, sotto specifici scenari applicativi, di oggetti di una certa classe: è il caso delle applicazioni per il *pedestrian detection*, il *face detection* o per il *text detection*.

3.2 Modelli per l'Object Detection

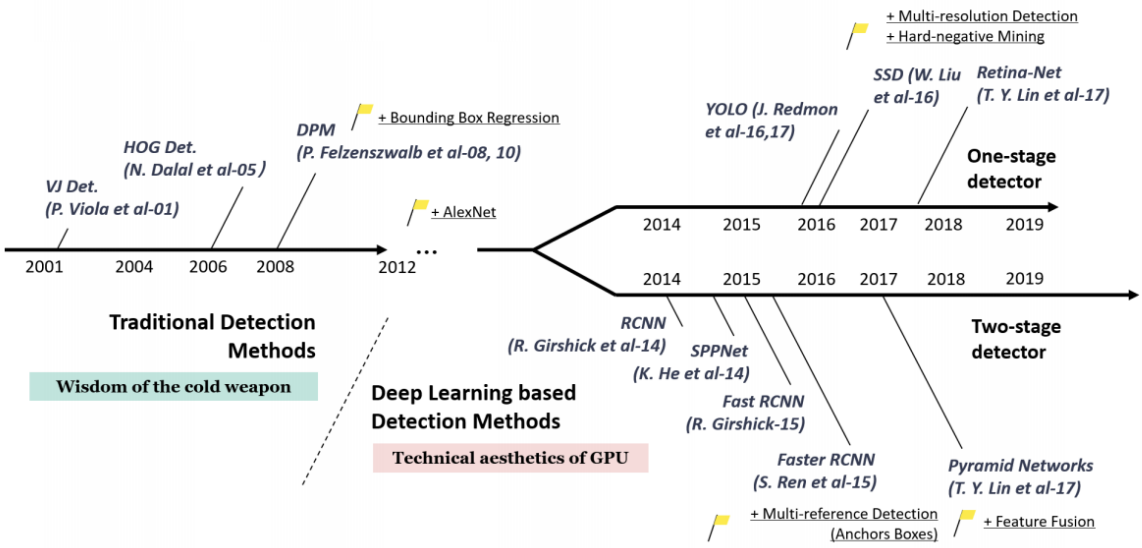

FIGURA 3.2: **La linea del tempo dell'evoluzione dei modelli di *object detection* negli ultimi 20 anni.** [32]

Attualmente i modelli per l'*object detection* possono essere divisi in due macro-categorie: *two-stage* e *one-stage detectors*. Nel primo caso, parliamo di modelli che dividono il compito di individuare oggetti in più fasi, seguendo una politica *"coarse-to-fine"*. Nel secondo parliamo di modelli il cui processo cerca di completare il riconoscimento in un passo unico con l'utilizzo di un'unica rete.
Come mostrato in figura esistono diversi modelli per ciascuna categoria. Vediamo adesso le caratteristiche fondamentali di ciascuno di questi, come spiegato in [32].
(**Nota**: *dove non specificato si tratta di modelli di tipo two-stage detectors.*)

3.2.1 RCNN

L'idea che si cela dietro le *RCNN* è relativamente semplice: iniziano con l'estrarre un insieme di proposte di oggetti (*object candidate boxes*) usando una *selective search* [25]. Dunque, per ogni proposta, viene ritagliata un'immagine a dimensione fissa che viene data in pasto ad una *CNN* addestrata per estrarne le caratteristiche fondamentali. Alla fine vengono usati classificatori lineari *SVM* per decidere la presenza di un oggetto in ciascuna regione e per riconoscere le categorie degli oggetti trovati.

Selective Search. Come *Jasper Uijlings et al.* sottolineano in [25], prima del loro modello di *selective search*, per determinare la posizione di un oggetto all'interno dell'immagine, si divideva l'immagine in un numero prefissato di regioni, o segmenti, e vi si applicava un classificatore per capire quanto fosse probabile che in ciascuna regione vi fosse un oggetto. Dopodiché si effettuava la fusione delle aree con predizioni simili e si ripeteva il procedimento sulle nuove regioni così ottenute. Con

questo metodo si tende a cercare di delineare con precisione i contorni di un oggetto, a discapito della capacità di riconoscere una buona parte degli oggetti presenti nelle immagini.

Con la *selective search* invece, la suddivisione delle aree da proporre avviene in modo gerarchico, così da catturare la presenza di oggetti in varie pose e dimensioni. In modo analogo all'*exhaustive search*, si divide l'immagine in un numero prefissato di aree o segmenti. A questo punto però viene applicato un algoritmo *greedy*[1] il quale iterativamente raggruppa, tra tutte, le due regioni più simili e, per procedere, calcola la *somiglianza*[2] tra queste nuove regioni ed i loro vicini. Tenendo in memoria tutte le regioni calcolate nei vari *step* in una rappresentazione che ne conservi la gerarchia, l'algoritmo procede fino a che l'intera immagine non diventa un'unica regione. Alla fine si considerano tutte le aree calcolate, in relazione alla loro posizione gerarchica, come potenziali locazioni di oggetti e quindi come proposte.

Questo approccio ha apportato un significativo miglioramento nelle performance rispetto le metriche di *VOC07*[3], con un avanzamento dal 33.7% (*DPM-v5* [7]) al 58.5%.

Pur avendo apportato un importante miglioramento nell'accuratezza del risultato finale, i modelli *RCNN* presentano una controindicazione importante: il grande numero di *features* da classificare, risultante dalla sovrapposizione delle tante aree proposte (più di 2'000 aree per ogni immagine), porta il modello a dover elaborare un grande numero di dati, compromettendo le *performance* dello stesso. L'architettura *SPP-Net* è stata successivamente proposta per far fronte a questo problema.

3.2.2 SPP-Net

Nel 2014 *K. He et al.* hanno proposto *Spatial Pyramid Pooling Networks* (*SPP-Net*) [8]. Prima di *SPP-Net*, i modelli di *CNN* richiedevano un input a dimensione fissa - per esempio immagini di 224x224 pixel per *AlexNet* [11].

Ciò comportava una perdita d'accuratezza nella classificazione di immagini di dimensione e proporzioni diverse da quelle prefissate, le quali, per poter essere classificate, andavano ridimensionate, con eventuale perdita di dettaglio o deformazione dell'immagine, oppure addirittura tagliate. Il motivo è che, per la loro natura, le *CNN* sono composte, negli ultimi livelli, da livelli interamente connessi (o *fully connected layers*), i quali lavorano su di un input di dimensione prefissata.

[1] letteralmente *"goloso"*, un algoritmo di tipo *greedy* risolve problemi di ottimizzazione tentando di costruire la soluzione ottima scegliendo sempre la strada più "ghiotta" tra quelle possibili. Non tutti i problemi di ottimizzazione sono risolvibili con tecniche *greedy*, si veda il problema del *commesso viaggiatore*.

[2] la somiglianza è stata definita dagli autori, in funzione di due aree a e b, come $S(a,b) = S_{size}(a,b) + S_{texture}(a,b)$; S_{size} è definita come la percentuale (espressa in frazione di 1) dell'immagine che viene occupata dall'intersezione delle due aree; $S_{texture}$ è invece un fattore determinato da quanto le texture delle due aree hanno in comune.

[3] la *PASCAL Visual Object Classes (VOC) challange* è stata una delle competizioni più importanti nelle prime community di sviluppatori di applicazioni per la *computer vision*. Consiste in diversi compiti, tra cui la classificazione di immagini, il riconoscimento di oggetti, la divisione semantica di immagini ed il riconoscimento di azioni. Nell'*object detection* sono due le versioni di *PASCAL-VOC* più usate: *VOC07* e *VOC12*; la prima consiste in 5'000 immagini con 12'000 oggetti pre-classificati, la seconda in 11'000 immagini con 27'000 oggetti pre-classificati.

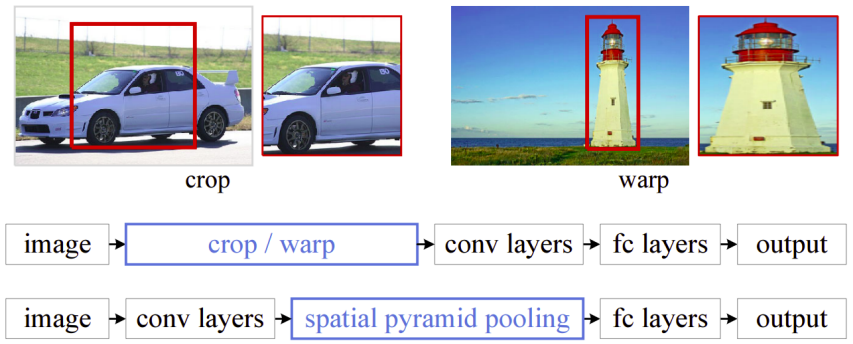

FIGURA 3.3: **Differenze tra architettura di *CNN* standard e *SPP-net*.** In alto, due esempi di come i modelli *CNN* trattano le immagini di dimensione arbitraria per poterle classificare. In mezzo, l'architettura standard di una *CNN*. In basso, la struttura di *SPP-Net*. [8]

La grande innovazione di *SPP-net* sta proprio nell'aver introdotto, tra i livelli convoluzionali e quelli interamente connessi, un livello di *Spatial Pyramid Pooling* (si veda figura 3.3), il quale permette di mettere insieme le *features* evidenziate dai livelli convoluzionali e di restituire un output di dimensione prefissata.

Spatial Pyramid Pooling [13] era un modello già esistente, estensione del *Bag of Words*, che veniva spesso utilizzato nell'ambito della *computer vision* prima dell'avvento delle reti convoluzionali.

Utilizzando *SPP-Net* per l'*object detection*, l'insieme delle aree proposte viene calcolato a partire dalle *features* estratte dai primi livelli convoluzionali. Successivamente vengono generate delle rappresentazioni di dimensioni prefissata sulle regioni proposte. Queste rappresentazioni possono essere dunque date in pasto ai classificatori.

SPP-Net è più di 20 volte più veloce dei modelli *RCNN*, senza sacrificare l'accuratezza nella classificazione (*VOC07 mAP=59.2%*). Nonostante *SPP-Net* abbia portato un'importante miglioramento nelle performance, ci sono ancora degli aspetti negativi non trascurabili: prima di tutto l'apprendimento è ancora diviso in più fasi (è un *two-stage detector*); in secondo luogo *SPP-Net*, durante l'apprendimento, addestra finemente i parametri dei livelli interamente connessi (*fully connected layers*), trascurando l'addestramento dei livelli precedenti.

Nel 2015 venne proposto *Fast R-CNN* [6], che pone una soluzione a questi problemi.

3.2.3 Fast R-CNN

Nel 2015, *R. Girshick* propose il riconoscitore *Fast R-CNN* [6], il quale apportò un ulteriore miglioramento rispetto alle *R-CNN* standard ed a *SPP-Net*.

Fast R-CNN consente di addestrare simultaneamente un riconoscitore ed un disegnatore di *bounding box*[4] all'interno di un unico modello.La *loss function* di *Fast R-CNN* infatti tiene conto dell'errore compiuto in ciascuna fase della propagazione

[4] si tratta di rettangoli disegnati sull'immagine di input al fine di delimitare l'area occupata da un oggetto.

in avanti.

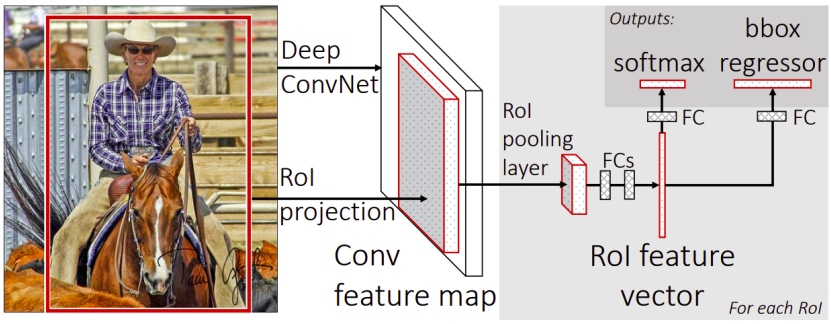

FIGURA 3.4: **Architettura di *Fast R-CNN*.** Un'immagine e diverse regioni di interesse (*RoI*) vengono date in pasto ad una serie di livelli convoluzionali. Ogni *RoI* è rimodellata in una *feature map* di dimensioni prefissate e quindi mappata in un vettore - anche questo di dimensioni prefissate - dai successivi livelli interamente connessi (*fully connected layers*). Per ogni *RoI* la rete da in output due vettori: le probabilità d'appartenenza a ciascuna classe riconoscibile e le informazioni spaziali delle relative *bounding box* (diverse in base alla classe). Questa architettura è addestrata *end-to-end* con una *loss function "multi-task"*. [6]

Fast R-CNN prende in input un immagine ed un insieme di *object proposal*, ossia di aree che si suppone contengano oggetti all'interno dell'immagine; la rete processa l'immagine attraverso una serie di livelli convoluzionali e di *max pooling* per estrarne le *features* e produrre una *convolutional feature map*. A questo punto un livello di *pooling*, chiamato *RoI Pooling Layer* (*Region of Interest*), estrae dalla mappa appena ottenuta un vettore di dimensione prefissata e lo processa attraverso due livelli interamente connessi. Ogni vettore così ottenuto si propaga in due direzioni: in entrambi i casi passa per una serie di livelli interamente connessi; nel primo caso l'output passa poi attraverso un livello *softmax* per stimare le probabilità, per ciascuna delle K classi di oggetti riconoscibili, che nell'area (vettore) ci sia un oggetto della *k-esima* classe; nel secondo i livelli producono quattro numeri reali per ciascuna delle K classi di oggetti. Questi valori codificano, per ciascuna classe riconoscibile, il centro e le dimensioni della corrispondente *bounding box*.

Sul dataset *VOC07*, *Fast R-CNN* ha migliorato la *mAP* dal 58.5% di *RCNN* al 70.0%, ed ha un tempo di esecuzione fino a 200 volte minore di un semplice modello *RCNN*.

Nonostante *Fast R-CNN* integri con successo i vantaggi dei modelli *RCNN* e *SPP-Net*, la sua velocità è ancora limitata dalla necessità di generare delle aree "proposta". La domanda dunque sorge spontanea: "possiamo generare proposte di oggetti usando un modello convoluzionale?".

La risposta venne successivamente fornita da *Shaoqing Ren et al.* con il modello *Faster R-CNN*.

3.2.4 Faster R-CNN

Nel 2015 *S. Ren et al.*, poco dopo la pubblicazione di *Fast R-CNN*, proposero il riconoscitore *Faster R-CNN* [24].

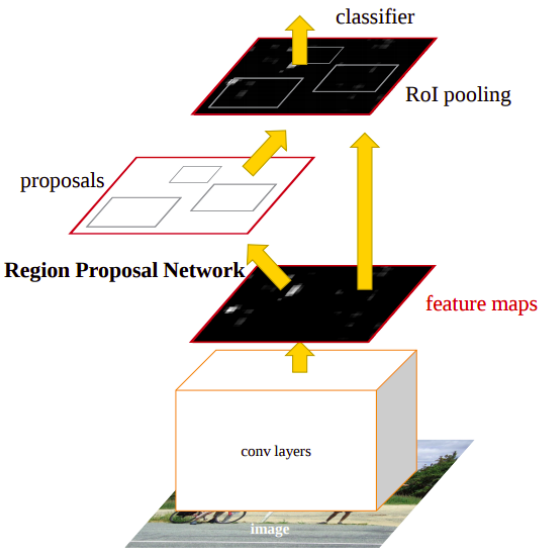

FIGURA 3.5: **Struttura di *Faster R-CNN*.** [24]

Faster R-CNN è il primo riconoscitore basato su *Deep Learning* ad essere interamente *end-to-end* e quasi *realtime* (*COCO mAP@.5=42.7%, COCO mAP@[.5,.95]=21.9%, VOC07 mAP=73.2%, VOC12 mAP=70.4%, 17fps con ZFNet, dati ottenuti da «Visualizing and understanding convolutional networks»*).

La principale innovazione apportata da *Faster R-CNN* è l'introduzione della rete *Region Proposal Network* (*RPN*) che implementa un sistema di *region proposal* molto efficiente, quasi privo di costi computazionali se paragonato ai modelli antecedenti.

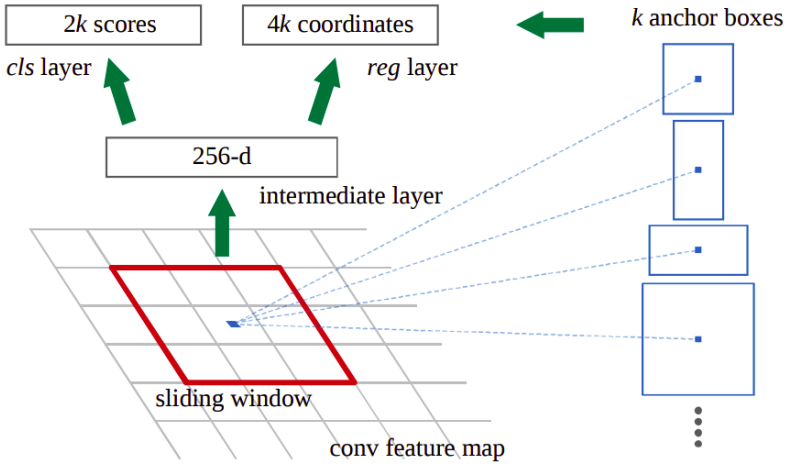

FIGURA 3.6: **Architettura del *Region Proposal Network*.** [24]

Region Proposal Network. L'idea dietro l'*RPN* è che la mappa convoluzionale di *features* ottenuta dai livelli convoluzionali - si veda sezione 3.2.3 *"Fast RCNN"* - possa essere utilizzata anche per generare "aree proposta" (o *region proposal*).

L'*RPN* consiste infatti in una serie di livelli convoluzionali applicati sulle *feature map* ottenute dai livelli convoluzionali iniziali della rete *Fast R-CNN*. Per generare le aree proposta, *S. Ren et al.* optano quindi per integrare una piccola rete alla fine dei livelli convoluzionali di *Fast R-CNN*. Questa rete prende in input una *feature map* di dimensione *nxn* ed è composta da tre livelli convoluzionali. Un primo livello condiviso di dimensione *nxn* e due livelli "gemelli" interamente connessi di dimensione 1x1 (vedi figura 3.6): fissato l'*iperparametro k*, il quale denota il numero massimo di proposte da avanzare per ogni locazione, uno dei due livelli gemelli (*reg*) produce in output le coordinate di *k "bounding box"*; l'altro livello gemello, (*cls*) restituisce, per ogni *bounding box* proposta, le probabilità che in essa ci sia o meno un oggetto.

A partire dai *R-CNN*, per arrivare ai modelli *Faster R-CNN*, la maggior parte dei singoli moduli di un sistema per il riconoscimento di oggetti - quali il *proposal-detection*, il *feature-extraction*, il *bounding-box-regression*, etc. - sono stati gradualmente integrati in un'unico *framework end-to-end*.

Sebbene *Faster R-CNN* superi di molto la velocità di *Fast R-CNN*, c'è ancora ridondanza nella computazione. Successivamente sono stati infatti introdotti una gran varietà di miglioramenti alla rete, tra cui *RFCN* [4] e *Light head RCNN* [15].

3.2.5 YOLO (one-stage detector)

YOLO [23] venne proposto da *J. Redmon et al.* nel 2015. È stato il primo *one-stage detector* nell'era del *Deep Learning*. *YOLO* è estremamente veloce: una versione veloce di *YOLO* funziona a 155 fps con una *mAP* = 52.7% su *VOC07*, mentre una versione più accurata funziona a 45 fps con una *mAP* = 63.4% su *VOC07*.

Il nome *YOLO* è l'acronimo di *"**Y**ou **O**nly **L**ook **O**nce"*. Già lo stesso nome lascia effettivamente intendere che gli autori hanno completamente abbandonato il paradigma preesistente di *"proposal detection + verification"*. Al suo posto *YOLO* segue una filosofia completamente differente: applicare un singolo modello all'intera immagine. *YOLO* infatti divide l'immagine in regioni, predice le *bounding box* e, per ciascuna di esse, determina le probabilità di appartenere ad una certa classe, il tutto utilizzando un'unica rete.

In un secondo momento *J. Redmon* fece una serie di miglioramenti alla rete, rilasciandone una seconda [21] ed una terza versione [22]. Queste nuove versioni hanno un'accuratezza migliore nel riconoscimento, pur mantenendo un'altissima velocità di esecuzione.

Nel prossimo capitolo ("4. *YOLO: You Only Look Once*") ci soffermeremo meglio sull'architettura e sulle caratteristiche di *YOLO*.

Ad ogni modo, nonostante il grandissimo miglioramento nella velocità di riconoscimento, *YOLO* soffre di una ridotta accuratezza nella localizzazione degli oggetti, se comparato ai *two-stage detector*, specialmente per quel che concerne gli oggetti più piccoli.

Gli sviluppatori delle versioni successive di *YOLO*, e successivamente di *SSD* [18], hanno concentrato la loro attenzione proprio su questo aspetto.

3.2.6 SSD (one-stage detector)

***S**ingle-**S**hot **D**etector*, o *SSD* [18], fu proposto da *W. Liu et al.* nel 2015. Fu il secondo *one-stage detector* nell'era del *deep learning*.

Il principale contributo di *SSD* è stato il cambio di prospettiva nei confronti della generazione delle *bounding box*: a differenza dei modelli precedenti che si preoccupavano di predire con esattezza la locazione di un oggetto all'interno dell'immagine, *SSD* parte da un insieme di *bounding box* di *default*. A partire da questo insieme, *SSD* predice, per ciascuna di queste *bounding box* di *default*, uno scostamento. Per ciascuna *bounding box*, traslata dello scostamento predetto, il modello effettua quindi la classificazione. Grazie all'utilizzo di filtri diversi, scelti in base alle proporzioni dell'immagine (*i.e. "larghezza"/"altezza"*), e grazie all'applicazione simultanea di tali filtri a *feature map* "multiple", ognuna ottenuta in un punto diverso dei livelli convoluzionali, *SSD* ottiene anche un'ottima accuratezza nella predizione delle classi di oggetti.

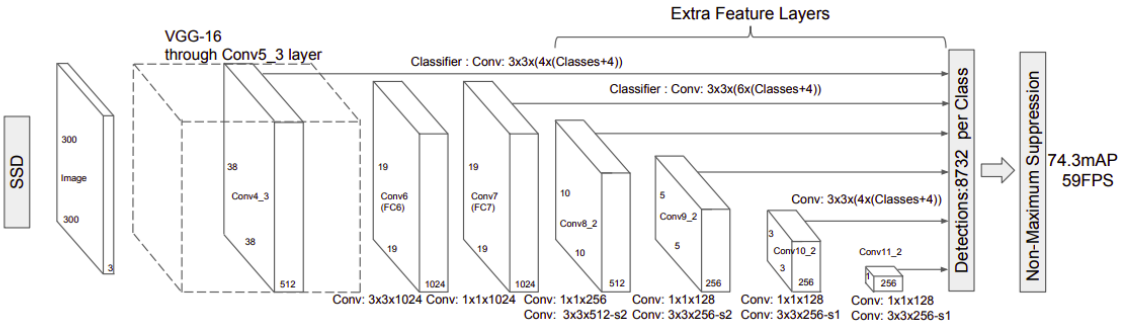

FIGURA 3.7: **Architettura di rete di *SSD*.** [18]

L'evitare la fase di predizione delle *bounding box* consente al modello *SSD* di risparmiare molto tempo d'esecuzione. L'utilizzo di *feature map* diverse invece migliora significativamente l'accuratezza del modello su immagini a bassa risoluzione, specialmente nel riconoscimento di piccoli oggetti.

SSD ha quindi vantaggi sia nella velocità che nell'accuratezza del riconoscimento (*VOC07 mAP=76.8%, VOC12 mAP=74.9%, COCO mAP@.5=46.5%, mAP@[.5,.95]=26.8%*), funzionando a 59 fps in una sua versione più veloce.

La differenza principale tra i modelli *SSD* ed i riconoscitori precedenti è che *SSD* riconosce oggetti di differenti dimensioni in diversi livelli della rete, mentre prima il riconoscimento era eseguito solo negli ultimissimi livelli.

3.2.7 (Feature) Pyramid Networks

Nel 2017 *T. Lin et al.* proposero *Feature Pyramid Networks (FPN [16])*. Prima di *FPN* la maggior parte dei riconoscitori basati su *deep learning* eseguivano il riconoscimento solo sulle *features* ottenute dagli ultimi livelli della rete.

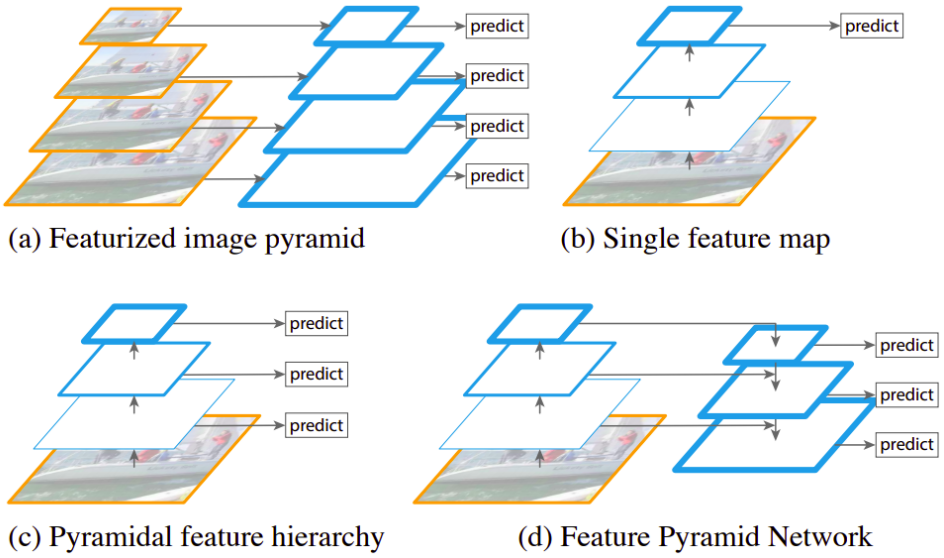

FIGURA 3.8: **Diversi approcci per l'estrazione di *feature map* in più risoluzioni.** Nell'immagine le *feature map* sono rappresentate come sezioni di piano con contorno azzurrino. Lì dove il contorno è più spesso ci si riferisce a *feature map* con un valore semantico più forte. [16]

FPN sfrutta il concetto di *"piramidi"* di *features*: prima dell'era del *deep learning* era un concetto largamente utilizzato, secondo cui, per estrarre le *features* da un'immagine, è utile partire dalla stessa immagine in più risoluzioni (si veda figura 3.8, (*a*)).

Successivamente, con l'avvento delle reti convoluzionali, questo modello è stato abbandonato: essendo queste più robuste e flessibili rispetto ai modelli precedenti, è possibile ottenere le *features* a partire da un'unica risoluzione di input (si veda figura 3.8 (*b*)). Nonostante la robustezza di questi modelli, questo metodo si è rilevato relativamente carente nell'accuratezza delle predizioni.

Per rendere le *feature map* più espressive si è pensato di effettuare le predizioni non solo sulle *features* ottenute dall'ultimo livello convoluzionale, ma anche su quelle ottenute da quelli intermedi (si veda figura 3.8 (*c*)). Ciò permette di fare predizioni su *feature map* che rappresentano l'immagine in diverse risoluzioni. Tuttavia questo approccio porta ad effettuare predizioni anche su *feature map* di scarso valore semantico, compromettendo l'accuratezza del modello.

La soluzione adottata da *T. Lin et al.* consiste invece nel generare nuove *feature map* che conservino il valore semantico di quelle di minor risoluzione.

Si parte appunto dalla *feature map* di minore risoluzione e maggior valore semantico, ossia da quella ottenuta nell'ultimo livello convoluzionale (*i.e.* la punta della piramide). A questa viene concatenata la *feature map* del livello successivo della piramide, che ha una risoluzione maggiore ma minor valore semantico. Si procede

così fino ad arrivare alla *feature map* che costituisce la base della piramide. Le predizioni vengono effettuate su ciascuna delle mappe di *features* così ottenute.

FPN può dunque contare su un'architettura che combina *features* di bassa risoluzione ma forte valore semantico, con *features* ad alta risoluzione ma scarso valore semantico. Questo grazie ad un percorso *top-down* che discende la piramide delle *features* combinandole tra loro con l'utilizzo di connessioni laterali (si veda figura 3.8 (*d*)). Il risultato è una piramide di *features*, ricca di semantica ad ogni livello, costruita velocemente a partire da un'immagine, senza che questa debba essere ridimensionata.

Usando *FPN* in combinazione con il modello *Faster R-CNN*, si ottengono ottimi risultati rispetto allo stato dell'arte (*COCO mAP@.5=59.1%, COCO mAP@[.5,.95]=36.2%*).

FPN è diventato una delle componenti alla base di molti riconoscitori nella *computer vision*.

3.2.8 Retina-Net (one-stage detector)

Nonostante l'alta velocità e la loro semplicità, i riconoscitori *"one-stage"* hanno sempre avuto problemi nell'accuratezza delle predizioni. *T. Lin et al.* hanno individuato alcune delle ragioni che si celano dietro questo *gap* ed hanno dunque proposto *RetinaNet* [17] come soluzione.

Essi sostennero che l'estremo squilibrio tra queste classi di riconoscitori fosse dovuto alla fase di addestramento dei livelli interamente connessi. Per ovviare a questo problema, gli autori introdussero in *RetinaNet* una nuova *loss function* chiamata *focal loss*. Questa nuova funzione è definita in modo tale da far risultare più influenti in fase di addestramento gli esempi difficili e quelli mal classificati.

Focal Loss Function. La *focal loss function* introdotta da *T. Lin et al.* è costruita a partire dalla *cross entropy loss function*. Quest'ultima è definita come segue:

$$CE(y, \hat{y}) = \begin{cases} -\log(\hat{y}) & \text{se } y = 1 \\ -\log(1-\hat{y}) & \text{altrimenti} \end{cases}$$

Ridefinendo $\hat{y}$ se ne può ottenere una versione semplificata:

$$\hat{y}' = \begin{cases} \hat{y} & \text{se } y = 1 \\ (1-\hat{y}) & \text{altrimenti} \end{cases}$$

$$CE(y, \hat{y}) = CE'(y, \hat{y}') = -\log(\hat{y}')$$

All'errore ottenuto da questa funzione, *T. Lin et al.* moltiplicano un fattore $\alpha_{example}$ - dipendente dall'esempio d'apprendimento - con lo scopo di bilanciare l'errore ottenuto da esempi positivi con quello predominante degli esempi negativi.

Tuttavia la funzione così definita ancora non fa differenza tra esempi facili e difficili da classificare.

Per implementare, ciò la *loss function* è stata ulteriormente modificata. Al fine di ridurre il valore dell'errore calcolato sulla classificazione di esempi facili e di aumentare quello derivante dalla classificazione di esempi difficili, *T. Lin et al.* introducono un ulteriore fattore definito come $(1 - \hat{y}')^{\gamma}$, con un parametro γ detto di *focusing* o di concentrazione.

Il risultato finale è la *focal loss function* così definita:

$$FL(\hat{y}') = -\alpha_{example}(1 - \hat{y}')^{\gamma} \log(\hat{y}')$$

Con la *focal loss function* di *Retina-Net* è possibile ottenere un'accuratezza paragonabile a quella dei riconoscitori *two-stage*, pur mantenendo la grande velocità di esecuzione che contraddistingue i modelli *one-stage* (*COCO mAP=59.1%*).

3.3 Ambiti d'applicazione dell'Object Detection

3.3.1 Videosorveglianza

Con l'aumento prestazionale ed il miglioramento nell'accuratezza dei modelli di *object detection*, di recente c'è stato un *boom* nell'utilizzo di modelli di *deep learning* per semplificare ed automatizzare l'analisi dei video di sorveglianza.

Più precisamente, molto utilizzata nella videosorveglianza è la tecnologia del *object tracking*[5].

I sistemi di *object tracking* si basano proprio sull'*object detection*: tengono conto non solo della collocazione spaziale di un oggetto in un dato momento, ma si preoccupano anche di tenerne traccia nel tempo - *i.e. si analizza l'output dell'object detection applicato a diversi frame di un video*.

3.3.2 Astronomia

Come in molte scienze, uno dei punti cardine dell'astronomia è senza dubbio l'osservazione. Ecco che i progressi nel *deep learning* hanno dato una marcia in più anche allo studio del cosmo. Sono diverse le applicazioni di reti neurali nel campo dell'astronomia: si va dal puro riconoscimento di entità astronomiche, alla rielaborazione di immagini o di dati.

Uno dei grandi problemi moderni dell'astronomia è appunto l'enorme quantità di dati raccolta dai vari strumenti elettronici. Progetti come lo *Square Kilometre Array (o SKA, si veda «Co-hosting the Square Kilometre Array»)* prevedono l'analisi e l'elaborazione di quantità impressionanti di dati - si parla di flussi di più di un *exabyte* giornalieri, più del totale del traffico mondiale di dati via internet [3]. L'utilizzo del *deep learning* sta aiutando molto in tal senso, grazie a modelli come quelli per il *"salient object detection"* o l'*"event detection"*.

[5] *"Object tracking is the process to track the object over the time by locating its position in every frame of the video in surveillance system. It may also complete region in the image that is occupied by the object at every time instant."* [10]

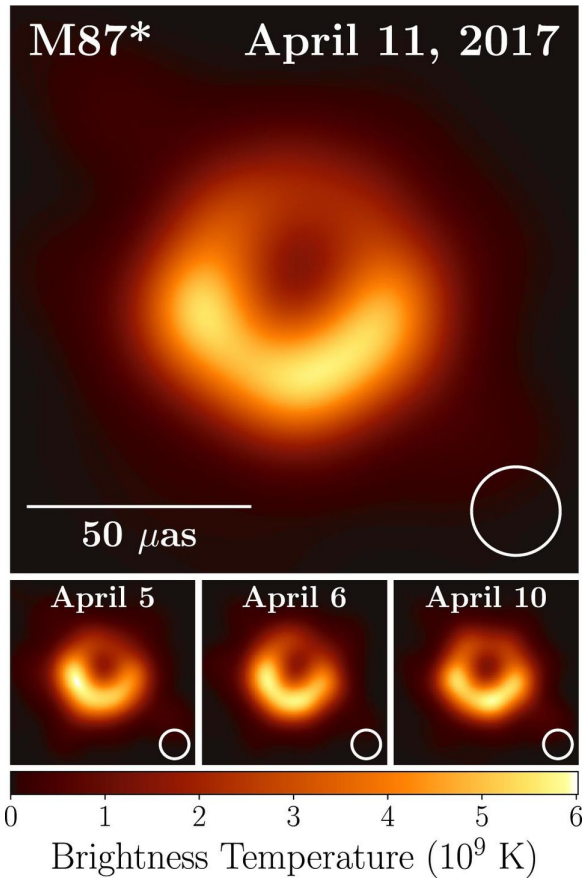

FIGURA 3.9: **Le immagini generate che rappresentano il buco nero M87.** Dopo l'assemblamento dei dati provenienti da diversi strumenti e luoghi, l'immagine è stata ripulita e la risoluzione aumentata grazie all'utilizzo di reti neurali (*Immagine ottenuta da «10 Deep Lessons From Our First Image Of A Black Hole's Event Horizon»*).

Un celebre esempio dell'innovazione apportata dal *deep learning* al mondo dell'astronomia si nasconde dietro l'elaborazione della prima immagine di un buco nero. Per generare quest'immagine astronomi e scienziati hanno messo insieme 5 *petabytes* di dati, ottenuti da un sistema di telescopi, ottici e radiofonici, progettato proprio per questo scopo: l'*Event Horizon Telescope array*.

La prima fase dell'elaborazione dei dati consiste nel selezionare, tra l'immensa quantità di dati, quelli relativi ad un preciso range di tempo, nell'ordine di pochi secondi. La seconda consiste invece nel mettere assieme i dati in modo da ottenere una rappresentazione fedele della realtà. Ed è da questa fase che entra in gioco il *deep learning*: modelli appositamente progettati vengono utilizzati per confrontare i dati ottenuti da diversi punti di vista, per ripulirli dal *rumore*, e per selezionare quelli realmente interessanti. L'ultima fase è quella in cui tutti i dati vengono assemblati e viene generata un'unica immagine in output.

Un altro esempio degno di nota è il *CMU DeepLens*, un modello di *supervised machine learning* costruito al fine di individuare autonomamente nuove galassie (*si veda «CMU DeepLens: deep learning for automatic imagebased galaxy–galaxy strong lens finding»*).

Citiamo infine un altro esempio importante dell'applicazione del *deep learning* in astronomia: si tratta del modello per l'individuazione e lo studio delle onde gravitazionali[6] (*si veda «Deep Learning for real-time gravitational wave detection and parameter estimation: Results with Advanced LIGO data»*).

3.3.3 Riconoscimento facciale

Ormai tutti i software moderni per il riconoscimento facciale si basano su modelli di *deep learning*. Un importante modello per il riconoscimento facciale è *DeepID3* [28], sviluppato nel 2015 a partire dalle sue versioni precedenti, ha ottenuto un grande successo grazie al basso contenuto di falsi positivi ed alla sua efficienza.

3.3.4 Applicazioni Mediche

Buona parte della ricerca sul *deep learning* si concentra proprio sulle possibili applicazioni mediche. Negli ultimi anni sono sempre di più i software che si propongono di essere uno strumento in più per medici e ricercatori del settore.

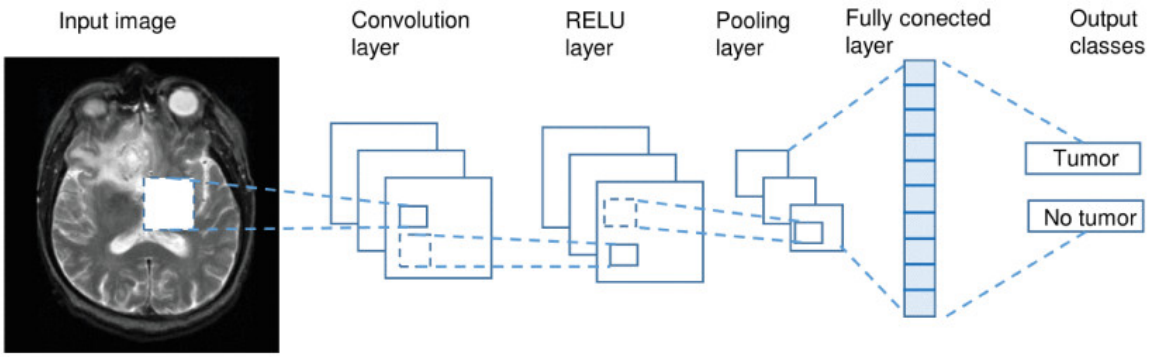

FIGURA 3.10: **Esempio di *CNN* per il riconoscimento di tumori.** L'estrazione delle *features* dall'immagine in input avviene attraverso i livelli convoluzionali, *RELU* e di *pooling*, la classificazione dal livello interamente connesso. *Immagine ottenuta da «Deep Learning Applications in Medical Image Analysis»*

Tra le diverse applicazioni citiamo modelli in grado di individuare tessuti cancerogeni, di riconoscere patologie a partire da dati biometrici o dalle descrizioni dei pazienti. Interessante è anche l'applicazione nell'ambito dello studio ambientale dei casi dell'insorgere di particolari patologie e/o malattie, reso possibile soprattutto grazie all'enorme mole di dati ottenuti dai dispositivi mobili.

3.3.5 Robotica e guida autonoma

Sono molti i modelli per l'*object detection* che sono stati sviluppati e su cui si lavora che si occupano di guida autonoma o, più in generale, di generare schemi comportamentali che rispondano a particolari esigenze ambientali.

[6] sono un importante fenomeno per comprendere gli equilibri e le leggi dell'universo. Sono prodotte da importanti eventi cosmici come la fusione di due stelle di elettroni.

Si tratta per lo più di reti ibride molto complesse, che si occupano prima di interpretare e capire l'ambiente e successivamente di scegliere quale risposta sia più opportuna nella circostanza rilevata.

Tratteremo più in avanti un modello di riconoscimento di oggetti in ambiente tridimensionale per lo sviluppo di modelli di guida autonoma.

Capitolo 4

YOLO: You Only Look Once

Presentiamo adesso la rete *"**Y**ou **O**nly **c**ook **O**nce"*, pubblicata nel 2016 da *Joseph Redmond et al.* [23]. L'architettura di *YOLO* si ispira a quella di *GoogLeNet* [29].

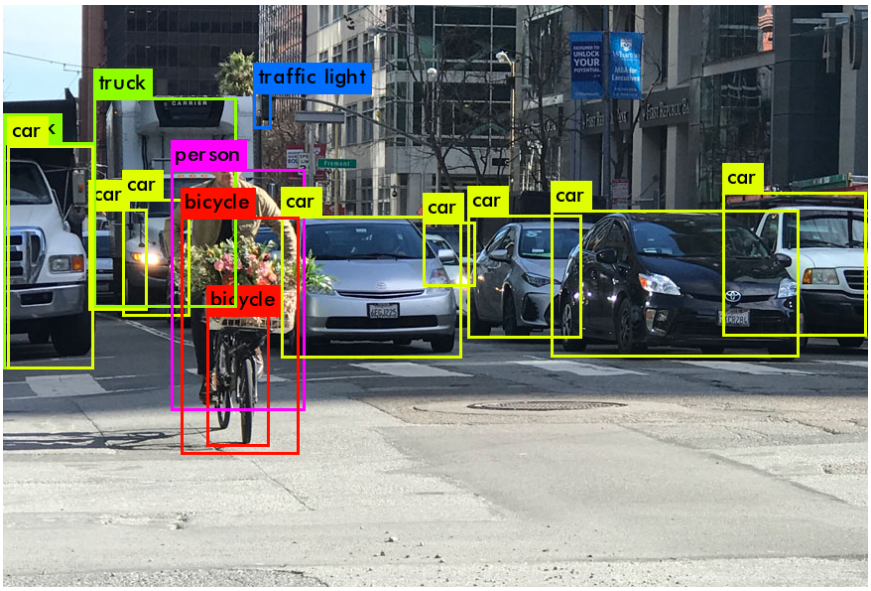

FIGURA 4.1: **Esempio dell'output di *YOLO***. [9]

Rispetto ai modelli di *object detection* preesistenti, *YOLO* è decisamente più veloce.

Questo è possibile principalmente grazie al fatto che *YOLO* non divide il riconoscimento in più fasi, ma predice *bounding box*[1], probabilità e classi degli oggetti presenti nell'immagine di input in un'unica fase.

Rispetto agli altri sistemi di *object detection*, *YOLO* commette più errori di localizzazione, ma al contempo è meno propenso a riconoscere falsi-positivi sullo sfondo dell'immagine, oltre ad essere notevolmente più veloce.

[1] letteralmente *"scatola di confine"*, è un rettangolo il cui centro è il centro dell'oggetto riconosciuto (vedi Fig. 4.4).

4.1 Progettazione ed architettura

In questa sezione parliamo delle scelte progettuali e dell'architettura di *YOLO*, cercando di soffermarci maggiormente sulle sue peculiarità.

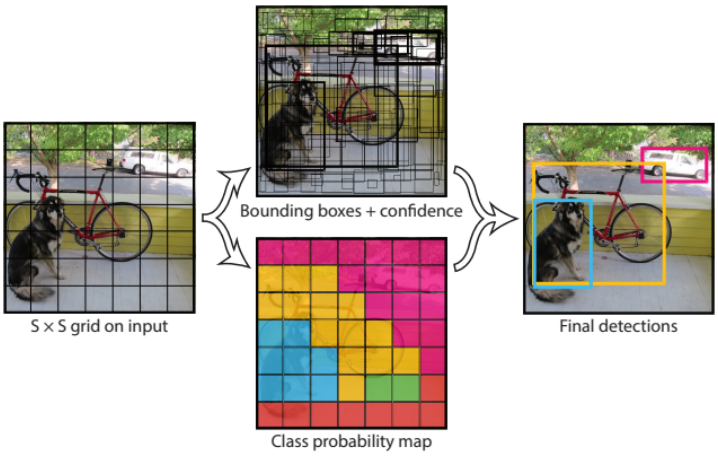

FIGURA 4.2: **Fasi dell'esecuzione di *YOLO*.** Il modello divide l'immagine in $S * S$ celle; per ogni cella predice B *bounding box*, ciascuna con un *confidence score* e C classi con le relative probabilità. [23]

4.1.1 Riconoscimento e classificazione unificati

YOLO è il primo sistema *"one-stage"* di *object detection*. *J. Redmon et al.* presentano una soluzione innovativa al problema dell'*object detection*, unificando le diverse componenti dei sistemi di *OD* in un'unica rete. Ciò fa sì che la rete ragioni contemporaneamente sull'intera immagine invece che su delle regioni specifiche, garantendo una maggiore comprensione dell'ambiente in cui le diverse classi di oggetti si trovano. Inoltre, così facendo, crea un legame implicito tra classi di oggetti effettivamente correlate.

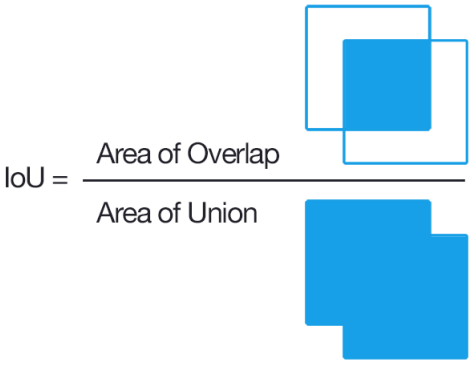

FIGURA 4.3: **Rappresentazione della metrica IOU[2].** (*Immagine ottenuta da «Intersection over Union (IoU) for object detection»*)

Fissati S e B, il modello divide l'immagine in una griglia di $S * S$ celle. La cella della griglia contenente il centro di un oggetto è definita come la cella responsabile per lo stesso (si veda figura 4.4). Ogni cella della griglia predice B *bounding box* ed assegna a ciascuna di esse un punteggio chiamato *"Confidence Score"*. Questo punteggio indica quante probabilità ci sono che la *bounding box* contenga un oggetto ($Pr(Object)$) ed anche quanto si ritiene che siano accurate le dimensioni e la localizzazione della *bounding box* stessa rispetto all'oggetto (IOU_{pred}^{truth} [2]).

Ogni *bounding box* (di ciascuna cella) è predetta dal modello con cinque valori: x, y, w, h ed il *confidence score* "$Pr(Object) * IOU_{pred}^{truth}$", dove (x, y) sono le coordinate del centro della *bounding box* rispetto ai bordi della cella della griglia, mentre w e h rappresentano rispettivamente larghezza (*width*) ed altezza (*height*) della *bounding box*, relativamente però all'intera immagine. Ogni cella della griglia, a prescindere dal numero B di *bounding box* generate, predice C classi con la relativa probabilità condizionata ($Pr(Class_i|Object)$).

A questo punto il modello, per ogni *bounding box*, moltiplica le probabilità condizionate delle classi (della cella) con il *confidence score* della *bounding box*, ottenendo così un *confidence score* specifico di ogni classe per ogni *bounding box*:

$$Pr(Class_i|Object) * Pr(Object) * IOU_{pred}^{truth} = Pr(Class_i) * IOU_{pred}^{truth}$$

Il valore così ottenuto codifica sia le probabilità che un oggetto di una data classe compaia nel *bounding box* in esame, che la precisione con cui la *bounding box* predetta delimita i contorni spaziali dell'oggetto.

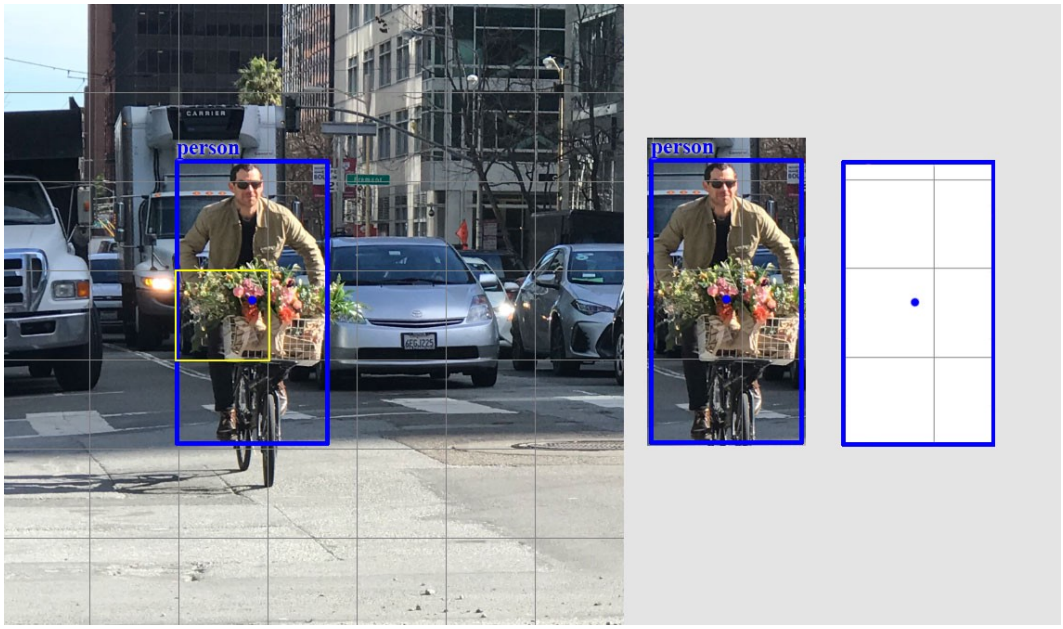

FIGURA 4.4: **Esempio di *bounding box* predetta da *YOLO*.** Ogni cella della griglia riconosce, alla fine del processo, un unico oggetto, ossia quello per cui è responsabile (*i.e. quello il cui centro è all'interno della griglia*). [9]

[2] ***I**ntersection **O**ver **U**nion* è una metrica utilizzata nei sistemi di *object detection* per determinare lo scostamento tra le *bounding box* previste dal modello e quelle reali.

TABELLA 4.1: **Architettura di *YOLO* nel dettaglio.** [19]

Name	Filters	Output Dimension
conv 1	7 x 7 x 64, stride=2	224 x 224 x 64
Conv 1	7 x 7 x 64, stride=2	224 x 224 x 64
Max Pool 1	2 x 2, stride=2	112 x 112 x 64
Conv 2	3 x 3 x 192	112 x 112 x 192
Max Pool 2	2 x 2, stride=2	56 x 56 x 192
Conv 3	1 x 1 x 128	56 x 56 x 128
Conv 4	3 x 3 x 256	56 x 56 x 256
Conv 5	1 x 1 x 256	56 x 56 x 256
Conv 6	1 x 1 x 512	56 x 56 x 512
Max Pool 3	2 x 2, stride=2	28 x 28 x 512
Conv 7	1 x 1 x 256	28 x 28 x 256
Conv 8	3 x 3 x 512	28 x 28 x 512
Conv 9	1 x 1 x 256	28 x 28 x 256
Conv 10	3 x 3 x 512	28 x 28 x 512
Conv 11	1 x 1 x 256	28 x 28 x 256
Conv 12	3 x 3 x 512	28 x 28 x 512
Conv 13	1 x 1 x 256	28 x 28 x 256
Conv 14	3 x 3 x 512	28 x 28 x 512
Conv 15	1 x 1 x 512	28 x 28 x 512
Conv 16	3 x 3 x 1024	28 x 28 x 1024
Max Pool 4	2 x 2, stride=2	14 x 14 x 1024
Conv 17	1 x 1 x 512	14 x 14 x 512
Conv 18	3 x 3 x 1024	14 x 14 x 1024
Conv 19	1 x 1 x 512	14 x 14 x 512
Conv 20	3 x 3 x 1024	14 x 14 x 1024
Conv 21	3 x 3 x 1024	14 x 14 x 1024
Conv 22	3 x 3 x 1024, stride=2	7 x 7 x 1024
Conv 23	3 x 3 x 1024	7 x 7 x 1024
Conv 24	3 x 3 x 1024	7 x 7 x 1024
FC 1	-	4096
FC 2	-	7 x 7 x 30 (1470)

4.1.2 Design di rete

L'architettura di *YOLO* si ispira molto a quella di *GoogLeNet* [29]. *YOLO* ha 24 livelli convoluzionali, seguiti da 2 livelli interamente interconnessi.

YOLO utilizza una funzione d'attivazione lineare per l'ultimo livello, mentre tutti gli altri livelli utilizza la seguente funzione d'attivazione:

$$\phi(x) = \begin{cases} x & \text{se } x > 0 \\ 0.1x & \text{altrimenti} \end{cases}$$

Come mostrato in figura 4.5 e nella tabella 4.1, *YOLO* ha un livello di output di dimensione 7x7x30.

J. Redmon et al. propongono inoltre una versione alleggerita della rete, *Fast YOLO*,

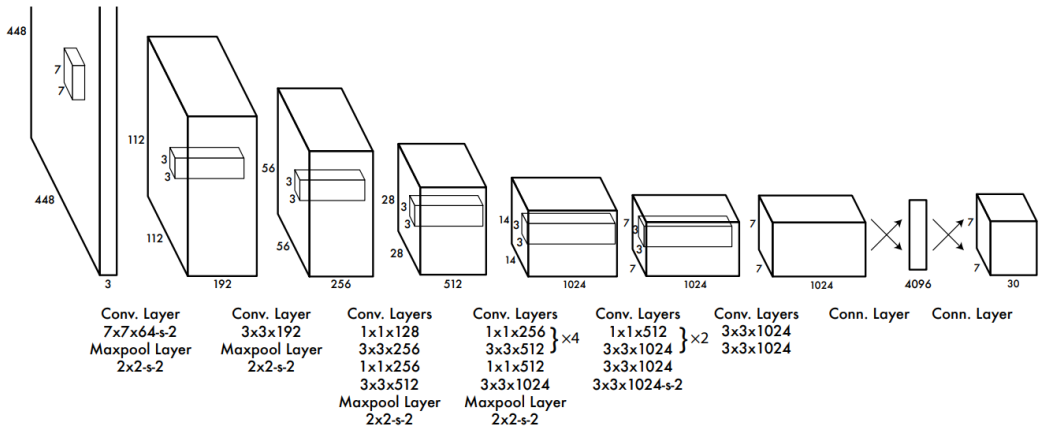

FIGURA 4.5: **Rappresentazione dell'architettura di *YOLO*.** *YOLO* ha 24 livelli convoluzionali seguiti da 2 livelli interamente connessi. Da notare l'alternanza di livelli convoluzionali di dimensione 1x1, usati per ridurre lo spazio delle attivazioni ottenute dai livelli precedenti.

ancora più veloce ma meno accurata. Quest'ultima presenta solo 9 livelli convoluzionali iniziali, invece di 24, tra i quali ci sono, tra l'altro, meno filtri[3].

4.1.3 Addestramento

L'addestramento di *YOLO*, così come concepito dagli autori, si divide in due fasi: una prima fase di *pre-training*, durante la quale solo i livelli più profondi vengono addestrati, ed una seconda fase di addestramento che coinvolge l'intera rete.

Pre-Training

Gli autori di *YOLO* iniziano l'addestramento allenando solo i primi 20 livelli convoluzionali a cui aggiungono, un livello di *average-pooling*[4] ed uno interamente connesso. Per tale scopo utilizzano il *dataset* della competizione *ImageNet 1000-class* con immagini in input di dimensione 224x224 pixel.

Hanno addestrato il modello per circa una settimana ottenendo un'accuratezza dell'88% circa sul *validation set* di *ImageNet 2012*.

Training

A questo punto *J. Redmon et al.* hanno eliminato i due ultimi livelli della rete - aggiunti appositamente per la prima parte dell'addestramento - per aggiungere altri 4 livelli convoluzionali, seguiti infine da 2 livelli interamente connessi.

[3] livelli "filtro" sono spesso interposti tra due livelli di una rete neurale per comprimere e mantenere bassi i valori delle attivazioni; in *YOLO*, ad esempio, ci sono 4 livelli di filtro *Maxpool*.

[4] è una tipologia di livello utilizzata per ridurre il flusso di dati nella propagazione in avanti di una rete neurale convoluzionale.

Essendo che i modelli per l'*object detection* richiedono informazioni precise per migliorare nell'accuratezza e nella generalità, il modello viene questa volta addestrato con immagini di risoluzione doppia *(i.e.* 448*x*448 *pixel)*.

La seconda fase di *training*, così come pensata dagli autori, prevede 135 epoche che utilizzano sia il *training* che il *validation set* delle competizioni *PASCAL VOC 2007* e *PASCAL VOC 2012*. Durante l'addestramento sono stati usati una dimensione di *batch*[5] di 64, un *momentum* di 0.9 ed un *decay* di $5 * 10^{-4}$.

Per ridurre l'*overfitting*[6] il modello ha un livello di *dropout* con tasso di riduzione del 50% posto dopo il primo livello interamente connesso per impedire la *co-adaptation* tra i livelli interamente connessi. Sempre al fine di ridurre l'*overfitting*, sono state introdotte anche tecniche di *data augmentation*: il modello prima di usare le immagini prese in input, le processa aggiungendo traslazioni e cambiando le proporzioni dell'immagine fino ad un 20% rispetto alle caratteristiche originali. Inoltre le immagini vengono modificate cambiando esposizione e saturazione di un fattore compreso tra 0 e 1.5.

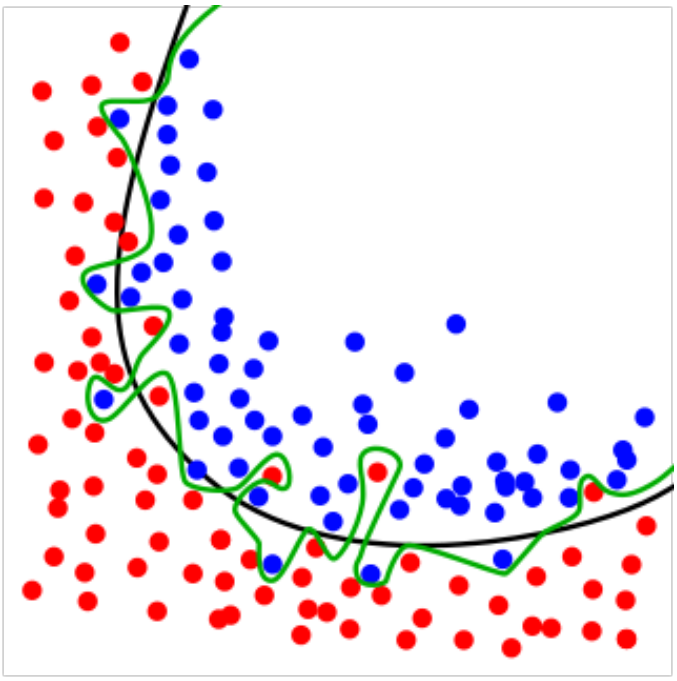

FIGURA 4.6: **Esempio di *overfitting*.** Mentre la linea nera rappresenta bene la distribuzione dei dati, la linea verde presenta un problema di *overfitting*. *(Immagine ottenuta da wikipedia.org)*

Il livello finale della rete predice sia le probabilità delle classi che le coordinate delle *bounding box* per gli oggetti riconosciuti. Il modello normalizza la larghezza e l'altezza delle *bounding box* rispetto alla dimensione dell'immagine presa in input in modo tale che i relativi valori ricadano nell'intervallo reale compreso tra zero e uno.

[5] è il numero di immagini passate come input al modello in ciascun passo d'apprendimento. Per processare più immagini contemporaneamente si usa il concetto di *vettorializzazione*.

[6] è la produzione di un'analisi che corrisponde in modo troppo fedele ad un particolare insieme di dati, al punto di rischiare di fraintendere dati provenienti da altre fonti.

Il centro delle *bounding box* è a sua volta normalizzato ed è espresso in funzione della cella (della griglia) d'appartenenza affinché anch'esso risulti appartenente all'intervallo tra zero e uno.

Per quel che riguarda il *learning rate* (o tasso di apprendimento), gli autori hanno notato che il modello spesso diverge verso gradienti instabili, se addestrato dall'inizio con alti livelli di questo parametro. Per questo motivo il *learning rate* parte da un valore basso, 10^{-3}, seguendo poi il seguente schema:

- il tasso aumenta lentamente nelle prime epoche fino ad arrivare al valore di 10^{-2};
- rimane costante fino all'epoca 75;
- per le successive 30 epoche si applica un tasso di 10^{-3};
- per le ultime 30 epoche si usa un *learning rate* di 10^{-4}.

Loss Function

Vista la complessità del modello, è opportuno scegliere con attenzione la funzione che calcola l'errore. Gli autori hanno deciso di utilizzare un errore quadratico in quanto facile da ottimizzare; tuttavia questa scelta non si addice molto all'esigenza di massimizzare la precisione media nelle predizioni. Con questa funzione infatti, gli errori di classificazione vengono posti sullo stesso piano di quelli di localizzazione. In più, in ogni immagine, molte celle della griglia possono non contenere nessun oggetto. Questo spinge il *confidence score* di queste celle verso lo zero, rendendo spesso più influenti del dovuto i gradienti ottenuti da quest'ultime. Tutto ciò, come spiegato dagli autori, può causare una maggiore instabilità del modello, portandolo a divergere.

Per rimediare a questo problema, gli autori aumentano il gradiente ottenuto dalle *bounding box* che contengono oggetti e riducono quello ottenuto dalle *bounding box* che non ne contengono. Per farlo, usano due parametri, rispettivamente λ_{coord} e λ_{noobj}. Di default questi parametri valgono $\lambda_{coord} = 5$ e $\lambda_{noobj} = 0.5$.

La *loss function* di *YOLO* si compone di:

- la ***classification loss***;
- la ***localization loss*** (l'errore spaziale tra la *bounding box* predetta e quella reale);
- la ***confidence loss*** (l'errore generale della *bounding box*, ovvero l'errore nel *confidence score* calcolato).

1. Classification Loss

Calcola l'errore nella classificazione di un oggetto specifico; indica ossia se la classe riconosciuta corrisponde o meno all'oggetto individuato nell'immagine. Data una cella della griglia, se questa viene considerata responsabile di un certo oggetto, allora è responsabile anche della sua classificazione. L'errore in questa operazione

è dato dalla funzione quadratica che calcola le probabilità condizionate della classe riconosciuta rispetto ad ogni classe:

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

dove $\mathbb{1}_i^{obj}$ vale 1 se un oggetto compare nella *i-esima* cella, 0 altrimenti;
$\hat{p}_i(c)$ denota le probabilità condizionate della *c-esima* classe di comparire nell'*i-esima* cella.

2. Localization Loss

Misura l'errore nella predizione delle *bounding box*, ovvero dell'altezza, della larghezza e delle coordinate del centro. *YOLO* calcola quest'errore solo per le *bounding box* responsabili di riconoscere un oggetto.
L'errore nella localizzazione è calcolato come segue:

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

dove $\mathbb{1}_{ij}^{obj}$ vale 1 se la *j-esima bounding box* dell'*i-esima* cella è responsabile per il riconoscimento di un oggetto, 0 altrimenti.
Notiamo che l'errore rispetto alle dimensioni della *bounding box* è calcolato sulla radice quadrata dei valori. Questo al fine di non dare lo stesso peso agli errori, a prescindere dalla dimensione della *bounding box*: per esempio un errore di 2 pixel potrebbe essere trascurabile se riscontrato su di una *bounding box* di 500 pixel, ma importante su una *bounding box* di 30 pixel.

3. Confidence Loss

Serve a quantificare l'obiettività della predizione fatta su di una singola *bounding box*. Se, data una *bounding box*, il modello vi ci riconosce un oggetto, la *confidence loss* è calcolata come segue:

$$\sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} (C_{ij} - \hat{C}_{ij})^2$$

dove $\mathbb{1}_{ij}^{obj}$ vale 1 se la *j-esima bounding box* dell'*i-esima* cella è responsabile per il riconoscimento di un oggetto, 0 altrimenti;
$\hat{C}_{ij}$ è il *confidence score* della *j-esima bounding box* dell'*i-esima* cella.

La maggior parte delle *bounding box* non contengono oggetti. Questo causa uno sbilanciamento del modello, il quale in pratica si allena più frequentemente a riconoscere gli sfondi piuttosto che gli oggetti. Per rimediare a ciò, la *confidence loss* per

le *bounding box* che non contengono oggetti viene modificata da un fattore λ_{noobj}; di default $\lambda_{noobj} = 0.5$. Otteniamo quindi:

$$\lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{noobj} (C_{ij} - \hat{C}_{ij})^2$$

dove $\mathbb{1}_{ij}^{noobj}$ è il complementare di $\mathbb{1}_{ij}^{obj}$;
$\hat{C}_{ij}$ è il *confidence score* della *j-esima bounding box* dell'*i-esima* cella.

Loss Function

Visti tutti gli elementi della *loss function* di *YOLO*, ecco come si presenta la *loss function* che calcola l'errore nella localizzazione, nel *confidence score* e nella *classificazione*:

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

4.2 Sperimentazione

Presentiamo adesso quelle che sono state le sperimentazioni effettuate sul modello da parte degli autori.

Questi si sono dapprima soffermati sul confrontare *YOLO* con gli altri sistemi *real-time* di *object detection* allora esistenti.

Dopodiché hanno confrontato a fondo le differenze con i modelli *R-CNN*, nello specifico con *Fast R-CNN*, uno dei modelli di *object detection* più performanti.

Infine presentano i risultati sui *dataset* della competizione *VOC 2012*, mettendo a paragone i risultati ottenuti con gli altri metodi correntemente in uso.

4.2.1 Comparazione con altri sistemi di Object Detection

Visto che gli altri sistemi di *object detection*, non raggiungevano prestazioni *real-time*, *Joseph Redmon et al.* hanno confrontato i risultati dei test in termini di *trade-off* tra velocità ed accuratezza dei diversi modelli esistenti.

VOC 2012 test	mAP	aero	bike	bird	boat	bottle	bus	car	cat	chair	cow	table	dog	horse	mbike	person	plant	sheep	sofa	train	tv
MR_CNN_MORE_DATA [11]	**73.9**	**85.5**	**82.9**	**76.6**	**57.8**	**62.7**	**79.4**	77.2	86.6	**55.0**	**79.1**	**62.2**	87.0	**83.4**	**84.7**	78.9	45.3	73.4	65.8	80.3	74.0
HyperNet_VGG	71.4	84.2	78.5	73.6	55.6	53.7	78.7	**79.8**	87.7	49.6	74.9	52.1	86.0	81.7	83.3	**81.8**	**48.6**	**73.5**	59.4	79.9	65.7
HyperNet_SP	71.3	84.1	78.3	73.3	55.5	53.6	78.6	79.6	87.5	49.5	74.9	52.1	85.6	81.6	83.2	81.6	48.4	73.2	59.3	79.7	65.6
Fast R-CNN + YOLO	70.7	83.4	78.5	73.5	55.8	43.4	79.1	73.1	**89.4**	49.4	75.5	57.0	**87.5**	80.9	81.0	74.7	41.8	71.5	68.5	**82.1**	67.2
MR_CNN_S_CNN [11]	70.7	85.0	79.6	71.5	55.3	57.7	76.0	73.9	84.6	50.5	74.3	61.7	85.5	79.9	81.7	76.4	41.0	69.0	61.2	77.7	72.1
Faster R-CNN [28]	70.4	84.9	79.8	74.3	53.9	49.8	77.5	75.9	88.5	45.6	77.1	55.3	86.9	81.7	80.9	79.6	40.1	72.6	60.9	81.2	61.5
DEEP_ENS_COCO	70.1	84.0	79.4	71.6	51.9	51.1	74.1	72.1	88.6	48.3	73.4	57.8	86.1	80.0	80.7	70.4	46.6	69.6	**68.8**	75.9	71.4
NoC [29]	68.8	82.8	79.0	71.6	52.3	53.7	74.1	69.0	84.9	46.9	74.3	53.1	85.0	81.3	79.5	72.2	38.9	72.4	59.5	76.7	68.1
Fast R-CNN [14]	68.4	82.3	78.4	70.8	52.3	38.7	77.8	71.6	89.3	44.2	73.0	55.0	**87.5**	80.5	80.8	72.0	35.1	68.3	65.7	80.4	64.2
UMICH_FGS_STRUCT	66.4	82.9	76.1	64.1	44.6	49.4	70.3	71.2	84.6	42.7	68.6	55.8	82.7	77.1	79.9	68.7	41.4	69.0	60.0	72.0	66.2
NUS_NIN_C2000 [7]	63.8	80.2	73.8	61.9	43.7	43.0	70.3	67.6	80.7	41.9	69.7	51.7	78.2	75.2	76.9	65.1	38.6	68.3	58.0	68.7	63.3
BabyLearning [7]	63.2	78.0	74.2	61.3	45.7	42.7	68.2	66.8	80.2	40.6	70.0	49.8	79.0	74.5	77.9	64.0	35.3	67.9	55.7	68.7	62.6
NUS_NIN	62.4	77.9	73.1	62.6	39.5	43.3	69.1	66.4	78.9	39.1	68.1	50.0	77.2	71.3	76.1	64.7	38.4	66.9	56.2	66.9	62.7
R-CNN VGG BB [13]	62.4	79.6	72.7	61.9	41.2	41.9	65.9	66.4	84.6	38.5	67.2	46.7	82.0	74.8	76.0	65.2	35.6	65.4	54.2	67.4	60.3
R-CNN VGG [13]	59.2	76.8	70.9	56.6	37.5	36.9	62.9	63.6	81.1	35.7	64.3	43.9	80.4	71.6	74.0	60.0	30.8	63.4	52.0	63.5	58.7
YOLO	57.9	77.0	67.2	57.7	38.3	22.7	68.3	55.9	81.4	36.2	60.8	48.5	77.2	72.3	71.3	63.5	28.9	52.2	54.8	73.9	50.8
Feature Edit [33]	56.3	74.6	69.1	54.4	39.1	33.1	65.2	62.7	69.7	30.8	56.0	44.6	70.0	64.4	71.1	60.2	33.3	61.3	46.4	61.7	57.8
R-CNN BB [13]	53.3	71.8	65.8	52.0	34.1	32.6	59.6	60.0	69.8	27.6	52.0	41.7	69.6	61.3	68.3	57.8	29.6	57.8	40.9	59.3	54.1
SDS [16]	50.7	69.7	58.4	48.5	28.3	28.8	61.3	57.5	70.8	24.1	50.7	35.9	64.9	59.1	65.8	57.1	26.0	58.8	38.6	58.9	50.7
R-CNN [13]	49.6	68.1	63.8	46.1	29.4	27.9	56.6	57.0	65.9	26.5	48.7	39.5	66.2	57.3	65.4	53.2	26.2	54.5	38.1	50.6	51.6

FIGURA 4.7: **PASCAL VOC 2012 Leaderboard.** *YOLO* comparato con la *comp4 public leaderboard* del 6 novembre 2015. Tra quelli elencati, *YOLO* è l'unico *real-time detector*. *Fast R-CNN + YOLO*, che tratteremo più avanti, è il quarto miglior modello per *mAP*, con uno scarto del 2.3% rispetto *Fast R-CNN*. [23]

TABELLA 4.2: **Sintesi prestazioni di sistemi *real-time* addestrati con *PASCAL VOC 2007*.** [23]

Real-Time Detectors	**Train**	**mAP**	**FPS**
100Hz DPM	2007	16.0	100
30Hz DPM	2007	26.1	30
Fast YOLO	2007+2012	52.7	**155**
YOLO	2007+2012	**63.4**	45
Less Than Real-Time			
Fastest DPM	2007	30.4	15
R-CNN Minus R	2007	53.5	6
Fast R-CNN	2007+2012	70.0	0.5
Faster R-CNN VGG-16	2007+2012	73.2	7
Faster R-CNN ZF	2007+2012	62.1	18
YOLO VGG-16	2007+2012	66.4	21

Infine gli autori hanno provato ad addestrare *YOLO* usando i *dataset* di *VGG-16*, ottenendo un modello più accurato ma significativamente più lento. Questo al fine di paragonare in modo oggettivo *YOLO* con gli altri modelli di *object detection* addestrati su questo *dataset*.

La versione più veloce del modello *DPM*, *Fastest DPM*, effettivamente migliora la velocità - senza sacrificare troppo l'accuratezza del riconoscimento - ma concretamente è ancora troppo lento per essere considerato un vero sistema di riconoscimento di oggetti *real-time*. Inoltre *DPM* ha un'accuratezza relativamente bassa nella localizzazione, se confrontato con modelli interamente basati su reti neurali.

R-CNN minus r [14] sostituisce la *Selective Search* con un sistema di *static bounding box proposal*. Sebbene sia molto più veloce dell'originale modello *R-CNN*, non è ancora considerabile un sistema *real-time* e rischia di perdere molto in accuratezza se la distribuzione degli oggetti nell'immagine non si sposa bene con le *bounding box* predefinite.

Fast R-CNN [6] migliora le prestazioni della fase di classificazione di *R-CNN* - sia in termini di tempo che di accuratezza - ma utilizza ancora *Selective Search* per ottenere delle proposte di *bounding box*, fase che può richiedere fino a 2 secondi di tempo per ogni immagine, raggiungendo una velocità di circa 0.5 fps.

Il sistema *Selective Search* per ottenere proposte di *bounding box*, è stato rimpiazzato da una rete neurale con l'avvento di *Faster R-CNN* [24]. Nei test effettuati dagli autori di *YOLO*, il modello *Faster R-CNN* più accurato raggiunge i 7 fps, mentre una versione più piccola e meno accurata riporta una velocità di 18 fps. La versione *VGG-16* di *Faster R-CNN* ha un'accuratezza migliore di 10 punti percentuali in *mAP*, ma è anche 6 volte più lenta di *YOLO*.

La versione di *Zeiler-Fergus* di *Faster R-CNN* è solo 2.5 volte più lenta di *YOLO*, ma è anche meno accurata.

4.2.2 Analisi degli errori

J. Redmon et al. analizzano l'errore commesso dal modello *YOLO* addestrato su *dataset VOC 2007* paragonandolo a *Fast R-CNN*, uno dei riconoscitori più performanti sulle configurazioni *PASCAL*.

In fase di test, per ogni categoria di oggetti, si identificano le N migliori predizioni. Ogni categoria può risultare corretta, oppure viene classificata in base al tipo di errore:

- **Correct**: classe corretta, $IOU > 0.5$;
- **Localization**: classe corretta, $0.1 < IOU < 0.5$;
- **Similar**: classe simile, $IOU > 0.1$;
- **Other**: classe sbagliata, $IOU > 0.1$;
- **Background**: $IOU < 0.1$, a prescindere dalla classe riconosciuta;

La figura 4.8 mostra come mediamente si distribuiscano le diverse tipologie di errore su un campione di 20 classi di oggetti, confrontando i risultati dei test effettuati su *YOLO* e su *Fast R-CNN*.

È subito evidente che *YOLO* fa fatica nella localizzazione degli oggetti. Infatti gli errori di localizzazione compiuti da *YOLO* sono maggiori rispetto alla somma di tutti gli altri.

Fast R-CNN fa decisamente meno errori di localizzazione, ma soffre molto gli errori di *background*. Tant'è vero che il 13.6% delle predizioni di *Fast R-CNN* sono in realtà falsi-positivi, in effetti *bounding box* che non contengono alcun oggetto.

Al termine del confronto, *J. Redmond et al.* sottolineano difatti che *Fast R-CNN* è quasi tre volte più propenso a fare predizioni sullo sfondo di quanto non lo sia *YOLO*.

4.2.3 La forte generalizzazione di YOLO

Una delle caratteristiche migliori di *YOLO*, oltre alla velocità, è senza dubbio la generalizzazione dei dati: lo stesso modello, allenato su immagini catturate dal mondo

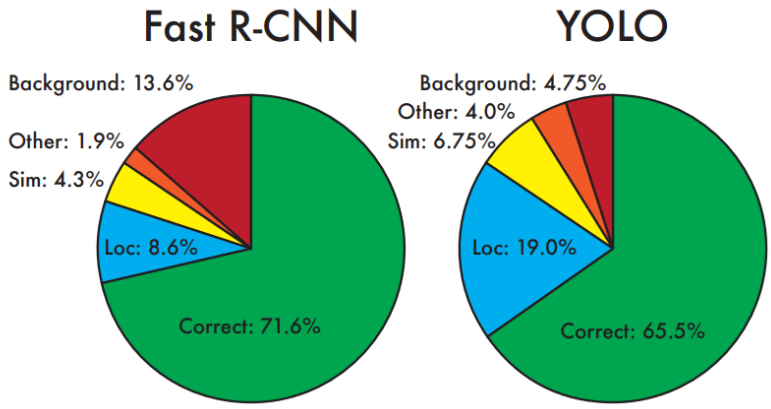

FIGURA 4.8: **Analisi degli errori: Fast R-CNN vs. YOLO.** Questi grafici a torta mostrano la percentuale di errori di localizzazione e di *background* nelle migliori N predizioni, per varie categorie (N =*[cardinalità dell'insieme degli oggetti nella data categoria]*). [23]

naturale, mantiene un buon livello di *performance* anche se usato per classificare immagini artistiche, ritraenti quadri e, più in generale, immagini artificiali.

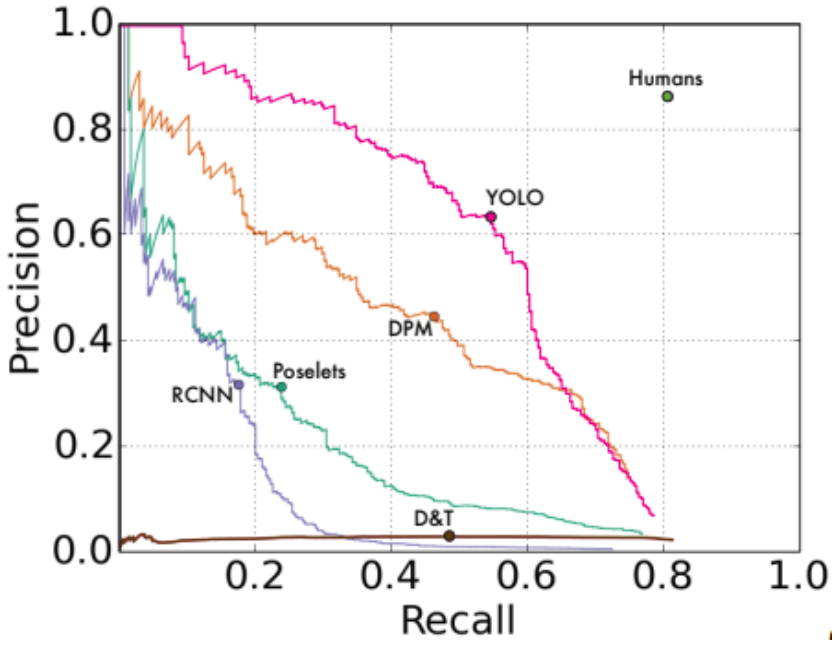

FIGURA 4.9: **Risultati nella generalizzazione sul dataset Picasso.** Curve precisione-chiamate sul *dataset Picasso*. [23]

Per testare la generalizzazione di *YOLO*, gli autori hanno confrontato le *performance* di *YOLO* con quelle di altri sistemi di *object detection* sul *Picasso dataset* e sul *People-Art Dataset*, due *dataset* ideati per testare sistemi di *person detection* su opere artistiche.

TABELLA 4.3: **Risultati qualitativi su *PASCAL VOC 2007*, *Picasso* e *PASCAL VOC 2007*.** Il *dataset Picasso* valuta sia sugli *mAP* che sul punteggio *Best* F_1.

	VOC 2007	Picasso		People-Art
	AP	AP	Best F_1	AP
YOLO	**59.2**	**53.3**	**0.590**	**45**
R-CNN	54.2	10.4	0.226	26
DPM	43.2	37.8	0.458	32
Poselets	36.5	17.8	0.271	
D&T	-	1.9	0.051	

La figura 4.9 e la tabella 4.3 mostrano i risultati della comparazione tra le *performance* di *YOLO* e quelle di altri sistemi di *object detection*.

I modelli usati per i test sul *Picasso dataset* sono stati addestrati con *VOC 2007*, mentre quelli per i test sul *People-Art Dataset* sono stati addestrati con *VOC 2010*.

R-CNN ha una buona precisione su *VOC 2007* ma questo dato cala considerabilmente quando il modello viene applicato ad immagini artistiche. *R-CNN*, come già spiegato, utilizza *selective search* per ottenere delle proposte di *bounding box*, sistema regolato su immagini naturali.

DPM conserva bene la propria accuratezza quando applicato ad immagini artistiche. Si teorizza che *DPM* si comporti bene con le immagini artistiche perché ha un modello spaziale molto forte per il riconoscimento di forma e disposizione degli oggetti. Gli autori di *YOLO* ricordano però che, per quanto *DPM* non perda precisione come *R-CNN* quando applicato ad immagini artificiali, questo parti da un livello inferiore di accuratezza nella valutazione di immagini naturali.

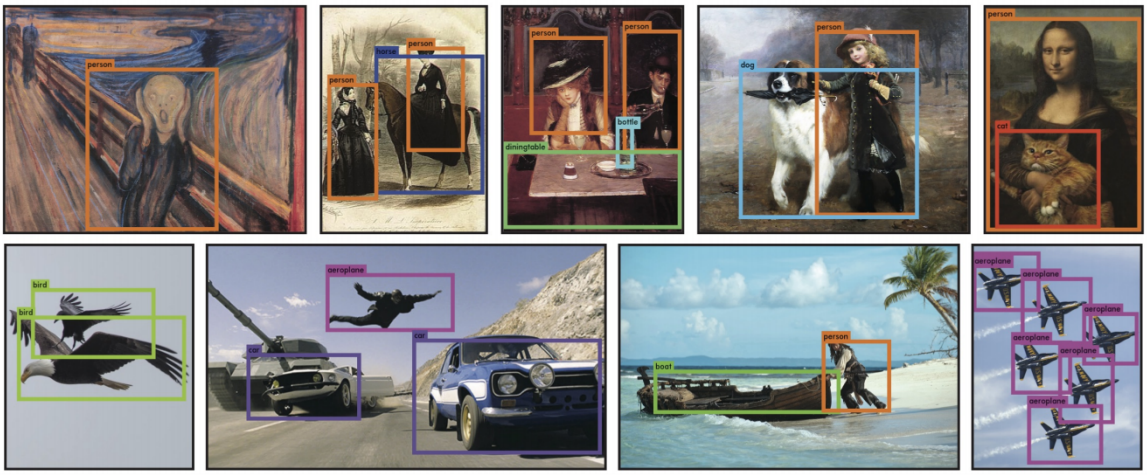

FIGURA 4.10: **Risultati qualitativi di YOLO:** YOLO eseguito su immagini artistiche ottenute da internet. È piuttosto accurato, nonostante pensi che una persona sia un aeroplano. [23]

YOLO ottiene buone *performance* su *VOC 2007* e, quando applicato ad immagini artistiche, perde meno accuratezza rispetto agli altri modelli presi in esame. Come

TABELLA 4.4: **Esperimenti di combinazione di modelli addestrati con *PASCAL VOC 2007*.** Gli autori esaminano gli effetti del combinare diversi modelli con la migliore versione di *Fast R-CNN*: combinandolo con altre versioni di *Fast R-CNN* si ottiene solo un piccolo miglioramento, mentre combinato con *YOLO* fornisce un significativo miglioramento nell'accuratezza. [23]

	mAP	Combined	Gain
Fast R-CNN	71.8	-	-
Fast R-CNN (2007 data)	**66.9**	72.4	.6
Fast R-CNN (VGG-M)	59.2	72.4	.6
Fast R-CNN (CaffeNet)	57.1	72.1	.3
YOLO	63.4	**75.0**	**3.2**

DPM, *YOLO* tiene conto sia della dimensione e della forma degli oggetti, che della relazione tra gli stessi e dove questi appaiano più frequentemente.

Redmond et al. evidenziano infine come, nell'ordine di pochi pixel, immagini naturali ed artificiali siano molto differenti, ma al contempo, come all'interno dell'intera immagine, le forme e le dimensioni degli oggetti siano molto simili. Vista la sua architettura, questo è sicuramente un fattore che aiuta *YOLO* a generalizzare meglio le classi di oggetti e quindi a dare buoni risultati quando applicato ad immagini artificiali.

4.2.4 YOLO combinato con Fast R-CNN

Essendo che *YOLO* fa meno errori di *background* e che *Fast R-CNN* ha un'accuratezza migliore, *J. Redmond et al.* hanno pensato di proporre un modello che combinasse *YOLO* e *Fast R-CNN*.

Questo modello combinato esegue *Fast R-CNN*; dopodiché controlla se le *bounding box* predette da *YOLO* siano simili. Se lo sono, il modello incrementa le probabilità di tale predizione, basandosi e sulle probabilità predette da *YOLO*, e sull'intersezione delle due *bounding box*.

Visto che il tempo impiegato per l'esecuzione di *YOLO* è trascurabile rispetto al tempo impiegato da *Fast R-CNN*, la combinazione dei due modelli non comporta cambiamenti significativi nel tempo di esecuzione di *Fast R-CNN*.

Il miglior modello *Fast R-CNN* raggiunge un *mAP* del 71.8% sul *test set* di *VOC 2007*; se combinato con *YOLO*, come mostrato in tabella 4.4, questo modello raggiunge un'accuratezza media del 75%, con un miglioramento maggiore del 3%.

J. Redmond et al. fanno notare come il miglioramento ottenuto nel modello ibrido non dipende dalla semplice combinazione di due computazioni, bensì come sia dovuto alla diversità dei modelli. Combinando diverse versioni di *Fast R-CNN* infatti non si ottengono miglioramenti significativi.

Piuttosto il miglioramento è dovuto al fatto che *YOLO* e *Fast R-CNN* fanno errori di diverso tipo in fase di test.

Capitolo 5

Sviluppi di YOLO

Presentiamo in questo capitolo quelli che sono stati i principali sviluppi di *YOLO* dal 2016 ad oggi.

Parliamo delle due versioni migliorate di *YOLO*, *YOLO-v2* (o *YOLO9000*) e *YOLO-v3*, pubblicate dagli stessi autori nei due anni successivi alla pubblicazione di *YOLO*.

Presentiamo inoltre *Fast YOLO*, una versione semplificata di *YOLO9000* che ne migliora le prestazioni in termini di tempo.

Infine diamo uno sguardo ad una diretta evoluzione di *YOLO*: *Complex-YOLO*, un sistema basato su *YOLO-v2* per l'*object detection* in ambienti tridimensionali.

5.1 Versioni successive di YOLO

Dal 2016 ad oggi, *J. Redmond et al.* hanno pubblicato due nuove versioni di *YOLO*: *YOLO-v2* [21] e *YOLO-v3* [22]. Contestualmente, un'altra versione di *YOLO-v2*, *Fast YOLO*, è stata pubblicata per proporre un modello che sposti il *trade-off* tra velocità ed accuratezza a favore della velocità.

Vediamo adesso nel dettaglio quali sono stati i miglioramenti nei vari aggiornamenti e come questi si riflettono sulle prestazioni del modello.

5.1.1 YOLO-v2 o YOLO9000

YOLO9000 o *YOLO-v2* è stato sviluppato e proposto da due degli autori di *YOLO*: *Joseph Redmond* ed *Ali Farhadi*. La principale novità introdotta nella seconda versione di *YOLO* è la capacità di riconoscere fino a 9'000 classi di oggetti differenti, pur rimanendo un sistema di *object detection real-time*.

YOLO9000 viene presentato dagli autori come *better* (migliore), *faster* (più veloce) e *stronger* (più forte). Migliore per i miglioramenti effettuati rispetto alla prima versione, più veloce perché costruito su di un *framework* diverso e con un architettura riorganizzata per lo scopo, più forte in quanto in grado di riconoscere una grande varietà di classi di oggetti.

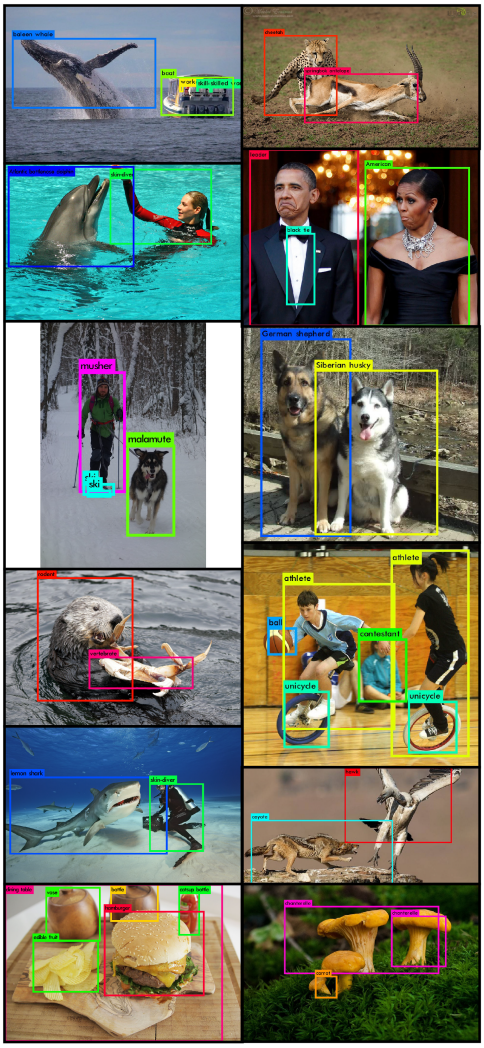

FIGURA 5.1: **Esempi di visualizzazione dell'output di YOLO9000.** *YOLO9000* può riconoscere in tempo reale una grande varietà di classi di oggetti. [21]

Better: *miglioramenti apportati rispetto alla prima versione*

Dal punto di vista strutturale, *YOLO9000* presenta diverse innovazioni:

- utilizzo della *Batch Normalization* su tutti i livelli convoluzionali, che comporta un miglioramento del 2% nella precisione (*mAP*).
- Sostituzione dei livelli finali interamente connessi con l'utilizzo di *anchor box* per la predizione delle *bounding box*. Contestualmente viene eliminato un livello di *pooling* per migliorare la risoluzione delle *features* estratte. Per l'addestramento vengono adesso utilizzate immagini di dimensione 416x416, e si ottiene una *feature map* di dimensione 13x13. Con l'utilizzo delle *anchor box*, l'accuratezza del modello migliora, in fase di *recall*, da un punteggio medio dell'81% ad uno dell'88%, penalizzando lievemente la precisione del riconoscimento (la *mAP* passa dal 69.5% al 69.2%).

- Utilizzo di dimensioni e proporzioni predefinite per la generazione delle *anchor box*, senza ottenere nessuna informazione dall'immagine, come avviene già in *Faster R-CNN* (si veda sezione 3.2.4 *"Faster RCNN"*).
- *YOLO-v1* non ha vincoli sulle predizioni delle *bounding box*, il che rende il modello instabile, in particolar modo durante le prime iterazioni della fase d'apprendimento. Questo perché le *bounding box* predette possono essere, o estendersi, lontane dalla cella della griglia a cui appartengono. *YOLO-v2* limita la locazione delle *anchor box* rispetto alla cella responsabile, usando una funzione d'attivazione $\sigma()$, la quale impone che il valore delle predizioni spaziali sia compreso tra 0 e 1 (si veda figura 5.2).
- Una *feature map* di dimensione 13x13 ha una risoluzione adatta per predire oggetti di grandi dimensioni; per fare buone predizioni anche su oggetti di piccola dimensione, la *feature map* di dimensione 26x26x512 ottenuta dai livelli precedenti è mappata in un tensore di dimensione 13x13x2048, quindi concatenata con la *feature map* di dimensione 13x13. Questa accortezza migliora l'accuratezza media del modello dell'1%.
- In fase di addestramento, ogni 10 *batches*, la dimensione delle immagini in input viene modificata casualmente; i valori possibili sono $\{320^2, 352^2, ..., 608^2\}$.

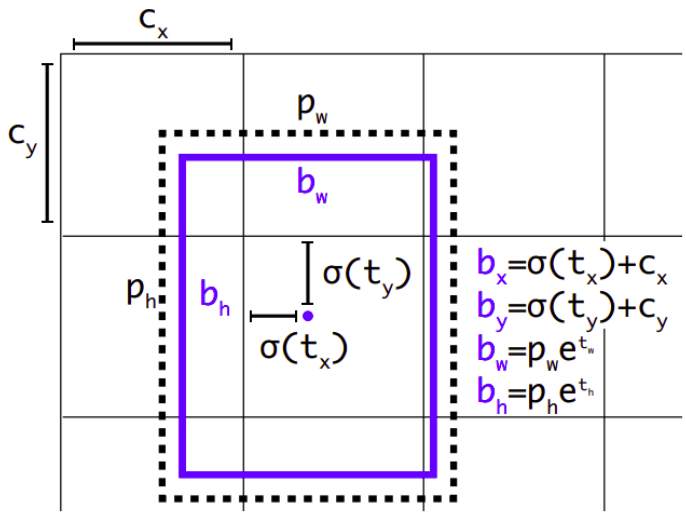

FIGURA 5.2: ***Bounding box* di *YOLO9000* con valori predetti (rispetto alla cella).** [21]

Ricapitolando, *YOLO9000* è più veloce su immagini di piccole dimensioni rispetto a quanto non lo sia *YOLO*. Per immagini a bassa risoluzione, ad una velocità di 90 fps, l'accuratezza media raggiunta da *YOLO9000* è simile a quella di *Fast R-CNN*, il che rende *YOLO9000* ideale per sistemi con poche risorse. Per immagini ad alta risoluzione invece, *YOLO9000* raggiunge un elevata accuratezza, totalizzando una *mAP* del 76.8%, pur restando un sistema di riconoscimento *real-time*.

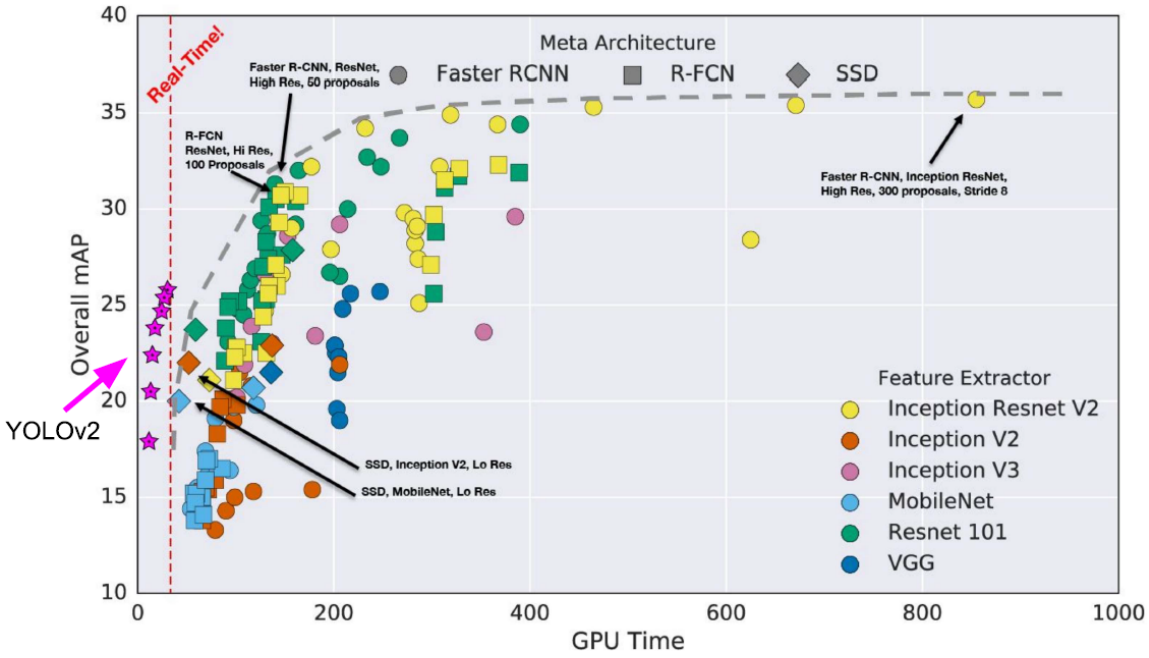

FIGURA 5.3: **Confronto tra modelli di *object detection* sul *dataset Microsoft COCO*.** [30]

Utilizzando il *dataset Microsoft COCO*, a differenza degli altri modelli presi in esame (si veda figura 5.3), *YOLO9000* è l'unico *object detector* a mantenere una velocità considerabile *real-time*.

Faster: *utilizzo di Darknet-19*

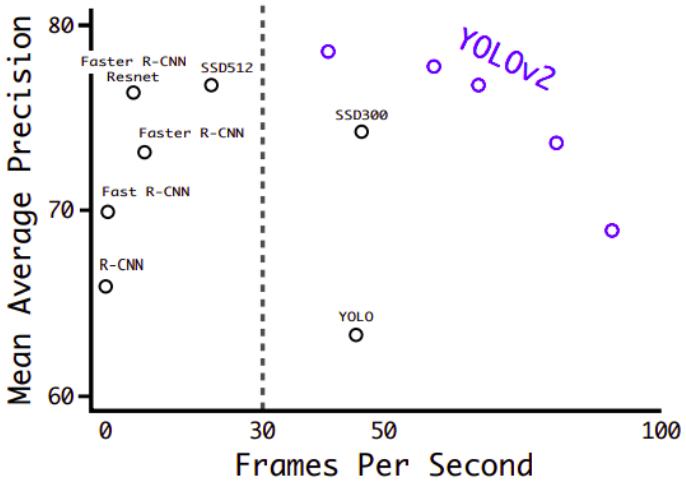

FIGURA 5.4: **Rapporto tra Accuratezza e Velocità.** Confronto tra diversi modelli per l'*object detection*, addestrati sul *dataset VOC 2007*, in termini di velocità ed accuratezza. [21]

Type	Filters	Size/Stride	Output
Convolutional	32	3×3	224×224
Maxpool		$2 \times 2/2$	112×112
Convolutional	64	3×3	112×112
Maxpool		$2 \times 2/2$	56×56
Convolutional	128	3×3	56×56
Convolutional	64	1×1	56×56
Convolutional	128	3×3	56×56
Maxpool		$2 \times 2/2$	28×28
Convolutional	256	3×3	28×28
Convolutional	128	1×1	28×28
Convolutional	256	3×3	28×28
Maxpool		$2 \times 2/2$	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Maxpool		$2 \times 2/2$	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	1000	1×1	7×7
Avgpool		Global	1000
Softmax			

FIGURA 5.5: **Architettura della rete *Darknet-19*.** Da notare la frequente alternanza di livelli di dimensione 1x1. Lo scopo è quello di ridurre il numero di parametri del modello. [30]

YOLO-v2 utilizza *Darknet-19 classification network* per l'estrazione delle *features* dall'immagine di input. L'architettura del classificatore *Darknet-19* è schematizzata in figura 5.5. Si noti che il modello ha molti livelli convoluzionali 1x1; questo riduce il numero di parametri rispetto al classificatore *"built-in"* della prima versione di *YOLO*.

	Top 1	**Top 5**	**FLOPs**	**GPU Speed**
VGG-16	70.5	90.0	30.95 Bn	100 FPS
Extraction (YOLOv1)	72.5	90.8	8.52 Bn	180 FPS
Resnet50	**75.3**	**92.2**	7.66 Bn	90 FPS
Darknet19	74.0	91.8	**5.58 Bn**	**200 FPS**

FIGURA 5.6: **Risultati sulla sfida 1000-classi di *Imagenet*.** Pur mantenendo un'accuratezza simile a *ResNet-50*, *Darknet-19* ha una rete meno complessa e quindi una velocità superiore al doppio di quella del modello *ResNet-50*. [30]

Darknet-19 ottiene ottimi risultati in termini di *trade-off* tra velocità ed accuratezza (si veda tabella 5.6).

Stronger: *utilizzo di dataset combinati per l'addestramento*

In giro per il web è possibile trovare tanti *dataset* da usare per modelli di riconoscimento e classificazione. Gli autori di *YOLO-v2* hanno pensato di combinarli tra loro in modo da poter addestrare il proprio modello su di una grande varietà di dati e classi.

Tuttavia la cosa non è così scontata: fare un semplice *merge* dei *dataset* porterebbe il modello a diventare instabile. Questo è dovuto soprattutto alla diversa specificità delle classi nei diversi *dataset*. Si prendano ad esempio i *dataset ImageNet* e *Microsoft COCO*: *ImageNet* ha classi molto specifiche, come ad esempio *"Yorkshire terrier"*, le quali non andrebbero d'accordo con le classi del *dataset* di *Microsoft COCO*, il quale, per completare l'esempio, possiede solo la classe più generale *"Dog"*.

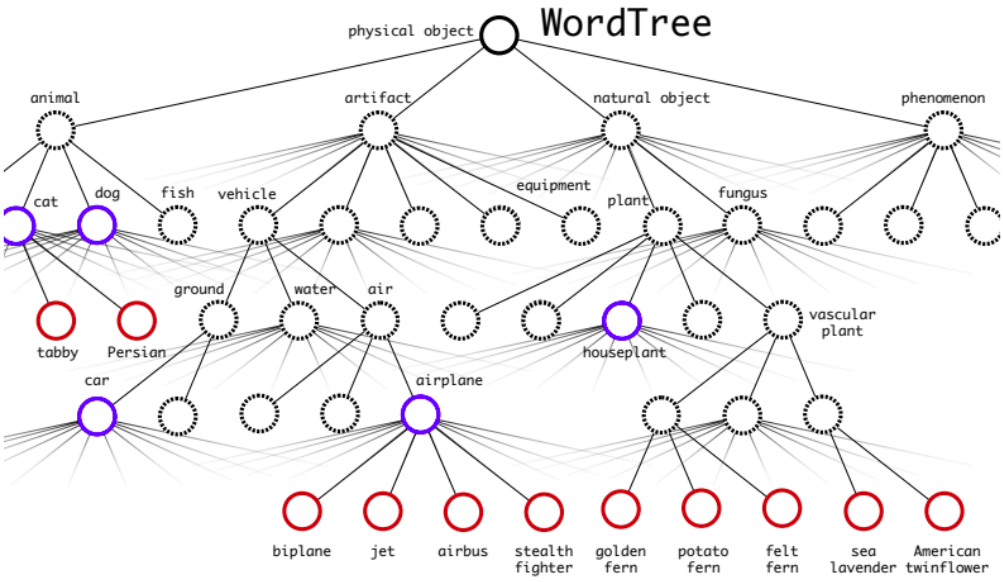

FIGURA 5.7: **Rappresentazione gerarchica *WordTree*.** Partendo dal concetto di *WordNet* gli autori di *YOLO-v2* hanno costruito un "albero dei concetti". Così facendo possono fondere i *dataset* di diverse fonti, mappando le classi all'interno dell'albero e creando una gerarchia tra le classi. L'immagine ritrae, a fini illustrativi, una versione semplificata del *WordTree* usato da *YOLO-v2*. [21]

Per ovviare a questo problema, l'approccio usato per fondere i *dataset* è analogo a quello usato, in un contesto diverso, da *Miller et al.* [20]: essi proposero una suddivisione gerarchica delle classi, simile a quella mostrata in figura 5.7.

Utilizzando lo stesso concetto di gerarchia tra classi, riprendendo l'esempio precedente, uno *"Yorkshire terrier"* sarà classificato prima come *"Mammal"*, poi come *"Dog"* ed infine come *"Yorkshire terrier"*.

Utilizzando questo metodo, *YOLO-v2* riesce a classificare 9′418 classi diverse di oggetti.

5.1.2 Fast YOLO

Fast YOLO è un'evoluzione di *YOLO-v2* proposta da *J. Shafiee et al.* [26]. *Fast YOLO* nasce con lo scopo di rendere possibile l'esecuzione di un modello di *object detection real-time* applicato a video-sequenze su dispositivi con risorse limitate.

Le differenze principali rispetto *YOLO-v2* sono riassumibili in due interventi sull'architettura:

1. ottimizzazione dell'architettura di rete di *YOLO-v2* con l'utilizzo di tecniche evolutive (il modello risultante è chiamato dagli autori *"O-YOLOv2"*, ovvero *"optimized YOLO-v2"*);
2. integrazione di un modello di inferenza *"motion-adaptive"*.

Vediamo sommariamente in cosa consiste il modello *motion-adaptive*.

Motion-Adaptive Inference

Il concetto alla base di questo modello è quello, in fase di classificazione dell'*n-esimo* frame, di tenere conto della classificazione già eseguita sui frame precedenti per semplificare la computazione.

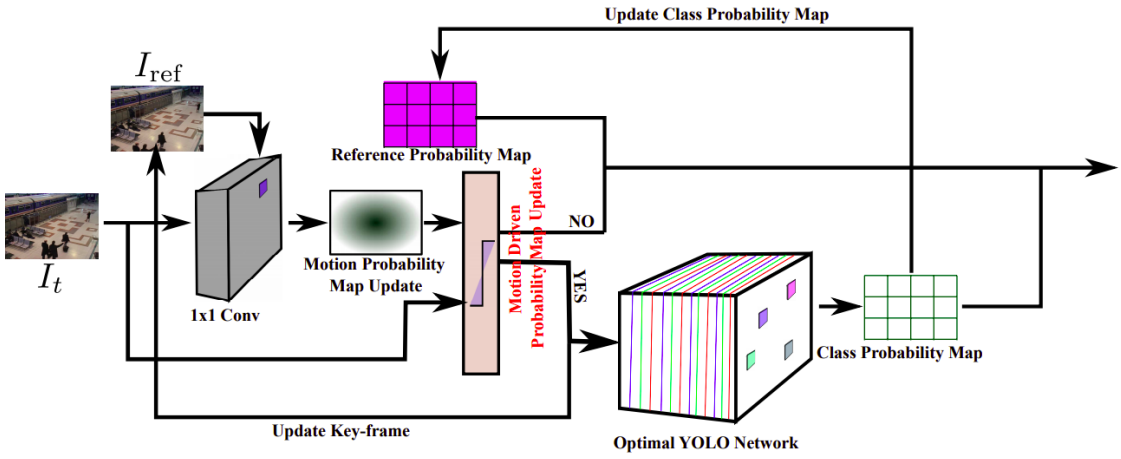

FIGURA 5.8: **Una visione d'insieme sull'architettura proposta con *Fast YOLO*.** [26]

L'architettura finale di *Fast YOLO* è descritta in figura 5.8.

Per ogni frame del video - diciamo il *t-esimo* frame I_t - la concatenzazione dell'immagine del frame I_t ed un frame scelto tra i precedenti, detto I_{ref}, vengono dati in pasto ad una semplice rete con un livello convoluzionale 1x1 per calcolare una mappa delle probabilità relativa al movimento all'interno dell'immagine.

A questo punto l'immagine I_t e la mappa delle probabilità appena calcolata vengono passate ad un modulo di inferenza *motion-adaptive* per decidere se l'inferenza è tale affinché si renda necessario classificare il frame:

- se è necessario classificare l'immagine, il frame I_t viene dato in pasto al modello ottimizzato *O-YOLOv2* per calcolare una mappa delle classi aggiornata che viene salvata. Il frame I_t viene salvato al posto del precedente I_{ref}.
- Altrimenti viene utilizzata direttamente la mappa delle classi salvata in precedenza, risparmiando il potere computazionale e la memoria necessari per calcolarla.

5.1.3 YOLO-v3

YOLO-v3 [22] è semplicemente una versione migliorata di *YOLO-v2*. Come spiegato dagli autori, con quest'ultimo aggiornamento per lo più hanno integrato idee altrui con il loro sistema e fatto piccole modifiche all'architettura del modello.

A differenza delle versioni precedenti, *YOLO-v3* predice le *bounding box* non solo su di un'immagine, ma sulla stessa immagine con tre dimensioni diverse.

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

FIGURA 5.9: **Architettura di *Darknet-53*.** [22]

Inoltre il *feature extractor Darknet-19* è stato modificando aggiungendo diversi livelli convoluzionali ed alcuni livelli residuali. Avendo un totale di 53 livelli, *J. Redmond* e *Ali Farhadi* chiamano questo nuovo *feature extractor "Darknet-53"*. L'architettura di *Darknet-53* è descritta nella tabella mostrata in figura 5.9.

Questa nuova rete è molto più potente di *Darknet-19* ma comunque più efficiente di *ResNet-101* o *ResNet-152* e migliora ulteriormente le prestazioni di *YOLO*.

5.2 Complex-YOLO

Per concludere presentiamo le caratteristiche principali di *Complex-YOLO* [27], un modello basato su *YOLO* per l'*object detection* in ambiente tridimensionale.

L'architettura della rete è simile a quella di *YOLO-v2*, estesa da un altro livello di regressione e da un *Euler Region-Proposal-Network* per determinare l'orientamento e le dimensioni degli oggetti.

Invece di processare semplici immagini, *Complex-YOLO* prende in input un'immagine generata che ritrae l'ambiente come fosse visto dall'alto - in effetti una *birds-eye-view RGB-map*. Questa mappa viene generata a partire da una scansione laser, la quale crea una mappa tridimensionale di punti.

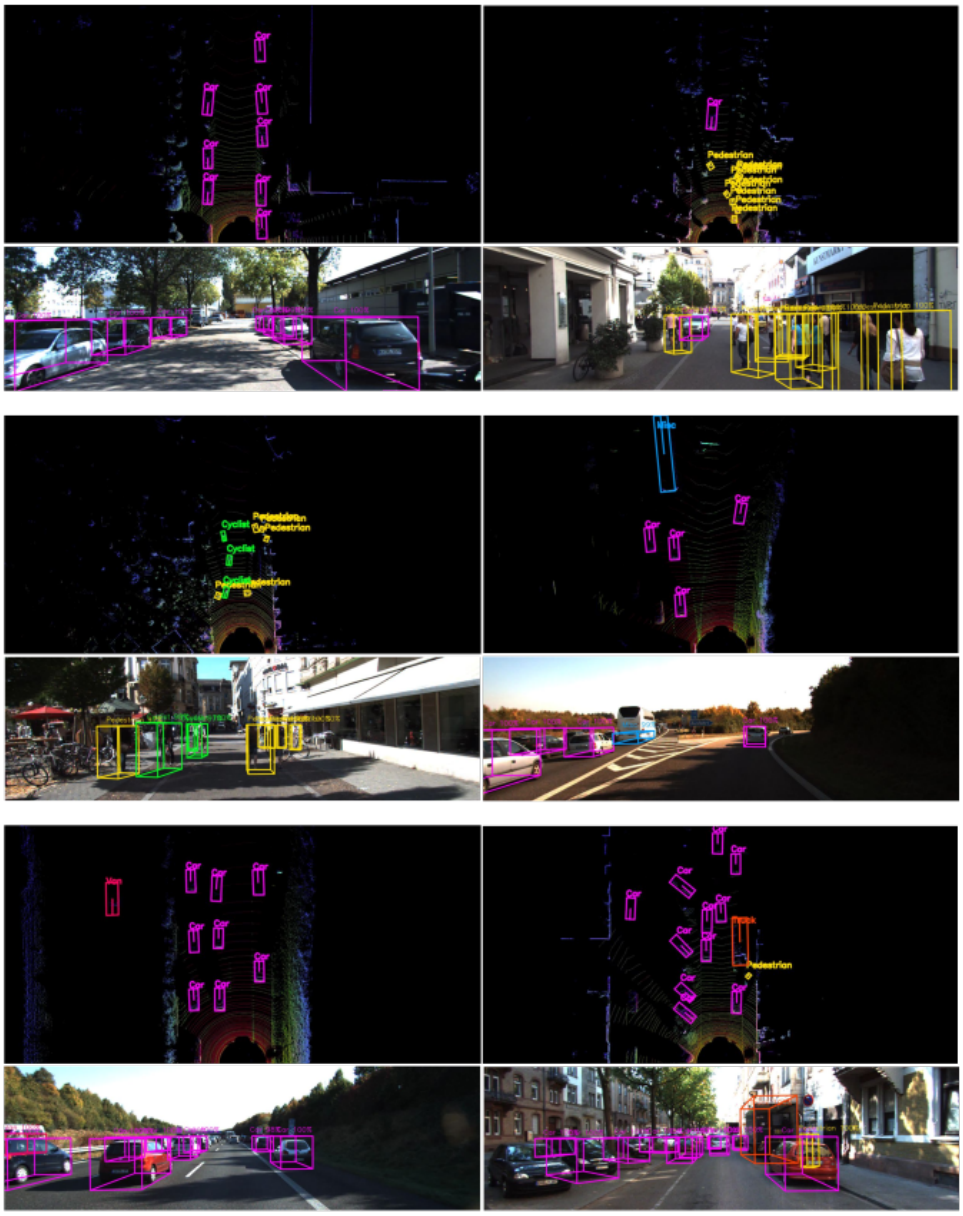

FIGURA 5.10: **Esempi visuali dell'output di *Complex-YOLO*.** Per ciascuno dei sei esempi, in alto è mostrata la *birds-eye-view* generata a partire dalla scansione laser, in basso la proiezione delle predizioni sulle corrispondenti immagini bidimensionali. Le predizioni vengono effettuate basandosi esclusivamente sulle immagini generate come *birds-eye-view*. Queste sono poi state proiettate sull'immagine della telecamera a fini illustrativi. [27]

TABELLA 5.1: **Architettura di *Complex-YOLO*.** Il modello presenta 18 livelli convoluzionali, 5 maxpool e 3 livelli intermedi per la riorganizzazione delle *features*. [27]

Layer	Filters	Size	Input	Output
conv	24	3 x 3/1	1024 x 512 x 3	1024 x 512 x 24
max		2 x 2/2	1024 x 512 x 24	512 x 256 x 24
conv	48	3 x 3/1	512 x 256 x 24	512 x 256 x 48
max		2 x 2/2	512 x 256 x 48	256 x 128 x 48
conv	64	3 x 3/1	256 x 128 x 48	256 x 128 x 64
conv	32	1 x 1/1	256 x 128 x 64	256 x 128 x 32
conv	64	3 x 3/1	256 x 128 x 32	256 x 128 x 64
max		2 x 2/2	256 x 128 x 64	128 x 64 x 64
conv	128	3 x 3/1	128 x 64 x 64	128 x 64 x 128
conv	64	3 x 3/1	128 x 64 x 128	128 x 64 x 64
conv	128	3 x 3/1	128 x 64 x 64	128 x 64 x 128
max		2 x 2/2	128 x 64 x 128	64 x 32 x 128
conv	256	3 x 3/1	64 x 32 x 128	64 x 32 x 256
conv	256	1 x 1/1	64 x 32 x 256	64 x 32 x 256
conv	512	3 x 3/1	64 x 32 x 256	64 x 32 x 512
max		2 x 2/2	64 x 32 x 512	32 x 16 x 512
conv	512	3 x 3/1	32 x 16 x 512	32 x 16 x 512
conv	512	1 x 1/1	32 x 16 x 512	32 x 16 x 512
conv	1024	3 x 3/1	32 x 16 x 512	32 x 16 x 1024
conv	1024	3 x 3/1	32 x 16 x 1024	32 x 16 x 1024
conv	1024	3 x 3/1	32 x 16 x 1024	32 x 16 x 1024
route	12			
reorg		/2	64 x 32 x 256	32 x 16 x 1024
route	22 20			
conv	1024	3 x 3/1	32 x 16 x 2048	32 x 16 x 1024
conv	75	1 x 1/1	32 x 16 x 1024	32 x 16 x 75
E-RPN			32 x 16 x 75	

5.2.1 Architettura

Complex-YOLO usa una rete chiamata *Euler-Region-Proposal* per processare le coordinate (b_x, b_y, b_w, b_l, rispettivamente ascissa ed ordinata del centro, ampiezza e lunghezza), la probabilità associata all'esistenza dell'oggetto (p_0), le probabilità che ciascun oggetto appartenga all'*i-esima* classe ($p_1, p_2, ..., p_n$) ed infine l'orientamento dell'oggetto, determinato dal fattore b_ϕ.

Per gestire in modo appropriato la predizione dell'orientamento degli oggetti, gli autori di *Complex-YOLO* hanno modificato l'approccio standard a griglie dei *RPN* attualmente più diffusi, aggiungendo alle *features* un angolo complesso $arg(|z|e^{ib_\phi})$:

$$b_x = \sigma(t_x) + c_x$$
$$b_x = \sigma(t_y) + c_y$$
$$b_w = p_w e^{t_w}$$
$$b_l = p_l e^{t_l}$$
$$b_\phi = arg(|z|e^{ib_\phi}) = \arctan_2(t_{Im}, t_{Re})$$

Con l'aggiunta di questo fattore, l'*E-RPN* è in grado di predire con accuratezza l'orientazione degli oggetti. Per ogni cella della griglia, *Complex-YOLO* predice cinque oggetti, con 75 *features* per ciascuno, come mostrato nella tabella 5.1.

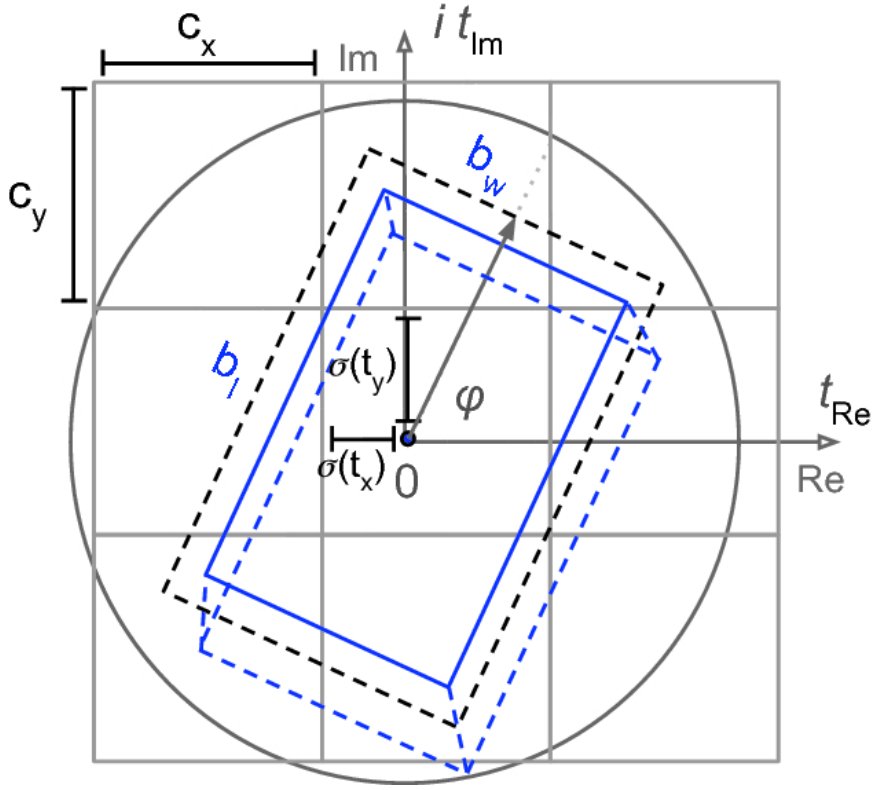

FIGURA 5.11: ***Complex-YOLO* 3D *Bounding Box.*** *Complex-YOLO* predice *bounding box* tridimensionali orientate basandosi sui parametri della regressione di *YOLO-v2* e sull'angolo complesso per l'orientamento della scatola. Il passaggio da *bounding box* bidimensionale a tridimensionale avviene utilizzando altezze predefinite associate alla classe dell'oggetto riconosciuto. [27]

La *Loss Function* di *YOLO-v2* viene utilizzata così come pensata dagli autori, ma l'errore totale nella propagazione in avanti è calcolato come la somma tra l'errore calcolato appunto dalla *Loss Function* di *YOLO-v2* ed una *Loss Function* che misura l'errore nelle *bounding box* proposte dall'*E-RPN*:

$$\mathcal{L} = \mathcal{L}_{YOLO} + \mathcal{L}_{Euler}$$

L'errore commesso nelle predizioni delle *bounding box* è calcolato come segue:

$$\mathcal{L}_{Euler} = \lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} \mathbb{1}_{ij}^{obj} [(t_{im} - \hat{t}_{im})^2 + (t_{re} - \hat{t}_{re})^2]$$

dove $\mathbb{1}_{ij}^{obj}$ vale 1 se la *j-esima bounding box* dell'*i-esima* cella è responsabile per il riconoscimento di un oggetto, 0 altrimenti;

λ_{coord} è un fattore variabile, analogo a quello della *loss function* di *YOLO* che serve ad assicurare che il modello converga durante le prime fasi dell'addestramento.

5.2.2 Addestramento

M. Simon et al. hanno addestrato il loro modello con un *weight decay* di $5 * 10^{-4}$ ed un *momentum* di 0.9. La loro implementazione, come quella di *YOLO*, si basa su di una versione modificata di *Darknet*.

Come di consueto, hanno iniziato l'addestramento con un tasso d'apprendimento piuttosto basso durante le prime epoche, per poi aumentarlo dopo qualche epoca. Da questo punto fino alla millesima (ed ultima) epoca, il tasso d'apprendimento decresce gradualmente.

Per la regolarizzazione del flusso hanno utilizzato la *batch normalization*.

A parte l'ultimo livello che utilizza una semplice funzione identità come attivazione, ogni altro livello della rete utilizza la funzione lineare *"leaky rectified"* come funzione d'attivazione (si veda funzione 4.1.2).

5.2.3 Sperimentazione

Per la sperimentazione gli autori hanno adattato il loro modello seguendo il *KITTI evaluation protocol* [5]: gli oggetti riconosciuti al di fuori del campo visivo dell'immagine sono stati scartati.

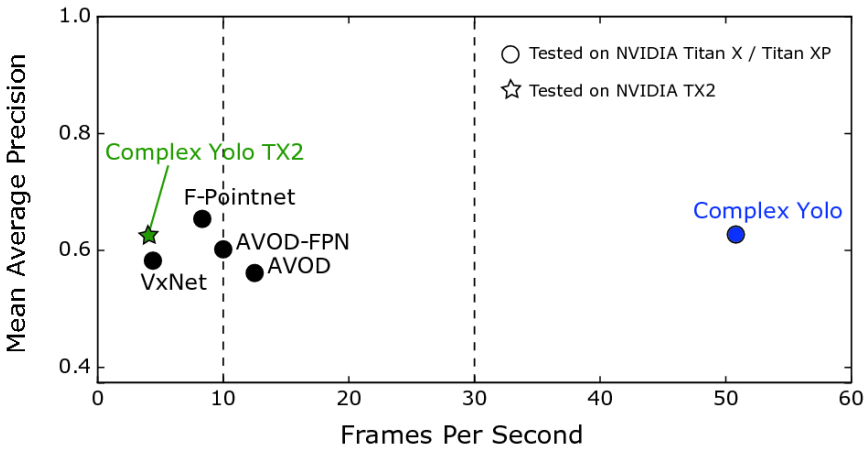

FIGURA 5.12: **Comparazione delle performance di *object detector* per scene 3D.** Questo grafico mostra il rapporto tra precisione (mAP) e velocità (fps) dei modelli. Le performance di *Complex-YOLO* sono state confrontate con quelle dei cinque modelli più famosi per il riconoscimento in tempo reale di oggetti in tre dimensioni. I risultati mostrano come *Complex-YOLO* sia molto più veloce degli altri modelli presi in esame, pur mantenendo un'ottima accuratezza media nelle predizioni. [27]

Dai test si evince che *Complex-YOLO* è senza dubbio il modello di *object detection* in tre dimensioni più veloce ed efficiente, ottenendo buoni risultati in termini di precisione ed accuratezza delle predizioni.

TABELLA 5.2: **Comparazione delle *performance* nel 3D *object detection*.** [27]

Method	**Modality**	**FPS**	**Car**			**Pedestrian**			**Cyclist**		
			Easy	**Mod.**	**Hard**	**Easy**	**Mod.**	**Hard**	**Easy**	**Mod.**	**Hard**
MV3D	Lidar+Mono	2.8	86.02	76.90	68.49	-	-	-	-	-	-
F-PointNet	Lidar+Mono	5.9	88.70	84.00	75.33	**58.09**	**50.22**	**47.20**	**75.38**	61.96	54.68
AVOD	Lidar+Mono	12.5	86.80	**85.44**	77.73	42.51	35.24	33.97		47.74	46.55
AVOD-FPN	Lidar+Mono	10.0	88.53	83.79	**77.90**	50.66	44.75	40.83	62.39	52.02	47.87
VoxelNet	Lidar	4.3	**89.35**	79.26	77.39	46.13	40.74	38.11	66.70	54.76	50.55
Complex-YOLO	Lidar	**50.4**	85.89	77.40	77.33	46.08	45.90	44.20	72.37	**63.36**	**60.27**

Complex-YOLO risulta essere 0.02 secondi più veloce rispetto ad *AVOD* [12]; rispetto a *VoxelNet* [31] è 10 volte più veloce e rispetto a *MV3D* [2], il rivale più lento, *Complex-YOLO* risulta essere ben 18 volte più veloce (si veda tabella 5.2).

Capitolo 6

Conclusioni

In questa tesi abbiamo trattato il problema dell'*object detection* e le principali soluzioni basate su reti neurali, integrando la letteratura già esistente riguardante l'*object detection* e gli specifici modelli e tecniche risolutive. In tal senso è stata condotta una ricerca sulle principali innovazioni degli ultimi anni apportate nel settore.

Abbiamo esposto le caratteristiche fondamentali dei modelli per l'*object detection* più recenti, soffermandoci sulle migliorie apportate da ciascuno di questi. Abbiamo inoltre approfondito *You Only Look Once*, il primo modello a trattare i problemi di individuare e classificare un oggetto in un'immagine come un unico problema di regressione. *YOLO* presenta un'architettura unificata ed è addestrabile *end-to-end*. Per concludere abbiamo indagato le modifiche apportate nelle ultimissime versioni di *YOLO* ed abbiamo presentato *Complex-YOLO*, un modello per l'*object detection* su scene tridimensionali che sfrutta l'architettura innovativa di *YOLO*.

Nonostante non ci si sia soffermarti su tutti i particolari di ciascun modello trattato, abbiamo cercato di trattare principalmente le caratteristiche che li distinguono dai precedenti, lasciando al lettore la possibilità di ottenere maggiori informazioni seguendo i riferimenti agli articoli di presentazione. La ricerca si ferma alla pubblicazione della *focal loss function* di *RetinaNet* del 2018.

Questa ricerca potrà essere integrata in futuro presentando le ultimissime innovazioni nell'ambito dell'*object detection* o approfondendo i temi qui presentati, come le innovazioni apportate dai diversi modelli trattati e gli eventuali sviluppi che queste potrebbero avere.

Ringraziamenti

Vorrei ringraziare il prof. *Andrea Asperti*, il quale ha reso possibile la realizzazione di questa tesi, di cui è relatore. La sua disponibilità ed il suo supporto sono stati fondamentali per il conseguimento di questo risultato.

Ci tengo inoltre a ringraziare i miei familiari, in particolar modo mia mamma e mio papà, i quali sono stati presenti durante tutta la mia vita e hanno reso possibile che io raggiungessi questo obiettivo e diventassi la persona che sono oggi.

Per ultimi, ma non per questo meno importanti, ringrazio i miei amici, vecchi e nuovi, e tutti coloro che in questi anni mi sono stati vicini, nei momenti belli ed in quelli di sconforto.

Un grazie sentito a tutti.

Erich Kohmann

Bibliografia

[1] Md Zahangir Alom et al. «The History Began from AlexNet: A Comprehensive Survey on Deep Learning Approaches». In: *arXiv.org > Computer Science > Computer Vision and Pattern Recognition* (2018). URL: https://arxiv.org/abs/1803.01164.

[2] X. Chen et al. «Multi-view 3D object detection network for autonomous driving». In: *CoRR abs/1611.07759* (2016).

[3] Thomas Crow. «The rise of machine learning in astronomy». In: *Particle @scitech.org.au/space/* (2018). URL: https://particle.scitech.org.au/space/the-rise-of-machine-learning-in-astronomy/.

[4] J. Dai et al. «R-fcn: Object detection via region-based fully convolutional networks». In: *in Advances in neural information processing systems* (2016), pp. 379–387.

[5] Andreas Geiger, Philip Lenz e Raquel Urtasun. «Are we ready for autonomous driving? The KITTI vision benchmark suite». In: *2012 IEEE Conference on Computer Vision and Pattern Recognition* (2012).

[6] R. Girshick. «Fast r-cnn». In: *Proceedings of the IEEE international conference on computer vision* (2015), pp. 1440–1448.

[7] R. B. Girshick, P. F. Felzenszwalb e D. McAllester. «Discriminatively trained deformable part models, release 5». In: (2012). URL: http://www.rossgirshick.info/latent/.

[8] K. He et al. «Spatial pyramid pooling in deep convolutional networks for visual recognition». In: *European conference on computer vision, Springer* (2014), pp. 346–361.

[9] Jonathan Hui. «Real-time Object Detection with YOLO, YOLOv2 and now YOLOv3». In: (2018). URL: https://medium.com/@jonathan_hui/real-time-object-detection-with-yolo-yolov2-28b1b93e2088.

[10] Kinjal A Joshi e Darshak G. Thakore. «A Survey on Moving Object Detection and Tracking in Video Surveillance System». In: *International Journal of Soft Computing and Engineering (IJSCE)* 2.3 (2012). URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.645.7492&rep=rep1&type=pdf.

[11] A. Krizhevsky, I. Sutskever e G. E. Hinton. «Imagenet classification with deep convolutional neural networks». In: *Advances in neural information processing systems* (2012), pp. 1097–1105.

[12] J. Ku et al. «Joint 3D proposal generation and object detection from view aggregation». In: *arXiv preprint arXiv:1712.02294* (2017).

[13] S. Lazebnik, C. Schmid e J. Ponce. «Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories». In: *CVPR* (2006).

[14] K. Lenc e A. Vedaldi. «R-cnn minus r». In: *arXiv:1506.06981* (2015).

[15] Z. Li et al. «Light-head r-cnn: In defense of two-stage object detector». In: *arXiv:1711.07264* (2017).

[16] T.-Y. Lin et al. «Feature pyramid networks for object detection». In: *Review of Scientific Instruments* 1.2 (2017), p. 4.

[17] T.-Y. Lin et al. «Focal loss for dense object detection». In: *IEEE transactions on pattern analysis and machine intelligence* (2018).

[18] W. Liu et al. «Ssd: Single shot multibox detector». In: *European conference on computer vision. Springer* (2016).

[19] Mauricio Menegaz. «Understanding YOLO». In: (2018). URL: https://hackernoon.com/understanding-yolo-f5a74bbc7967.

[20] G. A. Miller et al. «Introduction to wordnet: An on-line lexical database». In: *International journal of lexicography* 3 (1990), pp. 235–244.

[21] Joseph Redmon e Ali Farhadi. «YOLO9000: Better, Faster, Stronger». In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017), pp. 7263–7271. URL: http://openaccess.thecvf.com/content_cvpr_2017/html/Redmon_YOLO9000_Better_Faster_CVPR_2017_paper.html.

[22] Joseph Redmon e Ali Farhadi. «YOLOv3: An Incremental Improvement». In: *arXiv:1804.02767* (2018). URL: https://arxiv.org/abs/1804.02767.

[23] Joseph Redmon et al. «You Only Look Once: Unified, Real-Time Object Detection». In: *arXiv.org > Computer Science > Computer Vision and Pattern Recognition* (2016). URL: https://arxiv.org/pdf/1506.02640.pdf.

[24] S. Ren et al. «Faster r-cnn: Towards real-time object detection with region proposal networks». In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 6 (2017), pp. 1137–1149.

[25] K. E. Van de Sande et al. «Segmentation as selective search for object recognition». In: *Computer Vision (ICCV), 2011 IEEE International Conference on. IEEE* (2011), pp. 1879–1886.

[26] Mohammad Javad Shafiee et al. «Fast YOLO: A Fast You Only Look Once System for Real-time Embedded Object Detection in Video». In: *arXiv.org > Computer Science > Computer Vision and Pattern Recognition* (2017). URL: https://arxiv.org/abs/1709.05943.

[27] Martin Simon et al. «Complex-YOLO: An Euler-Region-Proposal for Real-Time 3D Object Detection on Point Clouds». In: *Computer Vision and Pattern Recognition – ECCV 2018 Workshops – arXiv:1803.06199* (2019), pp. 197–209. URL: https://arxiv.org/abs/1803.06199.

[28] Yi Sun et al. «DeepID3: Face Recognition with Very Deep Neural Networks». In: *arXiv > Computer Science > Computer Vision and Pattern Recognition* (2015). URL: https://arxiv.org/abs/1502.00873.

[29] C. Szegedy et al. «Going deeper with convolutions». In: *CoRR, abs/1409.4842* (2014).

[30] Sik-Ho Tsang. «Review: YOLOv2 & YOLO9000 — You Only Look Once (Object Detection)». In: (2018). URL: https://towardsdatascience.com/review-yolov2-yolo9000-you-only-look-once-object-detection-7883d2b02a65.

[31] Y. Zhou e O. Tuzel. «VoxelNet: end-to-end learning for point cloud based 3D object detection». In: *CoRR abs/1711.06396* (2017).

[32] Zhengxia Zou et al. «Object Detection in 20 Years: A Survey». In: *IEEE* (2019). URL: https://arxiv.org/pdf/1905.05055.pdf.