SAPIENZA
UNIVERSITÀ DI ROMA

Confronto tra le Reti Neurali YOLO e TINY YOLO per Object Detection in Ambiente Duckietown

Facoltà di Ingegneria dell'Informazione, Informatica e Statistica

Corso di Laurea in Ingegneria Informatica ed Automatica

Candidato

George Ciprian Telinoiu
Matricola 1796513

Relatore

Prof. Daniele Nardi

Anno Accademico 2019/2020

Confronto tra le Reti Neurali YOLO e TINY YOLO per Object Detection in Ambiente Duckietown
Tesi di Laurea. Sapienza – Università di Roma

Questa tesi è stata composta con LaTeX e la classe Sapthesis.

Email dell'autore: telinoiu.1796513@studenti.uniroma1.it

Sommario

Il documento che segue rappresenta il lavoro di diversi mesi basato su due dei corsi che ho frequentato durante il 2° Semestre del mio 3° Anno Accademico all'interno del mio Corso di Laurea, sul quale ho voluto basare il lavoro della mia Relazione Finale.
In particolare i corsi di interesse sono stati *Metodi Quantitativi per l'Informatica* e il *Laboratorio di Intelligenza Artificiale e Grafica Interattiva*, i quali mi hanno permesso, rispettivamente, di acquisire le fondamenta per quanto riguarda il mondo delle Reti Neurali e delle loro applicazioni; e di poterle poi sperimentare mediante il progetto *Duckietown* affiliato al Laboratorio.

Mi sono voluto dunque occupare di un problema di Object Detection utilizzando come dataset l'ambiente Duckietown sfruttando *YOLO* come algoritmo di riferimento. Date però le particolarità dell'ambiente simulato, ho dovuto considerare le diverse limitazioni di quest'ultimo andando ad analizzare anche una sua variante denominata *YOLO-TINY* che poteva rappresentare una soluzione ottimale al problema considerato.

Il lavoro svolto si compone di una prima parte pratica svolta in laboratorio, dove si è costruita la pista in base alle indicazioni documentate nel manuale *Duckietown*, il setup del Duckiebot che verrà descritto in particolare nella sezione *Metodologia* ed infine il campionamento delle immagini da affiancare al dataset utilizzato nella fase di allenamento delle reti. La seconda invece ha visto l'analisi di un dataset esterno sempre relativo all'ambiente Duckietown, utilizzato appunto per allenare le due reti di interesse, le quali sono state poi sfruttate per fare Object Detection e comprendere quale risultasse più adatta dato il problema iniziale, portando quindi al confronto sopraccitato.

Indice

Capitolo 1

Introduzione

Il mondo del software nell'ultimo decennio sta divenendo sempre più un pilastro all'interno di moltissimi ambiti lavorativi. Da sempre sono stato colpito dalla versatilità di questa disciplina, in particolare dalla sua capacità di rivoluzionare e rivoluzionarsi indipendentemente dal contesto applicativo. Per quanto non sia mai stato un amante dell'hardware, trovo comunque affascinanti i vari scenari offerti dal mondo dell'automazione, che proprio tramite l'intervento del software riesce a dare una nuova vita e nuove potenzialità all'hardware stesso. Uno dei rami che mi ha sempre stimolato è quello della guida autonoma, sia per l'ambizione che questa sfida porta con sé, che per il suo reale riscontro positivo laddove divenisse qualcosa di realmente affidabile.
Questo progetto infatti rappresenta un argomento sempre più reale e vicino, il quale richiede una forte attenzione da parte nostra in questo momento, al fine di poterla includere nella vita di tutti noi. Una delle componenti principali in questa vastissima problematica è quella della percezione, ossia la necessità di riuscire ad acquisire sempre più dati di natura fotografica e non, in modo da poter gestire tutto ciò che circonda l'autoveicolo. Problemi preesistenti come la classificazione e il riconoscimento di immagini, rappresentano un fondamentale punto di partenza per individuare pattern utili e soluzioni riutilizzabili. Non a caso, una delle tecnologie più utilizzate per risolvere questo tipo di problemi è quello dell'Intelligenza Artificiale, soprattutto l'introduzione delle cosiddette Reti Neurali che, rispetto agli approcci più classici di Machine Learning, riescono a dare risultati decisamente più prestanti e gestiscono in maniera ottima quantità di dati ben più voluminose.
Attratto quindi da questi argomenti e dalle loro peculiarità, ho deciso di approfondirli concretamente all'interno dei corsi di Metodi Quantitativi per l'Informatica e del Laboratorio di Intelligenza Artificiale e Grafica Interattiva (LABIAGI). Andando quindi più nel particolare nei confronti degli argomenti trattati, durante il primo corso, ho avuto l'occasione di introdurmi alle Reti Neurali, alle loro possibili applicazioni e, implementare sul dataset che verrà presentato più avanti, l'algoritmo di Object Detection YOLO; mentre per quanto riguarda il Laboratorio ci è stato fornito il materiale necessario per poter operare all'interno del mondo di Duckietown, un mondo ideale simulato che tramite l'uso di piccoli robot permette lo studio e lo svolgimento di esperimenti volti alla comprensione del mondo dell'automazione, andando ad affrontare tutte le tematiche necessarie al fine di simulare al meglio le sfida della guida autonoma.

I possibili approcci a questa disciplina sono diversi, seppur in media partono tutti da un ambiente supervisionato che gradualmente va a concedere sempre più autonomia al veicolo, considerati gli input richiesti al guidatore. Un riferimento cardine di

questo metodo è lo standard J3016 della SAE International *(Society of Automotive Engineers)*, che nel 2016 ha rilasciato un documento che racchiude in 5 livelli gli stadi di autonomia che ad ora sono stati concepiti. (vedi fig. 1.1)

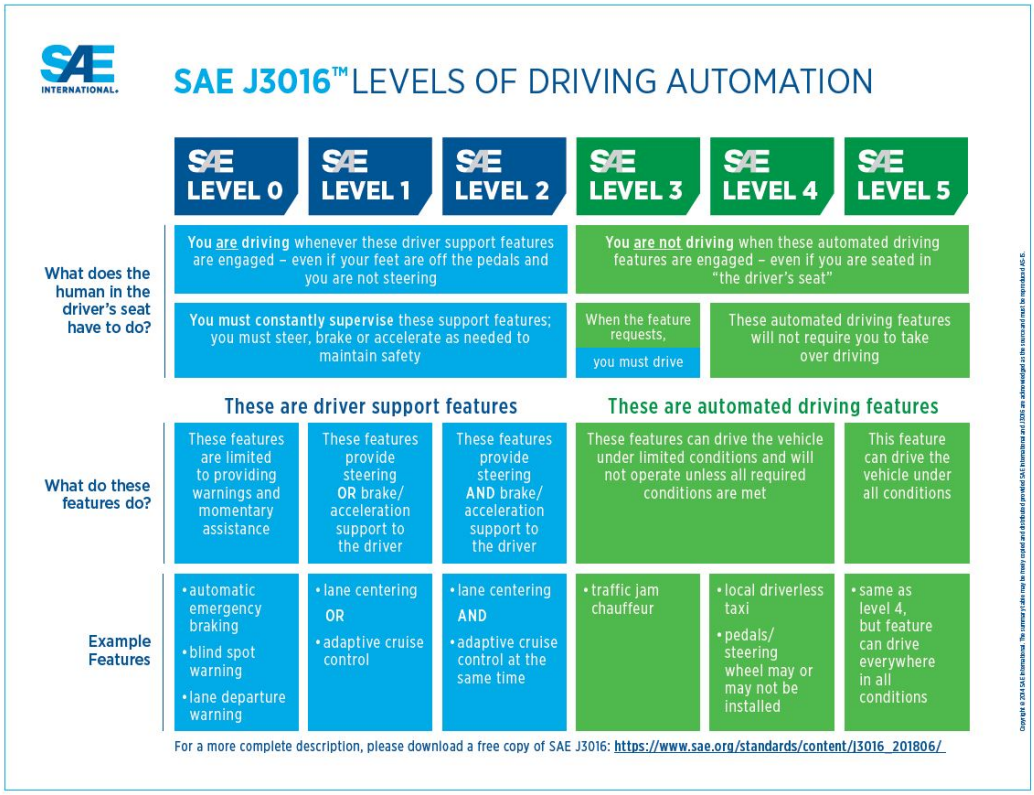


Figura 1.1. Livelli di automazione secondo la classificazione SAE

Come possiamo notare, una buona parte dei livelli richiede quasi sempre la presenza di un umano al fine di supervisionare il corretto funzionamento dei sistemi sviluppati. Questo però comporta anche una serie di rischi e costi, necessari per poter garantire un livello ottimo rispetto ai collaudi effettuati. E' qui che interviene il progetto *Duckietown* poiché, seguendo comunque il percorso progressivo della figura e riuscendo a traslare in problemi reali le considerazioni fatte in laboratorio; riduce drasticamente i costi necessari per sviluppare e testare nuove tecnologie, azzera i rischi che di solito sono associati a questo tipo di esperimenti e, dato il modo in cui è stato concepito, permette un forte approccio modulare verso gli argomenti che si vogliono affrontare. Nel caso di questo esperimento, infatti, ci si è concentrati soltanto su una piccola componente di quelle necessarie al fine di poter garantire un'esperienza di guida autonoma; in particolare è stato esaminato un algoritmo di Object Detection nei confronti di alcune classi del mondo simulato.

In breve, si è affrontato un problema multi-class su un dataset di terze parti relativo al mondo di Duckietown sfruttando le potenzialità di YOLO, in particolare andandolo a confrontare con la sua variante light, YOLO-TINY, al fine di ottimizzare una teorica implementazione di quanto sviluppato all'interno del nostro bot.

Qui di seguito la suddivisione del lavoro svolto:
Nel Capitolo 2 verranno descritti gli strumenti utilizzati al fine di poter svolgere il lavoro fatto, partendo dalle particolarità del mondo di Duckietown, continuando con il Dataset utilizzato e concludendo con l'introduzione all'algoritmo di YOLO;

A seguire, nel Capitolo 3 verrà affrontato tutto il processo di setup del robot al fine di poterlo utilizzare sulla pista costruita in Laboratorio. Nella seconda parte del capitolo invece verrà svolto il confronto tra le due reti, andando ad analizzare i vantaggi dell'una rispetto all'altra, descrivendo le necessità del problema affrontato ed infine mostrando le metriche dei test effettuati, i quali verranno mostrati e commentati all'interno del Capitolo 4.

In chiusura nel Capitolo 5 verranno riassunti i risultati ottenuti, proposti eventuali miglioramenti ed espresso un commento riguardo il lavoro svolto.

Capitolo 2

Strumenti Utilizzati

2.1 Ambiente Duckietown

L'ambiente Duckietown al fine di riuscire a simulare al meglio il mondo della guida autonoma posa le sue basi su due componenti: i Duckiebot e le Duckietowns. Andremo ora ad analizzare sommariamente l'hardware e le componenti software di maggiore interesse dei Duckiebot, mentre per quanto riguardo le Duckietown verrà esposto il lavoro che è stato svolto in Laboratorio, ossia la progettazione e l'assemblaggio di quest'ultima.

2.1.1 Duckiebot

Il Duckiebot sfrutta un telaio preesistente fornito già di 2 motori DC ed una struttura a due livelli che ben si presta a contenere sia la batteria che il cuore del nostro robot, ossia il Raspeberry Pie. Esso rappresenta appunto il centro di comando del Duckiebot, fornito di un processore quad-core ARMv8 a 1.2Ghz e 1GB di memoria RAM, nonché di una porta LAN ed un ricettore wireless fondamentali per il processo di setup del networking. Infine vi è la fotocamera 1080p da 5 Mega Pixels con un campo visivo di 160° che rappresenta il sensore principale del robot. Troviamo una rappresentazione di tutte le componenti citate nella fig. 2.1.

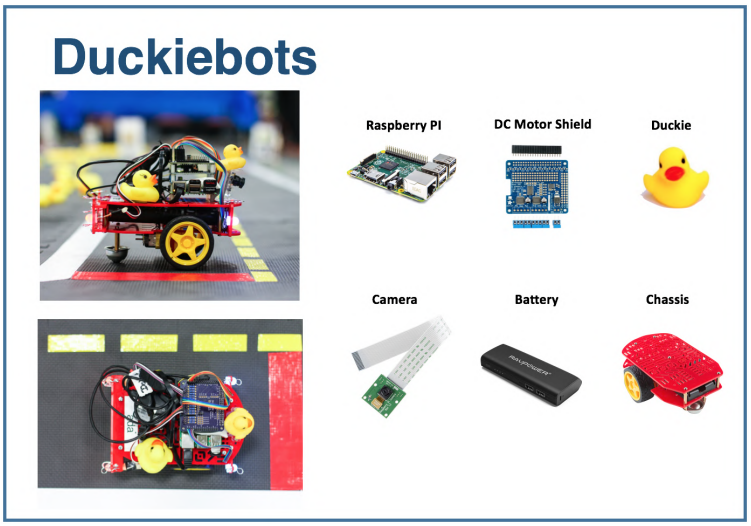


Figura 2.1. Hardware di maggiore interesse del Duckiebot

Per quanto riguardo il software del Duckiebot, uno dei concetti fondamentali che rendono il dispositivo così modulare e riducono al minimo i problemi di compatibilità è quello della containerizzazione. Il servizio scelto è stato quello di Docker, un progetto open source che, sfruttando i container default di Linux, è riuscito a costruire una piattaforma che rende agile e comoda la distribuzione di software. Docker, considera i container come macchine virtuali modulari estremamente leggere, offrendo la flessibilità di creare, distribuire, copiare e spostare i container da un ambiente all'altro, ottimizzando così le app per il cloud.
La tecnologia Docker utilizza il kernel di Linux e le sue funzionalità, come Cgroups e namespace, per isolare i processi in modo da poterli eseguire in maniera indipendente. Questa indipendenza è l'obiettivo dei container: la capacità di eseguire più processi e applicazioni in modo separato per sfruttare al meglio l'infrastruttura esistente pur conservando il livello di sicurezza che sarebbe garantito dalla presenza di sistemi separati.
Gli strumenti per la creazione di container, come Docker, consentono il deployment a partire da un'immagine. Ciò semplifica la condivisione di un'applicazione o di un insieme di servizi, con tutte le loro dipendenze, nei vari ambienti. Docker automatizza anche la distribuzione dell'applicazione (o dei processi che compongono un'applicazione) all'interno dell'ambiente containerizzato.Gli strumenti sviluppati partendo dai container Linux, responsabili dell'unicità e della semplicità di utilizzo di Docker, offrono agli utenti accesso alle applicazioni, la capacità di eseguire un deployment rapido, e il controllo sulla distribuzione di nuove versioni.

Per quanto riguarda il software che viene gestito da Docker, il quale rappresenta il vero middleware tra il codice che fornisce le istruzioni e l'hardware che aziona il robot, troviamo il sistema ROS (*Robot Operating System*) nella sua versione *Kinetic Kame*, approfondito anche durante le varie lezioni del Laboratorio di Intelligenza Artificiale e Grafica Interattiva.
I motivi principali che hanno portato a questa scelta sono stati diversi, tra cui, una grandissima disponibilità di librerie già pronte all'uso; l'indipendenza da un linguaggio di programmazione, considerando che ROS permette la comunicazione di moduli scritti in linguaggi diversi (*C++ & Python essendo i due principali*); una suite completa di strumenti per testare, visualizzare e salvare i risultati dei vari esperimenti e molti altri ancora.
L'architettura che ROS sfrutta si basa su pochi semplici elementi modulari, in particolare parliamo di *Nodes*, *Topics*, *Messages* e il concetto di *Publish/Subscribe*. Prima di analizzare i singoli moduli, consideriamo il nucleo che permette il funzionamento di ROS, ossia il cosiddetto *ROS Master*: sfruttando l'architettura TCP/IP, esso si occupa esclusivamente della gestione dei messaggi tra i vari Nodi, andando quindi a concedere l'accesso ai vari Topic che caratterizzano il nostro robot. Se il ROS Master non è attivo, l'applicazione non può funzionare.
I Nodi rappresentano gli eseguibili del nostro sistema, ognuno di essi viene gestito come un singolo thread e può pubblicare o sottoscriversi ad un certo Topic. Questi ultimi infatti non sono altro che dei canali di comunicazione di tipo many-to-many tra i vari Nodi. Ogni Topic si distingue per il tipo di messaggio associato ad esso, il quale può racchiudere tipologie primitive (*bool, string, float32, etc.*) oppure messaggi di natura più complessa, come geometry_msg (*Vector, Pose, Point, etc.*) e nav_msg (*Odometry, Path, etc.*).
Per riassumere con degli esempi concreti, osserviamo nella fig. 2.2 l'albero delle immagini Docker per un'istanza del Duckiebot, mentre la fig. 2.3 racchiude un esempio di Nodi (*Ovali*) e Topic (*Rettangoli*) per il sistema ROS di Lane Following, con le finestre di supporto visuale dei vari moduli (fig. 2.4).

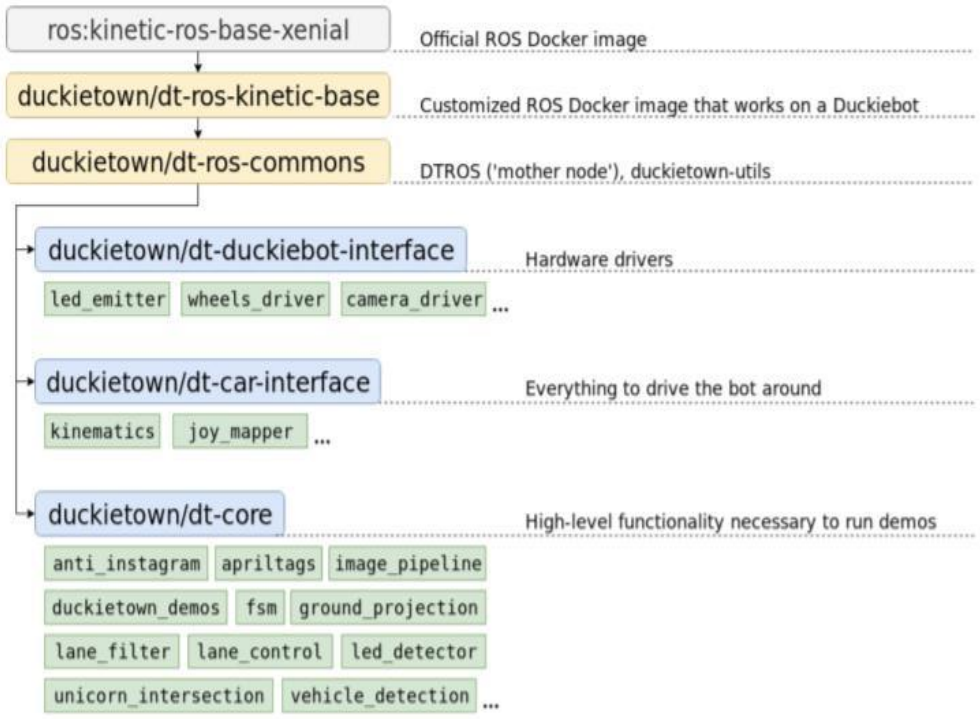


Figura 2.2. Albero delle Immagini Docker per Duckiebot

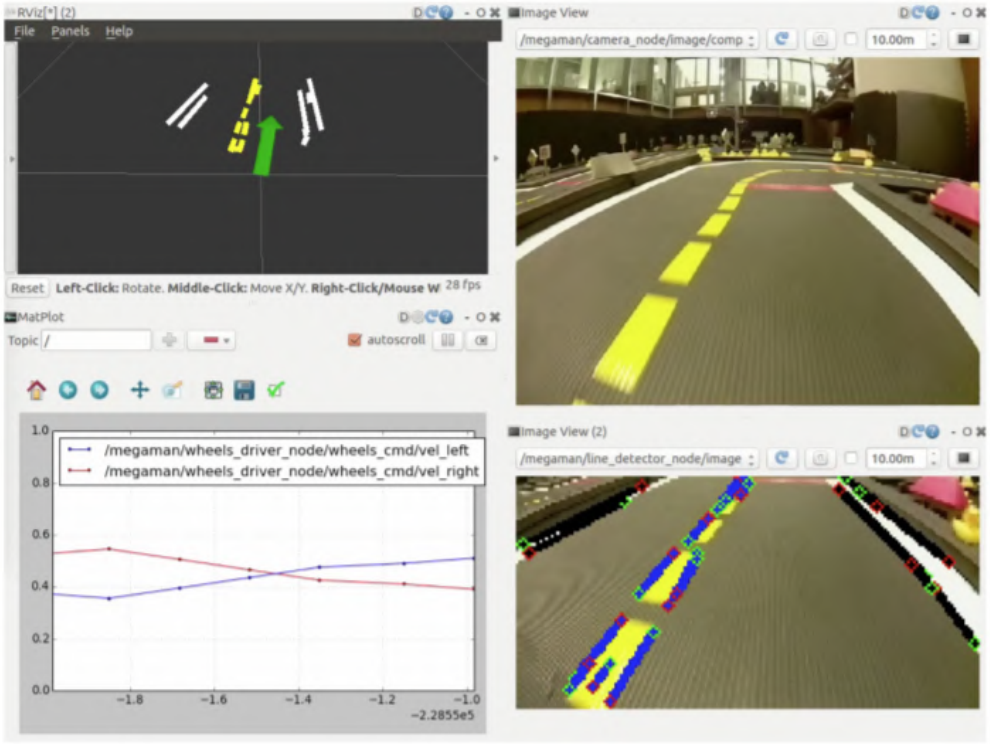


Figura 2.3. Moduli ROS Lane Following Duckiebot

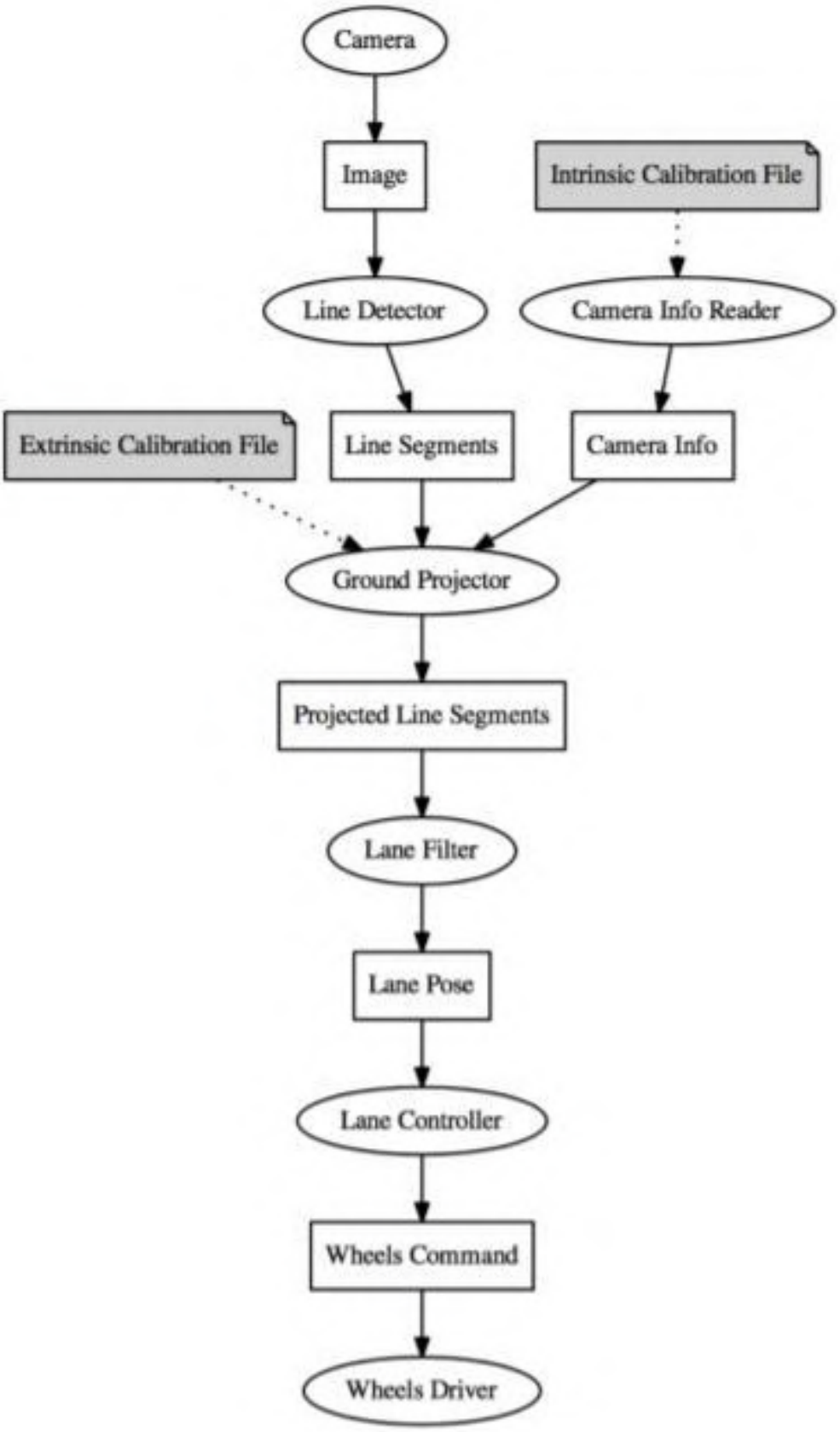


Figura 2.4. Moduli ROS Lane Following Duckiebot

2.1.2 Duckietown

Le Duckietown sono ambienti fortemente strutturati costruiti intorno al corretto funzionamento dei Duckiebot. Tutte le specifiche e le indicazioni esistono in funzione delle mansioni dei singoli bot. Questo tipo di approccio permette inoltre di modulare la difficoltà dei problemi affrontati, poiché buona parte delle strutture sono complementari e non necessarie nei confronti della sostenibilità del mondo simulato. Vi sono due componenti principali che danno vita alla città: il "floor layer" e il "signal layer *(vedi rispettivamente fig. 2.5 e fig. 2.6)*.
Il "floor layer" rappresenta la segnatura che definisce la topologia e la rete della strada. Questo strato è del tutto modulare poiché viene assemblato tramite l'incastro di semplici piastrelle, le quali vengono poi personalizzate utilizzando del nastro adesivo in base alla mappa che si vuole creare.
Il "signal layer" invece comprende tutti i segnali che i robot usano per navigare la Duckietown. Ci sono due elementi principali che compongono questo strato: i segnali e i semafori. I segnali contengono informazioni riguardo il tipo di incroci ed il nome delle strade. Ogni segnale è inoltre provvisto di un codice QR denominato "AprilTag" che semplifica il lavoro di percezione e riconoscimento. I semafori infine gestiscono l'accesso agli incroci.

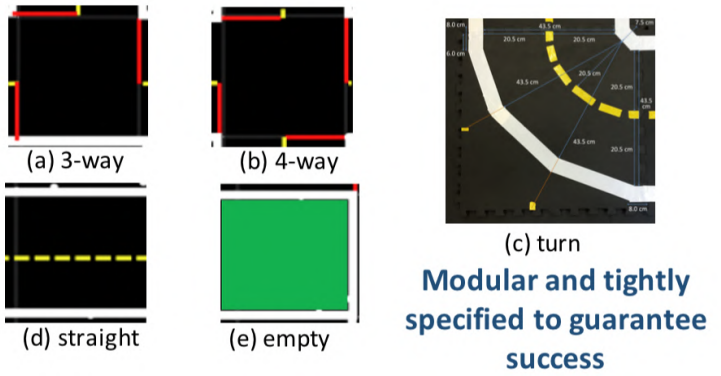


Figura 2.5. Floor Layer

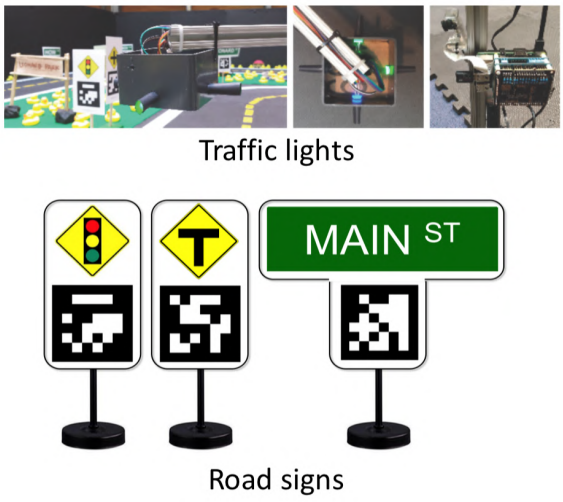


Figura 2.6. Signal Layer

Sulla base di queste conoscenze, ho avuto l'opportunità di costruire presso il Laboratorio del Dipartimento di Ingegneria informatica automatica e gestionale (DIAG) una piccola Duckietown, finalizzata a complementare il Test Set del Dataset (vedi sezione 2.2) utilizzato nell'analisi delle Reti Neurali a mia disposizione.

Avendo come obiettivo quello di costruire degli scenari ad hoc per poter testare determinate situazioni all'interno della Duckietown, si è deciso di procedere alla creazione di un ambiente generale che riuscisse a coprire la maggior parte dei casi ipotizzati. Il punto centrale è rappresentato dall'incrocio della fig. 2.7 che fa riferimento allo standard *a)* fig. 2.5. Esso si dirama in altre tre strade che non sono state collegate tra di loro data la mancata necessità di creare un circuito chiuso. Elementi come nomi delle strade e semafori sono stati omessi poiché andavano oltre lo scopo dell'esperimento. Per una descrizione dettagliata e precisa sulle dimensioni, posizioni e regole seguite nel processo di costruzione, si faccia riferimento alla guida [1] della Bibliografia.

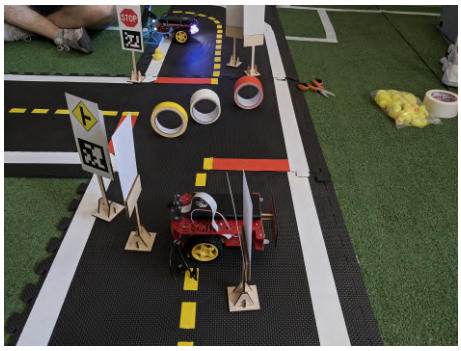
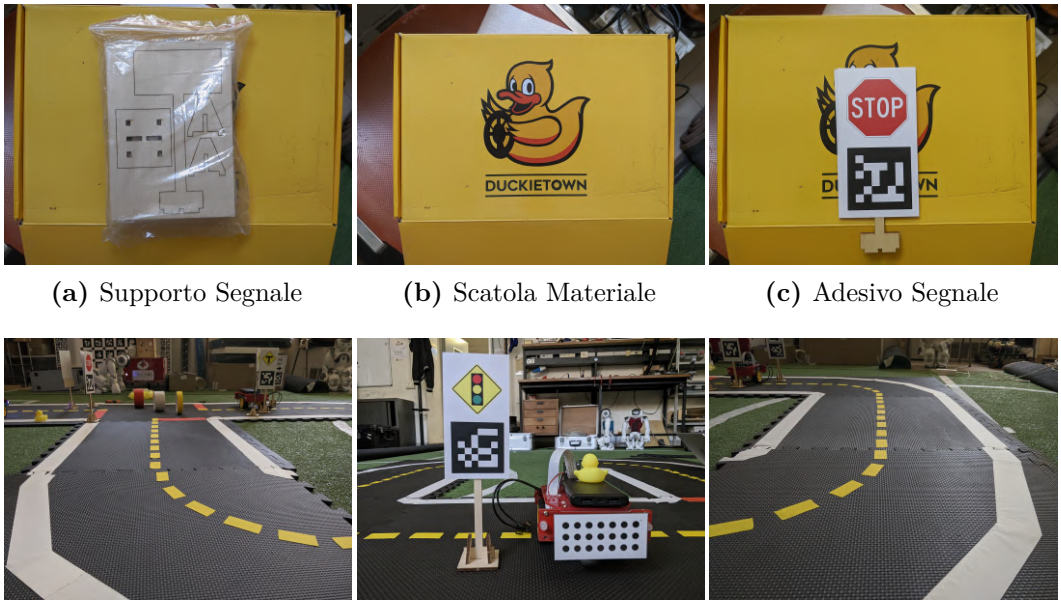


Figura 2.7. Incrocio Principale

Il processo di costruzione ha richiesto l'utilizzo di 9 piastrelle, 5 segnali stradali, 3 tipologie di nastro carta e moltissime paperelle. Una volta collegate tra di loro le varie piastrelle, sono stati tracciati i segni della carreggiata esterna *(nastro bianco)*, i divisori di corsia *(nastro giallo tratteggiato)* ed infine i segni di stop *(nastro rosso)*. Il passo successivo è stato posizionare i segnali stradali all'interno dell'incrocio, in particolare i segnali selezionati sono stati: stop, dare precedenza, strada a senso unico e obbligo di svolta. Le paperelle ed i bot erano sprovvisti di una posizione fissa poiché erano necessari al fine di creare i diversi scenari di interesse.



(a) Supporto Segnale **(b)** Scatola Materiale **(c)** Adesivo Segnale

(d) Strada Laterale **(e)** Bot e Segnali **(f)** Corsia Vuota

Figura 2.8. Materiale Laboratorio

2.2 Dataset

Il dataset utilizzato per la fase di allenamento delle Reti Neurali è stato recuperato da una repository su GitHub *(vedi Bibliografia [2])*. L'ambiente di interesse era appunto quello di Duckietown, in particolare si trattava di foto scattate da un Duckiebot in laboratorio, sotto condizioni di luci uniformi e con interesse nei confronti di 4 elementi: i bot, le papere, i segnali stradali e i segnali di stop.
Le immagini recuperate in totale sono 420, con una distribuzione di 378 immagini per il Training Set e 21, rispettivamente, per il Validation Set e per il Test Set.
Qui di seguito una tabella riassuntiva *(si faccia riferimento anche alle 2.11 e 2.12)*:

Class Name	Training Set *(378 img)*	Validation Set *(21 img)*	Test Set *(21 img)*
bot [0]	483	34	24
duckie [1]	820	63	33
stop_sign [2]	451	22	25
road_sign [3]	1198	74	69

Figura 2.9. Ripartizione Dataset per Allenamento

Ogni classe presenta delle particolarità rilevanti al fine del processo di Training, in particolare, prima di procedere con l'allenamento delle Reti, sono state concepite delle ipotesi, le quali verranno convalidate o meno nel paragrafo 5:

- bot: il numero di riferimenti a questa classe all'interno delle immagini risulta particolarmente basso, considerando le possibili variazioni in posizione ed aspetto. Infatti troviamo 4 possibili orientamenti, nonché una variazione nella presenza o meno delle luci e della papera che si trova alla guida del bot. Inoltre rappresenta l'unica classe in movimento tra quelle analizzate, rendendo decisamente più complicato il loro riconoscimento. Infine si presenta molto spesso in zone molto popolate dell'immagine andando a complicare il processo di isolamento delle sue features;

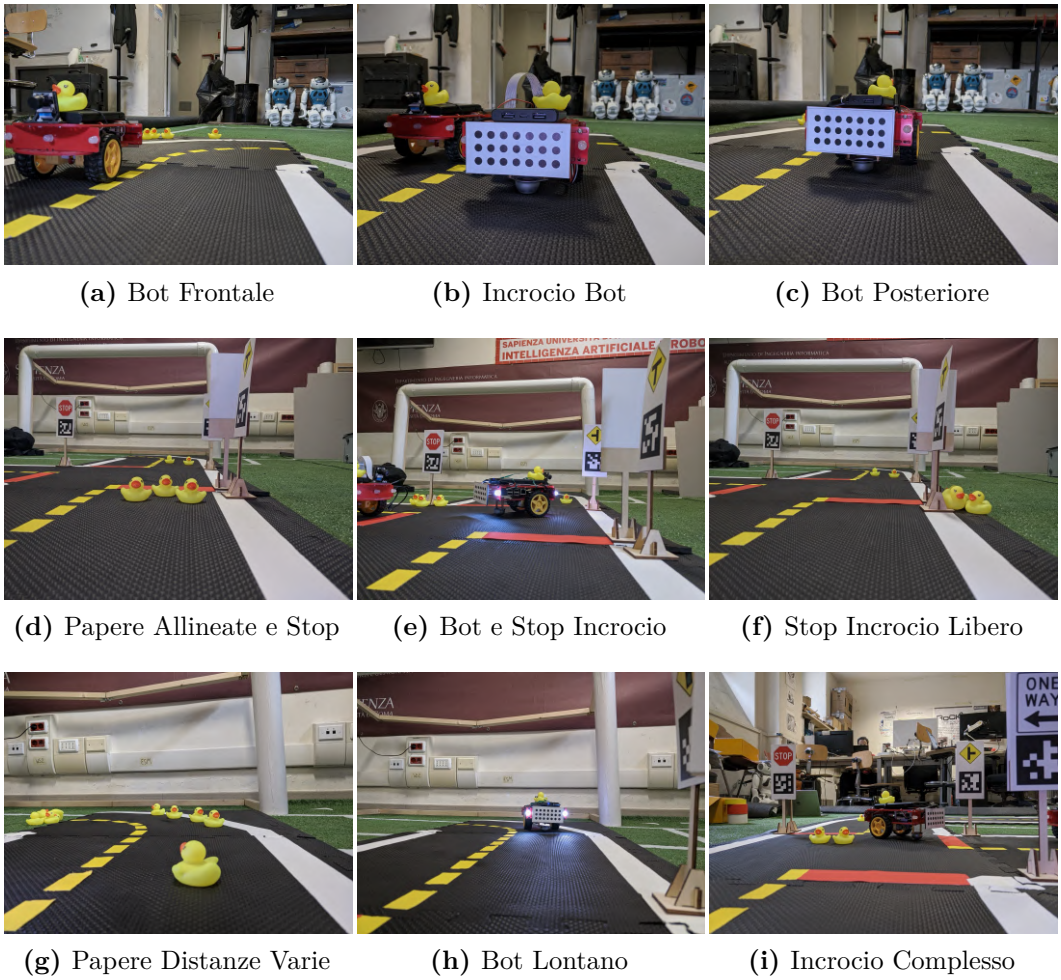
- duckie: seppur presenti nella maggior parte delle immagini, si trovano sempre in modo gruppi numerosi, quasi mai isolate e difficilmente in posizioni prevedibili. Un'ulteriore complicazione è la mancanza di esemplari all'interno della carreggiata, i quali dovrebbero rappresentare il fenomeno di un pedone che attraversa;

- stop_sign: la classe più stabile e costante all'interno del dataset, sia per forma che per posizione. Il numero limitato in questo caso potrebbe giusto portare ad eventuali problemi di overfitting, che però grazie all'uso di Data Augmentation possono essere mitigati;

- road_sign: la classe più numerosa ma anche più variabile tra tutte quelle presenti poiché non è stata fatta alcuna distinzione tra i vari possibili segnali; distinguendoli quindi soltanto dal segnale di Stop. Elemento di interesse è il

codice QR che troviamo sotto ogni segnale, che in questo caso potrebbe dare dei problemi poiché porta alla diversificazione di determinate features in quella zona.

Sommariamente il dataset risulta abbastanza diversificato, con un numero di immagini che, considerando il task di interesse, potrebbe causare problemi di overfitting, ossia i risultati ottenuti alla fine del training per quanto riguarda la precisione, saranno molto più alti di quello che le reti sono realmente capaci di predire su un dataset mai visto.
Per sopperire a questa eventualità, e per poter confermare o meno le problematiche avanzate nella descrizione precedente delle varie classi; è stato raccolto un piccolo campione di circa 20 immagini all'interno del percorso costruito ad hoc in laboratorio, come documentato nella sezione 2.1. Si è cercato di isolare laddove possibile le classi di interesse e presentare scenari più o meno complessi al fine di poter testare le capacità delle reti.
A seguire una raccolta di quanto descritto su questo nuovo Test Set *(fig. 2.10)*:



(a) Bot Frontale **(b)** Incrocio Bot **(c)** Bot Posteriore

(d) Papere Allineate e Stop **(e)** Bot e Stop Incrocio **(f)** Stop Incrocio Libero

(g) Papere Distanze Varie **(h)** Bot Lontano **(i)** Incrocio Complesso

Figura 2.10. Immagini di Interesse Test Set LABIAGI

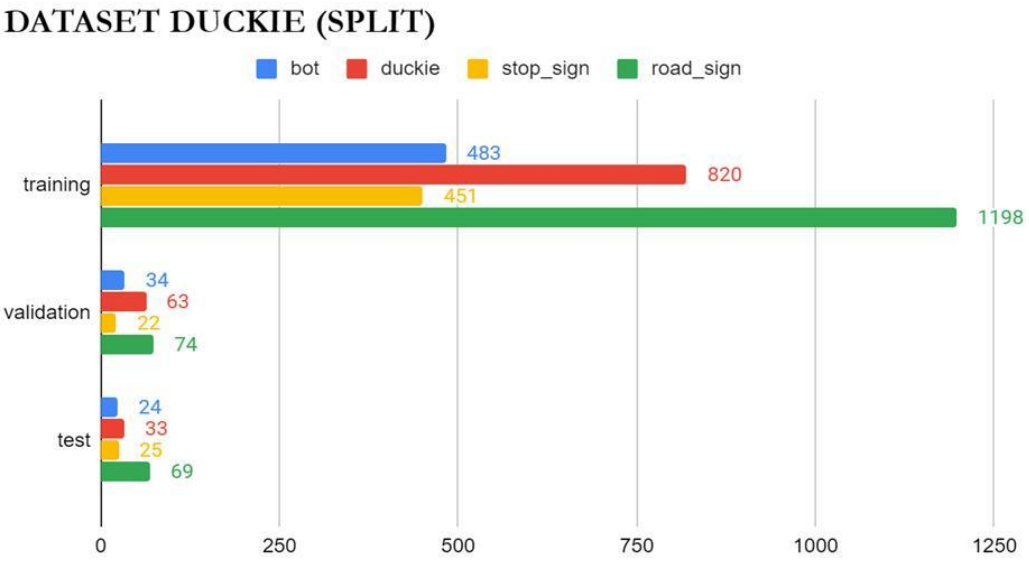


Figura 2.11. Split di Training, Validation e Test Set

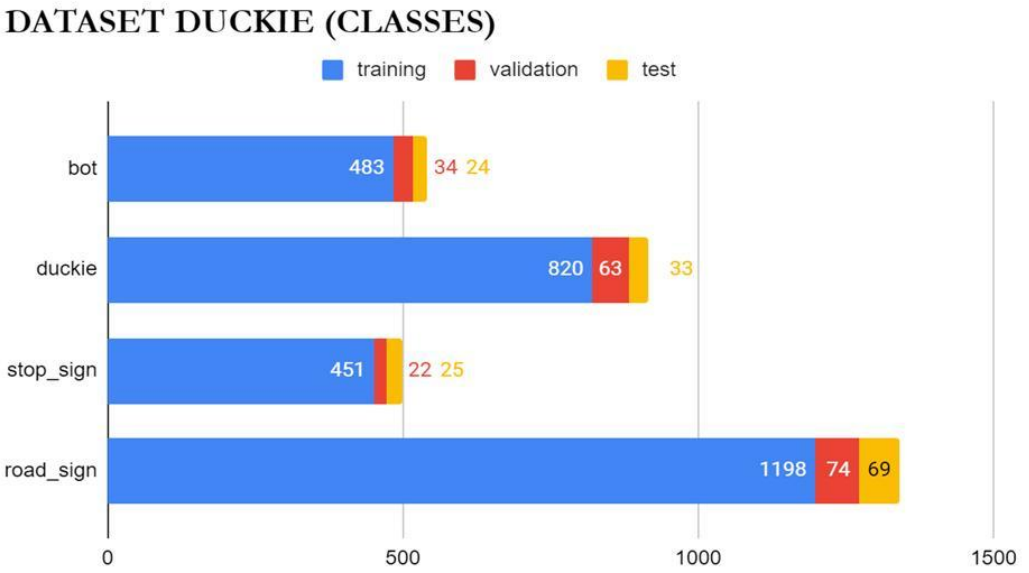


Figura 2.12. Distribuzione delle 4 Classi tra i vari Set

2.3 YOLO

Il processo di Object Detection può essere affrontato in diversi modi a seconda delle necessità della propria applicazione. Gli algoritmi utilizzati al momento per affrontare questo tipo di problema sono fondamentalmente due: Classificazione e Regressione.
I primi necessitano di un processo a due fasi, prima vengono selezionate le zone di interesse dell'immagine, per poi andare a classificarle usando delle Reti Convoluzionali. Questa tipo di approccio però può risultare lento poiché la predizione viene fatta per ogni singola regione. Alcuni famosi esempi per quanto riguarda questo tipo di algoritmo sono le Region-based convolutional neural network (RCNN), con delle loro varianti più preformanti denominate Fast-RCNN e Faster-RCNN.
I secondi invece provano a predire classi e bounding boxes *(BB)* per l'intera immagine in una singola fase, rendendo quindi molto più veloce il processo di detection. Uno degli algoritmi più famosi è proprio YOLO (You Only Look Once), il quale viene preferito per applicazioni di tipo real-time dove è necessario un tempo di risposta minore rispetto ad una precisione vicina all'ottimo.

L'idea di base è quella di suddividere l'immagine in una tabella *SxS*, dove ogni cella è responsabile nel predire un singolo oggetto. Ogni cella però a sua volta predice molteplici BB e molteplici probabilità condizionali per ogni classe.
La predizione da parte di YOLO quindi viene fatta nei confronti di una determinata classe e della sua BB rispetto della sua posizione. Ogni predizione può quindi essere descritta da 4 parametri (vedi fig. 2.13):

1. Centro della Bounding Box *(bx, by)*;
2. Larghezza *(bw)*;
3. Altezza *(bh)*;
4. Valore della classe di un certo oggetto *(c)*.

Inoltre ricordiamo anche il valore *(pc)*, ossia la probabilità che un oggetto di una certa classe si trovi in quella Bounding Box.

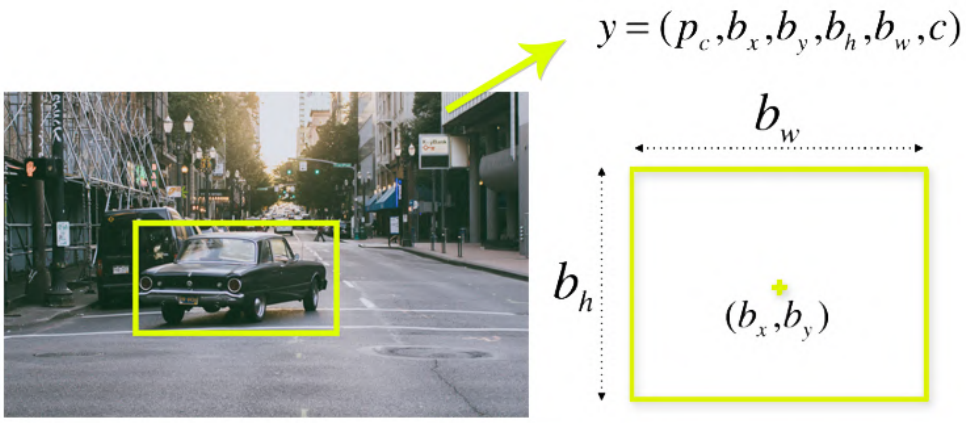


Figura 2.13. Parametri di Predizione Rete Neurale

Per poter calcolare la Loss nei confronti dei True Positive, ossia le predizioni corrette, dobbiamo selezionare quella con la maggior IoU *(Intersection Over Union)* rispetto al ground truth. Ogni predizione quindi si migliora nei confronti di determinate dimensioni e proporzioni.
YOLO utilizza l'errore quadratico medio tra le predizioni e il ground truth per calcolare la loss. Quest'ultima si sviluppa dall'unione di 3 componenti (vedi formula completa nella fig. 2.14):

- La loss della Classificazione;
- La loss della Localizzazione *(errore tra la BB e il ground truth)*;
- La loss della Confidenza *(l'oggettività della BB)*.

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

Figura 2.14. 1° e 2° membro Localizzazione, 3° e 4° Confidenza ed infine Classificazione

La versione utilizzata nei vari esperimenti è stata la v3, che presenta diverse migliorie sia dal punto di vista della precisione che della velocità rispetto alla precedente v2. Un cambiamento degno di nota è quello della rete utilizzata per estrarre le features, la quale utilizza strati convoluzionali 3x3 e 1x1 e risulta decisamente più robusta poichè passa da 19 a 53 strati. Ulteriori dettagli vengono illustrati nella fig. 2.15.

	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 2.15. Rete Darknet-53 Completa

Capitolo 3

Metodologia

All'interno di questo capitolo verrà descritto il lavoro svolto all'interno del Laboratorio per quanto riguarda il setup del Duckiebot e lo studio necessario alla preparazione delle due reti neurali per poter svolgere il loro allenamento; nonché la verifica delle loro rispettive performance sui Test Set disponibili.
La prima parte verterà quindi su una descrizione concreta delle mansioni effettuate al fine di rendere operativo il Duckiebot, tra successi e complicazioni; mentre la seconda vede un'analisi delle due reti neurali, la teorizzazione di alcun ipotesi correlate alla scelta dei parametri di interesse ed infine il metodo utilizzato successivamente nel capitolo Capitolo 4.

3.1 Setup Duckiebot

Una volta finalizzata la costruzione della Duckietown all'interno de laboratorio, è stato necessario effettuare un setup dei Duckiebot al fine di poterli utilizzare per il processo di bagging. Tutte gli step eseguiti si rifanno alla guida [3] della Bibliografia. Il processo di setup è stato svolto interamente in ambiente Linux, in particolare utilizzando la versione 18.04. E' stato inoltre necessario garantire varie dipendenze all'interno della propria macchina, tra le quali troviamo *"python3-pip git git-lfs curl wget"* ed alcune più specifiche come *"Docker"*, trattato all'interno della sezione 2.1.1, e la *"Duckietown Shell"*, utilizzata principalmente all'interno di questa sezione.
Da notare inoltre che, durante una prima parte del setup, è stata utilizzata una Virtual Machine (VM), la quale ha richiesto un processo di Bridge al fine di poter condividere la stessa sotto rete con il Duckiebot. Questo processo però ha causato diverse complicazioni, richiedendo in un secondo momento il passaggio ad un'installazione pulita su disco.

All'interno del laboratorio erano presenti già due esemplari di Duckiebot costruiti da altri studenti, denominati rispettivamente *ulysses* e *romolos*. Essendo essi già stati inizializzati da altri studenti, l'unico step richiesto al fine di poter operare uno dei due bot, sarebbe stato quello del Networking. Prima di poter procedere però, è stato necessario riconfigurare la rete di riferimento per il suddetto bot all'interno del file *"wpa_supplicant.conf"* della directory *"root/etc/wpa_supplicant/"*; questo poiché essa era stata impostata ad una rete differente da quella disponibile all'interno del Laboratorio. Sapere a quale rete è connesso il nostro bot risulta fondamentale poichè, oltre al nome, il suo unico altro riferimento utile è dato dal suo indirizzo IP che viene mostrato una volta effettuata la connessione alla rete. Una volta risolto questo problema, il prossimo passo è stato accendere il Duckiebot

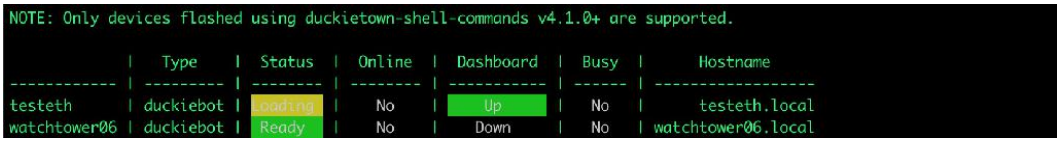


Figura 3.1. Esempio risultato dts fleet discover

e tramite il comando "dts fleet discover" monitorare l'avvio di quest'ultimo. In particolare, prima di poter procedere, bisogna aspettare che la voce "Dashboard" sia "Up" come nella fig. 3.1. Inoltre, se la connessione alla rete impostata è avvenuta con successo, anche la voce "Online" dovrebbe essere "Yes". Per testare il funzionamento della connessione internet è possibile operare in due varianti: provare ad eseguire "ping host_name","ping ip-host_name" oppure, laddove i ping venissero bloccati dalla rete, il comando "sudo curl google.com" all'interno del duckiebot. Per poter operare il bot tuttavia è necessario che anche lo stato "Status" sia "Ready" e per fare ciò bisogno effettuare il setup del Duckiebot tramite la Dashboard disponibile presso l'indirizzo "http://hostname.local/". Per poter accedere bisogna utilizzare il token fornito dal sito Duckietown una volta completata la registrazione. Una volta completato l'accesso, si avranno a disposizione una serie di strumenti fondamentali necessari a comandare il bot. Di particolare interesse risulta la sezione "Portainer", la quale permette di visualizzare tutti i container Docker attualmente attivi sul proprio Duckiebot. Una volta quindi finalizzato il processo di setup appena descritto, si è provato tramite la Duckietown Shell, ad utilizzare il comando "dts duckiebot keyboard_control DUCKIEBOT_NAME" il quale dovrebbe restituire una interfaccia grafica capace di muovere il nostro bot. Sfortunatamente però, questo comando ha restituito ad ogni prova un errore rispetto all'impossibilità di comunicare con il "ROS Core", rendendo inutilizzabile il robot. Andando a controllare lo stato dei container, effettivamente lo stato del container "roscore" risultava "unhealthy". Avendo provato tutti i consigli indicati dalla guida [3] e non avendo modo di sanare lo stato del container indicato; si è pensato di riprovare tutto il processo di Setup sull'altro Duckiebot disponibile. Quest'ultimo però presentava problemi nella fase di connessione ad internet, bloccando di conseguenza la possibilità di procedere e completare gli step necessari.
Ipotizzando una possibile corruzione del sistema operativo installato sul bot, avendo avuto conferma dal docente che il bot in questione non presentasse errori a livello hardware; si è provato ad installare da zero il sistema operativo sul Duckiebot. Per poter fare questo, come già indicato all'inizio della sezione, si è inoltre provveduto ad installare su disco il sistema operativo Linux, per evitare la necessità di operare su VM ed avere necessità di una connessione Bridge.
Una volta completato il processo di installazione, si è riusciti a superare il problema della connessione alla rete; ritornando però ad avere lo stesso problema nei confronti della comunicazione con il "ROS Core" a setup completato. Si è concluso quindi che l'immagine Docker caricata sul bot non venisse correttamente inizializzata, negando la possibilità di usufruire di quest'ultimo.

3.2 YOLO vs YOLO-TINY

Come già anticipato, lo studio e l'allenamento delle due reti è stato fatto in funzione di un'eventuale implementazione all'interno della piattaforma Duckietown. Trattandosi quindi ora di un problema real-time di Object Detection, i parametri di interesse non potevano essere semplicemente la precisione delle varie predizioni; ma andavano considerati anche i tempi nei quali queste predizioni venivano fatte. L'algoritmo YOLO rappresenta però già una delle migliori alternative tra tutte le possibili implementazioni di Object Detection (vedi fig. 3.2), sopratutto se messo a confronto con le classiche R-CNN. Tuttavia, esiste anche una variante TINY di YOLO, la quale avvalendosi di una rete più piccola e di alcune modifiche strutturali, conferisce una velocità molto più elevata al processo di detection.
La rete YOLO, come anticipato nella sezione 2.3, sfrutta 53 strati Convoluzionali per creare una mappa riassuntiva di tutte le feature di interesse all'interno dell'immagine. L'uso di un numero così alto di strati permette agli strati più profondi di ricordare feature di alto livello, come forme oppure oggetti specifici, mentre a quelli meno profondi di ricordare feature più primitive come le linee che compongono gli oggetti. La rete TINY si basa sullo stesso principio, ma riduce drasticamente il numero di strati a 19 e prima di utilizzare gli strati "yolo", al fine di migliorarne l'efficienza, introduce dei filtri denominati di "max_pooling" necessari a ridurre le dimensione della mappa calcolata dallo strato convoluzionale. Volendo paragonare le due reti così come sono state sviluppate, non si sono apportate modifiche agli iperparametri oppure agli strati che le costituiscono. In fase di allenamento inoltre, la dimensione delle immagini è stata di 416x416. Quest'ultima è funzionale sopratutto a colmare note difficoltà della rete nel lavorare con immagini di piccole dimensioni.

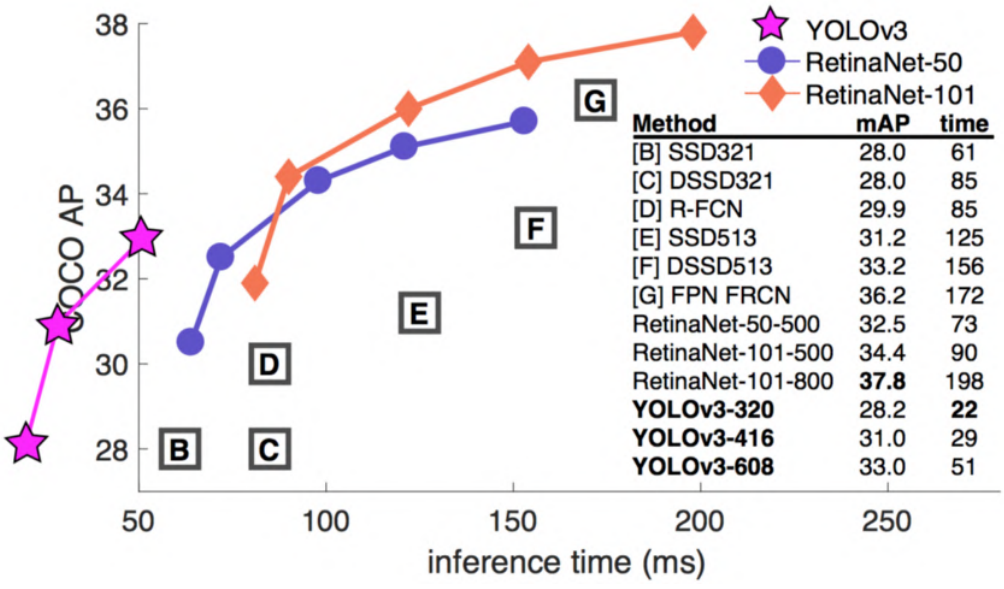


Figura 3.2. Performance Modelli Object Detection

Le metriche di interesse considerate per questo confronto, in ordine di importanza, sono state:

- la mAP *(Mean Average Precision)*;
- il tempo di detection;
- il tempo di allenamento;
- la dimensione dei pesi restituiti dagli allenamenti.

Il processo di training e preparazione del dataset risultano comuni alle due reti, tranne per l'utilizzo dei file cfg che ne descrivono le composizione. L'ambiente di lavoro è stato Google Colaboratory[1], una piattaforma online gratuita che offre un servizio di cloud hosting di Jupyter Notebooks[2], con il supporto a GPU. Avere a disposizione queste risorse è stato fondamentale per poter svolgere il processo di training in tempi utili che altrimenti tramite CPU sarebbe risultato aggravante.
La preparazione del dataset è stata fatta seguendo le indicazioni fornite nella guida [4]. I parametri di interesse, comuni ad entrambe le reti, sono stati la dimensione delle immagini, il numero di classi e la definizione dei vari set (training, validation e test) con le loro rispettive posizioni. Il dataset utilizzato era già provvisto dei file txt dove venivano descritte le posizione assolute delle BB. Il software utilizzato per la fase di training è stato "Darknet", un framework open source sviluppato dallo stesso autore di YOLO proprio per lavorare sulle Reti Neurali, sfruttando i linguaggi C e CUDA. Prima di caricare il dataset sul notebook tramite Google Drive, quest'ultimo necessitava un processo di setup comune anche qui ad entrambe le reti. Il primo step è stato quello di installare i driver CUDA di NVIDIA, fondamentali per poter sfruttare la GPU durante l'allenamento; a seguire è stato compilato il codice necessario per poter utilizzare la suite di comandi di "Darknet" ed infine, una volta concessi i permessi necessari, si è potuto far partire il processo di training.

[1] https://colab.research.google.com/notebooks/intro.ipynb#scrollTo=5fCEDCU_qrC0
[2] https://jupyter.org/about

Capitolo 4

Risultati

4.1 Allenamento YOLO

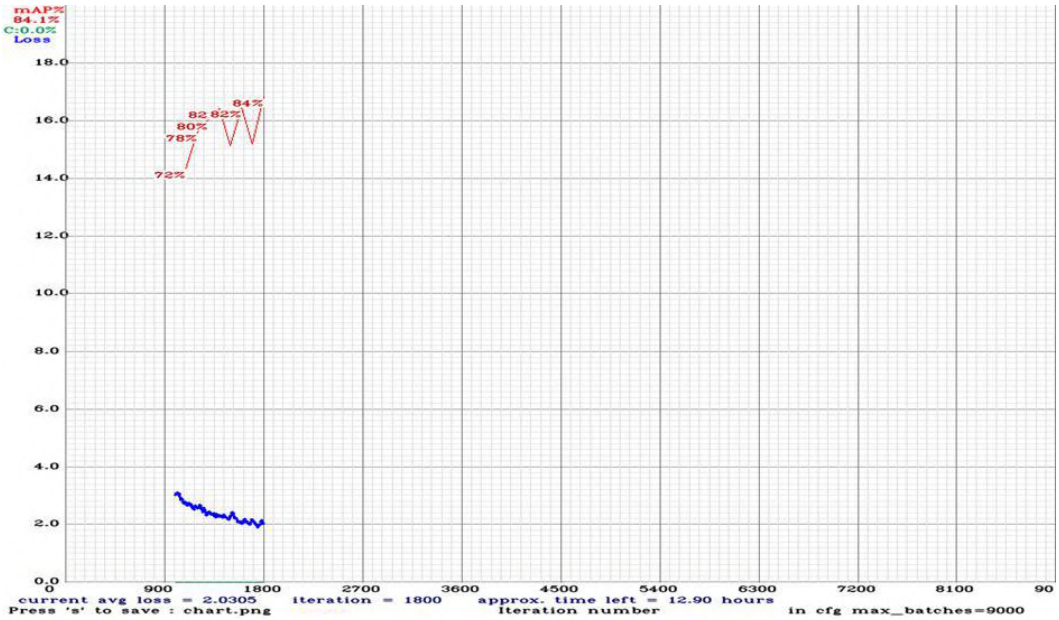


Figura 4.1. Chart Allenamento YOLO

```
 detections_count = 406, unique_truth_count = 193
class_id = 0, name = bot, ap = 86.98%           (TP = 30, FP = 5)
class_id = 1, name = duckie, ap = 79.08%        (TP = 49, FP = 12)
class_id = 2, name = stop_sign, ap = 95.45%     (TP = 21, FP = 0)
class_id = 3, name = road_sign, ap = 89.38%     (TP = 63, FP = 12)

 for conf_thresh = 0.25, precision = 0.85, recall = 0.84, F1-score = 0.85
 for conf_thresh = 0.25, TP = 163, FP = 29, FN = 30, average IoU = 60.26 %

 IoU threshold = 50 %, used Area-Under-Curve for each unique Recall
 mean average precision (mAP@0.50) = 0.877232, or 87.72 %
Total Detection Time: 12 Seconds
```

Figura 4.2. Output Fine Allenamento YOLO

4.2 Allenamento TINY

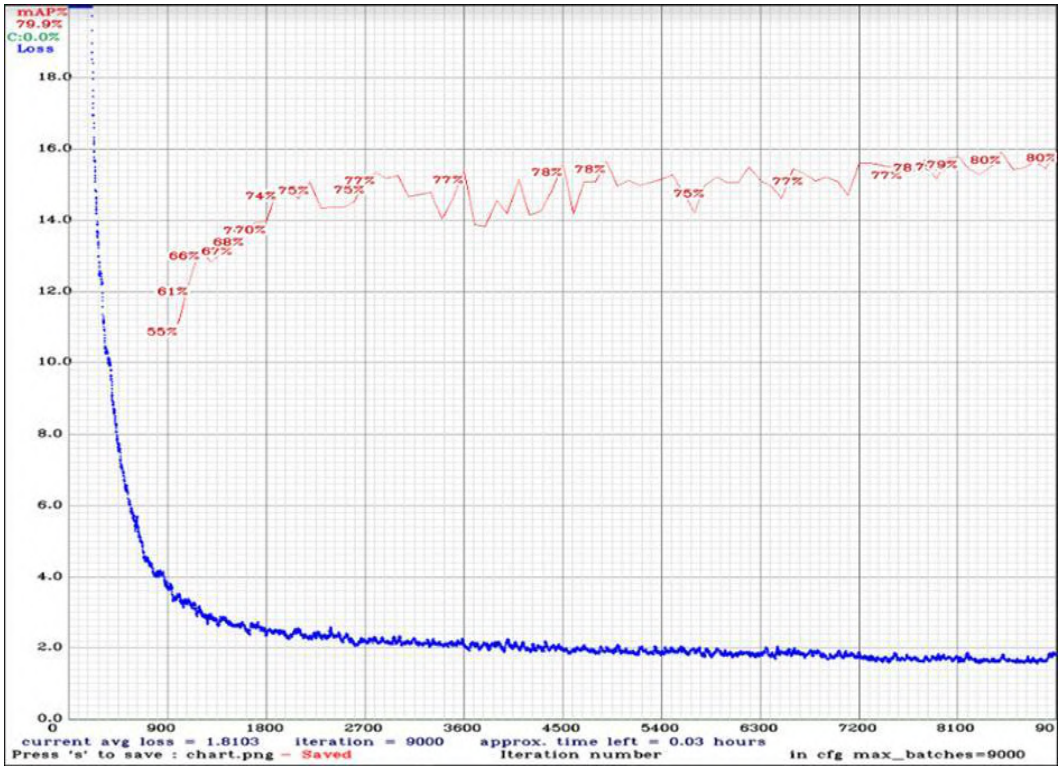


Figura 4.3. Chart Allenamento TINY-YOLO

```
 detections_count = 393, unique_truth_count = 193
class_id = 0, name = bot, ap = 86.24%          (TP = 29, FP = 8)
class_id = 1, name = duckie, ap = 59.34%       (TP = 44, FP = 17)
class_id = 2, name = stop_sign, ap = 87.45%    (TP = 20, FP = 1)
class_id = 3, name = road_sign, ap = 86.27%    (TP = 63, FP = 16)

 for conf_thresh = 0.25, precision = 0.79, recall = 0.81, F1-score = 0.80
 for conf_thresh = 0.25, TP = 156, FP = 42, FN = 37, average IoU = 55.12 %

 IoU threshold = 50 %, used Area-Under-Curve for each unique Recall
 mean average precision (mAP@0.50) = 0.798236, or 79.82 %
Total Detection Time: 1 Seconds
```

Figura 4.4. Output Fine Allenamento TINY-YOLO

L'allenamento è stato eseguito seguendo le indicazioni che troviamo nel post GitHub [4] della Bibliografia. Dovendo allenare la rete su 4 classi differenti, il riferimento era di utilizzare un numero di batch multiplo di 2000 × classi; in ogni caso non meno di 6000. Questo ha rappresentato un problema per quanto riguarda la rete YOLO, poichè, come mostrato nella fig. 4.1, l'allenamento ha raggiunto soltanto ~2000 batch.
Questo è dovuto a complicazioni riscontrate con la piattaforma Google Colab, la quale, dopo un certo periodo di tempo, rilascia le risorse allocate all'utente. Considerando che per raggiungere quei risultati ci sono volute ~7-8h; non è stato possibile, con i mezzi a mia disposizione, ottenere un allenamento ottimo nei confronti di questa rete. Il processo di allenamento però restituisce dei risultati graduali, che in caso di interruzione, possono essere comunque utilizzati. Sono stati proprio questi pesi quelli sfruttati nella fase di Detection sul nostro dataset. Queste considerazione verranno comunque sempre tenute in conto durante la sezione 4.3.

Fortunatamente l'allenamento della rete TINY-YOLO non ha richiesto un tempo di allenamento così estensivo, concludendosi in media intorno alle 3h, e non ha riportato alcun tipo di complicazione. Condividendo però buona parte delle feature con la rete YOLO (come descritto nella sezione 3.2), possiamo analizzare il suo processo di allenamento ed estendere buona parte delle considerazione anche alla rete principale.
Facendo riferimento alla fig. 4.3, osserviamo che la nostra mAP *(Mean Average Precision)* oscilla fortemente per buona parte del processo di allenamento: questo andamento è dovuto alla policy "Steps" del Learning Rate, la quale interviene soltanto intorno all'80% del numero massimo di batch e, solo allora, forza la convergenza all'ottimo ottenuto fino a quel momento.
L'andamento esponenziale della loss invece, è giustificato dal fenomeno di "transfer learning", dove per accelerare l'allenamento si utilizzano dei pesi pre-allenati (yolov4.conv.137); trasferendo appunto i risultati di quella rete sul nuovo dataset. Per ottimizzare questo processo inoltre, è stato introdotto un fattore di "burn-in" che blocca il valore del Learning Rate per le prime 1000 batch, dove la discesa della loss è considerevole. Indicativamente, una volta sotto il valore di 2.5 per la loss, i pesi restituiti dall'allenamento risultano utilizzabili per una prima fase di testing. Come già anticipato, ogni 1000 batch vengono restituiti dei nuovi pesi, fino a produrre un peso finale ed uno ritenuto migliore rispetto ai risultati ottenuti durante tutta la durata dell'allenamento. Andando invece a commentare i risultati restituiti dalle reti YOLO e TINY-YOLO, rispettivamente nelle fig. 4.2 e 4.4; si osservano, in merito alla mAP, risultati più che soddisfacenti per la prima, considerato il bassissimo numero di iterazioni, mentre per la seconda abbiamo un valore decisamente più basso che però, nella fase di testing, si è dimostrato più reale rispetto ai valori inflazionati delle rete YOLO. Come anticipato nella sezione 2.2, la classe del segnale di Stop è stata quella che ha restituito il minor numero di Falsi Positivi (FP), data appunto la loro costanza; mentre le paperelle ed i segnali stradali hanno avuto diversi problemi.
Per quanto riguarda la dimensione dei pesi, la rete YOLO ha restituito dei file di dimensioni ~240MB, mentre TINY-YOLO di ~30MB. Infine concludiamo che le percentuali ottenute, per entrambe le reti, rispetto ai risultati illustrati nella sezione 4.3, sono decisamente troppo alti e dimostrano una problematica di over-fitting; sopratutto per quanto riguarda il Test Set Originale se messo a confronto con quello del Laboratorio, che era stato pensato proprio per mettere in difficoltà le performance delle due reti allenate.

4.3 Predizioni YOLO vs TINY-YOLO

Per tutti gli esperimenti di questa sezione, la threshold base è stata del 40%. Le velocità di predizione sono funzione di utilizzi di GPU molto performanti, rispettivamente, una NVIDIA Tesla K80 per la sottosezione 4.3.1 e una NVIDIA Tesla P100 per 4.3.2.

4.3.1 Test Set Dataset

Fig. 4.5:
data/testset/204_frame1186.jpg: Predicted in 11.989000 milli-seconds.
stop_sign: 99% |road_sign: 71% |bot: 100% |duckie: 70% |
road_sign: 88% |road_sign: 89% |bot: 100% |duckie: 98% |bot: 97%

Fig. 4.6:
data/testset/204_frame1186.jpg: Predicted in 94.162000 milli-seconds.
stop_sign: 92% |bot: 99% | duckie: 66% | road_sign: 69% | bot: 88% | bot: 96%

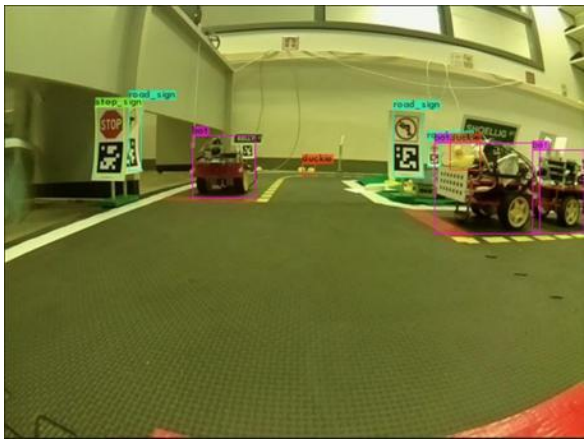


Figura 4.5. TINY-YOLO

Figura 4.6. YOLO

Fig. 4.7
data/testset/204_frame0286.jpg: Predicted in 11.932000 milli-seconds.
road_sign: 85% | stop_sign: 96%

Fig. 4.8
data/testset/204_frame0286.jpg: Predicted in 88.498000 milli-seconds.
stop_sign: 90%

Figura 4.7. TINY-YOLO

Figura 4.8. YOLO

Fig. 4.9
data/testset/203_frame0037.jpg: Predicted in 11.816000 milli-seconds.
stop_sign: 55% | bot: 81% | bot: 99%
road_sign: 75% | duckie: 89% | duckie: 91%
stop_sign: 86%

Fig. 4.10
data/testset/203_frame0037.jpg: Predicted in 88.861000 milli-seconds.
bot: 88% | bot: 62% | road_sign: 96%
duckie: 70% | duckie: 87% | stop_sign: 92%

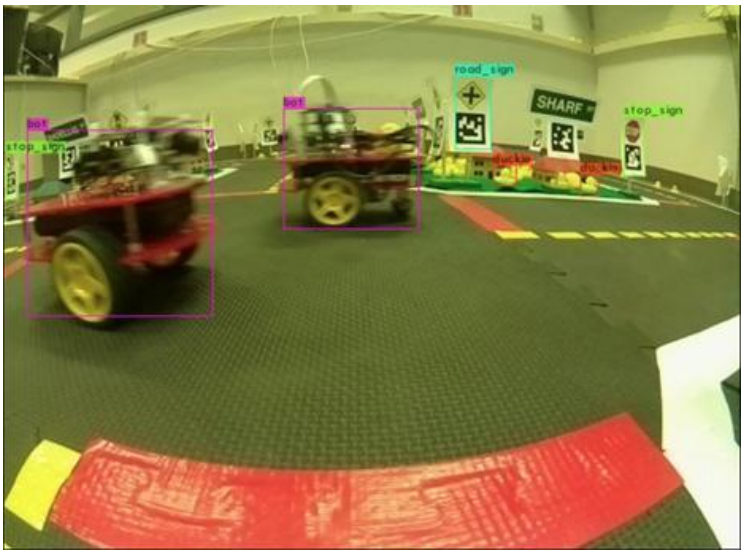


Figura 4.9. TINY-YOLO

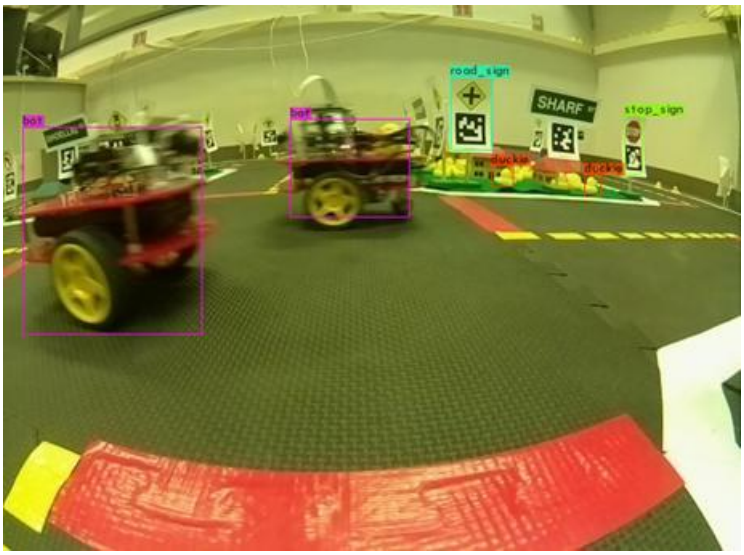


Figura 4.10. YOLO

Fig. 4.11
data/testset/203_frame0738.jpg: Predicted in 11.839000 milli-seconds.
duckie: 51% |road_sign: 94% | bot: 100% | road_sign: 91% | stop_sign: 84%
road_sign: 90% | duckie: 86% | duckie: 91% | stop_sign: 85%

Fig. 4.12
data/testset/203_frame0738.jpg: Predicted in 89.177000 milli-seconds.
duckie: 59% | road_sign: 91% | road_sign: 93% | bot: 93% | stop_sign: 84%
road_sign: 95% | duckie: 70% | duckie: 86% | stop_sign: 92%

Figura 4.11. TINY-YOLO

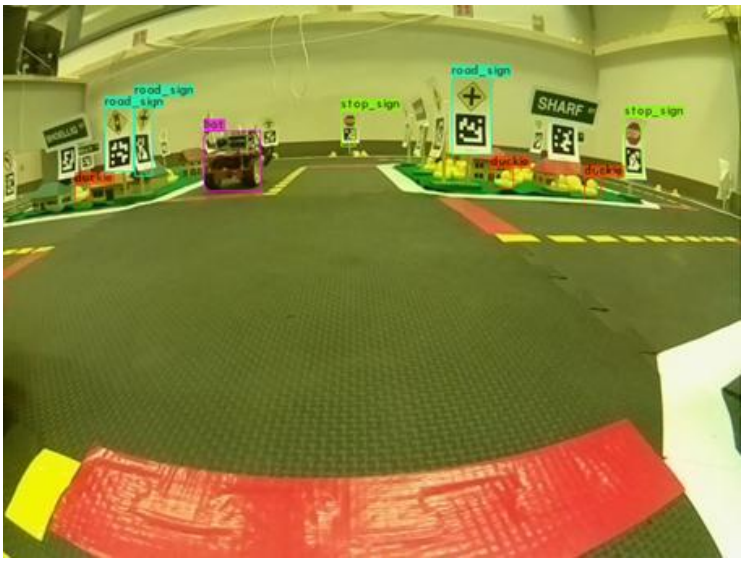


Figura 4.12. YOLO

Entrambe le reti su questo Test Set, seppur costituito da immagini del tutto assenti dal processo di allenamento; restituiscono risultati eccellenti. Questo è dovuto principalmente alla continuità nei confronti dei riferimenti che le Reti hanno sviluppato durante il processo di training. In particolare sorprendono i valori di precisione con la quale la rete YOLO riesce a riconoscere determinati elementi, date le considerazioni fatte nella sezione 4.1. Bisogna considerare infatti che, la rete TINY-YOLO intorno allo stesso numero di batch, risultava difficilmente utilizzabile; arrivando persino a confondere i divisori di corsia con delle paperelle ad un livello di confidenza vicino alla threshold.
Tuttavia l'allenamento completo della rete TINY permette ad essa di restituire risultati degni di nota, come nella fig. 4.5 dove riesce a discernere oggetti sovrapposti come i segnali stradali nella parte sinistra dell'immagine oppure la papera e il bot nella parte destra; cosa che invece YOLO non riesce a fare, riconoscendo solamente parte delle classi. Questo fenomeno si ripresenta anche nell'immagine 4.7 dove la rete TINY riconosce il segnale stradale in lontananza. Infine ritroviamo anche un caso di riconoscimento errato da parte delle rete TINY nella fig. 4.9, dove un segnale stradale viene scambiato per uno stop. Di per sé questo potrebbe rappresentare un problema, ma dato il grado di confidenza, la posizione del segnale ed il fatto che questi fenomeni sono stati decisamente rari, non è da considerarsi una debolezza della rete.
Sommariamente quindi, le due reti si comportano similmente con delle debolezze da parte delle rete YOLO contestualizzate però da un incompleto processo di allenamento. Le ipotesi fatte sulle singole classi sono state confermate, sopratutto per quanto riguarda i bot e le paperelle. Inoltre, una costante che supporta le ulteriori ipotesi fatte nella 3.2, sono i tempi di riconoscimento, i quali sono fortemente in favore delle rete TINY in tutti gli esperimenti effettuati.

4.3.2 Test Set Laboratorio

Fig. 4.13
data/foto_labiagi/TestSet/IMG_20200804_131740.jpg: Predicted in 5.597000 milli-seconds.
bot: 69% | duckie: 81% | duckie: 76%

Fig. 4.14
data/foto_labiagi/TestSet/IMG_20200804_131740.jpg: Predicted in 17.974000 milli-seconds.
bot: 50% | road_sign: 55% | road_sign: 51%
duckie: 87% | duckie: 43% | duckie: 84%

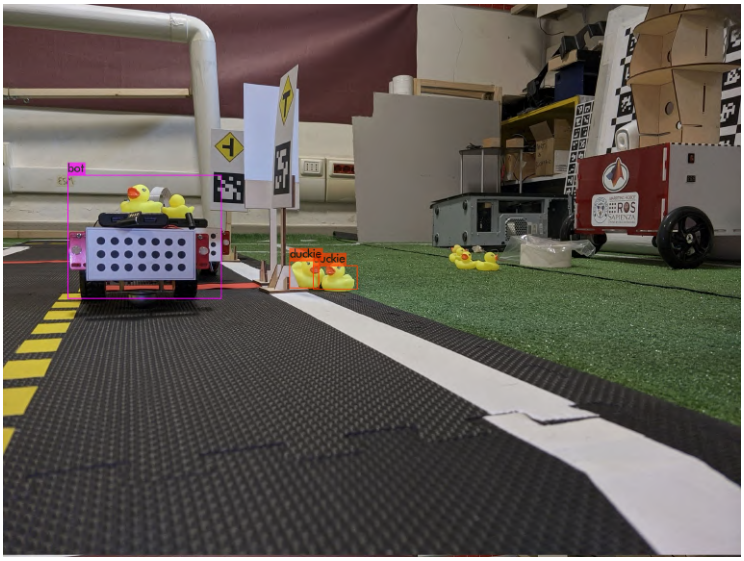


Figura 4.13. TINY-YOLO

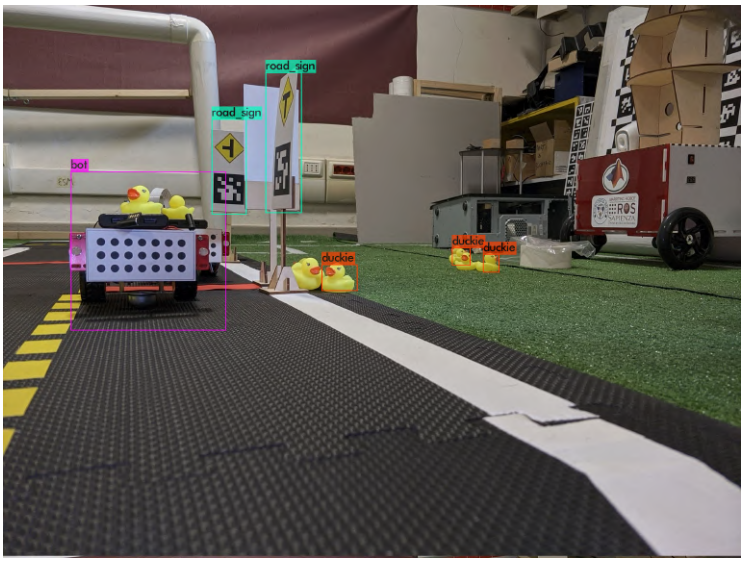


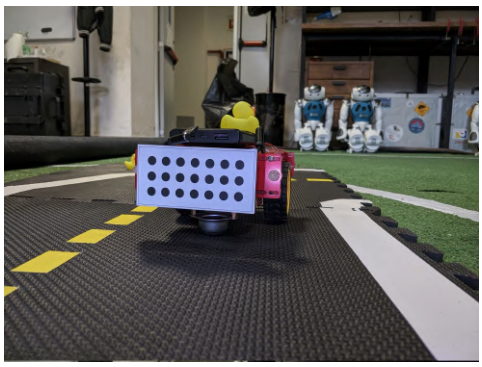
Figura 4.14. YOLO

Fig. a
data/foto_labiagi/TestSet/IMG_20200804_131905.jpg: Predicted in 3.254000 milli-seconds.
no detection over 40% accuracy

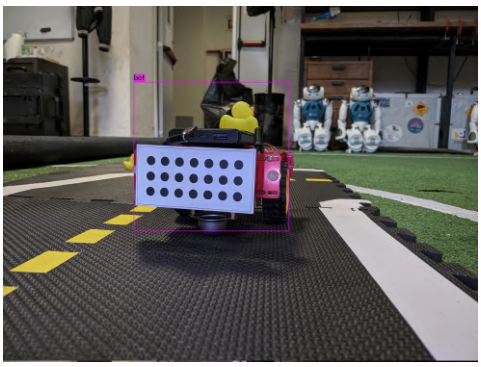
Fig. b
data/foto_labiagi/TestSet/IMG_20200804_131905.jpg: Predicted in 17.992000 milli-seconds.
bot: 58%

Fig. c
data/foto_labiagi/TestSet/IMG_20200804_131952.jpg: Predicted in 3.296000 milli-seconds.
no detection over 40% accuracy

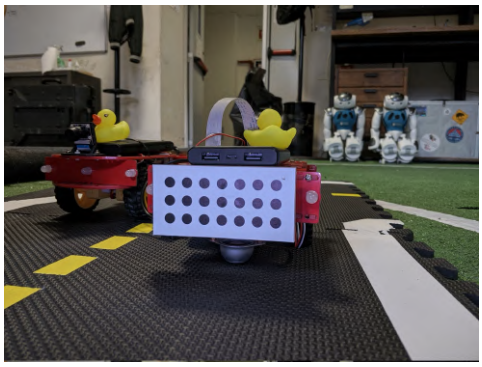
Fig. d
data/foto_labiagi/TestSet/IMG_20200804_131952.jpg: Predicted in 18.033000 milli-seconds.
bot: 57%



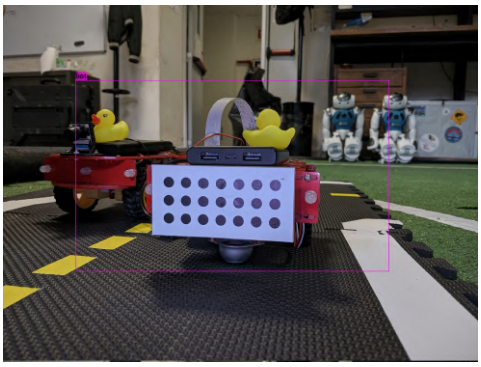
(a) TINY-YOLO



(b) YOLO



(c) TINY-YOLO



(d) YOLO

Figura 4.15. Dettaglio Bot

Fig. 4.16
data/foto_labiagi/TestSet/IMG_20200804_132535.jpg: Predicted in 3.181000 milli-seconds.
road_sign: 43% | duckie: 94% | road_sign: 95%

Fig. 4.17
data/foto_labiagi/TestSet/IMG_20200804_132535.jpg: Predicted in 18.019000 milli-seconds.
stop_sign: 60% | bot: 42% | duckie: 88% | road_sign: 86%

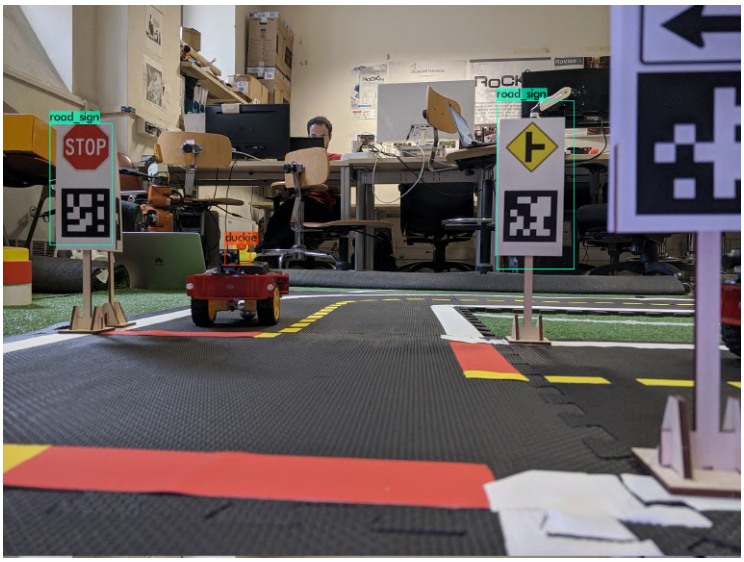


Figura 4.16. TINY-YOLO

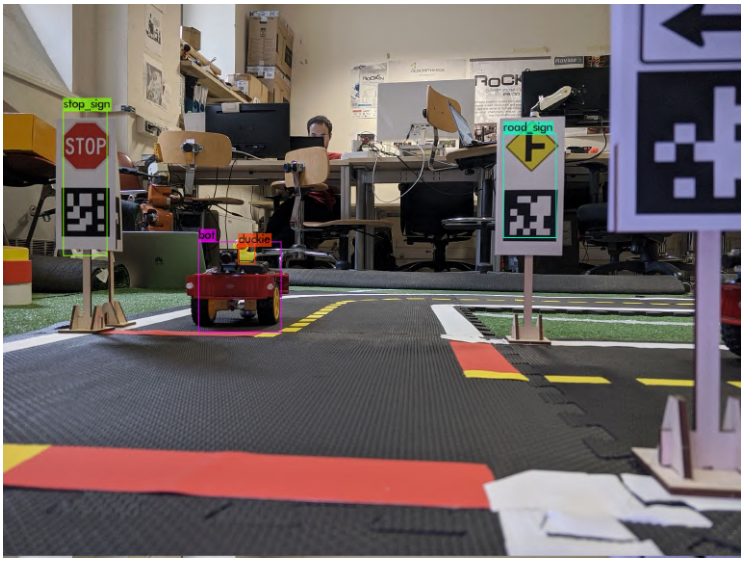


Figura 4.17. YOLO

Fig. 4.18
data/foto_labiagi/TestSet/IMG_20200804_132345.jpg: Predicted in 3.238000 milli-seconds.
duckie: 60% | duckie: 74% | duckie: 95%

Fig. 4.19
data/foto_labiagi/TestSet/IMG_20200804_132345.jpg: Predicted in 18.064000 milli-seconds.
duckie: 52% | duckie: 61% | duckie: 62% | duckie: 55%

Figura 4.18. TINY-YOLO

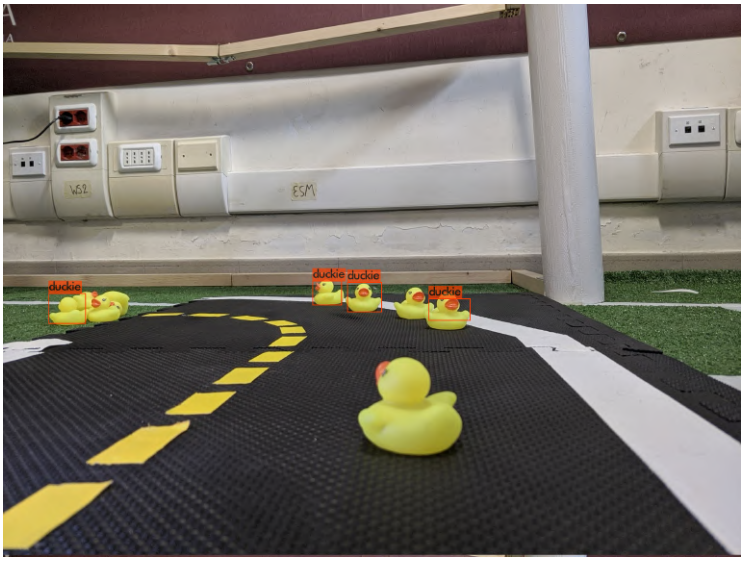


Figura 4.19. YOLO

I risultati degli esperimenti fatti sul Test Set alternativo permettono di trarre delle conclusioni più definite nei confronti delle reti in analisi. Innanzitutto, rimane la costante sui tempi di predizione che però, avendo a disposizione una GPU più potente, si riducono drasticamente; passando da un rapporto di ~8 a meno di 3. Questo permette di concludere che la rete YOLO riesce a sfruttare molto bene la potenza di calcolo laddove disponibile, riducendo di molto i vantaggi offerti dalla sua versione TINY.
Per quanto riguarda le capacità di detection, vediamo come YOLO riesca a riconoscere molti più oggetti rispetto a TINY, seppur con un grado di confidenza comunque basso relegato sempre alla stessa problematica della sotto-sezione precedente; per esempio nella fig. 4.14 i segnali stradali e il bot superano a malapena il 50%. Particolare anche il risultato della fig. 4.15, dove sopratutto per quanto riguarda il caso d, osserviamo una debolezza della rete nell'isolare soggetti sovrapposti; seppur riconoscendoli. Questo tipo di fenomeni risultano comunque piuttosto comuni durante l'utilizzo di YOLO, legati principalmente al modo in cui viene effettuata la detection.
D'altra parte, si ripropone un caso di riconoscimento errato nella fig. 4.16 da parte della rete TINY, anche qui però con un grado di confidenza inferiore al 50%. La rete YOLO invece riesce ad identificare correttamente il segnale di stop ed anche la coppia bot, paperella con i soliti limiti già esaminati. Concludo con un commento nei confronti delle fig. 4.18 e 4.19, dove vengono confermati i valori di maggiore IoU della rete YOLO data la maggiore precisione delle BB.

Capitolo 5

Conclusioni

Avendo commentato i risultati ottenuti da parte delle due reti, all'interno di questo capitolo si andranno ad illustrare i possibili miglioramenti da apportare al lavoro svolto, accompagnati da varie indicazioni finalizzate a continuare quanto elaborato. Infine si riassumeranno i contenuti della relazione, commentando quanto messo in luce dallo studio esposto.

5.1 Miglioramenti

Il problema principale che ha compromesso la possibilità di analizzare in modo più chiaro le due reti è stata decisamente l'impossibilità di allenare la rete YOLO nella sua interezza. Di conseguenza uno dei miglioramenti più sostanziali si ritroverebbe in un allenamento completo della suddetta rete. Una possibile soluzione, riutilizzando lo stesso hardware, sarebbe sicuramente ridurre la dimensione delle immagini passate alla rete in fase di allenamento. Questo ridurrebbe in generale il tempo di allenamento e permetterebbe il completamento di quest'ultimo. Tuttavia, grazie ai risultati ottenuti, è possibile comunque trarre delle conclusioni che, seppur non definitive, indicano sommariamente le debolezze e i vantaggi delle due Reti.
Anche il dataset presenta forti limitazioni, poichè decisamente contenuto rispetto al numero di variazioni delle classi analizzate e presenta in generale un numero di immagini piuttosto basso per questo tipo di applicazioni. Le classi che più hanno subito queste mancanze sono state quelle dei bot e delle paperelle; entrambi classi molto instabili nei confronti della detection da parte di entrambe le Reti. Da considerare infine l'impossibilità di apportare la maggior parte delle tecniche di Data Augmentation, poichè tutte le classi necessitano di presentarsi nelle stesse condizioni di quando sono state fotografate.

Se avessi avuto l'opportunità di continuare questa ricerca, avrei cercato quindi di ampliare il dataset tramite un processo di bagging molto più dettagliato all'interno del laboratorio al fine di colmare le lacune evidenziate dagli esperimenti fatti. Considerati i vantaggi della rete TINY per questo contesto applicativo, si sarebbe potuto anche provare a rivedere gli iperparametri che definiscono la rete per avvicinarla ancora di più alla rete YOLO. Alcuni parametri di interesse sarebbe stati sicuramente il Learning Rate, la sua policy, le variabili dello strato YOLO all'interno della Rete e anche la dimensione delle immagini utilizzate durante la fase di training. Una volta ottenuti dei risultati soddisfacenti, si sarebbe potuto passare al processo di implementazione della rete all'interno del sistema ROS e ridimensionare di conseguenza tutto il processo che ha portato all'ottenimento della rete utilizzata. In

conclusione allego un possibile grafico che illustra teoricamente l'architettura sulla quale avrei basato questo lavoro.

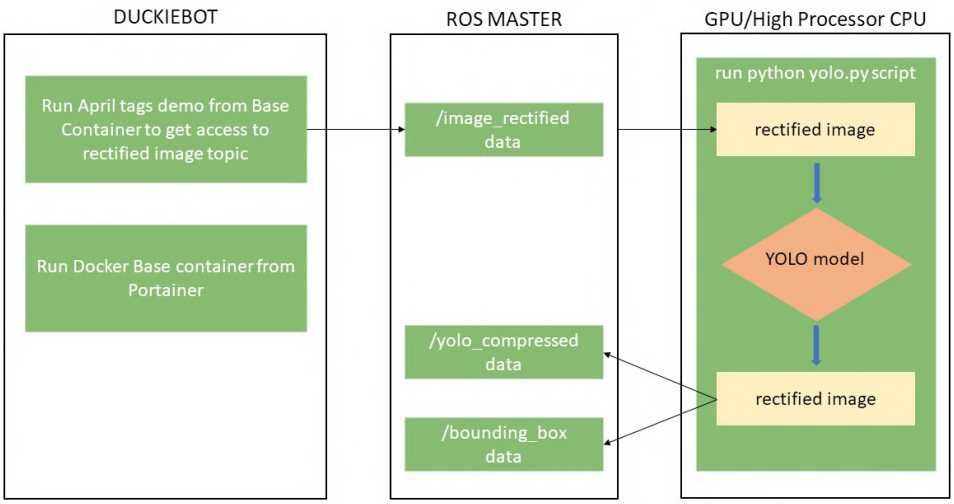


Figura 5.1. Utilizzo di YOLO su Duckiebot

5.2 Considerazioni Finali

Il lavoro presentato ha visto il confronto tra due varianti dello stesso algoritmo di Object Detection, YOLO; in particolare la sua versione standard e la sua versione TINY. Questo confronto è stato sviluppato sfruttando il mondo di Duckietown, dal quale è stato tratto il dataset utilizzato. Lo studio era finalizzato ad identificare il candidato migliore al fine di implementare le reti neurali come supporto del Duckiebot all'interno del sistema ROS, in particolare per quanto riguarda problemi di Object Detection. Questo tipo di lavoro ha quindi come scopo ultimo quello di creare moduli abbastanza prestanti ed affidabili da essere utilizzati in ambienti di guida autonoma.
Lo studio ha messo in luce i vantaggi offerti da parte dalla rete TINY-YOLO, la quale si presenta molto più veloce della sua controparte standard, sopratutto se utilizzata sfruttando hardware dalle potenzialità limitate. Il parametro che più ha marcato la superiorità di questa algoritmo è stato quella del tempo di detection, accompagnato sempre però da risultati superiori alla soglia selezionata per quanto riguarda la precisione nei confronti degli elementi identificati. I risultati della rete standard all'interno di questo studio sono stati fortemente limitati dall'impossibilità di ottenere un allenamento ottimo date le circostanze e le disponibilità hardware. Laddove si avesse avuto a disposizione una rete ben allenata, sicuramente i risultati offerti da quest'ultima avrebbe più che giustificato la sua maggiore complessità.
Ritengo quindi che per il contesto applicativo descritto all'interno di questa relazione, la rete TINY risulti migliore per semplicità e velocità. Laddove invece si avesse più tempo a disposizione e si volesse tentare di provare ad implementare la variante standard all'interno del Duckiebot, si raggiungerebbero sicuramente dei risultati prossimi agli standard necessari per traslare questo tipo di lavoro nel mondo reale.

Concludendo, questo progetto mi ha permesso di comprendere le complicazioni della computer vision applicate al mondo della guida autonoma, e quindi realizzare quanto lavoro ci possa essere dietro queste sfide. Un ambiente controllato e ben strutturato come Duckietown presenta comunque moltissimi imprevisti che, traslati nel mondo reale, non fanno altro che aumentare data la quasi impossibilità di replicare un ambiente chiuso e controllato.
C'è bisogno di creare sistemi robusti e provare a prevedere il maggior numero di variazioni possibili per poter ritenere questi sistemi affidabili in situazioni di necessità. Sarei curioso di provare a studiare il corrispettivo del lavoro fatto su un reale veicolo, andando a scoprire le tecniche, il software e l'ottimizzazione utilizzati per creare moduli capaci di far parte dei dispositivi che accompagnano la nostra quotidianità.

Bibliografia

[1] Appearance Specifications, https://docs.duckietown.org/DT17/opmanual_duckietown/out/dt_ops_appearance_specifications.html

[2] Darknet Fork on Duckietown, https://github.com/marquezo/darknet

[3] Duckiebot operation manual, https://docs.duckietown.org/daffy/opmanual_duckiebot/out/index.html

[4] How to train (to detect your custom objects) https://github.com/AlexeyAB/darknet#how-to-train-to-detect-your-custom-objects

[5] Real-time Object Detection with YOLO, YOLOv2 and now YOLOv3, https://medium.com/@jonathan_hui/real-time-object-detection-with-yolo-yolov2-28b1b93e2088

[6] SAE Standards News: J3016 automated-driving graphic update, https://www.sae.org/news/2019/01/sae-updates-j3016-automated-driving-graphic

[7] The Duckietown Platform, https://www.duckietown.org/about/platform

[8] What is Docker, https://www.redhat.com/it/topics/containers/what-is-docker

[9] YOLO Algorithm and YOLO Object Detection: An Introduction, https://appsilon.com/object-detection-yolo-algorithm/

[10] YOLO-TINY-v3 cfg, https://github.com/pjreddie/darknet/blob/master/cfg/yolov3-tiny.cfg

[11] YOLOv3 cfg, https://github.com/pjreddie/darknet/blob/master/cfg/yolov3.cfg

[12] YOLOv3 Paper, https://pjreddie.com/media/files/papers/YOLOv3.pdf