

# Implementazione sistema di rilevazione di Automobili con Sistema Embedded per la Park Detection

Francesco Pegoraro - Daniele Giacomelli  
Università degli studi di Firenze

## Introduzione

Questo lavoro viene svolto per la risoluzione del problema riguardante il rilevamento di veicoli posti in parcheggi o in particolari punti stradali (es. strisce pedonali) permettendo di capire quando i veicoli siano arrivati e partiti dai punti di interesse. Il programma deve essere in grado di funzionare su un dispositivo embedded (Raspberry Pi) munito di una videocamera che viene posto su un punto di osservazione rivolto verso la strada e deve poter monitorare le varie situazioni dando degli opportuni report sui cambiamenti di stato. Questo deve rimanere in funzione per tempi lunghi permettendo un frame rate sufficiente per lo scopo prefissato, inoltre deve essere in grado di evitare maggiormente i "falsi positivi" (es. auto in passaggio o altri elementi che possono interporsi nella visuale) e permettere una corretta elaborazione in qualsiasi situazione ambientale (es. di notte come di giorno).

- Implementazione sistema di rilevamento dei cambi di stato su computer tramite video di test
- Implementazione su Raspberry con test 'real-time'

Inizialmente la scelta del *car detector* è ricaduta su *Haar Cascade*, un classificatore usato spesso per fare 'face detection' che permette, con un numero esiguo di positivi e negativi per l'addestramento, di ottenere buoni risultati nel riconoscimento di oggetti. Successivamente però si è optato per *Yolo*, un sistema di Object detection innovativo basato su rete neurale poiché i risultati si sono rivelati più soddisfacenti al netto di un maggiore carico computazionale. Viene inoltre usata la libreria di Computer Vision *OpenCV* su linguaggio *Python* per la gestione e l'elaborazione di immagini e video. Si esaminano di seguito le tecniche usate nello sviluppo per poi ripercorrere i dettagli dell'implementazione del sistema di Park Detection.

## 1 Haar Cascade

Il classificatore *Haar cascade* per l'*Object detection* è una tecnica sviluppata da Paul Viola e Michael Jones basata su feature e machine learning dove una funzione *cascade* viene addestrata con raccolte di immagine positive e negative scalate a dimensione comune. Una volta raccolte le immagini per

## Strategia seguita

La strategia di risoluzione per soddisfare le richieste può essere riassunta nei seguenti passi:

- Ricerca di un Detector di Oggetti, in particolare un '*car-detector*'

il training è necessario estrarre le *features* da queste; ogni feature è un valore ottenuto sovrapponendo diversi tipi di filtri (Fig: 1) e calcolando la sottrazione tra la somma dei pixel dell'immagine che ricadono nel rettangolo bianco e la somma dei pixel sotto il rettangolo nero [4].

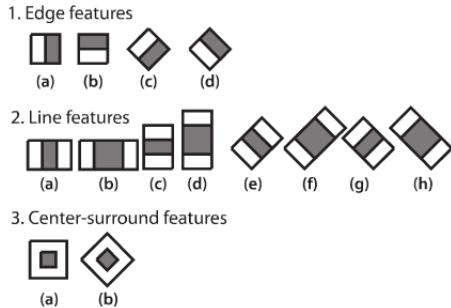


Figura 1: Features di Haar Cascade

Questo approccio di somme e differenze può essere molto dispendioso soprattutto perché vengono usate delle finestre relativamente piccole per cercare le features nelle immagini (es. 24x24). Per ridurne il costo computazionale viene usata la tecnica delle *immagini integrali*. Queste sono costruite con la tabella "Summed Area Table" nella quale l'elemento  $I_{\Sigma}(x)$  in posizione  $x = (x; y)^T$  rappresenta la somma di tutte le intensità dei pixel nell'immagine di input  $I$  all'interno della regione rettangolare formata dall'origine e  $x$ .

$$I_{\Sigma}(x) = \sum_{i=0}^{i \leq x} \sum_{j=0}^{j \leq y} I(i, j)$$

Una volta che l'immagine integrale è stata creata, sono necessarie solo tre addizioni per calcolare la somma delle intensità su qualunque area rettangolare. Quindi, il tempo di calcolo è indipendente dalle

dimensioni dell'area. Questo è molto importante poiché, vengono utilizzati filtri di grandi dimensioni [1].

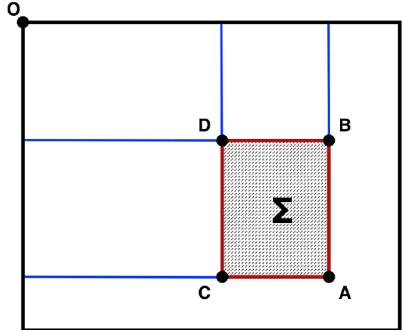


Figura 2:  $\Sigma = A - B - C + D$

Le features Haar Cascade sono applicate su tutte le immagini del training e per ognuna di esse viene trovata la migliore *threshold* che classifica le immagini come positive e negative. Ovviamente ci saranno degli errori di classificazione; vengono dunque selezionate le features con il minimo errore che sono quelle che classificano meglio le immagini che contengono o meno l'oggetto ricercato. Per fare ciò inizialmente viene assegnato uno stesso peso a ogni immagine e dopo ogni classificazione viene aumentato il peso delle immagini classificate erroneamente, questo processo è fatto ricorsivamente fino a che si raggiunge l'accuratezza richiesta o vengono trovate un numero di feature accettabile. Il classificatore finale quindi è una somma pesata di tutti i classificatori "deboli" (chiamati così poiché da soli non sarebbero in grado di classificare in maniera corretta). Per la ricerca dell'oggetto vengono utilizzate delle finestre di dimensione SxS (es. S=24), ma in un'immagine, la maggior parte dello spazio non è occupato dall'oggetto ricercato. È dunque preferi-

bile avere un metodo per verificare se una finestra è una regione dell’oggetto, se non lo è, essa viene scartata, altrimenti si concentra la computazione su di essa. A questo proposito viene introdotto il concetto di *”Cascade di classificatori”*: invece di applicare tutte le features su ogni finestra SxS, si raggruppano le features in stage differenti di classificazione che vengono applicati incrementalmente. Se una finestra fallisce al primo stage essa viene scartata; se la finestra passa, viene applicato il secondo stage di features e il processo continua. Una finestra che passa tutti gli stage è una regione dell’oggetto ricercato. Il classificatore inoltre è strutturato in modo che possa essere facilmente scalato così da riuscire a trovare oggetti di dimensioni differenti (più efficiente che ridimensionare l’immagine stessa), quindi per trovare un oggetto di dimensione sconosciuta la procedura dovrebbe essere fatta più volte con scale differenti [2].

## 1.1 Implementazione

Per la fase di *training* sono necessarie immagini di positivi (immagini contenenti automobili) e negativi (immagini di sfondi che non contengono automobili) che sono state reperite dal *UIUC Image Database for Car Detection*<sup>1</sup>. Le immagini possono essere di dimensione variabile; è comunque buona norma che queste siano di dimensioni maggiori rispetto alle finestre usate per il training. I campioni positivi sono generati dall’applicazione *opencvcreatesamples*, che a partire da una sola immagine positiva, riesce a creare artificialmente fino a

<sup>1</sup>UIUC Image Database for Car Detection, disponibile sul sito '<https://cogcomp.cs.illinois.edu/Data/Car/>'

100 campioni di diverse dimensioni e tagli. L’immagine viene ruotata randomicamente attorno ai tre assi (gli angoli di rotazione possono essere fissati) e successivamente posta su un’immagine scelta arbitrariamente tra quelle di background e infine ridimensionata al size desiderato. I comandi necessari per la creazione dei sample sono:

- -info < collection\_file\_name >
- -bg < background\_file\_name >
- -vec < vec\_file\_name >
- -num < number\_of\_samples >

dove *num* è il numero di campioni positivi che si vogliono creare e *vec* è il file ‘.vec’ che serve per addestrare il cascade contenente i sample positivi. Nel file *info* ogni linea corrisponde a un’immagine, dove il primo elemento è il nome del file, seguito dal numero di oggetti presenti nell’immagine e le coordinate di essi (*x, y, larghezza, altezza*); il file “info.lst” è strutturato come:

```
img/img1.jpg 1 140 100 45 45
img/img2.jpg 1 100 200 50 50
...
```

Una volta creati i samples si può procedere al training del cascade: questo viene fatto con il comando *opencv\_traincascade* e le seguenti *flag*:

- -data < cascade\_dir\_name >
- -vec < vec\_file\_name >
- -bg < background\_file\_name >
- -numPos < number\_of\_positive >
- -numNeg < number\_of\_negative >
- -numStages < number\_of\_stages >

‘-data’ rappresenta la directory di destinazione del classificatore, sono esplicitati inoltre i numeri di samples positivi e negativi e il numero di *cascade stages* che devono essere addestrati. Il processo di addestramento

può richiedere diverse ore e il risultato è un file ‘.xml’ che rappresenta il classificatore;

## 1.2 Test del classifier

Il classificatore viene testato su un file video di demo.

L’utente disegna dei *rettangoli target* che devono sovrapporsi ai relativi posti auto (uno per ogni parcheggio), quindi viene eseguito lo script che fa la detection solo nelle porzioni di immagine racchiusa dai target attraverso la funzione *detectMultiScale()*: partendo dalla dimensione delle finestre usate nel training cerca l’oggetto *auto* scalando la grandezza del blocco per la ricerca (elementi più piccoli di SxS non possono essere rilevati) [3].



Figura 3: Sistema in funzione su un solo parcheggio

Per valutare se l’auto è nel parcheggio si valuta la lunghezza del vettore che contiene i rilevamenti del classificatore, se questo è  $> 0$  allora è presente un veicolo, altrimenti no. Viene usato anche un contatore per gestire un tempo di stallo per cui se una macchina si ferma per poco tempo non viene rilevata come parcheggiata. Nonostante

la velocità del sistema sia ottima permettendo un elevato numero di frame per secondo e anche i risultati piuttosto soddisfacenti, vengono trovati troppi falsi negativi e bounding box lontani dall’oggetto, causando numerosi cambi di stato del parcheggio anche se l’automobile non si è mossa.

## 2 YOLO

You only look once (**YOLO**) è un object detection system real-time sviluppato nel 2016 che su un calcolatore dotato di scheda grafica Titan X riesce a processare 40-90 immagini al secondo, ottenendo un mAP<sup>2</sup> del 78.6% alla challenge VOC del 2007.

Per lo scopo di questo progetto si esaminano le specifiche della prima versione di Yolo (Yolo v1) nonostante siano già presenti approcci migliorati (Yolov2, YOLO9000..), per poi applicare la modalità Tiny Yolo costruita appositamente per dispositivi embedded con ridotte capacità Hardware come Raspberry Pi [5].

### 2.1 Approccio Innovativo

I detection System usuali utilizzano la classificazione e localizzazione per fare detection. Più precisamente fanno scorrere una sliding window sull’immagine e ad ogni passaggio usano un classificatore per avere delle predizioni sugli oggetti presenti in quella porzione di immagine. Questo approccio restituisce migliaia di predizioni e le regioni che ottengono un punteggio elevato vengono considerate detection.

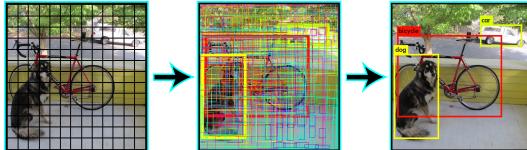
---

<sup>2</sup>mAP è la media delle Average Precision (TP/TP+FP) su tutte le immagini del dataset.

Ovviamente questo metodo funziona, ma non è molto efficiente perchè si deve applicare il classificatore un gran numero di volte. Un approccio più sofisticato potrebbe essere la tecnica detta di *region proposal*, in cui si cerca di predire prima quali parti dell'immagine contengono informazioni per poi usare il classificatore solo su di esse.

Ma queste routine sono complesse e difficili da ottimizzare poichè ogni componente individuale deve essere addestrato separatamente. Gli sviluppatori di YOLO usano un approccio completamente differente, applicando una singola rete neurale all'intera immagine.

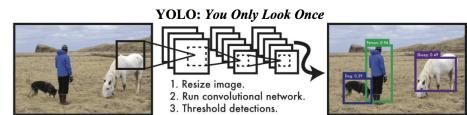
La Neural Network seziona l'immagine in regioni e predice bounding boxes e probabilità per ognuna di esse. Questi bounding box vengono pesati dalle probabilità predette.



Con questo approccio si ottengono risultati migliori rispetto ai sistemi basati su classificatore, infatti, si tiene conto dell'intera immagine e le predizioni sono informate dal contesto globale di essa. Inoltre si sfrutta una sola valutazione da parte della rete neurale al contrario di sistemi come R-CNN che ne richiede migliaia. Questo rende l'approccio estremamente veloce, quasi 1000 volte più di R-CNN e 100 più di Fast R-CNN.

## 2.2 Unified Detection

Questo sistema divide l'immagine di input con una griglia di SxS celle. Se il centro di un oggetto ricade in una delle celle della griglia, quella cella è responsabile della detection di quell'oggetto. Ogni cella predice B bounding boxes e i relativi score di confidenze per quei box. Questi score riflettono quanta confidenza ripone il modello sul fatto che quel box contenga un oggetto e anche quanto è accurato il box predetto.



Formalmente definiamo la confidenza come:  $Pr(\text{Object}) * \text{IOU}$ .

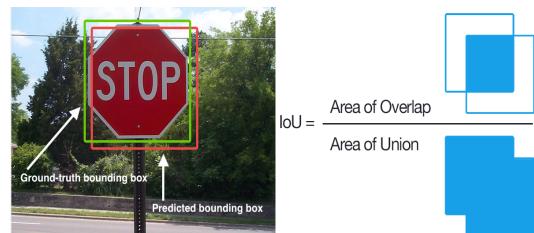


Figura 4: IOU è un'evaluation metric usata per misurare l'accuratezza di un object detector su un particolare dataset.

Se non ci sono oggetti in quella cella, la confidenza dovrebbe essere pari a zero, altrimenti vorremmo che fosse uguale alla IOU tra il box predetto e la ground truth. Ogni bounding box consiste di 5 predizioni:  $x, y, w, h$  più la confidenza. La coppia  $(x, y)$  rappresenta il centro del box relativamente ai bordi della cella, mentre l'altezza e la larghezza sono relativi all'intera immagine. Infine la predizione sulla confidenza

rappresenta l'IOU tra il box predetto e il box ground truth. Ogni cella predice inoltre C probabilità condizionali sulle classi:  $Pr(Class_i|Object)$ .

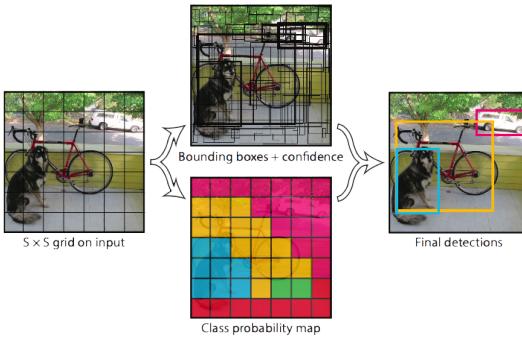


Figura 5: Il sistema modella le detection come un problema di regressione. Le predizioni sono codificate in un tensore di dimensioni  $S \times S \times (B * 5 + C)$

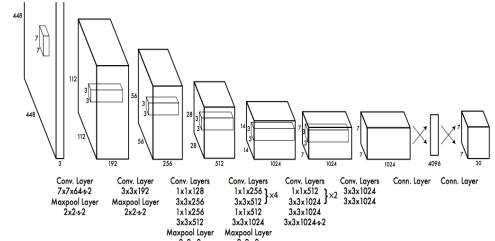
Quindi vengono moltiplicate le probabilità e le confidenze dei box individuali:

$$\begin{aligned} Pr(Class_i|Object) * Pr(Object) * IOU_{pred}^{truth} &= \\ &= Pr(Class_i) * IOU_{pred}^{truth} \end{aligned}$$

e questo ci dà le confidenze di ogni classe all'interno di ogni box. Questi valori codificano sia la probabilità che quella classe appaia nel box, sia quanto bene il box predetto racchiude l'oggetto. Per valutare YOLO su PASCAL VOC, sono usati  $S=7$ ,  $B=2$ . PASCAL VOC ha 20 classi, quindi  $C=20$ . La predizione finale è un tensore  $7 \times 7 \times 30$ .

### 2.3 Network Design

La rete è costituita da 24 layer convolutivi che estraggono le features dall'immagine e 2 layer completamente connessi che predicono le probabilità e le coordinate.



E l'output della rete è il tensore  $7 \times 7 \times 30$  delle predizioni.

### 2.4 Tiny YOLO

Le prestazioni della versione proposta di YOLO sono ottime ma sono comunque troppo dispendiose per dispositivi come il Raspberry Pi, quindi la scelta è ricaduta sulla versione Tiny YOLO. Questa è molto più veloce ma ovviamente meno accurata. Il training di questo modello è stato effettuato sul dataset PASCAL VOC, che può rilevare 20 classi, tra cui quella di interesse per questo progetto, le automobili:

- bicycle
- boat
- car
- cat
- ...

La versione usata ha solo 9 layer convolutivi e 6 pooling layer. Mentre la versione più attuale YOLO v2 ne ha il triplo. Questa, sempre su calcolatore con Titan X riesce ad elaborare circa 200 frame al secondo, ovviamente riducendo l'accuracy.

## 3 Sistema implementato

Come già detto in precedenza lo strumento utilizzato per la Park Detection in questo elaborato è *Yolo* che viene caricato su

un Raspberry PI 2 dotato di una PI camera. Il sistema viene poi fissato tramite un supporto fisso a una finestra e rivolto verso una particolare zona di interesse. Inizialmente per testare la bontà del lavoro, prima di essere caricato sul Raspberry, sono state eseguite, da computer, delle prove su dei video registrati da una telecamera che presentavano le stesse caratteristiche delle scene che sarebbero poi state monitorate dal sistema finale. Una volta visto che i risultati erano soddisfacenti si è passati all'implementazione sul Raspberry.



Figura 6: Raspberry posto su una finestra per monitorare i parcheggi paralleli alla strada

### 3.1 Funzionamento

La prima cosa da fare è andare a selezionare i parcheggi (o le aree di interesse) presenti nello scenario monitorato. Per fare ciò è sufficiente eseguire su Raspberry lo script Python '*waitCoordRASP.py*', questo scatta una fotografia della scena e tramite una connessione TCP/IP invia l'immagine al computer.

Da PC si esegue lo script '*selectCoordPC.py*' che riceve l'immagine scattata dal Raspberry, e la mostra a schermo. Su di

essa si possono tracciare le aree di interesse che si vogliono monitorare.



Figura 7: L'utente seleziona un numero arbitrario di *target* da computer

Banalmente, si disegnano dei rettangoli sopra ai parcheggi singoli, cliccando con tasto sinistro per indicare l'angolo alto sinistro del rettangolo e rilasciando sul vertice basso destro. Quindi si conferma premendo il tasto 'c', o 'r' per cancellare. Lo script ricava le coordinate dei rettangoli disegnati e li trasmette sempre tramite la connessione instaurata, al Raspberry. Una volta ricevute le coordinate, il dispositivo le salva in un file di testo con nome 'coordinates.txt'.

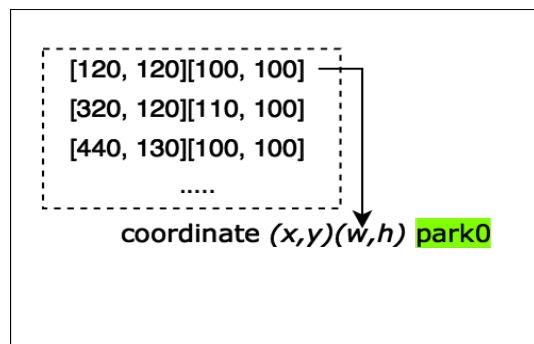


Figura 8: Coordinates.txt, rappresenta le coordinate dei rettangoli di *target*.

Dopo aver selezionato i punti di interesse non resta che monitorare la scena. Per questo si esegue lo script '*yoloRaspArray.py*' sul Raspberry, che dapprima inizializza i pesi e le funzionalità di Yolo con l'utilizzo di Keras per il Tensorflow backend, poi legge il file delle coordinate dei parcheggi, che vengono inserite in un array ('*position*'). Viene inizializzato l'array '*variables*', che verrà usato per valutare se la macchina sia presente o meno nel parcheggio con l'utilizzo di due variabili booleane e un contatore, infine viene creato l'array '*target*' che contiene le dimensioni dei rettangoli corrispondenti ai parcheggi. L'utilizzo di questi array multidimensionali è necessario per gestire un numero arbitrario di target disegnato dall'utente. Dopo questa fase di inizializzazione si incomincia a ricevere le immagini, frame per frame.



Figura 9: Visuale del Raspberry sui parcheggi.

Le immagini registrate dalla Pi Camera sono trasformate in formato PIL e ridimensionate, poichè la rete neurale di Yolo necessita immagini di dimensioni multiple di 32. Yolo dopo aver elaborato l'immagine risponde con i *bounding box*, l'*accuratezza* e il *nome della classe*.

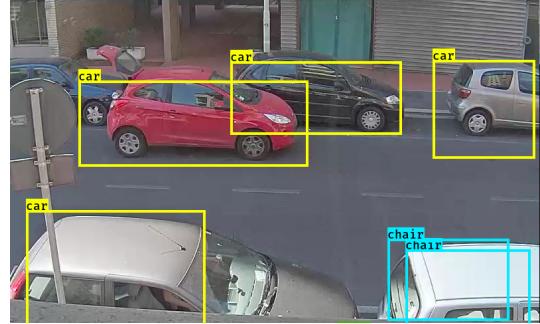


Figura 10: Risultato fornito da Yolo

Per ognuno dei *bounding box* si va a controllare che questi siano interni al target creato per il parcheggio (funzione '*testRectIn()*'): in pratica si fa un check sull'intersezione tra il target e il box, se l'area dell'intersezione è almeno 1/2 dell'area di target e se l'area del box non è più grande di due volte l'area del target, allora l'auto viene considerata all'interno del parcheggio. Quindi se il box corrisponde a una macchina, si va a inserire il dato nell'array '*carFound*', altrimenti viene scartato. Se nella posizione relativa a un certo target non sono state rilevate automobili, viene settato *False* all'indice corrispondente dell'array '*variables*' altrimenti viene messo a *True*.



Figura 11: Logica della funzione *TestRectIn()*

Per valutare se un'auto sia presente o meno nel parcheggio viene utilizzata questa logica, che si appoggia sulle tre variabili (2 booleane e una numerica) del vettore 'variables'. Ogni volta che viene rilevato lo stato di un target, questo viene confrontato con lo stato precedente, se non corrispondono, viene incrementato il contatore relativo; se questo avviene per 5 volte consecutive allora c'è stato un vero cambio di stato e quindi la macchina è appena uscita o arrivata a seconda del valore della variabile booleana carFound. A questo livello di esecuzione viene quindi salvata l'immagine con il parcheggio occupato se l'auto è arrivata o libero se questa se n'è andata. In pratica la logica che sta alla base è quella di valutare un cambio di stato, se questo è permanente per un certo quantitativo di frame allora va reso effettivo. Il contatore è utile per valutare questa permanenza, infatti nel caso ci sia una macchina in fermata e non in sosta, oppure nel caso di passaggio di veicoli che occludono la visuale del parcheggio il sistema non notifica un cambiamento (il contatore è azzerato ogni volta in cui non si verificano cambi di stato per evitare che dopo 5 rilevazioni di cambi anche a distanza temporale elevata venga notificato un nuovo stato erroneo), questo permette anche una buona resistenza nel caso di falsi positivi che sono più rari in 5 frame consecutivi. Vengono poi disegnati i rettangoli sia dei target che dei bounding box sull'immagine con la funzione di OpenCV `'cv2.rectangle()'`

## Conclusione

Per adempiere alla richiesta di creare un sistema embedded in grado di monitorare



Figura 12: Falso negativo dato da un auto in sosta in doppia fila

parcheggi o zone della strada in modo da individuare se queste siano o meno occupate da veicoli sono stati usati degli script in linguaggio Python che utilizzano il sistema Yolo per la detection, e un Raspberry Pi dotato di PiCamera per la registrazione e l'esecuzione.

Nonostante una prima implementazione prevedeva l'utilizzo di un classificatore Haar-Cascade, portando come fattore positivo la velocità di elaborazione, questo presentava un quantitativo di errori dovuti alla rilevazione di falsi negativi non accettabile. Si è optato per Yolo invece perchè si basa su una rete neurale (attuale e di elevato

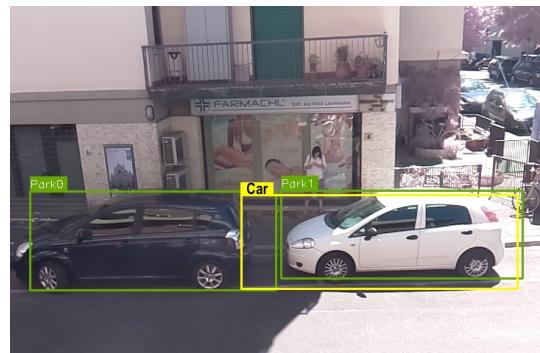


Figura 13: Falso negativo dato dall'ombra

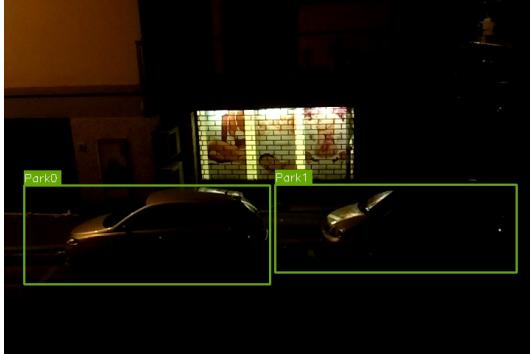


Figura 14: Falso negativo dato da scarsa luminosità nelle ore notturne

interesse), ha un’ampia categoria di oggetti individuabili e presenta un basso numero di falsi positivi. Quindi, nonostante si sia aggiunto un costo computativo di elaborazione, c’è stato un guadagno per quanto riguarda la robustezza e l’affidabilità del sistema.

Dai test effettuati è possibile valutare un eccellente funzionamento del sistema nelle ore diurne; infatti c’è un tasso di errore del 15% di cui il 9% è dovuto a falsi negativi generati da ombre e riflessi (Fig.13) e il resto è dato da auto parcheggiate in doppia fila (Fig. 12) o dal fatto che i target dei parcheggi non siano allineati perfettamente con le linee dei parcheggi.

Un discorso particolare invece lo meritano le rilevazioni notturne: poiché i test sono stati effettuati in una via poco illuminata, si riscontrano degli errori (il 40% dei report notturni) dati dall’effettiva impossibilità di identificare le automobili(Fig.14). Questo potrebbe essere risolto attraverso l’uso di una visione a infrarossi.

La velocità di elaborazione del video è di un frame ogni 4-5 secondi che per l’obiettivo preposto è accettabile, infatti essendo in

tempo reale e monitorando i parcheggi non è necessario un frame rate elevato.

## Riferimenti bibliografici

- [1] L. Angioloni. “Tecniche di riconoscimento e loro applicazione al controllo di sistemi mobili mediante piattaforme embedded”. Tesi di Laurea Triennale in Ingegneria Informatica. Università degli Studi di Firenze, 2016.
- [2] M Oliveira e V Santos. “Automatic detection of cars in real roads using haar-like features”. In: *8th Portuguese Conference on Automatic Control* (2008).
- [3] OpenCv. *Cascade Classifier Training*. [http://docs.opencv.org/trunk/dc/d88/tutorial\\_traincascade.html](http://docs.opencv.org/trunk/dc/d88/tutorial_traincascade.html).
- [4] OpenCv. *Face Detection using Haar Cascades*. [http://docs.opencv.org/trunk/d7/d8b/tutorial\\_py\\_face\\_detection.html](http://docs.opencv.org/trunk/d7/d8b/tutorial_py_face_detection.html).
- [5] Joseph Redmon. *Darknet: Open Source Neural Networks in C*, 2013–2016. <http://pjreddie.com/darknet/>.