UNIVERSITÀ
DEGLI STUDI
DI PADOVA


UNIVERSITÀ DEGLI STUDI DI PADOVA

**Dipartimento di Ingegneria Industriale DII**

Corso di Laurea Magistrale in Ingegneria Aerospaziale


---

# Sviluppo e test di un algoritmo per la navigazione autonoma di un rover basato su segmentazione semantica e SLAM

---


**Relatore**
Prof. Stefano Debei

**Correlatore**
Ing. Sebastiano Chiodini

**Laureando**
Giulio Polato 1234051



Anno Accademico 2021/2022

**Sommario**

La navigazione autonoma, in particolare nell'ambito dell'esplorazione planetaria, sta diventando sempre più un argomento di grande interesse. Un aspetto caratteristico della navigazione autonoma è l'analisi di attraversabilità di un terreno. In particolare, in questa tesi, si vuole proporre un algoritmo di SLAM (Simultaneus Localization And Mapping) che utilizza la segmentazione semantica a supporto della mappatura. Infatti, tramite una CNN (Convolutional Neural Network), è stato possibile mappare quale porzione di un terreno sia più adatta ad essere percorsa. La CNN è stata ottenuta allenando la rete neurale DeepLabv3+ e utilizzando come *dataset* le immagini dello stesso luogo in cui poi è stata realizzata la fase di *testing*. Esse sono state etichettate manualmente e la CNN è stata così allenata a distinguere il terreno asfaltato. Successivamente l'algoritmo è stato implementato per funzionare all'interno di ROS (Robot Operating System) con l'utilizzo del prototipo di rover, appartenente al progetto MORPHEUS. Infine, la mappatura realizzata è stata confrontata con le immagini satellitari e i risultati hanno dimostrato l'efficacia dell'algoritmo studiato, per realizzare l'analisi di attraversabilità di un terreno.

# Indice

# Elenco delle figure

# Elenco delle tabelle

# Capitolo 1

# Introduzione

Durante gli ultimi anni, l'interesse per la navigazione autonoma è sempre maggiore sia nelle applicazioni terrestri che spaziali, in particolare per quel che riguarda l'esplorazione planetaria e la navigazione di un rover.

In quest'ultimo caso il problema maggiore è l'enorme ritardo delle comunicazioni, dovuto alla distanza tra la Terra e il pianeta d'interesse e alla limitata finestra temporale in cui la comunicazione può avvenire [8].

Ad oggi, buona parte delle missioni di esplorazione planetaria per mezzo di rover sono state fatte proprio su Marte. La prima presenza robotica sul pianeta rosso avvenne venticinque anni fa con il rover **Sojourner** (4 luglio - 27 settembre 1997) della missione Mars Pathfinder.

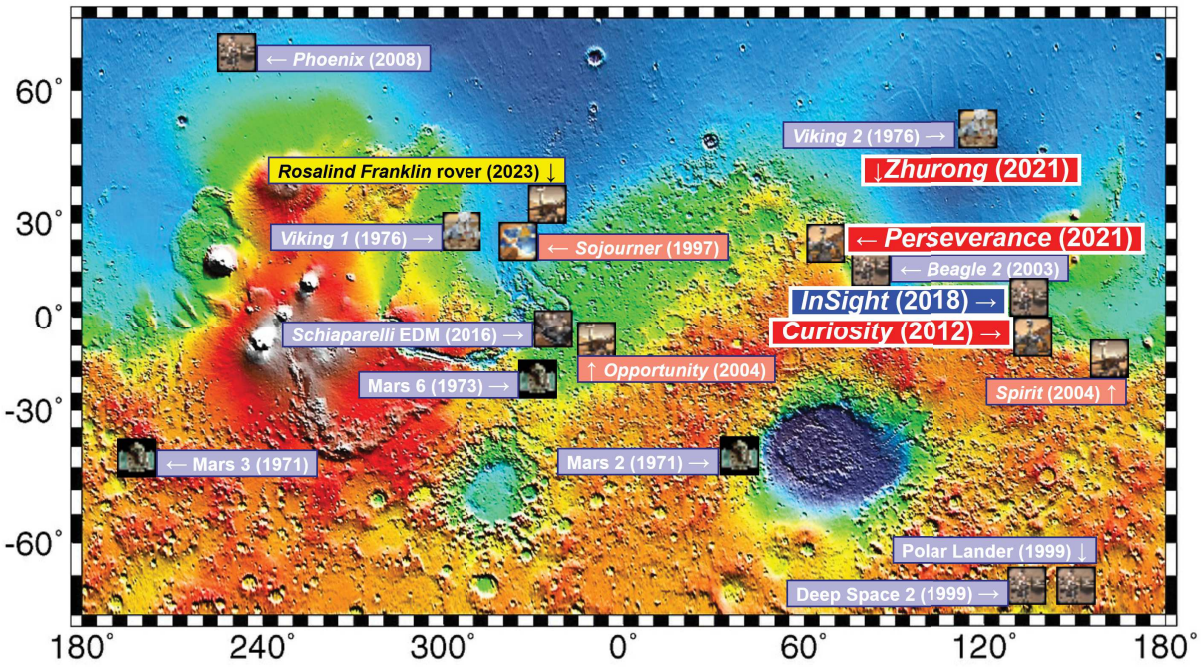



Figura 1.1: Siti di atteraggio dei rover e dei lander su Marte [1]

Seguirono altre missioni che portarono rover sempre più evoluti e sempre più autonomi come: **Spirit** (2004-2010) e **Opportunity** (2004-2019) all'interno della missione MER (Mars Exploration Rover), **Curiosity** (2012-in corso) all'interno della missione MSL (Mars Science Laboratory), **Perseverance** (2020-in corso) per la missione Mars 2020 e infine **Zhurong** (2021-in corso) della missione Tianwen-1 [9][1].

A settembre del 2022 era previsto il lancio del rover **Rosalind Franklin** facente parte del programma ExoMars, ma a seguito del conflitto in Ucraina la cooperazione tra ESA e Roscosmos è stata sospesa, posticipando così la data di lancio del rover [10].

Tutte queste missioni hanno affidato sempre più autonomia ai rover, tuttavia nella maggior parte di esse l'analisi dei rischi e la pianificazione degli obiettivi è stata demandata agli operatori di Terra, allungando così di molto i tempi per il raggiungimento di un determinato obiettivo.

La NASA definisce l'autonomia come "l'abilità di un sistema di raggiungere un obiettivo, operando in modo indipendente da un controllo esterno" [11]. Questo è quanto s'intende realizzare nelle future missioni, in particolare se vengono a verificarsi tre condizioni:[9]

- **Cambiamenti ambientali o nello spacecraft:** Esempi di cambi ambientali possono essere le eruzioni su Encelado o il vento di Marziano; cambiamenti nello spacecraft possono includere degradazione, danneggiamenti o rotture di componenti.
- **Cambiamenti imprevedibili:** Come ad esempio fenomeni non previsti da un modello fisico precedente, oppure difficili da modellare (come l'approccio a corpi di piccole dimensioni, dove i parametri devono essere aggiornati in tempo reale).
- **Tempi rapidi di risposta:** Questa condizione avviene quando lo spacecraft deve rispondere ad un evento, senza dover aspettare istruzioni da Terra. Ciò accadde ad esempio durante la fase di *touch-and-go* del modulo OSIRIS-REx oppure potrebbe essere la situazione di un rover che scivola lungo una superficie inclinata.

Per quanto riguarda la navigazione autonoma di un rover, si vengono a delineare quattro principali campi d'azione, come illustrati in Figura 1.2: *path planning, obstacle avoidance, localization, terrain traversability analysis* [2].

**PATH PLANNING** Consiste nella generazione di un percorso geometrico da un punto di partenza fino ad un punto di arrivo e la successiva generazione di una traiettoria, ovvero una legge oraria che ne definisce il moto lungo il percorso [12].

Inoltre esso viene generalmente suddiviso in due categorie: *global path planning e local path planning*. Il primo consiste nel costruire una traiettoria globale, ottimizzata sulla base di informazioni note a priori. Il secondo, invece, costruisce una traiettoria che segue il più possibile quella globale, ma tiene conto anche di nuove informazioni derivanti da un ambiente sconosciuto e dinamico, tramite l'utilizzo di appositi sensori. [2]

**OBSTACLE AVOIDANCE** Consiste nell'aggirare un ostacolo o un pericolo che impedisce lo svolgimento delle operazioni del robot. Tramite un sistema di percezione, composto da una stereo camera e/o da un LiDAR, il robot rivela la presenza e la posizione dell'ostacolo, permettendo così di elaborare una strategia per aggirarlo.

Nonostante esistano delle soluzioni per evitare ostacoli statici, in ambienti dove sono presenti un gran numero di ostacoli oppure dove questi sono dinamici, il problema si complica maggiormente [2].

**LOCALIZATION** Consiste nel localizzare il robot all'interno dell'ambiente circostante. Esistono diverse tecniche per questo tipo di operazione, le principali sono *wheel odometry, GPS (Global Positioning System), LiDAR odometry, VO (visual odometry) e SLAM (Simultaneous Localization And Mapping).*

Tutte queste tecniche presentano differenti vantaggi e svantaggi, quindi solitamente si preferisce utilizzarle insieme oppure in modo alternato [13].

**TERRAIN TRAVERSABILITY ANALYSIS** Per terreni sconosciuti e difficili da attraversare è importante che essi vengano analizzati, per poter valutare se il robot è in grado di percorrerli. Per esempio sul suolo marziano, il robot incontra rocce di natura e dimensioni diverse, quindi deve saper distinguerle per evitare eventuali pericoli[2].

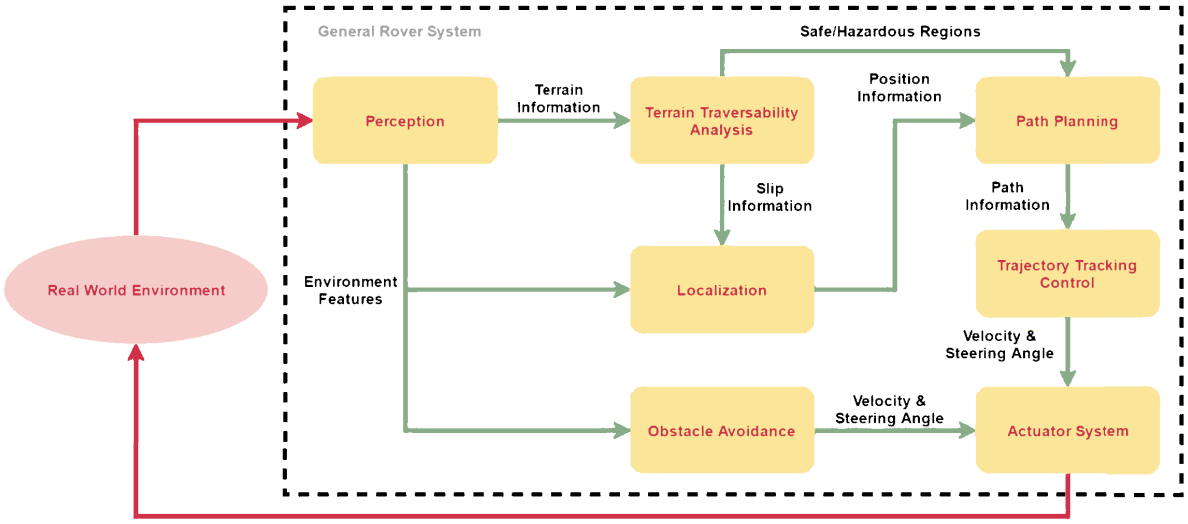



Figura 1.2: Scema generale del sistema di navigazione in un rover [2]

In questa tesi si vuole presentare un algoritmo innovativo a supporto dell'analisi di attraversabilità del terreno. Attraverso l'utilizzo di una rete neurale convoluzionale (CNN) viene riconosciuto, all'interno di un immagine, un terreno attraversabile e, grazie alla *point cloud* ottenuta da una stereocamera, è possibile costruire una *occupancy grid*. L'intero algoritmo viene testato sul campo utilizzando il prototipo di rover del progetto MORPHEUS.

La differenza principale rispetto ad altri algoritmi, già presenti in letteratura come [14], è la metodologia con cui viene ricostruita la mappa. Inoltre questo algoritmo permette d'identificare ostacoli "piani" che sarebbero difficilmente rilevabili, utilizzando un LiDAR oppure tramite il solo processamento della *point cloud*.

L'elaborato è strutturato nel seguente modo: nel Capitolo 2 vengono descritte il funzionamento della Visual Odometry e della SLAM; nel Capitolo 3 vengono introdotte le reti neurali ed in particolare le CNN; nel Capitolo 4 e nel Capitolo 5 vengono illustrati i passaggi principali per realizzare la rete neurale e il modo in cui è stato implementato l'intero algoritmo all'interno della rete ROS; nel Capitolo 6 vengono trattati i dettagli dell'intero apparato sperimentale e il modo in cui sono stati realizzati gli obiettivi della campagna sperimentale; infine nel Capitolo 7 vengono presentati i risultati ottenuti.

# Capitolo 2

# SLAM: Simultaneous Localization And Mapping

Con la SLAM si vuole rispondere al problema di un veicolo autonomo, che partendo da una posizione ignota in un ambiente ignoto, deve costruire una mappa dell'ambiente e deve essere in grado di localizzarsi all'interno di essa, rispetto ad un sistema di riferimento assoluto [15].

Per risolvere questo problema, sono state studiate negli anni diversi tipi di soluzioni nelle quali vengono utilizzati sensori differenti come sensori sonar, sensori IR, LASER scanner ed infine sensori visivi. Con l'utilizzo di questi ultimi sensori si è venuto a sviluppare un campo specifico noto come Visual SLAM oppure V-SLAM. Il vantaggio principale di quest'ultima tecnica è il minor costo dei sensori, ma allo stesso tempo essa richiede maggiori costi computazionali [13].

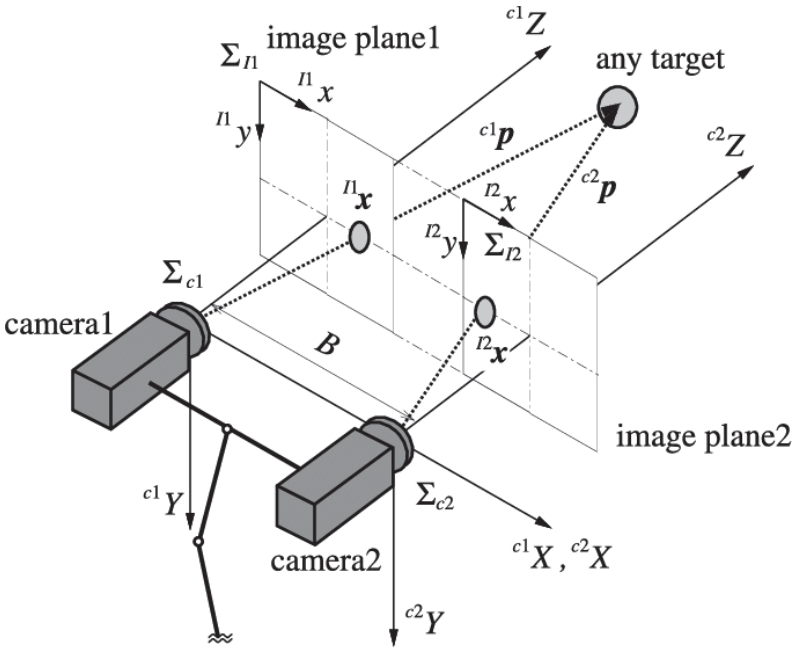



Figura 2.1: Setup di una stereo camera

Grazie allo sviluppo di CPU e GPU più potenti, è stato possibile utilizzare la V-SLAM anche per applicazioni in tempo reale. I sensori utilizzati possono essere mono camere, stereo camere, camere omni-direzionali e monocamere combinate con sensori di profondità (camere RGB-D). Per quanto riguarda i fini di questo elaborato, viene preso in considerazione soltanto il caso della Visual SLAM con stereo camera.

## 2.1 Concetti preliminari

Come mostrato in Figura 2.1 una stereo camera è composta da due fotocamere, una a sinistra e una a destra (camera 1 e camera 2). Il sistema di riferimento di ogni fotocamera ha l'asse z allineato all'asse ottico, l'origine coincidente con il centro ottico e gli assi x e y paralleli al piano dell'immagine e direzionati secondo l'ordine di numerazione dei pixel dell'immagine. I concetti di seguito illustrati fanno riferimento a [16].

### 2.1.1 Il modello di lente sottile

Una lente sottile è un modello matematico definito da un **asse ottico**, un piano perpendicolare ad esso, definito **piano focale**, con un'apertura centrata sul **centro ottico**, ovvero l'intersezione tra il piano focale e l'asse ottico, come mostrato in Figura 2.2.

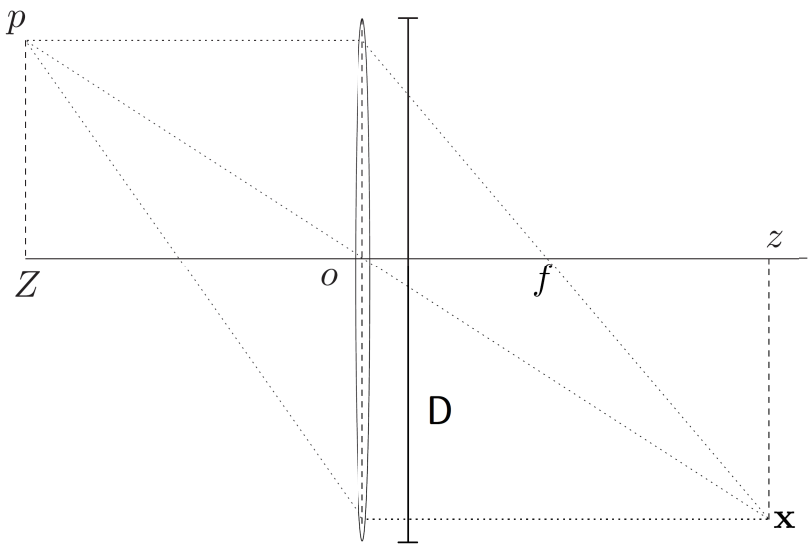



Figura 2.2: Modello di lente sottile

La lente sottile è descritta da due parametri, il diametro dell'apertura $D$ e la **lunghezza focale** $f$ ed è caratterizzata da due proprietà fondamentali. La prima afferma che tutti i raggi paralleli all'asse ottico ed entranti nell'apertura si intersecano in un punto dell'asse ottico, definito come **fuoco** della lente, ad una distanza dal centro ottico pari ad $f$. La seconda afferma che tutti i raggi passanti per il centro ottico non vengono deflessi.

Infine nel modello di lente sottile si considera solo la rifrazione della luce attraverso la lente, mentre si trascura per semplicità la diffrazione e la riflessione.

Immaginando di disegnare due raggi che partono da un punto p dello spazio e che passando attraverso la lente, come mostrato in Figura 2.2, vengono deflessi, essi si congiungeranno nuovamente nel punto **x** che giace sul **piano dell'immagine**.

Definita $Z$ la distanza di p lungo l'asse ottico e $z$ la distanza di **x** lungo lo stesso asse, tramite la similitudine dei triangoli, si ricava la seguente equazione fondamentale delle lenti sottili

$$\frac{1}{Z} + \frac{1}{z} = \frac{1}{f} \tag{2.1}$$

Si ricava anche il **campo di vista** della lente che è definito come $atan(D/2f)$.

### 2.1.2 Il modello di fotocamera con foro stenopeico

Se il diametro $D$ dell'apertura tende a zero, tutti i raggi sono costretti a passare per il centro ottico e quindi non saranno deflessi. Di conseguenza esiste una corrispondenza biunivoca tra il punto nello spazio $\mathbf{P} = (X, Y, Z, 1)_w$ ed il punto $\mathbf{p} = (u, v, 1)$ sul piano dell'immagine, espressi in coordinate omogenee.

In Figura 2.3 sono indicati i sistemi di riferimento utilizzati in questo modello, in particolare si notano il sistema inerziale $(X_w, Y_w, Z_w)$, il sistema della fotocamera $(X_c, Y_c, Z_c)$ e i due sistemi di riferimento sul piano dell'immagine, uno centrato sul **punto principale** di coordinate $(c_x, c_y)$ ed assi $(x, y, z)$, l'altro centrato rispetto all'angolo superiore sinistro dell'immagine con assi $(u, v)$, utilizzato per descrivere le coordinate dei pixel di un sensore digitale.

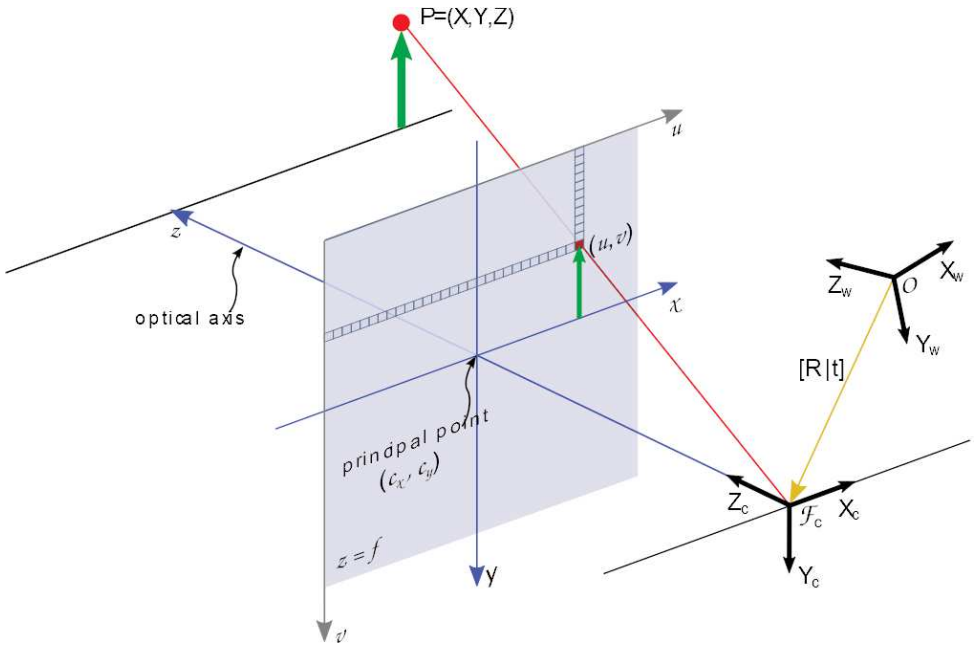



Figura 2.3: Modello di fotocamera con foro stenopeico

La relazione che lega il punto **P** al punto **p** è la seguente

$$\lambda \mathbf{p} = [A][P][G]\mathbf{P} \tag{2.2}$$

dove $\lambda$ è uno scalare definito **fattore di scala**, ovvero la coordinata $Z$ della Figura 2.2 generalmente ignota, $[A]$ è la **matrice dei parametri intrinseci** o **matrice di calibrazione** (Equazione 2.3), $[P]$ è la **matrice prospettica** (Equazione 2.4) e $[G]$ è la **matrice dei parametri estrinseci** (Equazione 2.5).

La matrice di calibrazione traduce le coordinate del punto **P**, espresse in metri nel sistema di riferimento della fotocamera, in coordinate sul piano dell'immagine espresse in pixel.

$$[A] = \begin{bmatrix} fs_x & fs_\theta & c_x \\ 0 & fs_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \tag{2.3}$$

dove $fs_x$ ed $fs_y$ sono rispettivamente la lunghezza focale espresse in pixel orizzontali ed in pixel verticali (se uguali, i pixel sono quadrati), invece $fs_\theta$ è espresso come $1/tan(\theta)$, con $\theta$ **angolo di skew**, ovvero l'angolo tra l'asse x e y. In generale questo parametro è nullo poiché solitamente $\theta = \pi/2$. Infine $c_x$ e $c_y$ sono le coordinate del punto principale espresse rispettivamente in pixel orizzontali e verticali.

La matrice prospettica, definita come segue, ha il solo scopo di raccordare la matrice di calibrazione con la matrice dei parametri estrinseci.

$$[P] = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{2.4}$$

Infine la matrice dei parametri estrinseci non è altro che la matrice di rototraslazione tra il sistema di riferimento inerziale e il sistema di riferimento della fotocamera.

$$[G] = \begin{bmatrix} [R] & [t] \\ 0 & 1 \end{bmatrix} \tag{2.5}$$

Solitamente, per una stereo camera ci si riferisce alla matrice dei parametri estrinseci come la rototraslazione tra la fotocamera di destra con quella di sinistra.

Per ottenere questi parametri, sia intrinseci che estrinseci, esistono alcuni metodi di calibrazione, nei quali viene tenuto conto anche delle distorsioni **radiali** e **tangenziali** della lente.

Tra i più utilizzati vi è il metodo di calibrazione Zhang [17], che consiste nel minimizzare gli errori di riproiezione degli angoli dei quadrati di una scacchiera, fotografata da diversi punti di vista.

### 2.1.3 Vincolo epipolare

Ipotizzando di avere una scena statica con luminosità costante e supponendo di avere due immagini della stessa scena osservata da due punti di vista differenti, è possibile identificare dei punti comuni alle due immagini, come mostrato in Figura 2.4.

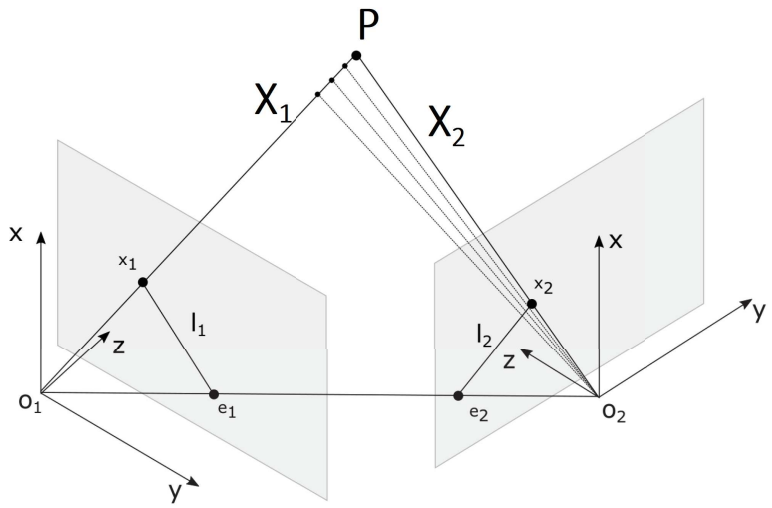



Figura 2.4: Rappresentazione dei sistemi di riferimento di due pose della fotocamera

Assumendo di assegnare alle due immagini i rispettivi sistemi di riferimento 1 e 2, e facendo combaciare il sistema inerziale con il sistema di riferimento 1, la matrice [G] diventa la rototraslazione tra i due sistemi di riferimento.

Definite $\mathbf{x}_1$ e $\mathbf{x}_2$ le coordinate omogenee della proiezione del punto $P$ sul piano dell'immagine delle due fotocamere e $\mathbf{X}_1$ e $\mathbf{X}_2$ le coordinate 3-D dello stesso punto rispetto ai due sistemi di riferimento, si ricavano le seguenti relazioni

$$\mathbf{X_2} = [R]\mathbf{X_1} + [t] \tag{2.6a}$$

$$\lambda_2\mathbf{x_2} = [R]\lambda_1\mathbf{x_1} + [t] \tag{2.6b}$$

Per poter eliminare la dipendenza da $\lambda_1$ e $\lambda_2$ bisogna moltiplicare ambo i membri dell'Equazione 2.6b per una matrice $[\widehat{T}]$ definita come segue

$$[\widehat{T}] = \begin{bmatrix} 0 & -t_3 & t_2 \\ t_3 & 0 & -t_1 \\ -t_2 & t_1 & 0 \end{bmatrix} \tag{2.7}$$

dove vale la seguente uguaglianza

$$[\widehat{T}]\mathbf{x}_i = [t] \wedge \mathbf{x}_i \tag{2.8}$$

infine moltiplicando ambo i membri dell'Equazione 2.6b per $\mathbf{x}_2^T$ l'equazione si semplifica in quanto $\mathbf{x}_2^T[\widehat{T}][t] = 0$ e $\mathbf{x}_2^T[\widehat{T}]\mathbf{x}_2 = 0$, poiché sono prodotti scalari di vettori perpendicolari tra di loro. Rimane quindi la seguente equazione

$$\mathbf{x}_2^T[\widehat{T}][R]\lambda_1\mathbf{x}_1 = 0 \tag{2.9}$$

e poiché $\lambda_1 > 0$ l'equazione si riduce a

$$\mathbf{x}_2^T[\widehat{T}][R]\mathbf{x}_1 = \mathbf{x}_2^T[E]\mathbf{x}_1 = 0 \tag{2.10}$$

che è la condizione di **vincolo epipolare**, dove [E] è chiamata **matrice essenziale**. L'Equazione 2.10 può anche essere scritta rispetto alle coordinate in pixel, utilizzando le matrici di calibrazione.

$$\mathbf{p}_2^T[A]_2^{-1,T}[\widehat{T}][R][A]_1^{-1}\mathbf{p}_1 = \mathbf{p}_2^T[F]\mathbf{p}_1 = 0 \tag{2.11}$$

dove [F] è chiamata la **matrice fondamentale**. L'interpretazione geometrica del vincolo epipolare è quella di imporre la complanarità tra i vettori $\mathbf{x}_1$, $\mathbf{x}_2$ e il vettore che congiunge i centri dei due sistemi di riferimento. Per completezza nella Figura 2.4 i punti $\mathbf{e}_1$ ed $\mathbf{e}_2$ sono chiamati **epipoli**, mentre i segmenti $l_1$ ed $l_2$ sono chiamate **linee epipolari**.

## 2.2 Visual Odometry

La Visual Odometry (VO) è il processo che stima il moto relativo di un veicolo dotato di una o più fotocamere, analizzando le sequenze di immagini ricevute. Questo termine, coniato nel 2004 da Nister [18], fu scelto per la sua somiglianza con la *wheel odometry*, che stima anch'essa in modo incrementale il moto di un veicolo, integrando il numero di giri delle ruote nel tempo.

Per lavorare in maniera efficace è opportuno che l'illuminazione sia sufficiente, che l'ambiente sia il più possibile statico e che ci siano sufficienti *features* per stimare al meglio il moto relativo. Inoltre è importante che le immagini siano il più possibile consecutive tra di loro, ovvero che non ci siano movimenti troppo veloci o troppo bruschi del veicolo.

Come per la *wheel odometry* alla VO sono associati degli errori cumulativi, ovvero che aumentano con lo scorrere del tempo, ma essa risulta lo stesso essere più affidabile e accurata della prima nel lungo periodo, anche perché non è affetta dagli errori dovuti all'accoppiamento ruota-terreno (come ad esempio lo slittamento).

La VO risulta essere un buon supporto a sistemi di navigazione come il GPS, le piattaforme inerziali (IMU), ecc. e in particolare è di cruciale importanza in ambienti dove è assente il segnale GPS come sotto acqua, sotto terra e sulla superficie di altri corpi celesti. I concetti di seguito illustrati fanno riferimento a [19], [20] e [21].

### 2.2.1 Formulazione generale del problema della VO

Facendo riferimento alla Figura 2.5 si definisce un sistema che si muove in un ambiente e che ad ogni istante temporale $k$ acquisisce delle immagini.

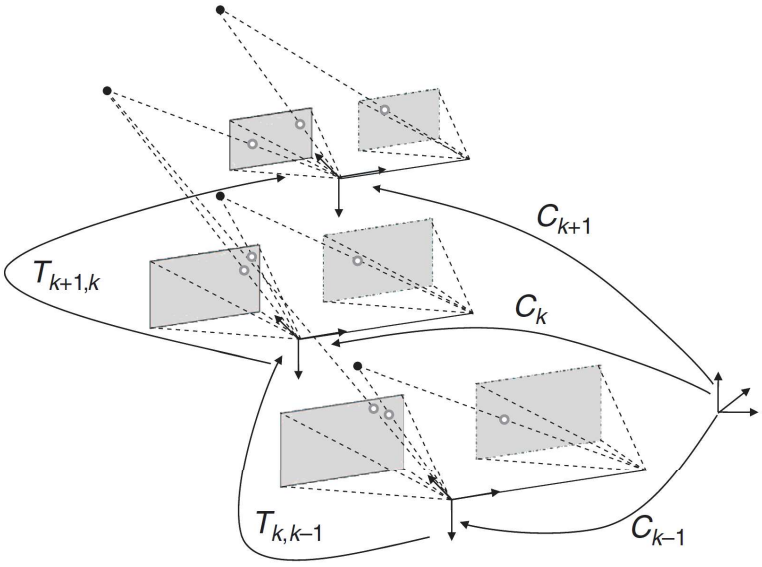



Figura 2.5: Rappresentazione del problema della VO

Nel caso di una fotocamera singola l'insieme delle immagini acquisite ad ogni istante è denotato come $I_{1:n} = \{I_1, ..., I_n\}$ mentre nel caso di una stereo camera i due insiemi delle fotocamere di sinistra e destra sono rispettivamente indicati come $I_{l,1:n} = \{I_{l,1}, ..., I_{l,1}\}$ e $I_{r,1:n} = \{I_{r,1}, ..., I_{r,n}\}$.

La trasformazione rigida di due pose della fotocamera tra due istanti temporali successivi $k-1$ e $k$ e denominata come $T_{k,k-1}$, come indicato nella formula seguente:

$$T_{k,k-1} = \begin{bmatrix} R_{k,k-1} & t_{k,k-1} \\ 0 & 1 \end{bmatrix}$$

dove $R_{k,k-1}$ è la matrice di rotazione tra i due sistemi, mentre $t_{k,k-1}$ è il vettore di traslazione. L'insieme delle trasformazioni rigide è definito come $T_{1:n} = \{T_{1,0}, ..., T_{n,n-1}\}$ e l'insieme delle pose, ovvero la trasformazione rigida tra la posa corrente e la posa iniziale, è indicato come $C_{0:n} = \{C_0, ..., C_n\}$. Vale quindi la seguente relazione

$$C_k = T_{k,k-1} C_{k-1} \tag{2.12}$$

Lo scopo principale della VO è proprio quello di calcolare la trasformazione $T_{k,k-1}$ in modo da poter ricostruire l'intera traiettoria della fotocamera.

Di seguito vengono riportati in Figura 2.6 i passaggi principali che devono essere compiuti per poter svolgere la VO.

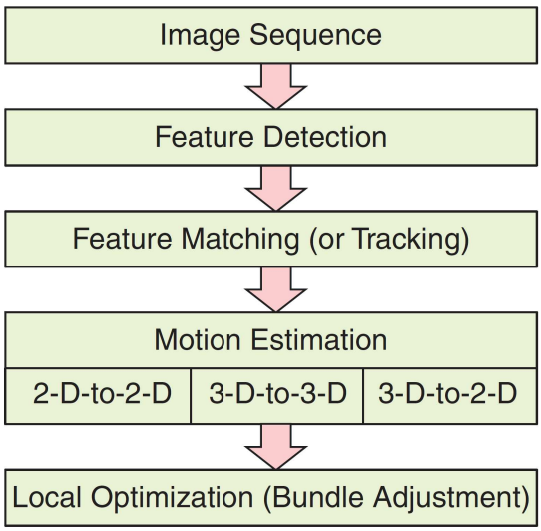



Figura 2.6: Schema riassuntivo dei passaggi della VO

Per ogni nuova immagine $I_k$ (*Image Sequence*), oppure coppia d'immagini $[I_{l,k}, I_{r,k}]$ se riferito ad una stereo camera, bisogna trovare (*Feature Detection*) e collegare (*Feature Matching*) oppure tracciare (*Feature Tracking*) le *features* 2-D comuni con l'immagine dell'istante precedente $I_{k-1}$. Il passo successivo consiste nel vero e proprio calcolo della trasformazione $T_{k,k-1}$ (*Motion Estimation*) attraverso tre metodi diversi, ottenendo così la posa della fotocamera $C_k$. Infine può essere fatta una ottimizzazione locale sulla stima della traiettoria utilizzando le ultime $m$ immagini catturate (*Bundle Adjustment*).

### 2.2.2 Feature Detection

All'interno di un'immagine esistono diversi punti che possono avere delle caratteristiche facilmente riconoscibili in altre immagini, come punti, angoli, linee dritte o curve, ecc. Quindi le caratteristiche principali per un buon algoritmo di *Feature Detection* sono:

- Accuratezza di localizzazione delle *features*: molto importante soprattutto per gli algoritmi di calibrazione, di ricostruzione dello spazio 3-D dalle immagini e di ricostruzione panoramica.
- Quantità: il numero ideale di *features* dipende dal tipo di applicazione. Nella maggior parte dei casi (rilevamento oggetti, recupero dell'immagine, ricostruzione 3-D) è importante avere un gran numero di *features* per poter avere una migliore accuratezza. Invece per applicazioni di classificazione degli oggetti sono sufficienti poche *features*.
- Invarianza: il riconoscimento delle *features* dovrebbe essere il più possibile resistente a cambi di prospettiva, d'illuminazione oppure di dimensione e posizione dell'oggetto (ingrandimento o traslazione).
- Efficienza computazionale: in particolare per applicazioni in tempo reale l'efficienza computazionale è un fattore critico per la scelta del ***feature detector***, soprattutto

per quel che riguarda non solo il tempo di elaborazione, ma anche il consumo energetico. Purtroppo il tempo impiegato dipende fortemente dal livello d'invarianza desiderato: un livello maggiore implica tempi più lunghi.

- Robustezza: le *features* rilevate devono essere il più possibile resistenti al rumore, ad effetti di discretizzazione, di compressione dell'immagine, di sfocatura e di deviazioni dal modello matematico utilizzato per ottenere l'invarianza, ecc.

Esistono diversi tipi di *feature detector*, ognuno con caratteristiche diverse come elencato in Tabella 2.1. Uno dei migliori è il SIFT (Scale Invariant Features Transform) inventato nel 1999 da Lowe [22] che, oltre a essere molto robusto, è anche molto preciso perché riesce a rilevare *features* con caratteristiche ben distinguibili dalle altre.

Naturalmente questo, il suo utilizzo implica un elevato costo computazionale e difficilmente viene utilizzato in applicazioni in tempo reale. Una buona alternativa è il SURF (Speeded Up Robust Features) proposto nel 2006 da Bay et al. [23], che si ispira al SIFT, ma è molto più veloce ed è abbastanza robusto.

| | Corner detector | Blob detector | Invarianza di rotazione | Invarianza di scala | Invarianza prospettica | Ripetibilità | Accuratezza localizzazione | Robustezza | Efficienza computazionale |
|---|---|---|---|---|---|---|---|---|---|
| **Harris** | ✗ | | ✗ | | | +++ | +++ | ++ | ++ |
| **Shi-Tomasi** | ✗ | | ✗ | | | +++ | +++ | ++ | ++ |
| **Harris-Laplacian** | ✗ | ✗ | ✗ | ✗ | | +++ | +++ | ++ | + |
| **Harris-Affine** | ✗ | ✗ | ✗ | ✗ | ✗ | +++ | +++ | ++ | ++ |
| **SUSAN** | ✗ | | ✗ | | | ++ | ++ | ++ | +++ |
| **FAST** | ✗ | | ✗ | | | ++ | ++ | ++ | ++++ |
| **SIFT** | | ✗ | ✗ | ✗ | ✗ | +++ | ++ | +++ | + |
| **MSER** | | ✗ | ✗ | ✗ | ✗ | +++ | + | +++ | +++ |
| **SURF** | | ✗ | ✗ | ✗ | ✗ | ++ | ++ | ++ | ++ |

Tabella 2.1: Confronto dei *feature detectors*

L'ultimo passaggio della *Feature Detection* è quello di convertire una regione di pixel intorno alla *feature* identificata in un ***feature descriptor***, ossia memorizzare quelle che possono essere le caratteristiche rilevanti di quella *feature* o della regione circostante (come il colore o l'intensità luminosa), in modo che possa essere facilmente associata alla *feature* di un'altra immagine.

### 2.2.3 Feature Matching e Feature Tracking

Contestualizzato all'interno della VO, ci si riferisce alla fase in cui si cercano *features* identiche, comuni tra due o più immagini consecutive (siano esse singole oppure stereo-coppie). In quest'ultimo caso è più corretto parlare di *Feature Tracking*, in quanto spesso si utilizzano algoritmi particolari di tracciatura delle *features* come ad esempio il Kana-deLucasTomasi (KLT) [24]. Per il *Feature Matching*, invece, il metodo è quello di cercare *feature descriptors* simili e di misurare la bontà della loro similitudine. Supponendo di avere due immagini $I_1$ e $I_2$ di cui si vogliono comparare due aree di grandezza $m \times n$ dove $m = 2a + 1$ e $n = 2b + 1$, centrate rispettivamente in $(u, v)$ e $(u', v')$ si possono utilizzare alcuni dei seguenti criteri:

- **Sum of Absolute Differences (SAD)**

$$SAD = \sum_{k=-a}^{a} \sum_{l=-b}^{b} |I_1(u+k, v+l) - I_2(u'+k, v'+l)|$$

- **Sum of Squared Differences (SSD)**

$$SSD = \sum_{k=-a}^{a} \sum_{l=-b}^{b} |I_1(u+k, v+l) - I_2(u'+k, v'+l)|^2$$

- **Normalized Cross Correlation (NCC)**

$$NCC = \frac{\sum_{k=-a}^{a} \sum_{l=-b}^{b} [I_1(u+k, v+l) - \mu_1] \cdot [I_2(u'+k, v'+l) - \mu_2]}{\sqrt{\sum_{k=-a}^{a} \sum_{l=-b}^{b} [I_1(u+k, v+l) - \mu_1]^2 [I_2(u'+k, v'+l) - \mu_2]^2}}$$

  dove

$$\mu_1 = \frac{1}{mn} \sum_{k=-a}^{a} \sum_{l=-b}^{b} I_1(u+k, v+l)$$

$$\mu_2 = \frac{1}{mn} \sum_{k=-a}^{a} \sum_{l=-b}^{b} I_2(u'+k, v'+l)$$

Infine viene applicato un filtraggio RANSAC [25] (RANdom SAmple Consensus) per eliminare gli outliers. L'algoritmo viene riportato in Appendice A.1.

### 2.2.4 Motion Estimation

La parte centrale della VO è proprio la stima del moto, ovvero l'ottenimento della $T_{k,k-1}$ grazie alla concatenazione di tutte le immagini precedenti. A seconda del tipo di

corrispondenza che viene fatto sulle *features* (2-D oppure 3-D), esistono tre metodi diversi per ottenere la trasformazione tra le due pose.

**2-D-to-2-D**

Le *features* in $I_{k-1}$ e in $I_k$ sono, in entrambe le immagini, identificate da coordinate 2-D. Sapendo che le loro coordinate devono rispettare la condizione di vincolo epipolare (Equazione 2.10), tramite alcuni algoritmi, è possibile stimare la matrice essenziale.

Il primo di tutti, in ordine cronologico, è l'*Eight-point algorithm* [26], che stima la matrice essenziale utilizzando almeno otto punti. Inoltre nel caso in cui si abbiano tante *features* è importante applicare un filtraggio (come ad esempio il RANSAC), per poter eliminare i falsi *matching* ed avere così una stima più accurata di [E].

Infine sfruttando la decomposizione ai valori singolari (SVD) si riesce ad ottenere la matrice essenziale $[\bar{E}]$ che rispetta la condizione imposta in cui $[E] = [\widehat{T}][R]$. La matrice di rotazione e la traslazione vengono ottenute come segue

$$\begin{aligned}[\bar{E}] &= [U][S][V]^T \\ [R] &= [U](\pm[W]^T)[V]^T \\ [t] &= [U](\pm[W])[S][V]^T\end{aligned}$$

dove

$$[W]^T = \begin{bmatrix} 0 & \pm 1 & 0 \\ \mp 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Infine bisogna scalare la traslazione per il fattore di scala relativo tra le due pose. Per far ciò bisogna triangolare tutti i punti $X_{k-1}$ e $X_k$ da due immagini consecutive ed infine calcolare il rapporto tra le distanze di una stessa coppia di punti.

$$r = \frac{||X_{k-1,i} - X_{k-1,j}||}{||X_{k,i} - X_{k,j}||}$$

**3-D-to-3-D**

In questo caso, utilizzando ad esempio una stereo camera, è possibile calcolare le coordinate dei punti ad ogni istante, tramite la triangolazione delle *features* comuni all'immagine di destra e di sinistra. Ottenuti così i punti $X_{k-1}$ e $X_k$, tramite le due coppie d'immagini $(I_{l,k-1}, I_{r,k-1})$ e $(I_{l,k}, I_{r,k})$, il problema si riduce a trovare la trasformazione $T_k$ che minimizza le distanze tra i punti della prima e della seconda coppia d'immagini

$$arg \min_{T_k} = \sum_i ||\widetilde{X}_k^i - T_k \widetilde{X}_{k-1}^i|| \tag{2.15}$$

dove $\widetilde{X}_k = [x, y, z, 1]_k$ sono le coordinate omogenee della $i$-esima *feature* al tempo $k$.

Una soluzione, proposta in [27] per un numero di *features* $n \geq 3$, consiste nel calcolare la traslazione come la differenza dei centroidi delle coordinate delle *features* $\overline{X}$, mentre la rotazione viene calcolata tramite la scomposizione SVD. Risulta quindi che la rotazione e la traslazione sono date da:

$$\begin{aligned} USV^T &= svd[(X_{k-1} - \overline{X}_{k-1})(X_k - \overline{X}_k)^T] \\ R_k &= VU^T \\ t_k &= \overline{X}_k - R_k\overline{X}_{k-1} \end{aligned}$$

Un'ulteriore soluzione è quella proposta da Horn in [28], dove la matrice di rotazione viene calcolata tramite la risoluzione di un problema di minimizzazione.

### 3-D-to-2-D

In questo caso le *features* in $I_{k-1}$ vengono date in coordinate 3-D, mentre in $I_k$ vengono fornite le loro riproiezioni sul piano dell'immagine.

Come affermato da Nister [18] la stima del moto 3-D-2-D è molto più accurata rispetto a quella 3-D-3-D, in quanto si minimizza l'errore di riproiezione (Equazione 2.17) invece che l'errore di posizione delle *features* (Equazione 2.15).

$$arg \min_{T_k} = \sum_i ||p_k^i - \hat{p}_{k-1}^i|| \tag{2.17}$$

dove $\hat{p}_{k-1}^i$ è la riproiezione dei punti 3-D $X_{k-1}^i$ nell'immagine $I_k$ in accordo con la trasformazione $T_k$. Anche con una singola fotocamera è possibile utilizzare questa tecnica, in cui la triangolazione viene effettuata utilizzando le due immagini precedenti ($I_{k-2}$ e $I_{k-1}$).

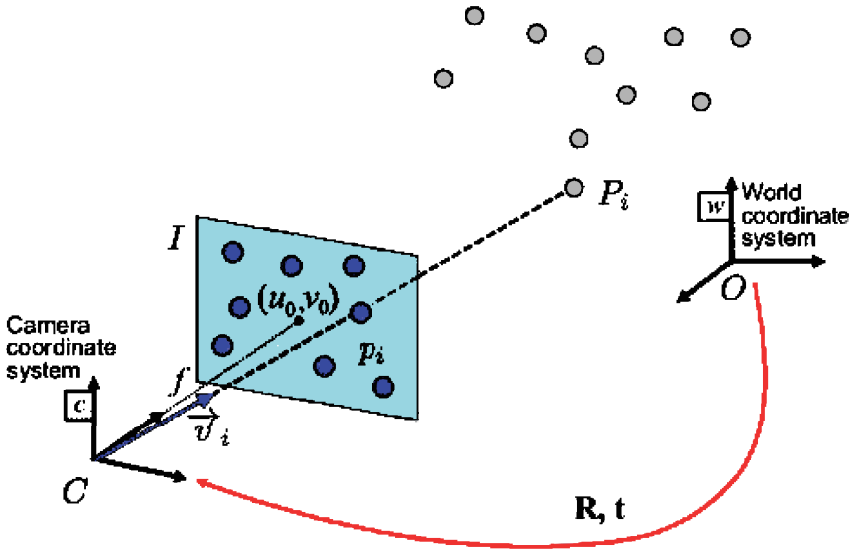



Figura 2.7: Rappresentazione del problema PnP

Il problema della riproiezione in 2-D dei punti 3-D è nota come "***P**erspective from **n** **P**oints*" (PnP) e il minimo numero di punti di vista che occorrono a trovare una soluzione è pari a tre ed è nota come "***P**erspective from **t**hree **P**oints*" (P3P).

In altre parole, facendo riferimento alla Figura 2.7, si vuole determinare la $[R]$ e $[t]$ del sistema di riferimento $C$ rispetto a $O$, tale per cui l'errore di riproiezione dei punti $P_i$ in $I$ (ovvero i punti $p_i$) sia il minore possibile.

La risoluzione del problema P3P necessita di conoscere i parametri intrinseci della fotocamera e le corrispondenze di almeno quattro *features*. In questo modo la soluzione è in generale unica. In letteratura esistono diversi algoritmi che risolvono il problema P3P, come ad esempio [29] e [30].

### 2.2.5 Bundle Adjustment

Calcolata in questo modo la rototraslazione tramite l'Equazione 2.12 è possibile calcolare le pose successive. Questo però comporta un continuo aumento dell'incertezza, come illustrato in Figura 2.8, e quindi un errore sul calcolo della traiettoria finale.

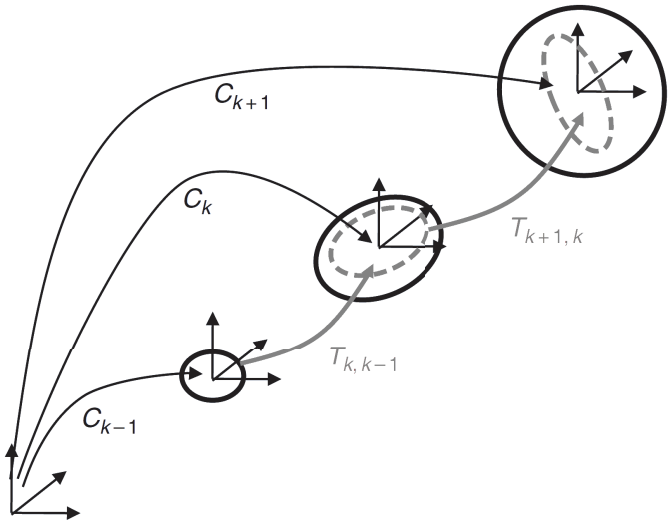



Figura 2.8: Propagazione dell'incertezza per le pose successive

La soluzione è ottimizzare la posa della fotocamera considerando anche le $n$ pose precedenti. Bisogna quindi rappresentare la successione delle pose come un grafo, dove i nodi ($C_i$,$C_j$) rappresentano le pose, mentre gli archi ($e_{ij}$) rappresentano le rototraslazioni ($T_{e_{ij}}$). Per ogni nuova trasformazione viene aggiunto un nuovo arco, il quale definisce la seguente funzione di costo:

$$\sum_{e_{ij}} ||C_i - T_{e_{ij}} C_j||^2 \tag{2.18}$$

Una prima ottimizzazione può essere fatta con quella che viene chiamata ***loop closure***, ovvero quando il robot riconosce delle *features* già rilevate in precedenza e aggiorna la propria traiettoria, ovvero aggiunge un ulteriore arco tra due nodi del grafo.

Con il *Bundle Adjustment* [31], oltre ad essere ottimizzata la traiettoria vengono ottimizzate anche le posizioni dei *landmarks*, ovvero quelle *features* che sono presenti all'interno di una sequenza di $n$ immagini. Quindi l'ottimizzazione è fatta a livello locale, perché dipende da un numero limitato d'immagini. La funzione di costo da minimizzare è la seguente:

$$arg \min_{X^i, C_k} = \sum_{i,k} ||p_k^i - g(X^i, C_k)||^2 \tag{2.19}$$

dove $p_k^i$ è la proiezione sul piano della k-esima immagine del punto 3-D della i-esima *features* $X^i$, mentre $g(X^i, C_k)$ è la funzione di riproiezione di $X^i$ sul piano dell'immagine data la k-esima posa $C_k$.

## 2.3 Visual SLAM con stereo camera

La differenza principale tra Visual SLAM e VO è la tecnica con il quale viene ricostruita la traiettoria del robot. La prima infatti colloca in una mappa 3D le posizioni delle *features* (chiamate in questo contesto ***landmarks***) e si localizza all'interno della mappa, calcolando la distanza dai *landmarks* stessi.

Ovviamente le misure saranno soggette ad un'incertezza, che però può essere essere ridotta se il robot passa per un luogo già precedentemente visitato (***loop closure***), in quanto sarà in grado di riconoscere i medesimi *landmarks*. Proprio il riconoscimento della *loop closure* ed il modo di correggere efficacemente la mappa e la traiettoria sono le caratteristiche che più differenziano la SLAM dalla VO.

Per quest'ultima infatti non esiste una vera e propria ottimizzazione a livello globale, ma solo a livello locale (*Bundle Adjustment*). Inoltre la traiettoria, come già descritto nella Sezione 2.2, viene ricostruita tramite una stima della rototraslazione tra due pose consecutive. I concetti di seguito illustrati fanno riferimento a [15], [32] e [21].

### 2.3.1 Formulazione generale del problema della SLAM

Considerando un robot che si muove all'interno di un ambiente sconosciuto e che ad ogni istante $k$ compie delle osservazioni, rilevando un certo numero di *landmarks*, si possono definire alcuni parametri come mostrato in Figura 2.9.

Nell'immagine $x_k$ rappresenta la posa del robot al tempo $k$, $u_k$ è il vettore di controllo necessario a far passare il robot dalla posa $x_{k-1}$ a quella $x_k$, $m_i$ è la posizione dell'i-esimo *landmark* considerato fisso, mentre $Z_{k,i}$ è la misura del i-esimo *landmark* fatta all'istante $k$-esimo.

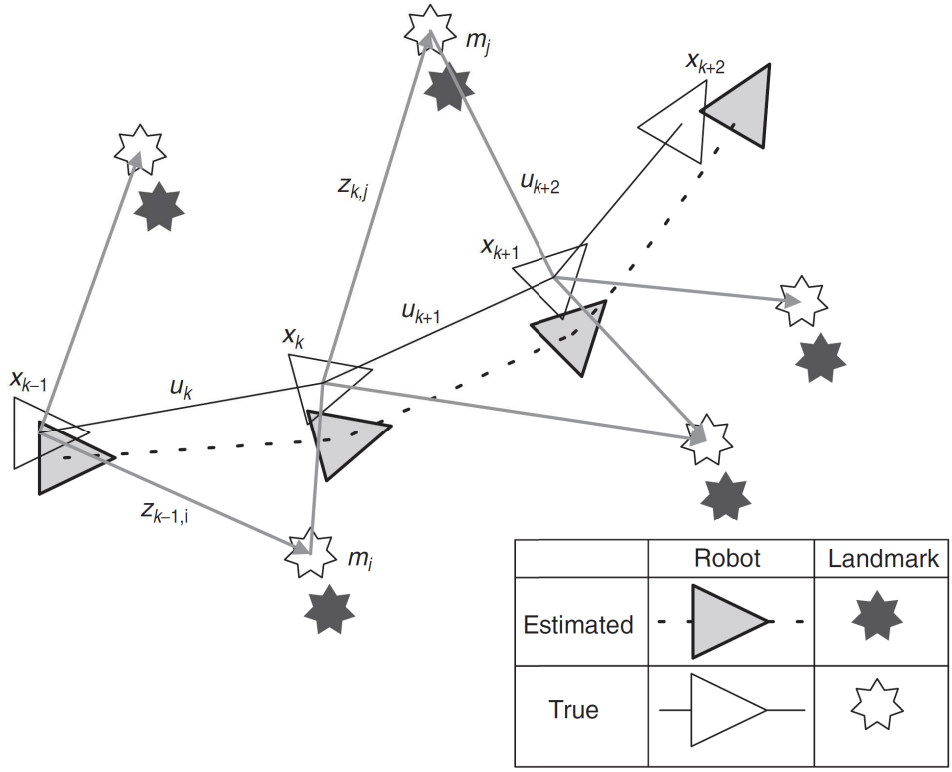



Figura 2.9: Rappresentazione del problema della SLAM

Dal momento che questo tipo di approccio si basa su delle misurazioni, è utile utilizzare una formulazione probabilistica del problema della SLAM. Il requisito quindi è che ad ogni istante temporale $k$ venga calcolata la seguente distribuzione di probabilità

$$bel(k) = P[(\mathbf{x}_k, \mathbf{m})|(\mathbf{Z}_{0:k}, \mathbf{U}_{0:k}, \mathbf{x}_0)] \tag{2.20}$$

L'Equazione 2.20 descrive la densità di probabilità sia dei *landmarks* $\mathbf{m} = \{m_1, m_2, ..., m_i\}$ che della posa del robot di trovarsi in una determinata posizione al tempo $k$, date le misurazioni fatte fino a quel momento sui *landmarks* $\mathbf{Z}_{0:k} = \{z_0, z_1, ..., z_k\}$, dati i vettori di controllo forniti fino al quel momento $\mathbf{U}_{0:k} = \{u_0, u_1, ..., u_k\}$ e data la posa iniziale del robot $\mathbf{x}_0$. Per lo scopo di questo elaborato $\mathbf{U}_{0:k}$ è la matrice di rototraslazione $T_{k,k-1}$ fornita dalla Visual Odometry.

Inoltre si vuole far notare che quest'ultima equazione racchiude sia il problema di localizzazione, dove la distribuzione di probabilità $P[\mathbf{x}_k|(\mathbf{Z}_{0:k}, \mathbf{U}_{0:k}, \mathbf{m})]$ assume come nota la mappa $\mathbf{m}$, sia il problema di mappatura, dove la distribuzione di probabilità $P[\mathbf{m}|(\mathbf{X}_{0:k}, \mathbf{Z}_{0:k}, \mathbf{U}_{0:k})]$ assume come note le pose $\mathbf{X}_{0:k}$.

La soluzione è ricorsiva, ovvero nel primo passaggio viene stimata la posizione basandosi sul controllo ricevuto, mentre nel secondo passaggio la stima della posizione finale viene migliorata, includendo le misure di distanza fatte sui *landmarks*.

É necessario avere sia un modello probabilistico del moto, sia uno modello probabilistico delle misure di distanza. Per far ciò bisogna definire la posa e le misure al tempo $k$ come segue:

$$\mathbf{x_k} = f(\mathbf{u_k}, \mathbf{x}_{k-1})$$
$$\mathbf{z_k} = g(\mathbf{x_k}, \mathbf{m})$$

così si possono definire le distribuzioni di probabilità nel seguente modo:

$$P[\mathbf{x}_k|(\mathbf{u}_k, \mathbf{x}_{k-1})] \tag{2.22a}$$
$$P[\mathbf{z}_k|(\mathbf{x}_k, \mathbf{m})] \tag{2.22b}$$

Assumendo che il processo sia di tipo markoviano, ovvero che le variabili al tempo $k$ dipendano solo ed esclusivamente dalle variabili al tempo $k-1$, si può costruire l'algoritmo della SLAM nel seguente modo:

- **Passaggio di predizione: aggiornamento dei tempi**

$$\begin{aligned} \overline{bel}(k) &= P[(\mathbf{x}_k, \mathbf{m})|(\mathbf{Z}_{0:k-1}, \mathbf{U}_{0:k}, \mathbf{x}_0)] = \\ &= \int P[\mathbf{x_k}|(\mathbf{u_k}, \mathbf{x}_{k-1})] \cdot bel(k-1)\, d\mathbf{x}_{k-1} \end{aligned} \tag{2.23}$$

- **Passaggio di correzione: aggiornamento delle misure**

$$bel(k) = \frac{\overline{bel}(k) \cdot P[\mathbf{z}_k|(\mathbf{x}_k, \mathbf{m})]}{P[\mathbf{z}_k|(\mathbf{Z}_{0:k-1}, \mathbf{U}_{0:k})]} \tag{2.24}$$

Nell'Equazione 2.23 si calcola $x_k$ basandosi solamente sul modello probabilistico del moto e sullo stato precedente $bel(k-1)$. Lo stato predetto al tempo $k$ si ottiene tramite il teorema della probabilità assoluta.

Infine, tramite il teorema di Bayes, nell'Equazione 2.24 si calcola lo stato al tempo $k$ aggiornando lo stato predetto $\overline{bel}(k)$ con le misure fatte al tempo $k$.

Per quanto riguarda gli errori sulla posizione dei *landmarks*, in generale gli errori relativi sono molto inferiori rispetto a quelli assoluti.

Solo per il caso lineare di distribuzioni di tipo gaussiano si è potuto dimostrare che gli errori sulla stima della posizione dei *landmarks* diminuiscono monotonicamente all'aumentare delle osservazioni [33].

In sostanza può essere fatta un'analogia con una rete di molle, come illustrato in Figura 2.10.

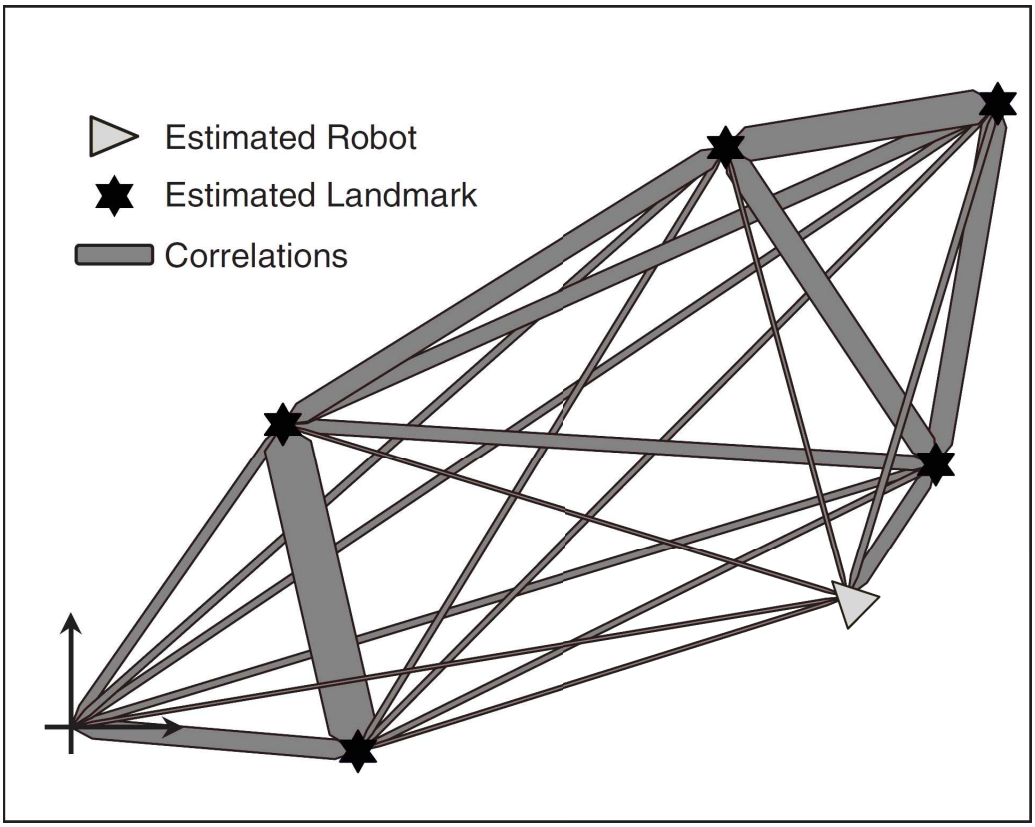



Figura 2.10: Analogia della rete di molle

Supponendo di collegare ciascun *landmark* tra di loro e tra le pose del robot con una molla, la rigidezza di quest'ultima è proporzionale all'accuratezza delle misure di posizione.

Ad ogni nuova misurazione viene aggiunto un nuovo vincolo sulla posizione del *landmark* che a sua volta va a influenzare anche la posizione degli altri, in base a quanto forte è la loro correlazione.

Continuando a fissare dei vincoli, le correlazioni diventano sempre più forti, e si riduce quindi l'errore di posizione dei *landmarks*.

### 2.3.2 Soluzione al problema della SLAM

La risoluzione al problema della SLAM consiste principalmente nel formulare degli appropriati modelli probabilistici del moto (Equazione 2.22a) e delle misure (Equazione 2.22b)

Le risoluzioni più comuni e più utilizzate in letteratura sono EKF-SLAM, FastSLAM e Graph-SLAM. Per semplicità di trattazione, verrà data una breve introduzione soltanto del primo tipo di risoluzione, in quanto è alla base delle altre due.

**EKF-SLAM**

L'algoritmo di risoluzione della SLAM basato sull'utilizzo di un filtro di Kalman esteso (EKF) è stato uno dei primi metodi ad essere utilizzato. La ragione principale sono state la semplicità d'implementazione e le grandi potenzialità del filtro di Kalman. Alla base c'è la formulazione delle Equazioni 2.22a e 2.22b come segue:

$$P[\mathbf{x}_k|(\mathbf{u}_k, \mathbf{x}_{k-1})] \iff \mathbf{x}_k = \mathbf{f}(\mathbf{x}_{k-1}, \mathbf{u}_k) + \mathbf{w}_k \tag{2.25a}$$

$$P[\mathbf{z}_k|(\mathbf{x}_k, \mathbf{m})] \iff \mathbf{z}_k = \mathbf{h}(\mathbf{x}_k, \mathbf{m}) + \mathbf{v}_k \tag{2.25b}$$

dove $\mathbf{f}(\cdot)$ e $\mathbf{h}(\cdot)$ sono rispettivamente il modello della cinematica del robot e il modello di misurazione, mentre $\mathbf{w}_k$ e $\mathbf{v}_k$ sono i rumori gaussiani bianchi additivi, associati rispettivamente alle matrici di covarianza $\mathbf{Q}_k$ e $\mathbf{R}_k$. Con questi parametri, tramite il metodo EKF, si possono calcolare il valore medio finale

$$\begin{bmatrix} \hat{\mathbf{x}}_{k|k} \\ \hat{\mathbf{m}}_k \end{bmatrix} = E \begin{bmatrix} \mathbf{x}_k \\ \mathbf{m} \end{bmatrix} |\mathbf{Z}_{0:k}$$

e la matrice di covarianza della distribuzione di probabilità (Equazione 2.20).

$$\mathbf{P}_{k|k} = \begin{bmatrix} \mathbf{P}_x x & \mathbf{P}_x m \\ \mathbf{P}_x m & \mathbf{P}_m m \end{bmatrix} = E \left[ \begin{pmatrix} \mathbf{x}_k - \hat{\mathbf{x}}_k \\ \mathbf{m} - \hat{\mathbf{m}}_k \end{pmatrix} \begin{pmatrix} \mathbf{x}_k - \hat{\mathbf{x}}_k \\ \mathbf{m} - \hat{\mathbf{m}}_k \end{pmatrix}^T |\mathbf{Z}_{0:k} \right]$$

Il calcolo avviene in due fasi:

- **Time-update**

$$\hat{\mathbf{x}}_{k|k-1} = \mathbf{f}(\hat{\mathbf{x}}_{k-1|k-1}, \mathbf{u}_k)$$
$$\mathbf{P}_{xx,k|k-1} = \nabla\mathbf{f}\, \mathbf{P}_{xx,k-1|k-1}\, \nabla\mathbf{f}^T + \mathbf{Q}_k$$

  dove $\nabla\mathbf{f}$ è lo Jacobiano di $\mathbf{f}$ valutato nel punto $\hat{\mathbf{x}}_{k-1|k-1}$. In maniera analoga a quanto visto nel paragrafo precedente, questa fase corrisponde al passaggi di predizione, dove viene ipotizzata una distribuzione di probabilità basandosi solo sul vettore di controllo.

- **Observation-update**

$$\begin{bmatrix} \hat{\mathbf{x}}_{k|k} \\ \hat{\mathbf{m}}_k \end{bmatrix} = \begin{bmatrix} \hat{\mathbf{x}}_{k|k} & \hat{\mathbf{m}}_k \end{bmatrix} + \mathbf{W}_k \left[ \mathbf{z}_k - \mathbf{h}(\hat{\mathbf{x}}_{k|k-1}, \hat{\mathbf{m}}_{k-1}) \right]$$
$$\mathbf{P}_{k|k} = \mathbf{P}_{k|k-1} - \mathbf{W}_k \mathbf{S}_k \mathbf{W}_k^T$$

dove

$$\mathbf{S}_k = \nabla \mathbf{h}\, \mathbf{P}_{k|k-1}\, \nabla \mathbf{h}^T + \mathbf{R}_k$$
$$\mathbf{W}_k = \mathbf{P}_{k|k-1}\, \nabla \mathbf{h}^T\, \mathbf{S}_k^{-1}$$

e dove $\nabla \mathbf{h}$ è lo Jacobiano di $\mathbf{h}$ valutato nel punto $\hat{\mathbf{x}}_{k|k-1}$ e $\hat{\mathbf{m}}_{k-1}$. Questa fase corrisponde al passaggio di predizione in cui la posa e la mappa vengono aggiornate prendendo in considerazione le misure effettuate. La matrice $\mathbf{W}_k$ è chiamata guadagno di Kalman e la sua funzione è proprio quella di dare maggiore rilevanza alle misurazioni fatte oppure alla stima del moto.

La EKF-SLAM è applicabile anche a modelli non lineari, se questi possono essere linearizzati localmente. Tuttavia se il modello è fortemente non lineare, possono insorgere problemi d'inconsistenza della soluzione. La convergenza e la consistenza sono garantite solo nel caso lineare [33]. Quindi quanto più il modello si avvicina alla linearità, tanto più è facile che il metodo converga ad una soluzione.

# Capitolo 3

# Reti Neurali Convoluzionali

La segmentazione semantica consiste nella suddivisione di un'immagine in categorie aventi proprietà simili tra di loro. Essa viene utilizzata in diversi campi, quali l'analisi delle immagini satellitari, l'ispezione industriale e il controllo dei processi e ovviamente la navigazione autonoma.

Questa viene realizzata tramite l'utilizzo delle reti neurali profonde (**D**eep **N**eural **N**etwork) e in particolare le cosiddette reti neurali convoluzionali (**C**onvolutional **N**eural **N**etwork). Per poter parlare delle CNN è quindi necessario fare un'introduzione alle reti neurali in generale e alle DNN [34].

## 3.1 Rete Neurale

Le reti neurali artificiali (**A**rtificial **N**eural **N**etwork) sono una tecnica di *Machine Learning* in cui viene simulato il meccanismo di apprendimento del cervello negli esseri viventi. In generale il loro scopo è quello di suddividere i dati ricevuti in input, in diverse categorie.

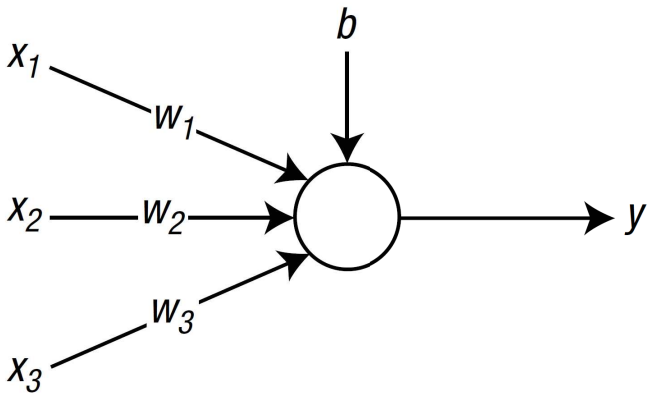



Figura 3.1: Un nodo che riceve tre input

Il sistema nervoso è composto da un insieme di neuroni connessi tra loro tramite le sinapsi. L'intensità della connessione tra le sinapsi spesso può cambiare in risposta a degli stimoli esterni, andando così a produrre una risposta dell'intera rete neuronale differente da prima [35].

In maniera simile l'intero meccanismo viene ricreato all'interno della ANN, dove i neuroni vengono chiamati **nodi**, mentre le sinapsi e l'intensità dei loro collegamenti sono i **pesi** ai quali viene associato un valore.

Si consideri un nodo che riceve tre input, come mostrato in Figura 3.1, dove il cerchio rappresenta il nodo, mentre la freccia indica la direzione del flusso del segnale passante per il nodo stesso. I segnali di input sono $x_1$, $x_2$ e $x_3$ e i pesi sono indicati con $w_1$, $w_2$ e $w_3$.

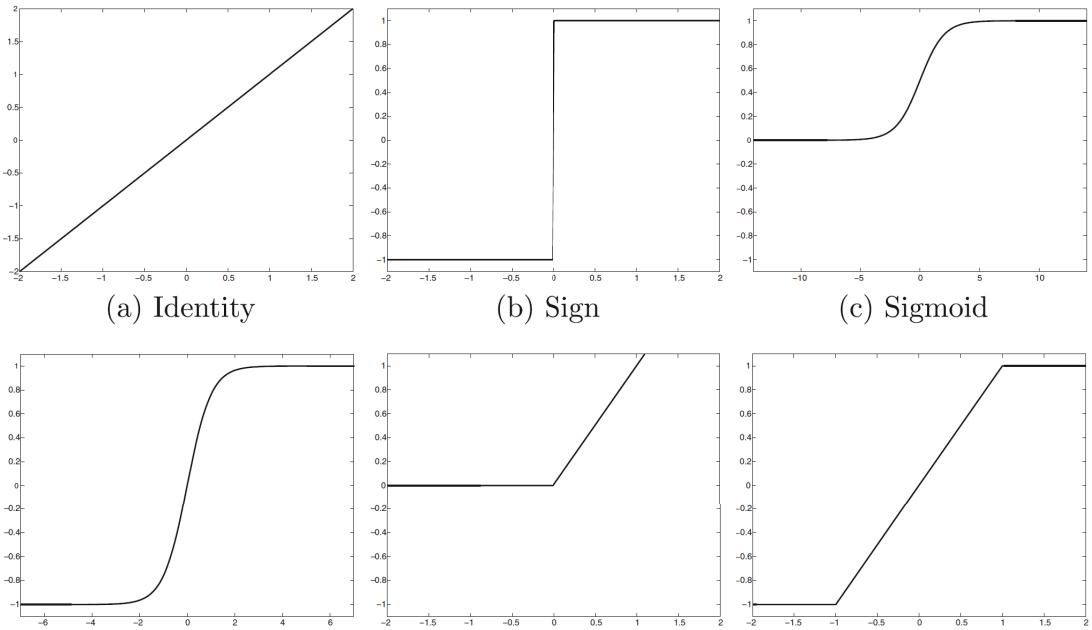
Infine $b$ è il **bias** del nodo, ovvero un valore fisso che viene assegnato a ciascun nodo. Nel complesso si avrà dunque che:

$$\nu = (w_1 \cdot x_1) + (w_2 \cdot x_2) + (w_3 \cdot x_3) + b \tag{3.1}$$

ed in generale denotando $\widehat{\mathbf{x}} = \{x_1, x_2, ..., x_n\}$ e $\widehat{\mathbf{w}} = \{w_1, w_2, ..., w_n\}$ si ha che:

$$\nu = \widehat{\mathbf{w}} \cdot \widehat{\mathbf{x}} + b \tag{3.2}$$

Infine l'output del nodo $y$ è il risultato della **funzione di attivazione** $\varphi(\nu)$, dove $\varphi(\cdot)$ è una funzione che determina il comportamento del nodo.



(a) Identity (b) Sign (c) Sigmoid

(d) Tanh (e) ReLU (f) Hard Tanh

Figura 3.2: Esempi di funzione di attivazione [3]

Esistono diverse funzioni di attivazione, come mostrato in Figura 3.2, la cui scelta può essere cruciale per la fase di allenamento della rete, in particolare per le reti neurali *Multi-Layer*. In generale si avrà quindi che:

$$y = \varphi(\nu) = \varphi(\widehat{\mathbf{w}} \cdot \widehat{\mathbf{x}} + b) \tag{3.3}$$

### 3.1.1 Rete Neurale Single-Layer

I primi studi sulle reti neurali cominciarono intorno alla metà degli anni '50 e alla base ci furono proprio le reti neurali a strato singolo chiamate **Perceptron**. Come illustrato in Figura 3.3 questa rete neurale è costituita da due strati: l'*input layer* e l'*output layer*.

Inoltre con $w_{ij}$ si indica il peso che collega il nodo $i$ dell'*input layer* al nodo $j$ dell'*output layer*. In questo modo è possibile raccogliere tutti i pesi all'interno di una matrice $n$x$m$, dove $n$ è il numero di nodi in output e $m$ il numero di nodi in input.

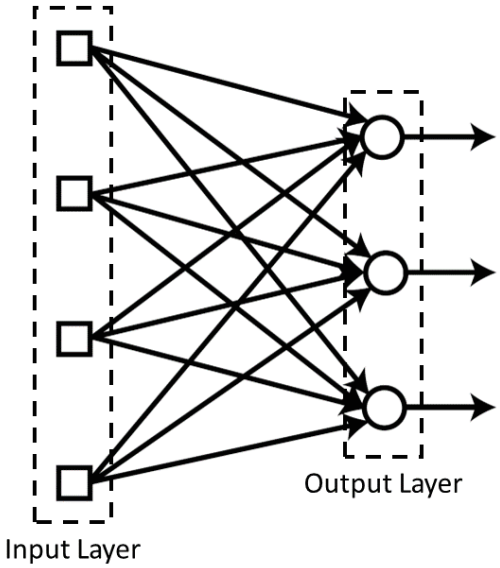



Figura 3.3: Single-Layer Neural Network

Le Equazioni 3.2 e 3.3 diventano:

$$\widehat{\nu} = [\mathbf{W}] \cdot \widehat{\mathbf{x}} + \widehat{\mathbf{b}}$$
$$\widehat{\mathbf{y}} = \varphi(\widehat{\nu})$$

Come accennato precedentemente, poiché lo scopo delle reti neurali è proprio quello di suddividere in differenti categorie i dati in input, bisogna "istruire" la rete attraverso un processo di *training*.

É essenziale quindi per questa fase avere dei *training data*, ovvero un insieme di input di cui è noto a priori il loro output.

In pratica l'output della rete viene confrontato con un output noto e si correggono i valori dei pesi in base alla differenza tra i due output. Il processo di *training* verrà approfondito alla Sezione 4.4, ma può essere riassunto nei seguenti passaggi:

1. Inizializzazione dei pesi.
2. Calcolo dell'output $y_i$ della rete, utilizzando un input di cui si conosce la soluzione $d_i$, e calcolo dell'errore tra i due output:

$$e_i = d_i - y_i$$

3. Calcolo della variazione del peso nel seguente modo (*delta rule*):

$$\delta_i = \varphi'(\nu_i)e_i$$
$$\Delta w_{ij} = \alpha\delta_i x_j$$

   dove $\alpha$ è chiamato ***learning rate***.

4. Aggiornamento del peso:

$$w_{ij} = w_{ij} + \Delta w_{ij}$$

5. Ripetizione dei passaggi da 2-4 su tutti i *training dataset*.
6. Ripetizione dei passaggi da 2-5, finché l'errore non raggiunge una tolleranza accettabile.

Questo tipo di rete neurale, nonostante le grandi potenzialità e la semplicità d'implementazione, presenta un grande limite: ha scarse *performance* con dati che non sono linearmente separabili, ovvero quei dati il cui confine non può essere tracciato da una semplice linea retta, come mostrato in Figura 3.4.

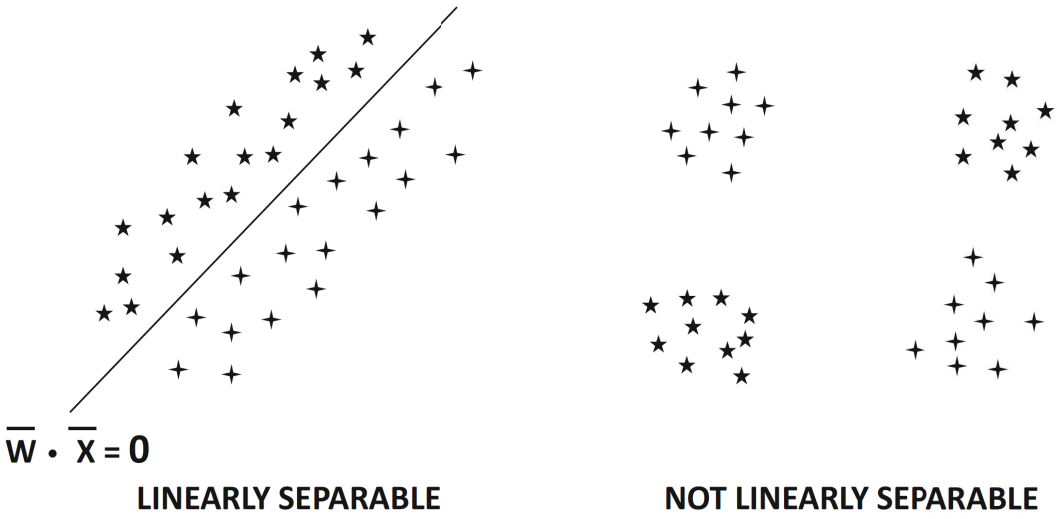



Figura 3.4: Esempio di dati linearmente separabili e il contrario

Infatti in [36] si dimostra che il "Perceptron" converge sempre fornendo un errore nullo per ogni *training data* se questi sono linearmente separabili. La convergenza invece non è garantita se i dati non sono linearmente separabili e addirittura la rete neurale alla fine della fase di *training* fornisce scarse prestazioni.

### 3.1.2 Rete Neurale Multi-Layer

L'unico modo per superare il problema sopra descritto è aumentare il numero di *layers* della rete creando in questa maniera una rete *Multi-Layer*, come mostrato in Figura 3.5. Così, oltre ad avere i due *layers* della rete precedente, questo tipo di rete ne ha un terzo chiamato **hidden layer**.

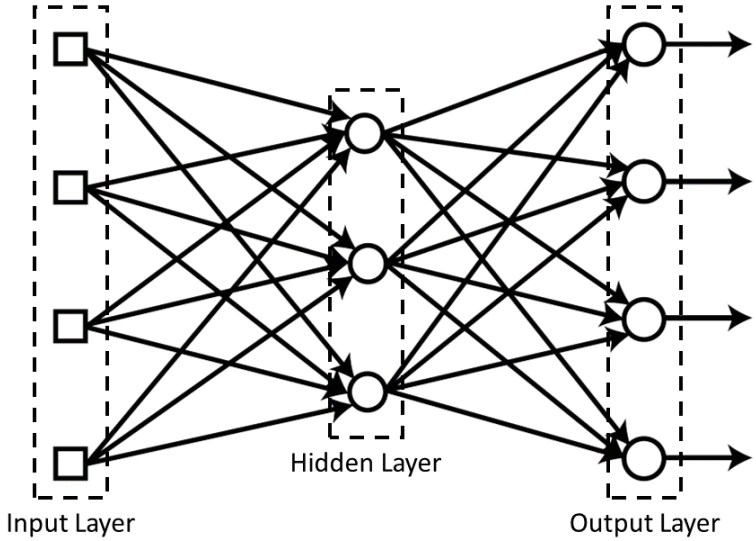



Figura 3.5: Multi-Layer Neural Network

In questa maniera, però, si presenta una nuova problematica: l'*hidden layer* non ha la possibilità di confrontare il proprio output con uno noto a priori, quindi non ha un termine di paragone per poter aggiornare i pesi tra i nodi del proprio *layer* e quelli dell'*input layer*.

Per risolvere questo problema e per riuscire ad aggiungere un solo strato nell'*hidden layer* ci sono voluti circa trent'anni, quando nel 1986 venne studiato l'algoritmo di retro-propagazione [37]. Il funzionamento dell'algoritmo può essere descritto tramite i seguenti passaggi:

1. Inizializzazione dei pesi.

2. Calcolo dell'output $y$ della rete, utilizzando un input di cui si conosce la soluzione $d$, e calcolo dell'errore tra i due output e di $\delta$:

$$e = d - y$$
$$\delta = \varphi'(\nu)e$$

3. Propagazione all'indietro di $\delta$ e calcolo del $\delta^{(k)}$ del k-esimo strato di *hidden layer*.

$$e^{(k)} = [\mathbf{W}]^T_{(k)} \delta^{(k+1)}$$
$$\delta^{(k)} = \varphi' \left(\nu^{(k)}\right) e^{(k)}$$

4. Ripetizione del passaggio 3 per tutti i *layer* dell'*hidden layer*.

5. Aggiornamento di pesi che collegano tutti i *layers*:

$$\Delta w_{ij} = \alpha \delta_i x_j$$
$$w_{ij} = w_{ij} + \Delta w_{ij}$$

6. Ripetizione dei passaggi da 2-5 su tutti i *training dataset*.

7. Ripetizione dei passaggi da 2-6, finché l'errore non raggiunge una tolleranza accettabile.

L'unica vera differenza dal metodo descritto in precedenza è l'aggiunta del terzo passaggio: infatti l'errore che va a ad aggiornare i pesi dei *layers* precedenti all'*output layer* è la retropropagazione dell'errore di output all'interno della rete.

## 3.2 Deep Neural Network

Quando si parla di Deep Neural Network s'intendono in generale le reti neurali *Multi-Layer* con due o più strati all'interno dell'*hidden layer*, come mostrato in Figura 3.6.

I problemi che si presentano con un numero di *hidden layer* superiore a uno utilizzando l'algoritmo di retropropagazione proposto sono principalmente tre:

- **Vanishing Gradient**: il gradiente è concettualmente simile al $\delta$ dell'algoritmo di retropropagazione. Il problema è che l'errore a mano a mano che avanza verso l'*input layer* decresce sempre più, fino a diventare così insignificante da non influire sul cambiamento dei pesi dei primi *layers*.

- **Overfitting**: la rete neurale tende a specializzarsi troppo sui *training data* all'aumentare dei *layers*. In questo modo la rete diventa di scarso utilizzo se vengono dati input al di fuori di quelli usati per il *training*.

- **Computational load**: Aumentando la complessità della rete, cresce anche il numero di nodi e di pesi che devono essere elaborati. Questo richiede quindi tempi di calcolo molto lunghi.

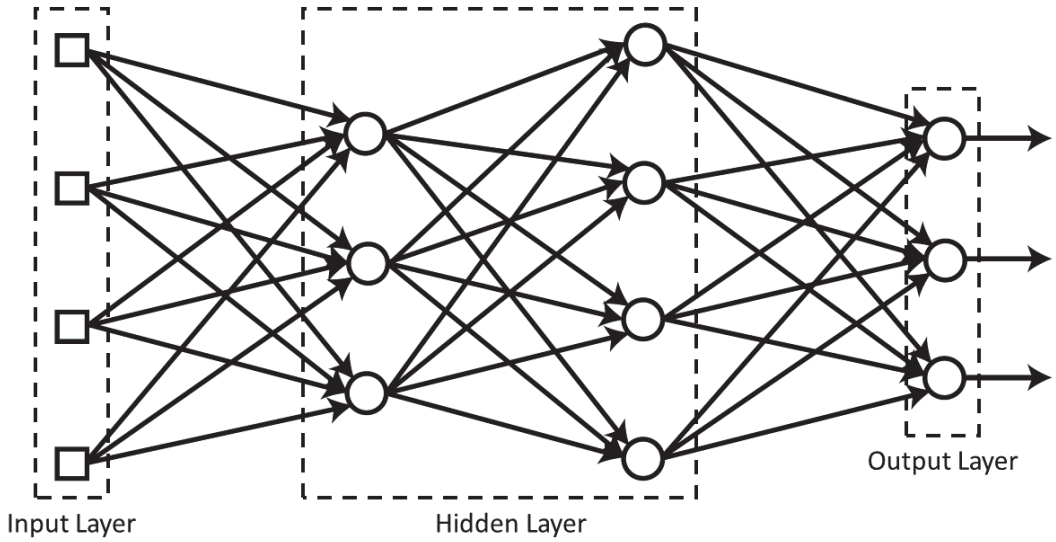



Figura 3.6: Deep Neural Network

Per poter risolvere questi problemi ci sono voluti ulteriori venti anni, in quanto queste problematiche sembravano così insormontabili da far diminuire l'interesse in questo campo.

Il problema del *Vanishing Gradient* è stato risolto cambiando la funzione di attivazione. Fino ad allora la funzione più utilizzata era proprio il "Sigmoide" (Figura 3.2c).

Cambiando questa funzione con la funzione ReLU (Figura 3.2e) l'errore non svaniva all'avanzare dell'algoritmo verso l'*input layer*.

Per quanto riguarda l'*Overfitting* la soluzione adottata è stato l'utilizzo del ***dropout***, ovvero durante il *training* una percentuale di nodi vengono disattivati casualmente, cioè il loro output viene posto pari a zero.

In questo modo l'effetto dell'*Overfitting* viene prevenuto, in quanto i pesi vengono continuamente cambiati durante l'allenamento, senza che questi convergano ad un valore specifico. Un altro modo per evitare l'*Overfitting* è utilizzare grandi *training dataset* in modo da avere un casistica molto più ampia, ma che allo stesso tempo potrebbe generare una minore accuratezza della rete.

Per quanto riguarda il *Computational load* esso è stato risolto con il miglioramento e lo sviluppo di nuovi hardware appositi per questo tipo di operazioni, come le GPU.

## 3.3 Architettura delle CNN

Le CNN sono delle reti neurali profonde, costruite per elaborare in input delle immagini da cui estrarre delle informazioni. Il modo in cui operano è tale per cui riescono a riconoscere dei *patterns*, delle *textures* o dei contorni che sono caratteristici di un determinato oggetto, riuscendo così distinguerlo dal resto dell'immagine.

La peculiarità di queste reti risiede proprio nella loro struttura, dove gli strati profondi sono costituiti da una successione di tre strati principali:

- strato convoluzionale
- strato di attivazione
- strato di pooling

L'architettura di maggior successo nelle CNN e che tuttora trova largo impiego nel campo della segmentazione semantica, è quella di tipo **Encoder-Decoder**.

Facendo riferimento alla rete SegNet in Figura 3.7 è possibile notare una prima parte dell'*hidden layer* composta da strati di matrici sempre più piccole ("Encoder") e una seconda parte composta da strati di matrici sempre più grandi ("Decoder").

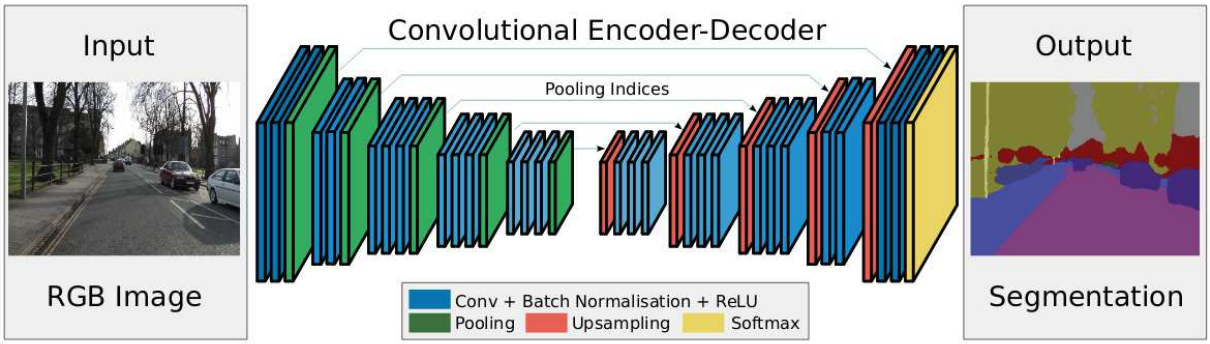



Figura 3.7: Rete neurale SegNet [4]

In generale la struttura di questo tipo di rete si compone di una successione dei tre strati elencati in precedenza nella seguente maniera:

$$Conv. \rightarrow ReLU \rightarrow Pooling \rightarrow ... \rightarrow Unpooling \rightarrow ReLU \rightarrow Deconv.$$

La successione dei primi tre strati è parte dell'"Encoder", mentre gli ultimi tre, che altro non sono che l'operazione inversa dei primi, sono parte del "Decoder".

L'architettura della rete **DeepLabv3+**, ovvero la CNN utilizzata in questo elaborato, presenta proprio questo il tipo di configurazione Encoder-Decoder, ma verrà illustrata con maggior dettaglio alla Sezione 4.4.

Fatta dunque questa premessa sulle CNN è possibile fare un'analisi più dettagliata del funzionamento dei tre strati elencati precedentemente.

### 3.3.1 Strato convoluzionale

Lo strato convoluzionale è l'operazione principale delle CNN, infatti esso è costituito da un filtro che viene applicato ad ogni pixel dell'immagine ed infine computa il prodotto scalare delle due matrici (immagine e filtro). Questa operazione si chiama convoluzione.

Quanto appena descritto è chiaramente illustrato in Figura 3.8 dove s'intuisce l'azione del filtro sull'immagine. Ogni filtro è caratterizzato da quattro **iperparametri** (*hyperparameters*): Altezza $F_q$, Larghezza $F_q$, Profondità $d$ e Numero $q$.

I primi due generalmente sono uguali, in modo da avere dei filtri quadrati. Di solito questi due numeri sono dispari e di piccole dimensioni (3 o 5). Il terzo parametro è strettamente legato alla profondità del *layer* di input. Infine il quarto è semplicemente il numero di filtri che si ritiene necessario utilizzare.

In altre parole poiché i filtri mettono in evidenza certe caratteristiche di un'immagine, maggiora sarà il loro numero, maggiori saranno le caratteristiche che la rete potrà rilevare.

È importante sottolineare che i valori dei filtri possono essere modificati durante il processo di *training*, quindi non è necessario preimpostare dei valori.

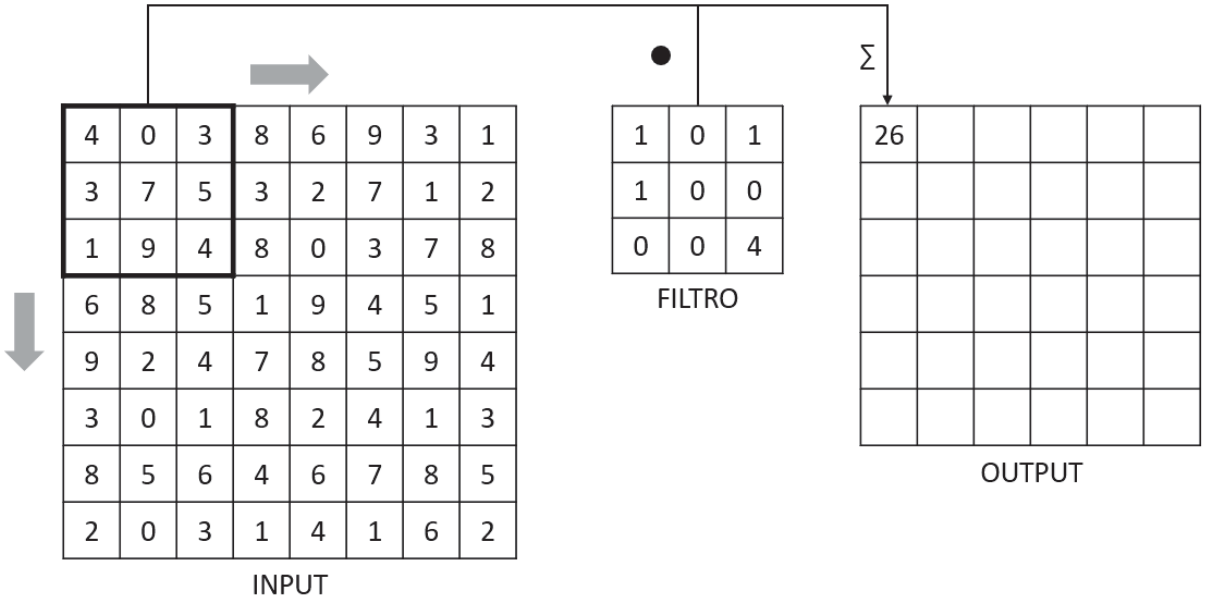



Figura 3.8: Esempio funzionamento dello strato convoluzionale [5]

L'output dello strato convoluzionale è una nuova immagine dove le dimensioni dipendono dall'immagine in input. Prendendo come riferimento la Figura 3.8, se l'input è di dimensioni $L_q$x$B_q$ = 8x8 ed il filtro $F_q$x$F_q$ = 3x3 ne risulta che l'immagine in output ha dimensioni

$$L_{q+1} = L_q - F_q + 1 = 6$$
$$B_{q+1} = B_q - F_q + 1 = 6$$

Oltre agli iperparametri ogni filtro è caratterizzato da un certo bias indicato con $b(q, p)$ con $q$ il numero del filtro del layer $p$.

#### Padding

L'operazione di convoluzione, come si può intuire dall'esempio precedente, riduce le dimensioni dello strato $q$ rispetto allo strato $q+1$. La conseguenza è una perdita di informazioni soprattutto sui bordi dell'immagine.

Per arginare il problema, si utilizza il *padding* ovvero si aggiunge all'immagine una cornice di pixel di valore nullo e di spessore $C_q = (F_q - 1)/2$ oppure $C_q = F_q/2$ a seconda che $F_q$ sia dispari o pari rispettivamente.

Questa tipologia di *padding* è definita "half-padding" poiché circa metà del filtro oltrepassa il bordo dell'immagine, quando collocato all'estremità.

#### Stride

Per velocizzare l'operazione di convoluzione, ma anche per poter catturare *patterns* più complessi in vaste porzioni dell'immagine si utilizza l'operazione di *stride*.

In pratica la convoluzione viene applicata saltando un numero $S$ di pixel, anziché ciascun pixel successivo. Le dimensioni dell'output saranno quindi

$$L_{q+1} = \frac{L_q - F_q}{S} + 1$$
$$B_{q+1} = \frac{B_q - F_q}{S} + 1$$

Ovviamente la frazione dovrà essere un numero intero. Solitamente si usano valori di $S$ pari a 1 o 2, valori più alti vengono utilizzati quando si hanno a disposizione spazi di memoria ridotti. L'utilizzo dell'operazione di *stride* permette una riduzione di un fattore circa $1/S$ delle dimensioni dell'immagine e di $1/S^2$ dell'area.

### 3.3.2 Strato di attivazione

Lo strato successivo è quello di attivazione in cui ciascun valore dell'output passa attraverso una funzione di attivazione non-lineare. Ne risulta un output delle stesse dimensioni del precedente.

Per la funzione di attivazione è molto utilizzata la funzione ReLU (**Re**ctified **L**inear **U**nit) per le ragioni spiegate già nella Sezione 3.2 e perché è computazionalmente molto più efficiente. ReLU è definita come segue:

$$ReLU = f(x) = max(x, 0) \tag{3.11}$$

### 3.3.3 Strato di pooling

Lo strato di pooling ed in particolare quello di max-pooling estrae il massimo valore contenuto all'interno di una matrice $P$x$P$ di ogni mappa di attivazione, producendo un altro *layer* di eguale profondità.

Anche in questo caso, come per la convoluzione, si fa uso dell'operazione di *stride* riducendo in questo modo le dimensioni dell'immagine finale.

Generalmente si utilizzano strati di dimensioni 2x2 e *stride* di 2, come illustrato in Figura 3.9, per evitare sovrapposizioni.

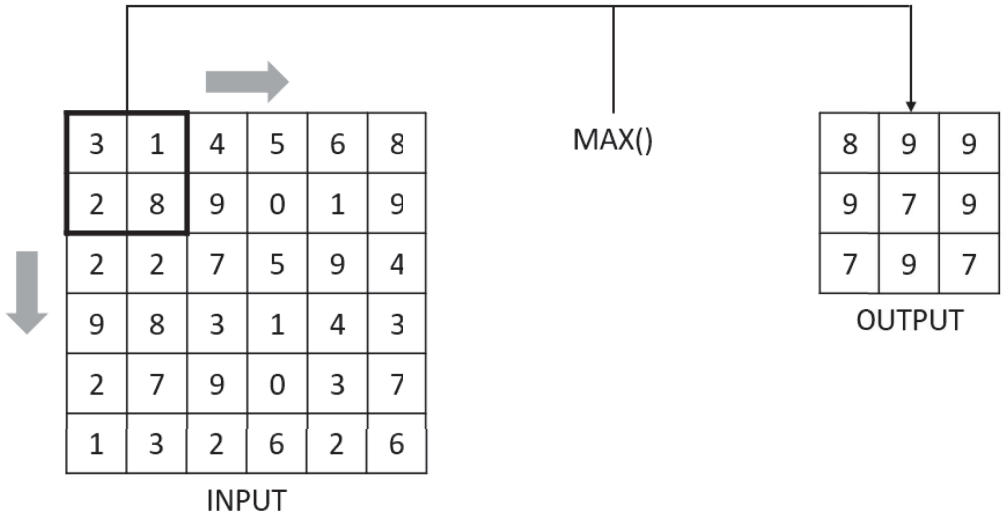



Figura 3.9: Esempio funzionamento dello strato di pooling [5]

Prendendo quindi come esempio una mappa di attivazione di dimensioni $L_q$x$B_q$ = 6x6, uno strato di pooling di dimensioni $P$x$P$ = 2x2 e *stride* $S = 2$, le dimensioni finali saranno:

$$L_{q+1} = \frac{L_q - P}{S} + 1 = 3$$
$$B_{q+1} = \frac{B_q - P}{S} + 1 = 3$$

L'utilizzo dell'operazione di *stride* nello strato di pooling è importante soprattutto per tre aspetti molto rilevanti:

- riduce l'impronta spaziale della mappa di attivazione, quindi si riducono i costi computazionali per i *layers* successivi. Proprio per questa sua caratteristica lo stato di polling non viene usato così spesso come i due precedenti, altrimenti si perderebbero troppe informazioni dopo pochi passaggi.
- presenta una certa invarianza alla traslazione. In questo modo oggetti identici che sono posizionati in posizioni diverse riescono ad essere identificati correttamente.
- aumenta il campo ricettivo, ovvero mette in evidenza caratteristiche complesse di grandi regioni.

Bisogna comunque sottolineare che negli ultimi anni si tende ad eliminare sempre più lo strato di *pooling* a favore dello strato convoluzionale che può svolgere lo stesso compito, ma in maniera molto meno incisiva.

Tuttavia lo strato di pooling presenta alcuni vantaggi, come la forte non-linearità e l'invarianza alla traslazione, pertanto non è stato del tutto abbandonato [3].

### 3.3.4 Strato completamente connesso

Infine, anche se non sempre presente in tutte le CNN, vi è lo strato completamente connesso, che mette in relazione gli output dello strato di pooling con gli output finali della CNN attraverso una semplice DNN.

Ovviamente questo tipo di rete aumenta a dismisura il numero di parametri da addestrare. In alcune reti però ciò è necessario, come ad esempio per LeNet-5 [38] una delle prime CNN ad essere state sviluppate, la cui struttura è riportata in Figura 3.10.

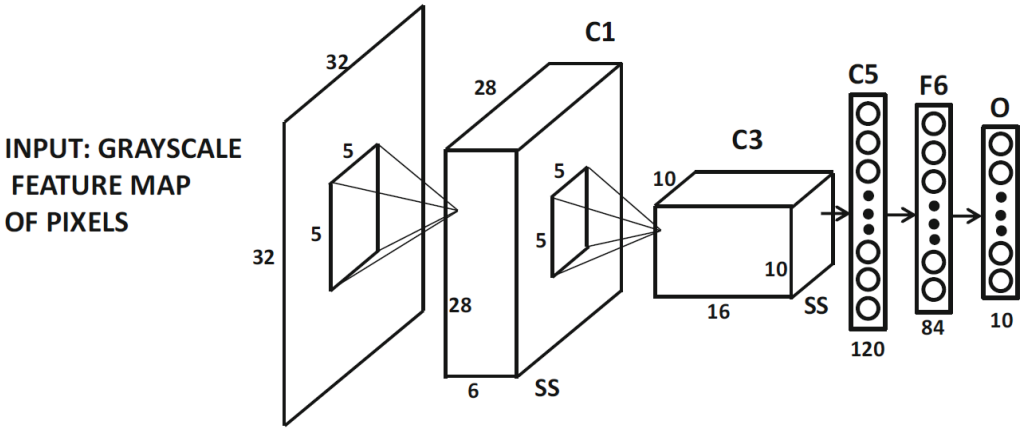



Figura 3.10: Struttura della rete LeNet-5

# Capitolo 4

# Implementazione della rete neurale

Per l'implementazione della rete neurale è stato utilizzato il programma MATLAB, che tramite l'utilizzo di appositi script e app ha permesso di fare i seguenti passaggi:

- Preparazione dello spazio di lavoro
- Labeling
- Data augmentation
- Training
- Testing
- Compilazione in linguaggio CUDA

La rete neurale svolge la funzione di riconoscere all'interno di un'immagine diverse tipologie di terreno. In questo modo il rover costruisce una mappa di cui tiene in considerazione soltanto la tipologia di terreno ritenuta idonea per essere attraversata.

Per quanto il rover sia pensato per applicazioni di tipo spaziale, come ad esempio l'esplorazione planetaria, dal momento che l'algoritmo è stato testato in ambiente terrestre, la rete neurale si basa su un *dataset* d'immagini, raccolte sul luogo in cui sono stati svolti i test.

Questo, per quanto abbia semplificato in maniera considerevole il lavoro, non esclude che l'algoritmo sviluppato sia valido solo per applicazioni terrestri. Infatti, utilizzando un *dataset* e classificazioni del terreno non necessariamente terrestri, non cambia l'algoritmo, ma solo la rete neurale.

## 4.1 Preparazione dello spazio di lavoro

Per gestire meglio il processo di implementazione della rete neurale sono stati creati degli script in codice MATLAB come illustrato nella Figura 4.1

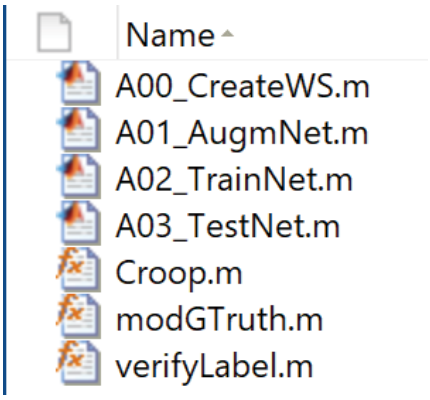



Figura 4.1: Elenco file MATLAB per L'implementazione della rete neurale

Il primo script "**A00_CreateWS.m**" (Appendice B.1) serve alla creazione dello spazio di lavoro, cioè vengono create delle cartelle specifiche per i successivi passaggi.

## 4.2 Labeling

Come motivato precedentemente, il *dataset* per la fase di *Training* è stato acquisito nel luogo in cui si sono svolti i test. In particolare le categorie scelte sono state: terreno asfaltato (*path*), terreno erboso (*grass*), vegetazione (*tree*) e "tutto-il-resto" (*other*).

La prima categoria è la più importante: l'algoritmo associa proprio a questo tipo di terreno la condizione di attraversabilità, infatti il rover costruisce la propria traiettoria solo attraversando un terreno asfaltato.

La seconda e la terza categoria sono state utilizzate per distinguere il terreno erboso dalla restante vegetazione, che potrebbe essere confusa come erba dalla rete neurale, come ad esempio i cespugli o le chiome degli alberi. Questa distinzione è utile soprattutto nel caso in cui si volesse provare ad utilizzare il rover solo sul terreno erboso invece che su terreno asfaltato.

Infine l'ultima categoria non ha uno scopo preciso, se non quello di assegnarne una a tutto ciò che non rientra in quelle precedenti. Infatti se ciò non fosse fatto, la rete neurale assegnerebbe lo stesso una categoria ai pixel dell'immagine, aumentando il rischi di avere delle associazioni errate.

Per fare l'operazione di *Labeling* sono stati fatti due passaggi: nel primo sono state utilizzate 49 immagini per creare una prima rete neurale; nel secondo sono state utilizzate 169 immagini le quali sono state pre-etichettate, utilizzando la rete neurale precedente e facendo in seguito alcune correzioni manuali.

Il secondo passaggio si è reso necessario per aumentare la robustezza della rete neurale stessa. Per far ciò, a differenza del primo passaggio, si sono utilizzate delle immagini con condizioni meteo differenti (soleggiato e nuvoloso). Bisogna però aggiungere che le correzioni manuali sono state fatte in modo più grossolano perché, per gli scopi di questo

elaborato, ciò era più che sufficiente e avrebbe determinato un eccessivo allungamento dei tempi.

MATLAB permette di fare questa operazione tramite l'app **Image Labeler**. Si sono così create le quattro categorie, ognuna del tipo **pixel label**. In questo modo è possibile assegnare ad ogni singolo pixel dell'immagine una delle quattro categorie, realizzando la cosiddetta *Pixel Segmentation*.

A mano a mano che si creano le categorie all'interno delle immagini, l'app crea automaticamente una cartella in cui vengono salvate delle immagini "livello", ovvero delle immagini dove a ciascun pixel viene assegnato il numero di una delle quattro categorie.

In ultimo l'associazione di questi dati viene salvata all'interno di una variabile di tipo **Ground Truth**, nominata *gTruthOrig*. Di seguito si riportano l'immagine dello spazio di lavoro alla fine di questa a fase e alcune immagini rappresentative dell'utilizzo dell'app *Image Labeler*.

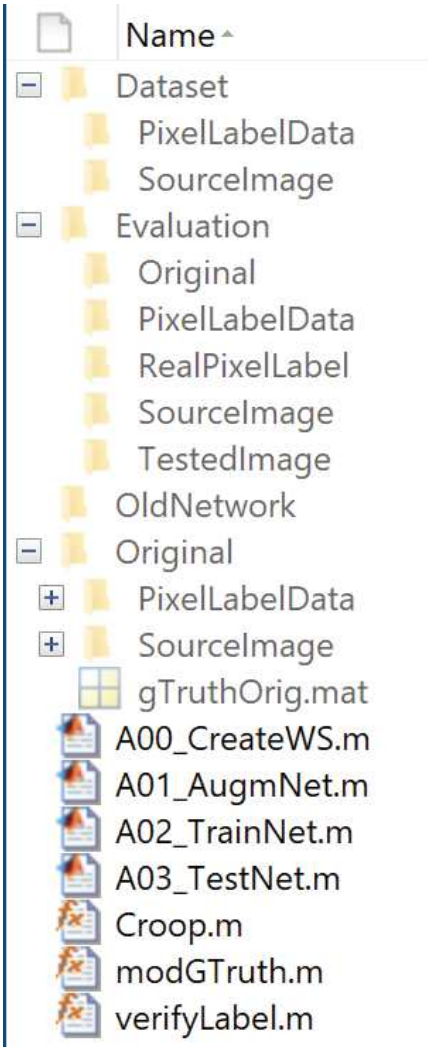
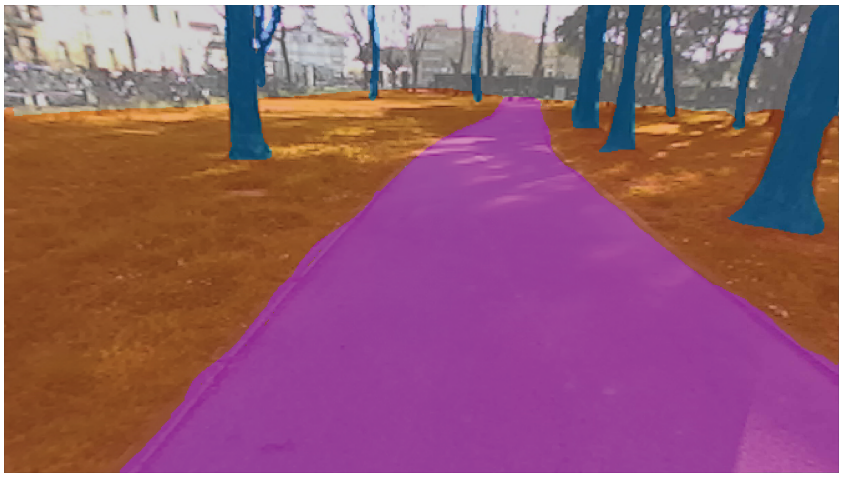
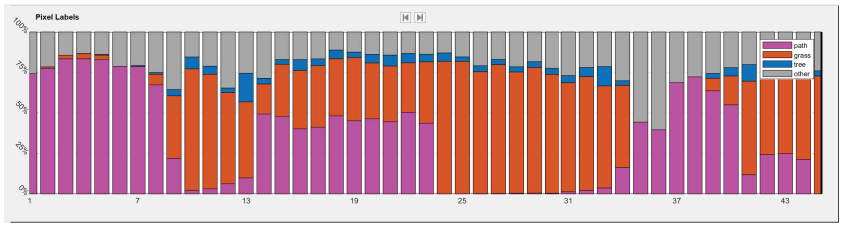



Figura 4.2: Spazio di lavoro alla fine del *Labeling*

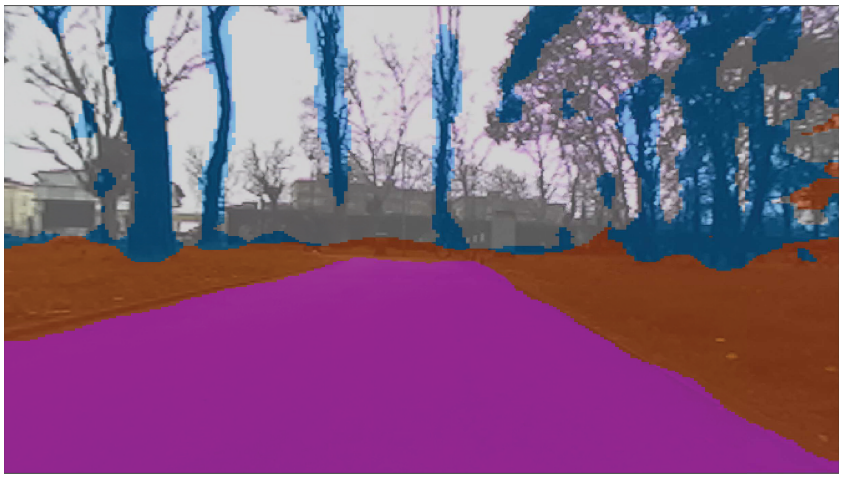


(a)

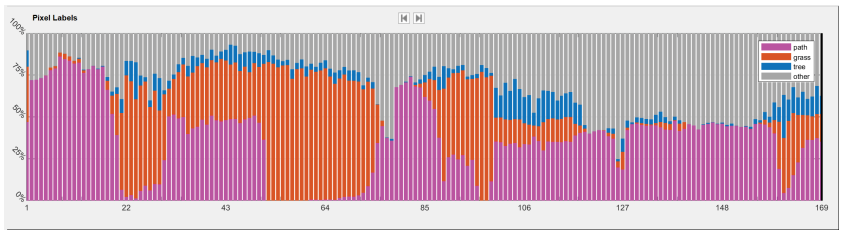



(b)

Figura 4.3: Primo passaggio: (a) immagine rappresentativa, (b) distribuzione delle categorie per ognuna delle 49 immagini del *dataset*



(a)




(b)

Figura 4.4: Secondo passaggio: (a) immagine rappresentativa, (b) distribuzione delle categorie per ognuna delle 169 immagini del *dataset*

L'operazione di *Labeling* manuale richiede molto tempo, in particolare se il numero d'immagini supera la cinquantina. Dal momento che la bontà di una rete neurale è caratterizzata dalla capacità di generalizzare, ovvero riuscire a riconoscere una determinata categoria all'interno di immagini diverse, è stato necessario utilizzare delle tecniche di **data augmentation** per riuscire ad avere un *dataset* appropriato.

Utilizzare un numero ridotto d'immagini porta la rete ad all'allenarsi su un numero limitato di caratteristiche delle categorie e quindi a non riconoscere eventuali cambiamenti di queste ultime.

## 4.3 Data augmentation

In questa fase le immagini vengono volutamente alterate, in modo da generare un *dataset* più ampio e rendere così la rete più robusta e accurata. Molte di queste trasformazioni, come traslazione, rotazione, riflessione, pur cambiando l'immagine non alterano le proprietà sostanziali degli oggetti rappresentati [35]. Inoltre molte di queste trasformazioni non richiedono elevata potenza computazionale. Di seguito vengono riportate alcune di queste trasformazioni [35]:

- **Smoothing:** Con questa operazione si vuole simulare l'effetto generato su immagini sfuocate, utilizzando dei filtri gaussiani. Il vantaggio è un leggero incremento della tolleranza degli effetti di *blurring*.
- **Blurring:** È l'effetto che si può ottenere da una fotocamera in movimento. In base alla ISO e velocità della fotocamera possono generarsi delle immagini degradate. Nel caso in esame questo effetto non è stato considerato proprio perché il robot si muove a velocità ridotte.
- **Sharpening:** Produce immagini dai contorni più nitidi e definiti. L'effetto si ottiene sottraendo all'immagine originaria quella sottoposta ad un effetto di *smoothing*.
- **Cropping:** L'immagine viene suddivisa in immagini più piccole.
- **Hue/Saturation/Value Changes** Nello spazio HSV le immagini sono caratterizzate da tre parametri: contrasto (*Hue*), saturazione (*Saturation*) e luminosità (*Value*). Variando questi valori la rete neurale viene allenata a riconoscere le stesse caratteristiche in condizioni di luminosità diverse. È da notare che una variazione troppo elevata del contrasto non sempre porta a dei benefici, perché l'immagine potrebbe risultare con colori troppo alterati e quindi non conforme alla realtà.
- **Resizing:** L'immagine viene ridimensionata, per simulare l'effetto della distanza. In questo modo la rete neurale può riconoscere più facilmente una stessa categoria, sia che essa sia lontana che vicina.
- **Mirroring:** L'immagine viene specchiata. In questo modo durante l'allenamento non prevale una direzione preferenziale della categoria.

Il secondo script "**A01_AugmNet.m**" (Appendice B.2) ha lo scopo di applicare alcune delle trasformazioni sopra elencate, in particolare le combinazioni: **cropping**, **cropping-mirroring**, **cropping-HSV** e **cropping-mirroring-HSV**.

La scelta è ricaduta su queste trasformazioni per la loro utilità e relativa semplicità d'implementazione. Per il *cropping* è stato deciso di ritagliare l'immagine in 6 quadrati, in quanto era il numero minimo che occorrevano per questo tipo di operazione.

Per l' *HSV*, invece, è stata utilizzata un'apposita function di MATLAB che cambia i tre parametri dell'immagine di un valore random. Infine, qualsiasi sia il tipo di trasformazione, le immagini sono state ridimensionate a una dimensione di [224x224x3], ovvero quelle dell'input richiesto dalla rete neurale.

Di seguito vengono riportate in Figura 4.5 tutte le trasformazioni fatte ad un'immagine del *dataset* originale.

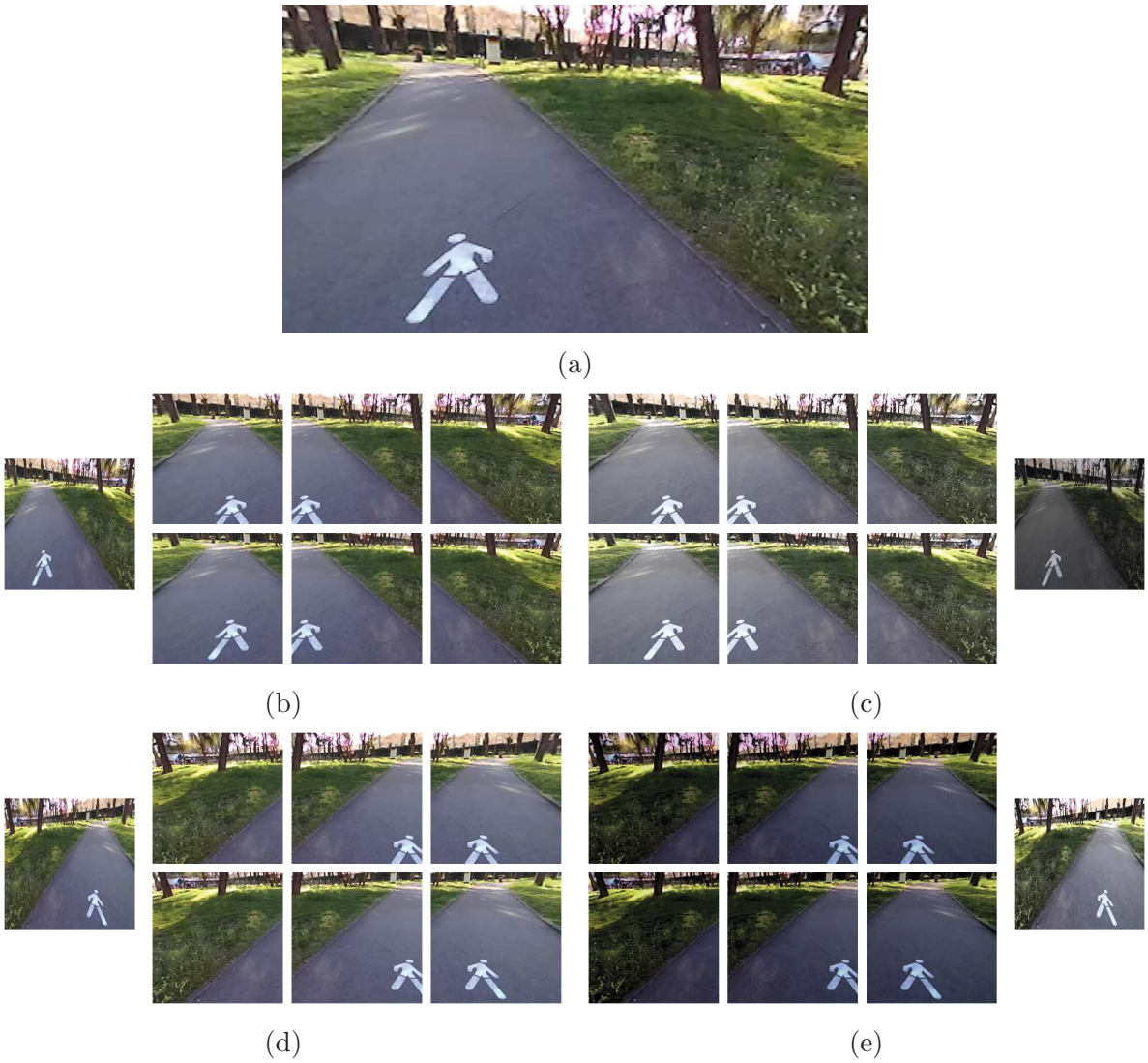



Figura 4.5: Trasformazioni ottenute da un'immagine: (a) immagine originale, (b) *cropping*, (c) *cropping-HSV*, (d) *cropping-mirroring*, (e) *cropping-mirroring-HSV*

Da una sola immagine ne vengono restituite 28, portando il *dataset* da 169 immagini iniziali a 4732, numero sufficiente per gli scopi di questo elaborato.

Le immagini vengono salvate all'interno della cartella *"Dataset\SourceImage"*, mentre le loro rispettive immagini "livello" vengono salvate nella cartella *"Dataset\PixelLabelData"*. Lo script infine genera due file, *Num.mat* e *gTruth.mat*, dei quali il primo serve per operazioni interne allo script stesso, il secondo contiene la variabile di tipo *"Ground Truth"* che serve per lo script successivo e a contenere tutte le informazioni utili del *dataset*.

## 4.4 Training

In questa fase avviene l'allenamento vero e proprio della rete neurale. Il terzo script "**A02_TrainNet.m**" (Appendice B.3) ha lo scopo di semplificare il lavoro da svolgere. Infatti, impostando alcuni parametri (spiegati in dettaglio di seguito), lo script esegue le tre parti principali di questa fase:

- **Data setup**: definizione del *dataset* e successivo partizionamento.
- **Network setup**: generazione dei *layers* della rete e definizione delle dimensioni dell'immagine di input e del numero di categorie.
- **Network training**: definizione di ulteriori opzioni per l'allenamento della rete.

### 4.4.1 Data setup

In questa fase il *dataset* di riferimento è proprio quello contenuto nella cartella *"Dataset"*. Le immagini vengono suddivise in tre gruppi: *training*, *validation* e *testing*. I primi due gruppi servono esclusivamente all'allenamento della rete neurale, ovvero la rete utilizza il primo per allenarsi e il secondo per validare l'accuratezza dell'allenamento stesso.

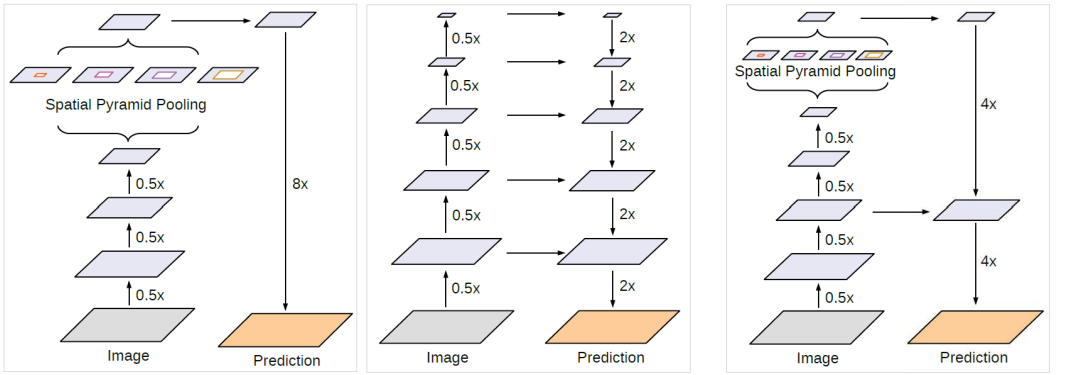
Infine il terzo viene utilizzato come riferimento per misurare l'accuratezza reale del *dataset*. Infatti siccome la rete non utilizza precedentemente queste immagini, in questo modo è possibile ottenere dei valori più realistici.

La proporzione con cui si è diviso il *dataset* nei tre gruppi è stata di 70%-15%-15%. Con questi valori si dovrebbe evitare maggiormente il rischio di sovradattamento della rete.

### 4.4.2 Network setup

Il passo successivo è quello di caricare i *layers* della rete neurale **DeepLabv3+** tramite una funzione apposita di MATLAB ed infine specificare le dimensioni del *layer di input* e del *layer di output/classification layer*.

La rete neurale **DeepLabv3+** è stata scelta perché tra quelle tuttora disponibili in MATLAB è una delle più performanti. Attualmente è la versione più recente sviluppata da Google ed è l'unione di due tipi di architetture principali la ASPP (Atrous Spatial Pyramid Pooling) e l'architettura Encoder-Decoder, come mostrato in Figura 4.6.




(a) Spatial Pyramid Pooling (b) Encoder-Decoder (c) Encoder-Decoder with Atrous Conv

Figura 4.6: (a) Architettura ASPP, (b) Architettura Encoder-Decoder, (c) Architettura di DeepLabv3+ [6]

La prima architettura permette di ridurre la complessità della rete, mentre la seconda fa da scheletro a tutta l'architettura di base, mantenendo alte le prestazioni della rete.

L'implementazione di DeepLabv3+ in MATLAB può adottare strutture di *"Encoder"* differenti. Nel caso in esame è stato scelto ResNet18 per avere tempi di allenamento più veloci, rinunciando tuttavia ad una migliore precisione della rete.

### 4.4.3 Network training

Prima di avviare l'allenamento vero e proprio, vengono impostati alcuni parametri. Il più importante è la scelta dell'algoritmo di retropropagazione, ovvero l'algoritmo che aggiusta i pesi della rete neurale fino a convergere ad una soluzione.

Nel caso in esame sì è adottato l'algoritmo SGDM (Stochastic Gradient Descent with Momentum), che è il più comune. Esso si basa sull'algoritmo SDG [34] che aggiorna i pesi della rete, cercando di minimizzare la funzione di perdita, cioè seguendo la direzione negativa del gradiente. I passaggi dell'SGD sono riportati in Appendice A.2.

Per l'SGDM si aggiunge un termine chiamato **momento** che viene sommato durante l'aggiornamento dei pesi e dipende dal valore del "momento" dell'iterazione precedente.

$$m_k = \alpha\, \delta_i\, x_j + \beta\, m_{k-1}$$
$$w_{ij,k} = w_{ij,k-1} + m_k$$

dove $\beta$ è un numero compreso tra zero e uno. Con l'aggiunta del "momento" si aumenta la stabilità dell'algoritmo e la velocità di convergenza della soluzione durante questa fase.

In generale l'allenamento di una rete neurale segue alcuni passaggi fondamentali, illustrati di seguito:

1. Il *training dataset* viene diviso in ulteriori sottoinsiemi chiamati **Mini Batch**.
2. Ogni "Mini Batch" ha un numero $N$ di immagini e per ognuna di queste viene calcolata la correzione ai pesi della rete.
3. Il calcolo della correzione dei pesi finale viene fatto mediando i valori ottenuti da ogni singola immagine del "Mini Batch".
4. Vengono aggiornati i pesi della rete e si passa al "Mini Batch" successivo. L'insieme dei passaggi 1-4 viene chiamata **Iterazione**.
5. Dopo un certo numero di "Iterazioni" viene eseguito il test di validazione, che verifica l'accuratezza ("**Accuracy**") e la perdita ("**Loss**") della rete.
6. L'allenamento si interrompe se viene raggiunto un valore massimo di perdita oppure un valore minimo di accuratezza oppure un numero massimo di **Epoche**. Un **Epoca** è il numero minimo di "Iterazioni" che occorrono per allenare la rete, utilizzando l'intero *training dataset*.

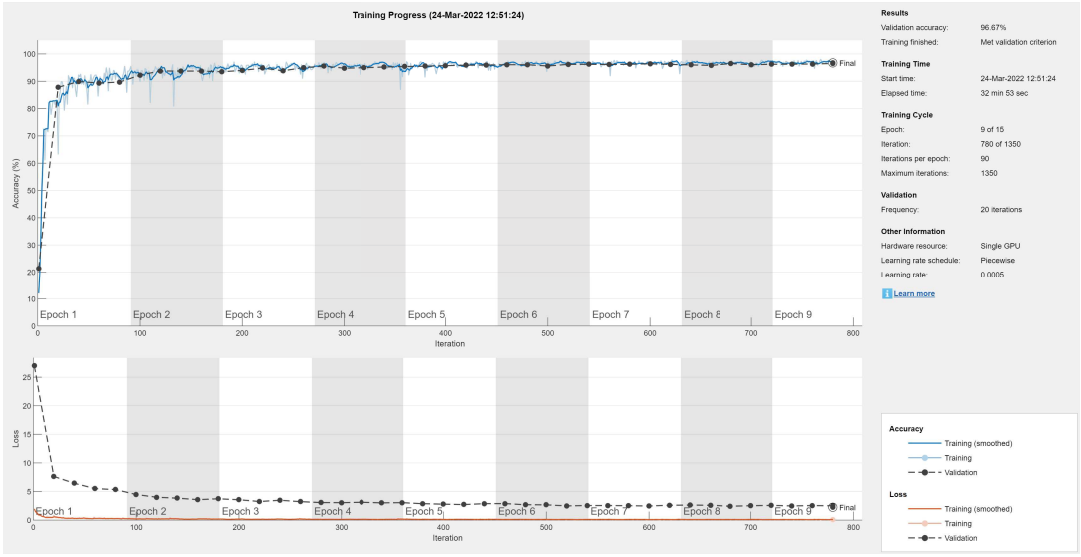
Si riporta in Tabella 4.1 i parametri impiegati per l'algoritmo di retropropagazione e per l'allenamento della rete:

| | **Descrizione** | **Valore** |
|---|---|---|
| **InitialLearnRate** | Il parametro $\alpha$ iniziale del SGDM. Maggiore è il *learning rate* e più velocemente variano i pesi della rete, ma allo stesso tempo è più facile che la soluzione diverga | $10^{-3}$ |
| **MaxEpochs** | Numero massimo di "Epoche" | 20 |
| **MiniBatchSize** | Numero di immagini che compongono un "Mini Batch" | 10 |
| **Shuffle** | Opzione che permette di rimescolare le immagini ad ogni cambio di "Epoca" oppure una volta sola | *every-epoch* |
| **ValidationData** | Insieme delle immagini di validazione | $pximdsVal$ |
| **ValidationFrequency** | Numero di "Iterazioni" dopo cui eseguire il test di validazione | 20 |
| **ValidationPatience** | Numero di volte in cui la "Validation Loss" può essere uguale o maggiore delle precedenti. Se questo valore viene superato l'allenamento s'interrompe | 5 |
| **LearnRateSchedule** | Abilita o meno il cambiamento del *learning rate* iniziale basandosi sui due parametri successivi | *piecewise* |
| **LearnRateDropPeriod** | Numero di Epoche che occorrono a ridurre il *learning rate* | 5 |
| **LearnRateDropFactor** | Fattore di riduzione del *learning rate* | 0.5 |
| **Plots** | Abilita o meno la finestra grafica per visualizzare il procedere dell'allenamento | *training-progress* |

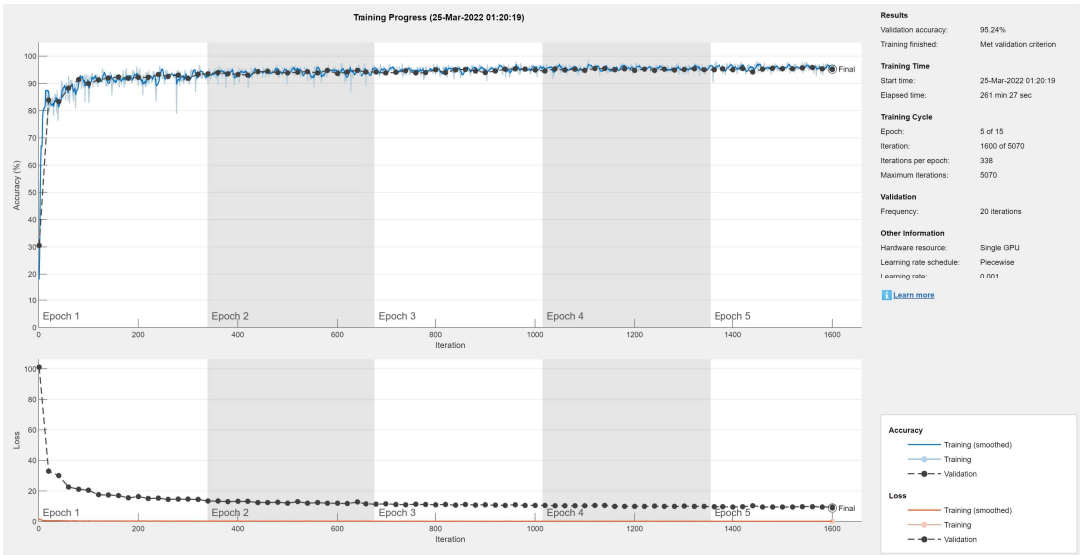
Tabella 4.1: Parametri per l'allenamento della rete neurale

Per quanto riguarda l'hardware sono stati utilizzati una GPU NVIDIA GeForce® GTX 1050 e un processore Intel® Core™ i7-7700HQ 2.80GHz.

Nelle immagini successive vengono mostrati i grafici dell'allenamento delle due reti neurali allenate. Dalla Tabella 4.2 è possibile notare che la prima rete, ovvero quella realizzata a supporto della seconda fase di *Labeling*, ha un'accuratezza maggiore e una perdita minore rispetto alla seconda, ovvero quella utilizzata all'interno dell'algoritmo.




(a) Allenamento della prima rete neurale




(b) Allenamento della seconda rete neurale

Figura 4.7

| | Accuratezza | Perdita | Iterazioni totali |
|---|---|---|---|
| **Prima rete** | 96.67% | 2.5 | 780 |
| **Seconda rete** | 95.24% | 9.5 | 1600 |

Tabella 4.2: Confronto risultati delle reti neurali

## 4.5 Testing

In questa fase le immagini del gruppo di *testing* e le loro rispettive immagine "livello" vengono salvate rispettivamente nelle cartelle *"Evaluation\Original"* e *"Evaluation\RealPixelLabel"*.

Grazie all'ultimo script "**A03_TestNet.m**" (Appendice B.4) è possibile misurare le prestazioni della rete neurale ottenuta. Per poter discutere dei parametri utilizzati [35] è necessario introdurre il concetto di "matrice di confusione".

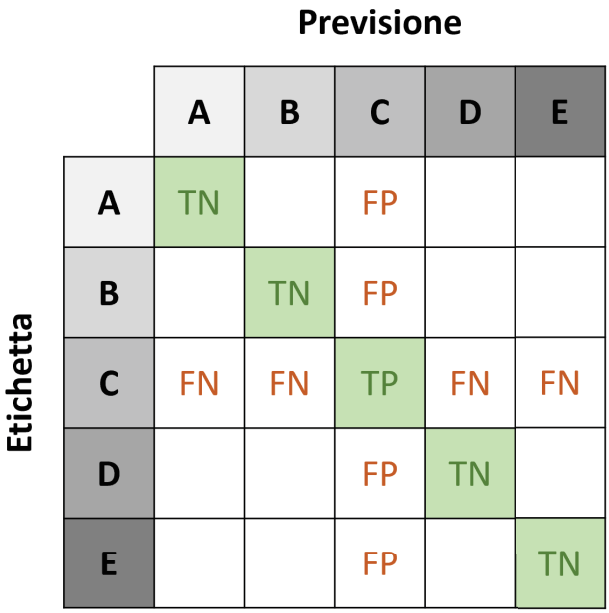



Figura 4.8: Matrice di confusione, in riferimento alla categoria C

In Figura 4.8 è indicata in modo schematico una "matrice di confusione". Facendo riferimento ad una categoria specifica si identificano quattro tipi di informazioni:

- **vero positivo (TP):** il numero di pixel previsti corretti e facenti parte della categoria di riferimento;
- **veri negativi (TN):** il numero di pixel NON previsti corretti e NON facenti parte della categoria di riferimento;
- **falsi negativi (FN):** il numero di pixel NON previsti corretti, ma facenti parte della categoria di riferimento;
- **falsi positivi (FP):** il numero di pixel previsti corretti, ma NON facenti parte della categoria di riferimento;

Dalla "matrice di confusione" si possono ottenere i parametri più interessanti per la valutazione della rete. Di seguito ne vengono elencati alcuni.

**ACCURATEZZA** Il modo più semplice per valutare una rete neurale è proprio quello di misurare quanti pixel vengono identificati correttamente sul totale di pixel etichettati. In altre parole, indicata con $\mathcal{M}$ la "matrice di confusione" si ha che:

$$Accuratezza = \frac{\sum_i \mathcal{M}_{ii}}{\sum_i \sum_j \mathcal{M}_{ij}}$$

L'accuratezza per quanto semplice da calcolare non sempre potrebbe fornire dei risultati rappresentativi, soprattutto nel caso in cui ci sia una sproporzione tra le diverse categorie.

**IoU** Il secondo parametro è l'"Intersection over Union" detto anche coefficiente di Jaccard. Per ogni categoria è definito come:

$$IoU_i = \frac{\mathcal{M}_{ii}}{\sum_j (\mathcal{M}_{ij} + \mathcal{M}_{ji}) - \mathcal{M}_{ii}}$$
$$IoU = \sum_i w_i \cdot IoU_i$$

questo parametro è molto simile all'"Accuratezza" con la differenza che penalizza molto di più i "falsi positivi". Proprio per questa ragione si preferisce utilizzare questo indice rispetto al precedente.

**PRECISIONE E RECUPERO** Entrambi sono parametri utili soprattutto per il calcolo del parametro **F1 score**. Entrambi fanno riferimento alle singole categorie, per poi eseguire una media ponderata, dove i pesi sono i rapporti tra i pixel della categoria di riferimento ed il numero totale di pixel. Per ogni categoria questi sono definiti come:

$$Precisione_i = \frac{\mathcal{M}_{ii}}{\sum_j \mathcal{M}_{ji}}$$
$$Recupero_i = \frac{\mathcal{M}_{ii}}{\sum_j \mathcal{M}_{ij}}$$

mentre per l'intero dataset:

$$Precisione = \sum_i w_i \cdot Precisione_i$$
$$Recupero = \sum_i w_i \cdot Recupero_i$$

Rispetto all'"Accuratezza" non c'è il problema della sproporzione tra le classi, in quanto viene fatta proprio una media pesata.

**F1 SCORE** Come accennato precedentemente per il calcolo di "F1 score" si utilizzano la "Precisione" e il "Recupero". Lo scopo è quindi quello di riassumere i due valori in uno unico e per far ciò si calcola la media armonica nel seguente modo:

$$(F1\ score)_i = \frac{2}{\frac{1}{Precisione_i} + \frac{1}{Recupero_i}}$$
$$F1\ score = \sum_i w_i \cdot (F1\ score)_i$$

In generale questo parametro è legato alla bontà del contorno tracciato rispetto al contorno segmentato: più "F1 score" è vicino a 1 e migliore è la capacità della rete di identificare il confine tra due categorie differenti.

Si riportano di seguito i risultati della matrice di confusione ottenuta (Tabella 4.3 e Tabella 4.4) e dei paramatri appena descritti (Tabella 4.5) in riferimento all'ultima rete neurale, perché è quella di maggior interesse.

| | **path** | **grass** | **tree** | **other** |
|---|---|---|---|---|
| **path** | 13424605 | 196253 | 6640 | 124485 |
| **grass** | 55862 | 8447393 | 93373 | 53595 |
| **tree** | 368 | 20798 | 1800082 | 79934 |
| **other** | 54538 | 57163 | 881239 | 10328632 |

Tabella 4.3: Matrice di confusione, numero totale dei pixel

| | **path** | **grass** | **tree** | **other** |
|---|---|---|---|---|
| **path** | 97.62 | 1.43 | 0.04 | 0.91 |
| **grass** | 0.65 | 97.65 | 1.08 | 0.62 |
| **tree** | 0.02 | 1.09 | 94.68 | 4.21 |
| **other** | 0.48 | 0.50 | 7.79 | 91.23 |

Tabella 4.4: Matrice di confusione normalizzata, percentuali

| **Accuratezza** | **IoU medio** | **IoU pesato** | **F1 score medio** | **F1 score pesato** |
|---|---|---|---|---|
| 95.44 | 85.78 | 92.04 | 91.71 | 95.67 |

Tabella 4.5: Percentuali dei parametri della rete neurale

Dalla matrice di confusione si può già notare che la rete neurale riesce ad identificare molto bene il terreno asfaltato, anche se una piccola percentuale viene riconosciuta come terreno erboso.

L'ultima categoria, invece, è quella che viene peggio riconosciuta. Una spiegazione potrebbe essere la enorme generalità di questa classe, che non ha delle caratteristiche ben identificate, come invece un terreno asfaltato oppure erboso.

Fortunatamente le ultime due categorie vengono molto di più confuse tra di loro rispetto alle prime due e questo permette di avere una rete neurale comunque affidabile, in quanto lo scopo principale è proprio quello di identificare il terreno asfaltato.

## 4.6 Compilazione in linguaggio CUDA

L'ultima fase consiste nel compilare l'intera rete neurale in linguaggio CUDA, per poter essere utilizzata dalla GPU rendendo l'applicazione finale molto più veloce. Inoltre viene creato un file DLL (Dinamic-Link Library) per poter essere implementato all'interno di ROS.

La libraria CUDA è stata sviluppata dalla NVIDIA permettendo di evitare la programmazione a basso livello delle proprie GPU. In questo modo molti programmi utilizzano i vantaggi di questa libreria, per applicazioni di *deep learning*.

Per quanto riguarda MATLAB esiste un'app apposita chiamata **GPU Coder** che permette di compilare in maniera semplice l'intera rete neurale in linguaggio CUDA.

Per il suo utilizzo bisogna preparare due script: il primo è una function (Appendice B.5) che ha in input un'immagine 224x224x3 e restituisce in output un'immagine 224x224x1, dove i pixel corrispondono al numero della categoria predetta; il secondo script, invece, serve ad inserire automaticamente le dimensioni e il tipo di input/output, ma quest'ultimo non è essenziale dal momento che può essere compilato manualmente.

Per utilizzare la libreria all'interno del computer, come in questo caso, è stato necessario installare anche alcune librerie come CUDA-toolkit 11.2 e CUDnn, senza le quali la rete non potrebbe funzionare.

Il risultato finale è una cartella con all'interno tutti i file necessari all'implementazione e in particolare il file "AutonavNet.so", cioè il file DLL creato per poter impiegare la rete neurale all'interno del nodo della rete ROS.

# Capitolo 5

# Implementazione della rete ROS

In questo capitolo si vuole descrivere i passaggi principali che sono stati realizzati per implementare la parte software dell'intero apparato sperimentale. In particolare grazie all'utilizzo di ROS (Robot Operating System) è stato possibile integrare tra di loro i diversi sensori e attuatori in dotazione al rover.

ROS è un insieme di *software libraries* e di strumenti utili a realizzare delle applicazioni robotiche [39]. Oltre al fatto che esso è totalmente *open source* e *multi-platform*, il vantaggio principale è che viene utilizzato nella gran parte delle applicazioni robotiche, grazie anche al supporto di una comunità molto ampia.

In linea di principio l'insieme dei processi svolti dai diversi pacchetti di ROS possono essere rappresentati da un grafo, dove i nodi sono gli elementi di elaborazione e i "topic/-server" sono gli elementi che gestiscono gli scambi d'informazione tra i nodi tramite dei messaggi, come rappresentato in Figura 5.1

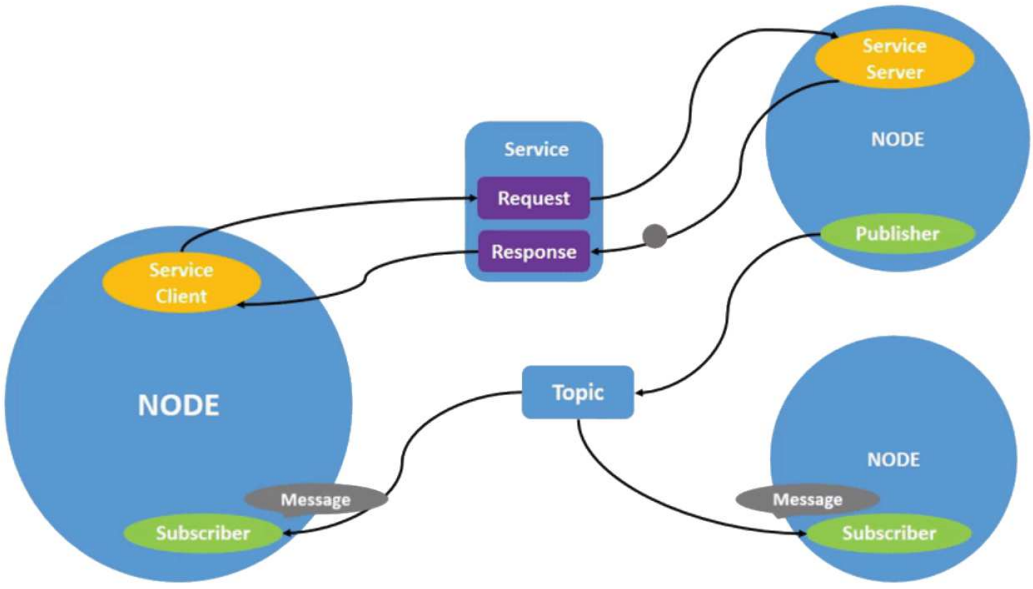



Figura 5.1: Rappresentazione di una rete ROS [7]

I nodi vengono realizzati principalmente in linguaggio C++ oppure Python, all'interno dei quali vengono incluse le librerie scaricate dai diversi pacchetti disponibili. Nel caso considerato da questo elaborato i pacchetti utilizzati sono stati: **zed_wrapper_ros** per l'utilizzo della stereo camera; **rtabmap_ros** per l'implementazione della SLAM; **octomap_server** per la generazione della mappa data una *point cloud*; **move_base** per il controllo della navigazione autonoma; **ethz_piksi_ros** per l'utilizzo del ricevitore GPS.

Mentre i pacchetti precedenti sono stati sviluppati da terzi e possono essere reperiti facilmente dalla Wiki di ROS, i seguenti pacchetti sono stati sviluppati appositamente per il rover, ovvero **locomotion** per il controllo dei motori e **autonav_ai** per gestire la rete neurale in esame.

Nella Figura 5.2 è rappresentata in modo schematico la rete ROS realizzata: in rosso sono indicati gli strumenti, in blu i nodi relativi ai pacchetti sopra elencati e in verde i messaggi che i nodi si scambiano tra loro.

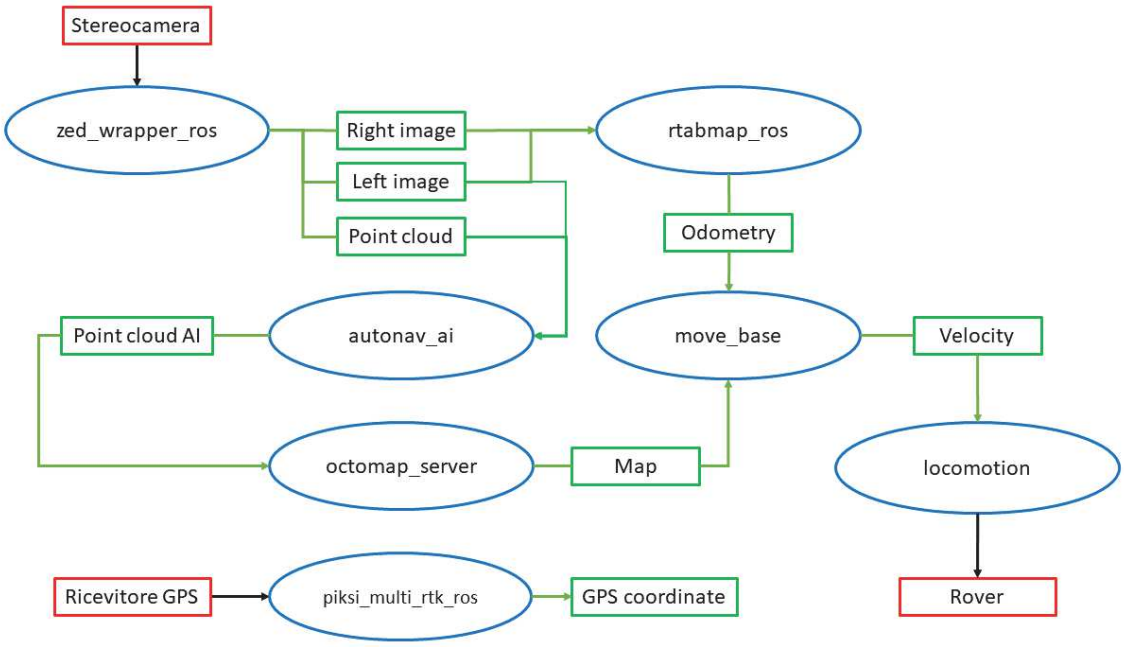



Figura 5.2: Rappresentazione della rete ROS finale

In seguito verranno riportati maggiori dettagli su tutti i pacchetti utilizzati e in particolare ci si soffermerà su quello appositamente creato per l'utilizzo della rete neurale.

## 5.1 Pacchetto zed_wrapper_ros

Il pacchetto [40] è stato sviluppato dalla StereoLab per poter integrare all'interno di ROS l'utilizzo delle proprie stereocamere ZED.

quiqui

Per poter funzionare deve essere installato l'SDK fornito dalla StereoLab stessa, il quale necessita che il computer sia dotato di una GPU della NVIDIA, per poter utilizzare la libreria CUDA.

Tra i "topic" di maggior interesse che il nodo restituisce ci sono la *point cloud*, la *depth image*, le immagini corrette dalle distorsioni (destra e sinistra), la *odometry*, i sistemi di riferimento dei due sensori delle fotocamere.

Per gli scopi di questo elaborato sono stati presi in considerazione soltanto tre "topic": *point cloud*, l'immagine di sinistra e di destra corrette dalle distorsioni.

I primi due vengono dati in input al nodo del pacchetto "autonav_ai", mentre il pacchetto "rtabmap_ros" riceve in input gli ultimi due "topic" per realizzare la SLAM.

## 5.2 Pacchetto autonav_ai

Questo pacchetto, come accennato in precedenza, è stato sviluppato appositamente per implementare la rete neurale all'interno di ROS. Innanzitutto è stato creato in linguaggio C++ un nodo ROS che include una serie di librerie per poter svolgere le operazioni necessarie al funzionamento.

Nel dettaglio sono state utilizzate le librerie di ROS per la costruzione della struttura generale del nodo, le librerie del pacchetto "sensor_message" per ricevere/trasmettere le immagini e la *point cloud*, le librerie di OpenCV per elaborare le immagini ed infine la libreria della rete neurale sviluppata con MATLAB. I passaggi principali del funzionamento del nodo sono i seguenti:

1. Ricezione il più possibile simultanea della *point cloud* e dell'immagine di sinistra;
2. Trasformazione dell'immagine per essere utilizzata da OpenCV;
3. Conversione dell'immagine da BGRA a RGB;
4. Ridimensionamento dell'immagine da 376x672x3 a 224x224x3 per poter essere elaborata dalla rete neurale;
5. Alterazione del canale "R" dell'immagine per identificare i punti che appartengono alla categoria di interesse, ovvero la categoria del terreno asfaltato;
6. Ridimensionamento dell'immagine da 224x224x3 a 376x672x3;
7. Confronto dell'immagine con i dati della *point cloud* e alterazione della coordinata "Z" dei punti a seconda che questi appartengano o meno al terreno asfaltato;
8. Trasmissione della *point cloud* modificata tramite un "topic" apposito.

Sono da fare alcune considerazioni relativamente ad alcuni passaggi. In particolare la stereo camera fornisce le immagini codificate in "BGRA", ovvero ogni pixel è descritto da quattro numeri, detti canali, descritti da otto bit ciascuno. Essenzialmente corrispondono ai canali "Blue", "Green", "Red" e "Alpha" dell'immagine.

La rete neurale però è stata allenata utilizzando un ordine dei canali diverso ovvero, “Red”, “Green” e “Blue” (RGB), quindi è necessario convertire l’immagine con quest’ultima codifica.

Inoltre, mentre in MATLAB l’immagine è rappresentata da una matrice tridimensionale, in linguaggio C++ l’immagine è descritta da un vettore di $n$ elementi, con $n = dim1 \cdot dim2 \cdot dim3$. Quello che però cambia è l’ordine con il quale vengono concatenati i dati. Facendo riferimento alla Figura 5.3, le tre dimensioni di un’immagine sono altezza, larghezza e profondità.

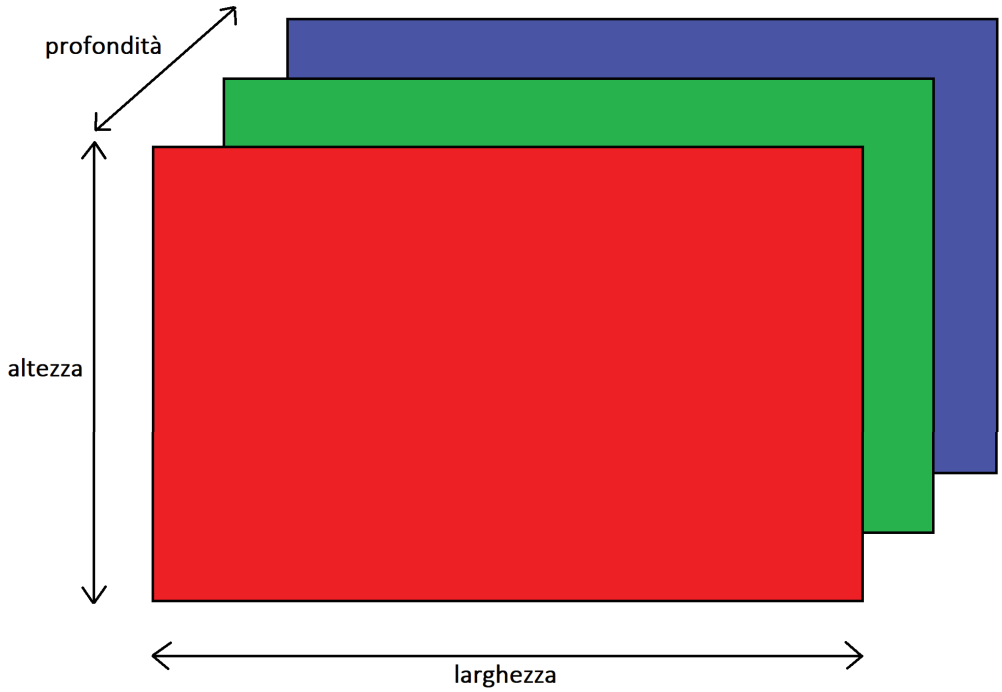



Figura 5.3: Dimensioni dell’immagine

Per quanto riguarda MATLAB l’immagine viene concatenata seguendo l’ordine altezza-larghezza-profondità, invece in ROS l’immagine è concatenata nell’ordine profondità-larghezza-altezza.

Per questo non solo è stato necessario cambiare la codifica, ma anche l’ordine dei dati all’interno dell’immagine, prima che essa sia data in input alla rete neurale.

Un’ultima considerazione è da farsi sul confronto tra immagine e *point cloud* e su come sono stati modificati i dati per variare la coordinata “Z”. Serve però spiegare come deve essere interpretato il vettore dei dati fornito dalla *point cloud*.

Ad ogni pixel dell’immagine vengono fatti corrispondere tre coordinate (“X”, “Y” e “Z”) ed il valore “BGRA” di quel pixel. Se il pixel ha coordinate pari a “NaN”, ovvero non è stato possibile misurare le sue coordinate, allora il valore del colore del pixel sarà anch’esso NaN = [255 255 255 127].

Ogni coordinata è rappresentata da quattro numeri da otto bit ciascuno, questo vuol dire che ogni coordinata è l’insieme di 32 bit, ovvero un numero *single precision*. In

Figura 5.4 [41] viene rappresentato in modo schematico la divisione dei bit in un numero di questo tipo.

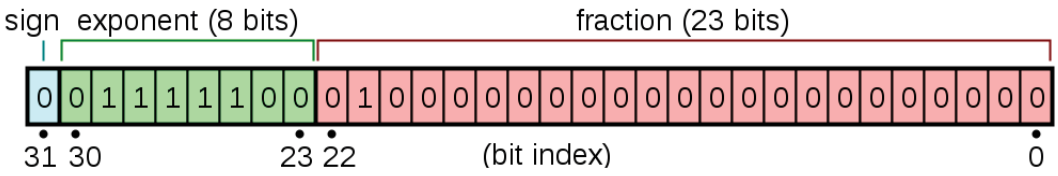



Figura 5.4: Schema numero *single precision*

Assegnando al bit $n$ la seguente notazione $b_n$ e sapendo che il valore potrà essere o 1 o 0, il numero rappresentato si ricava con la seguente formula:

$$value = (-1)^{b_{31}} + 2^{(b_{30}b_{29}...b_{23})_2 - 127} + \sum_{i=1}^{23} b_{23-i} \cdot 2^{-i} \tag{5.1}$$

In totale ogni pixel è descritto da 16 numeri da 8 bit ciascuno, 4 per la coordinata "X", 4 per la coordinata "Y", 4 per la coordinata "Z" e gli ultimi 4 per i canali "BGRA".

Ottenuta in output l'immagine originale con il canale "Red" modificato, è stato possibile riconoscere come modificare la coordinata "Z" di ciascun pixel dell'immagine.

Infatti se il pixel nel canale "Red" era pari a 255, la sua coordinata "Z" veniva modificata a -10 metri rispetto al sensore. Invece se il pixel nel canale "Red" era pari a 0, la sua coordinata "Z" veniva modificata a 0 metri rispetto al sensore.

In questo modo la *point cloud* modificata conserva la distanza "X" e "Y" dalla stereo camera, ma costruisce una nuvola di punti su due livelli di altezza ben distinti.

## 5.3 Pacchetto octomap_server

Questo pacchetto [42] [43] è stato creato per generare una *occupancy map* tridimensionale ed anche bidimensionale a partire da una *point cloud* in input. Nel caso in esame è stata utilizzata solo la mappa bidimensionale. Nella figura 5.5 viene illustrato un esempio di entrambe le mappe

Inoltre è possibile impostare l'altezza minima dei punti della *point cloud* da considerare come ostacoli. Questa è stata impostata a -2 metri, così da poter sicuramente escludere i pixel riconosciuti come terreno asfaltato che sono posizionati a -10 metri.

La ragione per cui gli altri pixel vengono posti ad un'altezza di 0 metri rispetto al sensore è dovuta al fatto che in questo modo la posizione degli ostacoli viene aggiornata molto più velocemente, quindi in presenza di ostacoli dinamici le celle che prima erano occupate si liberano più velocemente.

Infine è stata impostata una risoluzione della mappa di 0.1 metri/pixel per diminuire i tempi di calcolo. La mappa ottenuta in questo modo viene data in input al nodo "move_base", come mappa "statica" per realizzare la navigazione autonoma.

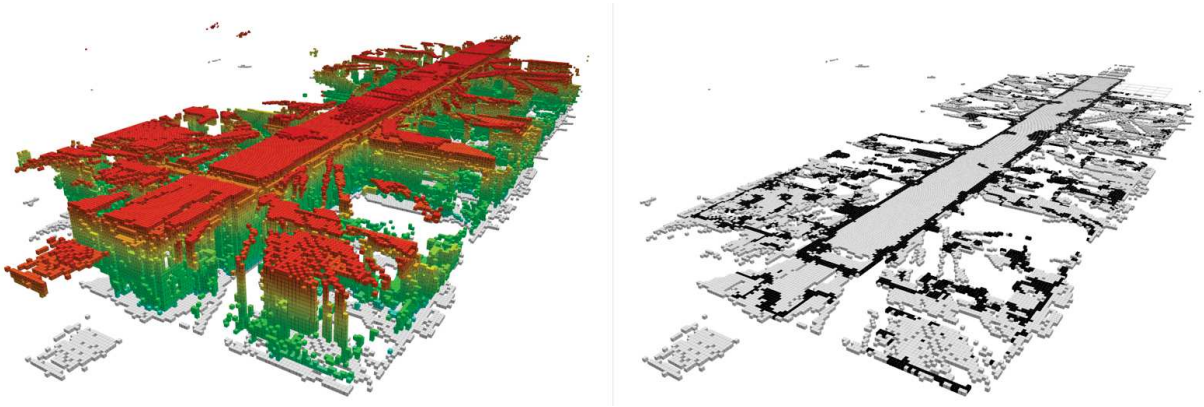


Figura 5.5: Esempio di mappa tridimensionale (destra) e la rispettiva mappa bidimensionale (sinistra)

## 5.4 Pacchetto rtabmap_ros

RTAB-Map [44] (Real-Time Appearance-Based Mapping) è una tecnica di SLAM basata sulla Graph-SLAM e può essere implementata sia con il LiDAR che con la stereo camera.

Per utilizzarlo su ROS è stato sviluppato un pacchetto apposito [45]. La scelta di utilizzare proprio questo pacchetto è dovuta alla sua flessibilità nel poter integrare più strumenti tra loro, come ad esempio GPS o *wheel odometry*. Nel caso specifico si è utilizzata la sola stereo camera.

In Figura 5.6 è rappresentato schematicamente il collegamento tra stereo camera e il nodo ROS. In particolare il pacchetto è composto da più nodi. Nell'immagine ce ne sono tre, ma per questo elaborato ne sono usati solo due ("stereo_odometry" e "rtabmap"), in quanto il terzo serve a togliere la distorsione dalle immagini, operazione che viene già svolta dal nodo "zed_wrapper_ros".

Inoltre non sono stati considerati i "topic" della mappa che RTAB-Map realizza, in quanto ai fini di quest'elaborato interessa solo quella generata dal nodo di OctoMap. L'unico "topic" preso in esame è l'"Odometry" ovvero la posizione della stereo camera rispetto a un punto iniziale, preso come riferimento per la localizzazione.

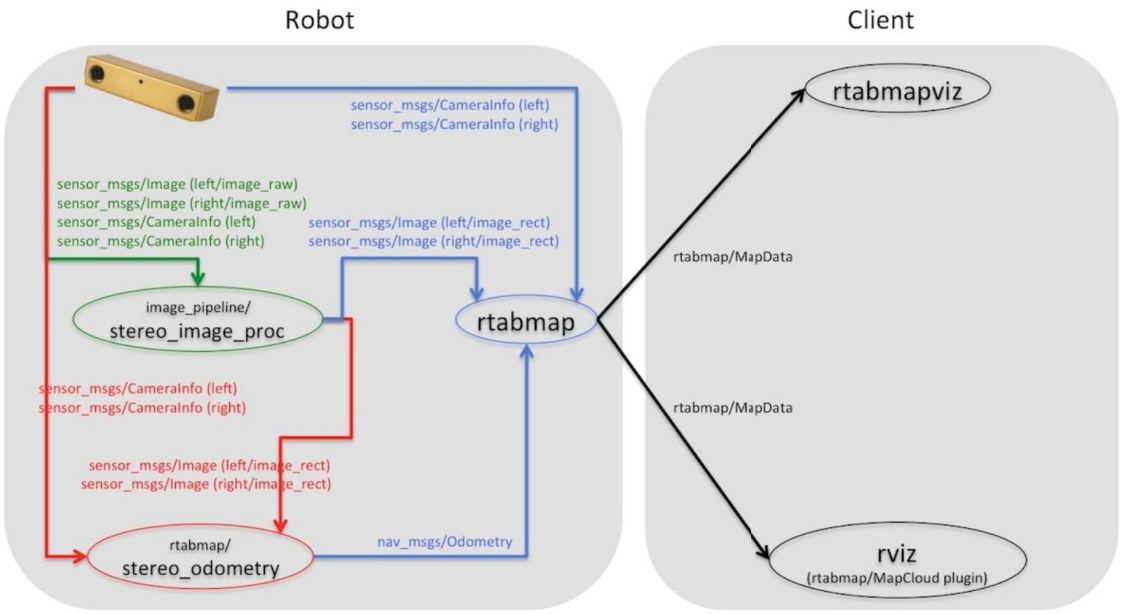



Figura 5.6: Nodo ROS di rtabmap_ros

## 5.5 Pacchetto move_base

Questo pacchetto [46] realizza la traiettoria che il rover deve seguire, impostato un punto di arrivo. Infatti, come schematizzato in Figura 5.7, il nodo in questione ha bisogno di quattro input: l'"Odometry", le trasformazioni tra i vari sistemi di riferimento (e.g. la posa della stereo camera rispetto al rover), la mappa dove sono presenti gli ostacoli "statici" e i sensori per poter rilevare gli ostacoli "dinamici". In generale il funzionamento del nodo è il seguente:

1. Il nodo genera una "global costmap" basandosi su due *layers*: lo "Static layer" che è la mappa degli ostacoli statici; l'"Obstacle layer" che è la mappa degli ostacoli dinamici.
2. Tramite una seconda mappa derivata dalle prime due, esso genera l'"Inflatious layer" ovvero una mappa in cui vengono segnate come occupate le celle poste entro una certa distanza dagli ostacoli. In questo modo si tiene conto dell'ingombro del rover e eventualmente di un margine di sicurezza che il rover deve mantenere dagli ostacoli.
3. Nota la posizione del rover all'interno della mappa, viene generata la traiettoria globale dal "global planner".
4. Viene inoltre generata una "local costmap". Questa a differenza di quella globale tiene conto di ulteriori ostacoli che possono essere rilevati dai sensori del rover.
5. In base agli ostacoli presenti nella "local costmap", il nodo produce tramite il "local planner" la traiettoria locale.
6. Infine esso traduce quest'ultima in comandi di velocità trasversale e rotazionale, i quali vengono inviati al nodo di controllo del rover.

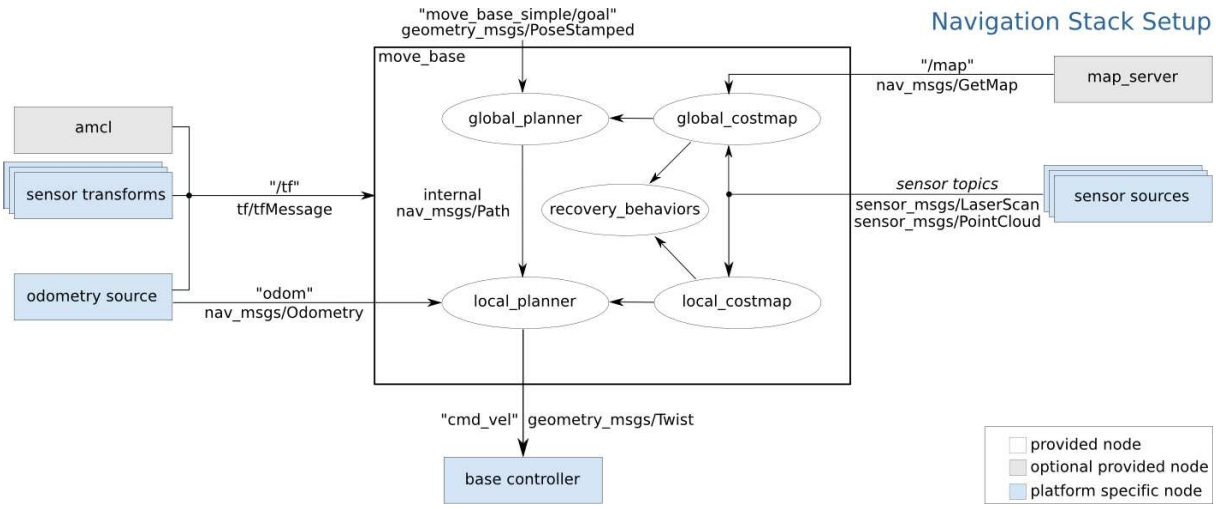



Figura 5.7: Rappresentazione del nodo ROS di move_base

Nelle tabelle di seguito si riportano i parametri impostati per la costruzione della *global/local costmap* (Tabella 5.1) e *global/local planner* (Tabella 5.2).

| | **Publish frequency [Hz]** | **Width [m]** | **Height [m]** | **Resolution [px/m]** | **Origin x [m]** | **Origin y [m]** | **Static map** | **Rolling window** | **Static layer** | **Obstacle layer** | **Inflatious layer** |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **global costmap** | 2 | 100 | 100 | 0.1 | -50 | -80 | ✓ | ✗ | ✓ | ✗ | ✓ |
| **local costmap** | 0.5 | 10 | 10 | 0.1 | -5 | -5 | – | ✓ | ✗ | ✗ | ✓ |

Tabella 5.1: Parametri delle *costmap*

| | **Type** | **Max vel x [m/s²]** | **Max vel backwards [m/s]** | **Max vel theta [m/s]** | **Acc lim x [m/s²]** | **Acc lim theta [m/s]** | **Min obs distance [m]** | **Footprint** |
|---|---|---|---|---|---|---|---|---|
| **global planner** | NavfnROS | – | – | – | – | – | – | – |
| **local planner** | TEB local planner | 0.25 | 0.2 | 0.25 | 0.2 | 0.2 | 0.7 | Rectangle |

Tabella 5.2: Parametri dei *planner*

La differenza principale tra le due "costmap" è la grandezza. Infatti la mappa locale è più piccola ed è l'unica che viene aggiornata rispetto agli ostacoli dinamici. Questi

ultimi dipendono dalla distanza massima a cui il sensore riesce a rilevarli con una buona accuratezza e ciò vincola la grandezza della mappa.

Per quanto riguarda i *layers* finali, generalmente lo "Static layer" viene utilizzato solo per la *global costmap*, mentre l'"Obstacle layer" viene impiegato per la *local costmap*.

Nel caso studiato in questa tesi l'"Obstacle layer" non è stato utilizzato in quanto la mappa ricevuta dal nodo di OctoMap mappa in *real-time*, quindi riesce a rilevare sia gli ostacoli statici che dinamici.

Per quanto riguarda il *local planner* è stato scelto TEB (Time-Elastic-Band) [47] [48] per la costruzione ottimale della traiettoria, adatta al tipo di rover utilizzato.

Infine come *global planner* è stato scelto quello standard del pacchetto "move_base" ovvero NavfnROS, dove i campi legati alla dinamica della traiettoria sono completamente vuoti, in quanto questa parte viene gestita dal *local planner*.

## 5.6 Pacchetto locomotion

Questo pacchetto è stato sviluppato appositamente per il rover e serve a controllare i motori delle ruote dello stesso, dati un comando di velocità lineare e uno di velocità angolare. In particolare il pacchetto e composto da due nodi:

- **loc_base_controller:** converte i comandi delle due velocità in un vettore di tre numeri. Il primo serve a far compiere una delle cinque operazione per il nodo successivo (0=Nessuna Operazione, 1=Setup dei motori, 2=Muovi i motori, 3=Controlla status motori, 4=Chiudi i motori), il secondo e il terzo sono rispettivamente le velocità dei motori di sinistra e di destra.
- **debug_Arduino:** gestisce le cinque operazioni descritte sopra e converte i messaggi di velocità dei motori di sinistra e destra nelle PRF (Pulse Repetiton Frequency) da fornire agli Arduino che controllano i motori.

In questo caso i comandi di velocità vengono assegnati in una scala da 0 a 1. Ciò è pensato per comandare il rover con il joystick per il quale non esiste una vera e propria conversione tra segnale dello stesso e velocità in metri al secondo.

## 5.7 Pacchetto piksi_multi_rtk_ros

Questo pacchetto [49] è stato sviluppato dall'ETH di Zurigo per il controllo di alcuni ricevitori GPS, come ad esempio quello prodotto dalla Swift e utilizzato in questo lavoro.

In pratica il pacchetto è pensato per poter gestire sia il solo ricevitore GPS, sia il ricevitore insieme alla *base station* come illustrato in Figura 5.8.

In altre parole esso può gestire la tecnica di misura del GPS differenziale: la *base station* rimane in una posizione fissa e nota, mentre il ricevitore viene montato sul rover; ricevendo

entrambi lo stesso segnale GPS è possibile effettuare delle correzioni sulla posizione del ricevitore, sottraendo il segnale ricevuto dalla *base station*.

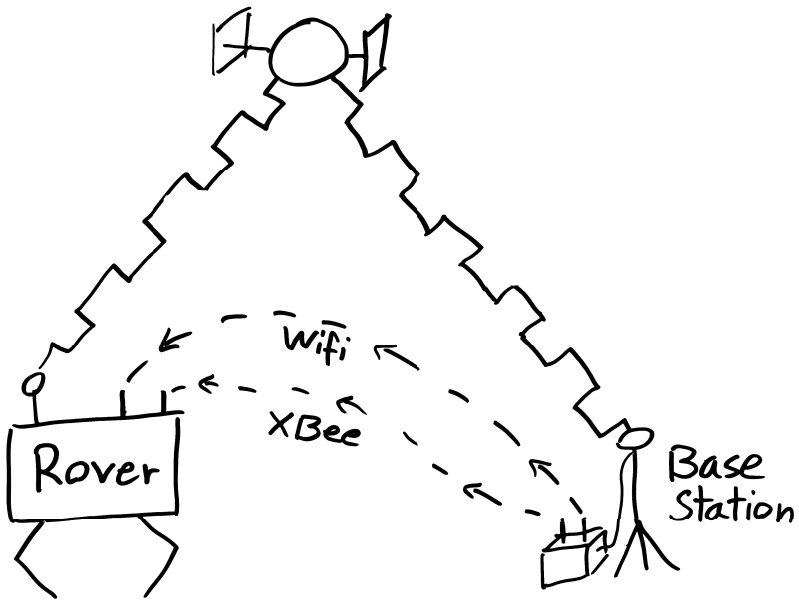



Figura 5.8: Rappresentazione GPS differenziale

Questa tecnica permette di ottenere misure di posizione molto precise, rispetto ad un solo ricevitore GPS. Tuttavia, poiché per questo studio non era disponibile una *base station*, essa non è stata utilizzata.

Per questa stessa ragione il segnale GPS non è stato integrato con la Visual Odometry di RTAB-Map, ma utilizzato solo come riferimento per la traccia a terra.

# Capitolo 6

# Apparato sperimentale

La sperimentazione sul campo dell'algoritmo è stata resa possibile grazie al rover sviluppato dal progetto studentesco MORPHEUS (Mars Operative Rover of Padova Engineering Students) [50].

Il progetto MORPHEUS nasce verso la fine del 2014 ed ha come obiettivo quello di progettare, costruire e testare un prototipo di rover in grado di eseguire operazioni tradizionalmente legate all'esplorazione planetaria.

Al momento il rover è composto da 6 motori, 6 arduino per il controllo dei motori, una scheda Jetson TX1 per la gestione di ROS, una stereo camera e un ricevitore GPS.

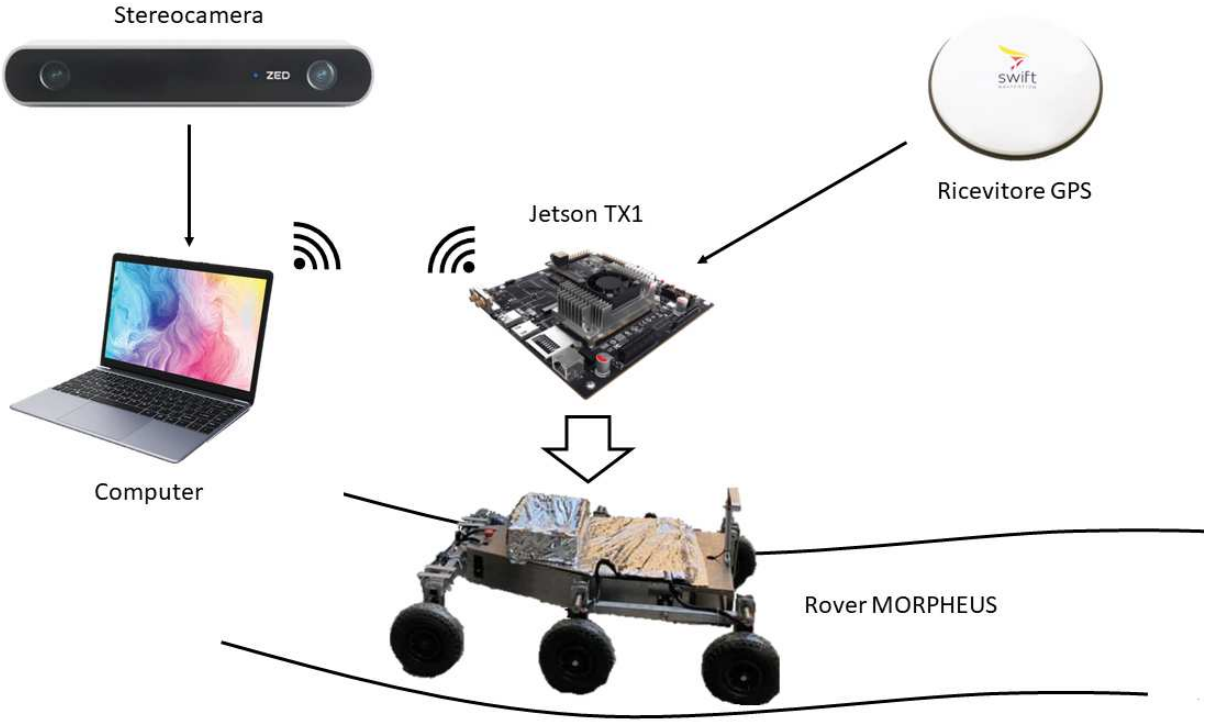



Figura 6.1: Rappresentazione dell'apparato sperimentale

Inoltre è stato utilizzato un computer collegato tramite Wi-Fi alla Jetson TX1, a supporto della stessa, in particolare per poter utilizzare versioni dei pacchetti di ROS

più recenti. Per questa ragione la stereo camera, a differenza del ricevitore GPS, è stata collegata direttamente al computer invece che alla Jetson TX1.

In ogni caso i pacchetti e la rete neurale sono pensati per essere utilizzati anche sulla Jetson TX1, bisognerebbe però prima aggiornare ROS e questo si tradurrebbe nell'installare la versione di Ubuntu più aggiornata. In Figura 6.1 viene rappresentato in modo schematico l'intero apparato sperimentale.

## 6.1 Rover

All'interno di questo paragrafo si vuole descrivere più nello specifico il funzionamento del rover. In particolare ci si concentrerà sul sistema di locomozione, che di fatto è l'unico motivo per cui si è utilizzato il rover.

Il sistema, consiste di sei ruote di diametro 26 centimetri, ciascuna motorizzata tramite un sistema di cinghie e pulegge. Le ruote hanno gli assi di rotazione tutti paralleli e sono disposte a coppie su tre bilancieri, come illustrato in Figura 6.2.

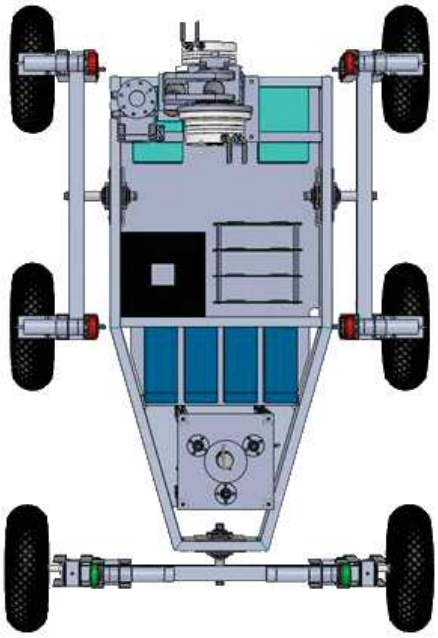


Figura 6.2: Vista in pianta del sistema di locomozione

Alla base della scelta di questo tipo di configurazione c'è la semplicità di realizzala e controllarla. Infatti per curvare, il rover si affida ad una guida *skid-steering*, invece che utilizzare ruote sterzanti, come è stato fatto ad esempio per il rover *Perseverance*.

La guida *skid-steering* si basa sul controllo differenziale delle ruote di destra e di sinistra che permette di ottenere meccanismi semplici e robusti, mantenendo comunque un'elevata manovrabilità su qualsiasi tipo di terreno.

Nonostante ciò, a causa delle complesse interazioni tra ruote e terreno, è complicato ricavare un modello cinematico e dinamico che descrivano con una buona accuratezza il moto del rover.

In particolare durante la manovra di rotazione il fenomeno dello slittamento è fondamentale per questo tipo di guida, perciò risulta difficile prevedere il comportamento del rover.

È comunque possibile fare una prima analisi cinematica del moto del rover in cui vengono messe in relazione le velocità angolari delle ruote ($\omega_R$ e $\omega_L$) con la velocità lineare ($v$) e di rotazione ($\omega$) nel seguente modo:

$$\omega_R = \frac{1}{r}(v + b \cdot \omega)$$
$$\omega_L = \frac{1}{r}(v + b \cdot \omega)$$

dove $r$ è il raggio della ruota (13 cm) e $b$ è la distanza della ruota dall'asse di simmetria (35 cm). Per un'analisi più dettagliata della cinematica si rimanda a [51].


Figura 6.3: Motore e riduttore planetario

Per quanto riguarda la parte meccanica del sistema di locomozione, essa si compone di sei ruote che vengono messe in movimento tramite una cinghia da sei motori *brushless* della Maxon, modello EC-max 30 ⌀30 mm da 60W, come mostrato in Figura 6.3

Ad essi vengono collegati dei riduttori planetari, sempre della Maxon, modello GP 32 HP ⌀32 mm, nella variante High Power. Nello specifico il riduttore è composto da tre stadi e permette di ottenere un rapporto di riduzione pari a 86:1.

Infine si riportano in Tabella 6.1 le specifiche del motore e in Figura 6.4 il range operativo in funzione della durata dell'operazione

| Motore EC-max 30 ⌀30 mm | | |
|---|---|---|
| Voltaggio nominale | [V] | 24 |
| Corrente a vuoto | [mA] | 191 |
| Velocità nominale | [rpm] | 8040 |
| Coppia nominale (max. coppia continua) | [mNm] | 60.7 |
| Corrente nominale (max. corrente continua) | [A] | 2.66 |
| Coppia di stallo | [mNm] | 458 |
| Corrente di stallo | [A] | 18.8 |
| Efficienza massima | [%] | 81 |

Tabella 6.1: Parametri del motore

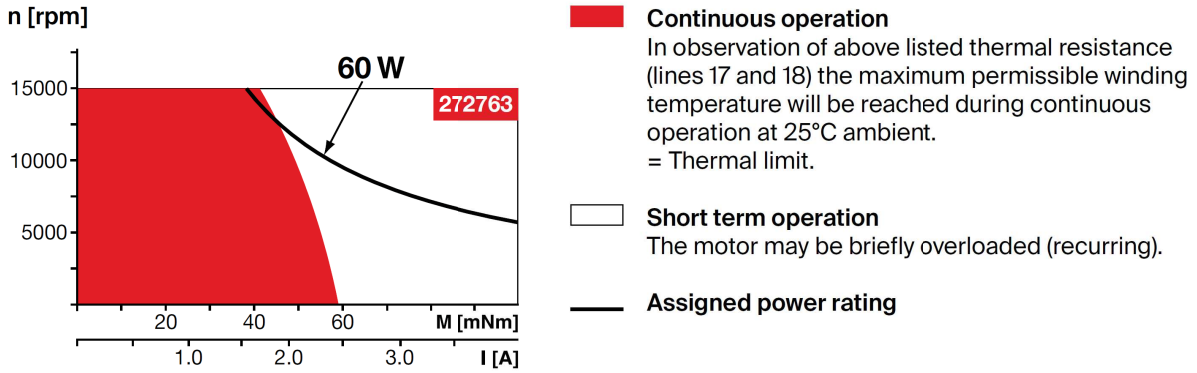



Figura 6.4: Range operativo del motore

## 6.2 Stereo camera ZED

La stereo camera viene prodotta da StereoLab, modello ZED mostrato in Figura 6.5. Essa si collega al computer tramite una presa USB 3.0 e può essere utilizzata solo se viene installato il suo SDK. Per funzionare ha anche bisogno di una GPU della NVIDIA, in quanto l'SDK utilizza la libreria CUDA. In Figura 6.6 vengono riportati i dati principali. La stereo camera è montata al centro della parta anteriore del rover, sollevata a 47 cm da terra tramite un profilato.


Figura 6.5: Stereo camera ZED

## Technical Specifications

**Video Output**

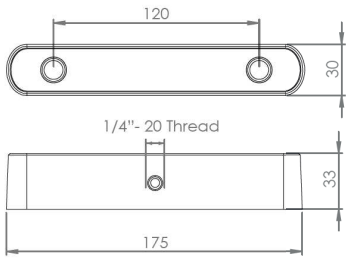
| | |
|---|---|
| Output Resolution | Side by Side 2x (2208x1242) @15fps<br>2x (1920x1080) @30fps<br>2x (1280x720) @60fps<br>2x (672x376) @100fps |
| Output Format | YUV 4:2:2 |
| Field of View | Max. 90° (H) x 60° (V) x 100° (D) |
| RGB Sensor Type | 1/3" 4MP CMOS |
| Active Array Size | 2688x1520 pixels per sensor (4MP) |
| Focal Length | 2.8mm (0.11") - f/2.0 |
| Shutter | Electronic synchronized rolling shutter |
| Interface | USB 3.0 - Integrated 1.5m cable |

**Depth Sensing**

| | |
|---|---|
| Baseline | 120 mm (4.7") |
| Depth Range | 0.5 m to 25 m (1.6 to 82 ft) |
| Depth Map Resolution | Native video resolution (in Ultra mode) |
| Depth Accuracy | < 2% up to 3m<br>< 4% up to 15m |

## Mechanical Drawing

*Dimensions are in mm*




## System Requirements

Win 10, Win 8 Win 7
Ubuntu 18.0/16.04
CentOS, Debian (via Docker)
USB3.0 Interface

### SDK Requirements

Dual-core 2.3GHz or faster
Minimum 4GB RAM Memory
Nvidia GPU (1) Compute capability ≥ 3.0

(1) Compatible with Nvidia Jetson Nano, TX2, Xavier

Figura 6.6: Caratteristiche della ZED

Come accennato precedentemente per lo scopo di questo elaborato è stata utilizzata una risoluzione di 376x672 per velocizzare i tempi di calcolo.

## 6.3 Ricevitore GPS

Per quanto riguarda il ricevitore GPS esso viene prodotto dalla Swift navigation, modello GPS500 mostrato in Figura 6.7.

Il ricevitore è montato nella parte anteriore del rover ed è collegato alla scheda di controllo del ricevitore tramite un connettore TNC. La scheda è a sua volta collegata alla Jetson TX1 tramite un cavo Ethernet. In Tabella 6.2 si riportano i dati del ricevitore.


Figura 6.7: Ricevitore GPS

| GNSS Antenna GPS500 | | |
|---|---|---|
| **Segnale ricevuto** | | |
| GPS | | L1/L2 |
| GLONASS | | L1/L2 |
| Galileo | | E1/E5b(1192-1229.64 MHz) |
| BeiDou | | B1/B2/B3 |
| QZSS | | L1/L2 |
| SBAS | | L1 |
| **Guadagno alla Zenit** | | |
| 1205-1278 MHz | [dBi] | 5.5 |
| 1559-1615 MHz | [dBi] | 5.5 |
| **Altri parametri** | | |
| Impedenza nominale | [ohm] | 50 |
| Polarizzazione | | RHCP |
| Rapporto assiale | [dB] | $\leq 3$ |
| Guadagno LNA | [dB] | 40 |
| Figura di rumore | [dB] | $\leq 2$ |
| Output/Input VSWR | | $\leq 2$ |
| Tensione operativa | [V] DC | 3.3-12.0 |
| Corrente operativa | [mA] | 45 (massimo) |
| Group Delay Ripple | [ns] | $\leq 5$ |

Tabella 6.2: Parametri del ricevitore GPS

## 6.4 Jetson TX1

La scheda Jetson TX1, mostrata in Figura 6.8, viene prodotta dalla NVIDIA. Si caratterizza come un *embedded computer*, ovvero un computer dove CPU e GPU sono integrati all'interno di un unico processore.

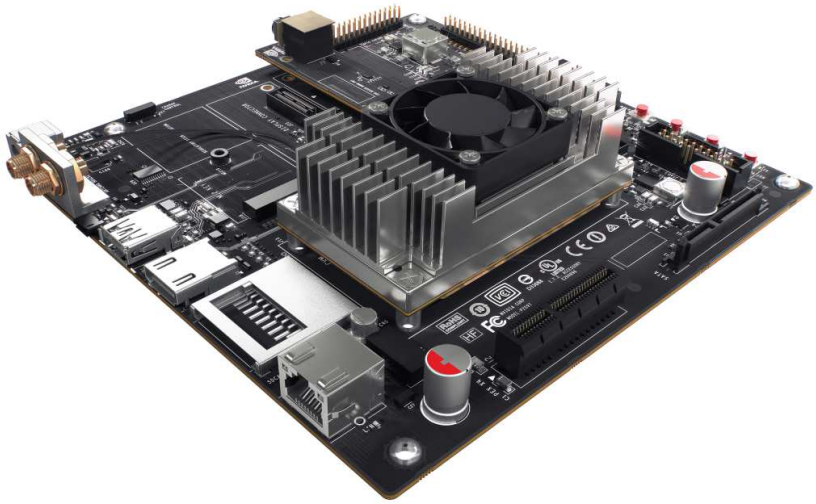


Figura 6.8: Jetson TX1

Nella Figura 6.9 vengono elencati i principali componenti della scheda, le funzioni dei quattro bottoni, le porte di input/output e alcune caratteristiche principali.

**JETSON TX1 MODULE**

- NVIDIA Maxwell™ GPU with 256 NVIDIA® CUDA® Cores
- Quad-core ARM® Cortex®-A57 MPCore Processor
- 4 GB LPDDR4 Memory
- 16 GB eMMC 5.1 Flash Storage
- 10/100/1000BASE-T Ethernet

**JETSON CAMERA MODULE**

- 5 MP Fixed Focus MIPI CSI Camera

**BUTTONS**

- Power On/Off
- Reset
- Force Recovery
- User-Defined

**I/O**

- USB 3.0 Type A
- USB 2.0 Micro AB (supports recovery and host mode)
- HDMI
- M.2 Key E
- PCI-E x4
- Gigabit Ethernet
- Full-Size SD
- SATA Data and Power

**POWER OPTIONS**

- External 19V AC adapter

Figura 6.9: Caratteristiche della Jetson TX1

## 6.5 Campagna sperimentale

Per quanto riguarda la fase di *testing* vera e propria dell'intero algoritmo, grazie all'apparato sperimentale appena descritto, si sono identificati due tipi di obiettivi:

- **Obiettivo primario - Mappatura:** l'algoritmo deve essere in grado di realizzare una mappa in cui sia ben distinguibile il terreno asfaltato da quello erboso. Il paragone viene fatto utilizzando un'immagine satellitare del luogo.
- **Obiettivo secondario - Navigazione:** il rover deve essere in grado di navigare autonomamente, passando solo attraverso il terreno asfaltato. Questo obiettivo è stato raggiunto impostando il punto di arrivo in una zona al di fuori dell'asfalto, per verificare che la navigazione autonoma utilizzasse veramente la mappa realizzata.

Bisogna specificare che per il primo obiettivo non è stato previsto l'utilizzo della navigazione autonoma. Infatti quest'ultima sarebbe stata eccessivamente lenta per poter coprire grandi distanze, poiché è stato necessario imporre dei limiti di velocità molto ridotti, sia per motivi di sicurezza generale sia per un adeguato funzionamento dell'applicazione.

Il secondo obiettivo è indicato come secondario, in quanto non è stato necessario per verificare l'effettiva bontà dell'algoritmo, quanto per constatare che questo potesse funzionare anche all'interno della navigazione autonoma.

Infatti nel caso in cui, per motivi di tempo, non si fosse riusciti a testare il rover per raggiungere questo secondo obiettivo, ciò non avrebbe compromesso il vero scopo di questo elaborato, ovvero testare l'algoritmo di mappatura.

Il luogo per eseguire i test è stato il giardino esterno del DIM (Dipartimento d'Ingegneria Meccanica), dove è presente una piccola stradina asfaltata che passa in mezzo al prato.

# Capitolo 7

# Risultati

In questo capitolo verranno illustrati e successivamente discussi i risultati ottenuti dai due tipi di verifica, compiuti durante la campagna sperimentale per soddisfare i due obiettivi imposti.

L'elaborazione dei dati è stata fatta utilizzando i dati forniti dai "topic" della rete ROS. Infatti grazie al comando **rosbag** è possibile registrare solo i "topic" di maggiore interesse, di seguito elencati:

- **mappa:** ovvero la mappa ottenuta dalla proiezione della *point cloud*, sicuramente essenziale per il confronto con la mappa satellitare.
- **immagine in output della rete neurale:** è stata utilizzata come *feedback* per verificare la precisione della rete neurale nel distinguere il terreno asfaltato da tutto il resto.
- **coordinate GPS:** utili per il confronto con la traiettoria del "global planner" e della SLAM oltre che per posizionare il rover sulla mappa satellitare.
- **traiettoria generata dal global planner:** utile per verificare che il rover costruisca la propria traiettoria all'interno del terreno asfaltato.
- **odometria:** ovvero i dati ottenuti dalla SLAM che permettono di confrontare con le coordinate GPS la posizione e orientamento del rover rispetto alla mappa.

## 7.1 Navigazione

Lo scopo di questa sezione è quello d'illustrare il funzionamento dell'algoritmo all'interno della navigazione autonoma, come già accennato nella Sezione 6.5, verificando così che la traccia del rover non fuoriuscisse dal terreno asfaltato. Un confronto tra la mappa generata dall'algoritmo con una mappa satellitare sarebbe stato utile, ma purtroppo il segnale GPS a disposizione era instabile e impreciso.

Ad ogni modo, come illustrato nella successiva Sezione 7.2, l'algoritmo è capace di riconoscere il terreno asfaltato dal restante e mapparlo; perciò si può presumere che il

rover sia in grado di rimanere in ogni caso all'interno del terreno asfaltato come illustrato in Figura 7.1.

(a) Immagine della CNN d'inizio percorso (b) Immagine della CNN di fine percorso

Figura 7.1

Inizialmente è possibile notare nella Figura 7.2 come la traccia del *global planner* venga generata per rimanere all'interno della zona bianca, per poi continuare lungo la zona grigia (ovvero quella sconosciuta) fino ad arrivare all'obiettivo. In questo caso, poiché l'obiettivo si trova sul terreno erboso, il rover non sarà mai in grado di raggiungerlo.

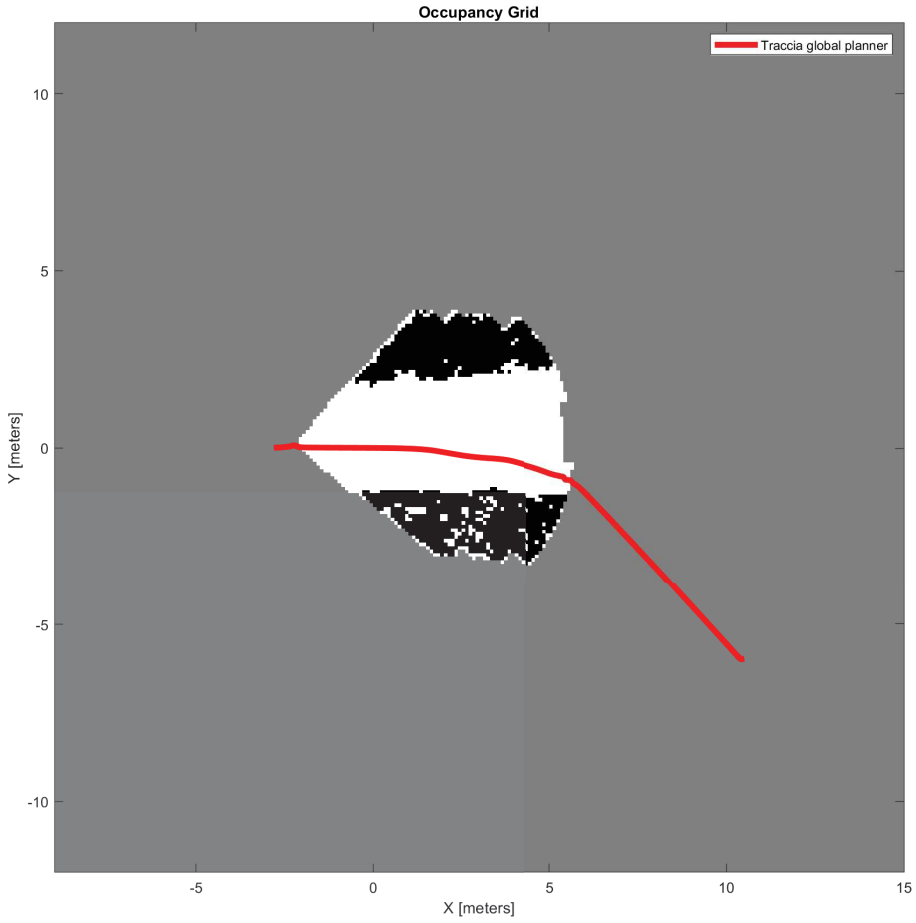



Figura 7.2: Mappa iniziale e traccia del *global planner*

Successivamente il rover viene guidato dalla navigazione autonoma fino al punto in cui esso deve entrare nella parte del terreno erboso, per poi bloccarsi. Infatti quanto appena descritto è ciò che viene illustrato in Figura 7.3, dove la traccia della SLAM s'interrompe proprio in prossimità del terreno erboso, che viene correttamente identificato come ostacolo.

Ulteriori test potrebbero essere svolti per approfondire meglio il funzionamento della navigazione autonoma. Infatti sarebbe interessante poter impostare degli ulteriori parametri, per verificare il ricalcolo della traiettoria, quando questa viene ostruita da un ostacolo.

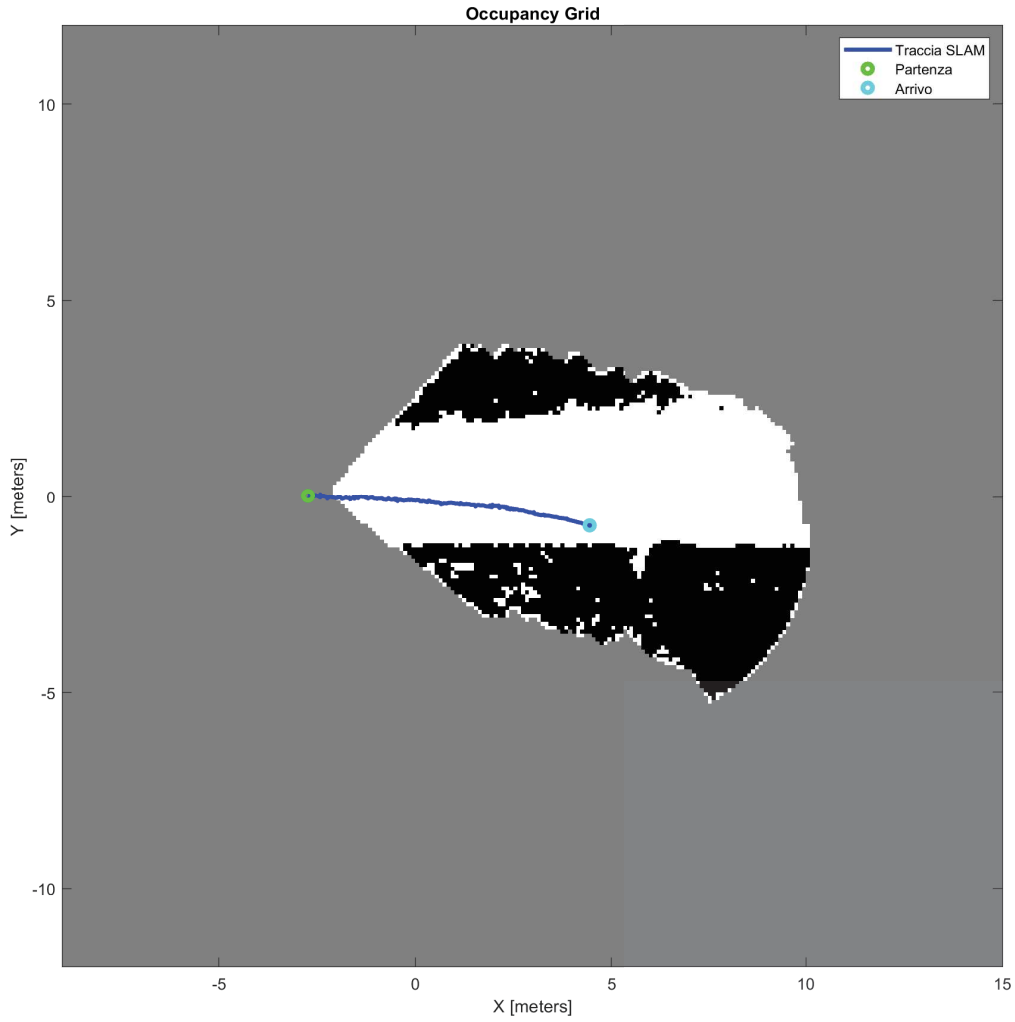



Figura 7.3: Mappa finale e traccia della SLAM

Infine in Figura 7.4 vengono messe a confronto la traccia della SLAM con la traccia del *global planner*. È possibile notare come non coincidano perfettamente, in quanto lo scopo del *global planner* è proprio quello di generare una traiettoria indicativa, ma che non vincola il rover a seguirla in maniera precisa.

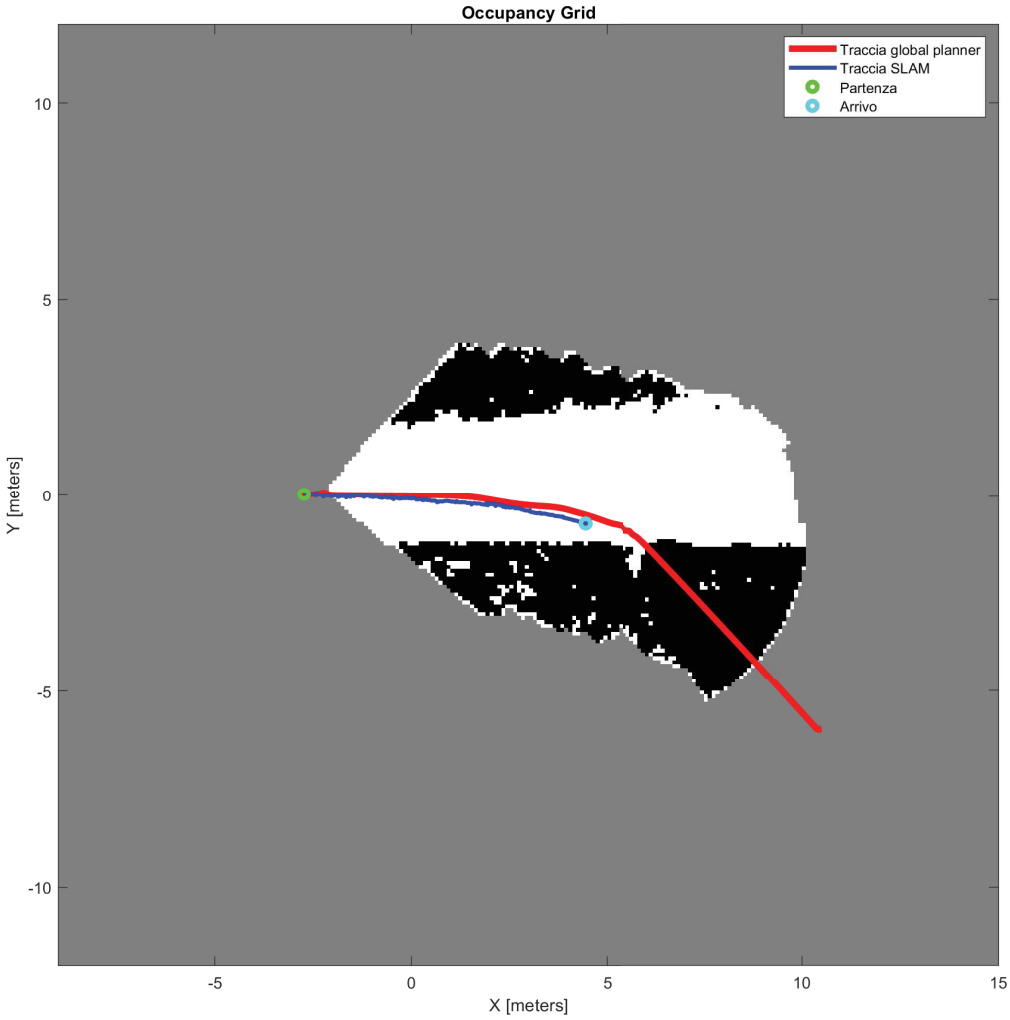


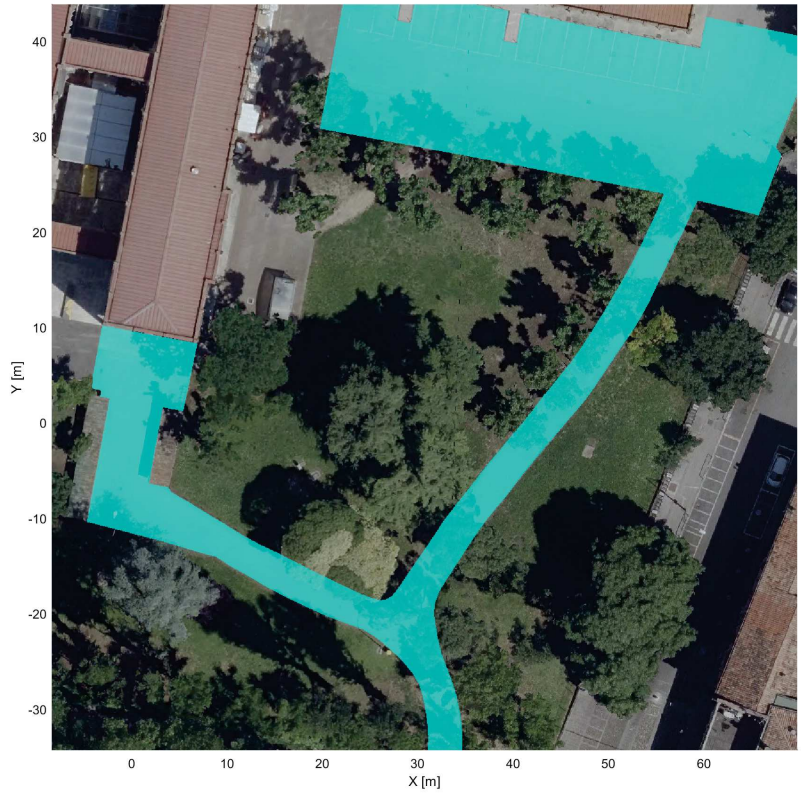

Figura 7.4: Confronto traccia SLAM e *global planner*

## 7.2 Mappatura

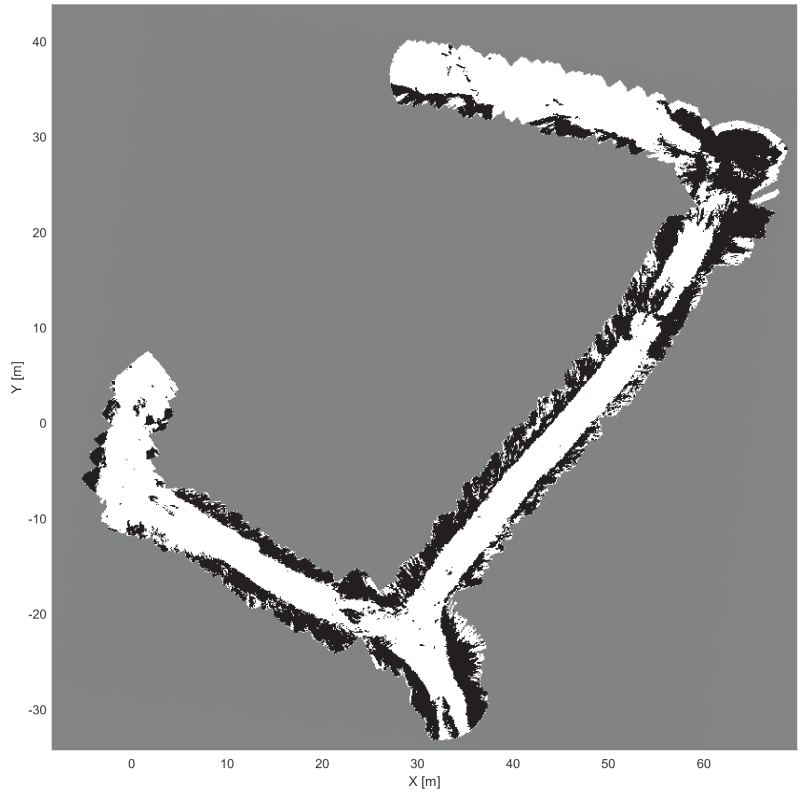
Per poter confrontare le immagini della mappa ottenuta e della mappa satellitare si sono dovute applicare delle trasformazioni (scala, rotazione e traslazione). Per essere realizzate sono state utilizzate le function dei pacchetto "Image Processing Toolbox" e "Mapping Toolbox" di MATLAB.

Infatti, è stato possibile assegnare ad ogni pixel dell'immagine satellitare (Figura 7.5a) due coordinate di riferimento (latitudine e longitudine), che sono poi state convertite in metri, utilizzando il punto partenza come centro del sistema di riferimento locale.

Nell'immagine satellitare è stata evidenziata in azzurro la parte di terreno asfaltato che il rover doveva essere in grado di riconoscere. Completati questi passaggi l'immagine della mappa (Figura 7.5b) è stata ruotata di $\alpha = -98.08^o$ per poter essere allineata con il sistema di riferimento globale.




(a) Mappa satellitare con evidenziato il terreno asfaltato (azzurro)




(b) Mappa ottenuta con la CNN

Figura 7.5

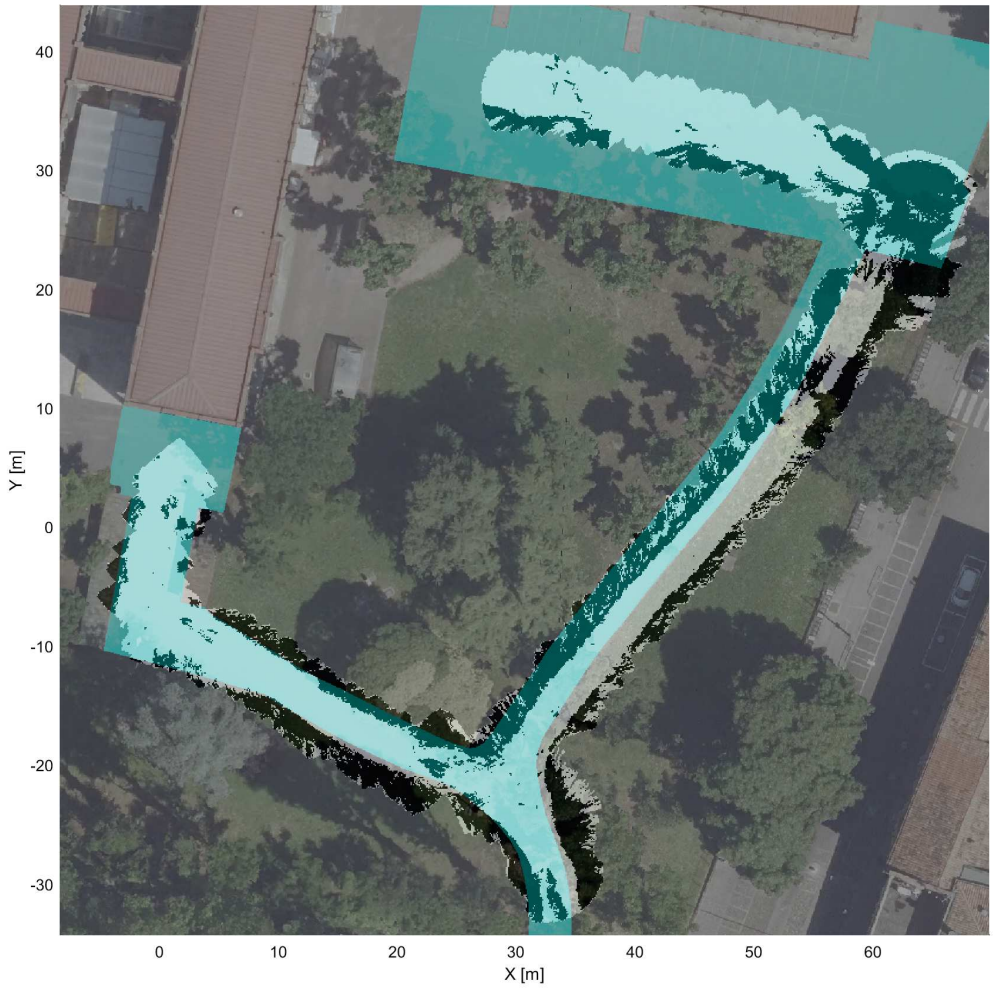



Figura 7.6: Confronto mappa satellitare e mappa ottenuta con la CNN

Dalla Figura 7.5b si può osservare come la mappa ottenuta rappresenti il terreno asfaltato come zona "libera" (bianco), mentre tutto il resto viene riconosciuto o come "ostacolo" (nero) oppure come "sconosciuto" (grigio).

Dal confronto in Figura 7.6, appare quindi molto chiara la capacità della rete neurale di riuscire a distinguere il terreno asfaltato. Infatti, se comparato alla porzione azzurra utilizzata come riferimento, si può notare una grande somiglianza.

Ci sono però delle differenze che meritano alcune attenzioni. La più evidente è proprio il graduale scostamento del terreno mappato dalla posizione reale, a mano a mano che il rover avanza lungo il percorso. Nonostante ciò, se si considerano piccoli tratti, l'andamento del terreno mappato rispecchia sufficientemente quello reale. Lo scostamento è dovuto proprio alla SLAM, la quale senza *loop closure* approssima bene localmente il percorso, ma non globalmente. Inoltre nella mappa sono presenti alcune zone che non vengono identificate come terreno asfaltato. Le cause sono principalmente due: il passaggio di persone e le ombre sull'asfalto, come illustrato nelle Figure 7.7 e 7.8.

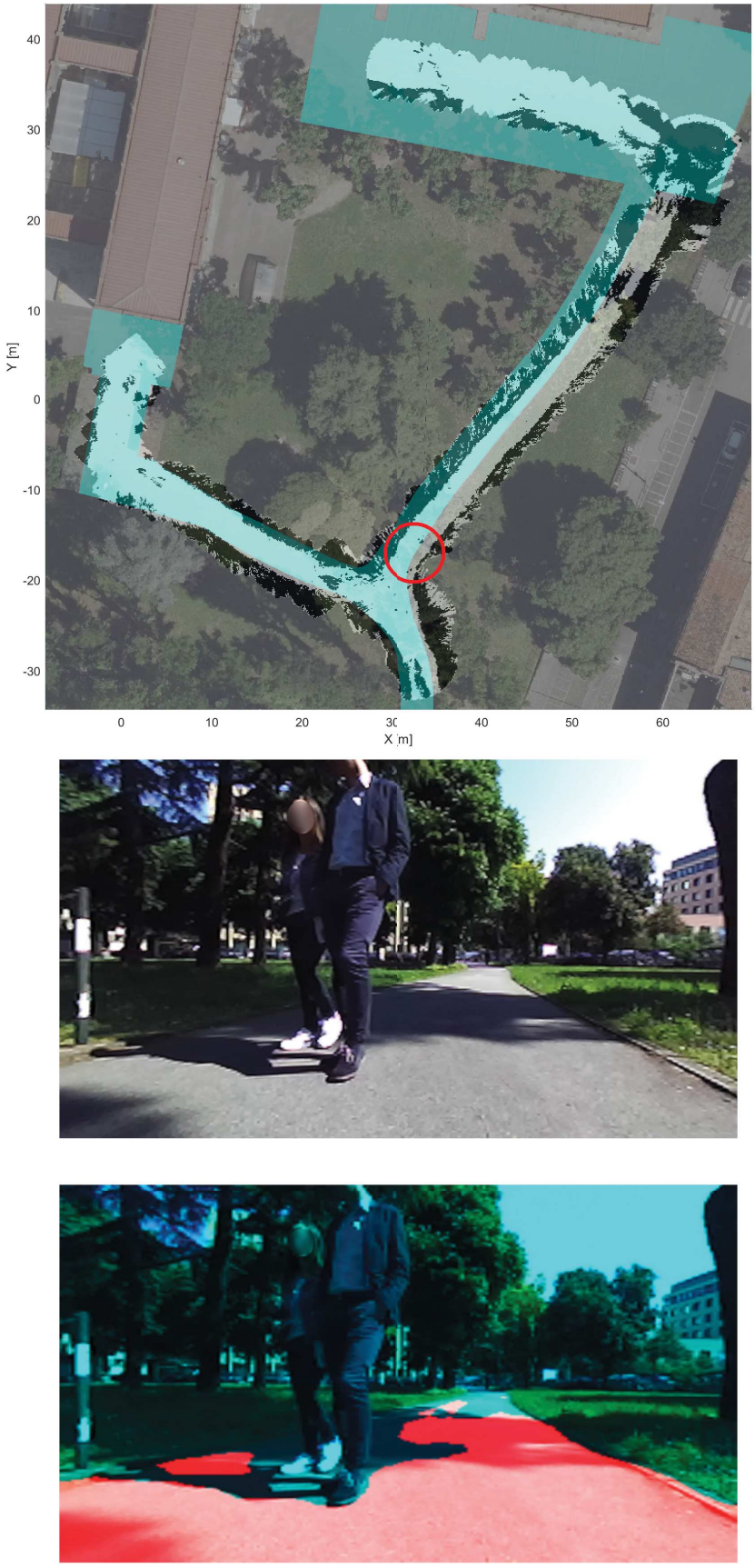



Figura 7.7: Errori dovuti alla presenza di persone


Figura 7.8: Errori dovuti alla presenza di ombre sull'asfalto

I disturbi dovuti al passaggio delle persone sono meno rilevanti rispetto alle ombre, come si può notare nelle due figure precedenti. La ragione è dovuta al fatto che, dopo il passaggio delle persone, le zone di terreno asfaltato nascoste vengono rietichettate e la mappa si aggiorna ripristinandosi quasi correttamente. Questo non accade per le ombre, in quanto essendo statiche non permettono alla mappa di aggiornarsi correttamente.

Infine in Figura 7.9 vengono messe a confronto la traccia ottenuta con il GPS e quella ottenuta con la SLAM.

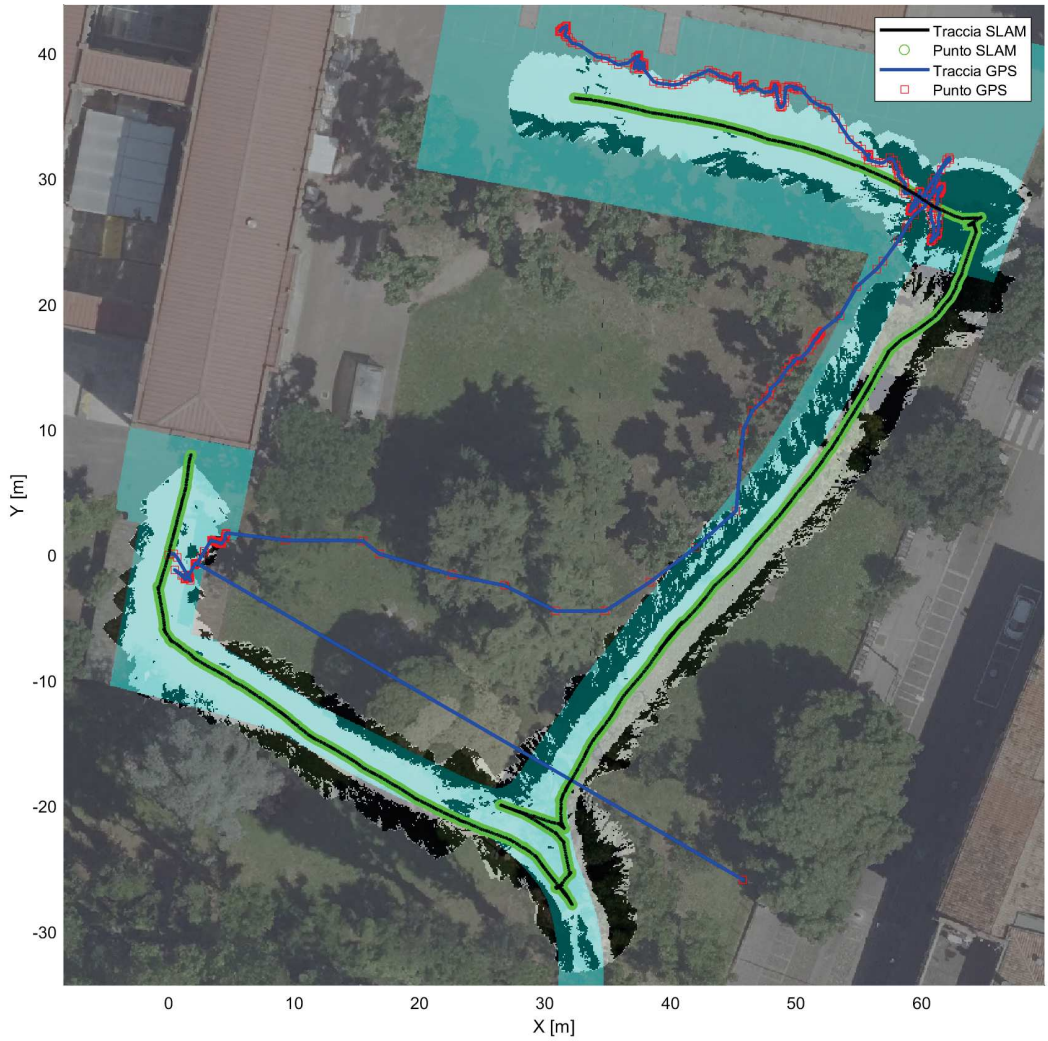



Figura 7.9: Confronto traccia GPS e traccia della SLAM

Bisogna notare come il GPS abbia avuto in generale delle pessime prestazioni, sicuramente per la presenza di edifici molto alti e degli arbusti che ha contribuito alla scarsa ricezione del segnale. Infatti, esclusi il momento iniziale ed il tratto finale, esso è risultato di scarsa utilità.

Invece la traccia ottenuta con la SLAM è più realistica, nonostante vada a scostarsi sempre più dalla posizione reale durante l'avanzamento. Questo però dimostra proprio uno dei punti di forza della SLAM: essa riesce a dare una buona stima della posizione relativa ad un punto iniziale, riuscendo quindi ad essere una valida alternativa al sistema GPS, se si tratta di distanze non troppo lunghe.

## 7.3 Prestazioni finali della rete neurale

Poiché finora sono stati presentati solo dei risultati qualitativi, si è ritenuto opportuno fornirne anche alcuni quantitativi. Infatti, dal momento che l'algoritmo dipende soprattutto dalla capacità della rete neurale di distinguere il terreno asfaltato da tutto il resto, sono state misurate nuovamente le prestazioni della rete, utilizzando le immagini raccolte durante la fase di "Mappatura".

I risultati ottenuti sono riportati nelle Tabelle 7.1 e 7.2 insieme a quelli che si erano ottenuti durante la fase di *Testing* della rete. In Tabella 7.3 viene invece riportata la matrice di confusione normalizzata, relativa alla fase di "Mappatura".

| | **Accuratezza** | **IoU medio** | **IoU pesato** | **F1 score medio** | **F1 score pesato** |
|---|---|---|---|---|---|
| **Testing** | 95.44 | 85.78 | 92.04 | 91.71 | 95.67 |
| **Mappatura** | 84.58 | 72.80 | 74.42 | 83.80 | 84.76 |

Tabella 7.1: Confronto percentuali dei parametri della rete neurale

| | | **Accuratezza** | **IoU** | **F1 score** | **weight** |
|---|---|---|---|---|---|
| **path** | **Testing** | 97.62 | 96.84 | 98.39 | 38.60 |
| | **Mappatura** | 92.07 | 91.90 | 95.78 | 34.02 |
| **grass** | **Testing** | 97.65 | 94.65 | 97.25 | 24.28 |
| | **Mappatura** | 95.22 | 69.59 | 82.07 | 7.82 |
| **tree** | **Testing** | 94.68 | 62.45 | 76.89 | 5.34 |
| | **Mappatura** | 68.27 | 64.86 | 78.68 | 33.46 |
| **other** | **Testing** | 91.23 | 89.20 | 94.29 | 31.78 |
| | **Mappatura** | 92.99 | 64.87 | 78.69 | 24.71 |

Tabella 7.2: Confronto percentuali dei parametri della rete neurale per categoria

| | path | grass | tree | other |
|---|---|---|---|---|
| **path** | 92.07 | 5.93 | 0.04 | 1.96 |
| **grass** | 0.18 | 95.22 | 3.69 | 0.91 |
| **tree** | 0.01 | 1.92 | 68.27 | 29.80 |
| **other** | 0.19 | 0.89 | 5.93 | 92.99 |

| | path | grass | tree | other |
|---|---|---|---|---|
| **path** | 14877015 | 958311 | 5884 | 316891 |
| **grass** | 6851 | 3537742 | 136962 | 33707 |
| **tree** | 731 | 305267 | 10851446 | 4735518 |
| **other** | 22631 | 105034 | 695862 | 10912458 |

Tabella 7.3: Matrice di confusione normalizzata e matrice di confusione, percentuali e numero totale dei pixel

Dal confronto è possibile notare come la rete non sia stata così performante come lo era stata durante la fase di *Testing*. Questo è principalmente dovuto al fatto che le immagini durante quest'ultima fase assomigliavano maggiormente a quelle usate durante il *training* della rete.

Nonostante quindi una diminuzione dell'accuratezza globale e dell'F1 score pesato di circa l'11% e dell'IoU pesato circa del 18% è possibile notare che, se si confrontano le categorie singolarmente (Tabella 7.2), per quanto riguarda il terreno asfaltato ("path") le differenze non sono così significative. È possibile quindi affermare che la rete è comunque in grado di distinguere molto bene il terreno asfaltato, anche se si trova in condizioni diverse da quello utilizzato per la fase di *Training*.

Un ultimo appunto è da farsi sulle percentuali relative al terreno erboso ("grass") e agli alberi ("tree").

Si nota infatti che per la prima di queste categorie l'accuratezza diminuisce di poco, mentre per l'IoU e per F1 score la percentuale diminuisce di parecchi punti. Ciò è dovuto alla sotto-rappresentazione di quest'ultimo tra le immagini. Infatti nella colonna dei pesi ("weight") esso rappresenta circa l'8% dell'intero numero di pixel, mentre durante la fase di *Testing* era circa il 24%. Questo porta ad avere uno sbilanciamento tra i veri positivi e i falsi positivi.

Invece per la categoria "tree" avviene una brusca diminuzione per quel che riguarda l'accuratezza, mentre aumentano di poco gli ultimi due parametri. Si verifica quindi la stessa situazione della categoria precedente, ma in termine opposti: gli alberi rappresentavano solo il 5% dell'interno numero di pixel durante il *Training*, mentre sono circa il 33% durante la fase di "Mappatura".

# Capitolo 8

# Conclusioni

Il lavoro descritto all'interno di questo elaborato ha permesso di mettere in pratica i concetti principali della navigazione autonoma ed in particolare quello che riguarda l'analisi di attraversabilità del terreno.

È stato sviluppato un algoritmo innovativo che, con il supporto di una rete neurale convoluzionale appositamente addestrata, ha permesso di mappare un tipo di terreno ritenuto "sicuro", riuscendolo a distinguere da tutto il resto.

L'addestramento è avvenuto utilizzando la rete neurale DeepLabv3+ e un *dataset* di immagini appositamente raccolte del luogo in cui si sarebbero svolti i test. Le immagini sono state ulteriormente modificate, tramite tecniche di *data augmentation*, per ottenere un *dataset* più ampio e quindi un addestramento più solido.

I risultati dell'addestramento sono stati ritenuti adeguati per poter implementare la rete sul campo: è stata così realizzata una rete ROS apposita che potesse integrare la rete neurale con un algoritmo di *visual odometry* ed uno di *path planning*.

La rete ROS è stata utilizzata per guidare in modo autonomo il rover del progetto MORPHEUS dell'Università di Padova ed è stato necessario integrare i pacchetti che gestiscono la sua strumentazione, ovvero la stereo camera, il ricevitore GPS e il sistema di locomozione.

La rete neurale è stata integrata all'interno di ROS creando un pacchetto apposito (autonav_ai), all'interno del quale essa identifica i pixel appartenenti al terreno asfaltato e il nuovo algoritmo modifica la *point cloud* della stereo camera, ponendo a diverse altezze i pixel a seconda che questi appartengano o meno alla categoria d'interesse.

In questo modo un pacchetto di mappatura basato su *point cloud* genera una mappa, dove i punti della *point cloud* al di sopra di una determinata altezza vengono indicati come "ostacolo".

La gestione della rete ROS, invece, è stata affidata al computer di bordo del rover (Jetson TX1) e ad un altro computer, collegati tra di loro tramite Wi-Fi. Questo ha permesso di raccogliere sul campo dati utili per la valutazione delle prestazioni dell'algoritmo.

Si sono quindi prefissati due obiettivi con priorità diverse volto, uno a valutare l'algoritmo vero e proprio, l'altro invece a valutare l'integrazione di questo con l'algoritmo di navigazione autonoma. Il secondo obiettivo è stato ritenuto secondario perché non è utile a dimostrare l'effettiva bontà dell'algoritmo.

La fase di *testing* è stata realizzata nel giardino del DIM (Dipartimento di Ingegneria Meccanica), dove è presente una piccola stradina asfaltata che passa in mezzo al prato. Lo scopo era quello di realizzare una mappa che mettesse in risalto il terreno asfaltato.

I risultati ottenuti evidenziano come l'algoritmo genera una mappa in cui il terreno asfaltato è l'unica parte ad essere indicata come "libera", ovvero l'unica all'interno della quale il rover può generare una traiettoria che gli permetta di raggiungere un determinato punto, passando esclusivamente su quella porzione di terreno.

Questi risultati sono stati quantificati, misurando nuovamente le prestazioni della rete neurale con alcune immagini ottenute durante questa fase. In particolare per quel che riguarda il solo terreno asfaltato si è notato che la rete ha un'accuratezza del 92.07%, un indice IoU del 91.90% e un F1 score del 95.78%.

Si è potuto comunque notare che la mappa è affetta da inesattezze, in particolare una dovuta alla SLAM che tende a scostare la posizione reale del terreno asfaltato a mano a mano che il rover avanza lungo il percorso.

Questo potrebbe essere risolto, facendo realizzare al rover dei circuiti in modo che l'algoritmo di SLAM possa effettuare la cosiddetta *loop closure*. Tuttavia questo non è stato possibile, perché l'intero algoritmo richiede un eccessivo utilizzo di risorse da parte del computer, e quindi limita l'autonomia della sua batteria.

Una seconda inesattezza della mappa è dovuta alla presenza delle ombre sul terreno asfaltato che rendono più difficile il riconoscimento del terreno per la rete neurale. Essa infatti non è stata sempre in grado di riconoscere porzioni con caratteristiche diverse da quelle su cui era stata allenata.

Il problema potrebbe essere risolto includendo nell'allenamento un numero maggiore d'immagini, in particolare di quelle in cui sono presenti queste caratteristiche; cosa che è stata fatta, ma forse in quantità non sufficiente.

Un'altra inesattezza, molto meno incisiva della precedente, è dovuta al passaggio delle persone lungo il percorso. Tuttavia la mappa riesce a ripristinarsi quasi del tutto e rimangono solo alcune piccole zone, segnate come "ostacolo".

Il problema potrebbe essere risolto creando un filtro per le persone, che riconosce all'interno dell'immagine la loro presenza e permette di sostituire i valori dei punti della *point cloud* corrispondenti con il valore "NaN". In questo modo le persone appariranno sulla mappa come punti sconosciuti che verranno aggiornati correttamente subito dopo il loro passaggio.

La criticità maggiore durante tutta la fase di *testing* è stata proprio la durata limitata della batteria del computer. Infatti, in termini di costi computazionali l'algoritmo è molto

oneroso, in quanto si utilizza un elevato numero di processi che devono essere elaborati sia dalla CPU che dalla GPU.

Ciò ha di certo rallentato molto questa fase e, per arrivare ad ottenere i risultati illustrati in questo elaborato, è stato necessario eseguire diversi tentativi.

Un altro aspetto problematico è stata la ricezione del segnale GPS, che è risultata disturbata da elementi esterni (edifici e alberi). Sarebbe stato più opportuno poter eseguire il test in uno spazio più aperto e più adatto alla ricezione del segnale. In ogni caso non è stato possibile realizzare più di due prove giornaliere e quindi risultava difficile raggiungere luoghi troppo distanti.

Possibili sviluppi futuri di questa applicazione possono riguardare innanzitutto la gestione della rete ROS. Essa potrebbe essere gestita soltanto dal computer del rover, riducendo così la complessità del sistema. Ciò permetterebbe di utilizzare batterie esterne che possono essere prontamente sostituite e allungare così i tempi di utilizzo del rover.

Altri miglioramenti possono riguardare l'efficienza dell'algoritmo e la riduzione del costo computazionale. Si potrebbe ad esempio realizzare una rete neurale ad hoc oppure implementare nuove tecniche di V-SLAM o addirittura provarne diverse o più sofisticate. Una di queste potrebbe essere quella che utilizza il LiDAR 3D, dove le misure di distanza sono molto più precise. La SLAM potrebbe essere migliorata anche integrando dati di diversi tipi di strumentazione, come il ricevitore GPS o anche un encoder per la *wheel odometry*.

Durante questo lavoro è emerso che si potrebbe migliorare l'algoritmo di *path planning* costruendone uno più specifico per questo tipo di rover. Anche il pacchetto che gestisce la locomozione probabilmente andrebbe ricalibrato meglio, mettendo a punto la correlazione tra la PRF data in input dai motori e la velocità lineare del rover.

In ogni caso il lavoro svolto all'interno di questo elaborato è sicuramente stato soddisfacente ed ha dimostrato la capacità di realizzare e gestire un nuovo algoritmo molto complesso da parte dell'intero apparato sperimentale, soprattutto considerando che si tratta di un'applicazione di tipo *real time*.

# Appendice

# Appendice A

# Algoritmi

## A.1 Algoritmo RANSAC

**Algoritmo 1** RANSAC implementazione generica

$data \leftarrow$ insieme di punti osservati
$model_{fin} \leftarrow$ modello finale utilizzato per determinare gli *outliers*
$nInliers_{old} \leftarrow 0$, numero di *inliers* massimo
$d_{max} \leftarrow$ distanza massima di un punto tollerata dal modello
$iter_{max} \leftarrow$ numero massimo di iterazioni
$iter \leftarrow 0$
**while** $iter \leq iter_{max}$ **do**
    $inliers \leftarrow$ seleziona $N$ punti casuali da $data$
    $nInliers_{new} \leftarrow N$
    $model_{ini} \leftarrow$ modello ottenuto con gli *inliers* selezionati
    **for** $i = 1 : numeroPuntiData - N$ **do**
        $d(i) \leftarrow$ distanza del punto $i$ rispetto al modello $model_{ini}$
        **if** $d(i) < d_{max}$ **then**
            $nInliers_{new}++$
        **end if**
        **if** $nInliers_{new} > nInliers_{old}$ **then**
            $model_{fin} \leftarrow model_{ini}$
            $nInliers_{old} \leftarrow nInliers_{new}$
        **end if**
    **end for**
**end while**

## A.2 Algoritmo SGD

**Algoritmo 2** Algoritmo SGD

$\mathbf{d} \leftarrow$ output corretto della rete
$[\mathbf{W}]_\mathbf{0} \leftarrow$ inizializzazione dei pesi della rete
$\mathbf{b} \leftarrow$ vettore dei bias dei nodi di Output
$\varphi(\cdot) \leftarrow$ funzione di attivazione del nodo
$N_{in} \leftarrow$ numero nodi in input
$N_{out} \leftarrow$ numero nodi in output
$\alpha \leftarrow$ learning rate
$e_{max} \leftarrow$ errore medio massimo ammissibile
$k \leftarrow 1$
**while** $e_{avg} \leq e_{max}$ **do**
  **for** $i = 1 : N_{out}$ **do**
    $e_i = d_i - y_{i,k-1}$
    $\nu_{i,k-1} = [W]_{i,k-1} X + b_i$
    $\delta_i = \varphi'(\nu_i)\, e_i$
    **for** $j = 1 : N_{in}$ **do**
      $w_{ji,k} = w_{ji,k-1} + \alpha\, \delta_i\, x_j$
    **end for**
  **end for**
  $\nu_k = [\mathbf{W}]_\mathbf{k} X + \mathbf{b}$
  $\mathbf{y}_k = \varphi(\nu_k)$
  $e_{avg} = \dfrac{1}{N_{out}} \sum_{i=1}^{N_{out}} e_i$
  $k{+}{+}$
**end while**

# Appendice B

# Codici MATLAB

## B.1 A00_CreateWS.m

```
clear; clc; close all;
mkdir('Dataset\PixelLabelData')
mkdir('Dataset\SourceImage')

mkdir('Evaluation\Original')
mkdir('Evaluation\PixelLabelData')
mkdir('Evaluation\RealPixelLabel')
mkdir('Evaluation\SourceImage')
mkdir('Evaluation\TestedImage')

mkdir('OldNetwork')

mkdir('Original\SourceImage')
```

## B.2 A01_AugmNet.m

```
clear; clc;

resEx = true;
cropdiv = {[1 1],[2 3]};
dimCropAI = [224 224];

% load gTruth to modify
gTcharOld = 'Original/gTruthOrig.mat';
gTcharNew = 'gTruth.mat';

for nTrasf = 1:4
    for nCrop = 1:length(cropdiv)
        Croop(resEx,nTrasf,cropdiv{nCrop},dimCropAI);
```

```
        resEx = false;
    end
end

gTruth = modGTruth(gTcharOld,gTcharNew,'Dataset');
```

### B.2.1 Croop.m

```
function [] = Croop(resEx,trasf,cropdiv,dimCropAI)

% reset ex
if resEx == true
    ex = 1;
    save('Num.mat','ex');
end

zer = '0000';
load('Num.mat'); %#ok<*LOAD>

dataDir = cd;
load('Original/gTruthOrig.mat');

% path to original images
imDir = fullfile(dataDir,'Original','SourceImage');
pxDir = fullfile(dataDir,'Original','PixelLabelData');

classNames = (string(table2array(gTruth.LabelDefinitions(:,1))))';
pixelLabelID = (str2num(char(string(table2array ...
    (gTruth.LabelDefinitions(:,4))))))';

imds = imageDatastore(imDir);
pxds = pixelLabelDatastore(pxDir, classNames, pixelLabelID);

% path where to save dataset
pathLabel  = fullfile(dataDir,'Dataset','PixelLabelData');
pathSource = fullfile(dataDir,'Dataset','SourceImage');

for i = 1:numel(imds.Files)
    % define image to crop
    I1 = imread(imds.Files{i});
    I2 = imread(pxds.Files{i});
    IC = categorical(I2,pixelLabelID,classNames);

    % select trasformation type
    switch trasf
        case 2
```

```
            I1 = flip(I1,2);
            IC  = flip(IC,2);
        case 3
            I1 = jitterColorHSV(I1,"Brightness",0.3,...
                "Contrast",0.4,"Saturation",0.2);
        case 4
            I1 = jitterColorHSV(I1,"Brightness",0.3, ...
                "Contrast",0.4,"Saturation",0.2);
            I1 = flip(I1,2);
            IC  = flip(IC,2);
    end

    % choose max win size for calculating stepX and step Y
    [~,szmin] = min(size(I1,1:2));
    [~,szmax] = max(size(I1,1:2));

    win1 = round(size(I1,szmin)/cropdiv(szmin))+ ...
        round(size(I1,szmin)/cropdiv(szmin)/2);
    win2 = round(size(I1,szmax)/cropdiv(szmax))+ ...
        round(size(I1,szmax)/cropdiv(szmax)/2);
    win = max([win1,win2]);

    een = 0;

    stepX = floor((size(I1,szmax)-win)/(cropdiv(szmax)-1));
    stepY = floor((size(I1,szmin)-win)/(cropdiv(szmin)-1));

    % initialize image dataset to croop
    IcropSource = cell(1,cropdiv(1)*cropdiv(2));
    IcropLabel  = cell(1,cropdiv(1)*cropdiv(2));
    Icrop_C     = cell(1,cropdiv(1)*cropdiv(2));

    for nn1 = 1:cropdiv(1)
        for nn2 = 1:cropdiv(2)
            % if is the right part of the image
            if nn2==cropdiv(2)
                % if is the bottom-right part of the image
                if nn1==cropdiv(1)
                    IcropSource{cropdiv(2)*(nn1-1)+nn2} = ...
                        imresize(imcrop(I1,...
                        [round(size(I1,szmax))-win+1 ...
                        round(size(I1,szmin))-win+1 win-1 win-1]), ...
                        dimCropAI);
                    Icrop_C{cropdiv(2)*(nn1-1)+nn2} = ...
                        imresize(imcrop(IC,...
                        [round(size(I2,szmax))-win+1 ...
                        round(size(I2,szmin))-win+1 win-1 win-1]), ...
```

```
                        dimCropAI);
                    IcropLabel{cropdiv(2)*(nn1-1)+nn2} = uint8(0);
                    for nn3 = 1:size(pixelLabelID,2)
                        IcropLabel{cropdiv(2)*(nn1-1)+nn2} = ...
                        IcropLabel{cropdiv(2)*(nn1-1)+nn2} + ...
                        uint8((Icrop_C{cropdiv(2)*(nn1-1)+nn2} ...
                        == classNames(nn3))*pixelLabelID(nn3));
                    end
                    een = 1;
                else
                    IcropSource{cropdiv(2)*(nn1-1)+nn2} = ...
                        imresize(imcrop(I1,...
                        [round(size(I1,szmax))-win+1 1+(nn1-1)*stepY ...
                        win-1 win-1]),dimCropAI);
                    Icrop_C{cropdiv(2)*(nn1-1)+nn2} = ...
                        imresize(imcrop(IC,...
                        [round(size(I2,szmax))-win+1 1+(nn1-1)*stepY ...
                        win-1 win-1]),dimCropAI);
                    IcropLabel{cropdiv(2)*(nn1-1)+nn2} = uint8(0);
                    for nn3 = 1:size(pixelLabelID,2)
                        IcropLabel{cropdiv(2)*(nn1-1)+nn2} = ...
                        IcropLabel{cropdiv(2)*(nn1-1)+nn2} + ...
                        uint8((Icrop_C{cropdiv(2)*(nn1-1)+nn2} ...
                        == classNames(nn3))*pixelLabelID(nn3));
                    end
                end
            % if is the bottom part BUT NOT the bottom-right part of
            % the image
            elseif nn1==cropdiv(1) && een==0
                IcropSource{cropdiv(2)*(nn1-1)+nn2} = ...
                    imresize(imcrop(I1, ...
                    [1+(nn2-1)*stepX round(size(I1,szmin))-win+1 ...
                    win-1 win-1]),dimCropAI);
                Icrop_C{cropdiv(2)*(nn1-1)+nn2} = ...
                    imresize(imcrop(IC, ...
                    [1+(nn2-1)*stepX round(size(I2,szmin))-win+1 ...
                    win-1 win-1]),dimCropAI);
                IcropLabel{cropdiv(2)*(nn1-1)+nn2} = uint8(0);
                for nn3 = 1:size(pixelLabelID,2)
                    IcropLabel{cropdiv(2)*(nn1-1)+nn2} = ...
                    IcropLabel{cropdiv(2)*(nn1-1)+nn2} + ...
                    uint8((Icrop_C{cropdiv(2)*(nn1-1)+nn2} ...
                    == classNames(nn3))*pixelLabelID(nn3));
                end
            else
                IcropSource{cropdiv(2)*(nn1-1)+nn2} = ...
                    imresize(imcrop(I1,...
```

```
                    [1+(nn2-1)*stepX 1+(nn1-1)*stepY ...
                    win-1 win-1]),dimCropAI);
                Icrop_C{cropdiv(2)*(nn1-1)+nn2} = ...
                    imresize(imcrop(IC,...
                    [1+(nn2-1)*stepX 1+(nn1-1)*stepY ...
                    win-1 win-1]),dimCropAI);
                IcropLabel{cropdiv(2)*(nn1-1)+nn2} = uint8(0);
                for nn3 = 1:size(pixelLabelID,2)
                    IcropLabel{cropdiv(2)*(nn1-1)+nn2} = ...
                    IcropLabel{cropdiv(2)*(nn1-1)+nn2} + ...
                    uint8((Icrop_C{cropdiv(2)*(nn1-1)+nn2} ...
                    == classNames(nn3))*pixelLabelID(nn3));
                end
            end
        end
    end

    % save image and label to dataset

    for n = 1:cropdiv(1)*cropdiv(2)
        name = 'ImData_';
        name = [name zer(1:4-length(char(num2str(ex)))) ...
        num2str(ex) '.png'];
        imwrite(IcropSource{n},fullfile(pathSource,name),'PNG');
        imwrite(IcropLabel{n},fullfile(pathLabel,name),'PNG');
        ex = ex+1;
    end
    save('Num.mat','ex');
end

end
```

### B.2.2 modGtruth.m

```
function gTruth = modGTruth(gTcharOld,gTcharNew,foldData)

load(gTcharOld); %#ok<*LOAD>

% path to original images
dataDir = cd;
imDir = fullfile(dataDir,foldData,'SourceImage');
pxDir = fullfile(dataDir,foldData,'PixelLabelData');

imds = imageDatastore(imDir);
pxds = imageDatastore(pxDir);

```

```
gDataSource = groundTruthDataSource(imds.Files);

gLabelData = table(pxds.Files,'VariableNames',{'PixelLabelData'});
gTruth = groundTruth(gDataSource,gTruth.LabelDefinitions,gLabelData);
    %#ok<*NODEF>

save(gTcharNew,'gTruth');

end
```

## B.3 A02_TrainNet.m

```
%% Data setup
clear; clc;
load('gTruth.mat')

dataDir = cd;
imDir = fullfile(dataDir,'Dataset','SourceImage');
pxDir = fullfile(dataDir,'Dataset','PixelLabelData');

classNames = (string(table2array(gTruth.LabelDefinitions(:,1))))';
pixelLabelID = (str2num(char(string(table2array ...
    (gTruth.LabelDefinitions(:,4))))))';

% Creating dataset
imds = imageDatastore(imDir);
pxds = pixelLabelDatastore(pxDir, classNames, pixelLabelID);

% Splitting dataset
Q = numel(imds.Files);
trainRatio = 0.70;
valRatio = 0.15;
testRatio = 0.15;

[trainingIdx, valIdx, testIdx] = dividerand(Q, trainRatio, valRatio,
    testRatio);

trainingImages = imds.Files(trainingIdx);
valImages = imds.Files(valIdx);
testImages = imds.Files(testIdx);

trainingLabels = pxds.Files(trainingIdx);
valLabels = pxds.Files(valIdx);
testLabels = pxds.Files(testIdx);

imdsTrain = imageDatastore(trainingImages);
```

```
imdsVal = imageDatastore(valImages);
imdsTest = imageDatastore(testImages);

pxdsTrain = pixelLabelDatastore(trainingLabels, classNames, ...
    pixelLabelID);
pxdsVal = pixelLabelDatastore(valLabels, classNames, pixelLabelID);
pxdsTest = pixelLabelDatastore(testLabels, classNames, pixelLabelID);

pximdsTrain = pixelLabelImageDatastore(imdsTrain, pxdsTrain);
pximdsVal = pixelLabelImageDatastore(imdsVal, pxdsVal);

% Save testing dataset
for n = 1:numel(imdsTest.Files)
    I1 = imread(imdsTest.Files{n});
    I2 = imread(pxdsTest.Files{n});
    nn0 = '00000';
    nn = char(string(n)); ln = length(nn);
    nn = [nn0(1:5-ln) nn];
    name1 = "ImgData_" + num2str(nn) + ".png";
    name2 = "ImgLabel_" + num2str(nn) + ".png";
    imwrite(I1,fullfile(dataDir,'Evaluation','Original',name1),'PNG');
    imwrite(I2,fullfile(dataDir,'Evaluation','RealPixelLabel', ...
        name2),'PNG');
end

%% Network setup

% Setting image size and number of classes
imageSize = [224 224 3];
numClasses = length(classNames);

% Balancing classes
tbl = countEachLabel(pxds);
imageFreq = tbl.PixelCount ./ tbl.ImagePixelCount;
classWeights = median(imageFreq) ./ imageFreq;

% Loading layers (DeepLabv3+)
layers = deeplabv3plusLayers(imageSize, numClasses, "resnet18");

% Replacing last layer
pxLayer = pixelClassificationLayer('Name', 'labels', 'Classes', ...
    tbl.Name, 'ClassWeights', classWeights);
layers = replaceLayer(layers, "classification", pxLayer);

%% Network training

opts = trainingOptions('sgdm', ...
```

```
    'InitialLearnRate', 1e-3, ...          % 1e-6 (DeepLabv3+)
    'MaxEpochs', 15, ...                   % 10 to 20
    'MiniBatchSize', 10, ...               % from 4 to 12 (DeepLabv3+)
    'Shuffle', 'every-epoch', ...
    'ValidationData', pximdsVal, ...
    'ValidationFrequency', 20, ...
    'ValidationPatience', 5, ...
    'LearnRateSchedule', 'piecewise', ...
    'LearnRateDropPeriod', 5, ...
    'LearnRateDropFactor', 0.5, ...
    'Plots', 'training-progress');

net = trainNetwork(pximdsTrain, layers, opts);

save('Evaluation/Net.mat','net');
```

## B.4 A03_TestNet.m

```
clear; clc;
load('gTruth.mat')
load('Net.mat')
zer = '0000';
foldSource = 'SourceImage';

dataDir = cd;

orDir = fullfile(dataDir,'Evaluation','Original');
imDir = fullfile(dataDir,'Evaluation','SourceImage');
pxRealDir = fullfile(dataDir,'Evaluation','RealPixelLabel');
pxDir = fullfile(dataDir,'Evaluation','PixelLabelData');
tsDir = fullfile(dataDir,'Evaluation','TestedImage');

dimCropAI = [224 224];

% Creating dataset
imdsOrig = imageDatastore(orDir);
I1 = imread(imdsOrig.Files{1});
szOrig = [size(I1,1) size(I1,2)];

%crop image
for n = 1:numel(imdsOrig.Files)
    I1 = imread(imdsOrig.Files{n});
    I1 = imresize(I1,dimCropAI);
    nn0 = '00000';
    nn = char(string(n)); ln = length(nn);
    nn = [nn0(1:4-ln) nn];
```

```
    name = "ImTest_" + num2str(nn) + ".png";
    imwrite(I1,fullfile(imDir,name),'PNG');
end

%% Creating dataset
classNames = (string(table2array(gTruth.LabelDefinitions(:,1))))';
pixelLabelID = (str2num(char(string(table2array ...
    (gTruth.LabelDefinitions(:,4))))))';

imdsTest = imageDatastore(imDir);

pxdsResults = semanticseg(imdsTest, net, "WriteLocation", ...
    pxDir, 'MiniBatchSize', 4);
pxdsReal = pixelLabelDatastore(pxRealDir,classNames,pixelLabelID);

if false
    for i = 1:numel(imdsOrig.Files)
        I2 = imread(pxdsResults.Files{i});
        IC = categorical(I2,pixelLabelID,classNames);
        IC = imresize(IC,szOrig);

        Icrop = uint8(0);
        for nn3 = 1:size(pixelLabelID,2)
            Icrop = Icrop + uint8((IC == ...
                classNames(nn3))*pixelLabelID(nn3));
        end

        imwrite(Icrop,pxdsResults.Files{i},'PNG');
        foldSource = 'Original';
    end
end

%% Save Test image

IB = cell(numel(imdsTest.Files),1);

for n = 1:numel(imdsTest.Files)
    IB{n} = verifyLabel('Evaluation',n,gTruth,foldSource);
    IB{n} = imresize(IB{n},szOrig);
    nn0 = '00000';
    nn = char(string(n)); ln = length(nn);
    nn = [nn0(1:4-ln) nn];
    name = "ImTest_" + num2str(nn) + ".png";
    imwrite(IB{n},fullfile(tsDir,name),'PNG');
end

%% Mesure results
```

```
76
77 metrics = evaluateSemanticSegmentation(pxdsResults,pxdsReal);
```

### B.4.1 verifyLabel.m

```
1  function IB = verifyLabel(foldData,i,gTruth,foldSource)
2
3  dataDir = cd;
4
5  imDir = fullfile(dataDir,foldData,foldSource);
6  pxDir = fullfile(dataDir,foldData,'PixelLabelData');
7
8  classNames = (string(table2array(gTruth.LabelDefinitions(1:4,1))))';
9  pixelLabelID = (str2num(char(string(table2array ...
10     (gTruth.LabelDefinitions(1:4,4))))))';
11
12 imds = imageDatastore(imDir);
13 pxds = pixelLabelDatastore(pxDir, classNames, pixelLabelID);
14
15 I1 = imread(imds.Files{i});
16 I2 = imread(pxds.Files{i});
17
18 IC = categorical(I2,pixelLabelID,classNames);
19 IB = labeloverlay(I1,IC,'Colormap',table2array ...
20     (gTruth.LabelDefinitions(:,3)));
21
22 end
```

## B.5 autonavNet.m

```
1  function data_out = autonavNet(ImgIn)
2
3  persistent net
4
5  if isempty(net)
6      net = coder.loadDeepLearningNetwork("autonavNet_02.mat");
7  end
8
9  imgOut = predict(net,ImgIn);
10 [~,imgOut] = max(imgOut,[],3);
11
12 data_out = imgOut;
13
14 end
```

# Appendice C

# Codice nodo ROS

## C.1 autonav_ai_node.cpp

```
#include <ros/ros.h>
#include <autonavNet.h>
#include <opencv2/imgproc.hpp>
#include <message_filters/subscriber.h>
#include <message_filters/time_synchronizer.h>
#include <cv_bridge/cv_bridge.h>
#include <sensor_msgs/Image.h>
#include <sensor_msgs/PointCloud2.h>
#include <sensor_msgs/image_encodings.h>

using namespace sensor_msgs;
using namespace message_filters;
using namespace cv;

ros::Publisher pub_image, pub_cloud;

int lparam, freq;
int cksum = 0;
double *num;
double imgOut[50176];

cv_bridge::CvImagePtr cv_brg;

Mat cv_data, cv_rgb1, cv_rgb2;

sensor_msgs::Image img_out;
sensor_msgs::PointCloud2 ptcld;

void callback(const ImageConstPtr& img, const PointCloud2ConstPtr& cloud
    )
{
```

```
31  // get point cloud pointer
32  ptcld = *cloud;
33
34  // from ROS image to OpenCV image
35  cv_brg = cv_bridge::toCvCopy(img, img->encoding);
36
37  // resize image for autonav Network (224x224)
38  resize(cv_brg->image, cv_data, Size(224, 224), 0, 0, INTER_LINEAR);
39
40  // change from BGRA to RGB
41  cvtColor(cv_data, cv_rgb1, COLOR_BGRA2RGB);
42  cvtColor(cv_data, cv_rgb2, COLOR_BGRA2RGB);
43
44  // change order of data (from [R G B R G B ... R G B] to [R R R ... G
     G G ... B B B ...])
45  for(int i=0; i<224; i++){
46    for(int j=0; j<224; j++){
47      for(int k=0; k<3; k++){
48        cv_rgb2.data[i + j*224 + k*50176] = cv_rgb1.data[i*672 + j*3 + k
     ];
49      }
50    }
51  }
52
53  // use autonav Network
54  autonavNet(cv_rgb2.data, imgOut);
55
56  // find pixel that correspond to category specified in num
57  for(int i=0; i<224; i++){
58    for(int j=0; j<224; j++){
59      cksum = 0;
60      for(int k=0; k<lparam; k++){
61        //std::cout << num[k] << " ";
62        cksum = (imgOut[i + j*224]==num[k]) ? cksum+1 : cksum;
63      }
64      //std::cout << std::endl;
65      cv_rgb1.data[i*672 + j*3] = (cksum>0) ? 255 : 0;
66    }
67  }
68
69  // resize image back to original shape
70  resize(cv_rgb1, cv_data, Size(img->width, img->height), 0, 0,
     INTER_LINEAR);
71
72  // transform axes z of point cloud: if pixel is NaN or is part of
     category
73  for(int i=0; i<(ptcld.height); i++){
```

```
74      for(int j=0; j<(ptcld.width); j++){
75        ptcld.data[8 + i*(ptcld.width)*16 + j*16] = 0;
76        ptcld.data[9 + i*(ptcld.width)*16 + j*16] = 0;
77        if(cv_data.data[i*(ptcld.width)*3 + j*3]<=127 && ptcld.data[11 + i
      *(ptcld.width)*16 + j*16]!=127){
78          ptcld.data[10 + i*(ptcld.width)*16 + j*16] = 0;
79          ptcld.data[11 + i*(ptcld.width)*16 + j*16] = 0;
80        }
81        else{
82          ptcld.data[10 + i*(ptcld.width)*16 + j*16] = 64;
83          ptcld.data[11 + i*(ptcld.width)*16 + j*16] = 192;
84        }
85      }
86    }
87
88    // convert from cv::Mat to cv_bridge image
89    std_msgs::Header header;
90    cv_bridge::CvImage img_brg = cv_bridge::CvImage(header, sensor_msgs::
       image_encodings::RGB8, cv_data);
91
92    // publish image
93    img_brg.toImageMsg(img_out);
94    pub_image.publish(img_out);
95
96    // publish point cloud
97    pub_cloud.publish(ptcld);
98
99    ROS_INFO("AutonavNetwork is running!!");
100
101 }
102
103 int main(int argc, char** argv)
104 {
105   ros::init(argc, argv, "autonav_ai_node");
106   ros::NodeHandle nh;
107
108   // set ROS parameters
109   nh.param("/lenght_category",lparam,1);
110
111   std::vector<double> cat_list(lparam,0);
112   double categ[lparam];
113
114   if(nh.hasParam("/category")){
115     nh.getParam("/category",cat_list);
116   }
117   else{
118     cat_list[0] = 1;
```

```
  }
  for(int x=0; x<lparam; x++){
    categ[x] = cat_list[x];
  }

  num = categ;

  nh.param("/frequency",freq,5);

  message_filters::Subscriber<Image> image_sub(nh, "/camera/left/
   image_rect_color", 1);
  message_filters::Subscriber<PointCloud2> cloud_sub(nh, "/camera/
   point_cloud/cloud", 1);

  TimeSynchronizer<Image, PointCloud2> sync(image_sub, cloud_sub, 20);
  sync.registerCallback(boost::bind(&callback, _1, _2));

  ros::Rate r(freq); // 10 hz
  while (ros::ok())
  {
    pub_image = nh.advertise<sensor_msgs::Image>("autonavAI/Image_mask",
    20);
    pub_cloud = nh.advertise<sensor_msgs::PointCloud2>("autonavAI/
   PointCloud2_mask", 20);

    ros::spinOnce();
    r.sleep();
  }

  return 0;
}
```

# Bibliografia


[1] Rover (space exploration). https://en.wikipedia.org/wiki/Rover_(space_exploration). (Ultimo accesso: 2022-03-14).

[2] Cuebong Wong, Erfu Yang, Xiu-Tian Yan, and Dongbing Gu. Adaptive and intelligent navigation of autonomous planetary rovers — a survey. In *2017 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*, pages 237–244, 2017.

[3] Charu C Aggarwal et al. Neural networks and deep learning. *Springer*, 10:978–3, 2018.

[4] Vijay Badrinarayanan, Alex Kendall, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for image segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 39(12):2481–2495, 2017.

[5] Luca Torresin. Sviluppo ed applicazione di reti neurali per segmentazione semantica a supporto della navigazione di rover marziani, 2019.

[6] Liang-Chieh Chen, Yukun Zhu, George Papandreou, Florian Schroff, and Hartwig Adam. Encoder-decoder with atrous separable convolution for semantic image segmentation. In *Proceedings of the European conference on computer vision (ECCV)*, pages 801–818, 2018.

[7] Understanding ros 2 nodes. https://docs.ros.org/en/humble/Tutorials/Understanding-ROS2-Nodes.html. (Ultimo accesso: 2022-05-23).

[8] Giovanni Giardini Mauro Massari and Franco Bernelli Zazzera. Autonomous navigation system for planetary exploration rover based on artificial potential fields. In *Proceedings of Dynamics and Control of Systems and Structures in Space (DCSSS) 6th Conference*, pages 153–162, 2004.

[9] Fesq L.M. Nesnas I.A. and R.A. Volpe. Autonomy for space robots: Past, present, and future. In *Current Robotics Reports*, volume 2, pages 251–263, 2021.

[10] Esa - press release. https://www.esa.int/Newsroom/Press_Releases/ExoMars_suspended. (Ultimo accesso: 2022-03-20).

[11] Nasa autonomous systems taxonomy. https://ntrs.nasa.gov/citations/20180003082. (Ultimo accesso: 2022-03-14).

[12] Lanzutti A. Gasparetto A., Boscariol P. and Vidoni R. Path planning and trajectory planning algorithms: A general overview. In *Motion and Operation Planning of Robotic Systems. Mechanisms and Machine Science*, volume 29, pages 3–27, 2015.

[13] Bab-Hadiashar A. Yousif, K. and Hoseinnezhad R. An overview to visual odometry and visual slam: Applications to mobile robotics. In *Intelligent Industrial Systems*, volume 1, page 289–311, 2015.

[14] Sebastiano Chiodini, Marco Pertile, and Stefano Debei. Occupancy grid mapping for rover navigation based on semantic segmentation. *ACTA IMEKO*, 10(4):155–161, 2021.

[15] H. Durrant-Whyte and T. Bailey. Simultaneous localization and mapping: part i. *IEEE Robotics Automation Magazine*, 13(2):99–108, 2006.

[16] Yi Ma, Stefano Soatto, Jana Košecká, and Shankar Sastry. *An invitation to 3-D vision: from images to geometric models*, volume 26. Springer, 2004.

[17] Z. Zhang. A flexible new technique for camera calibration. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(11):1330–1334, 2000.

[18] D. Nister, O. Naroditsky, and J. Bergen. Visual odometry. In *Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2004. CVPR 2004.*, volume 1, pages I–I, 2004.

[19] Davide Scaramuzza and Friedrich Fraundorfer. Visual odometry. part i: The first 30 years and fundamentals. *IEEE Robotics Automation Magazine*, 18(4):80–92, 2011.

[20] Davide Scaramuzza and Friedrich Fraundorfer. Visual odometry : Part ii: Matching, robustness, optimization, and applications. *IEEE Robotics Automation Magazine*, 19(2):78–90, 2012.

[21] Roland Siegwart, Illah Reza Nourbakhsh, and Davide Scaramuzza. *Introduction to autonomous mobile robots*. MIT press, 2011.

[22] D.G. Lowe. Object recognition from local scale-invariant features. In *Proceedings of the Seventh IEEE International Conference on Computer Vision*, volume 2, pages 1150–1157 vol.2, 1999.

[23] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. Surf: Speeded up robust features. In *European conference on computer vision*, pages 404–417. Springer, 2006.

[24] Jianbo Shi and Tomasi. Good features to track. In *1994 Proceedings of IEEE Conference on Computer Vision and Pattern Recognition*, pages 593–600, 1994.

[25] Martin A. Fischler and Robert C. Bolles. Random sample consensus: A paradigm for model fitting with applications to image analysis and automated cartography. *Commun. ACM*, 24(6):381–395, jun 1981.

[26] H Christopher Longuet-Higgins. A computer algorithm for reconstructing a scene from two projections. *Nature*, 293(5828):133–135, 1981.

[27] K. S. Arun, T. S. Huang, and S. D. Blostein. Least-squares fitting of two 3-d point sets. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-9(5):698–700, 1987.

[28] Berthold Horn. Closed-form solution of absolute orientation using unit quaternions. *Journal of the Optical Society A*, 4:629–642, 04 1987.

[29] David Nistér and Henrik Stewénius. A minimal solution to the generalised 3-point pose problem. *Journal of Mathematical Imaging and Vision*, 27(1):67–79, 2007.

[30] Bert M Haralick, Chung-Nan Lee, Karsten Ottenberg, and Michael Nölle. Review and analysis of solutions of the three point perspective pose estimation problem. *International journal of computer vision*, 13(3):331–356, 1994.

[31] Bill Triggs, Philip F McLauchlan, Richard I Hartley, and Andrew W Fitzgibbon. Bundle adjustment—a modern synthesis. In *International workshop on vision algorithms*, pages 298–372. Springer, 1999.

[32] T. Bailey and H. Durrant-Whyte. Simultaneous localization and mapping (slam): part ii. *IEEE Robotics Automation Magazine*, 13(3):108–117, 2006.

[33] M.W.M.G. Dissanayake, P. Newman, S. Clark, H.F. Durrant-Whyte, and M. Csorba. A solution to the simultaneous localization and map building (slam) problem. *IEEE Transactions on Robotics and Automation*, 17(3):229–241, 2001.

[34] Phil Kim. Matlab deep learning. *With machine learning, neural networks and artificial intelligence*, 130(21), 2017.

[35] Hamed Habibi Aghdam and Elnaz Jahani Heravi. *Guide to convolutional neural networks*. Springer, 2017.

[36] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[37] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[38] Y. Le Cun, L.D. Jackel, B. Boser, J.S. Denker, H.P. Graf, I. Guyon, D. Henderson, R.E. Howard, and W. Hubbard. Handwritten digit recognition: applications of neural network chips and automatic learning. *IEEE Communications Magazine*, 27(11):41–46, 1989.

[39] Ros. https://www.ros.org/. (Ultimo accesso: 2022-05-23).

[40] zed-ros-wrapper. http://wiki.ros.org/zed-ros-wrapper. (Ultimo accesso: 2022-05-24).

[41] Single-precision floating-point. https://en.wikipedia.org/wiki/Single-precision_floating-point_format. (Ultimo accesso: 2022-05-24).

[42] octomap-server. http://wiki.ros.org/octomap_server. (Ultimo accesso: 2022-05-24).

[43] Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. OctoMap: An efficient probabilistic 3D mapping framework based on octrees. *Autonomous Robots*, 2013. Software available at http://octomap.github.com.

[44] Rtab-map. http://introlab.github.io/rtabmap/. (Ultimo accesso: 2022-05-24).

[45] rtabmap-ros. http://wiki.ros.org/rtabmap_ros. (Ultimo accesso: 2022-05-24).

[46] move-base. http://wiki.ros.org/move_base. (Ultimo accesso: 2022-05-24).

[47] Christoph Rösmann, Wendelin Feiten, Thomas Wösch, Frank Hoffmann, and Torsten Bertram. Trajectory modification considering dynamic constraints of autonomous robots. In *ROBOTIK 2012; 7th German Conference on Robotics*, pages 1–6. VDE, 2012.

[48] Christoph Rösmann, Wendelin Feiten, Thomas Wösch, Frank Hoffmann, and Torsten Bertram. Efficient trajectory optimization using a sparse model. In *2013 European Conference on Mobile Robots*, pages 138–143. IEEE, 2013.

[49] piksi-multi-rtk-ros. https://github.com/ethz-asl/ethz_piksi_ros/tree/master/piksi_multi_rtk_ros. (Ultimo accesso: 2022-05-24).

[50] Davide Paganini. Morpheus: ideazione e progettazione del rover di ateneo, 2017.

[51] Enrico Morellato. Algoritmi e software di controllo per la trazione del rover "morpheus", 2016.