Object detection with Pepper: Gruppo 2

Luca Boccia, Francesco Chiarello e Michele Oliva

{l.boccia12, f.chiarello1, m.oliva26}@studenti.unisa.it

Università degli Studi di Salerno

Gennaio 2021

Indice

1 Introduzione

2 Descrizione della soluzione

2.1 Architettura

Il nostro software si compone di tre nodi: Un nodo che implementa il servizio di object detection, che riceve in input un'immagine da analizzare e, per ogni oggetto nell'immagine, restituisce il relativo bounding box, la classe ed il livello di confidenza relativo alla predizione. Un nodo che implementa tre servizi che forniscono un'interfaccia verso le funzionalità di text to speech, gestione della posa e movimento della testa di Pepper. Un nodo master che implementa la funzionalità richiesta dall'homework acquisendo le immagini dal topic della camera di Pepper e utilizzando i servizi offerti dagli altri due nodi. Per l'interfacciamento da e verso Pepper viene utilizzato il NaoQi SDK.

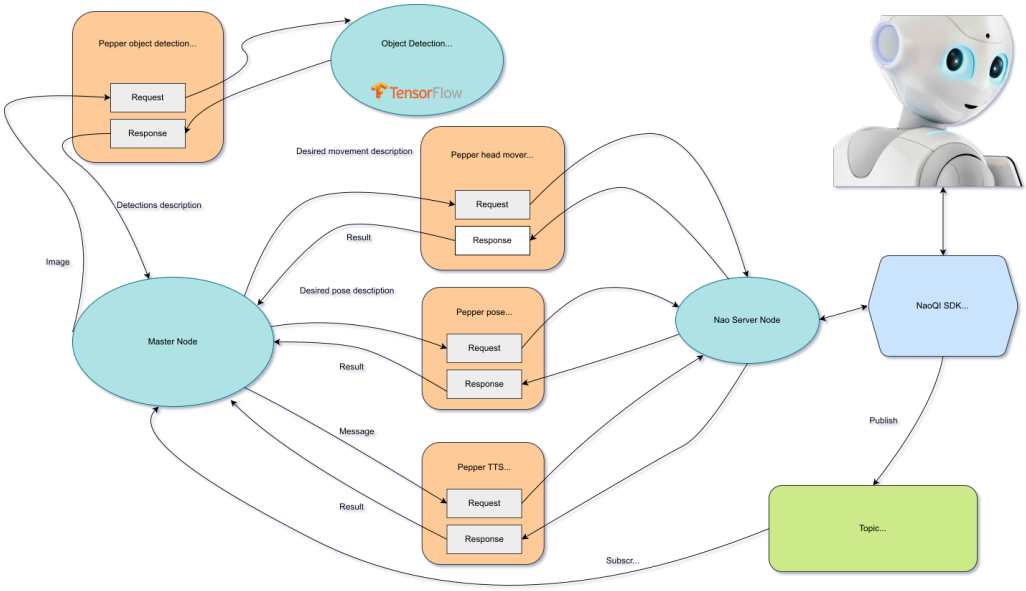

Figura 1: Architettura del software

2.2 Choices

BULLET 1 Abbiamo implementato le interfacce verso le funzionalità del robot (TTS, movimento della testa e gestione della posa) come servizi perché questo permette di modellare queste funzionalità come risorse condivise tra i vari nodi e di gestire gli accessi ad esse. L'utilizzo del paradigma request-response al posto di quello publish-subscribe permette inoltre di attendere e verificare l'esito di un'operazione, cosa molto utile quando si lavora con sistemi fisici. Non abbiamo utilizzato un'azione (non bloccante) perché per la nostra applicazione era comodo bloccare il sistema in attesa del completamento delle operazioni sul robot.

Per quanto riguarda le funzionalità di object detection, anch'esse sono fornite tramite un servizio, perché così facendo non è necessario inserire nel messaggio che trasporta il risultato della detection l'immagine di riferimento, in quanto il nodo che effettua la richiesta al servizio sa quale immagine sta fornendo in input, permettendo così di mantenere il messaggio più snello e di semplificare la sincronizzazione tra nodi che richiedono un'operazione di detection ed il nodo che la realizza. Abbiamo scelto un servizio e non un'azione perché per la nostra applicazione non è necessario fare altro mentre è in corso una detection. Come nota a margine, si fa notare che il fatto che il detector restituisca il risultato al solo nodo che invia la richiesta potrebbe sembrare limitante nel caso in cui il risultato della detection su di una stessa immagine dovesse servire per più nodi, perché si potrebbe pensare di dover invocare più volte il servizio fornendo la stessa immagine in input. In realtà, questa cosa può essere evitata facendo in modo che un solo nodo invochi il servizio e pubblichi il risultato ottenuto su di un apposito topic. Questo sistema, rispetto all'utilizzo del paradigma publish-subscribe, consente di conservare la possibilità di ricevere un feedback sul risultato dell'operazione.

BULLET 2 Per gestire i parametri di configurazione utilizziamo il parameter server di ROS e un file di configurazione che viene automaticamente caricato su di

esso tramite il nostro launch file.

BULLET 3 Per il task di visione abbiamo scelto di catturare più immagini della scena e di farne lo stitching. In tutto catturiamo 10 immagini, ruotando la testa tra 0.8 e -0.8 rad lungo l'asse z (yaw), e tra 0.2 e -0.2 rad lungo l'asse y (pitch).

Fare l'image stitching ha i seguenti vantaggi: * Permette di vedere oggetti grandi che occupano più spazio di una sola immagine * Permette di evitare che oggetti piccoli divisi su due immagini possano essere visti due volte o nessuna * Permette di allargare il campo visivo sia in orizzontale che in verticale, scegliendo opportunamente il numero e la posizione delle catture.

Avere una sola immagine fa sì che il detector lavori una sola volta. Se ciò sia effettivamente vantaggioso rispetto ad effettuare la detection su più immagini separate dipende da due fattori: il numero di immagini su cui fare la detection e il modello utilizzato. Nel caso di più immagini potrebbe bastare un detector leggero, visto che queste sono piccole e gli oggetti sono di meno e occupano gran parte della scena. Nel nostro caso, invece, l'immagine risulterà più grande, e soprattutto conterrà più oggetti che occuperanno, relativamente alle dimensioni, meno spazio. Come vedremo nella prossima slide, ci sarà bisogno di un detector più complesso.

3 Risultati sperimentali

3.1 Models comparison

La scelta del modello è stata dettata da tre parametri: la risoluzione delle immagini in ingresso, i tempi di esecuzione del detector per immagine e, infine, la sua capacità di individuare e classificare correttamente gli oggetti. In particolare, rispetto a quest'ultimo aspetto non ci è possibile fornire una valutazione quantitativa delle performance, in quanto davanti al robot si presenta sempre la stessa scena, pertanto abbiamo valutato i detector in base al loro comportamento relativamente a questa scena, e quindi i giudizi nella colonna "Performance" della tabella fanno riferimento a questa condizione. Inoltre, le immagini che abbiamo catturato, a valle dello stitching, hanno una dimensione di circa 900x400. Durante le nostre prove ci siamo accorti che questo tipo di immagini non sono tipici input per le reti a nostra disposizione. Alcune di queste, nonostante avessero valori di mAP di tutto rispetto sul dataset COCO (come si può vedere su https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf2_detection_zoo.md), non risultavano efficaci nella situazione in esame.

Inizialmente sono state provate due reti che lavorano con immagini aventi risoluzione 640x640, tali detector, facendo un downscale dell'immagine risultato del processo di stitching, hanno dimostrato pessime performance. Abbiamo quindi deciso di provare reti che lavorano con immagini aventi una risoluzione più grande. La Faster RCNN che usa come backbone Resnet 101 e che lavora con immagini aventi una risoluzione di 1024x1204, avendo performance discrete nella detection (non tutti gli oggetti venivano rivelati ed ad altri veniva assegnata l'etichetta sbagliata), ha dimostrato che la risoluzione delle immagini date in input ai modelli deve essere un fattore discriminante nella scelta del modello. Il passo successivo è stato, quindi, quello di provare una rete che lavorasse con immagini più simili, in risoluzione, a quelle che noi le diamo in input: la Faster RCNN che usa come backbone Resnet 50,

ha dimostrato, in tempi ragionevoli, ottime performance in termini di rilevazione degli oggetti, ma a qualche oggetto veniva assegnata una label sbagliata; la Faster RCNN che usa come backbone Resnet 101 ha dimostrato le migliori performance sia per la classificazione che la detection, ma in tempi meno ragionevoli.

Nelle immagini riportanti i risultati delle detection con le varie reti, le bande verticali di colore verde fluo evidenziano la suddivisione dell'immagine utilizzata per separare tra loro le regioni sinistra, destra e centrale.

I tempi di esecuzione riportati nella tabella fanno riferimento all'esecuzione della rete all'interno della VM "di riferimento" per il corso, senza l'utilizzo di accelerazione basata su GPU. Inoltre, sono tempi "di regime", cioè validi per tutte le esecuzioni del detector eccetto la prima, che, sperimentalmente, per motivi probabilmente legati a TensorFlow, impiega considerevolmente più tempo delle altre.

3.2 The chosen model

Si può notare come lo stitching permetta di fornire in input al detector una visione d'insieme della scena, che consente ad esempio di rilevare oggetti quali la scrivania, che sarebbero stati più difficili da individuare attraverso l'analisi di singole catture successive.

4 Conclusioni

Riportare nella relazione le conclusioni del lavoro svolto.